Defect Tolerant Logic Implementation onto Nanocrossbar-based Architectures

BY

YEHUA SU
B.S. Capital Normal University, Beijing, China, 2004
M.S. Chinese Academy of Sciences, Beijing, China, 2007

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2012

Chicago, Illinois

Defense Committee:

Wenjing Rao, Chair and Advisor
Shantanu Dutt
Kaijie Wu
Zhichun Zhu
John Lillis, Computer Science

ACKNOWLEDGMENTS

I have been blessed with numerous wise and caring people who have supported me through my Ph.D. as well as earlier education. I would like to take this opportunity to especially thank my advisor Professor Wenjing Rao for her unyielding support and guidance. Professor Rao is a great person, one of those rare idealistic people that always believe in perfection. She has helped me to develop analytical and problem-solving skills that have been of immense help in overcoming many of the difficulties. This thesis could not be possible without her supervising.

I thank Professor Shantanu Dutt, Professor John Lillis, Professor Kaijie Wu and Professor Zhichun Zhu for providing me very valuable guidance and also serving as committee members for this dissertation. I thank my fellow graduate students for sharing with me their research experiences and also their willingness to spend time with me on my research discussions. They are Yixin Shi, Hongzhong Zheng, Liang Han, Jingye Xu, Yu Liu, Kun Fang, Kun Ma and Suyu Zhang. It would be a long list to mention all the other friends I am indebted to. I gratefully thank all of them.

I wish to thank my entire family who taught me to believe that I have the ability to achieve anything as long as I put my mind to it. Their love has always been the source of my strength and inspiration.

YS

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

SUMMARY

Crossbar-based architectures are promising for the future nanoelectronic systems. Nanocrossbar logic implementation emerges as a new fundamental issue, because massive defects, resulting from self-assembly fabrication process, introduce irregular topological constraints on the otherwise regular nanocrossbars. Therefore, defect tolerance techniques become crucially important for the realization of nanocrossbar potentials. As an emerging challenge, defect-tolerant logic implementation onto nanocrossbars turns out to be a hard problem. It is therefore important to model the defect-tolerant logic implementation problem, analyze the cost and tradeoffs, and explore efficient defect tolerance methodologies.

In this dissertation, we first study the defect tolerant logic implementation problem by modeling it in a probabilistic way, so as to analyze the computational complexity and exploring the design quality. Both logic functions and nanocrossbars can be mathematically modeled by matrix model. The mapping based approach is then formulated with a matrix mapping problem, with the goal of finding a mismatch-free mapping (by permutating rows and columns of the matrices) between the logic function and the crossbar matrices. This way, the implementation process is translated into a constraint satisfiability problem with the complexity of NP-completeness. In order to quantitatively understand the defect-tolerant logic implementation and identity the design tradeoffs, beside a probabilistic view from the solution density point of view over the collection of mapping space, design quality is examined furthermore in a more precise way using mismatch number distribution over all the implementation possibilities. The mismatch number distribution reveals the probability that a valid logic implementation

exists and identifies the cost for finding a valid implementation. The number of mismatches turns out to be possible to be well modeled in probabilistic approaches, and the mismatch number distribution follows Normal/Poisson and Hypergeometric distributions, respectively.

With the knowledge of design quality indicated by mismatch number distribution, yield is analyzed and modeled when a large number of crossbars, each having a different defect pattern, are considered. Yield of nanocrossbars, different from traditional manufacturing yield in CMOS, depends on logic implementation algorithms as well as allowed runtime. Due to the high defect rate in nanocrossbars, the implementation process could take prohibitive runtime. Therefore we propose a practical concept of runtime-constrained yield, and identify the tradeoffs between yield and the impacting factors: runtime, defect rate and hardware cost.

In parallel to the analytical work on the modeling of defect tolerance quality, we propose low-cost aggressive approaches to further improve defect tolerance capability, namely logic morphing and fine-grained logic hardening. These novel approaches could be applied on top of the logic mapping technique. Logic morphing exploits the various equivalent forms of a logic function to tolerate defects, while logic hardening adds calculated redundancies to a logic function to make the hardened logic function inherently defect tolerable. Each approach explores an additional dimension of freedom in achieving defect tolerance, and both are orthogonal to and compatible with the existing mapping-based approach. In summary, all three approaches (logic mapping, morphing and hardening) are orthogonal to each other, and can be exploited simultaneously without offsetting each other's performance. We propose an integrated algorithmic framework, which employs mapping, logic morphing and logic hardening simultaneously, and efficiently searches for a successful logic implementation in the combined

solution space. Simulation results show that the proposed schemes boost defect tolerance capability significantly with many-fold yield improvement, while having no extra runtime over the existing approach of performing mapping alone.

CHAPTER 1

INTRODUCTION

This dissertation presents defect tolerance techniques that implement logic functions onto nanocrossbar-based systems and methodologies that model and analyze the defect-tolerant logic implementation process.

The current CMOS technology has scaled according to Moore's Law, allowing circuit designers to continue making advances. Researchers are devoted to continuing feature size scaling and also inventing new nanoscale electronic devices that can potentially replace the conventional photolithographic based CMOS designs. Scaling the device feature size provides faster, denser, and consequently more powerful systems that can run at higher speeds. Unfortunately, now that CMOS has entered the deep submicron range, scaling is becoming more difficult and will eventually cease because of fundamental physical properties of CMOS technology (1). The ITRS Roadmap has challenged the feasibility of CMOS scaling projections beyond MOSFET channel length of 9 nm and has indicated the need of non-CMOS nanoscale technologies.

Recently, a number of novel nanotechnologies have emerged to show the potential in improving the current CMOS platform or replacing the CMOS as an alternative, and also demonstrate promise to develop fundamentally new methodologies to computing systems. These nanotechnologies are proposed in the development of novel memory, configurable logic devices and even hybrid platforms. Nanoscale devices can have high device density, high switching speed and low power consumption. Yet, these improvements in area, performance, and power consumption also come with their own technical chal-

lenges. One of the main challenges, which is the subject of study in this dissertation, is reliability. It is expected that devices become less reliable in smaller feature sizes, and experience both more permanent defects due to the imperfect manufacturing process and more transient faults due to the effect of noise. Reliability issue is becoming constantly more challenging due to increase in both the device failure rate and system complexity. The conventional techniques will not be efficient enough or even capable of tolerating these errors for the future-generation computing systems.

1.1 Nanoelectronics background

Nanoelectronic devices are proposed as an alternative in the next generation electronic systems. Nanoelectronics encompasses an emerging set of technologies that offer the potential of device densities far greater than that of CMOS. These nanoelectronic candidates include silicon nanowires (NW) (2)(3)(4), single electron transistors (SET) (5)(6), quantum-dot cellular automata (QCA) (7)(8)(9), spintronics (SPIN) (10)(11)(12), resonant-tunnel diodes (RTD) (13)(14)(15), carbon nanotubes (CNT) (16)(17)(18). Each of these devices has its own unique characteristics. For example, RTDs offer the potential for multi-valued logic, while QCAs rely on the quantum interaction between electrons and use extremely low levels of power. However, despite the uniqueness of each of these devices, a set of characteristics span across them that help define nanoelectronics. These characteristics include a self-assembly bottom-up fabrication process(19)(20)(21), extremely high levels of defects(22), and inherent reconfigurability.

One common challenge is the fabrication methods that will lead to viable cost-effective nanomanufacturing processes. It is expected that the bottom-up self-assembly approach, which avoids the sophisticated and expensive lithographic process, is the basic way to construct nanoscale devices (3)(23)(24)(25). Self-assembly processes lower manufacturing costs at the expense of reduced control over the assembly

of materials and the placement of devices. Without fine-grained control, the resulting nanoscale devices will exhibit significantly higher defect rates than that in current CMOS technology. The defect rates in these emerging nanodevices are projected to be in the order of 10^{-3} to 10^{-1}, in comparison to that of 10^{-9} to 10^{-7} in CMOS technology (26). Consequently, defect and fault tolerance mechanisms need to be an integral part of nanoscale system designs (22)(27).

Nanoelectronics offer much denser circuitry, and logic implementations onto nanoelectronics generally are based on post-manufacturing configuration, due to the existence of potentially defective devices. Hybrid CMOS/nanodevice circuits (28)(29)(30)(31) were put forward recently for future nanoelectronics systems. Basically, CMOS is built up on top of nanodevices, and the configuration of nanodevices is achieved through reliable CMOS. Generally, the combinational logic is implemented in the underlying nanodevices, and sequential logic and routing are achieved through reliable CMOS. In such hybrid systems, the underlying reconfiguration-based nanodevices are the fundamental building blocks for the whole system.

1.2 Challenges and opportunities in defect tolerance of nanocrossbar systems

While bottom-up manufacturing approach is useful and promising for nanotechnology fabrication, it loses the fine-grained control of each device during the manufacturing process. Since variability and imprecision are inherent in such self-assembly processes, nanowires that are grown using bottom-up techniques may be broken or misaligned. Hence, these nanowires will become unusable. Additionally, for crossbar based architectures, crosspoint switches may contain defects and lose the configurability. The defect rate for Nanotechnology is expected to be much higher than current CMOS technology. This

reliability challenge, caused by massive defects, will fundamentally pose significant challenges on the design phase.

Due to the extreme high defect rate in the nanotechnology, the existing methodologies targeting the defect and fault tolerance in CMOS systems will not work for nanotechnology. Such a high defect rate may require fundamental changes in design paradigms and analytical models. Logic implementation onto nanocrossbar-based architectures is tremendously different from logic implementation on CMOS PLAs. In CMOS PLAs, the configuration phase is trivial, because all the devices at the crosspoints are configurable. However, with massive defects intrinsically residing in nanocrossbars, defect tolerance schemes need to be integrated into the design process. In addition, the defect pattern of each crossbar might be unique, which can be obtained, based on well established testing methodologies for PLA architectures (32; 33), hence the defect-tolerant implementation process has to be performed on every single chip. As a result, the defect-tolerant logic implementation phase becomes the critical bottleneck during the design and manufacturing of crossbar-based nano circuits. Figure 1 provides a comparison for the flows between traditional CMOS based PLAs and nanocrossbars.

With respect to defect tolerance in nanocrossbar-based systems, nanocrossbar architectures have two salient characteristics, which can help achieve the goal of defect tolerance. First, nanocrossbar-based architectures have their reconfigurability. We can use reconfigurability to change the nanocrossbar connectivity so that the logic function can be implemented, even though defects are contained in the nanocrossbars. There are two categories for defect tolerance with employing reconfigurability:

- *Defect-avoiding* means the logic is only implemented on the defect-free sub-circuits by avoiding the defective parts through reconfiguration.

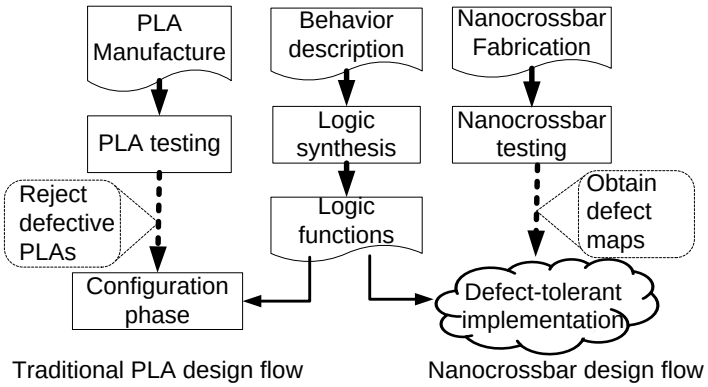

Figure 1. Design flow comparison

- *Defect-using* means the defects are utilized in the implementation as long as the defects do not affect the functionality of the original logic.

The second feature of nanocrossbar-based architectures lies in its regular structure. The regularity of nanocrossbars, in combination of reconfiguration, also significantly contributes to tolerating defects. Due to the regular structure, the implementation of a logic can be flexible, which provides an orthogonal opportunity for tolerating defects. Basically, all rows / columns can be viewed as equal in terms of implementing a logic function, and thus those rows / columns whose defects can be either avoided or utilized are chosen first for implementation.

1.3 Main contributions

In this dissertation, we have proposed and developed new approaches that can tremendously improve the defect tolerance capabilities over the state-of-the-art approaches. Besides the proposed techniques, we also developed multiple methodologies that model, analyze and evaluate the complex defect-tolerant

logic implementation process, and provide the guidelines for developing nanocrossbar-based systems. Some of the specific contributions are as follows:

- A new yield model (runtime-constrained yield), which practically takes into consideration the inevitably long runtime of configuring crossbar according to defects, is proposed. According to the proposed probabilistic model, the upperbound and lowerbound can be mathematically derived, which can reveal the design tradeoffs without having to perform the time-consuming simulations.
- A probabilistic approach is proposed to model and estimate the tradeoff space of runtime for finding a valid implementation. The proposed model reveals how runtime is affected by the impacting factors: defect rate, logic function size, crossbar size and so on.
- A highly effective model based on the number of mismatches in any mapping trial is proposed to quantitatively evaluate the defect-tolerate logic mapping quality.
- A new defect tolerance technique from the perspective of logic equivalence, logic morphing, is proposed, and the algorithmic framework is also developed to efficiently find a valid implementation with logic morphing.
- A new defect tolerance technique from the perspective of crossbar redundant implementation, logic hardening, is proposed, and an optimal fine-grained hardening for logic functions is developed mathematically.
- An integrated algorithmic framework is proposed to simultaneously employ all the proposed defect tolerance approaches, including logic mapping with heuristics, logic morphing and logic hardening.

1.4 Organization

The remaining chapters of the dissertation are organized as following. Chapter 2 describes the existing related research work and the state of arts on defect tolerance for nanocrossbar architectures, and continues to motivations for our work in this dissertation. Chapter 3 presents the modelings for the defect-tolerant logic implementation process, and the backtracking based logic mapping framework with our proposed heuristics. Chapter 4 dives into analyzing yield of nanocrossbars and evaluating runtime for finding a valid implementation. Chapter 5 is dedicated to developing the methodologies for evaluating the logic mapping quality. In Chapter 6, the proposed defect tolerance technique, logic morphing, is presented, including the efficient heuristics proposed specially for Logic Equivalence Checking targeting nanocrossbar logic mapping and the algorithm development integrating logic mapping and morphing. In chapter 7, the proposed defect tolerance technique, logic hardening, is introduced, and the methodology for achieving the optimal hardening for a specific logic function is developed. Following that, the integrated algorithmic framework exploring all proposed defect tolerance approaches is presented. Finally, Chapter 8 summarizes our work and discusses possible directions for future research.

CHAPTER 2

RELATED WORK AND MOTIVATIONS

In this chapter, we first introduce the related work in nanocrossbar architectures, and then discuss the recent publications on defect tolerance techniques targeting nanocrossbar-based systems. After reviewing these related work, the motivations for the new proposed techniques in the dissertation are described.

2.1 Nanocrossbar architectures

Among the proposed nanodevices, nanocrossbar-based architectures have been shown to have significant potential for nanotechnology systems. A large portion of nanoelectronics research has been focused on these nanocrossbar technologies, because they offer the possibility of creating circuits that have characteristics similar to CMOS circuits. This similarity offers two interrelated advantages. First, because we have a large body of knowledge on how to build CMOS-based circuits, devices that offer similar characteristics are better poised to take advantage of existing techniques. Second, as nanoelectronic devices begin to mature, one obvious possibility will be to integrate them into CMOS system to build hybrid circuits. Hybrid systems could leverage the strengths of both CMOS (e.g. reliability and precise fabrication) and nanoelectronics (e.g. cheap, abundant resources). Integrating CMOS with nanoelectronic devices that have familiar characteristics is a less daunting task than integrating devices with radical characteristics such as QCAs.

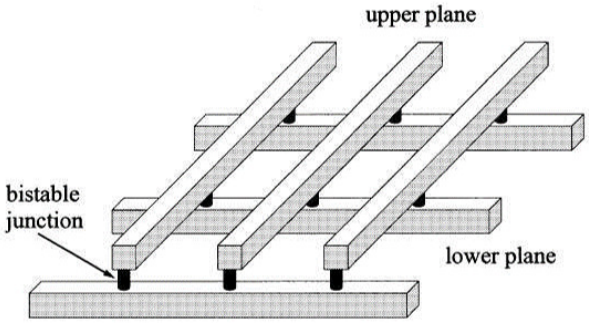

Figure 2. Nano crossbar array

2.1.1 Crossbar array

Because of the use of self-assembly for fabrication, nanoelectronic circuits will likely exhibit a regular structure. One of the most promising structures that has gained popularity is the crossbar array, as is shown in Figure 2. The crossbar array consists of two sets of perpendicular nanowires. At the crosspoint of any two nanowires is a bistable, reconfigurable device. When the configurable is in the“off” state, the wires are disconnected from one another. In the “on” state, the wires are connected to form either a diode or a transistor, depending on the type of nanowires used. Toggling between the two states can happen at any time by providing a large voltage difference between the two wires. This toggling is essentially what provides these devices configurability. It is important to note that these nanowires have a limited maximum length which determines the maximum size of the crossbar array.

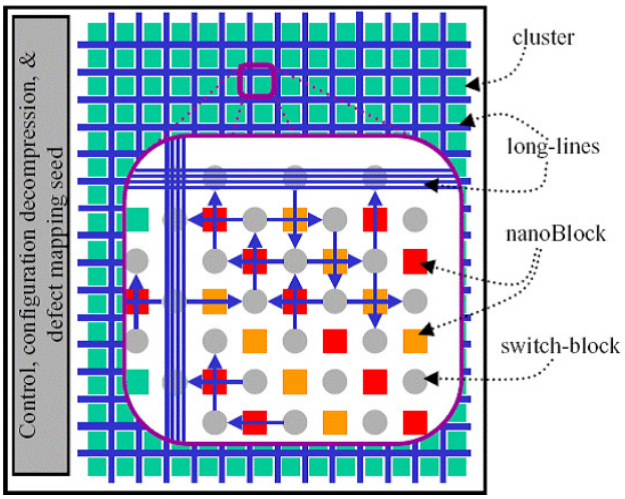

Figure 3. Nanofabric organization
(34)

2.1.2 Nanofabric

In this subsection, we review nanofabric which builds a crossbar-based nanoelectronics fabric. NanoFabrics is introduced in (34), and the organization of the NanoFabric, shown in Figure 3, is similar to that of an FPGA. A large number of reconfigurable blocks provide computation and can be connected together via reconfigurable interconnect. Unlike most FPGAs whose LUTs can implement any function, the capabilities of the nanoBlocks are severely limited. At the heart of each reconfigurable block is a molecular logic array which consists of a crossbar array with diodes at the crosspoints.

The reconfigurable blocks and switch blocks are arranged into a cluster in a checkerboard pattern. The switch blocks are simple crossbar arrays that could be configured to provide two non-overlapping routes between reconfigurable blocks. The placement and limited size of these switch blocks mean

that the connectivity between reconfigurable blocks is localized and limited. The organization of the NanoFabric mean that at least 50% of the devices available are used solely for interconnect. Because each reconfigurable block relies on diodes for operation, signal inversion and gain are not available. In order to provide these critical functions, negative differential resistors (NDRs) can be added to the output of each nanowire. Along with providing gain and inversion, these NDRs also allow for latching of data. Clearly, the shortcomings of diodes make them unattractive for use in a computational fabric. All of the logic operation of NanoFabrics lies in the nanoelectronic devices. A CMOS interface is provided within nanoBlocks to allow for clocking and to power the NDRs.

2.1.3 FET-based array architecture

A PLA-like approach to building a computational fabric has been proposed in (35). This FET-based architecture utilizes recent advances which allow two-terminal configurable diodes to be created at the intersection of nanowires. NOR gates are universal, and any boolean function can be computed using only NOR gates. Unlike the traditional PLA layout of an AND array followed by an OR array, the author uses either a NOR array followed by an OR array or two consecutive NOR arrays. The output wires of the NOR array can be placed orthogonal to the input wires of the proceeding NOR (OR) array. Figure 4 shows the general organization of the FET-based array with NOR-OR planes.

While the NOR and OR arrays are implemented with nanowires, the microscale wires are also used to address these wires as well as to read the result of their computation. Using the microscale wires to directly drive the nanowires is severely limited, because this will limit the density of the nanowires to that of the microscale wires. The proposed solution is to create a specialized decoder between the microscale wires and the nanoscale wires. The nature of the microscale to nanoscale interface leads to a

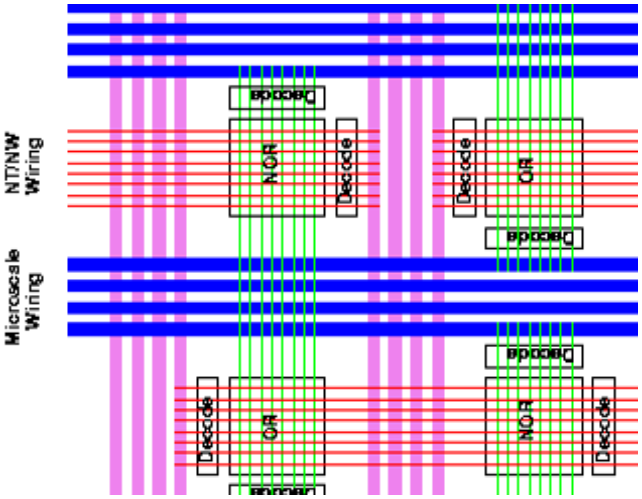

Figure 4. FET array
(35)

design tradeoff when it comes to designing a FET-based array architecture. In order to take advantage of the device density offered by nanoelectronics, large NOR/OR arrays are desired. However, as the size of the NOR/OR arrays increases, the utilization of the devices in the array decreases. This paradox makes achieving good overall device density in the FET-based array approach very difficult if not impossible.

2.1.4 CMOL FPGA

Achieving good device density in array-based approaches is a difficult task, due to the problem of providing microscale wires for addressing and long range interconnect. In the work on CMOL (29), the authors take a fundamentally different approach to interfacing between the nanoscale and microscale. The CMOL approach follows the concept of nano-on-CMOS hybrid systems. The idea of nano-on-CMOS is that rather than integrating micro and nano on the same level, the nanoelectronic part of the circuit is on a layer on top of the microscale.

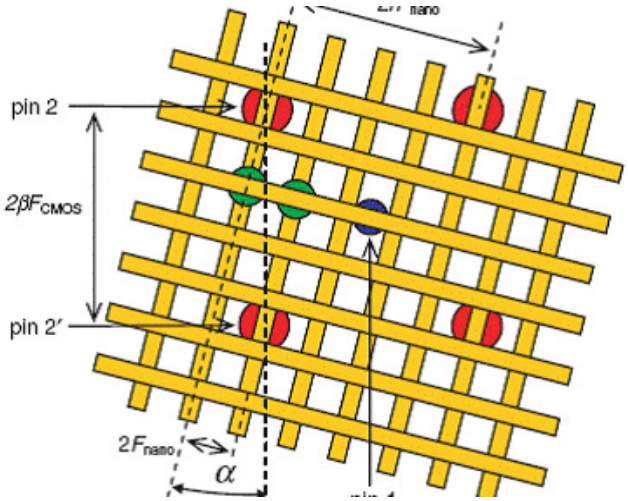

Figure 5. Interconnect between nano and CMOS in CMOL
(29)

The key feature of CMOL lies in how the nano and CMOS layer are interconnected. Figure 5 shows the nature of the nano/CMOS interconnect. In CMOL, there are two different types of vertical pins that connect between the nano and CMOS layers. As the crossbar array has two perpendicular sets of wires, one type of pin will always connect to a set of wires in one orientation while the other type of pin will connect to the perpendicular set of wires. The goal is to have each nanowire connect to exactly one pin.

In CMOL FPGA, the nano layer acts as a large OR array that can use diodes for operation rather than transistors. A NOR gate can be created by implementing an OR gate in the crossbar and having the input go into the CMOS layer for inversion. We can create a sea of these NOR gates and map functionality into them since the NOR gates are universal.

2.2 Related work on defect tolerance techniques

It has been shown that self-assembly techniques can overcome the limitations posed by lithography for the smallest feature size, and crossbars can be easily built due to fabrication regularity imposed by the self assembly process. One of the salient characteristics of nanoelectronic devices is their high levels of defects. Hewlett-Packard has recently fabricated 8×8 crossbar switches using molecular switches at the crosspoints (36). They observed that only 85% of the switches were programmable while the other 15% were defective. With such a high defect rate, defect tolerance methods have to be devised for the emerging nanotechnology devices. Various research groups are working on problems related to defect tolerance in nanocrossbar-based systems. As we stated in the previous chapter, broadly speaking there are two categories for defect tolerance targeting nanocrossbars: defect-avoiding and defect-using.

2.2.1 Defect-avoiding schemes

In defect-avoiding schemes, a (hopefully maximum area of) defect-free subset of the system is searched for and used. In (37), an appropriate system architecture consists of a compiler to arrange for desired circuit behaviors by only using correctly functioning components of a given crossbar circuit, as determined from a testing phase after manufacture. This approach of avoiding known defects gives a defect-tolerant system architecture. In (35), the author proposes the FET-based reconfigurable architectures, and circuits are built by avoiding the faulty wires and switches. In (38)(39)(40), an application independent scheme is proposed to search for a defect-free crossbar subset. Unfortunately, the chance of finding a large defect-free crossbar subset is low, given the high rate of defects. Furthermore, the complexity of finding a given sized perfect subcrossbar is NP-complete, and finding the maximum-sized one is NP-hard. This implies an inevitably high runtime cost. In (41)(42), the author proposes a Build-in

Self Mapping (BISM) algorithm, which tests the crossbar and map the logic in an interleaving way by avoiding the defects detected through BIST. The most salient feature of such a defect-avoiding design is the application-independent design flow. Overall, the defect-avoiding schemes become severely unviable due to the extremely high defect rate in nanotechnology.

2.2.2 Defect-using schemes

The other category is "defect-using", which basically utilizes the defects in the logic implementation as long as the defects do not affect the functionality of the circuit. In order to utilize the defects, these defects first have to be modeled and then can be considered in the logic implementation process. The crossbar defect model also has experienced a process going from simple to complex. Works in (43)(44)(45) solely consider the stuck-open defects, and propose the defect-tolerant algorithms to utilize the defects. Researchers in (46)(47)(48)(49) model both stuck-at-open and stuck-at-closed defects, and develop the approaches to improve yield. Besides crosspoint defects, works in (40)(50) also explore open and bridging line defects. In (50), the idea of using smaller possible crossbars in a larger crossbar is presented for defect tolerance.

Heuristics for expediting the logic implementation are proposed in (43)(51) to reduce the search runtime, where a nanocrossbar is modeled as a bipartite graph. Similarly, authors in (52)(53) also identify the stuck defects in the QCA based PLAs, and propose heuristic algorithms to tolerate these defect in realizing QCA-based circuits. In (54), the authors recommend using Built-In-Self-Test (BIST) to tolerate defects. During the mapping process, nano blocks in a system can be searched as long as any of them can be used for mapping a logic function.

In (55)(56), defect-tolerant logic implementation is translated into a SAT formulation. Work in (44) presents a yield model for logic mapping and identifies the threshold behavior in yield curves. Work in (47)(49) reveals the cost of finding a valid mapping as well as runtime cost involved in the mapping process. In (57)(58)(59)(60), logic mapping is performed under the constraints where defective switches are modeled with a delay cost. Work in (61) transforms the problem into a Bipartite SubGraph Isomorphism (BSGI) problem and also develops the mapping heuristic based on two dimensional sorting. Research work in (62) proposes to further tolerate defects by exploiting logical equivalences. In (63), the authors propose diversity mapping scheme by adding random operators into the greedy algorithm so as to improve the mapping success rate.

2.3 Other defect- and fault- tolerance schemes for nanotechnology

Unlike CMOS devices, nanotechnology is more susceptible to transient faults as well as permanent faults introduced by manufacturing, and has orders of magnitude higher defect rates (64)(44)(65). Hence, it is a significant design challenge to tolerate the high defect rates as well as fault rates, enhance on-line detection capability and to utilize the defective or faulty nanowire crossbar arrays for logic functions.

Various fault tolerance techniques have been proposed to compensate for the high defect rates of nanowire circuits. Fault masking methods can be applied to nanowire crossbars. A dynamically adaptive N modular redundancy (NMR) approach is proposed with reconfiguration algorithms in (65). This approach is shown to mitigate both permanent defects as well as online faults using flexible NMR and reconfiguration. Work in (66) proposes a fault tolerance technique using boolean logic tautology that focuses on the class of faults caused by missing devices at nanocrossbar crosspoints. Multi-level logic

approaches can also be applied to enhance their reliability, however, unlike in (65), the majority voter is not required. This approach can achieve a certain level of fault tolerance while significantly reducing hardware redundancy. Selective hardening of NanoPLAs is introduced in (67). In this method, high-probability faults are identified using an analytical approach as well as simulations. Following that, a selective hardening process is carried, with the goal of both improving the robustness of NanoPLA and reducing the cost for extra hardware added for fault tolerance.

For reliable operation of nanocrossbar architectures, a concurrent multiple error detection scheme using multistage nanocrossbar architectures is presented in (64)(68). The proposed dual-rail logic implementation can detect all single (transient and permanent) faults as well as most multiple (transient and permanent) faults. The effectiveness of hardware duplication, triple modular redundancy (TMR) and parity checking methods are also investigated as part of this work. Authors in (69) investigate the technique of von Neumann's NAND multiplexing, based on a massive duplication of imperfect devices and randomized imperfect interconnect, and reconfigurable architectures for highly unreliable nanoscale devices. It shows that the integration of reconfigurable architectures and NAND multiplexing can tolerate a fault rate of up to 10^{-2}. The authors in (70)(71) provide the probabilistic-based design methodologies for nanoscale logic circuits from the perspective of Markov Random theory.

2.4 Motivations

Whether a logic function can be successfully implemented onto a defective crossbar essentially depends on three factors: 1) logic function structure; 2) crossbar defect map, which basically depicts the structure of defect locations; and 3) logic mapping between the logic function and the defect crossbar, which specifies the correspondence between columns / rows in a crossbar and variables / products in a

logic function. All the existing research dedicated to defect tolerance for nanocrossbar-based systems has been mainly focused on logic mapping. What eventually logic mapping does is to tune the correspondence between columns / rows and variables / products so that the structure of the logic and the structure of the crossbar can "match". When the defect rate is high in nanoelectronics, such a structure match becomes very rare, resulting in increasingly long runtime in finding a valid logic implementation.

Logic mapping works by exploiting mapping flexibilities so as to not let any of the defects on the crossbar affect the logic functionality. In fact, there are other potential areas of achieving defect tolerance beside mapping flexibility, observing that a valid implementation exists as long as the structures from the two sides (logic function and defective crossbar) match. This essentially opens up two more opportunities to further improve the defect tolerance capabilities:

- Changing logic function structure: when a logic function exists in a certain form that is hard for defect-tolerant implementation, the logic structure can be adjusted so that it is better fit for defect tolerance purposes.
- Hardening the logic implementation: when the structure of defect locations in a crossbar is detrimental for implementing a logic function, duplicated columns / rows may be used to implement a single variable / product, so that the negative impact from crossbar defect structure can be mitigated.

In this dissertation, we propose two new defect tolerance approaches based on the above observations: logic morphing and logic hardening. Logic morphing examines the logic function structure, and changes the logic form to achieve better defect tolerance over the course of logic implementation. Obviously, the existing logic mapping framework can be used to support the implementation of any particular

logic form. This essentially means an amplified success rate in logic mapping when a logic function can exist in a large number of different forms, and this translates into great flexibility in defect-tolerant implementation process.

Logic morphing changes the logic structure in order that the structure of the logic can better fit into the crossbar. Similarly, we also want to find a way to "tune" the defect structure of the crossbar so that it becomes better fit for the logic function and tolerate more defects. Obviously, once the nanocrossbar is manufactured with certain defects distributed in it, there is no way to change the defects physically again. Yet, multiple rows / columns in the crossbar can be used jointly as a single row / column for implementation. The combined rows / columns, acting as a sing one, will exhibit different structure and connectivity, because defects of the same type may compensate for each other and defects of different type may cancel each other. Therefore, the structure of the crossbar can be reorganized by grouping different number of rows / columns or selecting different combinations of rows / columns.

Thus, multiple rows / columns are used for implementing a single variable / product, so that the structure of crossbar connectivity is changed, and the defects can be tolerated. Essentially, this is a hardening approach that uses redundant resources to make the implementation more robust. Since nanotechnology generally provides very high device density, such a hardening scheme is promising. From the perspective of a logic function, the logic variable / product is duplicated with multiple copies, each of which will be implemented using one column / row. Therefore, logic hardening provides another layer of defect tolerance capability, which is orthogonal to the existing defect tolerance approaches.

CHAPTER 3

DEFECT-TOLERANT LOGIC IMPLEMENTATION MODELING AND ALGORITHMIC FRAMEWORK

3.1 Problem formulation

We focus on implementing two-level logic functions in the form of SOP (sum of products) onto defective crossbars. Crossbar-based architectures have configurability as well as regular structure, which provides implementation flexibility through mapping different permutations of variables or products onto crossbar wires. In a defect-free crossbar, a logic function can be implemented arbitrarily because the crossbar is fully configurable. In a crossbar with every switch losing configurability and arbitrarily stuck to open and close, the only way to implement a logic function is through permutation of inputs and outputs. Then the logic implementation essentially becomes a Matching problem. In a partially defective crossbar, both configurability and mapping flexibility can be exploited to tolerate defects. Implementing a logic function onto a defective nanocrossbar entails formulating the logic function (relationship between variables and product terms) and the crossbar structure (relationship between two sets of perpendicular wires). We use a matrix model to formulate such relationship presented in the logic function as well as a defective crossbar.

3.1.1 Defect model for nanoscale crossbar based architectures

Even though it is widely acknowledged that defect level will be exceedingly high for nanoelectronic systems, precise defect models will rely on the further maturing in the technology. However,

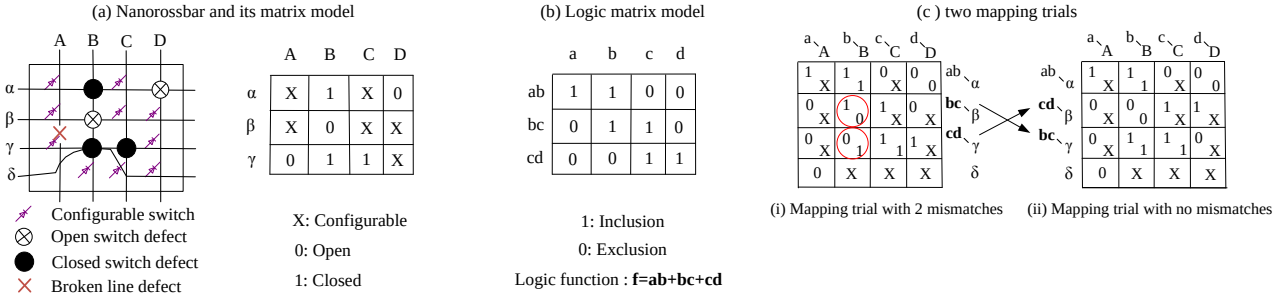

Figure 6. (a) A nanocrossbar with defects and its crossbar matrix, (b) Logic function matrix, (c) mapping trials

from a functional perspective, a particular set of defects are of importance for the crossbar systems: device(switch) defects and line defects. Despite the variety of physical defects that go beyond these cases, a two-fold basic strategy can be applied: 1) catastrophic defects, such as bridging lines, need to be avoided by leaving out the entire wire in the mapping process; 2) non-catastrophic defects, including defective and misplaced switches, can be exploited (thus is cost-efficient) in the mapping process.

Figure 6(a) shows an example, where a part of a broken line can still be used, while one of the two bridging lines has to be discarded as a catastrophic defect. In general, defective switches are not catastrophic, and can be dealt with in a cost-efficient way, such as switches being missing, losing configurability or shorting the two wires. Defects can be modeled functionally and integrated into the implementation process as constraints.

In this dissertation, we focus on the set of the non-catastrophic defects for the mapping approaches. After eliminating the catastrophic defects, a crossbar with such usable defects can be represented by a matrix of cells, each representing one of the 3 connection types:

- *Configurable(X)*: defect free, therefore the connection between the two perpendicular wires can be fully configured.
- *Closed(1)*: the perpendicular wires are permanently stuck closed.
- *Open(0)*: the perpendicular wires are permanently disconnected.

Figure 6(a) shows an example of the *crossbar matrix* with the corresponding cells: configurable (X), closed (1) and open (0).

A manufactured nanocrossbar has its defects distributed among the crossbar plane, forming a certain *defect pattern*. Even though logic implementation is defect pattern dependent, we use two defect formulations to capture the defect level: 1) *defect probability*, and 2) *defect percentage*.

Defect probability assumes that every device in the crossbar has an independent probability of being defective. The advantage of using defect probability lies in its simplicity in modeling, because all the switch devices in crossbars are treated equally. Alternatively, defect percentage is defined as the number of defects divided by the total number of switches to capture the overall defect level of a given crossbar. Defect percentage formulation characterizes defect level more accurately since it takes into consideration the exact umber of defects in crossbars. In this dissertation, *defect rate* is also used interchangeably with defect probability to describe the defect level, when we do not need to differentiate between defect probability and defect percentage in certain studies.

3.1.2 Logic function model

Any two-level logic function in the form of SOP (sum of products) can be modeled by a matrix based on the relationship between the variable set and product set. Connectivity falls into two types:

- *Inclusion*: a variable presenting in a product term, such as $a \in ab$.
- *Exclusion*: a variable not in a product term, such as $c \notin ab$.

Figure 6(b) shows its *logic matrix* for $f = ab + bc + cd$, where "1" indicates inclusion connectivity and "0" represents exclusion connectivity.

3.1.3 Logic implementation formulation

The problem of implementing a logic function onto a defective nanocrossbar translates into a *matrix mapping* problem: correspond the rows / columns of a logic matrix with that of a crossbar matrix, under the constraints imposed by the defects:

- *Configurable(X)* cells in a crossbar matrix are compatible with both *inclusion(1)* and *exclusion(0)* cells in a logic matrix.
- *Inclusion(1)/exclusion(0)* cells cannot be mapped onto *open(0)/closed(1)* cells. We denote these two cases as $1 \rightarrow 0$ and $0 \rightarrow 1$ mismatches.
- *Closed(1)/open(0)* cells, though defective, can be used to implement *inclusion(1)/exclusion(0)* cells in a logic matrix, representing the defect using case.

A valid mapping is one that contains no mismatches. Figure 6(c) shows an example of two matrix mapping trials. The first mapping fails with 2 mismatches, while the second one leads to a successful implementation with no mismatches at all. The objective of logic implementation is to search for a valid mapping (a solution) in the solution space. For a given logic function and a specific crossbar, all the mapping trial possibilities compose the entire *solution space*.

3.1.3.1 Solution space volume

When exploiting mapping flexibility, there are many different ways to map a logic matrix into a crossbar matrix since variables or products can be implemented by different wires. The volume of the solution space is huge, since (i) when the a crossbar matrix is large, the logic matrix can be mapped onto a subset of the crossbar matrix, and the number of such subsets are large, (2) rows / columns in a logic matrix can be mapped onto the same subset of the crossbar matrix with different permutations. Without loss of generality, mapping process can be viewed as permuting each set of rows / columns and then directly mapping the permuted logic matrix onto (a subset of) a crossbar matrix. When a logic matrix is of size $n \times m$ and a crossbar is of size $N \times M$, the volume of the solution space, i.e. the number of possible mappings, is $\frac{N!}{(N-n)!}\frac{M!}{(M-m)!}$. The volume of the solution space therefore grows exponentially to the number of rows / columns in both matrices.

To find one possible solution, the entire solution space might need to be explored. Due to the volume of the solution space, the searching process inevitably takes a long time when there are very few solutions. Heuristics to prune solution-less subspace were proposed in (43) to reduce runtime. However, a matrix mapping problem is NP-complete. There are no efficient algorithms to direct the searching process. Thus, finding a solution for a given logic and a specific defective crossbar is doomed with the challenge of prohibitive runtime and results in low yield.

3.1.3.2 Correlations in solution space

Different mapping trials imply different ways of implementing a logic function onto a crossbar. In fact, in a defect-free crossbar, all the implementations are valid. The presence of massive defects imposes severe constraints and results in the invalidation of most implementations. Essentially, correlation

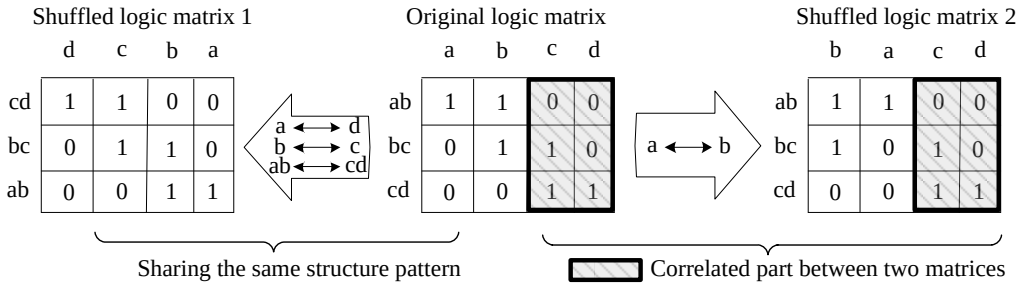

Figure 7. Structure pattern of different shuffled logic matrices for the same logic function

exists among mappings since we have to shuffle rows or columns, not individual cells. For instance, *repetitive mapping trials* occur when different rows / columns shuffles in the logic matrix have identical structure patterns. Figure 7 shows an example that shuffled logic matrix 1 and the original logic matrix have the same structure pattern which will lead to repetitive mapping trials. More generally, *correlated mapping trials* exist widely: rows and columns in an logic matrix are mapped onto the same rows and columns in a crossbar matrix. For example, shuffled logic matrix 2 and the original logic matrix in Figure 7 are correlated since columns c and d remain the same. Among correlated mapping trials, failure of one mapping trial likely denotes the failure of the other ones likewise.

Ideally, to search for a solution, only one instance among repetitive mapping trials needs to be explored. Furthermore, an invalid mapping trial likely indicates the failure of other correlated mappings. Correlation exists widely in the entire solution space. Highly correlated mappings are biased to being all valid or invalid. Within a practical runtime to explore only a limited number of mapping trials in the huge solution space, one would certainly wish to jump out of a set of highly correlated yet invalid ones. In other words, correlation influences the searching process in a significant yet complex way.

3.2 Backtracking-based algorithmic framework

The defect-tolerant logic implementation problem is essentially a constraint satisfiability problem, where the goal is to find a perfect mapping without any mismatches. Backtracking algorithms are typically used in such cases, and the efficiency depends on the use of good heuristics.

We provide the backtracking algorithm framework in Algorithm 1, which will serve as the main backbone for the new proposed mapping, morphing and hardening approaches in the following chapters. Essentially, the backtracking algorithm explores all the possible mappings (correspondence of rows / columns between the logic and crossbar matrices) recursively. Whenever one row (or column) x from a crossbar matrix is mapped to the row (or column) l from a logic matrix, the validity is checked (by $Mismatch\_Check$) to see whether any mismatches are introduced. When a perfect mapping is found, the backtracking mapping algorithm returns success. When no such mapping exists, the algorithm eventually returns failure.

In general, three types of heuristics can be used, as are noted in the backtracking framework.

- *Type 1 heuristics* concern the order of processing rows and columns in the matrices. We found that it typically works the best to map rows and columns in an interleaving way. Such an approach makes it easier to screen out impossible mappings at an early stage.
- *Type 2 heuristics* concern the priority in the row(or column) selection in the crossbar matrix. One way that contributes greatly to reducing search time is to "delay the use of configurable cells (X's)". Basically, when selecting which row (or column) to use in the crossbar matrix, the highest priority should be given to the ones that contain the most defects, yet are still mappable. This way essentially preserves the configurable cells to gain more flexibilities at the later stage.

Algorithm 1 Backtracking Framework for Mapping

Global variables: Logic_Matrix, Xbar_Matrix
BT_Mapping(mapped_set)

1. if all rows and columns of Logic_Matrix are in mapped_set

   return success

2. pick a row (or column), l, from Logic_Matrix, such that l is not in mapped_set yet
   //type 1 heuristics applicable here
3. For every unmapped row (or column), x, of Xbar_Matrix
   //type 2 and 3 heuristics applicable here

   if ***Mismatch_Check***($l \rightarrow x$, mapped_set) == $\emptyset$
   //map l to x, when current constraints are satisfied

   (a) add $l \rightarrow x$ to mapped_set
   (b) if ***BT_Mapping***(mapped_set) == success
   //recursive call for the rest of the mapping

   return success

   else remove $l \rightarrow x$ from mapped_set
   //$l \rightarrow x$ does not yield any solution, try a different x in Xbar_Matrix

4. return failure
   //failed to map l to any possible x, backtrack

Mismatch_Check(new_mapping, mapped_set)
//This subroutine checks whether adding new_mapping to mapping_set introduces mismatches, and returns mismatches.

- *Type 3 heuristics* concern various pruning techniques (43), which are used to screen out invalid mappings at the early stages to help reducing search time.

3.3 Discussions

The crossbar defect modeling has become an open issue in the recent published papers. In the manufacturing process, the defects can happen to any part of a crossbar plane, including nanowires and configurable devices sitting between two perpendicular sets of nanowires. For nanowire defects, there are two types: 1) broken line defects, and 2) bridging line defects. When it comes to configurable device defects, it may fall into one of those categories: stuck-at-open, stuck-at-closed, missing devices, misplaced device and etc. Researchers treat the defects in nanocrossbar differently. Some researchers assume that the stuck-at-open defects are way more frequent than stuck-at-closed defects (46). In a similar way, some other researchers presume that line defects are less likely than configurable device defects (45)(40)(50)(55)(44). Based on their proposed defect modeling, they propose various methodologies to deal with different categories of defects.

In fact, precise defect modeling will rely on the further maturing in the manufacturing technology. In this dissertation, we consider all types of defects, including line defects and various device defects. Yet, for those catastrophic defects, they can not be used at all for the implementation. For the research purposes, we model them from a perspective of its impact on the functionality of logic, and parameterize the relationship among various defect types by, for instance, the defect ratio between closed-to-open defects in the following chapters. In such a way, the methodologies in the dissertation can be used to analyze or predict the results for the emerging nanotechnologies in general.

We focus on two-level logic functions in the research for two reasons. First, it lies in its simplicity. Crossbar architectures are inherently suitable for the implementation of two-level logic functions. A two-level logic function is itself complete, since multi-level logic function can be converted into a two-level form. Second, in the case that crossbars are used to implement multi-level logic, the analysis on two-level logic implementation provides a fundamental basis that can be extended to nanocrossbar-based system designs without loss of generality.

CHAPTER 4

PROBABILISTIC ANALYSIS ON YIELD AND RUNTIME

The logic matrix to crossbar matrix mapping problem by its nature is equivalent to a subgraph isomorphism problem, which is known as a NP-complete problem. The difference between them lies in that the validity of such a mapping problem is determined by the mapping rule, as a configurable cell (representing a defect-free switch) is a "wildcard" to map both types of cells in a logic matrix. Since the complexity of validity checking stays the same, the logic matrix to crossbar matrix mapping problem, as an variation of the subgraph isomorphism problem, has the same computational complexity. For such a NP-complete problem, probabilistic information, such as yield and runtime, is of crucial importance, since it reveals the likelihood and the expected computational cost of finding a solution. Undoubtedly, runtime for the logic mapping problem depends on the specific algorithms. Therefore, in order to analyze runtime, an algorithm which can generally identify the computational cost should be adopted.

As discussed before, the defect-tolerant logic implementation is NP-complete, and there exist no efficient algorithms. On the other hand, this problem has two characteristics: (i) with an extremely large solution space, it is impossible to explore the entire solution space. In reality, only a very small portion of the solution space can be explored; (ii) mappings in the solution space are highly correlated. Within a practically acceptable runtime to explore only a limited number of mapping trials in the huge solution space, one would certainly wish to jump out of a set of highly correlated yet invalid ones and reach a solution quickly. Based on the above intrinsic difficulty of the problem, we adopt randomized algorithms to explore the solution space, according to which every part of the whole solution space has

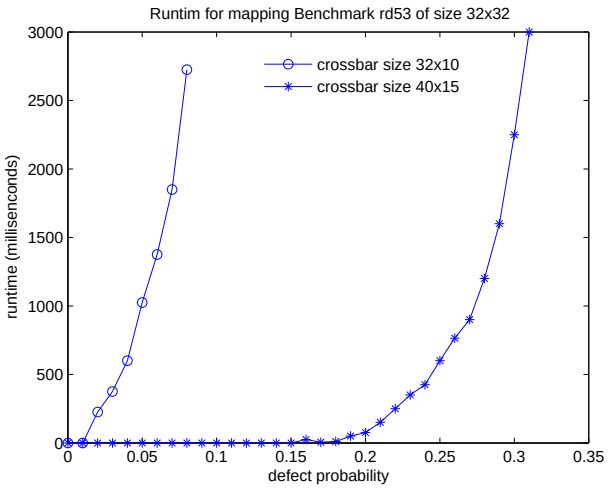

Figure 8. Explosion in runtime

equal opportunity to be sampled, and mappings chosen randomly tend to be less correlated. For these reasons, a randomized algorithm enables us to identify the computational cost in general, and can be used as a reference point for any future heuristic algorithms.

In the following sections, we first analyze the yield when defect-tolerant schemes are adopted. In addition, we examine yield from a practical perspective when runtime is limited. Finally, we investigate the runtime involved in finding a solution.

4.1 Observations from yield curves

Conceptually, yield is the percentage of "good chips", i.e. the percentage of crossbars which can be utilized to implement a given logic. With post manufacturing defect-tolerant reconfiguration, a "good crossbar" is one with valid mappings to implement the given logic function. This apparently has a great influence on yield, because theoretically speaking, as long as there is one mapping trial that is valid, the

crossbar should be considered as a "good" one. Unfortunately, such an ideal yield is both hard to model and in reality meaningless to apply, particularly when the level of defects is high, and finding a valid mapping might take prohibitively long time. Figure 8 shows that runtime cost grows dramatically when defect level increases beyond a certain value.

4.1.1 Phase transition in yield curves

It has been demonstrated that the yield curves of crossbar-based systems have a threshold behavior (44; 47). Yields curves based on both defect probability and defect percentage are shown Figure 9(a), with same-sized logic function and crossbars. A phase transition phenomenon can be observed in both defect probability based and defect percentage based curves. Essentially, the logic mapping problem falls into the category of Constraint Satisfaction Problems (CSP) (55), which have been widely acknowledged of the *phase transition* phenomenon in computational complexity (72). In this case, there apparently exists a threshold in defect probability/percentage, over which it becomes extremely hard to find a valid mapping.

4.1.2 Substantial yield improvements over hardware redundancy

Yield in crossbar-based systems essentially depends on the existence of valid mappings. Therefore, yield can be improved by increasing crossbar size, such that more wires are available for mapping, and the chance for existing a valid mapping is increased. Figure 9(b) shows how yield can be improved with the increase of hardware redundancy, where the logic function benchmarks (73) are mapped onto larger crossbars with the same defect probability of 15%. Sizes of enlarged crossbars are expressed as *size ratio* (crossbar size to logic function size).

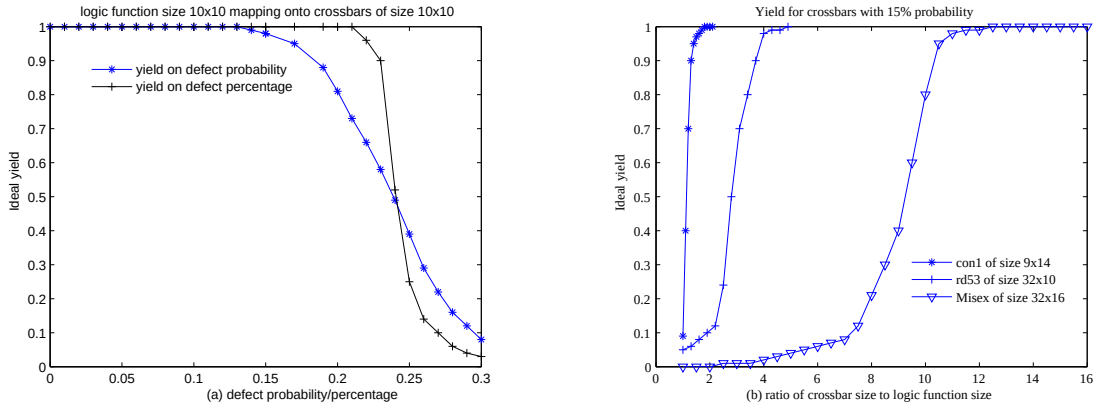

Figure 9. Phase transition in "ideal yield" assuming unlimited runtime of mapping algorithms

In fact, yield can be improved dramatically by keeping increasing crossbar size for all three cases, and a phase transition phenomenon can be observed int the curves as well. This indicates there is a certain "sweet spot" in adding hardware to boost yield significantly.

4.1.3 Problems with the "ideal" yield model for nanocrossbars

So far, the concept of yield takes into consideration the defect-tolerant approach of logic mapping, but is ignorant of mapping runtime. In fact, the logic mapping process is of NP-complete complexity, and each crossbar needs to be treated according to its unique defect pattern, therefore the lengthy logic mapping process has to be exercised on every single crossbar. As a result, the concept of yield will be of very little practical value in nanocrossbar-based systems, unless the cost of mapping runtime is also considered in the yield model. A crossbar with some rare valid mappings which in reality take forever to find, does not contribute to the "yield" part in any practical sense, and thus should not be counted in a mapping-aware yield model.

4.2 Mapping-aware yield

Mapping-aware yield model is a more coherent model for the nanocrossbar-based systems, where the cost of the logic mapping process is taken into consideration. Such a new yield model needs to satisfy two criteria: 1) efficiency: its calculation should be easy to carry out, preferably with some closed forms; and 2) precision: it needs to model the complexity and runtime cost with high fidelity. We propose such a new mapping-aware yield model - named *RunTime-Constrained yield* (denoted as RTC yield thereafter). Basically, the idea of RTC yield is still the percentage of "good" crossbars, yet a crossbar is counted as "good" only when a valid mapping can be found within some pre-set bound of runtime limit (i.e. a given number of mapping trials).

Assume all the possible mapping trials between a logic function and a crossbar constitute an entire *mapping space*. Then a "naive" yield model without mapping consideration essentially checks for the *existence* of any valid mappings in the *entire* mapping space, to count a crossbar as "good". RTC yield, instead of checking for the existence of valid mapping in the mapping space, is concerned with the "density" of valid solutions in the mapping space, and only needs to check a certain number of mappings. In other words, RTC yield is not concerned with the existence of a valid mapping in the entire mapping space, but rather, the percentage of mappings that are valid.

4.2.1 Modeling: solution density

We define *solution density* as the percentage of valid mappings in the mapping space. Typically, finding a valid mapping depends on both solution density and the runtime limit. However, for a given crossbar, its solution density is determined by its particular defect pattern and the target logic function, and is hard to compute. For a large number of crossbars (either with same defect probability or defect

percentage), the solution density of the batch is a distribution depending on defect probability/percentage and target logic function, and becomes possible to model. To study the characteristics of such a solution density distribution, we examine its expectation $E(sd)$ and variance and variation $V(sd)$, and how they jointly affect RTC yield.

4.2.1.1 $E(sd)$ based on defect probability $d_0 + d_1$: $E(sd)_{d_0+d_1}$

Assume each switch has probability d_0 of being defectively open and d_1 of being defectively closed. A valid mapping has no $1 \rightarrow 0$ or $0 \rightarrow 1$ mismatches. This means that any inclusion (1) cell of the logic function must not be mapped to an open defective switch. The probability of any single inclusion cell from logic matrix being mapped without mismatches is thus $1 - d_0$, and any simple exclusion (0) cell from logic matrix being mapped without mismatches is $1 - d_1$. For a logic function with logic inclusion ratio l_1 (defined as the percentage of inclusion cells in the logic matrix), and size $n \times m$ (n variables and m products), each mapping trial then has a probability of P_v to be a valid solution:

$$P_v = (1 - d_0))^{nml_1}(1 - d_1)^{nm(1-l_1)} \tag{4.1}$$

where nml_1 is the number of "1" cells and $nm(1 - l_1)$ is the number of "0" cells in the logic matrix.

Under the defect probability model, since each mapping has the same probability P_v of being valid, the expectation of solution density $E(sd)_{d_0+d_1}$ equals P_v, i.e.

$$E(sd)_{d_0+d_1} = P_v \tag{4.2}$$

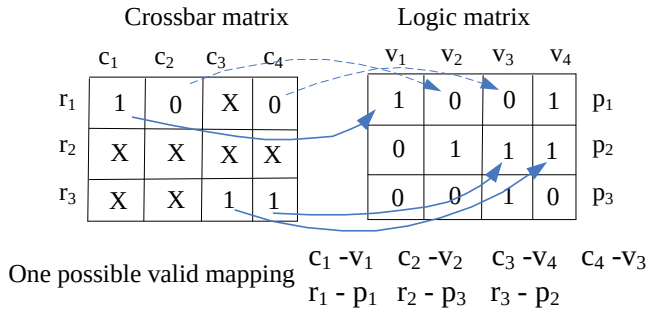

Figure 10. Cell mappings leading to no mismatches between logic and crossbar matrix

These two equations formulate the expected solution density among a large number of crossbars with the same device defect probability. Based on this model, the following observations can be made:

- $E(sd)_{d_0+d_1}$ drops sharply when defect probability increases.
- As logic function size increases, $E(sd)_{d_0+d_1}$ decreases exponentially.
- $E(sd)_{d_0+d_1}$ depends on defect probability and logic function, and is independent of crossbar size.

4.2.1.2 $E(sd)$ based on defect percentage $p_0 + p_1$: $E(sd)_{p_0+p_1}$

We introduce the defect percentage p_0 for open defects, and p_1 for closed defects. Different from $E(sd)_{d_0+d_1}$, $E(sd)_{p_0+p_1}$ varies according to crossbar size. We first consider the case with crossbars having equal size to the logic function. With crossbar size being nm, there are nmp_1 closed defects and nmp_0 open defects in a crossbar. Since the crossbar and the logic function have the same size, every cell in the crossbar matrix has to be mapped, and a valid mapping implies that every "1"("0") cell in the crossbar matrix has to be mapped without mismatches in the logic matrix. This means, for all the "1"

cells in a logic matrix, the same number of cells need to be chosen from a crossbar matrix, which can only come from two groups: “1” and “X”. Eventually, all the “1” cells in the crossbar matrix have to be chosen to map the inclusion cells in logic matrix, because otherwise some would have to be mapped onto “0” cells in logic matrix. In the same manner, all the “0” cells in the crossbar matrix will have to go with “0” cells in a logic matrix for the mapping to be valid.

For the given logic matrix and crossbar matrix shown in Figure 10, there are altogether $\binom{12}{6}$ ways to select from the 12 cells in the crossbar matrix to map to all the inclusion cells in the logic matrix, and thus the entire mapping space has $\binom{12}{6}$ possibilities. Among these possibilities, a valid mapping cannot have any $1 \rightarrow 0$ (or $0 \rightarrow 1$) mismatches, therefore all the “1” cells in the crossbar matrix have to be mapped to the inclusion cells (“1”) in the logic function matrix. The rest of the freedom ultimately depends on the number of choices to use “X” cells in the crossbar matrix for the remaining “1” cells in the logic function matrix. There exists only $\binom{7}{3}$ ways to choose among the 7 “X” cells in the logic matrix for the rest of three “1” cells in the logic matrix to be mapped. After that, the rest of “X” cells are deemed to be used for the “0” cells in the logic function matrix, and it is also implied that all “0” cells in the crossbar matrix are mapped to “0” cells in the logic matrix. This results in a solution density of $\binom{7}{3}/\binom{12}{6}$ = 0.038.

Figure 26 illustrates the mapping between logic matrix cells and crossbar matrix cells in general. Since there are altogether $\binom{nm}{nml_1}$ ways to select nml_1 cells from a crossbar matrix to map all the “1” cells in a logic matrix, the entire mapping space is of the size $\binom{nm}{nml_1}$. The number of valid mappings, however, ultimately depends on the number of choices to get the rest of “1” cells from the “X” group. There exist only $\binom{nmp_1}{nmp_1}\binom{nm-nm(p_0+p_1)}{nml_1-nmp_1}$ possible ways, where $\binom{nm-nm(p_0+p_1)}{nml_1-nmp_1}$ means to choose $nml_1 - nmp_1$ “X”

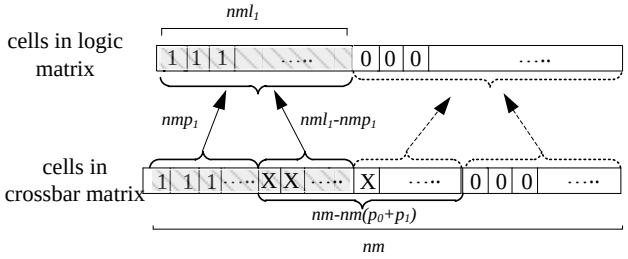

Figure 11. Logic mapping onto crossbars based on defect percentage

cells from $nm - nm(p_0 + p_1)$ cells to map the rest of "1" cells in a logic matrix. Therefore, the solution density expectation based on defect percentage can be formulated as:

$$E(sd)_{p_0+p_1} = \frac{\binom{nmp_1}{nmp_1}\binom{nm-nm(p_0+p_1)}{nml_1-nmp_1}}{\binom{nm}{nml_1}} = \frac{\binom{nm(1-p_0-p_1)}{nml_1-nmp_1}}{\binom{nm}{nml_1}} \tag{4.3}$$

Figure 12 shows a comparison between $E(sd)_{d_0+d_1}$ and $E(sd)_{p_0+p_1}$ with benchmark $xor5$ of size 16×10 and defect probability / percentage $[0 \sim 0.1]$. In general, $E(sd)_{p_0+p_1}$ is always lower than $E(sd)_{d_0+d_1}$, for the reason that will be explained in the following part of the paper.

When a crossbar that is larger than the logic function being considered, different parts of a crossbar may have various local defect percentage. If the crossbar size becomes large enough, defect percentage based model becomes similar to that based on defect probability. Therefore, as long as crossbar size is larger than the logic function size, the actual expectation of solution density of the crossbar sits between $E(sd)_{p_0+p_1}$ and $E(sd)_{d_0+d_1}$. The fact that $E(sd)_{p_0+p_1} < E(sd)_{d_0+d_1}$ essentially explains why increasing crossbar size can always improve solution density expectation.

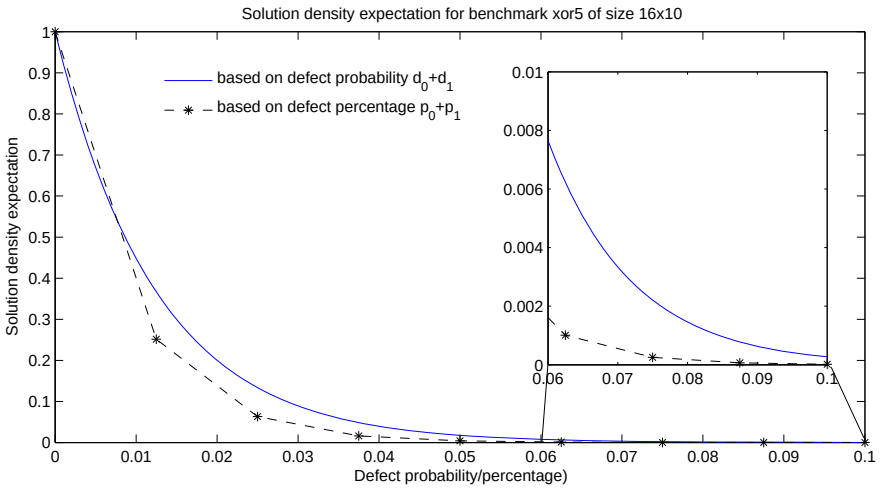

Figure 12. $E(sd)_{p_0+p_1}$ is lower than $E(sd)_{d_0+d_1}$.

4.2.1.3 Variations in solution density $V(sd)$

RTC yield not only depends on solution density expectation, but also on the variation $V(sd)$. Figure 13 shows the solution density distribution of crossbars based on defect probability, defect percentage, and crossbar size as well. A few interesting observations can be made from Figure 13. First, the distribution of solution densities based on defect percentage is more concentrated, as is shown in Figure 13(c) and (d), while the distribution based on defect probability is more widely scattered, as shown in Figure 13(a) and (b). Apparently, crossbars of the same defect probability may end up containing different numbers of defects, thus the variation of solution density among this group is larger. Second, when crossbar size increases ((a)→(b), (c)→(d)), $V(sd)$ decreases. The reason for this is twofold: i) as the size of a crossbar increases, the defect number variation across crossbars decreases; ii) since a logic function can be mapped to different parts of a larger crossbar (with various defect patterns), then an

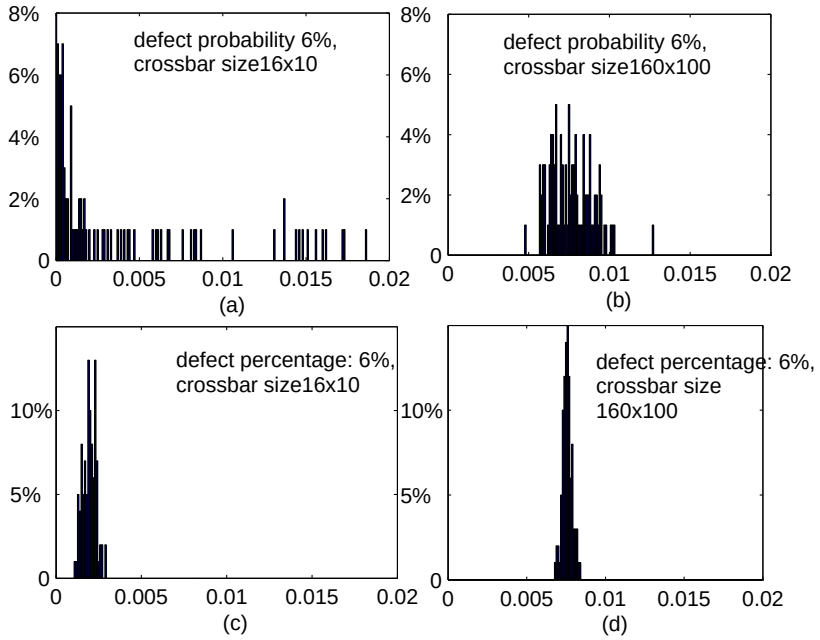

Figure 13. Solution density variation decreases when based on from defect probability to defect percentage ((a)→(c), (b)→(d)), and decreases as crossbar size increases ((a)→(b), (c)→(d))

enlarged crossbars, containing more varieties of local defect patterns, can mitigate the impact imposed by a specific defect pattern constraints more consistently.

4.2.2 Modeling: Runtime Constraint (RTC) yield

RTC yield is defined as the percentage of crossbars, with a valid mapping found within a limited number of trials. Using the concept of RTC yield, only a limited number of mappings in the entire mapping space need to be explored, therefore avoiding the difficulty faced by the traditional yield concept discussed in Section 3.

4.2.2.1 The impact of runtime limit on RTC yield

Figure 14 shows how the limit of mapping trials impacts RTC yield. It is clear that RTC yield is improved as the number of trials increases. With a fixed solution density expectation, RTC can be improved through investing more search runtime. Also, for any given number of mapping trials, RTC yield decreases as defect probability increases. This implies that as defect probability increases, the solution density decreases to the point where a valid mapping cannot be found within the given number of mapping trials.

It is also noticeable that the curves based on defect percentage model are steeper than that of defect probability model. The reason is that crossbars sharing the same number of defects have less variations in solution densities than crossbars with the same overall defect probability. In other words, the crossbars with the same number of defects tend to have similar mapping profile: either a valid mapping can be found or no valid mappings at all can be found within a given runtime. This indicates defect percentage based profile is a good index in predicting the yield of crossbars. Conceptually, RTC yield curves based on defect percentage switch from high to low dramatically as more defects are added into the crossbars.

Finally, there is always an overlap between RTC yield curves based on defect probability and curves based on defect percentage. For a large number of crossbars having defect probability, the average defect percentage is equal to defect probability, i.e. $E(p_0) = d_0$ and $E(p_1) = d_1$. So, the RTC yield curve based on defect probability $d_0 + d_1$ always overlaps RTC yield curve based on defect percentage $p_0 + p_1$.

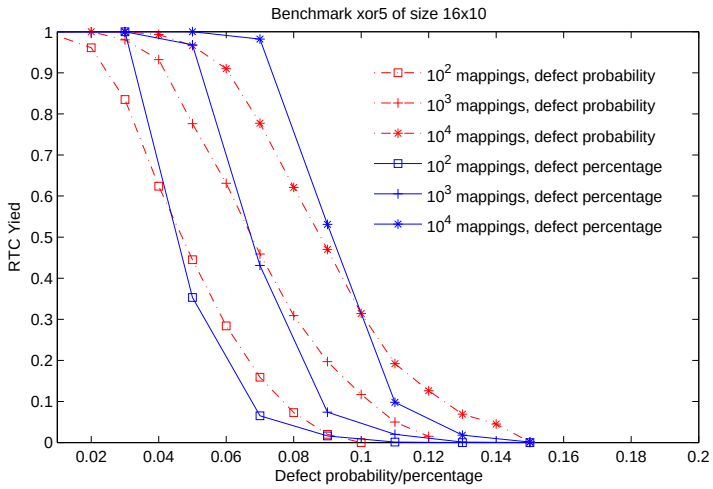

Figure 14. Runtime-constrained yield increases with the number of explored mappings

4.2.2.2 Establishing RTC yield upper / lower bound from the insight of crossbar size impact

Increasing hardware redundancy can significantly improve the conventional yield, since the mapping space grows exponentially and the entire mapping space is explored. For RTC yield, a fixed number of mapping trials are always explored regardless of the enlarged mapping space. Intuitively, more hardware redundancy does not necessarily improve RTC yield. Such a hypothesis is first examined and then followed by theoretical reasoning and analysis.

4.2.2.2.1 Exact upperbound for defect probability / percentage based models

Figure 15 shows RTC yields improving as crossbar size increases, despite the fact that the same number of mappings are explored. Moreover, as crossbar size increases, the difference between RTC yields based on defect probability and defect percentage gradually decreases. Apparently, RTC yields

do not increase unlimitedly with increasing crossbar size, and instead approach a certain "saturation" point even when crossbar size is boosted up to as large as 100 times.

With a fixed number of mapping trials, RTC yield essentially depends on the solution densities of each single mapping space. As shown in previous sections, the actual solution density expectation sits between $E(sd)_{p_0+p_1}$ and $E(sd)_{d_0+d_1}$, and variations do exist. The most improvement for RTC yield is achieved when solution density expectation is maximized to $E(sd)_{d_0+d_1}$ and $V(sd)$ reduced to zero. In such a case, all the mapping spaces are essentially as the same, with RTC yield equivalent to the probability of finding a valid mapping (with the given number of mapping trials). As a result, this gives an upperbound to RTC yield, which can be expressed as

$$RTC\_Y_{upperbound} = 1 - (1 - E(sd)_{d_0+d_1})^t \tag{4.4}$$

where t is the number of mapping trials to be explored.

As is shown in Eq.Equation 4.1, $E(sd)_{d_0+d_1}$ is independent of crossbar size. Therefore, in the probabilistic model, RTC yield improves as larger crossbar size reduces the $V(sd)$ among mapping spaces, as is shown in Figure 13. With $E(sd)$ remaining the same and $V(sd)$ diminishing, RTC yield improves. When it comes to the curve based on defect percentage, $E(sd)_{p_0+p_1}$ increases up to the level of $E(sd)_{d_0+d_1}$ with crossbar size increasing, therefore it is improved mainly because $E(sd)_{p_0+p_1}$ increases. In addition, $V(sd)$ also decreases as crossbar size increases. So, the increase of RTC yield based on defect percentage results from the combination of: 1) solution density expectation increasing, and 2) variation decreasing.

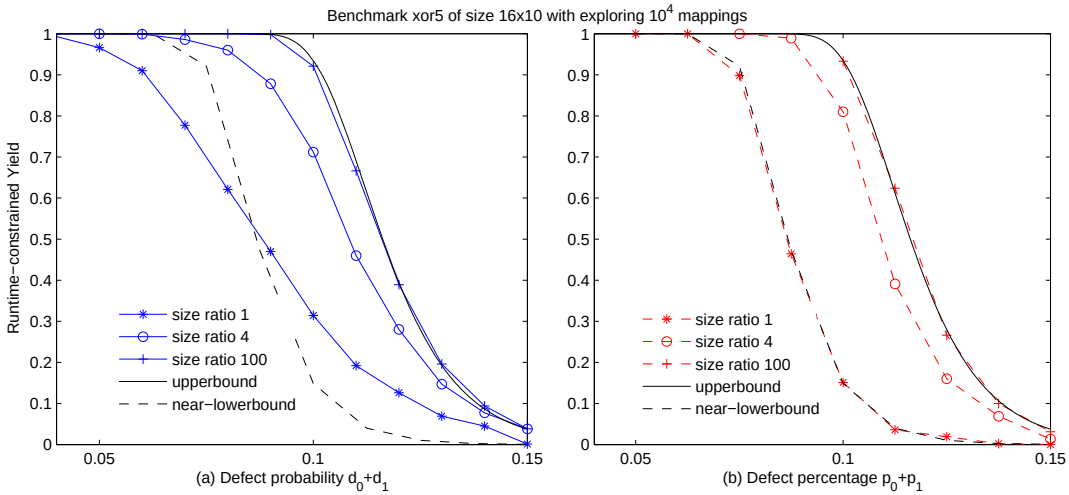

Figure 15. Runtime-constrained yield increases with crossbar size, expressed as size ratio

Lastly, this upperbound in Eq.Equation 4.4 holds for both RTC yield based on defect probability and defect percentage. The theoretical upperbound of RTC yield conforms with the saturation curve observed in Figure 15(a) and (b). With such a theoretical upperbound, RTC yield can be quantitatively estimated with a close mathematical form (Eq.Equation 4.4).

4.2.2.2.2 Exact lowerbound for defect probability based model and approximate lowerbound for defect percentage based model

While the RTC yield Upperbound shows the maximum benefits of yield improvement achievable through increasing crossbar hardware cost, we are also interested in the minimum yield achievement when no hardware redundancy is invested.

As is shown before, $E(sd)_{d_0+d_1}$ is always larger than $E(sd)_{p_0+p_1}$, regardless of crossbar size. In addition, solution density expectation based on defect percentage $E(sd)_{p_0+p_1}$ increases with crossbar

size, and ultimately reaches $E(sd)_{d_0+d_1}$. As a result, solution density expectation based on defect percentage $E(sd)_{p_0+p_1}$ with a minimum sized crossbar (equal to logic function size) can serve as the lowerbound for solution density expectation. Since $V(sd)$ based on defect percentage is negligible (as is shown in Figure 13), RTC yield based on defect percentage with minimum crossbar size can be approximated by:

$$RTC\_Y_{p_0+p_1} = 1 - (1 - E(sd)_{p_0+p_1})^t \tag{4.5}$$

.

With the fixed number of mapping trials, RTC yield based on defect probability overlaps RTC yield based on defect percentage (as is shown in Figure 15). As a result, Eq.Equation 4.5 servers as an approximate-lowerbound for RTC yields, based on both defect probability and defect percentage. As is shown in Figure 15, the lowerbound in Eq.Equation 4.5 nicely captures the minimum RTC yield when defect level formulation is based on defect percentage, and also intersects with the minimum RTC yield curve when defect level formulation is based on defect probability.

4.3 Probabilistic analysis on runtime

Essentially, the challenge of logic mapping comes from two sources: (i) the massive defects make the searching process impractically long, if there ever exists one solution, (ii) the defect pattern is distinctive for every nano fabric; thus such a mapping process has to be performed on a per-chip basis. Consequently, one cannot afford to run any mapping algorithm that is time consuming. With the searching problem shown to be NP-complete in nature (43)(44), it is important to know how likely it is to find a solution, captured by the yield analysis. Yield identifies the probability of finding a solution given the crossbar size and defect probability.

Nonetheless, the searching process could take extremely long runtime to find a solution even if yield is high. Analysis on runtime is necessitated, since it is crucially important to estimate how long it takes approximately to find one solution. Generally, achieving lower defect probability for a nanoelectronic system requires more fine-controlled and costly manufacturing. On the other hand, increasing the allowable defect probability will lead to more constraints on the mapping process and longer runtime. Having the knowledge of runtime before engaging in searching process and the impacting factors on runtime are highly important to nanoelectronic system designs.

As the solution space grows exponentially with increased crossbar size to improve yield, consequentially, runtime could boost up to an unacceptable level in reality. Thus, there is a limitation in trading off crossbar size to improve yield. In general, high solution density leads to reduced runtime regardless of the algorithms used in the searching process. Overall, runtime analysis based on solution density indicates how hard it is to find a solution.

A probabilistic approach, based on defective crossbars of a given defect probability and size, is adopted to evaluate solution density. Each switch is modeled as a random variable taking three possible states: configurable, open and close. Solution density sd for a defective crossbar becomes a random variable depending on the defect probability and size of crossbars. Solution density expectation and standard deviation are used and examined in detail to analyze the tradeoffs among runtime, hardware redundancy and defect probability.

4.3.1 Solution density expectation $E(sd)$

The solution space volume depends on (i) logic function size (determined by the number of variables and product terms) and (ii) crossbar size. When both are given, the solution space volume is fixed.

Defects in the crossbar pose constraints in the mapping process and result in a drastic decrease in the number of valid mappings in the solution space. Naturally, more defects in a crossbar lead to lower solution density. When a larger crossbar is used, there could be more solutions existing in the enlarged solution space as a result of a higher level of redundant resources being used. With solution space as well as the number of solutions increased resulting from large crossbar size, we examine the joint effect on the solution density expectation $E(sd)$.

When the solution space grows due to the increase of logic function size, it is easy to see that the number of solutions does not increase but decreases. First, there are no more redundant resources to be exploited. Furthermore, more rows / columns need to be considered at a time and thus more defects takes effect, making each mapping prone to be invalid. Therefore, when the logic function size gets larger, we expect $E(sd)$ to drop despite of the increased solution space.

As analyzed in the previous section, the solution density expectation can be expressed in Equation 4.1 by assuming that each device has a certain defect probability. We can draw the following observations:

1. $E(sd)$ is a polynomial function of defect probability. $E(sd)$ drops sharply with defect probability increasing when logic function size becomes large.
2. As logic function increases in size, $E(sd)$ decreases exponentially.
3. For a given defect probability and the desired logic function, $E(sd)$ is always the same regardless of crossbar size. This implies that the solution space volume and the number of solutions grow at the same speed.

4.3.2 Solution density standard deviation $V(sd)$

Besides $E(sd)$, which identifies the expectation of solution density, the standard deviation $V(sd)$ reveals the variability. In this particular problem, when $V(sd)$ is high, it indicates a dropping of accuracy when using $E(sd)$ for runtime prediction. As $E(sd)$ can be easily calculated by Equation 4.1, we focus on $V(sd)$ analysis so as to complete the discussion on runtime analysis. This entails investigation of how defect probability, logic function size and crossbar size determine $V(sd)$.

First, $V(sd)$ varies with defect probabilitys. When defect probability is low, all mapping trials tend to be valid regardless of the defect patterns, because the constraints imposed by the defects are very few. Consequently, the solution density of any mapping trials becomes stably high. On the other hand, when the defect probility is high, whether a mapping trial is valid depends highly on the defect pattern. Therefore, over all kinds of defect patterns under a certain (high) defect probability, $V(sd)$ is considerably large from the expected value $E(sd)$.

Besides defect probility, the aforementioned correlation in the solution space adds another layer of complexity and influences $V(sd)$ in an interesting way. Particularly, when a solution space is highly correlated, all mapping trials tend to be either valid or invalid, making high variation in solution density.

Mathematically, $V(sd)$ can be obtained as

$$\sigma_{sd} = \sqrt{\frac{\sum_{i=1}^{N}(sd_i - \mu_{sd})^2}{N}} \tag{4.6}$$

where sd_i is the solution density of an individual mapping and $E(sd)$ can be calculated. However, enumerating each sd_i for a specific crossbar is impractical due to an huge number of different mappings in

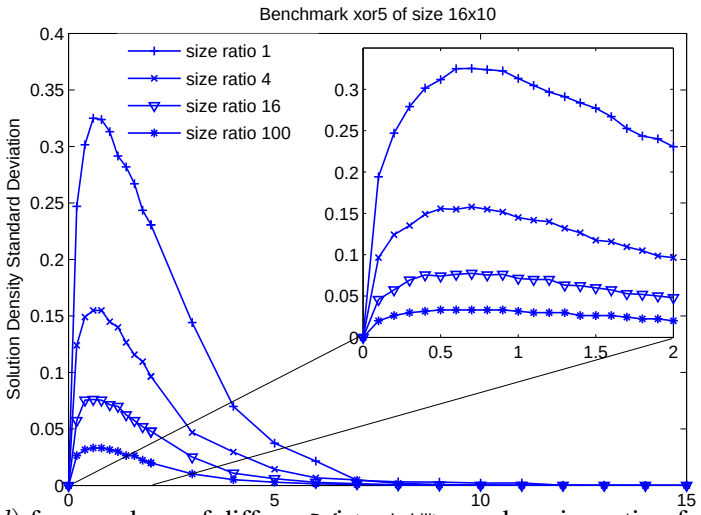

Figure 16. $V(sd)$ for crossbars of different sizes, expressed as size ratio of crossbar to benchmark.

the solution space. Consequently, $V(sd)$ is examined through simulations. A large number of defective crossbars generated under the same probabilistic model are used to obtain a sample space of solution densities. Logic synthesis benchmarks from the LGSynth93 benchmark set (73) are chosen to experiment with. To acquire solution density for each crossbar, 10^4 mappings are sampled randomly in the solution space. Extensive discussion on various factors impacting $V(sd)$ is provided in the following subsections, and we will show how crossbar size influences $V(sd)$ through correlation.

4.3.2.1 $V(sd)$ varies with defect probability

When the defect probability is low, $V(sd)$ is low. This is because solution densities are only slightly impacted when only a few defects exist. As is shown in Figure 16, $V(sd)$ increases as defect probability increases and reaches its peak roughly at defect rate 0.7%, beyond which $V(sd)$ begins to decrease. With increased defect probability, the impact of defect patterns on solution densities becomes significant. However, if crossbars contain a very high level of defects such that all the mappings tend to be invalid,

$V(sd)$ decreases since all solution densities are universally low. In Figure 16, $V(sd)$ becomes most significant at defect probability between $[0.2\%, 2\%]$, whose corresponding $E(sd)$ is within the range $[0.2, 0.8]$ according to Equation 4.1. Therefore, $V(sd)$ is maximized when the number of valid mappings is comparable to that of invalid mappings.

4.3.2.2 $V(sd)$ varies with logic function size

We examine how $V(sd)$ scales with logic function size. It turns out that at the same defect probability, different sized benchmarks have different $V(sd)$. The underlying reason is that under the same defect probability, various logic function size leads to different $E(sd)$. Since $E(sd)$ also depends on defect probability, we examine the combined impact of logic function size and defect probability on $V(sd)$. Figure 17 shows how $V(sd)$ varies as $E(sd)$ increases. As $E(sd)$ increases, $V(sd)$ increases and culminates at the point where $E(sd)$ is 0.5. When $E(sd)$ exceeds 0.5, $V(sd)$ begins to decrease. Though it seems that both logic function and defect probability influence $V(sd)$ individually, it turns out that $V(sd)$ is uniquely dependent on $E(sd)$. At the same $E(sd)$, $V(sd)$ is the same regardless of the difference on in logic function size.

4.3.2.3 $V(sd)$ decreases with crossbar size increasing

Figure 16 shows $V(sd)$ simulation results, where crossbar size is expressed in size ratio. Large crossbars lead to low $V(sd)$ uniformly at all defect probability, and $V(sd)$ reduction speed becomes slower as crossbar size further increases. For instance, $V(sd)$ decreases significantly when size ratio varies from 1 to 4, while the reduction is very limited for size ratio growing from 16 to 100. As crossbar size increases to infinity, $V(sd)$ approaches 0.

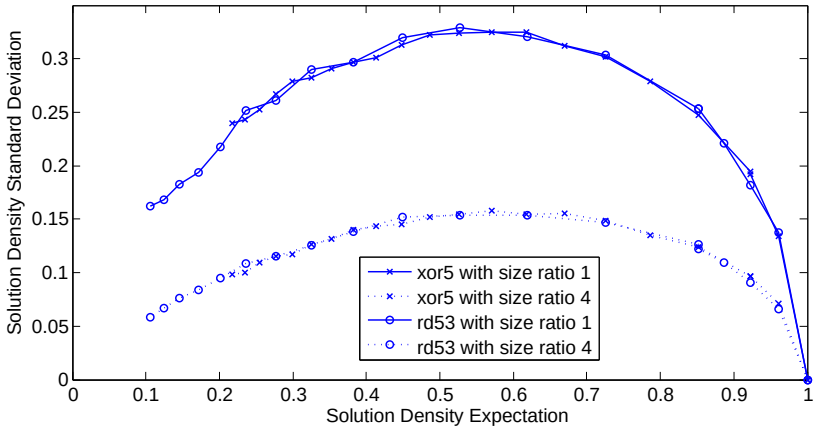

Figure 17. $V(sd)$ of two benchmarks mapped onto crossbars of different sizes.

The reason is large crossbars have less correlation in the solution space. When a logic function is mapped onto a small sized crossbar, the correlation in the solution space is very high, because of the limited number of rows / columns in a crossbar to be chosen from. With crossbar size increasing, the solution space grows and the correlation in the enlarged solution space is reduced. As a result of the impact from the correlation, larger crossbar size reduces $V(sd)$.

4.3.3 Solution density based runtime estimation

We use the number of mapping trials as the metric for runtime. When the solution space is explored with randomized algorithms, the expected number of mapping trials to hit one solution is inversely proportional to solution density. *Runtime expectation* is the expectation of variable $1/sd$ based on knowledge of random variable sd. Mathematically, expectation of $1/sd$ depends on both $E(sd)$ and $V(sd)$. We perform simulations to examine runtime expectation and compare the results with the curve of $1/\mu_{sd}$, which can be obtained from Equation 4.1, yet leaving out the impact from $V(sd)$.

Figure 18 shows runtime expectation results on benchmark $xor5$. The following observations are made:

- As $E(sd)$ increases, runtime expectation always decreases for any crossbar size. Obviously, higher solution density indicates it is easier to hit a solution in the solution space. We can conclude that lower defect probability leads to high $E(sd)$ and reduced runtime expectation. For the same reason, small sized logic functions reduce runtime expectation because of the high $E(sd)$.
- Large crossbar size leads to reduced runtime expectation at any $E(sd)$. The reason for this is large sized crossbars have small $V(sd)$ which, in turn, reduces runtime expectation. Ideally, if $V(sd)$ reduces to 0, then runtime expectation becomes $1/\mu_{sd}$. Since correlation always exists in the solution space and thus $V(sd)$ can not be 0, $1/\mu_{sd}$ is the lower bound for the actual runtime, as is shown in Figure 18.
- Runtime estimation based on $1/\mu_{sd}$ can predict runtime expectation accurately when crossbar size is above two times larger than the logic function size. This indicates that correlation in the mapping trials becomes very low as crossbar size increases to two times larger.

Overall, $1/\mu_{sd}$ is the lower bound for runtime expectation as runtime expectation depends on both $E(sd)$ and $V(sd)$. Thus solution density based runtime estimation entails taking into consideration not only expectation but also variance of solution density. With the same $E(sd)$, increased crossbar size leads to reduced runtime expectation.

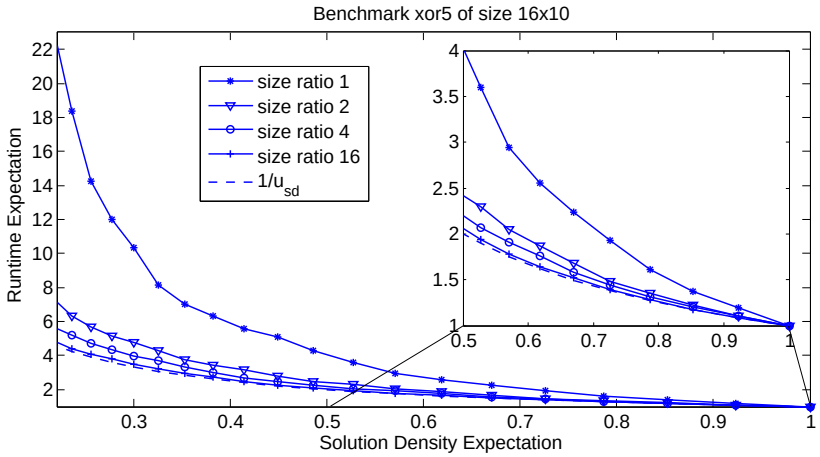

Figure 18. Runtime expectation for crossbars of different sizes.

4.4 Summary

We investigate the logic mapping problem in crossbar-based nanoelectronic systems. The main challenge is that finding a valid mapping takes prohibitive runtime when massive defects exist in a crossbar. Based on the mathematical model developed, probabilistic analysis is proposed to evaluate yield as well as runtime. The roots of the complexity of the problem lie in the existence of high correlations in the solution space.

Taking into consideration the correlations, we examine the interplay among crossbar size, defect probability, yield and runtime. We make the following conclusions:

- Yield can be improved by enlarging crossbar size. However, when runtime limit is applied, increasing crossbar size improves yield through decreasing correlations among the explored mapping trials, and a theoretical upperbound can be identified for runtime-constrained yield.

TABLE I

SUMMARY OF INFLUENCE OF DEFECT PROBABILITY, LOGIC FUNCTION SIZE AND CROSSBAR SIZE ON $E(SD)$, $V(SD)$ AND RUNTIME EXPECTATION.

Influence	Defect Probability ↗	Logic Function Size ↗	Crossbar Size ↗
$E(sd)$	↘	↘	constant
$V(sd)$	maximized when $E(sd)$=0.5		↘
Runtime	↗	↗	↘

- Runtime-constrained yield can also be improved by increasing the runtime limit. The improvement also has an upperbound. For large-sized logic functions, increasing runtime limit can significantly improve yield.
- Both the upperbound and lowerbound for runtime-constrained yield can be mathematically achieved.

Deterministically finding a correct mapping takes prohibitive runtime when massive defects exist in a crossbar. Influences of defect probability, logic function size and crossbar size on solution density expectation as well as variance and runtime are summarized in Table I. Particularly, the trade-off between runtime and crossbar size is shown. Large sized crossbars lower runtime by decreasing correlation between the explored mappings. This dissertation identifies the challenges with logic mapping onto defective nanoscale crossbars, and establishes a comprehensive model for the runtime of the problem. Capabilities of the defect tolerance on crossbar-based architectures are examined from the perspective of hardware cost and algorithm runtime, which provide the baseline and metric on evaluating future heuristic algorithms.

CHAPTER 5

EVALUATING QUALITY FOR NANOCROSSBAR LOGIC MAPPING THROUGH MISMATCH NUMBER DISTRIBUTION

5.1 Motivations

The complexity of defect-tolerant logic implementation problem is NP-complete(43; 44). As the success of logic implementation is under the severe constraint of a massive number of defects, the logic implementation process has unpredictable runtime when the defect rate is high. As a result, runtime estimation is exceedingly hard(48). Moreover, since each crossbar has its unique defect pattern, the lengthy logic implementation process has to be performed on every single crossbar. Consequently, yield analysis, which concerns the percentage of crossbars with valid mappings, is hard to model mathematically, and also expensive to derive through simulations(47).

Essentially, both runtime and yield analysis depends on the percentage of mismatch-free mappings, when defect-tolerant implementations are solely based on mapping schemes. However, such a percentage (defined as "solution density") is hard to achieve precisely with mathematical models (47; 48), due

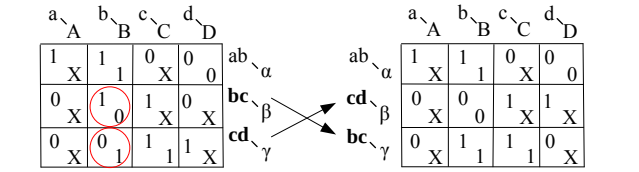

Figure 19. Two mapping trial examples

to the highly correlated mapping trials in a solution space. In fact, we need to move beyond focusing only on the percentage of the mismatch-free mappings. When both the mismatch-free and the *mismatched* mappings are considered together in an integrated picture, it is actually possible to establish mathematical models and derive meaningful evaluation methods for runtime and yield. For each mapping trial, no matter mismatched or not, its mapping quality can be represented largely by a "score", namely the number of mismatches in the mapping trial. A mismatched mapping has its specific number of mismatching cells, which essentially indicates how "bad" such a mapping trial is. It turns out that, when all the mapping trials are considered, such mismatch numbers follow certain mathematical distributions and can be well modeled. Since the percentage of valid mapping can be directly derived (by looking at the percentage with zero mismatch) once the distribution is available, the knowledge of mismatch number distribution serves as a basis for yield and runtime estimation.

In general, a mapping is valid (therefore useful) only when its mismatch number is zero. However, a limited number of mismatches do not necessarily lead to a bad logic implementation, as work in (62) shows that these mismatches can be "utilized" through logic morphing. In addition, when hardware redundancy is added, a logic function can be hardened and able to tolerate up to a certain level of mismatches (67). Therefore, mappings with certain mismatches can still result in a defect-tolerant implementation when they are treated specially. Research on mismatch number distribution thus provides insights in the tradeoffs among hardware cost, defect tolerant capability, and yield.

In this dissertation, we first examine mismatch number distribution Δ over a large number of solution spaces, when crossbars have the same defect probability. Second, we focus on mismatch number

distribution D in the solution space of one crossbar with a given defect pattern. Lastly, we show how D approaches Δ with the increase of hardware redundancy.

5.2 Mismatch number distribution among crossbars with the same defect probability: Δ

We begin with a given defect probability d for a set of crossbars. Considering the two types of defects (closed and open), we introduce *closed defect ratio* r, defined as the percentage of closed defects in both types of defects. Thus, each switch has probability:

- $p_0 = d(1 - r)$ of being open.
- $p_1 = dr$ of being closed.
- $p_X = 1 - d$ of being defect-free (configurable).

Similarly, we define *logic inclusion ratio* l as the percentage of inclusion cells among all cells in a logic matrix.

5.2.1 Probabilistic modeling of Δ

5.2.1.1 Binomial distribution

With a logic function of size $n \times m$ and logic inclusion ratio l, there are nml "1" cells and $nm(1-l)$ "0" cells in the logic matrix. In an arbitrary mapping, each "0" cell in the logic matrix has a probability $p_1 = dr$ of having mismatch $0 \rightarrow 1$, because a cell in the crossbar matrix has p_1 of being "1". Similarly, each "1" cell in the logic matrix has a probability of $p_0 = d(1 - r)$ having mismatch $1 \rightarrow 0$.

Mapping all "0" cells in a logic matrix onto a probabilistic crossbar matrix is essentially a series of Bernoulli events, and hence the distribution Δ_0 of mismatch $0 \rightarrow 1$ follows Binomial distribution.

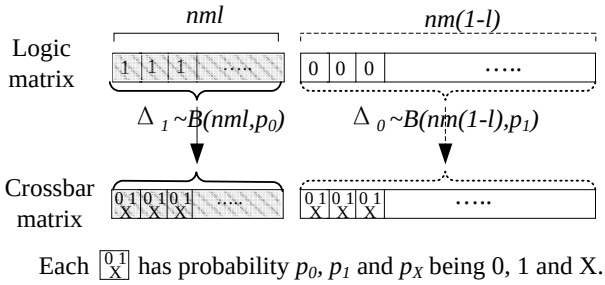

Figure 20. Mismatch number follows Binomial distribution model

Similarly, mapping all "1" cells in a logic function is another independent series of Bernoulli events, and the distribution Δ_1 of mismatch $1 \rightarrow 0$ also follows Binomial distribution. Therefore:

$$\Delta_0 \sim B(nm(1-l), dr), \Delta_1 \sim B(nml, d(1-r)) \tag{5.1}$$

Here, $B(n_B, p_B)$ denotes Binomial distribution with parameters n_B and p_B. A mapping trial can be viewed as two sets of independent bernoulli events Δ_0 and Δ_1, as is shown in Figure 20.

The distribution Δ of overall mismatches, including both $0 \rightarrow 1$ and $1 \rightarrow 0$, is the joint distribution of Δ_0 and Δ_1. Such a process of computing joint distribution can be time-consuming, yet it is possible to approach the distribution Δ with closed forms.

5.2.1.2 Closed form approximation of Δ using Normal and Poisson distributions

5.2.1.2.1 $\Delta \sim$ Normal distribution

According to de Moivre-Laplace theorem which is a special case of Central Limit Theorem, Binomial distribution can be accurately approximated by Normal distribution when the number of trials n_B is large. Therefore, Δ_0 and Δ_1 can be approximated as following:

$$\Delta_0 \sim N(\mu_0, \sigma_0^2), \Delta_1 \sim N(\mu_1, \sigma_1^2) \tag{5.2}$$

where

$$\mu_0 = nm(1-l)dr, \mu_1 = nmld(1-r) \tag{5.3}$$

$$\sigma_0^2 = nm(1-l)dr(1-dr) \tag{5.4}$$

$$\sigma_1^2 = nmld(1-r)(1-d(1-r)) \tag{5.5}$$

Due to the the linear attribute of Normal distribution, the distribution Δ can be approximated as:

$$\Delta \sim N(\mu, \sigma^2) = N(\mu_0 + \mu_1, \sigma_0^2 + \sigma_1^2) \tag{5.6}$$

where

$$\mu = nmd(l + r - 2lr) \tag{5.7}$$

$$\sigma^2 = nm[dr - d^2r^2 - l(d - r^2)(2d - 1)] \tag{5.8}$$

Based on the above closed forms, we can observe that:

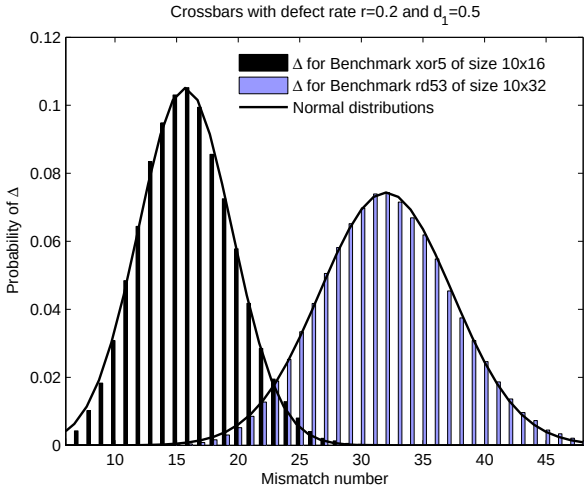

Figure 21. Mismatch number distributions Δ for different benchmarks

- the expectation of mismatch number increases with both defect probability d and logic function size nm.
- the expectation of mismatch number is affected by both closed defect ratio r and the logic inclusion ratio l. When both r and l approaches 1(or 0), the expectation is minimized. This is because more 1(0) cells in a logic matrix necessarily requires more 1 or X(0 or X) cells in a crossbar matrix.
- if l(or r) is 0.5, the expectation of mismatch number becomes $nmd/2$, determined only by logic function size and defect probability. This implies that when both types of defects occur with equal probability in the fabrication process, half of the defects are always tolerated regardless of logic function attributes. In other words, the expectation of mismatch number becomes $nmd/2$, which is only determined by logic function size and defect probability.

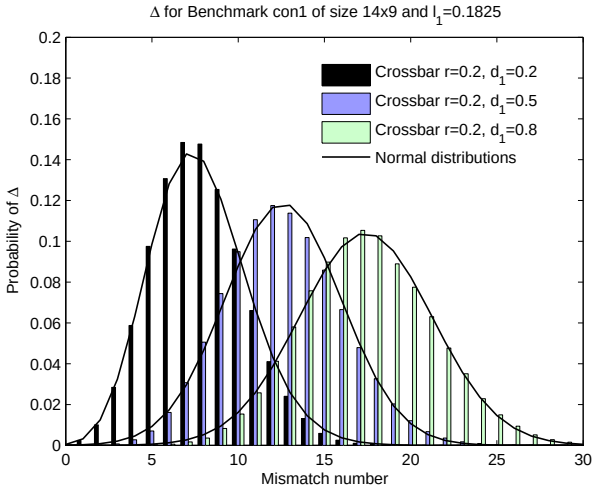

Figure 22. Distribution Δ when varying closed defect ratio r

5.2.1.2.2 $\Delta \sim$ Poisson distribution

The Poisson distribution with parameter $\lambda = n_B p_B$ can be used as an approximation for $B(n, p)$, when n_B becomes large while the product $n_B p_B$ remains fixed. Therefore,

$$\Delta_0 \sim Pois(\mu_0), \Delta_1 \sim Pois(\mu_1) \tag{5.9}$$

where μ_0, μ_1 are expressed in Equation 5.3. Since Poisson distribution also has linearity property, Δ can be approximated as:

$$\Delta \sim Pois(\mu) = Pois(\mu_0 + \mu_1) \tag{5.10}$$

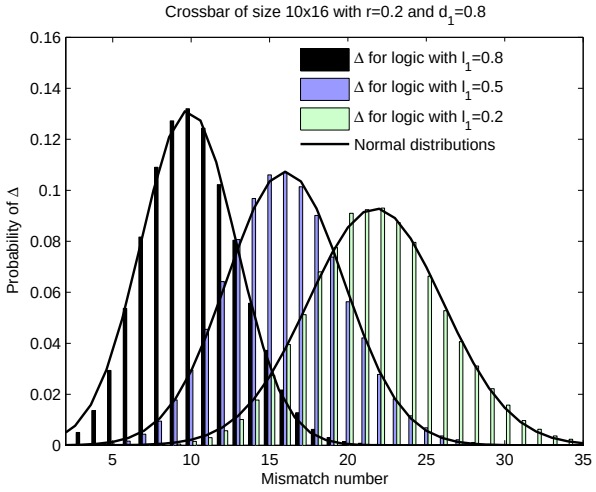

Figure 23. Distribution Δ when varying logic inclusion ratio l

5.2.2 Discussion and experimental results

To see how Δ changes upon various factors, we perform experiments with the LGSynth93 benchmark set (73). First, we show how Δ is affected by the logic function size. Simulation results for two logic function benchmarks are shown in Figure 21. Even though the crossbars have the same defect probability, the mismatch number distributions Δ of the two benchmarks are significantly different. Clearly, the large-sized benchmark *(rd53)* has more mismatches than the smaller one *(xor5)*.

Second, we show how l and r affect mismatch number distribution Δ. Figure 22 shows how closed defect ratio r impacts the distribution Δ. Basically, Δ shifts to the right and becomes flatter as closed defect ratio r increases. This is because the benchmark *con1* has logic inclusion ratio $l = 0.1825$ and thus can tolerate more open defects. When r is fixed, we generate logic functions of various inclusion ratio l to see how the distribution Δ is affected. Figure 23 shows that, with crossbar closed defect

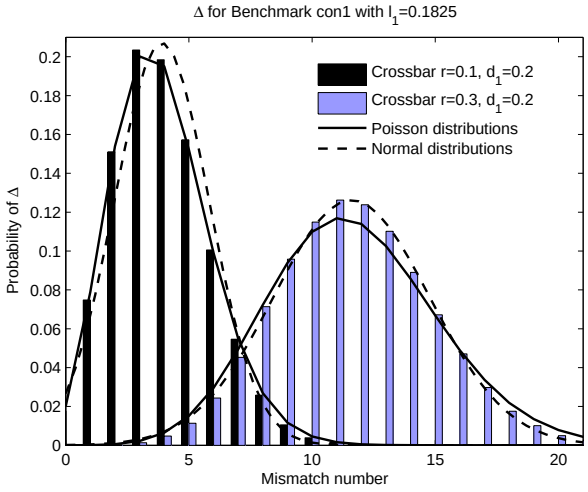

Figure 24. Comparison: Normal and Poisson in approximating Δ

ratio $r = 0.8$, distribution Δ for logic functions with larger l shifts to the left side, indicating reduced mismatches in solution spaces.

Lastly, we compare approximation accuracy of Normal distribution and Poisson distribution for Δ, and show how they fit into different region of defect probability r. Figure 24 shows two distinct mismatch number distributions with the expectation $n_B p_B$ being 3.9 and 11.7, respectively. It is clear to see that Poisson distribution approximates better when the expectation of mismatch number is small, while Normal distribution has better accuracy in the other case.

5.3 Mismatch number distribution of a single crossbar with a solution space: D

We have shown that the expected probability distribution across a large number of crossbars with same defect probability, Δ, is a combination of two independent binomial distributions, and can be accurately approximated by Normal/Poisson distributions. For a single crossbar with a given defect

pattern, the mismatch distribution (denoted as D) is much harder to model, since all the mappings within a solution space are not independent anymore. Figure 25 shows the mismatch distributions D of two specific crossbars, the size of which is the same as that of the benchmark logic function. First, the two distributions, even though both representing mapping trials of the same logic function onto same-sized crossbars, differs significantly. Second, Normal distribution fails to capture either of them. Particularly, the two distributions seem to have similar shape, but differ in overall mismatch number expectation. The discrepancies are then possibly caused by the variance of defect number and defect pattern.

Intuitively, large defect number increases the number of mismatches in mapping solution space, making D shift towards the right side. When it comes to different defect pattern, by which we mean defects with different locations on a crossbar, some defect patterns could help utilize more defects while other kind of defect patterns may make mappings more suffering the defects. Therefore, an understanding of mismatch distribution with one crossbar requires the insides on the roles played by: 1) defect number and 2) defect pattern.

5.3.1 Mismatch number distribution over crossbars with the same defect number: $\widetilde{D}$

To focus on a set of crossbars that are similar, we examine the distribution $\widetilde{D}$, when the number of defects is fixed for the crossbar set. To illustrate the sole impact from defect number, we assume a logic function and a crossbar have the same size.

When the defect numbers are known for crossbars, it is still hard to derive the mismatch number distribution theoretically, for the reason that the mismatch number of a mapping is determined by the complicated two-dimension matrix mapping. It is almost impossible to enumerate all the possible map-

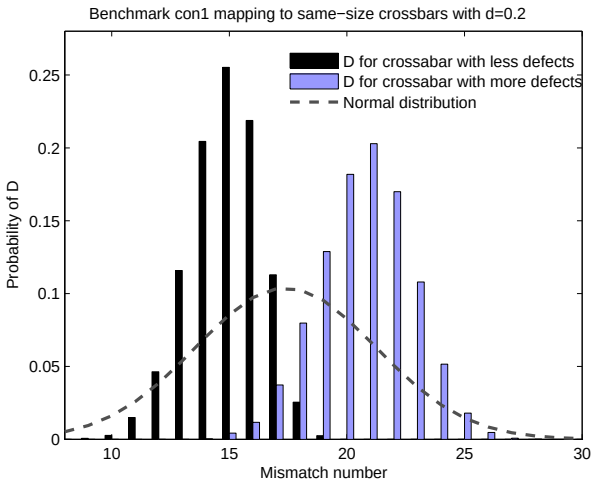

Figure 25. Two specific mismatch distributions vs. expected mismatch distribution

pings and obtain the mismatch numbers, and then derive the mismatch distribution. Yet, in this section, we consider the "average" distribution for a number of crossbars sharing the same number of defects, which basically mean that defects can happen at every location in the crossbar. This essentially tremendously reduces the impact from defect pattern. As such, we use a relaxed mapping model to approximate the complicated two-dimension logic mapping in analyzing the mismatch distribution.

Once the number of "0", "1"' and "X" are fixed, the distribution of mismatch number $\widetilde{D}$ with relaxing the defect pattern restrictions falls into the category of Hypergeometric distribution. Due to three types of switches existing in a crossbar, $\widetilde{D}$ turns out to be under bivariate Hypergeometric distribution model: one variable v_0 represents the number of $0 \rightarrow 1$ mismatches, and the other v_1 indicates the number of $1 \rightarrow 0$ mismatches. The probability mass function of the overall mismatch number distribution $\widetilde{D}$ can be solved by adding the corresponding joint probabilities of the two variable v_0 and v_1.

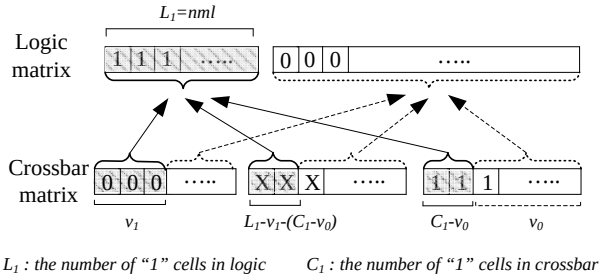

Figure 26. Mismatch number follows bivariate Hypergeometric distribution

Figure 26 illustrates how a bivariate Hypergeometric distribution model is developed. Since the sizes of the two matrices (logic function and crossbar) are the same, any mapping trial is essentially a one-to-one cell mapping between the logic and crossbar matrix. Cells of the same type in each matrix are put together for illustration purpose. First, we focus on all the "1" cells in a logic matrix, and choose the same number of cells from a crossbar matrix to match with the "1" cells. The chosen cells could come from three groups: "0","X" and "1". To ensure a mapping trial with v_1 mismatches of $1 \rightarrow 0$, there must be v_1 "0" cells from the crossbar matrix to match "1" cells in the logic matrix. In other words, the rest of "0" cells in the crossbar matrix have to go with "0" cells in a logic matrix. Similarly for v_0 mismatches of $0 \rightarrow 1$, v_0 "1" cells in the crossbar matrix have to map with "0" cells in the logic matrix, with the rest "1" cells mapping with "1" cells.

Thus, according to bivariate Hypergeometric distribution, the joint probability mass function can be expressed as:

$$\widetilde{D}(v_0, v_1) = \frac{\binom{C_0}{v_1}\binom{C_1}{C_1-v_0}\binom{C_X}{L_1-v_1-C_1+v_0}}{\binom{C_0+C_1+C_X}{L_1}} \tag{5.11}$$

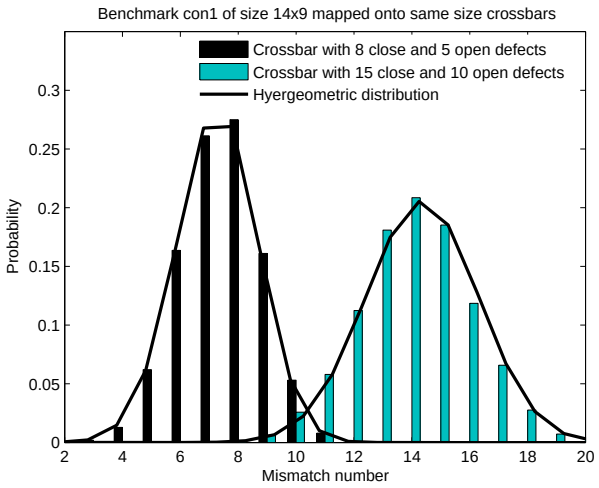

Figure 27. $\widetilde{D}$ distribution can be modeled as a Hypergeometric distribution

where L_1 is the number of "1" cells in the logic matrix and C_1, C_0, C_X are the number of "1", "0", "X" cells in the crossbar matrix. There are up to $min(L_0, C_1)$ mismatches of $0 \rightarrow 1$ and $min(L_1, C_0)$ mismatches of $1 \rightarrow 0$ in a mapping trial. Thus, the two-dimensional table for joint probabilities of variable v_0 and v_1 is of size $min(L_0, C_1) \times min(L_1, C_0)$.

We experiment with crossbars based on defect number to verify the above analysis. Figure 27 shows the experimental results of crossbars with different defect numbers, and the $\widetilde{D}$ in both cases matches the curves of Hypergeometric distribution. Since the Hypergeometric distribution model is a relaxed modeling for mismatch distribution, we will examine how good it is when using it to predict the mismatch distribution for a crossbar with a specific defect pattern in the next section.

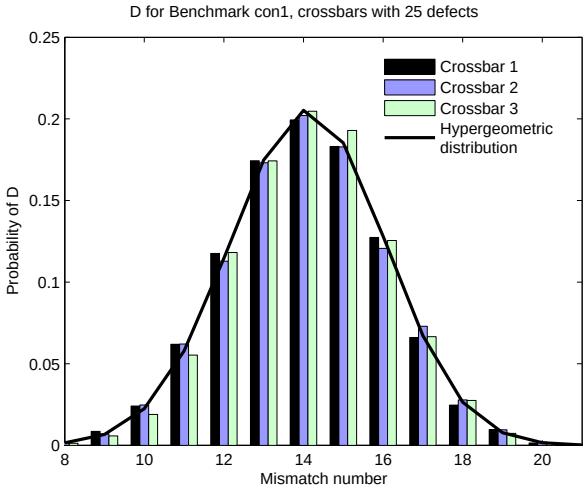

Figure 28. Three specific mismatch number distributions D vs. Hypergeometric distribution $\widetilde{D}$

5.3.2 Impact of defect pattern

We now move to discuss the mismatch number distribution D when a single crossbar is considered. Three specific crossbars (having the same size as the logic function) are used and the distributions of experiment results are shown in Figure 28, where two observations can be made. First, all the distributions can be approximated by the Hypergeometric distribution of $\widetilde{D}$ tightly. Second, the difference is negligible among crossbars of the same defect number but distinct defect pattern. In order to further confirm that Hypergeometric mismatch distribution model can be used to accurately approximate the mismatch distribution for any crossbar with a specific defect pattern, we experiment with a number of crossbars and see how they fit Hypergeometric distribution. With 100 randomly generated crossbars of the same number of defects and yet various defect patterns, we examine their mismatch distributions. Figure 29 shows the mismatch distribution variation among these crossbars and how they fit the Hyper-

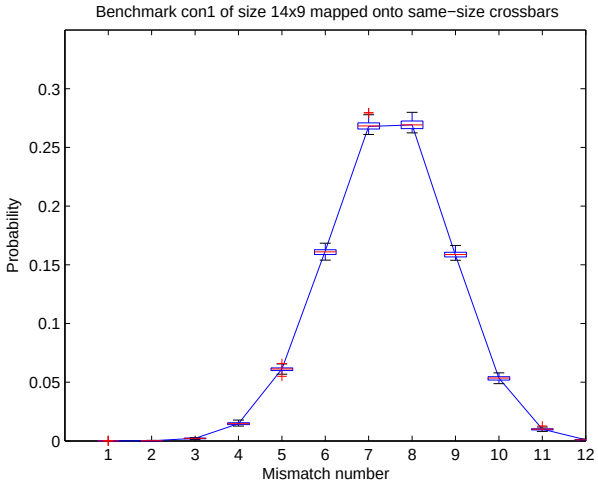

Figure 29. Mismatch distributions for crossbars with various defect patterns can be approximated by Hypergeometric distribution $\widetilde{D}$

geometric distribution model. It is clearly shown that variations among these crossbars are negligible, and basically the Hypergeometric distribution can be used to model the mismatch distribution for any one of these crossbars.

The above two sets of experiments imply that defect pattern has very little impact on mismatch number distribution. When considering a specific crossbar, mismatch number distribution D can be quite accurately approximated by Hypergeometric distribution of $\widetilde{D}$, i.e. mismatch distribution on crossbars with the same number of defects.

So far, we have examined the mismatch number distributions: Δ (over crossbars with the same defect probability), $\widetilde{D}$ (over crossbars with the same defect number) and D (on a specific crossbar). When only defect probability over a large number of crossbars is available, the distribution Δ follows

Normal/Poisson distribution. The information at this level can be used to estimate the quality of mapping a logic function onto a bunch of crossbars with the same defect probability. We can more precisely model the mismatch number distribution $\widetilde{D}$ as Hypergeometric distribution when the specific defect number is obtained through the chip-testing process.

In addition, discrepancy exists between the Hypergeometric distribution and Normal distribution. The Hypergeometric distribution is more centralized, indicating a small variation, and thus the number of mismatches in different mapping trials does not vary as much as that in Normal distribution. This means that if we obtain more deterministic defect information about crossbars, such as the total defect number instead of overall defect probability, the mismatch number of a mapping can be estimated more accurately. Mismatch number distribution essentially indicates the overall mapping quality, since it reveals the percentage of mismatch-free mappings, as well as the percentage of useful mappings with any number of mismatches.

5.3.3 Increasing crossbar size: $D \rightarrow \Delta$

We have examined D based on the assumption of the crossbar size being equal to the logic function size. Now let us consider the scenario that hardware redundancy is added, and only a part of the crossbar needs to be selected for logic mapping. Even though the entire crossbar has a fixed defect number, a part of the crossbar may contain more defects which make the local defect number larger than some other areas. Figure 30 shows the mismatch number distributions D when mapping a logic function onto crossbars of large size. We can see that, when crossbar size increases, D begins to deflect from the Hypergeometric distribution of $\widetilde{D}$ and finally conforms to Normal distribution of Δ.

On large crossbars, the fact that D converges to Δ indicts that it becomes equivalent to mapping onto a large number of crossbars based on the same defect probability. In other words, all the parts of a large crossbar used in the mapping process can be viewed as a collection of small-sized crossbars with the same defect rate. Distribution D sits between the Hypergeometric and Normal distributions when the crossbar size is not significantly large. Therefore, when crossbar size is larger than logic function size, D is always bounded by these two distributions $\widetilde{D}$ and Δ. Both these two bounds can be computed, thus providing an estimation for D when crossbar size is larger than the size of the logic function.

When the crossbar size grows, the mismatch number distribution becomes much wider. As a result, the probability of zero mismatch increases, and this is essentially how hardware redundancy contributes to yield directly when only mismatch-free mappings are considered. In addition, the probability of mappings containing small mismatch number also increases, and these lightly mismatched mappings can generally be used for defect-tolerant purposes through either logic morphing or hardening. Lastly, as we can see clearly that the improvement introduced by more hardware redundancy is ultimately bounded by Δ. Theoretically, this upperbound confirms to a crossbar of infinite size.

5.4 Summary

Defect-tolerant logic mapping onto nanocrossbars becomes a new fundamental challenge in the post-fabrication design phase, and is in nature a hard problem. Evaluation methodologies for such a challenging problem are necessitated so as to produce workable tools and platforms for nanocrossbar-based computing systems. Evaluation metric for the logic mapping process from the perspective of mismatch number distribution is presented in this chapter. Based on the presented methodologies and

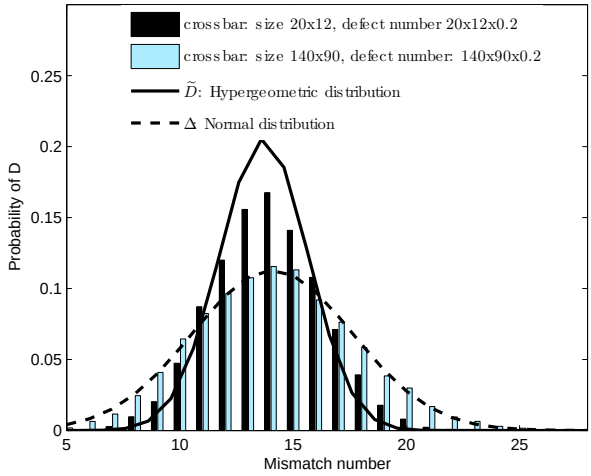

Figure 30. Distribution D sits between $\widetilde{D}$ and Δ and finally fits well with Δ

techniques, the mismatch number indicating the mapping quality can be well modeled, and the main outcomes are the following:

- Mismatch distribution across a large number of crossbars having the same defect rate Δ can be modeled precisely by independent Bernoulli events, and accurately follow Normal/Poisson distributions.
- Mismatch distribution for $\widetilde{D}$ for crossbars having the same size as the logic function and a fixed number of defects can be modeled by a bivariate Hypergeometric distribution.
- Mismatch distribution within a mapping solution space of one single crossbar D can be approximated accurately by mismatch distribution $\widetilde{D}$, and the impact of defect pattern on mismatch distribution is negligible.

- With hardware redundancy added in a crossbar, the mismatch distribution moves from Hypergeometric distribution to Normal/Poisson distribution, and is bounded by these two well-defined distributions. So, the benefits with increasing hardware redundancy can be quantified.

Overall, examining the mapping quality from mismatch number distribution serves as a starting point in analyzing logic mapping on nanocrossbars, as it reveals insights on understanding the various new factors involved in the logic mapping process (including defect probability, defect number, hardware redundancy and mismatch number), and the tradeoffs for achieving defect-tolerant logic implementations.

CHAPTER 6

DEFECT-TOLERANT LOGIC MORPHING

6.1 Motivations

Logic mapping schemes exploit the choice of selecting which column / row of a crossbar matrix to be used for a variable / product of a logic matrix, under the constraints of no mismatches. As the complexity of such a problem is NP-complete (43)(44), the runtime curve goes through a phase transition as defect level increases, because valid solutions become too rare. In such cases, exploiting an orthogonal dimension of freedom can help boost the number of valid solutions, by recognizing the various equivalent forms of any given logic function. This advantage comes from the fact that logic functions are inherently polymorphic, in the sense that a logic function can take various equivalent forms.

In other words, not all mismatches are created equal: some are actually "tolerable" - as long as the resulting logic form is equivalent to the original function. For instance, $f = a'c' + c'd + a'b$ is equivalent to $f^* = a'c'\underline{d'} + c'd + a'b$, therefore the mismatch resulting in an additional d' to product term $a'c'$ is tolerable. Allowing tolerable mismatches in a logic form essentially means morphing the logic form, because mismatches change the original structure of the logic. In other words, mismatches caused by mapping the original logic form to a crossbar can be "removed" through morphing the logic form, so that the mapping between the morphed logic form and the crossbar is mismatch-free.

The mismatch tolerance capability can exist in various forms and can be difficult to predict. As is shown in Figure 31, function $f = a'c' + c'd + a'b$ can tolerate the mismatch resulting in $a'c' \rightarrow a'c'\underline{d'}$

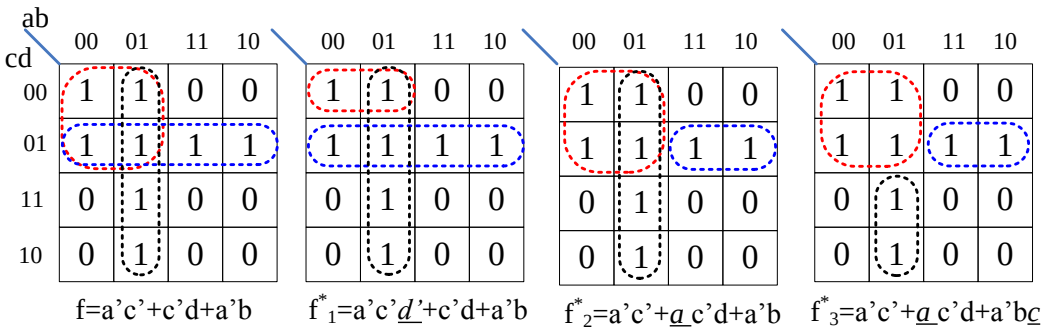

Figure 31. K-map showing the equivalent forms of a logic function

in f_1^*, or the mismatch $c'd \rightarrow \underline{a}c'd$ in f_2^*, but not both. In another case, f can tolerate two mismatches at the same time (illustrated in f_3^*). When the function is not fully optimized, there are also cases where the opposite is true, i.e., the combination of multiple mismatches can be tolerated, but not some subset of it.

6.2 Logic morphing

By realizing that a logic function can be represented by various equivalent forms, the flexibility of exploring these equivalent forms can be utilized for defect tolerance purposes. Then, basically a logic form will be restructured in the implementation process. Thus, two questions need to be answered: 1) how to make sure that the restructured form is equivalent to the original one, and 2) how to efficiently perform the logic morphing.

6.2.1 Logic equivalence checking

The first issue boils down to determining whether the restructured logic function due to mismatches is equivalent to the original one. Yet, one cannot derive which mismatches are tolerable by trivial calculation, and logic equivalence checking needs to be performed to determine tolerable mismatches. In

general, equivalence checking of any two logic functions is hard. However, when the two functions to be compared are similar, equivalence checking can be accomplished very efficiently. This observation opens up the possibility of exploring the morphing space during the mapping process. Basically, mismatches typically result in "morphed" logic forms that are very similar to the original form, as a mismatch only changes the function by adding or dropping a variable or product term.

6.2.2 Efficient algorithm for logic equivalence checking

Because one cannot derive which mismatches are tolerable by trivial calculation, logic equivalence checking needs to be performed to determine tolerable mismatches. In general, equivalence checking of any two logic functions is hard. However, when the two functions to be compared are similar, equivalence checking can be accomplished very efficiently. This observation opens up the possibility of exploring the morphing space during the mapping process. Basically, mismatches typically result in "morphed" logic forms that are very similar to the original form, as a mismatch only changes the function by adding or dropping a variable or product term.

We adopt a divide and conquer approach, which decomposes the original logic function into two subfunctions, according to Shannon Expansion (74). In every step, one *splitting variable* x_i is chosen to decompose the function:

$$f(x_1, x_2, \cdots, x_n) = x_i' f_{x_i=0} + x_i f_{x_i=1} \tag{6.1}$$

After decomposition, equivalence checking is performed on the subfunctions. f^* is equal to f if and only if the two pairs of subfunctions are equivalent: $f^*_{x_i=0} = f_{x_i=0}$ and $f^*_{x_i=1} = f_{x_i=1}$. When

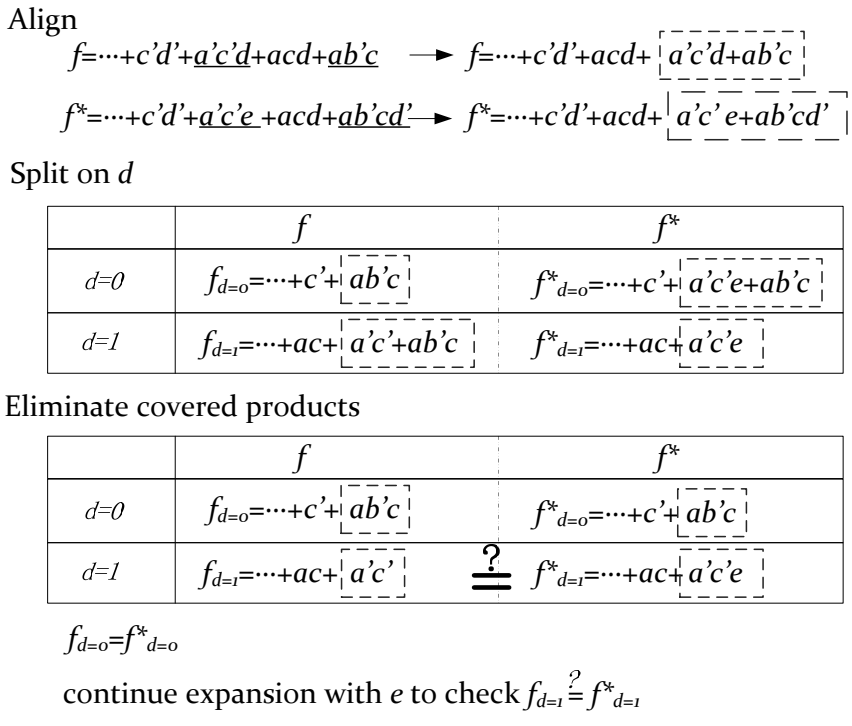

Figure 32. Logic equivalence checking with an example

the equivalence cannot be determined immediately, they are further decomposed, possibly until the leaf level.

When f^* is a mismatched form of f, the difference between f and f^* is caused by the mismatches. This means that f and f^* have a large number of common product terms, plus only a small part of the unique products that need to be compared. Figure 32 shows an example of the proposed equivalence checking between two functions f and f^*, with multiple mismatches: $a'c'\underline{d} \rightarrow a'c'\underline{e}$ and $ab'c \rightarrow ab'c\underline{d'}$. After the identification of the unique products of f^* and f, Shannon expansion is applied by choosing a

mismatched variable d to split on. Then, the products unique to f_d^* and f_d are compared, and the process continues to the next splitting variable, e.

A number of heuristics are adopted to help accelerate the equivalence checking process:

- An elimination process is performed to remove the covered[1] products in the unique part from the common products. For instance, product $a'c'e$ is covered by c' in $f^*_{d=0}$ in Figure 32.
- When the unique parts of two subfunctions have the same number of products after elimination, we check to see if they are of the same form. This leads to a quick decision if f_d and f_d^* are equivalent, such as the case of $f^*_{d=0} = f_{d=0}$ in Figure 32. Otherwise, further expansion needs to be performed.
- If the unique parts have a different number of products after the elimination process, then whether $f_d = f_d^*$ is unknown, and further expansion needs to be performed.

The logic equivalence checking algorithm is summarized in Algorithm 2, which recursively checks the equivalence of the subfunctions. By categorizing the product terms into 3 parts: common (pt_common), unique to f (pt_f), and unique to f^* (pt_f^*), only the unique parts (pt_f and pt_f^*) need to be focused on. In choosing the splitting variables, the mismatched variables have the highest priority. This makes it possible for the subfunctions f^* and f to be similar, so that they might be trivially compared to shorten the recursive algorithm. After the function splits over on the mismatched variables, the most frequently appearing variables are chosen next to further decompose until the equivalence checking is solved.

---

[1]Product p_1 is said to be covered by p_2 when p_2 is certain to evaluate to 1 as long as p_1 evaluates to 1.

Algorithm 2 Logic Equivalence Checking

Logic_Eq_Check (f, f^*)

1. pt_common = the common product terms of f and f^*
2. pt_f^* = products unique to f^* (because of mismatches)
3. pt_f = products unique to f
4. Select a variable v, according to the mismatch position
5. if ***Split_and_Compare***$(pt\_common, pt\_f, pt\_f^*, v)$==true
   return success
   else return false

Split_and_Compare$(pt\_common, pt\_f, pt\_f^*, v)$

1. $pt\_c0 = pt\_common$ with $v = 0; pt\_f0 = pt\_f$ with $v = 0$
   $pt\_f^*0 = pt\_f^*$ with $v = 0$
2. $pt\_c1 = pt\_common$ with $v = 1; pt\_f1 = pt\_f$ with $v = 1$
   $pt\_f^*1 = pt\_f^*$ with $v = 1$
3. $branch0$=***Trivial_Compare***(pt_c0, pt_f0, pt_f^*0)
   $branch1$=***Trivial_Compare***(pt_c1, pt_f1, pt_f^*0)
4. if($branch0$ == false OR $branch1$ == false)
   return false *//any subfunction not equal*
5. if($branch0$ == true AND $branch1$ == true)
   return true *//both subfunctions equal*
6. pick the next split variable u *//cannot be decided immediately*
7. if($branch0$ == unknown), then $branch0$ = ***Split_and_Compare***$(pt\_c0, pt\_f0, pt\_f^*0, u)$
   if($branch1$ == unknown), then $branch1$ = ***Split_and_Compare***$(pt\_c1, pt\_f1, pt\_f^*1, u)$
8. return ($branch0$ AND $branch1$)

Trivial_Compare$(pt\_common, pt\_f, pt\_f^*)$

1. remove products in pt_f, pt_f^* covered by pt_common
2. if pt_f and pt_f^* contain different numbers of products
   return false *//fast check without comparing products*
3. else perform direct product comparison
   a) if $pt\_f == pt\_f^*$, return true
   b) if $pt\_f \neq pt\_f^*$
      if all variables are split *//at leaf level*
      return false
      else return unknown

6.2.3 Mismatch-tolerating capability and analysis

Generally every logic function has a certain "resilient" structure, which is tolerable of mismatches and still leads to an equivalent form. In this section, we study how resilient the structure of a logic function is, so as to reveal the mismatch-tolerating capability.

We examine a number of logic function benchmarks (73) in order to learn the rough percentages of single mismatches that are inherently tolerable. As is shown in Table II, $2-10\%$ of the mismatches can be tolerated in single occurrence. Furthermore, it turns out that a function typically tolerates only one of the two mismatch types: either $0 \rightarrow 1$ or $1 \rightarrow 0$, but not both. Most functions tolerate the $0 \rightarrow 1$ mismatch type (resulting in appearing variables such as $ab \rightarrow ab\underline{c'}$), indicating that highly optimized functions have products containing only the minimum number of variables, thus tolerating mostly the $0 \rightarrow 1$ mismatches, but not otherwise.

6.3 Exploiting mapping and morphing simultaneously

By exploring the equivalent logic forms, morphing opens up a large space of logic implementations. In particular, the benefit is prominent when defect rate is high, and mismatch-free mappings become hard to find. However, the delivery of such potential advantages hinges on an integrated algorithm that explores the combined solution space of mapping and morphing efficiently.

In this section, we introduce an integrated algorithm to exploit mapping and morphing simultaneously. Logic morphing is performed throughout the mapping framework (in Algorithm 1). The difference lies in the way of dealing with the mismatches. In the mapping-only scheme, mismatches simply lead to invalid solutions. With morphing, mismatches might lead to valid solutions, and should be treated with logic equivalence checking. In the midst of the mapping process, if the equivalence

Benchmark	Size	Number of tolerable 0→1 mismatch	Number of single tolerable 1→0 mismatch	Percentage of all single tolerable mismatch
con1	9×14	6	0	4.76%
rd53	32×16	9	0	2.81%
sqrt8	40×16	86	0	13.43%
5xp1	75×14	98	1	9.42%
misez1	32×16	14	0	2.73%
bw	87×10	5	90	10.92%
9sym	87×18	25	0	1.59%
sao2	58×20	62	0	5.34%

TABLE II

PERCENTAGE OF SINGLE TOLERABLE MISMATCHES.

checking of a mismatch returns true, the backtracking process migrates to a different (but equivalent) logic form, thus performing "morphing". A new subroutine $Mismatch\_Tolerance\_1$ (shown in Algorithm 3) is used wherever $Mismatch\_Check$ is called in Algorithm 1. With the new logic form as the target implementation form, the backtracking process continues for the unmapped part of the logic function. Overall, logic morphing is performed whenever mismatches are found to be tolerable during the mapping process.

When exploiting both mapping and morphing simultaneously, the proposed algorithm framework traverses through only the logic forms triggered by the mismatches in the mapping process. Such a "morphing only when necessary" scheme avoids the overhead of checking for all the equivalent forms.

Since runtime is crucial in the defect tolerant logic implementation process, we use a hash table to cache the results of the logic equivalence checking. In step 3 of Algorithm 3, when mismatches

are encountered in the mapping process, the hash table is checked first, before going through the logic equivalence checking procedure. The results of this checking are added to the hash table. This means the runtime overhead is amortized throughout the entire process.

Algorithm 3 Mismatch Tolerance in Mapping + Morphing

Global variables: HashTable, Logic_Matrix
Mismatch_Tolerance_1($l \rightarrow x$, mapped_set)
//This subroutine replaces Mismatch_Check in Algorithm 1 whenever it is invoked

mm_set = Mismatch_Check($l \rightarrow x$, mapped_set) *//obtain mismatches of mapping* $l \rightarrow x$

1. if ($mm\_set == \emptyset$)

   return true *//no mismatch*

2. else if (HashTable(mm_set) has entry) *//hashtable hit*

   return HashTable(mm_set)

3. else, let $f = Logic\_Matrix$, construct f^* according to f and mm_set

   $mm\_tolerable$ = ***Logic_Eq_Check***(f, f^*)
   if ($mm\_tolerable$ == true)
   update Logic_Matrix with f^* *//logic morphing*
   add $mm\_tolerable$ into HashTable
   //update hashtable whether equivalent or not
   return $mm\_tolerable$

6.4 Simulation results

In this section, we examine the performance and cost of the proposed scheme:

- The performance is represented by yield, defined as the percentage of successful logic implementations over 10^4 defective crossbars, where defects are randomly distributed. In particular, we use the metric of *RunTime-Constrained (RTC)* yield, by setting a runtime upperbound for the process of finding a logic implementation.
- The cost is evaluated by the average runtime needed for obtaining a valid implementation.

The algorithms are implemented in Java on an Intel Core Duo 2.4GHz workstation with 2GB memory, and all the experiments are performed with the benchmarks in Table II.

6.4.1 RTC yield

In general, yield depends on many factors, including logic function size, defect rate, closed defect ratio, crossbar size, and runtime limit (47). We set the runtime limit to be 10 seconds for RTC yield comparison.

We first consider the case where crossbars are of the same size as the logic function. Figure 33 shows the yield over multiple defect rates on two benchmarks. The proposed scheme gains significantly higher yield over the mapping-alone scheme. Moreover, the improvement on yield is most significant when defect rate is high. This indicates the capability of finding more successful implementations in the "difficult region", due to the effect of morphing.

Closed defect ratio r also has significant impact on yield. Figure 34 shows the yield with two distinct closed defect ratios. When crossbars have mostly open defects (low r), yield is high. This means logic function implementation onto nanocrossbars can generally utilize more open defects than closed defects, for the reason that exclusion cells are always no less than inclusion cells in a logic function matrix.

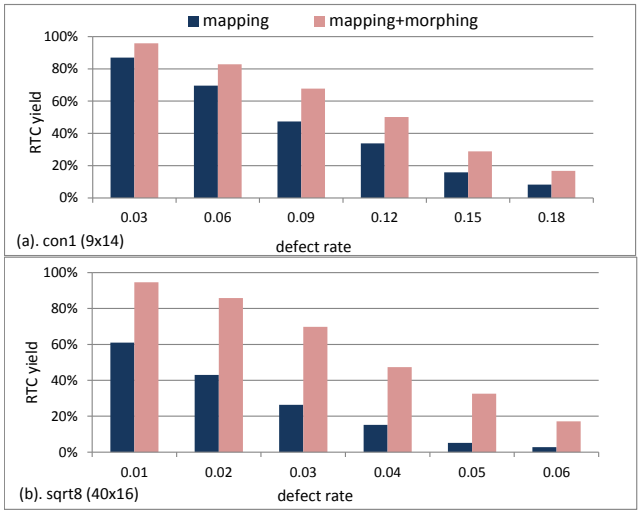

Figure 33. Yield comparison on (a) $con1$ and (b) $sqrt8$.

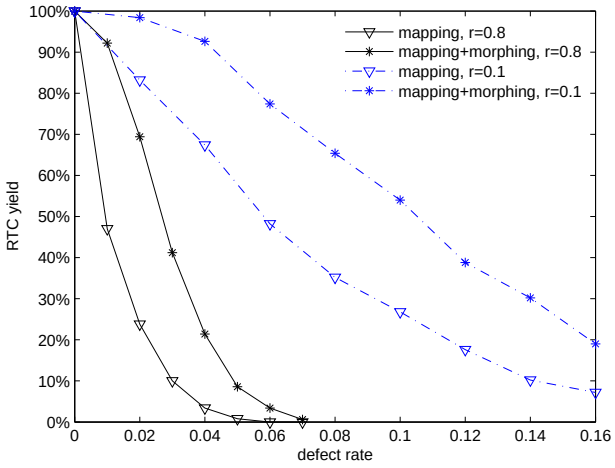

Figure 34. Yield comparison with different closed defect ratios

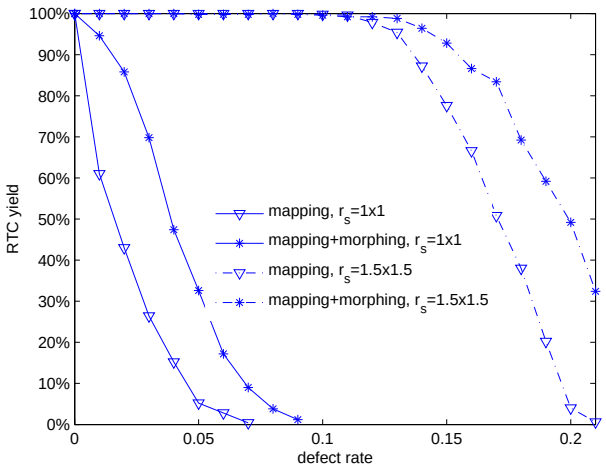

Figure 35. Yield improvement for various crossbar size

It has been shown that yield can be significantly improved when larger sized crossbars are used in a mapping based scheme. This is because adding hardware redundancy significantly increases the number of choices. Figure 35 shows yield of the same benchmark on two different-sized crossbars. We use size ratio r_s ((row ratio)$\times$(column ratio)) to indicate crossbar size in the simulation setup. Obviously, with larger crossbars of $r_s = 1.5 \times 1.5$, yield is higher for both schemes. This improvement mostly comes from the mapping dimension, when more available choices make it easier to find the most promising mappings. Nonetheless, the proposed scheme consistently outperforms the scheme of mapping alone, indicating the universal performance boost offered by exploring the morphing dimension.

The overall yield comparison over a set of benchmarks are shown in Figure 45. These data points are obtained at various defect rates with equal percentage of closed and open defects. In all the cases, the proposed scheme of mapping plus morphing outperforms the scheme of mapping only. Yield improve-

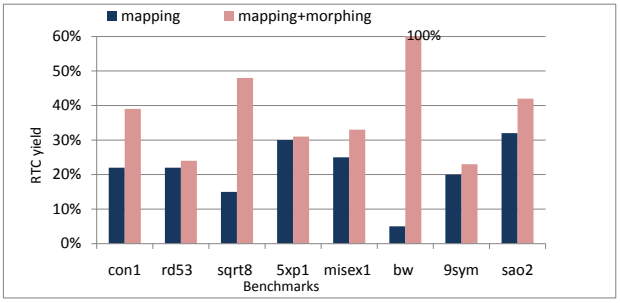

Figure 36. Yield improvement with morphing

ment varies across different benchmarks, some of which benefit significantly from morphing, because their logic forms can inherently tolerate more mismatches. For instance, the yield for benchmark bw achieves 100%, compared to the mapping-only solution of about 5% yield. In this benchmark, the percentage of tolerable single mismatch is as high as 10.92%, as is shown in Table II. In addition, these single mismatches turn out to be highly accumulative. Such characteristics make this benchmark benefit significantly from the morphing approach.

6.4.2 Runtime cost analysis

In this section, the runtime overhead of the proposed scheme is examined. We present the average runtime for the successful search, not counting the cases where the search gives up by hitting the preset runtime upperbound. Figure 37 shows the runtime for obtaining a valid implementation for both schemes, the yields of which are shown in Figure 33(a). It turns out that the average runtime for finding a successful logic implementation is basically the same for both approaches, and in many cases, the proposed scheme of mapping plus morphing actually takes less runtime. Therefore, the proposed

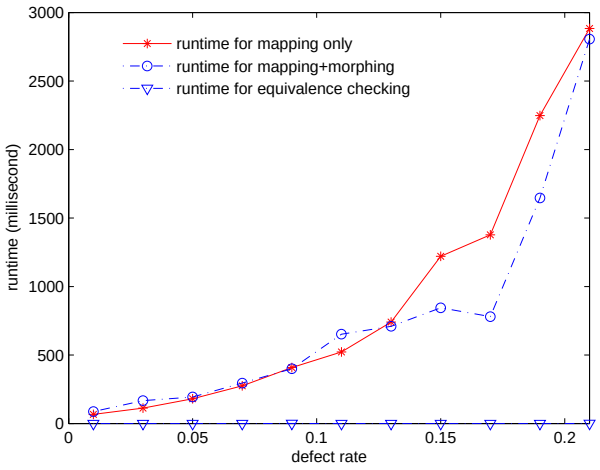

Figure 37. Runtime comparison for benchmark $con1$

scheme, incorporating both morphing and mapping, not only finds more solutions (as is indicated by the higher yield), but also finds them as quickly as (if not quicker than) the mapping-only scheme.

Figure 37 also shows the runtime overhead for logic equivalence checking, which is the curve that lies flat on the horizontal axis. Apparently, logic equivalence checking takes almost negligible time compared to the overall runtime cost. This is achieved through 1) exploiting the similarity in the logic forms, and 2) the efficient caching scheme of a hash table.

6.5 Summary

In this dissertation, we propose a new defect tolerance approach, namely logic morphing, by exploiting the various equivalent forms of a logic function. This approach explores a new dimension of freedom in achieving defect tolerance, and is compatible with the existing mapping-based approaches. We propose an integrated algorithmic framework, which employs both mapping and morphing simultaneously, and efficiently searches for a successful logic implementation in the combined solution space.

Simulation results show that the proposed scheme boosts defect tolerance capability significantly with many-fold yield improvement, while having no extra runtime over the existing approach of performing mapping alone.

CHAPTER 7

DEFECT-TOLERANT LOGIC HARDENING AND INTEGRATION WITH MAPPING AND MORPHING

7.1 Motivations

Nanotechnology can undoubtedly provide very high device density, and massive devices are available for a particular design. This allows redundancy-based designs for defect or fault tolerance purposes. One classic approach is based on N-Module Redundancy (NMR), which is too costly, and can compromise the benefits of high device density significantly. In fact, a fine-grained level of redundancy at a minimum cost can be achieved for defect tolerance purposes for nanocrossbar-base systems.

In nanocrossbar fabrication, it is generally acknowledged that defects which "short" horizontal and vertical nanowires are much less likely than defects caused by missing and misplaced switches (46). Therefore, it is expected that open defects are far more common than closed defects in the crossbar plane. As a result, open defects should be more necessarily targeted in defect tolerant schemes to achieve high yield and reduced runtime as well.

The aforementioned mapping and logic morphing techniques treat both open and closed defects indistinguishably in defect tolerance. Mapping techniques can utilize both types of defects as long as they do not lead to mismatches. In the same way, a morphing scheme changes the logic form to tolerate a defect of any type, when the defect leads to an equivalent function. Yet, it is revealed in the previous chapter that logic morphing tolerates mostly closed defects ($0 \rightarrow 1$ mismatches), for the reason that

logic functions are generally in a highly optimized form. Therefore, a technique to tolerate open defects becomes highly necessary to further boost the potential of nanocrossbars.

7.2 Hardening concept

When implementing a two level logic function onto a crossbar, open defects can cause $1 \rightarrow 0$ mismatches to the logic function. For instance, *f=ab+bc+ad+cd* may become a mismatched form *f'=~~ab~~+bc+ad+cd* because of an open defect. Yet, the logic function can be hardened for defect tolerance purposes. The logic function $f = ab + bc + ad + cd$ can be hardened as $F = abb_1 + bb_1c + ad + cd$, where b_1 is a redundant copy of the original variable b. Function F is more robust than f and can tolerate up to two $(1 \rightarrow 0)$ mismatches introduced to variable b. A hardened form containing mismatches *F'=~~ab~~*b_1*+~~bb~~*$_1$*c+ad+cd=*ab_1*+bc+ad+cd* is essentially equivalent as the original logic form f.

When more variables have a $1 \rightarrow 0$ mismatch, duplication can be applied to these mismatched variables to ensure an equivalent implementation. For instance, when two mismatched variables are present in *f'= ~~ab~~+bc+~~ad~~+cd*, then the hardened version can be *F'= ~~ab~~*b_1*+*bb_1c*+~~ad~~*d_1*+*cdd_1. When one extra copy still cannot satisfy the mismatch constraints, more copies can be added to a particular variable for defect tolerance purposes. For instance, *f'=~~abb~~*$_1$*+*bb_1c*+~~ad~~*d_1*+*cdd_1,which can be hardened as *F'=~~abb~~*$_1b_2$*+*bb_1b_2c*+~~ad~~*d_1*+*cdd_1.

In the process of mapping the logic function to a crossbar, a variable is implemented through a crossbar vertical wire, which intersects with every horizontal wire in the crossbar. So, if a variable is hardened for a particular product term, basically redundant vertical wires are used to implement the variable. Such a hardened implementation in the crossbar means that duplicated copies of the variable may show in all the product terms due to the intersection of every vertical wire and every horizontal

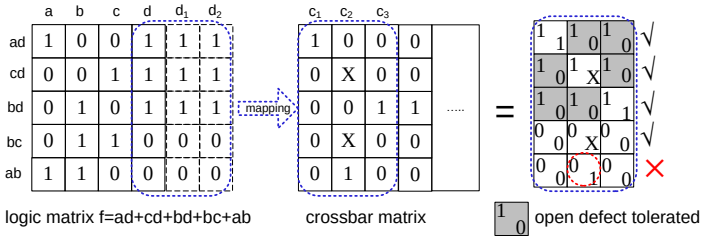

Figure 38. Logic hardening tolerates open defects and adds an extra $0 \rightarrow 1$ mismatch

wire. There are two scenarios, depending on whether the product term contains the hardened variable or not. First, it is obvious that duplicated copies do not have any negative effects for product terms containing the hardened variable. Second, for those product terms which do not contain the variable to be duplicated, extra copies may lead to extra $0 \rightarrow 1$ mismatches. For instance, in Figure 38, logic hardening through duplicating variable d tolerates a number of $1 \rightarrow 0$, and yet at the same time adds an extra $(0 \rightarrow 1)$ mismatch to the product term which does not contain variable d.

Thus, considering hardening technique, *hardening equality checking* needs to be performed throughout the whole mapping process to ensure that the hardened implementation of a logic function satisfies the following criteria.

- In the case that an inclusion cell (represented by 1) in a column is duplicated, as long as there is one out of these copies being mapped without a mismatch, then it is considered to be correctly mapped. In other words, hardening can allow many $1 \rightarrow 0$ mismatches to tolerate open defects
- In the case that an exclusion cell (represented by 0) in a column is duplicated, all copies have to be mapped without a mismatch. In other words, hardening cannot tolerate any closed defects, and add more constraints for mapping product excluding the hardened variable.

When crossbars contain mostly open defects, only inclusion cells in logic matrix can lead to mismatches $1 \rightarrow 0$. Hardening through variable duplication is helpful for tolerating $1 \rightarrow 0$ mismatches, and thus the most frequently appearing variables (with more inclusion cells in a column) tend to benefit most from the hardening. Therefore, as hardening heuristics, frequently appearing variables are chosen to be hardening with high priority to better tolerate defects.

7.3 Optimal hardening

Logic hardening through duplicating a variable is effective in tolerating open defects when mapping product terms containing the variable. Yet, hardening also adversely affects the mapping of product terms excluding the duplicated variable. Undoubtedly, it is hard to capture the benefits of hardening precisely, since the overall benefits are determined by the heuristic-based complicated mapping process, logic function structure, and crossbar size as well. In this section, we need to answer the fundamental question: how to harden a logic function in order to achieve the maximum defect tolerance capability.

We need to develop an assessment method to measure quantitatively the benefits brought by hardening, regardless of the specific mapping algorithms, the characteristics of the logic function and crossbar size. Such a benefit evaluation should provide the guidance for hardening a logic function effectively, in terms of which variables should be hardened and to what hardening degree they should be duplicated.

When a logic function is hardened with a resulting larger size, the searching path in the backtracking becomes longer and the corresponding solution space (containing all the possible mapping trials) grows also significantly. Yet, due to hardening, more mapping trials are turned into valid mappings - solutions. Thus, hardening has the disadvantage of increasing the solution space and the advantage of increasing solutions. In fact, what really matters essentially is *solution density*, which is the percentage of valid

mappings in the entire solution space. Regardless of the mapping algorithms, it is generally easy to find a valid mapping with a high solution density. Therefore, the key to evaluate the benefits of hardening, which in turn guides the hardening process, lies in identifying the solution density.

7.3.1 Solution density without hardening

Solution density can be analyzed in a probabilistic way, as shown in previous chapters. In crossbars, defects are assumed to be randomly distributed and each switch has a certain rate being defective. Given defect rate d, closed defect rate r, and a logic function with *logic inclusion ratio* l (the percentage of inclusion cells in a logic matrix) and size n (the total number of matrix cells), each mapping trial is valid with probability:

$$p = (1 - d(1 - r))^{nl}(1 - dr)^{n(1-l)} \tag{7.1}$$

where nl is the number of "1" cells and $n(1-l)$ is the number of "0" cells in the logic matrix. A cell in a crossbar matrix can be used to map an inclusion cell with probability $1 - d(1 - r)$, where $d(1 - r)$ is the probability for a cell to be an open defect. In the same way, a cell in a crossbar matrix can be used to map an exclusion cell with probability $1 - dr$, where dr is the probability for a cell to be a closed defect.

7.3.2 Solution density with hardening

Hardening degree k is defined as the number of columns in a logic matrix that are used to represent a single variable. The case of $k = 1$ basically mens no hardening. With a hardening degree k, k duplicated cells from the original cell in a logic matrix are mapped to k cells in a crossbar matrix.

- When an inclusion cell is duplicated with k copies, as long as there is one out of k copies being mapped without a mismatch, then it is considered to be correctly mapped. Thus, the probability that duplicating an inclusion cell with k copies can be successfully mapped is $1-(d(1-r))^k$.
- When an exclusion cell is duplicated with k copies, all k copies have to be mapped without a mismatch, and then it is considered to be correctly mapped. Thus, the probability that duplicating an exclusion cell with k copies can be successfully mapped is $(1-dr)^k$.

Having these two probabilities, the solution density associated with a hardened logic function of hardening degree k is:

$$p_k = (1-(d(1-r))^k)^{nl}((1-dr)^k)^{n(1-l)} \tag{7.2}$$

When $k=1$, $p_k = p$, which means no hardening is applied to the logic function. As long as hardening degree k goes beyond 1, $1-(d(1-r))^k$ is always greater than $1-d(1-r)$, indicating that hardening helps mapping inclusion cells. Yet, as k goes beyond 1, $(1-dr)^k$ becomes smaller than $1-dr$, which essentially means that hardening makes it even worse to map exclusion cells. Comparing $1-d(1-r)$ with $1-(d(1-r))^k$, the difference between them is decreasing with closed defect ratio r. In other words, hardening is more effective in tolerating open defects.

As hardening degree k grows, solution density p_k consists of two parts: the increasing part $1-(d(1-r))^k$ and the decreasing part $(1-dr)^k$. Obviously, when the exponent nl for the increasing part becomes larger or the exponent $n(1-l)$ for the decreasing part becomes smaller, p_k becomes increased.

In other words, when a logic function has a high logic ratio, leading to large nl and small $n(1-l)$, solution density p_k is more effectively improved.

7.3.3 Fine-grained optimal hardening

Solution density essentially is the probability indicating how likely to find a valid mapping. Solution density p_k with hardening technique has been shown previously when the whole logic function is considered. In fact, we can also calculate the probability for a column to be successfully mapped. Suppose the probabilities for all the columns in a logic matrix are p_{k1}, p_{k2},...,p_{kc}. Then the solution density for the whole function is $p_k = p_{k1} * p_{k2} * ... * p_{kc}$. In fact, the probability for each column being correctly mapped can be calculated in the same way as in Equation 7.2, if we know the number of cells in the column n_c, and the logic ratio of the column l_c. So, the Equation 7.2 for p_k also applies to calculating the success rate p_{kc} for mapping a column, when n and l are replaced with n_c and l_c, respectively.

Obviously, the objection of optimal hardening is to make the solution density p_k maximum so as to increase the chances to find a valid mapping. This indicates that we need to maximize the success probability for each column, p_{k1}, p_{k2},...,p_{kc}. So, for a particular column in a logic matrix, we can precisely calculate the hardening degree, given the column length, column logic ratio and crossbar defect rate and closed defect ratio. It is possible that the heavy columns, representing frequently appearing variables, should be duplicated to a large hardening degree, and there is no hardening required for sparse columns, representing non-frequent variables.

We analyze the specific benchmark $misex1$ from benchmark set (73) as a case study. The benchmark has 16 columns, and the length of each column is 32. Table III shows the logic inclusion ratio for all the columns. Suppose those columns are mapped onto a crossbar with closed defect ratio of

10% and defect rate of 20%, then the optimal hardening degrees for these columns can be developed, as shown in the Table III. It is obvious that when columns have logic ratio less than 10%, they must not be hardened (with degree of 1) since hardening only makes it worse. It is also found that the column with large logic ratio 62.5% can be hardened to the degree of 3 to produce the best performance. The figure in Table III shows the three specific logic inclusion ratios, which represent the optimal hardening degree being 1, 2 and 3, respectively. It is also shown that as logic inclusion ratio increases, the solution density improvement (p_k/p) becomes more significant.

7.4 Algorithmic framework exploiting mapping, morphing and hardening

Logic morphing and logic hardening are prominent when defect rate is high, and mismatch-free mappings become hard to find. However, the delivery of such potential advantages hinges on an integrated algorithm that efficiently explores the combined solution space of mapping, morphing and hardening efficiently.

In this section, we introduce an integrated algorithm to exploit mapping, morphing and hardening simultaneously. Backtracking-based logic mapping framework shown in Algorithm.1 serves as the basis to achieve a valid implementation. Morphing and hardening tolerate defects in different ways: one changes the logic function structure while the other adds redundancies. Both of them have different impacts on the mapping process.

Logic hardening technique is orthogonal to the two previous defect tolerant schemes, and in fact all of three techniques can be integrated together to achieve maximum benefits.

In the backtracking framework combining mapping and morphing, logic morphing is considered by performing LEC in every step, as long as a mismatch is encountered. To incorporate logic hardening, the

column	logic inclusion ratio	optimal hardening degree k	p_k/p
col_0	0%	1	1
$col_{1\sim3}$	3.13%	1	1
$col_{4\sim6}$	6.25%	1	1
col_7	9.38%	1	1
col_8	28.13%	2	2.78
col_9	31.25%	2	3.36
col_{10}	34.38%	2	4.04
col_{11}	40.63%	2	5.86
col_{12}	43.75%	2	7.05
col_{13}	50.00%	2	10.23
col_{14}	53.13%	2	12.34
col_{15}	62.50%	3	14.42

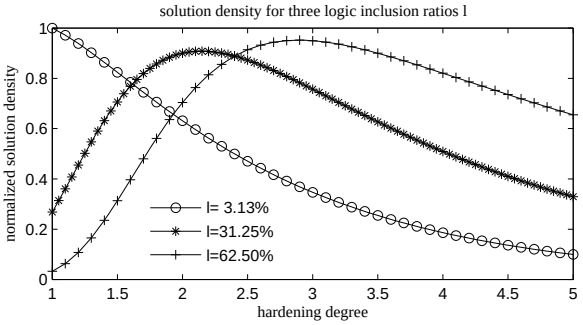

TABLE III

OPTIMAL HARDENING FOR $MISEX1$

difficult part lies in the integration of logic hardening and logic morphing, which essentially checks the logic equivalence between the original function without hardening and the hardened logic function containing mismatches. Such equivalence checking is hard, since the two logic functions contain different numbers of variable inputs.

We propose the following mechanisms to combine morphing and hardening in the backtracking mapping process 1. First, in the backtracking process, the logic morphing step should be performed only after all the redundant copies representing the same variable are mapped. This is because, if mismatches happen and are checked through LEC before all the redundant copies, the advantage of tolerating mismatches brought by hardening is not fully explored (since the mismatches can be tolerated through hardening). There are two implications: 1) all the copies representing the same variable need to be mapped in succession as one single mapping step; 2) there is no need to perform LEC in the midst of mapping redundant copies. When all the copies representing the same variable are already mapped, hardening equality check is performed first, and then followed by LEC when the mismatches cannot be tolerated through hardening.

Whenever one row (or duplicated columns) x from a crossbar matrix is mapped to the row (or multiple columns) l from a logic matrix, the validity of mismatches is checked (by $Mismatch\_Tolerance\_2$ in Algorithm.4) to see whether the mismatches can be tolerated. When a mismatch-free or equivalent mapping is found, the backtracking mapping algorithm returns success. When no such mapping exists, the algorithm eventually returns failure. In the process of mismatch checking, hardening equality checking (by $Harden\_Eq\_Check$) is performed first, and then LEC is performed (inside $Mismatch\_Tolerance\_1$) if mismatches cannot be tolerated through logic hardening.

7.5 Simulation results

7.5.1 Solution density improvement with hardening

Considering logic hardening, solution density p_k is affected by four factors: defect rate d, closed defect ratio r, hardening degree k, the logic function size n, and logic inclusion ratio l. The relationship

Algorithm 4 Mismatch Tolerance in Mapping + Morphing + Hardening

Global variables: HashTable, Logic_Matrix

Mismatch_Tolerance_2($l \rightarrow x$, mapped_set)

mm_set = Mismatch_Check($l \rightarrow x$, mapped_set) *//obtain mismatches of mapping* $l \rightarrow x$

1. if ($mm\_set == \emptyset$)

   return tolerable

2. else if (***Harden_Eq_Check****(mm_set, x)* == equivalent)

   return tolerable

3. else

   return ***Mismatch_Tolerance_1****(*$l \rightarrow x$*, mapped_set)*

Harden_Eq_Check(mm_set, x)

for every mm in mm_set $\{$

if($mm \in mismatch(1 \rightarrow 0)$)

if(x contains no 1 or X) *//mm cannot be tolerated*

return inequivalent

else *//mm is a* $0 \rightarrow 1$ *mismatch*

return inequivalent

$\}$

return equivalent

between p_k and defect rate d is clear, and p_k is a decreasing function of d. It is obvious that increasing defect rate will always make mapping process harder. So, for the following experiments on solution density, we fix the defect rate at 30%.

The first interesting part about logic hardening is to see what hardening degree should be adopted to effectively improve the success rate of finding a valid mapping. We will first explore the relationship between p_k and hardening degree k. In the paper, we experiment with logic function benchmarks to see the effectiveness of logic hardening. To see the solution density improvement in general, we set logic function size n to be 300, which is an average size, logic inclusion ratio to be 30%, which is also close to the average.

Four cases with different closed defect ratio r are examined, and the results are shown in Figure 39. We can see that when closed defect ratio is as low as 0%, increased hardening degree helps improve the solution density significantly. This essentially indicates that logic hardening can tolerate open defects effectively. Yet, even in this scenario, the hardening degree does not need to go beyond a certain value, since it finally saturates with diminishing returns. When closed defect ratio increases, the solution density can even become less when hardening degree increases beyond a certain value, as shown in cases of closed defect ratio being 10% and 20%. Thus, there is no need to invest a large hardening degree. Finally, as closed defect ratio increases, the hardening degree which leads to the maximum solution density becomes smaller. This essentially indicates that logic hardening is very effective to tolerate open defects.

Secondly, we examine the relationship between p_k and logic inclusion ratio r. The following settings are assumed as a general case: $n = 300$, $d = 20\%$, and $r = 20\%$. From the results shown in Figure 40,

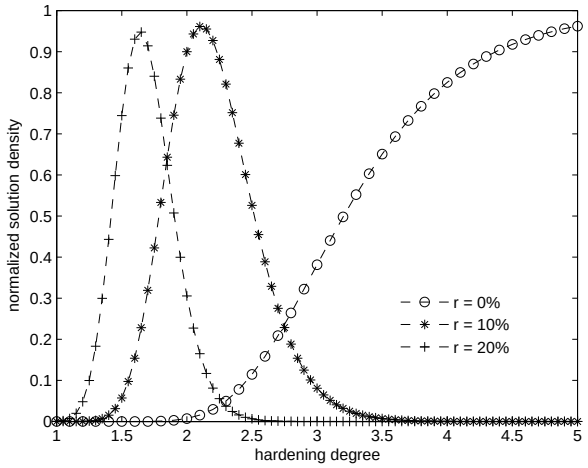

Figure 39. Hardening degree affects solution density with different closed defect ratio r

we can see that when the logic ratio is very low, the solution density for k being 2 or 3 is worse than that of no hardening ($k = 1$). But when the logic ratio is increasing beyond 0.2, the solution density in the case of hardening is greatly improved and far larger than that of no hardening.

Thirdly, we study how solution density is affected by closed defect ratio with logic hardening. The following settings are still assumed: $n = 300$, $d = 20\%$, and $l = 30\%$. We can see from results in Figure 41 that when the closed defect ratio is low, hardening brings about more benefits on solution density larger than without hardening. Yet, when closed defect ratio is larger than a certain value, which is roughly 0.25 in the setting, hardening just makes the solution density even worse. As a side-note, as closed defect ratio increases, both solution densities decrease significantly.

Fourthly, it is interesting also to see how solution density is improved with different logic function size. The following settings are still assumed: $d = 20\%$, $r = 20\%$ and $l = 30\%$. We can see from

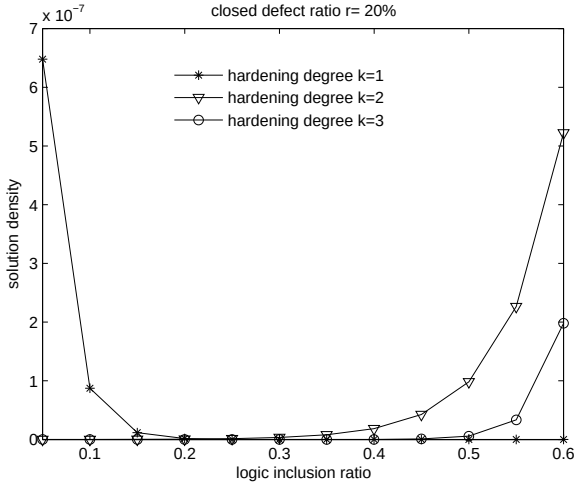

Figure 40. Logic inclusion ratio affects solution density with different hardening degree

results in Figure 41 that as logic function size increases, solution densities with hardening and without hardening decrease because large logic functions cause more constraints in the mapping process. Yet, as logic function size grows, the gap between hardening ($k = 2$) and non-hardening ($k = 1$) is significantly increased (note that solution density is in the logarithmic scale). This indicates that logic hardening technique can greatly improve the defect tolerance capability when large logic functions are mapped.

7.5.2 Yield improvement with logic hardening

In this section, we examine the performance of the proposed logic hardening scheme. The performance is represented by yield, defined as the percentage of successful logic implementations over 10^4 defective crossbars, where defects are randomly distributed. In particular, we use the metric of *RunTime-Constrained (RTC)* yield, by setting a runtime upperbound for the process of finding a logic implementation. In general, yield depends on many factors, including logic function size, defect rate,

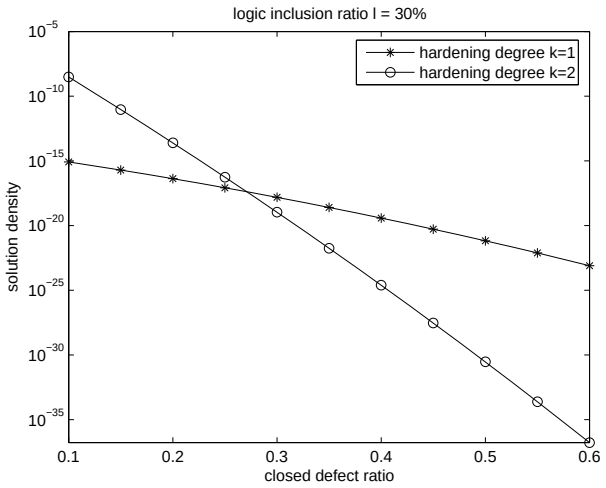

Figure 41. Closed defect ratio affects solution density with different hardening degree

close defect ratio, crossbar size, and runtime limit. The runtime limit is set to be 1 second for RTC yield comparison. Regarding crossbar size, we use size ratio (expressed as (row ratio)$\times$(column ratio)) to indicate crossbar hardware cost. The algorithms are implemented in Java on an Intel Core Duo 2.4GHz workstation with 2GB memory, and all the experiments are performed with the benchmarks.

We first illustrate how RTC yield can be improved by various hardening degrees. We expect that logic hardening can significantly improve yield when crossbars have mostly open defects with low closed defect ratio. Considering the logic hardening, crossbars of larger size are used for mapping. For a fair comparison, crossbars of the same size are used for both hardening and non-hardening cases. Figure 43(a) shows that RTC yield can be greatly improved as hardening degree increases, where closed defect ratio is 10%, and size ratio of crossbar versus logic function is 2×4. The following observations can be made based on the results.

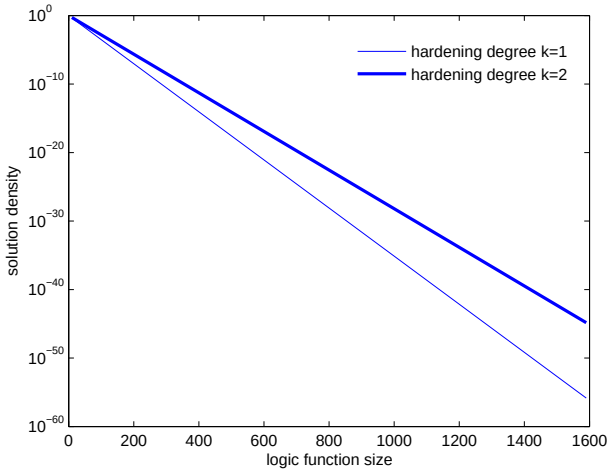

Figure 42. Solution density gap between hardening and nonhardening increases as logic function size grows

- As the hardening degree increases (below a certain value), the RTC yield is improved.
- Even though the yield is improved, the benefits get diminished as the hardening degree increases. For instance, yield improvement is more significant when the duplication level increases from one (no hardening) to two.
- When the level of duplications increases beyond a certain value, the RTC yield becomes even worse.

Figure 43(b) shows the corresponding solution density trend as the hardening degree increases. We can see that solution density is maximized when hardening degree is 3, which explains why yield becomes worse when hardening degree is increased to 4.

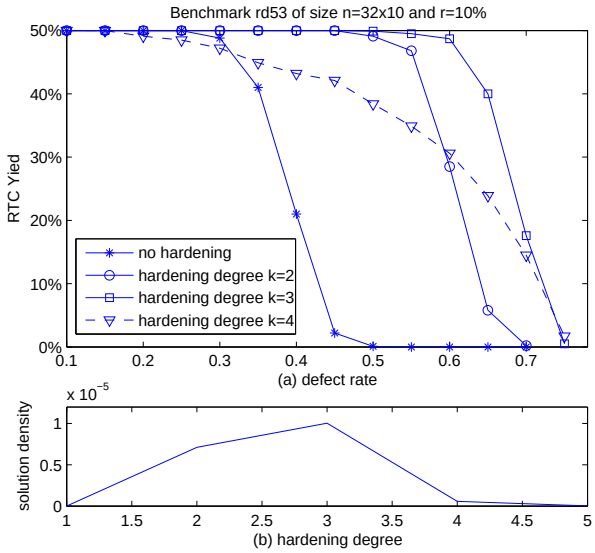

Figure 43. Yield improvements through logic hardening with different hardening degree $r = 10\%$

Second, we examine how a logic function can be effectively hardened tolerate defects by closely studying two benchmarks. We use the hardening heuristics, which chooses to harden frequently appearing variables, to see the effect of hardening on yield. The yield results of benchmark $misex1$ are shown in Figure 44, where the size ratio is 2×3, where frequently appearing variables are first chosen until all the variables are gradually selected.

As the first 4 or 8 frequent variables are hardened, the yield is greatly improved. When more other less frequent variables are hardened, the yield begins to become ever worse. We can see that hardening most frequent variables improves yield while hardening the non-frequent variables can even reduce the yield. As we can see from the case study in the previous section (Table III), only 8 variables need to be hardening so as to improve the yield at closed defect ratio of 10%. Besides, we can observe that yield

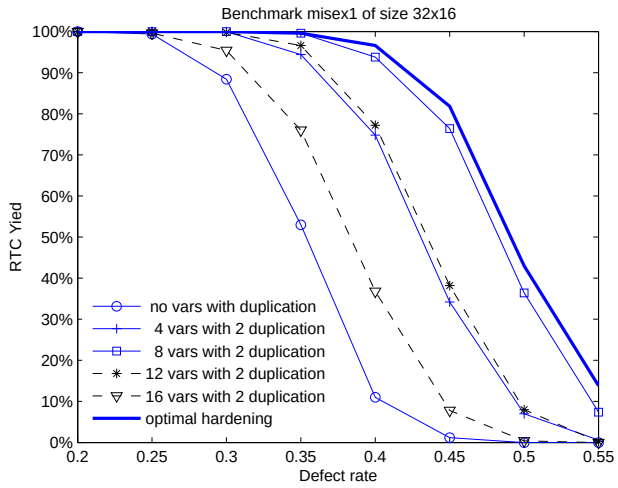

Figure 44. Yield improvements through logic hardening with different hardening degree and $r = 10\%$

curves with all kinds of hardening are bounded by the yield curve with optimal hardening of the average hardening degree 1.56 (the hardening degrees for all variables are: 3 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1).

So far, we have examined the logic hardening technique, and its significant impact on RTC yield on some benchmarks especially when defects in crossbars are open defects. Finally we examine the overall yield improvement using logic hardening over a set of benchmarks in Table IV at a relatively high closed defect ratio of 20% and with size ratio 2×3 of crossbar versus logic function. These data points shown in Figure 45 are obtained at various defect rates, and the last column in Table IV shows the average hardening degree with optimal hardening technique. In all the cases, the proposed scheme with hardening outperforms the schemes without hardening. Yet, different benchmarks have different yield improvements. From the perspective of a logic function, two factors come into play: logic function size and logic inclusion ratio. Benchmark $5xp1$ and $sao2$ have significant improvements

benchmark	size	logic inclusion ratio	optimal hardening degree
con1	126	18.3%	1.28
rd53	320	45.0%	2.20
sqrt8	640	20.9%	1.56
5xp1	1050	12.5%	1.71
misex1	512	26.7%	1.56
bw	870	16.6%	2.00
9sym	1566	8.6%	2.00
sao2	1160	11.3%	1.62

TABLE IV

BENCHMARK SIZE AND LOGIC INCLUSION RATIO.

on yield, mainly because these two benchmarks have very large size, compared to other benchmarks. Even though benchmark $9sym$ has a relatively large size, the yield improvement is still low due to very low logic inclusion ratio. Furthermore, benchmark $rd53$, having middle size, also has relatively significant yield improvement, mainly because of its high logic inclusion ratio.

Overall, RTC yield can be improved by many times with logic hardening at an average hardening degree of less 2. Even though more than one crossbar column are used for implementing a variable, the same amount of crossbar area is used for both the hardening scheme and the non-hardening scheme. It should be noted that RTC yield with non-hardening scheme is already much improved with hardware redundancy at size ratio of 2×3, compared to the case of no hardware redundancy at size ratio of 1×1. Thus, the actual hardware cost for both schemes is equal. So, logic hardening can greatly improve RTC yield when low-level redundancy can be provided at a minimum cost.

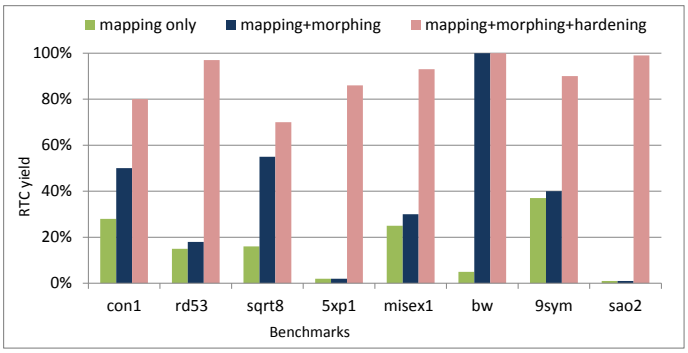

Figure 45. Yield improvements with hardening at $r = 20\%$

Overall, RTC yield can be improved by many times with logic hardening at an average hardening degree of less 2. Even though more than one crossbar columns are used for implementing a variable, the same amount of crossbar area (the same size ratio in Figure 45) is used for both the hardening scheme and the non-hardening schemes: 1) mapping only and 2) mapping and morphing combined. It is worth noticing that RTC yield with the mapping-only scheme is already much improved with hardware redundancy at size ratio of 2×3, compared to the case of no hardware redundancy at size ratio of 1×1. In other words, if crossbars of the same size as the logic function are used, the yield will be much lower than in Figure 45 for the mapping-only scheme. When the crossbar area is increased, the mapping-only scheme cannot take full advantage of the extra rows / columns. Yet, with logic hardening, these "previously-wasted" rows / columns in the mapping-only scheme can now be exploited.

7.5.3 Runtime cost analysis

In this section, the runtime overhead of the proposed schemes is examined. We present the average runtime for the successful search, not counting the cases where the search gives up by hitting the pre-set

runtime upperbound. Figure 46 shows the runtime for obtaining a valid implementation for three scenarios: 1) mapping only; 2) mapping with exploiting morphing; and 3) mapping with exploiting morphing and hardening. It turns out that the average runtime for finding a successful logic implementation is basically the same for scenario 1 and 2, and in many cases, the proposed scenario 2 actually takes less runtime. Therefore, the proposed morphing scheme not only finds more solutions (as is indicated by the higher yield), but also finds them as quickly as (if not quicker than) the mapping-only scheme. Figure 46 also shows the runtime overhead for logic equivalence checking, which is the curve that lies flat on the horizontal axis. Apparently, logic equivalence checking takes almost negligible time compared to the overall runtime cost. This is achieved through 1) exploiting the similarity in the logic forms, and 2) the efficient caching scheme of a hash table.

When logic hardening is exploited in the mapping process, runtime for finding a valid implementation is significantly reduced, as is shown in Figure 46. Logic hardening can quickly achieve a successful implementation, because it can tolerate many new mismatches, which cannot be tolerated through mapping and morphing. Besides, logic hardening does not incur any runtime overhead, since a logic function can be optimally hardened off-line, regardless of each individual crossbar defect pattern. Because of these two reasons, logic hardening can greatly reduce the runtime in achieving a successful implementation.

7.6 Summary

In this dissertation, we propose a new defect tolerant approach, namely fine-grained logic hardening, by adding calculated redundancy to a logic function to enhance the defect tolerance capabilities. An analytical framework is proposed to evaluate and fine-tune the amount of redundancy to be added to

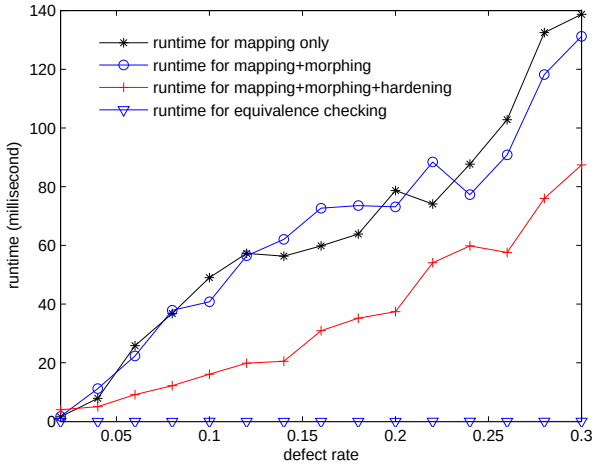

Figure 46. Runtime comparison for benchmark $con1$

a given logic function, and a method to optimally harden the logic function to maximize the mapping/-morphing yield. A holistic algorithmic framework is also developed to employ the three techniques and efficiently search for a valid solution in the combined solution space. Simulation results show that the proposed scheme boosts defect tolerance capability significantly with many-fold yield improvement without extra hardware cost, and also reduces the runtime.

CHAPTER 8

SUMMARY AND DIRECTIONS OF FUTURE WORK

The defect rate in self-assembly enabled nanotechnology is significantly higher than that of the lithography-based CMOS technology. Defect-tolerant logic implementation onto nanocrossbars becomes a new fundamental challenge in the post-fabrication design phase, and is in nature a hard problem. As a result, building reliable nanocrossbar-based systems press for new effective defect tolerance techniques and also new methodologies to model and evaluate the defect-tolerant logic implementation process.

8.1 Summary and conclusions

The conventional defect tolerance approaches for nanocrossbar architectures are solely based on the logic mapping technique, which rigidly tries to find a valid mapping between the logic function and the defective crossbar. However, even with various heuristic algorithms proposed, the runtime to find a valid mapping is still long and the yield for utilizing the nanocrossbars is low when the defect rate is high. In this dissertation, we have investigated the defect tolerant logic implementation problem in crossbar-based nanoelectronic systems from different perspectives. Essentially, the defect-tolerant logic implementation is to match the two parties, namely the logic function and the defective nanocrossbar, and the original functionality of the logic function can be represented in the defective crossbar through the implementation techniques.

In addition to the conventional perspective of strictly mapping the logic function and the crossbar without allowing any mismatches, two fundamentally different defect tolerance schemes can be explored when the defect-tolerant logic implementation problem in new perspectives. In this work, we have proposed and developed novel defect tolerance approaches, which are motivated from the perspectives of the logic function and the defective crossbar. Specially, these two categories of defect tolerance approaches are:

- *Logic morphing:* This technique has been proposed exploring the characteristics of a logic function to further improve the defect tolerance capabilities. In fact, the logic function can be represented by various equivalent forms, and all these forms can potentially be used for the final implementation. For defect tolerance purposes, one logic form can be chosen and it can tolerant most defects. Logic morphing is essentially the proposed technique to explore the right logic form to best tolerant the defects. Furthermore, in the work we have developed multiple heuristics especially tailored for expediting the logic morphing process targeting nanocrossbar logic implementation. Finally, the algorithmic work to integrate the logic morphing and mapping has also been developed to simultaneously exploit the two techniques.

- *Logic hardening:* From the perspective of redundancy in a crossbar, a logic function can be implemented via duplication, so that the redundant implementation can become more robust. Logic hardening has been motivated by redundant implementation, but it brings significant improvements to the traditional blind-duplication based hardening technique. In this technique, a logic function is analyzed quantitatively based on its structure and connectivity, and then hardening level for each variable or product is determined based on the criterion that such a hardening de-

gree has the highest probability for a successful implementation. Accordingly, the optimal logic hardening has been developed for a logic function, and such an optimally hardened logic function has a significant higher implementation success rate and results in great yield improvement.

The two above approaches are orthogonal to each other and can be exploited simultaneously. More importantly, they are also orthogonal to the existing logic mapping techniques. Compared to the incremental improvement to the logic mapping techniques by other research works, our work has proposed two fundamentally new approaches, which can significantly enhance the defect tolerance capabilities. Since the proposed techniques can be explored in combination with the existing techniques, we also developed a holistic algorithmic framework to exploit logic mapping with heuristics, logic morphing and logic hardening simultaneously.

Besides the proposed defect tolerance techniques, yield modeling has also been studies and a new yield modeling, which takes into consideration the runtime cost of logic implementation, has been proposed and analyzed. Both upperbound and lowerbound can be mathematically derived. Furthermore, we developed the methodologies to evaluate the logic design quality by modeling mismatch number distribution over all the implementation possibilities, in order to quantitatively understand the defect-tolerant logic implementation and identity the design tradeoffs. The mismatch number distribution reveals the probability that a valid logic implementation exits and identifies the cost for finding a valid implementation. It has been shown that the number of mismatches can be well modeled in probabilistic approaches, and the mismatch number distribution follows Normal/Poisson and Hypergeometric distribution, respectively.

8.2 Directions for future work

There are several possible directions to extend our work on the defect tolerance techniques of logic morphing and logic hardening.

8.2.1 Identification of optimal logic form: static logic morphing

We proposed logic morphing technique to change the logic form dynamically to tolerate the mismatches encountered over the course of logic mapping. At the end of the implementation process, the logic function is implemented in a different form. In this sense, the original logic form is not well positioned for defect tolerance purposes. In this dissertation, the proposed logic morphing technique began with any given form, and changed it dynamically as necessary as possible. One possible way to improve the logic morphing technique is to perform an offline study on the logic function, and change the logic function form so that it is optimally set for defect tolerance purposes. So, such a static logic morphing is aware of the defect tolerance capability. Some heuristics can include changing the logic inclusion ratio of a logic function according to the crossbar defect ratio, avoiding densely connected products or variables when the connectivity in the whole crossbar planes is sparse, and so on. With the identification of such a defect tolerance-aware logic form, the logic implementation process can begin with such an optimal form, which will inevitably reduce the steps of dynamic logic morphing, and result in reduced runtime.

8.2.2 Mismatch-directed logic synthesis

The proposed dynamic scheme always tries to satisfy the current defect constraints immediately by changing the logic form to counteract mismatches, without looking ahead at the further-unmapped part. In other words, the dynamic logic morphing technique treats all the mismatches indifferently, and it

is possible that one possible step of morphing to tolerate the immediate mismatch can result in a big sacrifice on the future logic implementation. The reason for this is not all the mismatches are equal in terms of their impact on finding a valid implementation. So, it is necessary to perform the logic morphing based on the evaluation of mismatches in terms of their benefits and costs. Some possible directions may include evaluating the impact of a mismatch on logic function structure, and also the impact of a mismatch on logic inclusion ratio. When a mismatch happens to a short product (containing less variables), it has large impact on logic function structure. Mismatches which can alter the logic inclusion ratio for matching the crossbar defect ratio are always chosen first. With such an evaluation, logic form is always changed in the better direction to obtaining the valid solution.

8.2.3 Defect pattern-aware logic hardening

The proposed logic morphing is statically to study the logic function and harden it to maximize the chance that the variables and products can be successfully mapped. This static process hardens the logic function optimally for a large number of crossbars sharing the same defect rate. Yet, such a hardening technique can possibly be further improved when each individual defect pattern of crossbars is taken into consideration. Basically, a logic function is hardened differentially for different crossbars, so that for each crossbar the logic function is hardened to the point where the defect tolerance capability is maximized. Over the course of hardening, Whether a particular variable or product will be hardened is determined by necessity, which requires to evaluate benefits and cost of each hardening step. One possible direction is to use the criteria developed by the static logic hardening to evaluate the benefits.

CITED LITERATURE

1. ITRS: International Technology Roadmap for Semiconductors, Emerging Research Devices. http://www.itrs.net, 2011.

2. Kamins, T., Williams, R., Chen, Y., Chang, Y. L., and Chang, Y.: Chemical Vapor Deposition of Si Nanowires Nucleated by TiSi2 Islands on Si. In Applied Physics Letters, 76:562–564, 2000.

3. Huang, Y., Duan, X., Cui, Y., Jauhon, L. J., Kim, K., and Lieber, C. M.: Logic Gates and Computation from Assembled Nanowire Building Blocks. Science, 294:1313–1317, Nov. 2001.

4. Islam, M. S., Sharma, S., Kamins, T. I., and Williams, R. S.: Ultrahigh-density Silicon Nanobridges Formed Between Two Vertical Silicon Surfaces. Nanotechonology, 15:L5–L8, 2004.

5. Kastner, M. A.: The Single-Electron Transistor. Review of Modern Physics, 64:849–858, 1992.

6. Likharev, K. K.: Single-Electron Devices and Their Applications. Proceedings of the IEEE, 87(4):606–632, 1999.

7. Lent, C. S., Tougaw, P. D., Porod, W., and Bernstein, G. H.: Quantum Cellular Automata. Nanotechnology, 4:49–57, 1993.

8. Snider, G. L., Orlov, A. O., Joshi, V., Joyce, R. A., Qi, H., Yadavalli, K. K., Bernstein, G. H., Fehlner, T. P., and Lent, C. S.: Electronic quantum-dot cellular automata. 2008 9th International Conference on SolidState and IntegratedCircuit Technology, page 549552, 2008.

9. Lent, C. S., Isaksen, B., and Lieberman, M.: Molecular Quantum-dot Cellular Automata. Journal of the American Chemical Society, 125(4):1056–1063, 2003.

10. Wolf, S. A., Awschalom, D. D., Buhrman, R. A., Daughton, J. M., von Molnar, S., Roukes, M. L., Chtchelkanova, A. Y., and Treger, D. M.: Spintronics: A Spin-based Electronics Eision for the Future. Science, 294:1488–1495, 2001.

11. Schmidt, G.: Spintronics in semiconductor nanostructures. Physica E: Low-dimensional Systems and Nanostructures, 25(2-3):150–159, 2004.

CITED LITERATURE (Continued)

12. Dietl, T., Ohno, H., and Matsukura, F.: Ferromagnetic Semiconductor Heterostructures for Spintronics. IEEE Transactions on Electron Devices, 54(5):945–954, 2007.

13. Mazumder, P., Kulkarni, S., Bhattacharya, M., Sun, J. P., and Haddad, G. I.: Digital Circuit Applications of Resonant Tunneling devices. Proceedings of the IEEE, 86(4):664–686, Apr. 1998.

14. Prost, W., Auer, U., Tegude, F. J., Pacha, C., Goser, K. F., Janssen, G., and Van Der Roer, T.: Manufacturability and Robust Design of Nanoelectronic Logic Circuits Based on Resonant Tunnelling Diodes. International Journal of Circuit Theory and Applications, 28(6):537–552, 2000.

15. Bergman, J. I., Chang, J., Joo, Y., Matinpour, B., Laskar, J., Jokerst, N. M., Brooke, M. A., Brar, B., and Beam, E.: RTD/CMOS Nanoelectronic Circuits: Thin-film InP-based Resonant Tunneling Diodes Integrated with CMOS Circuits. IEEE Electron Device Letters, 20(3):119–122, 1999.

16. Avouris, P., Appenzeller, J., Martel, R., and S.Wind: Carbon Nanotube Electronics. Proceedings of the IEEE, 91(11):1772–1784, 2003.

17. Avouris, P., Appenzeller, J., Martel, R., and Wind, S. J.: Carbon Nanotube Electronics. Proceedings of the IEEE, 91(11):1772–1784, 2003.

18. Avouris, P.: Carbon Nanotube Electronics and Photonics. Physics Today, 62(1):34, 2009.

19. Zhang, Z., Fu, Y., Li, B., Feng, G., Li, C., Fan, C., and He, L.: Self-Assembly-Based Structural DNA Nanotechnology. Current Organic Chemistry, 15(4):534–547, 2011.

20. Lee, Y. S.: Self-Assembly and Nanotechnology. John Wiley and Sons, Inc, 2008.

21. Ariga, K., Lee, M. V., Mori, T., Yu, X.-y., and Hill, J. P.: Two-dimensional nanoarchitectonics based on self-assembly. Advances in Colloid and Interface Science, 154(1-2):20–29, 2010.

22. Kamins, T. I. and Williams, R. S.: Trends in Nanotechnology: Self-Assembly and Defect Tolerance. MST News, pages 34–36, 2001.

23. Bachtold, A., Hadley, P., Nakanishi, T., and Dekker, C.: Logic Circuits with Carbon Nanotube Transistors. Science, 294:1317–1320, 2001.

CITED LITERATURE (Continued)

24. Butts, M., DeHon, A., and Goldstein, S. C.: Molecular Eletronics: Devices, Systems and Tools for Gigagate, Gigabit Chips. Int'l Conf. on Computer-Aided Design, pages 443–440, 2002.

25. Cui, Y. and Lieber, C. M.: Functional Nanoscale Electronics Devices Assembled Using Silicon Nanowire Building Blocks. Science, 291:851–853, 2001.

26. Beckett, P. and Jennings, A.: Towards Nanocomputer Architecture. Asia-Pacific Computer System Architecture Conference, pages 141–150, 2001.

27. Al-Yamani, A. A., Ramsundar, S., and Pradhan, D. K.: A Defect Tolerance Scheme for Nanotechnology Circuits. Fundamental Theory and Applications IEEE Transactions on Circuits and Systems, 54(11):2402–2409, 2007.

28. Ma, X., Strukov, D., Lee, J., and Likharev, K.: Afterlife for Silicon: CMOL Circuit Architectures. Proc. IEEE Conf. on Nanotechnology, pages 175–178, 2005.

29. Strukov, D. B. and Likharev, K. K.: CMOL FPGA: A Reconfigurable Architecture for Hybrid Digital Circuits with Two Terminal Nanodevices. Nanotechnology, 16:888–900, Apr. 2005.

30. Simsir, M. O., Cadambi, S., Ivanvcic, F., Roetteler, M., and Jha, N. K.: A Hybrid Nano-CMOS Architecture for Defect and Fault Tolerance. ACM Journal on Emerging Technologies in Computing Systems, 5(3):1–26, 2009.

31. Lee, B. H., Hwang, H. J., Cho, C. H., Lim, S. K., Lee, S. Y., and Hwang, H.: Nano-electromechanical Switch-CMOS Hybrid Technology and its Applications. Journal of Nanoscience and Nanotechnology, 11(1):256–261, 2011.

32. Bose, P. and Abraham, J.: Test Generation for Programmable Logic Arrays. 19th Design Automation Conference, pages 574 –580, June 1982.

33. Bushnell, M. L. and Agrawal, V. D.: Essentials of Electronic Testing for Digital, Memory, and Mixed-signal VLSI Circuits. 2000.

34. Goldstein, S. C. and Budiu, M.: NanoFabrics: Spatial Computing Using Molecular Electronics. Proceedings 28th Annual International Symposium on Computer Architecture, 00(June):178–189, 2001.

35. DeHon, A.: Array-Based Architecture for FET-Based, Nanoscale Electronics. IEEE Transactions on Nanotechnology, 2(1):109–162, 2003.

CITED LITERATURE (Continued)

36. Chen, Y., Jung, G.-Y., Ohlberg, D. A. A., Li, X., Stewart, D. R., Jeppesen, J. O., Nielsen, K. A., Stoddart, J. F., and Williams, R. S.: Nanoscale Molecular-Switch Crossbar Circuits. Nanotechnology, 14(4):462–468, 2003.

37. Heath, J. R.: A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology. Science, 280(5370):1716–1721, 1998.

38. Tahoori, M. B.: Defect Tolerance in Crossbar Array Nano-Architectures. Emerging Nanotechnologies: Test, Defect Tolerance, and Reliability, Springer, pages 121–151, 2007.

39. Tahoori, M. B.: Application-Independent Defect Toleranceof Reconfigurable Nanoarchitectures. ACM Journal on Emerging Technologies in Computing Systems, pages 197–218, 2006.

40. Tahoori, M. B.: Low-overhead Defect Tolerance in Crossbar Nanoarchitectures. ACM Journal on Emerging Technologies in Computing Systems, 5(2):1–24, 2009.

41. Tahoori, M. B.: BISM : Built-in Self Map for Hybrid Crossbar. Proceedings of the 19th ACM Great Lakes symposium, pages 153–156, 2009.

42. Tahoori, M. B.: BISM : Built-in Self Map for Crossbar Nano-Architectures. Workshop on Dependable and Secure Nanocomputing (DSN), 2008.

43. Rao, W., Orailoglu, A., and Karri, R.: Topology Aware Mapping of Logic Functions onto Nanowire-base Crossbar Architectures. IEEE/ACM Design Automation Conference, pages 723–726, July 2006.

44. Hogg, T. and Snider, G.: Defect-tolerant Logic with Nanoscale Crossbar Circuits. Journal of Electronic Testing, 23:117–129, Jun. 2007.

45. Hogg, T. and Snider, G. S.: Defect-tolerant Adder Circuits with Nanoscale Crossbars. IEEE Transactions on Nanotechnology, 5(2):97–100, 2006.

46. Naeimi, H. and DeHon, A.: A Greedy Algorithm for Tolerating Defective Crosspoints in NanoPLA Design. In Proceedings of Internal Conference on Field-Programmable Technology, pages 49–56, 2004.

47. Su, Y. and Rao, W.: Defect-tolerant Logic Mapping on Nanoscale Crossbar Architectures and Yield Analysis. IEEE International Symposium on Defect and Fault Tolerance (DFT) in VLSI Systems, pages 322–330, Oct. 2009.

CITED LITERATURE (Continued)

48. Su, Y. and Rao, W.: Runtime Analysis for Defect-tolerant Logic Mapping on Nanoscale Crossbar Architectures. IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), Jul. 2009.

49. Su, Y. and Rao, W.: On Mismatch Number Distribution of Nanocrossbar Logic Mapping. 2010 IEEE International Conference on Computer Design (ICCD), pages 132–137, Oct. 2010.

50. Huang, J., Tahoori, M., and Lombardi, F.: On the Defect Tolerance of Nano-scale Two-Dimensional Crossbars. In 19th IEEE International Symposiumon Defect and Fault Tolerance (DFT) in VLSI Systems, pages 96–104, 2004.

51. Rao, W., Orailoglu, A., and Karri, R.: Logic mapping in crossbar-based nanoarchitectures. IEEE Design Test of Computers, 26(1):68–77, 2009.

52. Crocker, M., Hu, S., and Niemier, M.: Defect Tolerance in QCA-Based PLAs. IEEE International Symposium on Nanoscale Architectures, pages 46–53, 2008.

53. Crocker, M., Hu, X. S., and Niemier, M.: Defects and faults in qca-based plas. ACM Journal on Emerging Technologies in Computing Systems, 5(2):1–27, 2009.

54. Rad, R. and Tehranipoor, M.: A Reconfiguration-based Defect Tolerance Method for Nanoscale Devices. 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006., pages 107 –118, 2006.

55. Zheng, Y. and Huang, C.: Defect-aware Logic Mapping for Nanowire-based Programmable Logic Arrays via Satisfiability. Design, Automation and Test in Europe (DATE), pages 1279–1283, Apr. 2009.

56. Zheng, Y. and Huang, C.: Fault-Tolerant Design for Nanowire-Based Programmable Logic Arrays. Lecture Notes in Electrical Engineering, 58:51–68, 2010.

57. Tunc, C. and Tahoori, M.: On-the-fly Variation Tolerant Mapping in Crossbar Nano-Architectures. VLSI Test Symposium (VTS), 2010 28th, pages 105 –110, 2010.

58. Tunc, C. and Tahoori, M.: Variation Tolerant Logic Mapping for Crossbar Array Nano Architectures. Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific, pages 855 –860, 2010.

59. Tunc, C.: Variation and Defect Tolerance for Nano Crossbars. Computer Engineering, 2010.

CITED LITERATURE (Continued)

60. Zamani, M. and Tahoori, M.: Variation-aware Logic Mapping for Crossbar Nano-architectures. Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific, pages 317–322, 2011.

61. Go andren, S., Ugurdag, H., and Palaz, O.: Defect-aware nanocrossbar logic mapping using bi-partite subgraph isomorphism and canonization. Test Symposium (ETS), 2010 15th IEEE European, page 246, may 2010.

62. Su, Y. and Rao, W.: Defect-Tolerant Logic Implementation onto Nanocrossbars by Exploiting Mapping and Morphing Simultaneously. International Conference on Computer Aided Design (ICCAD) 2011, pages 456–462, 2011.

63. Yuan, B. and Li, B.: Diversity Mapping Scheme for Defect and Fault Tolerance in Nanoelectronic Crossbar. Information Science and Technology (ICIST), 2011 International Conference on, pages 149 –154, march 2011.

64. Farazmand, N. and Tahoori, M.: Online Multiple Error Detection in Crossbar Nano-Architectures. IEEE International Conference on Computer Design 2009, pages 335 –342, 2009.

65. Rao, W., Orailoglu, A., and Karri, R.: Nanofabric Topologies and Reconfiguration Algorithms to Support Dynamically Adaptive Fault Tolerance. IEEE VLSI Test Symposium VTS, pages 214–221, 2006.

66. Rao, W., Orailoglu, A., and Karri, R.: Logic Level Fault Tolerance Approaches Targeting Nanoelectronics PLAs. Design Automation and Test in Europe DATE, pages 1–5, 2007.

67. Polian, I. and Rao, W.: Selective Hardening of NanoPLA Circuits. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT), pages 263–271, Oct. 2008.

68. Farazmand, N. and Tahoori, M. B.: Multiple Fault Diagnosis in Crossbar Nano-architectures. 2010 15th IEEE European Test Symposium ETS10, pages 94–99, 2010.

69. Han, J. and Jonker, P.: A Defect- and Fault-tolerant Architecture for Nanocomputers. Nanotechnology, 14(2):224–230, 2003.

70. Bahar, R., Mundy, J., and Chen, J.: A Probabilistic-based Design Methodology for Nanoscale Computation. International Conference on Computer Aided Design 2003, pages 480 – 486, 2003.

CITED LITERATURE (Continued)

71. Nepal, K., Bahar, R. I., Mundy, J. L., Patterson, W. R., and Zaslavsky, A.: Designing nanoscale logic circuits based on markov random fields. J. Electronic Testing, pages 255–266, 2007.

72. Prosser, P.: An Empirical Study of Phase Transitions in Binary Constraint Satisfaction Problems. Artificial Intelligence, 81(1-2):81–109, 1996.

73. North Carolina State University, D. o. C. S.: Collaborative Benchmarking Laboratory. 1993 LGSynth Benchmarks, 1993.

74. Shannon, C. E.: The Synthesis of Two-Terminal Switching Circuits. Bell System Technical Journal, 28:59–98, 1949.

VITA

NAME	Yehua Su
EDUCATION	Ph.D., Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, Illinois, December, 2012
	M.S., Electrical Engineering, Chinese Academy of Sciences, Beijing, China, July, 2007
	B.S., Electrical Engineering, Capital Normal University, Beijing, China, January, 2004
EXPERIENCE	Intern Researcher, Nvidia Corp. Santa Clara, CA, 05/2012 - 08/2012
	Intern Researcher, Motorola Corp. Arlington Heights, IL, 05/2011 - 08/2011
	Research Assistant, Dept. of ECE, University of Illinois at Chicago, 08/2007 - 05/2011
	Teaching Assistant, Dept. of ECE, University of Illinois at Chicago, 08/2007 - 05/2011
PUBLICATIONS	Y. Su and W. Rao: Defect-Tolerant Logic Implementation onto Nanocrossbars by Exploiting Mapping and Morphing Simultaneously. IEEE International Conference on Computer Aided Design (ICCAD), pages. 456-462, Nov. 2011.
	Y. Su and W. Rao: On Mismatch Number Distribution of Nanocrossbar Logic Mapping. IEEE International Conference on Computer Design (ICCD), pages. 132-137, Oct. 2010.
	Y. Su and W. Rao: Yield Modeling and Assessment for Nanocrossbar Systems. Invited paper. IEEE International Midwest Symposium on Circuits and Systems, pages 8-11, Aug. 2010.

VITA (Continued)

Y. Su and W. Rao: Runtime-constrained Yield Model in Nanocrossbar Systems. University Government Industry Micro/nano (UGIM) Symposium, Vol. 1, Jun. 2010.

Y. Su and W. Rao: Runtime Analysis for Defect-tolerant Logic Mapping on Nanoscale Crossbar Architectures. IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), pages 75-78, Jul. 2009.

Y. Su and W. Rao: Defect Tolerant Logic Mapping on Nanoscale Crossbar Architectures and Yield Analysis. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFTS), pages 322-330, Oct. 2009.