Hierarchical Strategies for Fault-Tolerance in Reconfigurable Architectures

David Michael Renwick Lawson

PhD

University of York
Electronics

September 2015

Abstract

This thesis presents a novel hierarchical fault-tolerance methodology for fault recovery in reconfigurable devices.

As the semiconductor industry moves to producing ever smaller transistors, the number of faults occurring increases. At current technology nodes, unavoidable variations in production cause transistor devices to perform outside of ideal ranges. This variability manifests as faults at higher levels and has a knock-on effect for yields. In some ways, fault tolerance has never been more important.

To better explore the area of variability, a novel reconfigurable architecture was designed: Programmable Analogue and Digital Array (PAnDA). By allowing reconfiguration from the transistor level to the logic block level, PAnDA allows for design space exploration, previously only available through simulation, in hardware. The main advantage of this is that design modifications can be tested almost instantaneously, as opposed to running time consuming transistor-level simulations.

As a result of this design, each level of PAnDA's configuration contains structural homogeneity, allowing multiple implementations of the same circuit on the same hardware. This potentially creates opportunities for fault tolerance through reconfiguration, and so experimental work is performed to discover how best to utilise these properties of PAnDA.

The findings show that it is possible to optimise the reconfiguration in the event of a fault, even if the nature and location of the fault are unknown.

Contents

List of Tables

List of Figures

Acknowledgements

This thesis would not have been possible without the tremendous support and encouragement from Andy Tyrrell, Martin Trefzer, James Walker and Simon Bale. I would like to thank everyone on the PAnDA project for making it such an enjoyable and rewarding experience.

I also owe thanks to my parents, my partner's parents and both of our families.

Lastly, words cannot express the gratitude I owe to Catherine for her love, support and the cups of tea she provided throughout the course of this work. I will be forever grateful.

Declaration

This thesis is submitted for the degree of Doctor of Philosophy and has not been submitted for any other award at this or any other institution. All work presented is entirely my own work, except where explicitly referenced.

Parts of this work have been presented in the following conference and journal papers:

- Campos, Pedro B.; Lawson, David M.R.; Bale, Simon J.; Walker, James Alfred; Trefzer, Martin A.; Tyrrell, Andy M., "Overcoming faults using evolution on the PAnDA architecture," in *Evolutionary Computation (CEC), 2013 IEEE Congress on* , vol., no., pp.613-620, 20-23 June 2013

- Lawson, D.M.R.; Walker, J.A.; Trefzer, M.A.; Bale, S.J.; Tyrrell, A.M., "A hierarchical fault tolerant system on the PAnDA device with low disruption," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on* , vol., no., pp.69-76, 14-17 July 2014

- Lawson, D.M.R.; Walker, J.A.; Trefzer, M.A.; Bale, S.J.; Tyrrell, A.M., "Evolving hierarchical low disruption fault tolerance strategies for a novel programmable device," in *Evolvable Systems (ICES), 2014 IEEE International Conference on* , vol., no., pp.77-84, 9-12 Dec. 2014

- Trefzer, M.A.; Lawson, D.M.R.; Bale, S.J.; Walker, J.A.; Tyrrell, A.M., "Hierarchical Strategies for Efficient Fault Recovery on the Reconfigurable PAnDA Device," in *IEEE Transactions on Computers* , vol. PP, no. 99, pp. 1–1, 2016.

Chapter 1

Introduction

Contents

1.1 Introduction

Electronic devices have become deeply embedded within our society, fulfilling a range of roles from entertainment to life support. The demands placed on the integrated circuits within these devices have increased as time has gone on and so a constant pressure for these devices to be smaller, faster and more efficient continues to push knowledge, technology and manufacturing techniques to their limits.

For the past few decades these demands have been met through making transistors smaller as the fabrication technologies improve to facilitate this, mostly adhering to Moore's Law [1]. By scaling the size of transistors more can be included on the same size chip, and chips can operate at a lower voltage. This allows designers to increase performance by adding features such as dedicated processing units and higher operating frequencies, both leading to performance increases. This strategy is becoming increasingly difficult to continue, however, as transistors in modern chips are now produced at the atomic scale, which brings about a number of challenges. It is not possible to consistently manufacture millions of transistors of this size and so there is variability in their construction and consequently their performance and reliability [2–4]. In addition, physical effects such as electromigration are more likely to cause issues in atomic-scale devices due to narrower interconnections and proportionally higher voltages compared with earlier devices. In general more faults can be expected to occur in both manufacture and operation when circuits are built at deep sub-micron scales, i.e. below 100 nm feature size, causing poor yields and potentially shorter lifespans in the field [5].

The decrease in integrated circuit reliability leads to a need for new methodologies to increase fault tolerance. Well established techniques such as Triple Modular Redundancy (TMR) [6, 7] are trusted to work well when reliability is critical, but the cost of triplicating an implementation for the functionality of one is not acceptable in many applications and destroys any benefits in cost or size achieved through transistor scaling. There is therefore a growing need for fault tolerance methodologies that improve upon TMR's resource utilisation.

Reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) are a common platform for experimental circuits and fault tolerance methodologies [8–13]. In the event of a fault affecting part of a configured circuit it is sometimes possible to configure an FPGA differently, avoiding faulty components and thus regaining functionality. Methodologies exploiting reconfigurable architectures have the potential to reduce the required power and resources compared to TMR since only one copy of the circuit needs to be active in any particular instant.

This thesis investigates how best to exploit such methodologies in the context of a novel reconfigurable architecture, the Programmable Analogue and Digital Array (PAnDA). In particular, this work focuses on mitigating transistor faults in transistor-level circuits, assuming that any reconfiguration logic is fault free. The justification for this is that prior work has addressed faults in FPGA configuration memory [14–16] and the observation that errors in the configuration memory of a transistor level reconfigurable architecture will manifest equivalently to faults in the transistors themselves.

Given that the problem of fault detection has been addressed many times in prior work [17–20], this research assumes that this is already in-place and focuses purely on the fault recovery side of fault tolerance. Since the faults being explored are at the transistor level, the work seeks a methodology which minimises the changes required to regain functionality to avoid having to initiate a new place-and-route sequence which is often a costly operation in terms of time and power.

1.2 PAnDA Project

PAnDA was an EPSRC funded project (EP/I005838/1) involving the Intelligent Systems Group (ISG) at the University of York and the Device Modelling Group at the University of Glasgow. It started in October 2010 with the aim of designing and using a hardware platform to investigate the effects of transistor variability and the effectiveness of techniques to overcome it. While much research already exists on this topic, much of it is carried out in computer simulation due to the high cost of fabrication. The PAnDA architecture allows research of this nature to move in to the hardware realm and thus benefit from significantly quicker evaluation. This is made possible by PAnDA's special reconfiguration architecture which enables circuits to be configured at both the digital level (as with standard FPGAs) and the analogue level (such as configuring transistor width). More details can be found in chapters 4 & 6.

This research aims to exploit the platform for the purpose of finding a novel fault tolerance methodology, where faults could be attributable to either variability or other sources. The features introduced in PAnDA for variability exploration made it a good candidate of this type of research given that the platform combines the features of a traditional FPGA with additional layers of configuration below.

1.3 Hypothesis

This thesis aims to prove the following hypothesis:

Hypothesis 1 *It is possible to include reconfiguration mechanisms exploiting both the analogue and digital abstraction layers of the PAnDA architecture in order to provide fault tolerance in devices.*

During the course of the research that was carried out around Hypothesis 1, a number of sub-hypotheses were investigated:

Hypothesis 2 *Removing the function decoder block from the PAnDA-Eins architecture provides benefits for fault tolerance.*

Hypothesis 2 explored whether making the initial PAnDA architecture more configurable would provide benefits for fault tolerance by increasing the number of options for implementing some logic functions (Chapter 5).

Hypothesis 3 *Applying transformations to a circuit configuration at random allows the identification of working circuits on a faulty substrate.*

Hypothesis 3 investigated whether the random application of a set of circuit reconfigurations could find a working circuit on a faulty PAnDA chip. This unguided approach to fault tolerance requires no knowledge about the current configuration or the nature of any faults present which means it could be applied in any scenario (Chapter 8).

Hypothesis 4 *Applying circuit transformations deterministically allows the identification of working circuits on a faulty substrate more efficiently than a random application.*

Hypothesis 4 takes the experiments conducted during the investigation of Hypothesis 3 further by using knowledge of the current circuit configuration to try to find a deterministic sequence of circuit reconfigurations can find a working circuit more efficiently than a random one (Chapter 9).

1.4 Contributions

The work in this thesis offers these main contributions:

- A novel hierarchical fault tolerance methodology using random sequences of non-destructive circuit transformations (Chapter 8).
- A novel hierarchical fault tolerance methodology using evolved lists of non-destructive circuit transformations (Chapter 9).

The advantage of these novel fault-mitigation techniques is that they are capable of finding functioning circuit configurations on a faulty reconfigurable device while minimising the disruption to the overall circuit structure. This means that during fault recovery, the methodologies will attempt to keep the circuit's configuration as close to its original as possible by seeking small changes to the implementation rather that resorting to another place-and-route (an expensive operation which maps a design to hardware).

One of the main difficulties of the approach is the testing of candidate circuits. If a conducting fault occurs it is possible that a short circuit situation will occur where `VDD` connects to `GND` through a series of transistors. This possibility does not necessarily cause an issue for two reasons:

1. Current will always flow through a number of transistors, which by providing some resistance will reduce the overall effect of a short circuit.
2. A fault may cause a short-circuit situation itself, meaning that if it hasn't broken already the fault mitigation process should not do any additional damage.

The novelty of these fault-mitigation techniques is in the cataloguing and application of non-destructive circuit transformations for the purposes of fault tolerance. The author is not aware of a similar technique in the literature being applied to reconfigurable transistor circuits.

1.5 Thesis Structure

Chapter 2 introduces the subject of transistor scaling and the problems that come with making transistors smaller. This leads into a discussion of fault tolerance and related techniques in the literature. This concludes by drawing together the concepts in the chapter and suggesting the need for a new hardware platform which could benefit work in these areas.

Chapter 3 presents Evolutionary Algorithms (EAs) and their use for exploring design spaces. Some examples from the literature are discussed which demonstrate how EAs are useful for finding novel solutions to problems.

Chapter 4 introduces the PAnDA-Eins architecture as a hardware platform tool for use in exploring the issues from Chapter 2.

Chapter 5 presents experimental work that uses an evolutionary algorithm to exploit the novel features of PAnDA-Eins to fix transistor level circuit faults.

Chapter 6 introduces a new iteration of the PAnDA architecture, PAnDA-Zwei. The differences between PAnDA-Eins and PAnDA-Zwei are explained along with the new features which provide more flexibility.

Chapter 7 discusses a software model of PAnDA-Zwei which gives the ability to simulate some functionality of the architecture quicker than using a full analogue simulation.

Chapter 8 presents an initial investigation into how the novel features of PAnDA-Zwei can be exploited to provide a new kind of resourceful fault tolerance using strategies at the transistor level.

Chapter 9 extends the work in Chapter 8 with a modification to the strategies and provides a comparison of the two methods.

Chapter 10 summarises the findings and contributions in this thesis and provides some concluding remarks. It continues with a discussion of possible future work, including specifically that of taking the work described in Chapters 8 and 9 further by adding higher level strategies to the existing systems.

Chapter 2

Transistor Scaling and Fault Tolerance

Contents

2.1 Introduction

Moore's famous observation in 1975 [21] stated that the transistor count in affordable integrated electronic circuits doubles every two years. This observation is often said to be a target for the semiconductor industry, driving progress at the same time as limiting the pace of advancements. To achieve this target, transistors are scaled down so that more will fit in the

In terms of digital processors, more transistors enable more features to be built in, such as dedicated multiplier units for a processor to use instead of using repeated addition for instance, and the capability for those features to efficiently handle larger numbers of bits at a time. In general, this facilitates better performance and is desirable.

Unfortunately we are rapidly approaching the physical limits of silicon and along the way there are many issues coming to light which hinder the development of performance increases as seen in the past. Many of these issues are related to reliability and are making circuit design significantly more challenging and thus expensive.

This chapter provides an introduction to transistor scaling and why it increases the probability of experiencing faults. It then discusses some approaches to fault tolerance found in the literature. The background described in this chapter is of particular importance for the work reported in this thesis as it provides evidence and context as to why novel fault tolerant methods are required in state-of-the-art semiconductor designs, from devices to systems. Given that the motivation of this work is to find an online fault recovery mechanism, the concept of testing to produce a fault map is outside the scope of the research and so will not be covered.

2.2 Transistor Scaling and Variability

Space on a chip's die is at a premium. The cost of fabrication is high and so as many chips as possible must be produced from a silicon wafer in order for it to be economical. This means transistors in an integrated circuit must be as tightly packed as possible and no space must be wasted. In order to increase the number of transistors on a chip therefore, the die must either be made bigger or the transistors made smaller. The latter approach is the focus for most semiconductor developments for many reasons, including:

- A larger die per chip means that fewer chips are manufactured per wafer, increasing unit cost.

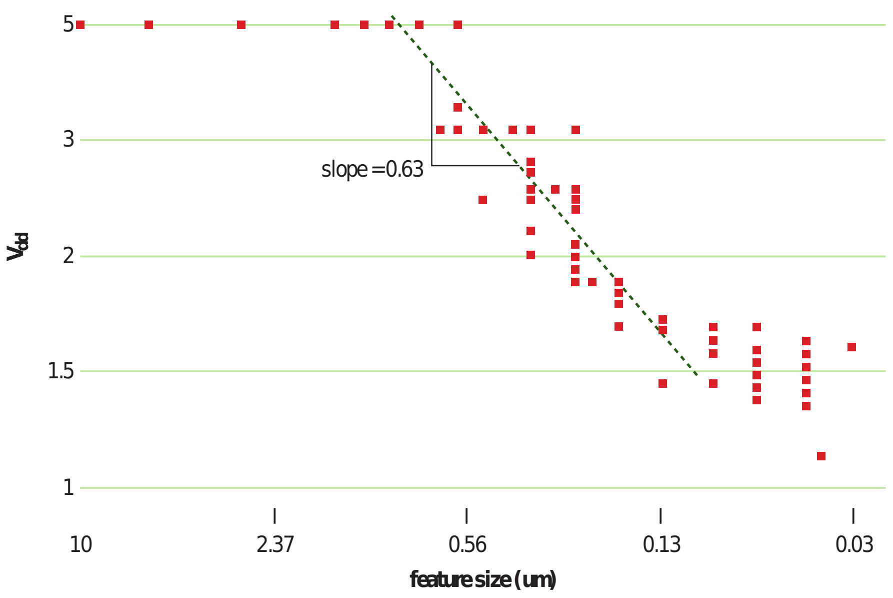


Figure 2.1: A graph showing how core voltages have decreased with feature size (adapted from [22]).

- Silicon wafers are not perfect, the probability of a die containing a defect increases with size, decreasing yield and increasing cost.
- The speed at which modern chips operate means that the physical distances that signals must travel begins to become relevant, thus a smaller chip is able to operate faster.

Putting more transistors into the same area increases power density if all other variables are kept the same. For this reason, voltages were historically reduced proportional to the size which allowed power density to remain constant (see Figure 2.1), a trend known as Dennard scaling [23]. This enabled a steady increase in both transistor density and an increase in clock frequency, producing exponential increases in performance (Figure 2.2). More recently however, Dennard scaling has broken down [24], meaning that voltages can no longer be reduced to the same degree if at all. This has necessitated an increase in dark silicon, chips which cannot be fully powered up due to a power budget, or else risk thermal destruction of the chip.

In addition to the breakdown of Dennard scaling, the scale at which transistors are now fabricated means that they can no longer be relied upon to always function within specification. Small variations in their physical structure, which were always present but small relative to the device size, can now produce significant deviations from their design characteristics. This means that circuit design now takes longer and thus costs more because of the need to ensure that circuits can cope with at least some degree of variability.

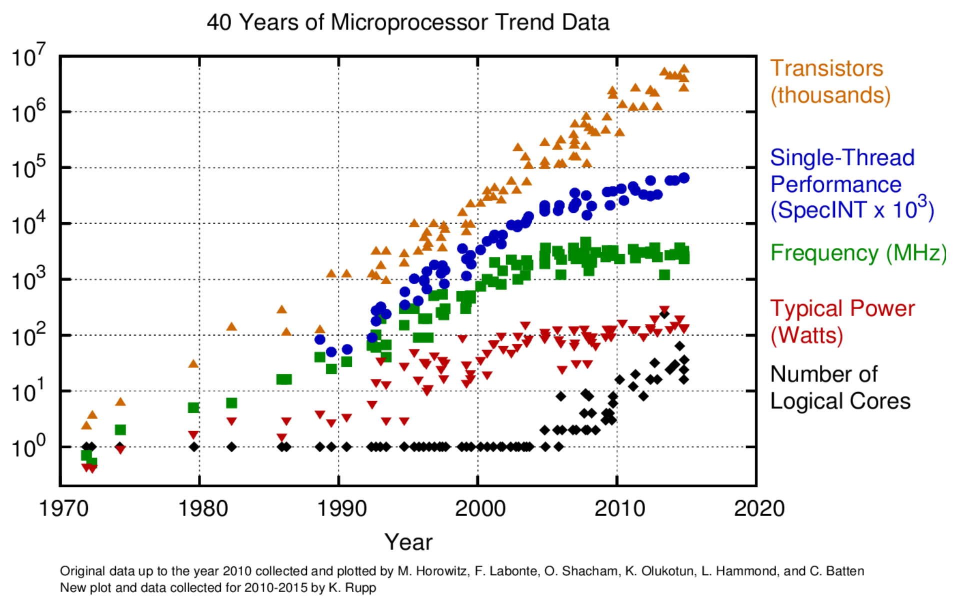


Figure 2.2: A graph showing how transistor count has increased exponentially since 1970, whereas power consumption has stayed approximately stable since around 2002 (taken from [25]).

2.2.1 Variability

For the past few decades, transistor scaling has provided manufacturers with a method of consistently improving performance each year. The progress in the last 10 years however has been made more difficult due to reaching the physical limits of what is possible with silicon based transistors. Firstly, the breakdown of Dennard scaling has meant that voltages cannot be reduced with each transistor size reduction meaning that, in order to keep within thermal budgets, it has not been possible to increase clock speeds as before. More recently, variability in manufactured transistors has meant that designers must take more care when designing circuits to take into account the fact that they will not all behave the same when realised.

This variability is due to the extremely small size of modern transistors, 10's of nanometres, and thus the inability of fabrication machinery to accurately produce them consistently in large quantities.

Sources

There are a number of sources of performance variability when making transistors smaller which are discussed below. The descriptions are based on [26] and [27].

Random Dopant Fluctuation When producing transistors larger than around 90nm, the small, unavoidable variations in the absolute number and placement of doping atoms in a

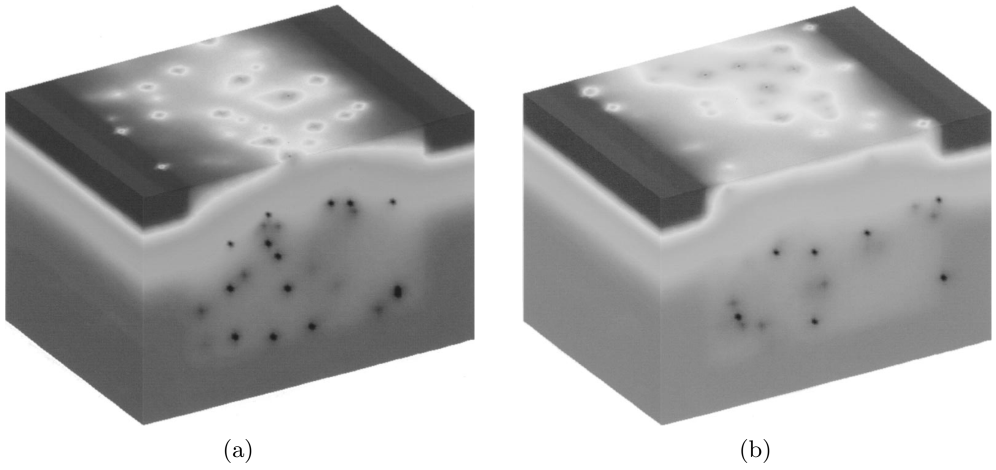


Figure 2.3: Taken from [2]. Two MOSFET transisors with the same number of dopant atoms (170) but different distributions. This leads to different threshold voltages of (a) 0.78 V (b) 0.56 V.

semiconductor are relatively insignificant. As transistors get smaller however, and given our inability to control the placement of individual atoms, these variations, though equal in magnitude, become more statistically significant and can have severe detrimental effects on the characteristics of these devices (Figure 2.3). Some areas will be more highly doped than others and the difference in performance between two identically designed transistors may be significant.

Poly-silicon/Metal Granularity MOSFET transistor gates were commonly made of poly-silicon until recently, and now metal gates are becoming more popular. Both materials are naturally granular and the size and orientation of the grains affect the electrical characteristics of the gate. With large transistors, variations in the grain structure average out, but at modern geometries the differences have a noticeable effect on the transistor's characteristics.

Line Edge Roughness The outline of a transistor during design will usually have straight edges, commonly a rectangle of doped semiconductor bridging two contacts. For any size transistor, zooming in on the edges of this rectangle will reveal that the edges are not straight at all and in fact are rough, due to manufacturing limits. Though the roughness is small, the average and minimum length of the gate will be affected. As transistors shrink, this roughness stays the same and so the effect increases in significance.

Surface Roughness Variations in the vertical plane of MOSFET transistors, such as the thickness of the oxide layer, begin to present more significantly than before. These can contribute to variations in V_T [26].

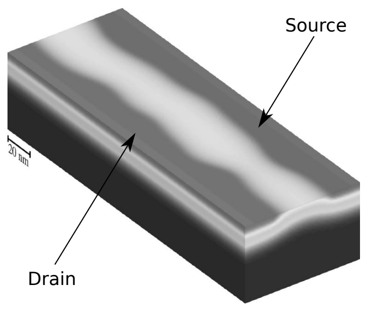


Figure 2.4: Example of a MOSFET exhibiting line edge roughness (taken from [3]). Note that the distance between source and drain (gate length) varies as the edge wobbles.

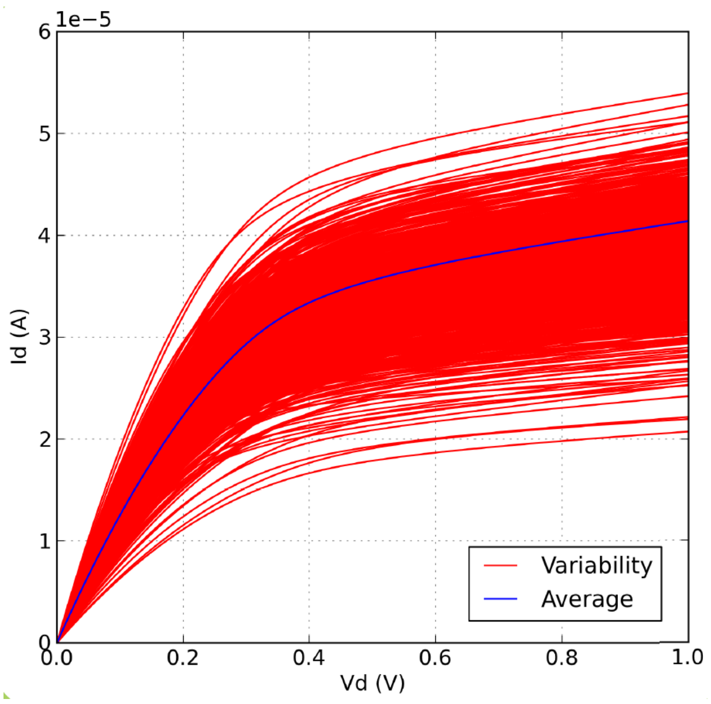


Figure 2.5: A graph showing how transistor variability can affect the Current/Voltage (IV) curve of a transistor (taken from [28]).

These sources all contribute to the overall unpredictability of nanometre scale transistor performance. Figure 2.5 illustrates this from the perspective of drain current versus gate voltage and shows that in 80nm FinFET transistors the current can vary up to 140mA/mm, where the unit of measurement represents the width of the transistor.

2.2.2 Coping Methods

Clearly a designer's job is difficult when the components they are using cannot be relied upon to perform as specified. In complex circuits, variability in some transistors will have more of an affect on performance and reliability than others. One way of improving the probability that a transistor will perform as specified is to simply make it bigger. Commonly this is achieved by increasing the transistor's width. Relating this to the effects described above, physical variations will average out more with a larger transistor, at the cost of power and space. Making all the transistors larger will remove any benefit of scaling and so careful consideration is required to determine the best candidates for enlargement.

One way to help these decisions is to use an optimisation algorithm. Work has shown that it is possible to improve the average delay and power consumption while also reducing variability in circuits by using optimisation techniques to help adjust the widths [29]. This could help a designer improve yields and average performance, but takes a long time due to many circuit simulations being required.

These inherent sources of variability cause faults in the final devices and systems and combined with other sources of faults can cause errors and ultimately failures of systems. The next section will consider a number of the more important sources of faults.

2.3 Sources of Faults

Electronic faults are an unwelcome but inescapable reality of integrated circuits. Though designed with ideal, well behaved components in mind, the physical products that we have to work with are imperfect, increasingly so as transistor scaling progresses. Reliability is important in both safety-critical situations, in which system failures could be life threatening, and consumer products, in which poor reliability reflects badly on a manufacturer. Faults can occur in any systems but don't necessarily lead to system failure in isolation. Consider a power output stage using BJTs in the following example presented by Tyrrell [30]:

> FAULT: A transistor Beta parameter changes due to age.
>
> ERROR: The transistor output current increases.
>
> RESULT: System output may not be affected as the output of the system including this faulty transistor is not taken outside its specification by the increase in current.
>
> Operation continues:
>
> FAULT: The increased current increases power dissipation in the transistor, increasing the transistors temperature.
>
> ERROR: Resistance of the base-emitter junction is reduced due to the heating effect (since Bipolar Junction Transistors have a negative thermal co-efficient)

SYSTEM FAILURE: The reduction in resistance increases the current and power dissipation, reducing the resistance, finally the transistor fails leading to a system failure.

This shows how a single *fault* (a change in a BJT transistor's Beta parameter, a problem with a component) which leads to an *error* (the output current changing, the consequence of the fault) may not lead directly to system *failure* (the whole system not working), which is usually caused by multiple faults or a chain reaction set off by a single fault [30]. Therefore, tolerance to faults is a crucial consideration during electronic design given the scale at which devices are now produced.

Fault tolerance is an area constantly researched in the contexts of both academia and industry. The remainder of this chapter introduces some of the areas and concepts around the topic, with a focus on fault recovery methods and reconfigurable devices. In addition, the topic of transistor scaling is introduced as additional motivation for investigating reliability and methods for fault tolerance.

There are many different kinds of fault which can occur in an electronic system, each with different causes.

2.3.1 Manufacturing

The process of manufacturing electronic circuits through photolithography is fraught with difficulties. Light is shone through a mask onto a photoresist material coated on the wafer. The parts of the resist that are exposed to the light react and will either be kept or removed in the etching phase depending on the technique. Historically, to get a better resolution and thus smaller feature size, the wavelength of the light used for the exposure was reduced. Since around 2001 state-of-the-art fabrication processes have used 193 nm laser light sources due to difficulties making smaller wavelength light sources commercially viable. This has necessitated the development of techniques such as immersion lithography (using a substrate other than air to fill the gap in projection lithography to achieve a refractive index >1), double patterning (exposing the wafer twice with two different patterns and resists to resolve features smaller than the resolution) and optical proximity correction (modifying the mask to compensate for errors) [31–34] in order to improve the resolution and accuracy achievable with a 193 nm light source, but it is still an imperfect process.

Despite improvements in lithography, as discussed in Section 2.2.1, the scale of modern transistors means their production is necessarily imperfect. Especially with modern nano-scale transistors, physical limitations prevent accurate and consistent production and mean that stochastic variation occurs, affecting the behaviour of transistors. Through the previously discussed mechanisms, circuits may be faulty to begin with.

2.3.2 Environment

The environment in which a circuit is operated can have an affect on its performance and can also cause faults. Heat and radiation are the main culprits in this case.

High temperatures increase power consumption and circuit delay. A number of phenomena occur when the temperature is increased:

- Semiconductors increase in conductivity, which increases current leakage in transistors. This increases power dissipation so leads to further heating.
- Metals decrease in conductivity, increasing delays in circuits and dissipating additional heat.
- Thermal noise increases, introducing processing errors.

Radiation can also cause processing errors and malfunction or even permanent failure. Ionising radiation can induce currents and cause unintended or illegal states to be entered by changing memory contents or computation results. High energy radiation can even alter the physical structure of transistors through lattice displacement, which can lead to permanent failure of components.

2.3.3 Time

Transistor ageing refers to a number of phenomena which can cause devices to “wear out”. As transistor scaling has progressed, these effects have become more significant and so manufacturers are careful to minimise stress on the devices. There are a number of different mechanisms by which a transistor can age which are described below (based on the information in [35] and [36]).

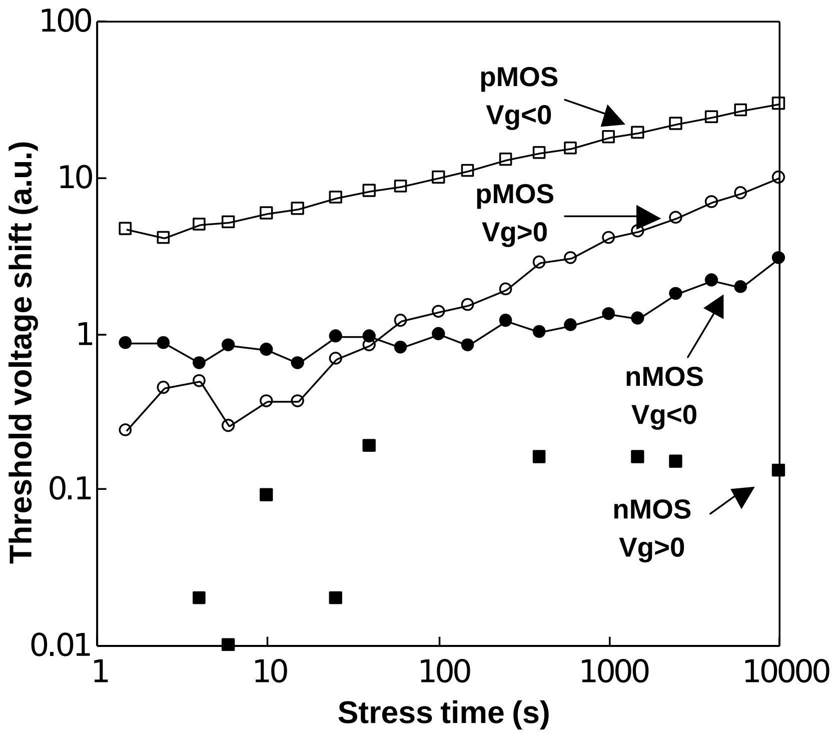


Figure 2.6: A graph showing the increase in threshold voltage over time for both PMOS and NMOS transistors with both negative and positive bulk bias voltages (taken from [37]).

Bias Temperature Instability (BTI)

Whenever a voltage is applied to the gate of a MOSFET, BTI can cause charge to build up in the dielectric. In PMOS transistors, this is Negative BTI (NBTI) and in NMOS transistors this is Positive BTI (PBTI), though NBTI is the predominant issue. Higher temperatures increase the phenomena, although the precise mechanism is still debated. BTI results in an increase in V_T (see Figure 2.6).

Hot Carrier Injection (HCI)

Charge carriers with high energy can get “injected” into the dielectric layer. As these build up over time, the resulting electric charge can increase the threshold voltage of the transistor, causing the transistor to switch more slowly and eventually stop working altogether.

Time-Dependent Dielectric Breakdown (TDDB)

The dielectric layer of a MOSFET is relied upon to be an insulator. Over time, the formation of many traps can cause this to break down and create a short between the gate the source-drain channel, which is likely to be destructive to the circuit.

Electromigration

Over time, the flow of electrons can physically move the atoms of metal conductors. This can ultimately result in breaks in circuit connections. This problem is increasingly relevant as integrated circuits are scaled down due to the connections becoming ever narrower and therefore reducing the number of atoms which would have to move to break a connection.

2.3.4 Summary of Fault Sources

This section has described a number of different sources of faults in devices. Considering that a single transistor failure could render a processor useless it is remarkable to contemplate the many billions of transistors on which we rely every day. Transistor scaling is increasing the amplitude of some of the fault sources mentioned in this section which presents some issues for the future - how can electronic devices remain reliable whilst still delivering the expected performance improvement? While technology is certainly improving all the time, higher levels of reliability are often required. The next section discusses approaches to fault recovery which is a part of the general subject of fault tolerance.

2.4 Fault Recovery Mechanisms

Given the reduction in reliability of modern electronic components, it could be argued that fault tolerance has never been so relevant. While the phrase “fault tolerance” can mean a complete system for tolerating faults (including detection and recovery, or else ensuring faults have no effect), this thesis will mainly focus on fault recovery mechanisms. It is noted that many techniques exist and have been proposed for fault detection [38–40] and so this is assumed to be possible.

Some previous work has investigated fault mapping, such as [41], which can be used to guide a fault tolerance methodology to work around known faults. This is out of scope for the work reported in this thesis as it focuses on finding online fault recovery mechanisms.

This section discusses a number of different methodologies for fault recovery including both standard and experimental techniques.

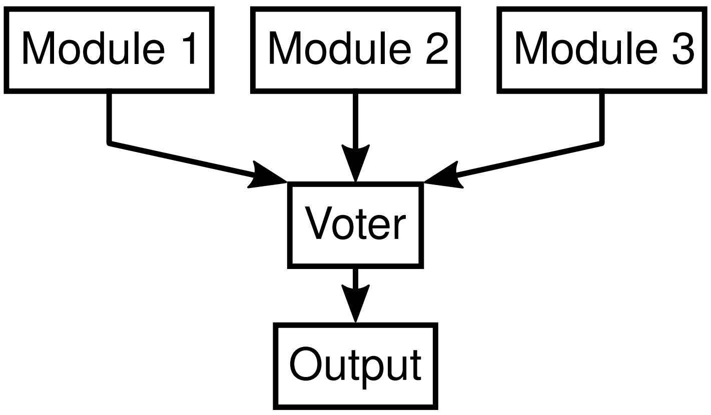


Figure 2.7: A block diagram of TMR. Three modules generate an output independently. The voter then compares these outputs and passes through the most common one to the output. If a minority of modules become faulty and begin outputting a different value, the voter will continue to output the correct value. If two modules develop faults in TMR, the voter can no longer know what the correct output should be, and so should report an error. If two modules fail simultaneously in the same manner, the voter will also not know this and so may output an incorrect value. For this reason, a common approach is to use different implementations of the modules so that failures are likely to manifest in different ways.

2.4.1 Triple Modular Redundancy (TMR)

Often looked to as the "gold standard" of fault tolerance, TMR (or higher modular redundancy) is a passive fault tolerance system used in safety critical situations. The implementation consists of three modules of identical functionality (though possibly different implementations) with their outputs connected to a voter (see Figure 2.7). If a majority of the modules' output values are the same, the voter outputs this value.

This means that if one module becomes faulty and outputs an incorrect value, the TMR system will mask this completely. If a second module becomes faulty, the voter will not be able to determine the correct output, but will be able to detect and alert as to the situation. The functionality of TMR therefore relies on the reasonable assumption that the probability of one module failing is higher than two failing in the same way (in which case the majority output would be a faulty one).

The advantages of TMR lie in its simplicity and ability to completely mask a single fault without intervention. This comes at the cost of $> 200\%$ overhead however, due to the redundant modules and voter.

Further to the large overhead, the voter remains a single point of failure and care must be taken to ensure its operation is as reliable as possible. If the voter fails, either by propagating the incorrect result or becoming stuck at one result, the whole system fails. Sometimes the voter itself is triplicated in order to provide additional protection.

The fault recovery mechanism of a system protected by TMR is static and so when faults occur the amount of redundancy is reduced; a single fault causes a whole module to be disregarded from the moment it is detected. If a fault occurs in a second module there is no way to tell which module (of the two remaining) has failed and so the whole system fails.

Despite these drawbacks, TMR (or a system with more than three modules) is mandatorily used in safety critical applications such as in the aerospace industry.

2.4.2 Reconfigurable Devices for Fault Tolerance

When devices fail permanently in fixed function integrated circuits there is usually nothing that can be done to recover functionality. The majority of components may be fault-free, but the failure of a single transistor or connection will render part of the circuit unusable.

Reconfigurable devices, such as FPGAs, appear to open up more possibilities when dealing with faults. Theoretically, if a fault occurs in part of a circuit configured on an FPGA, it may be possible to do a reconfigure that circuit to avoid using the faulty component. The difficultly comes in figuring out the minimal reconfiguration that can be made to both avoid the fault and regain functionality. It is desirable to avoid resynthesizing a design (converting a functional description of a circuit into a configuration for an FPGA device) as this is a computationally expensive procedure.

Many novel methodologies around fault tolerance using reconfigurable devices have been proposed. Some examples from the literature are presented below.

Relocatable Bitstreams

If a fault affects a small part of a module within a larger system on a reconfigurable device, it might be possible to use reconfiguration to recover from the fault. Montminy's methodology in [13] described how partial bitstreams could be made relocatable within a Xilinx FPGA, enabling a module to be relocated in the event of a fault. It was also shown that if the location of a fault was known, the same technique could be used to move a module implementing the same thing in a different way (avoiding the fault) to the same location. The work describes how these bitstreams are made relocatable, which means that the same bitstream can be implemented in multiple places as opposed to pre-generating bitstreams for all possible locations, thus saving time and memory.

Monminy's work references some similar work [11] which uses CLB columns rather than modules to achieve a similar goal, with pre-generated bitstreams for each column being produced at compile time. An interesting point noted in the paper is that should fault location data be unavailable, all possible column configurations can be tried until one is found that works. This avoids the complexity of including fault locating systems and reportedly performs well.

2.4.3 Bio-inspired Approaches

Nature has some inspiring solutions to fault tolerance which have enabled life to exist for millions of years. Most animals will sustain some sort of ailment during their lifetime and many recover fully from it. Of course, biology takes advantage of massive amounts of redundancy to make some of its repairs, but there are also mechanisms and features around the redundancy which we could take inspiration from.

Cell-based Systems

Upon detecting a fault, a cell based system will follow a set of rules for how to heal or work around the fault, based on the idea of cell replication in biology. For instance, the BioWall [42] implements a cell-like structure, consisting of thousands of LED display panels, each driven by an FPGA. In a Clock example, after a display cell becomes faulty, the functionality of the faulty cell is taken on by the cell to its right and the clock continues counting.

This approach is elegant in its simplicity, but does require the provision of spare cells in order to provide fault tolerance. One possible inefficiency in this methodology is that no matter what the fault, the whole cell is disregarded. This is not always necessary as it may be possible to reuse some components inside the cell, even if the original function stopped working. It should be noted however that this example was created to demonstrate a cell based system rather than efficiency.

Other cell based systems such as [43] and [44] work within a single chip and build functionality from a sea of homogeneous cells. As with the Biowall, the cells can reorganise around faults (Figure 2.8). These approaches appear to work well for fault tolerance so long as spare resources are available.

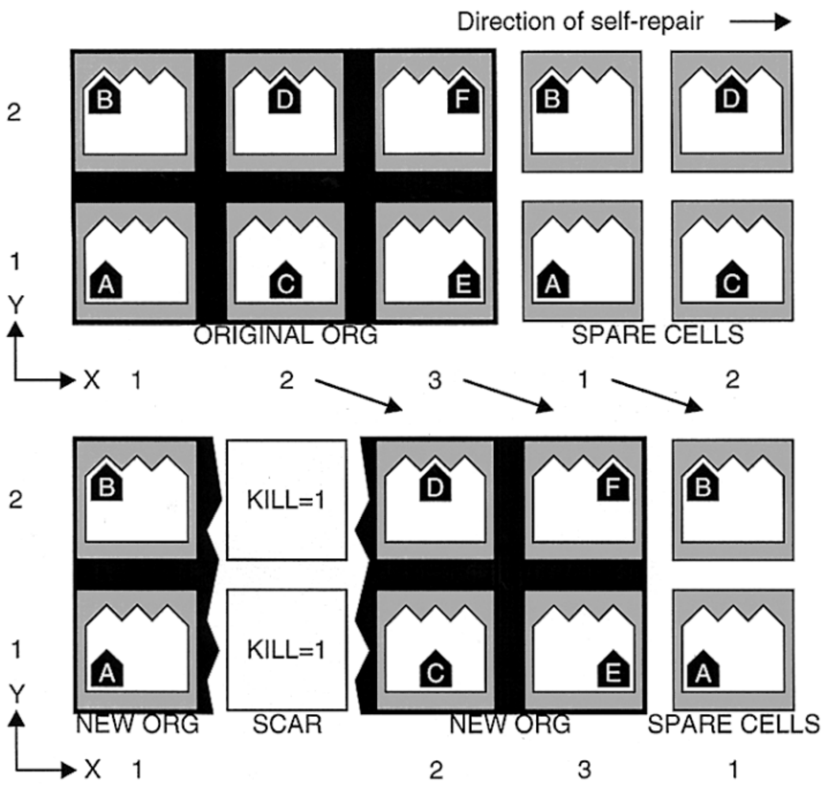


Figure 2.8: An illustration of cell based repair in the Embryonics approach [43]. The upper image shows a group of six artificial cells (an organism), each holding the same configuration data (like DNA) but each expressing a different part of it based on their position (configurations A-F). There are also four spare cells to the right of them. The lower image demonstrates how the middle column of cells in the organism can be killed (through becoming faulty for instance), triggering an automatic repair process. The rightmost cells of the original organism (configurations E and F) begin to express the configuration of the killed cells (C and D) and the first column of spare cells begin to express configurations E and F as replacements.

Beyond Cell Level

Mange's paper [43] goes beyond the cellular level previously mentioned to describe a novel architecture based upon the biology of multi-cellular lifeforms. In it, coarse-grained "cells" would be programmed by a global genome. Each cell contains a copy of the whole genome but, based on its physical location, would only express a certain part of it as its function (see Figure 2.8). The paper describes a few scenarios where this feature enables self-repair and self-reproduction. The work also introduces distinct layers above (organismic and population) and below (molecular) the cell level creating a hierarchy of configurable elements (see Figure 2.9). At the top level (called the population level), an array of processors can be configured to carry out work. Each processor (organism) is built from an array of cells which have the self repair abilities shown in Figure 2.8. Each cell is built from an chain of molecules which consist of very basic logic elements and a mechanism with which to detect faults. There are

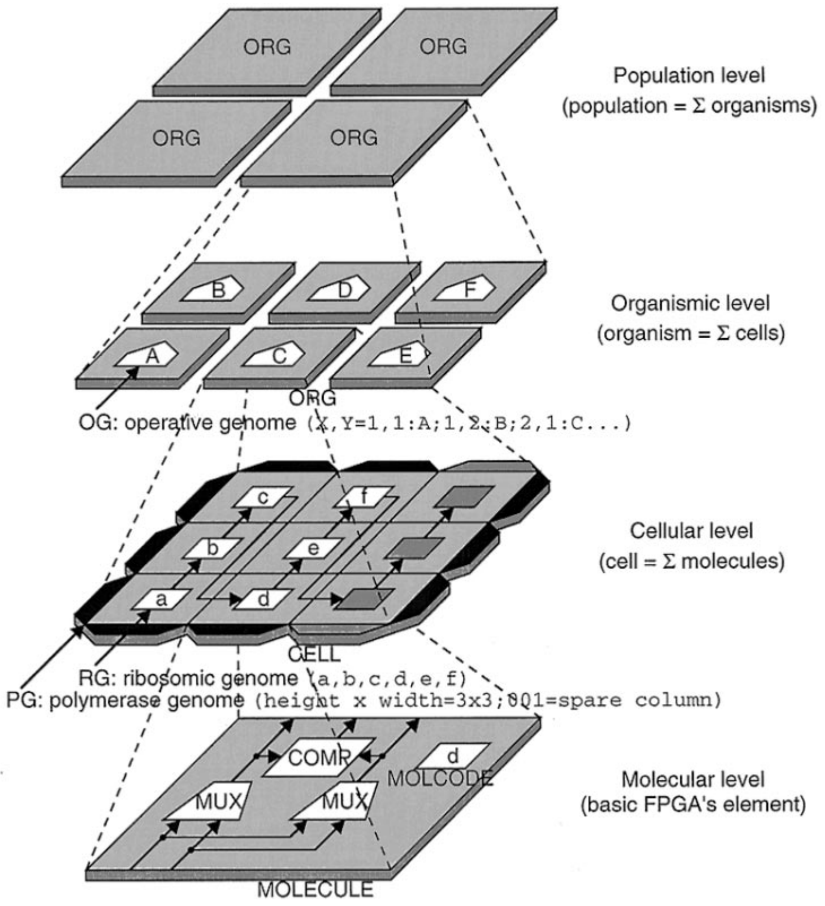


Figure 2.9: The hierarchy of the Embryonics architecture [43]. Figure 2.8 shows cells reorganising at the organismic level of this diagram. Each cell is made of molecules which in turn comprise of basic FPGA elements. Above the organismic level is the population level which consists of multiple organisms. This hierarchy of components enables the architecture to utilise redundancy at multiple levels.

spare resources at multiple levels and so if a molecule fails a repair can be attempted at the cellular level. If this is not possible, an organismic level repair can be attempted and so on. By performing repair operations at the lowest possible level, the number of repairs that can be carried out with the available resources is maximised.

The SABRE architecture [45] is another cell-based system which takes fault tolerance and re-use a bit further. The reconfigurable fabric is made up of Functional Units (FUs) built from Unitronics cells [46]. The cells themselves contain Built-In Self-Test (BIST) functionality and so can detect when a fault has occurred.

The FUs consist of a chain of cells with some of them configured to be spares. If a fault occurs in a non-spare cell, the cells beyond it in the chain are shifted along into the spare cells and the functionality from the faulty cell is transferred to the newly vacant one. Once all the spare cells are used up, the configuration for the whole FU is transferred to a spare one.

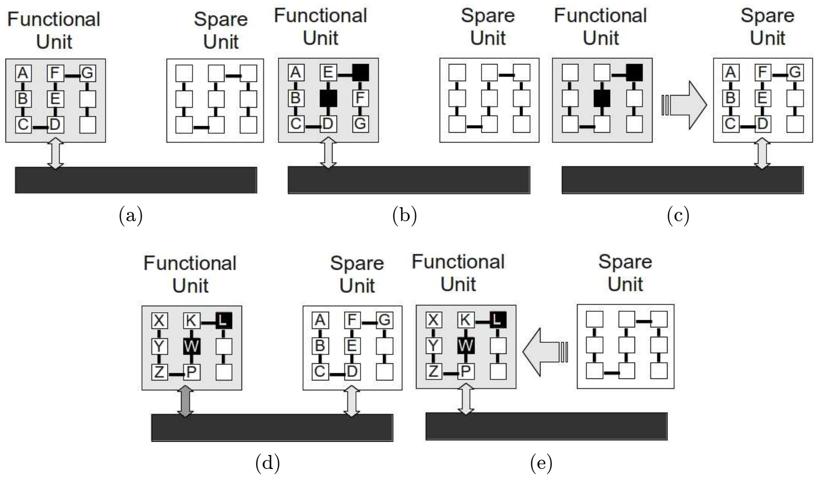


Figure 2.10: The reclaim procedure on the SABRE architecture [45].

After this happens, a reclaim algorithm attempts to find another configuration that will work in the faulty FU despite the faults. If one is found, this configuration is moved to the faulty FU to free up another spare.

This hierarchical strategy of making a larger change when there are no more smaller ones that can be made means that the available resources are used very efficiently and hence makes the system particularly tolerant to faults.

2.4.4 Reuse of Faulty Components

Some of the methods in this section have demonstrated the ability to reuse modules which have already found to be faulty by using them for other purposes that don't depend on the failed components. This seems an ideal strategy for keeping a system running for as long as possible and making the most of what's available. Some other pieces of work have focussed on this strategy more directly. DeHon demonstrates how partially faulty LUTs in FPGAs can be used in many situations despite errors [47]. It was found that the tolerable failure rate could be significantly increased through the use of some techniques such as permuting the inputs and changing their polarity to try to match faulty stuck-at outputs with desired outputs.

Emmert discusses using Partially Usable Blocks (PUBs) in [9, 48] as part of a system for fault tolerance. In the methodology, functional blocks are tested for faults on a continuous basis. If a block is found to be faulty it is further tested to identify the nature of the fault and whether the logic currently programmed in the block is compatible (able to perform the desired function) with the fault(s) that exist. If it is, the configuration is left there. If it is not, other configurations are tried until either one is found to work or else the block is marked

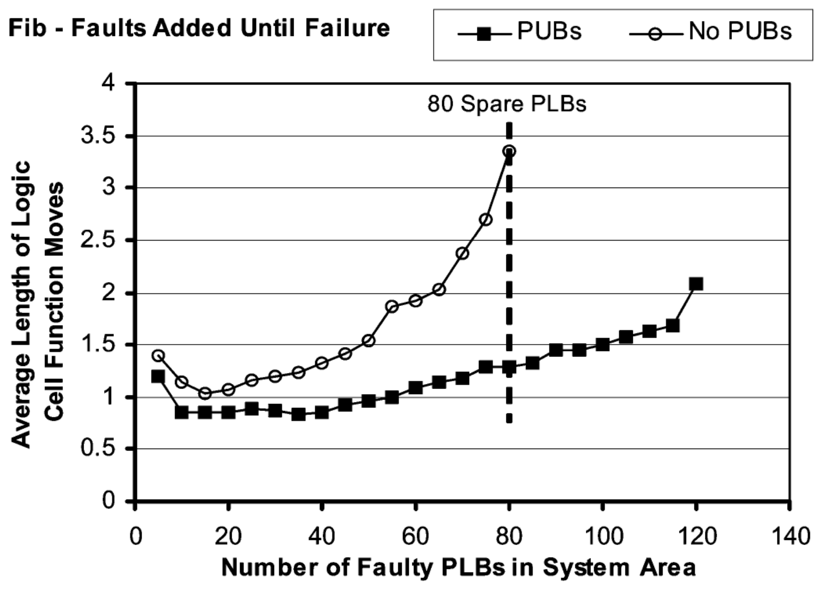


Figure 2.11: A graph showing the advantage of using Partially Usable Blocks (PUBs) (taken from [9]). The data is showing that for a Fibonacci Number Generator, faulty blocks could generally be replaced by physically close ones (by Manhattan distance) when PUBs were used and they could also handle more faults.

as unusable. Through reusing partially faulty blocks the method allows a system to tolerate a greater number of faults.

Both these techniques try to make the most of the hardware which is accessible to them, but are limited to functional blocks due to the architectures that they're targeting. If the faults that they're targeting are due to the characteristics of a single transistor, there is hardware which will go to waste. If a finer grained control were possible, this might reduce wasted hardware and increase fault tolerance at the cost of a higher overhead of extra configuration circuitry.

2.5 Summary

There has been a lot of research into how best to deal with what is an ever more unreliable technology. Variability tolerance is now something to be considered by circuit designers and must be measured during the design phase to optimise yield and performance variations. As discussed, initial performance is not the whole story and nanometre scale transistors are more prone to ageing effects, which will eventually lead to faults. Fault tolerance strategies have therefore never been more relevant in the world of electronics, not just in safety critical applications but in consumer electronics also where better reliability leads to more profit and customer satisfaction. While simply adding large amounts of redundancy to designs can achieve higher reliability, any benefit will quickly be defeated through increased cost and area taken up on the die. Similarly, making devices bigger would also make them more reliable due to lower intrinsic variability but with similar drawbacks in terms of area and power consumption.

Of the above mentioned techniques for fault tolerance, the ones which are most efficient in terms of hardware usage share some characteristics: fine-grained redundancy and reuse of faulty components. Neither characteristic, however, has been focussed on in particular and it seems there is room for significant improvement in both areas. A platform which could support reconfiguration at levels lower than a standard FPGA (like a Field Programmable Transistor Array (FPTA) [49]) but still be capable of implementing circuits of reasonable complexity would enable exploration of these two characteristics to a greater extent.

The next chapter discusses evolutionary algorithms which are a technique used in many novel approaches to fault tolerance. Subsequent chapters will then discuss how these evolutionary algorithms can be used with novel reconfigurable platforms to increase the reliability of devices and systems.

Chapter 3

Evolutionary Algorithms

Contents

3.1 Introduction

For the last few decades, evolutionary computing has been used for a wide variety of problems, from function solving to hardware optimisation.

Many resources exist on the topic of evolutionary computing, e.g. [50–52] thus this chapter gives a brief overview of the subject only and refers the reader to these publications for more detailed descriptions of specific areas. The focus of this chapter will be on some specific

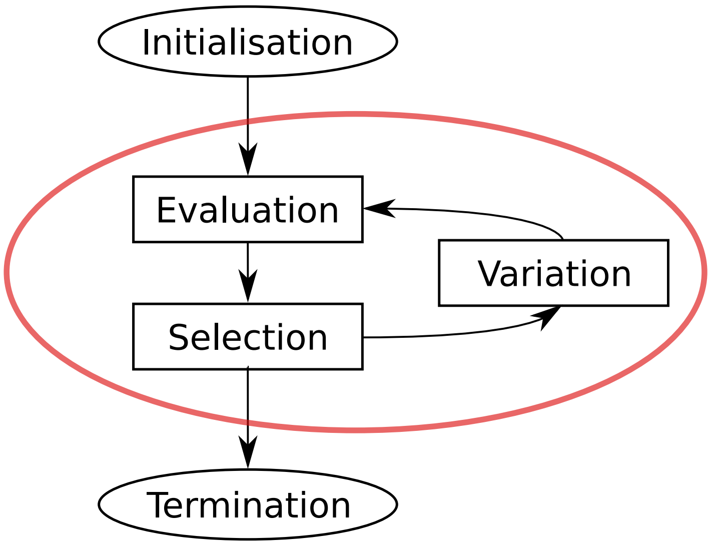


Figure 3.1: A basic evolutionary loop including Initialisation, Evaluation, Selection, Variation and Termination steps. The red oval highlights the algorithm's main loop.

techniques used in the work reported in this thesis and some previous work utilising these techniques.

Evolutionary computation is a general class of computational techniques inspired by biological evolution. The majority of these techniques involve meta-heuristic optimisation algorithms such as evolutionary algorithms: genetic algorithms [53], genetic programming [54] and evolutionary programming [55]. Other forms of evolutionary computing include swarm intelligence algorithms such as ant colony optimisation [56] and particle swarm optimisation [57]. For the work reported in this thesis the majority of this chapter will focus on genetic algorithms (GAs).

A basic overview of GAs is provided in this thesis, along with an explanation of Multi-Objective (MO) approaches. Multi-objective concepts are critical in most real-world problems as multiple, conflicting objectives must be optimised at the same time [58]. The concept of evolvable hardware is then introduced and some previous work that uses this concept for both optimisation and fault tolerance is discussed.

3.2 Genetic Algorithms

Genetic Algorithms (GAs) are a commonly used branch of evolutionary algorithms. They are simple to implement, and it is often straightforward to express the solution to a problem as a genome.

The general form (see Figure 3.1) consists of an initialisation stage followed by a cycle of three stages: evaluation, selection and variation. Finally, when some pre-defined criteria is met, the algorithm enters the termination stage and stops. The initialisation stage usually creates a random population of potential solutions which are then scored on how close they are to the requirements in the evaluation stage. The selection stage picks some of the best candidate solutions and the variation stage makes changes to these in particular ways. These are then evaluated and the cycle continues. By making changes to the solutions and then picking out the best, the algorithm mimics natural evolution in its ability to create successful individuals within a population which in many cases leads to an optimised solution. These steps are discussed in more detail below.

Some of the advantages of GAs over other optimisation techniques include:

- They can be applied to applications where little knowledge is encoded in the system.
- Problems with a high combinatorial complexity can be tackled efficiently.
- They are reliable and robust in finding good solutions to complex problems without getting stuck in local optima.
- GAs are amenable to parallelisation and are straightforward to implement.
- Many problems are straightforward to represent.
- The iterative process gives gradual improvements leading to good solutions even if a perfect one is not feasible.
- Varieties exist that can deal with multiple competing objectives (e.g. NSGAII [59]).

Disadvantages of GAs include:

- No guarantee of finding an optimum solution.
- Running times can be unpredictable and long.
- Setting of parameters (e.g. population size, mutation rate) involves guesswork.
- Poor choice of problem representation can affect results in a negative way.

When investigating a new problem where the solution landscape is unknown, GAs are good at getting a general feel for what solutions exist. For the work proposed by this thesis GAs

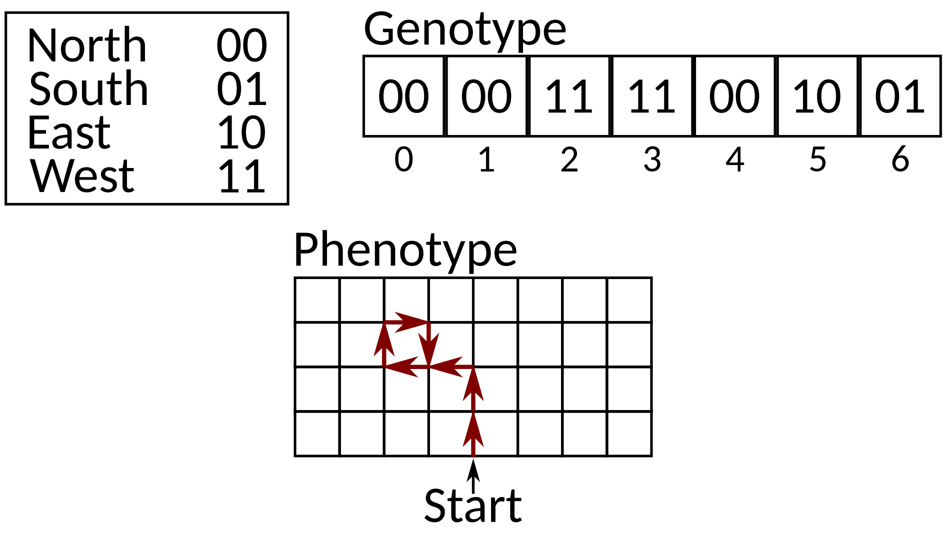


Figure 3.2: A basic genotype to phenotype mapping is presented where 2-bit values in a genotype are mapped to directions on a grid. If the mapping was changed, the phenotype generated from the genotype would be different.

are a good general purpose tool for exploring possible solutions. Additionally, multi-objective GAs are very useful in the electronic design space where good solutions are frequently a compromise between speed and power, and no optimal solution exists.

Techniques involving SAT solvers have been demonstrated in the past as methods of efficiently generating FPGA bitstreams that work around faults [60]. These generally work by formally specifying the required circuit and then using a SAT solver to find a solution. SAT solvers were not considered for the work in this thesis because they can only differentiate between feasible and infeasible solutions, there is no measure of how good a solution is nor can they optimise multiple objectives which is what was required for the methodologies proposed by this work.

3.2.1 Genetic Representation

Individual solutions in GAs are represented by a set of genes, similar to the way individuals are in nature. A fixed size structure of primitive types is commonly used, such as an array of bits, but variations of this have also been explored. A particular set of gene values is known as a genotype and maps through some process to a phenotype, which is the actual solution. An example of this genotype/phenotype mapping might be a representation of a path through a coordinate grid (Figure 3.2). The genotype may consist of an array of 2-bit values which, when mapped to the directions north, south, east, west and followed in order, creates a phenotype of a path around the grid. Obviously the mapping has a significant impact on the phenotype in this case, as changing the direction to which each 2-bit value refers to affects the path that is generated.

The genetic representation is a design decision to make which can have a significant impact on the results. Depending on the problem, a representation may perform poorly if similar genotypes have large differences in their phenotypes, as small changes (mutations) may result in large variations in fitness. The optimisation process in this case effectively becomes a guided search based around the selection and variation stages.

3.2.2 Evolutionary Loop

As in Figure 3.1, the main portion of an evolutionary algorithm is the loop from evaluation to selection to variation and back.

Initialisation

A population of individuals or potential solutions is initialised. These individuals can be generated completely at random, or through some strategy, perhaps maximising the spread along all of the problem dimensions or exploring the search space around an existing solution.

Evaluation

Each individual in the population is evaluated as a potential solution using some fitness function. This provides a measure of how close the individual is to the ideal solution, allowing comparisons between them. A simple, single objective fitness function may take a solution as an input and output a number, where a lower value is better. For example, evolving a simple logic gate circuit might use a fitness function which computes the truthtable of a candidate solution and reports the number of incorrect outputs where fewer is better. Many problems have more than one objective however and so more than one objective must be optimised. With the logic gate example, delay and power are usually also relevant and so multi-objective optimisation has to come in to play (see Section 3.3).

Selection

A subset of individuals from the population are picked to seed the next generation, or the result if the termination criteria is met. Most GAs use a fixed population size so there needs to be a way of going from (parents + offspring) to the next generation of the appropriate size. There are a number of different strategies for this selection stage, including:

- Roulette Wheel - individuals are selected randomly with a probability proportionate to their fitness.
- Tournament Selection - a group of individuals are picked at random and the fittest is selected for the next generation.
- $1 + \lambda$ - the single fittest individual is selected, where a new individual is given precedence over one carried over from the previous generation.
- Combinations of the above

In most cases a form of elitism is involved in the selection process which means that the best individual(s) are simply copied into the new population. This ensures that good solutions remain in the population.

Variation

This stage simulates the genetic mixing and mutations that happens during natural reproduction.

In nature, the genome of a living thing is normally made up from a mixture of two parent genomes. During the combination of these, random errors are occasionally introduced, known as mutations. The effect of this is to create variation in a population. In the next generation, individuals with mutations may pass them on to their offspring. Their offspring may also include further mutations as well as being the product of two parents. If mutations aid the survival of individuals (or improve their fitness), they are more likely to reproduce and pass on their traits (or in the case of an EA, be selected). On the other hand, if mutations hinder the survival of individuals (or diminish their fitness), they are less likely to reproduce and pass on their traits.

The net effect of this process is that a population evolves (over a long period of time) to a stronger state, with a higher probability of survival.

In genetic algorithms, this is modelled in the variation stage using genetic operators, functions which operate on a genome. Two commonly used operators are crossover and mutation, which represent the combination of two parent individual's genomes and the random mutation found

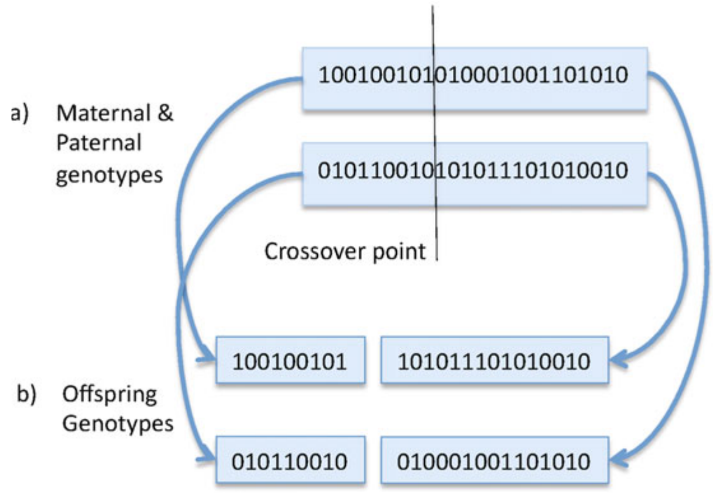


Figure 3.3: The Crossover genetic operator, which combines two genomes to produce offspring (taken from [30]). This diagram illustrates single point crossover but other variations exist.

in nature respectively. A simple crossover operator picks a point in the genome and copies all genes before it from one parent and all those after it from the other (see Figure 3.3). Mutation operators will often make changes stochastically to the genes, typically based on a low probability to minimise changes; large jumps around the search space tend to hinder convergence towards good solutions.

Most evolutionary algorithms, including GAs, use such variation operations to cause a selection drift towards better solutions. In many cases both variation operators will be used although it is also common to use only only one of them, either crossover or mutation.

Termination

Rather than run indefinitely, termination criteria is specified to exit the evolutionary loop when certain conditions are met. Common thresholds include the number of generations; the values from the fitness function; the relative improvement of the fitness since the last generation or some combination of these criteria. Termination may also be carried out manually if the current metrics can be monitored.

3.3 Multi-Objective Genetic Algorithms

Commonly, a problem will require optimisation against a set of conflicting objectives (see Figure 3.4). A naïve approach combining a set of fitness values via multiplication or addition often results in over-optimisation of one of the objectives. In the example given in Figure 3.4,

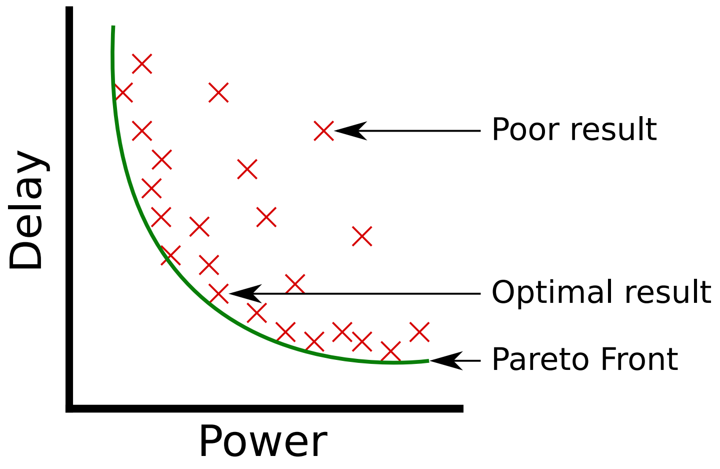


Figure 3.4: An example of a pareto plot. The pareto front is the imaginary boundary between feasible and infeasible designs. This figure illustrates a common goal in electronic design: to minimise both the power and delay of a circuit. These are typically opposing objectives in that one can build a very fast circuit which consumes a lot of power, or a slow circuit which is very power efficient. Thus a designer must pick a suitable trade-off between the two objectives based on their requirements. In this figure, the performance characteristics of some designs are plotted. An ideal design would appear at the origin, having both low power and delay, but this is not a feasible result. Instead, the optimum trade-offs appear on the pareto front.

a circuit may be found which possesses a very low delay, but consumes a large amount of power. If the delay is near 0, the fitness value will likely suggest this is a good candidate despite the fact that the power consumption is undesirably high.

If we consider a graph plotting delay against power consumption (Figure 3.4) of different designs for a transistor circuit, there is some imaginary line which bounds the performance. It is not possible to achieve the lowest power consumption along with the fastest performance (in this particular example the fundamental characteristics of the devices will not allow minimum power at the same same as having minimum delay), there is always a trade-off to decide upon. Multi-Objective optimisation algorithms attempt to find this line, allowing the designer to pick from a range of trade-offs for any particular set of requirements.

Multi-Objective (MO) GAs modify the evolutionary loop in order to avoid this over-optimisation. A popular example of an MO GA is Deb's NSGA-II [59]. This algorithm attempts to find a spread of solutions along the pareto front of a solution space by defining a sort order for individuals with more than one fitness value. An individual is preferred if it is better in at least one of the objectives. In addition to this, there is a measure of crowding distance: how close an individual is to its neighbours. By preferring to maximise the average crowding distance, a spread along the pareto front is formed, achieving better diversity in the population.

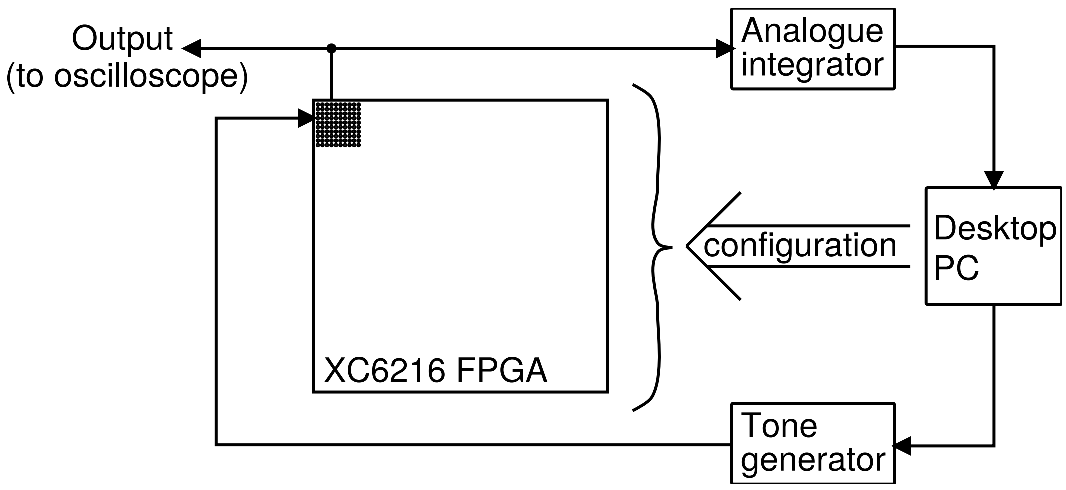


Figure 3.5: Diagram showing the experimental aparatus used to evolve and unclocked tone discriminator in an FPGA (taken from [61]). The Desktop PC configured the FPGA and controlled a Tone generator which output 500 ms bursts of 1 kHz and 10 kHz tones. The output voltage of the FPGA was integrated by the Analogue integrator (which was reset for each burst) and the result of this was fed back into the PC. The output of the FPGA was also sent to an ocsilloscope for observing the evaluations.

3.4 Evolvable Hardware

Evolutionary algorithms may be used to automatically explore hardware designs that would otherwise be extremely hard for a human designer. They allow physical variability in electronic circuits to be exploited to produce designs that are tuned for specific pieces of hardware as well as producing novel solutions to problems.

A classic example of evolvable hardware is Thompson's work [61] in evolving an unclocked tone discriminator on an FPGA. The work was able evolve an FPGA configuration that would output a low signal when the input was a 1 kHz tone and a high signal when the input was a 10 kHz tone. This was somewhat of an achievement as FPGAs are intended to implement digital circuits and no clock signal was involved, which could have been used for comparison. The evolved circuit appeared to take advantage of the physical properties of the FPGA being used for evaluation as when the same configuration was tried on a different FPGA of the same model, the circuit did not perform the same. This was an example of intrinsic evolveable hardware which means that physical hardware was used as part of the evolutionary loop (see Figure 3.5). The other type of evolvable hardware is extrinsic which means that the fitness of an individual is evaluated in simulation [30].

Figure 3.6 shows how the output from the FPGA evolved over 3500 generations from a having a fixed output to having the desired behaviour.

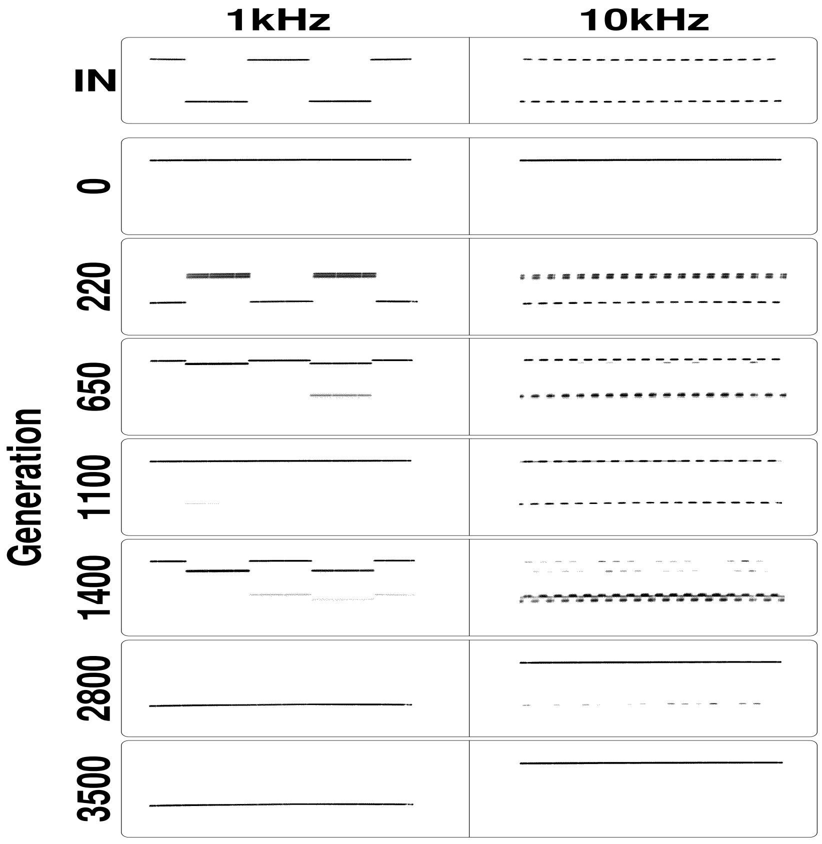


Figure 3.6: The output of the evolved tone disriminator at different generations as seen on the oscillioscope (adapted from [61]). The output begins as a constant high output and gradually changes to a consistent low when the input is 1 kHz.

3.4.1 Difficulties in Evolving Hardware

It is sometimes necessary to use some sort of simulation to evaluate individual solutions when evolving hardware (extrinsic evolution). For instance, integrated circuit design would require the fabrication of 1000s of candidate circuits which is simply infeasible both in time and financially. The simulation time for a single circuit is often in the order of minutes when using a simulator such as SPICE and hundreds of samples may be required for each circuit to achieve statistical significance with regards to variability.

As well as being time consuming, there is also a reality gap. Simulations make assumptions about parameters and ignore others in order to be usable, it's not worth simulating the universe to find out if an AND gate will function.

For both these reasons, it is often desirable to have physical hardware "in the loop" when attempting evolvable hardware, which often means the use of a reconfigurable device in the case of circuit design (see Figure 3.7).

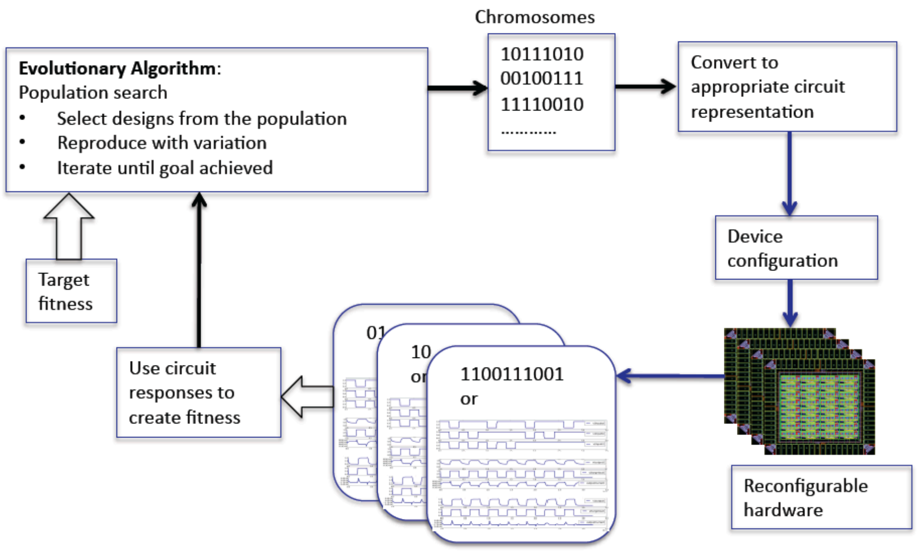

Chromosomes
10111010
00100111
11110010
…………
Evolutionary Algorithm:
Population search
• Select designs from the population
• Reproduce with variation
• Iterate until goal achieved
Convert to appropriate circuit representation
Device configuration
Target fitness
Use circuit responses to create fitness
01
10
or
1100111001
or
Reconfigurable hardware

Figure 3.7: An intrinsic evolutionary loop (taken from [30]). Candidate solutions are programmed onto a reconfigurable device and evaluated by measuring the circuit outputs. This is the method used in the work discussed in introduction to this section. Extrinsic evolution would replace the "Device configuration" and "Reconfigurable hardware" steps with a simulator such as SPICE, where the circuit representation might be a netlist.

3.4.2 Evolving Circuits at the Transistor Level

A desire to allow evolution more fine-grained control of led to the conception of the Field Programmable Transistor Array (FPTA) [62]. The device exposes a configurable array of programmable PMOS and NMOS transistor cells each of which have a selectable length and width and can be connected together in any orientation (see Figure 3.8).

Each programmable transistor can be configured to have one of fifteen different widths, achieved by enabling one or more parallel transistors. The effective width of the programmable transistor is the sum of any enabled parallel transistors. The length of the programmable transistor is selectable from five different options. These configuration options affect the analogue properties of the transistor and have enabled the construction of analogue devices such as Op-amps [64].

3.4.3 Optimising against Variability

As mentioned in Chapter 2, increased variability in transistor behaviour has forced designers to consider how circuits will behave when subject to stochastic variability. One approach to this is to alter the widths of individual transistors as this has an effect on the statistical performance of a circuit subject to variability [65]. This is a difficult task for a designer due

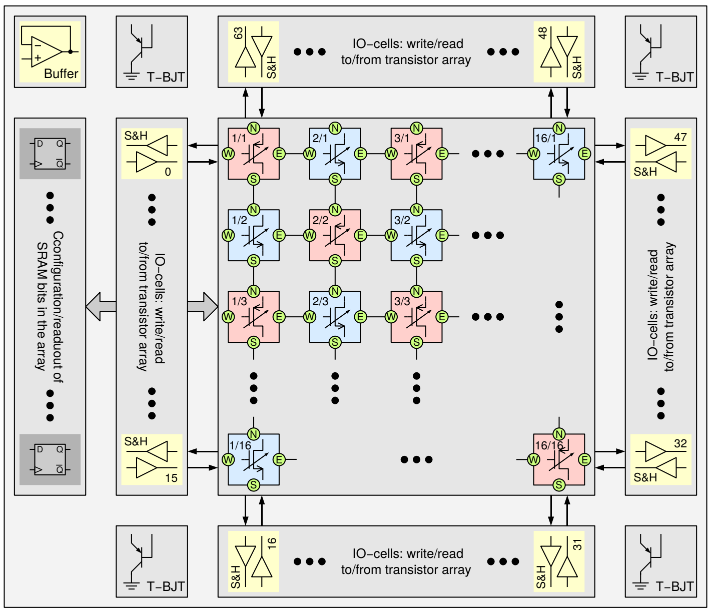

Buffer
T–BJT
IO–cells: write/read to/from transistor array
S&H
Configuration/readuout of SRAM bits in the array
N
W
E
S

Figure 3.8: The architecture of Heidelberg's FPTA (taken from [63]). The main feature to note is the chequerboard array of PMOS and NMOS programmable transistor cells, each of which can be enabled or disabled by the configuration. In addition to a programmable transistor, each of the four sides can be connected to any of the other three or to a node of the transistor.

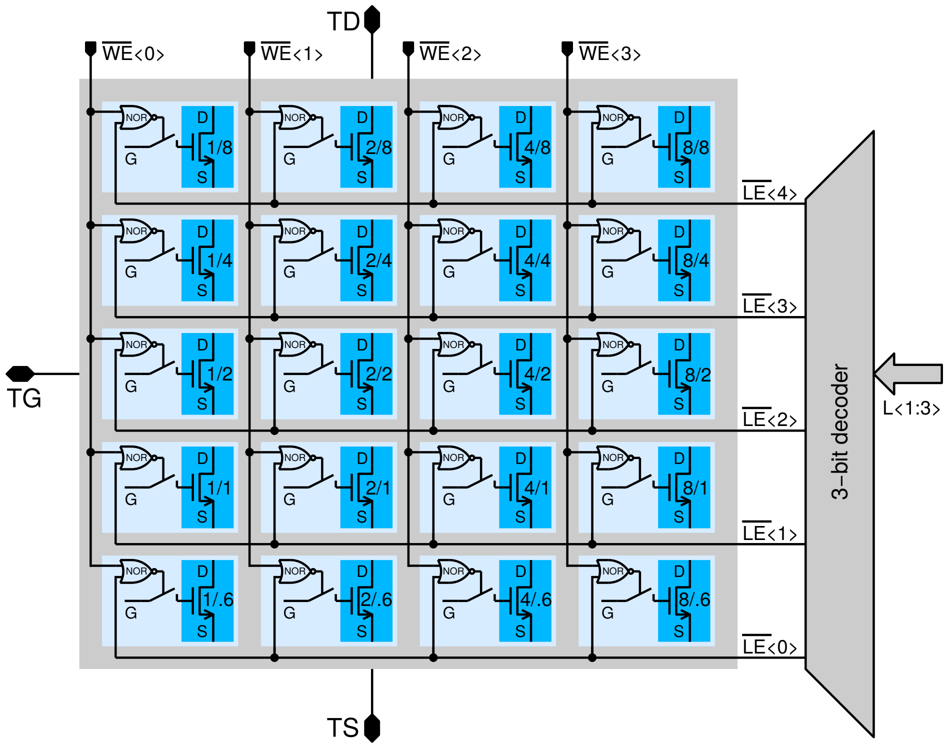


Figure 3.9: A programmable transistor from Heidelberg's FPTA (adapted from [63]). The length of the programmable transistor can be configured to one of five options: 0.6 µm, 1 µm, 2 µm, 4 µm or 8 µm and the width can be made up from any combination of 1 µm, 2 µm, 4 µm and 8 µm.

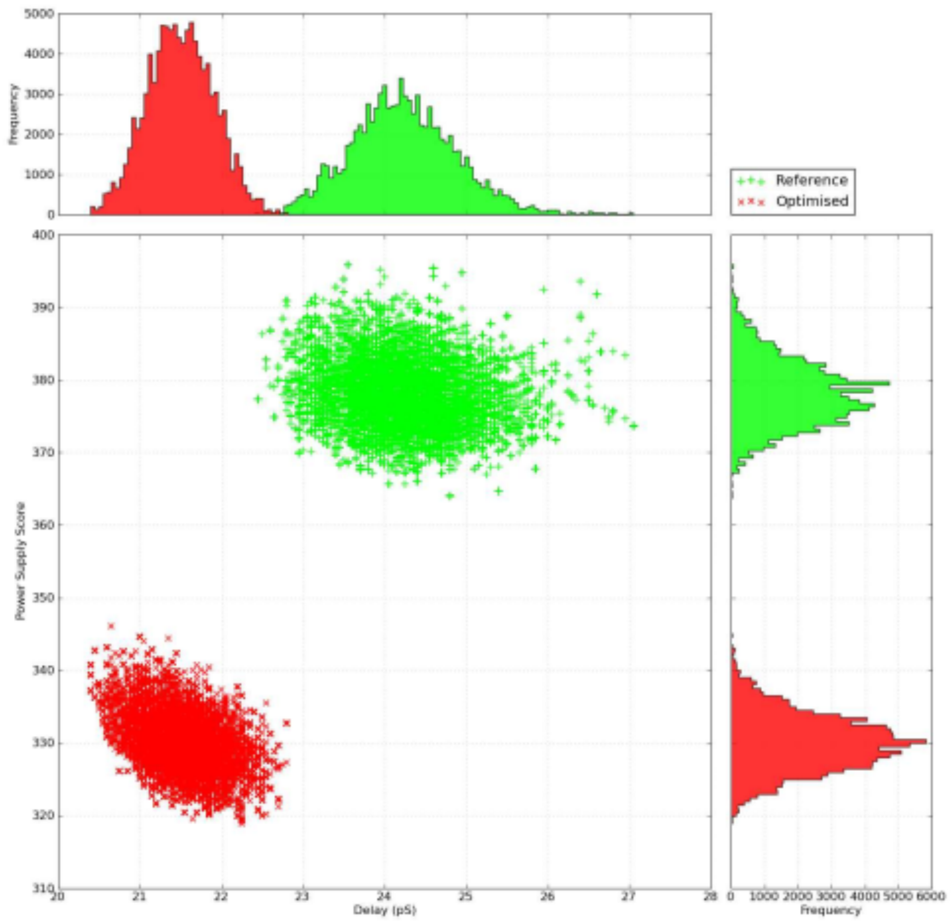


Figure 3.10: A graph showing how optimisation has improved the average performance of a buffer circuit when subject to transistor variability (taken from [65]). The green points represent different samples of the original circuit and the red points are the optimised circuit.

to the number of variables available and also the multiple objectives for the circuit such as power and speed requirements. Optimisation techniques, particularly multi-objective genetic algorithms, have been shown to be effective for this type of problem (see [65] & Figure 3.10).

3.4.4 Evolution for Fault Tolerance

Evolutionary approaches have also been used to add fault tolerance to a system. These generally take one of two forms: fault recovery or intrinsic fault tolerance. Keymeulen previously made a similar distinction in [66], calling the two forms fitness-based and population-based.

Fault Recovery Systems

Fault recovery is the fixing or recovering from a fault after it has occurred. Circuits on an FPGA or other reconfigurable architecture can sometimes suffer from faults which can be

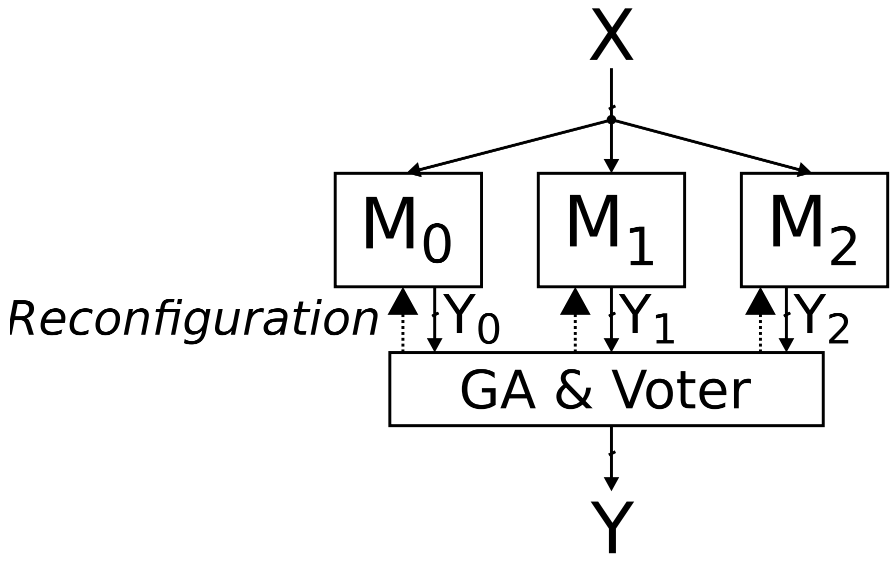


Figure 3.11: Diagram illustrating Garvie's system for fault tolerance (taken from [67]). The three modules M_0 to M_2 accept input X and output Y_0 to Y_2. The voter works like a standard TMR system to produce output Y. If a module is detected to be faulty (by producing a different output to the other modules) the GA implementation will attempt to reconfigure the module to make it work again.

overcome using reconfiguration. Various approaches can be found in the literature that use evolution to perform this reconfiguration automatically.

One example of using evolution for fault recovery is Garvie's "jiggling" method [10], which combines TMR with a minimal GA on an FPGA. After a permanent fault has occurred in one of the three modules, the system attempts to repair the faulty module whilst continuing to function with the other two. In order to repair the faulty module, the system uses the configuration bitstream as a genome and mutates it by bit-flipping (see Figure 3.11).

Intrinsic Fault Tolerance

Intrinsic fault tolerance is taken to mean using evolution during the design stage to produce a circuit that either tolerates faults naturally using redundancy, or degrades gracefully in the presence of faults.

The work in [8] examines whether evolving circuits in the presence of faults increases their fault tolerance compared with circuits evolved in a fault free environment. In their work, oscillator circuits showed a 12x increase in fault tolerance when evolved in the presence of faults.

Hartmann's thesis [68] describes an interesting approach to developing intrinsic fault tolerance in systems. Inspired by the seemingly intrinsic fault tolerance exhibited by nature, the work

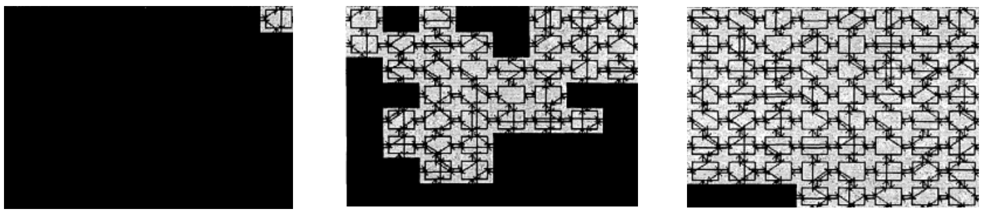


Figure 3.12: Diagrams showing the critical CLBs in oscillator circuits (adapted from [8]). 3.12a & 3.12b show the critical CLBs of oscillators evolved in the presence of faults, with slightly different definitions of critical - 3.12a a CLB that causes a failure when the output is forced to one or zero and 3.12b a CLB that causes a failure when it's output is forced to a one or a zero a the same time as another CLB. 3.12c shows the critical CLBs of a control oscillator which was not evolved in the presence of faults. The work found that the oscillators evolved in the presence of faults were much more tolerant to random faults being injected than the control oscillators.

attempts to evolve systems on an intentionally unstable platform. A novel idea used for this was the use of simulated "messy gates" (see Figure 3.13). These were essentially based on standard digital logic gates extended in two ways: firstly to give an analogue response at the output and secondly to produce a defined amount of random error. Connections between the messy gates were then evolved to give a desired, "non-messy" output. The work found that it was possible to successfully evolve small circuits, for example a NOR gate, and that these circuits exhibited a more graceful degradation in function when exposed to greater levels of noise and faults than standard circuits. It is suggested that this inherent fault tolerance could be similar to that found in nature, where systems have developed in noisy and fault-prone environments and perhaps as a result, are fairly resilient to faults.

3.5 Summary

This chapter has discussed GAs as an example of Evolutionary Computation. The basic elements of the method were discussed and it was explained how a GA can be used to explore difficult or unknown design spaces. A method of utilising multiple objectives in GAs was also introduced, which is especially helpful in electronic engineering where multiple conflicting design objectives are common, and any feasible design represented by a trade-off.

The concept of evolvable hardware was introduced and some of its applications discussed, in particular the use of evolution in circuit level fault tolerance.

Previous work that has experimented with these approaches has mainly focussed on either transistor level or function block level optimisation and fault tolerance. While focussing on

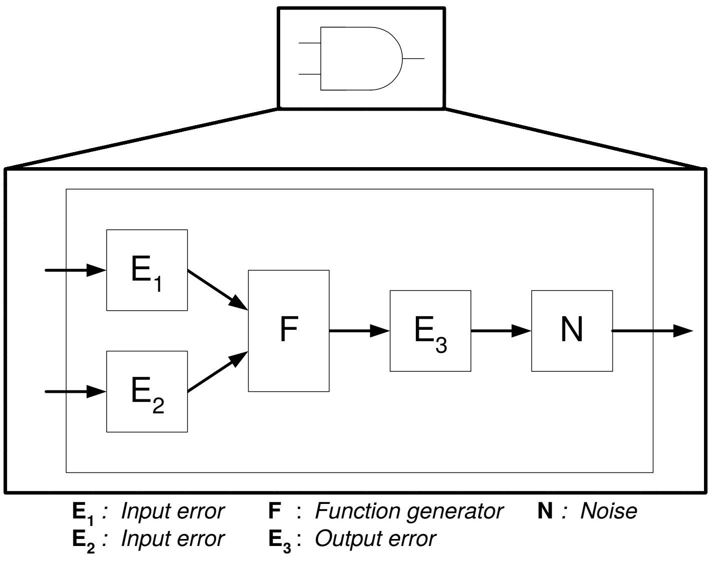


Figure 3.13: A diagram of one of Hartmann's messy gates (taken from [68]). The component functions similarly to a regular logic gate, but errors are introduced on the input and output as well as analogue noise being added to the output.

either of these levels of abstraction is useful, it would be interesting if it were possible to combine the two in order to both optimise the transistor level circuit and maintain functionality at the higher levels. Fault tolerance on an FPTA is interesting, but it is not feasible to build a system of modern complexity with existing FPTA architectures due to the relatively small number of functional transistors available given the overhead of the control circuitry. On the other hand, function block level fault tolerance can only do so much. A permanent fault or variability outside of expected values in a single transistor can make sections of a circuit unusable.

The next chapter discusses a novel reconfigurable architecture which allows for both transistor and function block level reconfigurations which potentially allows for the best of both types of approach.

Chapter 4

PAnDA-Eins

Contents

4.1 Introduction

PAnDA is a novel reconfigurable architecture similar to an FPGA but with additional layers of configuration. PAnDA-Eins is the first iteration of the architecture to be fabricated following its design by members of the Intelligent Systems Group at the University of York.

The design was inspired by the digital prototyping capabilities of traditional FPGAs and the low-level design exploration enabled by devices such as Heidelberg's FPTA device [62]. The combination of the two allows for unique exploration of the analogue design space which underpins the performance and behaviour of digital circuits. In doing so, this architecture presents many interesting options for reconfiguration which could also be exploited for other purposes such as fault tolerance.

This chapter serves to document the PAnDA-Eins architecture, beginning with an explanation of the motivations behind it and moving on to technical details regarding its implementation and features.

The PAnDA chip itself was designed by others in the research group but I designed all the experiments reported in this thesis (unless otherwise specified). Results from these experiments contributed significantly to improvements made to the original architecture.

4.2 Motivation

As transistors get smaller and less reliable (see Chapter 2), more must be done by the designer to account for the variability introduced. Standard simulation-based approaches are highly time consuming and restrict the designer in terms of what's feasible since many instances of the same circuit must be simulated in order to get meaningful statistics. There is also the issue of accuracy, and the results depend very much on the correlation between the transistor models and the performance of those actually produced.

One method of improving transistor circuits variability tolerance is to adjust the combinations of channel widths. This is a non-trivial process for which evolutionary algorithms have been applied with success [65], but the problems of simulation time are multiplied by the many different circuit evaluations which must be done for the evolutionary algorithm to work effectively.

Ideally there should be a way to test out different circuit designs physically using hardware to give an improvement in speed and accuracy, in the same way FPGAs are used for prototyping digital designs.

Heidelberg's Field Programmable Transistor Array (FPTA)[62] was one of two configurable transistor arrays [69] developed in the early 2000s. These reconfigurable devices allowed for configuration at the transistor level, at which both digital and analogue circuits could be built. The FPTA developed at Heidelberg in particular allowed for the channel length and width to be configured, post-fabrication, which could be used to vary certain properties of circuits built using it. In theory, this device could be used for testing designs with different transistor widths quickly and accurately.

Due to the complexity of having an array of fully configurable transistors however, it was not possible to include as many configurable transistors as would be necessary to build even a modestly complex circuit that would easily fit on a small FPGA. This creates a gap between

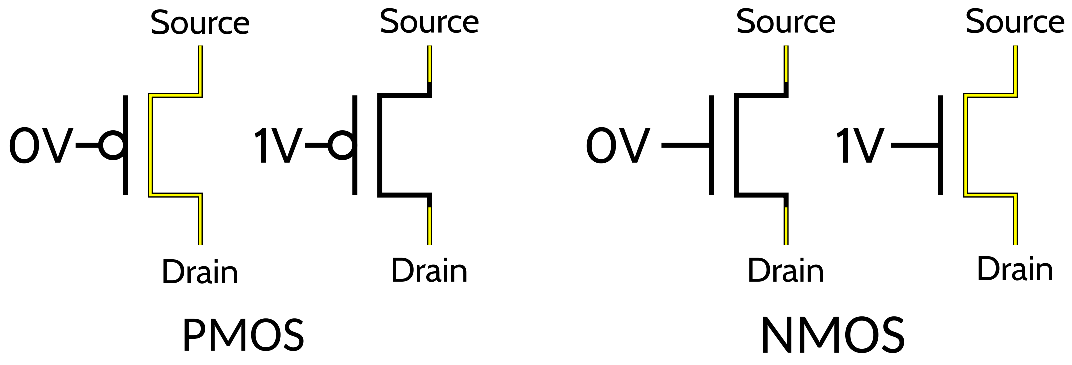


Figure 4.1: A comparison of PMOS and NMOS transistors. PMOS transistors conduct when the gate voltage is 0V, whereas NMOS conduct when the gate voltage is 1V.

the very fine grain detail of the FPTA and the larger circuit capabilities of FPGA, which is where PAnDA comes in.

PAnDA's design was based upon the idea of combining the digital reconfigurability of FPGA with the low-level flexibility of FPTA. By being configurable at both the digital and analogue levels, PAnDA allows exploration of the circuit design space in hardware, bringing both the speed and accuracy missing from current design flows.

4.3 Transistor Circuits

An ideal transistor can be simplified to a switch that conducts when its gate sees a certain voltage and insulates when it doesn't. In a 1 V circuit, an NMOS transistor conducts when the gate voltage is 1 V (the 'high' state), and a PMOS transistor insulates. Conversely, if the gate voltage is 0 V (the 'low' state), a PMOS transistor conducts and an NMOS transistor insulates (see Figure 4.1). Logic gates work using these assumptions, the simplest case being an inverter (see Figure 4.2). When a logic-high voltage is applied at the input, the PMOS transistor insulates and the NMOS transistor conducts, creating a path from the `GND` rail to the output, thus giving a logic-low output. When a logic-low voltage is applied at the input, the PMOS transistor conducts and the NMOS transistor insulates, creating a path from the `VDD` rail to the output, thus giving a logic-high output.

This principle is expanded to create more complex functions as will be discussed in later chapters.

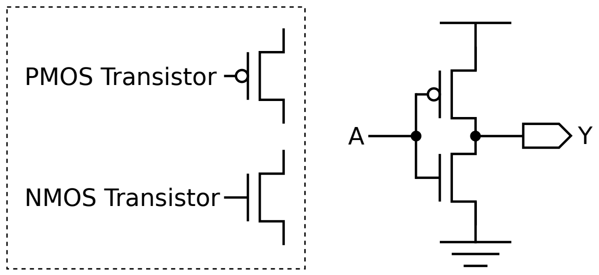


Figure 4.2: A CMOS inverter circuit (taken from [70]). The PMOS transistor will conduct and the NMOS transistor will insulate when the input A is 0 V, so the output Y will be driven to `VDD`. When the input is 1 V, the opposite is true and the output Y will be driven to `GND`.

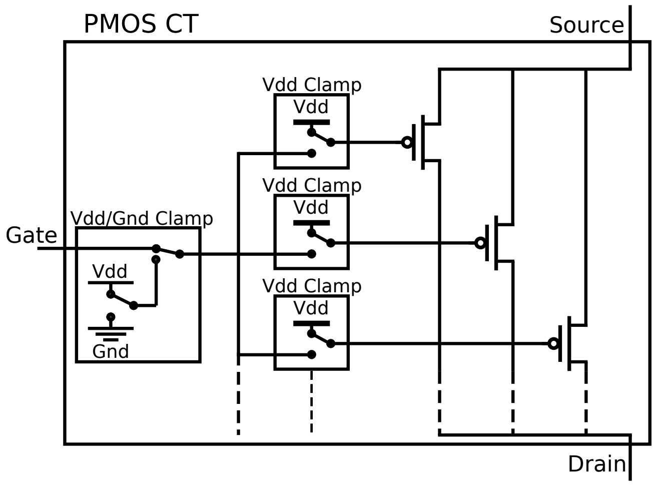


Figure 4.3: The structure of a PMOS flavour configurable transistor (CT) showing three of seven parallel transistors [71]. The "Vdd/Gnd" clamp controls whether the whole CT is enabled, insulating or conducting, and the "Vdd" clamps control the effective width by enabling or disabling individual transistors.

4.4 PAnDA Architecture

Traditional FPGAs have a number of levels in the hierarchy of configurable elements. Configurable routing connects CLBs, which in turn contain Slices made up of one or more Look-Up Tables (LUTs) and flip-flops. The PAnDA architecture has levels comparable to these, but with additional low level configurable layers.

4.4.1 Configurable Transistors

The lowest configurable level of PAnDA is the Configurable Transistor (CT). Consisting of seven parallel transistors and some transmission gates, a CT can be configured into a number of different states, which will be referred to in this thesis as:

- **Enabled** - behaving as an electronic switch, as used in traditional circuits.

- **Disabled-insulating** - not conducting, a permanent break in the circuit.
- **Disabled-conducting** - conducting, a permanent connection between two points in the circuit.

Figure 4.3 illustrates the basic implementation of a PAnDA-Eins PMOS CT here showing only three transistors, in the actual device there are seven. The seven parallel transistors, M0-M6, can be enabled in any combination, giving the effect of a transistor with a width of equal to the sum of the enabled transistors' widths. The widths of the seven transistors are 120 nm, 120 nm, 140 nm, 160 nm, 180 nm, 200 nm and 220 nm [72], combinations of which allow widths in the range 120 nm to 1140 nm.

All seven transistors have a fixed channel length of 40 nm. Since lengths cannot be combined in the same way as widths, separate sets would be required to provide the same functionality for configurable lengths. This implementation is used in Heidelberg's FPTA, but was out of scope for the PAnDA project for two reasons:

1. Hilder's findings [65] showed that optimising transistor widths alone could increase tolerance to variability.
2. Increasing the complexity of PAnDA's lowest layer would reduce space on the die for higher level structures.

4.4.2 Configurable Analogue Blocks

CTs are used to build two types of higher level structure, Combinatorial Configurable Analogue Blocks (CCABs) and Sequential Configurable Analogue Blocks (SCABs). Both types of CAB can be configured to act as digital logic gates, accepting inputs and generating outputs. CCABs can be configured to perform logic functions useful for combinatorial circuits whereas SCABS can be configured to perform logic functions useful for sequential circuits.

PAnDA-Eins uses sixteen CTs to create CCABs (see Figure 4.4). CCABs have three inputs and can be configured to act as one of eight logic functions. Table 4.1 lists the eight gates that CCABs can be configured as plus their inverted form, which is available through an inverted output. This brings the total number of available functions to sixteen. SCABs are made of twenty CTs (see Figure 4.5) and accept four inputs. Like CCABs, they can implement eight different functions, with an inverted output taking the total number of functions to sixteen (see Table 4.2).

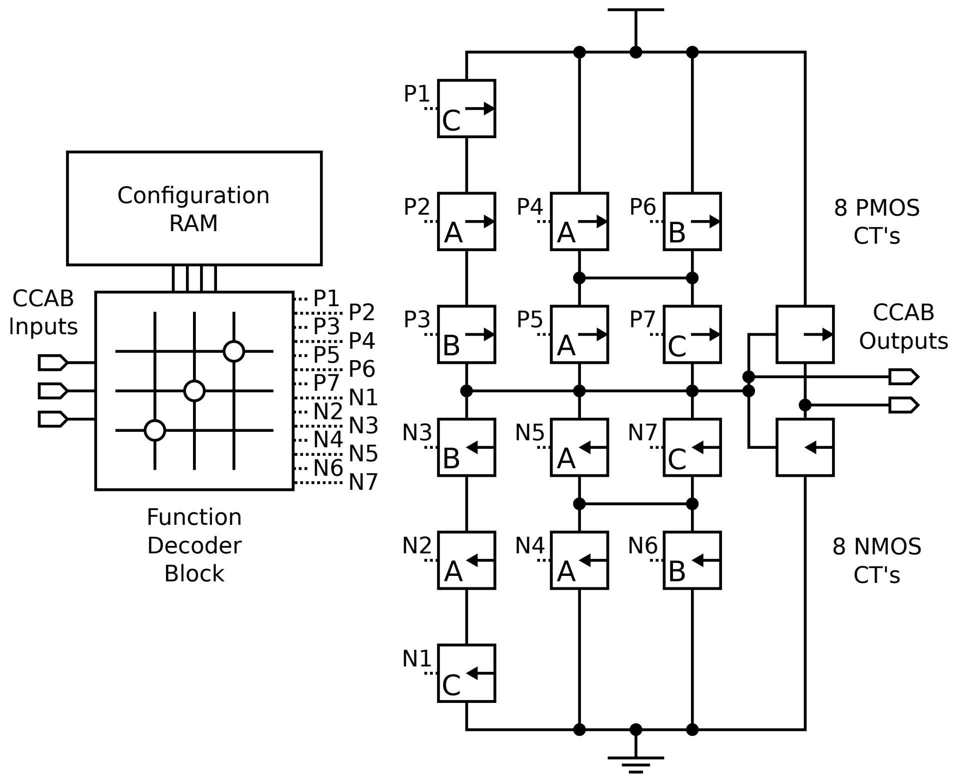


Figure 4.4: The structure of the PAnDA-Eins Combinatorial Configurable Analogue Block (CCAB) showing the layout of the PMOS and NMOS CTs, each consisting of seven transistors, and the Function Decoder Block (adapted from [72]). The letters A, B and C indicate which of the three functional CCAB inputs goes to the gate of each transistor. The rightmost PMOS and NMOS CTs (P8, N8) form an inverter to generate the complementary output. The Function Decoder Block takes three configuration bits as input (plus an enable bit) and outputs a clamp configuration for each of the CTs. This enables the user to configure one of eight functional configurations for the CCAB.

Configuration of a CAB's function is achieved through setting a 3-bit value in the configuration memory. This value is then decoded by a Function Decoder Block [73] into control signals for each of the CTs in the CCAB or SCAB.

Function Decoder Blocks

To simplify the configuration of the CABs, and the amount of configuration memory required, Function Decoder Blocks are used to convert 3-bit configuration values into predefined CT configurations. These are essentially large logic gates with a 4-bit input (three for the function and one enable bit) and either 12 or 10 outputs for CCABS and SCABS respectively [73].

The function decoder block simplifies the configuration of the CABs as only three bits are required to configure a function as opposed to each CT being configured independently.

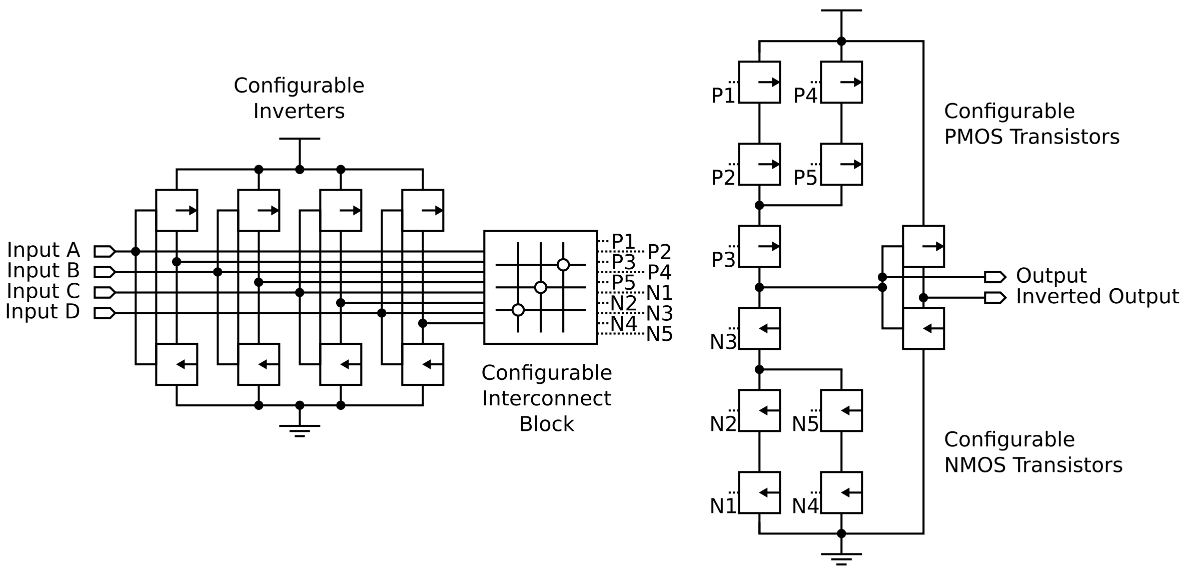


Figure 4.5: The structure of the PAnDA-Eins Sequential Configurable Analogue Block (SCAB) showing the layout of the PMOS and NMOS CTs (based on a figure from [74]). The letters A, B, C and D indicate which of the four SCAB inputs goes to the gate of each CT. The rightmost PMOS and NMOS CTs form an inverter to generate the complementary output. On the left, configurable inverters enable some functions to use inverted forms of the inputs in order to create some different functions.

Table 4.1: The 16 configurable functions of the PAnDA CCAB.

Configuration	Function (Standard Output)	Function (Inverted Output)
0	Inverter	Buffer
1	NAND-2	AND-2
2	NOR-2	OR-2
3	NAND-3	AND-3
4	NOR-3	OR-3
5	AND-OR-INV-21	AND-OR-21
6	OR-AND-INV-21	OR-AND-21
7	PROG-AND-OR-INV-2	PROG-AND-OR-2

Table 4.2: The 16 configurable functions of the PAnDA SCAB.

Configuration	Function (Standard Output)	Function (Inverted Output)
0	Inverter	Buffer
1	XNOR-2	XOR-2
2	XOR-2	XNOR-2
3	MUXI-2	MUX-2
4	INV-SW	INV-SW-INV
5	NOT-INV-SW	NOT-INV-SW-INV
6	CLK-MUX-2	CLK-MUX-2-INV
7	INV-x2	INV-X2-INV

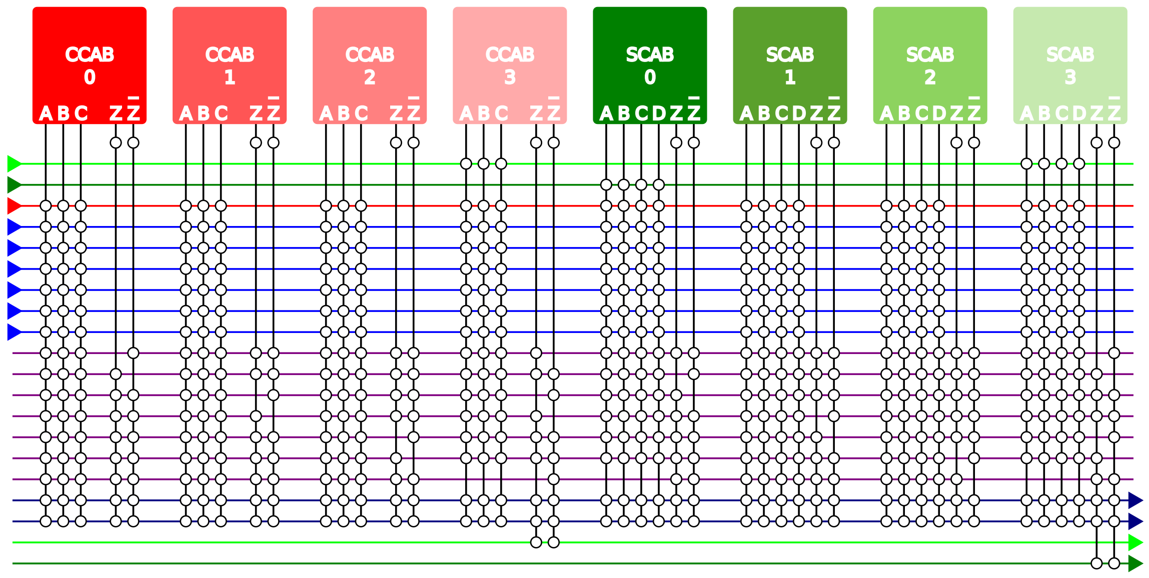


Figure 4.6: The routing inside a CLB (based on a figure from [28]). Each vertical wire can be configured to connect to one of the horizontal wires at the intersections with a circle. This enables CLB inputs (from the left) to be connected to the CCAB/SCAB inputs, the CCAB/SCAB outputs to be connected to other CCAB/SCAB inputs, and the CCAB/SCAB outputs to be connected to the CLB outputs.

4.4.3 Configurable Logic Blocks

Configurable Logic Blocks (CLBs) contains four CCABs, four SCABs and some configurable routing (see Figure 4.6). The routing can be used to connect the CLB's inputs to the CABs' inputs and the CABs' outputs to either the CLB outputs or the inputs of other CABs.

This enables small circuits to be built within a CLB, the outputs of which can be routed into another. CLBs are the highest level of configuration in PAnDA-Eins due to a fixed routing structure between them.

4.4.4 PAnDA-Eins Chip

Conceptually, the highest level of the PAnDA architecture is the chip level, containing an array of CLBs. The first iteration of the PAnDA architecture uses fixed routing between CLBs, as can be seen in the lowest layer of Figure 4.7. The routing is arranged in rows which allows for a number of functions to be performed on a set of inputs before they reach the output. There are also separate shift and carry lines which run perpendicular to the main routing to allow for interactions between the rows.

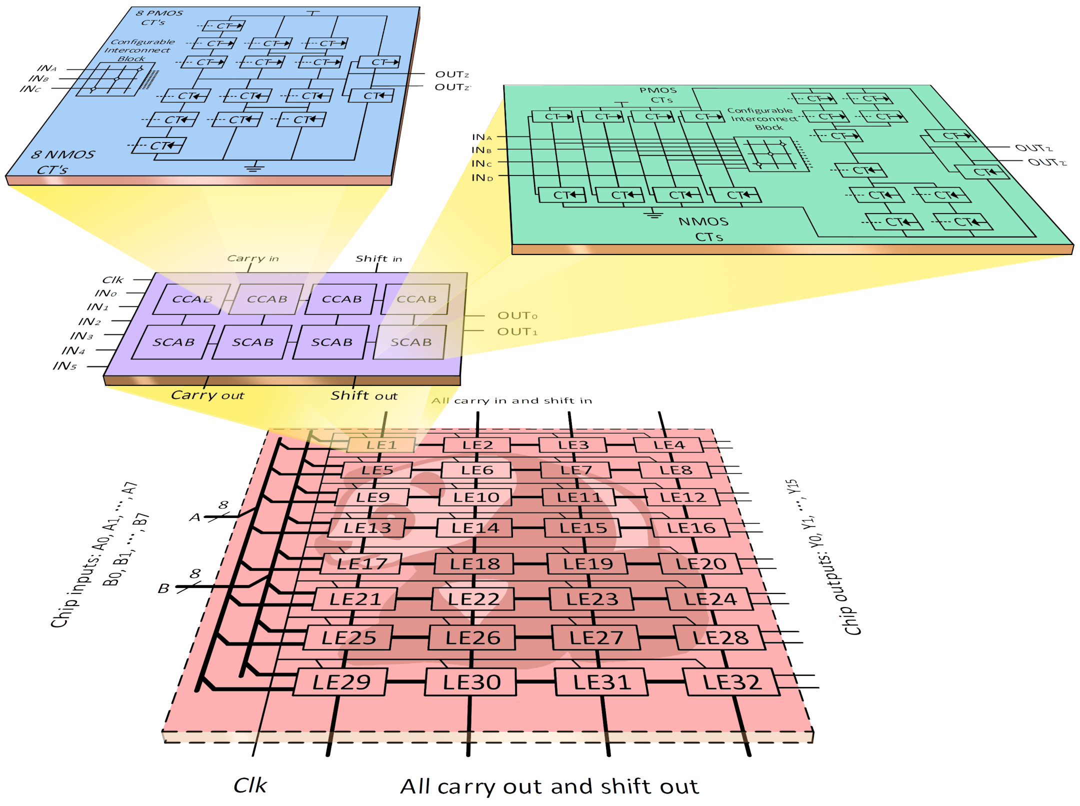

8 PMOS CT's
Configurable Interconnect Block
8 NMOS CT's
OUTZ
OUTZ'
PMOS CTs
NMOS CTs
Carry in
Shift in
Clk
CCAB
SCAB
OUT0
OUT1
Carry out
Shift out
All carry in and shift in
LE1 … LE32
Chip inputs: A0, A1, ···, A7 B0, B1, ···, B7
Chip outputs: Y0, Y1, ···, Y15
Clk
All carry out and shift out

Figure 4.7: Hierarchy of configurable structures in the PAnDA-Eins architecture [75]. The CLBs can be seen at the bottom of the diagram (labelled LE1 - LE32). Each CLB consists of eight CABs: four SCABs and four CCABs. Each SCAB and CCAB consists of a set of configurable transistors that allow various logic functions to be implemented (Tables 4.1 & 4.2). Each configurable transistor is build from seven different width transistors allowing configuration of the size of these transistors.

4.4.5 Differences to LUT-based FPGAs

Most common commercial FPGAs implement logic gates through Look-Up-Tables. The desired outputs of a function are stored in memory as 1-bit words and the inputs form an address lookup. This makes it possible to implement any logic function.

PAnDA-Eins is different in that logic gates are implemented in CMOS logic in the same or a similar way to how they would be implemented in ASIC. Given the motivations discussed in the previous section, this has the advantage of reacting to variability in ways comparable to ASIC circuits.

Figure 4.8 illustrates an example of a full adder circuit built using an PAnDA-Eins CLB. The logic gates configured in the CCABs and SCABs are made of CTs, each one containing seven transistors which can be individually enabled. This means that each CT of each logic gate

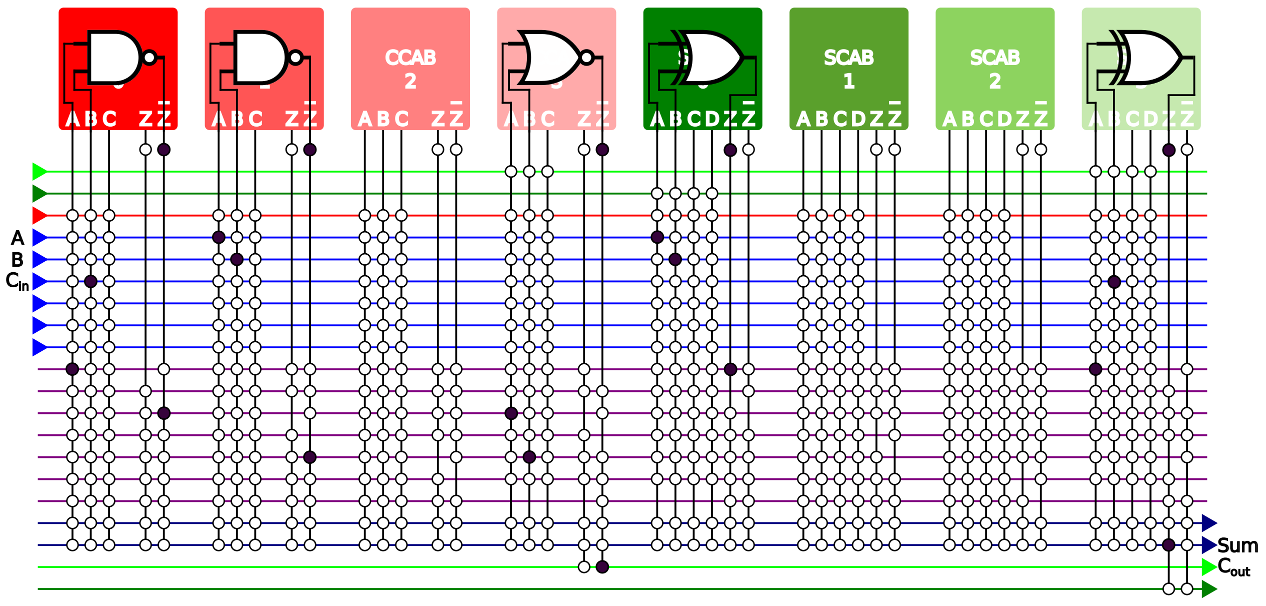


Figure 4.8: An example of a full adder implemented on a PAnDA-Eins CLB (based on a figure from [28]). Filled circles represent enabled connections.Tracing back from the Sum output, it can be seen that it is produced by the output of an XOR gate, which in turn takes its inputs from $A \oplus B$ and C_{in}. The inverted outputs of the CCABs are used to create AND and OR functions from the available NAND and NOR gate implementations.

can be reconfigured to give different behaviour at the analogue level, allowing for a wide range of parameter variations to be tested for this one circuit *in hardware*.

4.5 Summary

This chapter has introduced the motivations and design for a novel reconfigurable device, PAnDA-Eins, along with some details about the implementation. PAnDA-Eins architecture combines the strengths and flexibility of traditional FPGA devices with the transistor-level design exploration made possible by FPTAs. The combination of these should allow for experimentation with low-level parameters at a larger scale than is feasible with FPTA due to the higher-level structures that are built in.

Using function decoders to configure the CABs makes it simple to program them and ensures that any CAB configuration is valid. This simplicity comes at the cost of limiting the configurations that it is possible to access.

The next chapter describes an experiment that was carried out using the CCAB structure to find out whether removing the function decoder enables an EA to reconfigure the CTs to overcome a fault.

Chapter 5

Transistor Level Fault Tolerance

Contents

5.1 Introduction

Fault tolerance is a large area in the research community. With regard to electronic circuits, a number of techniques have been identified by the author which are thought to be relevant to the aim of the work reported in this thesis. Some details of these were given in Chapter 2. In this chapter the focus is on the specific area of evolvable hardware [30].

Evolutionary algorithms have been used in software since the 1950s [76] to optimise parameters and for use in the design of systems. Over the past 20 years, these techniques have been applied to hardware, in particular exploiting the opportunities presented by reconfigurable devices [77] and such techniques are particularly relevant to this thesis.

In separate work, Heidelberg and JPL simultaneously developed Field Programmable Transistor Array (FPTA) devices. These consisted of a transistor level reconfigurable fabric (more detail of the hardware device was given in Chapter 4). Work performed using these devices demonstrated that it is feasible to evolve circuits at the transistor level. The PAnDA architecture (as described in Chapter 4) shares some features with the FPTAs, such as the configurable transistors, so given the fixed routing between CTs (and thus a smaller design space) it is expected that evolving functions will be easier.

Logic functions on the PAnDA-Eins architecture (Chapter 4) are implemented by configuring CTs to one of three states: a switch, an insulator or a conductor. A function decoder block provides eight sets of these states for both CCABs and SCABs, giving sixteen possible functions between them. This simplifies configuration as the function decoders take a 3-bit value in order to configure a CAB as opposed to configuring each CT individually.

It was identified that the structure of the CCAB would allow for alternative implementations for some of its eight functions however the presence of the function decoder block makes it impossible to access these implementations as it prevents direct configuration of the CTs. It was hypothesized that in some cases, a fault affecting one of the predefined implementations could be worked around by using an alternative implementation of the same function.

The work in this chapter tests this hypothesis by simulating a fault that breaks a circuit and then uses an automated method to reconfigure and regain functionality. This work was carried out in collaboration with another student and was previously published in [71].

5.2 Removal of the Function Decoder

After the PAnDA-Eins architecture had been designed by other members of the research group, I investigated whether there was anything to be gained by removing the function decoder to allow direct configuration of the CTs. This work was conducted in close collaboration with another PhD student and I was the lead in aspects concerning the topic of this thesis such as defining the genetic representation and modelling the circuits and faults for the experiments. The work was published in [71].

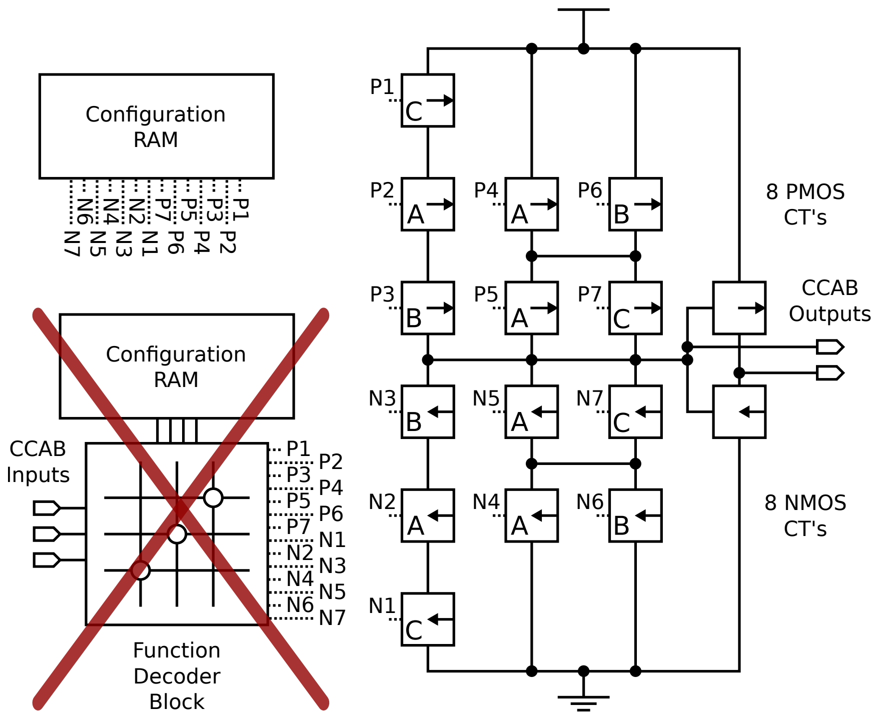
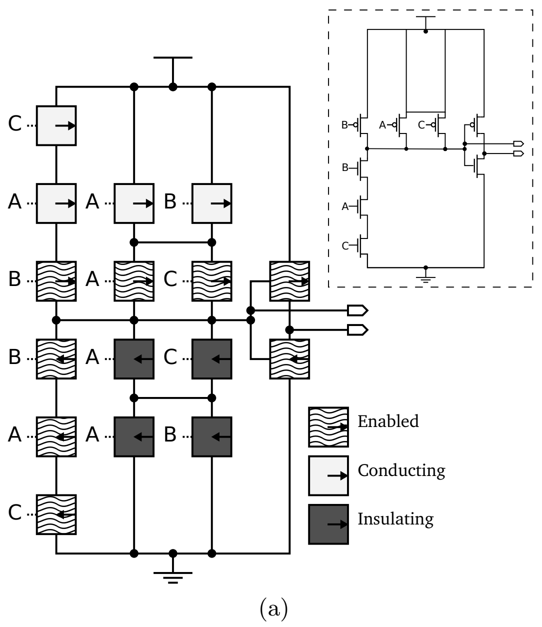


Figure 5.1: The function decoder block decodes three configuration bits (plus an enable bit) into a set of configurations for fourteen of the CTs in a CCAB (adapted from [72]). Removing it allows these CTs to be controlled directly and therefore for more functions or implementations of functions to be configured.

As described in Chapter 4 the PAnDA-Eins CCABs can be configured to one of eight functions based on a 3-bit value stored in the configuration RAM. This is achieved by decoding this value into configuration signals for 14 CCAB CTs, which consists of significantly more bits. This makes it easy to implement some commonly used standard cells, but impossible to configure anything else. It also limits the potential for reconfiguration in the event of a fault, when it may still be possible to implement a function in a different way using CTs that still work.

This chapter investigates what would be possible if the function decoder block were removed, instead giving the designer direct access to the CTs' configuration bits (see Figure 5.1).

Allowing direct configuration of the CTs enables the chip to enter states that would not have been possible with the function decoder block. This includes many useless configurations which will produce no output at all; potentially dangerous configurations that could short `VDD` and `GND` and alternative implementations of previously accessible functions. Figure 5.2 shows the standard PAnDA-Eins implementation of a 3-input NAND gate on a CCAB next to an alternative implementation which would only be possible if the Function Decoder Block were to be removed.



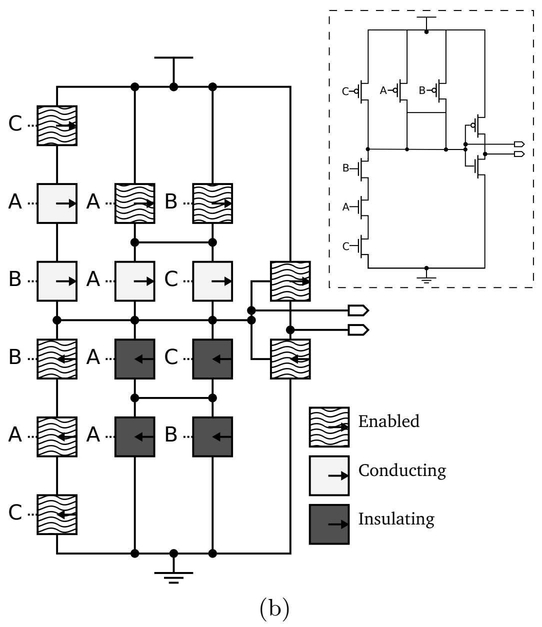


Figure 5.2: Figure (a) shows the implementation of a 3-input NAND gate offered by the PAnDA-Eins architecture. Figure (b) shows an alternative way to implement a 3-input NAND gate using the layout of CCAB which is functionally identical. In a standard PAnDA-Eins the presence of the Function Decoder prohibits this or other alternative implementations.

5.3 Hypothesis

Although it might be possible to build some analogue circuits with the PAnDA architecture, this work focuses solely on fault recovery in digital circuits. It does however consider some analogue aspects of the circuit, such as delay and power consumption, as secondary goals. The hypothesis investigated here is:

Hypothesis 2 *Removing the function decoder block from the PAnDA-Eins architecture provides benefits for fault tolerance.*

The next few sections describe experimental work that was carried out which attempted to fix faults and performance degradation of CCAB circuits in a way which would not be possible with the Function Decoder.

5.4 Experimental Method

The experiment chosen to investigate Hypothesis 2 attempted to use an automated approach to fix faults in a PAnDA-Eins CCAB. A SPICE netlist was constructed representing the CCAB circuit, with connections to the configuration memory replaced with voltage sources controlled by binary values. These binary values were to be controlled by a GA to provide the automation. We intended for the following sequence to take place:

1. The binary values would be used as genes for the GA.
2. The GA would evolve a specified logic gate by optimising the values of the genes to match an objective function.
3. A fault would then be introduced into the circuit by manually modifying the netlist.
4. The GA would again be used to evolve the logic gate with the now faulty CCAB circuit.

5.4.1 Tools

The CCAB structure was built as a SPICE netlist, with inputs where the configuration memory would normally connect controlled by binary values. The netlist was simulated using Ngspice [78]. A commercial implementation of NSGA-II was used to control the configuration inputs and read the outputs from Ngspice. 25 nm transistor models from GSS [79] were used for the simulations.

5.4.2 Fitness Function

The intention of this investigation was to not only recover logical functionality after a fault, but to restore the analogue performance of the circuit as well. For this to succeed multiple objectives for the fitness function were required. These are discussed in the following subsections.

Truth Table

The obvious choice for evolving a circuit to perform a digital combinatorial function is to measure the number of correct outputs against the desired truth table. This can be reversed into a minimisation problem by reporting the number of incorrect outputs, and so a value of 0 would mean that the circuit was performing the intended function.

Initial investigations found this approach to be insufficient for effectively evolving circuits in this setup. The reason for this is that unbalanced functions, such as NAND gates where all outputs are '1' except for when all inputs are '0', will score highly when all outputs are 1. There are many options for creating such as circuit, a conducting path simply has to be made from the `VDD` rail to the output.

Propagation Delay

The propagation delay is the measure between the input changing and the output changing to reflect the new state. We defined the start of this period as the point at which the input comes within 0.05 V of either 0 V or 1 V, and the end when the output comes within these levels.

In the experiment we calculated the mean delay over all of the transitions. One problem with this was deciding how to deal with input changes that don't result in a transition, or circuits that produced no transitions at all. If they were to report 0 ns this would skew the average calculation significantly and cause the GA to favour incorrect results. The solution to this problem was to insert a large delay where the measurement produced a 0 ns result. This way, incorrect circuits are punished while maintaining a measure of how good correct circuits were. Should any correct transitions be present, the circuit would still score better if these were faster than if they were slow.

Dynamic Power

Power usage was measured in order to be minimised by the optimisation. In switching logic circuits, dynamic power refers to the power dissipated during transitions as opposed to leakage dissipated when the input does not change. Including this in the objectives enabled the GA to find a balance between speed (propagation delay) and power. The alternative of including only one of the two would be that the GA would favour either fast circuits with transistors as large as possible that also use the most power or circuits with small transistors that dissipate little power but are slow.

RMS Error

To counter some of the issues with simply counting the number of incorrect truth table outputs, a measure of the RMS error versus the desired output was included. In the situation mentioned in the previous section with one output incorrect, the RMS error gives a smoother response as a circuit gets closer to the desired functionality.

One issue with this approach was that the transitions in our model output were unobtainable in reality. We had made them very quick so as to encourage a faster circuit. Unfortunately, this caused the GA to favour fast circuits more than anything else, so the RMS measurement was changed to calculate the error for regions of the output that should be stable, i.e. ignoring the transitions.

Number of CTs in Use

For evolving the initial circuit only, we included an additional objective which counted the number of CTs that were in the **enabled** state. This encouraged the algorithm to favour circuits that didn't include unnecessary CTs and produced much tidier individuals. This objective was removed from the fault recovery step as it was found to be too restrictive.

5.4.3 Test Circuits

The same experiment was performed on two different circuits: a 3-input NAND gate and an AOI21 gate. For both experiments the initial states, before evolving the circuits without any faults, were set the same. All the main functional CTs were set to the enabled state, with the PMOS CTs set to a width of 540 nm and NMOS CTs to 380 nm. The PMOS and NMOS CTs performing the output inversion were fixed at 380 nm and 240 nm respectively and were not evolved as part of the process because they do not affect the digital functionality.

For each circuit, a testbench was written with an input pattern tailored to the circuit's function. These were specifically designed to produce an alternating output so that there would be a transition on each change. This approach eliminated the possibility that a high scoring circuit would have any floating states as this would cause not cause the output to transition and so create large RMS error and propagation delay values.

NAND Gate

The first circuit that this experiment was attempted on was a 3-input NAND gate. This is one of the standard functions that the function decoder exposes, but only one implementation is accessible (see Figure 5.3).

AOI21 Gate

The same experiment was performed on using the AOI21 logic gate. The function decoder's implementation is seen in Figure 5.4.

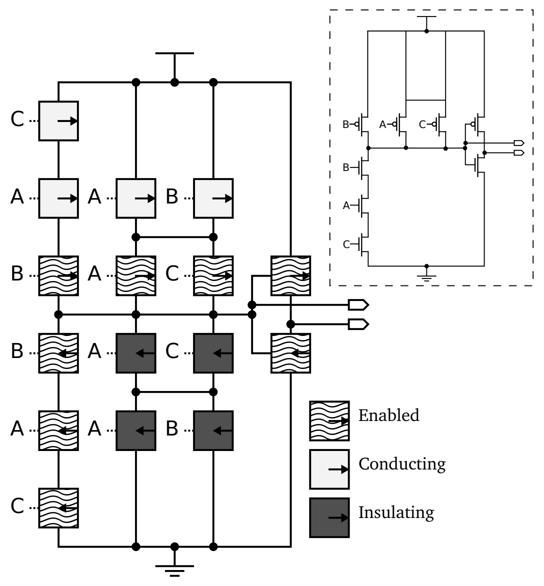


Figure 5.3: The original design for a 3-input NAND gate on PAnDA-Eins as implemented by the function decoder.

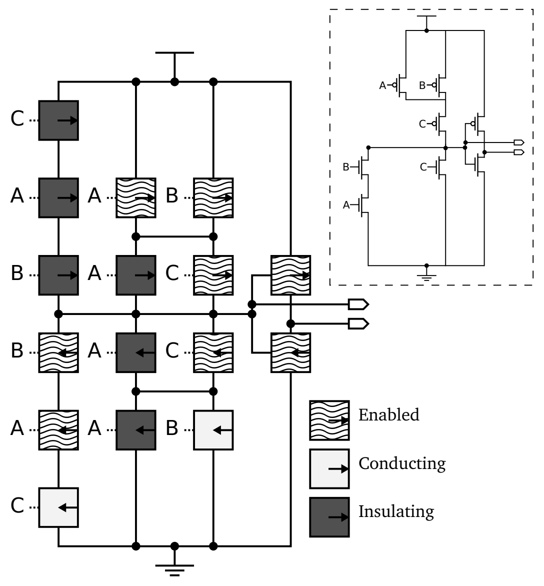


Figure 5.4: The original design for a AOI21 gate on PAnDA-Eins as implemented by the function decoder.

5.5 Results

For both circuit tested a version of the logic gate was evolved from the initial conditions stated in Section 5.4.3. An implementation of NSGA-II was used with a population of 60 and a mutation rate of 2% for 100 generations. From the set of results produced, individuals were hand picked from the middle of the delay/power pareto front. After adding faults to these which intentionally broke the circuits, they were evolved again. This time a population

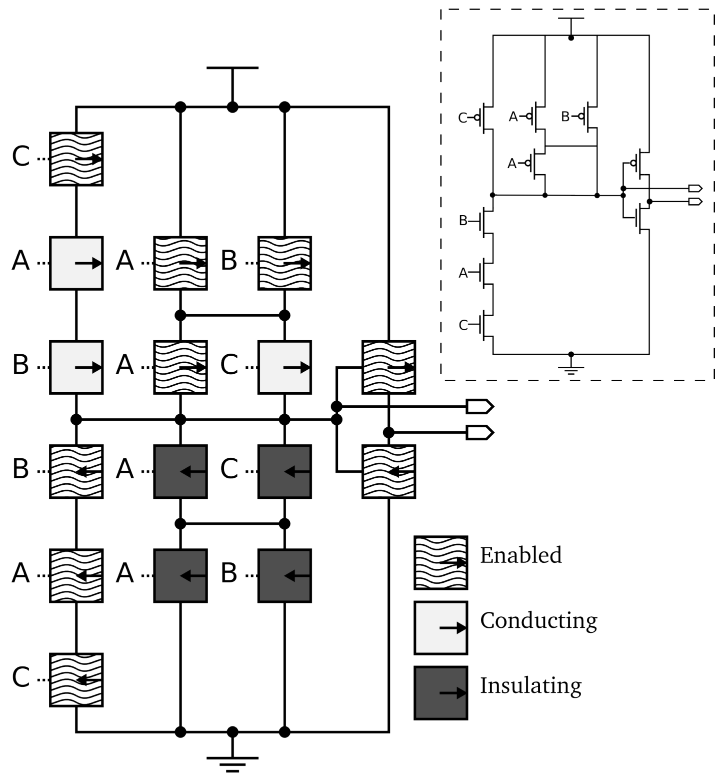


Figure 5.5: The evolved design of a 3-input NAND gate, with a simplification of the effective circuit on the top right corner.

of 116 was used (matching the number of available cores on the cluster being used) and 50 generations. Two individuals were picked from the final pareto front, one optimised for delay and the other for power.

5.5.1 3-input NAND Gate

The result of the initial evolutionary process is depicted in Figure 5.5. The GA has found a similar implementation to the function decoder's (Figure 5.3), with the NMOS side being exactly the same. On the PMOS side the GA has added a CT connected to the A input, which makes no functional difference but does increase the number of CTs that switch when the A input changes.

A fault was introduced into the netlist as shown in Figure 5.6, causing the PMOS side of the circuit to always conduct, giving an output of 1 V or a short-circuit when the input conditions caused the NMOS side of the circuit to conduct as well. The circuit was then re-evolved to negate the effect of this fault, the result of this being the circuit illustrated in Figure 5.7.

The behaviour of all three circuits (initially evolved, broken and fixed (optimised for speed)) is show in Figure 5.8. In Figure 5.8a the performance of the original evolved 3-input NAND

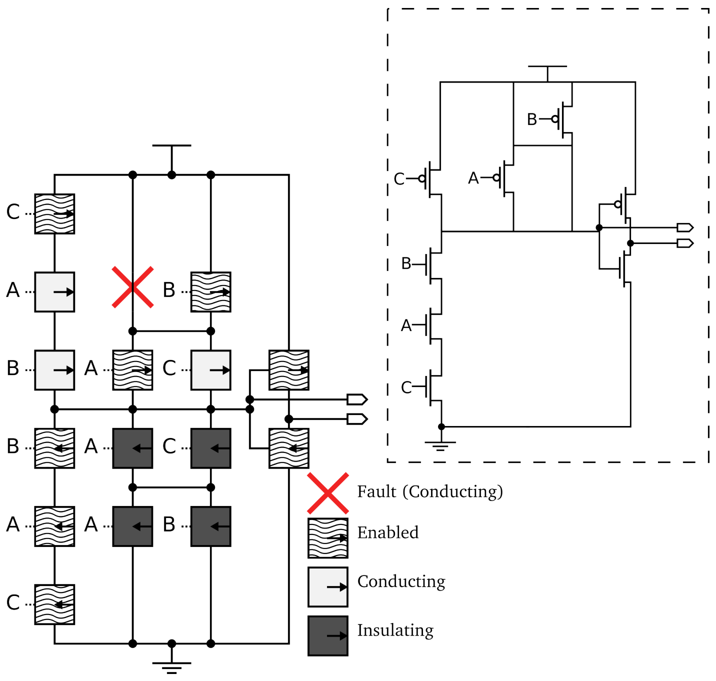


Figure 5.6: The fault was induced in a previously enabled transistor on the evolved 3-input NAND design, making it conduct all the time and disabling the gate's functionality. A simplification of the effective circuit can be seen in the top right corner.

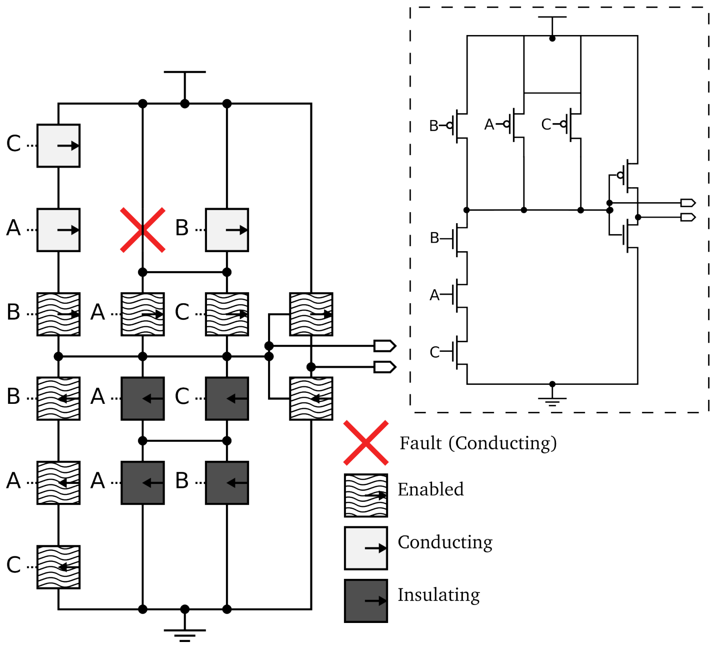
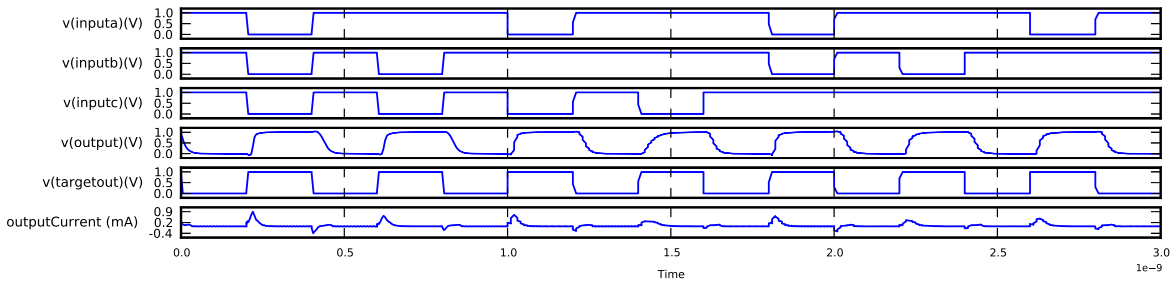
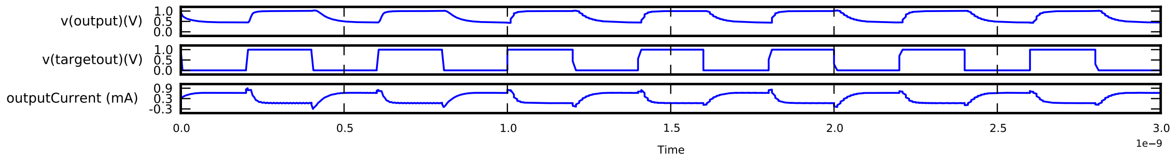


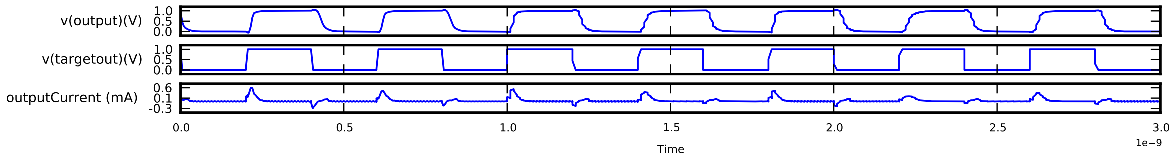
Figure 5.7: The 3-input NAND evolved on the faulty fabric and optimised for speed, with a representation of the effective circuit generated on the top right corner.



(a)



(b)



(c)

Figure 5.8: A simulation of (a) the evolved 3-input NAND gate on the CCAB with the waveforms of the 3 input signals – A, B and C –, (b) the faulty 3-input NAND gate, and (c) the recovered 3-input NAND gate, showing the actual and target outputs and output current.

circuit is shown. The output is close to the target output and the current measured across the circuit is generally low other than at the transitions. Comparing this to Figure 5.8b, it can be seen that the output voltage is not falling to 0 V when it should, and during these periods the current is high. This is due to the short-circuit that is occurring, and the circuit is acting as a potential divider. Finally, Figure 5.8c shows the circuit after functionality has been restored by the evolutionary algorithm's reconfiguration. The output once again is a good match for the target output and the current is generally low.

Table 5.1 details the circuits' performance. It includes two fixed circuits, one optimised for speed and the other for power.

The results clearly show the effect of the short-circuit fault as the power dissipation increases by approximately 10×. As the table shows, the circuit optimised for speed has an average delay value of around 5 ps less than the original design, though the power consumption has increased slightly. The circuit optimised for power has reduced the power consumption by around 2 µW but the average delay has increased. These results illustrate quite clearly the performance trade-off that must be made when designing electronics, where speed and power must be balanced appropriately.

Table 5.1: The widths of all the transistors of the initially evolved 3-input NAND gate, the faulty circuit, the one best optimised for speed, and the one best optimised for power. The last two rows describe their performance related to power consumption and propagation delay (average, per transistion). Transistor numbers relate to the CCAB schematic from Fig. 4.4 and enabled transistors have their widths written in bold.

Transistor ID	Evolved NAND3	Induced Fault	Evolved for speed	Evolved for power
P1	**1020**	**1020**	1140	1020
P2	1020	1020	1140	1020
P3	1020	1020	**720**	**300**
P4	**1140**	1140	1140	1140
P5	**620**	**620**	**840**	**380**
P6	1140	1140	**880**	**340**
P7	**1000**	**1000**	1000	1020
N1	**1140**	**1140**	**1140**	**860**
N2	**1140**	**1140**	**980**	**400**
N3	**1020**	**1020**	**840**	**320**
N4	340	340	660	0
N5	520	520	460	160
N6	280	280	520	480
N7	800	800	600	400
Avg Delay (ps)	50.764	N/A	45.451	61.780
Power(μW)	32.395	338.797	33.127	30.279
Enabled Transistors	7	6	6	6

5.5.2 AOI21 Gate

The AOI21 gate evolved on the non-faulty CCAB is shown in Figure 5.9. The design that the evolutionary algorithm has found matches the original function decoder version somewhat, but adds in some superfluous enabled CTs. The two **enabled** PMOS CTs on the left hand side of the diagram duplicate the functionality of the two on the right that are also connected to the B and C inputs. While this will increase power consumption, switching two transistors rather than one for inputs B and C, it will also increase the speed of the transition and perhaps created a better balance. On the NMOS side, the use of CT N4 as **disabled-conducting** rather than N7 is trivial and should make no discernible difference, but having the N7 in the **enabled** state is slightly wasteful as power must be dissipated when this switches despite its source and drain being shorted by N4.

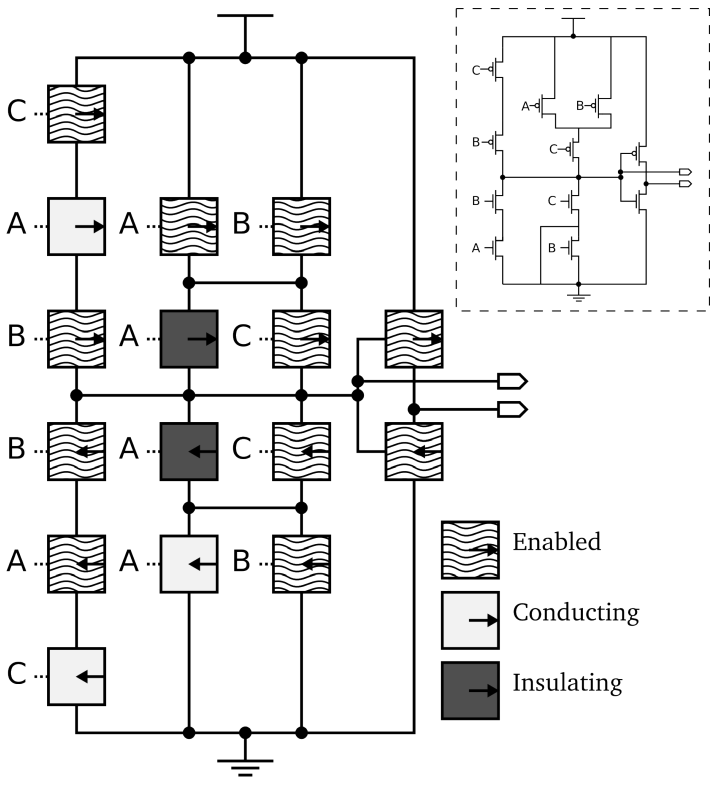


Figure 5.9: The evolved design of a 3-input AOI21 gate, with a simplification of the effective circuit in the top right corner.

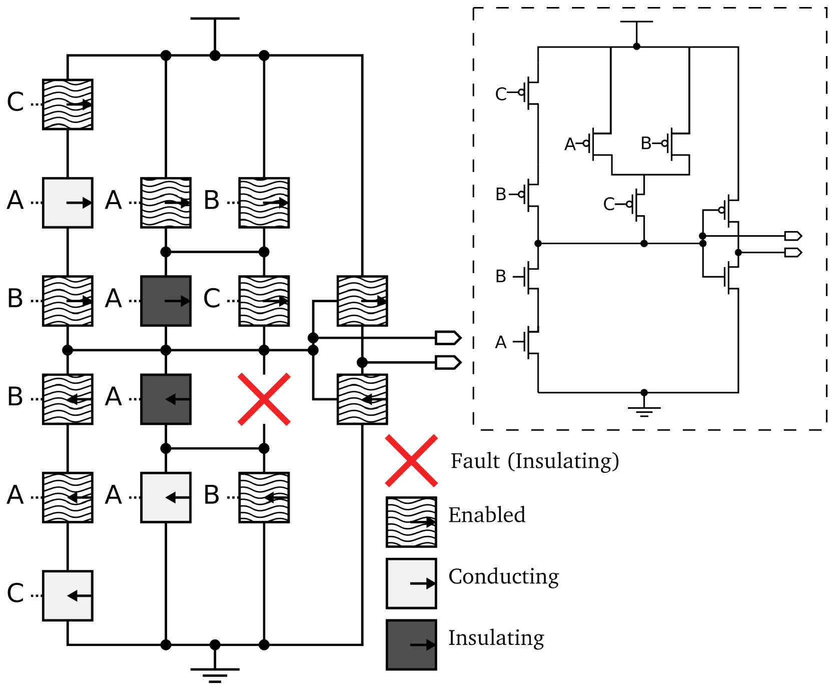


Figure 5.10: The fault was induced in a previously enabled transistor on the evolved AOI21 design, making it insulate all the time and disabling the gate's functionality. A simplification of the effective circuit can be seen on the top right corner.

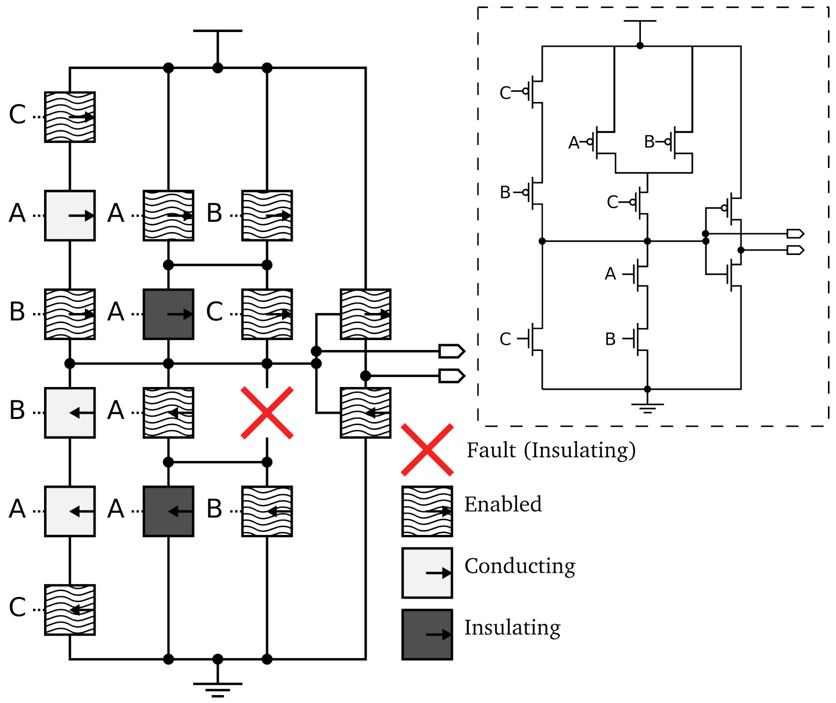
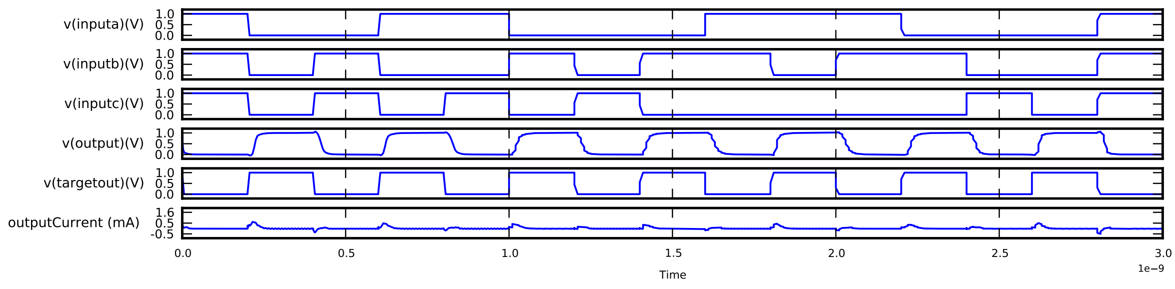
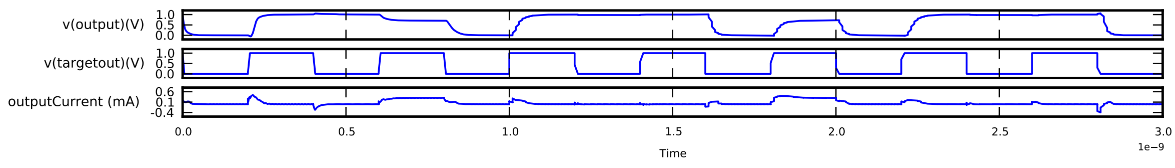


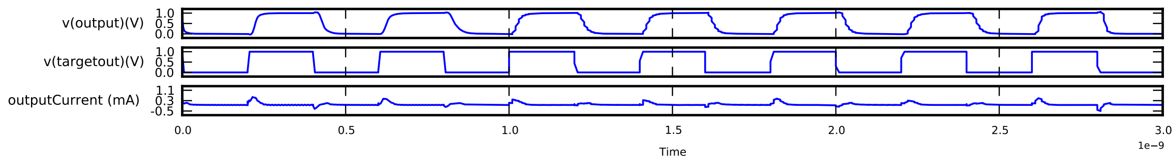
Figure 5.11: The 3-input AOI21 evolved on the faulty fabric and optimised for speed, with a representation of the effective circuit generated on the top right corner.



(a)



(b)



(c)

Figure 5.12: A simulation of (a) the evolved 3-input AOI21 gate on the CCAB with the waveforms of the 3 input signals – A, B and C –, (b) the faulty 3-input AOI21 gate, and (c) the recovered 3-input AOI21 gate, showing the actual and target outputs and output current.

Table 5.2: This table shows the widths of all the transistors of the initially evolved 3-input AOI21 gate, the faulty circuit, the one best optimised for speed, and the one best optimised for power. The last two rows describe their performance related to power consumption and propagation delay (average, per transition). Transistor numbers relate to the CCAB schematic from Fig. 4.4 and enabled transistors have their widths written in bold.

Transistor ID	Evolved AOI21	Induced Fault	Evolved for speed	Evolved for power
P1	**880**	**880**	**980**	340
P2	740	740	800	660
P3	**660**	**660**	**660**	640
P4	**1140**	**1140**	**1140**	**580**
P5	700	700	880	0
P6	**1020**	**1020**	**880**	**600**
P7	**1140**	**1140**	**840**	**680**
N1	1140	1140	860	520
N2	**1140**	**1140**	**920**	**260**
N3	**1020**	**1020**	1000	200
N4	1140	1140	0	0
N5	0	0	**460**	**380**
N6	**880**	880	880	880
N7	**1000**	**1000**	**1000**	**540**
Power(μW)	38.934	56.446	43.655	36.792
Avg Delay (ps)	41.812	N/A	50.369	76.058
Enabled Transistors	9	8	8	6

For the AOI21 circuit an insulating fault was introduced into the N6 CT (illustrated in Figure 5.10). This cuts off the branch which drives the output to 0 V when the C input is 1 V and so will cause a floating output in the truth table. Figure 5.11 illustrates the new evolved circuit in the presence of the fault.

Figure 5.12 illustrates the performance of the initial evolved AOI21 circuit (Figure 5.12a), the intentionally broken AOI21 circuit (Figure 5.12b) and the fixed AOI21 circuit, optimised for speed (Figure 5.12c). In the broken circuit, it can be see that the output does not change significantly on some occasions when it should be driven low. This is the result of the floating output caused by the induced fault - since the output is not driven (and no circuitry is attached to the output to drain the current), the voltage stays approximately the same. Figure 5.12c illustrates a return to expected functionality, similar to Figure 5.12a, with similar power performance, as illustrated by the output current.

The detailed results in Table 5.2 show a slightly different result to the other experiment. This time it was not possible for the evolutionary algorithm to find a solution with a delay lower than the original circuit, but a slower design that used less power was found. The savings possibly came from disabling the unnecessary PMOS CTs (P1 and P3).

5.6 Summary

The experiments described in this chapter confirmed that removing the function decoder block would allow the implementation of novel fault tolerant processes. The results show that it is possible to recover the same or similar performance from the logic gates after sustaining a permanent transistor fault, from which we learned alternative implementations of some CCAB functions are possible within the same structure. Other than an increase in configuration memory requirements, it seems that there is much to be gained from allowing direct control of the CTs.

The findings reported in this chapter had a direct impact on the new version of PAnDA designed by other members of the group. We had identified that the function decoder was a limitation of PAnDA-Eins in that it prohibited configuring working circuits and exploring alternate configurations that were useful. Without the function decoders the structure of the CCABs was unnecessarily limiting and it was realised that a simpler more regular structure would allow more functionality, such as other functions and multiple implementations of those functions.

The next chapter introduces the next iteration of the PAnDA architecture, PAnDA-Zwei, which builds on what was learned in this chapter and includes a more generic CAB structure with directly configurable CTs. The proceeding chapters use this new architecture to conduct experiments with a different novel fault tolerant methodology.

Chapter 6

PAnDA-Zwei

Contents

6.1 Introduction

PAnDA-Zwei is the second generation of the PAnDA architecture, designed by members of the Intelligent Systems Group at the University of York. Informed by the work in Chapter 5, the team made the CTs directly configurable (as opposed to being configured by the function decoder block in PAnDA-Eins) and the CCABs and SCABs have been replaced by MiniCABS, which offer less functionality individually but can be combined to provide more functionality than that available in the first version of PAnDA. This chapter describes this new architecture and discusses the differences between it and PAnDA-Eins, contemplating the potential benefits that the new features provide.

6.2 Architecture Features

The new architecture retains many of the concepts and ideas from PAnDA-Eins, but some things are implemented differently. This section describes the new features of PAnDA-Zwei from the bottom up, highlighting where things have changed.

6.2.1 Configurable Transistors

The CTs in PAnDA-Zwei are similar to the ones described in Section 4.4.1 in that they are constructed from parallel transistors which can be enabled and disabled. There are two key differences in PAnDA-Zwei CTs when compared with those of PAnDA-Eins:

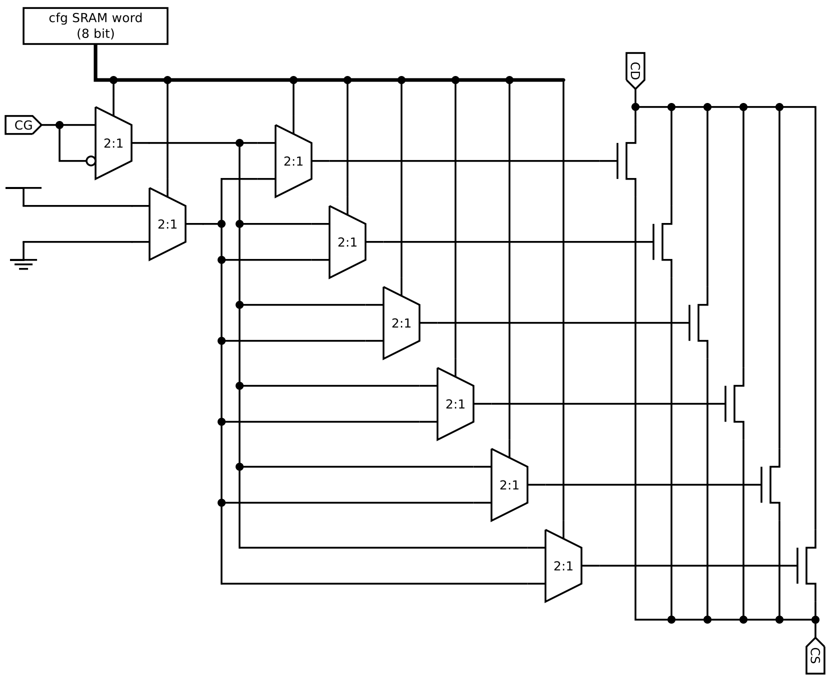


Figure 6.1: A PAnDA-Zwei NMOS CT with six transistors [70]. The gate input of each CT can be configurably inverted, similar to how PAnDA-Eins' SCABs operated.

1. The state of each transistor in the CT (**enabled** or **disabled**) and whether **disabled** means **disabled-conducting** or **disabled-insulating** for all transistors in the CT can be configured directly. In PAnDA-Eins the function decoder controls these settings, leaving only the ability to set individual transistors as **disabled-insulating** to control the CT's effective width.

2. The gate input can configurably be inverted, causing the transistor's switching behaviour to reverse (PMOS conducts on a 1 V, NMOS on 0 V). This new state will be referred to as **enabled-inverted**.

The first change was a design decision based on the findings of the work in Chapter 5. It was calculated that the number of transistors needed to implement the additional configuration memory required for direct control of the CTs was comparable to the number used to implement the function decoder blocks, therefore it was a simple decision for the team.

The second change was required to make it possible to replicate some of PAnDA-Eins' SCAB functions, which used inverters built from CTs. With an inverted Gate input, a PMOS CT will conduct when the input is 1 V and vice versa.

6.2.2 CT Branches

PAnDA-Zwei's CTs are arrange in 'branches' of four of the same type between either `VDD` (for PMOS types) or `GND` (for NMOS types) and the output (see Figure 6.2). This enables a branch of CTs to be configured to conduct given a certain input pattern and hence drive an output. One example of this would be if all the CTs in a branch were configured to be **disabled-conducting**, the branch would conduct for all inputs. If they were all **enabled**, the branch would only conduct when all the inputs were 0 V (for PMOS) or 1 V (for NMOS). The new **enabled-inverted** state for CTs enables branches of either kind to be configured to conduct on any product term with ≤ 4 variables in it.

6.2.3 Mini Configurable Analogue Blocks

The concept of CCABs and SCABs from PAnDA-Eins were combined in PAnDA-Zwei, creating a new kind of CAB called a MiniCAB. A MiniCAB contains two PMOS and two NMOS branches as shown in Figure 6.2 and is capable of implementing simple logic functions.

A logic function can be considered as two separate functions, one producing either a 1 V or floating output and the other 0 V or floating output. A MiniCAB is capable of implementing both simultaneously, where each function contains up to two minterms with up to four variables in its minimised form. An example of such a function is a two input NAND gate and is illustrated in Table 6.1.

Each minterm maps directly to a CT branch configuration. If a variable is present in the minterm, it means that an NMOS CT associated with that input is **enabled** or a PMOS CT

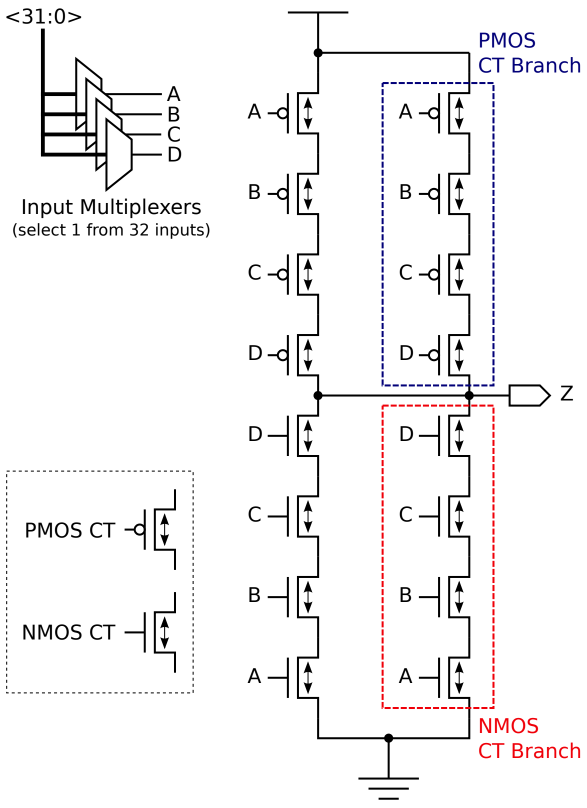


Figure 6.2: A MiniCAB with the two types of CT branches highlighted[70]. This structure allows for the implementation of simple logic gates but the output can also be chained to other MiniCAB outputs to create more complex gates. The Input Multiplexers allow for one of 32 inputs to be routed to each of the A, B, C and D CTs in each Branch.

is **enabled-inverted**. If a variable does not appear in a minterm, it means that the value does not matter (a “don’t care”) and so the CT associated with the missing variable should be configured to **disabled-conducting** so that it conducts for all inputs.

The inputs to each MiniCAB are also configurable at the MiniCAB level. Each of the four inputs (A, B, C and D), can be connected independently to one of 32 signals. This enables a MiniCAB to take any permutation of the inputs, so the same four inputs could be configured in 24 different orderings. Each of these 24 permutations can be used to implement the same MiniCAB functionality by also permuting the CT configurations to match the inputs since the AND operator in product terms is commutative. Furthermore the PMOS or NMOS branches in a MiniCAB can be swapped whilst maintaining functional equivalence, meaning there are $4 \times 24 = 96$ permutations for any given MiniCAB configuration.

Configuring a functional block at this level of granularity must be done carefully. If any overlap of the 0 V and 1 V functions exists (where they both conduct for some input pattern) this means that both a PMOS and an NMOS branch will try to drive the output, creating a short.

Table 6.1: These two tables represent (a) the minterms and (b) the minimised product terms of a 2-input NAND gate implemented with four inputs, as in a PAnDA-Zwei MiniCAB. A NAND gate outputs a logic high voltage unless all the inputs are high, in which case it outputs a logic low. To implement a 2-input gate where there are four inputs the minterms for all possible values of the unused inputs, C and D, need to be considered for each combination of A and B. The left-hand table lists all the minterms that exist for this function and the right-hand table lists the minimised product terms. The minimised form can be mapped directly to a MiniCAB.

(a)

1 V Outputs	0 V Outputs
A.!B.C.D	A.B.C.D
A.!B.C.!D	A.B.C.!D
A.!B.!C.D	A.B.!C.D
A.!B.!C.!D	A.B.!C.!D
!A.B.C.D	
!A.B.C.!D	
!A.B.!C.D	
!A.B.!C.!D	
!A.!B.C.D	
!A.!B.C.!D	
!A.!B.!C.D	
!A.!B.!C.!D	

(b)

1 V Outputs	0 V Outputs
A	A.B
B	

6.2.4 Slices

The Slices in PAnDA-Zwei are roughly comparable with the CLBs from PAnDA-Eins in that they consist of multiple CABs. A Slice contains four MiniCABs, takes 32 functional inputs and has four one bit outputs (see Figure 6.3). Each output can be driven by one or more of the MiniCABs, and so by connecting multiple MiniCABs to one output, more complex logic functions can be implemented. An example of this would be a four input NAND gate, compared to the two input version discussed in Section 6.2.3. Two MiniCABS are required for the minimized function because it contains four product terms for the 1 V output function, as shown in Table 6.2.

The implementation of the output switches uses transmission gates to combine the outputs as opposed to using digital logic and so care must be taken when connecting outputs together as a mistake in the configuration may result in the MiniCABs trying to drive the output voltage to opposing values and thus causing a short. In the case of the function described in Table 6.2 however, it is safe because only one type of branch will conduct for any given input value.

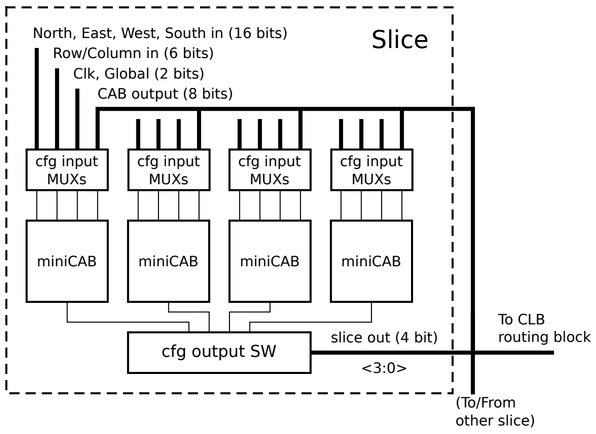


Figure 6.3: A PAnDA-Zwei Slice, showing the four MiniCABS, the input multiplexers and the configurable output switch. [70]

When connecting the outputs together in this way, and if the input multiplexers on both MiniCABs are set to the same set of inputs, we essentially create a larger MiniCAB. This means that CT Branch configurations could be swapped between connected MiniCABs whilst retaining functional equivalence. It would also be possible to connect two MiniCABs together with different sets of inputs selected to perhaps create > 4 input functions, although the possibilities of this had not been explored at the time of this thesis being written.

6.2.5 Configurable Logic Blocks

The Configurable Logic Blocks (CLBs) in PAnDA-Zwei are more comparable with components of the same name in commercial FPGAs than the CLBs in PAnDA-Eins. They contain two Slices and some routing to enable them to connect each other and to other Slices on the chip (see Figure 6.4). The work in this thesis will not focus on any of the implementation details higher than this level as they had not been finalised at the time of writing.

6.2.6 PAnDA-Zwei Chip

A PAnDA-Zwei chip consists of a number of connected CLBs as illustrated in Figure 6.5. The routing allows for 16-bits of data to be exchanged bi-directionally between neighbouring CLBs (four bits in each direction) and there are 3-bit row and column buses which can be connected to by either an input or an output from any of the CLBs on the same row or column.

PAnDA-Eins chips had fixed routing between CLBs which was a significant limitation when it came to implementing circuits (see Section 4.4.4). The configurable routing of PAnDA-Zwei allows for a much wider variety of circuits to be implemented.

Table 6.2: The minterms of a 4-input NAND gate using four variables. The left-hand table lists all the minterms that exist for this function and the right-hand table lists the minimised form.

1 V Outputs	0 V Outputs
A.B.C.!D	A.B.C.D
A.B.!C.D	
A.B.!C.!D	
A.!B.C.D	
A.!B.C.!D	
A.!B.!C.D	
A.!B.!C.!D	
!A.B.C.D	
!A.B.C.!D	
!A.B.!C.D	
!A.B.!C.!D	
!A.!B.C.D	
!A.!B.C.!D	
!A.!B.!C.D	
!A.!B.!C.!D	

1 V Outputs	0 V Outputs
A	A.B.C.D
B	
C	
D	

6.2.7 Summary of Improvements Over PAnDA-Eins

The architecture of PAnDA-Zwei improves upon that of PAnDA-Eins in a number of ways:

- The CTs are directly and independently configurable.
- The CTs can be configured to invert their gate input.
- Branches of CTs are homogeneous.
- All CABs have four inputs.
- Each of the four MiniCAB inputs can be configured to be driven by one of 32 possible signals.
- Routing between CLBs is configurable.

These changes allow for any 4-input function to be implemented on a PAnDA-Zwei Slice and for multiple different implementations of a function. The configurable routing between CLBs allows for more flexibility in building up larger circuits compared to the fixed routing between CLBs in PAnDA-Eins.

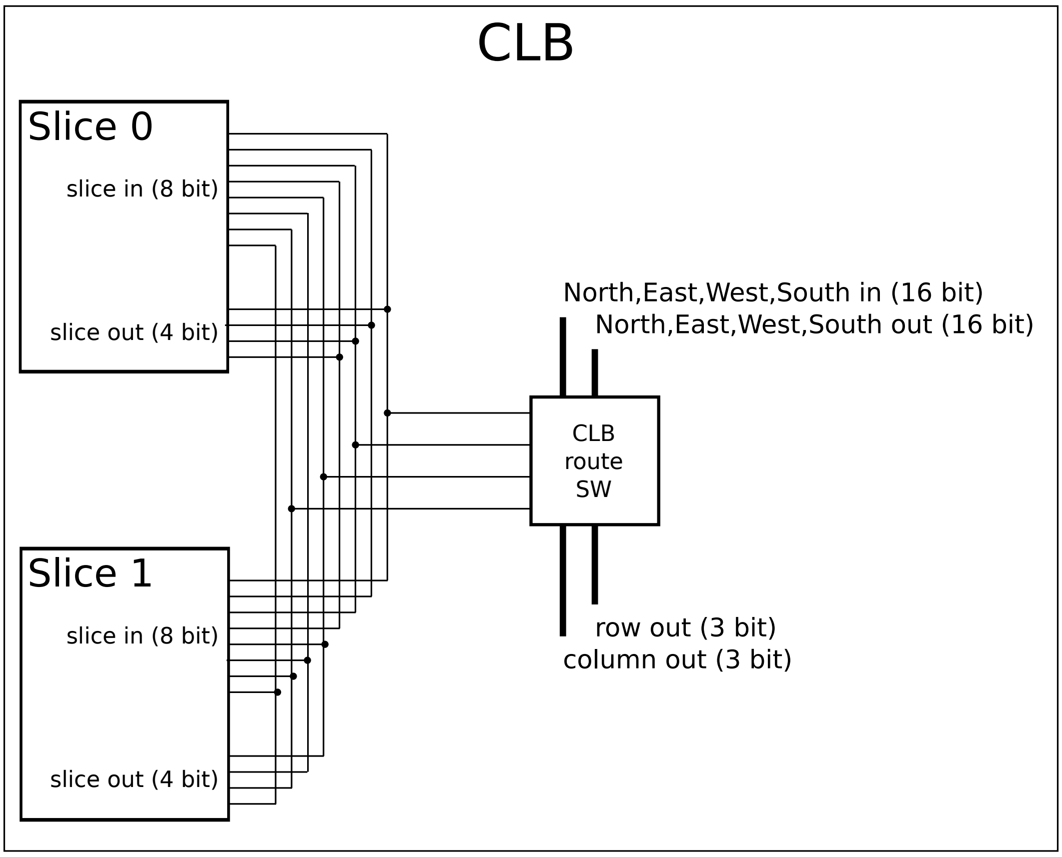


Figure 6.4: A PAnDA-Zwei CLB, showing the two Slices and the configurable routing switch. [70]

6.3 Comparison with FPGA

The PAnDA-Zwei architecture is comparable with commercial FPGA architectures. The main configurable parts of Xilinx and Altera FPGAs (other than routing) consist of an array of CLBs which consist of a number of Slices. Each FPGA Slice is a configurable logic component, built from SRAM Look-Up-Tables (LUTs), some storage components, such as flip-flops and possibly some multiplexers to allow the use of different combinations of these. The LUTs can be used to implement any logic function by setting the SRAM values to the desired output based on the address input. The flip-flops can be used as standard sequential components where required.

6.3.1 Implementation of Functions

The use of LUTs to implement logic gates in FPGA differs significantly from how gates are implemented in an ASIC. The output from an FPGA logic gate is controlled by an SRAM cell's stored value which gives the advantage of a consistent propagation delay for any logic

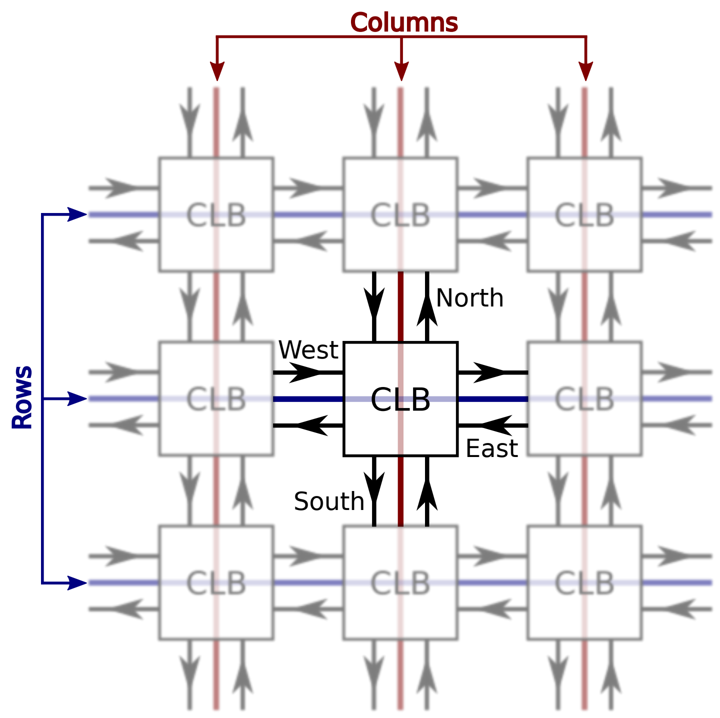


Figure 6.5: A number of CLBs connected together as they would be on a PAnDA-Zwei chip. The North, East, West, South (NEWS) routing allows a CLB to send and receive data with its immediate neighbours whereas each column and row has a bus which can be connected to by any CLBs on the same column or row.

function. In general this delay is higher than that of a standard cell in an ASIC circuit [80], which the logic gates built in PAnDA-Zwei are closer to in terms of implementation.

6.3.2 Granularity

From the top down to the Slice level, PAnDA-Zwei is conceptually very similar to FPGA. Additionally, Slices are comparable from a functional point of view as they can implement any 4-input function, just like a 4-input LUT, but the implementation differs significantly. Where LUTs are used for creating configurable logic gates in FPGA, PAnDA-Zwei implements functions by configuring MiniCABs and CTs. The use of CTs in PAnDA adds another layer of configuration to otherwise functionally compatible programmable arrays, as the CT widths can be altered as part of the design. A function can be made faster or to use less power, or even configured to have a longer rise time than fall time (for instance) on the output in PAnDA-Zwei because of these features. This extra level of configuration comes at the cost of additional silicon resources due to the transmission gates and SRAM cells required to implement it.

6.3.3 Tolerance to Faults

The LUTs used in FPGAs store the outputs for a function in SRAM whereas MiniCAB outputs are generated by real transistor logic circuits configured by SRAM. Given that SRAM is vulnerable to Single Event Upsets (SEUs)[81], where the stored value can be inverted, interesting comparisons can be made between the two styles or reconfigurable architecture within the context of fault tolerance.

If one value in a FPGA's LUT is flipped, this will lead to an incorrect value being output by a the LUT for a given input pattern, and propagated through a circuit (assuming no error correction). This could have serious consequences depending on the application, but should not cause damange to the chip itself.

A change in a single value of a PAnDA-Zwei MiniCAB's configuration could have a number of different effects. Each CT is configured by eight bits: one to configure whether the gate input is inverted, one to determine the behaviour of disabled transistors (conducting or insulating) and six to configure the size, enabling or disabling each of the parallel transistors to determine effective width. If the surrounding logic is ignored, this gives a 75% probability that if an SEU is sustained it will affect the configured width of a CT. In general, this will have little effect unless it disables the only transistor in use. If the SEU affects either of the other two bits, the effect would be more pronounced. In the worst case, the branch will start conducting when not intended and so will potentially cause a direct path between `VDD` and `GND`, consuming a lot of current and possibly causing damage to the chip. In a less severe scenario, the branch will not conduct when expected, causing a floating output for which the value read is not determinable.

Overall, if a LUT in an FPGA suffers an SEU there will be incorrect operation, whereas in PAnDA-Zwei there is a range of potential effects, the most likely being without functional impact but with the risk of chip destruction and incorrect operation.

6.4 Fault-Mitigation Opportunities

Chapter 5 discussed a method for overcoming faults on the PAnDA-Eins architecture. The technique was made possible by opening up direct configuration of the CTs which allowed some functions to be implemented in alternative ways. The PAnDA-Zwei architecture provides a great deal more opportunities for this by increasing homogeneity within the architecture, for example exchanging the CLB consisting of CCABs and SCABs for Slices consisting of homogeneous MiniCABs. All PMOS and NMOS CT branches are now the same, compared with the different structures present in PAnDA-Eins. The consequence of this is that functions and parts of functions can be built on any combination of branches and MiniCABs, and so this results in a much larger number of alternative implementations for any given function and thus potentially more opportunities for fault mitigation.

6.5 Potential for Short-Circuits

When creating any reconfigurable device the possibility of short circuits is always a concern. Commercial FPGAs rely on safeguards in the logic synthesis algorithms to prevent dangerous circuit configurations being generated, but this does not guarantee that it will never happen [82].

With finer-grained reconfigurable devices, there are often more opportunities for dangerous configurations. This was considered in the design of Heidelberg's FPTA which lead to precautions being implemented, such as a temperature sensor to shut down the chip should it get too hot. There is also a brief discussion in [63] suggesting that if `VDD` were connected to `GND` in the most direct way allowed by their architecture, the current would have to pass through at least one switch which would reduce its magnitude.

In PAnDA-Eins, it was impossible to implement dangerous configurations inside a CAB which would connect the `VDD` line to `GND` because of the restrictions imposed in the CAB's function decoder. It was however possible to configure the routing such that the outputs of two CABs were connected which does pose the risk of a short-circuit. There are situations such as creating certain sequential structures in which this would be desirable and so the ability to do it was not blocked. It was planned that a software component would verify configurations before programming the chip to minimise risks.

PAnDA-Zwei presents far more opportunities for short-circuits. It is quite straightforward to configure all the CTs in a MiniCAB to the **disabled-conducting** state, connecting `VDD` to `GND` as directly as possible, however the current would be subject to the resistance of the CTs. This resistance would decrease the more Branches that were fully conducting in this way and so the more Branches that are configured like this the higher the chance that damage will occur.

The work in this thesis assumes that potentially destructive configurations are not used intentionally and that, despite the possibility of bridging faults that may occur and cause shorts, the resistance of the CTs will be sufficient to prevent the Chip from self-destructing before the problem is fixed.

6.6 Summary

Following on from the design presented in Chapter 4 and the work in Chapter 5, this chapter has presented the next iteration of the PAnDA architecture, PAnDA-Zwei. Using the knowledge gained by experimenting with the PAnDA-Eins architecture without the function decoders the project team created a new design with more homogeneity and flexibility in its reconfigurability. The new features of the architecture allow for more straightforward implementations of more logic functions and the potential for novel approaches to fault tolerance, due to the large number of alternative implementations that an be achieved thanks to the homogeneity. These are good reasons for further experiments to be carried out on PAnDA-Zwei.

These new possibilities inspired an investigation into if and how they could best be exploited for the purposes of fault tolerance. The next chapter describes a simulator that was built to perform this investigation and is followed by two chapters detailing the experiments that were performed with it.

Chapter 7

PAnDA Slice Simulation

Contents

7.1 Introduction

In order to experiment with the PAnDA-Zwei architecture prior to chip fabrication a simulator was designed and built to evaluate function implementations in a Slice and also the effect of certain transistor faults.

The simulator is domain optimised for modelling four-input combinatorial logic gates implemented as they would be in PAnDA-Zwei and is capable of computing the effect of conducting (closed) and insulating (open) faults on the output of those gates. Sequential circuits with feedback loops were not covered directly since they would be built from smaller combinatorial components that are considered in this chapter.

Configuration circuitry is assumed to be fault-free and is not simulated. Prior work has addressed the issue of faults in FPGA configuration memory [17–20] and similar techniques could be applied in the context of PAnDA. Additionally, it is observed that errors in the configuration memory of transistor level configurable devices will manifest equivalently to faults in the transistors themselves.

The simulator was used for measuring the performance of algorithms described in subsequent chapters of this thesis. This chapter will describe the design and implementation of the software.

7.2 Motivation

The results from the experiments investigating evolutionary fault tolerance at the transistor level (Chapter 5) motivated research into whether more structures of the PAnDA architecture could be used in novel fault tolerance methodologies. Physical chips were not available at this point and so software simulation was necessary for experimentation.

Commonly, transistor level circuits are simulated using a SPICE simulator[78]. SPICE takes a netlist (a description of a circuit and any measurements to take) as input and produces an output file containing voltages, currents and other measurements. This is widely used across both research and industrial fields, but for a number of reasons it would be inconvenient to use for this work. The main reason for not using SPICE is that this work is considering many different levels of electronics - from the transistor level to higher level structures containing hundreds or thousands of transistors. Though this is quite possible using SPICE, it is very time consuming to create the netlists and the effort required to make changes to structures can be high. A second reason is the speed of simulation - large netlists can take many hours or days to simulate which limits what can be achieved on a reasonable timescale.

This motivated the building of a software model which could store, manipulate and simulate a representation of the PAnDA-Zwei architecture at the level of detail required for testing reconfiguration based fault tolerance methodologies. This includes the main functional

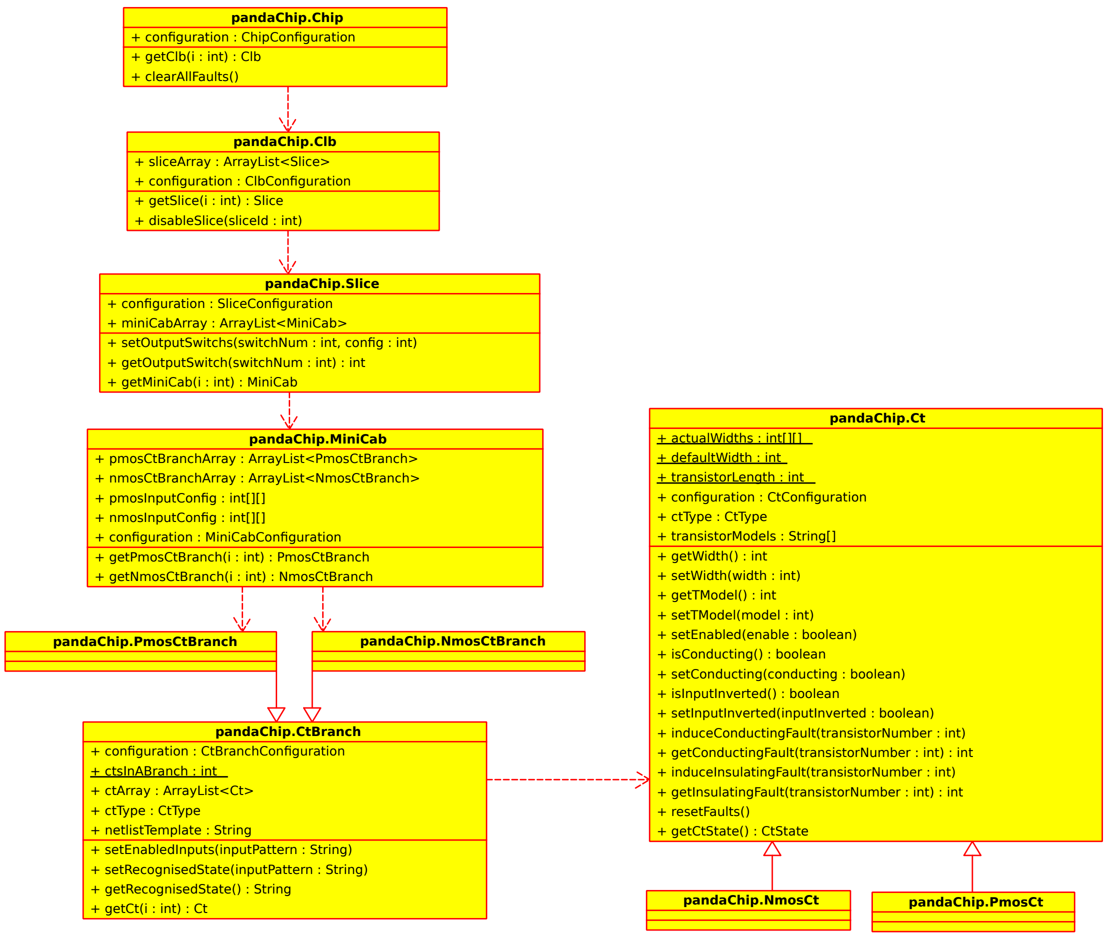

pandaChip.Chip
+ configuration : ChipConfiguration
+ getClb(i : int) : Clb
+ clearAllFaults()
pandaChip.Clb
+ sliceArray : ArrayList<Slice>
+ configuration : ClbConfiguration
+ getSlice(i : int) : Slice
+ disableSlice(sliceId : int)
pandaChip.Slice
+ configuration : SliceConfiguration
+ miniCabArray : ArrayList<MiniCab>
+ setOutputSwitchs(switchNum : int, config : int)
+ getOutputSwitch(switchNum : int) : int
+ getMiniCab(i : int) : MiniCab
pandaChip.MiniCab
+ pmosCtBranchArray : ArrayList<PmosCtBranch>
+ nmosCtBranchArray : ArrayList<NmosCtBranch>
+ pmosInputConfig : int[][]
+ nmosInputConfig : int[][]
+ configuration : MiniCabConfiguration
+ getPmosCtBranch(i : int) : PmosCtBranch
+ getNmosCtBranch(i : int) : NmosCtBranch
pandaChip.PmosCtBranch
pandaChip.NmosCtBranch
pandaChip.CtBranch
+ configuration : CtBranchConfiguration
+ ctsInABranch : int
+ ctArray : ArrayList<Ct>
+ ctType : CtType
+ netlistTemplate : String
+ setEnabledInputs(inputPattern : String)
+ setRecognisedState(inputPattern : String)
+ getRecognisedState() : String
+ getCt(i : int) : Ct
pandaChip.Ct
+ actualWidths : int[][]
+ defaultWidth : int
+ transistorLength : int
+ configuration : CtConfiguration
+ ctType : CtType
+ transistorModels : String[]
+ getWidth() : int
+ setWidth(width : int)
+ getTModel() : int
+ setTModel(model : int)
+ setEnabled(enable : boolean)
+ isConducting() : boolean
+ setConducting(conducting : boolean)
+ isInputInverted() : boolean
+ setInputInverted(inputInverted : boolean)
+ induceConductingFault(transistorNumber : int)
+ getConductingFault(transistorNumber : int) : int
+ induceInsulatingFault(transistorNumber : int)
+ getInsulatingFault(transistorNumber : int) : int
+ resetFaults()
+ getCtState() : CtState
pandaChip.NmosCt
pandaChip.PmosCt

Figure 7.1: A UML class diagram illustrating the main classes that implement the hierarchical model of PAnDA-Zwei.

elements down to and including the transistor width configuration, but does not include the configuration chain or most of the interconnect as this was beyond the scope of the planned experiments.

7.3 Virtual Hardware Implementation

The PAnDA model was implemented as a Java library which was used in a number of different projects. Using a heterogeneous tree structure, a hierarchy of classes represents each element of the architecture. The main functional classes of the model are presented using UML in Figure 7.1, with only the relevant methods for each shown.

Each element (CLB, Slice, MiniCAB, CT etc.) has an associated Configuration class, and each instance has its own configuration object instantiated. Configuration classes hold all the data required to define the behaviour of a particular element, just as the configuration memory would in a physical chip.

The separation of the configuration data from the architectural element enables modifications to the configuration without affecting data specific to a particular element, such as the presence of faults. This means that, as in the physical world, a configuration for a particular element may behave differently when applied to a different element of the same type due to the presence of faults, the data for which is held in the element object itself.

Access to each element is provided through getter functions. Starting with the Chip class, access to the contained Clbs is provided through the getClb(int clbNum) function and so on. This means that the Chip is stored as a tree and the elements can be traversed in a consistent manner.

7.3.1 CTs

PmosCt and NmosCt classes implement representations of PMOS and NMOS CTs respectively and extend the Ct class, each with a ctType variabled set to either CtType.PMOS or CtType.NMOS respectively. The functionality is contained within the Ct class and makes use of the ctType variable where behaviour should differ between the two.

The Ct class is perhaps the most significant as the configuration of the CTs is what creates the digital logic functions. For most of this work, the CTs are configured with just a few functions:

- `setConducting(boolean conducting)`
  Sets the default behaviour of non-enabled transistors to either conduct or insulate.

- `setInputInverted(boolean inputInverted)`
  Controls whether the gate input to the CT is inverted.

- `setWidth(int width)`
  Configures the width of the CT, where the least significant bits of the width integer corresponds one of the parallel transistors.

- `setEnabled(boolean enable)`
  Uses setWidth() to either enable or disable all the transistors in the CT as a shortcut to enabling or disabling it.

These settings are stored in an associated CtConfiguration object which separates the virtual hardware from the configuration. There are also functions for inducing transistor faults. Faults are stored in the Ct objects themselves as they affect the hardware, changing the behaviour of a particular CT instance and overriding affected CTs configuration. Two boolean arrays store whether each transistor in the Ct has sustained an insulating or conducting fault.

- `induceConductingFault(int transistorNum)`
  `induceInsulatingFault(int transistorNum)`
  Induces a conducting or an insulating fault on the transistor specified. An array in the Ct object stores the fault state of each transistor and overrides its configured behaviour during function calculation or netlist generation.

- `resetFaults()`
  Clears the fault arrays to return the CT to a fully working state.

The transistor model used when generating netlists can also be selected on a per CT basis, based on the SPICE models that have been included.

7.3.2 CT Branches

Branches of CTs are represented as PmosCtBranch and NmosCtBranch classes, which are both derived from the CtBranch class. CtBranches contain an array of CT objects (four by default).

CtBranches don't store any configuration themselves, but have a method for accessing the CTs contained within them, `getCt(int)`, and for configuring all the CTs at once, `setRecognisedState(String state)`. The `setRecognisedState(String state)` function is intended as a shortcut to configuring a "recognised state" for the branch, which means it will conduct when the input to the branch matches the given state. As an example, configuring a state of 1010 means that the branch will conduct when the inputs ABCD are set to 1010, driving the output of the MiniCAB to either Vdd or Ground depending on whether the branch is composed of PMOS or NMOS CTs respectively.

Branch states can be constructed such that they recognise more than one input. This is achieved by setting some CTs into the disabled-conducting state so that they conduct no matter what the input.

Configuring a CT branch with `setRecognisedState()` is achieved by creating a string where the length is equal to the number of CTs in a branch and the position of the characters represents the CT to be configured. The characters used to do this are:

- 0 - Recognises a low input. For NMOS CTs the input is inverted.
- 1 - Recognises a high input. For PMOS CTs the input is inverted.
- X - Recognises any input. The CT is put in the **disabled-conducting** state.
- Z - Recognises no input. The CT is put in the **disabled-insulating** state.

CtBranch objects also implement a `getRecognisedState()` function which returns a string in the format described above. The state is calculated from the configuration rather than simply returning what was set with `setRecognisedState()`. The calculation is performed by iterating through each CT in the branch and building up the string representation based on their configurations. Since this is essentially an estimate of the behaviour of an analogue circuit, the outputs produced are quite conservative such that unless the circuit's output is known, an error will be produced. The recognised states for each CT are calculated from the configuration using the following algorithm:

1. See if the CT contains any conducting faults or insulating faults. If all transistors have suffered from the same fault, the state is either "X" or "Z" respectively. If some have suffered from a fault, the state is reported as non-calculable.
2. If there are no faults, the width of the CT is checked. If it is above 0, the algorithm checks whether the CT is in the conducting state.
   (a) If it is, the recognised state is reported as "W" representing a "Weird" state. This means that if the width is less than the maximum, any disabled transistors will conduct while enabled ones will switch as normal, resulting in a CT that effectively always conducts but has a slightly different resistance depending on the input. This configuration is not considered by the work in this thesis as it creates an analogue behaviour.
   (b) If it is not in the conducting state, the CT is checked to see whether the input is inverted. If it is, the recognised state is reported as "1" for a PmosCt and "0" for an NmosCt. If the input is not inverted, the recognised state is reported as "0" for a PmosCt and "1" for an NmosCt.

7.3.3 MiniCab

The MiniCab class contains two PmosBranches and two NmosBranches by default. The only configurable element of the MiniCab class is the inputs, representing the input multiplexer part of the PAnDA-Zwei architecture.

The input configuration is stored as an array of integers in the MiniCab's configuration object with a length equal to the number MiniCAB inputs, which is the same as the number of CTs in a branch. The integers refer to which of the 32 potential inputs is mapped to each MiniCab input.

7.3.4 Slice

Slice classes contain an array of MiniCabs, four by default. As in the hardware, there are four configurable output switches which are each configured by an integer. The four least significant bits of these integers define which of the four MiniCABs are connected to the output.

7.3.5 Clb

Clb classes contain an array of Slices, two by default to match the hardware design. There is no additional configuration at the Clb level in the model developed for this work, although it could be extended to include the routing block to configure inter-Clb connections.

7.3.6 Chip

The highest level in the main model is the Chip class. There are no configurable options for the Chip class other than a variable that controls how many Clbs are instantiated. This allows the simulator to be scaled for larger versions on the chip at a later date.

7.4 GUI

For testing purposes and to provide a simple to use interface for the model, a GUI (see Appendix A) was created which allows the user to either program a Slice with a function, or edit the configuration of each element manually and then to simulate the configuration.

Each instantiation of each layer of the hierarchy is represented with a configuration screen which the user can interact with to configure a virtual PAnDA-Zwei. It is similar in concept to the GUI developed for Heidelberg's FPTA [62]. It could be used to gain an understanding of the architecture; how it works and the effect of each parameter, but would not be practical for implementing larger circuits due to the time it would take to configure them and the potential for user error.

7.5 Circuit Configuration

For the purposes of experimentation with the model it was required that automatic circuit configuration be possible. Once the simulation library had been developed, a configuration library was written to generate and implement automatically generated function configurations.

Given the definition of a 4-input function, it is possible to generate a circuit configuration that will implement it. This configuration can then be used to configure an instance of the PAnDA model using the functions exposed by the configurable components. The configuration library is able to do all of this automatically.

Configuring logic functions on PAnDA is different from the traditional Look-Up-Table (LUT) based FPGAs. A LUT is simply a memory, usually with a short word length such as 1-bit. The inputs to the LUT are essentially the address bus, and the output is whatever it stored at the addressed location. By storing a set of values in a LUT, it is possible to emulate any logic gate.

PAnDA works differently and gates are implemented by configuring structures of transistors, similar to how circuits are built in ASIC. In addition to binary logic, tri-state functions can be constructed as well as (generally undesirable) shorting functions, that will connect Vdd to Ground with some specific set of inputs.

7.5.1 Synthesizing PAnDA Logic Gates

PAnDA logic gates are implemented in a manner similar to the ASIC implementation using complementary PMOS and NMOS transistors to drive an output. As in most digital CMOS logic circuits, a gate in PAnDA consists of one or more inputs connected to transistor gates and an output which can be driven by either the `VDD` rail (for a '1' or 'high' output) or the `GND` rail (for a '0' or 'low' output). In order to perform a logic function on the structures inside a PAnDA Slice and MiniCAB, the input patterns that create a '1' output must make one or more branches of PMOS CTs conduct, so that the output is driven by the `VDD` rail, and vice-versa with regards to the '0' outputs and the NMOS CTs.

Any binary function can be described by a boolean expression. For n variables there are 2^n possible combinations of binary values that those variables can take. Each one of these combinations can be described by an expression or minterm. By selecting all the minterms associated with the variable combinations for which a function should output a '1' and summing them together (with an OR operation), an expression for the logic function can be found. This initial mapping can often then be minimized in order to reduce the number of minterms and the number of variables in each minterm. This is commonly achieved with the Quine-Mcklusky algorithm [83] in circuit design at the next abstraction layer up, building logic functions out of standard AND and OR gates.

An n-input binary logic function can be defined by a string of 2^n bits. Each bit of this string represents the function's output based on the input pattern equal to the bit's position in the string. Taking the function in Table 7.1 as an example, the string defining it is 1101100010100100 (the output column of the truthtable transposed). Table 7.2 shows the input states split into two groups, those that produce a 0 output and those that produce a 1. In this thesis, these will be referred to as 0-states and 1-states respectively.

As is common in CMOS design, PAnDA MiniCABs have PMOS transistors between the Vdd rail and the output and NMOS transistors between the Ground rail and the output. In order to make the logic gate output a '1' signal, a branch of PMOS CTs must conduct between the Vdd rail and the output. Similarly, for a logic '0' on the output, a branch of NMOS CTs must conduct between the Ground rail and the output. For this reason, the 1-States from Table 7.2 can also be thought of as "PMOS-States" as they will need to be recognised by (as defined in Section 7.3.2) a branch of PMOS CTs. Again, the same applies for the 0-States and branches of NMOS CTs.

At this point, a circuit could be built by implementing an NMOS or PMOS branch for each of the 0 and 1-States respectively, but it is usually possible to reduce the number of branches required in order to use resources more efficiently. This is often necessary as any unbalanced function will have more than eight 0 or 1-States and so will not fit into the eight Branches of a PAnDA-Zwei Slice.

Reducing the number of required branches essentially works by merging branches with common factors. The 1-states 0000 and 0001 differ by only one input which means that input D essentially doesn't matter. If these two states were implemented as branches, changing the configuration of the PMOS CTs connected to input D to be **disabled-conducting** would have no effect, they would now both conduct for both states, rather than one or the other. This means that one is now redundant and can be removed.

This process breaks down into two instances of boolean minimisation. If all the 1-states are written as minterms (e.g. !A!B!C!D, !A!B!CD, !A!BCD...), then a sum of products containing all of these represents the function required of the PMOS branches, i.e. whenever this sum of products is true, one or more PMOS branches must conduct. The same is true of the 0-states and the NMOS branches. Each product term can then be directly translated into a branch configuration using the following rules:

1. Any variables present in the product term represent **enabled** CTs.
2. Non-inverted variables map to **enabled** NMOS CTs and **enabled-inverted** PMOS CTs.
3. Inverted variables map to **enabled** PMOS CTs and **enabled-inverted** NMOS CTs.
4. Variables missing after minimisation map to **disabled-conducting** CTs.

Looking at the problem this way enables the use of standard boolean minimisation algorithms such as Quine-Mcklusky to minimise the expression and hence minimise the use of branches and CTs required. In addition to standard boolean functions, this method is capable of mapping tri-state and voltage divider circuits. For a tri-state function, the minterm(s) for any desired high-impedance states are not added to either the 0-states or 1-states lists. When the PMOS branch and NMOS branch functions are created, neither will conduct on the input(s) not included and so the output will be floating. Similarly, if desired, a minterm could be added to both lists and one of each type of branch could be made to conduct on that input. This is generally undesirable and could cause damage to the chip due to drawing large currents through the CTs.

Through minimising all 2^{16} boolean 4-input functions, it was found that all of them could be minimised into 8 product terms of each type or less, meaning that any 4-input boolean function can be implemented on a PAnDA-Zwei slice. In addition, there are only two functions that don't minimise at all: odd and even 4-parity functions, which are inverses of each other.

Table 7.1: Truthtable for the logic function used for the example in Section 7.5.1.

ABCD	Y
0000	1
0001	1
0010	0
0011	1
0100	1
0101	0
0110	0
0111	0
1000	1
1001	0
1010	1
1011	0
1100	0
1101	1
1110	0
1111	0

Table 7.2: The input states from Table 7.1 split into two groups based on the output.

0-States	1-States
0010	0000
0101	0001
0101	0011
0110	0100
0111	1000
1001	1010
1011	1101
1100	
1110	
1111	

7.6 Faults

The simulator was required to model the effect of faults on the circuits it was simulating. Similar to the work in Chapter 5, two types of fault were implemented: conducting (closed-circuit) and insulating (open-circuit). These two types represent extreme situations that can occur with faulty transistors but do not cover the whole range of effects that faults can cause such as bridging faults between arbitrary circuit nodes.

Faults in the configuration and routing logic were deemed out of scope for the simulator since the work was mainly focusing on gate level circuits in PAnDA. Configuration and routing can be implemented in different ways, but the novel aspect in PAnDA is the implementation of the logic gates. It is worth noting however that some faults in the configuration logic, such as stuck-at faults in the memory, will cause problems equivalent to the simple insulating or conducting fault model.

7.6.1 Injection

A Fault Tolerance package that works with the architecture model enables transistor faults to be injected into Chip instances. A base method, `injectTransistorFault()` injects a fault of a given type (conducting or insulating) into a specific transistor of a CT. Helper methods make use of this to provide more functionality, such as injecting a fault into a whole CT (the same fault into all transistors of a CT) and also injecting faults of a random type into a random CT of a Slice or Chip.

7.6.2 Simulation

The effect of a conducting fault is to make a transistor permanently conduct between source and drain, regardless of its gate voltage. When generating a SPICE netlist, this is implemented by adding a 0Ω resistor between source and drain. When deriving the function analytically as described in Section 7.7.2, the transistor will report the “X” recognising state regardless of its configuration (see Section 7.3.2).

Insulating faults are implemented in a similar fashion, except that in the netlist the gate node will be connected to either Vdd or Ground instead of the input multiplexer, depending on whether it is a PMOS or NMOS transistor respectively, causing the transistor to insulate permanently. When deriving a function, the transistor will report the “Z” recognising state regardless of its configuration (see Section 7.3.2).

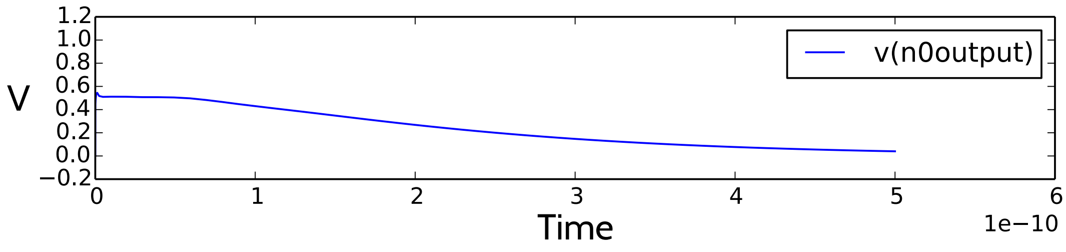
7.7 Simulation

Configurations held in the model can be simulated in two ways: SPICE simulation and analytically deriving the function. SPICE simulation is slower but gives more detail, such as voltages and propagation delays, whereas deriving the function is very quick but gives only digital information. However, this is adequate for testing the functionality of a circuit.

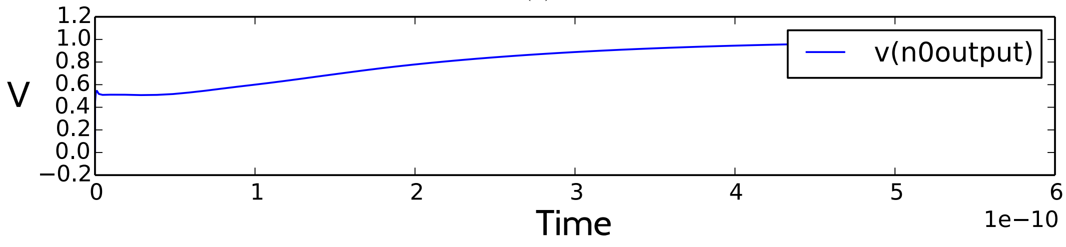
7.7.1 SPICE Simulation

For SPICE simulation, a netlist is generated using a Depth-First Search traversal of the PAnDA model tree. Each element implements a `getNetlist()` function which returns a netlist of its children and itself. This works by calling the same function in all child elements which do the same until the CT objects, which return a filled in netlist template. The software then simulates the netlist using ngspice and processes the results to get the truthtable.

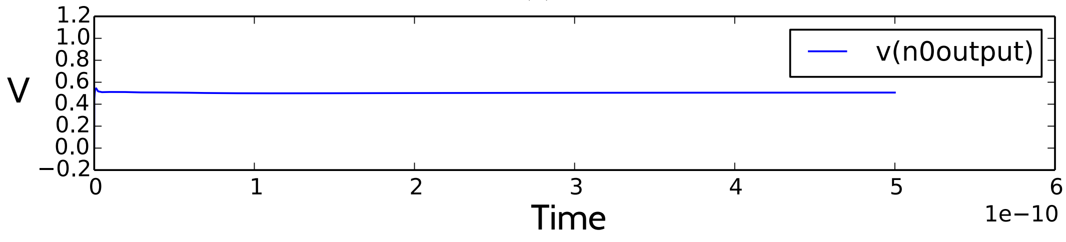
Originally this was to be the main purpose of the software, having an easy way of configuring and generating netlists for PAnDA-Zwei, but full-analogue simulation was found to be unnecessary for the purposes of the work and limited what was possible due to the simulation



(a)



(b)



(c)

Figure 7.2: The three expected outcomes from a SPICE netlist created by the PAnDA simulation tool. The SPICE simulation is started at 0.5V and will either be driven to (a) near 0V, (b) near 1V or (c) stay approximately where it is due to it floating (not being driven) or being driven both ways (a voltage divider).

time. The ability to produce and run the netlists however enabled verification of the model and subsequent faster simulation method.

To provide a reliable method of measuring the function performed by a Slice configuration, a SPICE simulation is run for each line of a 4-input truthtable. In the netlist testbench, the initial voltage of the output is set to 0.5V and the final voltage is measured after a fixed period of time. This makes it possible to distinguish between three possible situations: the output is driven from 0.5V to 1V, to 0V or stays the same. This leads to one of three conclusions:

1. <0.05V - Logic 0.
2. >0.95V - Logic 1.
3. Between 0.05V and 0.95V - High Impedance/Short

Figure 7.2 illustrates these three cases. Case 3 is ambiguous as it could be caused by either a high impedance state, where the output is not driven, or a short, where the output is driven by both a PMOS branch and an NMOS branch. Measuring the current was considered as a method of differentiating the two, but the difference in these cases was found to be too small to provide a reliable result.

SPICE simulation is too slow for running large experiments with many simulations. This motivated the building of a faster approach which would return the required information without running a full analogue simulation.

7.7.2 Deriving the Implemented Function

Deriving the function takes a very different approach and is the mode used for much of the work as the speed of running is orders of magnitudes faster than SPICE, enabling larger more complex experiments to become feasible.

The algorithm used for deriving the implemented function uses the `getRecognisedState()` function (described in Section 7.3.2) on each branch connected to an output to find out whether any will conduct for a given input pattern. For instance, the reported recognised state of a branch is 001X, the branch will conduct for the input pattern 0010 and 0011. The process works as follows for each line of the truthtable:

1. Find all MiniCABs that are connected to the output being examined.
2. For all of the PMOS and NMOS branches in those MiniCABs, evaluate whether any will conduct given the input pattern.
3. Derive one of the following four cases based on the results:
    (a) One or more PMOS branches conduct, no NMOS: Output = 1 (logic high).
    (b) One or more NMOS branches conduct, no PMOS: Output = 0 (logic low).
    (c) No branches conduct: Output = Z (high impedance).
    (d) One or more of both types of branch conduct: Output = X (short circuit).

This method has been verified by comparing its outputs for various circuits, with and without faults, to the outputs of SPICE simulations. As well as being faster, this method improves on the ngspice simulations method by providing a more reliable indication of whether a circuit is in a high impedance state or is short circuited.

7.8 Summary

The implementation of a simple software model of the PAnDA-Zwei architecture has been described. The model allows experimentation with the different layers of the architecture prior to fabrication. With regard to fault tolerance, this enables testing of different reconfiguration strategies in an easy to manipulate package. The next two chapters describe further research undertaken using this model which present novel fault tolerant methods, implemented on PAnDA, to improve reliability of its functionality in the presence of faults.

Chapter 8

Stochastic Swapping

Contents

8.1 Introduction

This chapter introduces the first attempt to exploit homogeneity in the layers of PAnDA-Zwei in order to fix faults. Two different non-destructive circuit transformations are defined and then used to explore the space of functionally identical circuits, in an effort to find one which works given a faulty environment. For this initial investigation the two transformations are applied at random to see if one is more successful than the other at fixing faults, or whether a combination of the two is better.

The method is comparable in outcome to Trimberger's work on alternative configurations [84], which attempts to make use of partially defective FPGAs. In his methodology multiple configurations for the same circuit are synthesized at compile time and stored in an enlarged configuration memory. During the programming stage, these configurations are tried in sequence until one is found to work as intended. The work in this chapter instead takes a single configuration and if it is found not to work the system will make small adjustments to the circuit until it does work. The difference is that only one initial configuration is required for the method presented here, rather than multiple being prepared in advance.

This chapter will discuss the concept of circuit transformations and then go on to describe the experiments that took place, along with the results and analysis.

8.2 Non-destructive Circuit Transformations

The large amount of homogeneity throughout the hierarchy of the proposed PAnDA-Zwei architecture is used to investigate how this could be exploited for the purpose of fault tolerance. At each of the configurable levels, structural repetition allows any circuit to be implemented in many different ways. In all but two of the 2^{16} logic functions implementable on a PAnDA-Zwei Slice, there exists redundancy that can be exploited by way of an alternative implementation in order to recover from at least one type of fault in the reconfigurable fabric.

The two that do not have this possibility are the most complex 4-input logic functions, odd and even 4-parity generators, when they are built as single logic gates. This is because they require all of the resources in a Slice, leaving none spare for redundancy. Multiple implementations of these circuits are possible however and it seems likely that transistor reconfigurations (inside a CT) could work around certain problems, although these will not be considered here as the simulator does not support changes at that level.

Although it is clearly possible to reconfigure around many different faults, it is not straightforward to know what reconfigurations must be made in order to fix an unknown fault. In order to be sure on the first attempt, one would have to know the exact location and nature of the fault in order to produce a circuit that works around it. This is often not convenient or possible in practice, so a novel method is proposed and tested that does not require a priori knowledge.

This novel method requires no knowledge of the nature of the fault, and only a general idea of where the fault has occurred. It does rely on having a method of testing functionality, but this is assumed to be feasible as in other work.

8.2.1 Input Swapping

Within a MiniCAB, the four inputs are each chosen from a set of 32. This allows a number of permutations for any given subset of inputs and the choice of permutation is functionally irrelevant, so long as the associated CTs are configured to match.

It can be seen therefore that the permutation of any Branch's input set can be changed to another, copying the CT configurations appropriately to match, and no knowledge of the current configuration is required as the resulting circuit will be functionally identical. This will be known as an Input Swap, and forms the lowest level transformation that this work considers.

Under certain conditions, Input Swapping allows fault recovery (see Figure 8.1). When a CT Branch has 1-3 CTs enabled, there are necessarily one or more CTs in the Disabled-Conducting state. Should any of these disabled CTs suffer a conducting fault, there will be no effect on the behaviour of the circuit, as they were already conducting. If a conducting fault occurs in an enabled CT, this clearly affects the behaviour of the circuit as the branch will conduct when it is not designed to. In this case, it is possible to take advantage of the previously mentioned case and swap the inputs and CTs around so that the **Disabled-Conducting** state configured for that CT is modelled by the fault and correct functionality is restored.

8.2.2 Branch Swapping

The Branch Swap transformation swaps Branches within a Slice. This is achieved by exchanging the configurations of all the CTs in a Branch with another. Branches can be swapped between MiniCABs as well as within them while maintaining a functionally equivalent circuit. If branches are swapped between MiniCABS, it is possible that the input multiplexers route the inputs in different orders. In this case, it is necessary to reorder the CT configurations as part of the swap. An example of a Branch swap is illustrated in Figure 8.1.

If there is only one function currently occupying the Slice, all eight Branches of each type can by freely exchanged. If more than one function exists in a Slice however, the branches can only be swapped between MiniCABs that form part of the same function. Although not explored here, the MiniCABs could be swapped around so that each function utilises a different set of them, allowing the Branches to be swapped more freely when more than one function exists in a Slice, but this is not explored here.

8.2.3 Higher Level Transformations

There is more homogeneity above the Branch level, and so more types of circuit transformation are possible. The next level up from Branch Swapping would be Slice swapping, where the configurations for the two Slices inside a CLB are swapped. This is a trivial operation which doesn't cause significant disruption and can easily be routed. The disadvantage compared to the lower level transformations is the significantly higher number of bits which must be read and written in order to exchange the configurations.

In this chapter we consider only Input Swaps and Branch Swaps as an initial investigation into the methodology. These were experimented with in order to evaluate the performance of this methodology.

8.3 Experimental Method

The two circuit transformations described in Section 8.2 were used as the basis for fault fixing strategies. A strategy here is used to refer to one or more circuit transformations that may be used to recover from a fault.

These strategies were as follows:

- Pick a MiniCAB at random and swap two random inputs.
- Pick two CT Branches at random and swap them.
- Shuffle the inputs of all the MiniCABs.
- Shuffle all the branches in the Slice.

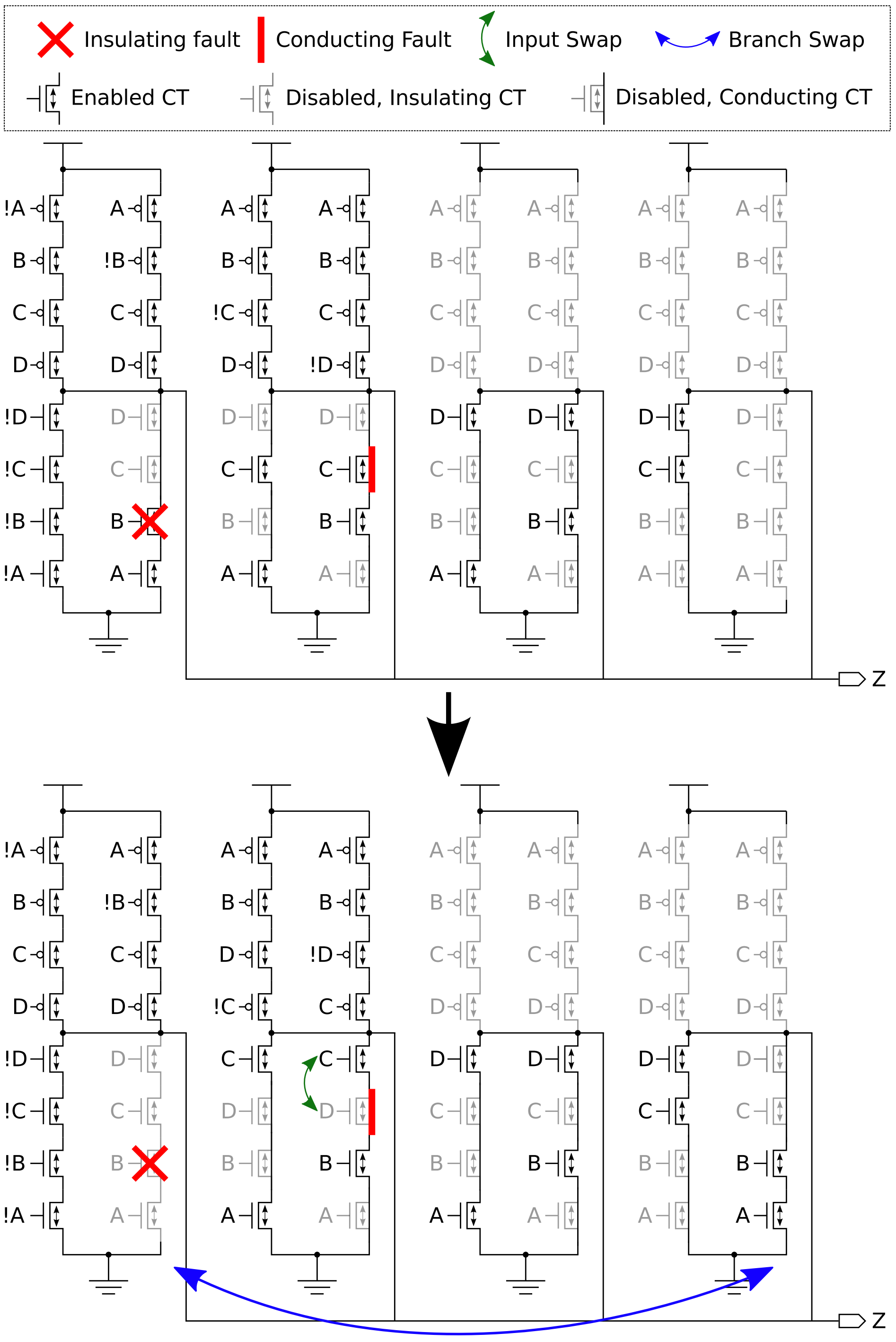


Figure 8.1: An illustration of the application of two circuit transformations. The rightmost NMOS branch in the first MiniCAB from the left suffers an insulating fault which is recovered from by being swapped with a spare in the third MiniCAB. The rightmost NMOS branch in the second MiniCAB from the left suffers a conducting fault, and is recovered from by swapping the mappings of inputs C and D.

One advantage of these strategies is that no prior knowledge of the configuration is required as the transformations are done at random.

In order to measure the effectiveness of these strategies in their ability to fix faults, an experiment was conducted which applied the strategies themselves at random to faulty circuits, until the fault was fixed or a pre-determined threshold was met.

8.3.1 Circuits to Test

Four different circuits were used for the testing which were chosen to represent a spread of different utilisations of a Slice. These are referred to as Z0 to Z3 in this thesis for ease of reference.

The circuits were selected to provide representative examples of a range of different utilisations of a PAnDA Slice that might be found in real circuit designs, from very low to very high. The circuits demonstrate a range of redundancy available for fault mitigation, from high to low. The simplest circuit, Z0, requires only two CTs and so has the most redundancy possible available for fault mitigation. Circuit Z3 uses most of the available CTs, leaving only one redundant branch of each type and eight of the fourteen used branches fully utilised. The other circuits, Z1 and Z2, represent midpoints of utilisation and different patterns of redundancy. It was felt that this set of circuits provide a sufficient spread of utilisation and redundancy levels for the investigations reported in this thesis.

Sequential circuits were not considered directly as they could be constructed from combinatorial components such as the ones under test. Significant additional effort would be required to construct and simulate sequential circuits on PAnDA and the results would not add anything conceptually to the concepts explained within the scope of this thesis.

Z0

The first circuit is the simplest possible binary logic gate, an inverter (Figure 8.2). This makes use of one CT and branch of each type in one MiniCAB in a Slice. This is was chosen for its simplicity, providing the highest possible amount of redundancy. All methods tested were expected to perform best on this circuit due to the amount of spare resources available.

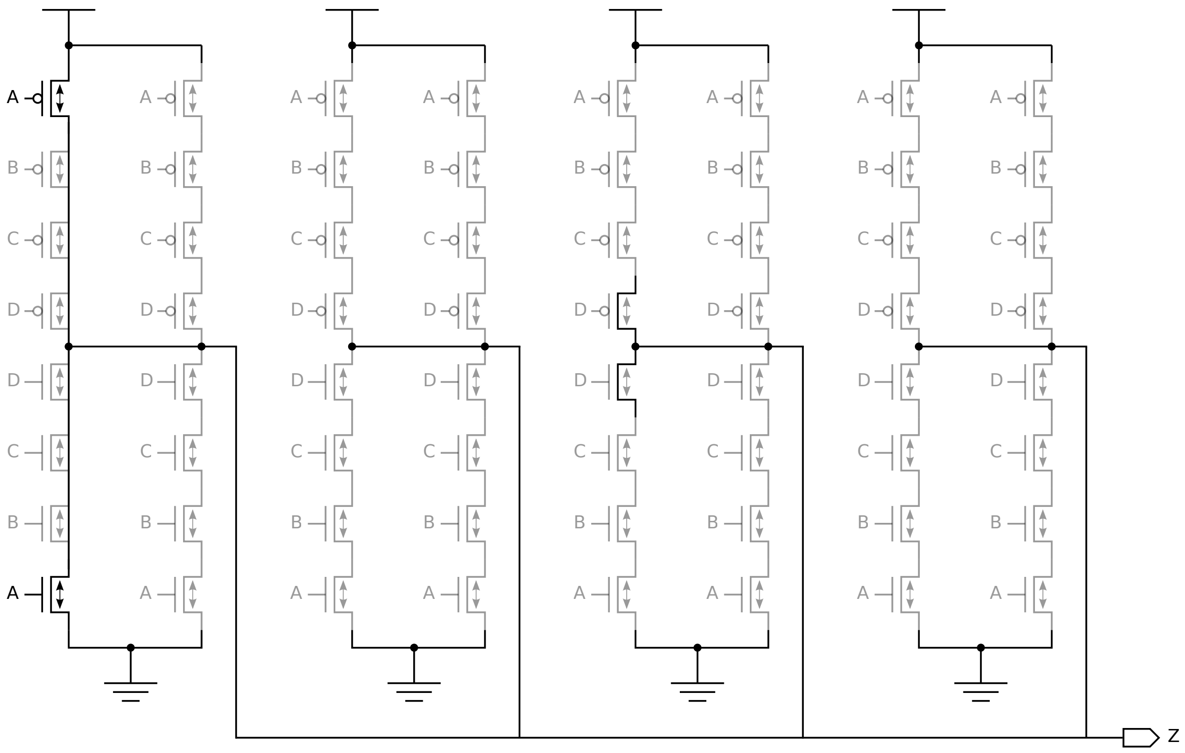


Figure 8.2: Circuit diagram of the Z0 configuration, a basic inverter. Only one CT of each type is used giving the maximum amount of redundancy.

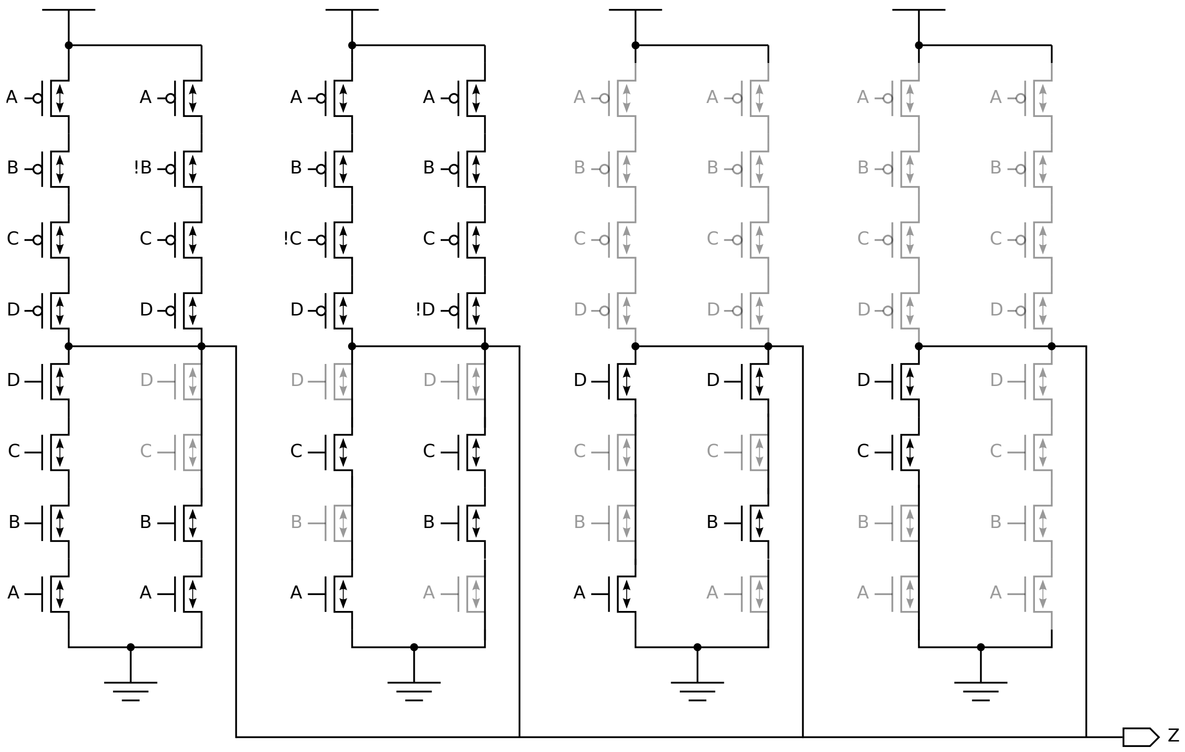


Figure 8.3: Circuit diagram of the Z1 configuration. Half of the CTs are utilized giving 50% redundancy.

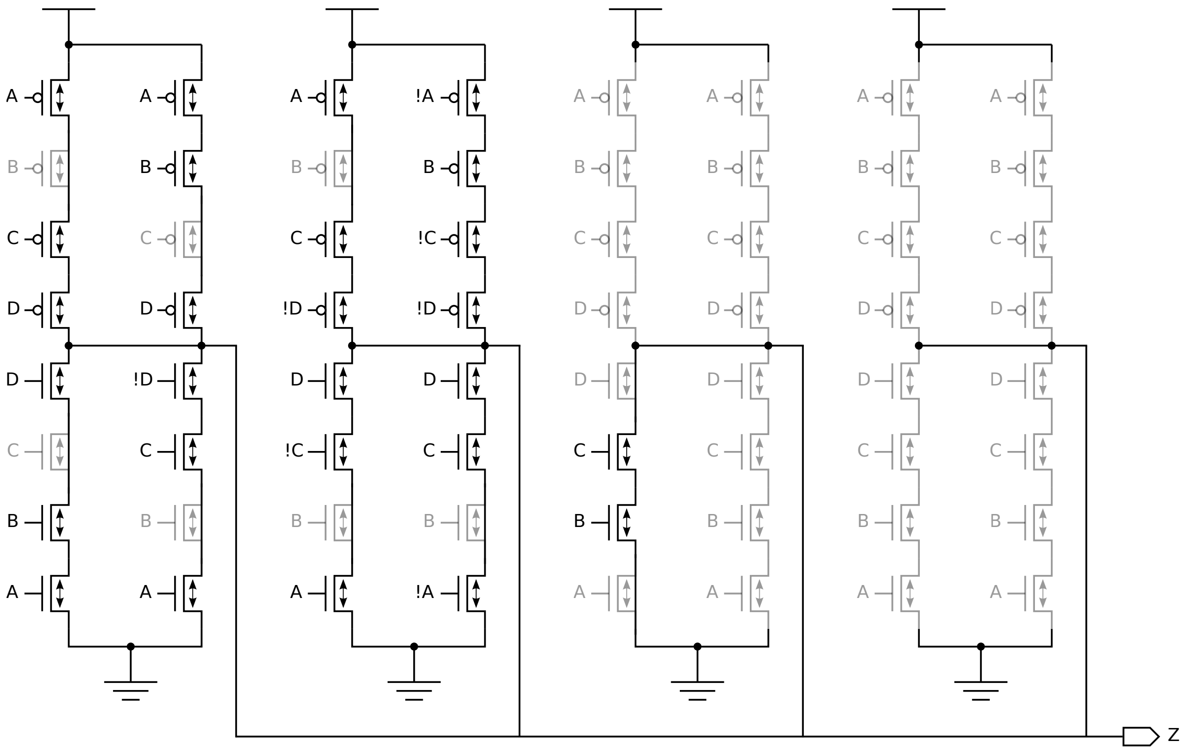


Figure 8.4: Circuit diagram of the Z2 configuration. Slightly less than half the CTs are utilised but the is only one complete Branch used compared with Z1 which has four.

Z1

This circuit is a four input "one hot detector", producing a high output if precisely one input is high (Figure 8.3). This circuit was chosen as it provides an interesting mix of redundancy. On the PMOS side, there are four fully occupied branches which means that input swapping will be ineffective, however there are four completely redundant branches which will allow branch swapping to recover from faults.

On the NMOS side, there is only one redundant branch, but all bar one of the utilised branches have two unused CTs which will enable input swapping to recover from some faults.

Z2

The third circuit was picked to represent a good mix of input and branch swapping possibilities (Figure 8.4). All bar one of the CT branches has at least one spare, enabling fault recovery via input swapping and there are four and three spare PMOS and NMOS branches respectively, allowing for fault recovery via branch swapping. It was expected that the proposed fault recovery methods would perform well on this circuit.

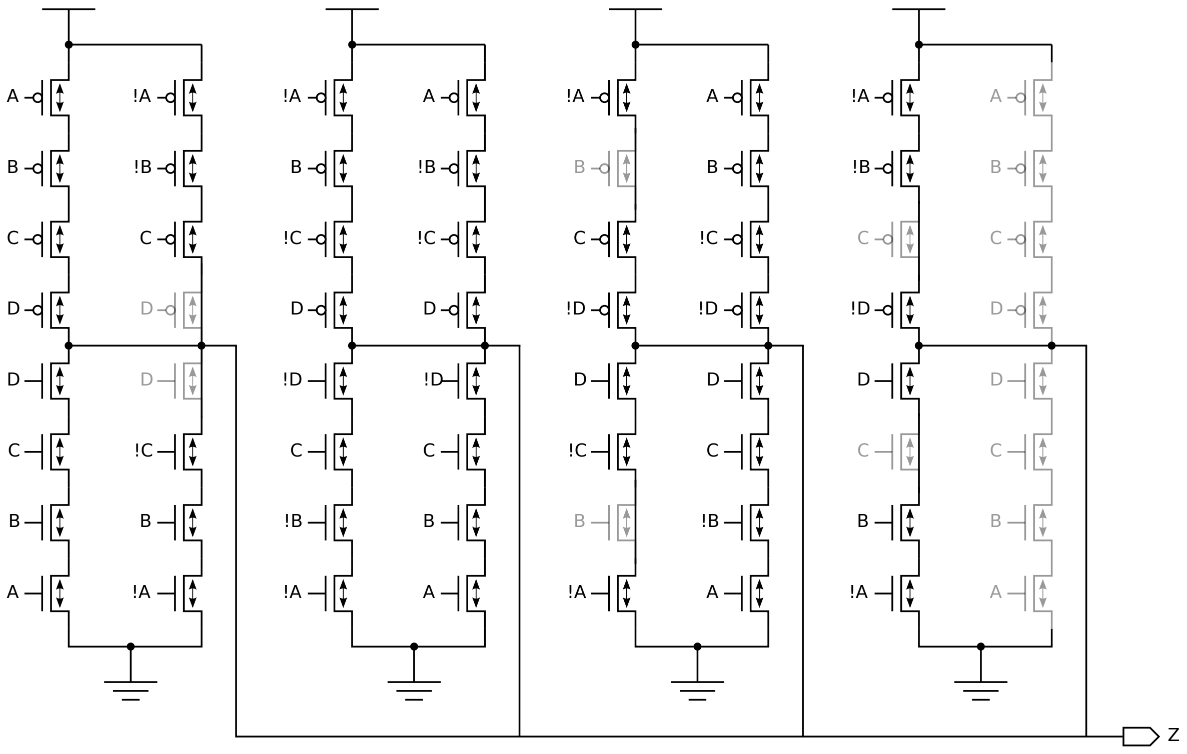


Figure 8.5: Circuit diagram of the Z3 configuration. This circuit uses most of a Slice's resources and so has far fewer spares than the other configurations.

Z3

The last circuit was picked as it has a high utilisation, with most utilised branches having no redundant CTs and only one redundant branch of each type (Figure 8.5). This comparatively small amount of redundancy does still provide some options for fault recovery, but it is expected to perform worse than the others during testing in terms of how often faults can be recovered from.

8.3.2 Fault Model

Experiments were conducted using the simulator discussed in Chapter 7. There were two types of fault used for experimentation, referred to here as Conducting and Insulating. These are intended to represent two extremes of what can happen when transistors fail and are commonly used in the literature as a basic fault model [85].

A conducting fault causes the faulty CT to permanently conduct between its source and drain nodes. This is suggestive of a short between the nodes, or perhaps the transistor suffering from variability and consequently having a very low or high threshold voltage, depending on whether it's NMOS or PMOS respectively. The functional effect of this is that the transistor conducts when not intended. In some situations, this will have no effect on the current configuration in PAnDA as CTs are sometimes used as conductors anyway. If this fault occurs in an active CT however, it will cause the CT to conduct on input patterns for which it is not supposed to. The effect of this is that for those input patterns, there may be a short circuit, for which there two consequences. The first is that damage could be caused to the chip as a large current flows directly from Vdd to Ground, potentially damaging more transistors. The second is that, if the chip survives, the logic output will be likely be either wrong or undefined ($\sim 0.5V$) due to the circuit essentially becoming a voltage divider. Since a CT suffering from a conducting fault acts a conductor, performing the input swap transformation can recover from the effects of this fault by moving an input from a faulty CT to a working one, if the configuration allows.

Insulating faults prevent the transistor from conducting between source and drain. In reality this could be due to a break in the circuit due to electromigration or the threshold voltage being exceptionally high or low (for NMOS or PMOS transistors respectively). If this type of fault occurs on an unused CT branch, there will be no functional effect. If it occurs anywhere on an active branch, it will cause one or more input patterns to produce a floating output, similar to a tri-state buffer. The branch is physically cut off and cannot be used for anything else. This situation necessitates a branch swap transformation to move the configuration to another branch without an insulating fault.

As mentioned before, each type of fault represents an extreme fault case and faults in reality may fall somewhere between the two, as well as manifesting in other ways (such as a transistor conducting between the gate and source or drain nodes). These other cases are not considered directly in this work but the methodology is still applicable since the precise nature of the fault is not considered during reconfiguration.

8.3.3 Fault Injection

Faulty circuits were created using the following process in the simulator:

1. The circuit was configured in a fault-free substrate.

2. The circuit was tested to ensure correct operation.

3. A fault of a random type (conducting or insulating) was injected into a random CT in the Slice.
4. The circuit was tested to see if the output is affected.
5. If the circuit still works the process is repeated from Step 3 and more faults are injected; if not, the process is finished.

Using this process, one or more faults are injected to break the functionality of the circuit. Faults which have no effect on the output are retained which may later affect the fault recovery. This was considered to be a realistic approach to modelling permanent faults - fault recovery would start after the first fault affects functionality, and faults not affecting functionality would accumulated unnoticed. It also makes the fault recovery more challenging as the exact number of faults in any one case is unknown.

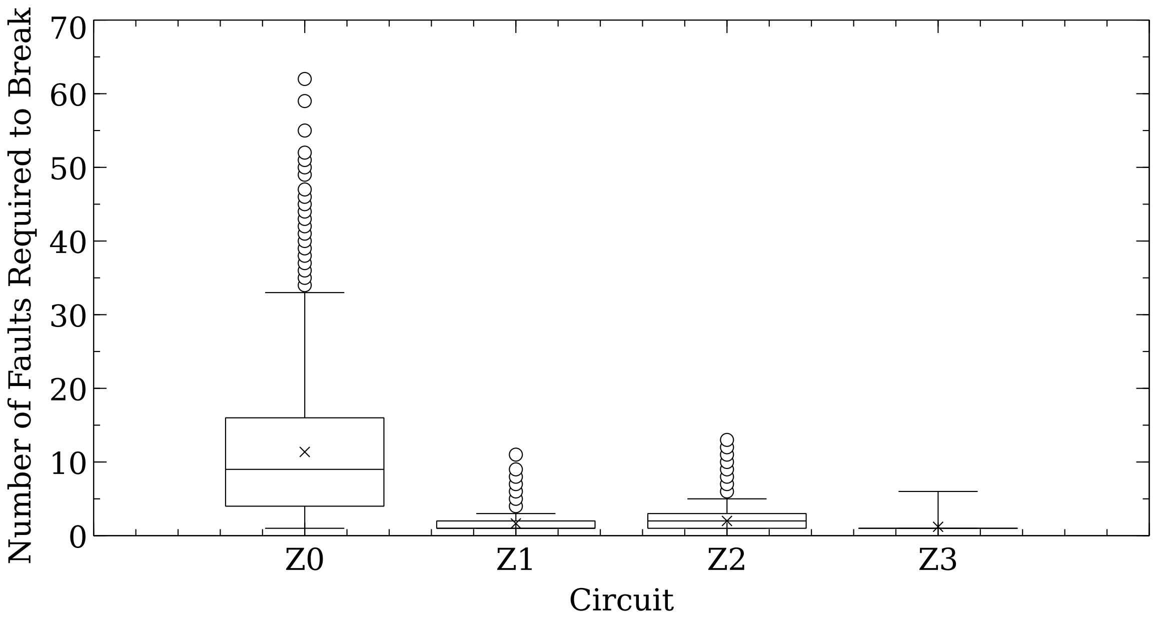
One observation of this approach is that a circuit with low utilisation, such as Z0, should on average receive more faults, and this may make fault recovery more difficult. This is because there is a large possibility that faults will be injected into CTs on unutilised branches, due to these forming the majority of the Slice configuration. These faults will accumulate until one of two things happens:

1. An insulating fault is injected into any of the CTs in the utilised branches.
2. A conducting fault is injected into either of the two active CTs.

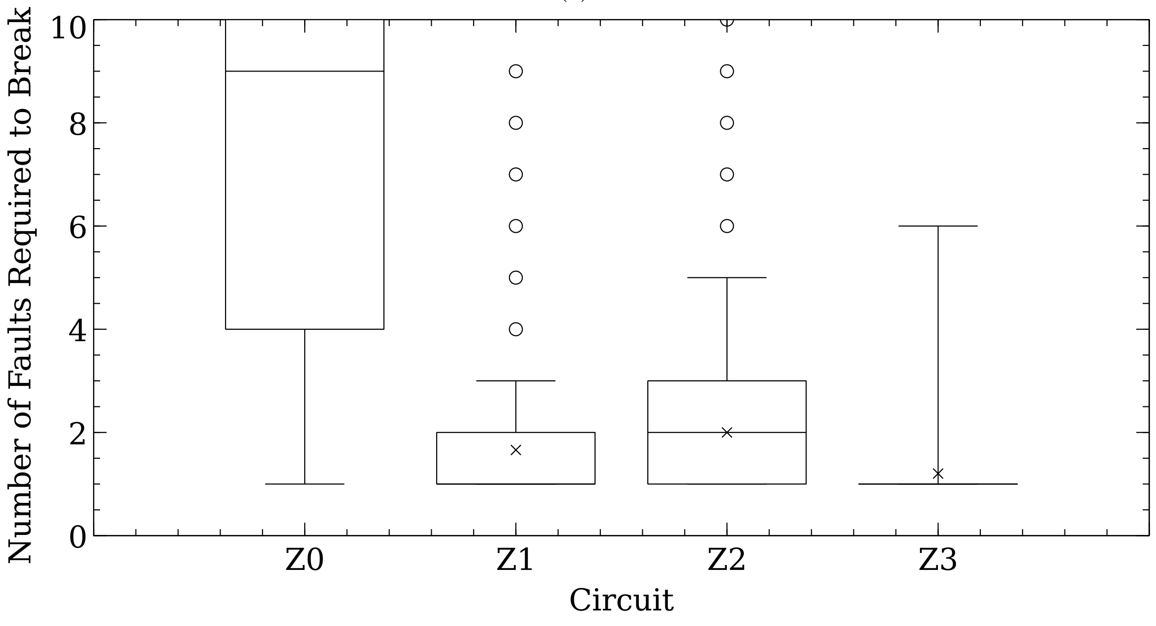
The first situation will cut off the affected branch and cause it to always insulate. When the input pattern is such that the branch should conduct, the output of the circuit will float and likely cause an undesirable value to be read by whatever is connected to it.

The second situation will cause the affected branch to conduct for all input patterns. In the case of the inverter, this means that for half the input patterns the output will be correct, and for the other half the output will be connected to both `VDD` and `GND` and therefore be undefined.

Using a whole Slice to implement an inverter is quite inefficient as the PAnDA architecture allows for the MiniCABs to be separated and used for separate functions, thus the three unused ones could implement other functions, but this design was chosen to measure the performance of the fault recovery in the best possible circumstances.



(a)



(b)

Figure 8.6: Statistical illustration of how many random faults are required to break each of the benchmark circuits. The results were obtained by injecting faults into fault-free Slices configured with the functions until functionality was lost. This was repeated 10000 times for each function. (a) The full results showing the large difference between Z0 and the other circuits. (b) The full results truncated to show more detail of Z1, Z2 and Z3.

Figure 8.6 illustrates how many random faults need to be injected into each circuit to break their functionality. The numbers are somewhat expected:

- **Z0** The number of random faults that need to be injected into Z0 to break it is higher than all the rest. This is due to the majority of the Slice being unused and so faults can accumulate without affecting functionality.

- **Z1** This circuit takes the second least number of faults to break, statistically. This is probably because there are four fully utilised Branches (a quarter of the Slice) any of which would be broken by a single fault.

- **Z2** A slightly higher number of faults is required to break this circuit than Z1. Since most of the branches have at least one CT in the **Disabled-Conducting** state, they are able to absorb some conducting faults without functionality being affected.

- **Z3** The least number of faults were required to break this circuit. This was expected as most of the Slice is utilised and so a random fault is most likely to affect an **Enabled** CT.

Parity Circuits

As mentioned in Section 8.2, four-bit even and odd parity circuits will not benefit from this methodology since they utilise all the CTs, leaving no spares. While this leaves no room for fault recovery within a single Slice, the functionality could be constructed by cascading two or three-bit parity circuits which do not utilise all the CTs. The number of faults that break these two circuits is shown in Figure 8.7.

The two-bit parity circuit fits inside a single MiniCAB, and the three-bit inside two MiniCABs. A four-bit parity circuit could be build with either a three-bit and a two-bit, or three two-bit circuits, either way taking up three MiniCABs and leaving spare resources. If these were split up between difference Slices then the fault tolerance methodology presented could be applied directly.

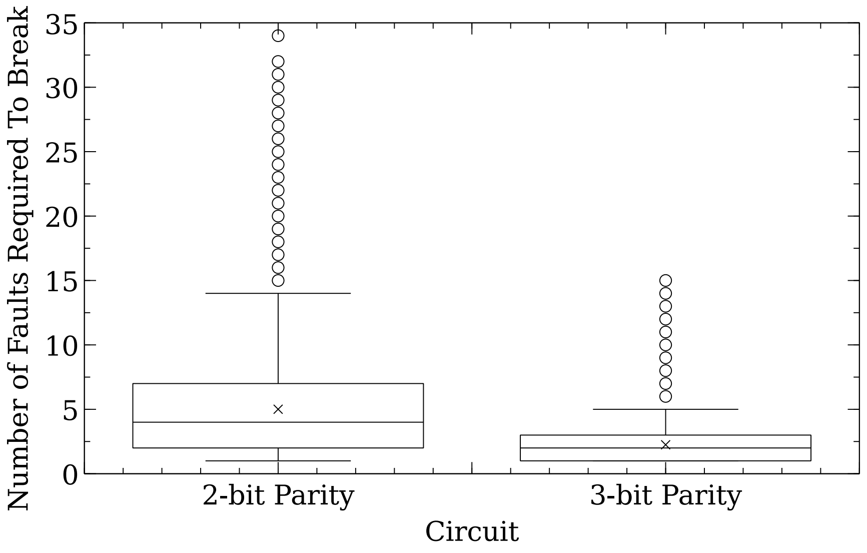


Figure 8.7: The number of faults that broke the two and three-bit parity circuits implemented on a Slice in 10000 samples. The three-bit parity circuit breaks after fewer faults than the two-bit circuit on average. Four-bit parity is not included as the circuit would always break after one fault.

8.3.4 Fault Recovery Strategies

For this experiment four different strategies were proposed, each using one of the circuit transformations described in Section 8.2. These strategies were as follows:

Random Input Swapping

One of the MiniCABs in the Slice is picked at random. Two of the four inputs are then picked at random and swapped. This operation is relatively cheap and not disruptive to the layout of the circuit, requiring the configurations of two input multiplexers to be swapped and eight CTs (two on each branch of the MiniCAB).

Input Shuffling

All the inputs of all of the MiniCABs in the Slice are shuffled using the Fisher-Yates shuffling algorithm [86]. This can be an expensive operation since the shuffle may require that all sixteen input multiplexers in the Slice be reconfigured.

Random Branch Swapping

A Branch type is first selected at random (PMOS or NMOS) and then two Branches of that type in the Slice are picked at random and swapped. This is a relatively cheap operation, requiring eight CT configurations to be exchanged (four for each branch). It can be disruptive to the layout of the circuit however, causing a previously compact circuit to spread out over a Slice.

Branch Shuffling

All of the branches of both types are shuffled within the Slice using the Fisher-Yates shuffling algorithm. This is the most expensive transformation, potentially requiring all 64 CTs to be reconfigured. It is also potentially the most disruptive to the circuit layout.

8.3.5 Fault Recovery

To recover from a fault, the fault tolerant methodology applies the strategies described in Section 8.3.4 to faulty circuits repeatedly and at random until either the circuit is fixed or a threshold of strategies applied (referred to from herein as steps) is met. Four experiments were conducted, testing each combination of one Input Swapping and one Branch Swapping strategy.

For each experiment, 101 runs were performed, each with a different bias value between and including 0 and 1. This bias controlled the probability of either strategy being chosen each time one was picked at random. For a bias of 0, only Input Swapping strategies would be used and for a bias of 1 only Branch Swapping strategies would be used. At a bias of 0.5, there would be an equal probability of either strategy being chosen.

It is assumed that the Input Swapping strategies are preferable to the Branch Swapping as they cause less disruption to the layout of the circuit. The advantage of this is that a small circuit, such as an inverter, does not begin to spread across multiple MiniCABS. The Branch Swaps are also assumed to be more complex due to the requirement to check and possibly swaps the order of the CTs if the input multiplexer configurations differ. For this reason, the power requirements are assumed to be higher despite the same number of CT reconfigurations being required.

For these reasons, it was hoped that the experiments would reveal an optimum bias value which achieved good fault recovery performance while using the Branch Swapping strategy as little as possible.

8.3.6 Experimental Procedure

Four different experiments were run, with the following combinations of strategies:

- Random Input Swapping and Random Branch Swapping
- Random Input Swapping and Branch Shuffling
- Input Shuffling and Random Branch Swapping
- Input Shuffling and Branch Shuffling

For each of these experiments the procedure was the same. As mentioned previously there were 101 bias values tested. Each bias value was tested 10,000 times to achieve meaningful statistics. Preliminary testing revealed this number of samples to be required for sufficiently stable results, whereas 1000 produced noticeable variance between repeats.

The threshold for the number of steps was chosen to be 10,000 to allow the algorithm a good chance to find a solution whilst not taking an infeasibly long time. In practice, this may be too high for real world use as it potentially involves tens of thousands of reconfigurations.

Each of the bias value measurements was conducted as follows:

1. Configure a function on a fault-free Slice.
2. Inject faults using the procedure described in Section 8.3.3.
3. Begin the fault recovery procedure.
4. Record the number of steps taken before the circuit works again if the procedure is successful.

Each set of experiments was repeated for the four circuits described in Section 8.3.1. This allows for comparison of how the methodology performs on different circuits given the same parameters and reveals whether one set performs well in general or if they can be tuned for different implementations.

8.4 Results

Figure 8.6 provides some context as to how many random faults break each of the four circuits, as each sample in the experiments may have suffered a different number.

Figures 8.8a-8.8d present the number of circuits fixed out of 10,000 for each of the four circuits (from Section 8.3.1) and for each of the four combinations of strategies (from Section 8.3.6). Figures 8.9a-8.9d present the mean number of steps taken to recover from a fault where the recovery was successful (where recovery was unsuccessful the number of steps would be 10,000 and so would skew the results in an unhelpful way). All these graphs have been truncated vertically, removing some of the extreme results on the left to emphasise the majority. Unabridged versions of these results can be found in Appendix B where some more details for each of the sixteen experiments are presented.

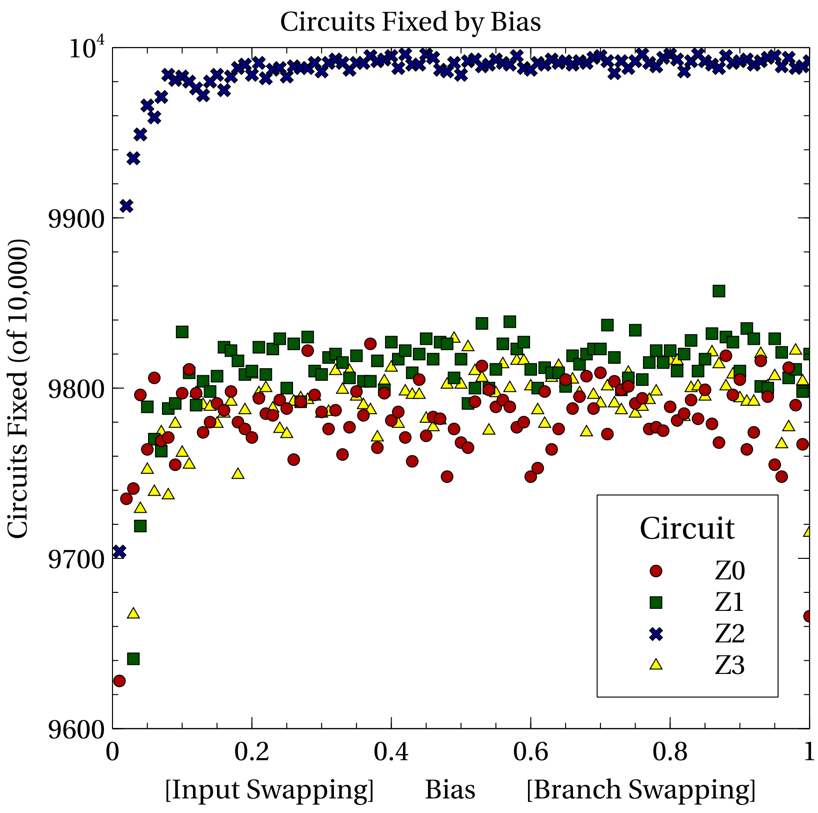
8.4.1 Circuits Fixed

The four graphs are very similar in their general shapes. The number of circuits fixed tends to increase as the bias varies towards the Branch Swapping strategy. From where the bias reaches ~ 0.2, the number of circuits fixed is approximately stable. In all four graphs, the results for Z2 are significantly higher than for the other three, averaging $\sim 9990 \pm 20$ in the stable section of the graph compared to $\sim 9800 \pm 50$ out of 10,000. All four graphs also show some significantly lower numbers of fixed circuits on the right hand side.

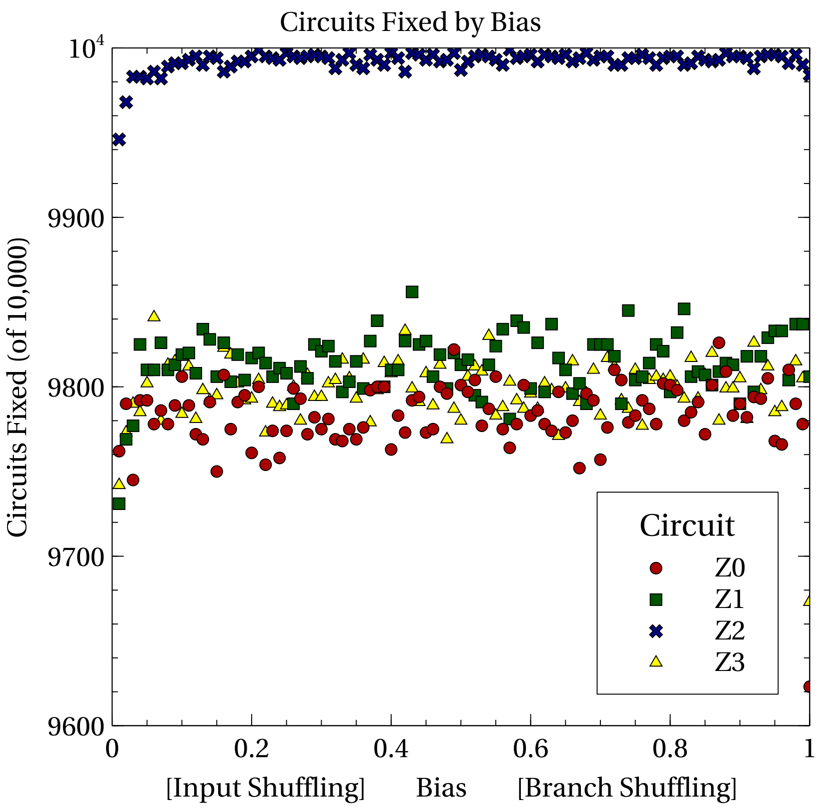
The general shape of the graphs was expected since Branch Swapping can be used to recover from either type of fault, whereas Input Swapping can only recover from conducting faults in Branches which have a spare CT. As mentioned previously, there is value in trying to use the Input Swapping strategies as much as possible since the disruption to the layout of the circuit is reduced. The results suggest that the same success rate for fault recovery can be achieved while still using a lot of Input Swapping.

It is clear from the graphs that all four combinations of strategies were significantly more successful at fixing faults on the Z2 circuit. It appears that the structure of this circuit is particularly suited to this form of fault recovery. There are a couple of reasons for this. Firstly, all but one of the active Branches contains a spare CT. This means that when a conducting fault is injected into a CT there is a high probability that recovery will be possible through Input Swapping. The second reason for the success is that there are four spare PMOS branches and three spare NMOS branches, giving a good amount of redundancy for Branch Swapping to take advantage of.

While circuit Z2 benefits from the redundancy, it may appear contradictory that circuit Z0, with the maximum amount of redundancy for a boolean function, does not. In fact for the majority of cases, fault recovery was much less successful on Z0 than any of the other circuits.

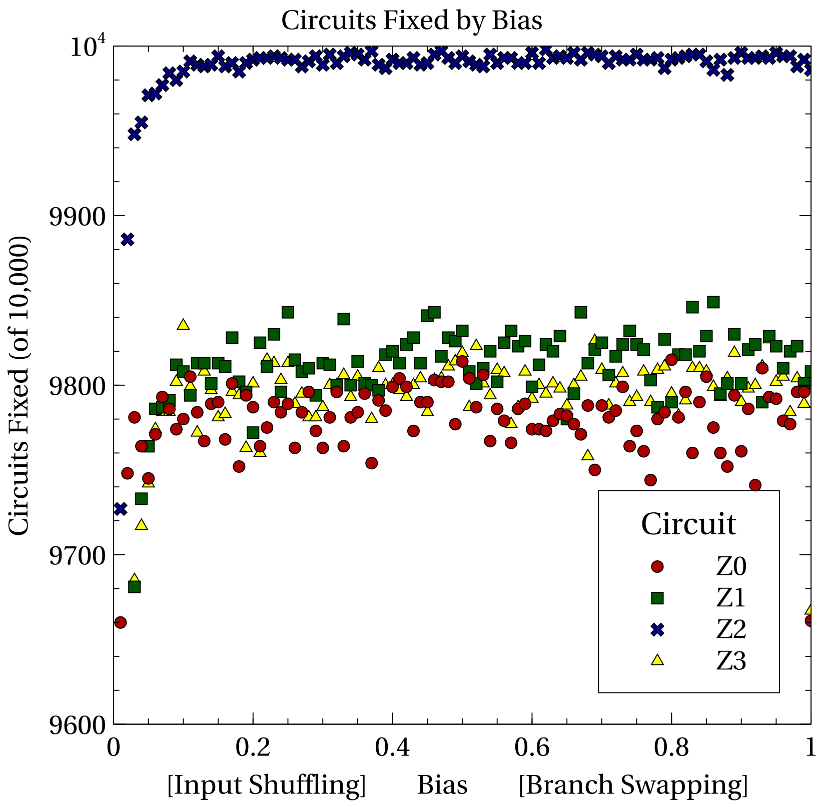


(a)

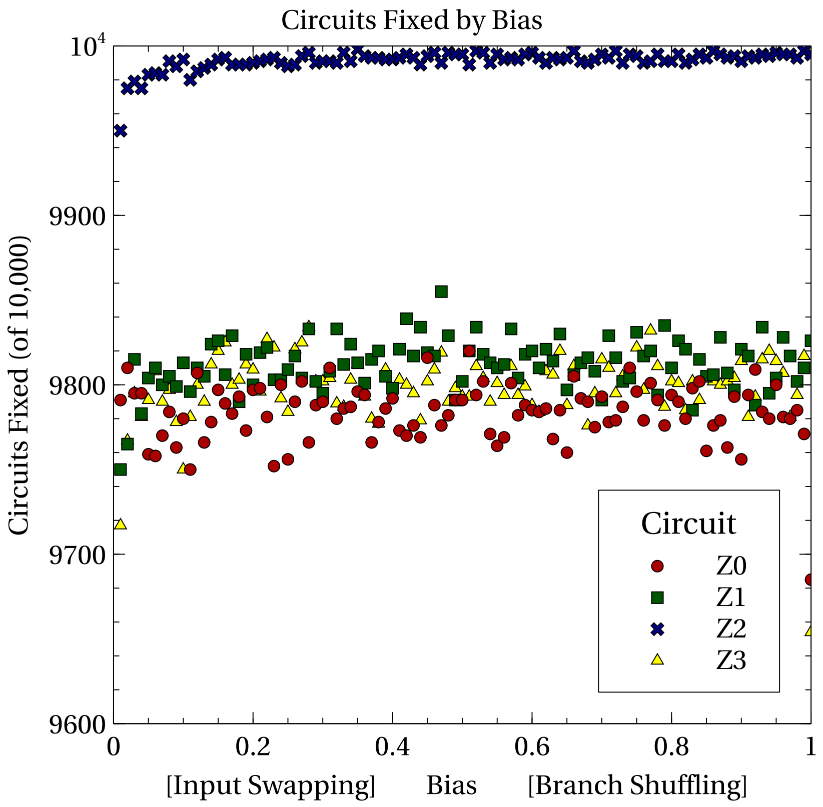


(b)

Figure 8.8: (a) Results of running the Random Input Swapping and Random Branch Swapping strategies on faulty instances of the test circuits. (b) Results of running the Input Shuffling and Branch Shuffling strategies on faulty instances of the test circuits.



(c)



(d)

Figure 8.8: (c) Results of running the Input Shuffling and Random Branch Swapping strategies on faulty instances of the test circuits. (d) Results of running the Random Input Swapping and Branch Shuffling strategies on faulty instances of the test circuits.

The reason for this is likely down to the method of fault injection. Since faults are injected at random without regard for where the active CTs are, the Z0 circuit is more likely to receive a higher number of faults on the unused CTs before finally being broken by a fault on one of the two active ones. When fault recovery starts, there are fewer fully functioning CTs to work with and so a workaround is either more difficult or impossible to find.

At the far right of the graphs, where the bias of 1 means that the recovery step can only use Branch Swapping, the success rate drops considerably. This is caused by specific combinations of faults which would be recoverable by using Input Swapping. An example of this would be one possible situation with Z0. Considering Figure 8.2, if all the PMOS CTs on the top row were injected with conducting fault, Branch Swapping alone would be unable to reach a working configuration again. It would require an Input Swap to move the CT configuration down to one of the lower rows. This shows that despite the Branch Swapping strategies being generally better than the Input Swapping strategies, the best performance is achieved using a mix of the two.

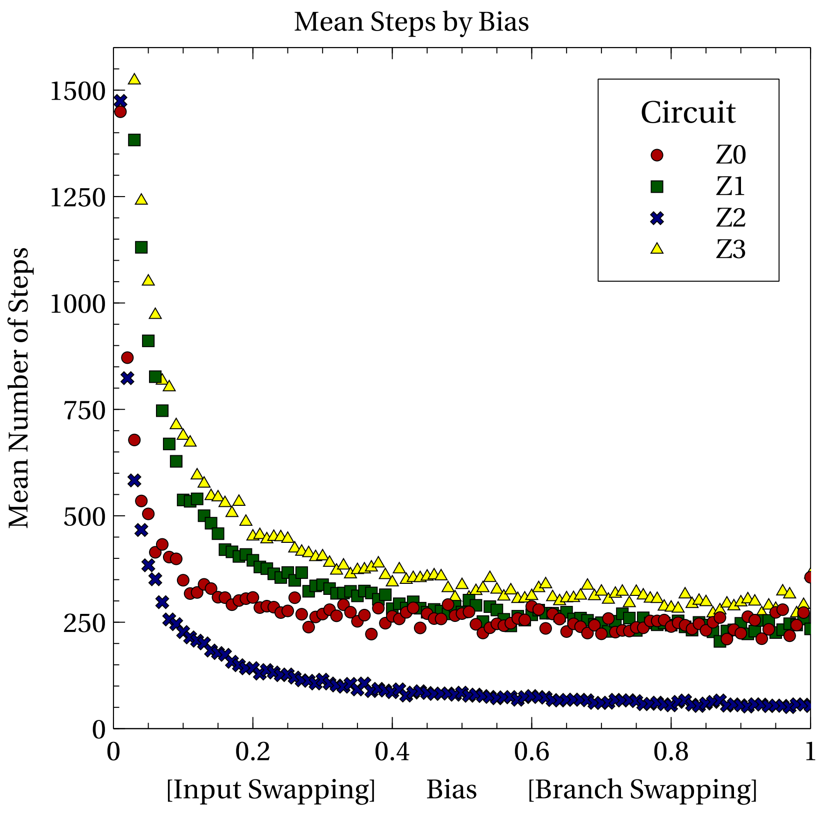
Figures 8.8a and 8.8c display worse performance at low values for the bias than Figures 8.8b and 8.8d. The difference between these two sets is the Branch Swapping strategy. The experiments in Figures 8.8b and 8.8d used Branch Shuffling which appears to have helped for low values of the bias. Since the shuffling strategies can perform one or more swaps during a step, the total number of swaps performed in a single step is increased bringing the performance closer to that of the higher biases.

8.4.2 Number of Steps Taken

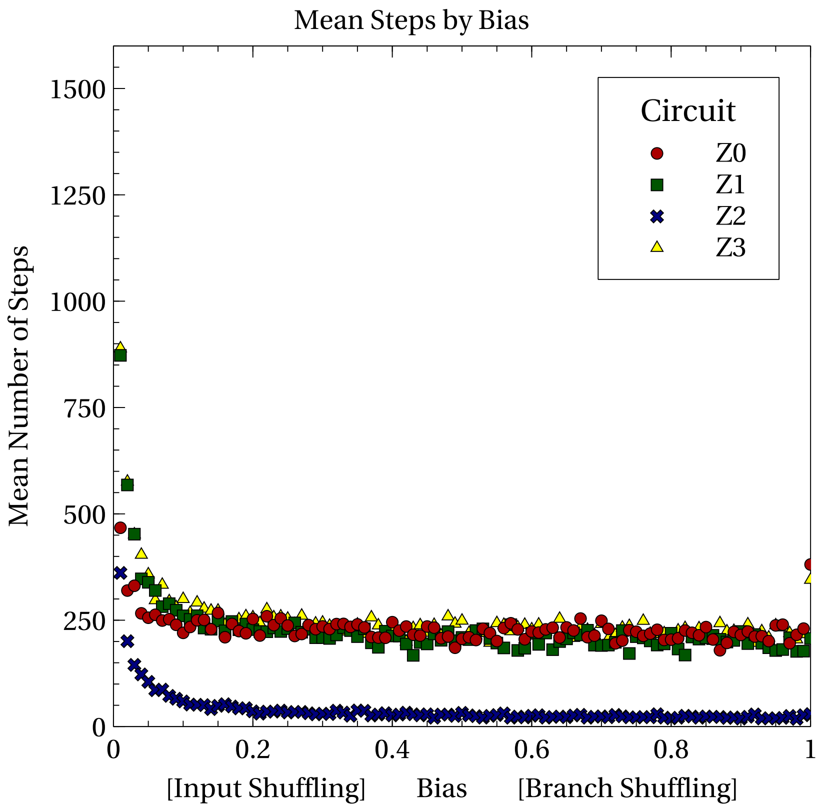
Figure 8.8 presents the number of steps taken to fix faults in the four experiments. Similar to the number of circuits fixed in the previous section there are not big differences in the general trends of the four graphs. The mean number of steps required to fix a fault reduces as the bias increases towards using more Branch Swapping, which represents a quicker fault recovery.

Comparing Figures 8.9a and 8.9c with 8.9b and 8.9d, the mean number of steps toward the left hand side is lower in the latter two, where each step involved shuffling all the Inputs on all the MiniCABs. This gives the algorithm a chance to fix any faults that are possible to fix with Input Swapping in one go. Due to the random nature however, it is also possible to put the circuit into a situation where faults cause more problems than the previous state.

The results for Z2 are again noticeably better than for the other three circuits, which all

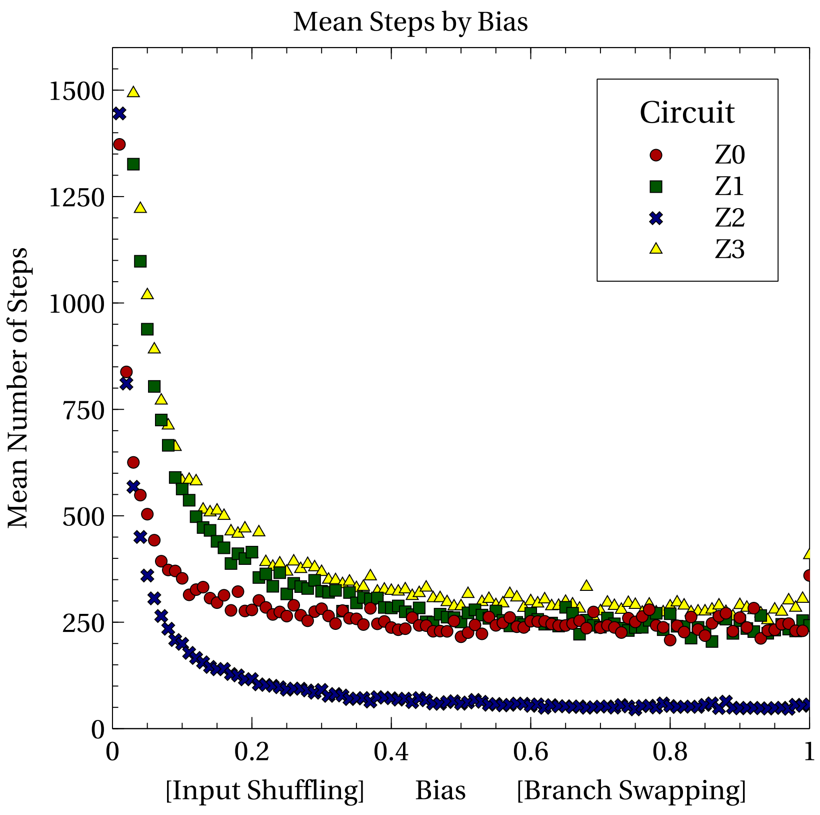


(a)

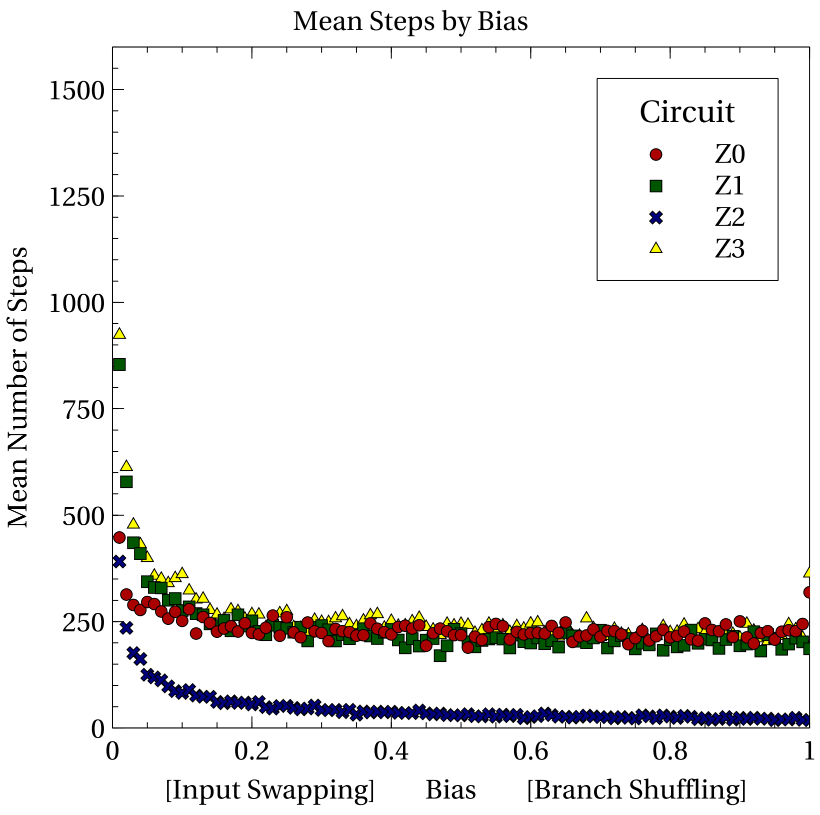


(b)

Figure 8.9: (a) Results of running the Random Input Swapping and Random Branch Swapping strategies on faulty instances of the test circuits. (b) Results of running the Input Shuffling and Branch Shuffling strategies on faulty instances of the test circuits.



(c)



(d)

Figure 8.9: (c) Results of running the Input Shuffling and Random Branch Swapping strategies on faulty instances of the test circuits. (d) Results of running the Random Input Swapping and Branch Shuffling strategies on faulty instances of the test circuits.

perform similarly. The balance of redundancy and utilisation in this circuit resonate with this methodology well for two reasons:

- By utilising slightly more than half of the Branches in the Slice, faults are more likely to occur in utilised branches, leaving redundant ones less likely to be faulty and increasing the likelihood that spares will be fault-free.
- Having slightly less than half of the MiniCAB branches unutilised leaves a lot of redundancy for Branches to move into. Random swaps have a good chance of putting active Branches into previously unutilised Branches, so fault recovery, if possible, is quicker.

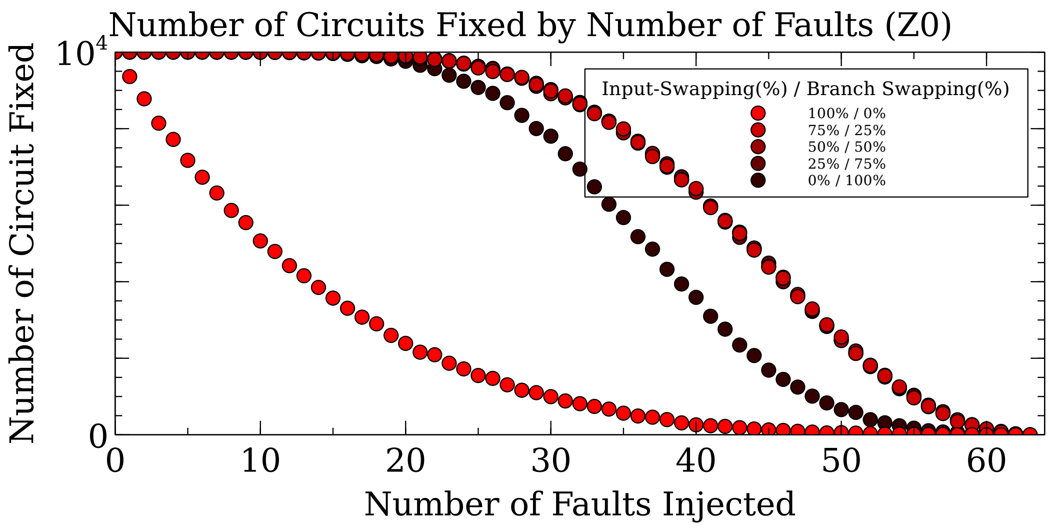
At the extreme right hand side of each of the graphs, performance is reduced somewhat in that the average number of steps to fix faults increases. This indicates situations where it was hard for the algorithm to find configurations that worked, and so more steps were taken to try to recover from the faults. Reducing the bias just slightly to include some Input Swapping clearly helps in these situations, perhaps where only one permutation of Branch configurations will work given the faults, whereas swapping the Inputs round allows other configurations to work.

8.4.3 Number of Faults

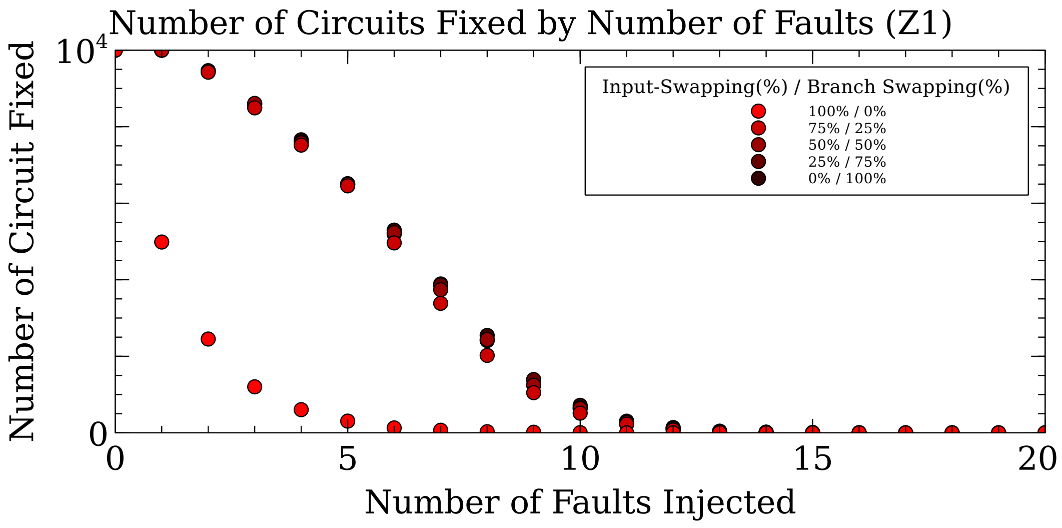
A separate experiment was run to measure how the different functions and biases performed with specific numbers of faults. For each of the functions Z0-Z3, 10000 experiments were run for increasing numbers of faults and with 5 different ratios between the two strategies. Figure 8.10 shows the results for these experiments.

The findings largely echo the other experiments in that Branch swapping was the most effective strategy. The circuit Z0 was found to be the most capable of recovering from faults. This is because there are not many resources required for the circuit to work and so many can be damage whist still making a fix possible. The requirements are:

- One working CT of each type.
- One working Branch of each type.
- No Branch of either type sustaining four conducting faults as this would create a permanent short.

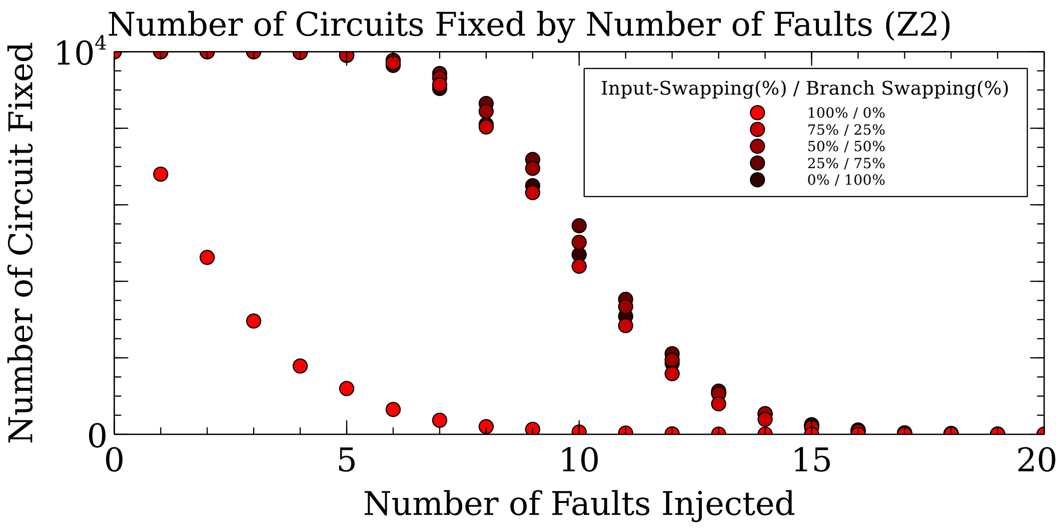


(a)

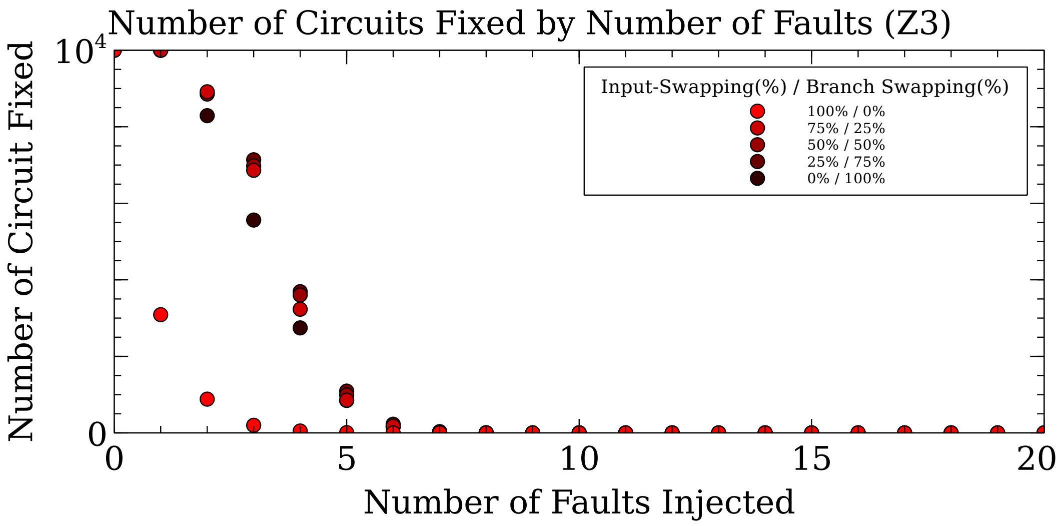


(b)

Figure 8.10: The number of circuits fixed out of 10000 samples when increasing numbers of random faults are introduced into (a) circuit z0 and (b) circuit Z1. Five strategies of combining the two circuit transformations were tried for each number of faults. The results for circuits Z1 were truncated to 20 faults as no more circuits were fixed after this point.



(c)



(d)

Figure 8.10: The number of circuits fixed out of 10000 samples when increasing numbers of random faults are introduced into (c) circuit Z2 and (d) circuit Z3. Five strategies of combining the two circuit transformations were tried for each number of faults.

This means that a working Z0 implementation can be guaranteed so long as no more than three faults have been sustained in the fault model used for this work. More faults than this would include the possibility of four Conducting faults in a single Branch, meaning that some of the input patterns would produce a short circuit and an undefined output. This is because only two transistors are required for the functions to work and so long as one branch of each type is free of insulating faults for these transistors to operate on, a fix is possible. The effect of this can be seen directly in the results in Figure 8.10a.

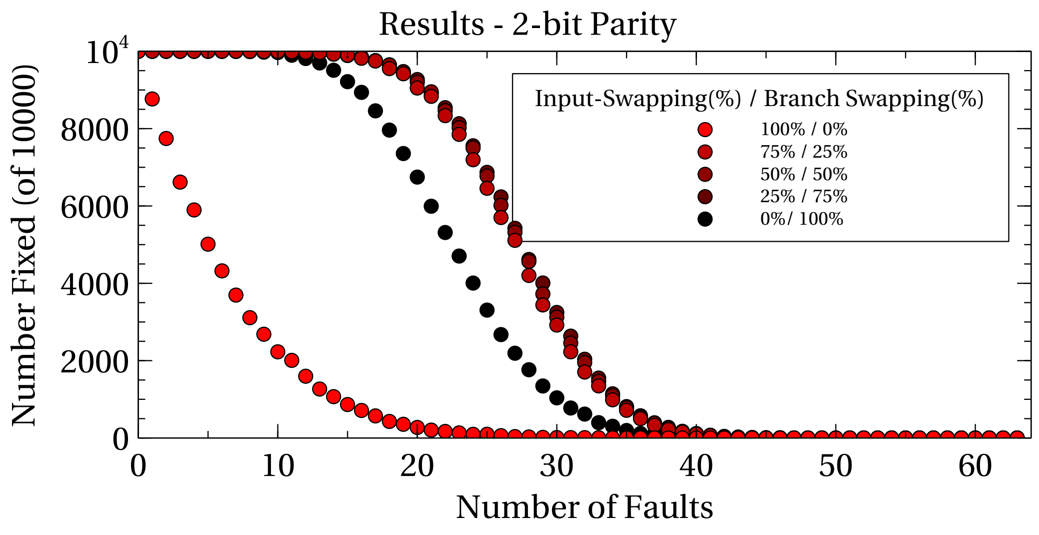
These experiments were also repeated on the two and three bit parity circuits (Figure 8.11). The results are markedly similar to each other apart from the different scales in the number of faults that they could handle. They exhibit the same general behaviour of the others (most obvious in 8.10a) where a total bias to one of the two swapping methods reduces the fault recovery performance. This reflects the earlier findings, adding evidence that both strategies are required for optimal performance.

8.5 Summary

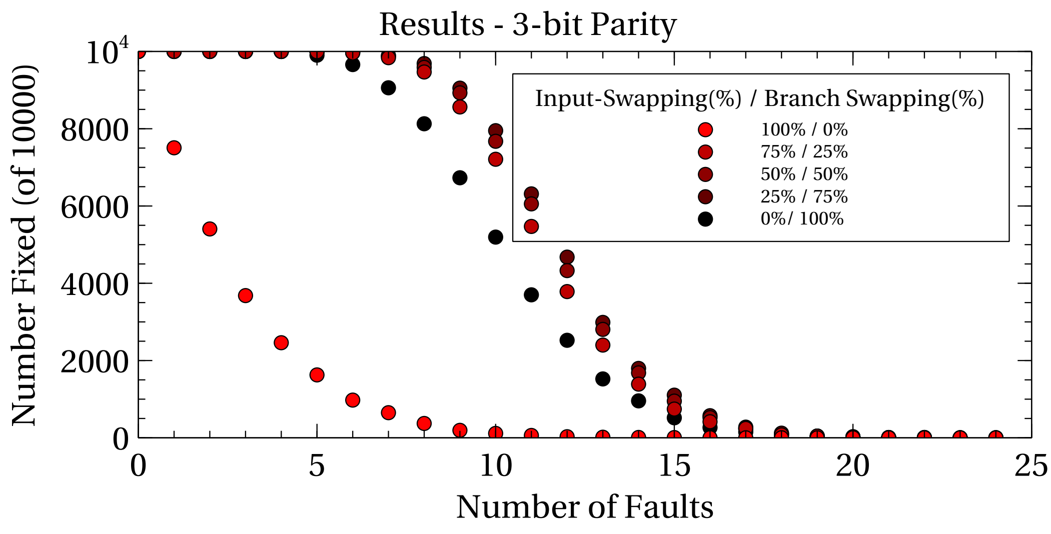
The results of these experiments have shown that even with *random* circuit transformations, it is possible to find reconfigurations of circuits in PAnDA-Zwei which work around faults. The methodology described has been demonstrated to work without any information about the configured circuit's implementation or the location or nature of any faults, simply the knowledge that a circuit is faulty. By exploiting the additional layers of configuration that PAnDA-Zwei has available, fault recovery has been performed in a simple yet effective way.

It has also been shown that the efficiency and effectiveness of this methodology can be controlled by simply biasing the random application of transformations between two different strategies. This is an interesting effect which could be used to optimise the methodology for a particular situation. If the quickest possible recovery is required, biasing towards Branch Swapping may be the optimum approach, whereas if the layout of the circuit should be left changed as little as possible, biasing towards the Input Swapping side can still yield successful results.

The experiment found that both types of transformation are required to achieve the best results. If biased strongly towards the Input Swapping strategy there is a drop in performance that becomes severe at the extreme. When biased towards the Branch Swapping strategy, the



(a)



(b)

Figure 8.11: The number of circuits fixed in (a) two and (b) three bit parity circuits when increasing numbers of faults were applied. Results from (b) were truncated at 25 faults due to no circuits being fixed.

results remain good until very high values of bias, at which point there is a significant (but not nearly as severe) drop in performance.

Some circuits performed better than others in the same circumstances. This is due to differences in their layouts changing the probability or difficulty of finding recovery configurations.

This chapter has presented some experimental work in applying random circuit transformation strategies to faulty circuits. It was found that this technique was successful in finding recovery mechanisms for faults and that the performance of this technique depended on a few different factors.

Compared to [84], a potential drawback of this method is that multiple configurations (as in Trimberger's work) could test very different circuit layouts quickly, whereas getting to a totally different circuit by making small changes (as in this method) will take longer. An interesting future experiment might be to combine the two, trying one configuration, making a few small changes and then trying a different configuration if that doesn't work.

The success of these experiments in actually finding working configurations after suffering from failure due to faults is the main point to take away. Having discovered this, the next chapter investigates some different strategies and ways of applying them.

Chapter 9

Deterministic Strategies

Contents

9.1 Introduction

The experiment in Chapter 8 utilised a random application of stochastic strategies for fault tolerance and was shown to successfully repair faults with varying levels of performance depending on the circuit it was applied to. It was hypothesized that an ordered application of deterministic strategies may have more scope for optimisation. This chapter discusses an experiment to test this hypothesis and measure any improvement.

9.2 Motivation

One drawback of the stochastic fault tolerance methodology presented in Chapter 8 is that there are often many unnecessary and unhelpful swaps being performed, which is expensive in terms of time and power due to producing more reconfigurations that is optimal. When trying to repair a fault without knowledge of its location, it is not possible to always take the most efficient route to a working configuration, but non-deterministic approaches potentially increase this as it takes no account of what components are actually active and which configurations have been tried before.

It was hypothesised that when the initial configuration of the circuit is known strategies may be more effective when applied deterministically. For instance, if a circuit is only using one PMOS and one NMOS branch, as in Z0, the majority of branches are unused and there is no effect in performing reconfigurations on them. If the fault recovery method was deterministic, the algorithm could avoid swapping between two unused inputs or branches, meaning less time and power wasted. This lead to the investigation of Hypothesis 3.

Hypothesis 3 *Applying circuit transformations deterministically can find working circuits on a faulty substrate more efficiently than a random application.*

This hypothesis prompted the thought that while keeping the two types of strategy, input swapping and branch swapping, the application of them could be predetermined in some way. By specifying the exact transformation that happens for each strategy, fine grained control over what would happen in the event of a fault can be achieved. By enumerating all possible swaps within a slice, these swaps could be chained together to form a list of actions to perform, and the resulting efficiency of this list could be optimised.

9.3 Strategies

The two types of circuit transformation described in Section 8.2 were again used to form strategies to repair faults. This time however, each strategy consisted of just a single input or branch swap. It was calculated that there were a total of 80 possible swaps that could be performed on a single slice, 24 input swaps and 56 branch swaps. A full list of the enumerated Strategies can be found in Appendix C.

9.3.1 Input Swapping

Each MiniCAB has four inputs which results in there being six distinct swaps that can be performed between them. This can be calculated using the combination $\binom{4}{2} = 6$. Input A can be swapped with B, C and D; Input B can be swapped with C and D, since the A with B swap is already covered and finally Input C can be swapped with D. Since there are four MiniCABs in a slice each with 6 possible swaps, there are 24 possible input swaps possible in a slice.

Each swap was assigned a number in order to refer to it conveniently later. The input swaps were assigned the numbers 0-23, where 0-5 refer to swaps in the first MiniCAB, 6-11 in the second and so on. Within each MiniCAB, the swaps are arranged in the order used above, where the first is swapping A with B. Figure 9.1 illustrates the input swaps and identifiers assigned to them for the left-most MiniCAB.

9.3.2 Branch Swapping

Each slice has sixteen branches, eight of each type. Branches can only be swapped amongst their own type to maintain the same functionality, so the total number of possible swaps is $2 \times \binom{8}{2} = 56$.

The branch swaps were assigned numbers in the same way as with the input swaps. PMOS branch swaps were numbered 24-51 and NMOS branch swaps 52-79. The first seven NMOS branch swaps are illustrated in Figure 9.1.

9.3.3 Strategy Lists

Associating every possible swap with a unique number means than they can be referred to in a convenient and unambiguous manner. Arranging these numbers into a list (as in Figure 9.2)

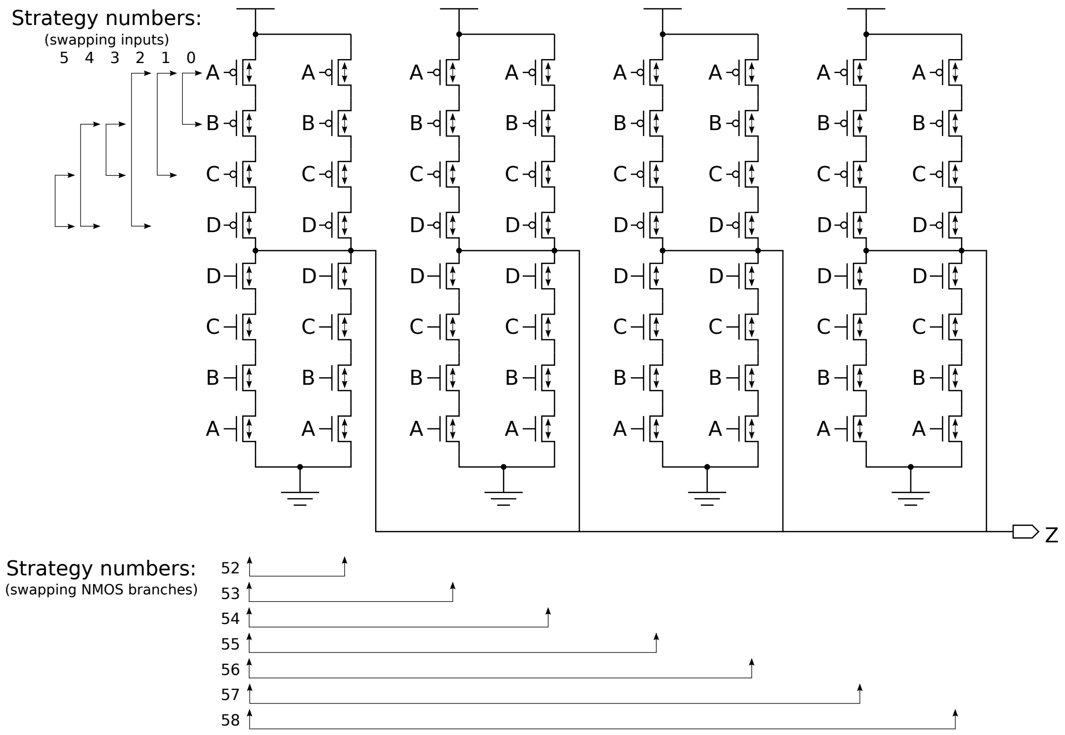


Figure 9.1: An illustration of how some of the strategies are mapped from numbers to circuit transformations. Strategy '0' for example will swap the input multiplexer configurations for inputs A and B, and also two CT configurations in each branch associated with inputs A and B. Strategies 6-11 will perform the same actions on the second MiniCAB from the left and so on. Strategy '53' will swap all four CT configurations in the leftmost NMOS CT branch with the ones in the third NMOS CT branch from the left, compensating for any differences in the order of the inputs between the two MiniCABs. Strategies 59-64 will swap the second NMOS branch with each of the others to the right and so on and strategies 24-51 enumerate all the possible PMOS CT branch swaps. This means that all possible transformations can be accessed by the GA. A complete mapping of numbers to strategies can be found in Appendix C.

defines a deterministic set of actions that can be performed upon a circuit. This thesis refers to lists of this type as Strategy Lists.

Given a particular strategy list, the methodology would work as follows when attempting to repair a fault:

1. Start with a faulty circuit and knowledge of the function it is meant to implement.
2. Read the first number from the strategy list.
3. Apply the strategy referred to by the strategy number to the circuit.
4. Test the circuit to see if it now works.
   (a) If it does, end the process.
   (b) If it doesn't, read the next number from the strategy list and continue from Step 3.

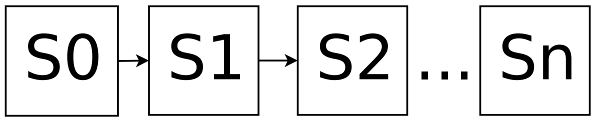


Figure 9.2: An abstract Strategy List. S{0..n} represent the strategy numbers which are applied in order.

An example list, using the numbers illustrated in Figure 9.1, could be [0, 3, 5, 52]. If the entire list is applied to the circuit Z0 (Section 8.3.1), the following circuit transformations are performed:

1. Inputs A and B are swapped in the first MiniCAB from the left (Strategy 0).
2. Inputs B and C are swapped in the first MiniCAB from the left (Strategy 2).
3. Inputs C and D are swapped in the first MiniCAB from the left (Strategy 5).
4. The first and second NMOS branches from the left are swapped (Strategy 52).

This process would result in the circuit presented in Figure 9.3. It should be noted that each of the strategies in this manually written list act upon the active CTs or the branches which contain them. The work in this chapter investigates whether it is possible to automatically create lists which repair faults in a more efficient way than the random method presented in Chapter 8.

9.4 Experimental Method

The methodology presented in the previous chapter performed a stochastic application of random strategies in order to find a working configuration. The newly proposed methodology performs deterministic strategies in a deterministic manner, specifically single-swap strategies stored in an ordered list. Whereas the previous method would apply a different sequence of circuit transformations each time it was used, this method will apply the same sequence of transformations each time. The experiments performed in this section will attempt to automatically find and optimise such sequences for repairing faults. If successful, these lists could be stored on-chip (for example) so that they are available to use for fault recovery.

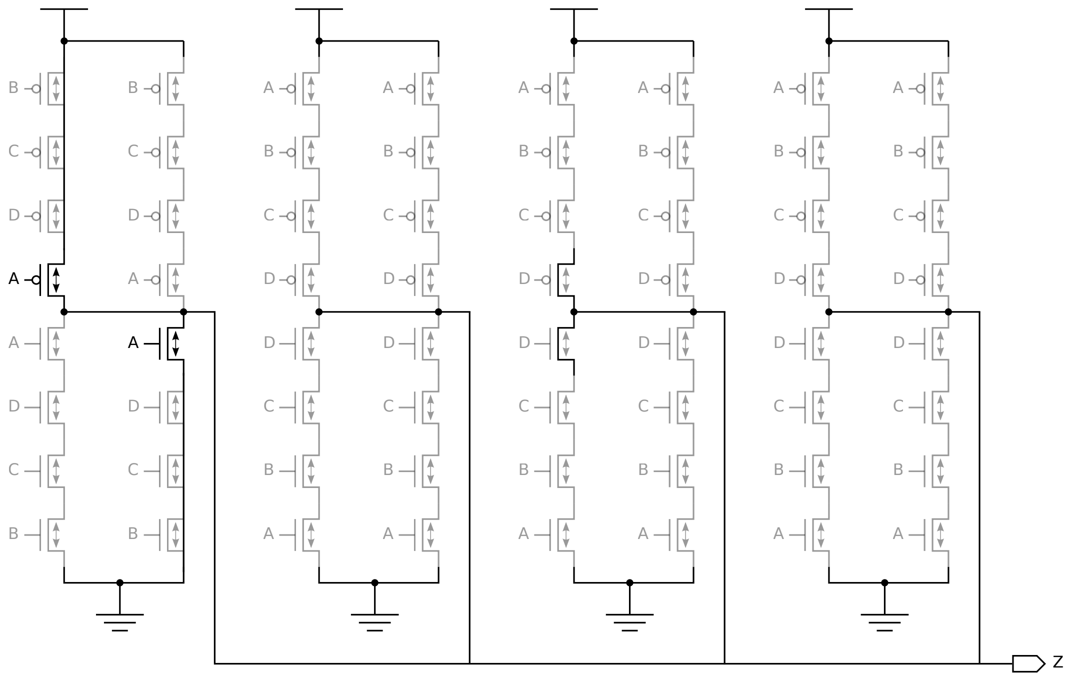


Figure 9.3: The result of applying the strategy list [0, 2, 5, 52] to circuit Z0. Compare with the original circuit in Figure 8.2.

9.4.1 Generating and Optimisating the List

The process of creating a strategy list for a given function is intended to be automatic. There are 2^{16} possible 4-input functions that could be implemented in a Slice, which makes it infeasible to develop them all by hand. In practice, these could all be pre-computed or lists for functions used in a circuit could be generated during synthesis. To achieve this automation, a Genetic Algorithm (GA) is employed which optimises lists iteratively. Multiple objectives for the optimisation, such as fixing as many faults as possible and doing it in the least number of steps, necessitate the use of a Multi-Objective algorithm, for which NSGA-II was used [59].

The length of the lists used in the experiments was 50, meaning that a maximum of 50 strategies would be applied in an attempt to fix a fault. This value, which is significantly lower than the 10,000 used in the previous method, was decided upon as in the previous experiment, circuit Z2 was using approximately 50 steps on average to fix faults while the other circuits used significantly more. The results should therefore show clearly any advantage this new approach might have.

Cost

In addition to the two measurements used in Chapter 8, a cost measurement was introduced. The purpose of this was to attempt to encourage the optimisation algorithm to favour Input swapping over Branch swapping since there is no longer a bias value to control this. The reason for this favouring is that if a fault can be fixed with just input swapping, it disrupts the layout of the circuit much less than branch swapping.

The input swapping strategies were assigned a low cost of 1, whereas the branch swapping strategies were assigned a high cost of 100 and the optimisation algorithm used the total cost expended as an objective to minimise when testing a list whether it succeeded or not. These numbers were chosen to significantly penalise the branch swapping and encourage input swapping.

There were a couple of options when defining how the average cost would be calculated. One way would be to calculate the cost based on the list alone, adding up the cost of all the strategies. The other was to add up the cost of the strategies actually used during the run. The second option was taken as it has the added benefit of encouraging better performing strategies since if a faulty circuit is fixed before applying the whole list (which is good), the cost will be lower and list's fitness will be better.

Implementation

The generation, optimisation and evaluation is performed using the commercial optimiser LS-OPT[87]. The algorithm used is the applications implementation of NSGA-II..

To enable LS-OPT to run the simulations, an input file was produced containing 50 variable placeholders representing a generic list of 50 elements. This was read and a number of concrete lists of 50 numbers between 0 and 79 (which represented a strategy list of length 50) were generated. These concrete input files were then evaluated by using them as input to a specially written experiment program. This program essentially wrapped the experimental logic and the PAnDA simulator described in Chapter 7 together to produce the required statistics. The statistics were then read back in by LS-OPT and used to produce the next generation.

Parameters

The parameters used for the GA are as follows:

- Population: 80
- Mutation rate: 0.1
- Generations: 400

The population and mutation rate are the default values suggested by the application. The experiments were set to run continuously until stopped manually.

Objectives

The objectives to minimise were as follows:

1. Number of unfixed circuits (out of 1000.)
2. Average number of steps taken before a circuit is fixed.
3. Cost/Effort expended to fix circuits.

Each strategy list was evaluated 1000 times on the same circuit with different faults.

The length of the strategy lists was fixed at 50 strategies. Given the results obtained in Chapter 8 where the lowest mean number of steps was approximately 50, this limit was chosen to make all the objectives a challenge for the GA and to encourage improved performance.

9.4.2 Strategy List Evaluation

Strategy lists generated by the GA were evaluated in simulation. The following process was repeated 1000 times for each list:

1. A fault-free PAnDA Slice model was prepared and programmed deterministically with a logic function.
2. The circuit was made faulty using the process described in Section 8.3.2.
3. The algorithm read the first strategy in the strategy list being tested.
4. The strategy was applied to the circuit.
5. The circuit was simulated to check whether it now functioned correctly.
6. If the circuit was still faulty and the strategy was not the last in the list, the algorithm read the next item in the strategy list and looped back to Step 4. Otherwise, if it was fixed or every item in the list had been used, the evaluation was over and the number of steps taken was added to a running total.

As in the experiment in Chapter 8, each of the 1000 evaluations of the list had a random number of different faults meaning that the solutions found would not be optimised for any particular set of faults.

9.4.3 Stochastic Method Comparison

To provide a more direct comparison between the Deterministic and Stochastic methods, results were obtained from a slightly modified version of the experiment in Chapter 8.

Input Swapping Strategy

There were two different input swapping strategies in Chapter 8's work: a single random input swap and a shuffle of all the inputs in the slice. The first is equivalent in this chapter's work to running one of the strategies numbered 0-23. The second would involve running up to three of these on each of the MiniCABs, so up to 18 single swap strategies.

To provide a straight comparison, this strategy was modified to make a single swap on a single MiniCAB by invoking a random deterministic strategy in the range 0 to 23.

Cost

The experiment in Chapter 8 did not consider the concept of cost, so this measurement was added into the new stochastic method for comparison. The total cost is accumulated over all successful and unsuccessful runs.

Measuring Unfixed Circuits

The LS-OPT software allows one to either maximise or minimise all the measured responses and so instead of measuring the number of circuits fixed out of 1000, the number of circuits that were not fixed is measured. This simply inverts the points on the graph compared to Chapter 8, so that lower values are better.

Average Steps Measurement

The measurement of the mean number of steps to fix a fault was altered slightly to better fit with the optimisation based approach. Instead of recording the number of steps taken in only the successful cases, the total number of steps used over all the repeats of the experiment was used. It was noticed that the old method would otherwise assign a high score to a list that did one useful swap and then 49 that would have no effect, as the average steps to fix a fault in this case would be 1. The other advantage of taking this approach is that it promotes lists that fix faults, in addition to the "unfixed" objective. This is because each time a list is tested and doesn't find a fix, it adds 50 (one whole list length) to the total number of steps used, whereas if it finds a fix part way though it adds only the number of steps it used, hence the more that are fixed, the lower the number of steps. The total number of steps was divided by 1000 to give an average per run.

9.5 Results

9.5.1 Deterministic Strategies Experiment

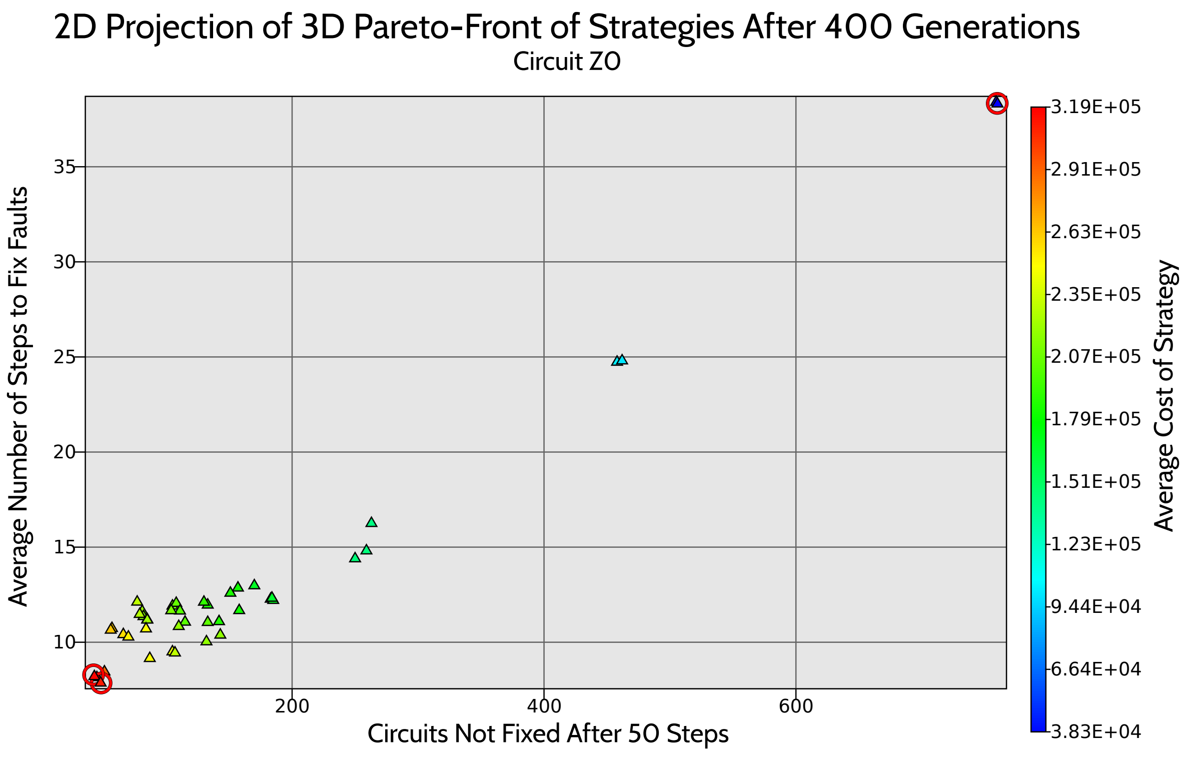
Results for each of the four functions are shown in Figure 9.4. Each of the points on the graphs represent a pareto-optimal solution (a strategy list) found after running 400 generations of the experiment described in Section 9.4.2. The experiments were stopped at this point due to the fact that a significant improvement had been seen and further improvements had slowed down.

The general trend that can be seen in all four cases is that solutions leaving fewer circuits unfixed (repairing more) do so, on average, in fewer steps. This comes at the expense of cost however, which can be seen increasing as the other two objectives decrease. This appears to mean that for faults that *are* repairable the strategy lists are able to repair them quickly. At the same time, these lists tend to be made up a large amount of Branch Swapping, and so this is a trade-off to be considered.

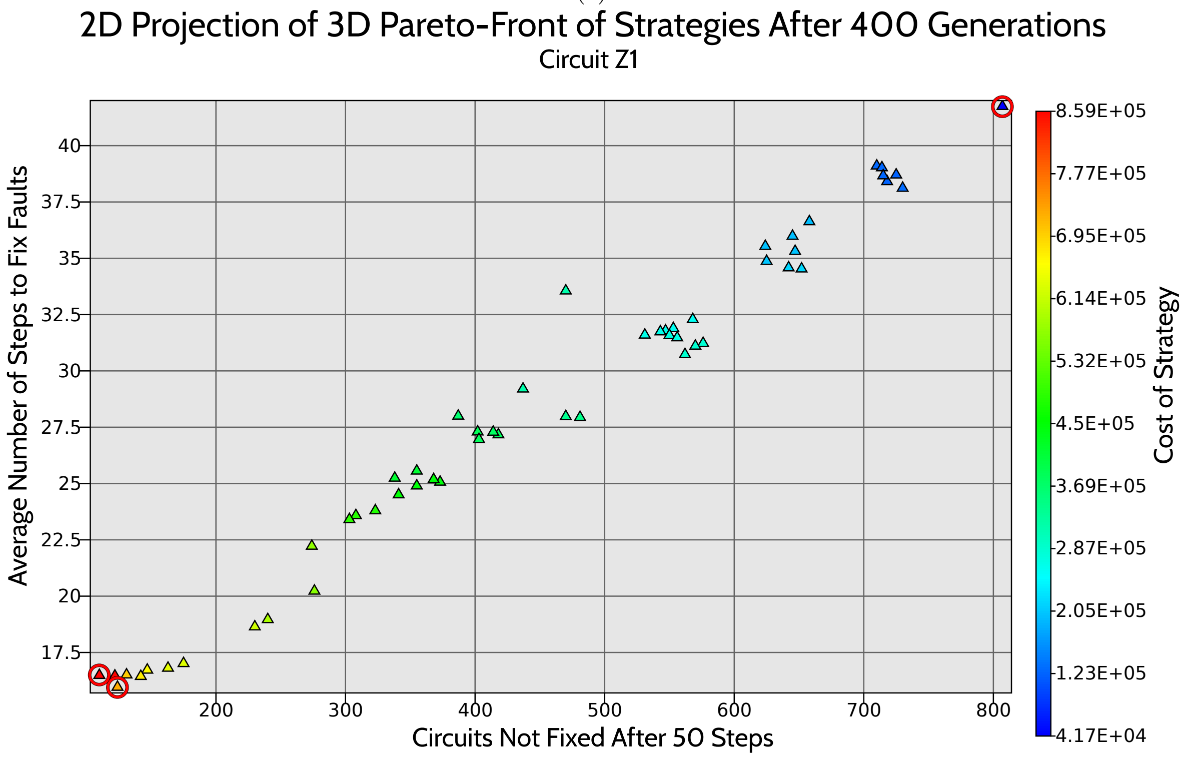
Table 9.1 presents twelve evolved Strategy lists. Three lists are presented for each of the four circuits, the optimal solution found for each objective individually. Appendix C provides a full details of the mapping between the numbers in the lists and the circuit transformations the represent.

Common Features of Evolved Lists

The most obvious trend in the evolved strategy lists is the low strategy numbers (<24) in the Cost-optimised lists, meaning that all the strategies are Input swaps. This is an intuitive result indicative of the experimental setup performing as designed. The cost associated with a Branch Swap was 100 compared with 1 for an Input Swap, meaning that the addition of just one Branch Swap would have significantly raised the average cost.

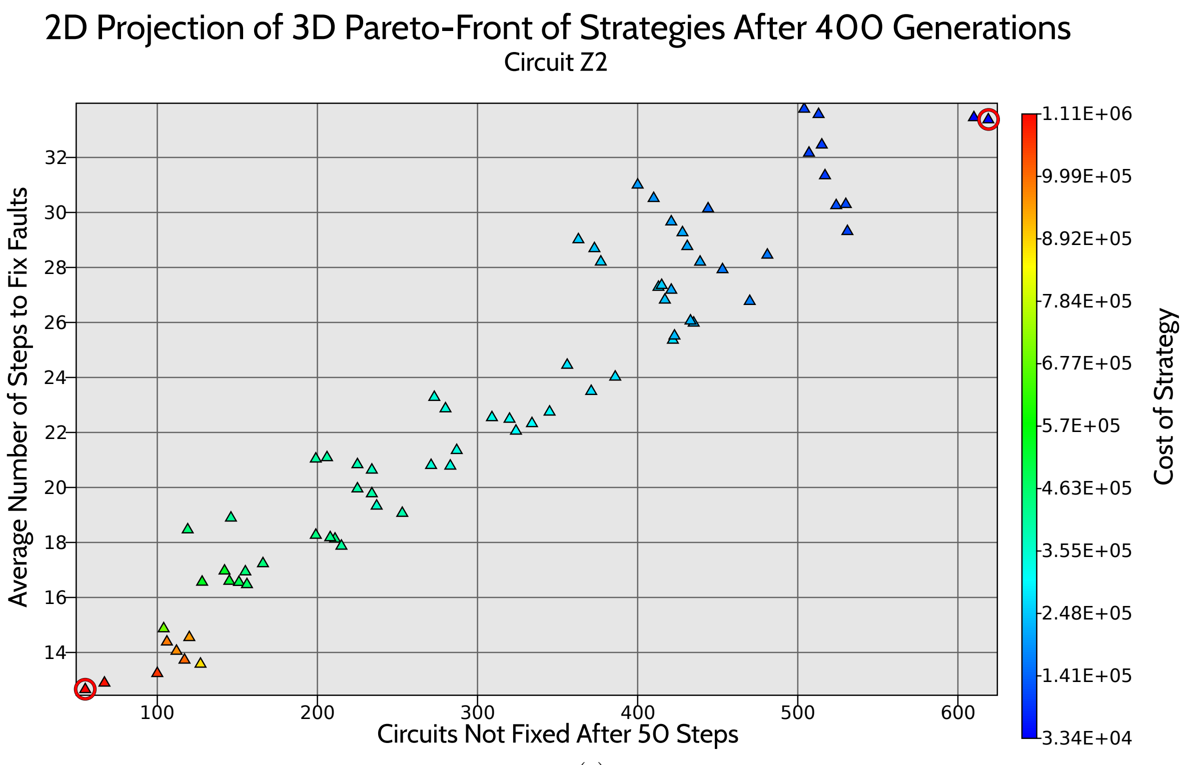

2D Projection of 3D Pareto-Front of Strategies After 400 Generations
Circuit Z0
Average Number of Steps to Fix Faults
Circuits Not Fixed After 50 Steps
Average Cost of Strategy
3.19E+05
2.91E+05
2.63E+05
2.35E+05
2.07E+05
1.79E+05
1.51E+05
1.23E+05
9.44E+04
6.64E+04
3.83E+04

(a)

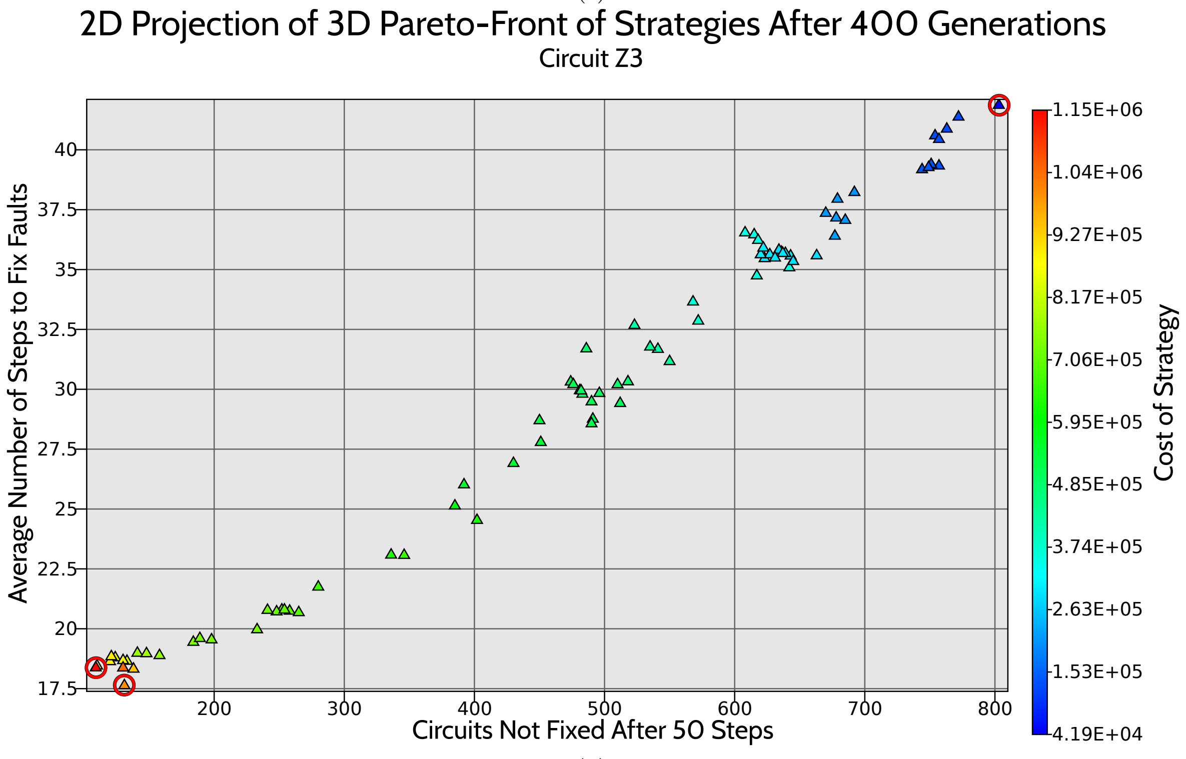

2D Projection of 3D Pareto-Front of Strategies After 400 Generations
Circuit Z1
Average Number of Steps to Fix Faults
Circuits Not Fixed After 50 Steps
Cost of Strategy
8.59E+05
7.77E+05
6.95E+05
6.14E+05
5.32E+05
4.5E+05
3.69E+05
2.87E+05
2.05E+05
1.23E+05
4.17E+04

(b)

Figure 9.4



(c)



(d)

Figure 9.4: The results of evolving strategy lists for circuits (a) Z0, (b) Z1, (c) Z2 and (d) Z3 after 400 generations. The solutions that are optimised for the number of unfixed circuits, mean number of steps and cost are circled in red.

Table 9.1: Strategy lists optimised for each of the three objectives, one for each circuit. Compare with the Strategy number mapping in Appendix C.

	Unfixed				Steps				Cost			
Index	Z0	Z1	Z2	Z3	Z0	Z1	Z2	Z3	Z0	Z1	Z2	Z3
0	2	44	61	45	2	42	61	45	2	17	3	5
1	55	15	44	75	55	12	44	23	4	11	5	1
2	17	7	6	78	2	7	6	78	0	7	6	18
3	5	78	74	2	5	78	74	2	4	6	0	0
4	74	16	63	74	74	17	63	74	1	21	2	16
5	3	67	59	36	3	67	59	1	0	5	7	8
6	26	9	33	2	26	9	33	3	0	1	1	21
7	6	66	30	66	5	65	30	66	2	10	3	6
8	10	3	76	53	10	3	76	53	4	0	1	3
9	12	37	35	37	12	37	35	37	5	6	9	4
10	37	5	53	4	37	13	53	5	0	0	13	5
11	56	55	74	13	56	54	74	46	2	3	3	14
12	31	25	71	25	31	25	71	25	6	16	1	13
13	1	7	45	57	1	2	45	57	0	12	10	12
14	3	24	48	10	3	24	48	5	0	0	0	0
15	1	57	57	48	1	57	57	51	1	9	0	2
16	5	30	76	22	5	27	76	72	2	5	1	18
17	18	21	33	47	19	21	33	48	1	20	2	16
18	3	70	8	70	3	69	8	75	0	4	6	3
19	7	72	68	14	7	70	68	16	0	8	3	20
20	33	33	74	3	33	3	74	4	1	0	13	2
21	5	11	25	3	3	13	25	5	1	10	4	4
22	53	74	72	72	53	75	72	76	1	12	4	6
23	59	8	59	7	59	8	59	7	10	8	1	1
24	5	60	34	27	5	9	34	27	3	2	1	4
25	1	44	61	47	1	47	61	47	0	11	12	13
26	5	2	75	35	3	5	75	38	4	4	8	2
27	0	8	38	4	0	2	38	3	0	1	5	0
28	6	64	75	7	6	61	75	5	0	13	7	2
29	3	2	67	48	1	9	67	49	3	2	2	2
30	18	35	52	13	17	8	52	13	1	1	2	7
31	7	9	6	60	8	11	6	60	0	10	7	10
32	1	7	53	34	1	8	53	34	0	0	0	1
33	48	5	23	59	48	3	23	54	0	2	9	8
34	7	69	9	0	7	0	9	2	1	5	3	6
35	6	3	12	3	6	3	12	3	1	0	0	4
36	51	2	65	0	53	3	65	1	1	0	2	1
37	75	74	31	73	77	75	31	0	1	1	1	13
38	28	2	70	2	28	1	70	2	1	1	5	3
39	7	23	3	24	7	0	3	4	2	0	5	3
40	72	5	55	1	75	4	55	1	0	2	6	4
41	69	60	66	63	74	8	66	74	0	1	3	15
42	67	0	77	4	69	1	77	4	0	0	12	0
43	4	68	59	76	1	21	59	75	4	8	2	8
44	44	15	13	1	49	14	13	1	0	16	1	9
45	8	3	4	0	8	3	4	0	0	6	2	2
46	1	72	25	60	1	11	25	60	0	6	0	1
47	57	0	73	5	57	0	73	5	0	2	0	18
48	45	8	50	19	48	2	50	18	5	3	3	20
49	2	4	76	2	2	4	76	3	0	4	3	3

List Work-Through

As an example of what the optimisation algorithm has come up with, the first eight steps of the "Unfixed"-optimised list for Z0 are described to illustrate the kind of reconfiguration sequence that the algorithm can devise. This list is presented in full in the leftmost data column of Table 9.1. Circuit Z0 is the inverter circuit with one PMOS and one NMOS CT on Input 0 of branches PMOS 0 and NMOS 0 (see Figure C.1 for an explanation of these numbers). The original circuit diagram can be found in Figure 8.2 (page 121). A visualisation the first five steps reconfigurations that this strategy list makes is presented in Figure 9.5.

0. Strategy 2 - Inputs 0 and 3 are swapped on MiniCAB 0. This is a sensible move as the two CTs used in the circuit are connected to input 0. If either had suffered just a conducting fault, this would recover from it.

1. Strategy 55 - NMOS 0 is swapped with NMOS 4. Again, this is sensible given the implemented circuit. The NMOS branch from the first MiniCAB is moved to the 3rd MiniCAB from the left. If the NMOS branch had suffered an insulating fault this would be a workaround. Though the NMOS CT inputs have been swapped on MiniCAB 0 in the first strategy, MiniCAB 2's inputs will still be in the same state and so the CT configurations on NMOS 0 will be reordered to match.

2. Strategy 17 - Inputs 2 and 3 are swapped on MiniCAB 2. This is essentially a NOOP as the four CTs connected to these inputs are all disabled. Though the NMOS Branch has been moved to MiniCAB 2, it is attached to Input 0 as MiniCAB 2 still has the default input order. This shows that this list has potential for further improvement.

3. Strategy 5 - Inputs 2 and 3 are swapped on MiniCAB 0. This moves the active CT on PMOS 0 to Input 2, potentially recovering from a conducting fault on Input 3 of PMOS 0.

4. Strategy 74 - NMOS 4 is swapped with NMOS 5. This move seems to be good evidence that the optimisation algorithm is successful in finding effective reconfigurations, given that it moves the only active NMOS branch to another that has not yet been tried.

5. Strategy 3 - Inputs 1 and 2 are swapped on MiniCAB 0, moving the PMOS CT on PMOS 0 to Input 1. This PMOS CT has now been tried in all four positions on this Branch.

6. Strategy 26 - PMOS 0 is swapped with PMOS 3. This is again an effective reconfiguration to make given that the previous strategy tried the active CT in the last possible position on PMOS 0 and so if the circuit still doesn't work the only option left is a Branch swap. The active CT was previously on Input B but will be translated to Input A when it moves to PMOS 3 due to the Input order of MiniCAB 1 not yet changing.
7. Strategy 6 - Inputs 0 and 1 are swapped in MiniCAB 1. After moving the PMOS CT to a Branch in MiniCAB 1 in the previous step, the Inputs are now swapped.

Up until this point, this strategy list appears to have evolved in a sensible way. Apart from the the third strategy, each reconfiguration has the potential to overcome a fault, and puts the circuit into a state that hasn't been tried before. This is the obvious approach one would take to finding a working circuit on a faulty substrate where the location and nature of the fault is unknown. This result proves that ordered lists of strategies that are effective for overcoming unknown faults can be generated automatically. This suggests that they could be generate for any given circuit, either at the time that the architecture is designed or at compile time for a particular circuit.

Number of Faults Fixed

As with the methodology in the Chapter 8, the number of faults that could be fixed in each circuit was measured. The results of these measurements are presented in Figure 9.6 and should be compared with Figure 9.7 rather than Figure 8.10 because the limit on the number of steps was reduced to 50 to match the newer experiments.

The newer method is clearly more successful that the older one, with all single faults being repaired for all circuits. The results from Figure 8.10 are more comparable with these new results, but this was through the use of up to 200× more reconfigurations.

9.5.2 Stochastic Strategies Experiment

For comparison, the method from Chapter 8 was re-tested (with differences explained in Section 9.4.3) on the same four circuits. The new results are presented in Figure 9.8.

The general trend for Figures 9.8a and 9.8b is that the average number of steps taken to recover from faults and the number of unfixed circuits left after the run decrease as the bias tends towards the Branch Swapping strategy. Figure 9.8c shows the opposite trend in that

Index	Z0	
0	2	On MiniCAB 0 (leftmost) swap inputs 0 (A) and 3 (D).
1	55	Swap NMOS Branch 0 (left, MiniCAB 0) with NMOS Branch 4 (left, MiniCAB 2).
2	17	On MiniCAB 2 swap inputs 2 (C) and 3 (D).
3	5	On MiniCAB 0 swap inputs 2 (C) and 3 (now A).
4	74	Swap NMOS Branch 4 (left, MiniCAB 2) with NMOS Branch 5 (right, MiniCAB 2).

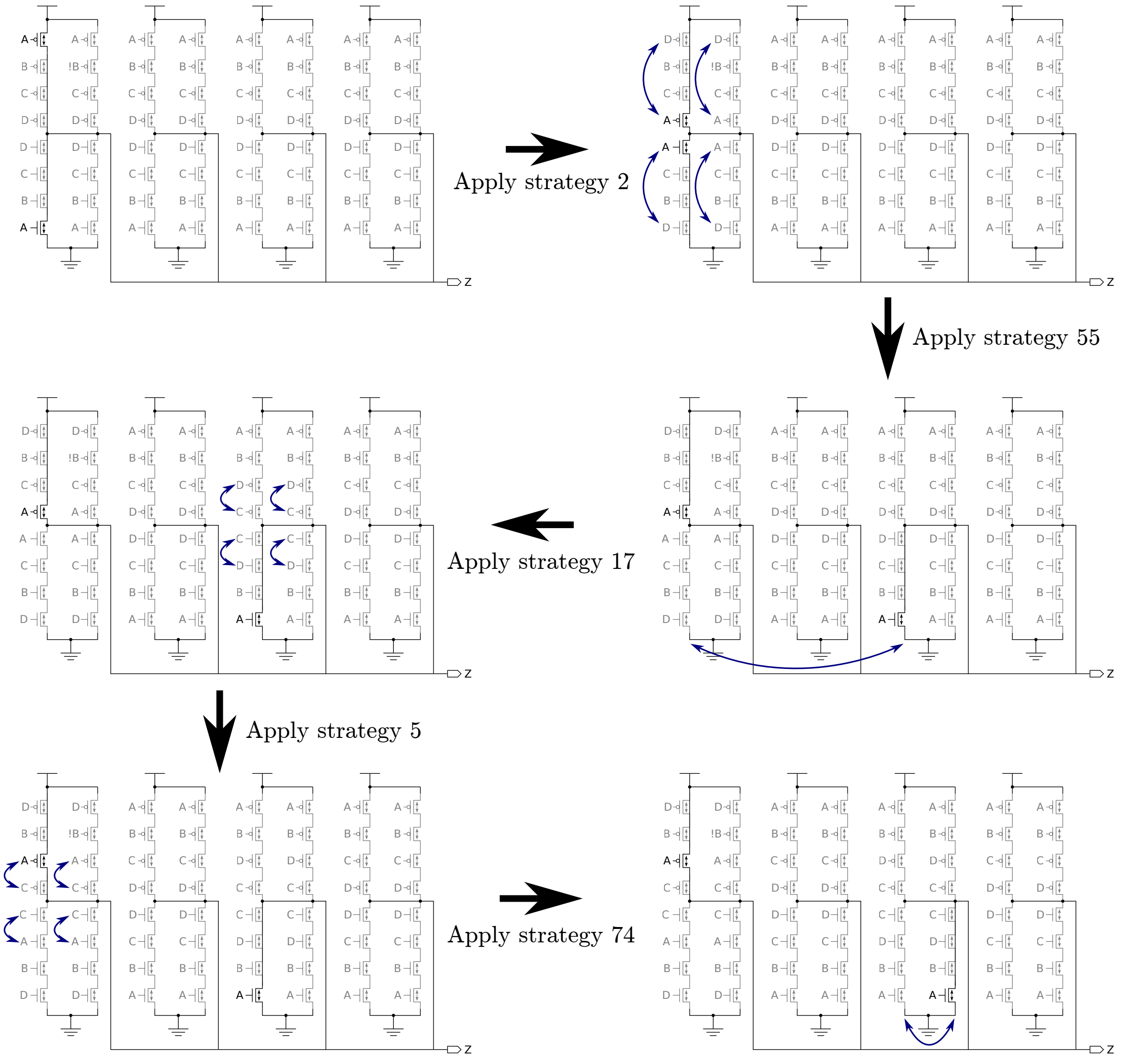


Figure 9.5: A visualisation of the first five steps of a strategy list applied to circuit Z0 (an inverter). Note that apart from the third strategy applied (index 2), all the strategies transform an active part of the circuit, suggesting that the evolutionary approach to generating the list converges towards a sensible outcome. A complete definition of the set of strategies can be found in Appendix C.

Table 9.2: A comparison of solutions from both methods optimised for the least unfixed circuits out of 1000.

	Solutions Optimised for Least Unfixed			
Circuit		Stochastic	Deterministic	Improvement
Z0	Unfixed	113	43	61.95%
	Steps	16.973	8.2	51.69%
	Cost	1629485	317575	80.51%
Z1	Unfixed	256	110	57.03%
	Steps	24.571	16.487	32.90%
	Cost	2457100	858779	65.05%
Z2	Unfixed	150	55	63.33%
	Steps	19.918	12.655	36.46%
	Cost	1850032	1106210	40.21%
Z3	Unfixed	383	109	71.54%
	Steps	31.991	18.374	42.57%
	Cost	3135740	1148560	63.37%

the average cost decreases as the bias tends towards the Input Swapping strategy. This is intuitive as the Branch Swapping strategy was intentionally made much more expensive than the Input Swapping strategy.

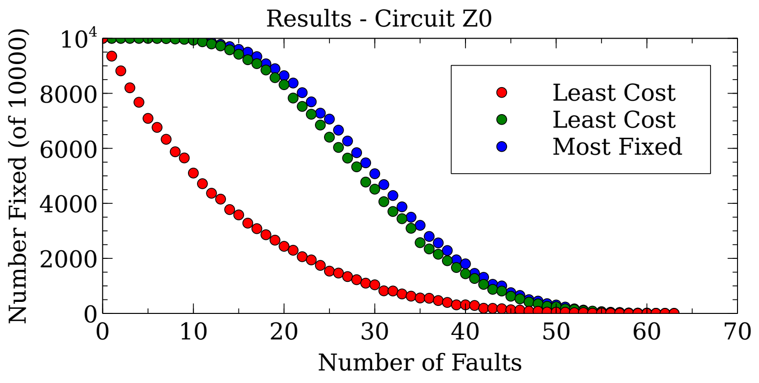
9.5.3 Comparison

Unfixed Circuits

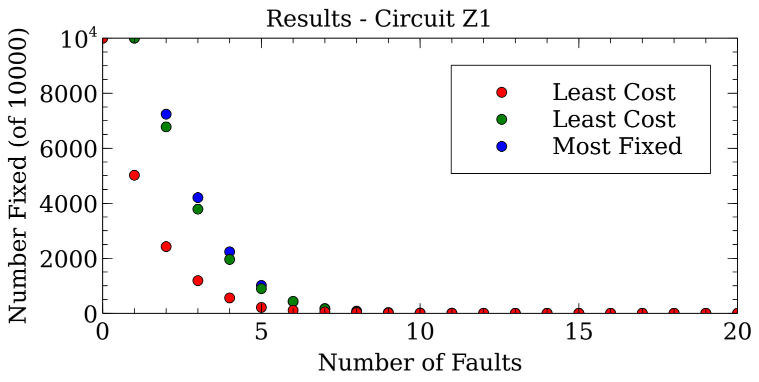
The number of circuits that were not fixed in the best cases of each approach is significantly improved by the new method (see Table 9.2). The improvement varies from between 57.03% and 71.54% in terms of the number of circuits, and the other objectives are also improved in these cases as well. This means that for the given number of steps, the deterministic method is able to recover more circuits from faults.

Number of Steps

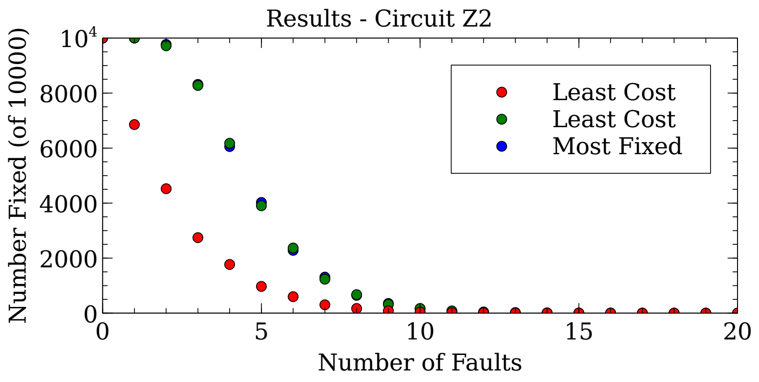
The deterministic method has decreased the average number of steps taken when fixing faults by between 35.04% and 52.2% (see Table 9.3). This means that if a fault is able to be fixed, the deterministic method will find the fix quicker, on average.



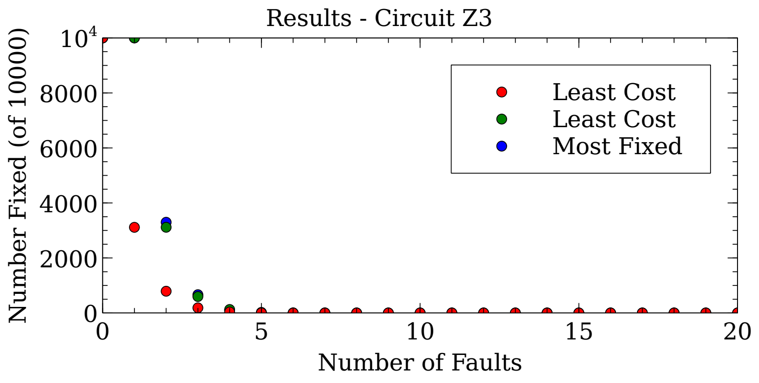
(a)



(b)

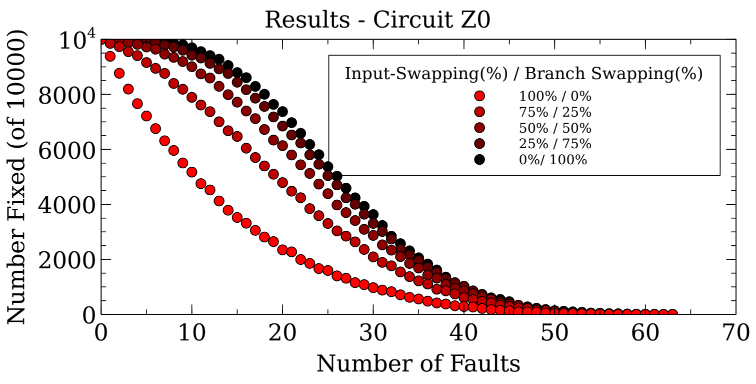


(c)

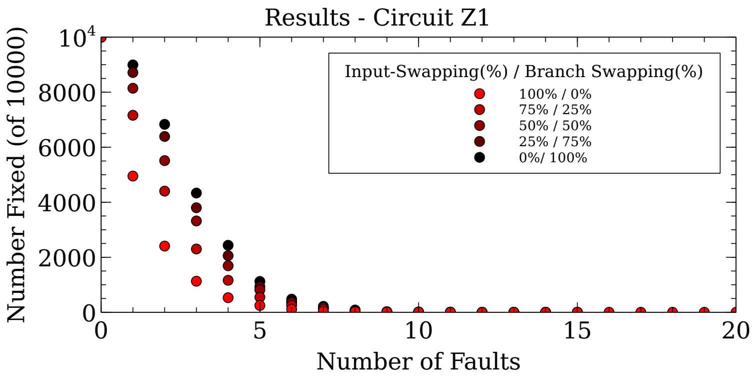


(d)

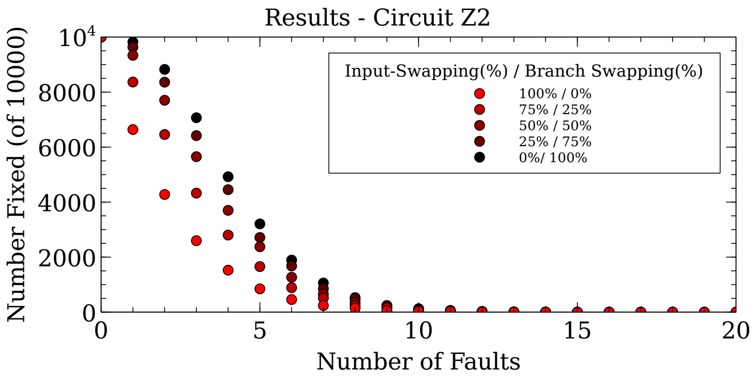
Figure 9.6: The number of circuits fixed (out of 10000 samples) by the new strategy lists when increasing numbers of random faults are introduced into circuits (a) Z0, (b) Z1, (c) Z2 and (d) Z3. Results are presented for each of the three optimised lists described in Table 9.1. The graphs for circuits Z1, Z2 and Z3 were truncated to 20 faults as no more circuits were fixed after this point.



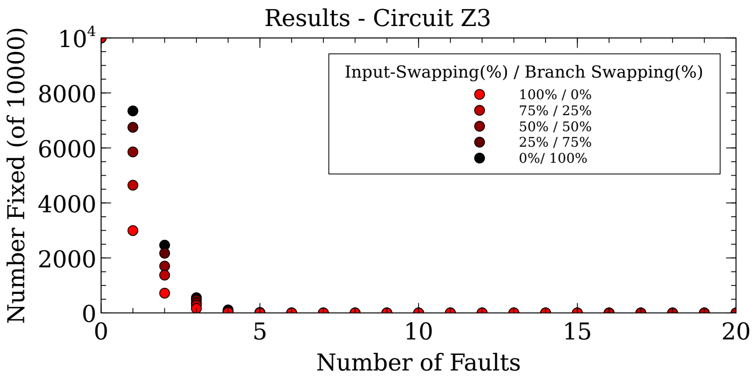
(a)



(b)

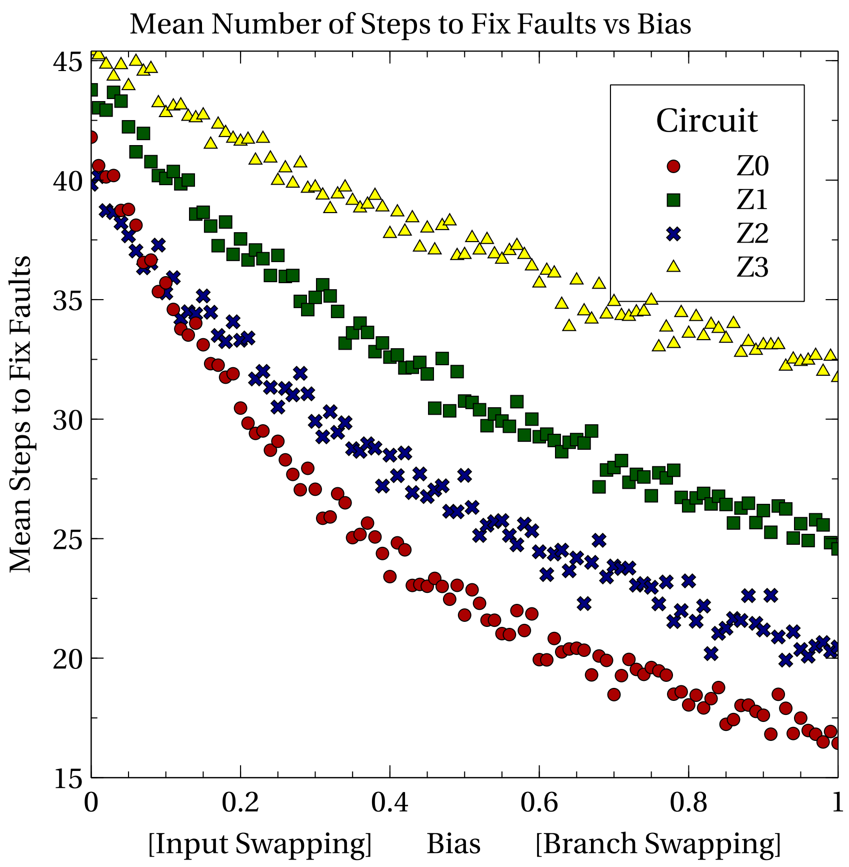


(c)



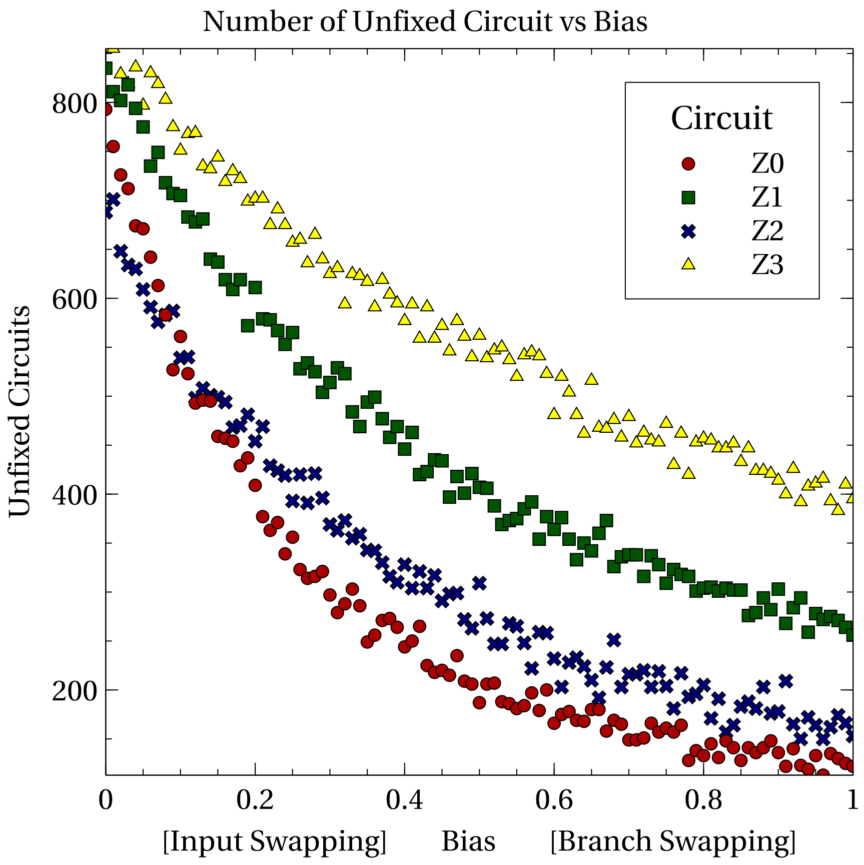
(d)

Figure 9.7: The number of circuits fixed (out of 10000 samples) by the methodology from Chapter 8 when increasing numbers of random faults are introduced into circuits (a) Z0, (b) Z1, (c) Z2 and (d) Z3. Results are presented for five bias values and used the random Input and Branch swapping strategies as opposed to shuffling. For a direct comparison with Figure 9.6 the limit on the number of steps was reduced from 10000 to 50. The graphs for circuits Z1, Z2 and Z3 were truncated to 20 faults as no circuits were fixed after this point.



Mean Number of Steps to Fix Faults vs Bias
Mean Steps to Fix Faults
45
40
35
30
25
20
15
0
0.2
0.4
0.6
0.8
1
[Input Swapping]
Bias
[Branch Swapping]
Circuit
Z0
Z1
Z2
Z3

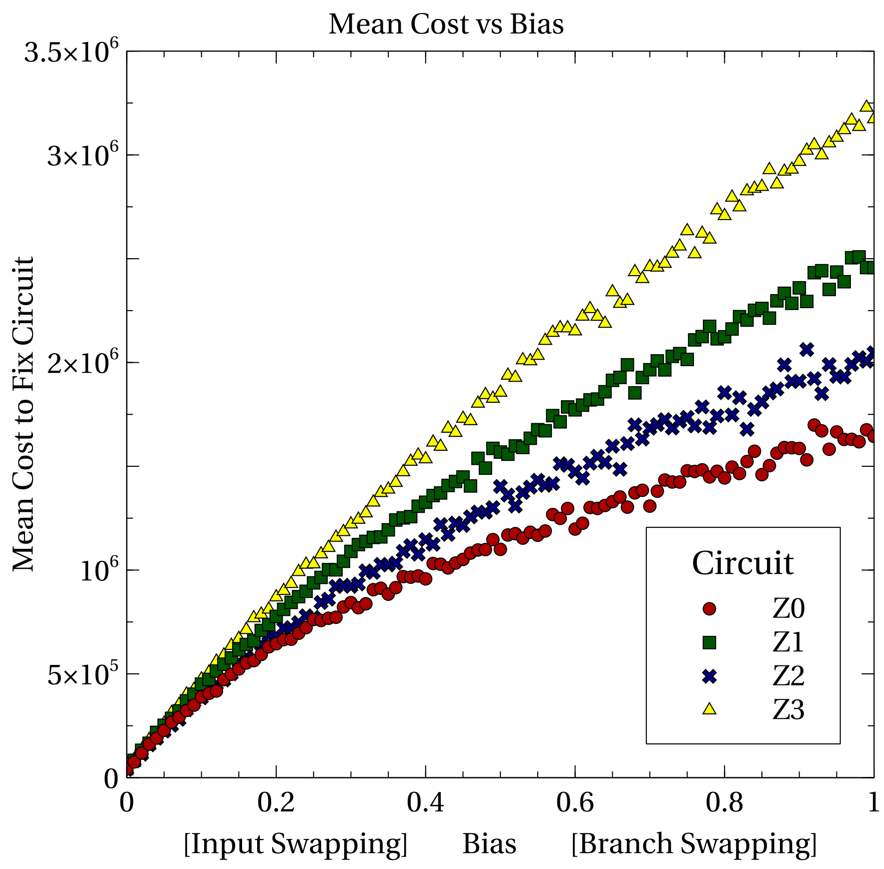
(a)



Number of Unfixed Circuit vs Bias
Unfixed Circuits
800
600
400
200
0
0.2
0.4
0.6
0.8
1
[Input Swapping]
Bias
[Branch Swapping]
Circuit
Z0
Z1
Z2
Z3

(b)

Figure 9.8



(c)

Figure 9.8: Results of running the experiment from Chapter 8 with the following differences (Section 9.4.3): the input swapping strategy performs only a single swap for each step; the total cost of each run is recorded; the number of unfixed (rather than the number fixed) circuits is reported and the average number of steps measurement includes cases where the circuit was not fixed. (a) The average number of steps taken to fix random faults. (b) The number of unfixed circuits out of 1000. (c) The total cost expended when using the particular bias.

Cost Expended

The new cost measurement shows less improvement than the other optimised solutions (see Table 9.4). The deterministic solution presented shows a small improvement in cost, between 4.64% and 16.2%, and similarly sized improvements across the other objectives. In all the stochastic and deterministic methods, the Steps measurement is 1000^{th} of the Cost measurement. This is because in each case, the solutions used only Input Swapping strategies which have a cost of 1. It's possible that setting the costs of each strategy type closer together may have changed this situation and provided more useful solution.

Table 9.3: A comparison of solutions from both methods optimised for the least number of steps taken during 1000 circuit repairs.

		Solutions Optimised for Least Steps		
Circuit		Stochastic	Deterministic	Improvement
Z0	Unfixed	122	48	60.66%
	Steps	16.44	7.859	52.20%
	Cost	1644000	311987	81.02%
Z1	Unfixed	256	124	51.56%
	Steps	24.571	15.961	35.04%
	Cost	2457100	720742	70.67%
Z2	Unfixed	150	55	63.33%
	Steps	19.918	12.655	36.46%
	Cost	1850032	1106210	40.21%
Z3	Unfixed	395	131	66.84%
	Steps	31.73	17.627	44.45%
	Cost	3173000	992381	68.72%

Table 9.4: A comparison of solutions from both methods optimised for the least cost expended after 1000 circuit repairs.

		Solutions Optimised for Least Cost		
Circuit		Stochastic	Deterministic	Improvement
Z0	Unfixed	793	760	4.16%
	Steps	41.798	38.309	8.35%
	Cost	41798	38309	8.35%
Z1	Unfixed	835	807	3.35%
	Steps	43.778	41.745	4.64%
	Cost	43778	41745	4.64%
Z2	Unfixed	688	619	10.03%
	Steps	39.826	33.375	16.2%
	Cost	39826	33375	16.2%
Z3	Unfixed	855	803	6.08%
	Steps	45.398	41.863	7.79%
	Cost	45398	41863	7.79%

9.6 Analysis

The results have shown that the deterministic method is able to perform better than the stochastic method in every situation tested. It is hypothesised therefore that this would be the case for every function (other than the two mentioned in Section 8.2). This follows since the stochastic method is likely to make many unnecessary and unhelpful reconfigurations, whereas it is undoubtedly possible to optimise the deterministic method so that it always makes a reconfiguration which moves the circuit to a different implementation and potentially fixes a fault.

9.7 Summary

The experiments reported in this chapter built upon the ones in Chapter 8 by taking all the possible circuit transformations that could be performed by the random swapping strategies and enumerating them into a dictionary of deterministic strategies. An evolutionary algorithm was then used to build ordered lists of these strategies which were optimised for finding working circuit configurations after a circuit failed due to faults.

It was shown that applying ordered lists of strategies provides a better performing fault tolerance methodology than a random application, even when the source of the fault is unknown. It appears that the evolutionary process is able to find sequences of non-destructive circuit transformations that move a specific circuit configuration through a series of functionally identical layouts which have a better than average probability of working given a random set of faults being present.

They have also shown that it is possible to optimise lists for specific circuit designs in terms of the time it takes to fix faults, or trade this off for lower disruption to the circuit configuration. A comparison between this new methodology and the one presented in Chapter 8 showed that the efficiency of deterministic method allows more faults to be fixed with fewer reconfigurations.

A cost measurement was added to the experiments which assigned a higher cost to the Branch Swapping strategies than the Input Swapping strategies given that Input Swapping is less disruptive to the layout of the circuit and is thus assumed to be more desirable. This is similar to the bias value used in the experiments in Chapter 8 in that it encourages one type of strategy over the other. The experiments from Chapter 8 were rerun with parameters matching the new experiments and the same cost measurement was taken. When comparing the two methods based on the results with the lowest cost (and so making use mainly if not entirely Input Swapping) the deterministic method still showed a small but consistent improvement over the stochastic method.

The deterministic method requires a specially optimised list for each circuit configuration and the time it takes for the lists to evolve makes it infeasible to produce them on the fly. This is due to the millions of simulations required during the evolutionary process. In contrast, the stochastic method can be used on any circuit with no preparation. One solution to this may be to compute the strategy lists in advance and to store them in memory so that they're available in the event of a fault. It seems feasible that a manufacturer could do this for every possible function.

Overall, these experiments have demonstrated a novel method for finding working circuits on a faulty reconfigurable device that performs better than random reconfigurations.

The following chapter presents some conclusions garnered from the work reported in this thesis and also some thoughts on how the findings could be used as a basis for further work.

Chapter 10

Conclusions and Further Work

Contents

10.1 Summary of Work

The increase in the number of issues surrounding integrated circuit fabrication, as outlined in Chapter 2, motivated members of the Intelligent Systems research group at the University of York to design a novel reconfigurable architecture, PAnDA (detailed in Chapters 4 & 6). By introducing multiple layers of configurable components, PAnDA allows for the exploration of digital circuits with different analogue properties, in hardware.

PAnDA's architecture intentionally allows for multiple implementations of the same circuit for the purposes of exploring the effects of transistor variability, but this also creates potential opportunities for fault tolerance. This thesis has presented research into using the PAnDA architecture for finding novel fault tolerant methodologies. Hypothesis 1 was the main focus of the work.

Hypothesis 1 *It is possible to include reconfiguration mechanisms exploiting both the analogue and digital abstraction layers of the PAnDA architecture in order to provide fault tolerance in devices.*

In working to explore Hypothesis 1, a number of sub-hypotheses were investigated. The work in Chapter 5, undertaken in collaboration with another PhD student, looked at Hypothesis 2.

Hypothesis 2 *Removing the function decoder block from the PAnDA-Eins architecture provides benefits for fault tolerance.*

Hypothesis 2 was found to be true and results were presented that showed fault mitigation was made possible by the removal of the function decoder. These findings contributed to a design change which was realised in PAnDA-Zwei, outlined in Chapter 6.

To fully explore the new design a bespoke simulator was developed, capable of simulating the digital function of a configured transistor circuit including the effects of conducting or insulating faults (Chapter 7). As part of this, a method of synthesizing transistor circuits for the PAnDA-Zwei architecture was developed, which enabled automatic configuration of any four-input logic function on a PAnDA-Zwei Slice. This made the process of configuring logic functions on PAnDA-Zwei Slices as straightforward as it is on traditional FPGA LUTs but with the additional ability to create tri-state circuits.

PAnDA-Zwei allows for configuration of any circuit in multiple ways due to increased homogeneity in the structure when compared to PAnDA-Eins. This feature inspired the investigation of Hypothesis 3, based on the idea of transformations to the configuration bitstream which would change the implementation of a circuit blindly, but without affecting the functionality. In this way an algorithm can move a configuration around the space of functionally equivalent circuits without needing knowledge of the current configuration.

Hypothesis 3 *Applying transformations to a circuit configuration at random finds working circuits on a faulty substrate.*

Chapter 8 demonstrated Hypothesis 3 to be true by applying such transformations randomly until a working circuit was found. The use of these transformations improves upon the random reconfigurations used in Chapter 5 as only valid circuits need ever be evaluated. Through the use of high and low level reconfigurations (Branch swapping and Input swapping respectively) in varying ratios, it was observed that the high level reconfigurations are better at fixing faults.

The random approach to applying circuit transformations lead to consideration of how to reduce unnecessary transformations and to the question of whether a non-random application could help at all. Hypothesis 4 was proposed and investigated to find out.

Hypothesis 4 *Applying circuit transformations deterministically can find working circuits on a faulty substrate more efficiently than a random application.*

In Chapter 9, all possible circuit transformations were enumerated and numbered so that they could be put into deterministic sequences and applied to faulty circuits. The sequences were optimised using an evolutionary algorithm in an attempt to automatically find the best order of reconfigurations to try for a given circuit when it had sustained one or more faults. The results of this experiment showed significant improvements over the previous method (55% on average) in terms of how successful the approach was in finding a fix and also how quickly a fix was found.

Overall, the results reported in this thesis demonstrate Hypothesis 1 is true. Novel reconfiguration mechanisms have been presented that exploit both the analogue (Chapter 5) and digital (Chapters 8 & 9) abstraction layers of the PAnDA architecture and provide fault tolerance for a device.

10.2 Contribution to Knowledge

The work in this thesis has produced a novel hierarchical fault tolerance methodology which enables fault mitigation whilst minimising disruption to circuit topology.

The methodology was shown to be effective in finding reconfigurations that worked around faults without any knowledge of the fault's nature or location.

It was also found that a sequence of reconfigurations could be optimised for a particular circuit, which improved the probability of finding a workaround for a fault and also reduced the average number of reconfigurations required to find one. This was demonstrated using lists (representing sequences of reconfigurations) optimised for particular circuits.

10.3 Evaluation

The fault tolerance methodology presented in this thesis has many advantages over more traditional methods, for example TMR, in the context that has been presented, but taking advantage of it in real world applications would require some additional consideration.

10.3.1 Low Overhead

Where TMR requires three instances of the same circuit, the proposed methodology requires only that there be unused resources available in a reconfigurable architecture. This is typically the case in PAnDA-Zwei since only a small percentage of logic functions require all the CTs in a Slice. Some additional memory may also be required for the deterministic method to store the strategy lists for each function used in a design, though this is implementation specific.

Though the proposed methodology has a lower overhead than TMR, it does not possess the built-in fault masking ability of modular redundancy meaning that a fault would make part of a circuit behave incorrectly or become unavailable while the fault recovery procedure is performed. This is perhaps countered somewhat by the procedure's ability to overcome multiple faults whereas TMR can only handle one.

Applying the hierarchical techniques to circuits on standard FPGAs would require that there be LUTs that are either unused or only partially used in order for there to be spare resources available for parts of designs to be swapped on to. In general, LUT utilisation is limited by routing [88][89] and so it is difficult to produce efficient designs that use all available resources. It is therefore likely that most FPGA circuits contain some spare resources that could be exploited using the techniques described in this thesis.

10.3.2 Low Circuit Disruption

One of the strengths of the hierarchical methodology is that changes will be kept at a low level where possible to avoid excessive disruption to higher level structures. The advantage of this is that changes to routing and expensive place-and-route operations can be kept to a minimum. Another benefit of keeping the circuit layout as similar as possible to the original design is that timing and power aspects of the circuit are affected as little as possible. When TMR recovers from a fault there is no disruption to the circuit structure, but this is because there are already complete redundant circuits.

10.3.3 Potential Damage

One drawback of making reconfigurations without knowing the nature of any faults is that the device may inadvertently be put into a short-circuit state if a conducting fault has occurred. This could occur after the fault but before any reconfigurations anyway, and so there is reason to believe it would not cause an instant problem. With TMR this risk only occurs if the fault itself causes a short.

10.3.4 Fault Recovery not Guaranteed

When beginning the algorithm, there is no way of knowing whether it will succeed in finding a solution. It is possible to estimate however based on how many step it takes on average to find a fix. After a fault has occurred there is therefore a trade-off in power limited situations regarding how likely it is that a fix will be found for the circuit that has become faulty. In this case, TMR is more reliable in terms of knowing that functionality can be maintained after the first fault, but after a second there is no opportunity to continue. The methodology presented in this thesis has the potential to keep a circuit working so long as there are sufficient spare resources.

10.4 Further Work

10.4.1 Extending the Hierarchy

So far, this hierarchical approach to fault tolerance has extended to only two levels of the PAnDA configuration hierarchy. In order to more thoroughly measure the performance of the hierarchical approach, it is necessary to add some additional levels into the mix of strategies to investigate whether this provides better performance.

CLB Level

The next level up from Slice level is the CLB level. Since a CLB contains two Slices, it is straightforward to try swapping the two Slices over and this could be added as an additional strategy number with a cost significantly higher than Branch swapping (Figure 10.1). Is is assumed that the implementation of the routing block inside the CLB is flexible enough to support either configuration.

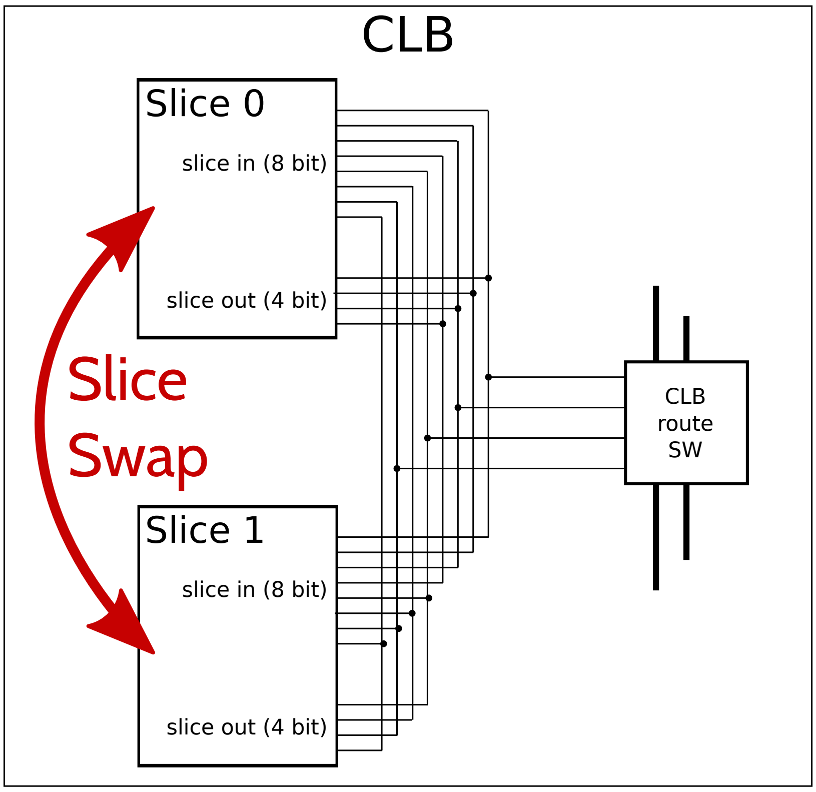


Figure 10.1: An illustration of how two Slice configurations would be swapped in a PAnDA-Zwei CLB (based on a figure from [70]).

Chip Level

A PAnDA-Zwei Chip contains CLBs. So long as the routing is flexible enough, the configurations for any two CLBs may be arbitrarily swapped around, perhaps finding a configuration where the circuits configured on a faulty CLB are fully functional (Figure 10.2). This operation could also be added as an additional strategy number with a cost significantly higher than Slice swapping. The cost of the swaps could increase with the distance between the two CLBs being swapped to encourage local swaps and minimise disruption to routing.

Chip Array Level

Taking the idea to its logical conclusion, an array of PAnDA chips could swap configurations around (Figure 10.3). The array could be composed of either redundant chips or chips with different functions as the process for repair would be the same (swapping component configurations and seeing if it works). If three chips were used where two were redundant, more fault tolerance could be achieved than with TMR since potentially numerous minor faults in a single chip before a chip swap was required, as opposed to a single failure preventing any further use of a whole chip.

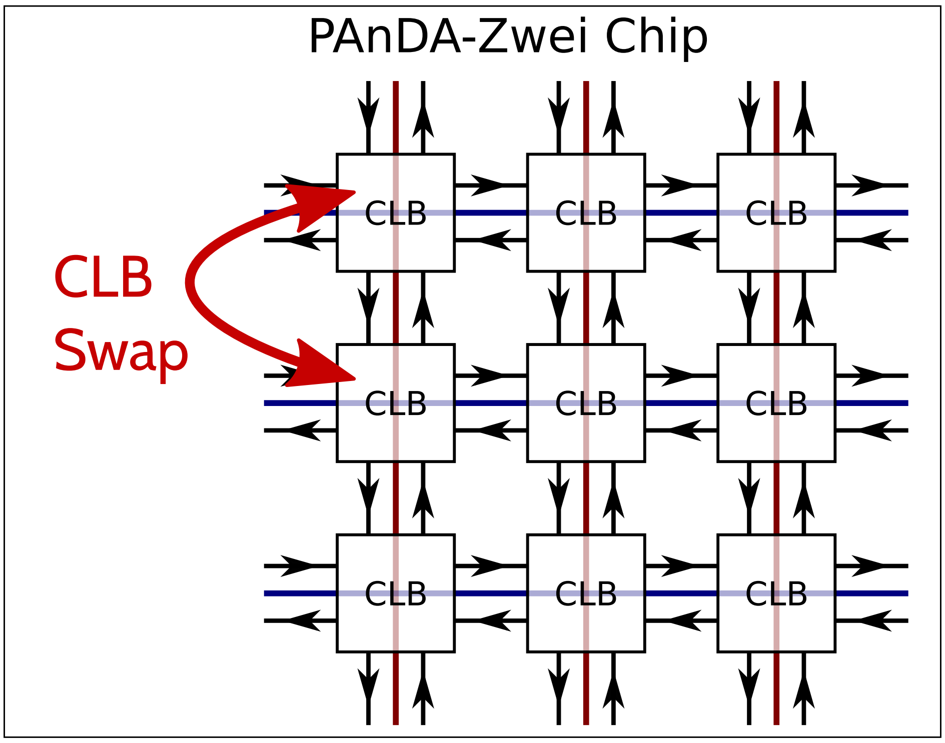


Figure 10.2: An illustration of how two CLB configurations would be swapped in a PAnDA-Zwei Chip. There is an assumption that the configurable routing in the final version would be sufficiently flexible to allow arbitrary CLBs to be swapped.

10.4.2 Routing

So far, routing has been assumed to be fault-free and sufficient for any reconfiguration of a particular circuit. It would be interesting to take routing faults into consideration in any future models. This might require certain higher level reconfigurations to take place in order to work around gaps in routing ability, but the same process of swapping and testing could be used.

10.4.3 Intermediate Levels

At the chip level there are potentially a large number of CLBs to swap around. Without any direction, this could cause a large amount of disruption to the circuit layout. One possible solution to this would be to break the chip down into groups of CLBs which would only be swapped with one another. The groups themselves could then be swapped as a whole in order to reach further configurations if necessary.

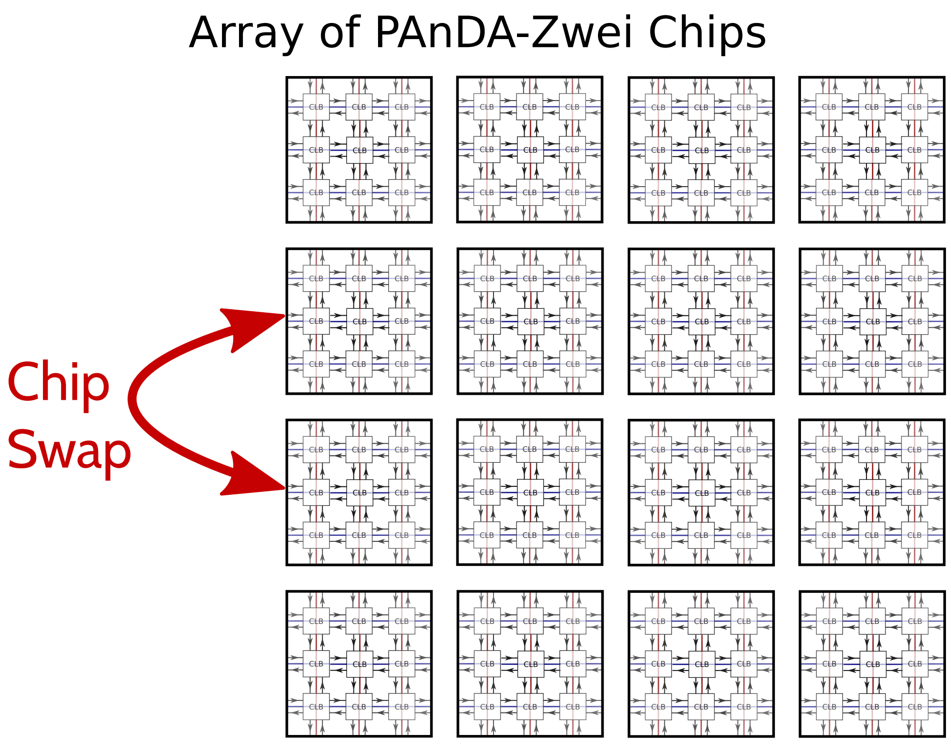


Figure 10.3: An illustration of how two whole Chip configurations would be swapped in an array of PAnDA-Zwei Chips. Certain circumstances may necessitate a high level of reliability for a long time so adding levels to the top of the configuration hierarchy allows the fault tolerance technique to scale further.

10.4.4 Scaleability

One issue with the proposed method for fault recovery is scalability. As the number of components involved in the technique increases, for instance when multiple CLBs are present, it quickly appears to become infeasible to find solutions to a hard problem, where there is perhaps one possible configuration that will work.

Chen's paper [90] proposes the use of dynamic programming, albeit at a much higher level of abstraction. By recording what configurations have been tried in which areas of the chip, certain known broken configurations can be skipped, reducing the time complexity of the fault mitigation algorithm. If a hash of the configuration bitstream that didn't work could be stored in a cache this could then be checked after each reconfiguration before testing whether or not it works.

10.4.5 FPGA Adaptation

PAnDA is an experimental platform which has leant itself well to the exploration of this new approach to fault recovery. It would be interesting, however, to apply the findings to a traditional LUT-based FPGA. There are hierarchical elements in these FPGAs for which swapping strategies could be developed:

- LUT inputs
- LUTs in a slice
- Slices in a CLB
- CLBs in a chip

If this could be done, the technique could be combined with Trimberger's work on multiple bitstreams [84] to improve the probability of finding a working configuration by attempting some fault mitigation on each bitstream before giving up and trying a different one.

10.4.6 Feedback

The current proposed method integrates little feedback except whether the circuit is working or not. It is possible that performance could be improved by feeding back any changes (or lack thereof) that occur to the output of a circuit after a change is made. For instance, if a particular input pattern produces an incorrect output and after a reconfiguration a different output is incorrect, it may be possibly to infer where the fault may lie. This information could then be used to decide on the best reconfiguration to try next. The strategy lists could become more like decision trees.

10.5 Real World Use

The real benefits of using the methodology described in Chapter 9 are that it provides an opportunity to make something work using any and all the resources available in a power efficient manner.

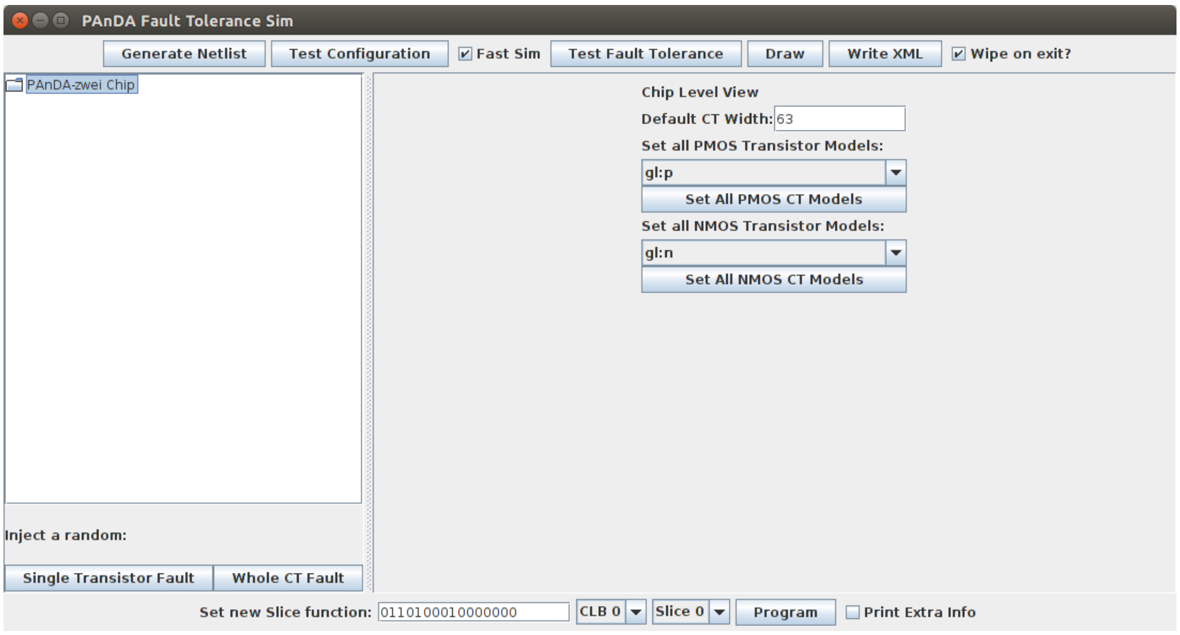
If a PAnDA chip was being used on a mission in space with no chance of physical access and limited power, and a fault occurred, strategy lists could be optimised remotely back on Earth where there is plenty of power and the trade-offs could be optimised for the specific situation. If there was not too much concern about power but there were not many spare resources on the chip due to other systems implemented on it, more time could be dedicated to trying very low level reconfigurations in an attempt to mitigate the fault. Alternatively if power was the main concern, the lists could be optimised to not spend too much time trying the low level reconfigurations and instead increase the probability of finding a working configuration by making higher level reconfigurations.

Appendix A

PAnDA Architecture Simulator

Contents

A.1 Graphical Interface


PAnDA Fault Tolerance Sim

✓ Fast Sim

✓ Wipe on exit?

PAnDA-zwei Chip

Chip Level View

Default CT Width: 63

Set all PMOS Transistor Models: gl:p

Set all NMOS Transistor Models: gl:n

Inject a random:

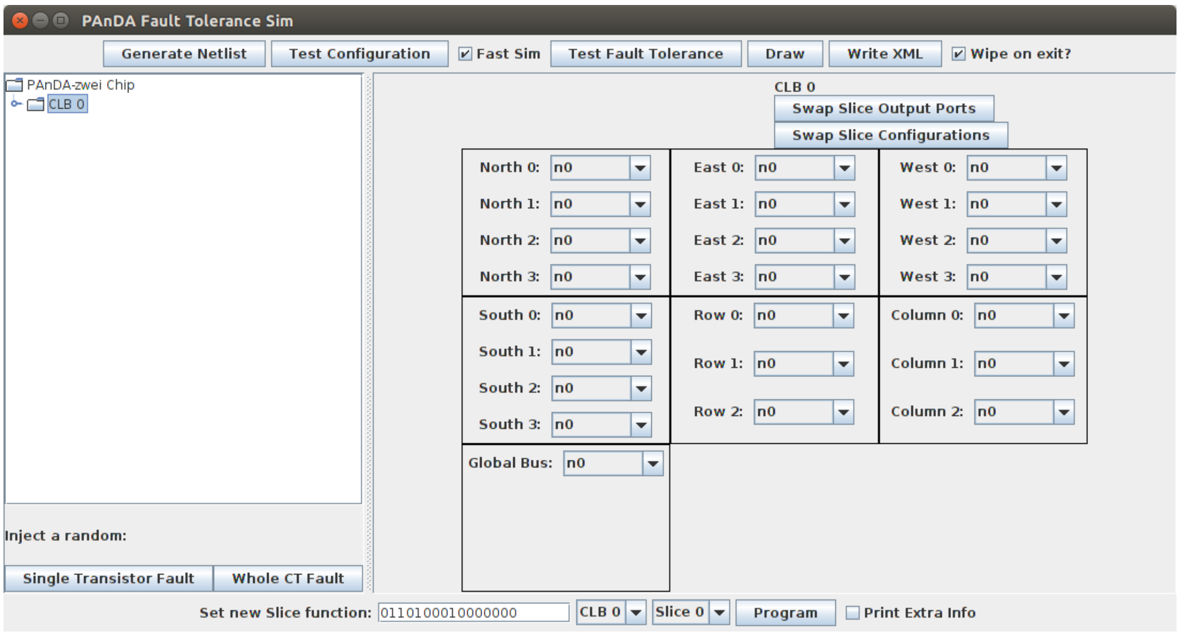
Set new Slice function: 0110100010000000

CLB 0

Slice 0

☐ Print Extra Info

Figure A.1: The Chip level interface which allows the user to configure some global options such as the default width for CTs when a function is automatically configured and the transistor model to use when generating a netlist.



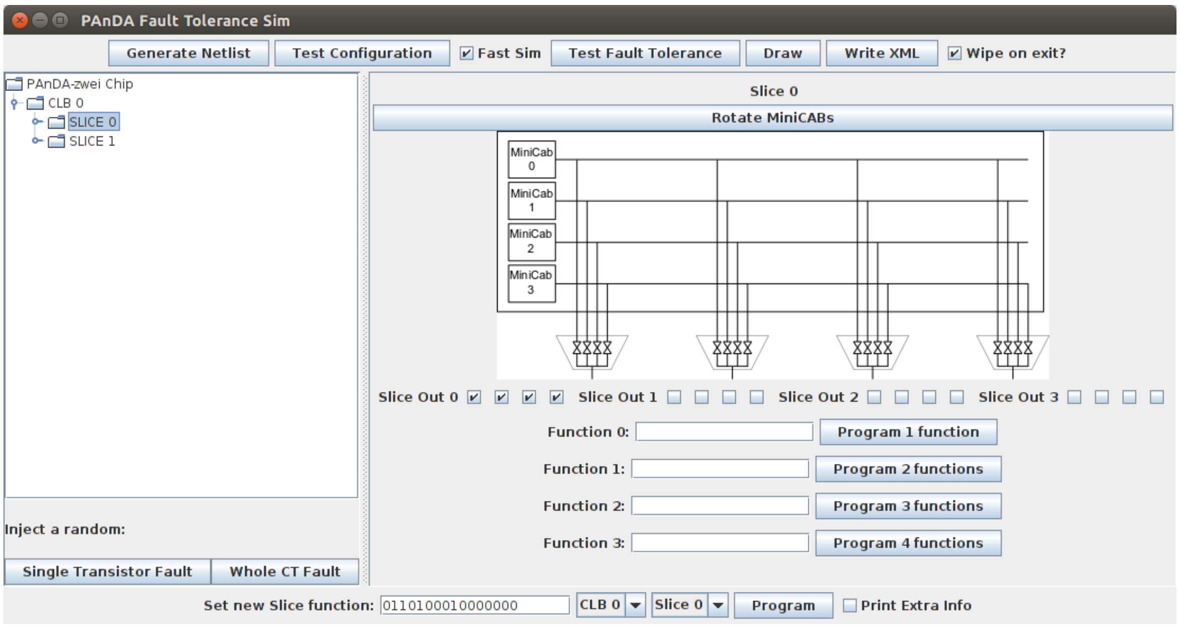
☑ Fast Sim ☑ Wipe on exit?

CLB 0

North 0: n0	East 0: n0	West 0: n0
North 1: n0	East 1: n0	West 1: n0
North 2: n0	East 2: n0	West 2: n0
North 3: n0	East 3: n0	West 3: n0
South 0: n0	Row 0: n0	Column 0: n0
South 1: n0	Row 1: n0	Column 1: n0
South 2: n0	Row 2: n0	Column 2: n0
South 3: n0		
Global Bus: n0		

Set new Slice function: 0110100010000000 CLB 0 Slice 0 ☐ Print Extra Info

Figure A.2: The CLB level interface with some example options for how the routing block might be configured, although this functionality was not implemented.



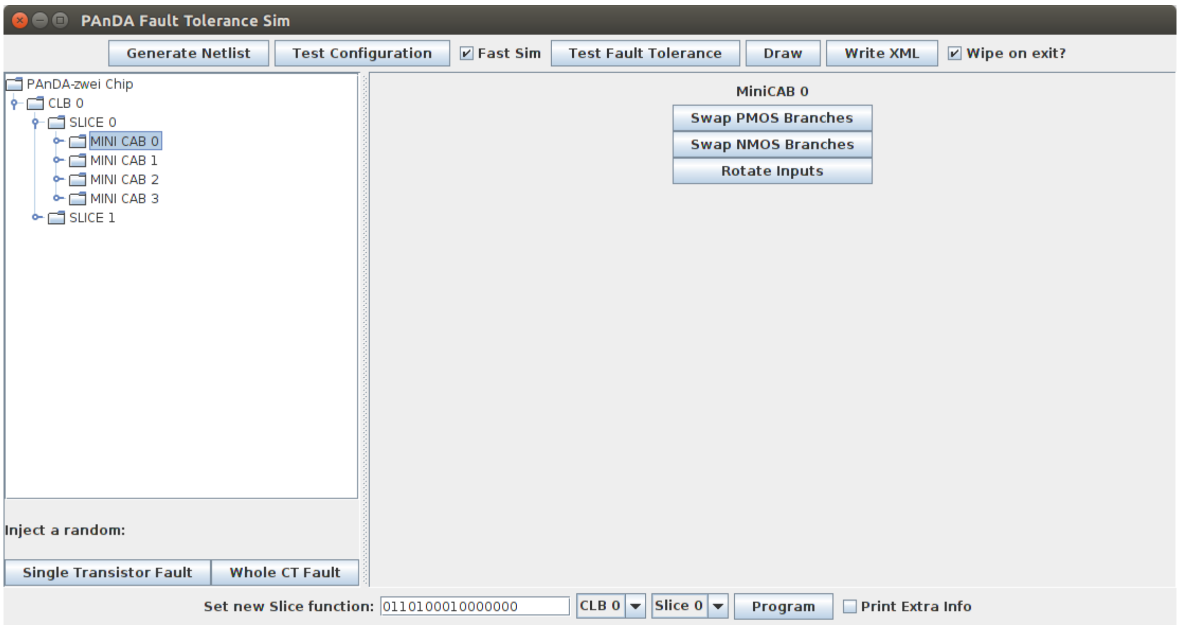
☑ Fast Sim ☑ Wipe on exit?

Slice Out 0 ☑ ☑ ☑ ☑ Slice Out 1 ☐ ☐ ☐ ☐ Slice Out 2 ☐ ☐ ☐ ☐ Slice Out 3 ☐ ☐ ☐ ☐

Function 0:
Function 1:
Function 2:
Function 3:

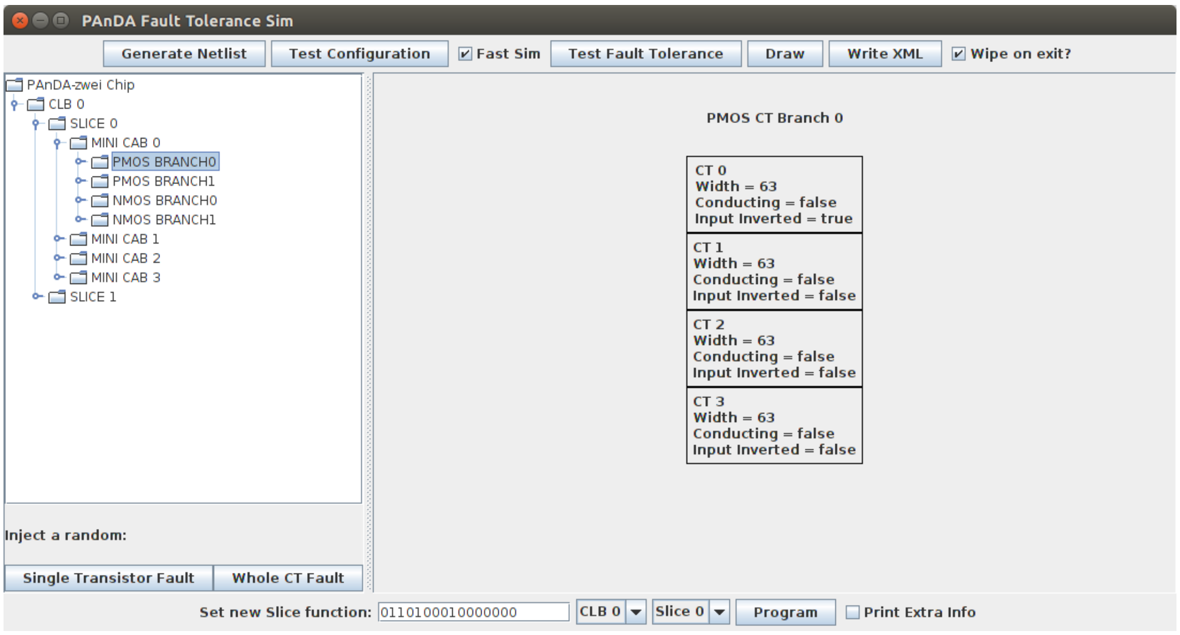
Set new Slice function: 0110100010000000 CLB 0 Slice 0 ☐ Print Extra Info

Figure A.3: The Slice level interface which allows for the configuration of which MiniCAB outputs are connected to which Slice outputs. Also allows multiple functions to be configured within one Slice if they will fit.



[✓] Fast Sim
[✓] Wipe on exit?
Set new Slice function: 0110100010000000
CLB 0
Slice 0
[ ] Print Extra Info

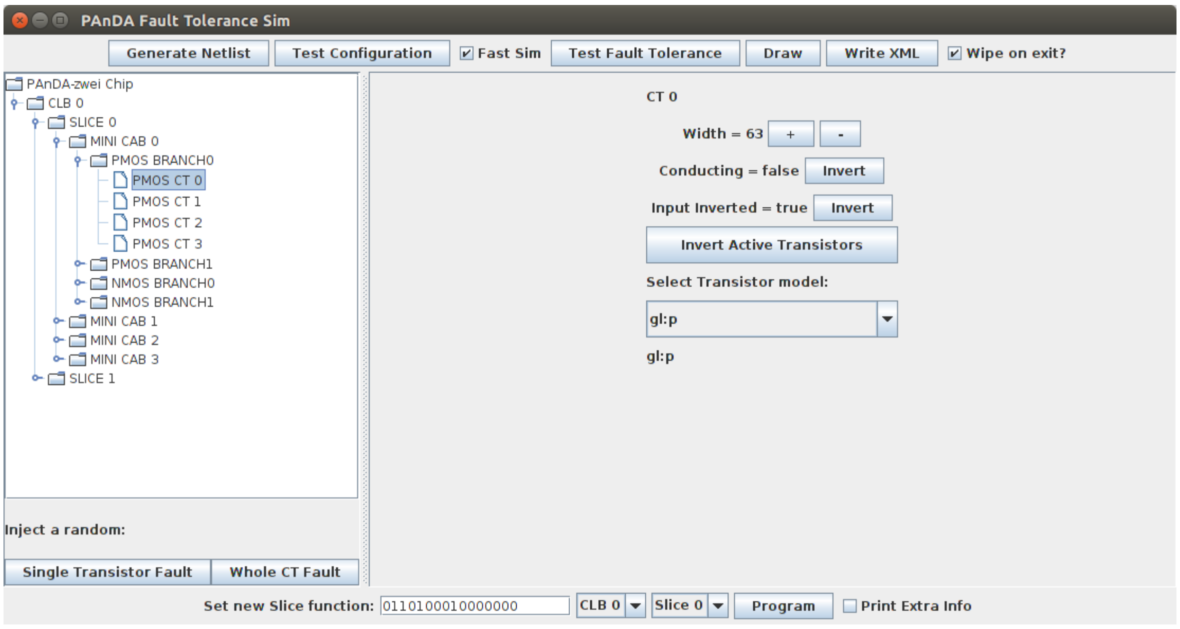
Figure A.4: The MiniCAB level interface which displays some options for switching around the configuration without affecting the functionality.



```
CT 0
Width = 63
Conducting = false
Input Inverted = true

CT 1
Width = 63
Conducting = false
Input Inverted = false

CT 2
Width = 63
Conducting = false
Input Inverted = false

CT 3
Width = 63
Conducting = false
Input Inverted = false
```

[✓] Fast Sim
[✓] Wipe on exit?
Set new Slice function: 0110100010000000
CLB 0
Slice 0
[ ] Print Extra Info

Figure A.5: The CT Branch level interface, summarising the configuration of the CTs that it contains.



CT 0

Width = 63

Conducting = false

Input Inverted = true

Select Transistor model: gl:p

gl:p

Set new Slice function: 0110100010000000 CLB 0 Slice 0

Figure A.6: The interface for configuring the properties of a CT.

Appendix B

Stochastic Swapping Results

Contents

This appendix present the full set of graphs from the experiments in Chapter 8. All graphs display similar features:

- Low "Circuits Fixed" and "Steps Taken" for bias of 0.

- Reasonably constant "Circuits Fixed" for biases >0.

- Steps taken results exhibiting a logarithmic decay as bias increases.

B.1 Swapping random Inputs and swapping random Branches

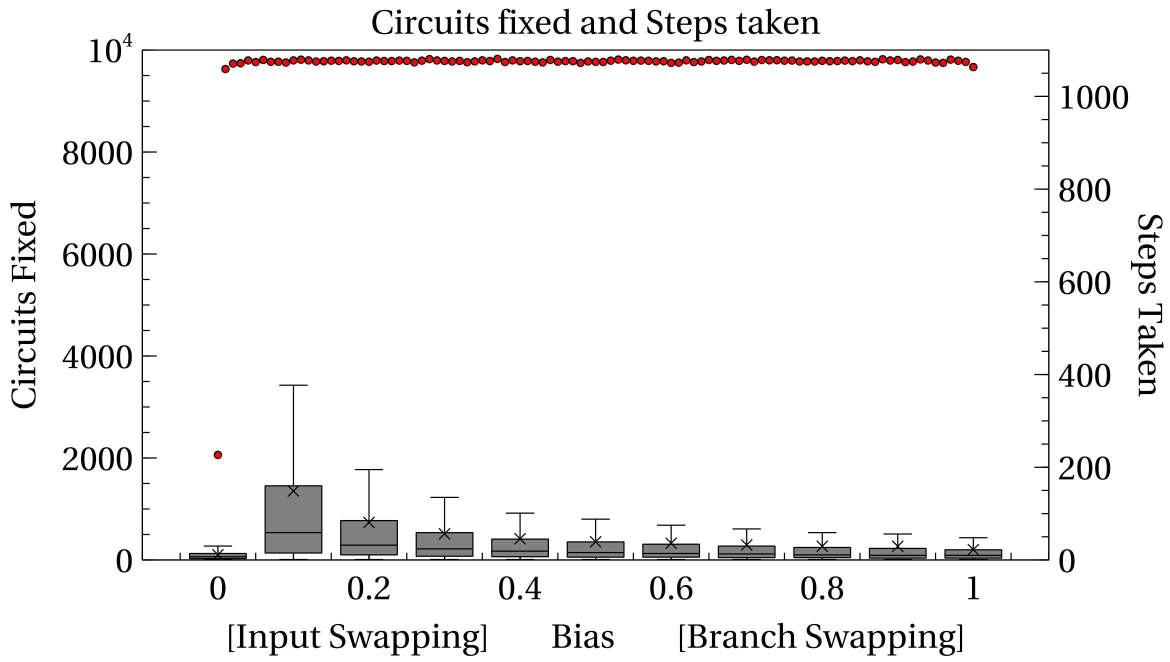


Figure B.1: Results of the stochastic swapping experiment when using the "Swapping Random Inputs" and "Swapping Random Branches" strategies on circuit Z0.

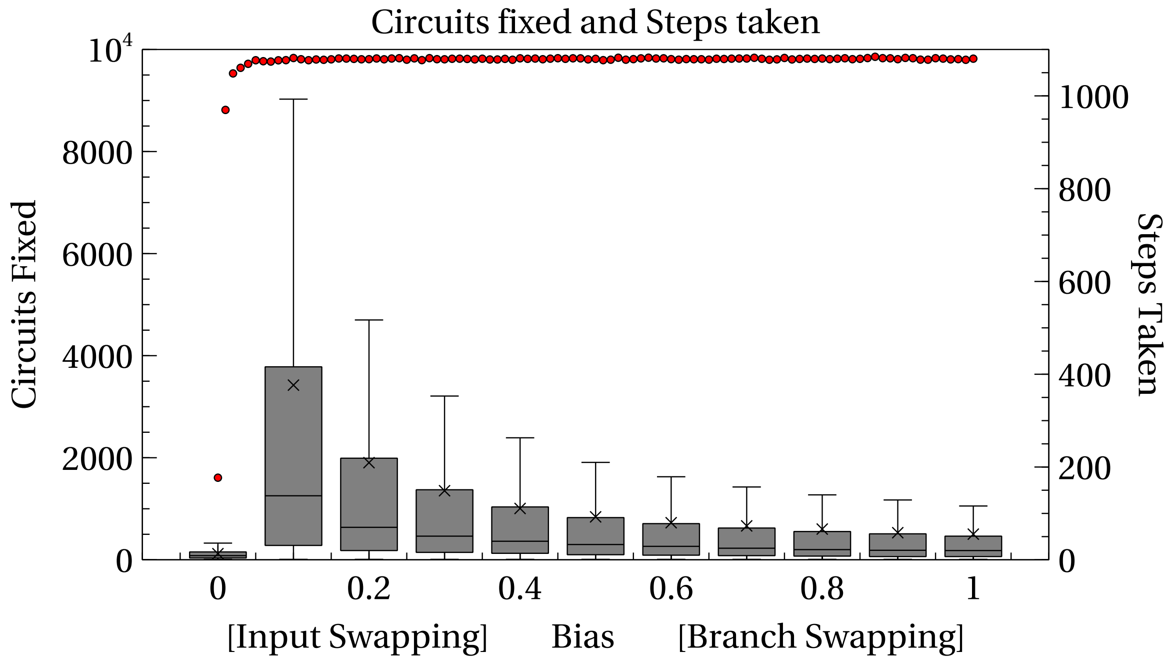


Figure B.2: Results of the stochastic swapping experiment when using the "Swapping Random Inputs" and "Swapping Random Branches" strategies on circuit Z1.

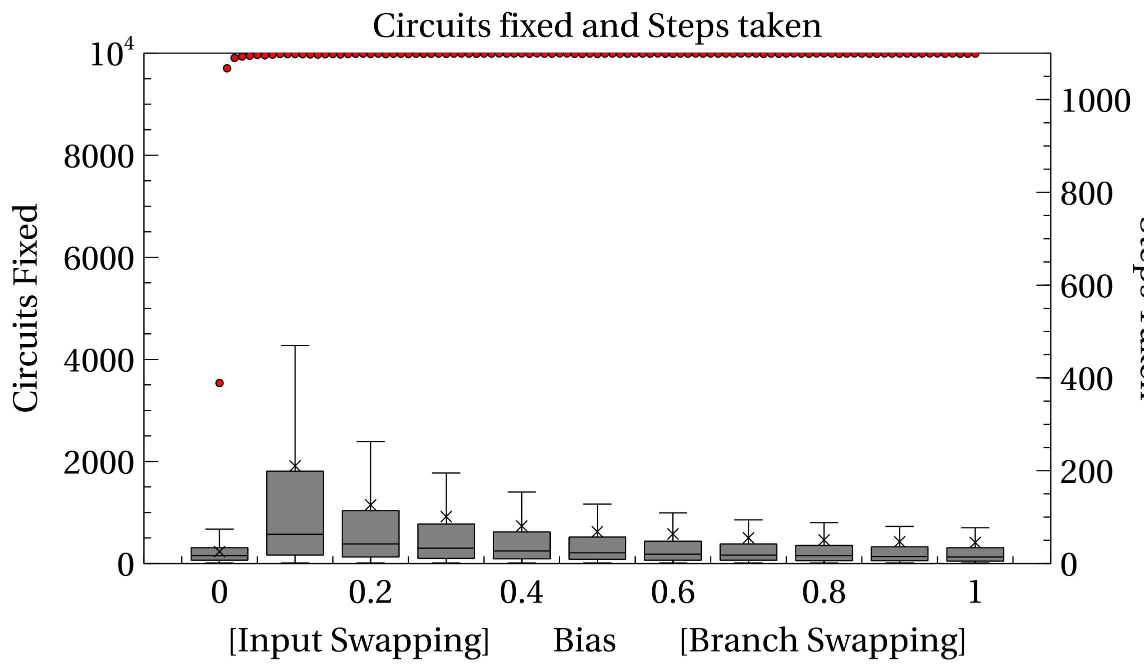


Figure B.3: Results of the stochastic swapping experiment when using the “Swapping Random Inputs” and “Swapping Random Branches” strategies on circuit Z2.

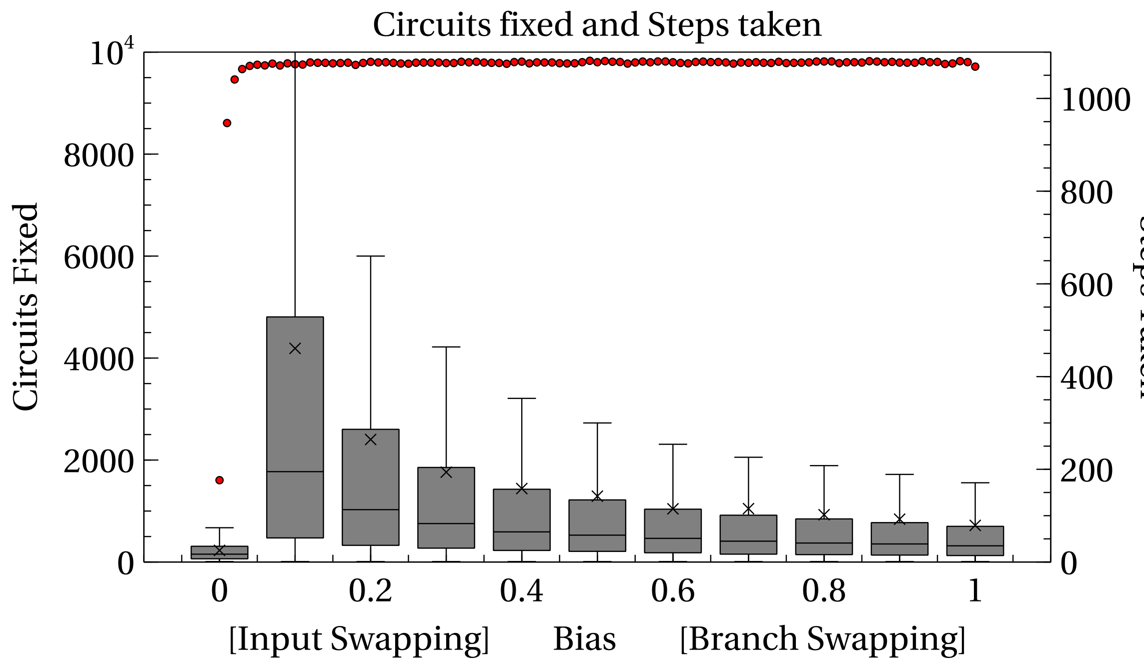


Figure B.4: Results of the stochastic swapping experiment when using the “Swapping Random Inputs” and “Swapping Random Branches” strategies on circuit Z3.

B.2 Shuffling all Inputs and Shuffling all Branches

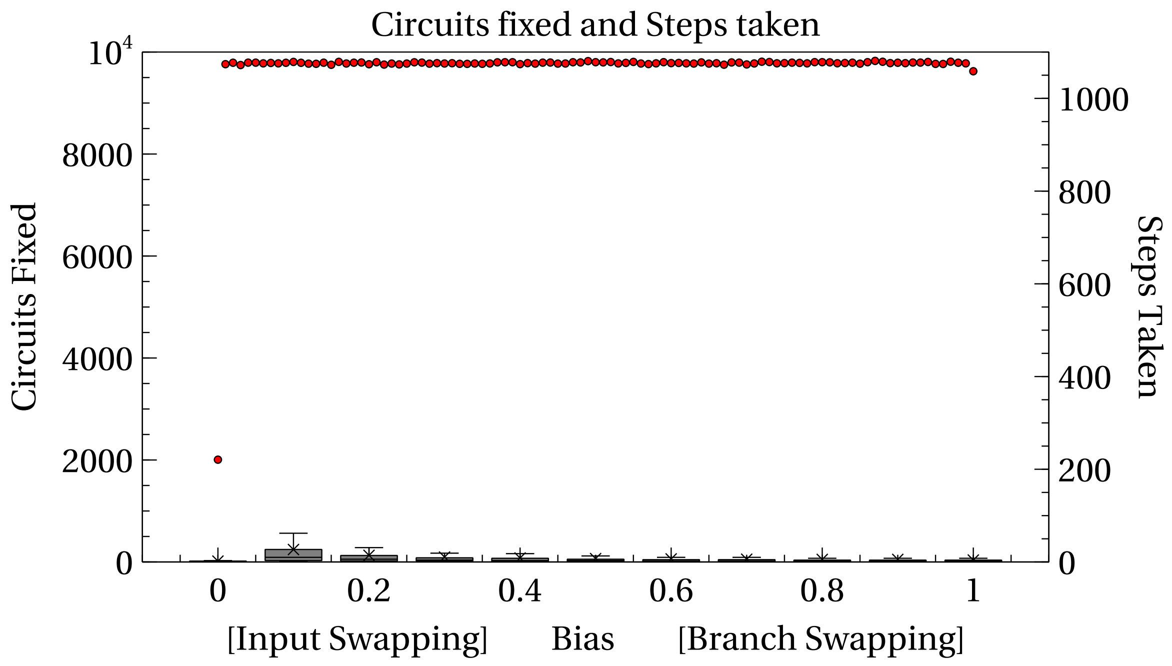


Figure B.5: Results of the stochastic swapping experiment when using the “Shuffling all Inputs” and “Shuffling all Branches” strategies on circuit Z0.

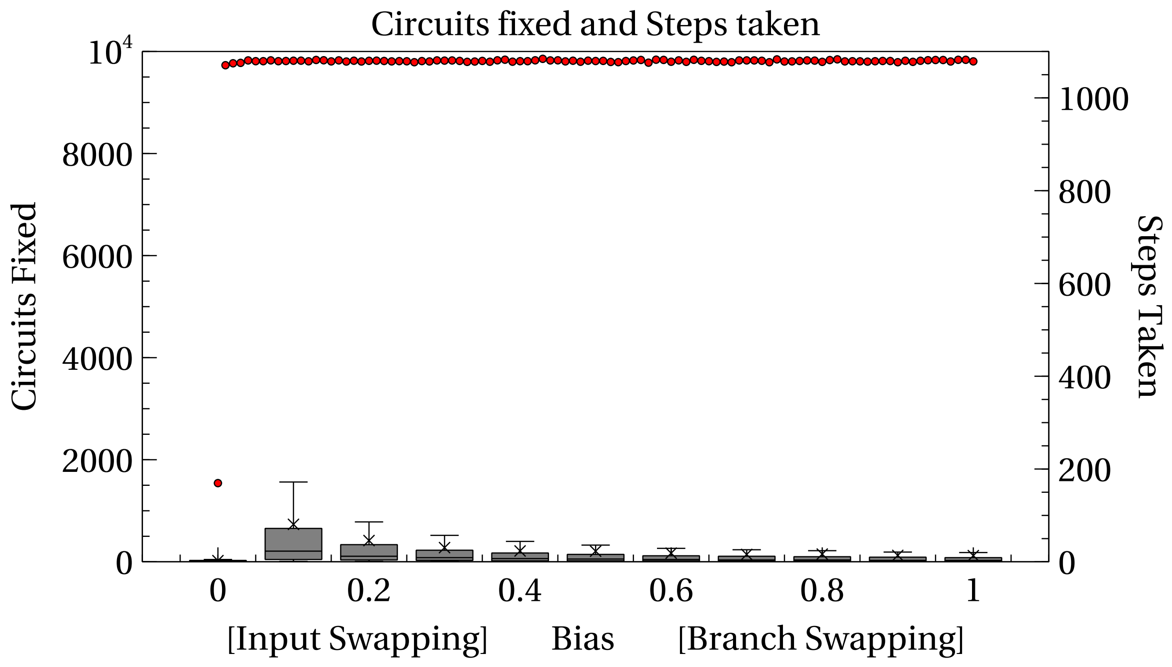


Figure B.6: Results of the stochastic swapping experiment when using the “Shuffling all Inputs” and “Shuffling all Branches” strategies on circuit Z1.

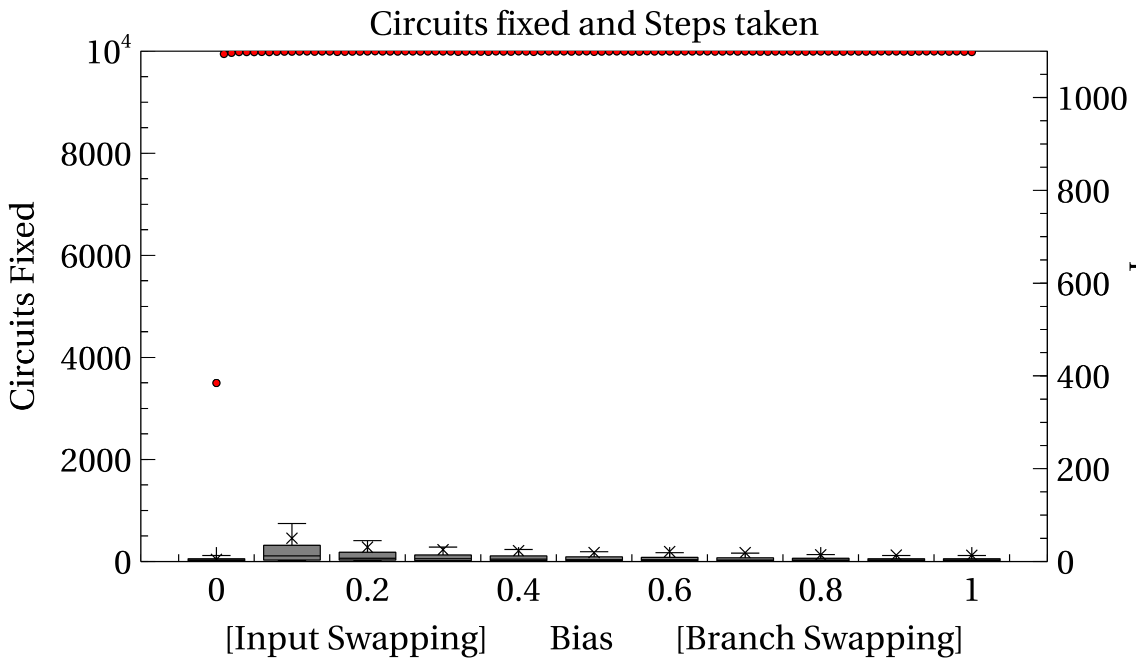


Figure B.7: Results of the stochastic swapping experiment when using the "Shuffling all Inputs" and "Shuffling all Branches" strategies on circuit Z2.

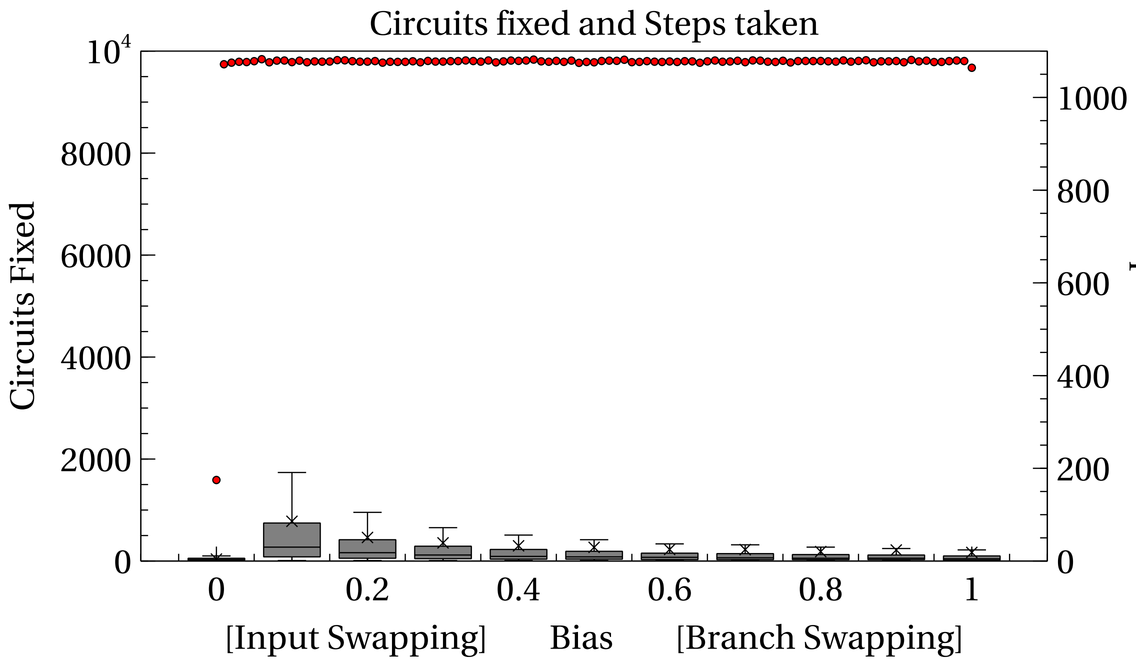


Figure B.8: Results of the stochastic swapping experiment when using the "Shuffling all Inputs" and "Shuffling all Branches" strategies on circuit Z3.

B.3 Shuffling all Inputs and Swapping Random Branches

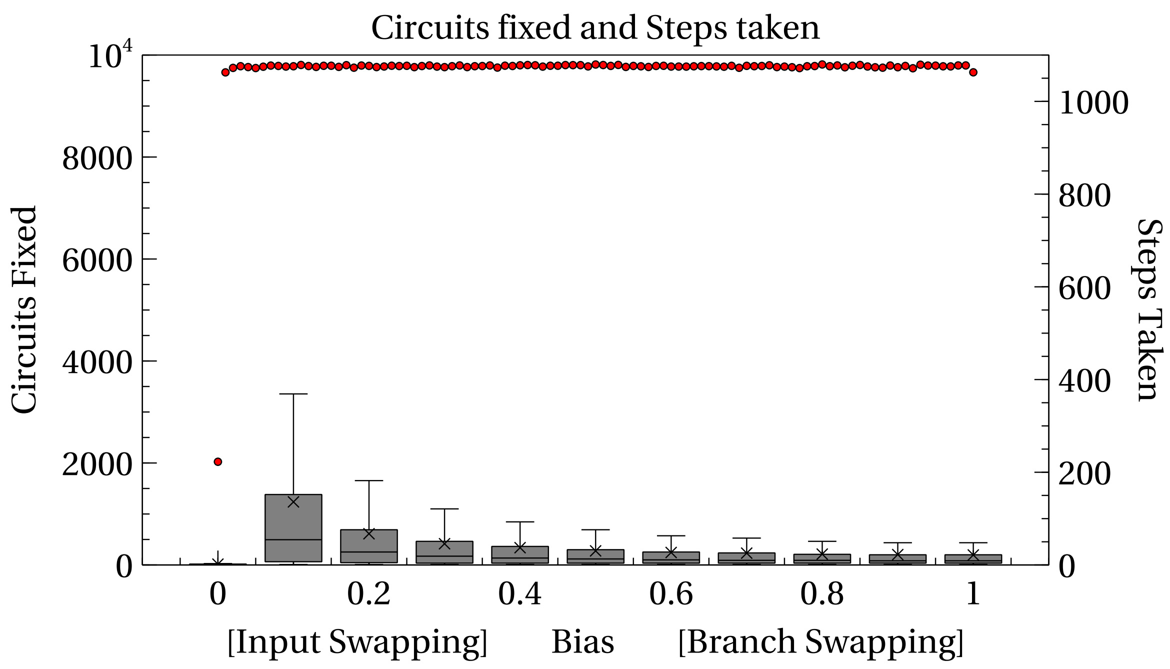


Figure B.9: Results of the stochastic swapping experiment when using the “Shuffling all Inputs” and “Swapping Random Branches” strategies on circuit Z0.

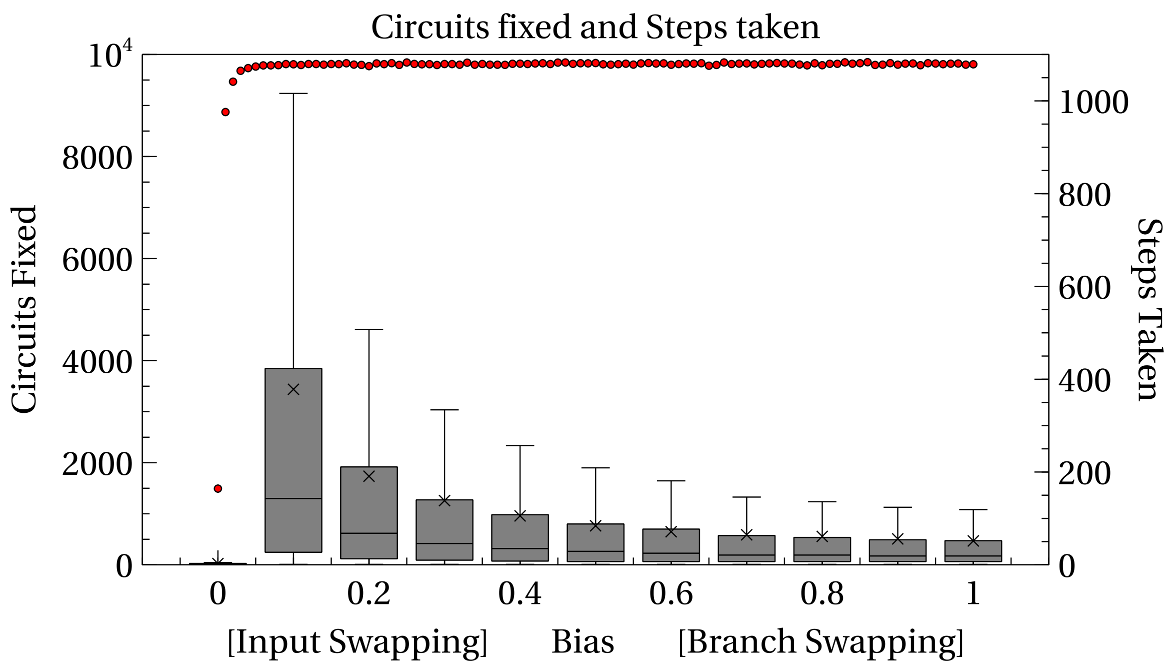


Figure B.10: Results of the stochastic swapping experiment when using the “Shuffling all Inputs” and “Swapping Random Branches” strategies on circuit Z1.

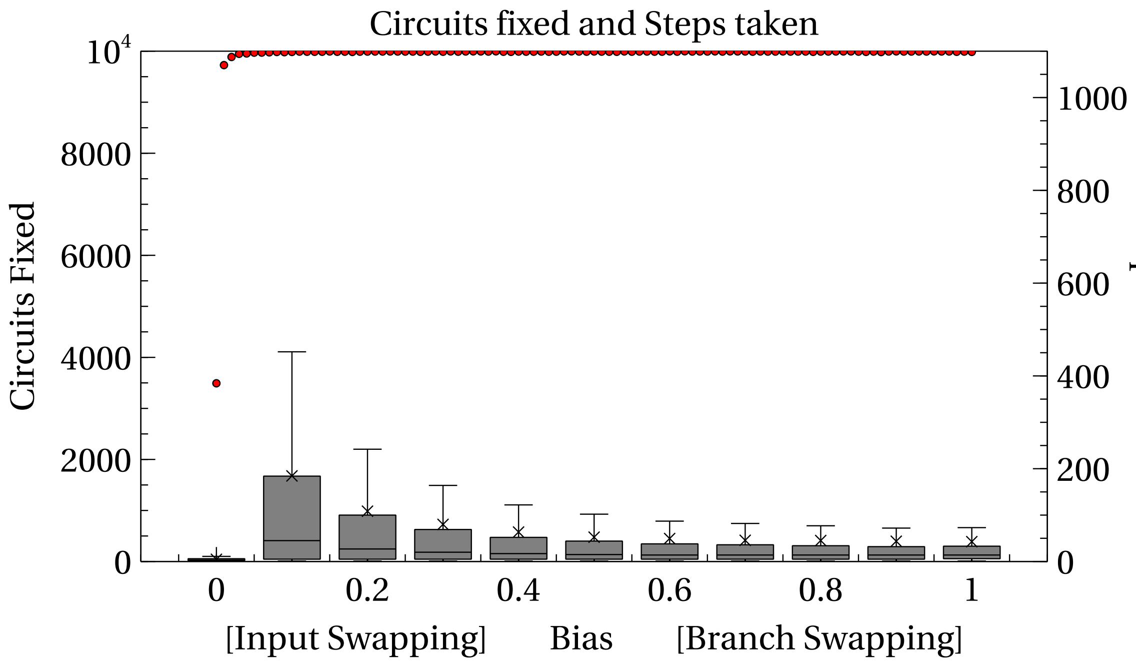


Figure B.11: Results of the stochastic swapping experiment when using the "Shuffling all Inputs" and "Swapping Random Branches" strategies on circuit Z2.

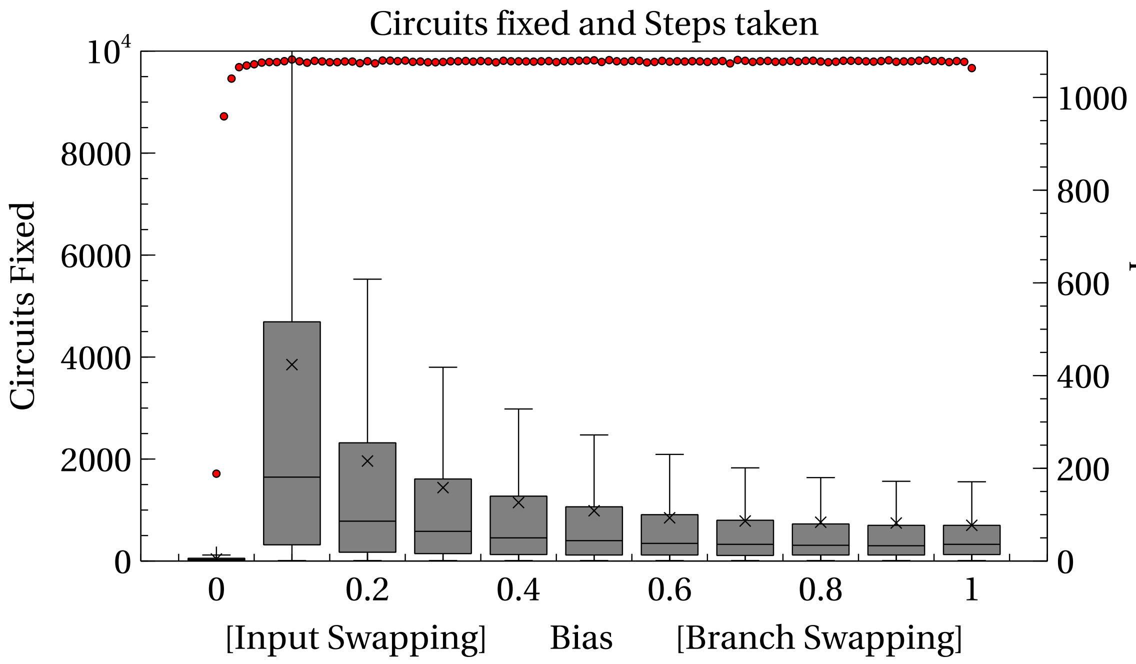


Figure B.12: Results of the stochastic swapping experiment when using the "Shuffling all Inputs" and "Swapping Random Branches" strategies on circuit Z3.

B.4 Swapping Random Inputs and Shuffling all Branches

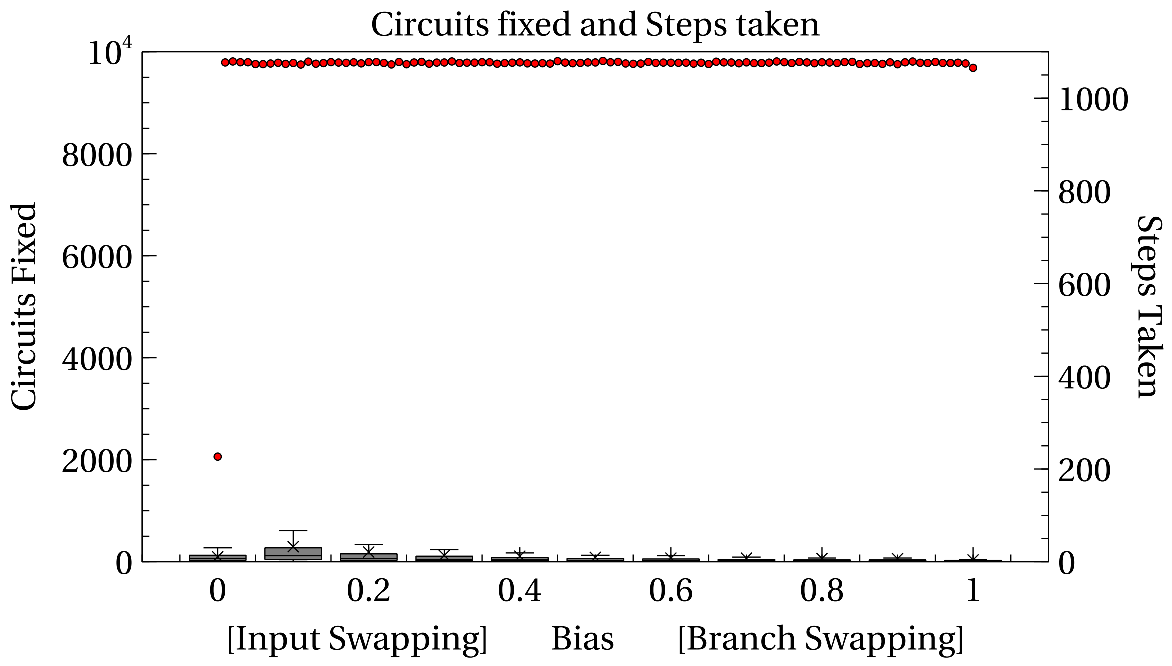


Figure B.13: Results of the stochastic swapping experiment when using the "Swapping Random Inputs" and "Shuffling all Branches" strategies on circuit Z0.

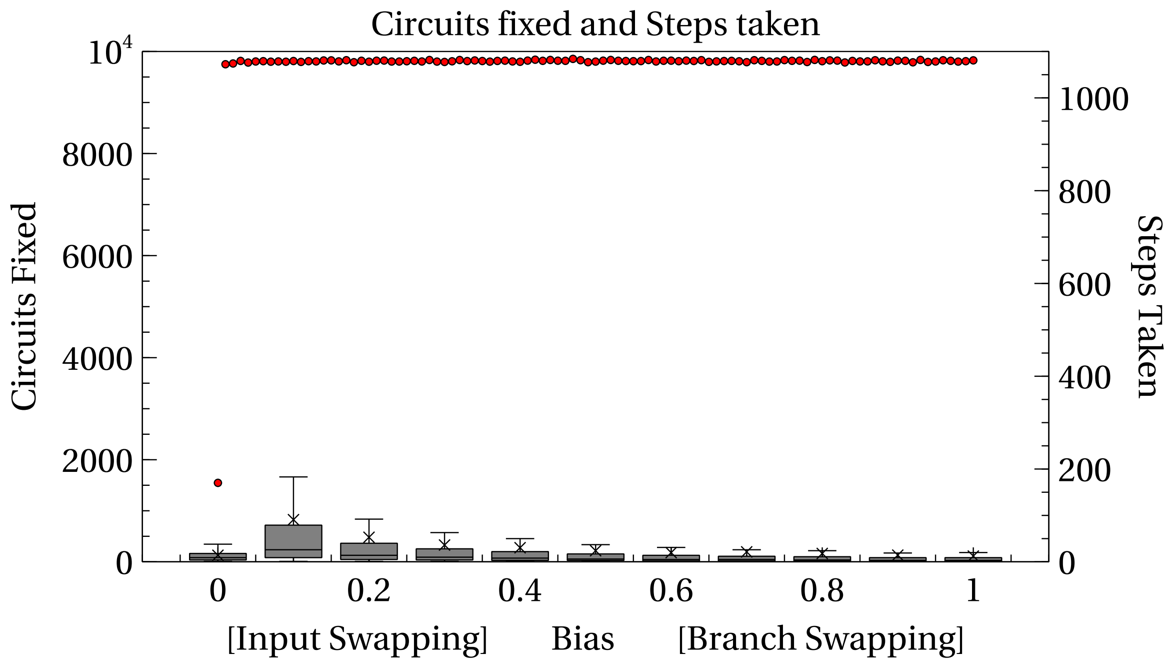


Figure B.14: Results of the stochastic swapping experiment when using the "Swapping Random Inputs" and "Shuffling all Branches" strategies on circuit Z1.

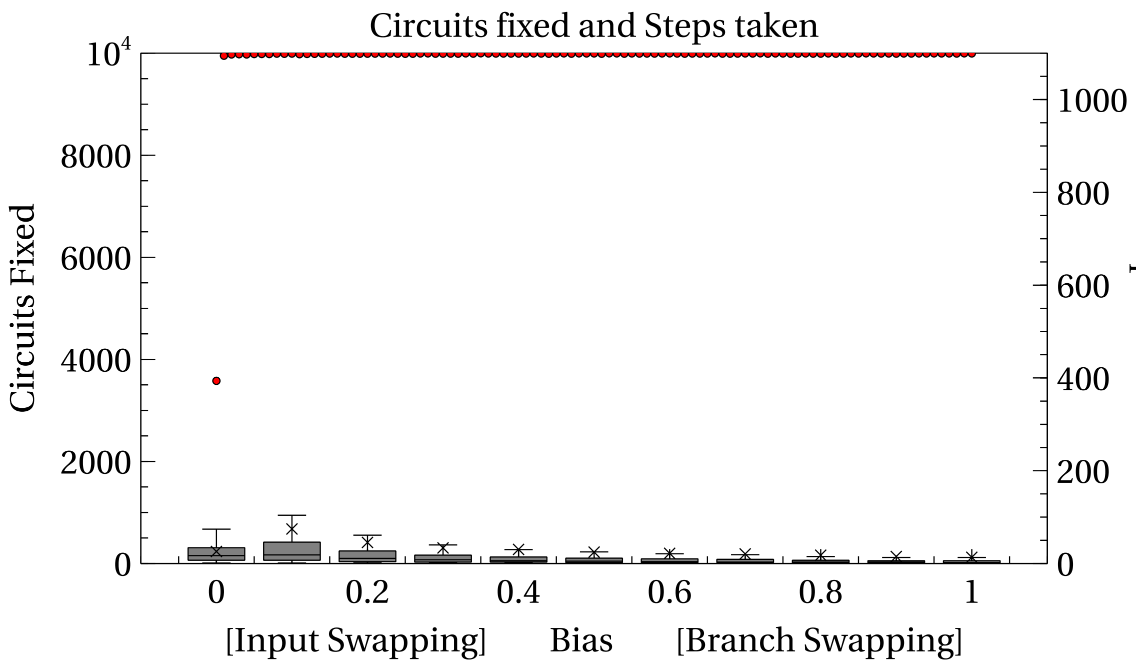


Figure B.15: Results of the stochastic swapping experiment when using the "Swapping Random Inputs" and "Shuffling all Branches" strategies on circuit Z2.

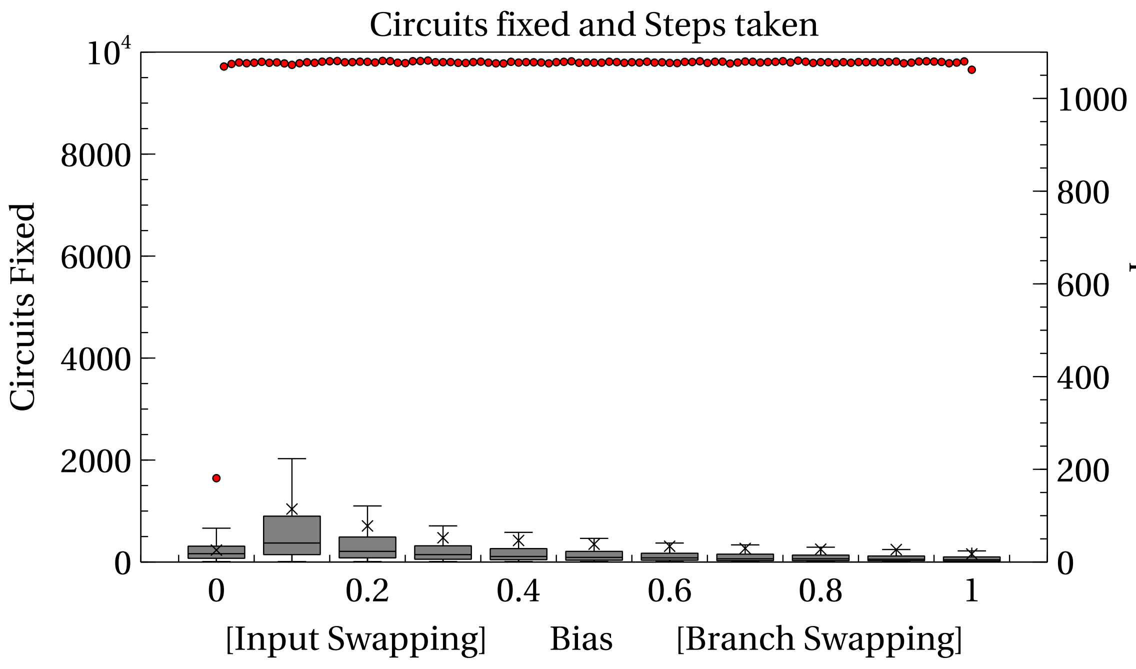


Figure B.16: Results of the stochastic swapping experiment when using the "Swapping Random Inputs" and "Shuffling all Branches" strategies on circuit Z3.

Appendix C

Deterministic Strategies

This appendix details the mapping between strategy numbers and their associated circuit transformation. When referring to MiniCABs by number, these are counted from 0-3 from the left hand side of the Slice diagrams. When referring to Branchs by numbers, these are counted from the left of the Slice diagrams and ignore the MiniCABs such that PMOS Branch 3 is the leftmost PMOS Branch in MiniCAB 1.

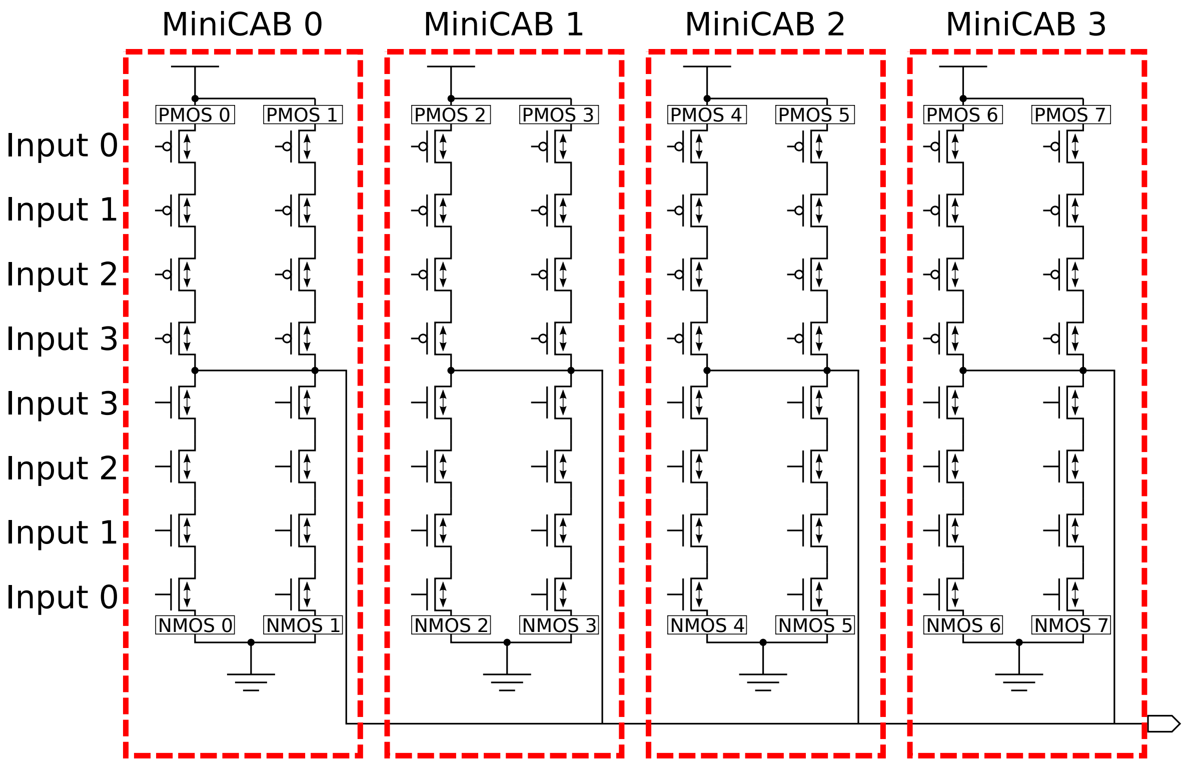


Figure C.1: The numbering schemes used when referring to MiniCABs, CT Branches and Inputs for the work in Chapter 9.

Table C.1: The enumerated Input Swapping strategies. For each strategy, the specified input A is swapped with the specified input B on the MiniCAB indicated. The MiniCAB numbering is shown in Figure C.1.

Input Swapping Strategies			
Strategy Number	MiniCAB	Input A	Input B
0	0	0	1
1	0	0	2
2	0	0	3
3	0	1	2
4	0	1	3
5	0	2	3
6	1	0	1
7	1	0	2
8	1	0	3
9	1	1	2
10	1	1	3
11	1	2	3
12	2	0	1
13	2	0	2
14	2	0	3
15	2	1	2
16	2	1	3
17	2	2	3
18	3	0	1
19	3	0	2
20	3	0	3
21	3	1	2
22	3	1	3
23	3	2	3

Table C.2: The 56 enumerated Branch Swapping strategies. Branch A and Branch B refer to the Branches to be swapped, starting from the left-hand side of Figure C.1.

Branch Swapping Strategies		
Strategy Number	Branch A	Branch B
24	PMOS 0	PMOS 1
25	PMOS 0	PMOS 2
26	PMOS 0	PMOS 3
27	PMOS 0	PMOS 4
28	PMOS 0	PMOS 5
29	PMOS 0	PMOS 6
30	PMOS 0	PMOS 7
31	PMOS 1	PMOS 1
32	PMOS 1	PMOS 3
33	PMOS 1	PMOS 4
34	PMOS 1	PMOS 5
35	PMOS 1	PMOS 6
36	PMOS 1	PMOS 7
37	PMOS 1	PMOS 3
38	PMOS 1	PMOS 4
39	PMOS 1	PMOS 5
40	PMOS 1	PMOS 6
41	PMOS 1	PMOS 7
42	PMOS 3	PMOS 4
43	PMOS 3	PMOS 5
44	PMOS 3	PMOS 6
45	PMOS 3	PMOS 7
46	PMOS 4	PMOS 5
47	PMOS 4	PMOS 6
48	PMOS 4	PMOS 7
49	PMOS 5	PMOS 6
50	PMOS 5	PMOS 7
51	PMOS 6	PMOS 7
52	NMOS 0	NMOS 1
53	NMOS 0	NMOS 2
54	NMOS 0	NMOS 3
55	NMOS 0	NMOS 4
56	NMOS 0	NMOS 5
57	NMOS 0	NMOS 6
58	NMOS 0	NMOS 7
59	NMOS 1	NMOS 2
60	NMOS 1	NMOS 3
61	NMOS 1	NMOS 4
62	NMOS 1	NMOS 5
63	NMOS 1	NMOS 6
64	NMOS 1	NMOS 7
65	NMOS 1	NMOS 3
66	NMOS 1	NMOS 4
67	NMOS 1	NMOS 5
68	NMOS 1	NMOS 6
69	NMOS 1	NMOS 7
70	NMOS 3	NMOS 4
71	NMOS 3	NMOS 5
72	NMOS 3	NMOS 6
73	NMOS 3	NMOS 7
74	NMOS 4	NMOS 5
75	NMOS 4	NMOS 6
76	NMOS 4	NMOS 7
77	NMOS 5	NMOS 6
78	NMOS 5	NMOS 7
79	NMOS 6	NMOS 7

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan 1998.

[2] A. Asenov, "Random Dopant Induced Threshold Voltage Lowering and Fluctuations in Sub-0.1µm MOSFET's: A 3-D "Atomistic" Simulation Study," *Electron Devices, IEEE Transactions on*, vol. 45, no. 12, pp. 2505–2513, Dec. 1998.

[3] A. Asenov, S. Kaya, and A. Brown, "Intrinsic parameter fluctuations in decananometer MOSFETs introduced by gate line edge roughness," *Electron Devices, IEEE Transactions on*, vol. 50, no. 5, pp. 1254–1260, May 2003.

[4] A. Asenov, "Random dopant induced threshold voltage lowering and fluctuations in sub 50 nm MOSFETs: a statistical 3D 'atomistic' simulation study," *Nanotechnology*, vol. 10, no. 2, pp. 153–158, Jun. 1999.

[5] M. White, B. Huang, J. Qin, Z. Gur, M. Talmor, Y. Chen, J. Heidecker, D. Nguyen, and J. Bernstein, "Impact of device scaling on deep sub-micron transistor reliability - a study of reliability trends using sram," in *2005 IEEE International Integrated Reliability Workshop*, Oct 2005, pp. 4 pp.–.

[6] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, Apr. 1962. [Online]. Available: http://dx.doi.org/10.1147/rd.62.0200

[7] J. M. Johnson and M. J. Wirthlin, "Voter insertion algorithms for fpga designs using triple modular redundancy," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 249–258. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723154

[8] R. O. Canham and A. M. Tyrrell, “Evolved fault tolerance in evolvable hardware,” in *Evolutionary Computation, 2002. CEC ’02. Proceedings of the 2002 Congress on*, vol. 2, 2002, pp. 1267–1271.

[9] J. Emmert, C. Stroud, and M. Abramovici, “Online Fault Tolerance for FPGA Logic Blocks,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 15, no. 2, pp. 216–226, Feb. 2007.

[10] M. Garvie and A. Thompson, “Scrubbing away transients and jiggling around the permanent: long survival of FPGA systems through evolutionary self-repair,” in *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International*, Jul. 2004, pp. 155–160.

[11] W.-J. Huang and E. J. McCluskey, “Column-based precompiled configuration techniques for FPGA,” in *Field-Programmable Custom Computing Machines, 2001. FCCM ’01. The 9th Annual IEEE Symposium on*, Mar. 2001, pp. 137–146.

[12] J. Lohn, G. Larchev, and R. Demara, “Evolutionary Fault Recovery in a Virtex FPGA using a Representation that Incorporates Routing,” in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, Apr. 2003, pp. 8 pp.–.

[13] D. Montminy, R. Baldwin, P. Williams, and B. Mullins, “Using relocatable bitstreams for fault tolerance,” in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, Aug. 2007, pp. 701–708.

[14] G. H. Asadi and M. B. Tahoori, “Soft error mitigation for sram-based fpgas,” in *23rd IEEE VLSI Test Symposium (VTS’05)*, May 2005, pp. 207–212.

[15] S. Sarkar, A. Adak, V. Singh, K. Saluja, and M. Fujita, “Seu tolerant sram for fpga applications,” in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 491–494.

[16] J. Y. Lee, C. R. Chang, N. Jing, J. Su, S. Wen, R. Wong, and L. He, “Heterogeneous configuration memory scrubbing for soft error mitigation in fpgas,” in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec 2012, pp. 23–28.

[17] M. Abramovici and C. E. Stroud, “Bist-based test and diagnosis of fpga logic blocks,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, pp. 159–172, Feb 2001.

[18] S. Jamuna. and V. K. Agrawal, “Implementation of bistcontroller for fault detection in clb of fpga,” in *Devices, Circuits and Systems (ICDCS), 2012 International Conference on*, March 2012, pp. 99–104.

[19] Y. Nishitani, K. Inoue, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, “Evaluation of fault tolerant technique based on homogeneous fpga architecture,” in *VLSI and System-on-Chip, 2012 (VLSI-SoC), IEEE/IFIP 20th International Conference on*, Oct 2012, pp. 225–230.

[20] S. U. Rehman, M. Benabdenbi, and L. Anghel, “Test and diagnosis of fpga cluster using partial reconfiguration,” in *Ph.D. Research in Microelectronics and Electronics (PRIME), 2014 10th Conference on*, June 2014, pp. 1–4.

[21] G. E. Moore, “Progress in digital integrated electronics,” in *Electron Devices Meeting, 1975 International*, vol. 21. IEEE, 1975, pp. 11–13. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1478174

[22] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, “Cpu db: Recording microprocessor history,” *Queue*, vol. 10, no. 4, pp. 10:10–10:27, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2181796.2181798

[23] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc, “Design of ion-implanted MOSFET’s with very small physical dimensions,” *Proceedings of the IEEE*, vol. 87, no. 4, pp. 668–678, Apr. 1999.

[24] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” *IEEE Micro*, vol. 32, no. 3, pp. 122–134, May 2012.

[25] “40 years of microprocessor trend data,” http://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/, accessed: 2016-10-3.

[26] J. A. Walker, J. A. Hilder, and A. M. Tyrrell, “Measuring the performance and intrinsic variability of evolved circuits,” in *Evolvable Systems: From Biology to Hardware: 9th International Conference, ICES 2010, York, UK, September 6-8, 2010. Proceedings*, G. Tempesti, A. M. Tyrrell, and J. F. Miller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–12.

[27] “Gold standard simulations ltd. :: Sources of statistical variability.” [Online]. Available: http://www.goldstandardsimulations.com/services/service-simulations/statistical-variability/variability-sources/

[28] P. B. Campos, “Variability-aware circuit performance optimisation through digital reconfiguration,” Ph.D. dissertation, University of York, 2015.

[29] J. A. Hilder, J. A. Walker, and A. M. Tyrrell, “Designing variability tolerant logic using evolutionary algorithms,” *Ph. D. Research in Microelectronics and Electronics*, pp. 184–187, Jul. 2009.

[30] A. M. Tyrrell, *Fault Tolerant Applications.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 191–207. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44616-4_7

[31] L. R. Harriott, “Limits of lithography,” *Proceedings of the IEEE*, vol. 89, no. 3, pp. 366–374, Mar 2001.

[32] A. K. Wong, “Microlithography: trends, challenges, solutions, and their impact on design,” *IEEE Micro*, vol. 23, no. 2, pp. 12–21, March 2003.

[33] R. F. Pease and S. Y. Chou, “Lithography and other patterning techniques for future electronics,” *Proceedings of the IEEE*, vol. 96, no. 2, pp. 248–270, 2008.

[34] K. Lucas, C. Cork, A. Miloslavsky, G. Luk-Pat, L. Barnes, J. Hapli, J. Lewellen, G. Rollins, V. Wiaux, and S. Verhaegen, “Double-patterning interactions with wafer processing, optical proximity correction, and physical design flows,” *Journal of Micro/Nanolithography, MEMS, and MOEMS*, vol. 8, no. 3, pp. 033 002–033 002–10, 2009. [Online]. Available: http://dx.doi.org/10.1117/1.3158061

[35] E. Maricau and G. Gielen, *Analog IC Reliability in Nanometer CMOS.* Springer Science + Business Media, 2013.

[36] J. Keane and C. H. Kim, “An odometer for CPUs,” *IEEE Spectrum*, vol. 48, no. 5, pp. 26–31, May 2011.

[37] V. Huard, M. Denais, and C. Parthasarathy, “{NBTI} degradation: From physical mechanisms to modelling,” *Microelectronics Reliability*, vol. 46, no. 1, pp. 1 – 23, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026271405000351

[38] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, “Razor: a low-power pipeline based on circuit-level timing speculation,” in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, Dec. 2003, pp. 7–18.

[39] N. Naber, T. Getz, Y. Kim, and J. Petrosky, "Real-time fault detection and diagnostics using FPGA-based architectures," in *2010 International Conference on Field Programmable Logic and Applications*, Aug. 2010, pp. 346–351.

[40] K. Inoue, Y. Nishitani, M. Amagasaki, M. Iida, M. Kuga, and T. Sueyoshi, "Fault detection and avoidance of FPGA in various granularities," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2012, pp. 539–542.

[41] S. Mitra, P. P. Shirvani, and E. J. McCluskey, "Fault location in fpga-based reconfigurable systems," in *IEEE Intl. High Level Design Validation and Test Workshop*, 1998.

[42] G. Tempesti, D. Mange, A. Stauffer, and C. Teuscher, "The biowall: an electronic tissue for prototyping bio-inspired systems," in *Evolvable Hardware, 2002. Proceedings. NASA/DoD Conference on*, 2002, pp. 221–230.

[43] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, "Toward robust integrated circuits: The embryonics approach," *Proceedings of the IEEE*, vol. 88, no. 4, pp. 516–543, Apr. 2000.

[44] M. Boesen and J. Madsen, "edna: A bio-inspired reconfigurable hardware cell architecture supporting self-organisation and self-healing," in *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, Jul. 2009, pp. 147–154.

[45] P. Bremner, Y. Liu, M. Samie, G. Dragffy, A. G. Pipe, G. Tempesti, J. Timmis, and A. M. Tyrrell, "Sabre: a bio-inspired fault-tolerant electronic architecture," *Bioinspiration & Biomimetics*, vol. 8, no. 1, p. 016003, 2013. [Online]. Available: http://stacks.iop.org/1748-3190/8/i=1/a=016003

[46] M. Samie, G. Dragffy, and T. Pipe, "Unitronics: A novel bio-inspired fault tolerant cellular system," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, Jun. 2011, pp. 58–65.

[47] A. DeHon and N. Mehta, "Exploiting partially defective LUTs: Why you don't need perfect fabrication," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec. 2013, pp. 12–19.

[48] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic fault tolerance in fpgas via partial reconfiguration," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, 2000, pp. 165–174.

[49] J. Langeheine, S. Fölling, K. Meier, and J. Schemmel, "Towards a silicon primordial soup: A fast approach to hardware evolution with a VLSI transistor array," in *Proc. 3rd Int. Conf. on Evolvable Systems From Biology to Hardware (ICES2000)*, J. Miller, A. Thompson, P. Thomson, and T. C. Fogarty, Eds. Edinburgh, Scotland, UK: Springer Verlag, Apr. 2001, pp. 123–132.

[50] D. Goldberg, *Genetic algorithms in search, optimization, and machine learning.* Reading, Mass: Addison-Wesley Publishing Company, 1989.

[51] M. Mitchell, *An introduction to genetic algorithms.* Cambridge, Mass: MIT Press, 1998.

[52] A. Eiben, *Introduction to evolutionary computing.* New York: Springer, 2003.

[53] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA: MIT Press, 1992.

[54] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA, USA: MIT Press, 1992.

[55] L. Fogel, A. Owens, and M. Walsh, *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons, 1966. [Online]. Available: https://books.google.co.uk/books?id=QMLaAAAAMAAJ

[56] A. Colorni, M. Dorigo, V. Maniezzo *et al.*, "Distributed optimization by ant colonies," in *Proceedings of the first European conference on artificial life*, vol. 142. Paris, France, 1991, pp. 134–142.

[57] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, Nov 1995, pp. 1942–1948 vol.4.

[58] K. Miettinen, *Nonlinear Multiobjective Optimization*, ser. International Series in Operations Research & Management Science. Springer US, 1999. [Online]. Available: https://books.google.co.uk/books?id=ha_zLdNtXSMC

[59] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[60] S. Thompson, A. Mycroft, G. Brat, A. Venet, N. AERONAUTICS, and S. A. M. F. C. A. R. CENTER., *Automatic In-Flight Repair of FPGA Cosmic*

Ray Damage. Defense Technical Information Center, 2005. [Online]. Available: https://books.google.co.uk/books?id=be2JDAEACAAJ

[61] A. Thompson, *An evolved circuit, intrinsic in silicon, entwined with physics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 390–405. [Online]. Available: http://dx.doi.org/10.1007/3-540-63173-9_61

[62] J. Langeheine, J. Becker, S. Fölling, K. Meier, and J. Schemmel, "A CMOS FPTA Chip for Intrinsic Hardware Evolution of Analog Electronic Circuits," in *Proc. of the Third NASA/DOD Workshop on Evolvable Hardware*. Long Beach, CA, USA: IEEE Computer Society Press, Jul. 2001, pp. 172–175.

[63] J. Langeheine, "Intrinsic Hardware Evolution on the Transistor Level," Ph.D. dissertation, University of Heidelberg, 2005.

[64] M. Trefzer, J. Langeheine, K. Meier, and J. Schemmel, *Operational Amplifiers: An Example for Multi-objective Optimization on an Analog Evolvable Hardware Platform*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 86–97. [Online]. Available: http://dx.doi.org/10.1007/11549703_9

[65] J. Hilder, J. Walker, and A. M. Tyrrell, "Optimising variability tolerant standard cell libraries," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 2273–2280.

[66] D. Keymeulen, A. Stoica, R. Zebulum, and V. Duong, "Results on the fitness and population based fault tolerant approaches using a reconfigurable electronic device," in *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, vol. 1, 2000, pp. 537–5441.

[67] M. Garvie, "Reliable electronics through artificial evolution," Ph.D. dissertation, University of Sussex, Brighton, 2005.

[68] M. Hartmann, "Evolution of Fault and Noise Tolerant Digital Circuits," Ph.D. dissertation, Norwegian University of Science and Technology, 2005.

[69] A. Stoica, D. Keymeulen, R. S. Zebulum, A. Thakoor, T. Daud, G. Klimeck, Y. Jin, R. Tawel, and V. Duong, "Evolution of Analog Circuits on Field Programmable Transistor Arrays," in *Proc. of the Second NASA/DOD Workshop on Evolvable Hardware*. Palo Alto, CA, USA: IEEE Computer Society Press, Jul. 2000, pp. 99–108.

[70] D. Lawson, J. Walker, M. Trefzer, S. Bale, and A. M. Tyrrell, "A hierarchical fault tolerant system on the panda device with low disruption," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, Jul. 2014, pp. 69–76.

[71] P. B. Campos, D. M. Lawson, S. J. Bale, J. A. Walker, M. A. Trefzer, and A. M. Tyrrell, "Overcoming Faults using Evolution on the PAnDA Architecture," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*, Jun. 2013, pp. 613–620.

[72] J. Walker, M. Trefzer, S. Bale, and A. M. Tyrrell, "Panda: A reconfigurable architecture that adapts to physical substrate variations," *Computers, IEEE Transactions on*, vol. 62, no. 8, pp. 1584–1596, Aug. 2013.

[73] J. Walker, M. Trefzer, and A. M. Tyrrell, "Designing function configuration decoders for the panda architecture using multi-objective cartesian genetic programming," in *Evolvable Systems (ICES), 2013 IEEE International Conference on*, Apr. 2013, pp. 96–103.

[74] M. Trefzer, A. Tyrell, and J. Walker, "Field-programmable gate array," Oct. 2 2014, uS Patent App. 14/355,859. [Online]. Available: http://www.google.com.ar/patents/US20140292369

[75] Y. Wang, "Circuit clustering for cluster-based fpgas using novel multiobjective genetic algorithms," Ph.D. dissertation, University of York, 2015.

[76] A. S. Fraser, "Monte Carlo analyses of genetic models," *Nature*, vol. 181, no. 4603, pp. 208–209, Jan. 1958.

[77] A. Thompson and C. Rijsbergen, *Hardware Evolution: Automatic design of electronic circuits in reconfigurable hardware by artificial evolution.* Springer-Verlag New York, Inc., 2001, vol. 1, no. 11. [Online]. Available: http://dl.acm.org/citation.cfm?id=558001

[78] "Ngspice circuit simulator," http://ngspice.sourceforge.net/, accessed: 30-05-2016.

[79] "Gold standard simulations ltd." http://www.goldstandardsimulations.com/, accessed: 30-05-2016.

[80] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb 2007.

[81] M. Nicolaidis, *Soft errors in modern electronic systems.* New York: Springer Science+Business Media, LLC, 2011.

[82] C. Beckhoff, D. Koch, and J. Torresen, “Short-circuits on fpgas caused by partial runtime reconfiguration,” in *2010 International Conference on Field Programmable Logic and Applications*, Aug 2010, pp. 596–601.

[83] E. J. McCluskey, “Minimization of boolean functions,” *Bell system technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

[84] S. Trimberger, “Multiple bitstreams enabling the use of partially defective programmable integrated circuits while avoiding localized defects therein,” Oct. 16 2007, uS Patent 7,284,229. [Online]. Available: http://www.google.com/patents/US7284229

[85] A. H. Madian, H. H. Amer, and A. O. Eldesouky, “Catastrophic short and open fault detection in mos current mode circuits: A case study,” in *2010 12th Biennial Baltic Electronics Conference*, Oct 2010, pp. 145–148.

[86] R. A. Fisher, F. Yates *et al.*, “Statistical tables for biological, agricultural and medical research.” *Statistical tables for biological, agricultural and medical research.*, no. 3th rev. ed, 1948.

[87] N. Stander, W. Roux, A. Basudhar, T. Eggleston, T. Goel, and K. Craig, *LS-OPT® User’s Manual: A Design Optimization and Probabilistic Analysis Tool for the Engineering Analyst, Version 5.0*, Livermore Software Technology Corporation, 2013.

[88] A. DeHon, “Balancing interconnect and computation in a reconfigurable computing array (or, why you don’t really want 100% lut utilization),” in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’99. New York, NY, USA: ACM, 1999, pp. 69–78. [Online]. Available: http://doi.acm.org/10.1145/296399.296431

[89] R. Tessier and H. Giza, “Balancing logic utilization and area efficiency in fpgas,” in *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, ser. FPL ’00. London, UK, UK: Springer-Verlag, 2000, pp. 535–544. [Online]. Available: http://dl.acm.org/citation.cfm?id=647927.739548

[90] J. Chen, Y. Lu, I. Comsa, and P. Kuonen, “A scalability hierarchical fault tolerance strategy: Community fault tolerance,” in *Automation and Computing (ICAC), 2014 20th International Conference on*, Sept 2014, pp. 212–217.