METHODOLOGIES AND TOOLS FOR YIELD IMPROVEMENT OF FIELD-PROGRAMMABLE LOGIC ARCHITECTURES

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

PONGSTORN MAIDEE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dr. Kia Bazargan, Advisor
December, 2009

© PONGSTORN MAIDEE 2009

Acknowledgments

I would like to thank all people who have helped me during my doctoral study. I especially want to thank my advisor, Prof. Kia Bazargan, for his guidance and constant support during my research and study at the University of Minnesota. He was always accessible and willing to help his students in any ways he can. All my friends in Minnesota made my time at the university filled with rewarding experiences. In particular, I would like to thank Cristinel Ababei for giving me a lot of good suggestions at the beginning of my graduate study and Satish Sivaswamy for spending his time discussing with me on various topics ranging from research problems to sports.

My deepest gratitude goes to my family for their love and support throughout my life; this dissertation is simply impossible without them. I am indebted to my father, Krittawit Maidee, for his care and love. He instilled the love of reading technical books in me by giving textbooks as my birthday gifts in my teenage years. Looking back, it is the most important thing that inspires me into a lifelong learning. I cannot ask for more from my mother, Pathum Maidee. She always gives me unconditional love and support. I admire her (among other things) for her vision in education for her children despite our family background and environment. She always finds ways to help her children succeed in education. I love you, Mom and Dad. I will always be indebted to my sister, Eed, Wilasinee Maidee, for her care to my parents while I was away for my study. She not only is a good daughter to my parents, but also fills my role perfectly on all occasions. Her efforts allowed me to spend most of my time for study.

Finishing up this thesis could be much harder without a countless number of encouraging words from my girlfriend, Pang, Thitiporn Chanla. She always cheers me up with her words of wisdoms. I cannot say thank enough for John and Jula St.Marie for their help and kindness over the years that I was in Minnesota. They always look after me as if I am one of their family members. There are many more people who I need to thank. Instead of listing some here and leaving out the rest, I would like all of you to accept my sincere gratitude. Thank you!

Dedication

To my parents.

Abstract

According to the international technology roadmap for semiconductors (ITRS) predictions, controlling manufacturing yield is going to be a challenging task in future technologies. It was shown that the effective yield of the future field programmable gate arrays (FPGAs) will be too low to make a profit [1]. Several FPGA yield improvement techniques have been proposed such as clustering, spare column and node covering. However, the challenges of future fabrication technologies are so great that these techniques cannot fully address high yield demands of the future. Thus, novel techniques should be explored.

We propose three approaches for FPGA yield improvement in this thesis: one adds redundant components to the FPGA architecture to tolerate permanent faults. Another technique speeds up a synthesis technique used in a rewiring engine to allow for replacement or enforcement of faulty wires in a circuit. The third technique uses a search space pruning technique to speedup the optimization of FPGA architecture development. The proposed yield improvement techniques can be applied in conjunction with other existing techniques, resulting in an effective framework for FPGA yield improvement.

Chapter 2 addresses fault tolerant FPGA architectures that introduce redundancies in the architecture to replace faulty components. A methodology is proposed for estimating the effectiveness. Effectiveness of existing defect-tolerant schemes such as clustering, spare column and node covering for contemporary FPGA architectures. Furthermore, a number of new schemes to further improve yield are proposed in Chapter 2. Several techniques for tolerating defects in

switch boxes were also introduced. The results show that the spare column scheme is very effective in maintaining a satisfactory yield. However, our studies show that to maintain reasonable yields, the number of spare columns must increase in the future. We also show that having redundancy for routing channels increases the absolute yield, but the benefit is outweighed by the area overhead for some types of routing channels. As a result, we show that redundancy must be judicially applied to the routing architecture to result in high yield numbers.

Chapter 3 addresses yield improvement at the synthesis level. Circuit rewiring is proposed in this thesis to enhance effectiveness of existing approaches, namely customization and design-specific approaches. The success of rewiring depends on both the quality and speed of the rewiring engine. Among several rewiring techniques that have been purposed, a Set-of-Pairs-of-Functions-to-be-Distinguished (SPFD)-based rewiring was shown to be more effective than the others both in theory and practice. However, due to its longer runtime, it is not a viable rewiring technique. A novel algorithm is proposed to avoid expressing SPFDs explicitly. Instead, a few satisfiability problem (SAT) instances are solved, allowing rewiring of one instance in the order of milliseconds. The experimental results show that our proposed technique's runtime is only a fraction of that of a conventional one and it scales well with the number of candidate wires considered. The existing SPFD-based rewiring approaches also limit where a new wire can be added. We present a theory that allows us to add a new wire virtually anywhere in the circuit structure. An algorithm based on this theory is also presented. Experiments show that the number of wires which can be rewired increases significantly and the number of alternate wires for a given wire also increases.

Chapter 4 deals with designing a family of FPGA chips. The goal of this chapter is to minimize area across a large number of designs. Minimizing area in

turn helps improve yield. We formulate the family selection process as an FPGA family composition problem and propose an efficient algorithm for solving it. The formulation can capture an increasingly complex specialized functional block selection problem for FPGA families. The technique is applied to an architecture similar to Xilinx Virtex FPGAs. The results show that a smart composition technique can significantly reduce the expected silicon area. The benefit of providing specialized blocks can also be investigated using the technique and thus it can be used as an important tool for determining the benefits of specialized blocks for future FPGAs.

Contents

Acknowledgments **i**

Dedication **iii**

Abstract **iv**

List of Tables **x**

List of Figures **xii**

1 Introduction **1**

1.1 Electronic Devices 1

1.2 VLSI Manufacturing Yield 5

1.3 Techniques for FPGA Yield Improvements 8

1.4 Thesis Contributions and Outline 12

2 Yield Improvement with Defect-tolerant FPGA architectures **18**

2.1 Introduction 18

2.2 FPGA Architecture 21

2.3 Defect-tolerant Architectures 24

2.3.1 Defect-tolerant Schemes 26

2.4 Yield Estimation 36

2.4.1 Basic Yield Computation 36

2.4.2 Yield of Each Component 37

2.4.3 Yield of Different Replacement Schemes 51
2.5 Simulation Results 53
2.5.1 Methodology 53
2.5.2 Results and Discussions 54
2.6 Summary 62

3 Yield Improvement by Circuit Rewiring 64

3.1 Introduction 64
3.2 Basic Terminology 67
3.3 Set-of-Pairs-of-Functions-to-be-Distinguished (SPFD) 67
3.4 Previous work on SPFD-based rewiring 70
3.5 SAT-based SPFD rewiring 71
3.5.1 Previous Work: Using SAT to Compute Minimum SPFD . 71
3.5.2 A New Distributing Miter 72
3.5.3 A Limitation of Distributing Miters 75
3.5.4 A Fast SPFD Rewiring Algorithm 76
3.5.5 Quality of the Proposed Technique 83
3.6 SPFD-based rewiring beyond dominator nodes 84
3.6.1 Generating SPFD Required for a New Wire 85
3.6.2 The Proposed Algorithm and its Correctness 89
3.7 Experimental Results 93
3.7.1 Experimental Results of the Proposed Efficiency Improvement 93
3.7.2 Experimental Results of the Proposed Efficacy Improvement 96
3.8 Summary 97

4 Yield Improvement of an FPGA Family 107

4.1 Introduction 107

4.2 Basic Application Modules and FPGA Building Blocks 109
4.2.1 Mapping Application Module Distributions to FPGA Block Distributions 111
4.2.2 Interactions between Different Types of Resources 111
4.3 FPGA Family Composition 114
4.4 A Minimum Area FPGA Covering a Given Percentage of Designs 116
4.4.1 Transforming the Hyperplanes 116
4.4.2 Finding the Minimum Area FPGA 117
4.5 Finding a Good FPGA Family 120
4.6 Experiments 122
4.7 Summary 128

5 Conclusions and Future Work 130

Bibliography 136

List of Tables

2.1 Sizes of each active component [2]. Items annotated with a through f correspond to elements in Figure 2.10. Others are for elements in an SB. 40

2.2 The appropriate values of u, v and s of $N_2(0,0)$ for an FPGA with odd width and even height. 49

2.3 Yield improvement when defect-tolerant schemes were applied to CLBs. 59

2.4 Further yield improvement when defect-tolerant schemes were applied to networks of segments of length one and long wires. 60

2.5 The effect of removing defect-tolerant SBs for networks of segments of length one. 61

3.1 Rewirability and runtime comparison of BDD and SAT -based SPFD rewiring. 101

3.2 Runtime details of results shown in Table 3.1. 102

3.3 Comparison when the circuit depth is limited. 103

3.4 Scalability on the number of candidate wires. 104

3.5 Rewiring ability of the proposed algorithm for the unlimited depth case. 105

3.6 Rewiring ability of the proposed algorithm for the limited depth case. . 106

4.1 Parameters of normal distributions of design modules. 124

4.2 Mapping from design module distributions to FPGA resource distributions. 124

4.3 The linear transformation used in the experiments. 124

4.4 A minimum area FPGA for selected percent coverage points. 125

4.5 Effects of the number of FPGAs in a family. 126

List of Figures

1.1 Because of defects, not all chips will work as expected and testing must be performed. (a) The process of testing fabricated chips (FPGAs), (b) Only observable defects that change the circuit behavior result in chip failure. 7

1.2 Defect-tolerant architectures. (a) The flow of testing and reconfiguring, and (b) an FPGA containing defects is configured to use spare resources in place of the defective resources. 9

1.3 The customization approach. (a) The flow of a customization approach, (b) an initial mapping of the design, (c) the adjusted mapping to avoid defects on one FPGA, and (d) the adjusted mapping to avoid defects on another FPGA. 15

1.4 The design specific approach. (a) The flow of the approach, and (b) an FPGA passes the test to be used for the given design as long as no defects appear on resources used by the design. 16

1.5 Circuit rewiring for yield improvement. (a) The flow of the approach, (b) an original circuit, (c) a rewired circuit with the same functionality as (b), and (d) a map of the design of the rewired circuit that avoids defect-prone resources. 17

2.1 Simplified Virtex-II-like architecture. 22

2.2 Abstract view inside a CLB. 23

2.3 Switches with in SB connecting a tracks
a) Abstract view of cross-point switches of a bi-directional track.
b) Abstract view of cross-point switches of uni-directional tracks.
c) Implementation view of cross-point switches of uni-directional tracks.
d) Implementation view of cross-point switches of uni-directional tracks for a simplified Virtex-II-like architecture.
e) Two back-to-back buffers connecting long wires from different orientations.
f) Connections between long wires and a CLB. 25

2.4 Staggered routing segments. 25

2.5 Spare column scheme [3]. a) without defective column. b) with one defective column. 27

2.6 Details of typical routing resources. a) no faulty columns. b) Column 2 is faulty. 28

2.7 Shifted replacement of CLBs [4] 30

2.8 If a track, its driver or receiver fails, OMUX and IMUX can shift the track to avoid the defect. 32

2.9 Defect tolerant SBs. A SB has four identical driving groups, but only one such group is shown. A driving group for S_2 or S_6 has four additional inputs from vertical and horizontal middle-point segments, but only two such inputs are shown for simplicity. Note that a shuffle MUX also has inputs from CLB, shown as one arrow for one MUX to keep the figure simple. a) Shuffle MUXes are added to a set of SB multiplexers b) Shuffle MUXes are added to four middle-point channels shared among four driving groups. These MUXes remove dependency propagated through the middle connection. c) Shuffle MUXes are also added to 4 incoming channels, removing dependency among 4 incoming channels. d) SB MUXes are merged with shuffle MUXes making incoming tracks from different directions independent. 35

2.10 Details of a configurable logic block containing N LUTs. There are I inputs to the CLB. Internal connection is assumed to be 50% sparse. MUXes drawn using dotted lines are for the grouping scheme and do not exist in other schemes. The number of inputs to each MUX is annotated next to it. Labels in parenthesis correspond to items in Table 2.1. Note that if there are spare segments, the total segments of length i including spare ones must be used instead of m_i. 39

2.11 The abstract view of a network of segments of length one containing 3x3 SBs. 44

2.12 Abstract view of one independent network out of $L^2/4$ for segments of length L. $L = 2$ is used in the picture. The following details in a SB are omitted from the picture for simplicity : 1) middle-point wires of track i connecting to all MUXes of track i, 2) connections from a MUX of track i to incoming wires of track i from three directions. 46

2.13 Abstract view of a defect tolerant long wire network. 50
2.14 Future yields of non-defect-tolerant architectures. 56
2.15 Effective yields in 2009 when defect-tolerant schemes are applied to CLBs. The bars show the yield for an architecture with no redundancy. 57
2.16 Effective yields in 2021 when defect-tolerant schemes are applied to CLBs. 57

3.1 The bipartite graph representing XOR. 68
3.2 A miter to compute a minimum SPFD at node n. Bold nodes constitute the separator set. 73
3.3 An example showing the shortcomings of the miter of Figure 3.2. The separator set is $\{c, d\}$. 73
3.4 A circuit showing $SPFD(n)$ propagation to POs through dotted edges. 75
3.5 Optional caption for list of figures 76
3.6 The distributing miter to compute SPFD(n) of Figure 3.4. Grey squares are priority encoders. Boxes labeled N/C correspond to inputs $\notin TFO(n)$, hence not connected (*e.g.* input f of gate i). 77
3.7 A flowchart of our rewiring algorithm. 78
3.8 A checking miter for validating the replacement of (n, g) of Figure 3.4 with (c, g). 80
3.9 A function miter used for computing the new $f(i)$ after replacing (n, g) with (c, g) of the circuit in Figure 3.4. 83
3.10 A circuit showing deficiency of BDD-based SPFD rewiring. The fanin order at each node is indicated below the node. 85
3.11 Decompositions of SPFD at edges into $SPFD^{valid}$ and $SPFD^{invalid}$. 88
3.12 Existence of $SPFD^{invalid}$ at any node $n \in TFI\{DOM(sk(w_r))\}$. . 88

3.13 Histogram of the number of alternative wires of $term1$. 98
3.14 Show $r(n)$ of Figs. 3.13 for rewiring with only dominator nodes and with any nodes. 98

4.1 Interactions between two types of resources. (See Section 4.2.2 for details) 112
4.2 Effects of a finite-member FPGA family. 113
4.3 Demonstration of (4.4) for two-variable case. 118
4.4 Show how to compute x^{k+1}. 119
4.5 The increase in coverage by adding one more FPGA (8,9,6) into a family is shown with bold solid lines. (The boundary of the previous coverage is shown using dashed lines) 121

5.1 Employing multiple yield improvement schemes to enhance the customization approach. 131
5.2 Employing multiple yield improvement schemes to enhance the design-specific approach. 132

Chapter 1

Introduction

1.1 Electronic Devices

Consumer electronic devices such as cell phones are becoming smaller, yet increasingly more powerful. This incredible improvement is possible because of the advances in several technologies, the most important one being the semiconductor fabrication technology.

An integrated circuit (IC), which is an integral part of electronic devices, is an electronic circuit containing many miniaturized devices manufactured on a common surface. The circuits are called monolithic or hybrid integrated circuits depending on whether their surfaces are semiconductor substrates or circuit boards. In this thesis, an integrated circuit will be used to mean a monolithic integrated circuit.

The process called very large-scale integration (VLSI) originally developed in the 1970s allows for combining millions of transistors onto a single chip making nowadays integrated circuits like microprocessors possible. A typical VLSI design process consists of several main steps: designing the functionality of the circuit [5, 6, 7], creating mask sets [8], and manufacturing the chip [9]. Circuit design involves deciding how to connect transistors together to accomplish the functionality required by the design. After the circuit has been designed to meet the given specifications, it will be built on a semiconductor substrate. Transistors are built

on the substrate in several steps. At each step, only parts of the transistors will be processed. Thus, a mask set is required for each step to expose those parts and to hide the others. Finally, the circuit will be manufactured with successive applications of these mask sets.

VLSI Manufacturing steps and theirs corresponding mask sets can be grossly divided into two groups as ones used to create transistors and ones used to connect the transistors. The first group of steps is used to form transistors and to adjust their properties, while the latter group is used to connect transistors by layers of metal.

An integrated circuit designed for a specific usage is called an application-specific integrated circuit (ASIC). ASICs can be classified by how they are designed into three main types: full-custom, standard cell and gate array designs. Each type of designs possesses different characteristics that suit for different sets of design constraints.

A full-custom design style lets ones to design and adjust each transistor to the full extent allowed by the manufacturing technology. As a result, all quality metrics of the circuit such as performance, area, and power consumption can be designed to be close to their optimums. However, as this design style offers a large number of parameters to be adjusted, it requires tremendous design time which translates into large design cost and slow time-to-market, on top of the manufacturing costs. For example, the Nvidia GeForce 4 is estimated to cost the company $100 million and take 12-month time-to-market [10].

A standard-cell design style reduces design cost by employing a set of well designed cells. In this design style, a handful of optimized cells will be provided. A designer can compose his/her designs with the given cells that come in different characteristics, resulting in a shorter design time compared to a full-custom design.

However, because the design space in this style is more restricted, circuit quality is bound to be worse than full-custom designs. Even though this design style offers lower design cost and shorter time to market, but it still incurs the same amount of manufacturing cost as a full-custom design.

As VLSI manufacturing cost is a large proportion of the total cost, the *mask programmable* gate array design style was developed to offer lower manufacturing costs by sharing some manufacturing steps among several designs, and hence amortizing the one-time overhead cost of making the mask sets among all such designs. In this design style, a configurable array of transistors is manufactured without specific knowledge of any particular design. To implement a particular design, one needs to only design mask sets to connect pre-fabricated transistors to collectively provide the functionality needed by the design. Thus, only a few mask sets are required to complete the design. Because of this feature, this class of designs can be called a mask programmable integrated circuits. As the pre-manufactured arrays can be used across many designs, the manufacturing cost can be averaged out among them. Furthermore, only a few mask sets are needed to apply to the arrays after the design process completes, time-to-market can be significantly reduced. However, in this design style, the design space is even more limited than that of a standard-cell design resulting in inferior designs in all aspects except cost.

In contrast to gate arrays that require mask programming, a programmable logic device (PLD) is an integrated circuit that can be configured after manufacturing. Its programmability can be accomplished by programmable read-only memories (PROMs), ultraviolet-erasable PROMs (EPROMs), electrically erasable PROMs (EEPROMs), or SRAMs. The early devices of this type of chips are the programmable arrays logic (PAL) and a generic array logic (GAL) which were

used as glue-logic for interfacing among larger integrated circuits. Recently, the complexity of such devices has increased significantly, and the new generation of such devices are called complex programmable logic devices (CPLD) that can be used to implement large integrated circuits.

A field-programmable gate array (FPGA) is another type of programmable logic devices. While the structure of other programmable logic devices resemble a sum-of-product logic array, FPGA structure is similar to the gate array design in that it contains a two-dimensional array of logic elements. FPGAs consist of electrically configurable logic blocks and configurable interconnects. FPGA has a higher memory to logic devices ratio than that of a CPLD making it suitable for implementing large state machines such as microprocessors.

Designing an application using FPGAs provides several benefits. FPGAs are designed and fabricated in large quantities similar to the gate-array design style and are not intended for any particular design. However, in the case of FPGAs, all mask sets are applied and the chip is packaged as well. Hence, implementing a design with FPGAs has a very fast time-to-market compared to other design styles. The greatest benefits of FPGAs are that their manufacturing cost is averaged out among large numbers of designs and customers, and that the startup cost of the design is very low compared to implementing it as an ASIC. As a result, the market for FPGAs has grown from $14 million in 1987 to an estimated $2.75 billion in 2010 [11]. However, FPGA's unit cost is higher than that of high volume ASICs. High-volume ASIC designs can amortize the high cost of mask sets across large numbers of chips, hence offsetting the cost benefits of FPGAs in high-volume applications. Furthermore, since FPGAs must cater to a large number of applications from different customers, they have to provide additional circuits to facilitate programmability to accommodate various designs incurring

lower silicon utilization. Thus, FPGAs are only well suited for low-to-medium volume applications.

Although there are still quality gaps between FPGAs and ASICs, these gaps are gradually shrinking over the years. It is well known and verified that the area efficiency and power consumption of FPGAs is about 35 times and 14 times worse respectively compared to ASICs [12]. Recognizing that most applications implemented on FPGAs use multipliers and memory units, FPGA vendors have included these specialized blocks in their recent products [13, 14]. As a result, the overall area utilization has improved because the introduced blocks are well optimized both in term of area and speed. For example, instead of using 50% of logic capacity of a Xilinx XC4025E to implement a 12 Kbit block of RAM, the block of RAM can now be implemented with an area efficiency close to that of a full-custom design [15].

1.2 VLSI Manufacturing Yield

The outcome of a VLSI fabrication process depends on three main factors: the design layout, the process control parameters, and randomly changing process parameters, called disturbances. The layout of a design affects manufacturing steps in several ways. For example, if two wires are too close together, it is harder to manufacture them without creating an unintentional short between them. Control parameters are used to manipulate the process to obtain desired results. In contrast, disturbances are random variations inherent in any fabrication processes and are usually modeled as a set of random variables.

Disturbances in VLSI manufacturing can be classified as [16]:

1. Human errors and equipment failures,

2. Instabilities in the process conditions,
3. Material instabilities,
4. Substrate inhomogeneities, and
5. Lithography spots.

Manufacturing yield, or simply yield, is defined as the ratio of the number of chips that successfully pass the test to the total number of chips at the beginning of the manufacturing process. Manufacturing yield is inversely proportional to the cost of the chips made, because the total cost of manufacturing is divided between all functional chips that can be sold. Using FPGAs as an example, chip testing can be shown in Figure 1.1. With the presence of inevitable faults, yield is always less than 100%. As a result, yield tops the important list of any fabrication process. The evidence that the manufacturing yield raises difficult challenges can be found in [17]. The overall yield of a VLSI manufacturing process can be described as a product of three components, material-defect limited yield (Y_M), systematic mechanism limited yield (Y_S) and random-defect limited yield (Y_R), as [17]

$$Y = Y_M \times Y_S \times Y_R = Y_M \times Y_S \times \left(\frac{1}{1 + \frac{A \cdot D_0}{\alpha}} \right)^{\alpha}, \tag{1.1}$$

where A is the chip area, D_0 is the random fault density and α is the cluster factor. In this thesis only the last type of yield will be addressed. The other types of yields can be improved through improving material and manufacturing process and they are beyond the scope of this thesis.

Because the manufacturing equipment is becoming more expansive, techniques for yield improvement is urgently required. As manufacturing a smaller transistor is more difficult and requires more sophisticated equipment, the manufacturing cost is constantly increasing [18]. Hence, more and more applications are not

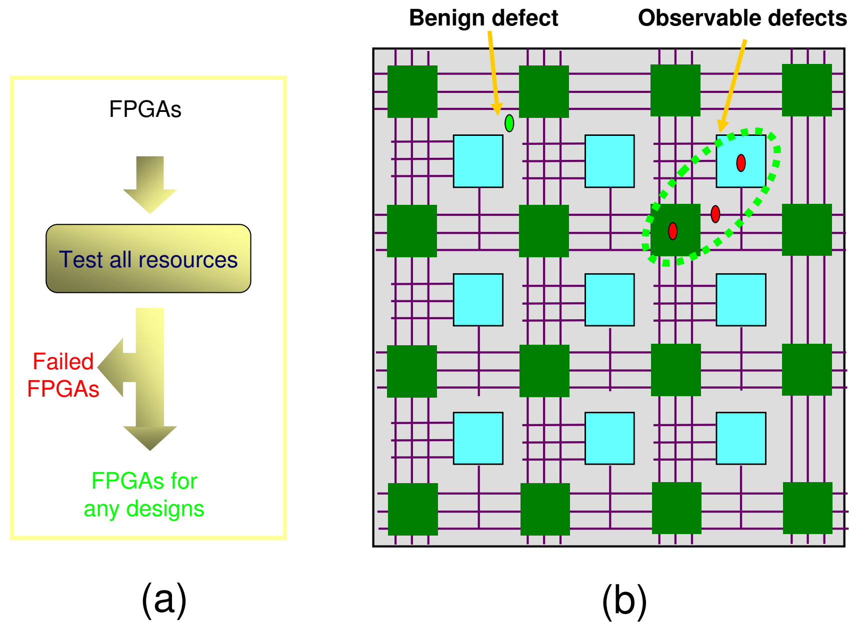

Figure 1.1: Because of defects, not all chips will work as expected and testing must be performed. (a) The process of testing fabricated chips (FPGAs), (b) Only observable defects that change the circuit behavior result in chip failure.

profitable unless they are implemented with FPGAs, spurring the growth of FPGA market in recent years.

Although FPGAs are gaining popularity partly due to high manufacturing costs, they also benefit from yield improvement in general. In the same manner, FPGA production cost is also inversely proportional to yield. Yield improvement techniques for other design styles are also applicable to FPGA manufacturing. Furthermore, with FPGA's unique characteristics, additional techniques may be further applied.

1.3 Techniques for FPGA Yield Improvements

Yield improvement techniques for FPGAs can be listed as

1. Defect-tolerant architectures,

2. Customization approach,

3. Design specific approach,

4. Rewiring for yield improvement, and

5. FPGA area reduction.

In the first approach, during the architecture design of a FPGA, spare resources will be added as redundancy into FPGA so that they can be used to replace failed resources easily. After fabrication, the FPGA will be tested. If the main resources fail, spare resources will be reconfigured to replace the failed ones. This approach can be demonstrated in Figure 1.2. The configuration is transparent to users and will not be observed by the users in any aspect of FPGA design, mapping and programming. The success of this approach depends on:

1. How we add a minimum number of resources to satisfy redundancy requirements, but not to the point to hurt yield due to added area. The minimum amount of redundancy is desirable because the total chip area increases with redundancy and the whole chip will be more susceptible to defects as can be inferred from Eq. (1.1).

2. The configuration to replace failed resources with spare ones must be simple enough to avoid large additional configuration costs.

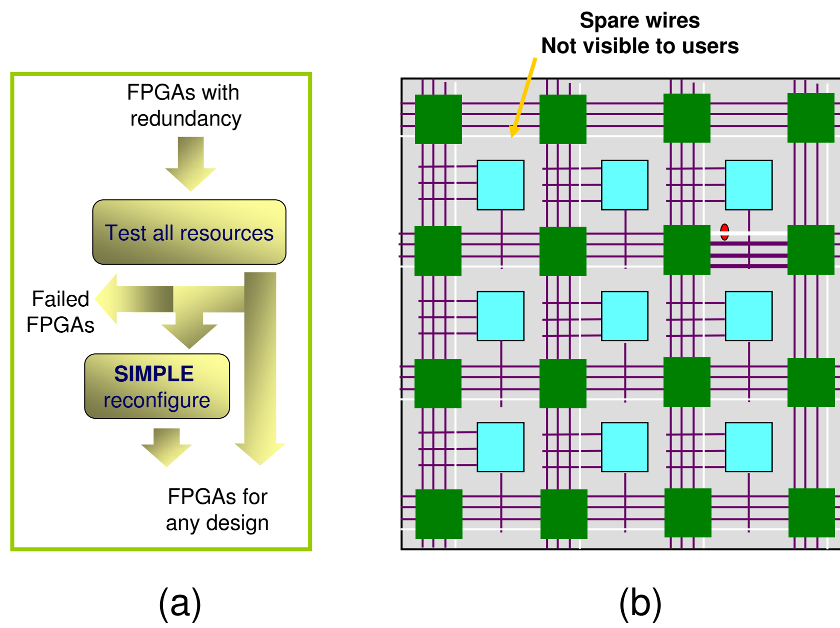

Figure 1.2: Defect-tolerant architectures. (a) The flow of testing and reconfiguring, and (b) an FPGA containing defects is configured to use spare resources in place of the defective resources.

In the customization approach, a design will be mapped for each FPGA chip to avoid its potentially unique defects. The summary of this approach can be shown in Figure 1.3. After a FPGA is tested, its defect locations will be recorded as its defect map. Thus, during the mapping of a design onto FPGAs, several possible mappings will be produced for different FPGAs depending on their defect maps. As a result, the same design may be implemented differently on FPGAs. The following issues have to be addressed in this approach.

1. How to compactly represent defect maps of defective FPGAs accurately? The number of resources on an FPGA is large. A compact defect map will require less memory and less time to be processed. On the other hand, the map has to be accurate enough so that the mapping process can use the defect map to efficiently implement the design.

2. How to efficiently generate different implementations of the same design for FPGAs with different defect maps? Mapping a design to an FPGA is a time consuming task, making mapping the design to each FPGA with different defect maps from scratch very inefficient. An efficient incremental technique to adapt an existing implementation to avoid defect locations given by a defect map is then required.

It can be seen that if an initial implementation uses resources that are not likely to fail, a smaller number of mappings can be used because the initial mapping can be used for many chips.

In the design specific approach, only resources that will be used for a design will be tested, as opposed to the previous two approaches that required all resources to be tested. In this approach, FPGAs will not be immediately tested after they are fabricated, but will be stored. Once a design is ready, only the FPGA resources

that are used by the design will be tested as seen in Figure 1.4. An FPGA will be declared as fault-free and can be used for that design as long as those resources are working, whether the rest of the resources contain defects or not. FPGAs that fail will be re-stored and will be tested for other designs. There are several benefits in this approach.

1. The testing cost depends on the amount of resources used by designs and defect density. If all resources on every FPGA are defect-free, only the resource used by designs will be tested, reducing testing cost. However, if defect density becomes higher, some FPGAs may need to be tested several times because they may fail to implement the earlier designs.

2. There are no additional steps such as re-synthesis, placement and routing and reconfiguration. Only testing, as shown in Figure 1.1, is performed several times.

The success of this approach depends on whether a design requires partial resource utilization. From a vendor's perspective, FPGA yield improves, and a customer can implement a design on FPGAs at a discount price. However, there is a utilization threshold on which using a smaller FPGA would cost customers less than the price a vendor can offer employing this approach. It is worthwhile to mention that if a design is implemented in such a way that resources with high yield are used, more chips will pass the test for the design avoiding undergoing multiple testing. As a result, testing cost reduces.

Circuit rewiring – removing some wires and adding other wires to maintain circuit functionality – can help improve the yield of a design implemented on FPGAs. Although a circuit is optimized for several criteria such as circuit delay, area and power, it may not optimized for yield because yield is not generally included

in design constraints. Circuit rewiring has been shown to help improve circuit quality [19]. A similar frame work can be easily adopted for yield improvement as shown in Figure 1.5. After a circuit is mapped onto a target FPGA architecture, the yield of each wire can be estimated. A wire with a low expected yield will be considered to be replaced by another wire with higher yield. As a result, the overall yield of the implementation increases. In practice, many constraints are imposed on a design. Thus, rewiring for yield improvement should not degrade other metrics, while helping increase yield. Hence, a powerful circuit rewiring technique is important for the success of this approach.

As FPGAs must be available for designs of various sizes and requirements, a FPGA family must contain FPGAs of appropriate sizes and resource combinations that suit customer needs at reasonable prices. Designs to be implemented on FPGAs cover a large spectrum of requirements. However, there are only a few FPGAs in a family for economic reasons. For example, XILINX Virtex-6 LXT family contains seven FPGA combinations [20]. As a result, some designs are forced to be implemented in an FPGA that has more resources than needed. From a customer's perspective, higher cost is incurred by the design not only because of the cost for unused resources but also because of the lower yield of a larger device, as inferred from Eq. (1.1). Therefore, by providing an FPGA family that minimizes the area overhead among all designs, more applications can be implemented profitably on FPGAs.

1.4 Thesis Contributions and Outline

The contributions of this thesis are as follows:

1. Efficacy of existing defect-tolerant schemes proposed for earlier FPGA ar-

chitectures will be analyzed for a contemporary FPGA architecture and new defect tolerance schemes will be proposed specifically for the architecture.

2. The problem of FPGA family composition design to minimize the total area wasted across all designs is proposed. An algorithm that can solve the problem efficiently is reported. The proposed technique can be used to compose a family with various specialized blocks.

3. Several improvements are proposed to existing rewiring techniques. The proposed novel approach outperforms the best known algorithm both in terms of quality and runtime.

The thesis can be outlined as follows: Chapter 2 focuses on defect-tolerant architectures. In this chapter, several existing schemes for various components of FPGAs will be summarized in the context of the FPGA architectures they were originally proposed for. Their applicability and efficiency when applied to a contemporary architecture, similar to XILINX Virtex-II, will be analyzed. Based on the analysis, some limitations of these schemes can be discovered. Several new schemes designed specifically to address these shortcomings will also be proposed in the chapter.

A Set-of-Pairs-of-Functions-to-be-Distinguished (SPFD), which is the underlying idea behind the most powerful rewiring techniques, will be introduced in Chapter 3. Existing rewiring techniques based on SPFD are described and the reason that they are not widely used will be explained in the chapter. Two improvements to SPFD-based rewiring will be presented. A novel approach for using SPFD to perform rewiring in a more efficient manner will be presented first. After that, a theory that improves the efficacy of SPFD-based rewiring will be presented next. An implementation of the proposed theory is also shown in this chapter.

In Chapter 4, the FPGA family composition problem aiming at minimizing the total area will be formally formulated. The problem will be shown to be not efficiently solvable. A transformation of the problem to an amenable one will be introduced. Finally, an efficient algorithm will be presented to solve the simplified problem.

The results of previous chapters will be concluded in Chapter 5. Not only the important results of the thesis will be highlighted, but also how they are related to the literature in general will be summarized.

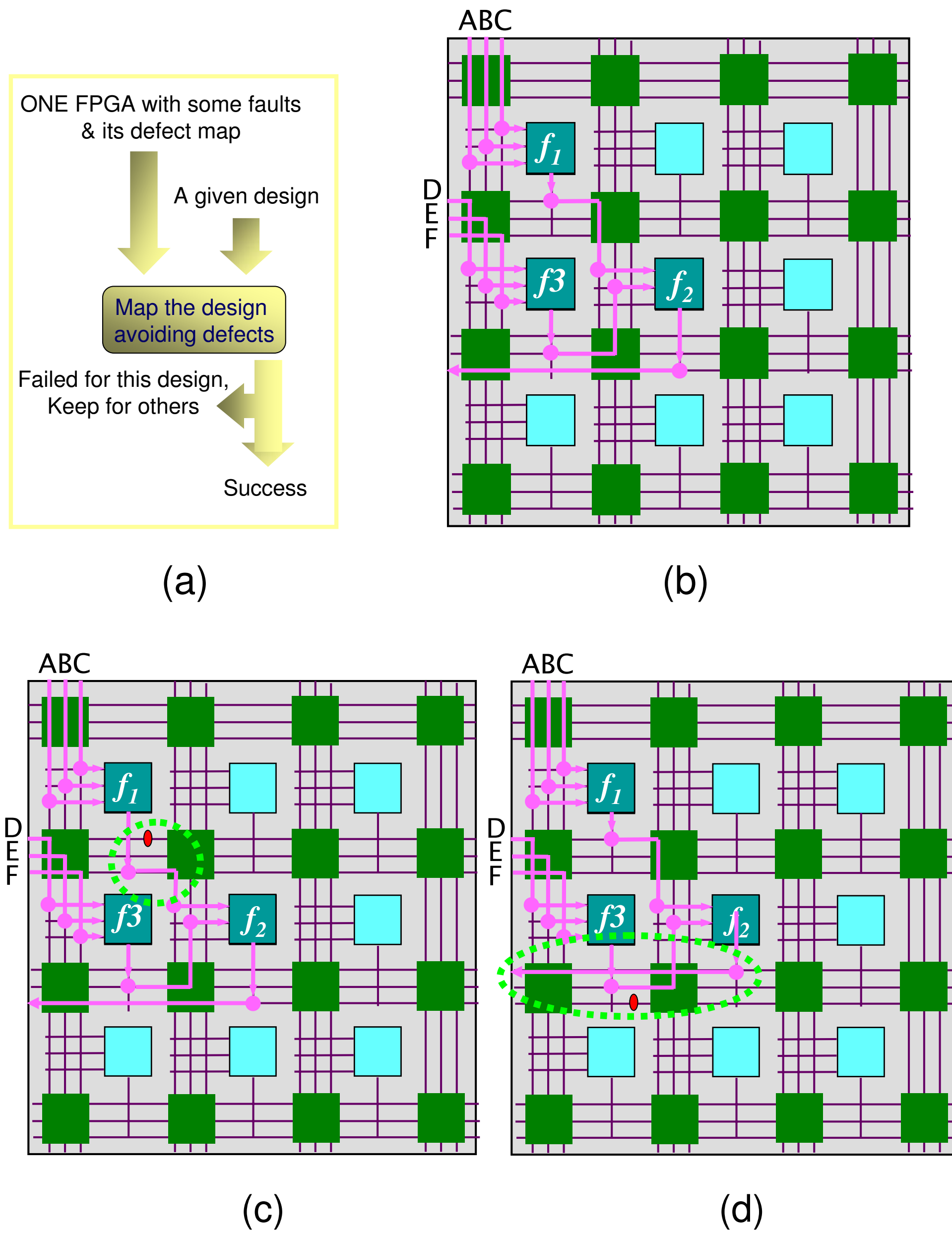

Figure 1.3: The customization approach. (a) The flow of a customization approach, (b) an initial mapping of the design, (c) the adjusted mapping to avoid defects on one FPGA, and (d) the adjusted mapping to avoid defects on another FPGA.

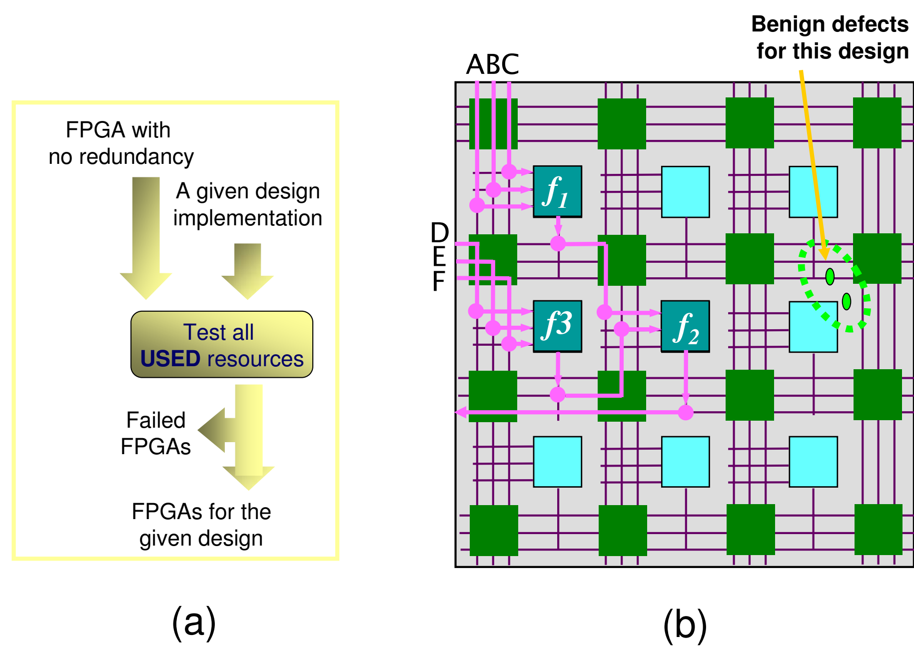

Figure 1.4: The design specific approach. (a) The flow of the approach, and (b) an FPGA passes the test to be used for the given design as long as no defects appear on resources used by the design.

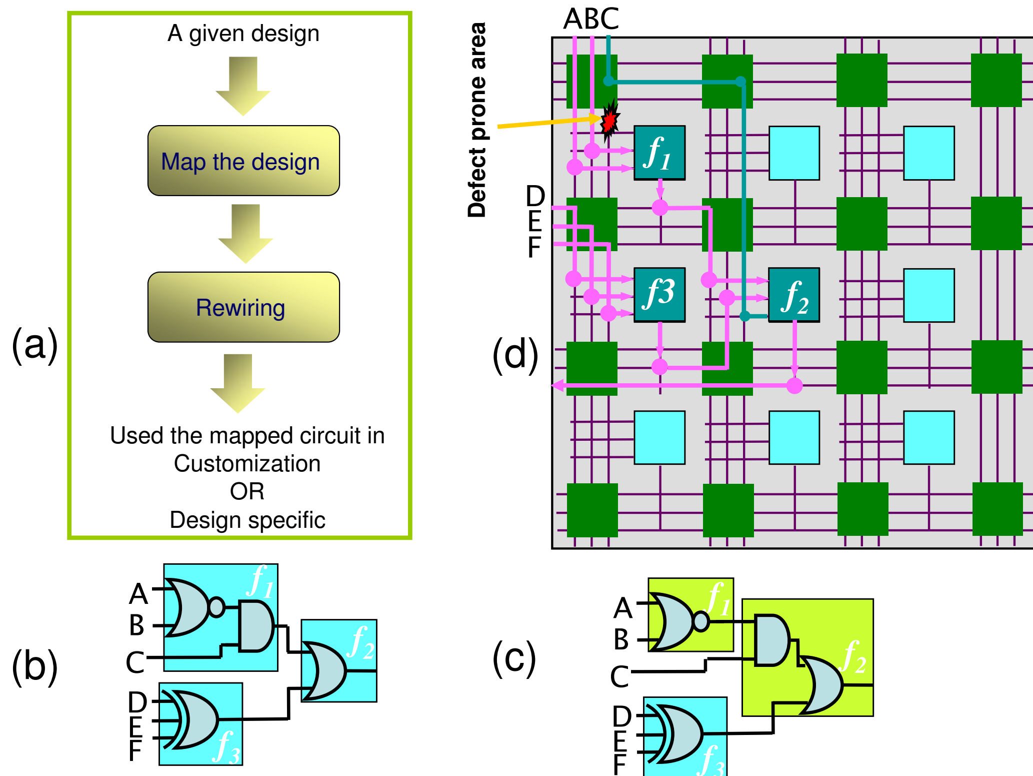

Figure 1.5: Circuit rewiring for yield improvement. (a) The flow of the approach, (b) an original circuit, (c) a rewired circuit with the same functionality as (b), and (d) a map of the design of the rewired circuit that avoids defect-prone resources.

Chapter 2

Yield Improvement with Defect-tolerant FPGA architectures

2.1 Introduction

A typical IC fabrication process contains many manufacturing steps. Inevitably, defects appear in every processing step in various forms. They can be classified in terms of their source of imperfection as human error, equipment failure, process condition instabilities, material imperfections and lithography spots, or in terms of their effects as catastrophic/structural and parametric defects [16]. However, in a mature process, most sources of defects can be controlled. The major source of remaining defects is caused by the lithography process which creates random defects. Catastrophic, random defects are considered in this thesis.

According to the ITRS, future technologies pose several challenges which are believed to be costly to address. To minimize chip cost, several changes have to be made to increase fabrication process utilization such as using larger wafers. As the chip yield is inversely proportional to the chip cost, yield improvement techniques have long been recognized as a method to help minimize chip cost.

A major reason in the success of field programmable gate arrays (FPGA) over the years has been their ability to provide a cost-effective solution for low-to-medium-volume products. Therefore, maintaining a competitive price is critical for FPGAs and is an important step towards expanding their market share.

Many techniques have been proposed for FPGA yield improvement in the last decade. Based on how comprehensively the chip should be tested, such techniques can be classified as:

1. Full test: all resources on the chip are tested to build a defect map which is used in either

    (a) Customization [21, 22] : Since not all resources will be used for a design, the design implementation is adjusted to avoid defective resources by using otherwise unused ones. However, each chip has a unique defect map. Therefore, customization has to be performed for each manufactured chip either by the FPGA vendor upon receiving design data or by customers themselves [23].

    (b) Redundancy : Spare resources will be allocated and a replacement scheme is determined during FPGAs' architecture design. After the defect map of each chip is known by testing, the spare resources will be swapped in for faulty ones at the manufacturing site or during bitstream uploads. The process is transparent to users.

2. Partial test [24, 25, 26]: a chip is tested to determine if specific circuits can be mapped. The chip will pass the test to be used for those designs as long as the resources used are not faulty even though unused resources may be defective. As a result, chip yield is higher and test costs are reduced. If a chip is not suitable for implementing one design, it may be usable for other designs. Thus, the total potential yield increases.

In this chapter, we will focus on only the full test approach. The customization scheme (1a above) individually configures each chip to implement a circuit around the defect [22]. This approach is obviously not scalable because generating each

configuration is time consuming and, therefore, not practical. Therefore, an FPGA with redundancy is the subject of our study.

FPGAs are composed of logic blocks, switch boxes, connection boxes, interconnects, IO buffers and configuration circuits. Redundancy schemes for each of these components have been proposed [3, 4, 27, 28, 29, 30, 31, 32, 33, 34, 35]. However, they were presented in a now-obsolete architecture and their yields were estimated independently from other components. In this chapter, these techniques are adapted to an architecture that resembles a commercial FPGA and new techniques are also proposed. This allows us to more accurately analyze the yield and gauge the impact of using redundancy for different FPGA resources.

Furthermore, although ITRS data indicates that defect densities in future technologies should remain the same, FPGA components and the size of critical defects become smaller. However, more functionalities and more components are added to modern FPGAs, FPGA die size in new technologies is comparable to those of the older ones. As a result, an effective defect tolerant architecture for the current technology may become ineffective in the future. Therefore, a rigorous study and search for effective defect tolerant FPGA architectures for these technologies is warranted.

The rest of the chapter is organized as follows : the FPGA architecture used in this work is described in Section 2.2. Existing replacement schemes will be summarized in Section 2.3. The effectiveness of each scheme is verified and our proposed enhancements are also presented in this section. Yield estimation of each major component is elaborated in Section 2.4, together with how they are applied to compute the yield of each scheme. Section 2.5 shows the efficiencies of defect tolerant schemes and their yields projected into the future. Finally, the conclusion and discussion is given in Section 2.6.

2.2 FPGA Architecture

Simplified architectures have long been used in studying various properties of FPGAs. XC4000-like architectures have served as the architecture model for numerous studies for a decade [2]. They consist of two-dimensional arrays of configurable logic blocks (CLB), switch boxes (SB), connection boxes (CB) and IO buffers. Wire segments running along a row/column of SBs are assumed to be bidirectional. Within an SB, wires of the same track from all four orientations can be connected to each other by switches shown in Figure 2.3a. A connection between two wires is implemented by a pass transistor or two back-to-back tristate buffers. The state of a switch is controlled by one SRAM cell.

Over the years, FPGA architectures have evolved and become more complex. Recently, leading FPGA vendors have adopted uni-directional interconnects for their FPGAs. It is shown that uni-directional interconnects can provide both delay and area improvements over their bi-directional counterparts [36]. A cross-point switch within an SB accommodating uni-directional wires can be shown at the abstract and implementation views as in Figures 2.3b and c, respectively. However, the arrangement of CLBs, SBs and CBs used in [36] is still in the XC-4000 fashion.

In this chapter, we assumed a simplified Virtex-II architecture arrangement as shown in Figure 2.1. A CLB is connected directly to routing segments through the SB, instead of through a CB in a X4000-like architecture. Therefore, therer is no CB in our architecture. An FPGA can be viewed as a two-dimensional array of tiles, each of which consists of one SB and one CLB. An SB next to the chip boundary also connects to an IO block.

A CLB consists of N basic logic elements (BLE) as shown in Figure 2.2. Each BLE contains one K-input look up table (K-LUT) and one D flip flop (D-FF) with

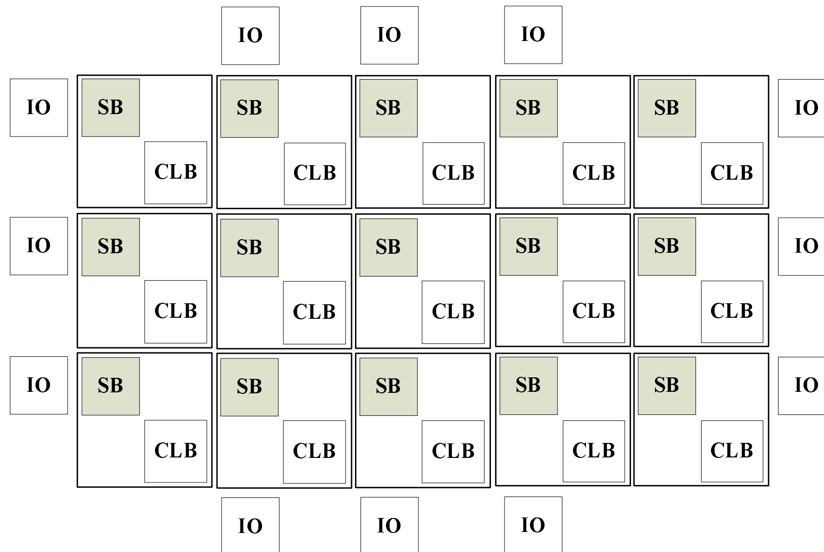

Figure 2.1: Simplified Virtex-II-like architecture.

one 2-input multiplexer to allow bypassing the D-FF. The number of inputs to a CLB is set to $I = (N+1)K/2$ to maximize BLE utilization [37]. Each input to a K-LUT can be chosen from any one of I inputs from SB or any one of N inputs of BLE outputs.

Modern FPGAs employ multi-length routing segments. A segment of length n, denoted as S_n, can reach the n-th SB from its driver, but not the $n+1$th. The Virtex-II architecture contains segments of length one, two, six and long wires. S_∞ is used to denoted a long wire. Each segment, except for the long wire type, can be driven only from one end but can be accessed either at the end or in the middle. The switches connecting to the end of a segment are called *end-point switches*, while the ones connecting to the middle of a segment are called *middle-point switches*. Note that these two types of switches may be implemented by the same multiplexer as seen in Figure 2.3d. In contrast, a long wire is bidirectional and can be driven and accessed at any SB along the wire. All segments, except for long

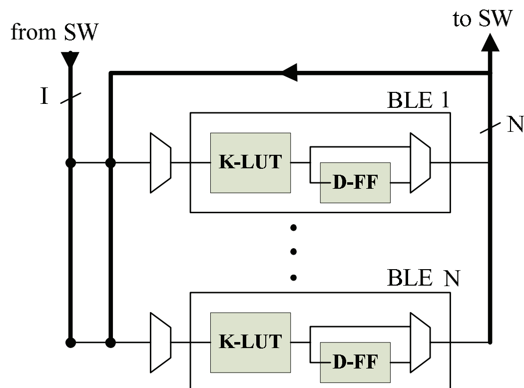

Figure 2.2: Abstract view inside a CLB.

wires, are organized into staggered patterns, *i.e.*, they originate from different SBs, as shown in Figure 2.4. For brevity, segments of length four are shown instead of length six. Within a row or column, there are two sets of segments going in opposite directions. Only wires in one direction are shown, wires in the opposite direction can be drawn similarly. In Figure 2.4, arrows indicate sources and destinations of segments (end-point switches), while a square within a SB on a segment signifies that the segment can be accessed by the SB through a middle-point switch. At a SB, the segments whose destinations are the SB are called *incoming* segments or tracks, while the segments that the SB can be connected through middle-point switches are called *middle-point* segments or tracks. From Figure 2.4, we can see that the number of S_n, going in one direction, must be a multiple of n. Let M_n be the number of S_n in one direction within a row or column. The number of S_n driven by a SB to one direction, m_n, is M_n/n which is also the number of S_n ending at one SB and the number of passing S_n accessible by one SB.

Disjoint switch boxes are used in our studies. Such switch boxes connect segments of the same type only. Within an SB, there is one set of multiplexers driving

segments of one type toward one direction; we call the set of these multiplexers a *driving group*. Note that in Figure 2.3d, a driving group contains only one multiplexer because $m_i = 1$ is assumed.

The Virtex-II architecture contains 40 and 120 tracks of segments of lengths two and six in each row or column. Thus, $\{m_2, m_6\} = \{10, 10\}$. An SB also has 16 direct wires to its surrounding eight SBs. We did not consider diagonal connections in our studies and we use $m_1 = 1$. The architecture also contains 24 bi-directional longwires for each column and row. A horizontal long wire is connected to a vertical long wire of the same track through two back-to-back tristate buffers as shown in Figure 2.3e.

2.3 Defect-tolerant Architectures

Fault tolerance techniques have been extensively studied on regular arrays for decades. They can be classified based on the levels at which they are applied as application, chip and circuit level techniques. Due to their uniform structures, yield of regular arrays can be modeled using closed form equations. Since FPGA architectures are similar to regular arrays, many techniques proposed for regular arrays are readily applicable to early versions of FPGAs containing only segments of length one. As the routing architecture of FPGAs becomes more complex, closed form solutions are difficult, if not impossible to obtain. However, most, if not all of previous work disregard defect tolerant schemes' applicability to complicated FPGAs. Hence, in this section, we will discuss in detail whenever the applicability of a scheme is not readily apparent or if it is not available in the literature.

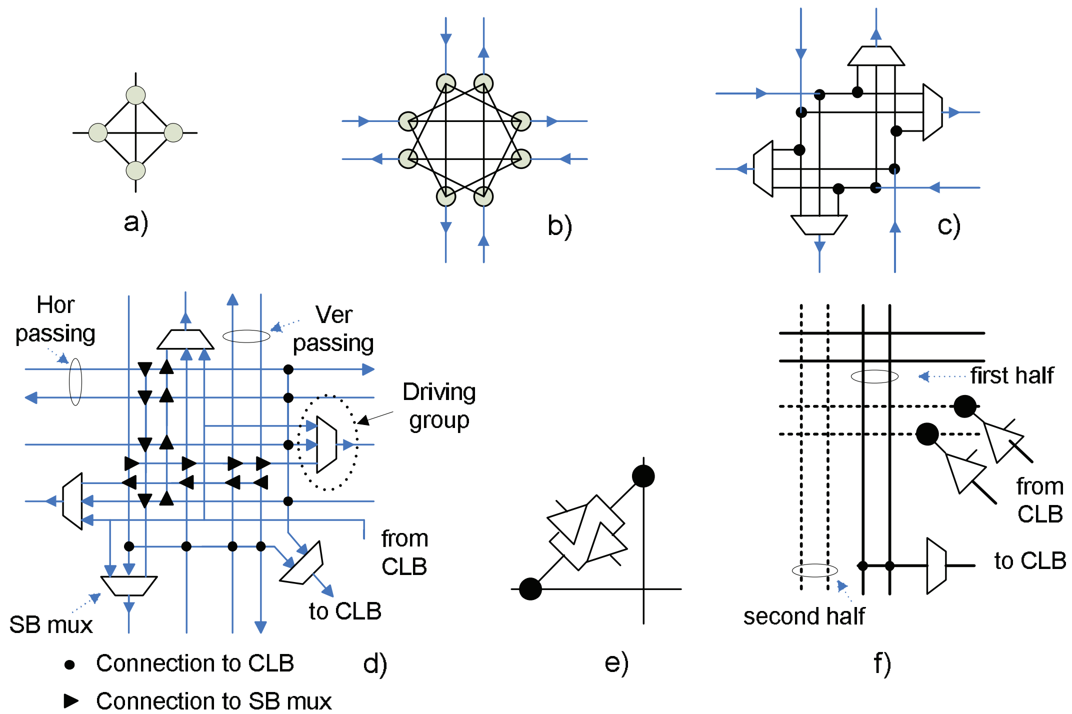

Figure 2.3: Switches with in SB connecting a tracks

a) Abstract view of cross-point switches of a bi-directional track.

b) Abstract view of cross-point switches of uni-directional tracks.

c) Implementation view of cross-point switches of uni-directional tracks.

d) Implementation view of cross-point switches of uni-directional tracks for a simplified Virtex-II-like architecture.

e) Two back-to-back buffers connecting long wires from different orientations.

f) Connections between long wires and a CLB.

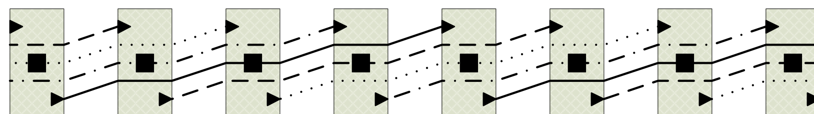

Figure 2.4: Staggered routing segments.

2.3.1 Defect-tolerant Schemes

Several schemes have been proposed in the past for different parts of FPGAs. Defect tolerant techniques for CLBs can be classified into four major categories as global, spare row/column [3], node covering [29] and clustering [38]. Shifting mechanism can be applied on segments in a channel [35].

Row / Column Replacement Scheme

Even though the spare column scheme has been discussed over the years, its applicability to two-dimensional mesh FPGAs with multi-length segments has never been verified[1]. Therefore, there is a need to validate the scheme in the context of such architectures and to study the scheme in more detail to be able to estimate its yield accurately.

An FPGA can be organized into row-wise or column-wise groups [3, 27, 34]. The column-wise organization can be shown in Figure 2.5a. The row replacement scheme is similar, so it is not separately discussed.

Segments must be extended by one extra column so that a faulty column can be skipped. As an example, the FPGA with $S2$ and $S4$ in Figure 2.5, which has one extra column, contains segments with physical lengths of three and five. After fabrication, if the 4th row of a given chip is faulty, the chip has to be programmed by the manufacturer so that the 4th row is skipped as shown in Figure 2.5b. However, users can still use the chip as if it had 8 consecutive working columns with segments of lengths 2 and 4. It is important to note that the amount of reconfiguration needed for such a chip is minimal.

To simplify the discussion, let us assume that the FPGA architecture has only

[1]This scheme has been successfully used for commercial FPGAs, although on different architectures from what we consider here [34, 39].

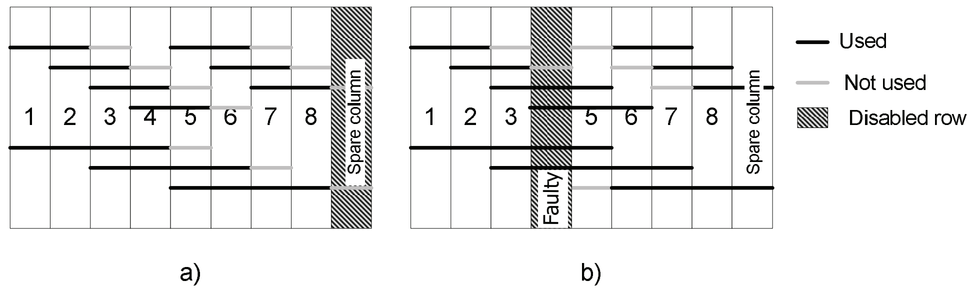

Figure 2.5: Spare column scheme [3]. a) without defective column. b) with one defective column.

segments of length four. Since segments of different types are independent, this assumption just makes the discussion simpler without losing the generality of our modeling framework. A typical routing architecture, consisting of only segments of length four and channel width of four is shown in Figure 2.6. Only wires going in one direction within each column and row similar to those in Figure 2.4 are shown in the figure. The fabric is designed to tolerate one faulty column. Thus, each segment is extended by one unit[2], shown in thick lines. Recall from Figure 2.3d that within a SB, segments that end at the SB can be connected to segments that start at the same SB. They are shown by four connected circles. Furthermore, the middle of segments at the SB can be accessed and connected to segments starting at the SB to perpendicular directions using switches called Middle-point switches, shown by squares. Figure 2.6a shows switches inside a SB when no columns are faulty. In this state, the extended parts of wires are not used.

[2]We cannot just design the architecture as it contains segments of length 5 because if there is a connection connecting two nodes at both ends of segment of length five, the placement and routing algorithms should not consider that to be a segment available to normal use. In the case that a column fails in between any connection of length one through four can be mapped to this segment of length five to compensate the fault.

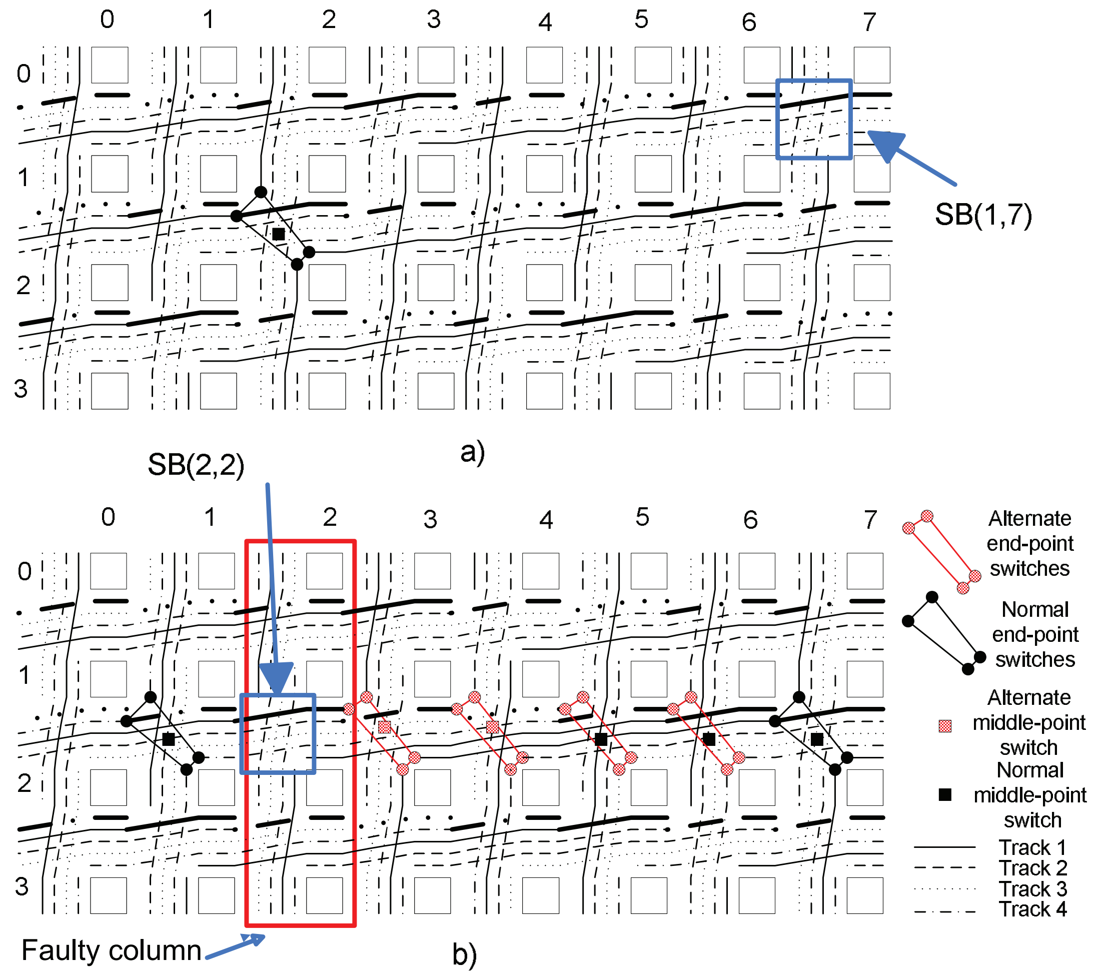

Figure 2.6: Details of typical routing resources. a) no faulty columns. b) Column 2 is faulty.

A column will be considered as faulty if 1) any CLB in that column fails, 2) any vertical and/or horizontal segments starting at a CLB in that column fails. If a column fails, all segments passing through that column have to be extended and the segments starting at that column have to be ignored. Let $SB(x, y)$ denote a switch box in row x and column y. Assuming that column 2 fails, Track 1 starting from $SB(2, 2)$ in both directions is assumed to fail. Column 3 will assume the role of Column 2. Thus, Track 2 starting from $SB(2, 3)$ will assume the role of Track 1. The end-point switch at $SB(2, 3)$ has to use the extended part of Track 1 from the left to skip Column 2. Furthermore, the middle-point switch at $SB(2, 3)$ which normally connect to Tract 4 has to change to connect Track 3 to imitate the middle-point switch at $SB(2, 2)$. In terms of programming the FPGA, this scheme helps us tolerate faults at the hardware level and program the chip as if no faults ever occurred.

It is interesting to note that this fabric can tolerate more than one faulty column. As seen above when Column 2 fails, switches of Column 3 to 6 are used to hide the failure. Thus, if Column 7 fails too, the same technique can be applied to skip Column 7 in addition to Column 2. But, we cannot skip Column 6 because we need to connect an extended segment which ends at column 6. In other words, Column 6 is in the same region of influence of the extra track length as in Column 2. Therefore, to tolerate m faulty columns, the above fabric can be used instead of extending routing segments by m units and m alternate states for end-point and middle-point switches which introduce much more area overhead.

It is obvious that both spare column and spare row can be used in an architecture. However, configuration shifting in such an architecture is more complicated than an architecture with two spare columns. Thus, providing both spare column and row will not be discussed further in this thesis and is left as future work.

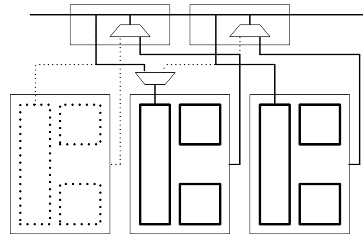

Figure 2.7: Shifted replacement of CLBs [4]

Grouping

If a group of n CLBs is connected to horizontal wires through m connection points, ($m < n$), faulty CLBs can be replaced by spare ones [4]. Figure 2.7 shows a group of three CLBs in which the leftmost CLB is a spare; only two CLBs will be used. Thus, there are only two connection points connecting them to the passing wires. However, at each connection point, there are multiplexers that can be used to select redundant input and output if the primary CLB fails.

Normally a tile contains one SB and a group of CLBs. Applying the grouping scheme to Virtex-like architecture defined in Section 2.2, a SB can be used as a connection point for the group. But, a super tile containing a group of CLBs and a group of SBs will be used instead because the numbers of SBs and CLBs are different.

Node Covering Scheme

In the node covering scheme, spare CLBs are provided similar to the spare column scheme. The differences between the two methods are as follows : 1) In the spare column scheme, all CLBs (from all rows) in a column will be skipped to tolerate defects that appear in a subset of CLBs in the column. But in the node covering scheme, each row can independently skip a CLB to tolerate defects appearing in

different columns. Therefore, vertical connections must remain fixed to maintain connectivity. 2) Segments are extended by one unit to facilitate column skipping in the spare column scheme. However, in the node covering scheme, the routing algorithm has to reserve segments to be used when some CLBs are skipped [29].

When the node covering scheme is applied to XC-4000-like architectures, 30% more segments are required [29]. Although the overhead is lower when it is applied to our architecture, it is difficult to be integrated into a multi-length segment architecture. Therefore, it does not provide more benefits than the grouping scheme discussed in the last section.

Clustering Scheme

Recent FPGAs incorporate a number of LUTs into a CLB as shown in Figure 2.2a. An input of a LUT is selected from all incoming inputs from the SB or feedback signals from other LUTs [2, 40] as shown in Section 2.2. As BLEs are densely connected inside a CB, extra BLEs can be provided and the shifting replacement scheme can be used to tolerate defects in some BLEs. As the number of BLEs within a CLB increases, having spare BLEs within a CLB is better than the grouping scheme as it allows for a tile-based design with smaller tile and requires less area overhead as MUXes at connection points are not necessary.

Shifting Scheme for SB and Routing Segments

Recently the approach of using shifting to tolerate faults both in SBs and routing segments has been shown to be efficient [35]. As an example, assume that there are three tracks and three MUXes in a driving group. Two defect-tolerant SBs can be shown in Figure 2.8. Two sets of MUXes have been added to a driving group : IMUX and OMUX [35]. If the middle track fails (or both MUXes B and

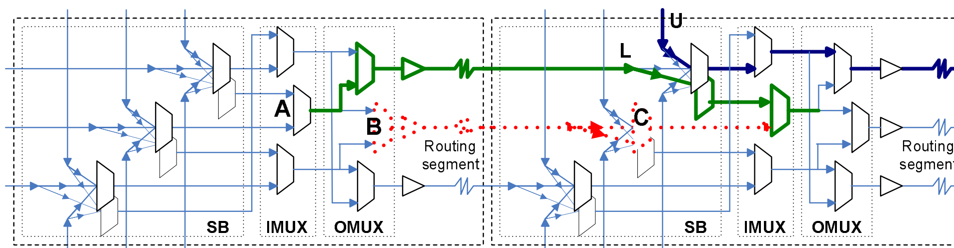

Figure 2.8: If a track, its driver or receiver fails, OMUX and IMUX can shift the track to avoid the defect.

C fail) and its adjacent track works, an incoming signal can be rerouted to avoid the faulty track as shown in Figure 2.8. However, two copies of SB MUXes are required to avoid contention after shifting (*e.g.*, signals L and U in Figure 2.8).

If some MUXes from IMUX and OMUX fail, (*e.g.*, MUXes A and B), the shifting scheme as shown in Figure 2.8 cannot be used to avoid the middle track failure. Thus, shifting has to be performed at a farther SB. Consequently, reconfiguration to tolerate defects on one routing channel depends on whether defects appear on other channels or not. Therefore, shifting becomes complicated and requires a complex reconfiguration scheme. As a result, this type of SBs can be used for the customization approach, but it is not ideal for a defect tolerant architecture.

In our defect-tolerant routing architecture, extra tracks are provided and shifting to avoid defective tracks must be performed by simple reconfiguration, in contrast to software intervention used in [35]. To better describe our proposed fault tolerant architectures, we define the term *dependency* as follows: two sets of routing resources (e.g., tracks entering the left side of a SB, and tracks exiting it from the right) are said to be independent if any combination of $n-r$ fault-free elements from the first set can be routed to any combination of $n-r$ fault-free elements of the second set, where r and n are the maximum tolerable faulty elements, and the maximum number of resources respectively. For example, in a dependency-free

SB with $n = 3$ tracks that can tolerate $r = 1$ faults, any two fault-free tracks entering the SB from left can be configured to connect to any permutation of two fault-free tracks exiting the SB from right. It is easy to show that permutation significantly improves fault tolerance. If no permutation is allowed, then only one bad segment on any given track of a disjoint SB architecture is enough to render all segments in that track useless for a long connection. However, if the segments are independent, then a long connection could be configured to use a track even if multiple segments on that track are faulty, by temporarily switching to other tracks wherever the segments of the original track are faulty.[3]

We propose four types of defect tolerant SBs that break dependencies between routing tracks and SBs (Figure 2.9). In the figure, there are three tracks, but one of them is a spare. Shuffle MUXes are responsible for breaking dependencies between SB Muxes and routing tracks. Using Figure 2.9a as an example, if either the shuffle MUX of Track O2 or the track itself fail, the track becomes unusable. However, any two signals among I1,I2 and I3 can be connected to O1 and O3. Therefore, shuffle MUXes make their outputs independent from their inputs.

Figure 2.9a shows the simplest version of our proposed architecture in which shuffle MUXes are inserted to break dependencies between SB MUXes and tracks. Thus, dependencies will be limited to only SB MUXes and its immediate incoming tracks. We call this defect tolerant SB a TypeA SB. Although requiring a small area overhead, it has one constraint. If the MUX driving I2 fails[4], all incoming tracks to the MUX are not usable. In other words, all incoming tracks from different channels must be shifted together, indicating dependency among them.

[3]In customization approach, if some track 1 are faulty, other track 1 can be used on chip-by-chip basis. But, our goal is to provide a fault-free chip. Thus, all track 1 have to be fault-free assuming disjoint SBs are used.

[4]This is equivalent to the case that an incoming track of the MUX is faulty.

TypeA SBs reduce dependency to only incoming channels. However, S_2 and S_6 also exhibit dependencies through middle-point connections. As a result, the region of dependency when applying TypeA SBs to such segments is still large. Shuffle MUXes can also be inserted between middle-point segments and the SB to break such dependencies. Since there are at most four middle-point channels, four sets of shuffle MUXes are inserted and they are shared among four driving groups. The resulting SB, call a TypeB SB, is shown in Figure 2.9b.

If the yield of tracks is low, it is beneficial to add shuffle MUXes to separate incoming channels from neighboring SBs, similar to those used to separate an SB from middle-point channels. The resulting SB is called a TypeC SBs, shown in Figure 2.9c.

We confine ourselves to using simple reconfiguration for the proposed SBs above. However, it is useful to study the effect of a more complex reconfiguration. If SB MUXes in Figure 2.9b are combined with shuffle MUXes, we obtain a TypeD SB as shown in Figure 2.9d. Note that middle-point tracks connect to another level of shuffle MUXes to reduce the number of inputs to the shuffle MUXes feeding the routing segment buffers. If a complex reconfiguration is used in a TypeD SB, all routing channels connecting to the SB are independent.

As seen in Section 2.2, S_∞ are bidirectional and are connected through back-to-back tri-state buffers as shown in Figure 2.3e. We can replace the set of these buffers with back-to-back shufle MUXes to enable a shifting mechanism.

All defect tolerant SBs proposed above can tolerate only one segment failure. To tolerate r faulty segments, a shuffle MUX used in each defect tolerant type has to be replaced with a MUX with $2r + 1$ inputs, one from the same track, r from the tracks above and r from the tracks below.

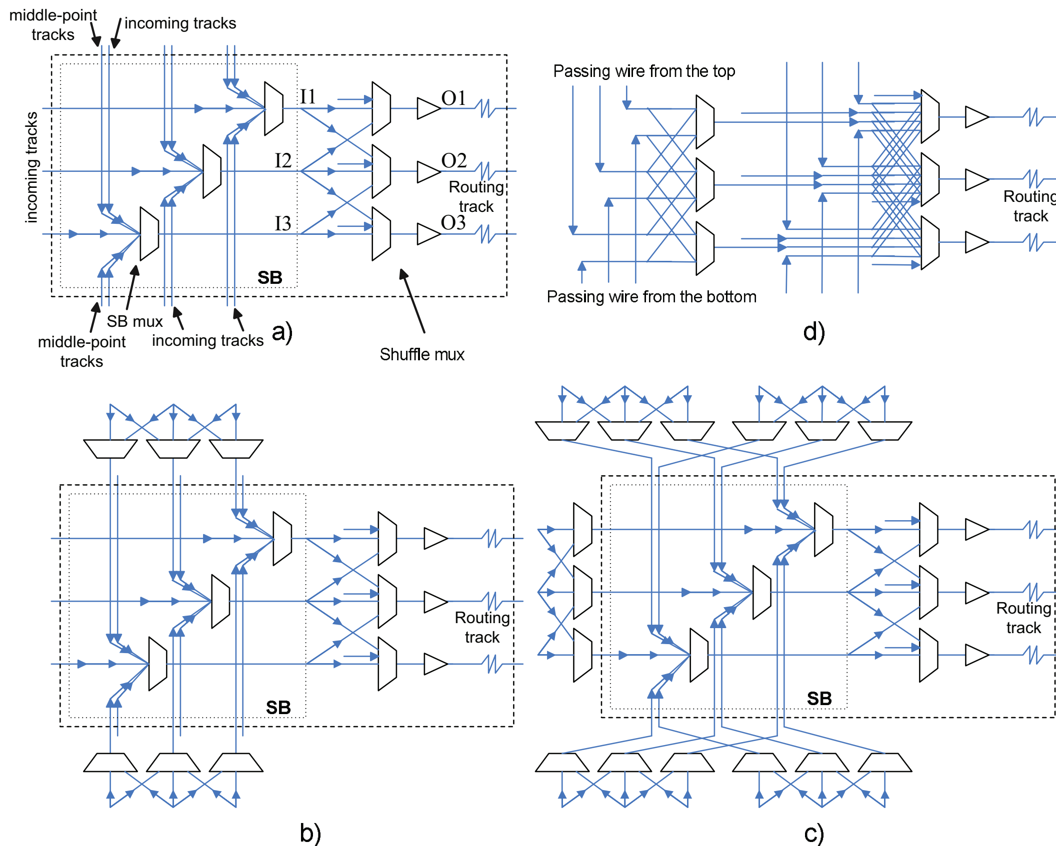

Figure 2.9: Defect tolerant SBs. A SB has four identical driving groups, but only one such group is shown. A driving group for S_2 or S_6 has four additional inputs from vertical and horizontal middle-point segments, but only two such inputs are shown for simplicity. Note that a shuffle MUX also has inputs from CLB, shown as one arrow for one MUX to keep the figure simple. a) Shuffle MUXes are added to a set of SB multiplexers b) Shuffle MUXes are added to four middle-point channels shared among four driving groups. These MUXes remove dependency propagated through the middle connection. c) Shuffle MUXes are also added to 4 incoming channels, removing dependency among 4 incoming channels. d) SB MUXes are merged with shuffle MUXes making incoming tracks from different directions independent.

2.4 Yield Estimation

2.4.1 Basic Yield Computation

The number of faults depends on both the circuit structure and defect sizes. Defect size can be described as a probability distribution function [41]

$$f_d(x) = \frac{2x_0^2 x_M^2}{(x_M^2 - x_0^2)x^3} \quad if x_0 \leq x \leq x_M \tag{2.1}$$

$$= 0 \quad otherwise. \tag{2.2}$$

where x_0, x_M are the minimum and maximum defect sizes, respectively. For a given defect size x, the area in which defects of type[5] i will cause faults if they appear within the critical area, $A_i^{(c)}(x)$. The effective critical area $A_i^{(c)}$, which quantifies the circuit layout sensitivity to defects of type i, can be described as :

$$A_i^{(c)} = \int_{x_0}^{x_M} A_i^{(c)}(x) f_d(x) dx \tag{2.3}$$

Note that $A_i^{(c)}(x)$ is a function of defect sizes, while $A_i^{(c)}$ is not. Let λ and d_i be the average number of faults on the chip and of defects of type i per unit area, respectively. We have $\lambda = \sum A_i^{(c)} d_i$. Assuming infinite independent regions, we obtain the Poisson distribution of random variable X as [41] :

$$P\{X = k\} = \frac{e^{-\lambda}\lambda^k}{k!} \tag{2.4}$$

where k is a constant. Therefore, the chip yield is :

$$P\{X = 0\} = e^{-\lambda} = \prod_i exp(-A_i^{(c)} d_i) \tag{2.5}$$

To capture fault clustering, we represent λ as a Gamma distribution with parameter $(\alpha, \alpha/\lambda)$. Integrating over possible defect sizes, we have a negative binomial

[5] Different defects can be classified as different types if they occur independently.

yield formula [41] :

$$P\{X = k\} = \frac{\gamma(\alpha + k)}{k!\gamma(\alpha)} \cdot \frac{(\lambda/\alpha)^k}{(1 + \lambda/\alpha)^{\alpha+k}} \tag{2.6}$$

Note that α is a clustering parameter ranging from 0.3 to 5, in practice. ITRS uses $\alpha = 2$. Thus, the chip yield can be computed as :

$$P\{X = 0\} = (1 + \lambda/\alpha)^{-\alpha} \tag{2.7}$$

It is important to note that yield computation can be decomposed into different independent layers and different independent sub-areas.

Consider a set of straight wires laid out as a bus. Let w and s be the wire width and spacing between adjacent wires, respectively. The probabilities that a wire will be open or short are [42] :

$$\theta_1 = \frac{x_0^2}{w(2w + s)}, \quad \theta_2 = \frac{x_0^2}{s(2s + w)} \tag{2.8}$$

The probability of a wire being neither open nor short to its neighbor is :

$$\left\{1 + \frac{d_1}{\alpha_1}\theta_1(w + s)L\right\}^{-\alpha_1} \times \left\{1 + \frac{d_2}{\alpha_2}\theta_2(w + s)L\right\}^{-\alpha_2}, \tag{2.9}$$

where α_1 and α_2 are clustering parameters of open and short defects, respectively.

2.4.2 Yield of Each Component

As seen in Section 2.2, an FPGA contains CLBs, routing segments, IO blocks as well as configuration memory. Since the two major FPGA components are CLBs and routing resources, their yield computation will be discussed in this chapter. The data will be used in computing the yield of each replacement scheme in Section 2.4.3. Memory elements in a configuration memory will be considered as part of CLBs and routing resources. We assume that global parts, *i.e.*, clock

buffer and set/reset logic are fault-free. This assumption should not affect the results much as their area accounts for only about 3% of a CLB containing eight BLEs. IO blocks will not be considered here. However, for a given yield numbers of these components, the overall yield is the product of them and those of CLBs and routing resources considered here. Thus, the overall yield trend should not change, but may be scaled by a small amount.

CLB Yield

Each cluster contains $N + R$ BLEs, in which R of them are spares. Therefore, at least N BLEs must be fault-free to make the cluster usable. The probability that m blocks of a cluster are fault free is :

$$y_m = \left(1 + m\frac{\lambda^b}{\alpha}\right)^{-\alpha} \tag{2.10}$$

where λ^b is the average number of faults in one BLE. Let the probability that exactly m out of M modules are working be F_m^M. By the Inclusion-Exclusion principle, we have [41] :

$$F_m^M = \begin{pmatrix} M \\ m \end{pmatrix} \sum_{k=0}^{M-m} (-1)^k \begin{pmatrix} M - m \\ k \end{pmatrix} y_{m+k}, \tag{2.11}$$

Therefore, the yield of a cluster, known as the M-out-of-N yield model, is :

$$Y = \sum_{i=N}^{N+R} F_i^{N+R} \tag{2.12}$$

Details of a typical cluster are shown in Figure 2.10 [2]. The circuit area is measured by the number of minimum size transistors, tr_{min} whose area can be found from the ITRS roadmap for a given technology. For a given number of BLEs and a number of inputs to a CLB, the CLB area can be computed using information from Table 2.1.

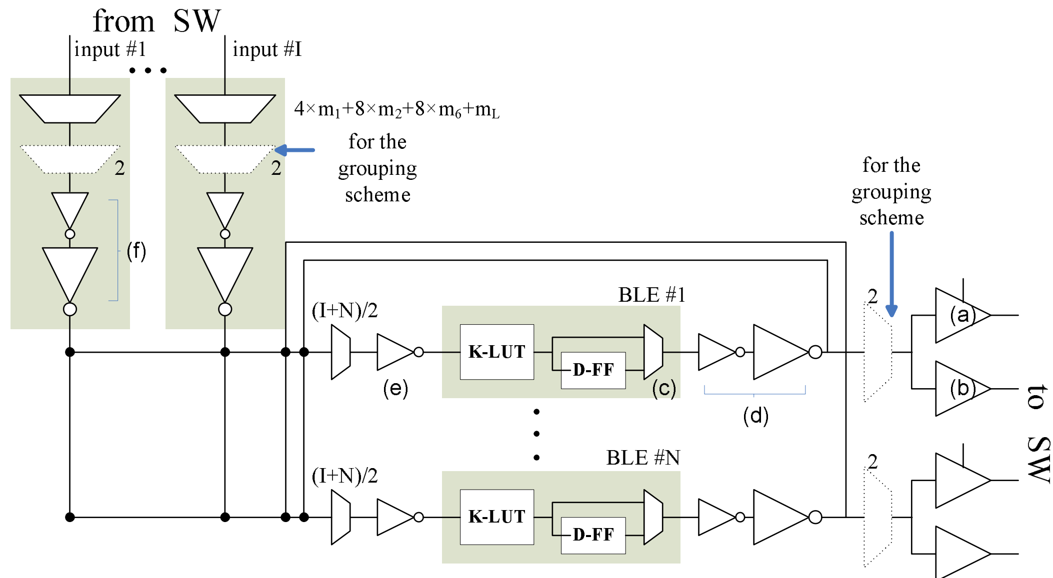

Figure 2.10: Details of a configurable logic block containing N LUTs. There are I inputs to the CLB. Internal connection is assumed to be 50% sparse. MUXes drawn using dotted lines are for the grouping scheme and do not exist in other schemes. The number of inputs to each MUX is annotated next to it. Labels in parenthesis correspond to items in Table 2.1. Note that if there are spare segments, the total segments of length i including spare ones must be used instead of m_i.

For a cluster of N BLEs with R spare BLEs, only $2(N + 1)$ inputs out of $2(N+R+1)$ inputs and N outputs out of $N+R$ output are required. We assume that the layout is uniform. Hence, CLB yield can be computed proportionally to its area. Inputs to a BLE connect to incoming inputs from an SB in similar way as cross bar switches. Thus, they are independent. As a result, a cluster yield can be computed in two parts: 1) inputs to a CLB and 2) BLEs, shown in Figure 2.10, each by using Eq. (2.12). Finally, the cluster yield is calculated as their product.

Table 2.1: Sizes of each active component [2]. Items annotated with a through f correspond to elements in Figure 2.10. Others are for elements in an SB.

Component	size(min Tr)
n-input MUX with SRAMs	$1.2\lceil log(n) \rceil + 2n - 2$
output driver of MUX in SB	8.45
sense buffer of SB for one track	2.70
tri-state buffer with SRAM	14.90
a) tri-state CLB output to long wire (shared buffer)	51.40
b) CLB output	42.40
c) BLE	194.00
d) BLE output driver	9.25
e) input buffer for each LUT input	2.35
f) driver for local routing	2.35
input buffer	9.15

SB Yield

A SB can be decomposed into four driving groups, one such group is shown in Figure 2.9. A shuffle MUX of a track, a track driver and a track itself are considered as one unit for the track's yield computation. Consider a SB MUX in Figure 2.9a. If one track fails, the remaining tracks of all other three directions have to work to make the FPGA work. Thus, there is dependency between adjacent routing channels. Therefore, the yield of an SB cannot be computed in isolation from routing resources and other SBs. Thus, SB yield will be revisited in the next section when routing yields are computed.

Routing Yield

In general, FPGAs are implemented in a tile-based fashion, in which one tile contains one logic cluster, one switch box and routing channels. Therefore, it is reasonable to assume that segmented wires are laid out in a straight manner as a bus[6]. However, approximating only the yield of a bus as done in [42] is not sufficient for the purpose of FPGA yield estimation because of the interactions between several routing channels and SBs.

Routing tracks of the same type can be laid out next to each other. However, if there is a bridging fault, a channel with one spare track would fail. Thus, it is beneficial to interleave tracks from different types. In this case, one bridging fault will translate to one fault for each type of segment. If different segment types are routed in separate layers, segments from opposite directions can be interleaved instead. We use m_i and r_i to denote the number of segments available to users and spare segments of length i starting at a SB, respectively.

[6]Even though the practical FPGA interconnect layout is staggered in nature, wires are staggered at the end. Therefore, it minimally affects our model.

Because we use disjoint switch boxes in our architecture, segments of different types can be considered separately in our yield estimation studies. Still, yield computation is complicated because of the interaction of elements associated with the same segment type. We call the set of SBs and segments of length L a *network*, denoted by N_L. The network of long wires is denoted as N_∞. For simplicity, let us assume for a moment that there is no middle-point connection. Hence, a segment of length L originating from $SB(a, b)$ would connect to segments starting at $SB(a \pm k \cdot L, b \pm k \cdot L)$, for $k \geq 1$, through k segments. Without the middle-point connections, there are L^2 independent networks within N_L. Each of them is denoted by $N_L(i, j)$, where i, j are the coordinate of its top-left SB within N_L. We also use $SB_{i,j}(x, y)$ to denote the SB with coordinate (x, y) relative to the top-left SB of $N_L(i, j)$. If we consider middle-point connections, the segments behave as segments of length $L/2$. Thus, there are $L^2/4$ independent networks for segments of length L. For example, there are nine independent $N_6(i, j)$, $0 \leq i, j < 3$.

Before we proceed, let us introduce some equations to simplify discussions of network yield computation. Consider elements arranged into a 2-dimensional array of width W and height H. Let the yield of each element be Y_e. A column is considered to be working if all elements in the column work. If each element is independent, the probability that there is exactly one faulty column is :

$$\begin{aligned} P_{1\mathrm{f}}(Y_e, H, W) &= Y_e^{H(W-1)} \cdot \sum_{i=1}^{H} \binom{H}{i} Y_e^{(H-i)} (1 - Y_e)^i \\ &= Y_e^{H(W-1)} (1 - Y_e^H) \end{aligned} \tag{2.13}$$

In general, the probability that exactly k columns fail – or exactly $W - k$ columns work – is

$$P_{kf}(Y_e, H, W) = \binom{W}{k} \cdot Y_e^{H(W-k)} (1 - Y_e^H)^k \tag{2.14}$$

If there are two types of elements in a column, the probability that a column works is $Y_{e_1}^{H_1} \cdot Y_{e_2}^{H_2}$, where Y_{e1} and Y_{e2} are yields of each type and H_1 and H_2 are the number of elements of each type in a column. Therefore, (2.14) can be generalized to

$$P_{kf}(Y_{e_1}, H_1, Y_{e_2}, H_2, W) = \binom{W}{k} \cdot \left\{Y_{e_1}^{H_1} \cdot Y_{e_2}^{H_2}\right\}^{(W-k)} \cdot (1 - Y_{e_1}^{H_1} \cdot Y_{e_2}^{H_2})^k \quad (2.15)$$

Note that (2.14) can be extended similarly for more than two types of elements.

Throughout this section, $Y_m(n)$ is the probability that one n-input MUX works and $Y_t(n, l)$ is the probability that a track unit, including an n-input shuffle MUX, a track driver and a wire of length l, works.

Network Yield Computation of Segments of Length One

For S_1, there is only one independent network. The network can be abstractly drawn as shown in Figure 2.11. Each circle represents one driving group with an arrow next to it indicating the direction it drives. Tracks coming from four directions are represented as incoming arrows. Note that the beginning of each arrow is not connected to its driving group, reflecting the fact that shuffle MUXes and associated tracks are independent from its originating SB as discussed in Section 2.4.2. A driving group of SBs in the middle of an FPGA has three incoming routing channels, each is represented by an incoming line. A driving group of SBs around the boundary may have fewer incoming routing channels.

Each type of defect tolerant SB shown in Figure 2.9 will be considered separately. But S_1 has no middle connections and they are relatively short. Thus, TypeB and TypeC SBs are not considered.

TypeA defect-tolerant SBs

Let us consider an SB in the FPGA which is not at a corner. We can see from

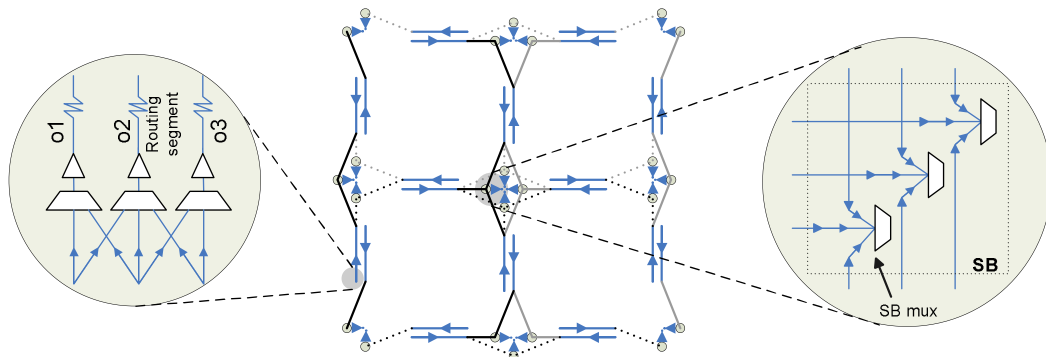

Figure 2.11: The abstract view of a network of segments of length one containing 3x3 SBs.

Figure 2.9a and 2.11 that track i is not usable if a MUX of track i in any driving groups or track i of any incoming channels fail. Therefore, MUXes of track i and track i constitute one column (in the context of (2.13)) and the yield of a SB is

$$Y_{SB} = \sum_{k=0}^{r} P_{kf}(Y_t(2r+1+Q,1), j, Y_m(3), j, m_1+r) \tag{2.16}$$

where $j = 3$ for a border SB, and 4 otherwise and Q is the number of CLB's outputs per track.

There are only two incoming channels for the corner SBs. Although these SBs are physically the same as those in the middle due to the tile-based design, only two driving groups are used and, therefore, considered in yield computation. Note that even though only one out of several inputs of a MUX is used, we assume that the MUX fails even if defects appear on those unused inputs. Thus, the yield of each corner SB is

$$Y_{SB} = \left\{ \sum_{k=0}^{r} P_{kf}(Y_t(2r+1+Q,1), 1, Y_m(3), 1, m_1+r) \right\}^2 \tag{2.17}$$

The yield of the network is

$$Y_{N_1} = \prod_{\text{all SB}\in N_1} Y_{SB} \tag{2.18}$$

TypeD defect-tolerant SBs

Since complex reconfiguration can be used, failed tracks in one incoming channel do not pose a constrain on the others when TypeD SBs are used. Furthermore, driving groups are merged with shuffle MUXes. Therefore, instead of computing the yield for each SB, the yield of each channel is computed as

$$Y_{channel} = \sum_{k=0}^{r} P_{kf}(Y_t(p(2r+1)+Q,1),1,m_1+r) \tag{2.19}$$

where p is the number of inputs to a SB MUX, which is 3 in this case. The yield of the network is

$$Y_{N_1} = \prod_{\text{all channels}\in N_1} Y_{channel} \tag{2.20}$$

Network Yield Computation of Segments of Length two and six

$N_L, L = \{2, 6\}$ can be divided into $L^2/4$ independent networks, one of which is abstractly shown in Figure 2.12 for routing channels with 2 tracks, one of which being a spare. Each driving group has three incoming and four middle-point channels. However, these details are omitted from the figure for simplicity. N_6 can be decomposed into nine independent networks, each of which behaving similar to N_2. Therefore, their yield can be computed in the same way as N_2 and will not be discussed further.

Before we proceed, it is important to note the difference between physical and logical segment lengths. Consider a S_2 starting at $SB(0,1)$ going toward $SB(0,0)$. Although logically it is a segment of length two, its physical segment length is one.

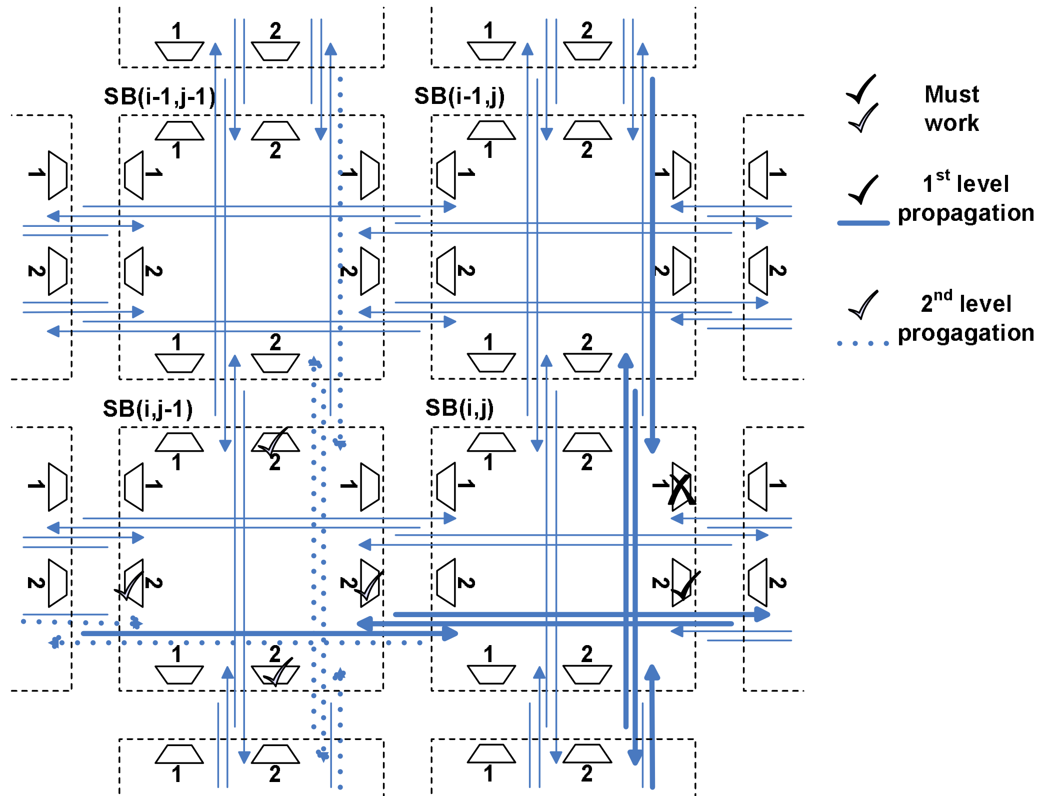

Figure 2.12: Abstract view of one independent network out of $L^2/4$ for segments of length L. $L = 2$ is used in the picture. The following details in a SB are omitted from the picture for simplicity : 1) middle-point wires of track i connecting to all MUXes of track i, 2) connections from a MUX of track i to incoming wires of track i from three directions.

We use C_L to denote a channel containing segments of physical length L, but their logical length depends on the context.

TypeA defect-tolerant SBs

If an SB MUX of a track fails, for example that of Track 1 of $SB(i,j)$, the other SB MUXes (Track 2 in our example) have to work. As a result, it is required that particular incoming and middle-point tracks, as shown by thick lines in Figure 2.12, should work. The constraint is further propagated to $SB(i,j-1)$ through the middle-point tracks at $SB(i,j)$. At $SB(i,j-1)$, as the middle-point track number i connects to the i SB MUXes of all driving groups, the constraint propagates to all of them and then along all incoming and middle-point tracks, as shown by dotted lines in Figure 2.12. The situation at $SB(i-1,j-1)$ is the same as that of $SB(i,j-1)$ just considered. Therefore, the constraint is propagated to $SB(i-1,j)$ and throughout the network in the same manner. In short, if one SB mux of track i fails, all SB MUXes and segments other than track i must work to make the network operational. Hence, the probability that the network works is

$$Y_{N_2} = \sum_{k=0}^{r} P_{kf}(Y_{t1}, C_1, Y_{t2}, C_2, Y_m(7), C_1 + C_2, m_2 + r), \tag{2.21}$$

where $Y_{t1} = Y_t(2r+1+O,1)$ and $Y_{t2} = Y_t(2r+1+O,2)$.

TypeB defect-tolerant SBs

TypeB SBs remove dependency of middle point connections. Therefore, N_2 can be decomposed into four independent networks: $N_2(i,j), i,j \in \{0,1\}$. The yield of each of them can be computed similar to that of N_1 with TypeA SB with the following differences:

1. Middle-point shuffle muxes are shared among several driving groups in the same SB.

2. A SB near the boundary of a network may have to drive a channel with

shorter physical length. For example, $SB_{1,1}(0,0)$ drives C_1 to $SB_{0,1}(0,0)$ and $SB_{1,0}(0,0)$, which are $SB(0,1)$ and $SB(1,0) \in N_2$, respectively.

After incorporating the first difference (the second one will be addressed below), the yield of one SB can be computed as

$$Y_{SB} = \sum_{k=0}^{r} P_{kf}(Y_t(2r+1+O,2)^v, 1, Y_m(2r+1)^u Y_m(7)^s, 1, m_2+r) \qquad (2.22)$$

where u, v and s are the number of middle-point, incoming channels and driving groups, respectively. The appropriate values of u, v and s depend on the location of an SB as well as the width and height of an FPGA. As an example, these values for $N_2(0,0)$ when the width and height of the FPGA are odd and even, respectively, are shown in Table 2.2.

Some SB in $N_2(i,j)$, *e.g.*, $SB_{0,0}(0,0)$, behave like a corner SB of N_1 with TypeA SB. Their yields are

$$Y_{SB} = \left\{ \sum_{k=0}^{r} P_{kf}(Y_t(2r+1+O,2), 1, Y_m(2r+1)Y_m(7), 1, m_2+r) \right\}^2 \qquad (2.23)$$

The yield of shuffle MUXes and track drivers of an incoming channel to a SB is computed using Eq. (2.22) and (2.23). However, some channels as described in the second difference above are corner SBs that do not feed incoming channels to other N_2 SBs and hence never be accounted for by Eq. (2.22) and (2.23). Thus, their yield must be computed using (2.19) with $p = 1$ and multiplied with the yields of all SBs to obtain the network yield.

TypeC defect-tolerant SBs

In addition to removing the dependency of passing channels, TypeC SBs also make SBs independent from incoming channels. As a result, yields of routing channels are independent from those of SBs; the yield of the network is the product of that of routing channels and that of SBs. The yield of a routing channel can be

Table 2.2: The appropriate values of u, v and s of $N_2(0,0)$ for an FPGA with odd width and even height.

SB location	u	v	s
top right	use (2.23)		
top left	use (2.23)		
bottom right	3	2	3
bottom left	3	2	3
right border	3	3	3
bottom border	4	3	4
top border	3	3	3
left border	3	3	3
middle	4	4	4

computed by (2.19) with $p = 1$ and using the physical length of the channel for Y_t. The yield of a SB can be computed by

$$Y_{SB} = \sum_{k=0}^{r} P_{kf}(Y_m(2r+1)^{u+v} Y_m(7)^s, 1, m_2 + r) \tag{2.24}$$

The appropriate values of u, v and s are the same as when TypeB SBs are used.

TypeD defect-tolerant SBs

TypeD switches affect N_2 in the same manner as they do to N_1. Hence, the yield of a channel can be computed by (2.19), but with $p = 4$ because there is one input from the shuffle MUX of passing channels. In contrast to N1, there are shuffle MUXes for passing channels. Because there are four sets of inputs for the shuffle MUXes at one SB, their yield can be computed by

$$Y_{SB} = \sum_{k=0}^{r} P_{kf}(Y_m(4(2r+1)), 1, m_2 + r) \tag{2.25}$$

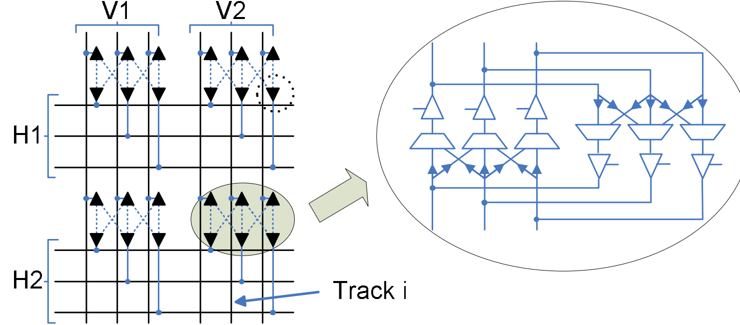

Figure 2.13: Abstract view of a defect tolerant long wire network.

Network Yield Computation for Long Wire Segments

Long wires span the whole width or height of the chip. To reduce area, long wires in each channel are divided into two groups. The first group of the vertical channel will connect to the first group of horizontal channels at their intersection by switches shown in Figure 2.3e. Effectively, long wire segments form two independent networks.

A section of one independent N_∞ is shown in Figure 2.13. Two vertical and two horizontal channels are shown. At the intersections, there are shuffle MUXes, similar to those used in Figure 2.9, in both directions to support bi-directional signals.

If only track i of channel $V1$ fails, there is no constraint on $H1$ or $H2$. However, other tracks of channel $V1$ have to work. With the property of shuffle MUXes, the yield of one column(row) of long wires can be computed as

$$\sum_{k=0}^{r} P_{kf}(Y_L, 1, Y_m(2r+1+O), M, m_L/2+r) \tag{2.26}$$

where Y_L is the yield of a long wire, computed by (2.9) in Section 2.4, M is the number of intersections on the column(row). The yield of the network is the product of yields of all columns and rows.

2.4.3 Yield of Different Replacement Schemes

As discussed in Section 2.3.1 that the clustering scheme requires less area overhead than the node covering and grouping schemes. Thus, only the clustering and spare column schemes will be considered for defect tolerance within CLBs. Defect tolerance of routing segments is performed by employing defect tolerant SBs, shown in Figure 2.9.

Clustering

To employ the clustering scheme, no change is required to the architecture, except adding more BLEs to CLBs. The area for transistors can be computed by applying information from Table 2.1 to compute the area for SBs and CLBs in Figures 2.9 and 2.10, respectively. To compute the area for segments, we assume that the wire width is equal to the minimum width allowed by the technology, but the wire spacing is twice larger than the wire width, providing the minimum coupling capacitances [2]. Let p be the yield of a CLB computed using Eq. (2.10)-(2.12). As CLBs and routing networks are independent, the chip yield can be computed as

$$p^{MN} \cdot Y_{N_1} \cdot Y_{N_2} \cdot Y_{N_6} \cdot Y_{N_\infty} \tag{2.27}$$

where M and N are the width and height of the FPGA array, respectively. Note that Y_{N_L} will be computed based on the redundant SBs that N_L uses.

Spare Row/Column

Referring to Section 2.3.1, in the spare column scheme a column works if all CLBs in the column, all vertical segments in the column and all horizontal segments starting from every CLB in the column work. Let the number of columns be W,

of which one column is a spare. Let P be the yield of one column. The total yield of this scheme can be computed by

$$W \cdot P^{W-1} \cdot (1 - P) + P^{W} \tag{2.28}$$

If both spare column scheme and defect-tolerant routing architecture are used, the total yield computation becomes complicated, making P difficult to be computed. Although in the spare column scheme, some segments are extended and extra hardware is provided to be able to skip some columns creating more physical connections than those of the clustering scheme, the hardware would be used only if there is a faulty column.[7] When FPGAs are used by the users, only $W - 1$ columns will be active. Thus, P^{W-1} can be computed as shown in Section 2.4.2 as if the FPGA contains only $W - 1$ columns. However, the physical length of a segment must be extended by one unit and extended SBs (see Section 2.3.1) must be used as required by the spare column scheme. A similar approach can be used to compute P^q for any value of q.

The modification to the FPGA architecture to incorporate one spare column can also be used to tolerate more faulty columns if enough spare columns are provided as discussed in Section 2.3.1. However, two faulty columns have to be separated by the length of the longest segments (ignoring long wires). Further modifications can also be applied to the architecture to tolerate several faulty columns without the separation limitation. But, it will introduce much more area overhead. Furthermore the spacing required between two faulty columns is only six columns. Therefore, such modifications are not justifiable. Both spare rows and columns can be employed. However, both end-point and middle-point switches must have more alternate configurations. As a result, it requires much

[7]This extra hardware affects the circuit delay. However, its effect is out of the scope of this work.

more area overhead. In addition, employing m spare rows and n spare columns does not statistically beneficial over employing $m+n$ spare columns. Hence, only the spare column scheme is considered.

Let I and k be the number of spare columns and the minimum spacing between two faulty columns, respectively. The yield of the FPGAs is

$$\sum_{j=0}^{I} \binom{W-(j-1)k}{j} \cdot \left\{P^{W-j} \cdot (1-P)^{j}\right\} \tag{2.29}$$

2.5 Simulation Results

In this section we describe how the experimental studies were conducted and present comparisons among different defect tolerant schemes. First we will explain the methodology, followed by comparison results of FPGA architectures using different defect-tolerant schemes will be reported. A MATLAB program was developed to compute the yield of each architecture.

2.5.1 Methodology

Critical defect size and density information for each technology can be found in ITRS documents [17], and be used in interconnect yield computations. However, even though defect density is known, the yield of an active area cannot be accurately estimated as its critical area is unknown due to lack of layout information (See Section 2.4). Therefore, we use the critical area as a parameter to study the yield of non-defect-tolerant architectures. Although the target yield specified in ITRS documents is 83%, we believe that the real yield of current generation FPGAs would be different. Therefore, we performed experiments for target yields of the current technology varying from 50% to 83%, referred to as the reference yield.

We assume that the layout topology of each component of an FPGA remains the same, but its size scales down in future technologies. In general, defect sizes relate to the minimum lithography feature size and therefore scale at the same rate, making critical areas of all components constant. Thus, the average number of faults on each component remains the same (see Section 2.4.1).

We assume that the array size of the current FPGA is 240×108, which is the largest array for Virtex-5 device. From ITRS data, chip sizes at production are constant in a three year cycle. Thus, we assume that future FPGAs will take roughly the same area as the current one, but array sizes will become larger as transistors get smaller. However, we assume that the routing complexity of circuits remains the same as that of circuits today, making the channel width constant over the period of interest.

Because adding redundancy will increase the chip area, there will be fewer chips per wafer. Therefore, to take the extra area into account, we define the effective yield as

$$Y_{eff} = Y_{yield} \frac{N_r^R(H_r^R, W_r^R)}{N_0^0(H_0^0, W_0^0)} \quad , N(H, W) = \frac{\pi R_e^2}{HW} e^{-\frac{H}{R_e}} \tag{2.30}$$

where H, W are the height and the width of the chip, and R_e is the wafer radius [1]. The ratio $N_r^R(H_r^R,\ W_r^R)\ /\ N_0^0(H_0^0,\ W_0^0)$ will always be less than one reflecting the area overhead. Note that we will use *yield* to refer to the yield of the architecture and *effective yield* to refer to yield after taking the area overhead into account.

2.5.2 Results and Discussions

In this section, we first estimate future yields of non-defect-tolerant FPGAs. After that, we calculate the yield for our defect tolerant architectures. As there are several types of FPGA components and each type has many defect tolerance options,

a large number of possible defect tolerant architectures have to be explored. As a result, we decomposed the search for the most effective architecture into three steps: first, defect tolerant schemes for CLBs are explored with no defect tolerance schemes applied to routing networks. Second, defect tolerance schemes for N_2 and N_6 are applied to the top three most effective architectures found from the first step. Finally, defect tolerance schemes for other segment types are investigated in a similar fashion.

Future yields of non-defect-tolerant FPGAs

The estimated future effective yields of non-defect-tolerant architectures are shown in Figure 2.14, its corresponding data are shown as the first value of 3-tuple entries in Table 2.3. Note that in these cases, effective yields are equal to yields obtained from the architectures because these architectures are non redundant (no area overhead). Regardless of reference yields, future yields always decrease and the reduction is more noticeable for low reference yields. For example, if the yield of current FPGAs is 75%, the yield will be only 21% at 2021. As a result, defect tolerant architectures are needed.

Exploring defect-tolerant architectures for CLBs

Defect tolerance schemes for CLBs were applied and their yields were computed and shown in Figure 2.15-2.16. The first item of the x-axis, a bar graph, is for non-redundant architecture, while the rest indicates the number of spare columns used for the defect-tolerant architectures. The y-axis of Figure 2.15-2.16 show references yields, while the z-axis indicate effective yields. As the clustering and spare column schemes can be applied together, for each architecture with a specific number of spare columns, the number of spare BLEs was varied from zero to three.

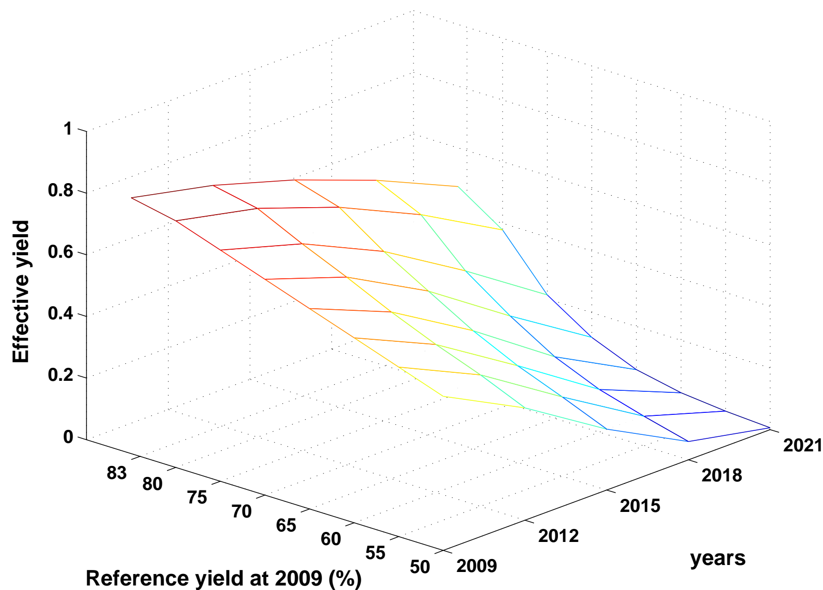

Figure 2.14: Future yields of non-defect-tolerant architectures.

Among these four variants, the maximum effective yield is shown as a point on the 3-dimensional space. The number of spare BLEs that gives the maximum yield were also recorded. However, only zero or one spare BLE are effective. Those with one spare BLEs are highlighted in the bottom of the surface (where $z = 0$), the others are architectures with zero spare BLE. Notice that the effective yields of non-redundant architectures and defect-tolerant architectures with zero spare column are the same only if the number of spare BLEs is zero. As we increase the number of spare columns of the defect-tolerant architectures, its area always increases linearly with the number of spare columns. However, at the same time, the (absolute) yield increase reduces due to diminishing returns. Thus, at some point, the yield increase becomes too small to overcome the area overhead. As a result, maximum points can be observed on the surface, marked by circles. For brevity, only the maximum yields and the corresponding numbers of spare columns are listed in Table 2.3 as the second and third values of 3-tuple entries.

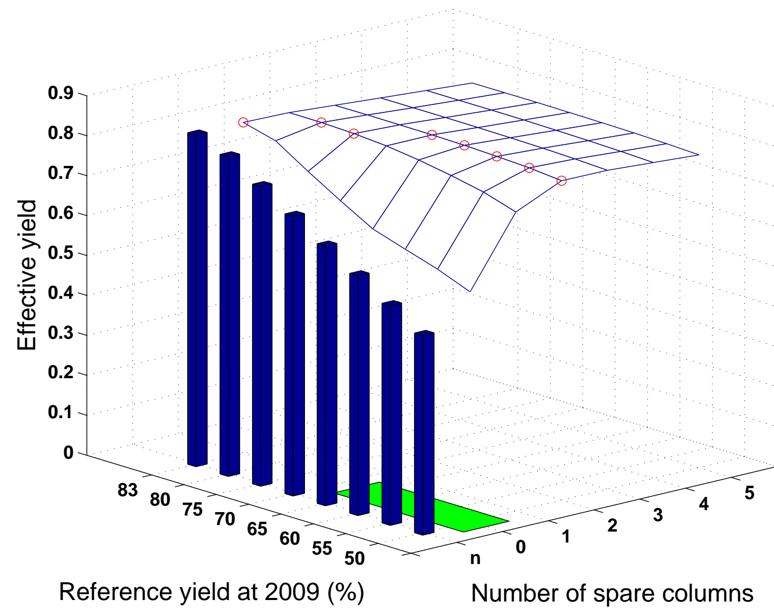

Figure 2.15: Effective yields in 2009 when defect-tolerant schemes are applied to CLBs. The bars show the yield for an architecture with no redundancy.

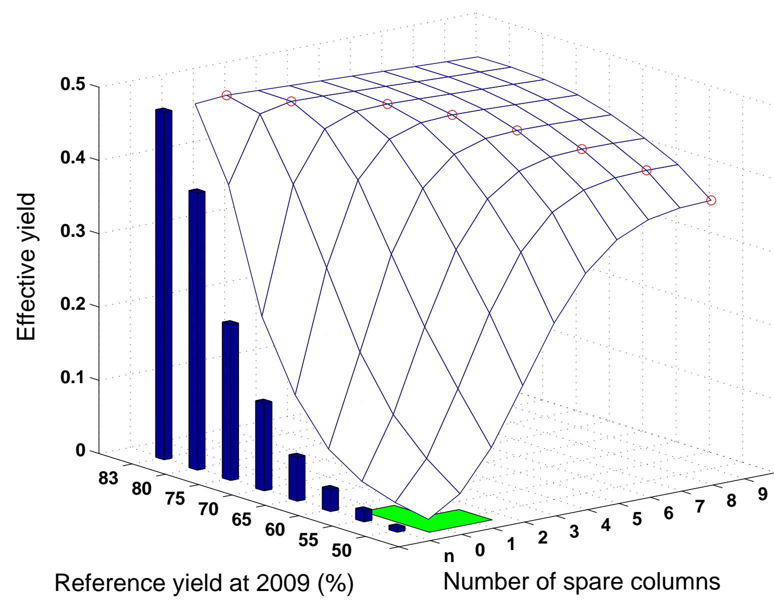

Figure 2.16: Effective yields in 2021 when defect-tolerant schemes are applied to CLBs.

As the results show, defect-tolerant architectures, especially the ones employing the spare column scheme, help improve the effective yield. It can improve effective yields of current FPGAs to more than 80% regardless of the reference yields (see Figure 2.15). More importantly, they can improve effective yields of FPGAs at 2021 from less than 10% to more than 40%. However, the number of spare columns that are required varies with reference yields and technology generations. The number of spare columns increases if (1) the reference yields reduce and/or (2) the feature sizes decrease. For example, the number of spare columns needed increases from two in 2009 to nine in 2021 for the 50% reference yield. Having spare BLEs was observed to be useful when there was not enough spare columns, ie., using one spare BLE was shown to be effective near the corner of both Figure 2.15 and 2.16.

Exploring defect-tolerant architectures for routing networks

Defect-tolerant schemes, as discussed in Section 2.3.1, for N_2 and N_6 were applied to the best architecture for a given technology for each reference yield obtained in the previous section (see Table 2.3). However, the area increase due to extra elements added for defect tolerance to the routing network might alter the best combination obtained previously. Therefore, the second and the third best combinations were also used. We assume that the number of redundant tracks for S_2 are the same all over the chip. The same assumption was also made for S_6.

The extra area (which is also prone to defects) of defect-tolerant SBs of TypeB, TypeC and TypeD is so significant that the yield improvements obtained by using these SBs cannot compensate for the fact that there are fewer number of chips per wafers. Although TypeA SBs require less extra area, they do not provide much improvement due to the interdependence between many SBs. As a result, having

Table 2.3: Yield improvement when defect-tolerant schemes were applied to CLBs.

	Years				
	2009	2012	2015	2018	2021
Ref. yields	basic yields, best yields, num. spare col.	basic yields, best yields, num. spare col.	basic yields, best yields, num. spare col.	basic yields, best yields, num. spare col.	basic yields, best yields, num. spare col.
0.50	0.500 , 0.800 , 2	0.364 , 0.741 , 3	0.198 , 0.653 , 4	0.059 , 0.529 , 6	0.006 , 0.366 , 9
0.55	0.550 , 0.807 , 2	0.427 , 0.748 , 3	0.257 , 0.664 , 4	0.097 , 0.550 , 6	0.015 , 0.401 , 8
0.60	0.600 , 0.812 , 2	0.481 , 0.752 , 3	0.314 , 0.671 , 3	0.139 , 0.563 , 5	0.030 , 0.425 , 7
0.65	0.650 , 0.815 , 2	0.541 , 0.759 , 2	0.382 , 0.678 , 3	0.200 , 0.574 , 4	0.060 , 0.444 , 6
0.70	0.700 , 0.816 , 2	0.609 , 0.763 , 2	0.466 , 0.683 , 2	0.287 , 0.582 , 3	0.120 , 0.460 , 5
0.75	0.750 , 0.822 , 1	0.673 , 0.765 , 2	0.550 , 0.688 , 2	0.389 , 0.588 , 3	0.213 , 0.469 , 4
0.80	0.800 , 0.825 , 1	0.742 , 0.771 , 1	0.648 , 0.692 , 1	0.526 , 0.592 , 2	0.378 , 0.475 , 2
0.83	0.830 , 0.830 , 0	0.772 , 0.772 , 0	0.692 , 0.693 , 1	0.593 , 0.595 , 1	0.475 , 0.478 , 1

Table 2.4: Further yield improvement when defect-tolerant schemes were applied to networks of segments of length one and long wires.

Ref. Yields	Years									
	2009		2012		2015		2018		2021	
	comb.	yields	comb.	yields	comb.	yields	comb.	yields	comb.	yields
0.50	NRC 0,1,2	0.952	NRC 0,1,3	0.948	NRC 0,1,4	0.939	ARC 1,1,6	0.906	ARC 1,1,9	0.827
0.55	NRC 0,1,2	0.959	NRC 0,1,3	0.956	NRC 0,1,4	0.951	NRC 0,1,5	0.929	ARC 1,1,8	0.877
0.60	NRC 0,1,2	0.964	NRC 0,1,2	0.961	NRC 0,1,3	0.959	NRC 0,1,5	0.945	ARC 1,1,7	0.909
0.65	NRC 0,1,2	0.966	NRC 0,1,2	0.967	NRC 0,1,3	0.966	NRC 0,1,4	0.959	NRC 0,1,6	0.936
0.70	NRC 0,1,2	0.968	NRC 0,1,2	0.971	NRC 0,1,2	0.971	NRC 0,1,3	0.969	NRC 0,1,5	0.959
0.75	NRC 0,1,1	0.974	NRC 0,1,2	0.973	NRC 0,1,2	0.976	NRC 0,1,3	0.976	NRC 0,1,4	0.972
0.80	NRC 0,1,1	0.977	NRC 0,1,1	0.979	NRC 0,1,1	0.980	NRC 0,1,2	0.980	NRC 0,1,2	0.981
0.83	NRC 0,1,0	0.979	NRC 0,1,1	0.980	NRC 0,1,1	0.982	NRC 0,1,1	0.983	NRC 0,1,1	0.984

Table 2.5: The effect of removing defect-tolerant SBs for networks of segments of length one.

item no.	year	ref. yield	best combination		best with no DT seg1		dif.
			comb.	yields	comb.	yields	
1	2018	0.500	ARC 1,1,6	0.906	NRC 0,1,6	0.904	0.002
2	2021	0.500	ARC 1,1,9	0.827	NRC 0,1,9	0.813	0.014
3	2021	0.550	ARC 1,1,8	0.877	NRC 0,1,8	0.870	0.007
4	2021	0.600	ARC 1,1,7	0.909	NRC 0,1,7	0.906	0.003

defect-tolerant SBs for S_2 and S_6 does not increase effective yields. For brevity, the data supporting this fact is omitted.

Defect-tolerant SBs were applied to N_1 and N_∞ for each of the top three best combinations from the last section, but none for N_2 and N_6. The combination that provides the best effective yield for each year and its yield is shown in Table 2.4 for each reference yield. A combination in the table is denoted by three letters, $xyz : x \in \{N, A, B, C\}$ indicates the type of defect-tolerant SBs used for S_1, while N signifies non-defect-tolerant, $y \in \{N, R\}$ indicates that normal or defect-tolerant SBs are used for S_∞, and $z = C$ indicates that the spare column scheme is used. The number of spare tracks and columns used are also listed in the table as $a, b, c :$ a and b are the number of spare tracks for S_1 and S_∞, respectively, while c is the number of spare columns. From 2009 to 2015, defect-tolerant SBs for S_1 are not needed and the number of spare columns that give the best effective yields are still the same as those in Table 2.3. However, having defect tolerant SBs for S_∞ improves the effective yield significantly. But, defect tolerant SBs of type A should be applied to N_1 after 2015 only when the reference yield is low.

Although the number of spare columns used increases in future technologies,

it increases at a much slower rate than the total number of columns. When N_∞ is non-redundant, the yield of N_∞ decreases in future technologies. As a result, effective yields decrease as seen in Table 2.3, but when applying defect-tolerant N_∞, the absolute yields (before taking extra area into account) degrade at a lower rate. Therefore, the effective yields may slightly increase for high reference yields.

Adding redundancy not only increases the chip's area but also increases circuit delay. Therefore, it is useful to see the effect of not applying defect-tolerant SBs to S_1. Effective yields of not using defect-tolerant SBs for N_1 were shown in Table 2.5. We can see that by not using defect-tolerant SBs for N_1, the effective yields drop by far less than 2%.

2.6 Summary

Previously proposed defect-tolerant schemes were revalidated for island-style FPGAs. Furthermore, a few additional schemes were also proposed in this chapter. The spare column scheme was shown to improve the yield. Other schemes, such as grouping, node covering and clustering schemes were also considered. Several defect-tolerant switch boxes were proposed with different degrees of absolute yield improvements and area overhead.

Yield estimations for different combinations of these defect-tolerant schemes were also studied. The yield estimation in conjunction with ITRS data was used to show that FPGA yield will drop significantly as technology scaling progresses. However, the spare column scheme can significantly improve effective yields to more than 80% for the current FPGAs, as long as the original yields are no less than 50%. By increasing the number of spare columns, this scheme can greatly help improve yield for future technologies. However, the effective yield still decreases over time.

The results also showed that applying defect-tolerant SBs to the networks of segments of length two and six will not improve FPGA's effective yield as each network exhibits intradependency within its routing structure and breaking this dependency requires a large area overhead. However, when applied to networks of segments of lengths one and long wires, defect-tolerant SBs help maintain effective yields above 80% in all cases. Further investigation suggested that to avoid delay increase over segments of length one, defect tolerant SBs can be used only for long wires, which results in just losing 2% or less of the maximum effective yield.

As the networks of segments of length two and six constitute the majority of an FPGA's area, it may seem counter-intuitive that only the spare column scheme and defect-tolerant networks of long wires can efficiently enhance effective yields. However, the spare column scheme can also cover some failures of routing channels (see Section 2.3.1). But, faults within the network of long wires cannot be covered by spare column schemes as its structure cannot be decomposed into column-wise sub-structures. Undoubtedly, using the spare column scheme and defect-tolerant SBs for the network of long wires provides the most efficient FPGA architecture.

Chapter 3

Yield Improvement by Circuit Rewiring

3.1 Introduction

The ever-shrinking transistor sizes improve device speeds but also impose many challenges throughout the VLSI design flow. Rewiring – which removes a number of wires and adds other wires at different locations in the circuit without changing its functionality – helps improve circuit characteristics at different levels of the design flow such as synthesis, physical synthesis and routing optimization. It allows for the exploration of a larger solution space by changing a circuit structure, hence providing better opportunities for optimization.

Several rewiring techniques have been proposed over the years. Among them, the automatic-test-pattern-generation (ATPG) based approach is the most popular due to its generality and runtime efficiency [43, 44]. More recently, Set-of-Pairs-of-Functions-to-be-Distinguished (SPFD) has been proposed to express a node's function in a circuit [45]. Researchers have used SPFD in rewiring [45, 46] and have shown its advantages both in terms of theory [47] and practice [46] compared to ATPG-based rewiring methods. Applying SAT to an ATPG-based rewiring helps improve its runtime and quality [48], although its quality is not in par with a SPFD-based rewiring [47]. The reason for the lower quality is that the solution space of an ATPG-based rewiring method is a subset of that of an SPFD-based one.

The first proposed SPFD-based rewiring algorithm requires that the destination of the new wire be the same as that of the removed wire, w_r [45]. Later, a technique to expand the destination to dominator nodes of w_r – nodes through which all paths from w_r to any POs pass – was proposed [46]. SPFD-based rewiring can be used in many applications. For example, it has been used to reduce the number of Look-Up Tables used in implementing an FPGA circuit [49] and has also been shown to help reduce FPGA power consumption by 12% [19].

Although SPFD-based rewiring is slower than its ATPG-based counterparts, it can provide more flexibility in exploring function implementation alternatives. One can use a two-tier rewiring scheme in which a quick ATPG-based rewiring is performed first, and then a more powerful SPFD-based rewiring is applied next if the former fails. Apart from slower runtimes, SPFD-based rewiring suffers from high memory requirements too. Memory usage can grow exponentially in the worst case due to its use of BDDs to explicitly represent SPFDs. Circuit partitioning techniques can be employed to curtail high memory requirements, but doing so will hurt the optimization quality because solutions to local subpartitions are not necessarily globally optimal.

The reason that the memory requirement of the existing SPFD-based rewiring techniques can grow exponentially in the worst case is that they use BDD to explicitly represent SPFD. Although the problem can be alleviated by partitioning the circuit, the locality of a partition undeniably impairs the rewiring result. We address this problem by introducing a SAT-based rewiring technique in the first half of the chapter. In our approach, a novel SPFD-based rewiring algorithm is presented that avoids computing SPFDs explicitly. In this algorithm, the feasibility of a rewiring instance is checked by solving a corresponding SAT instance only once. If that rewiring is feasible, then a number of auxiliary SAT instances

have to be solved to find new functions of affected nodes due to rewiring. The proposed algorithm processes one rewiring instance in the order of milliseconds.

Restricting the location of a new wire only to dominator nodes of the removed wire limits the applicability of SPFD-based rewiring in many applications. For example, consider a physical synthesis application in which we want to reduce routing congestion by rewiring. The goal would be to remove wires in congested regions and replace them with wires in non-congested regions. However, if all the dominator nodes are themselves in congested regions, then a dominator-only rewiring approach cannot alleviate congestion. We address this problem by introducing a technique that allows for adding a new wire to non-dominator nodes within a SPFD-based rewiring framework. Compared to an ATPG-based rewiring method, a SPFD-based method carries more information, requiring more computation. Thus, we are not aiming at making SPFD-based rewiring faster than ATPG-based ones. Our goal in the second half of the chapter is to improve the quality of rewiring at a reasonable runtime increase.

This chapter is organized as follows. Section 3.2 summarizes notations used in this chapter. Section 3.3 provides the background for SPFD. The previous SPFD-based rewiring approaches are summarized in Section 3.4. Section 3.5 describes our proposed efficient rewiring algorithm. Section 3.6 introduces the theory allowing rewiring outside dominator nodes. An algorithm for such rewiring and its correctness proof is also included in this section. Experimental results comparing both proposed methods to previous work are reported in Section 4.6. Finally, the chapter is summarized in Section 3.8.

3.2 Basic Terminology

A combinational circuit consists of nodes and directed edges between them. We use the notation (n_a, n_b) to represent a wire w that connects node n_a to node n_b, and n_a and n_b are known as $sr(w)$ and $sk(w)$, (the source and the sink of w) respectively. We also say that n_a is a fanin node of n_b and n_b is a fanout node of n_a. $FI(n)$ denotes the set of fanin nodes or wires of node n depending on the context. $FO(n)$ is similarly defined for fanouts. We use $TFI(n)$ and $TFO(n)$ to represent the sets of transitive fanin and fanout nodes of node n, respectively. Nodes with no fanout and no fanin nodes are called primary output and input nodes, PO, PI, respectively. The function of n in terms of its fanins and PIs is denoted as $f(n)$ and $g(n)$ and are called its local and global functions, respectively. A set of dominator nodes – nodes through which all paths from w_r to any PO pass – for node n is denoted as $DOM(n)$. In a graph context, $V(G)$ and $E(G)$ represent the set of vertices and edges of graph G, respectively.

3.3 Set-of-Pairs-of-Functions-to-be-Distinguished (SPFD)

Any binary function f of n variables can be represented using a complete bipartite graph that contains nodes f_i^{on} that represent the ON set on one side, and nodes f_j^{off} in the OFF set on the other side. Each node in the ON (OFF) set is an n-tuple that corresponds to the input combination that results in f being 1 (0). Edges connect every pair of (f_i^{on}, f_j^{off}) nodes. We can see that there is a one-to-one mapping between the set of n-input binary functions and the set of bipartite graphs. The bipartite graph of XOR can be shown in Figure 3.1. Thus, if the graph of a function is not bipartite, the function is not a one-output binary function. In 1996, Yamashita, *et al.*, generalized this concept by the following definitions [45].

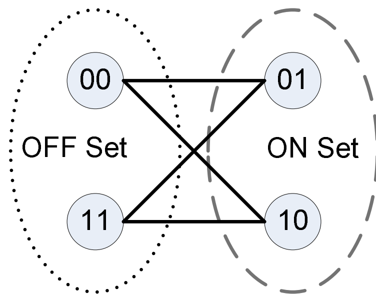

Figure 3.1: The bipartite graph representing XOR.

Definition 3.3.1 *[45] For any two boolean functions, f and g, let $FX = x \mid f(x) = 1, GX = x \mid g(x) = 1$, where x is a vector of primary input values. If $GX \subseteq FX$, then f* includes *g, written as $g \leq f$ or $g \rightarrow f$, which is equivalent to $g \cdot \overline{f} = 0$.*

Definition 3.3.2 *[50] A function f is said to* distinguish *a pair of functions g and h if either $g \leq f \leq \overline{h}$ or $h \leq f \leq \overline{g}$ is satisfied. Note that $g \leq \overline{f} \Leftrightarrow f \leq \overline{g}$ and if $g \cdot h \neq 0$, there is no function that satisfies the pair.*

Definition 3.3.3 *[50] A function f* satisfies *a* set of pairs to be distinguished *$SPFD = \{(g_1, h_1), ..., (g_n, h_n)\}$, if f distinguishes every pair of the set, i.e. $[(g_1 \leq f \leq \overline{h}_1) \vee (h_1 \leq f \leq \overline{g}_1)] \wedge ... \wedge [(g_n \leq f \leq \overline{h}_n) \vee (h_n \leq f \leq \overline{g}_n)]$.*

SPFDs at a node n and a wire (a, b) are denoted as $SPFD(n)$ and $SPFD(a, b)$, respectively. SPFD is in fact a collection of ISFs (incompletely specified functions). SPFD has been shown to express the flexibility of a function better than ISF [50].

Given a circuit and its output functions, the SPFD at an output pin can be constructed from its ON and OFF sets. Nodes will be processed in inverse topological order. At each node, an SPFD edge (an edge in the SPFD graph) will

be distributed to one of its fanins which can distinguish the edge [50]. At the end of this process, each node and edge will be associated with its own SPFD.

Algorithm 1 SPFD computation at a gate.

Require: $SPFD = (f_1^{on}, f_1^{off}), ..., (f_m^{on}, f_m^{off})$. The functions at the gate inputs are $y_1, y_2, ..., y_n$

1: **for** each $(f_q^{on}, f_q^{off}) \in SPFD(f)$ **do**

2: Construct all possible minterms on the inputs of the gate,*i.e.*, $b_0 = 0 \cdots 00 = \overline{y}_1(x) \cdots \overline{y}_{n-1}(x)\overline{y}_n(x), ..., b_{2^n-1} = 1 \cdots 11$.

3: Compute restricted minterms $a_i = b_i(f_q^{on} + f_q^{off})$. The set (f_q^{on}, f_q^{off}) is the care set. Thus, a_i describes all minterms needed to be distinguished.

4: Distribute all care minterms into two sets:

1. $F_1 = \{a_i \mid a_i \subseteq f_q^{on}, a_i \text{ is not constantly } 0\}$
2. $F_0 = \{a_i \mid a_i \subseteq f_q^{off}, a_i \text{ is not constantly } 0\}$

5: Build complete bipartite graph $F = F_1 \times F_0$.

6: **for** each $(a_i, a_j) \in F$ **do**

7: Add (a_i, a_j) to at least one input k based on an arbitrary *distributing order* such that $a_i \leq f_k \leq \overline{a}_j$ or $a_j \leq f_k \leq \overline{a}_i$.

8: **end for**

9: **end for**

The sketch of an algorithm that computes the SPFD at a node is shown in Algorithm 1. For each pair of functions in a given SPFD, *care* minterms are constructed (Line 2-3). b_i is a function (a set of minterms), not just a minterm, because it is a product of global functions. a_i is a set of care minterms that can be distinguished by one fanin. Therefore, a_i can be used in distribution after Line 4, instead of a minterm. Lines 4-5 categorize the minterms into two sets of functions to be distinguished. Each pair of functions (a_i, a_j) will be assigned to a fanin of the gate that can distinguish the pair at Line 7 based on the distributing order. The distributing order can be arbitrarily chosen without affecting the SPFD

correctness, because $SPFD(n) \subseteq \cup SPFD(m), m \in FI(n)$, it is guaranteed that each minterm pair of $SPFD(n)$ can be distinguished by at least one of $FI(n)$ nodes regardless of the order of the inputs. To reduce the number of minterm pairs that propagate in the circuit, we impose an arbitrary distributing order on the fanins of a node, and associate each pair of SPFD only to the smallest order fanin among the ones that can distinguish the pair.

We should note that if functions of some nodes in $TFI(n)$ change, it may make $SPFD(n)$ non-bipartite which cannot be implemented by any one-output binary function. To prevent this, a number of extra SPFD pairs must be added between different ISFs in the SPFD. One simple way to accomplish this is to reduce SPFD to only one ISF by the union operation among ISFs, used in [45, 46].

3.4 Previous work on SPFD-based rewiring

Previous approaches to SPFD-based rewiring can be classified by the number of wires added for a removed wire as follows:

1. *No Additional wire for one wire removal (0-for-1)* If $SPFD(w_r)$ can be redistributed to other fanins of $sk(w_r)$, then w_r can be removed without adding a new wire.

2. *One additional wire for one wire removal (1-for-1)* If another wire has to be added, the rewiring is called 1-for-1 rewiring. Rewiring proceeds by removing w_r and propagating the change through its fanout nodes until a dominator node of $DOM(w_r)$ is reached. A candidate wire w_a will be added as a new fanin of n_a and checked if the resulting $SPFD(n_a)$ covers its original SPFD before removing w_r [46]. If $n_a = sk(w_r)$, it is called a local rewiring, otherwise it is called a global rewiring.

3. *Many additional wires for 1 wire removal (m-for-1)* Even though w_{a1} and w_{a2} may not be used to individually replace w_r, they may collectively substitute w_r. The conditions that are likely to lead to this type of rewiring were suggested in [51].

3.5 SAT-based SPFD rewiring

In this section, we present a novel approach for SPFD-based rewiring that avoids explicit representation of SPFDs[1]. First, we describe previous work using SAT for SPFD evaluation and point to its shortcomings for the rewiring application. Then we will present our novel approach to SAT-based SPFD evaluation, followed by how to build the new rewiring engine.

3.5.1 Previous Work: Using SAT to Compute Minimum SPFD

A set of nodes dividing PIs from POs is defined as a cut $\mathcal{C}$. $\mathcal{C}\backslash n, n \in \mathcal{C}$ is also called a separator $\mathcal{S}$, of n. A minimum $SPFD(n)$ with respect to a separator $\mathcal{S}$ is $SPFD^{\mathcal{S}}_{min}(n) = \cup SPFD(p) - \cup SPFD(q), p \in PO, q \in \mathcal{S}$.

A miter is a circuit that computes $SPFD^{\mathcal{S}}_{min}(n)$. Figure 3.2 shows an example miter [53]. The miter is constructed as follows: (1) duplicate the circuit (we call these two copies *twins*). (The nodes corresponding to $\mathcal{S}$ are shown in bold.) The two sets of primary inputs (X, X') represent a pair of PI minterms, (2) add XORs between corresponding output nodes. If the output of any one of these XOR gates is one, it means that the corresponding output is able to distinguish the input pair, (3) OR the XOR outputs of Step 2 to see if any output distinguishes the pair, (4) XOR the output of each node in $\mathcal{S}$ with the output of its corresponding node in

[1]An earlier version of this part of the work appeared in [52]

the twin circuit, (5) OR these XOR outputs, and (6) AND the OR output of Step 3 and the inverse of the OR gate of Step 5. If the AND output is one, it means that the input pair (x, x') can be distinguished by at least one output but none in $\mathcal{S}$. The miter circuit can be converted to the CNF form and fed to a SAT solver, resulting in a runtime improvement of at least 23x compared to methods that use BDDs [53].

Although using the miter can be useful for some logic synthesis applications, it is not suitable for SPFD rewiring because the propagation of SPFD pairs cannot be well captured by the miter. If an arbitrary node m can distinguish few pairs of $SPFD(n)$, the existence of a path from m to a PO that needs those pairs is not guaranteed. Figure 3.3 shows an example of this shortcoming. Assume that we have computed $SPFD(g)$ and $SPFD(e)$. Based on a miter similar to Figure 3.2, we find $SPFD_{min}^{\{c,d\}}(e)$. Given that according to the definition of $SPFD_{min}^{\{c,d\}}(e)$ in [8] d and c are supposed to cover $SPFD(e) - SPFD_{min}^{\{c,d\}}(e)$, one might attempt to optimize node e to only satisfy $SPFD_{min}^{\{c,d\}}(e)$. However, doing so will result in 001 not being distinguished from 101 and 011, which leads to a wrong $SPFD(g)$. The reason is that $SPFD_{min}^{\{c,d\}}(e)$ is computed ignoring the fact that d is not connected to g. We will address this shortcoming in the next subsection.

3.5.2 A New Distributing Miter

Our goal in this section is to develop an auxiliary miter called *distributing miter* that helps us not only find out the set of pairs distinguished by a node, but also whether a path from such node to output nodes exist. The usage of this miter will be deferred to the next subsection. We will use the example circuit shown in Figure 3.4 for illustration purposes throughout this section.

A *distributing miter* contains two main parts: (1) the twin circuits, and (2)

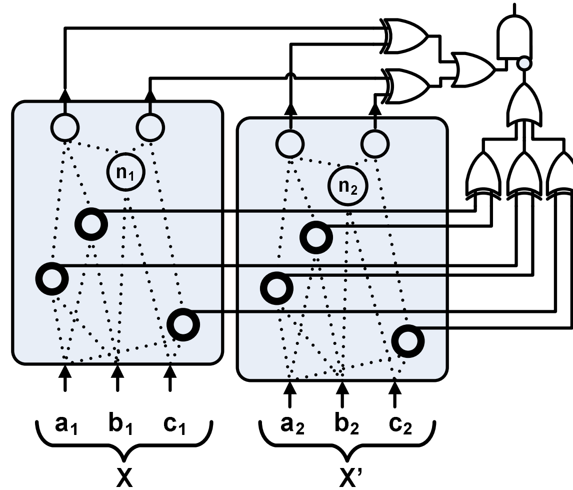

Figure 3.2: A miter to compute a minimum SPFD at node n. Bold nodes constitute the separator set.

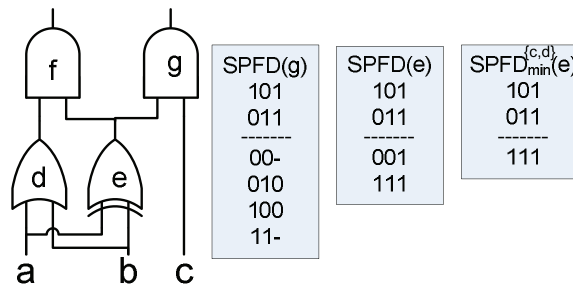

Figure 3.3: An example showing the shortcomings of the miter of Figure 3.2. The separator set is $\{c, d\}$.

an auxiliary circuit that regulates SPFD propagation. As the twin circuits are the same as those used in Figure 3.2, we will only describe components of the auxiliary part in the next paragraphs.

As seen in Algorithm 1 that a pair of $SPFD(n)$ can be distributed to more than one fanin of n. However, by reducing the number of pairs of $SPFD(w_r)$, the number of wires that can replace w_r increases. Therefore, in the distributing miter, we will make sure that a pair will be distributed to only one fanin. We first impose an order on the fanins of every node T $\in TFO(n)$. The fanins of node T (e.g., T = node i in Fig. 3) either lie outside TFO(n) (e.g. the fanin coming from f), or inside TFO(n) (e.g., inputs coming from g and h). The following rules are used to minimize the number of pairs in $SPFD(n)$: (1) fanins of T that fall within $TFO(n)$ have priority over fanins that are not in the set, (2) among fanins of the same type, use an arbitrary order.

The distributing order of a node can be implemented using a regular priority encoder circuit, shown in Figure 3.5. The priority encoder for a node with k fanins has k inputs and k outputs, designated by I and O, respectively. Its operation is described as $O(0) = I(0), O(i) = 1$ iff $(I(i) = 1) \wedge \{\wedge_{\forall j<i}(I(j) = 0)\}$. Distinguish-ability of fanin node i is obtained by XOR($i1, i2$), where $i1$ and $i2$ are the corresponding nodes of i in the twin circuits.

As mentioned before, we are interested in finding out whether there is a distinguishing path from n to a PO. To do so, each node in $TFO(n)$ will be accompanied by an auxiliary circuit with one output called *indicator node*. If the indicator gate of node i (called $\alpha(i)$) evaluates to one, there is a distinguishable path from n to i. The auxiliary circuit is built using the following rules.

1. For node n, $\alpha(n)$ is XOR($n1, n2$).

2. For a PO $\mathcal{P}$ and its driver $\mathcal{D}$, the auxiliary circuit has only one gate $\alpha(\mathcal{P})$

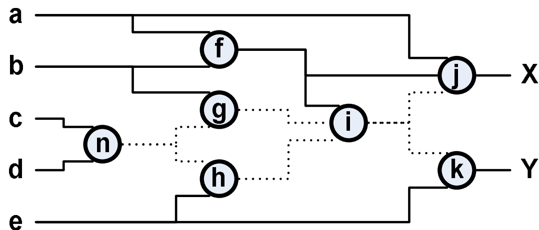

Figure 3.4: A circuit showing $SPFD(n)$ propagation to POs through dotted edges.

which is AND(XOR($\mathcal{D}1, \mathcal{D}2$) , $\alpha(\mathcal{D})$). (e.g., $\alpha(X)$ and $\alpha(Y)$ in Figure 3.6)

3. For node $\mathcal{N} \in TFO(n) \backslash \{PO \cup n\}$, let $\mathcal{Q} = \{\mathcal{R} | \mathcal{R} \in FI(\mathcal{N}) \wedge \mathcal{R} \in TFO(n)\}$. The auxiliary circuit is an AND between OR($\forall \alpha(\mathcal{C})$), $C \in \mathcal{Q}$ and OR($\forall O(i)$), where $O(i)$ is the output of $I(i)$, where the i input connects to a node in $\mathcal{Q}$.

Adding an OR to $\alpha(\mathcal{P}), \forall \mathcal{P} \in PO$ will complete the miter. Using this construction, we can guarantee that there exists a path from n to a PO for any pair of minterms $(m0, m1) \in SPFD(n)$. Such paths consist of nodes $\mathcal{N}$ whose $\alpha(\mathcal{N}) = 1$. The existence of the path is confirmed when the final OR output is '1'. An AND gate of the final OR and n can be inserted to avoid symmetry of the twin circuit and reduce the SAT solution space by half.

The distributing miter to compute $SPFD(n)$ of the circuit in Figure 3.4 is shown in Figure 3.6. The number of additional gates is $c|V(G)|$, where c is a constant. A miter to compute SPFD of an edge can be built by first adding a dummy node on that edge and then building a distributing miter for the dummy node.

3.5.3 A Limitation of Distributing Miters

Although a distributing miter can capture the flow of SPFD accurately, it has an inherit limitation. In Algorithm 1, $SPFD(n)$ was classified into $F0$ and $F1$.

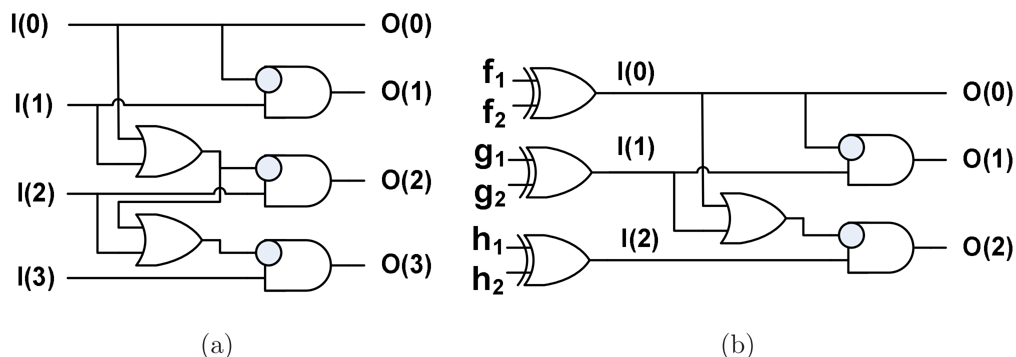

Figure 3.5: a)A priority circuit at a node with four fanins. b)The priority encoder circuit for node i of Figure 3.4.

Although some minterms in $F0$ may not originally be required to be distinguished from minterms of $F1$, the algorithm forces them to be distinguished to make sure the graph is bipartite (Step 5). Imposing such a condition is doable because all pairs in the SPFD are known at once. In contrast, a distributing miter cannot generate such extra pairs because it can only see one pair at a time. Therefore, the SPFD computed using a distributing miter is a subset of that computed using Algorithm 1.

3.5.4 A Fast SPFD Rewiring Algorithm

Although SPFD are used to describe the functionality of a circuit in both a conventional SPFD-based rewiring and our new algorithm, they use such information differently. Assuming that the wire to be removed and the new wire (or no new wire) is given. Our fast SPFD rewiring algorithm can be summarized in Figure 3.7 and will be described in this section.

The distributing miter introduced in the previous subsection can be used to enumerate all pairs of SPFD in the same manner that the miter in Subsection 3.5.1 is used to compute a minimum SPFD. However, the number of such pairs is

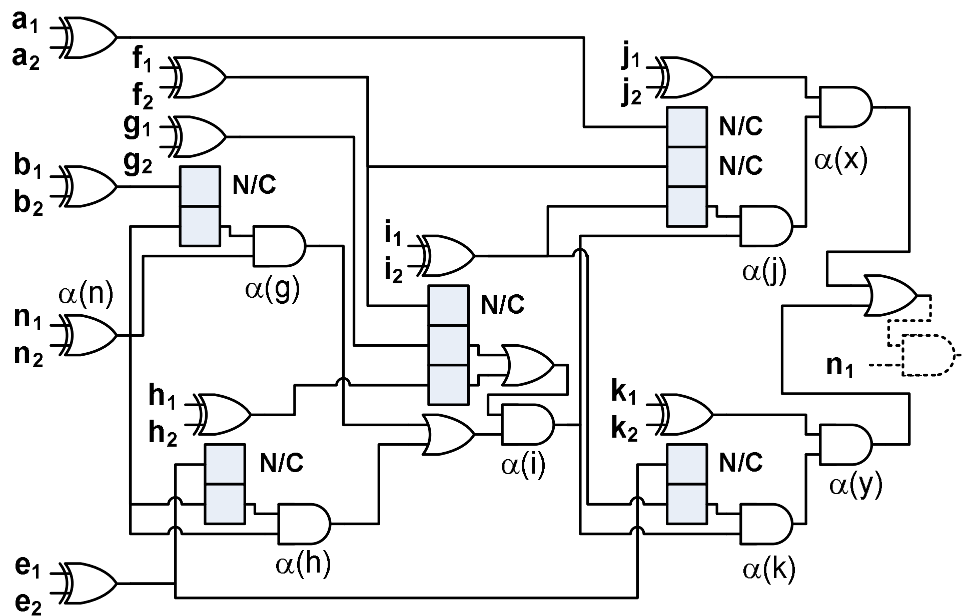

Figure 3.6: The distributing miter to compute SPFD(n) of Figure 3.4. Grey squares are priority encoders. Boxes labeled N/C correspond to inputs $\notin TFO(n)$, hence not connected (*e.g.* input f of gate i).

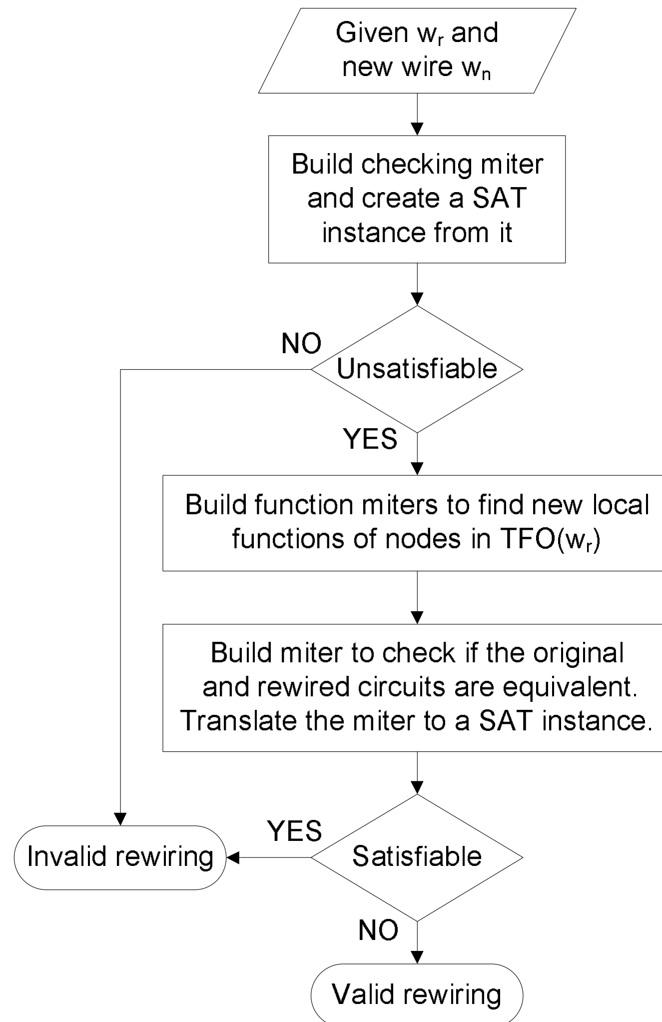

Figure 3.7: A flowchart of our rewiring algorithm.

extraordinarily large. Our rewiring technique that does not require the list of SPFD will be explained in this section.

Our rewiring algorithm consists of 3 steps. The proposed rewiring will be checked first by solving a SAT instance of a distributing miter once. Then, the new node functions can be computed by solving other auxiliary SAT instances. Because a distributing miter may not capture bipartite enforcing edges, our technique requires an equivalence checking between the original and the rewired circuits as the last step. As an illustrating example, in the following subsections, let us assume that we want to replace (a, b) with (c, d), where $d \in DOM(b)$.

Screening the Proposed Rewiring

A new wire, w_n, can be used to replaced w_r, if $SPFD(w_r) \subset SPFD(w_n)$. This requirement can be verified by finding both $SPFD(w_r)$ and $SPFD(w_n)$ using distributing miters. But, the number of pairs in each SPFD could be quite large, we would like to determine if a rewiring proposal fails without enumerating all pairs. A rewiring proposal cannot be accepted if the SPFD of the removed wire after rewiring is not empty. The distributing miter discussed in the previous section cannot be used directly for this purpose. A *checking miter*, discussed in the next paragraph, can be used to address this shortcoming.

With minor modifications to a distributing miter, an invalid rewiring proposal can be identified using a *checking miter*. We use the structure of $TFO(n)$ to guide the building of the auxiliary circuit for computing $SPFD(n)$ described in Subsection 3.5.2. We can extend the distributing miter for computing $SPFD(a, b)$ to build the checking miter by adding (c, d) as the first input of the priority encoder at d. Note that (c, d) does not appear in the twin circuits of the miter. Since (c, d) has the highest priority, it absorbs all SPFD pairs that it can distinguish. For

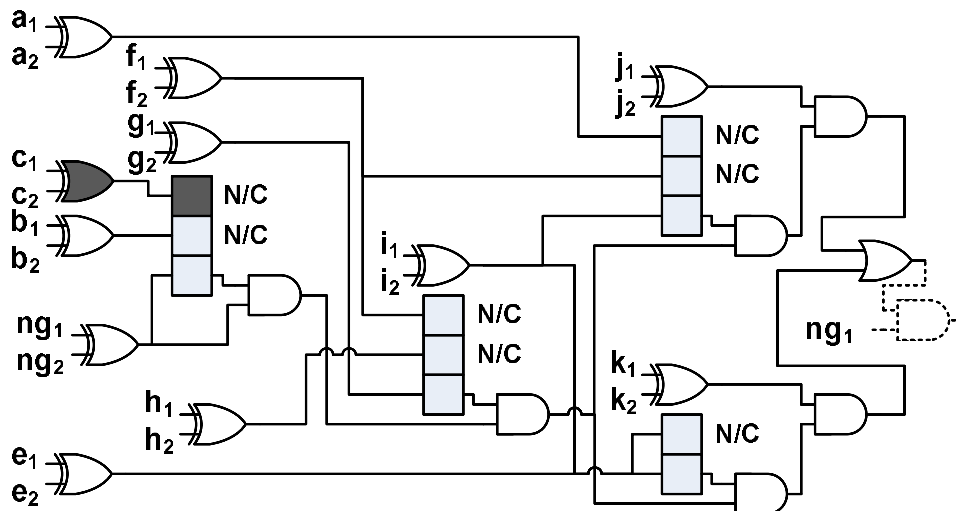

Figure 3.8: A checking miter for validating the replacement of (n, g) of Figure 3.4 with (c, g).

the rewiring proposal to be valid, $SPFD(a, b)$ obtained from the checking miter must be empty, which can be tested by solving the corresponding SAT instance once. The checking miter for validating the replacement of (n, g) with (c, g) of the circuit in Figure 3.4 is shown in Figure 3.8.

Finding New Node Functions

To complete the rewiring process, we have to compute new functions at some nodes in $TFO(Sk(wr))$ to reflect the change in SPFD propagation due to rewiring. At any node, its care local minterms are those necessary to propagate the SPFD. They can be identified by using a distributing miter to enumerate all pairs in the SPFD that the node has to propagate. However, as the number of pairs could be large, such a method could incur a large runtime penalty.

Considering the fact that several SPFD pairs are likely to be projected into the same local minterm, an efficient way to compute local functions is summarized in

Algorithm 2, which uses a *function miter* to determine the new function of nodes. Because nodes in $TFO(Sk(wr))$ are affected by the change in SPFD propagation, their new functions will be computed in topological order. To prevent a SAT solver from re-discovering the same solution, boundary nodes of $TFO(Sk(wr))$ will be used to establish blocking clauses. However, if the number of such nodes is larger than the number of PIs, the PIs will be used instead. To maintain consistency of SPFD propagation, the function miter must use the same distributing order as in the checking miter. The function miter for node i is shown in Figure 3.9.

Discovering care minterms at a node is performed incrementally. Let U and V be the sets of ON and OFF minterms at node n, respectively ($e.g., U = \{x, y\}, V = \{w, z\}$). To discover new OFF minterms, we force SAT variables corresponding to fanins of n in one copy to be one minterm from U and add a blocking clause using variables in another copy for each minterm in V ($e.g.$, adding a forcing clause for x and blocking clauses for w and z in one SAT instance, and adding a forcing clause for y and blocking clauses for w and z in another). The new ON minterms are found similarly. Solving this SAT instance will discover P and Q as the new ON and OFF minterm subsets, respectively. At this point, the ON and OFF sets are $U \cup P$ and $V \cup Q$, ($e.g.$, the ON and OFF sets becomes $\{x, y, u\}$ and $\{w, z, v\}$), respectively. Now, to discover more OFF minterms, each minterm in P will be used as a forcing clause, but the blocking clause will be added for each minterm from both V and Q ($e.g.$, adding a forcing clause for u and blocking clauses for w, z and v. This process stops when we cannot expand either the ON or OFF sets.

After obtaining ON/OFF minterms of each node, its local function can be computed by $\sum_{r=1}^{l_1} \prod_{s=1}^{l_0} P_{rs}$, where l_0/l_1 are the number of minterms in OFF/ON sets, respectively. Let k be the first fanin in the order that distinguishes the pair (m_r, m_s). $P_{rs} = y_k$ if f_k includes m_r and $P_{rs} = \overline{y_k}$ if $\overline{f_k}$ includes m_r [54].

Algorithm 2 Finding new node functions.

1: Mark Sk(wr) as *'changed'*.
2: **for** each $n \in TFO(Sk(wr))$ marked *'changed'*, in topological order **do**
3: Build a function miter for n, using the same propagation order as the checking miter.
4: Find boundary nodes, nodes with $FO \in TFO(Sk(wr))$ or PIs, whichever is smaller.
5: Create the SAT instance from the miter.
6: Solve the SAT instance.
7: Trace the distinguishable path and collect ON/OFF minterms at each node as follows. Collect values from $FI(n)$ from the first copy and store in ON or OFF set depending on the value of n in that copy. Do the same for the second copy.
8: **repeat**
9: **for** each ON minterm found in the last iteration, m **do**
10: Add a forcing clause for m.
11: Add a blocking clause for each OFF minterm.
12: **repeat**
13: Solve SAT and collect new OFF minterm.
14: Add blocking clause for boundary nodes.
15: **until** UNSATISFIABLE
16: **end for**
17: Repeat the above loop for OFF minterms.
18: **until** UNSATISFIABLE
19: Compute new local function from the collected minterms.
20: If the new function changes, mark $FO(n)$ as *'changed'*.
21: **end for**

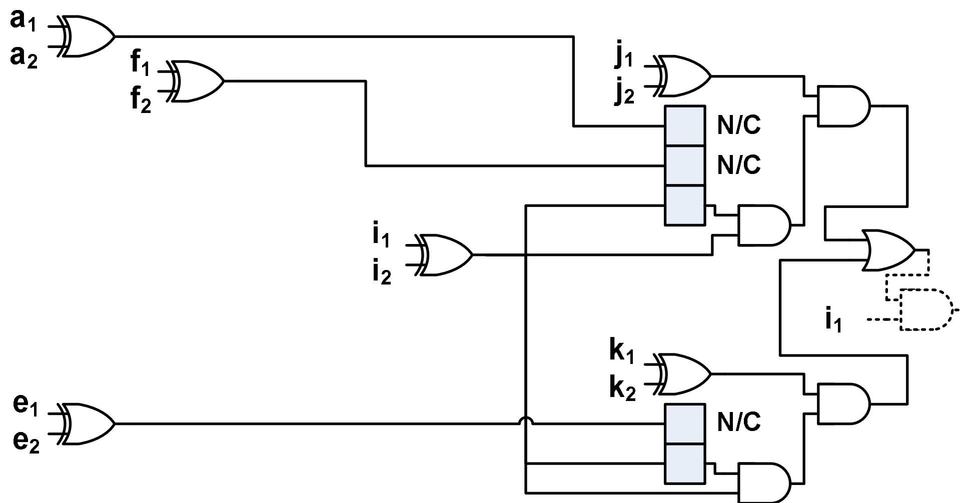

Figure 3.9: A function miter used for computing the new $f(i)$ after replacing (n, g) with (c, g) of the circuit in Figure 3.4.

Equivalence Check

Because extra SPFD is unobservable by function miters, new local functions computed in the last subsection may not allow propagation of extra SPFD. Therefore, the equivalence between the original and the rewired circuits must be checked. The miter for checking the equivalence of the circuits before and after rewiring contains XORs between each PO pair from the twin circuits and an OR of all these XOR outputs. The two circuits are equivalent if the SAT instance of the miter is unsatisfiable.

3.5.5 Quality of the Proposed Technique

In the rewiring context, we use the term *rewirability* to refer to the number of wires that can be rewired in the whole circuit. Both the BDD-based approach and the proposed technique use the SPFD theory. But, the miters have a limitation as mentioned above. Thus, it might lead one to think that rewirability of BDD-based

rewiring approaches must be the upper bound for the SAT-based ones. However, the BDD-based implementation of the SPFD theory also has its own limitations. In-depth investigation revealed that in some cases, the BDD-based rewiring fails, while ours succeeds. For example, consider the circuit in Figure 3.10. We would like to replace (b, q) with (p, q). A BDD-based approach would use Algorithm 1 to produce $SPFD(b, q)$ observed at q, in which a is *don't care* (Line 2). As a result, it would conclude that $SPFD(b, q)$ is not distinguishable by p ('11' cannot distinguish between '-1' and '-0'). This is not the case, because a is always 1 in $SPFD(b, q)$ observed at t, and hence p can distinguish $SPFD(b, q)$. Our SAT-based method can indeed conclude that (b, q) can be replaced with (p, q). Using the checking miter, each pair of $SPFD(b, q)$ observed at t will result in $\alpha(p), \alpha(q)$ and $\alpha(t)$ evaluating to '1'. As (p, q) is the highest priority at q, (p, q) absorbs all such pairs and we can conclude that the rewiring is valid.

3.6 SPFD-based rewiring beyond dominator nodes

SPFD-based rewiring involves two steps: (1) implicitly or explicitly generating a requirement for a new wire, and (2) checking each candidate source for a new wire against the requirement. A new way to generate such a requirement for any nodes is described in Section 3.6.1. The proposed algorithm and its correctness will be proved in Section 3.6.2. Throughout this section, we assume that (1) SPFDs have been annotated onto each node using Alg. 1 and (2) each SPFD has only one ISF.[2]

[2]If the incoming $SPFD(n)$ has multiple components, the second assumption is satisfied by applying a union operation. The second assumption is required to ensure that a change in $f(m), m \in TFI(n)$, will not make the local $SPFD(n)$ non-bipartite.

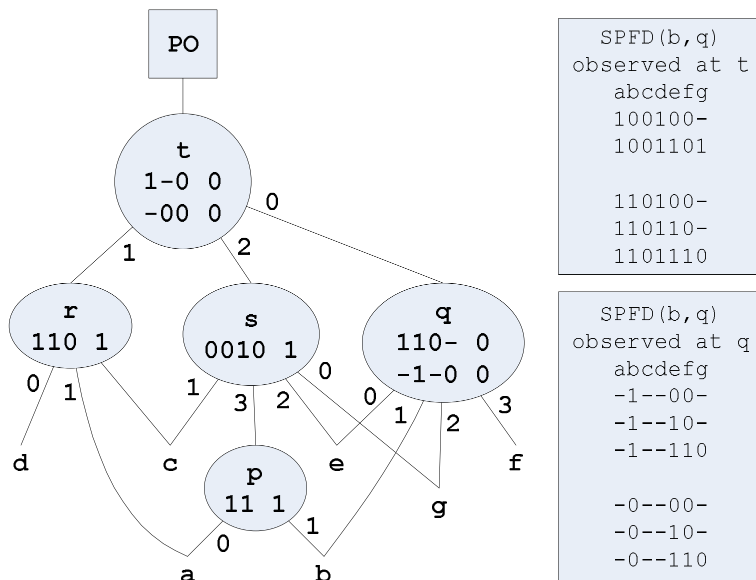

Figure 3.10: A circuit showing deficiency of BDD-based SPFD rewiring. The fanin order at each node is indicated below the node.

3.6.1 Generating SPFD Required for a New Wire

If w_r is removed, the global functions of nodes in $TFO(w_r)$ may change and they may not distinguish all SPFD originally assigned to them. The following definition classifies the SPFD at a node reflecting the effect of removing w_r. The SPFD at an edge can be defined similarly.

Definition 3.6.1 *1. $SPFD^{valid}(n)$ is defined as the part of $SPFD(n)$ that can still be distinguished by n after removing w_r.*

2. $SPFD^{invalid}(n)$ is the edge-induced subgraph of $SPFD(n)$ by $E\{SPFD(n)\} - E\{SPFD^{valid}(n)\}$.

Lemma 3.6.2 shows how $SPFD(n)$ can be decomposed into $SPFD^{valid}(n)$ and $SPFD^{invalid}(n)$. Intuitively, $SPFD^{invalid}$ can be computed from w_r toward POs because $SPFD^{invalid}(w_r) = SPFD(w_r)$. At a particular node n, a

Lemma 3.6.2 *$SPFD^{valid}$ and $SPFD^{invalid}$ at any edge and node in the circuit can be defined recursively from w_r toward POs as follows.*

1. *$SPFD^{invalid}(w_r) = SPFD(w_r)$ and $SPFD^{valid}(w_r) = \emptyset$. For any other fanin wire w of $sk(w_r)$, $SPFD^{invalid}(w) = \emptyset$ and $SPFD^{valid}(w) = SPFD(w)$.*

2. *Let $SPFD^{valid}_{in}(n) = \cup_{wi \in FI(n)} SPFD^{valid}(wi)$ be $SPFD^{valid}$ that need to be guaranteed distinguishable at n. Let X be a maximal complete bipartite such that $X \supseteq SPFD^{valid}_{in}(n)$ (X distinguishes $SPFD^{valid}_{in}(n)$). Thus, $SPFD^{valid}(n) = X$ and $SPFD^{invalid}(n) = SPFD(n) - X$.*

3. *For each fanout wire, wo, of n, $SPFD^{invalid}(wo) = SPFD(wo) \cap SPFD^{invalid}(n)$ and $SPFD^{valid}(wo) = SPFD(wo) \cap SPFD^{valid}(n)$*

part of $SPFD^{invalid}(w), w \in FI(n)$ may not come from $PO(n)$, but may have been introduced to prevent n from becoming non-bipartite. Therefore, to find $SPFD^{invalid}(n)$, we need to find a function, whose corresponding SPFD distinguishes most of $SPFD(n)$. The $SPFD^{invalid}(w), w \in FO(n)$ is the part of $SPFD(w)$ that falls into $SPFD^{invalid}(n)$.

Once $SPFD^{invalid}$ is computed for each node between $sk(w_r)$ and its dominator node, appropriate wires can be added to nodes along any cut between the two nodes to rectify the effect of removing w_r. Furthermore, $SPFD^{invalid}$ can be transferred to another node. For example, consider Figure 3.11, where $SPFD^{invalid}$ and $SPFD^{valid}$ are shown in grey and black arrows, respectively. Two appropriate wires can be added to node s and u. Also, $SPFD^{invalid}(s)$ can be combined with $SPFD^{invalid}(u)$ to generate a requirement such that w_r can be replaced by one wire connecting to u. Although manipulating $SPFD^{invalid}$ for an

arbitrary number of additional wires is beneficial, it is discouraged because (1) it is computationally expensive, (2) the successful rate of using more than one wires is low [55] and (3) it requires more resources, negatively affecting an optimization objective at hand. As a result, such manipulations will not be discussed further.

Lemma 3.6.2 is inefficient because X is difficult to find and it has to be computed for all nodes along the path from $DOM(sk(w_r))$ to $sk(w_r)$. A more efficient way to compute X is to remove w_r from the local function of $sk(w)$ and to modify the function accordingly. As a result, the new global function at each node can be used as X at the node. However, the maximality of X is not guaranteed and unlikely to be met because functions of nodes other than $sk(w_r)$ are left unexploited. Consequently, the size of $SPFD^{invalid}$ may be larger than that obtained using a maximal X and an algorithm may not find a successful rewiring even though it would have been possible if a maximal X was used.

As only one new wire will be considered, instead of computing $SPFD^{invalid}$ at every node and then combining them to allow rewiring by one new wire, $SPFD^{invalid}(n)$ is directly computed under the convention that n will be the only destination for the new wire using Alg. 3. Note that Lines 1-6 of Alg. 3 are the same as Lines 2-7 of Alg. 1 except the computation of G which is the set of minterms missing from the original SPFD as a result of remvoing w_r. The $SPFD^{invalid}(n)$ obtained is denoted as $SPFD^{invalid*}(n)$ to signify the convention used. As a result, if a wire that distinguishes $SPFD^{invalid*}(n)$ is added to n, the effect of removing w_r can be rectified.[3] Let us denote $SPFD$ distributed by Alg. 3 as $SPFD'(n)$ to signify that the $SPFD$ is computed on a modified circuit after removing w_r. Note that $SPFD'(n)$ is $SPFD^{valid}(n)$.

[3]This statement will be proved in the next section.

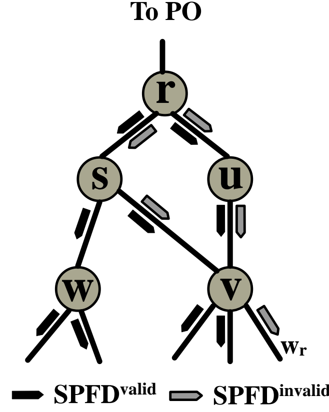

Figure 3.11: Decompositions of SPFD at edges into $SPFD^{valid}$ and $SPFD^{invalid}$.

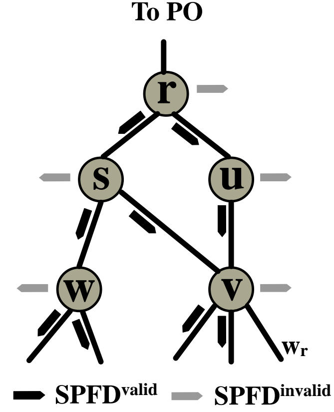

Figure 3.12: Existence of $SPFD^{invalid}$ at any node $n \in TFI\{DOM(sk(w_r))\}$.

If $n \in DOM(w_r)$, n has to distinguish $SPFD(n)$ to mask the effect of removing w_r. Thus, $SPFD(n)$ is used in Alg. 3 to compute $SPFD^{invalid*}(n)$ as the part that cannot be distributed to $FI(n)$. However, if $n \notin DOM(w_r)$, it has to distinguish $SPFD'(n) \cup SPFD^{invalid*}(m)$, where $m \in FO(n)$. If two nodes, u and v, share the same fanout node m as their only fanout, both nodes will use $SPFD^{invalid*}(m)$. Thus, if an appropriate wire is added to either u or v, $SPFD^{invalid*}(m)$ will be satisfied, making rewiring with one wire possible (see the next section for details).

Our approach outperforms the existing SPFD-based rewiring in two ways:

1. $SPFD^{invalid*}(n)$ exists even if $n \notin DOM(w_r)$. For example, $SPFD^{invalid*}$ exists at s and u, in Fig. 3.12.

2. $SPFD^{invalid*}(n)$ exists even if $n \notin TFO(w_r)$ as long as $n \in TFI(DOM(w_r))$. For example, $SPFD^{invalid*}(w)$ exists in Fig. 3.12.

Algorithm 3 $SPFD^{invalid}$ computation at a gate after w_r is removed.

Require: $SPFD = (F_1, F_0)$, computed by Alg. 1 before removing w_r.

1: Construct all possible minterms on the inputs of the gate,*i.e.*,
$b_0 = 0 \cdots 00 = \overline{y}_1(x) \cdots \overline{y}_2(x)\overline{y}_3(x), ..., b_{2^n-1} = 1 \cdots 11$.

2: Compute restricted minterms $a_i = b_i(F_1 + F_0)$. The set (F_1, F_0) is the care set. Thus, a_i describes all minterms needed to be distinguished.

3: Distribute all care minterms into two sets:

1. $F_1' = \{a_i \mid a_i \subseteq F_1', a_i \text{ is not constantly } 0\ \}$
2. $F_0' = \{a_i \mid a_i \subseteq F_0', a_i \text{ is not constantly } 0\ \}$
3. $G = \{(F_1 \cup F_0) \backslash (F_1' \cup F_0')\}$

4: Build complete bipartite graph $F' = F_1' \times F_0'$.

5: **for** each $(a_i, a_j) \in F$ **do**

6: Add (a_i, a_j) to at least one input k such that $a_i \leq f_k \leq \overline{a}_j$ or $a_j \leq f_k \leq \overline{a}_i$.

7: **end for**

8: $SPFD^{invalid} = (G \cap F_1, G \cap F_0)$

3.6.2 The Proposed Algorithm and its Correctness

Our proposed rewiring algorithm can be shown in Alg. 4. The algorithm assumes that each node has been assigned a priority to be used as a sink of a new wire by an application. First, the SPFD of each node $n \in DOM(sk(w_r))$ must be computed in Line 1 so that the correct functionality of the circuit can be captured. If $SPFD(w_r) = \emptyset$, w_r can be removed without adding any wires. After that we will try to rewire the circuit using only one node as the destination for a new wire, as mentioned in the previous subsection. The search for the new wire to be added is done following a depth-first-search (DFS) order. To help guide the DFS traversal, we compute $v(n)$ as the traversal priority, which is calculated as the smallest priority among nodes in $TFI(n)$. Rewiring will be tried by modifying $f(sk(w_r)$ to $f_{\bar{w}_r}, f_{w_r}, f_{\bar{w}_r} + f_{w_r}$ and $f_{\bar{w}_r} \cdot f_{w_r}$, one at a time, where $f_{\bar{w}_r}$ and f_{w_r} are

Algorithm 4 Rewiring w_r.

Require: $p(n)$, a priority ordering for using node n as the sink of the new wire. Smaller p means higher priority.

1: Compute SPFD using Alg. 1.
2: **if** $SPFD(w_r) = \emptyset$ **then**
3: remove w_r and exit.
4: **end if**
5: Find $v(n) = min\{p(n), min_{m \in FI(n)}\{v(m)\}\}$, in reverse topological order, ignoring w_r.
6: **for** each type of function modifications on $sk(w_r)$ **do**
7: Make a copy of the circuit and work on the copy.
8: Remove w_r and modify the local function of $sk(w_r)$.
9: **for** each $n \in DOM(sk(w_r))$, with $v(n) < \infty$, in increasing order of v. **do**
10: Annotate $SPFD^{valid}$ by applying Alg 3 to $TFI(n) \cap TFO(sk(w_r))$, in reverse topological order, obtaining $SPFD^{invalid*}(n)$.
11: **if** DFS checking n (Alg. 5) is successful **then**
12: Use the modified circuit as the output and Return.
13: **end if**
14: **end for**
15: **end for**

Algorithm 5 DFSchecking .

Require: a node n and $SPFD^{invalid*}$ from one of its fanouts.

1: **if** $p(n) < \infty$ **then**

2: If $\exists p$, such that $SPFD'(p)$ can distinguish $SPFD^{invalid*}(n)$, add wire (p, n). Update f(n) and return success.

3: **end if**

4: Find $SPFD^{invalid*}(n)$ by Alg 3.

5: **for** each $m \in FI(n)$, in increasing $v(m)$. **do**

6: **if** DFSchecking($m, SPFD^{invalid*}(n)$) is successful. **then**

7: Update $g(m)$ and $f(n)$ to satisfy $SPFD'(m) \cup SPFD^{invalid*}(n)$ and return success.

8: **else**

9: Restore $f(m)$ and $SPFD(m), m \in FI(n)$.

{They were altered by Line 4}

10: **end if**

11: **end for**

12: return fail

$f(sk(w_r))$ with the variable corresponding to w_r set to 0 and 1, respectively. The algorithm considers rewiring in a fanin cone of a dominator node in the increasing order of $v(n)$, breaking the tie by the shorter distance to $sk(w_r)$. $SPFD^{valid}$ is computed in Line 10. A DFS traversal starts from a dominator node by calling DFSchecking (Alg. 5).

DFSchecking takes a node, n, and $SPFD^{invalid*}(n)$ as inputs. First, if n is designated as a possible sink, we will try to find a compatible source m such that $g(m)$ distinguishes $SPFD^{invalid*}(n)$ (see Definition 3.3.3) and return after $f(n)$ is updated. Otherwise, $FI(n)$ will be considered. Before calling DFSchecking on a fanin node, m, $SPFD^{invalid*}(m)$ must be computed. Note that if DFSchecking call at m fails, the original $f(m)$ and $SPFD(m), m \in FI(n)$ must be restored because they were altered by the call to Alg. 3. However, if the call is successful,

the altered values are correct as $SPFD^{invalid*}$ passes through node m.

Let us prove that Alg. 4 can indeed rewire the circuit correctly. The rewiring is considered correct if (1) the functionalities at POs are maintained and (2) there exists a one-output binary function at each node. First, let us invoke a theorem and state a corollary that will be heavily used in our proof.

Theorem 3.6.3 *(Theorem 1 in [54]) Given* $SPFD(n)$*, if* $\forall k \in FI(n)$*,* f_k *satisfies* $SPFD(k, n)$ *(as defined in Section 3.3), which is calculated using Alg. 1, there exists a one-output binary function* f *that satisfies* $SPFD(n)$*.*

Corollary 3.6.4 *Let* $SPFD(n) = \{f^{on}, f^{off}\}$*. If each pair of minterms* $\{(a_i, a_j)|a_i \in f^{on}, a_j \in f^{off}\}$ *can be distinguished by a fanin of* n*, there exists a one-output binary function* $f(n)$ *that satisfies* $SPFD(n)$*.*

The correctness of our algorithm can be shown in the following theorem.

Theorem 3.6.5 *Alg. 4 correctly rewires a circuit.*

Proof 3.6.6 *If* $SPFD(w_r) = \emptyset$*,* w_r *can be removed without adding a wire [45]. Thus, we need to prove that adding a wire proposed by Alg. 4 retains circuit's functionality.*

Line 10 of Alg. 4 annotates $SPFD^{valid}$ *at each node affected by removing* w_r*. Note that by definition,* $SPFD^{valid}(n)$ *is the part of the original* $SPFD(n)$ *that can still be distinguished after* w_r *is removed.*

Assume that the new wire will be added to a node n *in the fanin cone of a dominator* d*.* $SPFD^{invalid*}(d)$ *is passed on to* n *through a path containing nodes* $d, n_1, ..., n_k, n$*. When* $SPFD^{invalid*}(d)$ *is passed to* n_1*, Alg. 3 is used to compute* $SPFD^{invalid*}(n_1)$*. Thus,* $SPFD^{valid}(n_1) \cup SPFD^{invalid*}(n_1)$ *is a complete bipartite. The same can be said with other nodes along the path, including* n*.*

A wire, w, is added to n only if $sr(w)$ distinguishes $SPFD^{invalid}(n)$. Thus, by Lemma 3.6.4, there exists $f(n)$. At n_k, both $SPFD^{valid}(n_k)$ and $SPFD^{invalid}(n_k)$ can be distinguished ($g(n)$ distinguishes $SPFD^{invalid}(n_k)$). Invoking the same lemma, we can conclude that $f(n_k)$ exists. Applying the same argument to each node along the path until d, $SPFD^{valid}(d) \cup SPFD^{invalid}(d)$, which is the original $SPFD(n)$, can be distinguished. As a result, the rewired circuit has the same functionality as that of the original circuit.

3.7 Experimental Results

We presented two improvements to SPFD-based rewiring in two distinct directions in this chapter. In the first part (Section 3.5), a novel approach to reduce rewiring runtime was proposed and a technique to improve the effectiveness of an SPFD-based rewiring was proposed in the second part (Section 3.6). Therefore, in this section, the experimental results will be presented accordingly.

3.7.1 Experimental Results of the Proposed Efficiency Improvement

A conventional SPFD-based rewiring algorithm that uses BDDs, and our SAT-based algorithm were implemented in C and all experiments were run on a 3GHz-Intel Xeon machine with 4GB RAM running Linux. In the first experiment, each wire w_r in the circuit was selected to be removed and be replaced by a candidate source wire from the set of nodes $n \notin TFO(sk(w_r))$. Regardless of the rewiring result, the circuit was left intact and the next wire was selected. The depth of the circuit was allowed to increase in this experiment. Its results can be summarized in Table 3.1. The benchmark circuits are listed in the first column, with their numbers of nodes and wires shown in the second and third columns, respectively. Rewirability results and runtimes of a BDD-based SPFD rewiring are shown in

the forth and fifth columns, respectively. Results of our SAT-based method are shown in the sixth and seventh columns. The conventional BDD-based method implemented in the experiments should be considered comparable to SPFD-GR ([46]), but the distributing order used may be different.

We can observe that SAT-based rewiring is faster, especially for large circuits, while yielding comparable rewiring results. For large circuits, the BDD-based method cannot finish within 20 hours, while our method finished in reasonable amount of time. Excluding these large cases, we conclude that SAT-based SPFD rewiring is faster, while yielding comparable rewiring results.

Table 3.2 shows the runtime break-down of the two methods, giving more insight into the runtime behavior of Table 3.1. The second column shows the percentage of runtime the BDD-based implementation spent for SPFD computation. The rest of the columns correspond to our SAT-based method. The third and fourth columns show the runtimes for computing local functions and performing equivalence checking when the proposed rewiring passes the screening test (using checking miters). The fifth and the sixth columns show the number of rewiring instances considered and the number of instances that passed screening. The last column shows the average time spent on each instance. We can see that in the BDD method, SPFD computation took a significant portion of runtime, especially for *C499,C1908,apex6*. The runtime of our technique depends on the number of instances considered and the number of instances that pass the screening. On average, it spends about 2.5ms per instance.

Rewiring can be applied after logic synthesis for area and/or delay improvements [46]. At that stage, the circuit depth is used as the measure for circuit delay. Thus, we conducted another experiment similar to the first one, but in which the circuit depth is not allowed to increase. In effect, the circuit depth is

a constraint used to eliminate some rewiring proposals. The results are shown in Table 3.3. The meaning of each column is similar to that of the previous tables. As expected, the rewirability measure decreases, but the results of the two engines are comparable. While the runtime of the BDD-based implementation remains virtually the same, the runtime of the SAT-based one reduces when the number of instances decreases.

Other applications may have more constraints. These constraints can be used to screen out most of the rewiring proposals. As a result, fewer candidates are needed to be tested by our rewiring engine. To understand how our algorithm would behave in such applications, we performed another experiment to see how its runtime scales with the number of candidate wires. In this experiment, for each wire to be removed, all candidate sources for the wire were collected as in the first experiment. But, only a number of candidates were randomly picked. The same set of selected candidates was used in both rewiring engines in the same order.

Table 3.4 shows that our technique scales well with the number of candidates. We can see that as the number of candidates decreases, the rewirability of both approaches decline but are still comparable. The result confirms our observation that SPFD computation is the bottleneck in the BDD-based SPFD rewiring as its runtime decreases very slowly, while its percentage slightly increases with decreasing number of candidates. In contrast, the runtime of the SAT-based rewiring reduces almost linearly with the number of candidates as can be seen from the total runtime and the total number of instances. In other words, the average time per instance is virtually constant.

3.7.2 Experimental Results of the Proposed Efficacy Improvement

Alg. 4 was implemented using BDD for SPFD computation and experiments were conducted to investigate the benefits of not restricting the set of destinations for a new wire to only dominator nodes using similar setting as that in Section 4.6.

All experiments were run on a 3GHz-Intel Xeon machine with 4GB RAM running Linux. The results are summarized in Table 3.5 and 3.6. The benchmark circuits listed in the first column are the same as those used in Section 4.6. Experiments were conducted for both the cases of unlimited and limited circuit depths (in the same manner as those in Table 3.1 and 3.3) and their result are shown in Table 3.5 and 3.6, respectively. All rewired circuits are confirmed by checking their PO functions in the experiments. Rewirability is reported in the second to fifth columns of both tables. The results in the second columns was obtained by restricting the destinations of a new wire to dominator nodes. The results of not restricting the location of destination nodes are listed in the third columns labeled "Any node". In some cases, not restricting the destinations does not show a significant improvement, such as *cordic* and *apex*7. In the "Any node" column, dominator nodes were given priority over non-dominator ones. Thus, there might have been a non-dominator node which could have been used in rewiring, but because of the preference of dominator nodes was never explored. To emphasize this fact, another run ignoring dominator nodes was performed and its results are listed in the fourth columns labeled "No dom". The percentage improvements of "Any node" over "Only dom" (the third column over the second one) are shown in the fifth column. The total runtime of our algorithm over a particular circuit is shown in the last column of each table. Although the numbers are different due to different ordering of SPFD distributions, the results in Table 3.6 should be considered equivalent to SPFD-GR (Table 1 of [46]).

We can see that the proposed rewiring algorithm improves the number of wires that can be rewired for almost all circuits. The average improvement is about 13% for both depth-limited and unlimited cases, while that of circuits with more than 100 LUTs is 20% and 17% for unlimited and limited circuit depths, respectively. This is because for a small circuit, there are fewer number of non-dominator nodes, resulting in lower improvements. Although, the cut-off point is arbitrary, it emphasizes that the improvements are larger for larger circuits.

It was observed that our algorithm not only increased the number of wires that can be rewired, but it also increased the number of alternative wires for replacing a wire. This observation can be conceptualized by histograms of the number of alternative wires in Figs. 3.13 for $term1$. Note that the number of wires that have 0 alternative wires between the two approaches are the same as the wire itself can be removed without adding any wires. Rewiring with only dominator nodes has a higher number of wires with one alternative wire. Although hard to observe, rewiring with any nodes has a histogram spread out toward the higher numbers of alternative wires. Let us define $r(n)$ as the number of wires which have at least n alternative wires. The plots of $r(n)$ computed from Fig. 3.13 can be shown in Fig. 3.14. We can see that $r(n)$ of rewiring using any nodes as sink nodes dominates that of rewiring with only dominator nodes for all n. Note that $r(0)$ are 126 and 139 which are the total numbers of wires that can be rewired when using only dominator nodes and any nodes for $term1$ shown in Table 3.6, respectively.

3.8 Summary

This chapter proposed improvements to SPFD-based rewiring in two directions. In one direction, to improve its efficiency, we showed how to overcome an un-

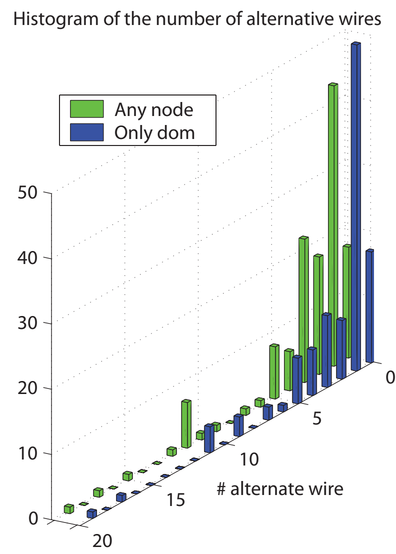

Figure 3.13: Histogram of the number of alternative wires of $term1$.

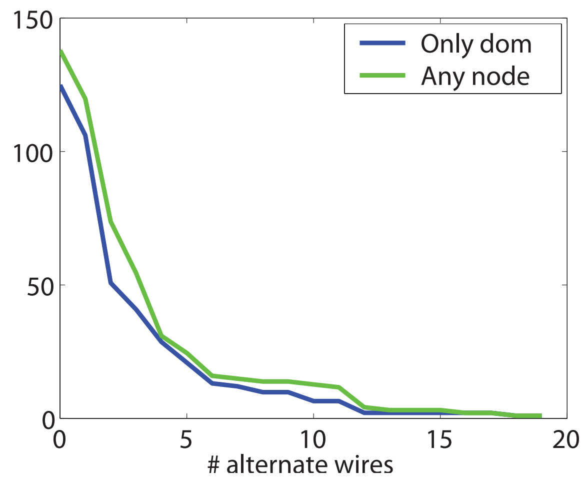

Figure 3.14: Show $r(n)$ of Figs. 3.13 for rewiring with only dominator nodes and with any nodes.

predictably large memory requirement of BDD-based SPFD rewiring by not expressing SPFDs explicitly. The proposed algorithm performs rewiring by solving a few SAT instances. Experimental results showed that our approach took less time than a BDD-based one and can rewire large circuits that the BDD-based one cannot finish in reasonable amount of time. Furthermore, while the runtime of BDD-based one remains virtually the same regardless of the number of candidate wires due to the bottleneck in SPFD computation, the runtime of our approach reduces almost linearly with the reduction in the number of candidate wires. Our approach requires about 13% of BDD-based runtime when there are at most 25 candidate sources for new wires for each to-be-removed wire (compounded with the number of candidate sinks, the number of candidate wires was observed to be up to about 50). Because our approach evaluates one rewiring instance in an order of milliseconds, it can be employed in a two-tier rewiring scheme that takes advantage of both efficiency from an ATPG-based rewiring and effectiveness from an SPFD-based rewiring without a large runtime penalty. Furthermore, because of its efficiency, it may be used in an optimization loop.

In another direction, a theory that allows considering a node $n \notin DOM(w_r)$ as a candidate destination for rewiring was presented. An algorithm based on the theory was also proposed. The experimental results show a large improvement over SPFD-based rewiring using only dominator nodes as destinations. Thus, the proposed algorithm makes more rewiring instances possible in several applications in which all dominator nodes may reside in congested areas. The proposed algorithm would be beneficial to an application that can indicate desirable locations of the new wires with priorities. However, as the theory requires SPFD to be represented explicitly, we suggest using a multi-tier approach in which a fast ATPG-based rewiring is used first, then a SAT-based SPFD rewiring is applied

next, and finally this algorithm is applied last if the former two fail. Regarding runtime of the proposed algorithm for larger circuits, it is not possible to apply this proposed algorithm to the whole circuit at once. Therefore, a windowing technique that extracts only a relevant part of the circuit should be used. However, an appropriate window should be selected in the context of an application at hand and thus is beyond the scope of this work.

Table 3.1: Rewirability and runtime comparison of BDD and SAT -based SPFD rewiring.

			BDD		SAT	
circuits	#node	#edge	rew	runtime (sec)	rew	runtime (sec)
cordic	21	77	61	1.51	**63**	2.72
b9	45	156	68	3.46	**75**	5.92
i3	46	172	32	6.77	32	44.87
frg1	51	183	68	10.13	**79**	79.56
C499	74	280	16	1189.31	16	**88.91**
apex7	95	322	173	26.36	**192**	**24.96**
i4	102	348	84	39.01	**88**	168.45
term1	72	244	127	11.08	**137**	22.28
C1908	127	432	172	1926.39	171	**161.57**
alu2	152	510	329	24.48	261	441.59
C432	154	538	389	217.91	359	244.38
ex2	135	433	199	100.47	**216**	**46.87**
x1	170	557	306	65.12	**307**	73.67
apex6	286	1025	458	2009.49	409	**331.07**
alu4	284	939	561	294.64	439	3447.72
x3	268	958	N/A	>20hr	**422**	**258.75**
dalu	373	1338	N/A	>20hr	**760**	**1292.43**
C5315	534	1772	N/A	>20hr	**694**	**9200.90**
total (cordic-alu4)			3043	5926.13	2844	5184.54

Table 3.2: Runtime details of results shown in Table 3.1.

	BDD	SAT				
	% SPFD comp. T	local fcn comp. T(sec)	eq check T(sec)	#inst (x1000)	#pass check	T/inst (ms)
cordic	47.68	0.34	0.33	2.4	70	1.12
b9	47.11	0.24	0.71	10.8	158	0.55
i3	31.02	0.34	0.76	51.0	160	0.88
frg1	55.58	31.24	19.52	22.8	3767	3.50
C499	79.62	1.04	0.07	25.8	16	3.45
apex7	57.81	1.61	2.38	27.8	490	0.90
i4	30.66	16.04	11.64	115.8	2484	1.45
term1	54.24	1.25	2.04	23.5	408	0.95
C1908	76.22	10.31	4.52	54.0	548	2.99
alu2	64.38	242.17	55.21	79.3	8218	5.57
C432	71.02	71.44	14.37	63.3	1551	3.86
ex2	72.37	1.9	2.86	66.7	614	0.70
x1	57.08	3.61	3.9	97.7	890	0.75
apex6	93.09	33.78	23.73	367.2	5056	0.90
alu4	68.66	1735.25	506.43	319.6	47323	10.79
x3	N/A	53026	8.57	326.7	1584	0.79
dalu	N/A	42.25	32.44	660.5	6057	1.96
C5315	N/A	3335.53	670.25	1371.8	72737	6.71
		total cordic-alu4		1327.6	avg	2.56

Table 3.3: Comparison when the circuit depth is limited.

	BDD		SAT			
circuits	rew	runtime (sec)	rew	runtime (sec)	#inst (x1000)	T/inst (ms)
cordic	27	1.75	**28**	2.72	2.19	0.96
b9	36	3.59	**40**	5.92	9.39	0.51
i3	32	6.59	32	44.87	45.08	0.87
frg1	22	10.99	**40**	79.56	20.19	3.30
C499	0	1142.41	0	**88.91**	17.82	3.90
apex7	158	25.47	**176**	**24.96**	24.54	0.83
i4	84	35.19	**88**	168.45	98.81	1.37
term1	126	10.80	**133**	22.28	20.55	0.93
C1908	109	1901.49	109	**161.57**	45.44	2.79
alu2	292	25.60	233	441.59	67.46	5.37
C432	198	230.36	**207**	244.38	58.97	3.79
ex2	178	100.44	**185**	**46.87**	62.90	0.65
x1	294	60.24	**296**	73.67	88.15	0.72
apex6	445	1996.70	395	**331.07**	341.90	0.86
alu4	499	288.6	380	3447.72	268.08	8.48
total	2500	5840.22	2342	3740.92	1171.46	(avg)2.36

Table 3.4: Scalability on the number of candidate wires.

	BDD			SAT			
limit to	total rew	total T (sec)	avg. %spfdT	total rew	total T (sec)	total #inst	avg. T/inst
100	2272	5699.75	66.29	2125	2340.32	637825	2.51
50	1640	5535.91	69.00	1650	1302.97	354841	2.57
25	1126	5447.98	70.56	1100	707.50	189671	2.64

Table 3.5: Rewiring ability of the proposed algorithm for the unlimited depth case.

circuits	rewiring ability (limited depth)				Total time (sec)
	Only dom	Any node	No dom	%inc of "Any node" over "Only dom"	
cordic	61	65	20	6.6	1.5
b9	68	70	53	2.98	3.5
i3	32	32	0	0.0	6.8
frg1	68	73	21	7.4	10.6
C499	16	16	0	0.0	1215.0
apex7	173	173	48	0.0	27.0
i4	84	136	76	61.9	94.1
term1	127	141	58	11.0	12.0
C1908	172	190	24	10.5	2080.5
alu2	329	364	137	10.6	26.0
C432	389	412	212	5.9	225.7
ex2	199	240	90	20.6	100.1
x1	306	409	210	33.7	73.8
apex6	458	538	177	17.5	2049.6
alu4	561	638	265	13.7	315.9
indicate		avg all		13.5	
large circuits		avg large		20.1	

Table 3.6: Rewiring ability of the proposed algorithm for the limited depth case.

circuits	rewiring ability (limited depth)				Total time (sec)
	Only dom	Any node	No dom	%inc of "Any node" over "Only dom"	
cordic	27	29	2	7.4	1.7
b9	36	37	3	2.8	3.5
i3	32	32	0	0.0	6.5
frg1	22	28	12	5.8	11.5
C499	0	0	0	0.0	168.0
apex7	158	158	47	0.0	26.5
i4	84	112	52	33.3	78.1
term1	126	139	53	10.3	11.2
C1908	109	122	19	12.0	2043.3
alu2	292	316	118	8.2	27.1
C432	198	230	80	16.1	243.0
ex2	178	220	76	23.6	99.6
x1	294	389	195	32.3	66.1
apex6	445	512	153	15.1	2027.6
alu4	499	540	210	8.2	315.0
indicate		avg all		13.1	
large circuits		avg large		17.7	

Chapter 4

Yield Improvement of an FPGA Family

4.1 Introduction

Field-programmable gate arrays (FPGAs) have recently been widely used in digital systems, especially for low- to medium-sized applications. However, it is well known and verified that the area efficiency of FPGAs is about 35 times worse than that of the standard-cell implementation [12]. Furthermore, their power consumption is about 14 times higher. Recognizing that most applications implemented on FPGAs use multipliers and memory units, contemporary FPGAs provide these specialized functional blocks. Since density gains of specialized blocks could be up to two orders of magnitude compared to the FPGA implementation, the area efficiency gap could be significantly reduced if such blocks are well utilized. Furthermore, power consumption reduces as a lot of programmable components are removed and circuits can be optimized. There are several works dedicated to specialized blocks. Architectural aspects have been studied in [56, 57, 58], to name a few. Mapping tools supporting various specialized blocks have also been reported [59, 60]. Although these papers showed promising results for specialized blocks, they did not consider the interaction between different specialized blocks. As a result, effects of specialized blocks considering a large set of applications cannot be inferred.

A typical flow of an FPGA architecture design begins with evaluating new features over a set of benchmarks for performance, area and power consumption among other metrics and then forming a family of several FPGAs from the same base architecture so that they satisfy designs of various sizes and requirements. Although widely ignored in the literature, a selected set of FPGA sizes in a family affects the overall cost resulting from extra unused area and manufacturing yields. Thus, determining an appropriate set of FPGAs that contain the right mix of resources is an important problem. We call this problem "FPGA family composition" in this work[1]. It becomes even more important and more difficult for contemporary FPGAs containing specialized blocks, especially with the fact that FPGAs are entering more domains, hence requiring FPGA architects to study more dedicated hardware types to be embedded in FPGAs. FPGA family composition under the influence of specialized blocks is also considered in this work. To the best of the our knowledge, this is the first time that the FPGA family composition problem is formally studied.

Previous work on choosing an appropriate FPGA size has mostly focused on first choosing a specific domain, and then selecting the number of resources such as cross-bars, floating point units, etc., on the FPGA fabric. Simulated annealing (SA) and integer linear programming (ILP) technique were used for architecture exploration [62, 63]. These approaches have two limitations : 1) SA and ILP would take a very long time and depending on the mixture of resources, SA might not even converge to a good solution. 2) the set of "representative" applications to be implemented by the FPGA should be determined in advance. Thus, although the resulting architecture is well tuned for these representatives, it would not necessarily provide good estimates on how the architecture would perform if a new

[1]The preliminary version of this work can be found in [61]

application from the same application domain is mapped to the architecture [62]. Note that even though timing and congestion can be estimated using floorplanning during architecture exploration [63], the estimations are not accurate due to the lack of detailed routing information. Furthermore, power consumption is not considered because several important factors are not known [63]. This work introduces a high-level architecture exploration which considers all designs in the domain. The main objective is to minimize the total area by appropriately composing an FPGA family. Even though timing and power consumption cannot be directly captured in the process, they are generally reduced with the FPGA area as a result of shorter connections. The technique can be used in conjunction with the existing approaches to ensure that the resulting architecture suitable for the selected representatives is also reasonable for other designs.

The rest of the chapter is organized as follows. First, mapping from designs to FPGA functional blocks is discussed in Section 4.2. Interaction among these blocks is also formalized in this section. The FPGA family composition problem is formally stated in Section 4.3. The problem is solved in two steps: finding the candidates to be included in the family and choosing a good set from the candidates. The first step is elaborated in Section 4.4, while the second step is discussed in Section 4.5. Experimental results are reported in Section 4.6. Finally, the work is summarized in Section 4.7.

4.2 Basic Application Modules and FPGA Building Blocks

Basic application modules are defined as a set of well-defined modules from which a designer can compose a particular design. For example, a set of modules in a digital filter may include multipliers, adders and FFs. In this work, the basic application modules are assumed to be LUTs, multipliers and RAM blocks, although

our technique could be used to study any set of specialized blocks. Considering all applications, the number of modules used for each type can be plotted as a histogram. We analyzed many designs collected from [12, 56, 59] and the histogram of each module was plotted and fitted with a normal distribution.[2] In the rest of this chapter we use LUT, MEM and MUL to represent random variables (RVs) of modules LUT, block RAM and multiplier, respectively. Each of the RVs has a normal distribution.

FPGA building blocks are a set of distinct functional blocks provided in FPGAs. Different FPGA families may have different sets of building blocks. FPGA building blocks may differ from an application's basic modules. However, every module must be implementable by at least one of FPGAs' building blocks. The mapping between application basic modules and the building blocks may not be one-to-one: two modules could be mapped to the same type of block, and one module could be implemented by several blocks. RVs of basic modules have to be mapped to RVs of FPGA resources so that the requirement of each resource type can be computed and FPGA resources can be allocated to satisfy the demand of users.

[2]Although the histogram of all circuits ever implemented in FPGAs may not be a normal distribution, a large set of their representative circuits used in FPGA architecture design is a normal distribution by the central limit theorem. The area of each resource type is at least an order of magnitude lower than an FPGA area. Thus, assuming continuous distribution is reasonable as the quantization error is limited.

4.2.1 Mapping Application Module Distributions to FPGA Block Distributions

Assuming that the distributions of design modules are given (the distributions may be dependent), they must be translated to distributions of FPGA blocks. Let A, B and C be design modules and W, X, Y and Z be resources available in an FPGA. The RVs corresponding to modules and resources are denoted by their name with subscript rv. Design modules implementable by a resource type X can be described by either

1. $X = S\{n_a A, n_b B, n_c C\}$ which means that one functional block of type X can implement n_a, n_b, n_c of module A, B, C, respectively, at the same time, or
2. $X = E\{n_a A, n_b B, n_c C\}$ which means one block of type X can implement just one type of modules A, B, and C at a time with the quantity specified.

For a given FPGA, we can describe its resources using above descriptions. Although one module type can be mapped to several combinations of resources, mapping design module distributions to those of FPGAs' building blocks to minimize the required area can be performed using resource areas. As our objective is to minimize the FPGA area, once such map is obtained for each module type, it will used throughout this chapter.

4.2.2 Interactions between Different Types of Resources

Some resource types are partially compatible, resulting in one being able to implement the other. Examples of resource usage interactions are shown in Figure 4.1 for two types of resources. Part (a) shows an example in which an FPGA has two types of resources: CLB4 and CLB8, each with 4 and 8 LUTs respectively.

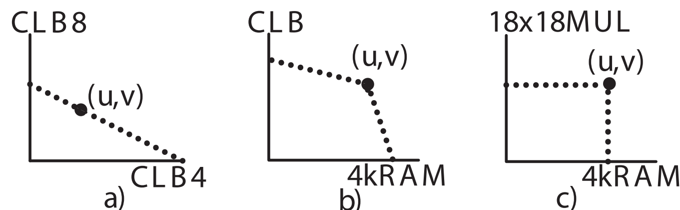

Figure 4.1: Interactions between two types of resources. (See Section 4.2.2 for details)

An FPGA having u resources of type CLB4 and v units of CLB8 can be shown as Point (u, v). A design requiring $u - 2$ and $v + 1$ resources can still fit on the FPGA because two CLB4 blocks can be combined to implement a CLB8. Thus, the interaction between these two resources is a line passing through (u, v) with the slope of -0.5 as shown in Figure 4.1(a). Any design that falls under the line can fit on the FPGA.

Now, assume that the resources are CLB and 4kRAM as shown in part (b) of the figure. If we move from (u, v) toward the y-axis, we use some 4kRAMs to implement CLBs. If we move toward the x-axis, some CLBs are used to implement 4kRAMs. However, the efficiency of using CLBs to implement 4kRAMs is different from that of using 4kRAMs to implement CLBs. Thus, the two lines have different slopes as shown in the figure. If the resources are 18x18MULs and RAMs which cannot implement each other, designs have to use no more than u and v of RAMs and 18x18MULs respectively to be able to fit in the FPGA and the interaction is shown in Figure 4.1c).

The resources exhibit a relation in Figure 4.1(a) if the two resources can implement each other with the same silicon area efficiency, or Figure 4.1(b) for different efficiencies. If they cannot implement each other, we have Figure 4.1(c). In general, the interaction of one resource with others can be described by a hyperplane,

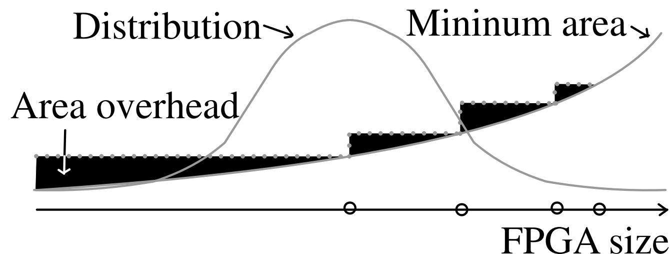

Figure 4.2: Effects of a finite-member FPGA family.

one for each resource type, as follows.

$$x_i + \sum_{j \neq i} a_{ij} \cdot x_j = b_i \tag{4.1}$$

, where $a_{ij} \in (0, \infty]$ is the efficiency of using x_i to implement x_j and b_i depends on a particular FPGA under consideration. For example, if the resource of type i and j are 4-LUT and 9x9 multiplier, respectively, one 9x9 multiplier can be implemented using 81 LUTs. Thus, $a_{ij} = 1/81$. If $a_{ij} = 0$, the resource of type j cannot be implemented using type i.

The maximum number of hyperplanes is n, where n is the number of resource types on an FPGA. However, it is possible that two resource types will have the same hyperplane, such as that in Figure 4.1(a). Thus, the minimum number of hyperplanes is one. In conclusion, interactions among n resource types can be described by m n-variable linear equations, where $m \leq n$.

4.3 FPGA Family Composition

Assume that an FPGA has n types of functional blocks (resources). An FPGA can be represented as an n-tuple vector, $\vec{x}$, in which x_i is the number of blocks of type i available in the FPGA. Let $pdf(\vec{x})$ be the distribution of all mapped designs and $A(\vec{x})$ be the minimum area of an FPGA that can fit a design requiring $\vec{x}$, in this case x_i represents the number of resource of type i that the design needs. The total area of all FPGAs to implement all designs in $pdf(\vec{x})$ is

$$TotalArea = N \cdot \int A(\vec{x}) \cdot pdf(\vec{x}) d\vec{x} \tag{4.2}$$

where N is the number of designs, which is supposedly a large number. The probability density of the distribution toward the upper end is small because few applications exist that require huge resource requirements. Thus, it may not be practical to provide FPGAs for designs in that range. As a result, there is a specific amount of design coverage, say PMAX that our FPGA architecture should be able to implement.

As $pdf(\vec{x})$ is assumed given, thus the total area is determined by $A(\vec{x})$. The effect of $A(\vec{x})$ on the total area can be seen as follows. For simplicity, let's consider an FPGA with one type of resource. Consider Figure 4.2 which implements all designs using an FPGA family containing 4 FPGAs, shown as circles on the X-axis. The area overhead is shown as shaded region. It can be easily seen that as the number of FPGAs in the family increases, the area overhead reduces and vanishes if there are infinitely many FPGAs in a family. FPGA family composition problem can be stated as follows.

Problem statement

For a given distribution of all designs, find a set of FPGAs containing at most M FPGAs that requires the minimum total area for covering PMAX of all designs.

Solving the problem is equivalent to choosing M points (FPGAs) in an n-dimensional space that collectively minimize the area overhead. Thus, due to its high complexity, the problem is solved in two steps:

1. Finding candidate FPGAs detailed in Section 4.4. The FPGA architecture solution space is multi-dimensional and hard to optimize. To reduce the number of architectural candidates, the solution space is projected to a one dimensional solution space by mapping the multi-dimensional space to a number indicating application coverage percentage. For each selected coverage percentage, a minimum area FPGA is determined.

2. Choosing at most M FPGAs from the candidates that minimize the total area. (See Section 4.5)

The above steps do not result in an optimal solution. Although the second step guarantees an optimum solution, its solution can be affected by the selected candidates. The overall solution may not be optimum because

1. there are a limited number of selected coverage points, and

2. the minimum area FPGA for a given coverage may not be unique, due to the given probability distribution and / or the introduction of errors in the numerical algorithm used.

4.4 A Minimum Area FPGA Covering a Given Percentage of Designs

For a given set of hyperplanes obtained from Section 4.2.1, the total probability of all designs being covered by the FPGA can be computed by multiple integrals. However, such an approach would be complicated and inefficient. Thus, in this section, the transformation of the problem space to be efficiently solved will be discussed first and the optimization algorithm that can be applied in the transformed space will be presented next.

4.4.1 Transforming the Hyperplanes

The percentage coverage by a specific size FPGA can be computed by multiple integrations with limits from hyperplanes. If hyperplanes are transformed to be axis-aligned ones, the percentage coverage becomes a cumulative distribution function (CDF), which can be computed efficiently. Although the CDF of a multivariate normal distribution has no closed form, an efficient computation for it is known [64].

The transformed hyperplanes will become axis-aligned if the normal vectors of the original hyperplanes are used in the linear transformation. This transformation will change the probability distribution. The transformed RVs may be correlated even if the original RVs are uncorrelated. The transformed distribution is a normal distribution if and only if the resulting covariance matrix is positive definite. This can be guaranteed if the linear transformation is linearly independent. However, if there are hyperplanes similar to case b) of Figure 4.1, it may happen that the covariance matrix is not positive definite and the obtained probability is not normal anymore.

4.4.2 Finding the Minimum Area FPGA

After the transformation, the probability that an FPGA with a specific amount of resources, $\vec{y}$, covers designs can be computed using a CDF. Furthermore, the area of the FPGA which is the summation of areas of all resources in the original problem, *ie.*, $\vec{y} \cdot \vec{1}$, becomes $\vec{x} \cdot \vec{C}$, where $\vec{x}$ and $\vec{C}$ are $\vec{y}$ and $\vec{1}$ after the transformation. Thus, the minimum area FPGA that covers $a\%$ of all designs can be formulated as

$$\begin{aligned} minimize \quad f(\vec{x}) &= \vec{x} \cdot \vec{C} \\ s.t. \quad CDF(\vec{x}) &= a \\ B^{-1}\vec{x} &\succeq 0 \end{aligned} \tag{4.3}$$

where B is the linear transformation used in Section 4.4.1 and $\succeq$ means $\geq$ for every component. Since there is no closed form for the CDF of a multivariate normal distribution, it has to be computed numerically [64]. Thus problem (4.3) is in an oracle form. Thus, the optimization problem is difficult to solve because an initial feasible solution is difficult to obtain. The problem can be extended to (4.4). Later on we will show that the solution to (4.4) is the same as that of (4.3).

$$\begin{aligned} minimize \quad f(\vec{x}) &= \vec{x} \cdot \vec{C} \\ s.t. \quad 1 - CDF(\vec{x}) &\leq 1 - a \\ B^{-1}\vec{x} &\succeq 0 \end{aligned} \tag{4.4}$$

Let $x_2 = \mathcal{C}(x_1, \rho)$ defined by $P\{X_1 \geq x_1, X_2 \geq x_2\} = \gamma$, where (X_1, X_2) has a bivariate normal distribution with mean (0,0) and variances $\sigma^2_{X_1}, \sigma^2_{X_2}$ and correlation ρ. It is shown that $\mathcal{C}(x_1, \rho)$ is strictly concave over x_1 for $\gamma \in (0, 1)$ and any value of ρ [65].

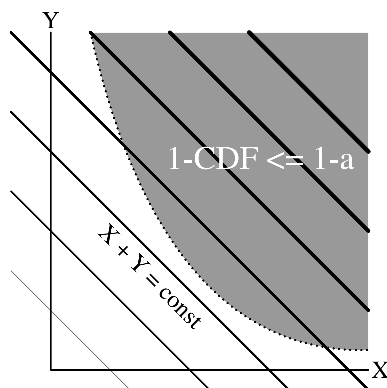

Figure 4.3: Demonstration of (4.4) for two-variable case.

Furthermore, $\mathcal{C}(x_1, \rho + \epsilon) > \mathcal{C}(x_1, \rho)$, for $\epsilon > 0$ [65]. However, $CDF(x_1, x_2) = P\{X_1 \leq x_1, X_2 \leq x_2\}$. By linear transformation, a contour of $CDF(x_1, x_2) = \beta$ is strictly convex over x_1. As a result, $1 - CDF(\vec{x}) \leq 1 - a$ is also convex for 2-dimensional case as shown in Figure 4.3. Any marginal distribution of a multivariate normal distribution is also a normal distribution. Thus, we can infer that the feasible set is also convex in a higher dimension.

In the original problem, $\vec{y} \succeq 0$ is convex and thus its transformed counterpart is also convex. Because intersection of convex sets yield a convex set, the constrain of (4.4) is convex. A local optimum point x^* must satisfy

$$\nabla f(x^*)'(x - x^*) \geq 0, \quad \forall x \in X. \tag{4.5}$$

If f is convex over a feasible set X, x^* is also a global minimum. In our problem, (4.4), the objective function is linear, thus convex, and the feasible set is strictly convex, a local optimum x^* is the global optimum.

The algorithm we use belongs to the feasible direction method which proceeds

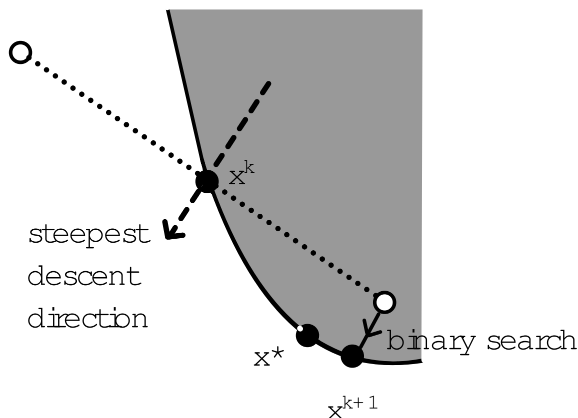

Figure 4.4: Show how to compute x^{k+1}.

in an iterative manner. Let x^k be the current feasible solution, at iteration k.

$$x^{k+1} = x^k + \alpha^k d^k. \tag{4.6}$$

d^k is a feasible and descent (regarding objective function) direction, ie., $\nabla f(x^{k+1})' d^k < 0$ and α^k is a step size. If $x^{k+1} = x^k$, it means that $d^k = 0$; $\nabla f(x)' d^k > 0, \forall x \in X | \|x - x^k\| < \epsilon$. Therefore, x^k is the optimum solution. Traditionally, x^{k+1} is obtained by solving a subproblem. However, since we have the feasible set only in oracle form, we have to find x^{k+1} numerically by searching around x^k. Using the fact that our objective function is linear, a simple, yet effective search can be performed. Consider Figure 4.4. A hyper-ball of radius ϵ in $n-1$ dimensions centered at x^k can be drawn, where n is the dimension of the solution space. x^{k+1} is determined in 2 steps :

1. Randomly select a point on the ball. There are two points, shows as white circles, on the hyper-ball in Figure 4.4, and

2. Find a point x' along the contour $1 - CDF(\vec{x}) \leq 1 - a$ by performing a binary search along the steepest descent direction, $\nabla f = C$.

These two steps will be repeated until $x' \cdot \vec{C} < x^k \cdot \vec{C}$ and $B^{-1} \cdot x' \succeq 0$ and that x' is used as x^{k+1}. Note that the chosen x^{k+1} may not yield the most cost reduction. However, the optimum solution is guaranteed to be found as the feasible set is convex, but may take more steps. It is possible that x^* is in an obscure location such that the feasible points oscillate between x^k and x^{k+1}. To ensure convergence, ϵ must be small and gradually decrease in size.

If the feasible region is contained by the perpendicular plane in the $-\vec{C}$ direction, it implies that we are at the optimal solution because $C' \cdot (x - x^k) > 0, \forall x \in X$ (C' is the transpose of C). Therefore, the optimality checking can be combined with x^{k+1} finding. In particular, if there is no feasible point during x^{k+1} finding, x^k is the optimal solution. However, to check for optimality, ϵ must be small. In contrast, using a large ball (i.e., using a large ϵ) in finding x^{k+1} may reduce the number of steps to get to the optimum solution. Thus, we choose to use two separate balls.

An initial solution can be any point within the feasible region. The choice will not affect optimality, but may affect run time. Note that not all the points in the solution space correspond to a valid FPGA due to discretization but in practice discretization had negligible effects in our implementation.

4.5 Finding a Good FPGA Family

The selected minimum area FPGAs for various percent coverage points may not be well-formed. As a result, adding a minimum FPGA that covers 80% of the applications to a family covering 60% of the applications does not necessarily mean

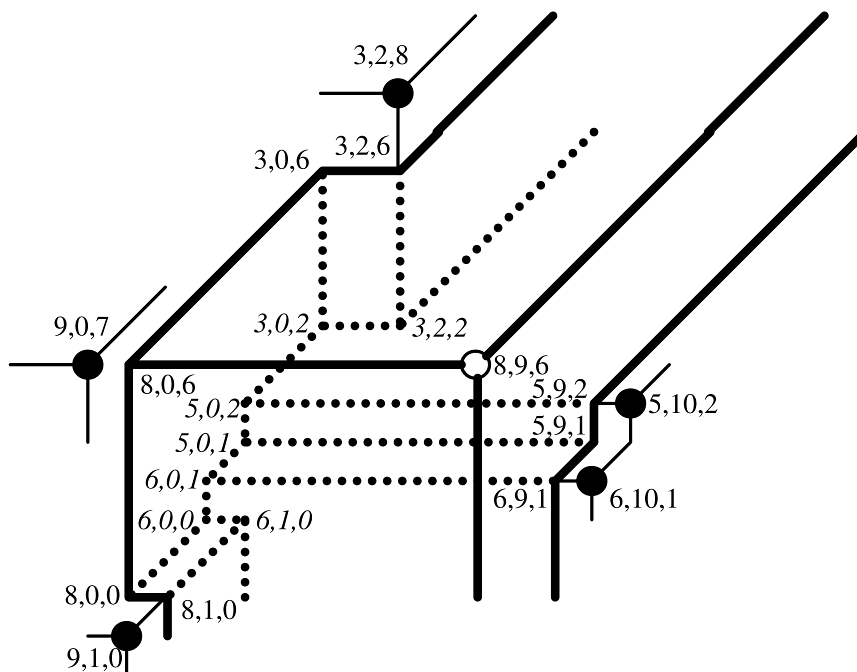

Figure 4.5: The increase in coverage by adding one more FPGA (8,9,6) into a family is shown with bold solid lines. (The boundary of the previous coverage is shown using dashed lines)

that the family will cover 80% of the applications. Consider Figure 4.5. Imagine that only point (3,2,8), which covers 60%, is included in the family. Adding point (8,9,6) that covers 80% to the family will result in a total coverage of more than 80% because (3,2,8) covers some space which is not covered by (8,9,6).

The selection can be performed in a branch-and-bound manner. When one more FPGA is added into the family, the increase in coverage must be computed. Consider Figure 4.5 again. Assuming that (3,2,8),(9,0,7),(9,1,0),(6,10,1) and (5,10,2), all in black dots, are included, and (8,9,6) is going to be included. The volume of the increased coverage is enclosed by bold lines and dash lines, which are parts of the boundary of the coverage of the partial solution. As this

volume is not a hyper-box, the plane sweep technique is used to partition it down to several hyper-boxes so that the probability they represent can be computed as CDFs. In general cases, included points may reside in the volume. Thus, the plane sweep technique is also used to break down these inner volumes so that their probability content can be computed, using CDF, and subtracted from the outer volume. For brevity, the algorithm is omitted. It is important to note that a minimum FPGA that covers PMAX applications must be included in a family.

4.6 Experiments

The design basic modules used in our experiments were LUT, RAM and 9x9MUL (they were also used as RV names for the corresponding modules) which are in units of number of LUTs, memory bits and multipliers, respectively. Many designs were collected from [59, 56, 12] and histograms of basic modules were plotted and fitted with normal distributions. The parameters of obtaining distributions are shown in Table 4.1. We assume that these distributions are independent. However, our proposed technique does not pose any restrictions on the dependency of distributions.

In the experiments, two types of FPGAs are considered: one Virtex-like FPGA containing only CLBs and block RAMs, and another with additional 18x18 bit multipliers. A CLB has 4 4-input LUTs and a block RAM is of size 4k bits. The RVs of FPGA resources are named CLB, 4kRAM and 18x18MUL for CLBs, block RAMs and multipliers, respectively. In the FPGA without the 18x18MUL, we have $CLB = E\{4 * LUT, MUL/20.5\}$ and $4kRAM = E\{4096 * MEM\}$. Thus, mapping from design basic module distributions to those of FPGA resources can be shown in Table 4.2. In contrast, for the FPGA with 18x18MUL, basic modules can be directly mapped to their corresponding blocks.

The logic part of a Virtex CLB implemented in $0.13\mu m$ technology takes $3660\mu m^2$ [66]. Projected to $65nm$, it takes $0.000915mm^2$. Based on the Virtex routing architecture description, routing resource per CLB is about $0.0011mm^2$. The CLB area is the summation of the logic and routing area. Block RAM of 4k bits takes $0.0208mm^2$ at $65nm$, projected from [58]. An 16x16 bit Booth multiplier takes $0.837mm^2$ in $0.6\mu m$ technology [67]. We believe that using an industrial, more aggressive design flow, 18x18 multiplier[3] should fit in the same area. Projected to $65nm$, the embedded multiplier should take $0.0098mm^2$.

Modules usually implemented in one resource type can also be mapped to other types. Although slower, particular logic functions can be implemented in multipliers. If there are multipliers already on critical paths, MUXs not on critical paths can be implemented using multipliers without performance degradation [59]. On average, the number of LUTs reduces by 70, with the increase of 15 9x9 multipliers. A 4-input LUT is essentially a 16x1 memory. Thus, LUTs can be packed into larger memory blocks [60, 68]. One 9x9 multiplier can be implemented using 81 LUTs (with associated logic)[4]. Note that a multiplier can be implemented using memory but it requires exorbitant amount of memory [69]. Thus, implementing a multiplier using memory blocks is not considered in our experiments. Using this information, a linear transformation of FPGAs with 18x18 multipliers can be set up as shown in Table 4.3. For FPGAs without multipliers, the same transformation is used, but with the last column and row removed.

---

[3]Although benchmark circuits contain only 9x9 multiplier, recent commercial FPGAs provide dedicated 18x18 multipliers. Thus, an 18x18 multiplier will be used to implement one 9x9 multiplier

[4]Although there are several compact multiplier implementations, they are irregular and not suitable for implementation on LUTs. The carry save multiplier construction is the most common implementation.

Table 4.1: Parameters of normal distributions of design modules.

	LUT	RAM	9x9MUL
mean	5.824×10^3	1.743×10^4	4.460×10^1
sigma	2.055×10^4	2.042×10^4	8.862×10^2

Table 4.2: Mapping from design module distributions to FPGA resource distributions.

without MUL18x18	CLB	=	0.25*LUT + 20.25*MUL
	4kRAM	=	MEM/4096
with MUL18x18	CLB	=	0.25*LUT
	4kRAM	=	MEM/4096
	MUL18x18	=	MUL

Table 4.3: The linear transformation used in the experiments.

	CLB	4kRAM	18x18MUL
CLB	1	32/4096	1/20.25
4kRAM	15/4	1	0
18x18MUL	70/60	0	1

The algorithm detailed in Section 4.4.2 was applied to find a minimum area FPGA for each selected coverage point. The resulting resource area of the minimum area FPGAs with and without 18x18 multipliers are shown in Table 4.4. Data in each row corresponds to one percent coverage point. It can be observed that minimum area numbers increase with the percent coverage for both architectures. Furthermore, the minimum area increases faster for higher coverage percentages. The most important observation is about the area difference (column 9) between the two architectures (columns 4 and 8). We can see that FPGAs with 18x18 MUL require less area to cover the same percentage compared to FPGAs without multipliers. In addition, the difference tends to increase with percent coverage.

As a basis for a comparison, a baseline family selection was performed as follows. The designs collected were divided into 4 categories: 1) those use only CLB, 2) those use CLB and RAM, 3) those use CLB and MUL and 4) those use all 3 resources. Within each category, 3 designs were selected in such a way that other designs are dominated by the selected designs in term of resource usage. In some categories, only 2 designs were enough. But to reduce the area overhead (recall Figure 4.2), one more design is selected from the design at the median of the category. The module usages of the selected designs were mapped to resource usages of FPGAs with and wihout MUL using data from Table 4.2. At this point, there are 12 FPGAs for each architecture. For each FPGA, its coverage over the distribution shown in Table 4.1 was computed. Some FPGAs far away from the center of the distribution are large but have the comparable coverage as other small FPGAs. For example, one selected FPGA of size 22.7 mm^2 covers 61%, while another FPGA of size 0.27 mm^2 covers 62%. Thus, those FPGAs were eliminated from the family. As a result, the baseline family for each architecture contains 10

Table 4.4: A minimum area FPGA for selected percent coverage points.

% cover	W/O 18x18 MUL CLB area	4kRAM area	tol area (mm^2)	With 18x18 MUL CLB area	4kRAM area	MUL area	tol area (mm^2)	area dif. (mm^2)
0.35	0.237	0.063	0.300	0.014	0.222	0.018	0.254	0.046
0.40	0.902	0.009	0.911	0.289	0.051	0.428	0.767	0.144
0.45	1.528	0.135	1.663	0.872	0.121	0.391	1.384	0.279
0.50	2.162	0.082	2.244	1.476	0.235	0.257	1.969	0.275
0.55	2.786	0.020	2.807	1.990	0.020	0.534	2.545	0.262
0.60	3.427	0.032	3.459	2.660	0.009	0.138	2.807	0.652
0.65	4.072	0.177	4.248	3.163	0.287	0.537	3.987	0.262
0.70	4.771	0.004	4.775	3.898	0.099	0.112	4.109	0.666
0.75	5.514	0.119	5.634	4.562	0.412	0.190	5.164	0.470
0.80	6.350	0.082	6.433	5.344	0.178	0.182	5.704	0.729
0.85	7.328	0.245	7.573	6.216	0.373	0.208	6.797	0.776
0.90	8.547	0.150	8.696	7.432	0.154	0.183	7.770	0.927
0.95	10.358	0.020	10.378	9.085	0.073	0.129	9.287	1.091
0.98	12.384	0.074	12.458	10.936	0.204	0.501	11.641	0.817

Table 4.5: Effects of the number of FPGAs in a family.

# in a family M	without 18x18 MUL in a family				
	family	expected area (mm^2)	over 1 FPGA	over conventional	
2	00000100000001	6.805	0.454	-0.051	
4	10000100001001	4.479	0.640	0.308	
6	10001001010101	3.891	0.688	0.399	
8	10010101010111	3.668	0.706	0.434	
10	11010101011111	3.545	0.715	0.453	
12	11101101111111	3.471	0.721	0.464	
14	11111111111111	3.418	0.726	0.472	
# in a family M	with 18x18 MUL in a family				over w/o MUL
	family	expected area (mm^2)	over 1 FPGA	over conventional	
2	00000001000001	6.101	0.476	0.026	0.103
4	10000100010001	3.943	0.661	0.371	0.120
6	10010001010101	3.335	0.713	0.468	0.143
8	10000100111111	2.889	0.752	0.539	0.213
10	11010011011111	2.770	0.762	0.558	0.219
12	11111101101111	2.752	0.764	0.561	0.207
14	11111101101111	2.752	0.764	0.561	0.195

FPGAs. The baseline family of FPGAs with and without MUL covering 98% has an expected area of 6.266 and 6.475 mm^2, respectively.

Using minimum area FPGAs listed in Table 4.4 , the algorithm, highlighted in Section 4.5, was used to solve the FPGA family composition problem, stated in Section 4.2, for family sizes, M, between 2 to 14. The result is shown in Table 4.5. The table is divided into an upper and a lower sections for FPGAs without and with 18x18MULs, respectively. Each row contains data for a specific value of M, shown in the first column. The second column lists FPGAs without MUL from Table 4.4 selected for the family. The leftmost and rightmost bits represent FPGAs covering 35% and 98%, respectively. If the minimum area FPGA covering a particular percent coverage is included, the corresponding bit is marked 1 and 0 otherwise. The third column shows expected areas (eq. (4.2)/N). Area improvements, computed by $(ref - new)/ref$, of the families over having only the FPGA covering 98% are shown in the fourth column. The fifth column shows area improvements over the baseline selection. To highlight the effect of having 18x18MUL in FPGAs, area improvements for families of the same size from the two architectures are shown in the last column. In general, we can see that there are more 1s around FPGAs with higher percent coverage. This is because the area increases faster for the higher coverage (See Table 4.4). Thus, to minimize area overhead, more FPGAs were selected from the higher percent coverage (See Figure 4.2). Interestingly, for FPGAs with 18x18MUL, the same family is obtained for both $M = 12$ and $M = 14$ because the volume in the transformed space covered by the minimum area FPGAs for 65% and 80% coverage can be collectively covered by FPGAs with smaller areas. For both architectures, the expected area decreases as M increases. Hence, the area improvement over one FPGA and conventional selection are also increased. The effectiveness of the proposed technique can be

observed from the area improvements over the baseline (Column 5) even when M is smaller than that of the baseline family, whose M is 10. The benefit of having MUL in FPGAs is confirmed by the area improvement (Column 6) and it increases with M from 10% at $M = 2$ to about 20% when $M \geq 8$.

4.7 Summary

An FPGA family composition problem was formulated in this chapter. Due to the difficulty of the problem, it was solved in two steps. Minimum area FPGAs of selected percentage coverage points were chosen from the multi-dimensional solution space. For each percentage coverage point, a minimization problem was set up and shown to be convex, although in an oracle form. Thus, we presented an algorithm to solve it optimally within numerical errors. Among the resulting FPGAs, a family of limited FPGAs was formed by a simple branch-and-bound technique. However, as the solution space is multi-dimensional, minimum area FPGAs are not totally ordered. Thus, an algorithm using a plane sweep technique was created to compute the probability increase for including one more FPGA into the family. The result showed that the technique can produce families with the same number of members as the baseline families, but with area improvement of 0.453 and 0.558, for an architecture with and without multipliers, respectively. Although the experiment was carried out on a distribution projected from a small set of designs, the methodology proposed can be used for the more accurate distribution and the same benefit should be obtained. The process of mapping design distributions to those of FPGA resources was also discussed. Interactions among resources were described by hyperplanes making the FPGA family composition efficiently solvable after the transformation. However, the interactions may be nonlinear and should be address in future work.

Chapter 5

Conclusions and Future Work

It was shown in this thesis that FPGA yield will significantly reduce in future technologies. For example, the yield will be only 21% at 2021 if the yield is now at 75%. Thus, given higher fabrication equipment costs, effective yield improvement techniques are needed.

A combination of defect-tolerant schemes was shown here to maintain a satisfactory yield level until 2021. Although the spare column scheme, shown here to be applicable to a modern FPGA architecture, can improve FPGA yield because it helps recover from faults in both logic blocks and interconnect networks, its effectiveness degrades as fabrication technologies progress. Several schemes were proposed to tolerate faults in interconnect networks. However, interweaving medium-length interconnect networks of modern FPGAs creates interdependency among their components. Breaking these dependencies requires additional area, making most defect-tolerance schemes ineffective due to their high area overheads. But, applying the proposed scheme to a network of segments of length one and long wires helps maintain the FPGA yield at about 90% up to the technology of 2020. As manufacturing cost significantly increases in the future generations, reducing product cost demands even higher yield which can be obtained by applying a combination of the presented techniques. Possible combinations of the proposed yield improvement techniques can be seen in Figure 5.1-5.2. Both defect-tolerant architecture and rewiring for yield improvement studied in this thesis can

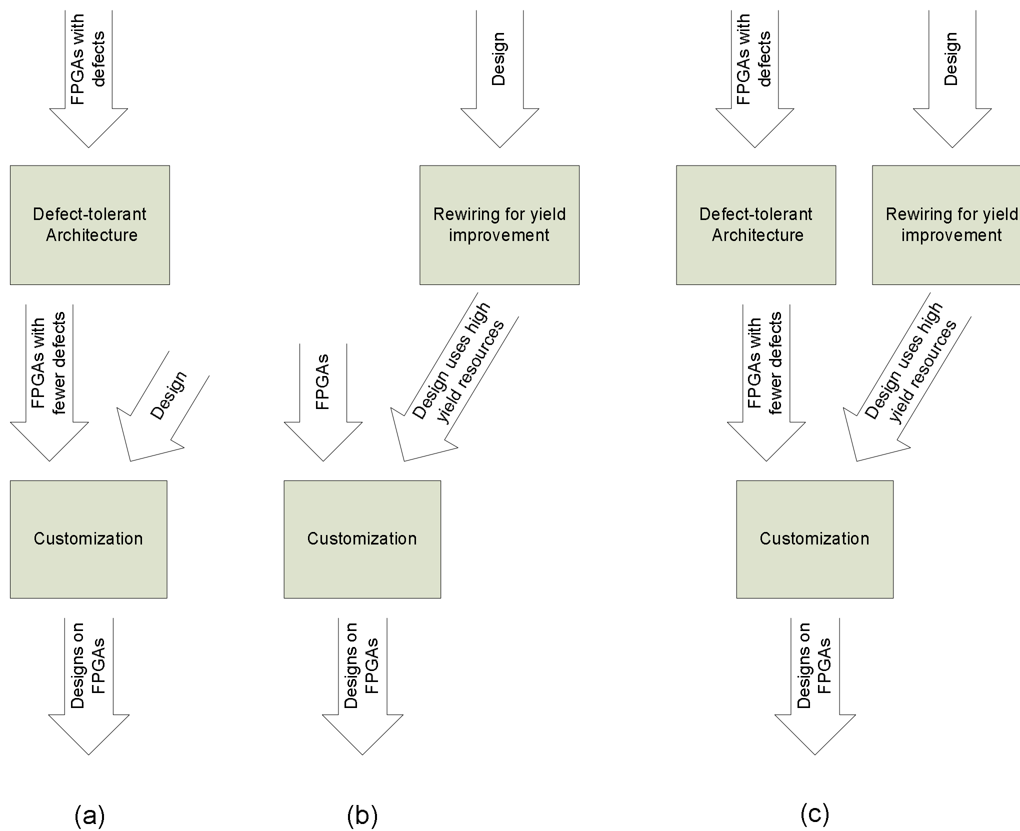

Figure 5.1: Employing multiple yield improvement schemes to enhance the customization approach.

be used in conjunction with either customization or design-specific approaches.

Defect-tolerant architecture reduces the expected number of faults into the range that customization can be applied effectively. Instead of applying customization approaches on FPGAs without defect-tolerance, they should be applied to defect-tolerant FPGAs as shown in Figure 5.1(a). In future technologies, even though the expected number of faults on FPGAs would be high, defect-tolerant architectures will mask some of those faults. As a result, the expected number of faults will be small enough to make customization approaches effective.

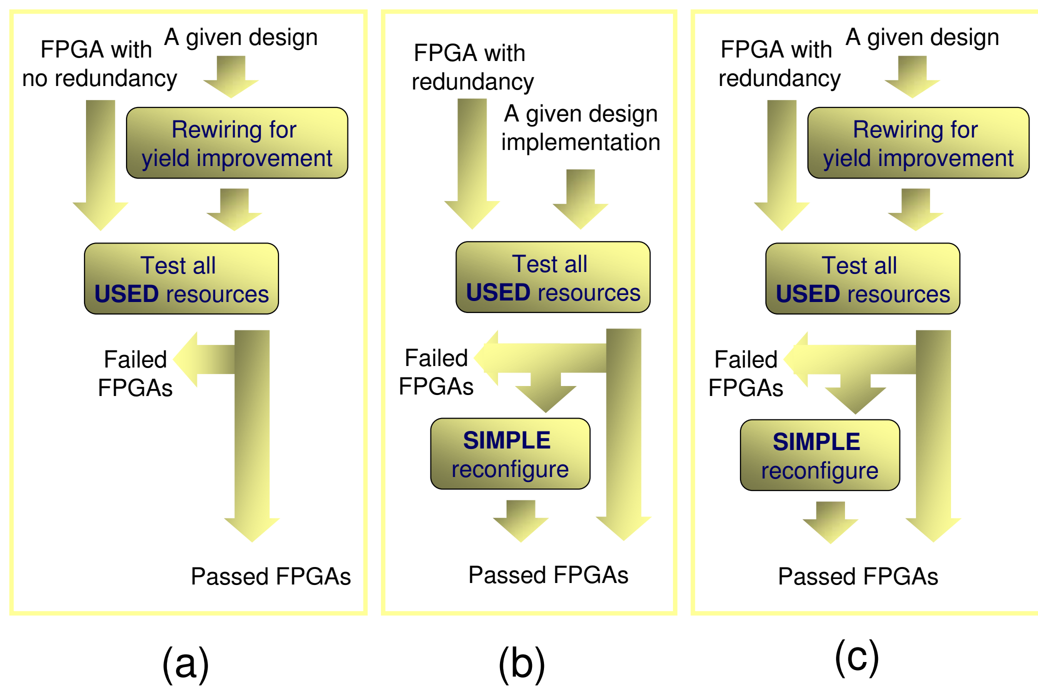

Figure 5.2: Employing multiple yield improvement schemes to enhance the design-specific approach.

Compared to a design with no rewiring, faults are more likely to be on the unused resources if rewiring for yield improvement is applied, significantly reducing remapping cost. An incremental mapping adjustment employed in a customization approach incurs overhead. For example, one re-mapping may take several minutes for one FPGA with 441 logic blocks [21]. Modifying a design before applying a customization approach, as shown in Figure 5.1(b), reduces the number of FPGAs that need remapping. As a modern FPGA contains up to several thousand logic blocks, *e.g.*, the largest XILINX Spartan-6 has 23,098 logic blocks [70], the improvement would be significant. If both defect-tolerant architectures and rewiring are applied in addition to customization approaches as shown in Figure 5.1(c), even fewer number of FPGAs will need remapping.

Rewiring for yield improvement has a positive impact on the efficiency of design-specific approaches in a similar way that it does on a customization approach. Similarity between circuits is defined as the ratio between the number of resources shared among different designs and the number of resources used in the largest design. It was shown that the resulting yield of the design-specific approach increases with similarity of circuits to be mapped [24]. Rewiring for yield improvement will replace low yield connections with higher yield ones. Thus, the similarity between designs increases as their used resources converge on to resources with higher yield, especially with the powerful rewiring technique presented in this thesis. Furthermore, as the commonly used resources have higher yield compared to those before rewiring, using rewiring in conjunction with a design-specific approach as shown in Figure 5.2(a) increases the overall yield.

The design-specific approach can be used on defect-tolerant architectures, as shown in Figure 5.2(b). Every resource of an FPGA will be tested in the conventional defect-tolerant architecture approach, but only the resources used by a given

design will be tested in design-specific approach. Thus, to reduce testing cost, the design-specific approach can be applied before applying the defect-tolerant architecture. A fabricated FPGA will be tested using the design-specific scheme. If some used resources fail, the FPGA will not be declared as faulty immediately, but spare resources will be reconfigured to be used if possible. Only if reconfiguration cannot mask all the faults, the FPGA will be tagged as faulty for implementing the design. Thus, it is easily to see that the number of FPGAs passing the test increases compared to the case that there is no redundancy in FPGAs. As a result, the number of failed FPGAs that need to be stored and retested for other designs decreases, resulting in testing cost reductions. If both defect-tolerant architectures and rewiring are used as in Figure 5.2(c), the benefits are even higher.

Different FPGA family compositions affect how FPGAs are chosen for designs. An FPGA family contains several FPGA members sharing a common architecture but deploying different numbers of resources of each type. In the discussion above, we implicitly assumed that an FPGA member is selected for a given design. Generally, because every FPGA member shares a common architecture, the main criterion of selecting an appropriate member for a family is the number of its resources. For example, ALTERA Stratix IV EP4S40G2 should be used for a design using 1,250 multipliers and 20 144k memory blocks, while EPS100G5 is more appropriate for a design with 1,000 multipliers and 60 144k memory blocks [71].

Usually, the selected FPGA has more resources than required by a design, degrading FPGA yield. The yield of implementing the design on the selected FPGAs is lower than in the best case that there are no extra resources because such an FPGA would be smaller and has higher yield, inferred from Eq.(1.1). More unused resources tranlating to less utilization seems to benefit customization and

design-specific approaches. However, it was shown that yield improvement even with optimal redundancy diminishes as the area overhead increases [41]. Thus, in any cases, composing an FPGA family to minimize the total area of unused resource will result in higher overall yield.

Significant progress has been made on FPGA yield improvements in this thesis, but many challenges still remain. In this thesis, we quantitatively showed that the future yield will significantly drop and argued that yield improvement techniques are important to help maintain a profitable yield. Although the improvement by defect-tolerant architecture was quantified, those of the others considered in this thesis were not because (1) the techniques cannot be modeled except for the simplest one which is the least effective in general, and (2) their efficiencies depend on the detail of the circuits. For example, effectiveness of a technique in customization approach has to be shown empirically [72]. However, their quantitative measures are imperative to determine the effectiveness of a combination of techniques for a given situation. Therefore, not only new approaches for yield improvement are needed to be further devised, but also their quantitative benefits should be further pursued.

Bibliography

[1] N. Campregher, P. Y. K. Cheung, G. A. Constantinides, and M. Vasilko, "Analysis of yield loss due to random photolithographic defects in the interconnect structure of FPGAs," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 2005, pp. 138–148.

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

[3] F. Hatori, T. Sakurai, K. Nogami, K. Sawada, M. Takahashi, M. Ichida, M. Uchida, I. Yoshii, Y. Kawahara, T. Hibi, Y. Saeki, H. Muroga, A. Tanaka, and K. Kanzaki, "Introducing redundancy in field programmable gate arrays," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, San Diego, CA, 1993, pp. 1–7.

[4] D. E. Jefferson and S. T. Reddy, "Redundancy circuitry for programmable logic devices with interleaved input circuits," U.S. Patent 6 107 820, Aug. 22, 2000.

[5] R. L. Geiger, P. E. Allen, and N. R. Strader, *VLSI design techniques for analog and digital circuits.* McGraw-hill, 1990.

[6] B. Razavi, *Design of Analog CMOS integrated circuits.* McGraw-hill, 2001.

[7] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective.* Prentice Hall, 1995.

[8] C. Saint and J. Saint, *IC Mask Design: Essential Layout Techniques.* McGraw-hill, 2002.

[9] S. A. Campbell, *The Science and Engineering of Microelectronic Fabrication.* Oxford University Press, 2001.

[10] R. Wilson, "Designers report on multi-million gate asics," June 2002. [Online]. Available: `http://www.eetimes.com`

[11] D. McGrath, "FPGA market to pass $2.7 billion by '10, In-stat says," May 2006. [Online]. Available: `http://www.eetimes.com`

[12] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Febuary 2007.

[13] ALTERA, "Cyclone III FPGA: Embedded memory," 2009. [Online]. Available: `http://www.altera.com`

[14] XILINX, "Using embedded multipliers in Spartan-3 FPGAs," 2009. [Online]. Available: `http://www.xilinx.com`

[15] ALTERA, "Benefits of embedded RAM in FLEX 10K devices," 2009. [Online]. Available: `ftp://ftp.altera.com`

[16] S. Director, W. Maly, and A. Strojwas, *VLSI Design for Manufacturing: Yield Enhancement*. Kluwer Academic Publishers, 1990.

[17] The international technology roadmap for semiconductors (ITRS), "Yield enhancement," 2007. [Online]. Available: `http://www.itrs.net`

[18] D. McGrath, "Gear costs to derail moore's law in 2014," June 2009. [Online]. Available: `http://www.eetimes.com`

[19] B. Kumthekar and F. Somenzi, "Power and delay reduction via simultaneous logic and placement optimization in FPGAs," in *Proceedings of the International Conference on Design And Test in Europe*, 1998, pp. 202–207.

[20] XILINX, "Virtex-6 FPGA family," 2009. [Online]. Available: `http://www.xilinx.com`

[21] S. Dutt, V. Shanmugavel, and S. Trimberger, "Efficient incremental rerouting for fault reconfiguration in field programmable gate arrays," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, 1999, pp. 173–176.

[22] V. Lakamraju and R. Tessier, "Tolerating operational faults in cluster-based fpgas," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2000, pp. 187–194.

[23] S. Trimberger, "Methods for using defective programmable logic devices by customizing designs based on recorded defects," U.S. Patent 7 047 465, May 16, 2006.

[24] N. Campregher, P. Y. K. Cheung, G. A. Constantinides, and M. Vasilko, "Yield enhancements of design-specific fpgas," in *Proceedings of the International Symposium on Field-programmable Gate Arrays*, 2006, pp. 93–100.

[25] Z.-M. Ling, J. Cho, R. W. Wells, C. S. Johnson, and S. G. Davis, "Method of using partially defective programmable logic devices," U.S. Patent 6 664 808, Dec. 16, 2003.

[26] Xilinx, "Easypath solutions," 2006. [Online]. Available: `http://www.xilinx.com`

[27] M. Chan, P. Leventis, D. Lewis, K. Zaveri, H. M. Yi, and C. lane, "Redundancy structures and methods in a programmable logic deivice," U.S. Patent 7 180 324, Feb. 20, 2007.

[28] A. Doumar and H. Ito, "Design of switching blocks tolerating defects/faults in FPGA interconnection resources," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Yamanashi, 2000, pp. 134–142.

[29] F. Hanchek and S. Dutt, "Methodologies for tolerating cell and interconnect faults in FPGAs," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 15–33, 1998.

[30] N. Howard, A. Tyrrell, and N. Allinson, "The yield enhancement of field-programmable gate arrays," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, no. 1, pp. 115–123, 1994.

[31] V. Kumar, A. Dahbura, F. Fischer, and P. Juola, "An approach for the yield enhancement of programmable gate arrays," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, Santa Clara, CA, 1989, pp. 226–229.

[32] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 6, no. 2, pp. 212–221, 1998.

[33] ——, "Algorithms for efficient runtime fault recovery on diverse FPGA architectures," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems*, Albuquerque, NM, 1999, pp. 386–394.

[34] C. Lane, K. Zaveri, H. Yi, G. Powell, paul Leventis, D. Jefferson, D. Lewis, T. Nguyen, V. Santurkar, M. Chan, A. Lee, B. Johnson, and D. Cashman, "Programmable logic device with redundant circuitry," U.S. Patent 6 965 249, Nov. 15, 2005.

[35] A. Yu and G. Lemieux, "Defect-tolerant FPGA switch block and connection block with fine-grain redundancy for yield enhancement," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2005, pp. 255–262.

[36] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in fpga interconnect," in *Proceedings of the IEEE International Conference on Field Programmable Technology*, 2004, pp. 41–48.

[37] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 12, no. 3, pp. 288–298, 2004.

[38] P. Maidee and K. Bazargan, "Defect-tolerant FPGA architecture exploration," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2006, pp. 467–472.

[39] Altera, "Apex redundancy," 2004. [Online]. Available: `http://www.altera.com`

[40] G. Lemieux and D. Lewis, "using sparse crossbars within LUT clusters," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 2001, pp. 59–68.

[41] I. Koren and Z. Koren, "Defect tolerance in VLSI circuits: techniques and yield analysis," *Proceedings of the IEEE*, vol. 86, no. 9, pp. 1819–1838, 1998.

[42] I. Koren, Z. Koren, and D. Pradhan, "Designing interconnection buses in VLSI and WSI for maximum yield and minimum delay," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 3, pp. 859–866, 1988.

[43] S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng, "Perturb and simplify: Multilevel boolean network optimizer," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 12, pp. 1494–1503, 1996.

[44] S.-C. Chang, L. P. van Ginneken, and M. Marek-Sadowska, "Circuit optimization by rewiring," *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 962–969, 1999.

[45] S. Yamashita, H. Sawada, and A. Nagoya, "A new method to express functional permissibilities for LUT based FPGAs and its applications," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1996, pp. 254–261.

[46] J. Cong, Y. Lin, and W. Long, "SPFD-based global rewiring," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2002, pp. 77–84.

[47] S. Sinha, "SPFDs: A new approach to flexibility in logic synthesis," Ph.D. Dissertation, University of California, Berkeley, CA, USA, 2002.

[48] C.-A. Wu, T.-H. Lin, S.-L. Huang, and C.-Y. Huang, "SAT-controlled Redundancy Addition and Removal - A Novel Circuit Restructuring Technique," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2009, pp. 191–196.

[49] T. Kouda, S. Yamashita, and Y. Kambayashi, "Reduction of the number of FPGA blocks by maximizing flexibility of internal functions," *IEICE Transactions on Fundamentals*, vol. E81-A, no. 12, pp. 2554–2562, 1998.

[50] R. K. Brayton, "Understanding SPFDs: A new method for specifying flexibility," in *Proceedings of the International Workshop on Logic Synthesis*, 1997.

[51] K. Tanaka, S. Yamashita, and Y. Kambayashi, "SPFD-based flexible transformation of LUT-based FPGA circuits," *IEICE Transactions on Fundamentals*, vol. E88-A, no. 4, pp. 1038–1046, 2005.

[52] P. Maidee and K. Bazargan, "A Fast SPFD-based Rewiring Technique," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2010.

[53] A. Mishchenko, J. Zhang, S. Subarnarekha, J. R. burch, R. Brayton, and M. Chrzanowaka-Jeske, "Using simulation and satisfiability to computeflexibilities in boolean networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 5, pp. 743–755, May 2006.

[54] S. Yamashita, H. Sawada, and A. Nagoya, "SPFD: A new method to express functional flexibility," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 8, pp. 840–849, 2000.

[55] P. Maidee and K. Bazargan, "A generalized and unified SPFD-based rewiring technique," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2007, pp. 305–310.

[56] P. Jamieson and J. Rose, "Enhancing the area-efficiency of FPGAs with hard circuits using shadow clusters," in *Proceedings of the IEEE International Conference on Field Programmable Technology*, 2006, pp. 1–8.

[57] K. Rajagopalan and P. Sutton, "A flexible multiplication unit for an FPGA logic block," in *Proceedings of the International Symposium on Circuits and Systems*, 2001, pp. 546–549.

[58] J. R. Tony Ngai and S. Wilton, "An SRAM-programmable field-configurable memory," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1995, pp. 499–502.

[59] P. Jamieson and J. Rose, "Mapping multiplexers onto hard multipliers in FPGAs," in *Proceedings of the International IEEE-NEWCAS Conference*, 2005, pp. 323–326.

[60] S. J. Wilton, "Heterogeneous technology mapping for FPGAs with dual-port embedded memory arrays," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 2000, pp. 67–74.

[61] P. Maidee and K. Bazargan, "Fpga family composition and effects of specialized blocks," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2008, pp. 101–106.

[62] K. Eguro and S. Hauck, "Resource allocation for coarse-grain FPGA development," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1572–1581, October 2005.

[63] A. Smith, "Heterogeneous reconfigurable architecture design : An optimisation approach," Ph.D Dissertation, Imperial College, University of London, London, UK, 2006.

[64] A. Genz, "Numerical computation of multivariate normal probabilities," *Journal of Computational and Graphical Statistics*, vol. 1, no. 2, pp. 141–149, June 1992.

[65] D. P. Tihansky, "Properties of the bivariate normal cumulative distribution," *Journal of the American Statistical Association*, vol. 67, no. 304, pp. 903–905, December 1972.

[66] V. Garg, V. Chandrasekhar, M. Sashikanth, and V. Kamakoti, "A novel CLB architecture and circuit packing algorithm for logic-area reduction in SRAM-based FPGAs," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2005, pp. 791–794.

[67] A. A. Katkar and J. E. Stine, "Modified booth truncated multipliers," in *Proceedings of the ACM Great Lakes symposium on VLSI*, 2004, pp. 444–447.

[68] J. Cong and S. Xu, "Technology mapping for FPGAs with embedded memory blocks," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 1998, pp. 179–188.

[69] M. Wirthlin, "Constant coefficient multiplication using look-up tables," *Journal of VLSI Signal Processing*, vol. 36, no. 1, pp. 7–15, January 2004.

[70] XILINX, "Spartan-6 family overview," 2009. [Online]. Available: http://www.xilinx.com

[71] ALTERA, "Stratix IV FPGA family overview," 2009. [Online]. Available: http://www.altera.com

[72] J. Narasimhan, K. Nakajima, C. Rim, and A. Dahbura, "Designing interconnection buses in VLSI and WSI for maximum yield and minimum delay," *IEEE Transactions on Computers*, vol. 13, no. 8, pp. 976–986, 1994.