Reliable Integration of Terascale Systems with Nanoscale Devices

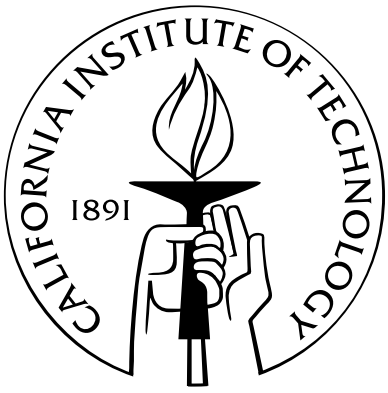
Thesis by

Helia Naeimi

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2008

(Submitted January 24, 2008)

Acknowledgements

I would like to thank my adviser, Professor André DeHon. He is a great person; one of those rare idealistic people that always believe in perfection. Learning more in the field of computer science was my first goal from coming to Caltech and working with a great scientist like André, and I have indeed learned many things from him and my other friends in IC group. However, I have learned many things from my adviser besides the technical matters. Specially during the unexpected challenge that our group experienced in the recent years.

I would like to thank my parents Bagher Naeimi and Azam Saatchi for helping me grow up free and boundless. They constantly encourage me to learn and grow. They thought me how to make my very first step and since then, they have supported me to make the steps, one little step at a time, in every aspects of my life specially my education. It was my mother who first thought me how to be a researcher, how to think freely , and her amazing power of thinking creatively has always been a great inspiration for me. I am truly indebt to them for having a vision to observe the nature and a mind to learn from her and respect her. My parents have a true passion for the nature. They always learn something new from the nature and respect her. They grow with any single bud and bloom in every single spring.

I would like to thank Amir Dana, my husband and my friend. who gives my life meaning and challenge. He has been a great support for me during my study at Caltech. From listening to many rounds of practice talks to bringing survival kits during long nights of deadlines to discussing a new idea or carrying heavy books, he was always there.

Abstract

Nanotechnology design has attracted considerable attention in recent years and seems to be the technology for the future generation of the electronic devices, either as scaled and more restricted conventional lithographic technology [1], or as emerging sublithographic technologies, such as nanowires, carbon nanotubes, NDR (Negative Differential Resistance) devices, or other nanotechnology devices. Each of these technologies provides one or more design benefits including feature-size scaling, high ON–OFF ratios, and faster devices. However, all of these techniques share their most challenging design issue: *reliability*. Providing reliability is becoming constantly more challenging due to increases in both the device failure rate and system complexity. This work develops techniques that make achieving reliability in such systems feasible with practical area overhead and considerable improvement in area overhead and system reliability compared to related techniques.

Conventional reliability techniques focus on low defect and fault rates, i.e., single event upset (SEU). These techniques cannot simply be scaled to larger systems with more unreliable devices. If these techniques are directly applied to the high defect and fault rate of the nanotechnology regime, they suffer impractically high overhead, or they may not achieve the desired reliability. Our approach in this thesis exploits the following design patterns to achieve a considerable area reduction compared to related works and achieve high reliability:

(1) *Fine-grained reliability*: In this technique, the system is partitioned into fine–grained blocks, and the reliability is provided for each block. This technique is used to contain the area overhead and bound the impact on the throughput.

(2) *Using alternative resources*: This technique improves the design quality by

sparing other resources when system is tight on one resource. In our work we replace some of the spacial redundancies with temporal redundancy to limit the area overhead. We further improve the system throughput to limit the throughput cost as well.

(3) *Defect pattern matching*: With this techniques, the defective resources are located and the design is reconfigured considering the defect pattern of the chip. Then the design configuration is mapped to the chip. This technique isolates the defective resources and make use of most of defect free resources.

(4) *Global reliability*: This technique is used to unify the reliability techniques used in different parts of the system. When using one unified technique to protect the system, the area overhead provided to protect one resource can be reused to protect other resources as well.

In the present work, we report considerable improvement in the area overhead using the above techniques. We show that using *Fine-Grained Reliability*, *Alternative Resources*, and *Defect Pattern Matching*, high permanent defect rates (e.g., 10%) which is the result of imperfect manufacturing can be tolerated with moderate area overhead (about 30% on average for typical designs). Again *Using Alternative Resources* and *Fine-Grained Reliability* improve the area overhead of the transient fault-tolerant designs by close to an order of magnitude compared to recent reliable works. Finally we report a fully reliable memory system that employs a *Global Reliability* scheme to tolerate permanent defects and transient faults, both in the memory and in the supporting logic and still achieves 100 Gbit/cm^2 density for fault rate of 10^{-18} errors per bit per cycle and 10% junction defect rate.

Contents

Chapter 1

Introduction

This thesis presents reliability techniques that make designing in sublithographic and nanometer scale practically feasible.

Considerable amount of research and work is devoted to continue feature size scaling and also invent new nanoscale electronic devices that can potentially replace the conventional lithpgraphic–based designs. Scaling the device feature size provides faster, denser, and consequently more powerful system that can run at higher speeds. In some of emerging technologies with high ON–OFF ratios it is also expected to bound power consumption by cutting off the leakage power. These improvements in area, performance, and power consumption, bring us technical challenges of their own. One of the main challenges (that is the subject of study in this work) 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. Providing reliability is becoming constantly more challenging due to increase in both the device failure rate and system complexity up to the point that the conventional techniques will not be efficient enough or even capable of tolerating these error rates and complexity for the future generation systems.

This thesis presents practical techniques with limited area overhead to achieve reliable systems. To implement such a reliable system we exploit these new design patterns for reliability:

- *Fine-grained reliability*: This technique is used to limit the area overhead and

bound the impact on the throughput. When the error rate is high, the system must be partitioned into fine-grained block size, where the errors strike in smaller number. Protecting blocks with few errors requires less area overhead.

- *Using alternative resources*: With this technique, we can shift some of the redundancy from area domain to time domain to limit the area overhead. The time redundancy can then be reduced with some improvement techniques shown in chapter 3 and chapter 5.

- *Defect pattern matching*: This technique is used to maximize resource utilization, and make use of almost all the defect free resources, isolating defective ones. In this technique the design configuration is restructured to match the defect pattern of the chip.

- *Global reliability*: When using one unified technique to protect the system, the area overhead provided to protect one resource can be reused to protect other resources as well. An instance of this technique, is using the same Error-Correcting Codes (ECCs) to protect against transient faults and permanent defects. With this technique the redundancy in the code will be used more efficiently to protect both errors. Another example is protecting the memory and its supporting logic (e.g., detector circuit), with single ECC.

Before going into the application of these design patterns in our reliable techniques, we review the sources of unreliability. The sources of failure are divided into two main categories, by the nature of the sources and their impacts on the system:

1. *Permanent defects*

2. *Transient faults*

Permanent defects: As the result of imperfect fabrication process, devices may have variation in shape and size. When the device structure is very different from the designed structure, the device will not perform as intended and will make a permanent defect. The probability that a node is defective is called the defect rate.

These defects could be of the form of a broken interconnect or junction because of lack of deposited molecules, or a too-high resistance interconnect due to lack of proper number of doping atoms, or a misplaced connection between devices because of extra molecule deposition.

In general the feature size scaling reduces control over the fabrication process which result in higher defect rates. Even for well-studied and largely manufactured conventional lithographic systems, the technology is facing reliability challenges. It is expected that Design Rules will become more restricted and require more regular design structure to bound the rate of manufacturing defects [2]. Furthermore, circuit designers can no longer design simply by technology design rules and expect a functional, let alone a scalable design. Designers must know when to use more relaxed rules and not simply relax the rules on the entire design, which negates physical scaling [3]. For emerging nano-technologies the defect rate is even higher due to the small feature size and bottom-up nature of the design. For example, imprint lithography, which provides one of the most reliable nanowire fabrication techniques, is reported to have 15% defective wires [4]. Although defect rate for emerging technologies is expected to decrease once the technologies are more mature, due to the nature of the fabrication process it is still expected to be high [5][4].

Transient faults: When a node in the system loses its effective charge due to ionized particle hit or various source of noises, it may cause the value of a node to be flipped in the circuit. However, the error does not permanently change the circuit, and it only generates a faulty bit value at the node that can last for one or few cycles. The transient fault rate is the probability that a single node loses its correct value during one clock cycle. Feature-size scaling, faster clock cycles and lower power designs increase the transient fault rate. Feature-size scaling and voltage level reduction shrinks the amount of critical charges holding logical state on each node; this in turn makes each node more susceptible to transient faults, e.g., an ionized particle strike has higher likelihood of being fatal as the critical charge is reduced in a node [6], which may cause a glitch or bit-flip. Furthermore operating at higher clock frequency increases the probability that a glitch in the signal is latched and

propagated as an erroneous value throughout the circuit.

In the rest of this chapter, we review our defect- and fault-tolerant techniques to protect nanotechnology systems against the sources of unreliability explained above.

1.1 Defect-Tolerant Approaches

Traditionally chips were tested and any defective chip would be discarded. However, as the device defect rate increases and the systems become larger, the probability of having a perfect chip will become unreasonably low and removing all chips with any defective node will dramatically decrease the system yield. Therefore, currently in large and regular systems (e.g., memory or PLA), the system is tested and part of the system that is defective is isolated and the rest of the chip will be functional. For example, in memory systems, a row or a column that contains a defective cell will be burnt-out and the rest of the system performs correctly [7][8]. Tolerating a few defective rows or columns in the system increases the system yield. However, as the defect rate increases the probability of having even a single defect free row will become very slim. For example, in a *NanoPLA* block that contains a 100×100 programmable devices and a device defect rate of 10%, the probability that a row of 100 devices is perfect is about $(0.90)^{100} \approx 2 \times 10^{-5}$, which makes it very unlikely that there will even be a single perfect row in a block (about 0.2%). Therefore, in order to keep the system yield high and make use of a defective chip with reasonable area cost, one must use a more *Fine–Grained* defect-tolerant technique, which tolerates defects at the device level. We have to make use of the defect free devices on even the nonperfect rows that also contain defective devices. Therefore, we use a *Fine-Grained Defect Pattern Matching* technique, which makes use of almost all the defect free devices in the system, although they belong to a defective part. In this technique the defect pattern of the chip is extracted and then the design configuration will be restructured to be matched to the defect pattern of the system. The system will be configured by this matching design configuration.

This technique requires postfabrication configurability which can be achieved with

various emerging technology devices, e.g., [2]catenane-based molecule [9], mechanical nanotube switch [10] or conventional programmable devices like SRAM-based reprogrammable switches, floating gate transistors, or fuses. The system also requires the appropriate hardware to test and locate defects in the chip.

Defect Pattern Matching is the main focus of chapter 4. In this chapter the defect pattern matching problem is modeled by a graph, and a matching algorithm with low complexity is proposed. Chapter 4 illustrates that the *Fine–Grained Matching* technique can tolerate 10% defect in the wires and another 10% defect in the programmable devices, in less than 3-fold area for the worst-case design. In contrast, techniques that are not based on fine–grained matching, require larger area overhead; e.g., Gate Multiplexing requires 100-fold area to tolerate up to a 3×10^{-3} device defect rate [11]. The matching technique is feasible since the defect pattern is static and fixed; and once the defect pattern is discovered, it can be used to configure the system.

chapter 3, *Cost of ignorance and cost of knowledge*, quantifies the benefits of exploiting the static defect map with methods like Fine–Grained Matching from area overhead perspective. It illustrates the overhead reduction achieved by exploiting this knowledge compared to the impact of ignoring it. We further introduce the technique to detect permanent defects in a system and discover the defect configuration pattern. The cost of defect pattern extraction is also presented and contrasted with the design costs associated with ignoring this knowledge.

1.2 Fault-Tolerant Approaches

Conventional fault-tolerant systems target Single Event Upsets (SEUs). However, due to the increase in the fault rate and system complexity this is not a valid assumption for future systems. For example, in a system with 10^{12} susceptible nodes and fault rate of $P_f = 10^{-7}$ per node per cycle, the system fails every cycle and the number of failures in the system is about $100,000$ errors per cycle in the average which is many more than single failure. Therefore, the traditional system-level SEU tolerant

techniques can not provide adequate reliability. When large number of errors occur in the system it is hard to detect the errors. To be able to detect the large number of errors in the system, one must focus on *Fine–Grained* block sizes, where errors occur in smaller number and can be detected more easily with less area overhead.

To detect errors in a block, the simplest way is to duplicate the block and compare the outputs, or to triplicate and perform the majority to actually correct the error. When using replication, the amount of replication factor grows super-linearly as the unit block size under detection and correction grows. Therefore, it seems that using *Fine–Grained* block size, minimizes the replication factor. Chapter 2 explains this fact in more detail. However, there is a constant area overhead associated with each block that grows with the number of blocks in the system. Therefore, there is an optimum block size that minimizes the total area cost (more detail will be shown in chapter 5). The replication technique is shown not to be the most efficient technique in communication coding theory. However, in the context of combinational logic it is not clear if other complex coding technique can outperform the simple replication scheme.

Furthermore, using only area redundancy to achieve the reliability may increase the area overhead dramatically. It is important to consider *Using Alternative Resources*, e.g., time redundancy, to bound the area overhead. One way of using time redundancy is to repeat the operation to generate a correct result, once an error is detected. The benefits of exploiting time redundancy is that we can only detect errors of a block in area domain and correct it in time domain which reduces the area overhead because the error detection circuit takes less area than error correction circuit (more detail will be shown in chapter 5).

In chapter 5 we suggest a *Fine–Grained Rollback* technique to tolerate high fault rates in a complex system. The *Rollback* technique is essentially detect-and-repeat; i.e., the outputs of the blocks are checked and once an error is detected in a block output, the operation of the block is repeated to generates the correct output. This technique exploits the *Fine–Granularity* error detection as well as *Using Alternative Resources* (time redundancy). In chapter 5 we show that the fault-tolerant system

protected with rollback technique will have a more compact, implementation (up to six times less area) compared to a *Feed-Forward* technique, which is mostly used in recent nanotechnology fault-tolerant design in the form of Gate-Multiplexing. *Feed-Forward* techniques correct potential errors with enough information redundancy without requiring recomputations. Furthermore, since in most of the cycles each block runs error free and errors are detected with low frequency, the impact of the rollback technique on the system performance is minimal.

As mentioned above, in our *Fine–Grained* Rollback design, we use replication to detect errors. The area overhead of this technique can be greatly improved by using a more efficient error detection technique. Error-correcting codes tend to be more efficient for error detection and correction of individual bits; as is the case for data transmission and data storage. However, ECC does not necessarily performs better than replication for protecting arbitrary combinational logic. A potential future work to this thesis would be to find a more efficient error detection and correction technique compared to replication, to further improve the area overhead of our *Fine–Grained Rollback* technique (chapter 8).

In fact, we have solved this problem for a subclass of combination logic circuits: encoder, corrector, and detector circuits [12]. Conventionally, only memory bits were protection against transient faults, however, as the combinational logic are becoming more susceptible to faults, the supporting logic of the memory system must also be protected against faults. Here we can use our fault-tolerant encoder, corrector, and detector circuit to satisfy this demand. In chapter 6, we define a new restriction on error-correcting codes, that guarantees *Fault-Secure* Detector circuit. The *Fault-Secure* detector can detect any error in the received code-vector, despite having faults in the detector circuitry. This is a breakthrough in fault-tolerant design, since the fault-tolerant capability of the detector is achieved by exploiting the structure of the design and using the redundancy already available in the circuit, compared to the traditional approach where the extra circuitry has to be added to the circuit under protection. In chapter 6, we also present a *Global Reliability* technique that protects memory cells and supporting logic all in one-shot, and also uses a unified technique

to tolerate permanent defect and transient fault together. We have shown codes that can tolerate fault rate of up to 10^{-18} faults/bit/cycle and defect rate of 1% and still achieve memory density of 100 Gbit/cm^2. Codes with higher error-correction capability can also be used to tolerate higher fault and defect rates.

All of the above techniques are general enough that they can be applied to any architecture and device substrate. In order to perform a detail analysis of the defect and fault-tolerant technique we analyzed the implementation of the above techniques on NanoPLA [13] and Nanomemory [14] architecture models. Chapter 2 reviews NanoPLA and Nanomemory architectures along with other nanotechnology based architecture models. This chapter also reviews some of the emerging technology devices and substrates.

In summary in this work we design a reliable system for nanotechnology designs, where the conventional techniques of tolerating SEU would not suffice due to high defect and fault rates and large system integration. We present approaches that achieve high reliability with practical area overhead for nanotechnology system.

Chapter 2

Background

Most of the defect- and fault-tolerant schemes for conventional designs assume Single Event Upset, SEU. This assumption is valid when the error rate is low enough that with very high probability only single error occurs in the system. However, due to the certain decline in the device reliability this assumption may not be valid for future designs. In this chapter, we review some sources of defect and fault and review some of the related works. We illustrate the reasons why we believe the future generations of the electronic systems must have a more robust defect- and fault-tolerant designs.

To analyze the details of our proposed reliable designs, we implement these techniques on two nanowire-based architectures, *NanoPLA* and *Nanomemory*. The second part of this chapter reviews these two architectures and the devices and substrate model that they are built on.

2.1 Reliability in Nanoscale Designs

Chemists have successfully shown fabricating nanoscale devices that are below 10 nm wide. At this scale devices are composed of only tens of atoms. We know that atoms and molecules have statistical behavior and we can only control the statistical behavior of these particles. When the devices are made of thousands of atoms and molecules, the shape and the behavior of the devices follow the statistical prediction, almost all the times. However, at the scale of devices with tens of atoms and molecules, the variation in the device shape and behavior will be more visible and at

higher rate.

The variation in shape will cause permanent defect, and the variation in behavior will cause noise and transient fault consequently. Sources of defect and fault will be explained in more detail in the following section.

2.1.1 Permanent Defects

The statistical structure of the devices will results in the following categories of defects in the system.

- Nanowires may break along their axis during assembly. The integrity of each nanowire depends on about 100 atoms in each radial cross-section and the lack of some atoms or atomic bonds may result in a break or high resistance nanowire.
- Nanowire to lithographic scale wire junctions depend on a small number of atomic scale bounds which are statistical in nature and subject to variation.
- Programmable device located between crossed nanowires will be composed of only tens of programmable molecules. The lack of functional molecules results in a high resistance device which is no longer programmable.
- Statistical doping of nanowires may lead to high variation among nanowires. If the doping location or density is sufficiently different from the designed device, a nanowire current may not be controlable or the wire may not conduct properly.

In the above list, we particularly gathered the defects that are related to nanowire-based architectures e.g., NanoPLA and Nanomemory. At this scale, we expect wires and devices to be defective in the $1\% - 10\%$ range.

However, since these technologies has not been mass produced and are still under active research, not many device defect rates has been reported. Some of the reported defect rates are given next. For example, an array of 250 nanowire fabrication is reported in [15], the nanowires has width of 100 nm and can be spaced 100 nm apart. The work in [15] reports 95% of the wires measured had good contacts. Another work,

reported fabrication of 6×6 nanowire crossbar with 40 nm wide nanowires [16]. They reported that 85% of cross-point junctions measured were usable. Both of these are early experiments and we expect the yield rates to improve. However, based on the physical phenomena involved, we anticipate the defect rates will be closer to the few percent range which is quite higher than the defect rates of conventional lithographic processing.

2.1.2 Transient Faults

A transient fault is an event that lasts for about one cycle. If a charge disturbance on a circuit node is smaller than the noise margin, the circuit will continue to operate properly. Otherwise, the disturbed voltage may be interpreted as the opposite logic state and the circuit will malfunction. A transient fault on any node has a finite probability of causing a glitch. There is a difference in the response of static circuits and actively clocked circuits. In a static circuits like memory, a glitch on a node may cause a bit flip. In actively clocked circuits, the glitch may propagate to an input of a sequential cell, get latched as a wrong value, and affect the machine operation. In precharged combinational circuits, a glitch on a node if happens after the precharge phase, gets latched and cause an erroneous signal that can propagate through the system.

However, many transient faults will not be latched. Some of the latched data may not be relevant to machine operation and there will be no perceivable error in the program operation. Hence, the effective error rate of a large combinational circuit needs to be derated. Three types of derating are applied to a typical circuit for calculating its error rate [17].

- Logical masking: If the transient fault strike happens on an input of a 2-input NAND gate, but one of the other inputs is 0, the strike will be completely masked and the output will be unchanged, i.e., this particle strike will not cause a soft error. In order for an error to propagate, there must be a sensitized path from the input to the output.

- Temporal masking: A glitch on a node may be outside the latching window of all the latches in the subsequent paths. Hence, the error will not be latched, and there will be no soft error.

- Electrical masking: The glitch pulse amplitude may reduce after passing through some logic stages, which may cause the glitch to attenuate. This phenomenon is called electrical masking.

In the fault-tolerant analysis in this work, we do the worst-case analysis and assume that any transient fault strike will result in a soft error. However, the real fault rate is less than the worst-case analysis. In order to achieve the exact analysis one must consider the detail of the circuit design and check for the above masking processes.

Many different sources can give rise to transient faults including: high-energy ionized particles impacts, thermal noise, and shot noise. In advanced VLSI systems feature size and voltage scaling lead to small node capacitance and voltage, resulting in decreased critical charge on nodes holding logical states. With fewer electrons representing states, each node in the system becomes more susceptible to charge disruption, that may be caused by any of the sources below.

2.1.2.1 High-Energy Particles

High-energy particles come from two sources

- High-energy alpha particles

- High-energy neutrons.

Alpha particles contains two protons and two neutrons. Alpha particles are emitted by various radioisotopes undergoing radioactive decay [18]. Metals such as lead that are used in packaging materials, emit low-energy alpha particles that can go as deep as 15 to 30 μm in silicon and are very effective in causing upsets in circuit. Alpha particles are also generated by elements such as some isotope of natural boron that is contained in the doping material. Boron-related upsets have been demonstrated in

DRAMs [19] and SRAMs [20]. It is shown that boron can result in as many as 81% of the SEUs in a 0.25 μm SRAM [21]. Alpha particles caused by radioactive impurities from packaging or doping materials remain an important source of errors in SRAMs and other sensitive circuits.

Charged particles, like alpha particles, create a direct ionization in semiconductor devices, causing a current surge that is responsible for errors in the memory and processing elements. However, high-energy neutrons do not have electrical charges; their effects occur through nuclear collisions that give rise to charged particles, which in turn cause ionized particles; thereby causing SEUs and the degradation of electrical properties. The probability of nuclear collisions occurrence is extremely low. It is reported in [22], that one out of 40,000 neutrons hits a silicon nucleus. However, once it hits the silicon nucleus it is very effective and will cause an error.

As mentioned above, not all the high-energy particle hits are fatal, but as the critical charge reduces the probability that a high-energy particle hit becomes fatal increases. It has been shown that the transient fault rates per chip induced by alpha particle increases 30 times as the manufacturing process goes from 0.25 μm to 0.18 μm and the supply voltage drops from 2 V to 1.6 V; at the same time the transient fault rates per chip due to the neutron's impact increases by 20% [6].

2.1.2.2 Shot Noise

Besides high-energy particles, noise is another source of electrical property disruption. Shot noise is one of the sources of noise that increases as we increase the clock frequency and further reduce supply voltage, and can become a significant source of transient faults. It is shown in [23] how to compute the error rate due to shot noise using Rice's generalized formula [24]. The error rate of a single transistor is

$$Bit\_Error\_Rate = \frac{2}{\sqrt{3}} f_c \exp\left(\frac{-0.09I}{f_c e}\right), \tag{2.1}$$

where f_c is the clock frequency, e is the electron charge, and I is the ON-state current of the device. This is with the assumption that the noise margin is 60% of the ON-

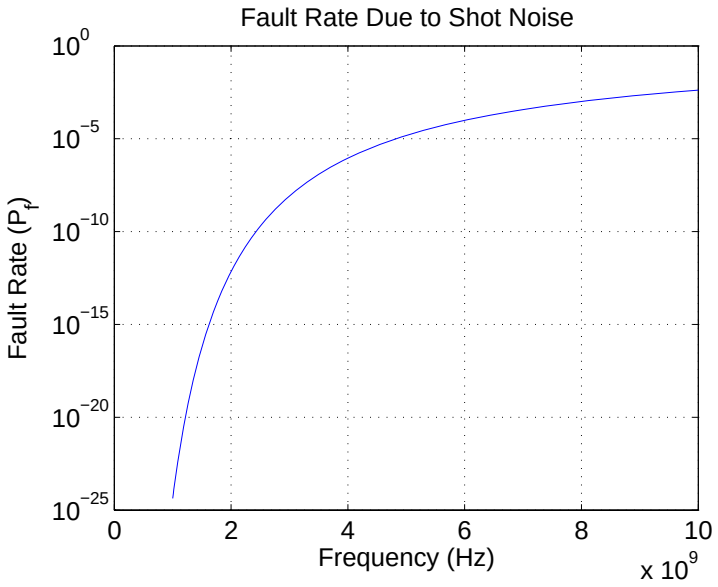


Figure 2.1. The device (e.g., transistor, nanowire) failure rate increases with the system frequency, for current per nanowire of 100 nA.

state current. Fault rate due to shot noise increases as the ON-state current decreases. Next we estimate the ON-state current to predict the device fault rate. The ON-state current is estimated using the maximum tolerable power dissipation in the system. Based on ITRS 2005, the maximum tolerable power dissipation in the system is around $250\,W/cm^2$ [1]. If the system works at 1 V, the maximum current consumed per area is $250\,A/cm^2$. Using the nanoPLA structure (section 2.3), the nanowire density of this structure is $\approx 240 \times 10^7\,/cm^2$. Therefore, each nanowire has a drive current around 100 nA. Using (2.1), we can estimate the device fault rate of such a system. Figure 2.1 shows how the fault rate grows as a function of system frequency when ON-state current equals 100 nA. For operating frequencies in the 1–5 GHz range, this model suggests that the individual device (e.g., transistor, nanowire) fault rates, P_f, are in the 10^{-20} to 10^{-5} range.

The effect of any of the above transient fault sources, ionized particle hit or noise, may ultimately cause a node in the logic circuit or memory unit loses its effective charge and consequently obtain a faulty value. This node can be a transistor or a precharged nanowire. To design a fault-tolerant system we assume the following general fault model for transient faults: A node (e.g., a transistor, or a precharged

nanowire) may lose its correct value and holds an erroneous value with random probability. The node holds an erroneous value for as many cycles before obtaining the new value. Further we assume the transient fault has identically independent distribution (iid).

2.1.3 Related Reliable Designs

Now that we know the sources of unreliability, we review some of the conventional reliable designs. We also present the reasons that these conventional reliable design may come short for high defect and fault rate nanotechology designs.

2.1.3.1 Defect-Tolerant Works

In conventional VLSI designs, a system undergoes a chip level test and all the non-perfect chips are discarded. However, as mentioned above, the defect rate of nanotechnology design is expected to be few percentages. With this defect rate almost every system have significant number of defective devices.

For example, assume a system with 10^{11} transistor and the defect rate of 1%. The expected number of defects in this system is 1 in every 100 nodes or 10^9 defective node in total, and with probability of 99%, the number of defective nodes is at least 10^8.

At this point most of the fault-tolerant techniques tolerate single error; a simple example is *Triple Modular Redundancy* (TMR) [25], which provides three copies of the system followed by a reliable voter. TMR technique guarantees correcting any single error and multiple errors, as long as they fall in one copy. The reliability of TMR technique can be improved by generalizing the number of copies to N. In N Modular Redundancy, NMR, technique, all defects that cause at most $\lceil (N-1)/2 \rceil$ erroneous copies are detected. The NMR technique, however, is not practical for the above example, where the system is expected to have 10^9 defective nodes. The system would need nine orders of magnitude area overhead!

To tolerate defects in this system with replication based techniques the system

must be partitioned into *Fine-grained* units, and NMR must be applied on each unit. For example, the above system can be partitioned into 10^{10} units each of size 10 nodes, where each node can be defective with 1% probability. In this case, each unit is defective with 9% probability,

$$P_{f\_unit} = 1 - (1 - P_f)^{10} = 1 - (1 - 0.01)^{10} = 0.09. \tag{2.2}$$

For the sake of simple presentation, we assume that the interconnects are reliable and the error does not accumulate in the system. These may not be practical assumption, however, even with this relaxed assumption the NMR results in impractically large area overhead. As mentioned, the probability of having a reliable system is the probability of having less than $\lfloor \frac{N-1}{2} \rfloor$ erroneous units.

$$\sum_{i=0}^{\lceil (N-1)/2 \rceil} \left(\begin{array}{c} N \\ i \end{array} \right) P_{f\_unit}^{i} (1 - P_{f\_unit})^{N-i} = 0.99.$$

Equating this probability to our target value, we can find the minimum required number of copies of $N = 45$.

The replication factor of 45 is not the final area overhead of the system. Implementing a reliable voter for each unit of size 10 nodes, require a considerable amount of area. The first approach to design a reliable voter would be to replicate it N times. Furthermore a circuit to take the majority of 45 signals is a huge circuit compared to the 10-node unit. In best case the voter grows linearly with the input size (N), so the voter area in total would grow as N^2. Therefore, we should expect seeing the final area overhead of this technique at about 2025 times the original area! This is huge area overhead, however, it is a considerable improvement from nine order of magnitude overhead of the system level NMR! In short, the above example, illustrates that NMR technique, even fine-grained NMR technique, will not be practical for nanotechnology defect-tolerant designs.

Another approach which uses the resources more economically compared to NMR, is *Sparing and Reconfiguration* technique. This technique is very popular for regular

structures like memory systems [7] [8]. In this technique the resources are overpopulated, and only the nondefective units are used. For example in the memory system some extra columns are considered, the columns are then tested, and those that contain defective cells will be isolated. So this technique can tolerate a few defective cells in the system. However, when the defect rate is high, it will be challenging to find even a single defect free column, let alone a large enough set of columns that can perform as a full memory system. For example in the memory system with columns of length 1000 cells, and the defect rate of 1%, the probability that a single column is defect free is 4×10^{-5}! So using *Sparing and Reconfiguration* technique alone will not provide a suitable defect-tolerant scheme for nanotechnology designs either.

However, we believe that we can develop a compact defect-tolerant approach for nanotechnology designs, and we present this in chapter 4. We use *Defect Pattern Matching* reliability design pattern, to achieve a reasonable area overhead for practical designs. In this technique the design of each unit is matched with the defect configuration of that unit, so we can still make use of defective units. The fact that make this possible is that permanent defects are statically located at the system. So if the defects are located, the system can program the design around the defects and still make use of defect free resources even in defective units. This technique is applicable for systems with regular structure and fixed configuration like Read Only Memories (ROM) and Programmable Logic Arrays (PLA), where the rows and columns configuration do not change during the operation. Once the defect pattern of the system is discovered, the row configurations of the system will be compared with the row defect pattern. Each configuration will be mapped to the first row with the compatible defect pattern, i.e., the configuration is mapped to the defect free nodes. Using this matching technique we show the most compact results compared to the related works; it can tolerate 10% defect rate with about 30% area overhead on average (chapter 4).

2.1.3.2 Fault-Tolerant Works

Transient fault-tolerant techniques, can be divided into to two classes: *Rollback* techniques and *Feed-Forward* techniques [26][27, 28].

Generally in *Rollback Recovery* techniques, errors are *detected* with *spatial* redundancy (e.g., a duplicated copy of the logic) and *corrected* with *temporal* redundancy (e.g., repeating the operation). The system runs at high speed when there are no errors, but when an error is detected, the system stops and repeats the affected operation to generate the correct result. *Rollback Recovery* schemes exploit the fact that most of the operation cycles pass with no error occurrence, and therefore, the recovery process occurs infrequently and the throughput impact is potentially low. In contrast, *Feed-Forward Recovery* schemes provide enough *spatial* redundancy in the system to *detect* and *correct* errors with no *temporal* redundancy. NMR is a feed-forward technique.

Similar to the discussion on the permanent defect tolerant scheme, the first requirement for our design is *Fine–Grained* reliability. One recent feed-forward approach that has received considerable amount of attention in nanotechnology community is *Majority-Multiplexing*, which provides reliability at the device–level. This technique was originally invented by von Neumann at 1956 [29], and there has been some improvement on the original technique recently [30][31][11]. This technique was the first fault-tolerant approach that specifically targets nanotechnology designs. We bring a brief review of this work in the following section.

This technique is suggested to tolerate errors, cause by permanent defects and transient faults [31][11]. However, the analysis provided for these scheme is more acceptable for permanent defects. The reliability goal of this system is set at 90% defect rate which is reasonable chip yield, but too low for a valid fault-tolerant target. It is shown in [30] that the device defect rate of 10^{-5} to 10^{-2} requires 100 to 1000, replication factor. More efficient Majority-Multiplexing is provided in [11]. However, they show great improvement for lower defect rates (e.g., 10^{-8}) and for higher defect rates (e.g., 10^{-4} <) it stays close to the original Majority-Multiplexing. Next section

shows the details of the Majority-Multiplexing technique and the replication factor required for this technique, which can go as far as 1000 for the defect rate of 0.01. Later in chapters 4 and 5 we show that we can design defect- and fault- tolerant techniques that are more efficient than Majority-Multiplexing.

2.1.3.3 Majority Multiplexing for Nanotechnology Designs

The common, fine-grained *Feed-Forward* fault-tolerant techniques for nanotechnology designs are based on *Multiplexing* the logic gates, which was originally developed by von Neumann as *Nand-Multiplexing* in 1956 [29]. In the *Multiplexing* technique, reliability is achieved by logic replication. Each bit is replicated M times and represented by the bundle of M wires. Computations are also replicated M times. Majority voting corrects errors in the logic. To prevent the voters from becoming a single point of failure, the voters are replicated as well. The trick is to make sure that a stage of computation and voting reduces the number of errors which exist in the bundle of wires which represent each bit.

For the multiplexing scheme, each processing unit (NAND gate) is replaced by replicated copies of the processing unit and voters. Each of the M wires of an input bundle has a separate and independent path through the multiplexed unit. A multiplexed unit consists of two stages, each using M processing units (NAND gate) (see figure 2.3). The first stage is the executive stage which performs the actual logic operation and generates replicated results of the logic (NAND function). The second stage is the restorative stage. The restorative stage performs the redundant voting on the output of the executive stage and is responsible for improving the output reliability. The executive stage is connected to the restorative stage through a randomized interconnect; this randomization improves the reliability of the design by guaranteeing errors arriving at the restoration stage are statistically independent (figure 2.3). In recent work [31], it is shown that *Majority* gates perform better than NAND gates, resulting in more compact fault-tolerant designs. All the devices in the first and second levels and the randomized interconnects fail with equal probability. The total area overhead of this design is lower-bounded by its replication factor. The replication

factor of this design is $2 \times M$. It is shown in [31] that majority multiplexing can be further optimized by sharing one restoration stage among multiple executive stages. Let L be the number of executive stages that share a restorative stage. The value of L, impacts the reliability of the system, and there is a lower bound on it based on the desired system reliability. For a system with M multiplexing factor and L executive stages for one restoration stage, the replication factor is $((L+1)/L) \times M$ [11]. We have to note that the total area overhead is larger than the replication factor when considering the wiring area required by the randomized interconnects, particularly when M is large.

This technique is suggested to tolerate both permanent defects and transient faults [32][11]. In the defect-tolerant application, this method is oblivious to the defect location map, and tolerates defect the same way that it may tolerate transient faults. Depending on the source of the failure (permanent defects, or transient faults), the system has different reliability target; i.e., for tolerating permanent defects a chip yield of around 90% is acceptable yield, while the expected failure probability for transient fault-tolerant application is around 10^{-18}, which makes the Failure In Time (FIT) of 360 for a typical system. *FIT* is an standard way of representing system reliability, and it is the number of failure in 10^9 hours of operation. A typical FIT of a commercial system is around couple of hundreds, so we believe that FIT=360, is the right target.

The area overhead of this technique is computed for both permanent defects and transient faults application [11]. The chip size, N_{total}, is assumed 10^{12} in [11]. In the same work the system is partitioned into units of size 10^6 nodes. They assume that the units have a logical depth $D = 10$. The graph in figure 2.3 plots the area overhead for two cases: (1) when tolerating permanent defects with target chip yield of 90%; (2) when tolerating transient faults with FIT of 360. We would expect that the fault rate would go as far as 10^{-7} at most and the defect rate would be above this value up to a few percent.

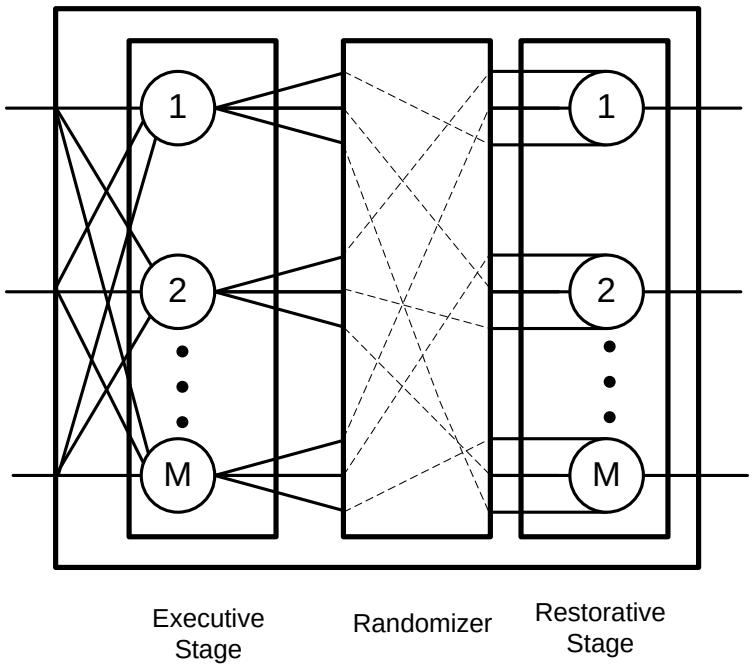


Figure 2.2. A reliable multiplexed unit to implement using von Neumann multiplexing technique

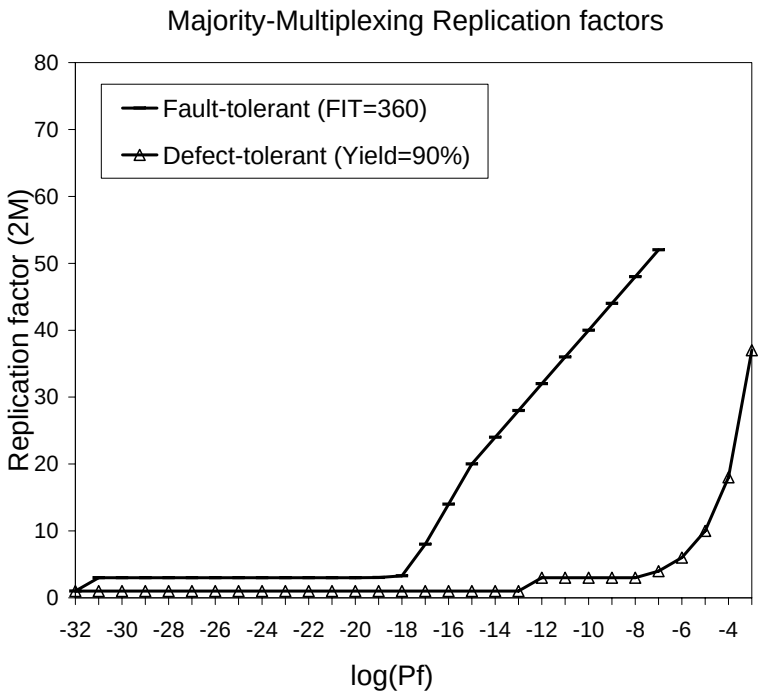


Figure 2.3. Replication factor of majority multiplexing for transient fault-tolerant (FIT=360) and defect-tolerant (yield=90%).

2.2 Substrates

In the rest of this chapter we review some of the promissing nanotechnology devices, and main building blocks. We then review the a computation (*NanoPLA* [33]) and memory (*Nanomemory* [14]) model built upon these devices. In the following chapters we show the detail implementation of our defect- and fault-tolerant techniques on these architectures.

2.2.1 Wires

This section reviews two of the most studied and developed interconnect wires using nanotechnology: *Nanowire* [34] and *Nanotube* [35]. Both of these wires can be semiconductor and metallic wires, and therefore, they can be both used as interconnect or as active devices, as will be shown later in this chapter.

2.2.1.1 Nanowires

Chemical Technologies have been developed to grow silicon and germanium nanowires [34, 36] which are only nanometers wide. These nanowires can be hundreds of micrometers long [37]. Atomic-scale nanowires can be fabricated in chemistry labs to have a variety of conduction properties (i.e., semiconductor or metal). Nanowires can be grown in the controlled environment using seed catalysts (e.g., gold particles). The size of the seed catalysts define the nanowires diameter. The catalyst constrains the growth of the semiconductor to only one dimension [34]. Nanowires with diameters down to 3 nm have been demonstrated [36, 34]. Seed catalysts with controlled diameter can be produced by self-limiting chemical processes (e.g., [38]). The composition of a nanowire can be varied along its axis and along its radius, which offers a single two-dimensional structure that works as an interconnect and a controllable device at the same time.

Langmuir-Blodgett (LB) flow techniques can be used to align a set of nanowires into a single orientation, close pack them, and transfer them onto a chip surface [15][39]. The resulting wires are all parallel with nematic alignment. By using wires with an

oxide sheath around the conducting core, the wires can be packed tightly. The oxide sheath defines the spacing between conductors and can, optionally, be etched away after assembly. The LB step can be rotated and repeated so that we get multiple layers of nanowires [15][39] such as crossed nanowires for building a crossbar array or memory core (section 2.3.1).

The other successful fabrication technique is imprint lithography. Nanowires with sub-10 nm feature size can be made using imprint lithography [40]. This new technique has high throughput and low cost. Imprint lithography includes little damages to sensitive circuit components, including active molecules, which are used in making programmable cross-points (section 2.2.3). Chen et al. have developed an inexpensive process to fabricate nanoscale devices and circuits utilizing imprint lithography, shown in [16]. This technique for fabricating aligned metal nanowires through a one-step deposition process without subsequent etching or lift-off is demonstrated in [41]. Their technique uses Molecular Beam Epitaxy (MBE) to create physical template for nanowire patterning. The template is a selectively etched GaAs/AlGaAs superlattice. The wires are defined by evaporating metal directly onto the GaAs layers of the superlattice after selective removal of the AlGaAs to create voids between the GaAs layers. By depositing the metal solely on the GaAs layers, the wire width is defined by the thickness of the GaAs layers and the separation width by AlGaAs layers. Transfer of the metal nanowires to a silicon wafer is performed by contacting the metal-coated template to a silicon oxide surface with subsequent heating process. Wires deposited with this technique were uniform and continuous over 2 to 3mm length, with very few defects.

2.2.1.2 Nanotubes

Carbon nanotubes that are nanometers in diameter and micrometers long can be fabricated in chemistry labs [42]. Carbon nanotubes can also be grown from seed catalysts with diameters down to roughly 1 nm in diameter and microns long. They can be semiconducting, allowing field-effect control, or metallic, perhaps offering superior electrical properties to silicon nanowires or even copper. To date, we cannot control

the conducting properties of nanotubes; i.e., metallic and semiconductor nanotubes cannot be differentiated during the growth process. Unlike nanowires, nanotubes are not rigid, and consequently it is more challenging to align carbon nanotubes into straight, parallel arrays. Some techniques are being developed (e.g., [35]) and more techniques are under research.

2.2.2 Field-Effect Controllable Cross-Point

By controlling the density of doping material in the environment during growth, semiconducting nanowires can be doped to control their electrical properties [34]. Heavily doped nanowires always conduct, while conduction through lightly doped nanowires can be controlled via an electrical field similar to Field-Effect Transistors (FETs) [15]. OFF resistances can be over 10 GΩ and ON resistances under 0.1 MΩ, OFF/ON resistance ratios are at least 10^4 [43]. A nanowire field-effect gating has sufficient gain to build restoring gates [44]. The threshold voltage for the nanowires can be controlled by material properties (e.g., doping or composition) and geometry factors.

By changing the density of doping material in the environment during growth, a doping profile can be made along one nanowire, e.g., in a highly doped nanowire that can always conduct only one region can be lightly doped to control the conduction. The differentiated doping profile gives the ability to selectively control the conduction of one nanowire among other nanowires in an array (section 2.3.2).

2.2.3 Programmable Cross-Point

Chen et al. demonstrate a nanoscale Pt-rotaxane-Ti/Pt sandwich which exhibits hysteresis and nonvolatile state storage showing an order of magnitude resistance difference between ON and OFF states for several write cycles [4]. After an initial "burn"-in step, which permanently reduces the the high resistance of > 100 MΩ to 9 MΩ, the state of these devices can be switched at ± 2 V and read at ± 0.2 V. [4] reports a 40 nm×40 nm junctions, with the ON resistance of roughly 500 kΩ

, and the OFF resistance of 9 MΩ . The exact nature of the physical phenomena involved is the subject of active investigation. The basic hysteretic molecular memory effect is not unique to the rotaxane. Many technologies have been demonstrated for nonvolatile, switched cross-points, where the common features include: (1) resistance which changes significantly between ON and OFF states; (2) the ability to turn the device ON or OFF by applying a voltage differential across the junction; (3) the ability to be placed within the area of a crossed nanowire junction.

LB techniques can also be used to place the switchable molecules between crossed nanowires (e.g., [9]). The molecules are formed into a single monolayer in an LB trough and then transferred onto a set of parallel nanowires [45]. An orthogonal set of nanowires is then transferred on top creating the conductor-device-conductor sandwich for the cross-point array. The 8×8 molecular crossbar was constructed using this approach [4].

This programable cross-point is comparable with a SRAM-based programmable switch for reconfigurable systems (e.g., FPGA). However, a typical SRAM-based switch might take about 2500 nm^2 compared to a 5 nm×5 nm bottom level metal wire crossing of molecular switches. Consequently, the molecular cross-points offers 100 times smaller switches.

2.3 NanoPLA

In this section we review the building blocks that can be constructed using the techniques in the previous section. These building blocks are useful in developing many system architecture including *NanoPLA*, which will be reviewed later in this section.

2.3.1 Programmable Crossbar Array

Nanowires can be fabricated in tight-pitched parallel arrays, but we cannot fabricate arbitrary geometries with equally tight conductor and device pitches. Assembly process allows crossed nanowires array the switchable resistance from section 2.2.3 in between (see figure 2.4). The effective diode in the junction comes from the crossing

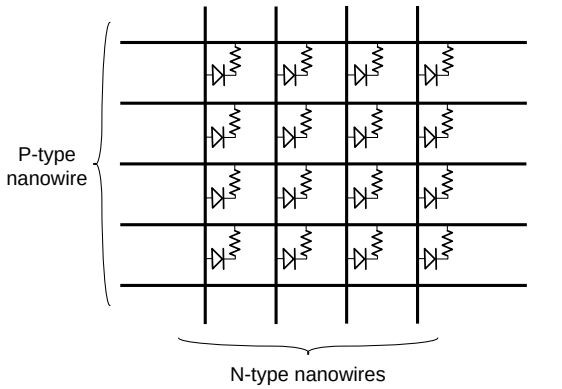


Figure 2.4. Programmable nanowire crossbar

nanowires doping. A low resistance junction between an N-typed nanowire and a P-type nanowire works as a diode from the first nanowire to the second one. This nanowire crossbar with programmable junctions is the core of many nanotechnology architectures, including nanoPLA and nanoMemory. This crossbar can be used as memory cores, programmable logic cores, and programmable crossbar interconnect arrays. When crossbar is used as logic core, each horizontal nanowire implements the wire-OR logic function of the selected vertical nanowires with turned ON switches. When the crossbar is used as memory core, each junction stores single memory bit and it can be accessed by putting the appropriate voltage on both nanowires passing the junction. When the crossbar is used as interconnect array, each junction is programmed to route the incoming signals in vertical nanowires to horizontal nanowires.

2.3.2 Restoration and Inversion Array

As noted in section 2.3.1, the programmable, wired-OR logic is passive and nonrestoring, drawing current from the input. Further, OR logic is not universal. To generate any arbitrary combinational logic, we need universal building gate and further we need to restore signals to maintain the appropriate voltage level in multilevel circuit implementation. We can achieve all of the above with a restorable inverter integrated with the OR gates, which generate a universal NOR gate. As developed in section 2.2.2, nanowires can be field-effect controlled. This gives us the potential to build FET-like

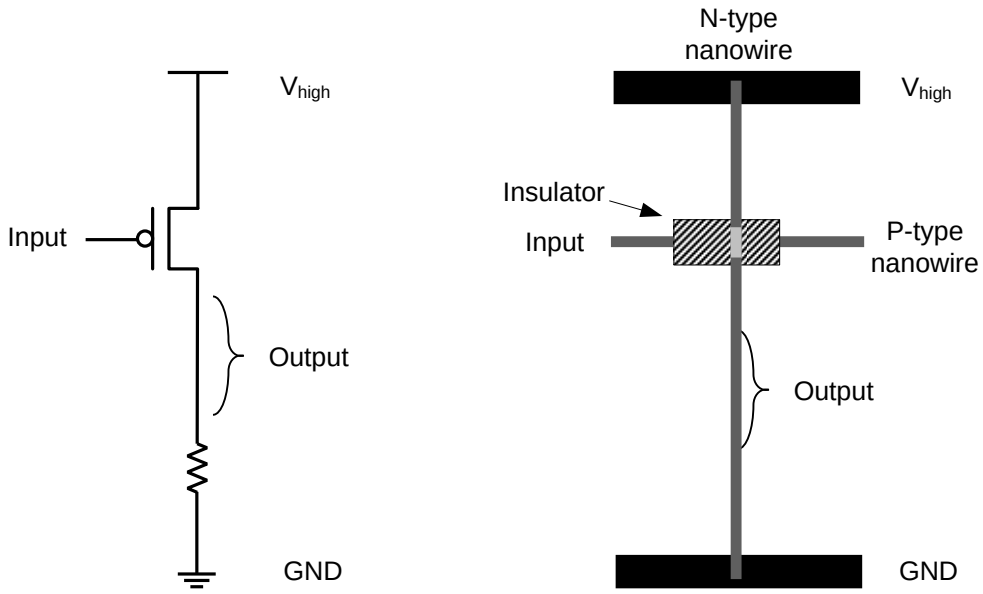


Figure 2.5. Structure of nanowire inverter

gates for restoration. Using a highly doped nanowire with a lightly doped region as shown in figure 2.5 can provide a gatable junction, where the electrical voltage on the horizontal nanowire, passing through lightly doped region, controls the current on the vertical voltage. If the vertical nanowire is P-type doped and the V_{high} and GND signals are connected as figure 2.5 then the inverted and restored value of the horizontal nanowire is transferred into the vertical nanowire. If the location of V_{high} and GND are swapped the vertical nanowire only holds the restored value with no inversion.

These restoration gates can be closed packed to generate a nanowire restoration inverter crossbar as in figure 2.6. In ideal case, each vertical (output) nanowires has exactly one controllable lightly doped region that can be controlled by exactly one horizontal nanowire. However, due to limited control over alignment of nanowires the nanowires with different doping profile is stochastically located. Furthermore, to generate a nanowire set with different doping profiles, each nanowire has to be selected from a bach of grown nanowires with a unique doping profiles. However, with our nanowire assembly technique (section 2.2.1.1) this is practically impossible.

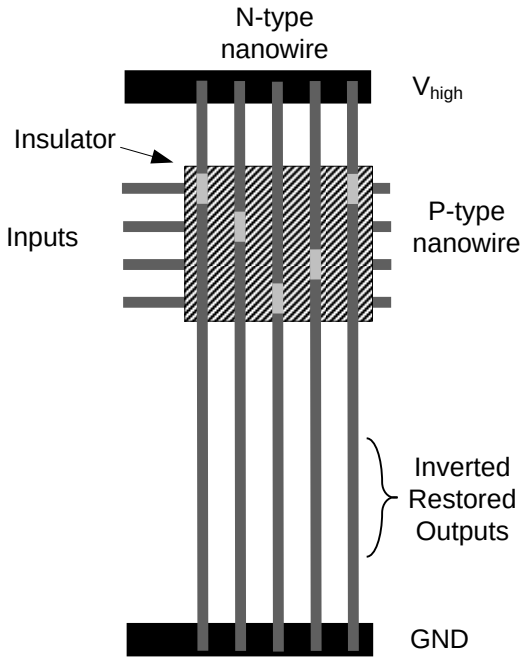


Figure 2.6. Restoration crossbar with nanowire

Instead we can overpopulate the vertical nanowires with various doping profile to make sure that there exists enough correctly aligned and uniquely controlled vertical nanowires (figure 2.5).

2.3.3 Lithographic to Sublithographic Decoder

An important challenge is to access nanowires from lithographic scale wires and be able to programm nanowire junctions by applying appropriate differential voltage on the nanowires of each junction. Furthermore, we must be able to drive and sense each nanowire to read back the status of each junction (e.g., in memory cores). A decoder structure from lithographic scale wires to nanowires proposed in [44] provide these functionalities.

To interface with lithographic-scale wires, address bit regions are marked off at the lithographic pitch. Each such region is then either doped heavily so that it is oblivious to the field applied by a crossed lithographic-scale wire or is doped lightly so that it can be controlled by a crossed lithographic scale wire (see figure 2.7). In this way, the nanowires will only conduct if all of the lithographic-scale wires crossing

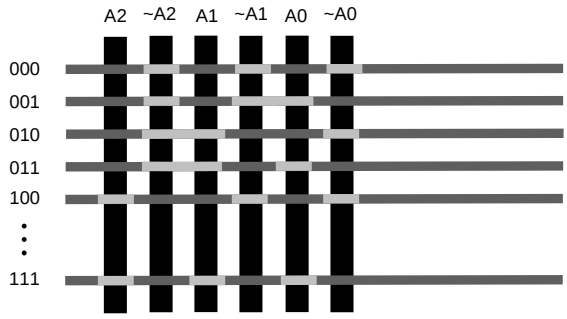


Figure 2.7. Lithographic to sublithographic Decoder

its lightly doped, controllable regions have a suitable voltage to allow conduction. If any of the lithographic-scale wires crossing controllable regions provide a suitable voltage to turn off conduction, then the nanowire will not be able to conduct.

Now the conduction through the nanowires can be controlled with lithographic scale wires, each nanowire can be selected separately with the following procedure. All the nanowires are either precharged or weakly pulled to a nominal voltage. We then apply the desired nanowire address to the lithographic-scale address lines. We also apply the desired drive voltage to a common line attached to all the nanowires. If the selected address is present in the array, it will allow conduction from the common line into the array charging up the selected nanowire (see left side of figure 2.8). The other nanowire crossing the junction can be selected with the same procedure, and the junction can be programmed by applying the appropriate voltage at the crossing nanowires. Note that there is no directionality to the decoder. Consequently, this same unit can also serve as a multiplexer. That is, when we apply an address to the lithographic scale wires, it allows conduction through the addressing region for only one of the nanowires. Consequently, we can sense the voltage on the common line rather than drive it.

2.3.3.1 Nanowire Codes

A dual-rail binary code is used for each logical lithographic address bit. That is, for each logical address bit, we provide the value and its complement. This results in two bit positions on the nanowire for each logical input address bit, one for the true sense and one for the false sense. To code a nanowire with an address, we simply code either bit position to be sensitive to exactly one sense of each of the bit positions (figure 2.7). This results in a decoder which requires $2\log_2(N)$ address bits to address N nanowires. The technique in [44] shows a denser addressing using $N_a/2$-hot codes (N_a is the number of address bits). That is, it simply requires that half of the address bits, N_a, be set to a voltage which allows conduction and half to be set to a voltage that prevents conduction. This scheme requires only $\lceil 1.1\log_2(N)\rceil + 3$ address bits.

If each nanowire in the array has a unique address in our selected coding scheme, we can uniquely address each individual nanowire in the array. However, similar to issue in the restoration plane, our nanowire assembly techniques (section 2.2.1.1) do not allow us to uniquely select and place particular nanowires in particular locations. However, if the code space for the nanowires is large compared to the size of the nanowire array, it is statistically guaranteed that with arbitrarily high probability every nanowire in an array has a unique address. That is, we start with growing a very large number of nanowire codes. We mix up the nanowires before assembly, and randomly select an array of coded nanowires. As long as the array formed is sufficiently small compared to the code space, with high probability each array contains nanowires with unique codes [44]. It turns out that we do not need a large number of address bits in order to guarantee this uniqueness. For example, the $N_a/2$-hot codes need a total of only $\lceil 2.2\log_2(N)\rceil + 11$ bits to achieve over a 99% probability that all nanowires in an array will have unique addresses. If a few duplicates are tolerable, then the codes can be much tighter [46][47]. More information about tolerating nanowire code misalignment can be found in [44].

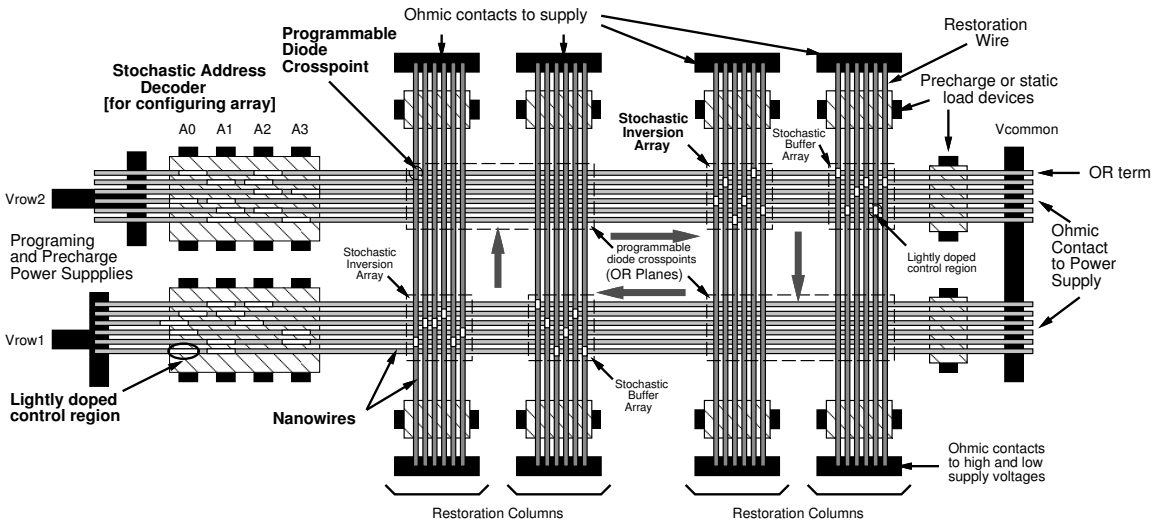


Figure 2.8. Single nanoPLA block

2.3.4 Architecture

Combining the building blocks introduced above, we can construct complete, programmable logic architectures with all logic, interconnect, and restoration occurring in the atomic-scale nanowires. Programmable switch crossbar (section 2.3.1) provides wired-OR programmable logic, field-effect restoration arrays (section 2.3.2) provide gain and signal inversion, and the nanowires themselves provide interconnect among arrays. Lithographic scale wires provide a reliable support infrastructure which allows device testing and programming, by addressing individual nanowires using the decoders introduced in section 2.3.3. Lithographic-scale wires also provide power and control logic evaluation.

figure 2.8 shows a simple nanoPLA block organization with no interblock routing. The array forms a two-plane PLA cycle. Each plane consists of a programmable cross-point OR array (section 2.3.1) followed by a restoration and inversion array (section 2.3.2). Consequently, each plane is a programmable NOR. The combination of NOR-NOR planes is essentially an AND-OR PLA. As figure 2.8 shows, each horizontal wire forms a wired-OR. Crossed nanowire inputs connected to a horizontal nanowire through a junction programmed into the low-resistance ON state can potentially pull up the nanowire, whereas inputs connected through high-resistance OFF junctions do

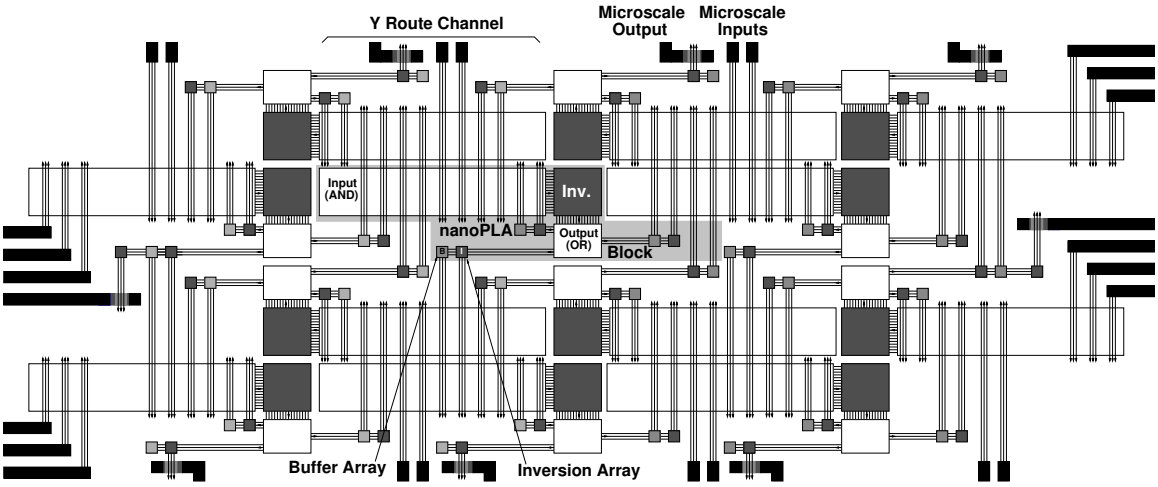


Figure 2.9. Interconnected nanoPLA blocks

not allow sufficient current to pass through the junction to pull up the nanowire. We precharge the nanowire to a low voltage, using the lithographic scale connections (right side of figure 2.8) so that the output is appropriately low when none of the programmed input nanowires is high. The vertical nanowires serve as buffers or inverters to restore and potentially invert the signals formed on the horizontal, wired-OR nanowires. Each horizontal nanowire can act as a field-effect gate for a vertical wire so that each vertical wire output is simply a restored, potentially inverted, version of a horizontal wire (section 2.3.2).

The lithographic to sublithographic decoder lets the lithographic superstructure test each nanowires connectivity. Furthermore, we can exploit the restoration connections to set or reset a single cross-point in either the top or bottom OR plane by using both of the decoders and via one of the restoration plane. In principle, all the nanowires should be equivalent. Therefore, we can assign a given OR function to any of the nanowires within the array, avoiding defective nanowires.

A typical array has 100 nanowires in each plane. To build large components, we can extend these nanoPLA blocks to include I/O to other nanoPLA blocks and assemble them into a large array, as figure 2.9 shows. The logic structure is basically the same. However, a given nanoPLA block now has horizontal input nanowires from arrays above and below it. By carefully arranging the overlap between these

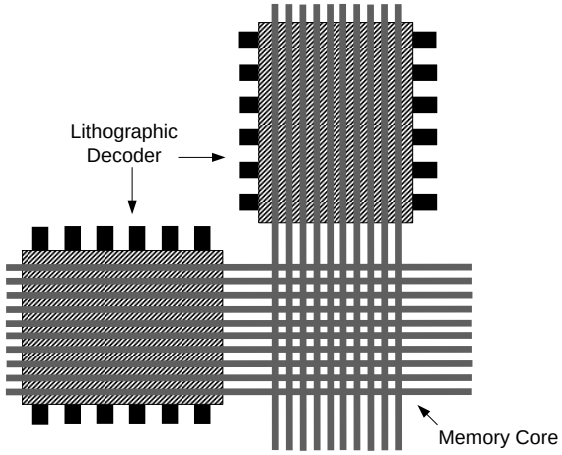


Figure 2.10. Nanomemory organization

arrays, we can support arbitrary Manhattan routing (orthogonal routing on a 2D x, y grid) [33]. At a high level, this provides a structure similar to conventional, Island-style FPGAs, [48] where logic blocks consist of lookup tables (LUTs) rather than PLAs but otherwise comprise bit-level logic blocks inside a bit-level, configurable network. Switching occurs through the nanoPLA logic blocks, and the OR arrays now serve as both wired-OR logic and crossbar switching points. A more complete description of these architectures, their fabrication, and their operation, is available in [33][13].

2.4 Nanomemory Architectures

Combining the programmable crossbar core (section 2.3.1) with a pair of decoders (section 2.3.3), we can build a tight-pitch, nanowire-based memory array [49]. Figure 2.10 shows how these elements come together in a small memory array. The entire array is formed using crossed, tight-pitch nanowires. Programmable diode cross-points are assembled in the nanowire-nanowire crossings. Lithographic-scale address wires form row and column addresses that can access each nanowire junction separately through the lithographic stochastic decoder (section 2.3.3). However, as explained in section 2.3.3 due to the stochastic selection of codded nanowires, and high defect rate, the present and functional nanowire codes are unknown.

In order to address the present and functional nanowires, we must either (re)discover the present addresses when we need to access the device, or we need to store away the set of addresses for known good nanowires so we can address them directly. For applications like nanoPLA programming, rediscovering the addresses when we want to (re)program the device may be viable. However, for data storage applications, it is unreasonable to search through an address space which is $O(N^2)$ large in order to find a particular address. Consequently, we will need to store a translation table which maps the good addresses within the total address space. Unfortunately, the size of this translation table is too large to store in a lithographic-scale memory without negating much of the density advantage of the sublithographic memory core.

A programmable deterministic address decoder is proposed to resolve this problem [50][51]. It provides deterministic addressing in the face of random assembly to make the operational address decoders deterministic and programmable. This way, we can assign each address we would like to see in the array to a good wire in the array. However, in order to program up a nanoscale junction, we generally need to address just that junction; that is, we need to place a programming voltage differential across only the micro-wire and nanowire which make up the junction. Consequently, we will need to start with nanowire addressability in order to bootstrap the programming process. To achieve this addressability, we build a pair of address decoders on each set of nanowires (see figure 2.12). The first address decoder is built using the previously mentioned stochastic decoder scheme (section 2.3.3) to achieve unique addressability of nanowires. We can then use this address decoder to configure the programmable address decoder (see figure 2.11). During operation, we use only the programmed address decoder. Once we have this programmable address decoder scheme, we compose a pair of them to build a deterministically addressable memory.

Write operations into the memory array can be performed by driving the appropriate write voltages onto a single row and column line. Read operations occur by driving a reference voltage onto the common column line, setting the row and column addresses, and sensing the voltage on the common row read line. To provide multiple-bit access to and from a single memory bank, we simply split the common read line

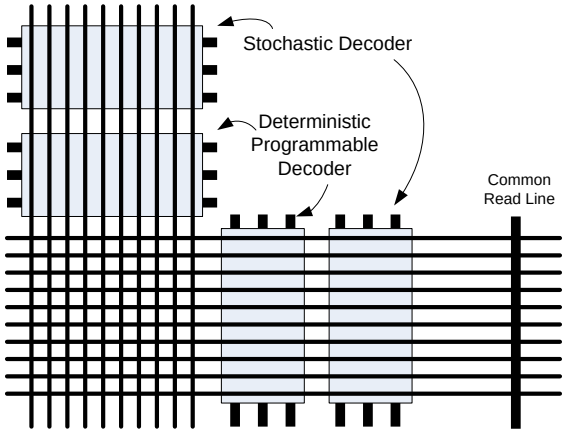


Figure 2.11. Nanomemory with programmable deterministic address decoder

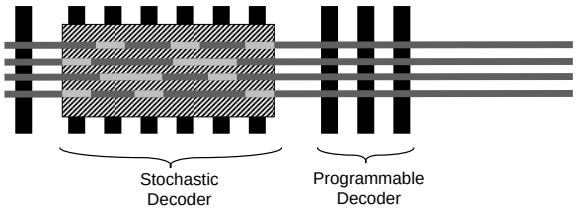


Figure 2.12. Programmable deterministic address decoder

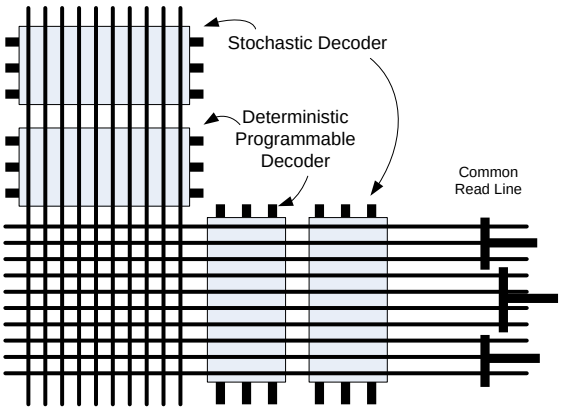


Figure 2.13. Multi-bit memory access

into separate microscale connections to the nanowire array (see the right-hand side of figure 2.13). We then program up the nanowire addresses so that the same address is present in each of the nanowire bundles associated with a distinct microscale output contact.

Limitations on reliable nanowire length and the capacitance and resistance of long nanowires prevent us from building arbitrarily large memory arrays. Instead we break up large nanowire memories into banks similar to the banking used in conventional DRAMs. Reliable, lithographic-scale wires provide address and control inputs and data inputs and outputs to each of the nanowire-based memory banks. We expect to yield only a fraction of the nanowires in the array due to wire defects. Error-correcting codes (ECC) can be used to tolerate nonprogrammable cross-point defects as will be shown in more detail in chapter 6. After accounting for defects, ECC overhead, and lithographic control overhead, net densities on the order of 100 Gbits/cm^2 appear achievable, using nanowire pitches around 10 nm [49].

2.5 More Nanotechnology-Based Architecture

Several groups have been studying variants of these nanowire-based architectures. Heath et al. articulated the first vision for constructing defect-tolerant architectures based on molecular switching and bottom-up construction [52]. Luo et al. elaborated the molecular details and diode-logic structure [53]. HP introduced a random particle decoder scheme for addressing individual nanowires from lithographic-scale wires [54]. These early designs assumed diode logic was restored and inverted using lithographic scale CMOS buffers and inverters. CMU described an interconnected set of these chemically-assembled diode-based devices [55]. They uses only two-terminal nonrestoring devices in the array, but add latches based on resonant-tunneling diodes (RTDs) for clocking and restoration [56]. HP suggests nanoFET-based logic and also tolerates nonprogrammable cross-point defects by matching logic to the programmability of the device [57]. RPI also explore cross-point programmable nanowire-based programmable logic [58]. They use lithographic-scale buffers with an angled topology

and nanovias so that each long nanowire can be directly attached to a CMOS-scale buffer. These designs all share many high-level goals and strategies as described in this article. They suggest a variety of solutions to the individual technical components including the cross-point technologies, nanowire formation, lithographic-scale interfacing, and restoration. The wealth of technologies and construction alternatives identified by these and other researchers increases our confidence that there are options to bypass any challenges which may arise realizing any single technique or feature in these designs.

Chapter 3

Cost of Ignorance and Cost of Knowledge

Exploiting the knowledge of the defect map in system design, can lead to better results from area, time, and power point of view. The system can have more compact area, because the design program can be tailored to the available defect free resources. It can have faster clock cycles, because the critical path does not pass through extra resources and are shorter. Finally it can consume less power, because only the defect free and required devices are turned ON.

Due to the above reasons, there are many defect-tolerant approaches that extract the defect location map and exploit this information to develop a closer-to-optimum defect-tolerant design [59][60]. However, extraction of defect location map becomes more challenging due to the increase in the system complexity and expected high defect rate in nanotechnology designs. Therefore, many recent works suggest defect-tolerant techniques that are oblivious to the defect location maps. It is true that extracting the defect location map can become more challenging for nanotechnology systems, however, it is important to understand and compare the cost paid for gaining this knowledge and the cost paid for ignoring it. This chapter is an effort to make this cost and gain quantified.

Defect-tolerant approaches can be partitioned into two categories based on the approach that they take against using defect location map: (1) *knowledge-based* approach, which discovers the defect location map, and configures the system around the defect locations and (2) *ignorant-based* approach, which designs the reliable systems

independent of the defect locations.

The *knowledge-based* approaches which exploits the defect location maps requires the following capabilities in the system

1. The system must have postfabrication configurability, to isolate defects.
2. There must be enough redundant resource in the system to guarantee the system functionality even after removing the defective parts.
3. The defect location map must be discovered before programming the system.

For the first condition, the postfabrication configurability can be achieved with various emerging technology devices, e.g., [2]catenane-based molecule [9], mechanical nanotube switch [10] or conventional programmable devices like SRAM-based reprogrammable switches, floating gate transistor, and fuses. To satisfy the second condition, the critical resources must be overpopulated until it statistically guarantees the system functionality. Probably the most challenging requirement is to satisfy the third condition which is discovering the defect locations map. Testing integrated circuits is generally a time consuming process, however, testing and localizing the defective nodes demands even larger set of test vectors and is potentially even more time consuming.

The alternative defect-tolerant approach would be to ignore the defect location map and provide reliability, oblivious to the defect locations. This method does not demand discovering the defect node locations. However, since it is oblivious to the defect locations, it cannot exploit the full capability of the hardware, and therefore, results in suboptimal designs. This method provides enough resource redundancy that if some of the resources are defective the correct value can still be extracted as long as the majority of the replicated resources holds the correct value.

We partition the defect-tolerance techniques based on the part of the circuit that they protect, into two categories. The defect-tolerant designs protecting

1. Computation resources (logic circuit)
2. Interconnect resources.

In this chapter we analyze different fault-tolerant techniques to protect each of the above resources in a separate section, and compare the techniques using ignorant-based and knowledge-based approaches for each of them. This chapter focuses on the area overhead and analyze the impact of ignoring the defect locations on area overhead.

The knowledge-based techniques (detect-and-configure techniques) are another instantiations of *Using Alternative Resource* design pattern described in chapter 1. In these techniques the cost of extra area overhead in the ignorant-based technique is shifted to the cost of defect map extraction and design mapping technique in the knowledge-based techniques. Knowledge-based techniques require extra hardware support for testing the circuit. Some time is also needed to extract the defect map, match the design configuration with the defect pattern, and map the design. By spending these resources (time and test hardware), the cost is essentially shifted from area redundancy to time redundancy (test and configuration time). Then the time redundancy can be further reduced by the technique shown in section 3.3.

3.1 Cost of Ignorance in Interconnect

This section analyzes the defect-tolerant schemes protecting interconnect resources. We compare the two type of defect-tolerant schemes introduced above and quantify the advantage of exploiting defect map. We compare these schemes over a simple routing example. In ignorant-based approach, each interconnect channel in this technique is implemented with a reliable network that guarantees connectivity with the existence of defects. In knowledge-based approach the interconnect defect pattern has to be discovered using a comprehensive test and localization technique. Then the system will be configured around the defects. This technique is similar to conventional routing that prevents congestion, while some of the resources are already taken (in this case these resources are marked as defective).

In our routing example, we assume a channel with W wires and S buffering segments, where each of the buffer sets is preceded with a fully connected switch

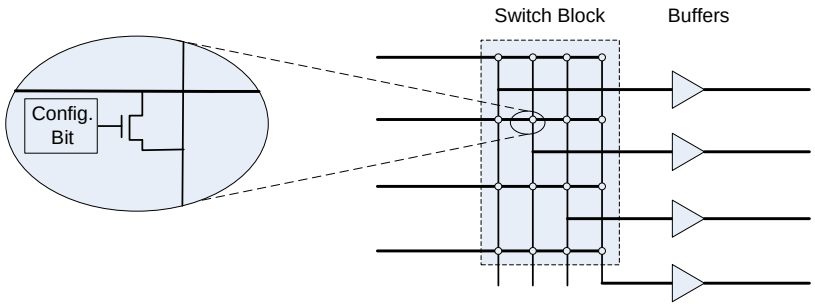


Figure 3.1. Fully connected switch box and buffers

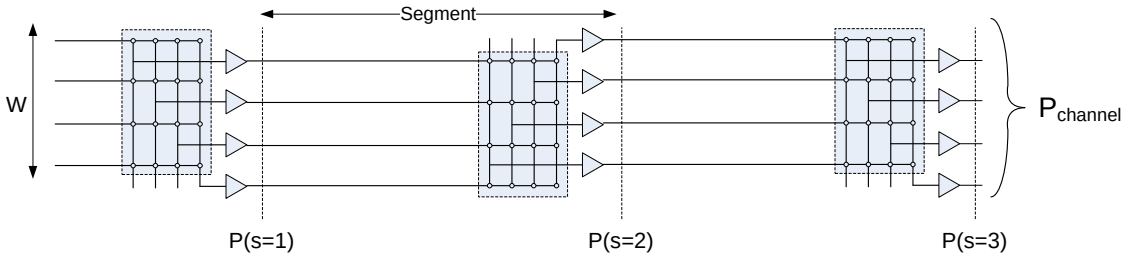


Figure 3.2. Channel of width $W = 4$ with $S = 3$ segments, each containing fully connected switch block. $P(s = i)$, is the probability that N signals are routed up to the ith segment.

block (figures 3.1). The switches are configured to ON and OFF states to route the signals, similar to a simplified FPGA channel (figure 3.2). In this example we assume that the defects are in the wires or buffers and the switches are defect free. The goal is to route N independent paths from N input points on one end of the channel to N output points on the other end of the channel.

3.1.1 Ignorant-Based Interconnect Defect-Tolerant Scheme

In this section we use a general ignorant-based scheme based on Majority-Multiplexing scheme [29]. A complete review of this scheme is provided in section 2.1.3.

Based on the majority-multiplexing technique, each wire is implemented using a bundle of size M. Figure 3.3 shows a reliable unit implementing a single buffer and

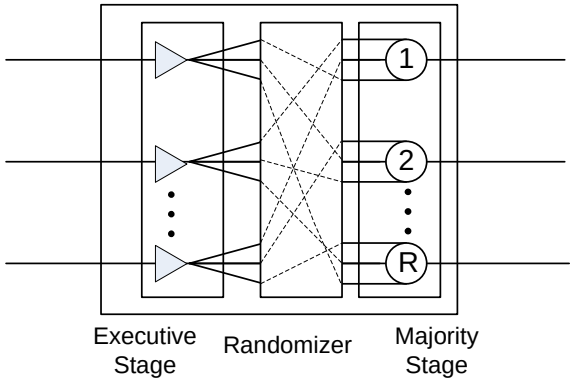


Figure 3.3. Single wire protected with majority multiplexing

interconnect stage using majority-multiplexing technique. All the W wide channels with S buffering stages, are implemented with majority-multiplexing technique as shown in figure 3.4. Each of the W wires is replaced with a wire bundle of size M. Since each of the W routes guarantees the connection, there is no need to overpopulated the channel; i.e., for routing N signals we need exactly N routes. Therefore, $W = N$.

The M-wide bundle passes through an executive stage, randomizer interconnect, and restorative stage, to implement a reliable processing unit. The executive stage is essentially replicated copies of the unit under protection, since in this case the unit is a wire followed by a buffer and a wire does not have any functionality, the executive stage contains of only buffers (figure 3.3).

In the majority multiplexing, the resource redundancy and the effective integration in each bundle tolerates any potential defects in the wires, buffers, or majority gates, and guarantees the correct routing. Therefore, there is no need to discover the defect location map of each chip and a fixed configuration is used for all the chips.

Now we analyze the system reliability based on this technique. In one bundle an output of a majority gate at segment s is correct if the majority gate is correct and the majority of the input signals into the gate is also correct. This is shown in the

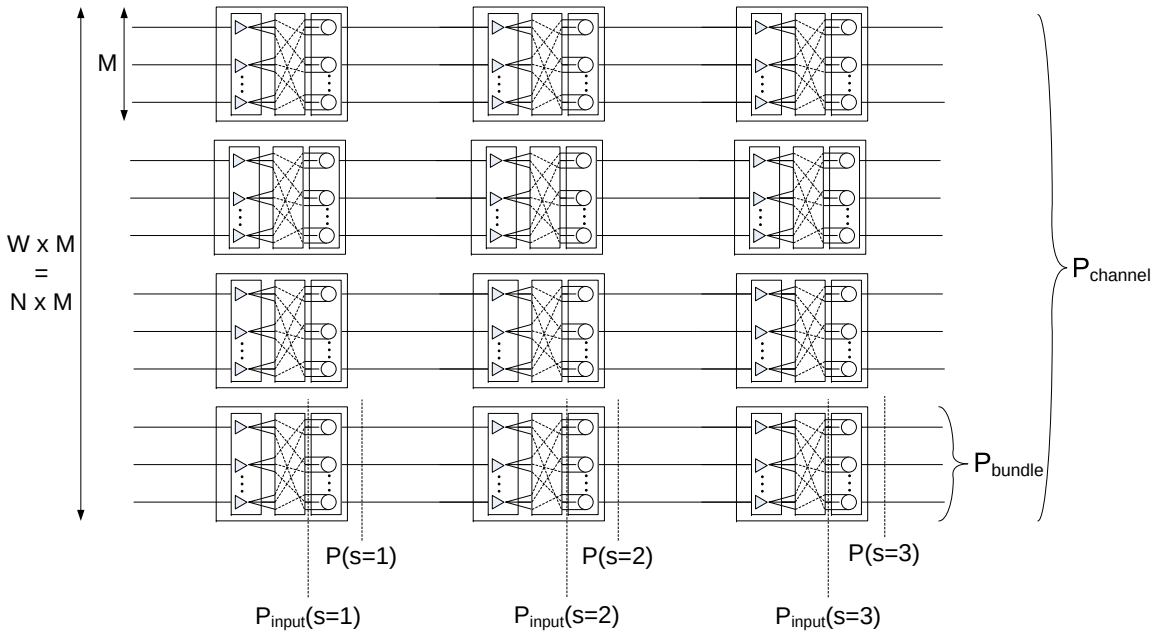


Figure 3.4. A channel implemented with majority multiplexing buffers

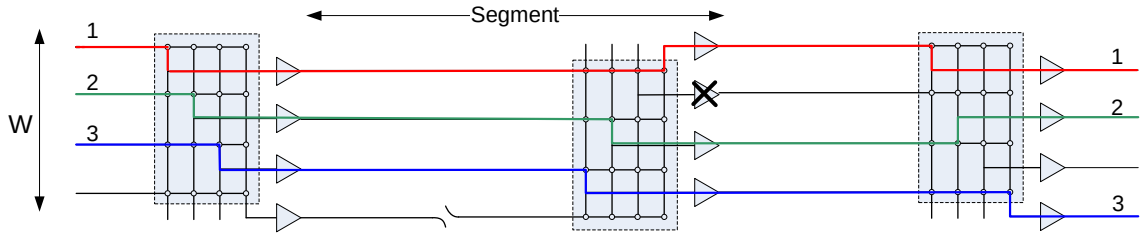


Figure 3.5. One possible routing of $N = 3$ paths, isolating a broken wire and a defective buffer

following equation.

$$P(s) = P_{gate} \sum_{i=\lceil \frac{M+1}{2} \rceil}^{M} \begin{pmatrix} M \\ i \end{pmatrix} P_{input}(s)^i (1 - P_{input}(s))^{M-i}. \tag{3.1}$$

$P(s_0)$ is the probability that a wire coming out of a segment $s = s_0$ is correct. $P_{input}(s_0)$ is the probability that a wire entering the majority stage of majority-multiplexing unit at segment $s = s_0$ is correct (figure 3.4). Thereby, $P_{input}(s_0)$ can be computed from the $P(s_0 - 1)$ as follows

$$P_{input}(s_0) = P(s_0 - 1) \times P_{good\_buffer} \times P_{good\_wire}. \tag{3.2}$$

Here we assume that $P(0)$ is 1; the primary input signals are defect free. At the final segment, each bundle holds the correct value if at least half of the wires of the bundle are correct. Therefore, the probability that a bundle yield through the final segment S is

$$P_{bundle} = \sum_{i=\lceil \frac{M+1}{2} \rceil}^{M} \begin{pmatrix} M \\ i \end{pmatrix} {P_{(S)}}^i (1 - P_{(S)})^{M-i}. \tag{3.3}$$

The final segment is fed into a reliable majority gate to generate the output signal. The channel yields when all the N bundles yield through the final segment, which is computed with the following approach.

$$Channel\_Yield\_Maj = {P_{bundle}}^N. \tag{3.4}$$

Here we assume that $P_{good\_buffer} = P_{good\_wire}$. The channel width growth as the function of P_{good_wire} to route $N = 100$ signals is plotted in figure 3.6. For this simulation the target $Channel\_Yield\_Maj$ is 0.999, and the channel width that guarantees this yield is computed for different values of P_{good_wire} and is plotted in figure 3.6. We compare this channel width with the channel width of the knowledge-based technique in the coming section.

3.1.2 Knowledge-Based Interconnect Defect-Tolerant Scheme

Here we perform the above routing operation on overpopulated channel with full knowledge of the defect locations. For detect-and-configure techniques the defect location map has to be discovered first and all the defective nodes have to be localized. Once the defect location map is available N independent and defect free paths must be routed through the W wires of the channel. Figure 3.5 shows a simplified example of routing $N = 3$ wires over $W = 4$ wide channel.

Remember from the problem assumption stated in section 3.1.1, that the switch block is fully connected and fault free. Using this assumption, to find N independent defect free paths among W wires, we must find N defect free paths in each segment. The N independent defect free paths of each segment can then be connected using the fully connected defect free switches. The channel yield with this technique is computed with the following approach.

$$P_{seg} = \sum_{i=N}^{W} \begin{pmatrix} W \\ i \end{pmatrix} P^i (1-P)^{W-i}. \tag{3.5}$$

This is the probability that each segment yields N paths, where P is the probability that a wire and the corresponding buffer is defect free,

$$P = P_{good\_wire} \times P_{good\_buffer}, \tag{3.6}$$

The channel yields when all the segments yield at least N wires and buffers, therefore,

$$Channel\_Yield\_Config = (P_{seg})^S, \tag{3.7}$$

figure 3.6 shows the channel width required to route $N = 100$ signals with 0.999 yield. This graph shows that for 11% defect rate the knowledge-based technique result an order of magnitude narrower channel width compared to the ignorant-based technique. The curves show that the channel width of ignorant-based technique grow considerably faster than the knowledge-based technique.

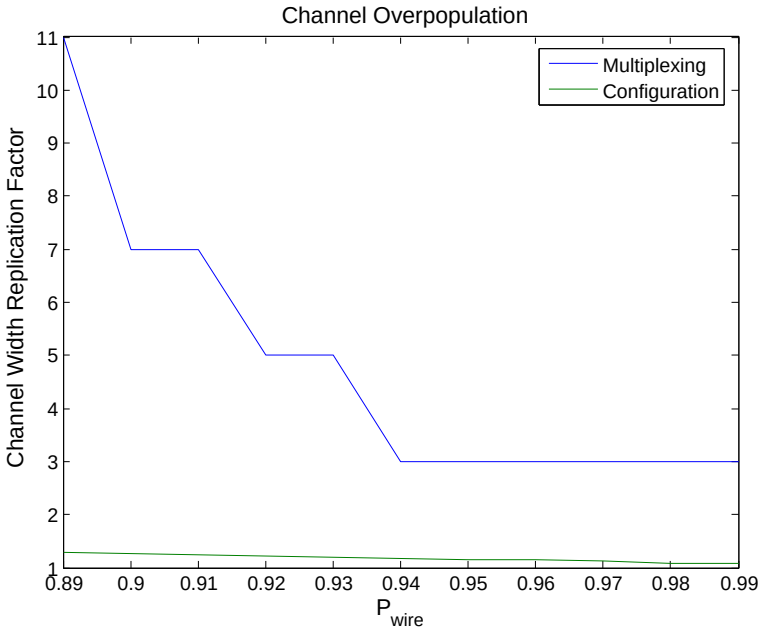


Figure 3.6. The overhead on the number of wires in a channel for defect-tolerant design. The yield is 0.999 and defect rate of all the defect-prone components range from 0.10 to 0.01. The required number of paths to be routed is $N = 100$.

This simple example illustrates the advantage of a knowledge-based technique over ignorant-based technique from area point of view. For this reason there are many defect-tolerant work on programmable substrates like FPGA based on detect-and-configure technique [59][60]. The defect pattern can be discovered with the test and defect localization techniques [61][62][63]. Once the defect configuration map is present then the signals can be routed to avoid defective wiring resources. This is similar to the normal routing problems that avoid congestions. We use a detect-and-configure based technique to program logic in nanowire-based substrate. This techniques shows about 30% overhead on average to tolerate 10% defect rate. The detail of this technique will be shown in chapter 4

3.2 Cost of Ignorance in Logic

In this section we analyze defect-tolerant approaches protecting computation (logic circuit). Similar to the previous section, we compare the knowledge-based and ignorant-

based techniques.

We compare the above two technique using a simple example. Assume we want to have a single reliable logic gate. Using a simple ignorant-based technique, the gate is replicated R times followed by a reliable majority gate. This unit compute the logic function reliably if the majority of the R gates are defect free. Therefore, with P being the failure rate of a logic gate the probability that the unit works correctly is

$$Gate\_Yield\_Maj = \sum_{i=\lceil \frac{R_M+1}{2} \rceil}^{R_M} \left(\begin{array}{c} R_M \\ i \end{array} \right) {P_{gate}}^i (1 - P_{gate})^{R_M - i}. \tag{3.8}$$

For knowledge-based technique, the gate is also replicated into R_C copies and performs correctly if at least one defect free gates exists. The defect free gates are located and assuming defect free configuration, one of them is configured as the single logic gate and the rest of the copies will be isolated. Therefore, the probability that this technique performs the logic function correctly is when it yields at least one gate,

$$Gate\_Yield\_Config = 1 - (1 - P_{gate})^{R_C}, \tag{3.9}$$

figure 3.7 shows the area overhead of the above two techniques (ignorant-based and knowledge-based techniques) to yield a single reliable gate with the probability 99.9%. For 10% defect rate the ignorant-based majority scheme requires 17 copies, while the knowledge-based configuration scheme takes only 3 copies.

In the above analysis we fixed the target yield and the number of yielded gates (at least 1 reliable gate), and then we compared the final area overhead. We can analyze the above comparison from another point of view. With the same reliability target for both techniques, we replicate the defect-and-configure techniques with R_M copies, equal to the replication of the majority technique. Then the detect-and-configure technique can potentially yield at least half of the gates $\lceil R_M + 1 \rceil / 2$ while majority only yield a single gate. Therefore, with the same amount of resource the detect-and-configure technique provides more computational power compared to the majority technique.

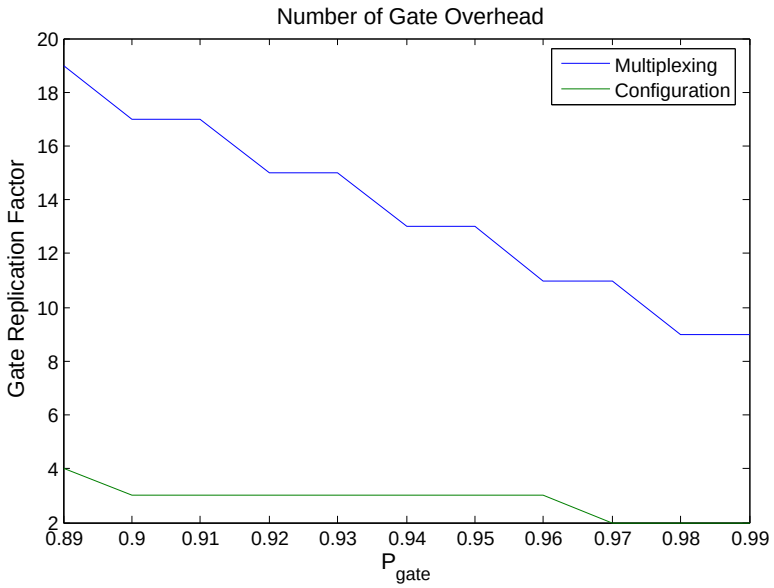


Figure 3.7. Number of logic gates required for detect-and-configure technique and majority technique to yield at least one gate with 99.9% probability.

The comparison of majority scheme and configuration scheme can be generalized further to the case that the majority gate and the configuration devices are not reliable either. These cases are addressed in von Neumann-type majority-multiplexing scheme [29] [11] (section 2.1.3.2) and defect-tolerant matching technique (chapter 4).

The above simple examples for interconnect and logic defect-tolerant techniques showed an absolute advantage of the knowledge-based defect-tolerant techniques over ignorant-based techniques in area overhead. In the following section we review the potential cost to gain this benefit: *cost of extracting defect location map.*

3.3 Cost of Knowledge

Above we reviewed the benefits of having defect map and how defect-tolerant techniques can improve the area overhead by exploiting this knowledge. However, extracting the defect map is not a cheap process, especially as the chips become more complex and the defect rate increases. Localizing the defects compared to the normal test operation is a more complicated process. The normal test process is an accept-or-reject process. I.e., as soon as an error is detected, during the test operation, the

chip is marked as defective and will be discarded. For defect localization in the other hand, all the nodes of the chips have to be tested and the defect map is generated. Below we review our test and localizing approaches. In this technique, the defective nanowires is localized in NanoPLA architecture this work suggests a technique that combines the detection process with mapping process, which will expedite the detection process and reduce the time costs.

3.3.1 NanoPLA Test and Defect Localization

To test and localize defective points in nanoPLA [64] structure we use a built-in microprocessor that performs the test operation. The test is performed by an on-chip microprocessor. The microprocessor is responsible to test and detect the manufacturing defects and remap the configuration to nanoPLA blocks. The microprocessor need to be free of defect to perform the test operation correctly. Therefore, it must be implemented in reliable CMOS. This processor will only occupy a tiny fraction of the die area on the chip. The microprocessor serves the following purposes: (1) Discovering the live nanowire addresses existing in the stochastic lithographic scale to nanowire decoder (section 2.3.3). (2) Identifying the usable nanowires which are correctly restorable/invertable. (3) Identifying the programmable cross-points in usable nanowires and isolating the defective ones during mapping.

Since addressing and restoration is stochastic, we will need to discover the live addresses and their restoration polarity. Further, since we will have defective nanowires, we must identify which nanowires are usable and which are not. We use the restoration columns (see figures 2.8) to help us identify useful addresses. The gate side supply (e.g., top set of lithographic wire contacts in figures 2.8) can be driven to a high value, and we look for voltage on the opposite supply line (e.g., bottom set of lithographic wire contacts in figure 2.8; these contacts are marked V_{high} and GND but will be controlled independently as described here during discovery). There will be current flow into the bottom supply only if the control associated with the P-type restoration wire can be driven to a sufficiently low voltage. We start by driving all the row lines high using the row precharge path. We then apply a test address and

drive the supply (V_{row}) low. If a NW with the test address is present, only that line will now be strongly pulled low. If the associated row line can control one or more wires in the restoration plane, the selected wires will now see a low voltage on their field-effect control regions and enable conduction from the top supply to the bottom supply. By sensing the voltage change on the bottom supply, we can deduce the presence of a restored address. Broken nanowires will not affect the bottom supply. Nanowires with excessively high resistance due to doping variations or poor contacts will not be able to pull the bottom supply contact up quickly enough. We sense the buffering and inverting column supplies separately so we will know whether the line is buffering, inverting, or both. We need no more than $O(K^2)$ unique addresses for K logical P-terms to achieve virtually unique row addressing [14], so the search will require at most $O(K^2)$ such probe tests. A typical address width for the nanoPLA blocks of 90 is $N_a = 14$ which provides $3,432$ distinct 7-hot codes. We might thus need to probe $3,432$ addresses to find all the live row wires. This process has to be done for both of the decoders in each nanoPLA block (see figure 2.8). So for one nanoPLA we have to perform $6,863 = 2 \times 3,432$ distinct address check.

Once we know all the present addresses in an array and the restoration status associated with each address, we can assign logic to each nanowire of programmable crossbars. To program each nanowire, each of the junctions has to be selected and programmed one at a time. In order to program a nanowire in the top programmable crossbar (figure 2.8), the top address decoder selects the nanowire and the other address decoder selects each junction via the restoration plane. When the suitable programming voltage is applied to the nanowires of each junction, the junction will be programmed. If any of the junction is broken then the microprocessor adds this junction to its defect configuration map and tries mapping the OR function on another nanowire. In this way the defect location map is generated accumulatively and can be used for mapping the next nanowires.

The logical configuration for each nanoPLA block (the "bitstream") is given to the microprocessor, and it runs the lightweight greedy algorithm [65] to discover the present and functional addresses and perform the local matching (chapter 4) necessary

to assign block configuration to physical nanowires. This technique takes linear time in the number of nanowires to map the whole OR-plane. In this way, the device never stores the entire defect map for the component, but simply rediscovers it one nanoPLA block at a time, and it never exposes the user to the defect details of the array.

3.4 Summary

The defect rate of VLSI design is increasing and the defect rate in emerging nanotechnology designs is expected to be even higher. High defect rate decreases the perfect system yield, and the low system yield increases the chip cost and price. The alternative would be to relax the chip quality condition, i.e., instead of having perfect chip, allowing limited number of defects in the system. This approach has already found its place in memory systems and FPGA chips. The memory chips are overpopulated in the number of rows and columns to compensate for potential defects in the rows or columns. If there are a few defective rows or columns those rows or columns are isolated and the rest of the memory is functional. Therefore, during the test process, only memory chips with too many defects are rejected and memory chips with a few defects that can be isolated is accepted.

In FPGA technology, Xilinx is the pioneer in using defective chips by the technique called EasyPath [66]. It matches the configuration bitstream to the defect pattern of the defective chips. Therefore, once the users know their design is fixed and no longer requires the full programmability of a standard FPGA, the design configuration will be mapped to the FPGA chip. This approach provides a great total cost reduction.

As the above two examples illustrate there is a tradeoffs between yield and cost, which is determined by the quality (number of accepted defect) of the system. Tolerating zero defects in the system results in highest chip quality with lowest yield and consequently highest price. Tolerating some defects, however, increases the number of chip yields and consequently the costs decline. Therefore, by improving the design to tolerate some number of defects, the quality condition can be more relaxed and

more defective chips can be accepted, which increases the system yield and reduces the cost.

Tolerating defects in the system is potentially a costly process. The current chapter revealed, that knowledge-based techniques can result in an order of magnitude more compact designs. These techniques, however, demands the defect locations map. Section 3.3 reviewed some of the approaches to extract this information. Although the test and defect localization techniques are potentially costly process, this chapter reviewed a technique that can reduce these costs, and make it more practical.

Chapter 4

Permanent Defect-Tolerant Design Using Reconfiguration

Tolerating defects is one of the most challenging issues in design nanoscale systems. In this chapter we propose using reconfiguration to tolerating the defective wires and devices in the system. The location of the defective wires and devices in the system is extracted and the system is configured around them. Based on the discussion of chapter 3, since this technique exploits the knowledge of defect locations, it can improve the final design area overhead.

chapter 2 reviewed some of the potential sources of permanent defects. In this chapter, these potential defects are abstracted to the following defect models

1. Wire defects

2. Nonprogrammable cross-point defects.

(1) *Wire defects.* A wire is either functional or defective. A functional wire has good contacts on both ends, conducts current with a resistance within a designated range, and is not shorted to any other nanowires. Broken wires will not conduct current. Poor contacts will increase the resistance of the wire leaving it outside of the designated resistance range. Excessive variation in nanowire doping from the engineered target can also leave the wire out of the specified resistance range. We can determine if a wire is in the appropriate resistance range during testing (chapter 3) and can arrange not to use the ones which are defective, applying the technique in this chapter.

(2) *Nonprogrammable cross-point defects.* A cross-point is programmable, nonprogrammable into ON state, or shorted into the ON state. A programmable junction can be switched between the resistance range associated with the ON-state and the resistance range associates with the OFF-state. A nonprogrammable junction can be turned OFF, but cannot be programmed into the ON-state; a nonprogrammable junction could result from the statistical assembly of too few molecules in the junction or from poor contacts between some of the molecules in the junction and either of the attached conductors. A shorted junction cannot be programmed into the OFF-state. Based on the physical phenomena involved, we consider nonprogrammable junctions to be much more common than shorted junctions. Further, we expect fabrication can be tuned to guarantee this is the case. Consequently, we will treat shorted junctions like a pair of defective wires and avoid both wires associated with the short. We do not currently consider bridging of adjacent nanowires as a major defect source. Radial shells around the (semi)conducting nanowire cores prevent the shorting of adjacent nanowires. At present, there is insufficient experience to determine if variations in core shell thickness, imperfect planar nanowire alignment, or other effects may, nonetheless, lead to bridging defects between adjacent nanowires. If such bridging were to occur, it could make a pair of nanowires indistinguishable, perhaps effectively giving two addresses to the nanowire pair. These bridged nanowire pairs could be detected and avoided but their occurrence would necessitate slightly more complicated testing and verification algorithms than the ones detailed in chapter 3.

For our analysis, we assume the defects have random and identically independent distribution (iid). This assumption is consistent with broken wires resulting from assembly strain, poor contacts made by statistical assembly, and statistical distributions of molecules and connections in junctions. The defect mechanisms anticipated here are very different from those typical in lithographic-scale assemblies, and there is insufficient experience with manufacturing of these arrays to suggest more sophisticated models (e.g., clustering) at this point.

In the rest of this chapter we start by showing how to tolerate defective wires and then nonprogrammable cross-points in each nanoPLA block, followed by the

experimental results.

4.1 Tolerating Defective Wires

Tolerating wire defects is a simple matter of provisioning adequate spares, separating the good wires from the bad, and configuring the nanoPLA blocks accordingly. For a given PLA design, we want each block to have a minimum number of usable wires (OR-term and interconnect wires). Since we will have wire losses, we design the physical array to include a larger number of physical wires to assure the yield of enough usable wires to meet our logical requirements.

For the detailed architecture described in section 2.3.4, wires actually work in pairs. A horizontal OR-term wire provides the programmable computation or programmable interconnect and a vertical restoration wire provides signal restoration and perhaps inversion. Since the gatable junction between each restoration wire and its associated OR-term wire is not programmable, a defect in either wire will result in an unusable pair. Consequently, each logical OR-term or output will yield only when both wires yield. Let P_{wire} be the probability that a wire is not defective. The probability of yielding each OR-term is

$$P_{or} = P_{out} = (P_{wire})^2 . \tag{4.1}$$

We can perform an M-choose-N calculation to determine the number of wires we must physically populate (N) to achieve a given number of functional wires (M) in the array. The probability that we will yield exactly i restored OR-terms is

$$P_{yield}(N, i) = \left(\binom{N}{i} (P_{or})^i (1 - P_{or})^{N-i} \right) . \tag{4.2}$$

That is, there are $\binom{N}{i}$ ways to select i functional OR-terms from N total wires, and the yield probability of each case is

$$(P_{or})^i (1 - P_{or})^{N-i} . \tag{4.3}$$

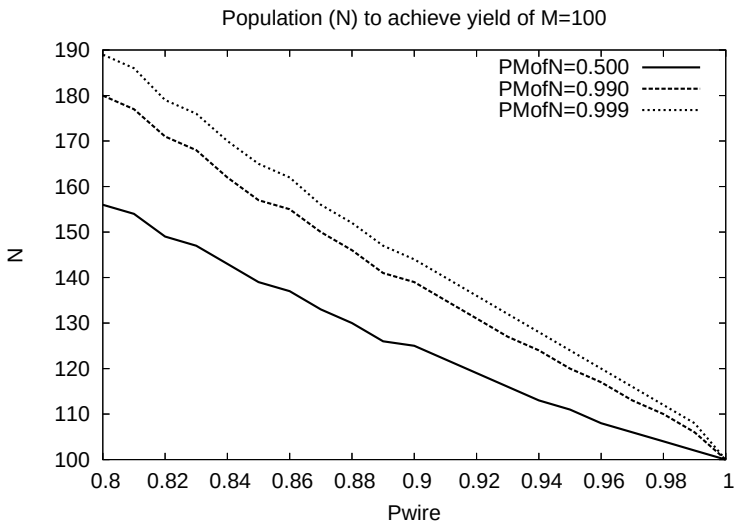


Figure 4.1. Physical Population (N) of wires to Achieve 100 Restored OR-terms (M)

We yield an ensemble with M items whenever M or more items yield, so our system yield is actually the cumulative distribution function,

$$P_{MofN} = \sum_{M \leq i \leq N} \left(\binom{N}{i} (P_{or})^i (1 - P_{or})^{N-i} \right), \tag{4.4}$$

Given the desired probability for yielding at least M functional OR-terms, P_{MofN}, equation 4.4 gives us a way of finding the number of physical wires, N, we must populate to achieve this. For our interconnected nanoPLA blocks, the product terms and interconnect wires will be the M's, and we will calculate a corresponding N to determine the number of physical wires we must place in the fabricated nanoPLA block. Figure 4.1 plots the N required to achieve 50%, 99%, and 99.9% yield rates (P_{MofN}) as a function of P_{wire} when building $M = 100$ wire arrays.

Once we know the number of physical wires to populate, we can build physical area models to calculate the size of a given array. From this we can calculate the area overhead associated with sparing for a given wire defect rate. figure 4.2 plots the area overhead as a function of P_{wire} for a typical array assuming the reliable, lithographic substrate uses 105 nm pitch wires (e.g., 45 nm technology node) and the nanowires have 10 nm pitch. If the design consisted *only* of nanowires, we would expect the area to scale as $\left(\frac{N}{M}\right)^2$; however, since the nanowires only make up a fraction of the

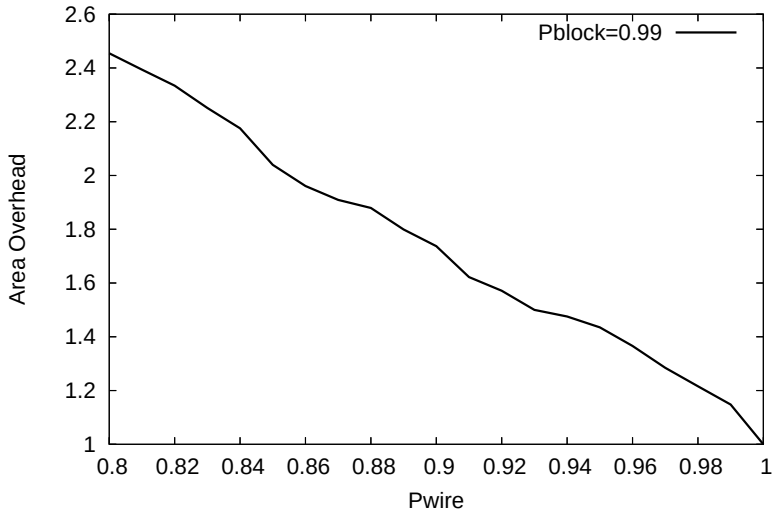


Figure 4.2. Impact of Wire Yield Rate (P_{wire}) on Area for an Interconnected nanoPLA Block with 80 net product terms and 25 net interconnect wires per channel (nanowire pitch is 10 nm; reliable superstructure pitch is 105 nm.)

area (see figure 2.8), the area overhead scales more slowly; at 80% wire yield rate and the target yield rate (P_{MofN}) of 99%, we only see an overhead around 2.4 instead of $(1.8)^2 = 3.2$.

The lithographic-scale addressing units and lithographic-scale contacts allow us to identify the addresses of individual, restored OR-term wires. A typical testing routine would sequentially test all the possible addresses for nanowires in an array, attempting to charge one wire at a time for conduction and restoration. By watching the voltage on the lithographic-scale supply contact attached to the far end of a restoration column (see figure 2.8), it is possible to determine if the address is present, properly restored, and identify the resistance associated with the nanowire's signal path. We record the address of each nondefective wire discovered and use only those addresses during subsequent device configuration. Section 3.3.1 shows more detail of the testing process.

4.2 Tolerating Defective Cross-Points

Since each wire in a nanoPLA block has around 100 logical junctions, it is unlikely that any single wire is free of defective junctions. For example, at a 10% nonprogrammable

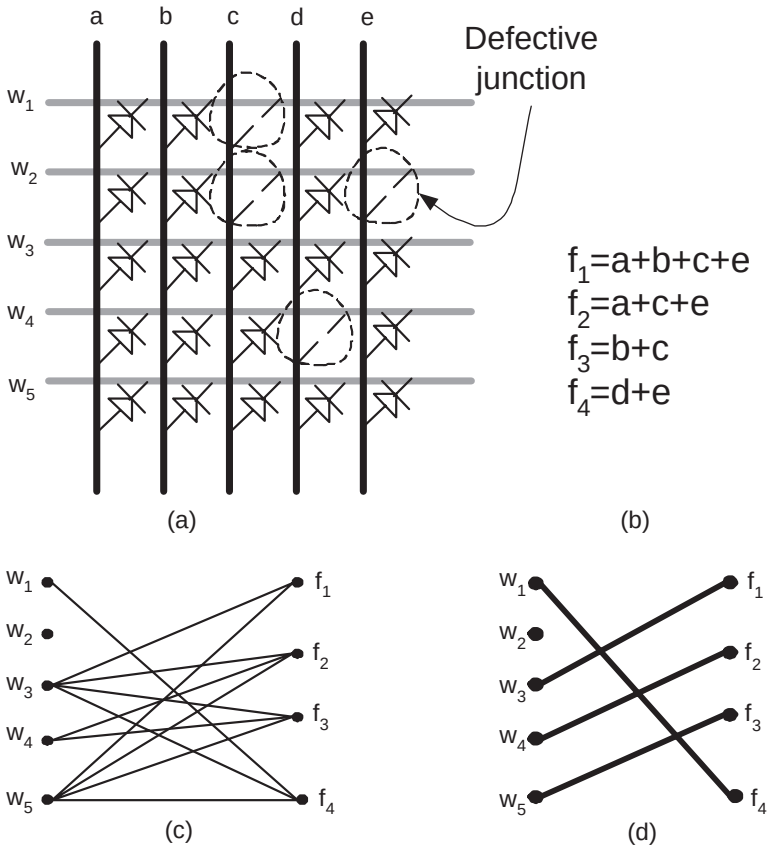


Figure 4.3. (a) A logic array with defective junctions. (b) An example set of OR-terms. (c) A bipartite graph with OR-terms in the right and nanowires in the left. An edge shows that an OR-term can be mapped to a nanowire. (d) One possible assignment of the OR-terms to nanowires.

cross-point defect rate, the likelihood of a wire with 100 junctions having no defects is $(0.9)^{100} = 3 \times 10^{-5}$; this further suggests there is only a 0.3% chance that there is even one defect free wire in an array of 100 nanowires. Consequently, in order to cope with these high junction defect rates, we must be able to use wires even when they contain defective junctions.

In practice the OR-terms are sparsely programmed. Each OR-term receives both the true and complement sense of each input, and in most terms only a subset of all the input variables appear. Consequently, most OR-terms will need fewer than 50% of their junctions enabled, and we can tolerate nonprogrammability defects in the rest.

The main idea is to match the logic of an OR-term to the defect pattern of a nanowire. An OR-term is compatible with the defect pattern of the nanowire if and

only if the inputs of the OR-term are a subset of the nondefective junctions of the nanowire. For example if a programmable crossbar of a nanoPLA has defective junctions as marked in figure 4.3, the OR-term $f_1 = a + b + c + e$ can be assigned to nanowire w_4, despite the fact that it has a defective (nonprogrammable) junction at (w_4, d)—i.e., the OR-term f_1 is compatible with the defect pattern of nanowire w_4. We can combine this idea with the fact that the OR-term inputs are sparse and turn nanowire assignment into a matching problem.

4.2.1 Algorithms

Assume we have a logic array with a defect pattern similar to figure 4.3(a) and we want to program it to the set of OR-terms in figure 4.3(b). We first determine which OR-terms are compatible with the defect pattern of each nanowire. We then make a graph, like the one in figure 4.3(c), showing which OR-terms can be assigned to which nanowires. The nodes on the right side of the graph in figure 4.3(c) are the OR-terms and the nodes on the left side are the nanowires. An edge between an OR-term and a nanowire indicates that the OR-term is compatible with the defect pattern of the nanowire. Next we try to find a complete assignment from the OR-terms to the nanowires. Figure 4.3(d) shows one possible assignment. Finding an assignment is equivalent to identifying a bipartite graph matching. While we could find the matching with a standard, optimal bipartite matching algorithm, we find a linear-time greedy heuristic provides reasonably good results for these defect rates while running in substantially less time [65].

Let F be the set of OR-terms and W be the set of nanowires; let f_i represent an OR-term in F and w_j represent a nanowire in W. Our heuristic algorithm (shown in figure 4.4) picks the f_i's in decreasing order of their fanin size (because larger fanin OR-terms are harder to map) and the w_j's randomly. When the number of ON junction per nanowire is bound to a constant, the number of wires tested in line 5 for each OR-term, f_i, is a constant and this algorithm runs in linear time, $O(|F|)$. Note that this is even smaller than the $O(|W|^2)$ or $O(|W|^5)$ operations which would be required to diagnose the programmability of the cross-points in the array, which

```
1   do {
2     f_i = unmapped OR-term in F with largest fanin
3     do {
4       w_j = nanowire randomly selected from unused nanowires in W
5       if (f_i can be mapped to w_j)
6         assign f_i to w_j
7         mark f_i as mapped
8         mark w_j as used
9       } while (f_i unmapped)
10    } while (there are unmapped f_i in F)
```

Figure 4.4. Greedy Algorithm for Matching OR-terms to Nanowires with Potentially Defective Crosspoints

would be necessary to construct the full graph used for matching in the optimal algorithm [49].

4.2.2 Fanin Bounding

In order for the preceding algorithm to succeed, there must be enough nanowires in the logic array to allow every OR-term to be assigned to a nanowire. We can derive a lower bound on the number of spare nanowires we need in the array based on the OR-term fanin. The probability that an OR-term f_i can be mapped to a nanowire in the graph is $(P_j)^{c_i}$ because all the c_i junctions need to be programmable. The expected number of nanowires connected to f_i in the graph is $|W| \cdot (P_j)^{c_i}$, where $|W|$ is the number of nanowires. In order to find a complete assignment from OR-terms to nanowires, the expected size of the node degree of each OR-term must be at least one. Assuming c_i's are bounded by C,

$$|W| \cdot (P_j)^C > 1. \tag{4.5}$$

This immediately implies that $|W|$ should be greater than $(P_j)^{-C}$. When C is large, $|W|$ must be unacceptably large. For example with $P_j = 0.85$ and $C = 40$ the above bound suggests $|W| > 665$. To allow reasonably sized logic arrays, the maximum fanin, C, must be bounded. For example, if we want to map $|F| = 100$ with little overhead ($|W| \approx 100$), we must keep $C < 28$ since $(0.85)^{-28} \approx 95$.

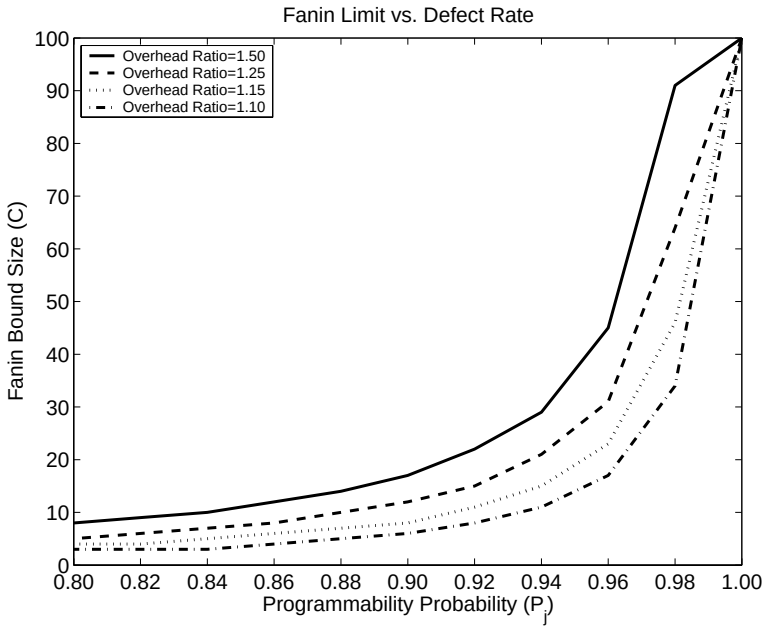


Figure 4.5. Fanin Limit to Tolerate Different $\frac{|W|}{|F|}$ Overhead Ratios versus P_j Values for $|F| = 100$

Ultimately, we must map all the OR-terms in each array; consequently, equation 4.5 is a weak bound. In [65] we derive tighter bounds useful for design. figure 4.5 show the fanin limits for different area overhead ratios based on these tighter bounds.

4.2.3 Guaranteeing Sparseness during Mapping

We control the fanin bound, C, during logic mapping. A typical step in mapping for clustered PLA designs, such as these nanoPLA blocks, is to group the logic into clusters which can be feasibly implemented by the base logic blocks. For example, PLAMAP [67] can take in a netlists of primitive logic gates and cover the logic while assuring that each logic cluster does not exceed architectural limitations, including:

- I the maximum number of inputs to a cluster
- P the maximum number of P-terms in each cluster
- O the maximum number of outputs from a cluster
- P_{max} the maximum number of P-terms which fan in to any OR-term

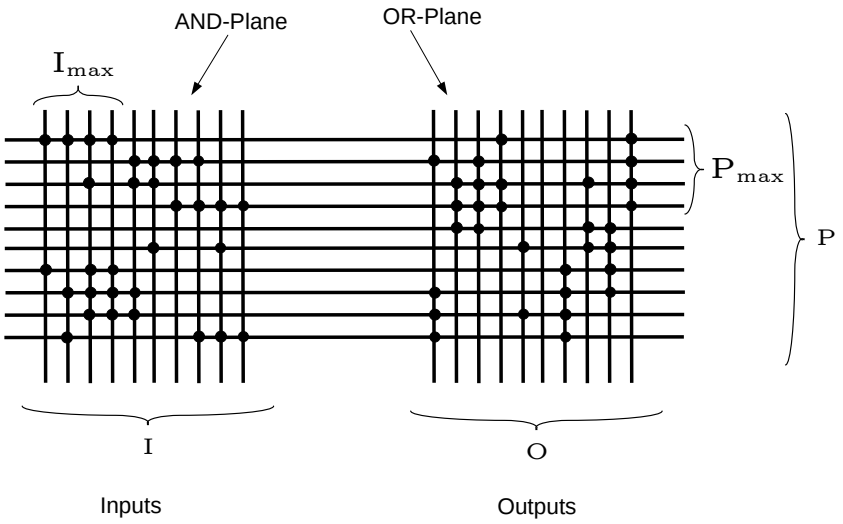


Figure 4.6. The parameters of PLAMAP on a nominal PLA: I, P, O, I_{max}, and P_{max}

We might also wish to separately bound:

- I_{max} the maximum number of inputs which fan in to any particular P-term

figure 4.6 illustrates these parameters on a nominal PLA.

In our nanoPLAs, the critical P-term fanin limit, C, corresponds to the number of inputs, I_{max}, for the AND-plane and the OR-term fanin, P_{max}, for the OR plane. Consequently, once we know the effective defect rate for a given fabrication technology we can set P_{max} and I_{max} appropriately and synthesize logic clusters that are guaranteed to meet the appropriate fanin bounds (i.e., equation 4.5 and extensions in [65]). Restricting I_{max} and P_{max} may lead to an increase in the number of blocks needed by the circuit and the logic depth; this must be compared against the alternative of using less sparse logic and paying a larger mapping overhead.

4.2.4 Interconnect Nanowire Integration with Logic Resources

section 2.3 and [33], show interconnect signals are routed through the same nanoPLA blocks that are used as computation block at the same time. Once the design is par-

titioned and and clustered into nanoPLA blocks, the interconnect signals are routed through the nanoPLA blocks. The interconnect routing configuration is added to the logic configuration of the corresponding PLA blocks. Since the routing configuration only has single cross-point per nanowire, they will have the smallest fanin, $C = 1$, and therefore, will be mapped once all the P-terms of the PLA configuration is mapped. Routing interconnects through the logic blocks need very limited area overhead for defect-tolerance purpose, since the fanin is only 1 (equation 4.5 and extensions in [65]).

4.3 Experimental Results

To characterize the impact of defective cross-points, we mapped 16 of the benchmarks in the Toronto20 [68] place-and-route challenge suite to nanoPLA arrays with varying cross-point and wire defect rates. We used PLAMAP [67] to create clusters. To control I_{max}, we asked PLAMAP to generate ($I = I_{max}, P = P_{max}, O = 1$) single-output covers, then used T-Vpack [48] to combine the single-output covers into $I = 20$, $P = 64$ clusters.

Table 4.1 shows relative area versus defect rate for the 16 designs. For all designs, the additional overhead of fanin bounding and defect-tolerance is below 120% up to a defect rate of 10% ($P_j \geq 0.9$).

The modest overhead for most designs arises from the fact that they can be mapped to bounded fanin without significantly increasing the number of clusters required to cover the logic task. Table 4.2 shows how the design sizes scale as we map to different fanin bounds: $C = I_{max} = P_{max}$. Many of the designs show no increase as we tighten the fanin bounds. In fact larger fanin bounds can lead to excessive logic duplication and increased area in many cases [67]. For a few designs there is an appreciable increase in block count as the fanin bound is tightened, and this is the major factor accounting for the area overhead jumps for `ex1010`, `pdc`, `s298`, and `spla`.

Table 4.3 summarizes the composite area overhead of `pdc`, the design with the

Table 4.1. Relative Area versus P_j (nanowire pitch is 10 nm; reliable superstructure pitch is 105 nm.)

	P_j			
Design	0.85	0.9	0.95	1
alu4	1.81	1.64	1.00	1.00
apex2	1.19	1.19	1.00	1.00
apex4	1.30	1.16	1.00	1.00
bigkey	1.00	1.00	1.00	1.00
clma	1.00	1.00	1.00	1.00
des	1.00	1.00	1.00	1.00
dsip	1.00	1.00	1.00	1.00
elliptic	1.00	1.00	1.00	1.00
ex1010	3.81	2.15	1.00	1.00
ex5p	1.00	1.00	1.00	1.00
frisc	1.00	1.00	1.00	1.00
misex3	1.31	1.31	1.00	1.00
pdc	4.75	1.79	1.00	1.00
s298	1.84	1.84	1.00	1.00
seq	1.20	1.12	1.00	1.00
spla	3.46	1.83	1.00	1.00

Table 4.2. nanoPLA Mapped Block Count versus Fanin Bound

	$C = I_{max} = P_{max}$						
Design	4	6	8	10	12	16	48
alu4	107	110	98	81	52	26	19
apex2	136	150	152	167	179	178	183
apex4	108	115	123	47	48	34	36
bigkey	99	111	139	147	132	165	168
clma	454	513	557	634	663	646	520
des	102	110	114	130	149	158	120
dsip	70	88	82	108	87	108	79
elliptic	162	209	225	289	338	397	262
ex1010	380	404	402	233	272	351	81
ex5p	88	98	50	50	46	30	8
frisc	213	229	244	285	326	376	276
misex3	97	109	105	102	83	44	37
pdc	291	322	267	292	304	160	41
s298	81	85	84	88	90	102	57
seq	121	135	140	143	138	111	76
spla	211	235	201	219	221	109	33

Table 4.3. Relative Area for PDC Benchmark as a Function of P_{wire} and P_j

P_{wire}	P_j				
	0.8	0.85	0.9	0.95	1
0.80	25.01	21.07	17.11	14.52	11.02
0.85	10.75	9.17	7.34	6.28	4.71
0.90	4.06	3.46	2.79	2.39	1.80
0.95	2.26	1.91	1.55	1.31	1.00
1.00	2.26	1.91	1.55	1.31	1.00

largest overhead, as a function of both wire and junction defects. The composite area overhead at $P_{wire} = 0.90$ and $P_j = 0.90$, is less than a factor of three. After accounting for overheads at these defect rates, the design achieves over 100 times greater density than a 4-LUT based FPGA implementation in 22 nm CMOS. This density benefit is typical of these designs [33].

4.4 NanoPLA Block Sparing

On top of the individual wire sparing described above, we will likely still need to spare entire nanoPLA blocks.

- Larger scale contaminants during assembly or large cluster faults may leave an entire nanoPLA block unrepairable; these defects are not appropriately modeled as independent, random junction or wire defects.

- As discussed, we can guarantee high statistical yield of each nanoPLA block, but with millions of nanoPLA blocks in an array, some blocks will not be repairable.

Now that we have used the wire sparing techniques to bring the yield of the nanoPLA blocks to a respectable level (e.g., P_{MofN}=99%), we can use the M-choose-N sparing idea at a higher level on the nanoPLA blocks. For example, Lach et al. describe a strategy for tolerating FPGA defects by omitting one logic block from a $k \times k$ tile of FPGA logic blocks and generating logic configurations which accommodate the failure of each physical logic block in the tile [69].

section 3.3 suggested using a built-in reliable microprocessor responsible for localizing defects and performing the procedure of figure 4.4. We will give the device

the logical configuration for each nanoPLA block. The on-chip reliable microprocessor will then run the test to discover the present and functional addresses of each nanoPLA block and perform the local matching procedure (figure 4.4) necessary to assign logical configuration to physical nanowires. In this way, the device never stores the entire defect map for the component, but simply rediscovers it one nanoPLA block at a time, and it never exposes the user to the defect details of the array.

4.5 Summary

In this chapter we illustrated how to tolerates defect in the nanoPLA architecture. The defects are abstracted to two defect models: broken wires, and nonprogrammable open cross-points. We used detect-and-reconfigure technique to isolate the defective nodes and program around them. This technique takes only three-fold area to tolerate 10% broken wires and 10% nonprogrammable cross-points in the worst-case, where in the average for typical design it is about 30% overhead. Using a built-in reliable microprocessor to discover the defect map and stochastic nanowire addresses facilitates the testing process. The microprocessors never stores the whole defect map, it simply discovers it one nanoPLA block at a time. This microprocessor is further used to perform the greedy mapping procedure to program the nanoPLA blocks.

Chapter 5

Transient Fault-Tolerant Design with Rollback Technique

Chapter 2 pointed out that the transient fault increases as the reason of feature size and supply voltage scaling. In this chapter we propose a fault-tolerant design to tolerate high fault rates. We propose a fine-grained rollback technique to tolerate transient faults. We used temporal redundancy to contain the spatial redundancy. The potential impact on the system throughput is further reduced by using streaming buffers between the blocks. We mainly compare our fine-grained fault-tolerant design with Majority-Multiplexing technique which gathered much attention recently. We illustrate that this technique can bring close to an order of magnitude improvement compared to the related nanotechnology fault-tolerant works. Our technique provides the flexibility to choose the right area–throughput trade–off to adjust the design with the desired area overhead or throughput loss limit.

5.1 Design Structure

Rollback recovery has been widely used for large block sizes with coarse-grained recovery, typically at the processor level [26][27, 28]. In this section we design a *Fine-grained* rollback technique that can tolerate higher fault rates than previous rollback techniques and achieve highly reliable system. Later in this section, we show how the block size affects the reliable system design and why small blocks (i.e., at logic level size) are essential, to achieve reasonable area overhead and system

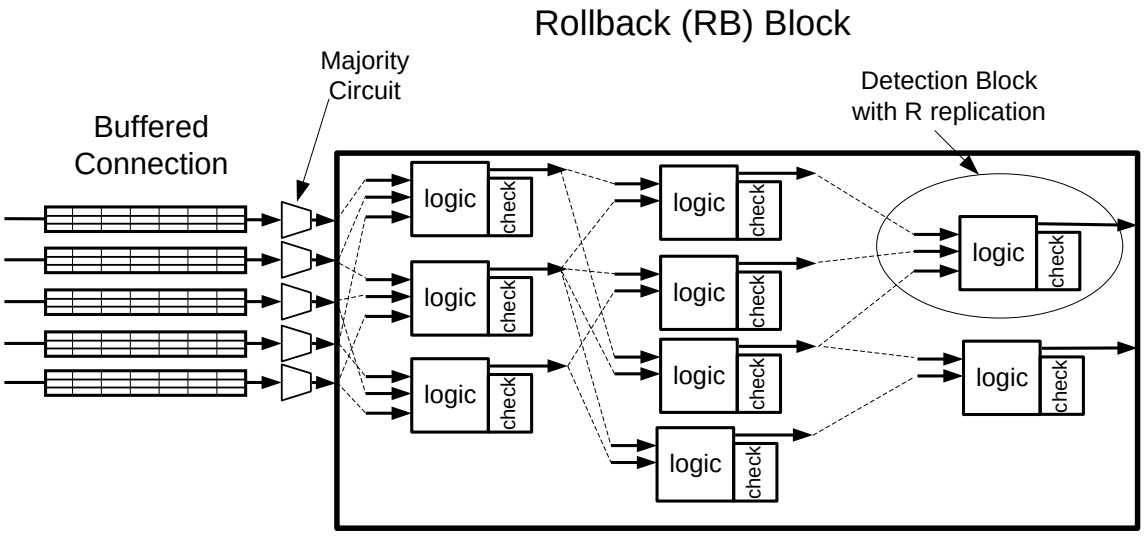


Figure 5.1. The RB block consisting of detection blocks.

throughput.

This fined-grained rollback design has a two-level hierarchical structure, as shown in figure 5.1. At the base, the system is partitioned into fine-grained blocks called *Detection Blocks*. Each detection block has an embedded fault detection circuit to guarantee detection of a certain number of errors inside the block. At the next level the detection blocks are clustered to form a *Rollback* (or *RB* for short) Block. Each RB block guarantees the correctness of its output signals by performing rollback operations. Once a detection block inside an RB block signals an error, all the blocks inside the RB block stop their normal processes and the RB block *rolls back*, meaning it returns to a previously error free state, recovers the inputs which arrived subsequent to that state, and repeats the affected operations to generate the correct result.

The interconnects between the RB blocks are *Buffered Connections* that are designed to facilitates relatively independent operation flow between the RB blocks; i.e., the buffered connection provides buffer capacity between RB blocks, allowing an RB block to continue while an adjacent RB block is in rollback mode.

The above building blocks: *Detection Block*, *RB Block*, and *Buffered Connections* are developed in detail in the rest of this section.

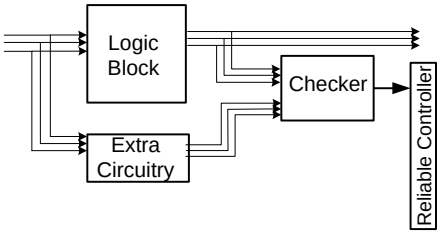


Figure 5.2. The general structure of concurrent error detection scheme.

5.1.1 Detection Block

The detection block consists of the logic circuit block protected with enough redundant data to make errors in the logic circuit identifiable. A checker circuit follows the original circuit block and the redundant logic circuitry to detect any error at the output signals of the logic circuits. The main idea behind error detection is to compute redundant data *concurrently* with the main computation and compare the main and the redundant output signals, detecting any error in the main computation (figure 5.2). There are many different ways to generate the extra information to protect the main block [70]; e.g., parity signals, error correcting codes, and logic replication.

Here we use a simple error detection technique, *Replication with Comparison*. It consists of multiple (R) independent copies of the main logic block, followed by a checker, which detects any disagreement among the copies of the logic block. We select R based on the device fault rate, P_f, and the desired FIT rate.

The *Replication with Comparison* technique is a general-purpose structure and does not demand any special design specification, while design-specific alternatives may provide more lightweight and less expensive solutions. In the present article we show that even with this basic and nonoptimized detection scheme the rollback recovery will require less overhead than feed-forward fault-tolerant technique. The area overhead can be further reduced by using more optimized detection techniques, such as a multiple parity scheme [70], as long as the encoder and the decoder take small area and short delay.

If the checker block is equally error prone as the logic blocks then the checker needs to be protected as well (see figure 5.3). Replicating the checker block and

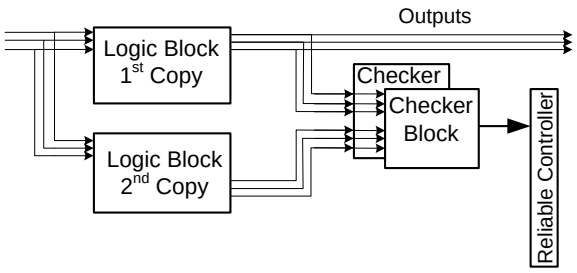


Figure 5.3. A detection block, consisting of two copies of logic block followed by two copies of checker blocks.

reporting an error when any of the checker block copies reports an error decreases the probability that errors in the checker will go undetected.

A checker design which can detect any disagreement between R copies of the logic block is simple. It basically computes the AND and OR functions of the R copies of each logic block outputs. If the R signals are identical then the AND and OR functions of those signals have the same value. However, if there is any disagreement between the signal values, the AND function holds "zero" and the OR function holds "one". This is illustrated in figure 5.4 by a truth table. This implementation of the checker is minimal for the nanoPLA structure and other two-level implementation as will be shown in section 2.3.

A detection block is the combination of the R copies of logic blocks with the R copies of checker blocks. The structure of the detection block is shown in figure 5.3 for $R = 2$. In this example each detection block detects any single error and most cases of multiple errors inside the block. For any value of R, each detection block detects any $R - 1$ errors and most cases with greater number of errors. One important feature of this design is that the checker blocks are placed off the normal computational path, hence the latency of the checker block does not add to the latency of the normal system operation; checker latency only affects the operational latency when an error is detected.

Signals	Values		
$a_i's$	all 0	all 1	mixed
OR(a_i)	0	1	1
AND(a_i)	0	1	0
AND(OR(a_i),$\overline{\text{AND}(a_i)}$)	0	0	1

Figure 5.4. The truth table of the checker block logic. The checker block reports any disagreement among the inputs, $a_i's$. The inputs $a_i's$ are R copies of an output signal from a logic block. If all of the inputs hold the same value, the outputs of the AND and OR will be the same, otherwise the outputs of the AND and OR signals will be complements of each other. The last row of the table shows the error indicator function; on detecting and error, it holds the value of "1".

5.1.2 Rollback Block

When an error is detected in one of the detection blocks inside an RB block, the control circuit stops the computation of all the detection blocks inside the RB block and forces the RB block to repeat the affected process and generate the correct result. The control circuit guarantees the correctness of the rollback flow and uses the result of the checker block to switch the block operation between rollback and normal modes. The correctness of the system flow depends on the reliability of the control circuit, and therefore, the control circuit must be designed with higher reliablility. For example we can implement the control circuitry with reliable, coarse-grained CMOS even when otherwise using nanoscale sublithographic devices for the compute block. The reliable devices take greater area but since the control circuit is a small fraction of the detection block, its area overhead is negligible compared to the area of the compute blocks.

When an error is detected, the reliable controller stops the normal operation of the circuit, resets the pointer of the input buffer to the input data associated with the last correctly retired output, and recomputes the operation from that state to recover the corrupted data. How far the inputs roll back depends on the depth of the RB block and the latency of the logic blocks and checker blocks.

figure 5.5 illustrates the latencies of different parts of an RB block that affect the rollback design. When an error is detected, the detection is delayed by the checker block latency (D_c cycles). Furthermore the data needed to recover the erroneous

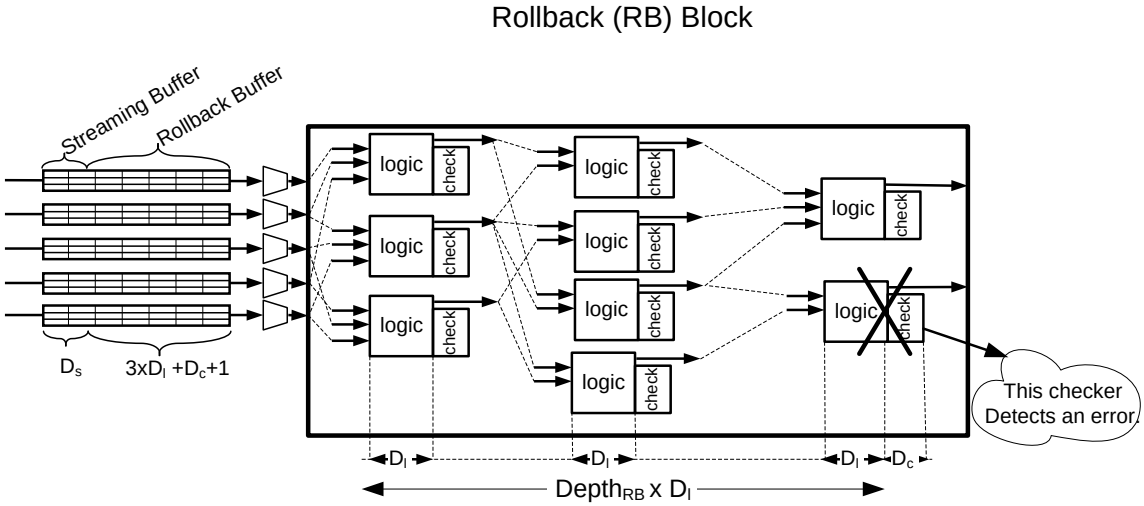


Figure 5.5. RB block with timing parameters.

computation may have come from the RB block inputs after multiple levels of the logic block latency; e.g., figure 5.5 shows a case where the error is detected 3 levels deep in the block and each level has delay of D_l cycles. So the inputs should rollback for $3 \times D_l + D_c + 1$ cycles (with one extra cycle being for the reliable controller to perform the feedback). In general the inputs of the RB block must be registered to support correct rollback operation for the following number of cycles,

$$D_R = Depth_{RB} \times D_l + D_c + 1, \tag{5.1}$$

where $Depth_{RB}$ is the number of levels in the RB block (figure 5.5). Therefore, we need a D_R-deep buffer for any of the RB block inputs. We call these D_R buffers *Rollback Buffers*.

The system runs fully pipelined at high throughput until an error is detected. Then the system freezes and spends a relatively long time (i.e., $D_R = Depth_{RB} \times D_l + D_c + 1$) recovering from the error. Although this situation happens infrequently, it can have a severe impact on the system throughput. In the next section we describe how streaming buffer interconnects reduce the impact on the system throughput.

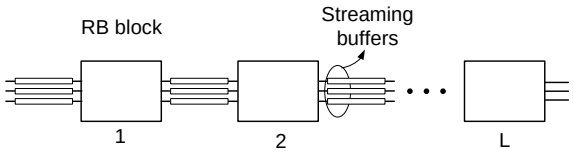


Figure 5.6. A simple structure model designed with buffered connections between RB blocks.

5.1.3 Streaming Buffer

When an RB block stops to rollback, the other RB blocks in the system must also stop due to the data dependencies between the blocks. Consequently the system throughput drops to zero whenever any of the blocks is in rollback mode. In large systems with many RB blocks, this can potentially cause high throughput loss.

In order to avoid much of this throughput loss in large systems, we use streaming connections or *Streaming Buffer* between the RB blocks. The *Streaming Buffers* allow most of the RB blocks to continue their normal process while some of them are in rollback mode. Note that the *Streaming Buffer* are extra buffers added to the required *Rollback Buffers* of size D_R (equation (5.1)). For example, if a *Streaming Buffer* of depth D_s is embedded at the inputs of an RB block, then the total depth of the buffer at the inputs of this block is $D_s + D_R$.

To build intuition on how the streaming buffers prevent throughput loss, we consider a simple chain structure as an example (see figure 5.6). This structure is also considered in [32] and [31]. It is a chain of L levels of RB blocks separated by an adequate number of buffers. Specifically let us consider a simple scenario that reveals the improvement in throughput due to the streaming buffers. Assume some errors are detected inside the Lth and the 1st RB blocks and they start the rollback process at time t_1 and t_2 respectively (figure 5.7). If the rollback time takes D_R cycles, in the case of no streaming buffer the system is idle for $2 \times D_R$ cycles. Therefore, the system throughput during t_1 to t_3 is $(t_3 - t_1 - 2 \times D_R) \,/\, (t_3 - t_1)$.

In the presence of streaming buffers the blocks before the Lth block continue their normal process while the Lth block is in rollback mode from t_1 to $t_1 + D_R$ and the data is stored in the intermediate buffer between the $L-1$st and the Lth blocks.

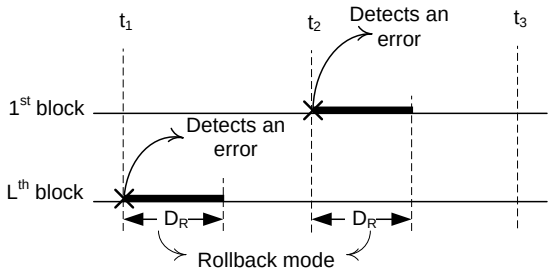


Figure 5.7. This is a timing diagram of the system. It shows a simple scenario where the streaming buffer can improve throughput. The L^{th} block detects an error at time t_1 and stays in rollback mode for D_R cycles, then the 1st block detects an error at time t_2 and switches to rollback mode.

Later when the first block is in rollback mode during t_2 to $t_2 + D_R$ the Lth block continues its normal process by consuming the saved data in the buffer between the $(L-1)$st and the Lth blocks. Therefore, the total throughput loss is only D_R cycles, and the throughput during this period is $(t_3 - t_1 - D_R) / (t_3 - t_1)$. The streaming interconnects allow the blocks in the chain to run more independently and therefore, as you can see, the final throughput of $(t_1 - t_3 - D_R) / (t_1 - t_3)$ is the same as the average throughput of a single block. That is, the streaming buffers reduced the throughput loss by half in this example. Section 5.4 uses a simulation to estimate the best depth of the streaming buffers, D_s, to achieve acceptable throughput with reasonable area overhead.

5.1.3.1 Reliable Buffered Interconnect

Each buffered interconnect consists of two parts: Streaming Buffer and Rollback Buffer, each similar to a shift register of length D_s and D_R respectively. Figure 5.8 shows how the two shift registers are connected to generate the buffer structure. In normal operation mode the data flow through the Streaming Buffer. One new data values are shifted into the streaming buffer from the previous RB block and one datum is shifted out to the next RB block. As long as both the previous and the next RB block are in normal operation mode, the number of data elements in the streaming buffer stays the same. The number of data elements in the streaming buffer can be anything from 0 to $D_s - 1$.

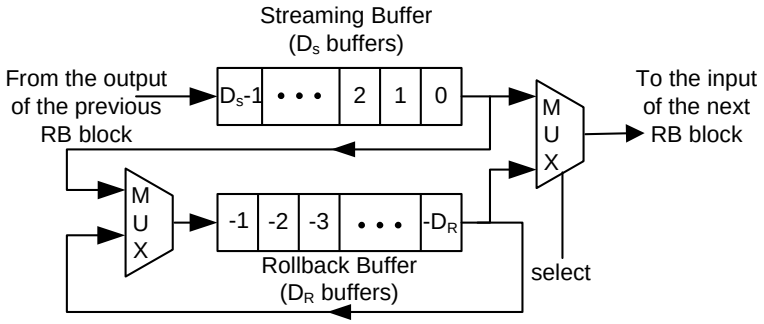


Figure 5.8. This figure shows a simple block diagram of a buffered connection. Each buffered connection consists of two parts: the rollback buffer and the streaming buffer. The numbers on the buffer elements represents the order of the data, "0" representing the data currently being processed in the RB block following the buffer. Each of the buffer elements in the Streaming buffer or Rollback Buffer has structure similar to figure 5.9.

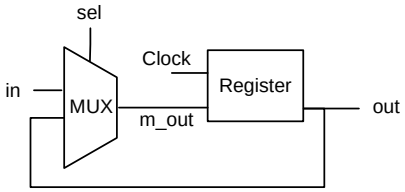


Figure 5.9. A simple block diagram of a buffer element.

If the next block detects an error and starts the rollback operations, it will stop consuming data from streaming buffer and will start consuming the data from D_R cycles ago which is stored in rollback buffer. During the period that the next RB block is in rollback mode, the previous RB block continues generating data and storing them in the streaming buffer until it fills up.

The streaming and reliable buffers are each composed of a chain of buffer elements shown in figure 5.9. Each buffer element consists of a register and a multiplexer (figure 5.9). The multiplexer allows either the new input or the current value into the buffer. If the buffer is in *shift* mode, the multiplexer selects the new input value, which replaces the current value. If the buffer is in *keep* mode the multiplexer selects the current value and the current value will be restored.

The data coming out of the buffered connections into the RB block must be correct. To guarantee the correctness of the buffered connection data, an error correcting

technique is embedded in the buffers.

For simplicity and consistency with the error detection technique in the logic blocks we use the majority voting scheme for error correction in the reliable buffer. In this scheme multiple copies of the data are stored and a voter circuit following the multiple copies determines the majority among these copies. This scheme needs large data redundancy (i.e., minimum of 3) but the encoder (replicator) and the decoder (voter circuit) are relatively cheap when the replication factor is small.

We call the replication factor for each buffer element R_{buf}. The minimum R_{buf} for majority voting is 3 and it grows for high fault rates. The voter circuit receives all the R_{buf} copies of buffer element. It computes the majority of the R_{buf} input signals. This is the value of at least $(R_{buf}/2 + 1)$ of the inputs.

If there were a single voter circuit for every R_{buf} copies of the buffered data, the voter circuit would be a single point of failure and the reliability bottleneck; the reliability improvement achieved by multiple copies of buffer element will be wasted. To prevent this effect, the computation of the voter circuitry must also be protected. Therefore, similar to the logic blocks the voter circuit is replicated into R copies and the correctness of the results is verified by checker blocks following them. When a checker block identifies a disagreement among the voter results, the recovery process is similar to the case when an error is detected in a logic block; that is, the process of the following RB block is stopped, and the voter circuits repeat the operation to identify the correct value of the majority of the incoming signals from the buffered connections.

5.1.4 Block Size

Key parameters in rollback system design are the detection block size and RB block size. The detection block affects the likelihood of detecting transient faults and, hence, determines the reliability of the system. The RB block size controls the latency of rollback and the rate at which rollback occurs and, hence, is largely responsible for determining the throughput of the system. By treating the detection and RB block sizes independently, we can separately engineer the system for reliability and

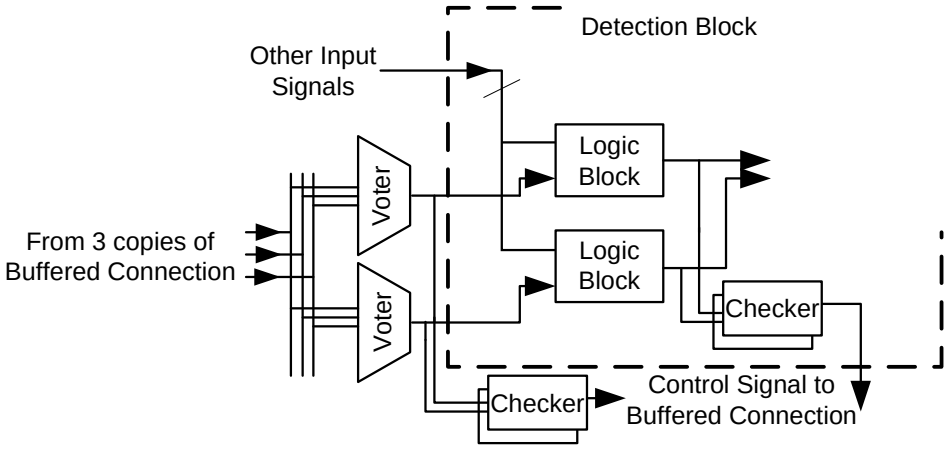


Figure 5.10. This figure shows how the reliable voters are structured and connected to separate replicas of logic blocks in each detection block.

performance. Both block sizes affect the overhead in the system.

As we will see in section 5.3 the reliability of a detection block for a fixed device fault rate depends on the replication factor and the block size. Larger replication factors and smaller block sizes increase the reliability of the detection block. Therefore, for a fixed reliability target and device fault rate, we have to limit the detection block size to keep the required replication factor small. Nevertheless interconnect locality, fixed block overheads, and reliable control circuitry make the smallest block sizes (e.g., single P-terms or even P-terms with only two inputs) inefficient [64]. Therefore, there is a practical lower bound on efficient block sizes. The area minimizing block sizes for various nanoPLA designs is shown in [64]. These efficient designs have fine-grained block size (i.e., logic level). Here we try to design the rollback system where the size of the detection blocks is close to this efficient size.

The RB block size affects the throughput and area overhead of the rollback system. The impacts of the RB block size are summarized in the following categories:

1. In rollback mode, the operation of the block will be recomputed. The main part of the rollback latency is the latency of the main block, which was shown in equation (5.1). Small block size, or more specifically small block depth, $Depth_{RB}$, helps keep the rollback latency short and, in turn, keeps D_R small.

2. The larger the block is, the higher the probability of transient fault occurrence

in the block, and therefore, the higher rollback frequency. If the device failure probability is P_f and the block has N devices, the block fails with the probability below if we ignore fault masking,

$$P_{RB} = 1 - (1 - P_f)^N. \tag{5.2}$$

When $N \times P_f \ll 1$, the failure probability is approximately $N \times P_f$, and we see that rollback frequency grows linearly with the size of the block.

3. The RB block size also affects the area overhead; but in different directions. Larger block size results in smaller area overhead by reducing the number of buffered connections. Large blocks tends to enclose connections between the detection blocks inside it, thus reducing the number of inter-RB-block connections which are implemented in buffered connections.

As you can see, the first two effects above favor small RB block size to achieve high system performance, while the last one favors large RB block size to reduce area overhead; this suggests the RB block size selection provides a tradeoff between area and time. When fault rates are low, we can employ large RB block sizes to minimize area overhead, but as fault rates increase, the RB block sizes must decrease to maintain performance, at the cost of additional area overhead. Section 5.4 quantifies this tradeoff.

Note that the optimum size of the RB block is much larger than the detection block size. This is the main motivation for designing fine-grained rollback system in two hierarchical levels with two different block sizes. We can have larger RB blocks which amortize the overhead of streaming inputs without decreasing reliability or increasing error detection overhead.

5.2 NanoPLA Implementation

In this section, the implementation of the fine-grained streaming rollback design will be demonstrated on a nanoPLA substrate. As explained in chapters 2 and 4, the

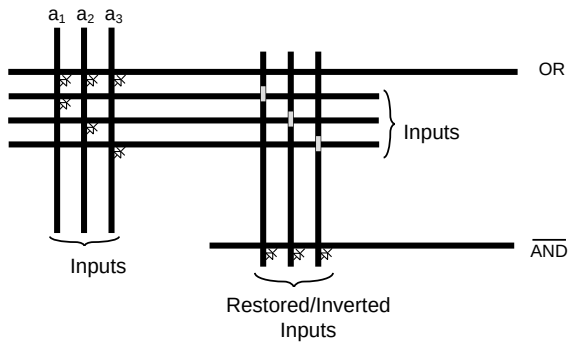
nanowires operate in pairs; the nanowires in the logic plane generate the wired-OR logic and the nanowires in the following restoration plane invert their value and restore their voltage level. We consider each pair of nanowires and the corresponding input diode switches and the gate-controlled junction in between nanowires as a unified element. We define the fault rate, P_f, the probability that this unified element is erroneous. We also measure the area of our system based on the number of nanowire pairs.

5.2.1 Detection and Rollback Block

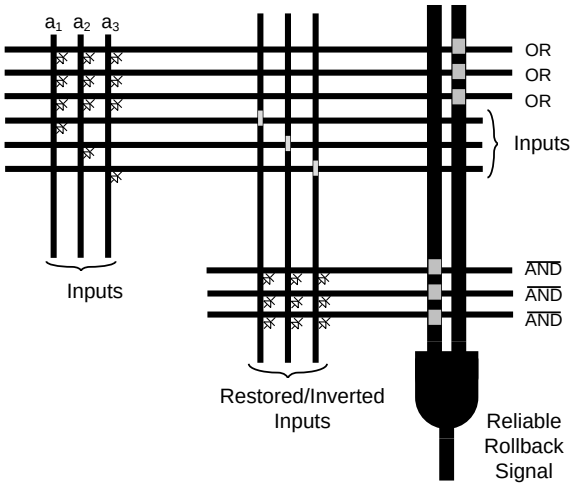
The detection block developed in the previous section is implemented on the nanoPLA substrate. Multiple logic blocks may be implemented by each nanoPLA block; each logic block is replicated R times and followed by the checker blocks which are also implemented in nanoPLA blocks. The checker function consists of an R-input AND function and an R-input OR function and can easily be implemented in two-level logic as described in section 5.1.1. Figure 5.11(a) shows the checker design implemented in a nanoPLA block with $R = 3$. The nanoPLA checker block needs one P-term to implement the R-input OR function and R P-terms and one OR-term to implement the R-input AND function (figure 5.11(a)). Overall a checker circuit needs R P-terms and 2 OR-terms to check the agreement between R signals, which in total takes $R+2$ pairs of nanowires.

Since the checker size is relatively small the R copies may be integrated into one nanoPLA block. As shown in figure 5.11(b), the R copies of the checker takes, $R \times (R + 2)$ nanowire pairs.

The final outputs of the checker block connects to reliable control circuitry through a wired-OR (figure 5.11(b)) to generate the final reliable feedback control signal. That is, we want to signal a rollback when any of the checker ouputs signals an error; the nanoscale checker outputs are wired via diode connections to a reliable, lithographic-scale wire so that it is pulled high when any of the checker outputs is high. Strictly speaking the efficient implementation shown in figure 5.11(b) implements $\left(\overline{and_0} + \overline{and_1} + \overline{and_2}\right) \cdot (or_0 + or_1 + or_2)$ rather than $\overline{and_0} \cdot or_0 + \overline{and_1} \cdot or_1 + \overline{and_2} \cdot or_2$,



(a)



(b)

Figure 5.11. (a) The checker block implemented with nanowires taking $R+2$ ($R+2=5$ in this example). (b) The R copies of the checkers integrated with reliable lithography-scale circuitry.

where and_i's are the ANDs and or_j are the ORs; the extra cross terms should also always be zero in a fault free case, so these additions do not cause any false rollbacks.

The detection blocks, including logic blocks and checker blocks, are clustered to form an RB block. The interconnect signals among the detection blocks inside an RB block are routed in the bundle of R nanowires. The interconnect signals are implemented on the nanoPLA planes. The details of how interconnect routing can be implemented on nanoPLA planes is provided in [13].

5.2.2 Buffer Connection

The buffered connection, as described in section 5.1.3.1, is a chain of buffer elements each consisting of a multiplexer and a register. Figure 5.12 shows how this can be implemented on a nanoPLA substrate. The details of the buffered connection implemented on the nanoPLA can be found in [13]. This design takes 4 pairs of nanowires per cell and multiple buffer element can be implemented in one nanoPLA plane.

The voter circuit following a buffered connection is an OR function of all the possible $(R_{buf}/2+1)$-input AND gates from R_{buf} signals. Therefore, the number of AND gates in the voter circuit is

$$A_{voter}\left(R_{buf}\right) = \left(\begin{array}{c} R_{buf} \\ (R_{buf}/2+1) \end{array} \right). \tag{5.3}$$

When R_{buf} is small, the above number is not very large. For large values of R_{buf}, there are alternate options that can provide more compact implementations (as small as $O\left(R_{buf}\right)$) at the expense of greater checker latency, D_c. Figure 5.13 shows the voter circuit for $R_{buf}=3$. It has 3 AND gates (P-terms) followed by an OR-term.

Using the above design, the number of the nanowire pairs required for a buffered connection of depth D_R+D_s including R copies of the voter circuit is

$$A_{buffer}+A_{vote} = Size_{buf} \times (D_R+D_s) \times R_{buf} + R \times \left(\begin{array}{c} R_{buf} \\ (R_{buf}/2+1) \end{array} \right).$$

where $Size_{buf}$ is the number of nanowire pairs in one buffer element which equals 4.

5.3 Reliability and Area Analysis

In this section we analyze the area and reliability of our fine-grained rollback scheme. The main goal in this section is to determine how large the replication factor must be to achieve a desired FIT rate. To do so, this section is organized as follows: we first

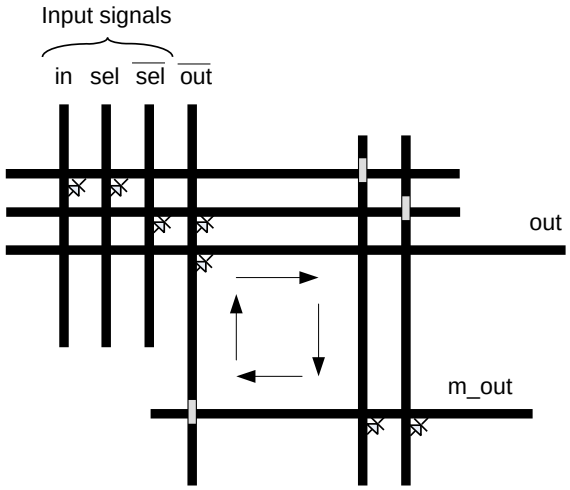


Figure 5.12. This figure shows a shift register element implemented with nanowires. The schematic view of this design is shown in figure 5.9. The *m_out* signal is an intermediate signal (output of the MUX), which is routed into the input plane to generate the final output signal. This implementation needs 4 pairs of nanowires.

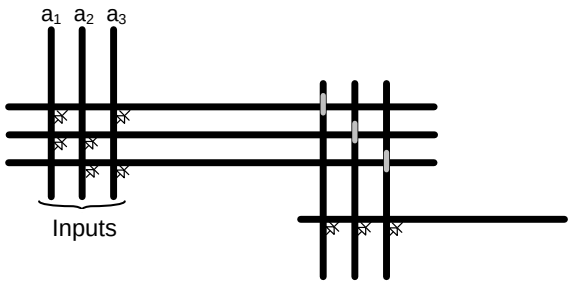


Figure 5.13. The voter circuit designed with nanoPLA, for redundancy factor $R_{buf} = 3$.

compute the *undetected error probability* of the system using a bottom-up approach; i.e., we compute the undetected error probability of the building blocks of the system from the base-level *detection block*, to *RB block*, to the complete system. Once we have the undetected error probability of the system and know the system frequency, we can compute the expected number of undetected errors in one billion operation hours, which is the FIT rate of the system.

5.3.1 Error Probability of a Detection Block

To compute the undetected error probability of a detection block, we first have to compute the error probability of its building blocks: *logic blocks* and *checker blocks*.

Here we consider each logic block as the *logic cone* of each output signal. The *logic cone* of an output signal is the set of all the logic elements required to generate the output signal and therefore is the only part influencing the output signal.

With a conservative estimate, an OR-term (an output signal of a logic block in the nanoPLA architecture) has an erroneous result if any element inside the block is erroneous. It is conservative since it does not consider the effects of any kind of error masking, e.g., *logic masking*, *electrical masking*, or *latching-window masking* [71]. *Logic masking* is when the error might not propagate to the output because a gate on the path is not being sensitized to facilitate the propagation. *Electrical masking* is when an error is attenuated passing through multiple gates on the path to the output. Finally *latching-window masking* is when the fault effect reaches the output but the latch is not open to store the erroneous value.

Using this conservative assumption, any fault in the logic block will result in an error in the OR-term signal. Therefore, the probability that an OR-term has an erroneous value is

$$P_{or\_err} = 1 - (1 - P_f)^{N_{logic}}, \tag{5.4}$$

where N_{logic} is the size of the logic cone of the OR-term. With a similar calculation

the error probability of a checker block is

$$P_{cb\_err} = 1 - (1 - P_f)^{R+2}, \tag{5.5}$$

where $R + 2$ is the size of the checker block as shown in section 5.2.1.

Now that we know the error probability of building blocks of a detection block, we can compute the probability of an undetected error in a detection block. In a detection block with R copies of a logic block and R copies of a checker block, an erroneous OR-term is undetected under two scenarios: First, when all the R copies of the OR-term are erroneous and all the checker blocks are correct, in this case no disagreement among the OR-term copies can be detected. Second, when at least one of the OR-term copies are erroneous but all the R checker copies are erroneous and fail to detect the error. These two cases generate the undetected error probability of a detection block as below,

$$P_{det\_block\_und\_err} = (P_{or\_err})^R \times {P_{cb\_crr}}^R + \left(1 - (P_{or\_crr})^R\right) \times (P_{cb\_err})^R . \tag{5.6}$$

Note that P_{or_crr} and P_{cb_crr} are the probability that an OR-term signal or a checker block is correct, these are the complement of P_{or_err} and P_{cb_err} respectively, which are computed in equations (5.4) and (5.5).

Remember that the reliability of the voter circuitry following each buffered connection at the input of an RB block is provided by replication of the checker circuitry. Therefore, the voter circuitry generates an undetected error in the same scenario as a logic block in a detection block does: (1) When all the R copies of the voter circuitry are erroneous, which results in identical erroneous output signals, and all the checker copies are correct. (2) When at least one of the R copies of the voter signal is incorrect but all the checker copies fail to detect the erroneous voter circuit copy. This probability is similar to equation (5.6)

$$P_{vote\_block\_und\_err} = (P_{vote\_err})^R \times {P_{cb\_crr}}^R + \left(1 - (P_{vote\_crr})^R\right) \times (P_{cb\_err})^R . \tag{5.7}$$

P_{vote_crr} and P_{vote_err} are the probabilities that a voter circuit is error free or erroneous, respectively, which is essentially the same as a logic block's with $N_{logic} = \begin{pmatrix} R_{buf} \\ (R_{buf}/2+1) \end{pmatrix}$ nanowire pairs.

5.3.2 Undetected Error Probability of an RB Block

Each RB block includes a number of detection blocks. It also includes a number of voter blocks following any incoming buffered connection. An RB block has an undetected error in it if any of its detection blocks or the voter blocks has an undetected error. Therefore, the undetected error probability of an RB block with B detection blocks and I inputs is

$$P_{rb\_block\_und\_err} = \left(1-(1-P_{det\_block\_und\_err})^{B}\right) \bigcup \left(1-(1-P_{vote\_block\_und\_err})^{I}\right).$$

Note $\bigcup$ is used here to denote a probability union calculation, where we avoid counting the overlap probability twice; that is

$$A \bigcup B \equiv A + B - A \cdot B. \tag{5.8}$$

5.3.3 Buffered Connection Reliability

The error probability of a buffer element depends on the number of consecutive cycles that a buffer element holds a single logic value in the system, and therefore, it is susceptible to errors. In order to have a realistic estimate on the number of consecutive cycles that a buffer element holds a single value, we simulate the performance of the system. This simulation is explained in section 5.4 for the same chain structure introduced in section 5.1. The error probability that a buffer element has an erroneous value in a single cycle is

$$P_{buf\_elem\_err\_per\_cycle} = 1-(1-P_f)^{Size_{buf}}, \tag{5.9}$$

where $Size_{buf}$ is the number of devices in one buffer element. Once we have the maximum number of consecutive cycles that a buffer element holds a single value, we can compute the error probability of a buffer element as below, where c is the number of those cycles

$$P_{buf\_elem\_err} = 1 - (1 - P_{buf\_elem\_err\_per\_cycle})^c, \tag{5.10}$$

A protected buffer element with replication (R_{buf}) has an undetected error when the number of erroneous replicas are more than half of the replication factor (R_{buf}), and therefore, the majority computes the wrong value. This probability is written below,

$$P_{buf\_und\_err} = \sum_{i=\lceil R_{buf}/2 \rceil}^{R_{buf}} \begin{pmatrix} R_{buf} \\ i \end{pmatrix} {P_{buf\_elem\_err}}^i \left(1 - P_{buf\_elem\_err}\right)^{R_{buf}-i}. \tag{5.11}$$

5.3.4 Undetected Error Probability of the Complete System

The undetected error probability of the system will be computed similarly to the undetected error probability of an RB block. There is an undetected error in the system if there is an undetected error in any of the RB blocks of the system or any of the buffered connections of the system. An undetected error in an RB block results from an undetected error in its constituent detection blocks, and an undetected error in a buffered connection results from an undetected error in any of its constituent buffer elements. Therefore, we can conclude that any undetected error in the system results from either an undetected error in any of the detection blocks or the buffer elements of the system. In a system with the total of S_D detection blocks and S_B buffer elements, the probability that the system has at least one undetected error is

$$P_{sys\_und\_err} = \left(1 - (1 - P_{det\_block\_und\_err})^{S_D}\right) \bigcup \left(1 - (1 - P_{buf\_und\_err})^{S_B}\right). \tag{5.12}$$

equations (5.4) through (5.12) develop the undetected error probability in the whole system. Once we have the undetected error probability of the whole computation and having the system frequency, we can compute the FIT rate of the system, which is

the number of undetected errors in 10^9 hours of system operation,

$$FIT = P_{sys\_und\_err} \times 10^9 \times \text{System Frequency.} \tag{5.13}$$

Later in this section, using the above analysis, we show the required replication factor of R for a sample system specification. The complete area overhead including the buffered connections will come in the following section, at section 5.4.

5.3.5 Redundancy Analysis

Using the above analysis, we show the required replication to achieve the desired FIT rate for a sample system. In this section we focus on the logic replication factor R and compare this value with a feed-forward fault-tolerant approach. The detailed complete area overhead analysis including the buffered interconnect will be shown in the next section.

In order to use the equations (5.4) through (5.12), we have to specify the following system parameters:

- N_{logic}, logic block (logic cone) size: The logic block size depends on the design substrate. For the nanoPLA architecture model, we identify the efficient logic block sizes for permanent defect-tolerance in [65]. In [65] we bound the mapping redundancy for defects by limiting the fanin size of each OR-term. From the experiments in [64], we see that a logic block size of $N_{logic} = 16$ achieves compact systems close to the minimum size. Here we keep the same $N_{logic} = 16$ in our analysis since it is small enough to minimize the replication factor, R, as explained in section 5.1.4.

- S_D, the system size: The value of S_D, the number of detection blocks in the system, can be computed from the total number of devices in the system, N_t, divided by the size of a detection block. The size of a detection block is $R \times (N_{logic} + (R+2))$, consisting of R logic blocks and R checker blocks. Estimating the number of devices in the system built on the nanoPLA substrate, excluding

the buffered connections, around $N_t = 10^{12}$, the number of detection blocks in the system would be

$$\begin{aligned} S_D &= N_t/(R \times (N_{logic} + (R+2))) \\ &= 10^{12}/(R \times (16 + (R+2))). \end{aligned}$$

The size of S_B, the number of buffer elements in the system, is determined through the simulation described in section 5.4.

- The system frequency: The system runs at 10 GHz frequency, which is a reasonable expectation for future system design.

- Desired FIT rate: The desired FIT rate in this example is 360. With the above system frequency of 10 GHz the undetected error probability for the system will be $P_{sys\_und\_err} = 10^{-20}$.

- P_f, device failure rate: The device failure rate, ranges from 10^{-32} to 10^{-7} similar to previous studies [31][23].

We compare our rollback recovery results with feed-forward recovery results of [31] (reviewed briefly in section 2.1.3.3). In [31] the analysis was done for system reliability rate of 90%. Here we perform the calculation in [31] with the new $P_{sys\_und\_err} = 10^{-20}$ (for FIT=360, and system frequency of 10 GHz), which is much lower than the 10% target used in [31].

figure 5.14 plots the value of R for different values of P_f. These curves compare the replication factor of rollback recovery and feed-forward recovery. For fault rates smaller than 10^{-32} the system with no protection satisfies the system reliability goal of $(1 - 10^{-20})$. For higher fault rates just above 10^{-32} (left side of the graph) the rollback recovery has a replication factor of 2 (the minimum replication factor for error detection) and the feed-forward recovery has a replication factor of 3 (the minimum replication factor for Majority Multiplexing feed-forward recovery technique as described in section 2.1.3.3). As the fault rate increases the gap between the rollback and the feed-forward technique increases. The gap starts to grow dramatically for P_f

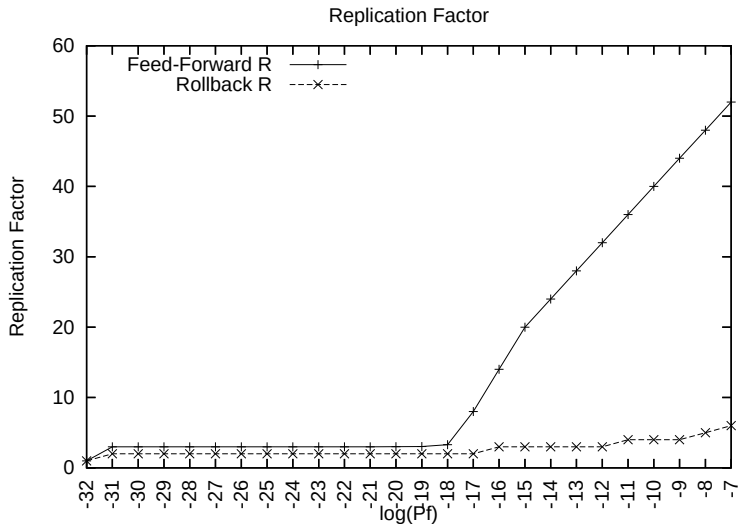


Figure 5.14. This graph compares the replication factor of rollback and feed-forward recovery. The feed-forward recovery data is from the majority-multiplexing shown in [31]. In their analysis the system reliability goal is 90%. We recomputed their results for a system reliability of $\left(P_{sys\_und\_err} = 1 - 10^{-20}\right)$, which is equivalent to a FIT of 360 used for our system specifications.

larger than 10^{-18}. The feed-forward replication factor grows to almost an order of magnitude greater than rollback recovery for $P_f \geq 10^{-9}$.

In this section we analyzed the replication factor of the rollback technique and demonstrated that the rollback recovery technique requires about one order of magnitude lower replication factor than feed-forward recovery technique. In the next section we see how the complete area including the checker and the buffered connections compare against feed-forward recovery technique. We also estimate the system throughput and see how rollback impacts the system performance.

5.4 Simulation and Comparison

In this section, we simulate our proposed reliable technique in the presence of random transient faults with various fault rates. We measure the system throughput and demonstrate the complete area overhead including the checker blocks and the buffered connections area. There are two variable parameters in our system specification that need to be specified to achieve the desired area-time tradeoff: the RB block size and

the streaming buffer depth. The RB block size, as explained in section 5.1.4, has two different effects on the system: First, larger RB block sizes, enclose more interconnects inside them, and therefore, reduce the total number of buffered connections in the system. As a result larger RB blocks allow compact system implementation. The second phenomenon has the the opposite effect; larger RB blocks tend to have more logic levels in the block, which increases the rollback latency, and a higher frequency of rollbacks. Therefore, using smaller RB blocks results in higher system performance. In our simulation we will find the best RB block size which balances these effects to minimize the area overhead and maximize the system throughput.

In order to meaningfully estimate the number of interconnects and the number of logic levels in an RB block, we tune our estimation with the *toronto20* benchmark set [68]. We map the designs in this benchmark set to nanoPLAs using a logic block size, N_{logic}, of 16. Figures 5.15 and 5.16 show the results of this mapping. Figure 5.15 shows the number of primary inputs and outputs of a design as the function of the design size in P-terms. In this figure, each data point represents a design from the benchmark, and the trend shown is a fitted Rent's Rule [72] curve (i.e., $IO = c \cdot (N_{blocks})^p$) to the data points. Similarly, figure 5.16 shows the logic depth of a design as the function of the design size. The data points represent the designs from the benchmark and are fitted to a logarithmic curve. In our simulation, we use the fitted curves from figure 5.15 and figure 5.16 to estimate the number of buffered connections at the boundary of an RB block or the number of logic levels in an RB block respectively.

We simulate the throughput of the system on the chain structure introduced in section 5.1.3. The building blocks of the chain are RB blocks and the length of the chain is 100 blocks. This is the same structure that was used in [31] to estimate redundancy factors required in the feed-forward approach.

The rest of the system parameters are the same as the previous section: $N_t = 10^{12}$, the system frequency is 10 GHz, and the FIT rate is 360.

During the simulation, random faults are injected into the system with probability of P_f. For each P_f we use the simulator to examine a range of RB block sizes and pick

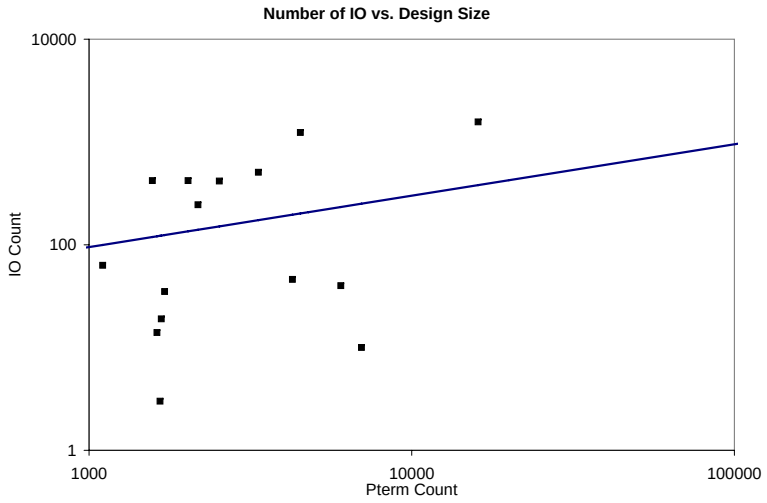


Figure 5.15. This graph shows the number of primary inputs and outputs (IO) versus the number of P-terms in a design. The data is from toronto20 benchmark set implemented on nanoPLA substrate with logic block size of 16. The curve shows the exponential function fitted to the data points, which is $IO = 3.2 \times (p - terms)^{(0.51)}$

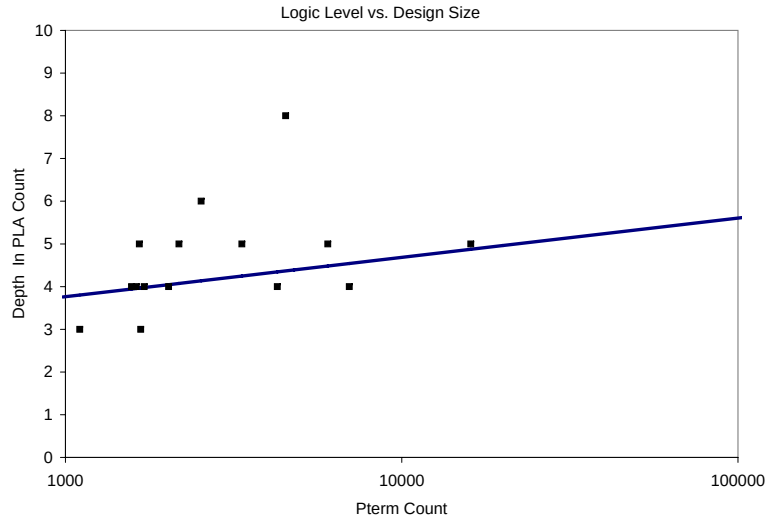


Figure 5.16. This graph shows the depth of the design in the number of nanoPLA planes, versus the number of P-terms in the design. The data is from toronto20 benchmark set implemented on nanoPLA substrate with logic block size of 16. The curve shows the logarithmic function fitted to the data points. This function is $Depth = 0.92 \log_{10}(p - terms)$

the best RB block size. For each RB block size we compute the area overhead and simulate the system throughput; this operation starts with the streaming buffer depth $D_s = 1$, and if the throughput is not high enough, increments D_s by one for each trial until the desired throughput is achieved. Here we set our throughput threshold at 98% for $\leq 10^{-9}$, and 90% for $> 10^{-9}$ (i.e., we add buffers until the throughput is at least 98% (or 90% for $> 10^{-9}$) of the throughput of the fault free case). Table 5.2 shows the RB block sizes which achieve the minimum area overhead while keeping the throughput above 98% (or 90% for $> 10^{-9}$). Table 5.1 shows the required streaming buffer depth to achieve the throughput target.

The RB block size and transient error rates determine the probability that each RB block detects an error and rolls back and, consequently, determines the throughput sustainable by the RB block. Table 5.2 shows the probability that an RB block detects an error (P_{detect}). For each P_f the RB block size is made small enough to keep P_{detect} low while not increasing the area overhead impractically large. We observed that for low P_{detect}, small streaming buffer depth is required (e.g., $D_s = 1$) while larger P_{detect} demands larger streaming buffer depth. The system needs the minimum of $D_s = 1$ to achieve high system throughput even for smaller fault rates. With no buffering, a single rollback stalls all the logic on the chip; however, the elasticity provided by even the minimum size D_s limits the impacted number of RB blocks. For example, let D_s be 1, and the rollback latency (D_R) be 4 (which is the minimum rollback latency). Then if the ith RB block detects an error and stops to rollback at time t, the rollback wave expands to the $i-4$th RB block over the period of 4 cycles, such that, the $i-1$st block run for one more cycle after cycle t, filling up the single streaming buffer following that block and stopping at cycle $t+1$. The $i-2$nd block runs for another cycle, filling up the single streaming buffer following this block and stopping at cycle $t-2$. This continues until the $i-4$th block stops at $t+4$, after filling up its following streaming buffer. The ith block had zero throughput from cycle t to $t+4$, however, 4 data elements are stored in the 4 streaming buffers distributing over 4 stages. So if any block preceding the $i-4$th stage detects an error and stops to rollback in the future, the 4 data element will be consumed by the following blocks, preventing these

downstream blocks from sitting idle for another D_R cycles, the same effect that was shown earlier in section 5.1.3. Consequently, this minimum buffering guarantees that RB blocks further away in the chain are not impacted by this failure; if we see only one rollback occurring at a time, only the few RB blocks immediately adjacent to the affected RB block stall, while the majority of RB blocks continue their operation.

We observed the following interesting effect of the streaming buffer depth and the rollback block size on the system throughput: The simulation shows that the impact of RB block size on the throughput is stronger than the depth of the streaming buffers. This means that in a nominal design, reducing RB block size yields a larger throughput improvement than increasing the depth of the streaming buffers between the RB blocks. Therefore, to achieve high throughput and keep area overhead low, it is more beneficial to minimize the RB block size and use the minimum required streaming buffer depth. Note that the RB block size reduces to 300 detection blocks, or 188000 P-terms, by $P_f = 10^{-7}$; these results show how the strong dependence of RB block size on device fault rate drives us to fine-grained rollback blocks for designs at these fault rates. We also note that, even at this high transient fault rate and relatively high rollback overhead, the RB block size does not reduce to a single detection block, underscoring the value of keeping the detection block size separate from the RB block size (section 5.1.4).

5.4.1 Area and Throughput Simulation Results

The areas determined from the simulation are plotted in figure 5.18. Figure 5.18 shows the replication factor, R, and the total area overhead of the rollback recovery technique. The figure also plots the replication factor of the feed-forward technique for comparison. The replication factors are computed as explained in section 5.3.5. The total rollback area overhead curve includes the complete area of the RB blocks and the buffered connections.

figure 5.17 plots the throughput of the system. As you can see for $P_f \leq 10^{-9}$ the impact on the throughput is almost negligible and for higher fault rate the drop in throughput is less than 10%. This minimal impact on the throughput is achieved

$\log(P_f)$	$\leq$-16	-15	-14	-13	-12	-11	-10	-9	-8	-7
D_s	1	1	1	1	1	1	2	2	3	3
R_{buf}	3	5	5	5	5	5	7	7	9	9

Table 5.1. This table shows the depth, D_s, and the replication factor of buffered connections, R_{buf}.

while reducing the area required by a factor of 6 compared to the feed-forward recovery technique.

In order to understand the area curve in figure 5.18, it is helpful to understand how the system area is distributed over different part of the system. Table 5.3 summarize the equations used to compute the area of each component in an RB block; area is calculated in terms of nanowire pairs. Table 5.4 shows how the area of the system is distributed over different parts of the system for different fault rates, P_f. As you can see the logic and checker area is the dominant portion of the total system area for moderate fault rates ($P_f < 10^{-9}$). The buffered connection area (A_{buffer}) plus the voter area (A_{voter}) increase as the fault rate P_f increases. Achieving high throughput with high fault rate, demands smaller RB block size, and smaller RB block size results in more buffered connection in the system, which also increases the overall system area. This effect can also be seen in the area curve in figure 5.18. This figure shows that for $P_f < 10^{-9}$ the total area is dominated by the logic replication factor which is the minimum possible area overhead. The area curve follows the replication curve closely. For these fault rate, we also see a very small drop in the system throughput (figure 5.17). For higher fault rates the RB block size is reduced to prevent throughput loss. Reducing the rollback block size, however, results in more streaming interconnects in the system. Therefore, the buffered connections start to consume a larger fraction of the total area. This fact causes the divergence of the total area overhead curve from the replication factor curve around $P_f = 10^{-9}$.

figure 5.19 plots the area/throughput ratio for rollback recovery and feed-forward recovery techniques. As you can see our rollback technique, not only reduces the area overhead by up to a factor of 6, but from an area-time product point of view it is also a more efficient design.

$\log(P_f)$	RB Block Size	P_{detect}
≤ -11	10000	$\leq 9.3 \times 10^{-6}$
-10	3500	3.2×10^{-5}
-9	3000	2.7×10^{-4}
-8	2500	3.1×10^{-3}
-7	300	4.9×10^{-3}

Table 5.2. This table shows the number of detection blocks in an RB block.

One RB block area in the number of nanowire pairs	
A_{logic}	$R \times B \times N_{logic}$
$A_{checker}$	$(R-1) \times (R+2) \times B$
A_{buffer}	$IO \times R_{buf} \times Size_{buf}$ $\times(D_s + Depth_{RB} \times D_L + D_C + 1)$
A_{voter}	$IO \times \begin{pmatrix} R_{buf} \\ R_{buf}/2+1 \end{pmatrix} \times R_{buf}$

Table 5.3. In A_{logic} the value of B is the number of logic blocks in RB blocks. The value of N_{logic} is 16 nanowires. In A_{buffer}, IO is the number of buffered connections of an RB block, which is estimated by the curve in figure 5.15. For nanoPLA detection block $D_L = 1$ and $D_C = 2$. The streaming buffer depth, D_s, is defined by the throughput simulation and Table 5.1 shows the selected values of D_s for different fault rate values, generated by our simulation.

P_f Range	A_{logic}	A_{check}	A_{buffer}	A_{voter}
$10^{(-29)} - 10^{(-17)}$	66.56	10.40	22.51	0.52
$10^{(-16)} - 10^{(-11)}$	60.85	15.21	22.87	1.06
$10^{(-10)}$	49.75	16.32	30.07	3.86
$10^{(-9)}$	48.40	15.88	31.66	4.06
$10^{(-8)}$	27.88	11.15	53.61	7.36
$10^{(-7)}$	15.28	7.16	67.86	9.69

Table 5.4. This table shows the distribution of the area over different parts of an RB block.

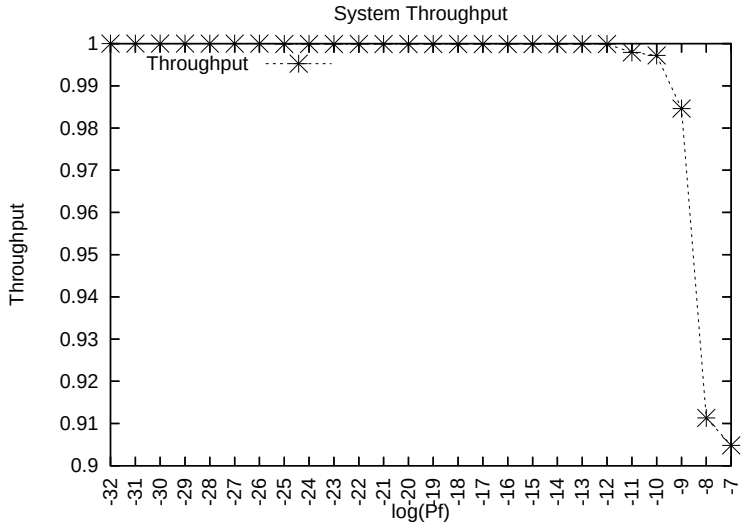


Figure 5.17. This graph shows the system throughput as the function of failure rate, P_f.

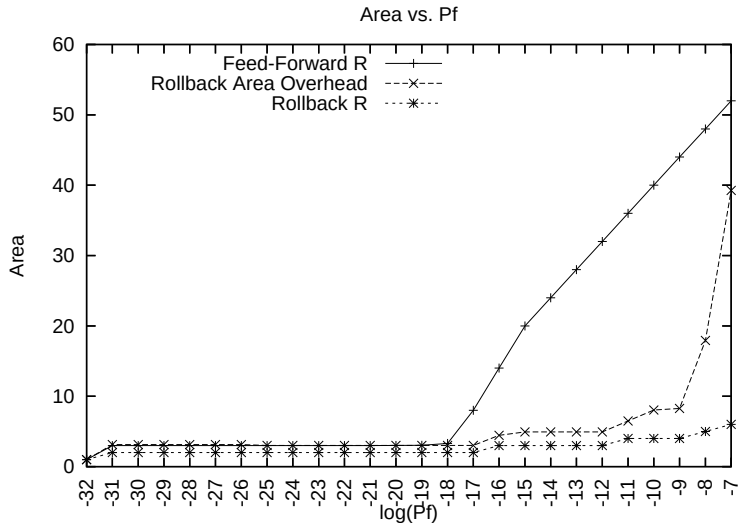


Figure 5.18. The solid curve with "+" markers show the replication factor of the feed-forward technique from [31] with higher reliability goal of FIT=360. The curve with "$\ast$" markers is the replication factor of the rollback recovery. The third curve with "$\times$" markers show the total area of the rollback recovery technique.

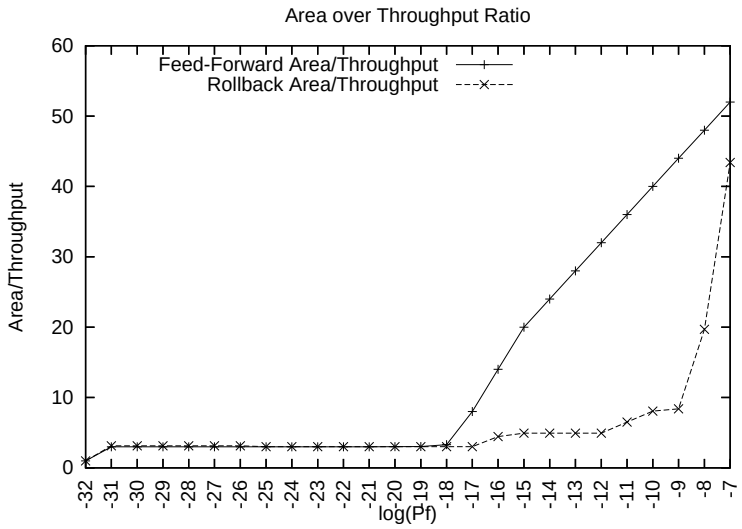


Figure 5.19. This graph plots the area/throughput for the rollback and feed-forward recovery techniques.

5.5 Summary

Reliability techniques, such as *Feed-Forward Recovery*, rely only on spatial redundancy. These techniques require large area overhead as the device failure rate increases. Here we developed and analyzed a recovery technique, *Fine-Grained Rollback Recovery*, that exploits redundancy in time as well as space. This technique has lower area overhead with negligible impact on performance for fault rates as high as $P_f = 10^{-10}$. At $P_f = 10^{-9}$ the replication factor is almost an order of magnitude smaller in rollback recovery than feed-forward recovery. For $P_f \leq 10^{-9}$, even the total area overhead of rollback can be about 6 times smaller than feed-forward replication factor—and consequently much smaller than the complete area overhead required for a feed-forward implementation. At these fault rates, we show that detection is best performed using fine-grained detection blocks using 88 P-terms to protect 16 logical P-terms and rollback is best performed on larger blocks containing 450K P-terms to protect 56K logical P-terms.

Although the replication factor of rollback recovery remains relatively low for high fault rates, the total area overhead becomes large due to the streaming buffers. At high fault rates, buffer area is the dominant area in streaming design. E.g., for

$P_f = 10^{-7}$, the buffered connection takes almost 2/3 of the total area. Therefore, techniques which reduce this buffer overhead could offer even greater area benefit.

We used replication with comparison as the error detection technique because it has compact encoder and decoder circuits and allows general-purpose analysis. The total area overhead may be further reduced by using smarter technique, as long as the encoder and decoder circuits remain small.

Chapter 6

Defect and Fault-Tolerant Nanomemory Design

6.1 Introduction and Motivation

Chapter 5 introduced fine-grained rollback technique to protect any arbitrary logic circuit against transient faults. The fine-grained rollback technique use general-purpose replication-and-compare to detect errors. The area overhead can be further reduced by using a more complex error-detection techniques. One potential approach is using error-correcting codes. Error-correcting codes are shown to be more efficient than replication in data storage and communication applications. However, for combinational logic it is not yet shown if error-correcting/detecting codes can outperform replication. A special case of protecting logic with error-detecting codes is solved for single error detection in arbitrary logic by using parity prediction [73][70]. In this technique the parity function of the output signals is predicted concurrently using the input signals. Checking the output signals against the parity signals reveals a potential error in the outputs or parity signals. However, generalization of this scheme to tolerate multiple errors in the output set is not straight forward. Despite the active research in this field there has not been any work to report an error-detecting/correcting technique for arbitrary logic that outperforms replication in the area overhead. In this chapter we solve this problem for one subset of combination logic: encoder, corrector and detector circuits. We show how to protect these units exploiting ECC to outperform the area overhead compared to replication scheme [12][74].

Fault-tolerant encoder, corrector, and detector can be used to design a completely fault-tolerant memory system. Conventionally, only memory bits were protected against transient faults, however, since combinational logic is becoming more susceptible to faults as the feature size scales [6], the supporting logic of the memory system must also be protected against transient faults as well. Therefore, there have been an increasing amount of research on designing fault-tolerant encoders and decoders [75][76]. Most of these techniques consider only single event upset, and using conventional fault-tolerant techniques which are based on adding extra circuitry to the encoder, corrector, and detector.

We suggest a technique that can tolerate multiple errors in these units. The unique characteristic of this technique is that it does not add any extra protection circuitry to the encoder, corrector and detector units. There is a class of ECCs that guarantees fault-tolerant encoder, corrector, and detector design. This class is defined with the new restricted definition for ECC. The redundancy accumulated in these units to perform their ECC-related operation is enough to detect multiple errors in these units. This subclass of ECCs has the property that their detector circuitry is *Fault-Secure*, i.e., It can detect any error in the input code vector successfully, despite experiencing errors in its own circuitry. We call this type of error-correcting codes, *Fault-Secure Detector capable ECCs*, *FSD-ECC*. The FSD-ECC, allows detection of as many errors in the received code-word and detector circuitry as the ECC minimum distance allows, without compromising the detection capability of the system when adding the protection for the supporting circuitry. The other important advantage of this technique is its flexibility. Once an FSD-ECC code is found, the usual detector circuit, which is based on the parity-check matrix of the code, has the fault-secure property.

The fault-secure detection unit is used to design a fault-tolerant encoder and corrector by monitoring their outputs. One fault-secure detector monitors the outputs of each encoder and corrector. If a detector detects an error in either of these units, that unit has to repeat the operation to generate the correct output vector. Using this rollback technique, we can correct potential transient errors in the encoder

and corrector outputs and provide fault-tolerant memory system with fault-tolerant supporting circuitry.

The ECC mentioned above can also be used to correct erroneous bits in the memory words due to the permanent defects. We present a unified technique for tolerating transient faults and permanent defects. When using a single ECC to correct both transient and permanent errors, the first advantage is that only one encoder and corrector unit is required. If using separate ECC for defect- and fault-tolerant technique, then each memory word has to pass through two encoders, one for the defect-tolerant ECC and one for the fault-tolerant ECC. Similarly each memory word has to pass through two corrector, one for defect-tolerant ECC and one for fault-tolerant ECC. Therefore, the unified technique, which requires only one encoder and corrector unit saves area, time, and power. The analysis and implementation detail of this approach will be presented later in this chapter.

In the rest of this chapter we first review the related works and compare our approach with other fault-tolerant encoder and corrector techniques. We then show the restricted ECC definition (FSD-ECC) which guarantees having a fault-secure detector. Then the encoder, corrector, and detector designs and implementation is presented. Later in this chapter we show a unified technique based on FSD-ECC to tolerate permanent defects and transient faults together. Finally we present the reliability analysis and the area and performance of the design.

6.2 Related Works

Traditionally, memory cells were the only susceptible part to transient faults. However, the supporting logic of memory system is also expected to be affected by transient faults as well [6]. Consequently, developing fault-tolerant encoders, correctors, and detectors for memory system attracted considerable attentions recently [75][76]. Almost all of the techniques use the conventional fault-tolerant schemes(e.g., parity-prediction) to protect the encoder and corrector circuitry similar to other general purpose fault-tolerant schemes. In contrast the technique introduced in this work

does not use the conventional fault-tolerant schemes. It exploit the structure of the Error-Correcting Code and based on the specific structure of the code guarantees the fault-secure detector unit.

The work presented in [75] is based on parity-prediction. It predicts a set of parity bits for the generating code-word from the information bits, which are the inputs of the encoder unit. Then the predicted parity bits will be compared against the similar parity bits of the encoder outputs, which are the parity bits of the code-word. The general structure of this technique is illustrated in figure 6.1. For cyclic codes, the parity predictor essentially performs a vector division on the k-bit information vector or $(n-k)$-bit ECC parity bits, and the parity function is an $(n-k)$-bit remainder. To implement a compact parity generator (divider) unit, the right denominator vector must be found, which depends on the specific ECC. For this reason, the related works are usually designed for one specific code.

In our suggested technique, however, any codes that satisfies the FSD-ECC condition, guarantees the fault-secure capability of the detector without demanding any other fault-tolerant technique. This characteristic make this technique very compact and more important flexible; once a code is proved to be FSD-ECC, it has the fault-secure detection capability without requiring to go through any design detail to make the detector fault-secure.

6.3 System Overview

In this section we outline our memory system which can tolerate errors in any part of the system, including the storage unit, encoder and corrector circuit, using the fault-secure detector. Let E be the maximum number of error bits that the code can correct and D be the maximum number of error bits that it can detect, and in one error combination that strikes the system let e_e, e_m, and e_c be the number of errors in encoder, memory word, and corrector. In conventional designs, the system would guarantee error correction as long as $e_m \leq E$ and $e_e = e_c = 0$. In contrast, here we guarantee that the system can correct any error combination as long as $e_m \leq E$,

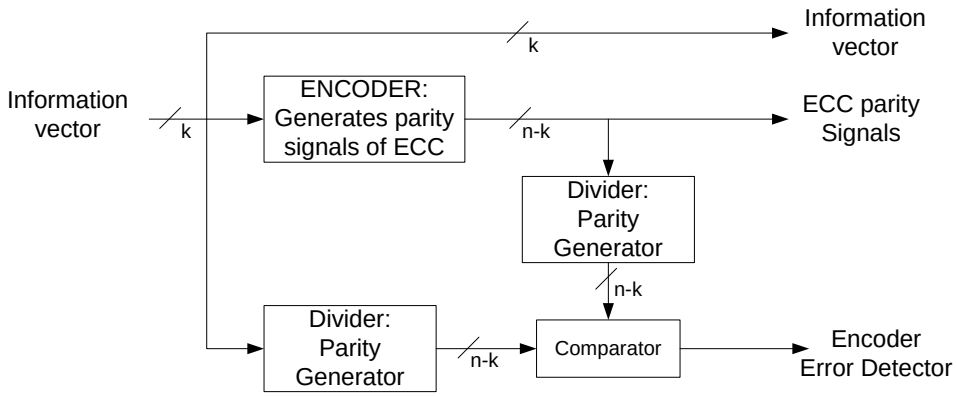


Figure 6.1. The fault-tolerant parity-prediction based encoder

$e_e + e_{de} \leq D$, and $e_m + e_c + e_{dc} \leq D$, where e_{de} and e_{dc} are the number of errors in the two separate detectors, monitoring the encoder and corrector units. This design is feasible when the following two fundamental properties are satisfied:

1. Any single error in the encoder or corrector circuitry can at most corrupt a single code-word digit (i.e., cannot propagate to multiple code-word digits).

2. There is a fault secure detector that can detect any combination of errors in the received code-word along with errors in the detector circuit. This fault-secure detector can verify the correctness of the encoder and corrector operation.

An overview of our proposed reliable memory system is shown in figure 6.2, and is as described below: The information bits are fed into the encoder to encode the information vector, and the fault secure detector of the encoder verifies the validity of the encoded vector. If the detector detects any error, the encoding operation must be redone to generate the correct code-word. The code-word is then stored in the memory. Later during operation, the stored code-word will be retrieved from the memory unit. Since the code-word is susceptible to transient faults while it is stored in the memory, the retrieved code-word must be fed into the detector to detect any potential error and possibly to the corrector to recover any erroneous bits. In the

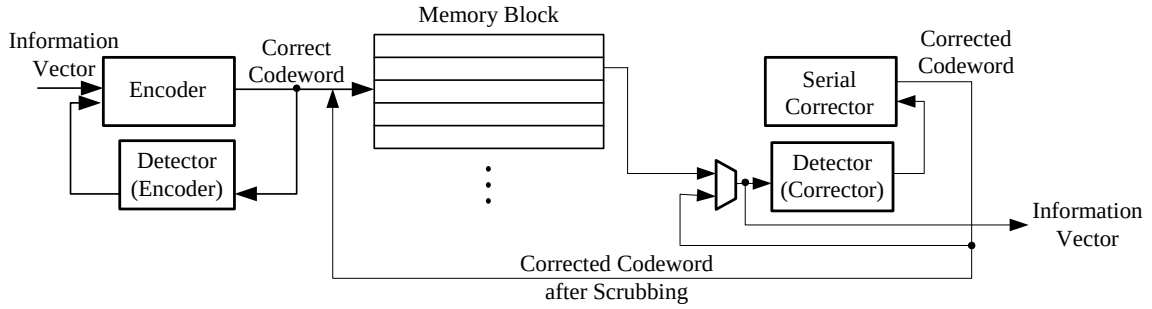


Figure 6.2. The overview of our proposed fault-tolerant memory architecture.

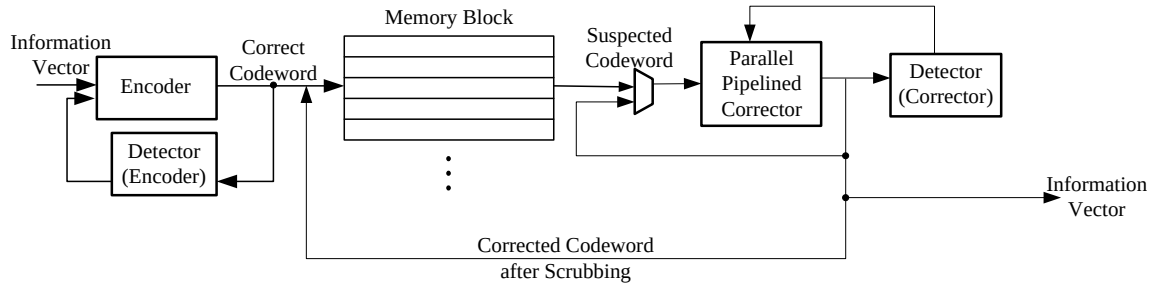


Figure 6.3. The overview of our proposed fault-tolerant memory architecture, with pipelined corrector.

design flow shown in figure 6.2, all the memory words pass through the detector. If an error is detected in a memory word, that word is sent to the serial corrector to be corrected. The detector then checks the output of the corrector and if memory word is correct it sends the information bits out. The detector circuit is implemented in parallel for each code-bit and fully pipelined, and therefore memory words are pipelined through the detector circuit one codeword every cycle. However, when an error is detected in a codeword, the normal pipeline flow of the detector is stopped and waits for the serial corrector to generate the corrected memory memory word. This serial corrector can take n (code length) cycles to correct all the n bits of the memory word. Therefore, this technique is useful when errors happen with low frequency. If memory words are error free most of the times and only rarely are detected with error, then spending n cycles plus the latency of the detector will not impact the access throughput. However, when errors happen in higher frequency, e.g., the system also correct errors due to permanent defects, a more efficient design, which has shorter corrector latency, must be used.

Figure 6.3 shows the structure of a system with faster corrector. In this design the corrector circuit has parallel structure and is implemented fully pipelined similar to the detector. All the memory words are pipelined through the corrector and then detector, therefore, one corrected memory word is generated every cycle. The detector following the corrector, would raise an error-detection flag only if a transient fault occurs in the corrector or detector circuitry. Due to the relative lower transient fault rate compared to the permanent defects and the relative small corrector and detector circuitry, this happens with low frequency. Therefore, the potential throughput loss of this system is low.

Data bits stay in memory for number of cycles and during this period each memory bit can be hit by transient fault with certain probability. Therefore, transient errors accumulate in the memory words over time. In order to avoid accumulation of too many errors in the memory words that surpasses the code correction capability, the system has to perform memory *scrubbing*. Memory scrubbing is periodically reading memory words from the memory, correcting any potential errors and writing them

back into the memory [77] (figures 6.2 and 6.3). To perform the periodic scrubbing operation, the normal memory access operation is stopped and the memory goes under the scrubbing operation, where each memory word is read, checked and corrected if necessary and then written back into the memory. The number of faults that accumulate in the memory is directly related to the scrubbing period. The longer the scrubbing period is, the larger number of errors can accumulate in the system. Therefore, to control the number of errors in the memory word, the right scrubbing period must be selected. If the fault rate is high the scrubbing interval must be short for reliability purpose, furthermore short scrubbing interval can decrease the system performance; since the memory system has to be idle frequently to go through the scrubbing operation. One approach to reduce the impact on the system performance for the case where the fault rate is high and consequently the scrubbing is frequent, is to make the scrubbing operation fast. This can be done by providing multiple detectors and correctors that correct the memory words in parallel (similar to the corrector structure of figure 6.3). Later in this chapter we will explain more details of scrubbing operation and potential optimization to achieve high performance and high reliability. We also explain each of the above units and memory operations in more details in the following sections.

6.4 ECCs with Fault Secure Detector

In this section we present our novel, restricted Error-Correcting Code definition for our fault-secure detector capable codes. Before starting the details of our new definition we briefly review the basic linear Error-Correcting Codes.

6.4.1 Error-Correcting Code Reviews

This section provides a brief introduction to linear block ECCs. Let $i = (i_0, i_1, ..., i_{k-1})$ be the k-bit information vector that will be encoded into a n-bit code-word, $c = (c_0, c_1, ..., c_{n-1})$. For linear codes the encoding operation essentially performs the

following vector-matrix multiplication,

$$c = i \times G, \tag{6.1}$$

where G is a $k \times n$ generator matrix. Checking the validity of a received encoded vector is done by employing the *Parity-Check* matrix, which is an $(n-k) \times n$ binary matrix named H. The checking or detecting operation is basically summarized as the following vector-matrix multiplication,

$$s = c \times H^T. \tag{6.2}$$

The $(n-k)$-bit vector s is called *syndrome* vector. A syndrome vector is zero if c is a valid code-word, and non-zero if c is an erroneous code-word. Each code is uniquely specified by its generator matrix or parity-check matrix.

A code is a *systematic code* if any code-word consists of the original k-bit information vector followed by $n-k$ parity-bits [78]. With this definition, the generator matrix of a systematic code must have the following structure,

$$G = [I : X], \tag{6.3}$$

where I is a $k \times k$ identity matrix and X is a $k \times (n-k)$ matrix that generates the parity-bits. Figure 6.4 shows a systematic generator matrix. This generator matrix generates a Hamming code of (7,4,3). The advantage of using systematic codes is that there is no need for a decoder circuitry to extract the information bits. The information bits are simply available in the first k bits of any encoded vector.

A code is said to be a *cyclic code* if for any code-word c, all the cyclic shifts of the code-word are still valid code-words. A code is cyclic *iff* the rows of its parity-check matrix and generator matrix are the cyclic shifts of their first rows. It is shown in [79] that any generator matrix of a cyclic code can be transformed into systematic generator matrix.

The *Minimum Distance* of an ECC, is the minimum number of code-bits that are

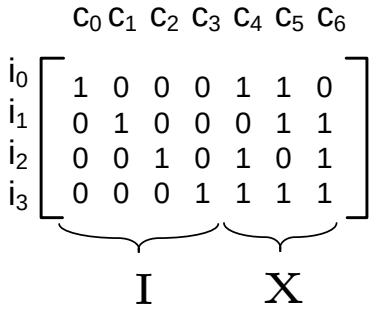


Figure 6.4. A systematic generator matrix: The generator matrix of (7,4,3) Hamming code.

different between any two code-words, and it is shown by d. The maximum number of errors that an ECC can detect is $d-1$, and the maximum number that it corrects is $\lfloor d/2 \rfloor$.

6.4.2 FSD-ECC Definition

The restricted ECC definition which guarantees a *Fault-Secure-Detector* capable ECC (FSD-ECC) is as follows:

Definition I: Let C be an ECC with minimum distance d. C is FSD-ECC if it can detect any combination of overall $d-1$ or fewer errors in the received code-word and in the detector circuitry.

Theorem I: *Let C be an ECC, with minimum distance d. C is FSD-ECC iff any error vector of weight $e \leq d-1$, has syndrome vector of weight at least $d-e$.*

Note: The following proof depends on the fact that any single error in the detector circuitry can corrupt at most one output (one syndrome bit). This can be easily satisfied for any type of circuitry by implementing the circuit in such a way that no logic element is shared among multiple output bits, therefore, any single error in the circuit corrupt at most one output (one syndrome bit).

Proof: The core of a detector circuitry is a multiplier that implements the vector-matrix multiply of the received vector and the parity-check matrix to generate the

syndrome vector. Now if e errors strike the received code-vector the syndrome weight of the error pattern is at least $d-e$ from the assumption. Furthermore, the maximum number of tolerable errors in the whole system is $d-1$ and e errors already exist in the encoded vector, therefore, the maximum number of errors that can strike in the detector circuitry is $d-1-e$. From the above note, this many errors can corrupt at most $d-1-e$ syndrome bit, which in worst-case leaves at least one non-zero syndrome bit, and therefore, detects the errors. **Q.E.D**

The difference between FSD-ECC and normal ECC is simply the demand on syndrome weight. That is, for error vector of weight e, a normal ECC demands non-zero syndrome weight while FSD-ECC demands syndrome weight of $\geq d-e$.

6.5 FSD-ECC Example: Euclidean Geometry and Projective Geometry Codes

In this section we prove that two classes of known error-correcting codes, are FSD-ECC. These two classes are *Euclidean Geometry* and *Projective Geometry* codes. We first review these two codes and then prove they have the FSD-ECC property.

6.5.1 Euclidean Geometry Code Review

This section reviews a simple construction of Euclidean Geometry and Projective Geometry codes based on the lines and points of the corresponding two finite geometries [80]. Since these two codes generate Low-Density Parity-Check (LDPC) matrices [81], they are called EG-LDPC and PG-LDPC codes. We follow the notation of [79] here. We start by showing the construction of EG-LDPC. The construction of PG-LDPC which is very similar to EG-LDPC will come afterward.

Let **EG** be a Euclidean Geometry with n points and J lines. **EG** is a finite geometry that is shown to have the following fundamental structural properties:

1. Every line consists of ρ points
2. Any two points are connected by exactly one line

3. Every point is intersected by γ lines

4. Two lines intersect in exactly one point or they are parallel, i.e., do not intersect.

Let H be a $J \times n$ binary matrix, whose rows and columns corresponds to lines and points in **EG** Euclidean geometry, respectively, where $h_{i,j} = 1$ if and only if the ith line of **EG** contains the jth point of **EG**, and $h_{i,j} = 0$ otherwise. A row in H displays the points on a specific line of **EG** and has weight ρ. A column in H displays the lines that intersect at a specific point in **EG** and has weight γ. The rows of H are called the incidence vectors of the lines in **EG**, and the columns of H are called the intersecting vectors of the points in **EG**. Therefore, H is the incidence matrix of the lines in **EG** over the points in **EG**. It is shown in [79] that H is a Low-Density Parity-Check matrix, and therefore, the code is an LDPC code.

A special subclass of EG-LDPC code, type-I two-dimensional EG-LDPC, is considered here. It is shown in [79] that type-I two-dimensional EG-LDPC has the following parameters for any positive integer $t \geq 2$:

- Information bits, $k = 2^{2t} - 3^t$
- Length, $n = 2^{2t} - 1$
- Minimum distance, $d_{min} = 2^t + 1$
- Dimensions of the Parity-Check matrix, $(2^{2t} - 1) \times (2^{2t} - 1)$
- Row weight of the Parity-Check matrix, $\rho = 2^t$
- Column weight of the Parity-Check matrix, $\gamma = 2^t$

It is important to note that the rows of H are not necessarily linearly independent. The rank of H is $n - k$ which makes the code of this matrix (n, k) linear code. The $(2^{2t} - 1) \times (2^{2t} - 1)$, parity-check matrix H of an **EG** euclidean geometry, can be formed by taking the incidence vector of a line in **EG** and its $2^{2t} - 2$ cyclic shifts as rows, therefore, this code is a *cyclic code*. A cyclic code's parity-check matrix is the cyclic shifts of the first row [79]. Figure 6.12 shows a parity-check matrix for $(15, 7)$

EG-LDPC code. You can see that all the rows of this parity-check matrix are the cyclic shift of the first row.

6.5.2 Projective Geometry Code Review

Construction of Projective Geometry codes are very similar to the Euclidean Geometry codes. The main difference is that the Euclidean Geometry codes are based on the Euclidean geometry, and the Projective Geometry codes are based on Projective geometry. Let **PG** be a projective geometry. The parity-check matrix of Projective Geometry code is the incidence matrix of the lines in **PG** over the points in **PG**. The projective geometry **PG** has the same fundamental structural properties as Euclidean geometry, and it is shown that the code constructed based on **PG** is a Low-Density Parity-Check code. The special subclass of these codes that are considered here is type-I two-dimensional PG-LDPC codes. It is shown that this subclass of codes has the following parameters, for any positive integer $t \geq 2$:

- Information bits, $k = 2^{2t} + 2^t - 3^t$
- Length, $n = 2^{2t} + 2^t + 1$
- Minimum distance, $d_{min} = 2^t + 2$
- Dimensions of the Parity-Check matrix, $(2^{2t} + 2^t + 1) \times (2^{2t} + 2^t + 1)$
- Row weight of the Parity-Check matrix, $\rho = 2^t + 1$
- Column weight of the Parity-Check matrix, $\gamma = 2^t + 1$

It is shown in [79] that PG-LDPC codes are cyclic codes. Therefore, the $(2^{2t} + 2^t + 1) \times (2^{2t} + 2^t + 1)$, parity-check matrix H of a **PG** projective geometry, can be formed by taking the incidence vector of a line in **PG** and its $2^{2t} + 2^t$ cyclic shifts as rows.

6.5.3 FSD-ECC Proof for EG-LDPC and PG-LDPC

Theorem II: *Type-I two-dimensional EG-LDPC and PG-LDPC codes are FSD-ECC.*

Proof: Let C be an EG-LDPC or PG-LDPC code with column weight γ and minimum distance d. We have to show that any error vector of weight $e \leq d-1$, corrupting the received encoded vector, has syndrome vector of weight at least $d-e$.

Now a specific bit in the syndrome vector will be one if and only if the parity-check sum corresponding to this syndrome vector has an odd number of error bits present in it. Looking from the Euclidean geometry perspective, each error bit corresponds to a point in the geometry and each bit in the syndrome vector corresponds to a line. Now we are interested in obtaining a lower bound on the number of lines that pass through an odd number of error points. We further lower bound this quantity by the number of lines that pass through exactly one of the error points. Based on the definition of the Euclidean geometry, γ lines pass through each point; so e error points potentially impact γe lines. Also at most one line connects two points. Therefore, looking at the e error points, there are at most $\begin{pmatrix} e \\ 2 \end{pmatrix}$ lines between pairs of error points. Hence the number of lines passing through a collection of these e points is lower bounded by $\gamma e - \begin{pmatrix} e \\ 2 \end{pmatrix}$. Out of this number, at most $\begin{pmatrix} e \\ 2 \end{pmatrix}$ lines connect two or more points of the error points. Summarizing all this, the number of lines passing through exactly one error point is at least $\gamma e - 2\begin{pmatrix} e \\ 2 \end{pmatrix}$.

From the code properties in the previous two sections that $d = \gamma + 1$, we can derive the following inequality

$$\gamma e - 2\begin{pmatrix} e \\ 2 \end{pmatrix} = e(\gamma + 1 - e) = e(d-e) \geq d - e.$$

The above inequality says that the weight of the syndrome vector is at least $d-e$ which is the required condition of Theorem (I). Therefore, EG-LDPC and PG-LDPC are FSD-ECC. **Q.E.D.**

Table 6.1. The detector circuit area in the number of 2-input gates

Type	EG	PG	EG	PG	EG	PG	EG	PG
Code	(15,7)	(21,11)	(63,37)	(73,45)	(255,175)	(273,191)	(1023,781)	(1057,813)
Det. Area	45	48	501	584	3825	4368	31713	33824

6.6 Design Structure

In this section we provide the design structure of the encoder, corrector, and detector units of our proposed fault-tolerant memory system. We also present the implementation of these units on a sublithographic nanowire-based substrate. Later in this section we presents a demultiplexer that integrates the nanoMemory core with the supporting logic in nanoscale devices.

6.6.1 Fault Secure Detector

The core of the detector operation is to generate the syndrome vector, which is basically implementing the following vector-matrix multiplication on the received encoded vector c and parity-check matrix H,

$$s = c \cdot H^T. \tag{6.4}$$

Therefore, each bit of the syndrome vector is the product of c with one row of the parity-check matrix. This product is a linear binary sum over digits of c where the corresponding digit in the matrix row is 1. This binary sum is implemented with an XOR gate. Figure 6.5 shows the detector circuit of $(15, 7)$ EG-LDPC code, the parity-check matrix of this code is shown in figure 6.12. Since the row weight of the parity-check matrix is ρ, to generate one digit of the syndrome vector we need a ρ-input XOR gate, or $(\rho - 1)$ 2-input XOR gates. For the whole detector, it takes $\gamma(\rho - 1)$ 2-input XOR gates. Table 6.1 illustrates this quantity for some of EG-LDPC and PG-LDPC codes. Note that implementing each syndrome bit with a separate XOR gate satisfies the assumption of Theorem I on no logic sharing in detector circuit implementation.

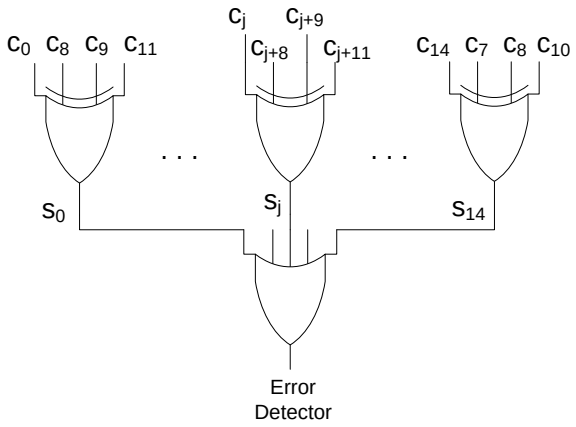


Figure 6.5. The fault-secure detector for $(15, 7)$ EG-LDPC code

An error is detected if any of the syndrome bits has a non-zero value. The final error detection signal is implemented by an OR function of all the syndrome bits. The output of this n-input OR gate is the error detector signal. In order to avoid a single point of failure, we have to implement the OR gate with a reliable substrate, e.g., in a system with sublithographic *nanowire* substrate, the OR gate is implemented with reliable lithographic technology– I.e., lithographic-scaled wire-OR. This n-input wired-OR is much smaller than implementing the entire $n \times (\rho - 1)$ 2-input XORs at the lithographic scale. The area of each detector is computed using the model of nanoPLA and nanoMemory form [33] and [50] accounting for all the supporting lithographic wires.

figure 6.7 shows the implementation of the detector on a nanoPLA substrate. The framed block shows a γ-input XOR gate, implementing a $\log_2(\rho)$-level XOR tree in spiral form (figure 6.6). The solid boxes display the restoration planes and the white boxes display the wired-OR planes of nanoPLA architecture model [82][33]. The signals rotate counter clock-wise, and each round of signal generates the XOR functions of one level of the XOR-tree. The final output then gates a robust lithographic-scale wire. This lithographic-scale wire generates a wired-OR function of all the n ρ-input XORs and is the final output of the detector circuit. The XOR gate is the main building

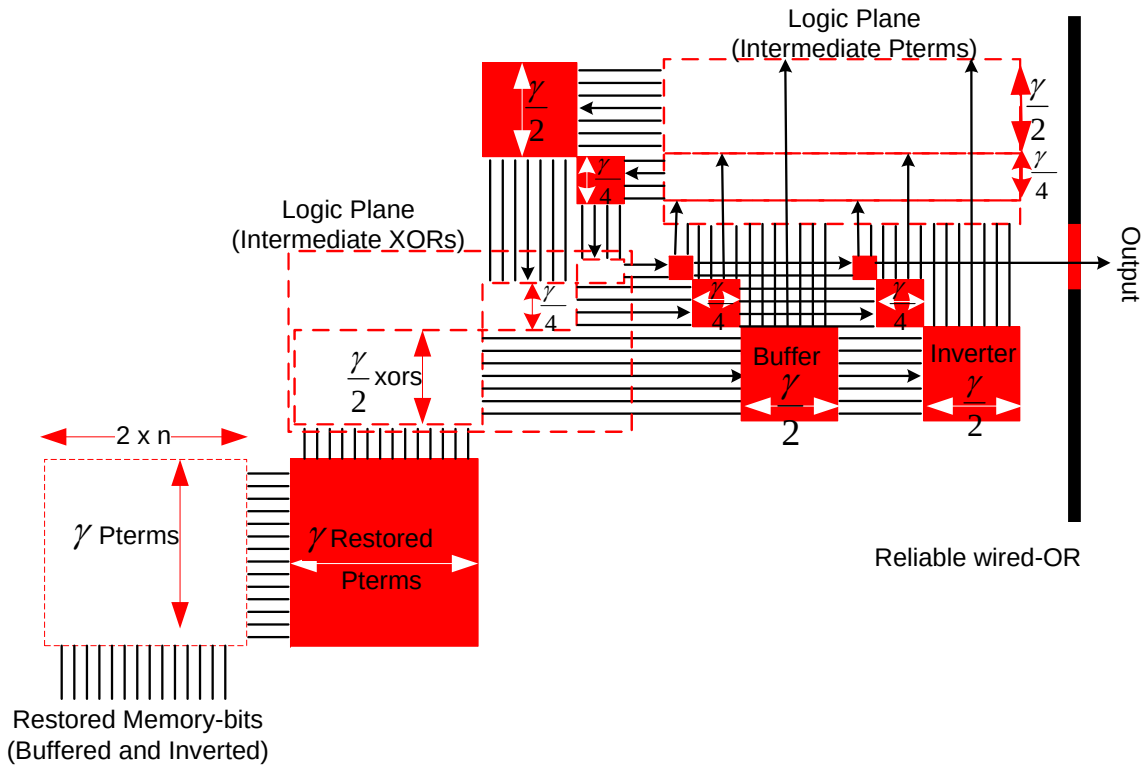


Figure 6.6. A γ-input XOR tree implemented on nanoPLA structure.

block of the encoder and corrector as well.

6.6.2 Encoder

An n-bit code-word c, which encodes k-bit information vector i is generated by multiplying the k-bit information vector with $k \times n$ bit generator matrix G, i.e., $c = i \cdot G$.

From section 6.5 we know that the EG-LDPC and PG-LDPC codes are cyclic codes. Figure 6.8 shows the generator matrix of $(15, 7)$ EG-LDPC code. As you can see all the rows of the matrix are cyclic shifts of the first row. This cyclic code generation does not generate a systematic code and the information bits must be decoded from the encoded vector, which is not desirable for our fault-tolerant approach due to the further complication and delay that it adds to the operation. The generator matrix of any cyclic code can be converted into systematic form ($G = [I : X]$) as shown in [78] and [79]. We used the procedure presented in [78] and [79] to generate systematic generator matrices of all the EG-LDPC and PG-LDPC codes

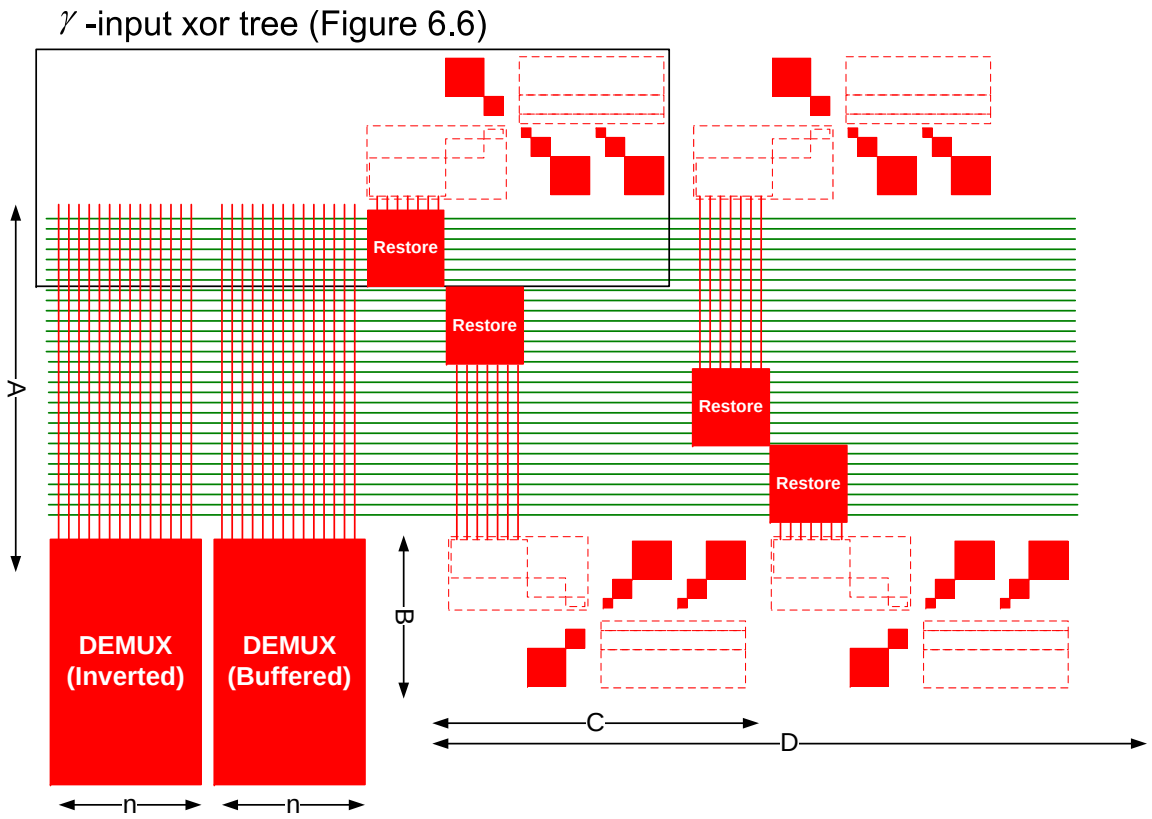


Figure 6.7. A detector circuit implemented on nanoPLA: The parameters in the figure are $A = n \times \gamma$, $B = 2 \times Pl(\gamma - 1)$, $C = Pl(2 \times \gamma - 2) + 2 \times Pl(\gamma - 1)$, and $D = n/2 \times C$.

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}
i_0	1	0	0	0	1	0	1	1	1	0	0	0	0	0	0
i_1	0	1	0	0	0	1	0	1	1	1	0	0	0	0	0
i_2	0	0	1	0	0	0	1	0	1	1	1	0	0	0	0
i_3	0	0	0	1	0	0	0	1	0	1	1	1	0	0	0
i_4	0	0	0	0	1	0	0	0	1	0	1	1	1	0	0
i_5	0	0	0	0	0	1	0	0	0	1	0	1	1	1	0
i_6	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1

Figure 6.8. The generator matrix of EG-LDPC code of $(15, 7)$ in cyclic format

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}
i_0	1	0	0	0	0	0	0	1	0	0	1	1	1	0	1
i_1	0	1	0	0	0	0	0	1	1	0	0	1	1	1	0
i_2	0	0	1	0	0	0	0	0	1	1	1	0	0	0	1
i_3	0	0	0	1	0	0	0	1	0	1	1	1	0	0	0
i_4	0	0	0	0	1	0	0	0	1	0	1	1	1	0	0
i_5	0	0	0	0	0	1	0	0	0	1	0	1	1	1	0
i_6	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1

Figure 6.9. The generator matrix of EG-LDPC code of $(15, 7)$ in systematic format, which consists of identity matrix in the left columns

under consideration.

figure 6.9 shows the systematic generator matrix to generate $(15, 7)$ EG-LDPC code. The encoded vector, which is generated by the inner product of the information vector and the generator matrix, consists of information bits followed by parity bits, where each parity bit is simply an inner product of information vector and a column of X, from $G = [I : X]$. Figure 6.10 shows the encoder circuit to compute the parity bits of the $(15, 7)$ EG-LDPC code. In this figure $i = (i_0, ..., i_6)$ is the information vector and will be copied to $c_0, ..., c_6$ bits of the encoded vector, c, and the rest of encoded vector, the parity bits, are linear sums (XOR) of the information bits.

If the building block is two-input gates then the encoder circuitry takes 22 two-input XOR gate. Since the systematic generator matrix of EG-LDPC and PG-LDPC codes does not have the standard row and column density, we have to construct all the generator matrices to compute the encoder area. To compute the area of an encoder circuitry the corresponding systematic generator matrix has to be constructed using the procedure in [78] and [79]. Once the systematic generator matrix is constructed

Table 6.2. The encoder circuit area in the number of 2-input gates

Type	EG	PG	EG	PG	EG	PG	EG	PG
Code	(15,7)	(21,11)	(63,37)	(73,45)	(255,175)	(273,191)	(1023,781)	(1057,813)
Enc. Area	22	47	355	643	6577	7429	93823	98730

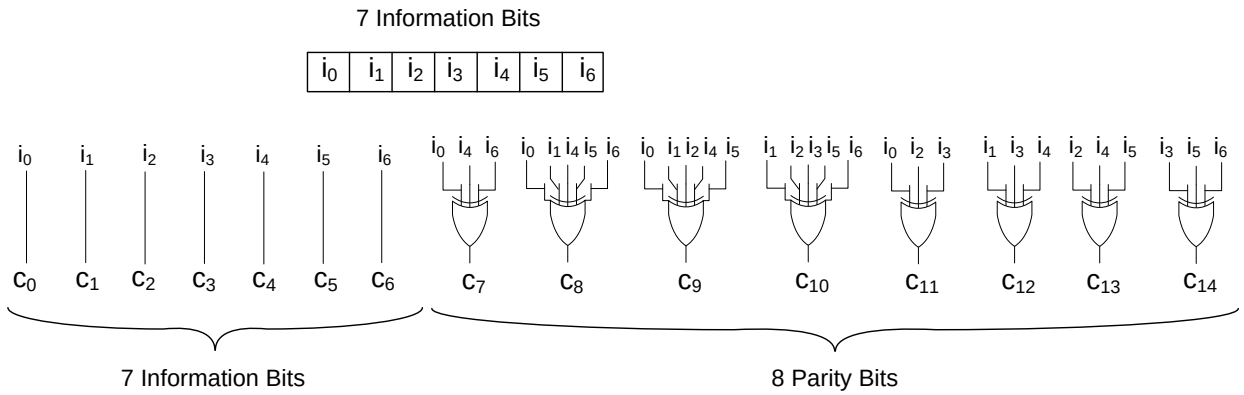


Figure 6.10. The structure of an encoder circuit for $(15, 7)$ EG-LDPC code. Each of the XOR gates generate one parity bit of the encoded vector. The code-word consists of 7 information bits followed by 8 parity bits.

the fanin size of the XOR gates can be determined by the column densities of the generator matrix. For example, consider the generator matrix of figure 6.9 which represents the systematic generator matrix of $(15, 7)$ EG-LDPC code. Columns 7 to 14 determines the XOR function of parity bits c_7 to c_{14}, as shown in figure 6.10. Table 6.2 shows the area of the encoder circuits for each EG-LDPC and PG-LDPC code under consideration. The fanin size of the XOR gates are determined by the column densities.

Once the XOR functions are known, the encoder structure is very similar to the detector structure shown in figure 6.7, except it consists of $(n - k)$ XOR gates. Each nanowire-based XOR gate has structure similar to the XOR tree shown in figure 6.6.

6.6.3 Corrector

One-step majority-logic correction is a fast and relatively compact error-correcting technique [79]. There is a limited class of ECCs that are one-step-majority cor-

rectable which includes type-I two-dimensional EG-LDPC and PG-LDPC. In this section we present a brief review of this correcting technique. Then we show the one-step majority-logic corrector for an EG-LDPC and PG-LDPC codes.

One-step majority-logic correction is the procedure that corrects one bit of the received encoded vector at a time. This method consists of two parts: (1) Generating a specific set of linear sums of the received vector bits. (2) Finding the majority value of the computed linear sums. The majority value shows the correctness of the code-bit under consideration. If the majority value is 1 the bit is inverted, otherwise it is kept unchanged.

A linear sum of the received encoded vector bits can be formed by computing the inner product of the received vector and a row of a parity-check matrix. This sum is called *Parity-Check* sum. A set of parity-check sums is said to be orthogonal on a given code bit if each of the parity-check sums include the code bit but no other code bit is included in more than one of these parity-check sums. If for each code bit there are j parity-check sums that are orthogonal on it, then the code is *one-step majority-logic correctable* up to $\lfloor j/2 \rfloor$ bit errors. This proof can be found in [79]. In a cyclic code, a set of j parity-check sums orthogonal on a code-word bit is orthogonal on all the n code-word bits. Therefore, using one set of parity-check matrix rows orthogonal on one code bit, we can design a majority circuit that corrects all the other bits, serially.

The one-step majority logic error correction is summarized in the following procedure. These steps correct a potential error in one code bit, e.g., c_{n-1}.

1. The j parity-check sums orthogonal on c_{n-1} are formed by computing the inner product of the received vector and the appropriate rows of parity-check matrix.

2. The J orthogonal check sums are fed into a majority gate. The output of the majority gate corrects the bit c_{n-1}, by inverting the value of c_{n-1} if the output of majority gate is "1".

It is shown in [79] that type-I two-dimensional EG-LDPC and PG-LDPC codes are one-step majority logic correctable up to $\gamma/2$ bits, meaning that there are γ

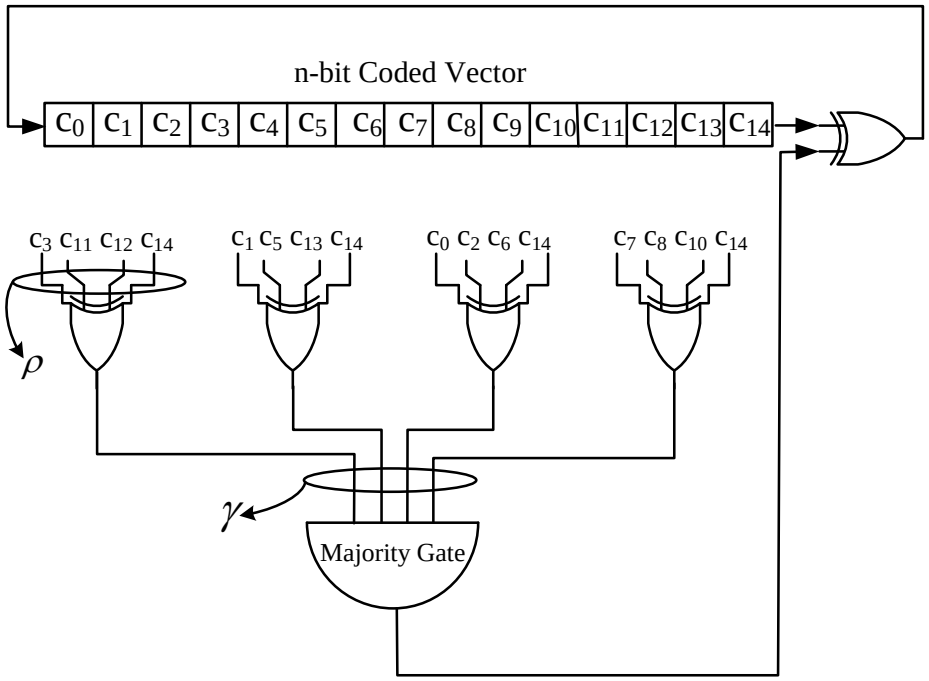


Figure 6.11. The serial one-step majority logic corrector structure to correct last bit (bit 14th) of 15-bit $(15, 7)$ EG-LDPC code.

parity-check sums orthogonal on a code-bit, therefore$j = \gamma$.

The circuit implementing a serial one-step majority logic corrector for $(15, 7)$ EG-LDPC code is shown in figure 6.11. This circuit generates γ parity-check sums with γ XOR gates and then computes, the majority value of the parity-check sums. Since each parity-check sum is computed using a row of the parity-check matrix and the row density of EG-LDPC codes are ρ then each XOR gate that computes the linear sum has ρ inputs. The single XOR gate on the right, corrects the code bit c_{n-1}, using the output of the majority gate. Once the code bit c_{n-1} is corrected the code-word is cyclic shifted and code bit c_{n-2} is placed at c_{n-1} position and will be corrected. The whole code-word can be corrected in n rounds.

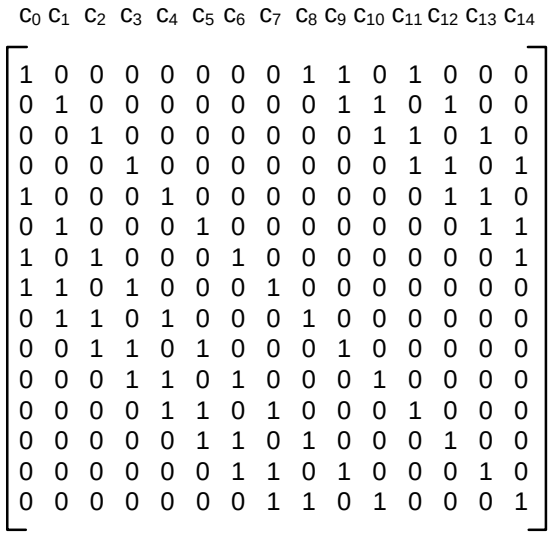


Figure 6.12. The parity-check matrix of $(15, 7)$ EG-LDPC code.

If implemented in flat, two-level logic, a majority gate could take exponential area. The majority gate is implemented by computing all the $\begin{pmatrix} \gamma \\ \lceil \frac{\gamma+1}{2} \rceil \end{pmatrix}$ product terms that has $\frac{\gamma+1}{2}$ ON input and one $\begin{pmatrix} \gamma \\ \lceil \frac{\gamma+1}{2} \rceil \end{pmatrix}$-input OR-term. For example, the majority of 3 inputs a, b, c is computed with $\begin{pmatrix} 3 \\ 2 \end{pmatrix} = 3$ product terms and one 3-input OR-terms as below,

$$Majority(a, b, c) = ab + ac + bc. \tag{6.5}$$

In the following section we present a compact implementation for the majority gate using *Sorting Networks* [83]. The majority gate has application in many other error-correcting codes, and this compact implementation can improve many other applications.

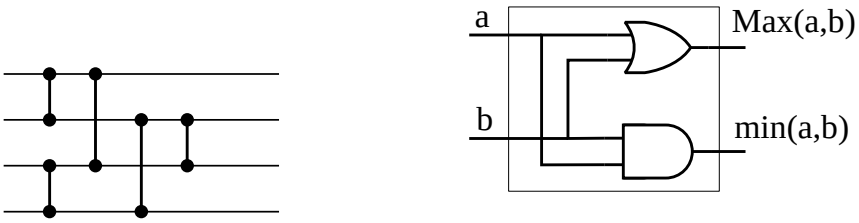
6.6.3.1 Majority Implementation

A majority function of γ binary digits is simply the median of the digits (were we define the median of even number of digits the $\gamma/2 + 1$st smallest digit).

To find the median of the γ inputs we do the following:

1. Divide the γ inputs into two halves with size $\gamma/2$.
2. Sort each of the halves
3. The median is 1 if for $i = 1, 2, ..., \gamma/2$ the ith element of one half and the $(\gamma/2 + 1 - i)$th element of the other half are both 1.

We use binary *Sorting Network* to do the sort operation of the second step efficiently. An n-input sorting network is the structure that sorts a set of n bits, using a 2-bit sorter building blocks. Figure 6.13(a) shows a 4-input sorting network. Each of the vertical lines represents one comparator, which compares two bits and assign the larger one to the top output and the smaller one to the bottom (figure 6.13(b)).



(a) (b)

Figure 6.13. (a) A 4-input sorting network Each of the vertical lines shows a 1-input comparator. (b) One comparator structure

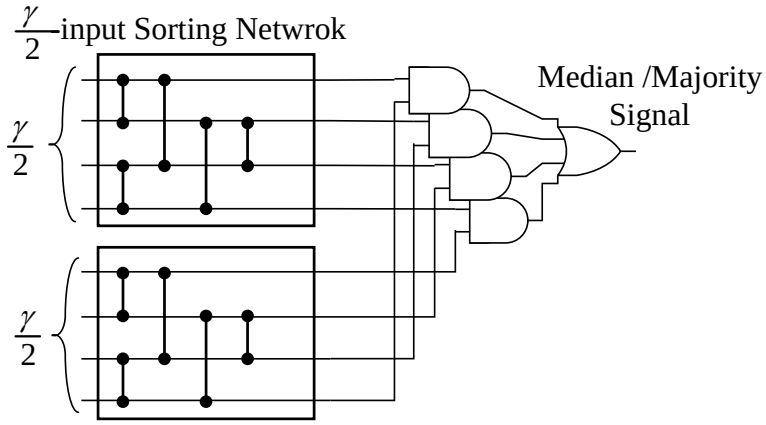


Figure 6.14. An 8-input majority gate using sorting network

The 4-input sorting network has 5 comparator blocks, where each block consists of two 2-input gates, so overall the 4-input sorting network consists of ten 2-input gates in total.

To check the condition in the third step we use $\gamma/2$ 2-input AND gates followed by a $\gamma/2$-input OR gate. Figure 6.14 shows the circuit implementing the above technique to find the median value of 8 bits. It has two $\gamma/2$-input (4-input) sorting networks followed by combinational circuitry, consisting of four 2-input AND gates and a 4-input OR gate, which can be implemented with three 2-input OR gates. Therefore, in total an 8-input majority gate implemented with sorting networks take 27 2-input gates; In contrast, the two-level implementation of this majority gate takes $\begin{pmatrix} 8 \\ 5 \end{pmatrix} = 56$ 5-input AND gates with one 56-input OR gate.

As mentioned earlier, the same one-step majority-logic corrector can be used to correct all the n bits of the received code-word of a cyclic code. To correct each code-bit, the received encoded vector is cyclic shifted and fed into to the XOR gates as shown in figure 6.11. The serial majority corrector takes n cycles to correct an erroneous code-word. If the fault rate is low the corrector block is used infrequently, and since the common case is error free code-words, the latency of the corrector will not have a severe impact on the average memory read latency. Figure 6.2 shows the system integration with serial corrector. The serial corrector is placed off the normal memory read path. The memory words retrieved from memory unit is checked by detector unit. If the detector detects an error, the memory word is sent to the corrector unit to be corrected, which has the latency of the detector plus the n round latency of the corrector. Since the serial corrector is used when the fault rate is low, the corrector is used infrequently and consequently not having severe throughput impact. However, for high error rate (e.g., when tolerating permanent defects in memory words as well), the corrector is used more frequently and its latency can impact the system performance. Therefore, we can implement a parallel one-step majority corrector which is essentially n copies of the single one-step majority-logic corrector. Figure 6.3 shows a system integration using the parallel corrector. All the memory words are

Table 6.3. The corrector circuit area in the number of 2-input gates

Type	EG	PG	EG	PG	EG	PG	EG	PG
Code	(15,7)	(21,11)	(63,37)	(73,45)	(255,175)	(273,191)	(1023,781)	(1057,813)
Cor. Area	19	32	83	108	331	376	1263	1358

pipelined through the parallel corrector. This way the corrected memory words are generated every cycle. The detector in the parallel case monitors the operation of the corrector, if the output of the corrector is erroneous, the detector signals the corrector to repeat the operation. Detecting a fault in the corrected memory word is not because of the fault in the memory words, it is solely because of faults in the detector and corrector circuitry. Since detector and corrector circuitry are relatively small compared to the memory system, the failure rate of these units is very low, and therefore, the error detection and repeat process happens very infrequently, and do not impact the system throughput.

Assuming our building blocks are 2-input gates, the XOR gates to generate γ number of ρ-input parity-check sums will take $\gamma \times (\rho - 1)$ 2-input XOR gates. The size of the majority gate is defined by the sorting network implementation. Table 6.3 shows the overall area of a serial one-step majority-logic gate in the number of 2-input gates for the codes under consideration. The parallel implementation consists of exactly n copies of the serial one-step majority-logic.

Generating the linear binary sums (XORs) of the one-step majority-sum is the same as figure 6.7. The majority gate is simply computed following the structure shown in figure 6.14 using the nanowire-based substrate.

6.6.4 Banked Memory

Large memories are conventionally organized as sets of smaller memory blocks called banks. The reason for breaking a large memory into smaller banks is to trade-off overall memory density for access speed and reliability. Excessively small bank sizes will incur a large area overhead for memory drivers and receivers. Large memory banks require long rows and columns which results in high capacitance wires that consequently increases the delay. Furthermore long wires are more susceptible to

breaks and short defects when they are excessively long. Therefore, excessively larger memory banks have higher defect rate and lower performance. The organization of nanoMemory is not different than the conventional memory organization, except that the overhead per bank is larger due to the scale difference between the size of a memory bit (a single wire crossing) and the support structures (e.g., microscale wires for addressing and bootstrapping). The work presented at [49] provides more detail on memory banks and shows how the banks would be integrated into a complete memory system.

The memory system overview shown in figure 6.3 can be generalized to multiple banks as shown in figure 6.15. The encoder encodes the information bits and send it to the memory banks. The bank that contains the "write" address will be selected and the memory word is written into the memory bank. A fault-secure detector monitors the operation of the encoder. If the output vector is not a valid code-word, then the detector sends a feedback to the encoder to repeat the encoding process. When reading a memory word form the memory, the memory word is pipelined through the corrector unit. A fault-secure detector following the corrector unit, monitors the output vector of the corrector unit. If it detects an error, it send a signal to the corrector unit and the corrector unit must perform its process again.

Memory words have to be scrubbed frequently to prevent error accumulation in them. However, scrubbing memory words one by one can take long time especially if scrubbing has to occurs often due to high fault rate, and therefore, it can seriously reduce the system performance. To prevent that we can potentially scrub all the memory banks in parallel. For this, each memory bank requires a separate corrector and detector unit. This model is shown in figure 6.16. This model takes one-quarter of the original scrubbing latency. However, having one corrector and detector for each bank decreases the density of the memory dramatically. Furthermore the system performance is often high enough even when more than one memory bank share one corrector and detector. Therefore, we can cluster a number of memory banks together and consider a corrector and detector unit for each *Cluster*. Figure 6.17 shows a memory system with two parallel corrector units. Here each cluster contains

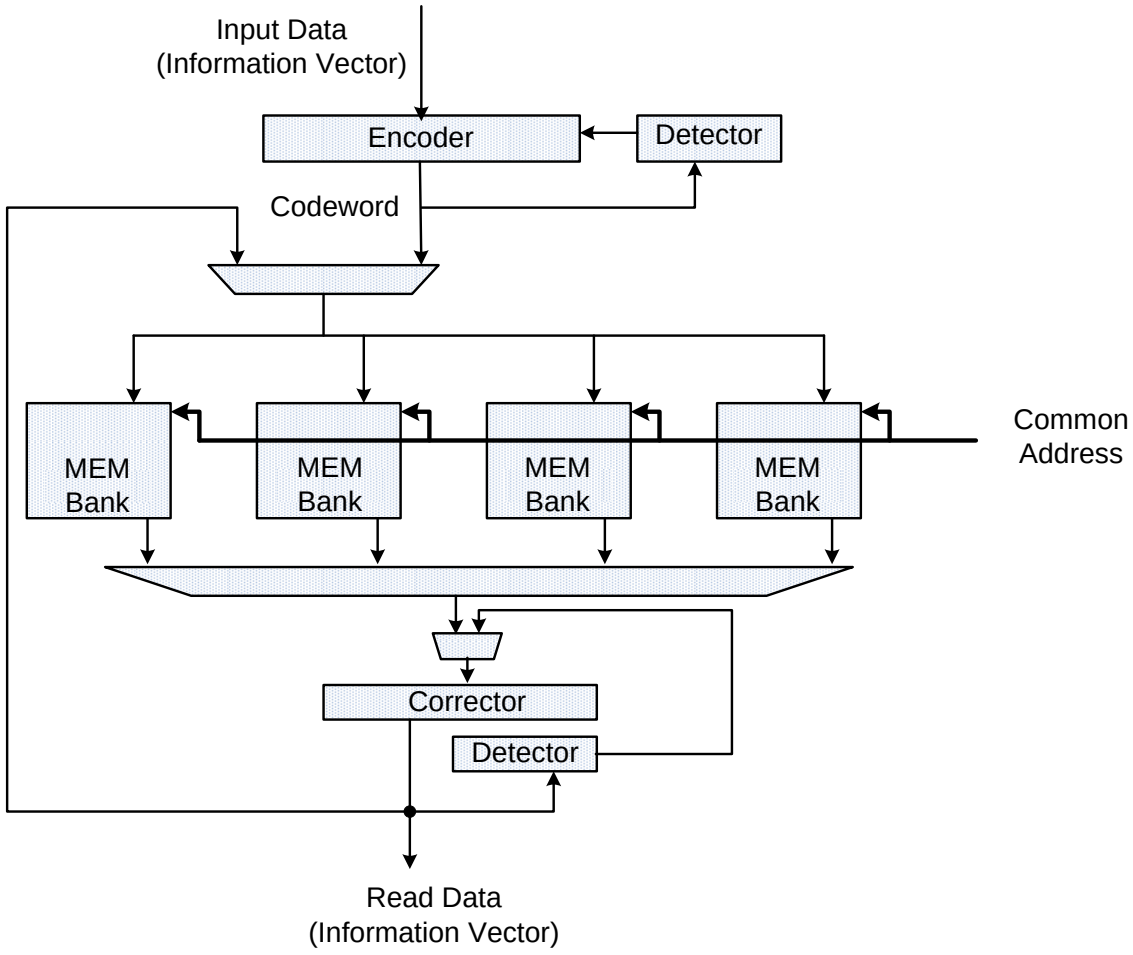


Figure 6.15. Banked memory organization, with single global corrector.

two memory banks. This model takes half of the original scrubbing latency.

6.6.5 Nanoscale Demultiplexer

The nanoMemory architecture introduced in [50] and reviewed in section 2.4 has a lithographic scale interface. Each memory bit (a nanowire cross-point) is addressed from two lithographic scale to sublithographic scale decoders and the value of the bit is read by a common read line (see figure 2.11). To provide multiple-bit access to and from a single memory bank, we simply split the common read line into separate microscale connections to the nanowire array (see the right-hand side of figure 2.13). We then program up the nanowire addresses so that the same address is present in

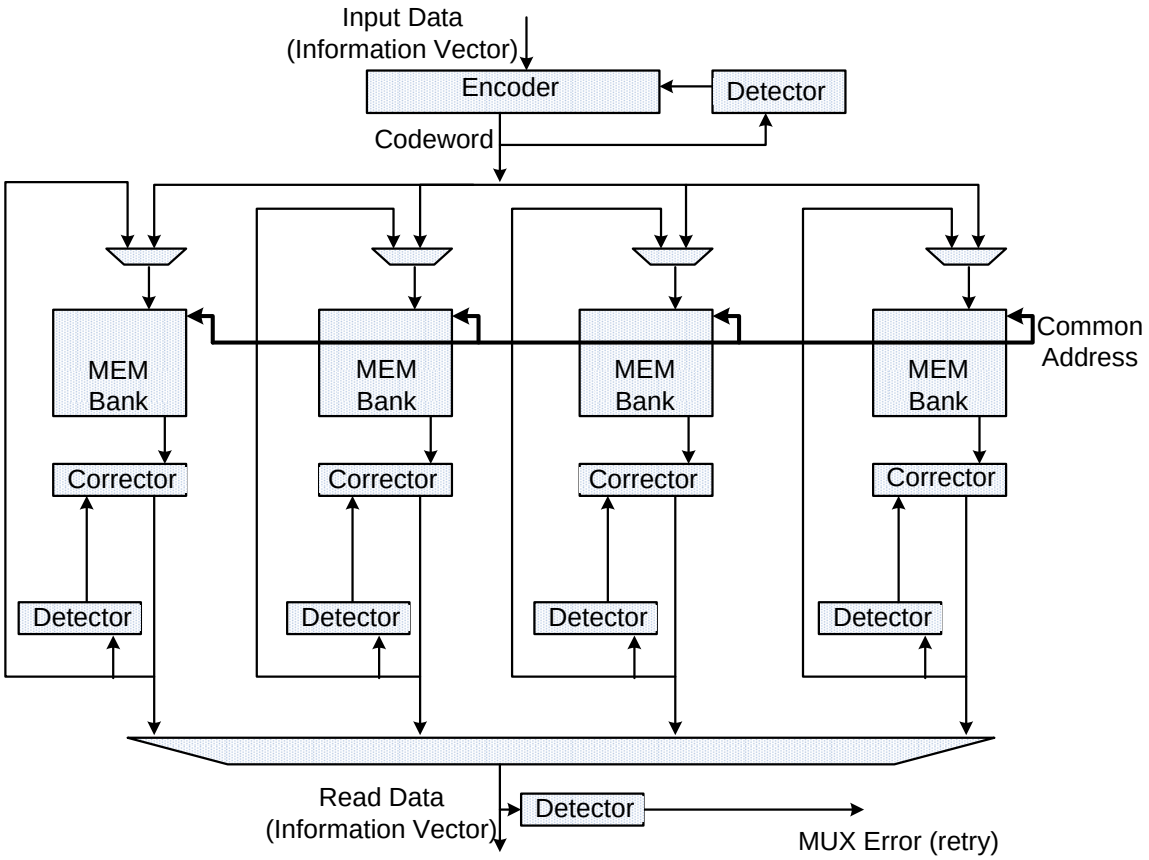


Figure 6.16. Banked memory organization with, fully parallel correcting units (cluster of size 1).

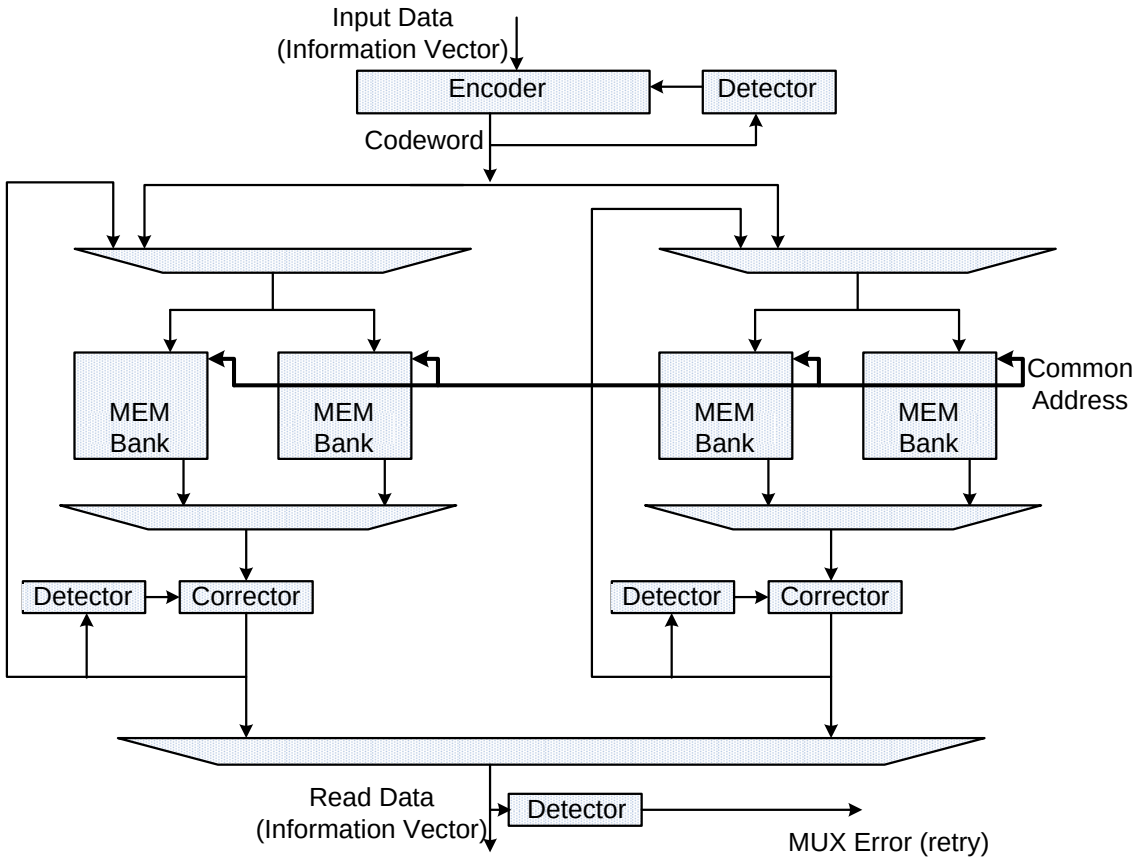


Figure 6.17. Banked memory organization, with cluster size of 2.

each of the nanowire bundles associated with a distinct microscale output contact.

For our fault-tolerant system where the supporting logic is implemented at the sublithographic scale, the read and write operation to the memory core is done through sublithographic scale units. Therefore, there is no need to have lithographic scale to sublithographic scale memory read line. To integrate the supporting sublithographic units (i.e., detector, encoder, and corrector) with the memory core we need a sublithographic demultiplexer (DEMUX). The r-wide rows of the memory core are demultiplexed into n-wide memory words. The main part of the DEMUX is the gate-able nanowires (section 2.2.2 and [84]). A simplified overview of this DEMUX is shown in figure 6.18 with $r = 12$ rows and memory word width of $n = 3$. Each of the DEMUX nanowires (vertical wires) is gate-able by r/n nanowires of the memory row, where r is the number of memory rows, and n is the memory word width. To access a memory word with this DEMUX, the column address is placed on the lithographic address wires of the column decoder. The row address decoder is programmed in a way that selects exactly one nanowire in each DEMUX lightly doped regions. To read a memory word, the DEMUX outputs are precharged to high value. During the evaluation phase, the memory word nanowires are selected by the row and column address decoders. The memory word nanowires representing each memory word controls the current of the DEMUX nanowires. Those memory word nanowires that represents "0" memory bits, carry low voltage, and therefore, let the DEMUX wires conduct (when DEMUX wires are P-typed) the low voltage from the "GND" wire to their outputs. Those nanowires that carry high voltage do not let the current go through, and the DEMUX outputs will keep the high precharged voltage.

figure 6.19 shows an example of selecting a memory word of width $n = 3$. The column address selects a single nanowire. The row address selects the nanowires representing the memory word bits at their junctions with the selected column nanowire. Each row nanowire representing a memory bit of the selected memory word, gates exactly one DEMUX output. Therefore, the $n = 3$ memory word signals are present at the DEMUX output.

figure 6.20 (a) shows a DEMUX with ideal nanowire placement. However, similar to

the restoration plane (section 2.3.2) and lithographic to sublithographic decoder (section 2.3.3) the DEMUX can only be stochastically fabricated. We do not have control over the placement, alignment, or even the doping pattern selection of nanowires in an array. Therefore, it is not possible to select and align of the DEMUX nanowires perfectly as shown in figure 6.18 (a). The idea is to increase the number of valid pattern combinations and placements; so that when nanowires are selected and placed randomly, with higher probability a valid configuration exists. The perfect DEMUX shown in figure 6.18 (a), has only $3! = 6$ valid configuration among all the possible alignments and selections. If we separate the doping patterns as shown in figure 6.18 (b), there are more valid configurations, i.e.,

$$\begin{pmatrix} 12 \\ 4 \end{pmatrix} \begin{pmatrix} 8 \\ 4 \end{pmatrix}. \tag{6.6}$$

figure 6.18 (b) shows a perfect alignment and selection with disjoint doping regions, however, with stochastic placement the practical DEMUX would be similar to figure 6.18 (c). Note that the DEMUX nanowires are overpopulated to guarantee existence of all the required patterns. In this demonstration example in figures 6.18 (c), the three left most nanowires make a functional DEMUX and the rightmost nanowire is not used.

6.7 Reliability Analysis

In this section we analyze the reliability of the system. To measure the system reliability, we estimate the probability that system fails, i.e., system experiences more number of errors in a memory word than the number of errors the error-correcting code can tolerate. With this analysis we then show the impact of protecting the ECC supporting logic in section 6.7.2.

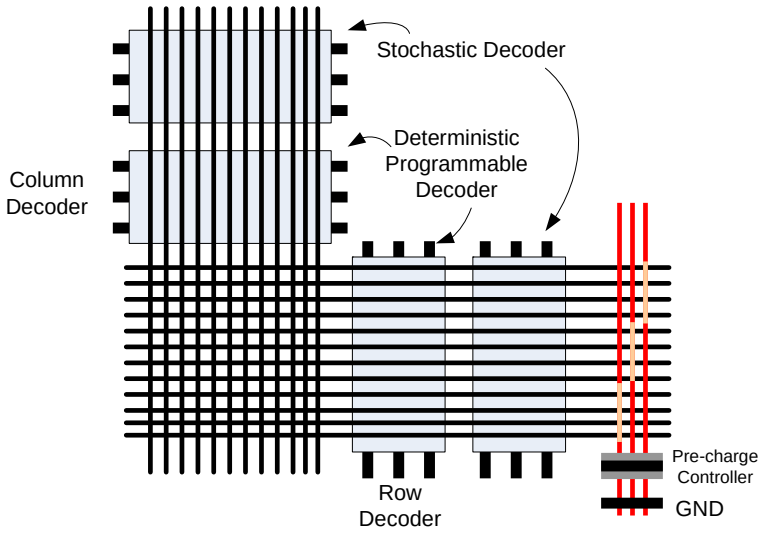


Figure 6.18. A simple DEMUX for $n = 4$, and $r = 12$

6.7.1 Analysis

We assume the fault probability of each device at each cycle (P_f) has i.i.d. and random distribution over the devices of the memory system. Recall e_e and e_{de} are the nominal number of errors that occurs in encoder and decoder during memory write operation. Similarly, e_m, e_c, and e_{dc} are the number of errors that occur in a memory word and its corresponding corrector and detector. Let n_e, n_c, and n_d be the size of the circuitry involved in an operation on a single code-bit in the encoder, corrector, or detector, respectively. This is the size of the logic cone of a single output of each of the above units. For example, in a detector each logic cone is a ρ-input XOR gate generating a single bit of the syndrome vector.

Let a nominal unit have a logic cone size of x. With worst-case analysis the output of the logic cone fails when any of the devices in the logic cone fails. So when at least one of the x devices inside the cone is erroneous, the output of the logic cone, which is a code-bit, would be erroneous. Therefore, the probability that a code-bit is

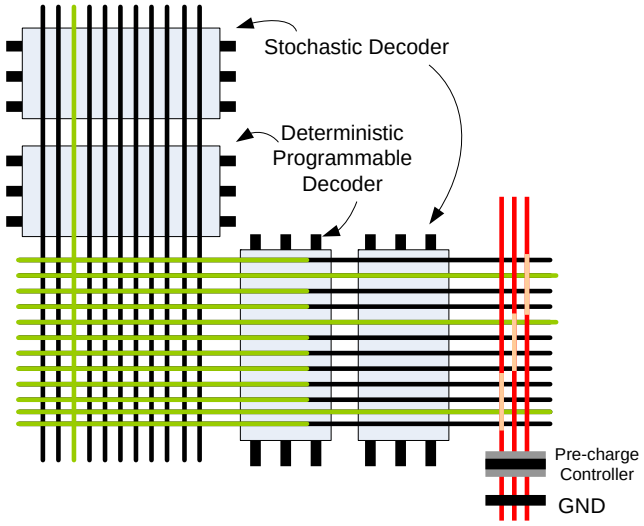


Figure 6.19. An example of the memory word selection with the nanoscale DEMUX

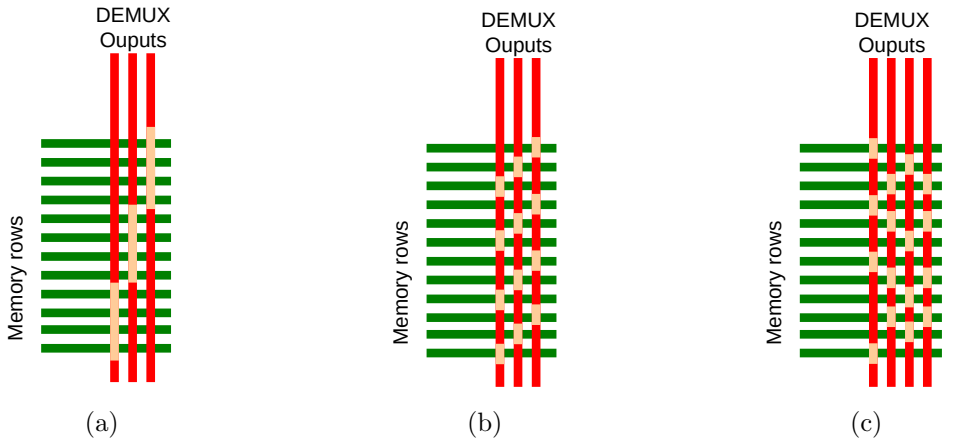


Figure 6.20. Stochastic DEMUX structure connected region, (a) Ideal DEMUX, (b) Ideal DEMUX with repeating regions, (c) Stochastic DEMUX with random pattern.

erroneous in any of the above units is

$$P_{bit\_circuit} = 1 - (1 - P_f)^x, \tag{6.7}$$

where x has one of the values: n_e, n_c, or n_d. Similarly the probability that a memory-bit is erroneous in scrubbing interval of s cycles is

$$P_{bit\_mem} = 1 - (1 - P_f)^{xs}, \tag{6.8}$$

where x is the number of devices contained in one memory cell. If using SRAM cells, $x = 6$. If using a nanoMemory, then the memory bit is essentially a single nanowire cross-point. However, since accessing each memory bit requires reading the signal value through a pair of nanowires (see figure 6.19), the correctness of each memory bit depends on the correctness of two nanowires. Therefore, for the nanoMemory design $x = 2$. Each unit or a memory-word experience e errors among n bits of the code-word with the probability

$$P_{unit} = \begin{pmatrix} n \\ e \end{pmatrix} {P_{bit}}^e (1 - P_{bit})^{n-e}, \tag{6.9}$$

which is simply a binomial distribution, and n is the code-length, P_{bit} is either P_{bit_mem} or $P_{bit\_circuit}$ and e is e_e, e_m, e_c, e_{de}, or e_{dc}.

As explained in section 6.6.1, errors in the encoder unit are detected by its following detector, and are corrected by repeating the encoding operation to generate a correct encoded vector. The detector can detect up to $d - 1$ errors overall in these two units. Where d is the code distance. With worst-case assumptions, the detector fails to detect the errors if there are more than $d - 1$. Therefore, we define the first reliability condition as

(Condition I) $\qquad e_e + e_{de} < d,$

which states that the total number of errors in the encoder unit and the following detector unit must be smaller than the minimum distance of the code. The detector

of the corrector is also capable of detecting up to $d-1$ errors accumulated from memory unit, corrector unit and the second detector unit. Similarly with worst-case assumption, detector fails to detect errors when they are more than $d-1$. Therefore, the second reliability condition is defined as

(Condition II) $e_m + e_c + e_{dc} < d.$

Furthermore the corrector can recover a memory-word with up to $\gamma/2$ errors from the memory unit. If more than $\gamma/2$ errors are accumulated in a memory word, then the EG-LDPC and PG-LDPC codes cannot correct the memory word. Therefore, the third reliability condition is formulated as

(Condition III) $e_m \leq \gamma/2,$

which states that the maximum number of tolerable errors in each memory word is $\gamma/2$. Satisfying the three above conditions guarantees that the memory system operates with no undetectable or uncorrectable errors. We calculated the probability of each of the above conditions employing equation (6.9), for all the various EG-LDPC and PG-LDPC codes. Section 6.9 illustrates the reliability of the system for different device failure rate, P_f. It also presents the optimum design points for optimizing reliability, area and throughput.

6.7.2 The Impact of Providing Reliability for Supporting Logic

It is important to understand the impact of protecting the supporting logic on the system FIT rate. Could the system FIT rate be low enough if only memory words were protected? What is the potential cost of protecting the supporting logic? We answer these question with the example below.

Figure 6.21 shows the FIT rate of the system decomposed into the contribution from the memory bank and the contribution from the supporting logic. The FIT in the supporting logic is without a fault-secure detector (*i.e.*, any error in the supporting logic results an erroneous output, with worst-case analysis). Obviously the FIT of the whole system with no logic protection is the sum of the above two FITs, illustrated

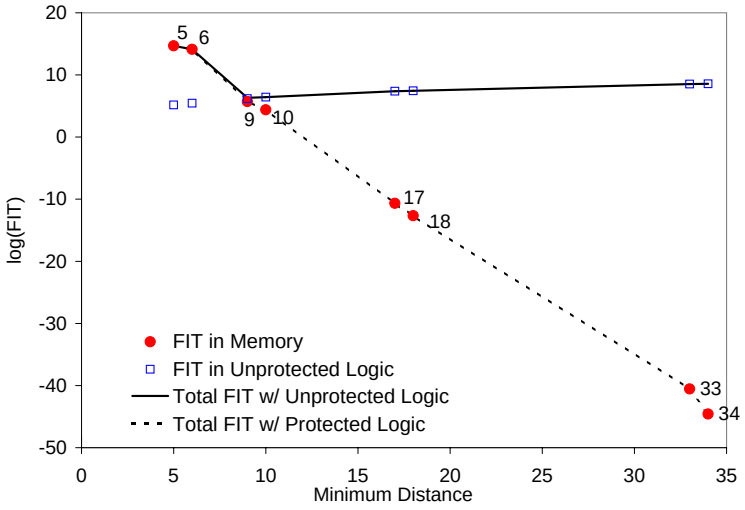


Figure 6.21. The impact of protecting logic on system reliability, for $P_f = 10^{-19}$.

with a solid line. This graph is plotted for a device fault rate of $P_f = 10^{-19}$ with memory scrubbing interval of 10 minutes. As you can see, for codes with minimum distance larger than 9, the FIT of the system with no logic protection is dominated by the FIT of the unprotected logic. Using codes with greater redundancy will decrease the FIT of memory bank; however, since the unprotected logic has a non-trivial FIT rate, increasing the code redundancy without protecting the logic does not decrease the FIT of the composite system. To achieve the higher reliability the logic must also be protected. The FIT of such system with fault secure logic is illustrated with the dashed line, and as you can see, the FIT of this system follows very closely the FIT of the memory bank. Protecting supporting logic, is essentially done by the fault-secure detector and the only cost to achieve the logic protection is the cost that we pay for the detectors. Table 6.4 shows that the detector takes a negligible fraction of area compared to the memory core, encoder, and corrector. Therefore with minimal area overhead the system reliability can be greatly improved.

6.8 Tolerating Permanent Defect in Memory Cells

In this chapter, so far we focused on a transient fault-tolerant memory structure. Equally important is to tolerate permanent defects in the memory systems. Since the memory unit has a regular structure, most of the defect tolerant designs are based on row and column sparing. This means that we overpopulate the rows and columns based on the expected defect rate so that after removing all the defecting rows and columns the memory has the desired size. For example, if we want to have a 1K×1K memory, and the junction defect rate is 0.001% which results in nanowire defect rate of $P_{wire} = 1000 \times 10^{-5} = 1\%$, then with 2% row and column overpopulation (20 more rows and columns) the system yields 1K×1K memory core with 99.6% probability

$$P_{row\_yield} = P_{column\_yield} = \sum_{i=0}^{20} \begin{pmatrix} 1020 \\ i \end{pmatrix} P_{wire}^{i}(1 - P_{wire})^{1020-i} \approx 0.9981, \quad (6.10)$$

and the memory yields when both rows and columns yield which is the product of the two probability P_{row_yield} and P_{column_yield},

$$P_{row\_yield} \times P_{column\_yield} = 0.9981^2 = 0.996. \quad (6.11)$$

So when the defect rate is small (0.001%) even with very small area overhead (2% overpopulation) the system can yield a perfect 1K×1K with high probability. However, with higher defect rate, the column and row sparing can be very costly. For example, a 1% cross-point defect rate on a nanowire with 1000 junctions implies that with almost 100% probability any nanowire in the memory block has at least one defective junction. At this defect rate we cannot afford discarding nanowires with any defective junctions. The work presented in [49] suggests a defect tolerant technique that is more efficient than the column and row sparing. This technique suggest discarding nanowires that have more defective junctions than a set threshold. These nanowires will be replaced with spare nanowires. The limited number of defective junctions on the remaining nanowires will be tolerated using ECCs.

For the above example with junction defect rate of 1%, and keeping nanowires with up to 12 defective junctions, then the system would require only 31% row and column overpopulation to achieve the 99% yield. The computation is shown below:

The probability that a wire is accepted, i.e., has at most 12 defective junction is

$$P_{wire} = \sum_{i=0}^{12} \begin{pmatrix} 1000 \\ i \end{pmatrix} (0.01)^i (0.99)^{1000-i} \approx 0.792. \tag{6.12}$$

With only 31% column and row sparing we can get 99% yield,

$$P_{row\_yield} = P_{column\_yield} = \sum_{i=0}^{310} \begin{pmatrix} 1310 \\ i \end{pmatrix} P_{wire}^i (1 - P_{wire})^{1310-i} \approx 0.9953. \tag{6.13}$$

and the final memory yield is

$$P_{row\_yield} \times P_{column\_yield} = 0.9953^2 = 0.99. \tag{6.14}$$

In [49], it is suggested that a reliable lithographic-scale encoder and decoder be used to tolerate limited defective bits in each row.

Here we use the same EG-LDPC and PG-LDPC codes that we use for transient faults for tolerating permanent defects as well, and the error correction capability of the ECC is partitioned between fault-tolerant and defect-tolerant requirement. For example the EG-LDPC code of $(255, 175, 17)$ can correct up to 8 errors in each memory word. This can be partitioned to tolerate 4 transient faults and 4 permanent defect in each junction. With this technique we allow each memory word to contain up to 4 defective junctions which with a row width of 1000 is about 12 defective junctions per row, since each row takes three memory words and each memory word tolerates 4 defects.

This reduces the area overhead compared to solely row and column sparing. The important point is that this area overhead reduction is achieved with almost no extra cost, since it uses the structure which already exists for transient fault-tolerance, i.e., encoder, corrector, and detector units. The only potential drawback point for this

technique is that it increases the effective FIT rate of the memory block relative to the case where we had all 8 errors available to tolerate transient upsets. Therefore, to guarantee the desired reliability, codes with larger minimum distance which can tolerate larger number of defects and faults can be used. Section 6.9 shows more detail results on how the reliability and area overhead costs are balanced.

In section 6.6.3.1 we showed serial and parallel corrector integration for low and high fault rates respectively. However, when we also have a limited number of defective junction in memory words, this means that with high probability a memory word has erroneous bits and must be corrected. Therefore, as mentioned in section 6.6.3.1, we use a parallel fully pipelined corrector to prevent throughput loss. The probability that a memory word has a limited number of defects is computed below. Remember that the memory words with more than a set threshold is removed. Let the set threshold of the number of defects in each memory word be D_{thr}, and the defect rate be P_d, therefore, the memory words have 0 to D_{thr} defective bits. The probability that a memory word is defective and require correction is

$$P_{def\_mem\_word} = \frac{\sum_{i=1}^{D_{thr}} \begin{pmatrix} n \\ i \end{pmatrix} P_d^i (1-P)^{(n-1)}}{\sum_{i=0}^{D_{thr}} \begin{pmatrix} n \\ i \end{pmatrix} P_d^i (1-P)^{(n-1)}}. \tag{6.15}$$

For example for 1% defect rate and tolerating up to 4 defective bits per memory word, the probability that a memory word has defective bit is $P_{def\_mem\_word} = 0.91$. Therefore, 91% of the memory words must be corrected, which justifies the parallel and fully pipelined corrector. In the following section we show the effect of D_{thr} on area, throughput, and reliability.

6.9 Area and Performance Analysis and Results

There are three design aspects that are specifically important when designing fault-tolerant designs: system reliability, area overhead, and performance. It is important

to see how these three factors interact and generate various design points. There are multiple design parameters that determines the balance between reliability, area, and performance. Among these parameters, those that we keep variable in our simulations are:

- The maximum number of defects which exist per memory word (D_{thr})
- The scrubbing interval (S)
- The cluster size of memory banks for scrubbing (C)

For a fixed memory size, bank size, and transient fault rate, we find the right value for D_{thr}, S, and C to achieve the desire area, performance, and reliability. Here we review the impact of each of these parameters on area, performance, and reliability.

The threshold on the number of defects per memory word affects area, reliability, and performance all together. It is first suggested to reduce the area overhead of the nanoMemory core [49]. The example presented in section 6.8 in equation (6.13) shows how the area overhead can be reduced by large enough D_{thr}. Increasing D_{thr} however, can potentially decrease the system throughput. Equation (6.15) shows the frequency that the memory word must be corrected. As mentioned in the previous section, the throughput loss is improved by using a parallel and fully pipelined corrector, which takes greater area. So increasing D_{thr}, increases the throughput loss or area overhead. The third impact of the value of D_{thr} is on the reliability. D_{thr} shows the part of the code error-correction capability that is assigned to defect-tolerant. This means that the code has weaker capability for tolerating transient faults, and therefore, has lower system reliability.

The scrubbing interval length, S, impacts the system performance and the reliability. The longer the scrubbing interval is, the less reliable the system will be, because more errors can accumulate in each memory word during longer scrubbing intervals. Equation (6.8) shows how scrubbing interval impacts the reliability of each single memory bit. However, the shorter the scrubbing interval is the lower the throughput will be, because the system puts more time on performing the scrubbing operation.

Below we show how the value of S impacts the system throughput. Assume a memory system has bank size of B and cluster size of C. This means that every C memory banks share one corrector and detector to perform the scrubbing operation. During each scrubbing operation all the $C \times B$ memory words in each cluster will be read, corrected, and written back into the memory. With fully pipelined parallel corrector this takes about $B \times C$ cycles. If the scrubbing interval is S cycles then the system throughput loss is

$$Throughput\ loss = \frac{B \times C}{S}. \tag{6.16}$$

If S is too small, the throughput loss is large. The impact of memory bank cluster size and memory bank size is also clear on the system throughput, larger memory bank size and cluster size increase the throughput loss. In this work we set the memory bank size to 1K×1K, to achieve the high enough memory density, which follows the detail analysis on memory bank size provided in [50]. Here we vary cluster size to optimize the throughput and area overhead. When the cluster size is large, the parallel corrector is shared among a large number of memory words, and therefore, the area of the corrector and detector is amortized out over a large number of memory words. However, the throughput loss can increase for large cluster size (equation (6.16)). In contrast, small cluster size, reduces the throughput loss but increases the net area per bit.

For our simulation we set the limit of throughput loss to $< 0.1\%$ and reliability to < 1000 FIT, and then minimize the area overhead. Figure 6.22 shows the reliability of different codes for different D_{thr}. We also provided the case with $D_{thr} = 0$, which means that the EG-LDPC and PG-LDPC codes are solely used for transient faults and the permanent defects are handled with a separate ECC. Setting the limit on the throughput and reliability, determines the values of S, scrubbing interval and C, cluster size of memory banks.

figure 6.22 plots the reliability of the systems that satisfy the throughput loss limit and reliability limit while achieving the minimum area overhead. The decomposed area of these design points is shown in Table 6.4. All of the above design points are

for memory size of 10^{12} bits. For these simulation, we assume a memory unit with the following parameters: lithographic wire pitch of 105 nm, nanowire pitch of 10 nm, defect rate of 0.01 per memory junction, and the memory bank size of 10^6 bits.

figure 6.23 plots the total area per bit for different memory sizes, when the fault rate is $10^{(-28)}$. The area of the memory banks are computed following the area model provided in [50]. The area of the supporting units (encoder, corrector, and detector) is computed using the area model provided in section 2.1.3.3 for each of the units.

The codes $(255, 175)$ tolerating 4 defects per memory words and $(63, 37)$ tolerating 2 defects, are the minimum design points for EG-LDPC codes. The area overhead of the code in the flat part of the curve is defined by multiple factors:

1. The code overhead (n/k)

2. The constant area overhead of the memory bank that decrease for larger memory size

3. The number of accepted defective junctions per memory word, D_{thr}

The final area per bit shown in figure 6.23 is the combined result of the above factors. The code overhead (n/k) is smallest for larger codes; e.g., $(273, 191)$ has the lowest code overhead. If all the other costs where amortized out over the large number of memory bits we would expect that the largest code shows the lowest area per memory bit. However, one important fact that dominates the code overhead is the limited memory bank size. We set the memory banks size fixed at 1 Mbit, and for a cluster of memory banks, which can be from 1 to 1000 banks per cluster. We dedicate one corrector and one detector (figure 6.17). The area overhead of these units are amortized over the memory bits of a cluster and is not reduced by the total memory size increase. The only costs that are reduced as the total memory size grow is the global encoder and its detector units that are shared among all the memory bit.

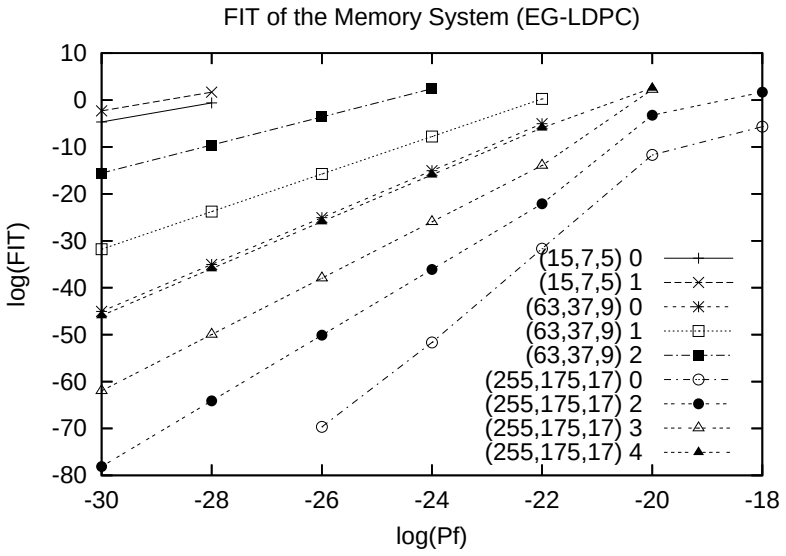
Finally the memory core area per bit plays an important roll in the area overhead of the system, which is greatly influenced by D_{thr}, the more defective junctions we can

tolerate per memory row, the fewer spare rows the system requires and the smaller the area will be. This effect is visible when comparing the same codes, e.g., $(255, 175)$ for different D_{thr}, 2, 3, and 4. The combination of all of these factors results in the curves shown in figure 6.23.

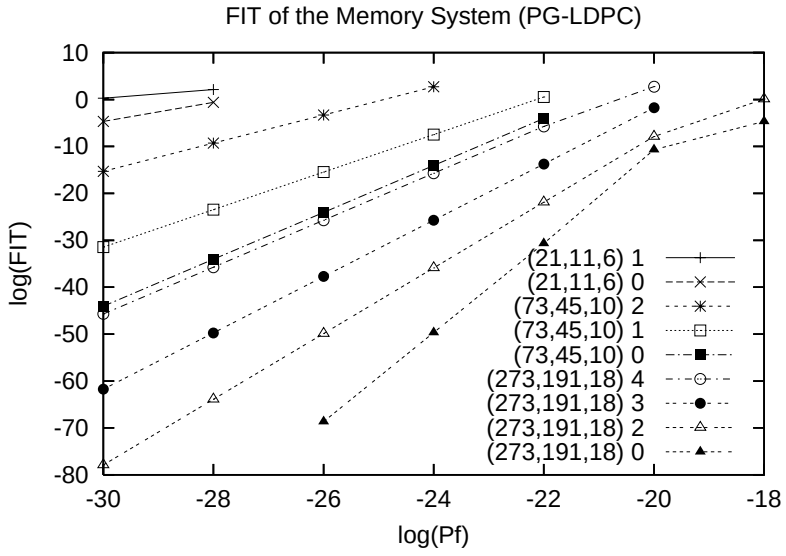
In order to understand how each part of the memory system contribute to the final area overhead and how they change with the cluster size, we show a decomposed area of a memory system of size 1 Mbit, which is essentially one memory bank. First of all let us look at the memory core area per bit. The memory core area per bit is fixed for the code and D_{thr} pair. It does not change with the cluster size or with the total memory size increase. For one code the larger number of defects it can tolerate the smaller the area of the core will be, because it can use more defective wires and requires less wire sparing (Compare rows 3 and 5 of Table 6.5). The maximum number of defective junctions that the smaller codes can tolerate per row is larger than the maximum number of defective junctions of the larger codes. For our system with memory rows of 1000 bits, a (15,7) code that tolerates 1 defect per memory word essentially tolerates 66 errors per row, and a (255,175) that tolerates 4 defective bits, tolerates 12 errors per row. Therefore, generally smaller codes could result in lower area because they can tolerate more defective bits in a row. However, larger codes have better code rates (n/k). The combination of these factors makes the memory core using codes $(63, 37)$ and $(73, 45)$ and tolerating 2 defective junctions, the smallest memory cores, (rows 3 and 4 of Table 6.5).

The second part is the area overhead due to the corrector and detector per bank cluster. The area of the corrector and detector is amortized out over the memory bits of one cluster. Table 6.5 shows the area of the corrector and decoder amortized over 1 Mbit-memory bank (one cluster). For larger clusters these net area per memory bit decrease. For example for cluster of size 1000 memory banks, the per bit area of the corrector and detector is less than 1 nm^2. However, if the cluster size is fixed, the per bit area of these units does not decrease with memory size increase.

The global encoder and its detector, are shared among the whole memory system. Therefore, their net per bit area of these units decrease as the memory size increases.

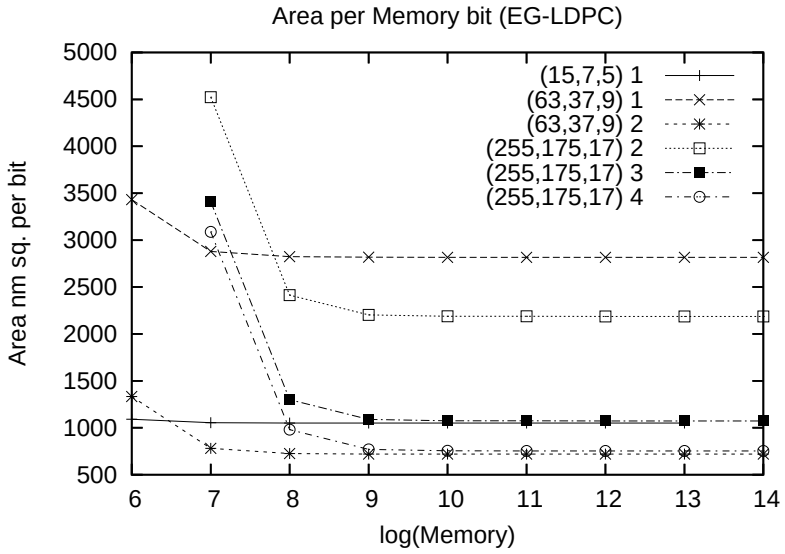


(a)

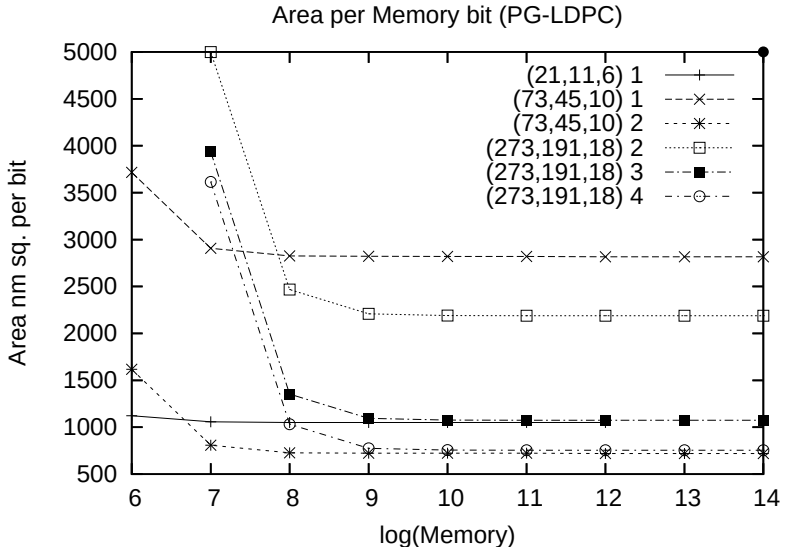


(b)

Figure 6.22. FIT of EG-LDPC codes for a system with 10^{12} memory bit, memory bank size of 1 Gbit, system frequency of 1 GHz, and the defect rate of 1%. The curve labels are of the form: "$(n, k, d)D_{thr}$".



(a)



(b)

Figure 6.23. The area of the memory system vs. the memory size. The fault rate is $P_f = 10^{-28}$, the scrubbing interval and the cluster size is set to values that satisfy FIT$<$ 1000 and throughput loss $<$ 0.001. The curve labels ar of the format: "$(n, k, d)D_{thr}$".

Table 6.4. The decomposed area per bit of the design points selected for the curves at figure 6.22. The memory size for this designs is 10^{12} bit. The unit of area is nm^2/bit.

Column Number	Code Type,(n,k,d)	Original Memory	Final Memory	Global Enc.+Det.	Cor. Units	Det. Units	C	D_{thr}	S (min)	Thr Loss	FIT
1	EG,(15,7,5)	626.0	1050.0	31.5E-6	0.18	0.03	100	1	10	0.001	-1.88
2	PG,(21,11,6)	626.0	1050.0	53.1E-6	0.34	0.05	100	1	10	0.001	-1.88
3	EG,(63,37,9)	523.0	719.0	7.2E-5	0.273	0.02	1000	2	120	0.0008	-9
4	PG,(73,45,10)	523.0	719.0	21.1E-4	0.376	0.02	1000	2	120	0.0008	-9
5	EG,(63,37,9)	2050.0	2817.0	7.2E-5	0.273	0.02	1000	1	120	0.0008	-23
6	PG,(73,45,10)	2050.0	2817.0	21.1E-4	0.376	0.02	1000	1	120	0.0008	-23
7	EG,(255,175,17)	521.0	746.0	8.65E-3	7.36	0.15	1000	4	120	0.0008	-35
8	PG,(273,191,18)	521.0	746.0	1.017E-2	8.96	0.17	1000	4	120	0.0008	-35
9	EG,(255,175,17)	745.0	1066.0	8.65E-3	7.36	0.15	1000	3	120	0.0008	-49
10	PG,(273,191,18)	745.0	1066.0	1.017E-2	8.96	0.17	1000	3	120	0.0008	-49
11	EG,(255,175,17)	1524.0	2180.0	8.65E-3	7.36	0.15	1000	2	120	0.0008	-64
12	PG,(273,191,18)	1524.0	2180.0	1.017E-2	8.96	0.17	1000	2	120	0.0008	-64

Table 6.5. The decomposed area per bit of memory size is 1 Mbit, and cluster size of 1. The unit of area is nm^2/bit.

Column Number	Code Type,(n,k,d)	Final Memory	Global Enc.+Det.	Cor. Units	Det. Units	Total Area OH	C	D_{thr}
1	EG,(15,7,5)	1050.0	5.414	18.19	3.9	1.75	1	1
2	PG,(21,11,6)	1050.0	8.941	34.51	5.8	1.82	1	1
3	EG,(63,37,9)	719.0	73.8	273	21.4	2.64	1	2
4	PG,(73,45,10)	719.0	144.6	376	26.9	3.19	1	2
5	EG,(63,37,9)	2817.0	73.8	273	21.4	1.70	1	1
6	PG,(73,45,10)	2817.0	144.6	376	26.9	1.84	1	1
7	EG,(255,175,17)	746.0	8693	7360	153.0	46.96	1	4
8	PG,(273,191,18)	746.0	1232.2	8960	175.2	57.12	1	4
9	EG,(255,175,17)	1066.0	8693	7360	153.0	33.27	1	3
10	PG,(273,191,18)	1066.0	1232.2	8960	175.2	40.38	1	3
11	EG,(255,175,17)	2180.0	8693	7360	153.0	16.99	1	2
12	PG,(273,191,18)	2180.0	1232.2	8960	175.2	20.47	1	2

For large enough memory (0.1 Gbit<) these units have negligible area overhead per bit compared to other parts of the system.

The overall area overhead per memory bit for large enough memories is dominated mainly by the memory core area which is both the result of defect-tolerant over-population and ECC rate. In the second order, the corrector and detector of each cluster contributes to the area and finally the global encoder. The curves in figure 6.23 plots the total area per bit for a range of memory size, and shows how it decrease as the memory increases and shows its final value.

6.10 Summary

In this chapter we presented a fully fault-tolerant memory system that is capable of tolerating errors not only in the memory bits, but also in the supporting logic including the encoder and corrector. We used Euclidean Geometry and Projective Geometry codes. We proved that these codes are part of a new subset of ECCs that have *Fault-Secure* Detectors. Using these fault-secure detectors we design a fault-tolerant encoder and corrector, where the fault-secure detector monitors their operation. We also presented a unified approach to tolerate permanent defects and transient faults. This unified approach improves the area overhead. If these two techniques were not combined then to tolerate defects, we required reliable (and consequently lithographic scale) encoder and decoder, since we could not tolerate any transient fault in those circuits. Accounting for all the above area overhead factors, all the codes considered here achieve memory density of 20 to 100 Gbit/nm^2, for large enough memory (0.1 Gbits).

Chapter 7

Summary

The conventional defect- and fault-tolerant techniques is developed for low defect and fault rate, where the error happens infrequently in the system (i.e., Single Event Upset). However, if the microelectronic system is going to continue scaling down to the level that the interconnect wires and active devices are only a few atoms wide, then the defect and fault rate is going to be much higher and the system would experience many more than one single error in the system. The conventional reliability techniques do not simply scale to less reliable regime and larger system integration. Thereby, the engineers have to redesign the reliability design paradigm. This process has already started by constantly restricting the Design Rules of VLSI designs from one feature size to the next feature size and it is expected that Design Rules will become more restricted and require more regular design structure to bound the rate of manufacturing defects [2].

In this work we have developed defect- and fault-tolerant techniques that target the reliability challenges in emerging nanoscale technologies and protect systems against permanent defects and transient faults. To design a reliable system with practical area overhead, we have exploited number of conventional design patterns and developed some new patterns:

- *Fine–granularity*: With high fault rate the reliability must be applied to *Fine–Grained blocks* to bound the area overhead require to detect or correct the errors. When the error rate is low the system may experience few errors in the entire system, however, when the error rate is high the system experience

errors in almost every small subsets. So the detection or correction techniques most be applied to Fine–Grained blocks where the errors are not masked and are detectable with reasonable area overhead.

- *Defect pattern matching*: We cannot afford discarding any nonperfect chip, since the yield will be very low. When the defect rate is high we have to make use of any defect free resources. The systems structure and the placement-and-routing techniques must facilitates *Matching* the design on the defect free resources and isolating the defective devices, to make use of the system despite having many defective parts.

- *Using alternative resources*: If only using area to protect systems against defects and faults, the area overhead can be impractically large. In this case we can use other resources besides area to protect the system and bound the area. In this work we exploit time redundancy to contain the area overhead. It is shown in this work that error correction generally takes more area than error detection. So to bound the area overhead, we only implement error detection with extra circuits and then repeat the operation to correct a potential detected error. This approach takes less area compared to the case that the errors are corrected with extra circuits.

- *Global reliability*: When using one unified technique to protect the system, the area overhead provided to protect one resource can be reused to protect other resources as well. One example, is to use the same ECC to protect different part of the system. This will save encoder and decoder time and area overhead to encode/decode data from one ECC to another ECC moving from one part of the system to another part of the system. Another example is using the same ECC to protect against transient faults and permanent defects. With this technique the redundancy in the code will be used more efficiently to protect both errors. We have taken this pattern one step further: We have defined FSD-ECC, a new subclass of ECC, that guarantees enough redundancy in the code that not only protects the codewords but also protects the detector circuitry. Therefore, by

applying ECC to memory we can simply protect the supporting logic as well.

Using the above design patterns, we can tolerate high defect and fault rates with practical area overhead. We have presented a defect tolerant scheme exploiting *Fine-Grained Reliability*, *Alternative Resources*, and *Defect Pattern Matching*, that can tolerate high permanent defect rates with acceptable area overhead. For example, we reported an average of 30% area overhead for tolerating 10% defect rate in NanoPLA programmable junctions.

We have designed a fault-tolerant technique, exploiting *Fine-Grained Reliability* and *Alternative Resources* design patterns that can tolerate up to 10^{-7} transient fault rate. Our fault-tolerant technique can tolerate up to 10^{-9} fault rate with less than 9 times area overhead. This technique can achieve close to an order of magnitude less area overhead compared to the Majority-Multiplexing technique that is the other fault-tolerant technique proposed for nanotechnology designs [11].

With the above two techniques we protect interconnect resources and logic circuits. To perform computation, a system needs computational units (logic circuit), interconnect resources, and memory. In the last part of this dissertation we developed a unified defect- and fault-tolerant techniques to protect the memory, that use single ECC to tolerate both permanent defect and transient faults. This technique is based on the new *Fault-Secure Detector* capable ECCs, FSD-ECC, that guarantees fault-secure detector without any extra logic protection. Using this technique, we reported achieving reliable memory system, that can tolerate up to 10^{-18} fault rate and achieve density of 100 Gbit/cm^2.

The above techniques show that reliability in the nanotechnology systems can be achieved with practical area overhead. Although providing reliability will be more challenging for these technologies compared to conventional microelectronics, it will not be the bottleneck to scaling or using emerging nanotechnology devices.

Chapter 8

Future Work: Using ECC to Protect Logic Circuit

In chapter 5 we presented a multiple-error-detection technique based on replication. We used replication-based error-detection to keep our design general for any logic circuit. In information and coding theory, replication is known for providing suboptimal way of protecting information. However, this is not always true for protecting logic circuit. For single-error detection in logic circuit there have been some works, reporting more compact error-detection compared to replication-based technique. For multiple-error-detection, however, there is no known approach better than replication-and-comparison.

For single-error-detections, Tuba [73] presents a error-detection technique based on parity-prediction that outperforms duplication-and-comparison. In his technique the outputs of the logic circuit is partitioned into one or few parity groups, and each parity group is protected with one parity signal (figure 8.1). If considering the output bits as information bits of an error-correcting code, then the parity groups can be represented as the *Parity-Check* matrix (H matrix) of the error-correcting code. Each row of the H matrix corresponds to one parity group and consequently one parity-signal. The 1's in each row mark the output signals and the parity signal of each parity group (figure 8.1). The number of parity groups can vary from one to the number of outputs. When circuit has one parity signal for each output, the parity

	o1	o1	o3	P1	o4	o5	P2	o6	o7	P3
H=	1	1	1	1	0	0	0	0	0	0
	0	0	0	0	1	1	1	0	0	0
	0	0	0	0	0	0	0	1	1	1

Figure 8.1. The parity signals are represented with P and the output signals are represented with o. This example shows 7 output signals divided into 3 parity groups.

signals make the duplicate of the logic circuit.

The parity signals are generated with parity-prediction functions, concurrently with the original circuit. The works presented at [73] and [70] report that for some of the design points in the MCNC benchmark the area overhead of the parity-protected logic is larger than the area overhead of the duplicated logic. Therefore, for those circuits it is better to use the duplication compared to parity function.

The above works protect the circuit against single error, however, there has not been any analysis for multiple-error-detection for general logic circuits, that can perform better than replication. In this chapter, we show an approach that can potentially outperform replication in multiple-error-detection applications. This technique is based on integrating ECC with the logic circuit using parity-prediction technique. This technique is still under research, and here we show our approach, analysis, and achievements so far.

In the rest of this chapter we show our criteria to select the right error-correcting codes, followed by our random error-correcting code generation procedure. Then we review the challenges to integrate the ECC with the original circuit. Finally we report some of our preliminary results.

8.1 Code Selection Criteria

Our approach is to predict the parity signals of the code concurrently with the original circuit and encode the output bits of the circuit with these parity bits. This is the

same approach used in [73] for single-error-detection. Any systematic ECC can be used to encode the output bits with the above technique. In this section we show the criteria to select an ECC for multiple-error-detection in logic.

Each ECC has different parameters, including *Code Length*, *Minimum Distance*, *Code Rate*, and *Information Bits Length*. For our purpose, which is protecting the logic circuit with ECCs, the information bits length is determined by the number of outputs of the logic circuit. The minimum distance of the code is determined by the maximum number of errors that need to be detected in the circuit. Lastly, the code rate partially determines the area overhead, and therefore, will be minimized after satisfying the code length and minimum distance. Although in coding theory, the code rate essentially shows the overhead of the code, in logic circuit it shows only part of the overhead. The code complexity can dominate the overall area overhead of the ECC-integrated logic circuit, by determining the amount of logic required in the parity signal generation. Based on the complexity of the code and the original logic circuit, the logic implementation of the ECC-integrated circuit may have much larger overhead than the code rate. Another substantial factor in fault-tolerant logic design, is the checker area overhead which grows as the code complexity grows, and is not captured by code rate. So the best ECC is the one that optimizes the minimum distance, code rate, and code complexity.

To satisfy the above conditions, we need control over the above code parameters. Therefore, we use random ECC generation to develop codes with desired properties. We consider random code construction to develop multiple-error-detection technique. We design codes with relatively low complexity, which potentially result in simple and compact ECC-integrated circuits and compact checker. By randomly generating codes, we have control over the code parameters, such as code distance, code rate, and code complexity. So we can find the optimum design points between the code rates and code complexity. For example for a fixed relative distance, lower code rate with lower complexity may result in smaller area overhead than higher code rate with higher complexity.

In order to minimize the area and time overhead, we use *Systematic* codes (sec-

tion 6.4.1). Systematic codes have the advantage that no decoding is necessary for the original output bits. In this article we suggest a random technique to develop *small, low-complexity,* and *systematic* codes that is efficient for our fault-tolerant logic application.

8.2 Random Code Construction

In this section we construct small, low-complexity and systematic codes with random technique, that achieves the desired code complexity, minimum code distance, and code rate.

Each error correcting code is defined by its parity check matrix H. Figure 8.2 shows a 10×4 parity check matrix H of code $(n, k, d, c) = (10, 6, 3, 5)$, where n is the code block size, k is the number of information bits, d is the code minimum distance, and c is the column weight of the parity check matrix H; i.e., the number of ones in each column. Each parity check matrix H, can be represented with a bipartite graph, called *Tanner graph*. The Tanner graph representation of this parity-check matrix is illustrated in figure 8.3.

In an $n \times m$ parity-check matrix H (m is the number of parity bits, $m = n - k$), each row of H is a left vertex of the graph and each column is a right vertex. If $h_{ij} = 1$, then there is an edge between the ith left vertex and the jth right vertex. The right vertices are called the *constraint* or *checker* vertices, and each performs an XOR function over all the incoming edges. A code-word $c = [x_0, x_1, ..., x_{n-1}]$ is associated with an assignment to the left vertices. The corresponding assignment to the right vertices is the result of the product $c \cdot H^T$. Therefore, an assignment to the left vertices is a code-word if the assignment induced on the right vertices is all 0's.

The systematic code property, as in this case, enforces the lower m vertices in the left set to have degree of 1, which represents the identity part of the parity check matrix in the left m columns of the matrix H. The m single-degree vertices on the left side represent the parity check bits and the rest of the vertices on the left side are the information (normal output) bits. There is a one-to-one assignment between

$$\begin{vmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Figure 8.2. Parity-Check matrix of $(10, 6)$ code.

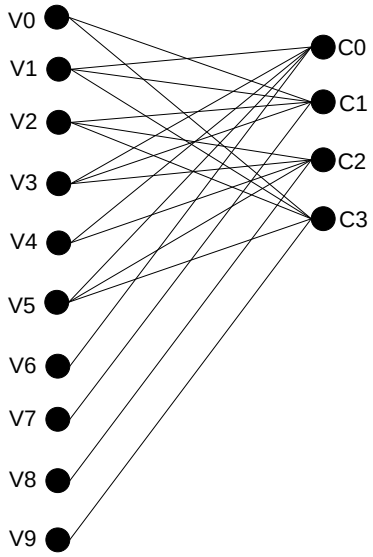


Figure 8.3. Tanner Graph presentation of the parity-check matrix of the code $(10, 6)$ illustrated in figure 8.2.

the parity check bit vertices on the left and the checker vertices on the right. The information bits connecting to a checker vertex are called the *parity group* of the corresponding parity check bit. By construction, the parity check bit is the parity function of its parity group bits.

The low-complexity requirement enforces the checker vertices degree to be small and constant. In other word it means that the parity groups are small and of the same size. In parity check matrix H this means small and fixed number of 1's in each column, which is shown by c in the above code representation, and also shows the checker vertices degree. The limited degree correspond to smaller parity groups and less complicated parity signals corresponding to those groups.

In this section we generate code with the desired specifications (i.e., (n, k, d, c)) from a randomly generated bipartite graph. We start with an empty bipartite graph with n left vertices and $m = n - k$ right vertices. Each of the lower m left vertices

is connected to a separate right vertex, to generate the one-to-one mapping shown in figure 8.3. Then we randomly add edges to the graph, from the top k left vertices to the right vertices following the limitations on each vertex degree. We do so until all the right vertices has degree c, and all the top k left vertices have degree v, such that

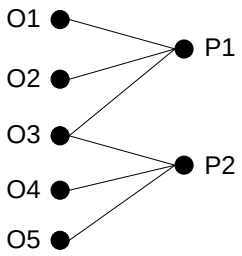
$$\lfloor m \times c/n \rfloor \leq v \leq \lceil m \times c/n \rceil. \tag{8.1}$$

It is important to note that due to the limitation on the vertices degrees, random edge selection will not always generate the desired graph; so in the middle of the graph construction, the graph may be discarded and the construction process restarts. Once the graph with the desired specification is generated it will be examined for the code minimum distance.

In fact we examine our generated codes for the FSD-ECC property, introduced in section 6.4 to protect errors in the checker circuit as well as the main logic circuit. So for each randomly generated graph, we check all the possible error vector of weight $e < d$. The error vectors are assigned to the left vertices of the graph. The code is FSD-ECC with minimum distance d or above if any error vector of weight e has syndrome vector of weight at least $d - e$. It is important to note that errors solely in the parity signals are exempt from this property. If errors in the parity signals are masked and not detected, since the code is systematic, the output bits are still correct, and the masking only prevents a false positive detection.

Once the code with desired rate, distance and complexity is generated, it will be integrated into the circuit logic to protect the normal output bits. The parity check bits are appended to the normal output bits to generate the code.

In the next section we show how to generate the parity check bits in two-level logic concurrently with the normal function of the circuit and how to synthesis logic to optimize area overhead.



(a)

O1 = ab+cd
O2 = bcd
O3 = abc
O4 = bc+cd
O5 = acd

(b)

P1=O1 xor O2 xor O3
P2=O3 xor O4 xor O5

(c)

$P1 = a\,b\,\overline{c} + a\,c\,d + \overline{b}\,c\,d$
$P2 = \overline{a}\,b\,c + \overline{a}\,c\,d + b\,c\,d$

(d)

Figure 8.4. (a) The error-correcting code structure, showing two parity signals for five information bits. (b) The function of the logic circuit. (c) The parity function. (d) The two-level logic implementation of the parity signals.

8.3 Integrating ECC to Logic

8.3.1 Parity Check Bit Generation

To append the parity check bits to the normal output bits, the parity check functions are calculated independantly by the primary inputs of the circuit and are implemented in two-level logic (sum of product) similar to the normal output bits.

figure 8.4 shows the steps to generate the parity functions. Figure 8.4(a) shows the structure of the two parity groups. Figure 8.4(b) shows an example for the outputs of a logic circuit. The parity signals are the XOR of the output signals in their parity groups. Figure 8.4(c) shows how these parity functions are generated. The parity signals are synthesized in two-level logic as represented in figure 8.4(d) generated concurrently from the original inputs.

8.3.2 Logic Synthesis and Area Optimization Challenge

The area of the circuit can usually be reduced substantially from the canonical sum of min-term forms. There are efficient logic minimization tools to reduce the area of the two-level logic circuit, such as espresso [85]. There are two main parts in two-level logic optimization:

1. The min-terms of each output bit can be combined to form a product-term, following the boolean logic rules, to reduce the number of AND functions. This is 2-level single-output logic function minimization.

2. Similar product-terms between different outputs can be shared and replaced by one product-term, to further reduce the number of product-terms. This is called product-term sharing.

However, the area of the concurrent error detecting circuit developed above can only be minimized using the first optimization step. The logic of each outputs or parity signals is optimized separately with no logic sharing between the signals. If logic sharing was allowed, single error in product term can propagate into multiple output and parity signals and may become masked. Assume we are using a code with minimum distance d, which detects $d-1$ errors in the output. In order to make sure that any combination of $d-1$ faults in the logic circuit (including the product-terms) is detected, we have to guarantee that any combination of $d-1$ faults in the logic will result in a detectable error pattern in the output and parity check bits. When there is no logic sharing, $d-1$ errors in the product-terms will propagate into at most $d-1$ erroneous outputs, which can be detected with the code with minimum distance of d. In the presence of product-term logic sharing, $d-1$ errors in the product-terms may propagate into more erroneous outputs. For example when single product-term is shared between two outputs, if the product-term is erroneous, it may cause both of the output signals be erroneous. Therefore, one fault corrupts two output bits. Similarly $d-1$ faults can corrupt more than $d-1$ output bits, and since our code could at most detect $d-1$ errors, then it cannot detect this error pattern, although it

Table 8.1. P-term reduction using logic sharing.

Design Name	5xp1	9sym	alu4	apex1	b12	clip	duke2	ex1010	inc	misex1
Sharing	65	86	575	206	43	120	86	446	30	12
No Sharing	74	86	631	902	53	148	200	452	44	32

was originated from $d-1$ errors. Therefore, no logic (product-terms) can be shared among the outputs.

The restriction on logic-sharing can greatly increase the area overhead. Table 8.1 shows some designs from MCNC benchmark, with both logic sharing and no logic sharing optimization. You can see for a design like APEX1 sharing can reduce the number of P-terms by more than factor of 4.3. Therefore, one of the challenges in fault-tolerant logic is to overcome the area overhead of the no-logic sharing. For replication in contrast, each copy can be optimized exploiting logic sharing, since the error is detected comparing the copies. To tolerate $d-1$ errors as the above example, the system needs d replication copies. With d replication copies any combination of $d-1$ or smaller errors can be detected, and there is no restriction on logic sharing among the outputs of each circuit copy. Since logic sharing is acceptable for replication and is not acceptable for ECC-based technique, achieving an area improvement with the ECC-based techniques compared to replication technique is more challenging.

8.3.3 Output Permutation

The logic complexity of the parity check bits, can vary with the assignments of the physical outputs to the information bits of the ECC. Each mapping determines the output bits belonging to each parity groups, and therefore, impacts the parity function complexity. Here, for each ECC and logic circuit, we map the physical output bits to the information bits with random permutation, and perform the synthesis. We repeated this procedure for large number of permutations, and selected the permutation with minimum area.

Table 8.2 shows a 4-bit ALU protected with 10 bits ECC parity bits with minimum distances of 3, 4, and 5. The reported area is the minimum area of 100 random output permutation. The improvement achieved by changing the mapping of the physical

outputs to the information bits can be very large. In the above simulation the ratio between the maximum and the minimum area of the ECC-integrated circuit can be more than 1.6. Compared to the number of all the possible permutations of output bits, 100 is small number. For example in this case an ALU with 8 outputs, have $8! = 40,320$ permutations. So one potential path for area improvement would be to search this area more efficiently and find closer to optimum output permutation.

8.4 Preliminary Results

We randomly generated ECC with different (n, k, d, c), with the procedure explained in section 8.2, and integrated the codes with the circuits. Table 8.2 illustrates the number of P-terms of a small ALU that is protected with ECCs of minimum distances of 3, 4, and 5. The ALU has 8 outputs, however, the selected codes have 10 information bits. Larger number of information bits gives us the opportunity to reduce the code complexity, by reducing the parity group size.

The table shows the decomposed area of the design in the number of P-terms. The "Org. Logic" column shows the original number of P-terms. The third column shows the number of P-terms in ECC-integrated logic. This is the minimum number, over 100 output permutations. The next column is the number of pterms in the replication case, which is d times the original logic. The following two columns are replication checkers and ECC-based checkers. Note that the replication checker is replicated d times to have the same reliability as the logic (see section 5.1.1 for more details). The ECC checker however, does not need to be replicated because the checker is fault secure as defined in section 6.4. The total ECC-based design (column 9) is smaller than the replication-based design in the first and second row, for which tolerates 2 and 3 errors respectively. However, the difference is very minimal. In the last row which can tolerate 4 errors, the result is reversed, replication-based design is slightly more compact than the ECC-based design. The last column, shows the area of the ECC-integrated circuit and the checker area, when the circuit is synthesized with no restriction on logic sharing. You can see that logic sharing can reduce the area

Table 8.2. Decomposed area of ECC-integrated circuit.

(n, k, d, c)	Org. Logic	ECC Logic	$d\times$ Logic	Rep. Checker	ECC Checker	Rep. Total	ECC Total	Shr.
(20,10,3,4)	575	1686	1725	240	120	1965	1806	1348
(25,10,4,5)	575	2269	2300	480	240	2780	2509	1502
(28,10,5,7)	575	3353	2875	800	432	3675	3785	2395

overhead by up to factor of 1.6. This suggests there is great room for improvement. The question is that "How much of the logic sharing we can exploit? Is it possible to exploit some logic sharing, or is no logic sharing acceptable?"

One idea is to have larger minimum distance code and allow limited logic sharing. For example, let e be the number of errors that a circuit must be able to detect. Then the circuit must use ECC with minimum distance $d = e + 1$ and no logic sharing. Or the circuit can use ECC with minimum distance $d = e \times f + 1$, and allow logic sharing with restriction on P-term fan out to f. This way each error in a P-term can fan out to at most f output. Therefore, a combination of e errors in the circuit can at most corrupt $e \times f$ outputs, and the code can still detect these many errors since $d = e \times f + 1$. We have not performed any simulation and logic synthesis using this approach, it may result some improvement.

8.5 Summary

In this section we presented some of our ongoing research to generate multiple-error-detection for general purpose logic circuit that can outperform the replication with comparison. In our approach we construct random ECC with lightweight parity check bits, and integrate the codes with the logic circuit. The main challenging issue is to integrate the ECC with logic circuit and bound the area overhead, that caused by restriction on logic-sharing. Our preliminary results, show that our ECC-based technique and replication-based technique consumes almost the same amount of area overhead.

Bibliography

[1] "International Technology Roadmap for Semiconductors," <http://www.itrs.net/Links/2005ITRS/Home2005.htm>, 2005.

[2] R. Sinnott, A. Asenov, D. Berry, D. Cumming, S. Furber, C. Millar, A. Murray, S. Pickles, S. Roy, A. Tyrrell, and M. Zwolinksi, "Meeting the Design Challenges of Nano-CMOS Electronics: An Introduction to an Upcoming EPSRC Pilot Project," in *Proceedings of the UK e-Science All Hands Meeting, Nottingham, UK*, September 2006.

[3] B. P. Wong, A. Mittal, Y. Cao, and G. Starr, *Nano-CMOS Circuit and Physical Design*. John Wiley and Sons, Inc., 2005.

[4] Y. Chen, G.-Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams, "Nanoscale Molecular-Switch Crossbar Circuits," *Nanotechnology*, vol. 14, pp. 462–468, 2003.

[5] Y. Huang, X. Duan, Y. Cui, L. Lauhon, K. Kim, and C. M. Lieber, "Logic Gates and Computation from Assembled Nanowire Building Blocks," *Science*, vol. 294, pp. 1313–1317, November 9 2001.

[6] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, "Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes," in *Proceedings of Symposium on VLSI Digest of Technology Papers*, 2001, pp. 73–74.

[7] B. Keeth and R. J. Baker, *DRAM Circuit Design: A Tutorial*, ser. Microelectronic Systems. IEEE Press, 2001.

[8] S. E. Schuster, "Multiple Word/Bit Line Redundancy for Semiconductor Memories," *IEEE Journal of Solid-State Circuits*, vol. 13, no. 5, pp. 698–703, October 1978.

[9] C. Collier, G. Mattersteig, E. Wong, Y. Luo, K. Beverly, J. Sampaio, F. Raymo, J. Stoddart, and J. Heath, "A [2]Catenane-Based Solid State Reconfigurable Switch," *Science*, vol. 289, pp. 1172–1175, 2000.

[10] T. Rueckes, K. Kim, E. Joselevich, G. Y. Tseng, C.-L. Cheung, and C. M. Lieber, "Carbon Nanotube Based Nonvolatile Random Access Memory for Molecular Computing," *Science*, vol. 289, pp. 94–97, 2000.

[11] S. Roy and V. Beiu, "Majority Multiplexing–Economical Redundant Fault-Tolerant Design for Nanoarchitectures," *IEEE Transactions on Nanotechnology*, vol. 4, no. 4, pp. 441–451, 2005.

[12] H. Naeimi and A. DeHon, "Fault Secure Encoder and Decoder for Memory Applications," September 2007, pp. 409–417.

[13] A. DeHon, "Nanowire-Based Programmable Architectures," vol. 1, no. 2, pp. 109–162, 2005.

[14] A. DeHon, P. Lincoln, and J. Savage, "Stochastic Assembly of Sublithographic Nanoscale Interfaces," vol. 2, no. 3, pp. 165–174, 2003.

[15] Y. Huang, X. Duan, Q. Wei, and C. M. Lieber, "Directed Assembly of One-Dimensional Nanostructures into Functional Networks," *Science*, vol. 291, pp. 630–633, January 26 2001.

[16] Y. Chen, D. A. A. Ohlberg, X. Li, D. R. Stewart, R. S. Williams, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, D. L. Olynick, and E. Anderson, "Nanoscale Molecular-Switch Devices Fabricated by Imprint Lithography," *Applied Physics Letters*, vol. 82, no. 10, pp. 1610–1612, 2003.

[17] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson, "On latching probability of particle induced transients incombinational networks," in *International Symposium on Fault-Tolerant Computing*, 1994, pp. 340–349.

[18] J. Patel, P. Hazucha, and T. Karnik, "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, 2004.

[19] J. R. Lamarsh and A. J. Baratta, *Introduction to Nuclear Engineering.* Prentice Hall, 1999.

[20] P. J. Griffin, T. F. Luera, F. W. Sexton, P. J. Cooper, S. G. Karr, G. L. Hash, and E. Fuller, "The role of thermal and fission neutrons in reactor neutron-inducedupsets in commercial SRAMs," *IEEE Transactions on Nuclear Science*, vol. 44, no. 6, pp. 2079–2086, 1997.

[21] R. C. Baumann and E. B. Smith, "Neutron-induced boron fission as a major source of soft errors indeep submicron SRAM devices," in *IEEE International Reliability Physics Symposium*, 2000, pp. 152–1257.

[22] J. F. Ziegler, "Terrestrial cosmic rays and soft errors," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 19–39, 1996.

[23] J. Kim and L. Kish, "Error Rate In Current-Controlled Logic Processors With Shot Noise," *Fluctuation and Noise Letters*, vol. 4, no. 1, pp. 83–86, 2004.

[24] S. O. Rice, "Mathematical Analysis of Random Noise," *Bell System Technical Journal*, vol. 24, pp. 46–156, 1945.

[25] R. E. Lyons and W. Vandekulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM Journalof Research Development*, vol. 6, no. 2, p. 200, 1962.

[26] N. S. Bowen and D. K. Pradham, "Processor- and memory-based checkpoint and rollback recovery," *Computer*, vol. 26, pp. 22–31, February 1993.

[27] G.-M. Chiu and C.-R. Young, "Efficient rollback-recovery technique in distributed computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 565–577, June 1996.

[28] D. K. Pradhan and N. H. Vaidya, "Roll-forward and rollback recovery: performance-reliability trade-off," *IEEE Transactions on Computers*, vol. 46, pp. 372–378, March 1997.

[29] J. V. Neumann, "Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956.

[30] A. Sadek, K. Nikolić, and M. Forshaw, "Parallel information and computation with restitution for noise-tolerant nanoscale logic networs," *Nanotechnology*, vol. 15, pp. 192–210, 2003.

[31] S. Roy and V. Beiu, "Majority Multiplexing–Economical Redundant Fault-Tolerant Design for Nanoarchitectures," *IEEE Transaction on Nanotechnology*, vol. 4, pp. 441–451, 2004.

[32] K. Nikolić, A. Sadek, and M. Forshaw, "Fault-tolerant techinques for nanocomputers," *Nanotechnology*, vol. 13, pp. 357–362, 2002.

[33] A. DeHon, "Design of Programmable Interconnect for Sublithographic Programmable Logic Arrays," February 2005, pp. 127–137.

[34] Y. Cui, L. J. Lauhon, M. S. Gudiksen, J. Wang, and C. M. Lieber, "Diameter-Controlled Synthesis of Single Crystal Silicon Nanowires," *Applied Physics Letters*, vol. 78, no. 15, pp. 2214–2216, 2001.

[35] H. D. A. Javey, "Regular Arrays of 2 nm Metal Nanoparticles for Deterministic Synthesis of Nanomaterials," *Journal of the American Chemical Society*, vol. 127, pp. 11 942–11 943, 2005.

[36] A. M. Morales and C. M. Lieber, "A Laser Ablation Method for Synthesis of Crystalline Semiconductor Nanowires," *Science*, vol. 279, pp. 208–211, 1998.

[37] Y. Wu and P. Yang, "Germanium Nanowire Growth via Simple Vapor Transport," *Chemistry of Materials*, vol. 12, pp. 605–607, 2000.

[38] Y. Tan, X. Dai, Y. Li, and D. Zhu, "Preparation of Gold, Platinum, Palladium and Silver Nanoparticles by the Reduction of their Salts with a Weak Reductant–Potassium Bitartrate," *Journal of Material Chemistry*, vol. 13, pp. 1069–1075, 2003.

[39] D. Whang, S. Jin, and C. M. Lieber, "Nanolithography Using Hierarchically Assembled Nanowire Masks," *Nanoletters*, vol. 3, no. 7, pp. 951–954, July 9 2003.

[40] S. Y. Chou, P. R. Krauss, and P. J. Renstrom, "Imprint Lithography with 25-Nanometer Resolution," *Science*, vol. 272, pp. 85–87, 1996.

[41] N. A. Melosh, A. Boukai, F. Diana, B. Gerardot, A. Badolato, P. M. Petroff, and J. R. Heath, "Ultrahigh-Density Nanowire Lattices and Circuits," *Science*, vol. 300, pp. 112–115, April 4 2003.

[42] C. Dekker, "Carbon Nanotubes as Molecular Quantum Wires," *Physics Today*, pp. 22–28, May 1999.

[43] Y. Cui, Z. Zhong, D. Wang, W. U. Wang, and C. M. Lieber, "High Performance Silicon Nanowire Field Effect Transistors," *Nanoletters*, vol. 3, no. 2, pp. 149–152, 2003.

[44] A. DeHon, "Array-Based Architecture for FET-based, Nanoscale Electronics," *IEEE Journal of Nanotechnology*, vol. 2, no. 1, pp. 23–32, March 2003.

[45] C. L. Brown, U. Jonas, J. A. Preece, H. Ringsdorf, M. Seitz, and J. F. Stoddart, "Introduction of [2]Catenanes into Langmuir Films and Langmuir-Blodgett Multilayers. A Possible Strategy for Molecular Information Storage Materials," *Langmuir*, vol. 16, no. 4, pp. 1924–1930, 2000.

[46] B. Gojman, E. Rachlin, and J. E. Savage, "Decoding of Stochastically Assembled Nanoarrays," in *Proceedings of the 2004 International Symposium on VLSI*, February 2004.

[47] A. DeHon, "Law of Large Numbers System Design," in *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, S. K. Shukla and R. I. Bahar, Eds. Boston: Kluwer Academic Publishers, 2004, ch. 7, pp. 213–241.

[48] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, 02061 USA: Kluwer Academic Publishers, 1999.

[49] A. DeHon, S. C. Goldstein, P. J. Kuekes, and P. Lincoln, "Non-Photolithographic Nanoscale Memory Density Prospects," *IEEE Journal of Nanotechnology*, vol. 4, no. 2, pp. 215–228, 2005.

[50] A. DeHon, "Deterministic Addressing of Nanoscale Devices Assembled at Sublithographic Pitches," *IEEE Journal of Nanotechnology*, vol. 4, no. 6, pp. 681–687, 2005.

[51] A. DeHon and H. Naeimi, "Deterministic addressing of nanoscale devices assembled at sublithographic pitches," US Patent Application 7242601, 2007.

[52] J. R. Heath, P. J. Kuekes, G. S. Snider, and R. S. Williams, "A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology," *Science*, vol. 280, no. 5370, pp. 1716–1721, June 12 1998.

[53] Y. Luo, P. Collier, J. O. Jeppesen, K. A. Nielsen, E. Delonno, G. Ho, J. Perkins, H.-R. Tseng, T. Yamamoto, J. F. Stoddart, and J. R. Heath, "Two-Dimensional Molecular Electronics Circuits," *ChemPhysChem*, vol. 3, no. 6, pp. 519–525, 2002.

[54] S. Williams and P. Kuekes, "Demultiplexer for a Molecular Wire Crossbar Network," United States Patent Number: 6,256,767, July 3 2001.

[55] S. C. Goldstein and M. Budiu, "NanoFabrics: Spatial Computing Using Molecular Electronics," in *Proceedings of the International Symposium on Computer Architecture*, June 2001, pp. 178–189.

[56] S. C. Goldstein and D. Rosewater, "Digital Logic Using Molecular Electronics," in *ISSCC Digest of Technical Papers*. IEEE, February 2002, pp. 204–205.

[57] G. Snider, P. Kuekes, and R. S. Williams, "CMOS-like Logic in Defective, Nanoscale Crossbars," *Nanotechnology*, vol. 15, pp. 881–891, June 2004.

[58] D. B. Strukov and K. K. Likharev, "CMOL FPGA: a Reconfigurable Architecture for Hybrid Digital Circuits with Two-Terminal Nanodevices," *Nanotechnology*, vol. 16, no. 6, pp. 888–900, June 2005.

[59] J. Huang, M. Tahoori, and F. Lombardi, "Routability and Fault Tolerance of FPGA Interconnect Architectures," in *Proceedings of the International Test Conference*, 2004.

[60] A. Yu and G. Lemieux, "Defect-Tolerant FPGA Switch Block and Connection Block with Fine-Grain Redundancy for Yield Enhancement," in *International Conference on Field-Programmable Logic and Applications, Tampere, Finland*, 2005, pp. 255–262.

[61] D. Mark and J. Fan, "Localizing open interconnect defects using targeted routing in FPGAs," in *Proceedings of International Test Conference*, 2004, pp. 627–634.

[62] W.-J. Huang, S. Mitra, and E. J. McCluskey, "Fast Run-Time Fault Location in Dependable FPGA-Based Applications," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE Computer Society, 2001, p. 206.

[63] W.-J. Huang and E. J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA," *FCCM*, pp. 137–146, 2001.

[64] A. DeHon and H. Naeimi, "Seven Strategies for Tolerating Highly Defective Fabrication," vol. 22, no. 4, pp. 306–315, July–August 2005.

[65] H. Naeimi and A. DeHon, "A Greedy Algorithm for Tolerating Defective Crosspoints in NanoPLA Design," in *ICFPT*. IEEE, December 2004, pp. 49–56.

[66] G. Krishnan, "Flexibility with EasyPath FPGAs," *Xcell Journal*, vol. 0, no. 4, pp. 96–98, 2005.

[67] D. Chen, J. Cong, M. Ercegovac, and Z. Huang, "Performance-Driven Mapping for CPLD Architectures," *TRCAD*, vol. 22, no. 10, pp. 1424–1431, October 2003.

[68] V. Betz and J. Rose, "FPGA Place-and-Route Challenge," <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>, 1999.

[69] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 1998, pp. 105–115.

[70] S. Mitra and E. J. McCluskey, "Which Concurrent Error Detection Scheme to Choose?" in *Proceedings of the International Test Conference*, 2000, pp. 985–994.

[71] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings of International Conference on Dependable Systems and Networks*, June 2002, pp. 389–398.

[72] B. S. Landman and R. L. Russo, "On Pin Versus Block Relationship for Partitions of Logic Circuits," *IEEE Transactions on Computers*, vol. 20, pp. 1469–1479, 1971.

[73] N. A. Tuba and E. J. McCluskey, "Logic Synthesis of Multilevel Circuits with Concurrent Error Detection," *IEEE Transaction On Computer-Aided Design of Integrated Circuit and Systems*, vol. 16, no. 7, 1997.

[74] H. Naeimi and A. DeHon, "Fault Tolerant Nano-Memory with Fault Secure Encoder and Decoder," in *International Conference on Nano-Networks*, September 2007.

[75] S. J. Piestrak, A. Dandache, and F. Monteiro, "Designing fault-secure parallel encoders for systematic linear error correcting codes," *IEEE Transactions on Reliability*, vol. 52, no. 4, pp. 492–500, 2003.

[76] G. C. Cardarilli, "Concurrent Error Detection in Reed-Solomon Encoders and Decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, pp. 842–826, 2007.

[77] A. Saleh, J. Serrano, and J. Patel, "Reliability of Scrubbing Recovery-Techniques for Memory Systems," *IEEE Transaction on Reliability*, vol. 39, no. 1, pp. 114–122, 1996.

[78] R. J. McEliece, *The Theory of Information and Coding.* Cambridge University Press, 2002.

[79] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Prentice Hall, 2004.

[80] H. Tang, J. Xu, S. Lin, and K. A. S. Abdel-Ghaffar, "Codes on Finite Geometries," *IEEE Transaction on Information Theory*, vol. 51, no. 2, pp. 572–596, 2005.

[81] Y. Kou, S. Lin, and M. P. C. Fossorier, "Low-Density Parity-Check Codes Based on Finite Geometries: A Rediscovery and New Results," *IEEE Transaction on Information Theory*, vol. 47, no. 7, pp. 2711–2736, 2001.

[82] A. DeHon and M. J. Wilson, "Nanowire-Based Sublithographic Programmable Logic Arrays," February 2004, pp. 123–132.

[83] D. E. Knuth, *The Art of Computer Programming*, 2nd ed. Addison Wesley, 2000.

[84] Y. Cui, X. Duan, J. Hu, and C. M. Lieber, "Doping and Electrical Transport in Silicon Nanowires," *Journal of Physical Chemistry B*, vol. 104, no. 22, pp. 5213–5216, June 8 2000.

[85] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Norwell, MA: Kluwer Academic Publishers, 1984.