LUCAS GAIA DE CASTRO

Defect Tolerant Routing Elements for Non-Volatile Field-Programmable Gate Arrays

São Paulo
2022

LUCAS GAIA DE CASTRO

Defect Tolerant Routing Elements for Non-Volatile Field-Programmable Gate Arrays

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro de Computação.

São Paulo
2022

LUCAS GAIA DE CASTRO

Defect Tolerant Routing Elements for Non-Volatile Field-Programmable Gate Arrays

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro de Computação.

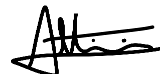
Área de Concentração:

Engenharia de Computação e Sistemas Digitais

Orientador:

Prof. Dr. Bruno de Carvalho Albertini

Versão revisada pelo orientador e de acordo.



São Paulo
2022

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, ______ de ____________________ de __________

Assinatura do autor: ________________________

Assinatura do orientador: ________________________

Catalogação-na-publicação

ACKNOWLEDGMENTS

To my advisors both at TU Darmstadt and USP: M.Sc. Tobias Schladt and Prof. Dr. Bruno de Carvalho Albertini, for insightful comments throughout the work. I asked for a challenging topic and their support made me learn much more I thought I could for the time window.

To my mother Raquel, my sister Daniela and Mavi, all law graduates who spent countless days listening to long monologues about computer architecture and gave their wholehearted belief in my work albeit not understanding most of what was being said.

To Felipe and Guilherme, soon to be engineers who made the above mentioned monologues into amazing discussions. Apart from great invidivuals they also were much needed friends inside USP's halls.

To all the faculty from USP and TU Darmstadt, for the opportunities and resources provided. All of my future success as a computer engineer is mostly thanks to all these people and study halls.

And finally to my father Ricardo, who stopped me from attending law school.

"Do only what only you can do"

-- Edsger W. Dijkstra

RESUMO

Field Programmable Gate Arrays são dispositivos compostos por elementos de lógica, memória e roteamento que permitem a implementação de um circuito arbitrário através de sua programação. Suas aplicações são inúmeras e incluem prototipagem de circuitos integrados para aplicações específicas (*Application Specific Integrated Circuits*) e implementação de aceleradores em hardware. Duas desvantagens de FPGAs comerciais são a área de substrato dedicada ao roteamento de circuitos e o fato da memória de programação ser geralmente volátil, fazendo com que o dispositivo deva ser reprogramado sempre que for reiniciado. Memórias não-voláteis baseadas em memristores possívelmente contornam ambas questões, porém esses elementos são mais suscetíveis a defeitos de fabricação do que transistores.

Esse trabalho propõe uma nova arquitetura de elementos de roteamento baseados em células de memória não-voláteis que conseguem mitigar taxas de defeito moderadas antes de inviabilizar a implementação de um circuito devido à falhas no roteamento. A arquitetura é simulada e comparada no quesito funcional ao estado da arte de memórias não-voláteis baseadas em memristores (células *2-Transistor-2-Memristor*) utilizando *Verilog to Routing* (VTR) e um simulador próprio desenvolvido em Python. Em comparação com memórias do tipo *Static Random Access Memory*, com 6 transistores por célula, a arquitetura proposta possui um ganho de área aproximado de 14,29%, além de fornecer *gate-boosting* às chaves de roteamento, diminuindo o atraso de circuitos implementados. Em comparação com memórias *2-Transistor-2-Memristor*, a arquitetura proposta suporta taxas de defeito até três vezes maior.

Palavras-Chave – FPGA, *Place and Route*, Arquitetura de Roteamento, Memristores, Tolerância a Defeitos, Memória Não-Volátil.

ABSTRACT

Field Programmable Gate Arrays are devices composed of logic, memory and routing elements that allow the implementation of an arbitrary circuit through its programming. Their applications are numerous and include Application Specific Integrated Circuits prototyping and the development of hardware accelerators. Two disadvantages of commercial FPGAs are the substrate area dedicated to the routing architecture and the fact that the program memory is usually volatile, meaning the device must be reprogrammed every time it is power-cycled. Non-volatile memristor-based memories can possibly overcome both issues, but these elements are more susceptible to manufacturing defects than transistors.

This work proposes a new architecture of routing elements based on non-volatile memory cells that can mitigate moderate defect rates before making a circuit implementation infeasible due to routing failures. The architecture is simulated and compared functionally to state-of-the-art non-volatile memristor-based memories (2-Transistor-2-Memristor cells) using Verilog to Routing (VTR) and a proprietary simulator developed in Python. Compared to Static Random Access Memory type memories with 6 transistors per cell, the proposed architecture has an approximate area gain of 14.29%, and also provides gate-boosting to routing switches, decreasing the delay of implemented circuits. Compared to 2-Transistor-2-Memristor memories, the proposed architecture supports up to three times higher defect rates.

Keywords – FPGA, Place and Route, Routing Architecture, Memristors, Defect-Tolerant, Non-Volatile Memory.

LIST OF FIGURES

1 FPGA architecture example. White blocks are I/O, orange blocks are logic blocks, blue blocks are memory blocks and green blocks are multipliers. The black lines are a rough representation of wiring segments and routing elements. Source: author. 15

2 Configurable logic block example. Source: author. 16

3 FPGA routing architecture in detail. A logic block is again represented in orange, while connection blocks are in green and a switchbox is shown in blue. Source: author. 17

4 Routing multiplexer from [1]. From input to output there are always two pass-transistors acting as routing switches controlled by memory cells. At the output there is a buffer for signal restoration. The buffer presence makes the multiplexer unidirectional. Source: [1]. 18

5 Comparison between 6T SRAM cell and 2T2R ReRAM cell. 19

6 Exemplification of TMR design from [2]. 21

7 VTR standard flow. Source: author. 23

8 Routing resource graph representation from [3]. Dotted lines are switchbox connections and dashed lines are connection blocks connections. 25

9 Proto-Voter cell schematic 30

10 Charging profile of a NMOS device used as pass transistor. Source: author. 31

11 Low-level UML class diagram of fault-tolerant-routing-mux module. Source: author. 34

12 High-level UML class diagram of fault-tolerant-routing-mux module. Source: author. 35

13 VTR CAD flow with integrated defect simulation of routing resources . . . 39

14 SA0 cell probabilities obtained for each cell versus mathematical ground truth. Source: author. 46

15 SA1 cell probabilities obtained for each cell versus mathematical ground truth. Source: author. 47

16 UD cell probabilities obtained for each cell versus mathematical ground truth. Source: author. 48

17 Percentage of defect routing edges and memory cells using 2T2R. The amount of defect edges grows quadratically due to the presence of UD errors or multiple SA1 errors in the same multiplexer. Source: author. . . . 49

18 Percentage of defect routing edges and memory cells using proto-voter cell. The amount of defect edges grows linearly due to the mitigation of SA1 and UD error types. Source: author. 49

19 Number of designs routed in a 20x20 grid for each memory cell type with increasing defect probabilities. Source: author. 50

20 Number of designs routed in a 30x30 grid for each memory cell type with increasing defect probabilities. Source: author. 50

21 Critical path delay variation before and after defect simulation. Source: author. 51

LIST OF TABLES

1 2T2R error based on pull-up or pull-down errors. Source: author. 19

2 Effect of memory cell errors on a routing multiplexer 29

3 Programmable switch gate behavior using a proto-voter cell as NV-memory. Source: author. 30

4 Command line options to run defect simulation with VTR. Source: author. 38

5 FPGA utilization for designs evaluated on a 20x20 grid with a channel width of 60. Source: author. 42

6 FPGA utilization for designs evaluated on a 30x30 grid with a channel width of 80. Source: author. 43

ABBREVIATIONS

2T2R 2-Transistor-2-Memristor

ASIC Application Specific Integrated Circuit

BEOL Back End-Of-Line

BLE Basic Logic Element

BLIF Berkley Logic Interchange Format

CAD Computer Aided Design

CLB Configurable Logic Block

DSP Digital Signal Processing

FF Free of Failure

FPGA Field Programmable Gate Array

FTRM Fault-Tolerant Routing Mux

HRS High Resistive State

LRS Low Resistive State

LUT Look-Up Table

NRE Non-Recurring Engineering

QMR Quintuple Modular Redundancy

RR Routing Resources

SA0 Stuck at 0

SA1 Stuck at 1

SHA Secure Hash Algorithm

SEU Single Event Upset

SRAM Static Random Access Memory

TMR Triple Modular Redundancy

UD Undefined

VPR Versatile Place & Route

VTR Verilog to Routing

CONTENTS

1 Introduction **12**

2 Background and Previous Work **14**

2.1 FPGA Architecture 14

2.2 Memristive Systems 16

2.3 Fault-Tolerant FPGA design 19

3 Verilog to Routing (VTR) **22**

3.1 VTR Standard Flow 22

3.2 FPGA Architecture representations 23

3.3 Command Line usage 25

4 Implementation **28**

4.1 Proto-Voter Cell 29

4.2 Fault Tolerant Routing Mux Python module 31

4.3 VTR Integration 37

5 Evaluation **41**

5.1 Evaluation metrics 43

5.2 Results 45

6 Conclusion **52**

References **54**

1 INTRODUCTION

Since their introduction, Field Programmable Gate Array (FPGA) have excelled at prototyping complex digital circuit designs while avoiding Non-Recurring Engineering (NRE) costs until the design is robust enough for fabrication. The current challenge of device scaling in FPGA devices is not the area dedicated to hard blocks, but the area dedicated to routing elements. Today, more than half of the FPGA total area and delay in implemented designs is dedicated to routing. Great part of this contribution comes from the area occupied by the programmable memory cells, which store information about which routing elements are enabled or disabled. The state-of-the-art for programmable memory cells is a 6-transistor Static Random Access Memory (SRAM) cell, which means any programmable switch needs to fit at least 7 transistors in the FPGA fabric area. Another disadvantage of this design is that the information is only retained in the memory cells as long as the device is connected to power sources, which can lead to unnecessary power usage in idle moments.

An alternative to transistor-based SRAM cells is memristor-based memory cells. Memristors are passive elements whose resistance is given by the historical current flow through the device. This resistive element can be used as non-volatile storage and its fabrication process is CMOS compatible. Memristors can not only be integrated with CMOS devices but also do not share the same fabric area of transistors by being constructed between Back End-Of-Line (BEOL) metal lines. A disadvantage of the use of memristors is that the yield of the fabrication process for memristors is prone to considerable failure rates. In an FPGA, these failures lead to completely unusable memory cells and, consequently, also unusable routing elements.

This work proposes a novel routing element architecture based on non-volatile memristive memory cells. The architecture is robust enough to withstand minimal to moderate defects in the memory cells while still successfully routing designs with high FPGA utilization. A Python framework is also developed to simulate and evaluate the proposed architecture.

This work is organized as follows: Chapter 2 introduces technical background and work of art for FPGA architecture, memristive systems, and fault-tolerant design in FPGAs. Chapter 3 explains the most relevant aspects of the Verilog to Routing (VTR) tool used for FPGA architecture simulation. Chapter 4 describes the novel memory cell architecture and the developed Python module to simulate it and how it is integrated into VTR. Chapter 5 explains the chosen evaluation metrics, results achieved and a mathematical proof for verification. Lastly, Chapter 6 summarizes findings and presents challenges met and possible extensions for this work.

2 BACKGROUND AND PREVIOUS WORK

2.1 FPGA Architecture

FPGAs are devices composed of programmable logic and routing elements and input and output (I/O) interfaces. Most modern devices include memory blocks and complex hard logic blocks such as Digital Signal Processing (DSP) blocks, multipliers as well as entire co-processors such as the UltraScale device family from Xilinx [4,5]. This advance enables the design of large and complex digital circuits through (re-)programming the device.

A Basic Logic Element (BLE) in an FPGA consists of a K-input Look-Up Table (LUT) and a flip-flop which can be bypassed to use the registered or unregistered output of the LUT as the output of the BLE [6]. Groups of N BLEs are usually bundled together in an FPGA with routing multiplexers for local interconnect to define a so-called Configurable Logic Block (CLB), illustrated in Figure 2. By implementing the truth table of any given part of a digital circuit in a LUT, CLBs can mimic the exact functionality of small parts of a circuit employing programming. The FPGA routing elements are then responsible to interconnect CLBs and realize the complete design.

Device scaling would allow fitting more LUTs with more inputs into a CLB in the same area. Such an idea would benefit from fast local interconnects by implementing more logic inside a CLB and relying less on the FPGA routing resources to minimize the delay. Nonetheless, it has been proven empirically in [7] that the best area-delay product is achieved with LUTs with 4-6 inputs and clusters of 3-10 BLEs, which are the values used in commercial devices [8].

One of the possible classifications of modern FPGAs is *island-style* FPGAs, which this work focuses on. On *island* FPGAs, logic and hard blocks are surrounded by vertical and horizontal channels of pre-fabricated, directed wiring segments and programmable routing switches [6]. The channel width defines the number of wire segments in a channel, and the wire length defines how many logic blocks a wire segment spans. At every intersection,

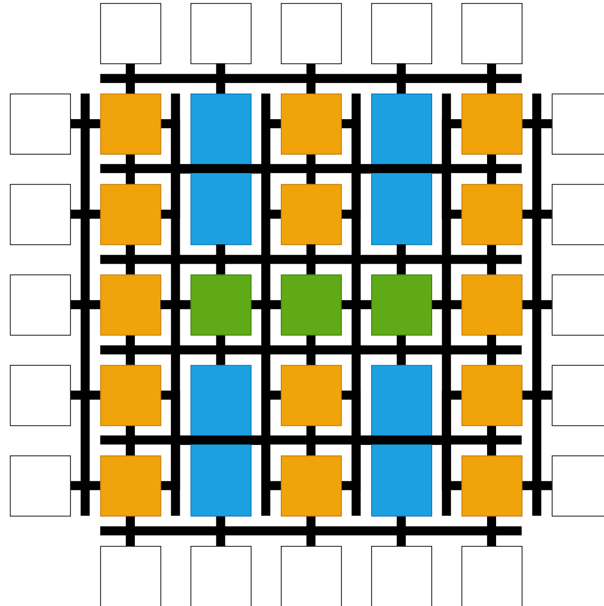
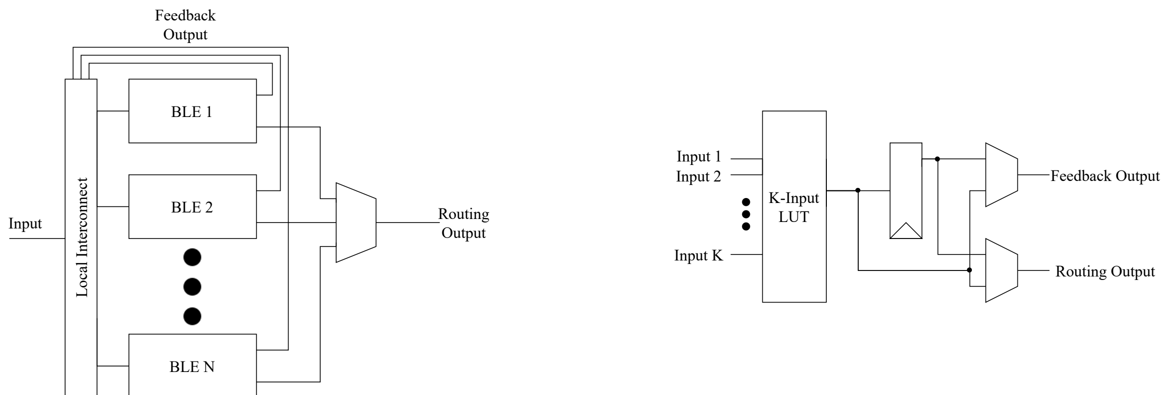


Figure 1: FPGA architecture example. White blocks are I/O, orange blocks are logic blocks, blue blocks are memory blocks and green blocks are multipliers. The black lines are a rough representation of wiring segments and routing elements. Source: author.

there is a switch box (SB) responsible for connecting an incoming wiring segment to an outgoing wiring segment at another channel, while the pins (input or output) of a logic block are connected to wiring segments through a connection block (CB). An overall view of *island* FPGAs is shown in Figure 1, and its routing architecture in detail is shown in Figure 3.

In modern devices, a single programmable switch is usually implemented as a Static Random-Acess Memory (SRAM) cell controlling a pass-transistor or a tri-state buffer. Both switch boxes and connection blocks are a collection of programmable switches and may present different architectures. Currently, they are usually implemented as a two-level pass-transistor based multiplexer as it was shown to provide the best area-delay product and is currently used in commercial devices [8, 9]. A patent filed by Xilinx is shown in Figure 4 that exemplifies such topology [1].

The physical limitation of FPGA devices is usually regarding routing elements. Previous work done by Chiasson and Betz has proven that routing elements account for more than 50% of the fabric area and critical path delay of designs in FPGAs [10]. Optimizing the routing architecture is then crucial to the device's performance. To describe the parameters of the routing architecture, the terminology proposed by Rose and Brown is used [11]. Here the previously introduced channel width is denoted by W, the number of logic blocks a wire segment spans is denoted by L for length, the number of wires a given CLB can connect to via a CB is denoted by $F_{c,input}$ and $F_{c,output}$ for connection block flex-



(a) Configurable logic block architecture (b) Detailed basic logic element design

Figure 2: Configurable logic block example. Source: author.

ibility of the respective pins, and the number of outgoing wiring segments able to connect to an incoming segment in an intersection is defined as F_s for switch box flexibility.

Optimizing the routing architecture is not a trivial task. Increasing the number of resources may lead to underuse and wasted area, while not enough resources do not allow enough flexibility to route complex designs. In [11] it was shown that the best area-delay product was achieved by F_s of 3 or 4 and F_c of $0.7W$ to $0.9W$. Later research then suggested lower $F_{c,input}$ and $F_{c,output}$, which is going to be adopted in this work [6]. Furthermore, in [10] was also shown that using transmission gates instead of pass-transistors could lead to a better area-delay product if gate-boosting is not available for the programmable switches.

2.2 Memristive Systems

The memristor was first proposed by Chua in 1971 as a nonlinear resistor with memory [12]. A memristor behaves as a resistor at any given instant in time, but its resistance is determined by the history of the current that flowed through the element. Later, Chua generalized the concept of memristive devices and systems to any circuit that could mimic the functionality of the original memristor [13]. Resistive Random Access Memories (ReRAM) is one such system and is going to be taken into account in this work. ReRAM is used nowadays for "in-memory computation", especially for deep learning workloads [14], but its fabrication process is CMOS compatible and could be used in FPGAs to increase device density, lower power consumption, and enable scenarios where the device is not connected to power at all times.

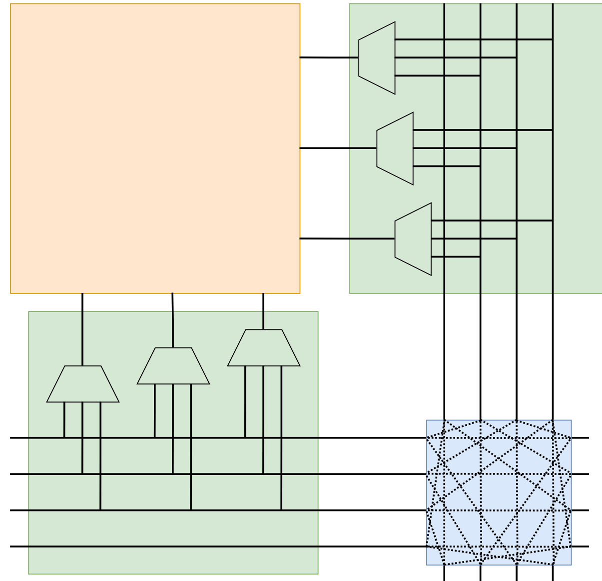


Figure 3: FPGA routing architecture in detail. A logic block is again represented in orange, while connection blocks are in green and a switchbox is shown in blue. Source: author.

FPGAs store their programmable information in SRAM cells, the most common array being a 6 transistor (6T) cell. This type of storage is volatile, meaning information is only retained as long as there is an external source providing power to the device. ReRAM, on the other side, can keep information stored in memristive elements indefinitely, potentially decreasing power consumption when not needing to be connected to a power source. By being compatible with CMOS fabrication process, ReRAM cells do not share the same fabric area as transistors by being built between the backend-of-line metal lines, increasing device density. In this work, the 2-Transistor-2-Memristor (2T2R) model proposed by [16] is going to be considered as the minimal resistive unit for information storage. In this model exemplified in Figure 5 two memristors are placed in series between V_{dd} and ground. The node between the two memristors is connected both to the gate of a routing switch and to another pass-transistor responsible to set their resistive states through a programmable voltage V_P. In an optimal use case, setting V_P accordingly and enabling the transistor connected to it sets one memristor in High Resistive State (HRS) and the other in Low Resistive State (LRS). If the pull-up memristor is in LRS and the pull-down memristor is in HRS then a logic "1" is stored in the cell and the routing switch is enabled. Following this terminology, it is also possible to name a memristor in LRS as turned "ON" or in a logic "1" state and analogously naming a memristor in HRS as turned "OFF" or in a logic "0" state. This terminology is important to explain memristor faulty behavior.

The effectiveness of the 2T2R cell depends on the ratio between HRS and LRS as

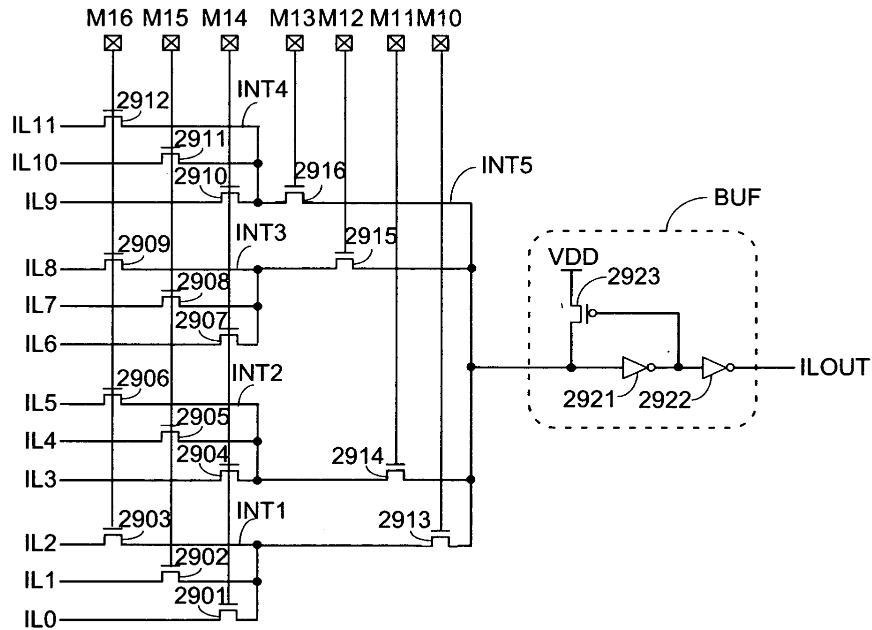
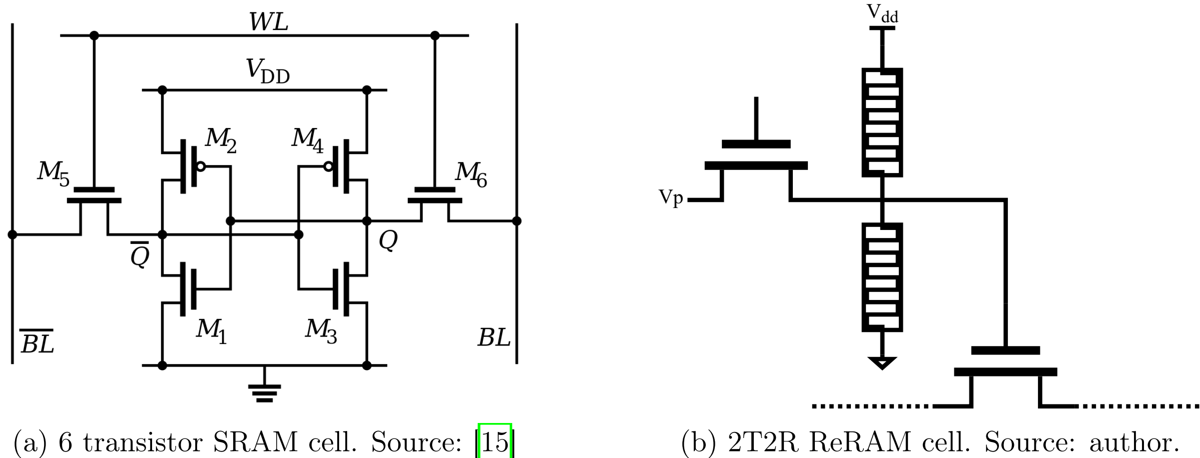


Figure 4: Routing multiplexer from [1]. From input to output there are always two pass-transistors acting as routing switches controlled by memory cells. At the output there is a buffer for signal restoration. The buffer presence makes the multiplexer unidirectional. Source: [1].

the memristors act as a voltage divider and current technologies offer a big enough ratio of about 10^4. Nonetheless, production defects in memristor may affect expected storage cell behavior and this work focuses on three specific defect cases also explored in [17]. If independently of V_P the memristor remains at LRS or HRS it is considered that the element is Stuck at 1 (SA1) or Stuck at 0 (SA0), respectively. If the memristor remains fixed in an intermediary resistive state that does not provide the expected difference ratio for the 2T2R to function properly it is said that the element presents Undefined (UD) behavior. Lastly, if the memristor does not present any kind of defect, it is then said to be Free of Failure (FF). From this definition, it is possible to determine the error propagated to the routing switch gate based on the error of the individual memristors of the 2T2R cell, shown in Table 1. If any memristor presents an undefined error or if both pull-up and pull-down present the same type of error, the memory cell will present undefined behavior and the routing switch cannot be used. If one of the memristors is SA1, the cell error takes preference on the pull characteristic: if pull-up is SA1 then the cell error is SA1, if pull-down in SA1 then the cell error is SA0. Analogously, if one memristor is SA0, the cell error takes preference on the unaffected pull network: a SA0 pull-up results in a SA0 error for the memory cell while a SA0 pull-down results in a SA1 cell error.

Non-volatile resistive RAM-based FPGAs were already explored in literature [18–20].



(a) 6 transistor SRAM cell. Source: [15]

(b) 2T2R ReRAM cell. Source: author.

Figure 5: Comparison between 6T SRAM cell and 2T2R ReRAM cell.

		Pull-up			
		FF	SA0	SA1	UD
Pull-down	FF	FF	SA0	SA1	UD
	SA0	SA1	UD	SA1	UD
	SA1	SA0	SA0	UD	UD
	UD	UD	UD	UD	UD

Table 1: 2T2R error based on pull-up or pull-down errors. Source: author.

Results are heavily dependent on the exact non-volatile technology used but aggressive improvements of up to 96% lower routing resource area, 67% less delay, and 79% less power consumption can be achieved. Although NV-FPGAs were already explored, the reliability of process yields for ReRAM technologies is rarely discussed. Even currently available processes such as the 40nm ReRAM process from TSMC [21] do not present 100% of yield. The presence of defects in routing elements directly affects the efficiency of an FPGA. The novel architecture proposed can withstand minor fabrication errors while still enabling circuit designs with high FPGA utilization to be routed and fully implemented.

2.3 Fault-Tolerant FPGA design

This work builds mainly on top of the work done by Freiberger on the evaluation of design routability in the presence of defect routing elements [17]. Freiberger simulated a 2T2R-based FPGA with memory blocks and fracturable inputs LUTs. With increasing error probabilities, he then simulated the effect of individual memristor defects in the global FPGA routing architecture and ultimately how it impacted the routability designs from the VTR and MCNC20 benchmarks that were successfully implemented in a error-

free FPGA. His results showed that the amount of routing edges grew quadratically to the number of defect memristors, as usually a single memory cell is responsible for controlling multiple routing switches. Additionally, it was shown that even for small designs with about 5% of the FPGA's logic blocks utilization could only be successfully routed with error rates of up to 0,375% per memristor.

Concerning fault-tolerant design in FPGAs, the usual trade-off is robustness and area-delay product. The most trivial solution is to increase the number of available routing resources in the device, but as shown before most of the FPGA area is dedicated to then routing area as this solution can lead to both increased area and a high amount of unused resources. Another solution is to implement the robustness not in the FPGA fabric but in the CAD tools, that identify defect elements through the use of test designs and can realize placing and routing of designs that partially or completely avoid defect areas, but finding a specific place and route solution for each device can be time consuming [22]. There may even be cases where a valid placement solution does not provide any viable routing path between two FPGA nodes due to defects and improving the CAD algorithms does not lead to a solution. Both of these solutions take advantage of a priori defect knowledge to handle them at the programming stage.

Another kind of approach is scenarios where FPGAs are exposed to a high amount of radiation (i.e. aerospace applications) and the configuration bitstream is corrupted by what is called Single Event Upset (SEU). In this case, Triple Modular Redundancy (TMR) or Quintuple Modular Redundancy (QMR) is used to replicate logic modules and depend on a majority voter circuit to forward the expected design implementation to the next domain [2]. TMR and QMR are designed to handle FPGA defects while the application is running in-field and there is no means of reprogramming the device. The approach is extremely robust as in TMR 2 out of the 3 replicas of a single logic module need to fail for the error to be propagated to the next stage while in QMR 3 out of 5 replicas need to fail. The downside is having to replicate each logic module thrice or five times in addition to adding a voter circuit between each set of replicas. For this work, the defects can be known beforehand and the information is used to take advantage of lower circuit redundancy, but the voter mechanism is somewhat adapted to add robustness to the memristor-based routing element this work presents in Chapter 4.

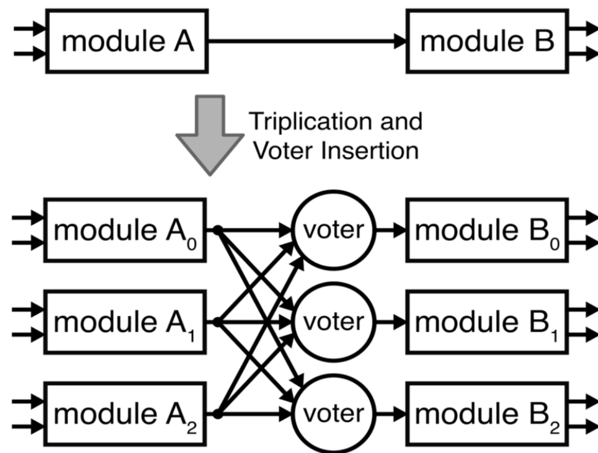


Figure 6: Exemplification of TMR design from [2].

3 VERILOG TO ROUTING (VTR)

3.1 VTR Standard Flow

VTR is a set of open-source Computer Aided Design (CAD) tools for FPGA architecture exploration and CAD algorithms research [3]. The standard execution flow of VTR shown in Figure 7 takes a Verilog description of a circuit design and an XML-file describing the target FPGA architecture and produces output files for the packing, placing and routing of the given design as well as area, speed, and power results. VTR starts by using Odin II to synthesize the high-level description of the circuit into a flattened netlist of logic gates, flip-flops, and complex logic blocks such as adders, multipliers, and RAM slices. As an option, VTR can also be built to use Yosys together with Odin II in the elaboration stage to increase Verilog-2005 support and optimize device utilization. Next, a technology-independent logic optimization is done by ABC, which maps the circuits into LUTs, flip-flops, and blackboxes. Using the technology-mapped circuit file, Versatile Place & Route (VPR) parses the architecture file to generate a routing resources graph for the target architecture and runs packing, placement and routing on the design. The XML architecture files do not necessarily specify the device size, only the arrangement of logic and hard blocks. By default, VPR creates a big enough device to fit all blocks in the technology-mapped netlist with the smallest channel width that allows successful routing.

The toolflow is highly modular as every stage produces an output file that can be independently fed into the following stage by other sources. Odin II takes a Verilog circuit description and an XML architecture file as input and produces the netlist in a Berkley Logic Interchange Format (BLIF) file, and ABC uses BLIF netlists both as input and output. The complete flow of VPR takes a BLIF netlist and an XML architecture file as input and produces output files for packing, placing and routing stages, as well as a final output file with statistics on each stage. Alternatively, VPR can also save an XML file of the routing resources graph from the target architecture represented as routing

nodes and edges. In this work, the Routing Resources (RR) XML file is used for the simulation, edited with defect elements and fed back into VPR, which also accepts the RR file directly. Ultimately, VPR can run any stage independently using an adequate packing, placement or routing file as input.

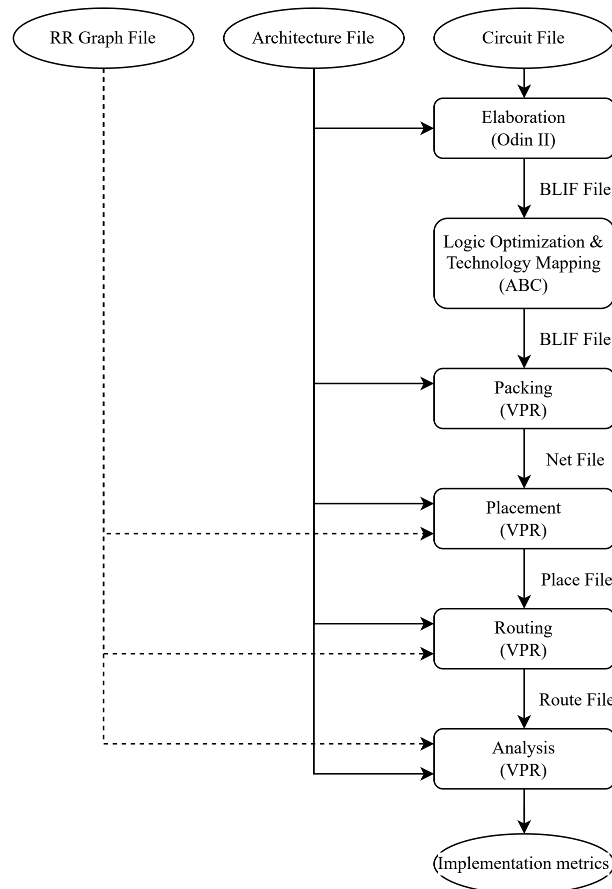


Figure 7: VTR standard flow. Source: author.

3.2 FPGA Architecture representations

The FPGA architecture file provided to the VTR flow has all the information needed to describe the device's architecture. The file's format is defined by VTR and although extensive, some points are crucial to this work and therefore briefly explained. The first tag in the file is always *<architecture>*. The possible top-level tags are the following:

- **<models>** Contains a list of supported BLIF sub-circuit (*.subckt*) model names. Important for using dedicated hard blocks.
- **<tiles>** Contains a list of functional blocks and their properties to compose the FPGA grid. All blocks need to be defined in the *<complexblocklist>* tag.

- <**layout**> Defines a pattern of how tiles should be arranged in the FPGA grid. If an *<auto_layout>* is defined, VPR creates a device following the defined pattern with the minimal size needed to fit a chosen design, otherwise, if a *<fixed_layout>* is defined, an arbitrary width and height need to be written as tag parameters to create a device with a fixed grid size.

- <**device**> Contains the resistance of minimum-width transistors, the routing switch type used, the name identifying a connection block switch in *<switchlist>* and the area occupied by a 1x1 logic block.

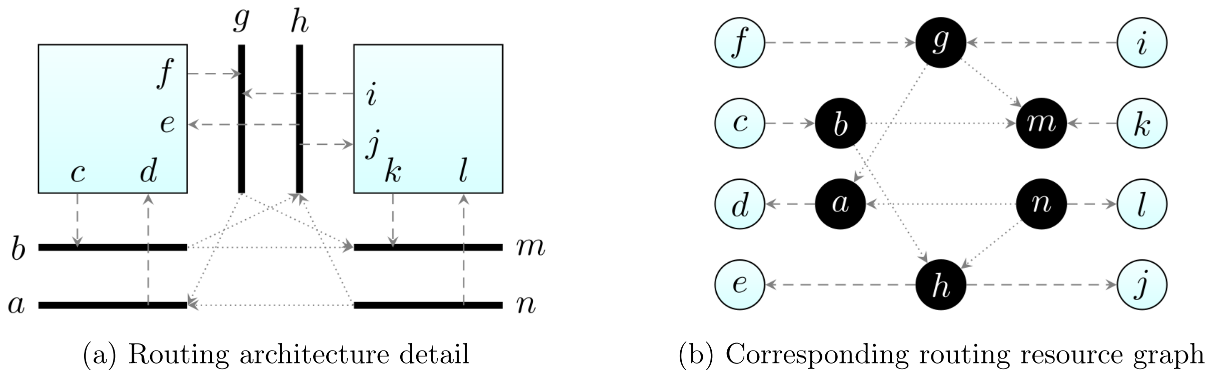
- <**switchlist**> Specifies the routing switches used. Each switch has a type, a name and physical properties.

- <**segmentlist**> Specifies each type of wiring segment used in the architecture.

- <**complexblocklist**> Defines each functional block to be used in the *tiles* section.

What is of most importance in this work are the *<device>* and *<switchlist>* tags. The *<device>* tag defines both the switchbox type, meaning how an incoming wire is routed to another track, and the name of the connection block defined in the *<switchlist>* tag. The available switchbox types are the planar switchbox used in Xilinx 4000 FPGA, the Wilton [23] switch, the universal [24] switch, and a custom-defined switch. For this work, the chosen switchbox type is the Wilton switch. The *<switchlist>* tag can have at least two switches, and with the information from the *<device>* tag, it is possible to know which switch "id" represents a connection block, while the remaining switches represent switchboxes. Regarding switch types, the available options are a multiplexer, a tristate buffer, a pass transistor, a simple wire or a non-isolating buffer. For this work, the only switch type considered is the multiplexer.

From the architecture file, VTR creates an intermediary representation of routing resources in a so-called RR graph file. This file is also in an XML format and it represents pins as nodes and routing elements connections as edges. The relation between the actual architecture and the RR graph is illustrated in Figure 8. The most important tag in the RR graph file is the *<rr_edges>* tag, which contains a list of all routing edges in the device generated by VTR. Every *<edge>* tag inside *<rr_edges>* has the id of its source node, sink node, and routing switch. With the three parameters, it is possible to identify if the edge is from a CB or a SB and it is possible to recreate the element to later simulate its defects by grouping edges that share the same sink node.



(a) Routing architecture detail

(b) Corresponding routing resource graph

Figure 8: Routing resource graph representation from [3]. Dotted lines are switchbox connections and dashed lines are connection blocks connections.

VTR is bundled with multiple sets of architecture files, ranging from FPGAs with only logic blocks to more complex FPGAs with memory slices and DSP blocks. The architecture used is described in the "k6_frac_N10_mem32K_40nm.xml" file. The architecture is chosen to provide a direct comparison with the work done by Freiberger. It provides a reasonable delay model for a 40 nm technology device based on the commercial Stratix IV with fracturable 6-input LUTs organized in clusters of 10 LUTs each. The architecture also has 32 kilobits of memory slices with variable data width and multipliers with 36 and 18 bits of width. As described in the VTR documentation, the transistor size is modeled after the iFAR architectures [25] which is based on 45 nm node technology and linearly scaled down. The routing parameters of the architecture are $L = 4, F_{c,input} = 0.15, F_{c,output} = 0.1$. The channel width can be user-defined or is automatically set to the minimum needed to route a chosen design.

3.3 Command Line usage

VTR is run through the command line and a comprehensive explanation of all of its execution options is available in its documentation. This work will only explain relevant options used to ensure reproducibility. The first important script used in the tool is "run_vtr_flow.py", which has two required arguments: a circuit description in Verilog and an FPGA architecture file. Without further arguments, "run_vtr_flow.py" executes the complete flow described in Figure 7. To run specific stages, the options `--starting_stage` or `--end_stage` can be used, followed by the desired starting or end stage name. The `--device` option allows the definition of a fixed layout for the target architecture. The prerequisite is defining the layout inside the *<layout>* tag in the architecture file and using its name following the option. Every architecture needs to have

at least one <*auto_layout*> defined which VPR uses as default. To set a fixed channel width the option `--channel_width` followed by the desired channel width is used. Additionally, `--write_rr_graph` and `--read_rr_graph` followed by a file name can be used to save or load a RR graph file. "run_vtr_flow.py" automatically parses all options and forwards them to their respective stage. If options unknown to VTR are included, "run_vtr_flow.py" forwards them to VPR. An example of how the script is run in this work is in Listing 3.1, where `$VTR_ROOT` is the directory where VTR is installed.

```
$> $VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py circuit.v fpga_arch.xml --
   ↪ device fixed20x20 --channel_width 60 --write_rr_graph rr_graph.xml
```

Listing 3.1: VTR example usage where $VTR_ROOT is the directory where the tool is installed.

The second script used is "run_vtr_task.py", which takes a VTR task name as input. VTR tasks provide a framework to run scripts over a set of benchmark and architecture files with multiple parameters. To setup a task, a directory with the task's name needs to be created under *$VTR_ROOT/vtr_flow/tasks* and it needs to contain a folder called "config" with a single configuration file inside called "config.txt". The configuration file sets the architectures and benchmarks directories through *archs_dir* and *circuits_dir* directives. Each architecture and benchmark file is then set through *arch_list_add* and *circuit_list_add* directives, respectively. The script to be run in the task can have two sets of options to be passed to each invocation. The first set is passed to every invocation and the second set has each entry added to the first set separately. This is useful to define a common set of options along with a second range of options to be swept. Executing "run_vtr_task.py" runs the defined script ("run_vtr_flow.py" by default) over the vectorial product of the architecture files list, benchmark files list, and script parameters list. The results of all executions are parsed at the end by defining a parse file, which is a collection of files and regular expressions to be parsed from all script executions and compiled in a single file as tabular data. An example of a VTR task "config.txt" file is shown in Listing 3.2.

```
# Path to directory of circuits to use
circuits_dir=benchmarks/verilog

# Path to directory of architectures to use
archs_dir=arch/timing

# Add circuits to list to sweep
```

```
circuit_list_add=ch_intrinsics.v
circuit_list_add=diffeq1.v

# Add architectures to list to sweep
arch_list_add=k6_N10_memSize16384_memData64_stratix4_based_timing_sparse.
    ↪ xml

# Parse info and how to parse
parse_file=vpr_standard.txt
```

Listing 3.2: VTR task configuration file example from [3].

The script also accepts options of its own. The number of tasks to be run in parallel can be defined with the option `-j` followed by the number of threads to create and additional parameters can be passed to the task script by using the option `-s` followed by the options set. An example of the command-line usage of "run_vtr_task.py" is shown in Listing 3.3.

```
$> $VTR_ROOT/vtr_flow/scripts/run_vtr_task.py task_name -j 8 -s --
    ↪ channel_width 60
```

Listing 3.3: VTR task example usage. $VTR_ROOT is the installation directory of VTR.

4 IMPLEMENTATION

The 2T2R arrangement from [16] and Freiberger's evaluation [17] are taken as a starting point. The 2T2R arrangement uses two memristors and one transistor as an information storage unit and a second transistor as a programmable switch. Freiberger considered three possible types of error affecting these memristors - SA0, SA1 and UD - and their effects in the memory cell (Table 1). Connection blocks and Switch boxes use 2-stage pass-transistor multiplexers as routing switches (Figure 4), and a cell error can have multiple effects on the structure.

In the most trivial case, if one memory cell is afflicted by a SA0 error, all switches controlled by the storage cells and their respective paths cannot be used as they will never be enabled. If one memory cell is afflicted by a SA1 error, then the switches controlled by it are always going to be active, and so should their respective routing paths. This situation implies two consequences: if in a given stage only one cell is afflicted by a SA1 error, then the affected path is the only possible usable path in the multiplexer; if, on the other hand, more than one path if afflicted by a SA1 error, then both paths are active and a potential short circuit is present at the end of the stage and the routing multiplexer cannot be utilized. If a storage cell is affected by a UD error then the programmable switch behavior is undefined and the routing multiplexer cannot be used to avoid short circuits as well. This error relation is summarized in Table 2.

From this behavior, it is possible to define an error tolerance per multiplexer stage. Given a routing multiplexer stage with n inputs, the stage can tolerate up to $n-1$ SA0 errors, up to 1 SA1 error, and no UD error. If both stages respect this tolerance the multiplexer is still usable. Clearly, the most critical type of error to a multiplexer stage is UD due to its prevalence over the other types, followed by SA1 errors which cannot occur more than once, and SA0 being the most permissive type of error. As long as there is a single routing path in every stage not afflicted by SA0, the routing multiplexer is still usable using solely the path not affected by SA0. Note that in this situation the viable path may even be affected by a SA1 error.

Second stage error	First stage error: FF	SA0	SA1	UD
FF	All paths usable	SA0 paths unusable	Only SA1 paths usable	Multiplexer unusable
SA0	SA0 paths not usable	SA0 paths not usable	Only SA1 paths usable excluding SA0 paths	Multiplexer unusable
SA1	Only SA1 paths usable	Only SA1 paths usable excluding SA0 paths	Only SA1 paths usable	Multiplexer unusable
UD	Multiplexer unusable	Multiplexer unusable	Multiplexer unusable	Multiplexer unusable

Table 2: Effect of memory cell errors on a routing multiplexer

4.1 Proto-Voter Cell

A novel architecture of defect-tolerant routing element is proposed in Figure 9 inspired by the voter design of TMR. In this architecture, there are two memory cells controlling a single routing switch, which are called the main and the control memory cell. Each of these individual cells follows the same layout as the 2T2R cell without the programmable switch. The control cell selects which signal to drive the routing switch gate and the main cell is one possible option of driving signal. If the control cell is storing a logic "1", the main cell can directly controls the programmable switch. If otherwise the control cell is storing a logic "0" the programmable switch gate is connected to ground and therefore disabled. The control cell acts as voter to enable programmability or completely disable a programmable switch if the main cell presents harmful errors to the routing multiplexer architecture. The complete arrange is from now on called proto-voter cell.

In the proposed design, if the main cell has any error that would hinder the routing multiplexer utility, such as a UD error or a second SA1 error in a single stage, the control cell can select the ground terminal to drive the routing switch and force a SA0 error. On the other hand, if the control cell is afflicted by a UD error, the main cell can be programmed to store a logic 0 and thus paired with the ground so there is no signal to be driven and the programmable switch is again safely "always OFF". A third scenario is a main cell with a SA1 error and a defect-free control cell, which can freely switch between ground and the SA1 main cell to virtually pose as a defect-free cell from the programmable switch point of view. The programmable switch gate behavior using a proto-voter cell arrangement is summarized using Table 3

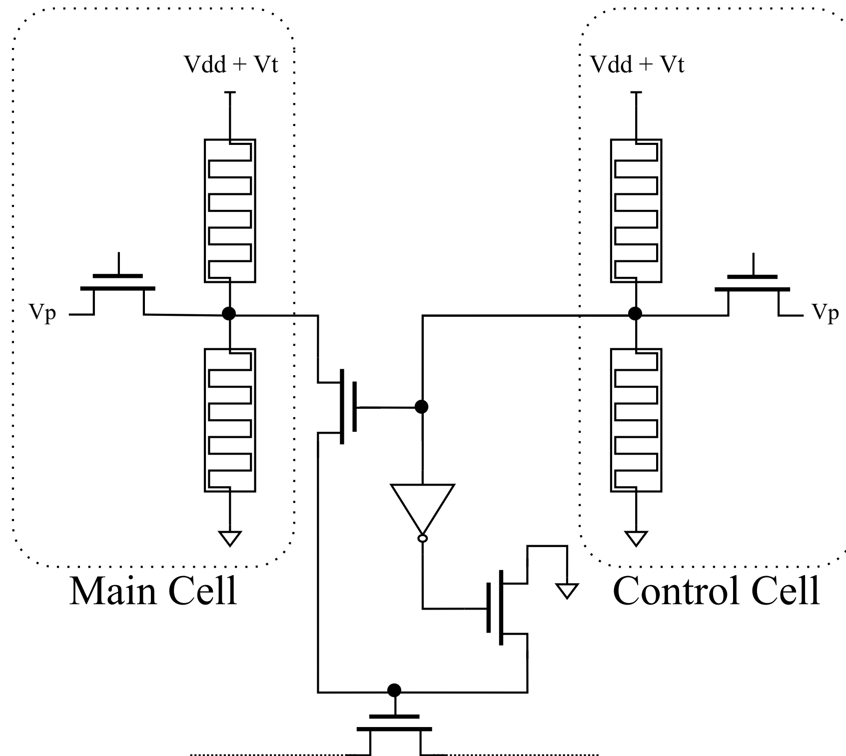


Figure 9: Proto-Voter cell schematic

		Control cell			
		FF	SA0	SA1	UD
Main cell	FF	FF	SA0	FF	SA0
	SA0	SA0	SA0	SA0	SA0
	SA1	FF	SA0	SA1	UD
	UD	SA0	SA0	UD	UD

Table 3: Programmable switch gate behavior using a proto-voter cell as NV-memory. Source: author.

The proto-voter cell prioritizes disabling defect routing paths to keep the routing multiplexer usability. In contrast to the base 2T2R cell, a proto-voter cell will only forward a UD error to the routing switch if both the main and control cells are afflicted by a UD error or if the pair has both SA1 and UD errors. For SA1-type of error, the robustness is even better: a SA1 error will only be forwarded to the routing switch gate if both the main and control cell are afflicted by a SA1 error. If only one of the cells is afflicted, the other can still provide a logic "0" state, directly through the main cell or by selecting the transistor connected to the ground through the control cell.

The proto-voter cell arrangement presents some design challenges. First of them is the pass-transistor cascade arrangement. For NMOS the charging profile of the device is shown in Figure 10. The output of the pass-transistor can only be charged up to $V_{dd} - V_t$

where V_t is the threshold voltage of the device, which is a considerable signal degradation for state-of-the-art technologies. As an example, 22 nm node technology is used in [10] and for 0.8 V of V_{dd}, the output can only switch between 0 V and 0.55 V, with a very slow slew rate above 0.45 V. For the proto-voter cell, the routing switch gate signal may not be strong enough when the main cell is driving a logic "1". To solve the challenge, gate-boosting is proposed, where a voltage greater than V_{dd} is applied at a transistor's gate.

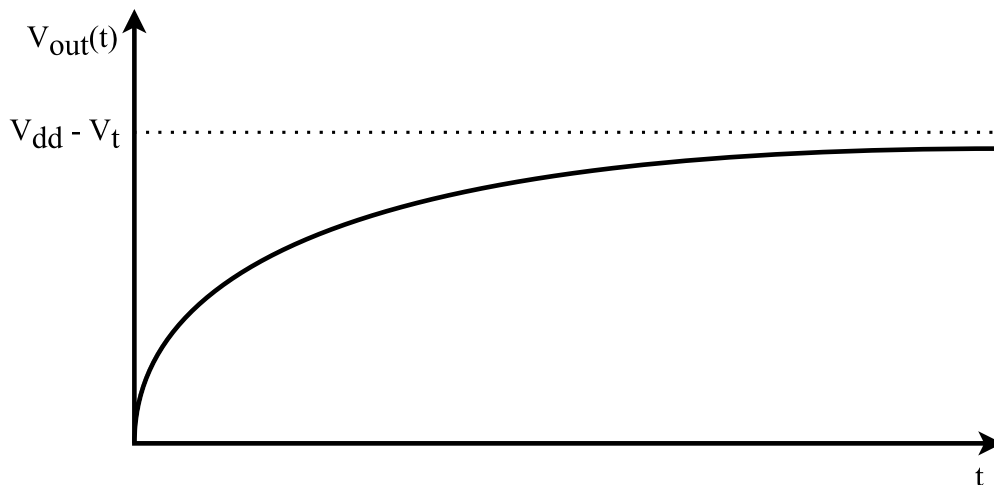


Figure 10: Charging profile of a NMOS device used as pass transistor. Source: author.

Gate-boosting itself is also a challenge because it degrades the device due to hot carrier effect and an extra voltage domain is needed for the FPGA. It can be argued that the presence of NV-storage will allow the device to not be connected to a power source thus prolonging the device's lifespan. For the extra voltage domain, it can also be justified through the use of the memristors. The characteristics of the 2T2R require +3 V and -3 V for programming, the pull-up memristor to be connected to +1.2 V and the pull-down memristor to be connected to 0 V [16]. As the memristor already requires different voltages, the extra power domain is justifiable. Using 1.2 V as V_{dd} means the routing switch gate will drive at logic "1" about 0.95 V, meaning 150 mV of gate-boosting, which is close to the 200 mV Betz found experimentally to yield a better area-delay product when using pass-transistors instead of transmission gates in [26].

4.2 Fault Tolerant Routing Mux Python module

To simulate both the base 2T2R memory cell and the proto-voter cell, a Python module is developed called Fault-Tolerant Routing Mux (FTRM). The choice of Python

is for ease of integration with VTR scripts and previous knowledge from the author. The module is able to run a standalone simulation of an arbitrary number of independent routing multiplexers or run the simulation based on a given RR Graph file.

The module has a high-level wrapper represented in figure 12 and some low-level support classes represented in figure 11. The *Errors* class simply stores an enumerated definition of each memristor error type, including the error-free (FF) state. "FF" is stored as a 0, "SA0" is stored as a 1, "SA1" is stored as a 2, and "UD" is stored as a 3. The *RandomNumberGenerator* class stores absolute probabilities for SA0, SA1 and UD errors and generates a pair of errors following the given distribution through the use of the instance function *gen* for generate. The *MemCell* class represents a 2T2R memory cell without both transistors. It has two instance members representing the pull-up and pull-down memristor states and a 2D matrix as a class member acting as a LUT for the cell error based on Table 1 and the *Errors* class structure. The *set_errors* function set each memristor state from errors received from a *RandomNumberGenerator* instance and the *get_cell_error* function returns the error matrix value at the position given by the pull-up and pull-down state. The *ProtoVoterCell* class represents the novel cell design. It has the same structure as the previous class: a lookup matrix for the cell error and two instance members, this time being *MemCell* classes representing the main and control cells in the structure. It also has a *set_errors* and *get_cell_error* which work analogously to the *MemCell* functions with the addition of receiving 2 pairs of errors to be set and forwarding each pair to one of the *MemCell* members.

To represent a routing multiplexer, three classes are used: *RoutingMux*, *FirstStageBlock*, and *SecondStageBlock*. The last two classes represent a first-stage multiplexer block and a second-stage multiplexer block. Both classes have as instance members the id of its sink pin, a list of source pins, a list of memory cells, and a boolean variable to show if the block is usable or not. The main difference between both stages is that the second stage block has a list of lists for source pins, as each of its input pins controls multiple source pins from the first stage. The available functions for the classes are *set_errors*, *compute_block_error* and *get_defect_edges*. The first function takes a *RandomNumberGenerator* instance as input and forwards error pairs to each element in the memory cell list, the second retrieves each cell error and computes the block error following the logic in Table 2 while also setting if the block is unusable and the third function returns which source-sink pair cannot be used due to errors.

The *RoutingMux* class contains its sink (output) node, a list of source (input) nodes, and its memory cell type. After being created, an instance of this class computes the

```
def compute_optimal_block_size():
        mux_size = len(source_nodes)
        block_size = mux_size # initial block size is mux_size
        n_mem_cells = mux_size # initial number of cells is mux_size
        partial_block = False # Partial block

        for new_block_size in range(1, mux_size + 1):
            # Check if a partial block is needed
            partial_block = (mux_size % new_block_size) != 0
            # Calculate new number of memory cells
            new_n_mem_cell = new_block_size + (mux_size // new_block_size)
                ↪  + partial_block

            # If less memory cells are used, save new block size
            if (new_n_mem_cell < n_mem_cells):
                n_mem_cells = new_n_mem_cell
                block_size = new_block_size

        return block_size
```

Listing 4.1: Algorithm to find optimal block size. Source: author.

optimal block size for the first stage to use a minimal number of memory cells. The algorithm used for this calculation is shown in Listing 4.1, taken from [17]. It tests the multiplexer block size ranging from 1 up to the number of inputs, creating blocks of equal input number plus a possible last block with fewer inputs if the integer division of inputs by block size has a remainder. From the 2-stage structure shown in Figure 4, the number of memory cells in the first stage is the maximum number of inputs from a first-stage block, and the number of cells in the second stage is the number of firsts stage blocks. The smallest block size with an optimal number of memory cells is chosen and the optimal number of memory cell instances is created according to the chosen type. Each cell is then stored in separate lists from the first and second stages to correctly compute their error effects in the structure. To set and retrieve the errors in each cell the instance functions *set_errors* and *get_cell_errors* recursively call the functions with the same name from each memory cell. For testing reasons, the effect of cell errors on the routing multiplexer is only computed after calling the function *compute_block_errors* which again recursively calls the function with the same name inside a block type class.

The *FaultSimulator* class serves as a wrapper for both simulation types, containing a *RRGraphParser*, a *RandomErrorGen*, a list of *RoutingMux* instances, and the desired cell type, being the latter either *MemCell* or *ProtoVoterCell*. *RRGraphParser* is a class responsible for parsing a RR graph file and storing its information in Python data struc-

tures. It uses the Python library *ElementTree* to load, update and rewrite the RR graph file. After loading the file, it searches for the *switches* node, where it can identify the *id* number of switchboxes and connection blocks. The connection block id is identified by the switch whose name is *ipin_ cblock*, and the switchbox is identified by the remaining switch whose name does not have "delayless" in it, as otherwise are dummy switches used by VTR to create the device. After each switch id is identified, every *node* in *rr_ nodes* is parsed and a dictionary is created indexing source pins by their sink pin. Each dictionary entry represents the pins from a routing multiplexer used in a SB or CB. Together with the memory cell type, the pins dictionary is used to create *RoutingMux* class instances representing a routing multiplexer.

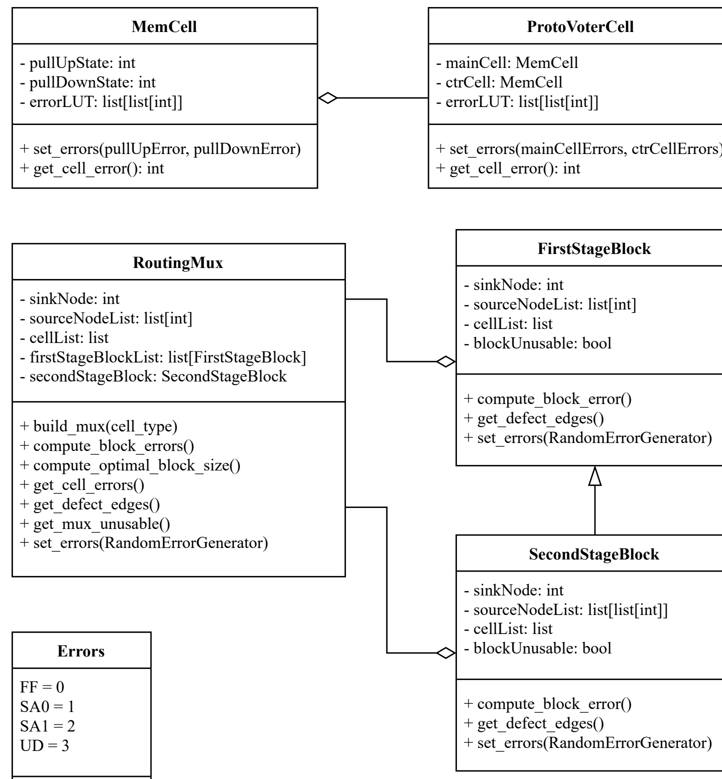


Figure 11: Low-level UML class diagram of fault-tolerant-routing-mux module. Source: author.

To run the simulation based on the loaded RR graph file, the *FaultSimulator* class has to be instantiated with a cell type, an RR graph file name, and either a single probability P_{equal} or individual probabilities for each type of error P_{SA0}, P_{SA1} and P_{UD}. If a single probability P_{equal} is given, its value is attributed to each error type. This means for $P_{equal} = 0.01$ a given memristor will have $P(SA0) = P(SA1) = P(UD) = 0.01, P(FF) = 0.97$ where $P(SA0)$ is the probability of a SA0 error occurring, $P(SA1)$ is the probability of a SA1 error occurring, $P(UD)$ is the probability of a UD error occurring, and $P(FF)$ is the probability for the memristor to remain error-free. The error probabilities are

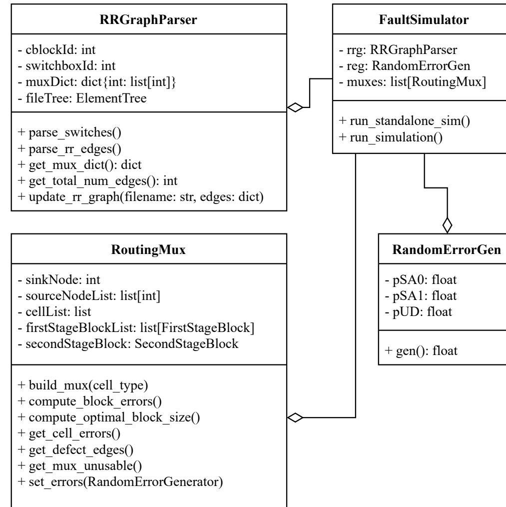


Figure 12: High-level UML class diagram of fault-tolerant-routing-mux module. Source: author.

considered to be independent between memristors, which means for example that the probability for a 2T2R cell to stay error-free is given by $P_{pull-up}(FF) \cdot P_{pull-down}(FF)$.

The RR graph file is then loaded and parsed by a *RRGraphParser* instance and a *RandomErrorGen* instance is created with the input probabilities. With the parsed RR graph and the given memory cell type, routing multiplexers are created and stored in a list. By calling *run_ simulation*, the function *set_ errors* is called in each multiplexer providing the aforementioned *RandomErrorGen* instance to simulate memristor errors in each cell of the routing multiplexer. Then the block errors are computed and for each multiplexer, the defect cells are stored in a list and the defect edges are stored in a dictionary indexed by sink pin. The simulation also keeps a counter of how many multiplexers are unusable.

After the simulation, the defect edges dictionary is used to update the RR graph file. The *ElementTree* class stores the XML-structured information from the RR graph file as Python lists and dictionaries, which are not efficient to iterate through due to their size. For efficiency, two Python sets are created. The first set contains all routing edges from the *rr_ edges* node and the second is a set of source-sink tuples created from the defect edges dictionary as the pairs are always unique. The latter is used to find every node corresponding to a defect edge and save these nodes in a new set of defect edges. Then the set of defect edges is subtracted from the set of all edges to keep only the defect-free routing edges, already typed as *ElementTree* nodes. For speed, the complete *rr_ edges* node is removed from the original *ElementTree* root and a new one with the same name is created using the defect-free edges set as child nodes. The RR graph file is then updated through the *write* method of the *ElementTree* class.

The last step in the simulation is to generate a report. The report header contains the architecture and circuit design files used, the simulation time, the time spent to update the RR graph file and each error probability considered. The report body contains the absolute and relative number for each error occurrence in the memory cells and defect edges, as well as the percentage of unusable multiplexers. For debugging reasons the report also provide the total number of routing edges in the device and the total number of routing multiplexer edges.

The second type of simulation is implemented as a class function and can be run independently. The standalone simulation takes as input a number n of multiplexers to be created with a chosen cell type T and an array of failure probabilities P_{equal} which works the same as above. The standalone simulation creates the n multiplexers with the given memory cell type and simulates each memristor error as independent probabilities. At the end of the simulation, the effect of the error in each memory cell is computed followed by the effect on the multiplexers. As an output, the simulation delivers a report containing the number of defect cells detailed with the count of each error, the number of defect routing paths, and the number of unusable multiplexers. All of these metrics are given for each error probability in the input array. Listing 4.2 shows an example usage of the python module by itself.

```
import fault-tolerant-routing-mux as ftrm
import numpy as np

# Standalone simulation
p_array = np.arrange(0., 0.0006, 0.0001)
num_muxes = 10000
ftrm.FaultSimulator.standalone_sim(p_array, num_muxes, ftrm.MemCell)

# RR graph file simulation
fault_sim = FaultSimulator(ftrm.ProtoVoterCell, "arch_file.xml", p=0.005)
fault_sim.run_simulation()
```

Listing 4.2: FTRM example usage. Source: author.

4.3 VTR Integration

The Python module is also integrated into the VTR toolflow for ease of use. The toolflow described in Chapter 2 is implemented in "$VTR_ROOT/vtr_flow/scripts/flow.py" and the "run_vtr_flow.py" script acts as a wrapper for it. As both the toolflow and the developed module are written in Python, the first step is to import the *fault-tolerant-routing-mux* module inside the "flow.py" file. As the module is related to the routing part of the toolflow, it is integrated in the VPR stage, and six additional command line arguments are added. As the new options are not recognized by VTR, they are automatically forwarded to the VPR stage by the "run_vtr_flow.py script".

The first argument is `--fault_sim`, which when used, runs the fault simulation with a default $P_{equal} = 0.0005$ considering 2T2R memory cells. The second new argument is `--cell_type`, which has to be followed by either `MemCell` or `ProtoVoterCell` to set which class is to be used as memory cells in the fault simulation. The last four arguments are responsible for setting the fault probabilities. Using `--equal_fault`, `--psa0`, `--psa1` and `--pud` followed by a number between 0.0 and 1.0 respectively sets P_{equal}, $P(SA0)$, $P(SA1)$ and $P(UD)$. If `--equal_fault` is used with a value greater than 0.33 it defaults to 33% chance for each error type (and 1% of a chance of staying error-free). The use of `--equal_fault` also overwrites any other command for individual probabilities. A summary of added options in shown in Table 4 and two usage examples of this integration are shown in Listing 4.3.

```
$> $VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py circuit.v fpga_arch.xml --
   ↪ device fixed20x20 --channel_width 60 --write_rr_graph rr_graph.xml
   ↪ --fault_sim --equal_fault 0.0001 --cell_type ProtoVoterCell

$> $VTR_ROOT/vtr_flow/scripts/run_vtr_flow.py circuit.v fpga_arch.xml --
   ↪ device fixed20x20 --channel_width 60 --write_rr_graph rr_graph.xml
   ↪ --fault_sim --pud 0.0001 --psa1 0.0002 --psa0 0.0003
```

Listing 4.3: VTR example usage with FTRM integration. $VTR_ROOT is the directory where VTR is installed. Source: author.

The expected execution flow assumes the user has defined a fixed device channel length. If in addition to a fixed channel width VPR is set to write a RR graph file, execute routing or execute analysis, flow.py runs VPR twice to ensure the produced files in the first run can be re-loaded into VPR. To take advantage of this behavior, an extra

Command line option	Explanation
`--fault_sim`	Runs defect simulation
`--cell_type MemCell|ProtoVoterCell`	Define memory cell type to use in simulation
`--psa0 p`	Set $P(SA0) = p$
`--pSA1 p`	Set $P(SA1) = p$
`--pSA1 p`	Set $P(UD) = p$
`--equal_fault p`	Set $P(SA0) = P(SA1) = P(UD) = p$

Table 4: Command line options to run defect simulation with VTR. Source: author.

condition is added to run VPR a second time if `--fault_sim` is used. Listing 4.4 shows the final execution flow. If `--fault_sim` is used, all other simulation arguments are parsed and removed from the arguments list and `--write_rr_graph` is added if not yet included as a RR graph file is needed for the simulation. Then VPR is run once, followed by the defect simulation, and then VPR is run again with the modified RR graph file from FTRM as input. An important implementation detail is that between VPR runs the `--write_rr_graph` argument is removed and swapped by `--read_rr_graph`, otherwise VPR generates another RR graph file in the second run and compares lazily to the RR graph from the first run. Even in the case of no defect cells being removed from the original file, FTRM may swap node order in the original file and the VPR check fails. The final VTR flow with the defect simulation is shown in Figure 13.

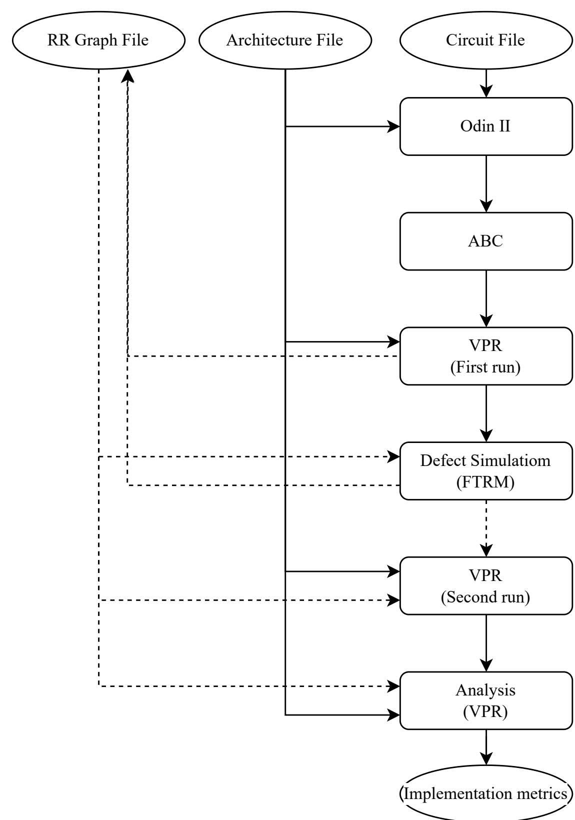


Figure 13: VTR CAD flow with integrated defect simulation of routing resources

```
import fault-tolerant-routing-mux as ftrm
...
   # Pack/Place/Route
   if should_run_stage(VtrStage.VPR, start_stage, end_stage):
       ...
       if route_fixed_w:
           # The User specified a fixed channel width
           do_second_run = False
           do_fault_sim = False
           second_run_args = vpr_args

           if (...) or "fault_sim" in vpr_args:
               do_second_run = True

           if "fault_sim" in vpr_args:
               # Set sim run and pop argument for VPR
               do_fault_sim = True
               ftrm_args = parse_and_pop_ftrm_args(vpr_args)

               # Save rr_graph for simulation
               if "write_rr_graph" in vpr_args:
                   # gets current defined RR graph filename
                   fault_sim_rr_graph = temp_dir / vpr_args["
                       ↪ write_rr_graph"]
               else:
                   # defines a default RR graph file to be saved
                   fault_sim_rr_graph = architecture_copy.name + ".xml"
                   vpr_args["write_rr_graph"] = fault_sim_rr_graph

           # First VPR run
           vtr.vpr.run(...)

           if do_fault_sim:
               # Run simulation before second VPR run
               fault_sim = ftrm.FaultSimulator(ftrm_args)
               fault_sim.run_simulation()

               # pop write argument to avoid second run check fail
               vpr_args.pop("write_rr_graph")
               # add read_rr_graph in place
               vpr_args["read_rr_graph"] = Path(fault_sim.
                   ↪ get_faulty_rr_graph()).name

           if do_second_run:
               # Run vpr again with additional parameters.
               # This is used to ensure that files generated by VPR can be
                   ↪  re-loaded by it
               vtr.vpr.run_second_time(...)
       ...
```

Listing 4.4: Integration of fault-tolerant-routing-mux module into flow.py. Some code parts omitted for simplicity. Source: author.

5 EVALUATION

To evaluate the proposed architecture, the defect simulation is run on the chosen architecture from Chapter 3 and on circuits from two benchmarks: MCNC20 and the VTR circuit benchmark [3,27]. The MCNC20 benchmark is composed of small to medium sized circuits for today's standards. Most of them only make use of logic blocks, but they are nonetheless used to directly compare the work done by Freiberger and to demonstrate the novel architecture effect on simpler designs. The VTR benchmark on the other hand is composed of medium to large designs which make use of hard blocks such as multipliers and memory slices. The designs are a good baseline of analysis as there are relevant implementations such as the Secure Hash Algorithm (SHA). A subset of circuits from the VTR benchmark is chosen as some are too large to enable multiple test runs due to simulation time.

To ensure VPR creates a device of fixed sized, two fixed FPGA layouts are included in the target architecture file used in this work (k6_frac_N10_mem32K_40nm.xml): *fixed20x20* which defines a device of 20 blocks of width and height and *fixed30x30* which defines a device with 30 blocks of width and height. Both layouts follow the same tiling pattern as the *<auto_layout>* already defined in the architecture file. The relation of which circuit was implemented in which layout is shown in Tables 5 and 6 with the block utilization of each circuit in the device. For the 20x20 grid a channel width of 60 is chosen and for the 30x30 grid a channel width of 80 is used. These values were obtained experimentally by Freiberger to be the minimum channel width that allowed routing the largest design on the layout.

From both tables, it is seen that the total FPGA utilization is not representative of the design load on the FPGA. Three clear examples of designs with high device load are the "mkPktMerge.v" on the 30x30 grid and the "diffeq2.blif" and "elliptic.v" designs on the 20x20 grid. The first and second designs have less than one-fifth of total FPGA utilization but use a considerate amount of complex blocks. The utilization of memory blocks from "mkPktMerge.v" is 94% in the 30x30 grid and the utilization of multipliers

20x20 grid, 60 channel width

Design	FPGA Utilization				
	Global	I/O	Logic	Multipliers	Memory
elliptic.blif	0.65	0.43	0.98	0	0
des.blif	0.43	0.87	0.46	0	0
bigkey.blif	0.42	0.74	0.50	0	0
apex2.blif	0.37	0.07	0.60	0	0
dsip.blif	0.36	0.74	0.39	0	0
seq.blif	0.35	0.13	0.56	0	0
s298.blif	0.33	0.02	0.56	0	0
alu4.blif	0.28	0.04	0.47	0	0
diffeq.blif	0.27	0.18	0.41	0	0
apex4.blif	0.26	0.05	0.42	0	0
ch_intrinsic.v	0.25	0.40	0.28	0.11	0
misex3.blif	0.25	0.05	0.42	0	0
diffeq1.v	0.22	0.45	0.15	0.62	0
ex5p.blif	0.22	0.12	0.33	0	0
tseng.blif	0.21	0.30	0.27	0	0
diffeq2.v	0.19	0.28	0.12	0.88	0
stereovision3.v	0.05	0.07	0.06	0	0

Table 5: FPGA utilization for designs evaluated on a 20x20 grid with a channel width of 60. Source: author.

by "diffeq2.v" reaches 88% of the blocks in the 20x20 grid. By looking only at the total FPGA utilization one may think these designs can withstand high defect rates in routing elements due to the low utilization, which does not hold.

The best example of the importance of looking into the utilization of separate functional blocks is given by "elliptic.blif" with 65% global device utilization. By looking into individual block types the design has 98% of logic block utilization and it was shown experimentally during the simulations that the chosen channel width is heavily dependent on a good placement solution. VPR uses simulated annealing to find a placement solution. The algorithm starts with a random initial placement and a temperature parameter. To find a solution, swaps are made between blocks, and the cost of the move is calculated to define the probability to accept it. If a move has a negative cost and lowers the current temperature, it is always accepted, otherwise the higher the cost, the lower the acceptance probability. If the move is accepted, the temperature is updated and another move is attempted until the temperature reaches a set minimum threshold. Every parameter of simulated annealing can be user-defined through the command line, but the default parameters in VPR were obtained experimentally and are usually better than any custom parameter provided [3]. The best-found alternative to ensure a viable placement and consequent routing from VPR was to use different "seeds" for the initial

30x30 grid, 80 channel width					
	FPGA Utilization				
Design	Global	I/O	Logic	Multipliers	Memory
or1200.v	0.41	0.87	0.43	0.05	0.12
mkSMAdapter4B.v	0.28	0.44	0.29	0	0.31
sha.v	0.24	0.08	0.35	0	0
raygentop.v	0.23	0.58	0.18	0.43	0
mkPktMerge.v	0.19	0.52	0.05	0	0.94
boundtop.v	0.13	0.34	0.14	0	0
ch_intrinsics.v	0.11	0.26	0.11	0	0.06
diffeq1.v	0.10	0.29	0.06	0.24	0
diffeq2.v	0.09	0.18	0.05	0.33	0
stereovision3.v	0.02	0.05	0.02	0	0

Table 6: FPGA utilization for designs evaluated on a 30x30 grid with a channel width of 80. Source: author.

placement.

From the previous insight, each combination of architecture and circuit is run multiple times with different random seeds to avoid local minima in the placement and routing algorithms when finding a valid solution. To implement such behavior in the simulation flow, the custom-defined VTR tasks are run with the script option `--seed X` appended to it, where X varies from 1 to 10. The `--seed` option is used by VPR to set a random seed responsible for defining the initial placement. An example of the simulation runs is shown in Listing 5.1

```
$> $VTR_ROOT/vtr_flow/scripts/run_vtr_task.py fault_sim_20x20 -j 8 -s --
    ↪ seed 1

$> $VTR_ROOT/vtr_flow/scripts/run_vtr_task.py fault_sim_20x20 -j 8 -s --
    ↪ seed 2
```

Listing 5.1: Example of running a VTR task providing a different seed at each task run.

5.1 Evaluation metrics

With the simulation environment set up, the evaluation metrics for the defect simulations are defined. Following Freiberger, the first metric analyzed is the number of designs successfully routed with a given error probability. Considering 0% defect probability all chosen designs are successfully routed in their respective architecture and device grid. All defect simulations are run with equal probabilities for error types SA0, SA1, and UD,

and consider the defect probabilities of any two memristors as independent. The second metric analyzed is the percentage of defective memory cells versus the percentage of defect routing edges in a given device grid. The third metric considered is the relative variation of the critical path delay between the error-free solution and the design implementation post-defect simulation. The critical path delay metric is considered a relative variation because the architecture proposed by this work is purely functional. No modeling was done regarding the area, delay, or power of the new routing element. This assumption also justifies the use of a 40 nm technology architecture while considering 22 nm node technology with 150 mV of gate-boosting in Chapter 4.

Although no circuit modeling was done for the proto-voter, its area and delay can still be evaluated. Regarding the transistor area of the proto-voter cell, the number and type of circuit elements used can be used to estimate the real gain against a 6T SRAM cell. The proposed cell has one transistor for each memory cell inside it, two driven by the control cell and two more to build the inverter, totaling six transistors plus the routing switch. As discussed in Chapter 2, the memristors do not share transistor fabric area, and even though the proto-voter cell has the same number of transistors as a standard SRAM cell all but one of them are NMOS transistors. An SRAM cell has usually two PMOS transistors for signal latching while the proto-voter cell only has one needed for the inverter. As electron mobility is lower in PMOS-type devices, they are usually larger than NMOS devices to drive the same current. Additionally, the NMOS transistors in the word line in SRAM cells are usually larger than NMOS in the latch structure so that they can overwrite the information latched during a write operation. In the proto-voter cell, all NMOS transistors are of the same size. From the assumptions just given, it is safe to say that the transistor area of the proto-voter cell can be smaller than the transistor area of a 6T-SRAM cell. Assuming half of the electron mobility in PMOS devices against NMOS devices, the proto-voter cell has at least 14,29% area gain versus 6T-SRAM.

Concerning the cell delay, the routing switch connected to a wiring segment is still an NMOS pass-transistor. As gate-boosting is considered to justify the cascade design of the proto-voter cell, it is also possible to assume the proposed architecture is faster than a device with the same size technology node using pass-transistors or transmission gates as switches and does not have gate-boosting available as shown by [10].

A final metric is used to verify the correctness of the simulation implementation. As previously defined, the defect probabilities of memristors are independent. From Table 1, the probability of a SA0 error in the 2T2R cell is given by:

$$P(SA0) \cdot P(FF) + P(SA0) \cdot P(SA1) + P(FF) \cdot P(SA1)$$

As both pull-up and pull-down memristor error probabilities are the same, the SA1 error probability is given by the same relation (in the 2T2R cell). Following this logic, the probabilistic models for every type of error in both cells are plotted against the obtained error probabilities in the defect simulation. This way if the simulation values are close to the mathematical ground truth it is safe to assume the simulations are indeed correct.

5.2 Results

The defect simulation is run for both the 2T2R cell and the proto-voter cell. The range of defect probability chosen was from 0% to 3% with multiple step sizes. Between 0% and 0.01% there is a 0.001% increment. Between 0.01% and 0.1% there is a 0.005% increment. From 0.1% to 0.25% there is a 0.05% increment and from 0.25% to 3% there is a 0.25% increment. Once more this probability is used for P_{equal}, which means for a considered probability of 3% there is 91% of probability of a memristor staying error-free and 9% probability of it presenting one of the 3 possible error types, each of them being equiprobable with $P(SA0) = P(SA1) = P(UD) = 3\%$.

Graphs 14, 15 and 16 represent the obtained error probabilities in the defect simulations against the probabilistic model assumed. As there is randomness involved in generating each memristor error, the values are not exact but follow the ground truth with great precision. An important insight from this metric is that for the probability range chosen, the 2T2R cell present similar probabilities for all error types. For greater error probabilities UD type errors actually grow faster as only one memristor needs to present this error for the complete cell to present an UD error as well. In contrast, the proto-voter cell presents almost ten times more SA0 errors but much less SA1 and UD errors. This behavior is explained by the attempt to turn defect cells into SA0 cells as it is the most permissive error type in a multiplexer stage. The expected probability for a proto-voter cell with 3% of memristor defect probability to present a SA1 error is approximately 0.31% against 5.55% in the 2T2R cell. For UD errors, the expected chance from the proto-voter cell is 1.05% against 6.09% in the 2T2R cell, again considering 3% probability of memristor defect. Although not shown, the chance of staying free of error is slightly reduced: 77.77% in proto-voter cells against 82.81% in 2T2R cells considering 3% defect chance.

Overall, given the same defect chance for both architectures, the proto-voter cell will

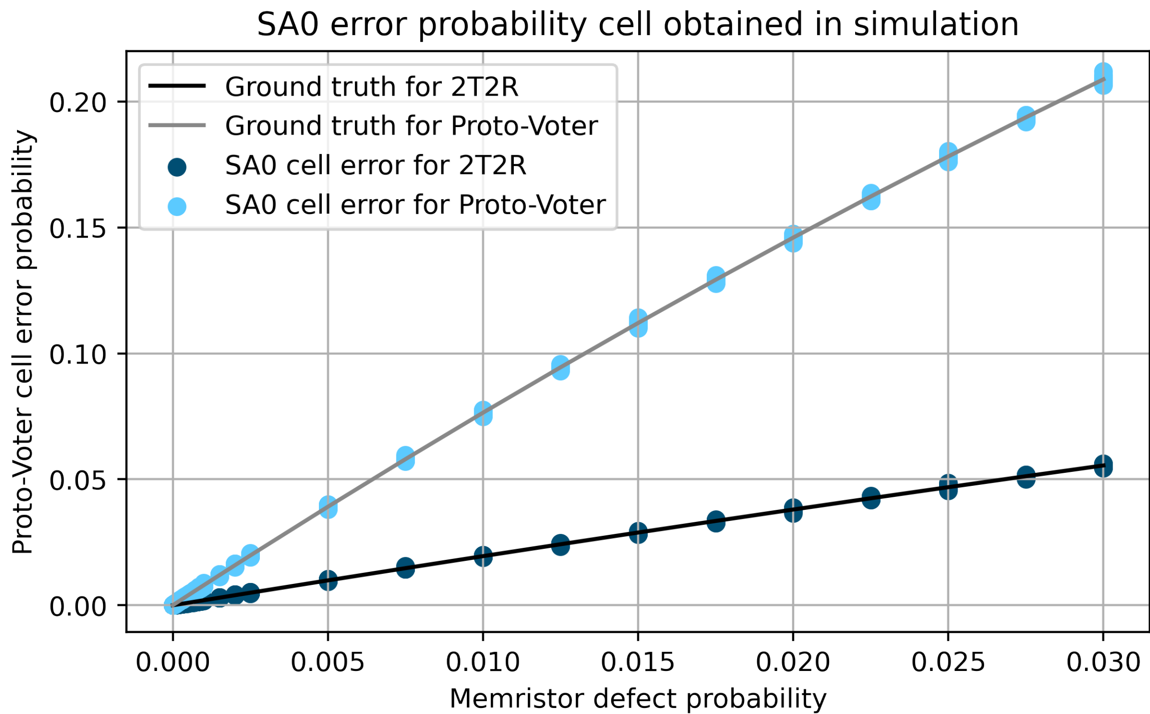


Figure 14: SA0 cell probabilities obtained for each cell versus mathematical ground truth. Source: author.

present fewer defect edges as shown in Graphs 17 and 18. Both graphs show the relative amount of defective memory cells and routing edges versus defect probability for both cell architectures. The 2T2R cell presents a linear increase in defect memory cells and a quadratic increase in defect edges as UD errors affect the whole multiplexer and in the studied defect probability range, it is as equiprobable as the other errors. The proto-voter cell presents a linear increase in both defect memory cells and routing edges, as SA0 errors are more common and their effect is restricted solely to a single routing edge. It is also important to note that the number of defective memory cells grows faster in the pro-voter cell: considering 3% of defect probability the proto-voter cell has 22.23% of defective memory cells versus 17.19% in 2T2R cells. Here the relevance of fewer SA1 and UD errors is highlighted.

Graphs 19 and 20 show the number of successfully implemented designs with a given defect probability. Designs implemented with the 2T2R cells are in dark blue bars and designs implemented with the proto-voter cell are in light blue. In the 20x20 grid 17 designs can be implemented free of errors and in the 30x30 grid 10 designs can be implemented free of error. The first important insight of both graphs is that for every scenario where defect probability is in place, the proto-voter cells enable at least two more designs to be successfully routed. The second insight is that there is some deviation where higher

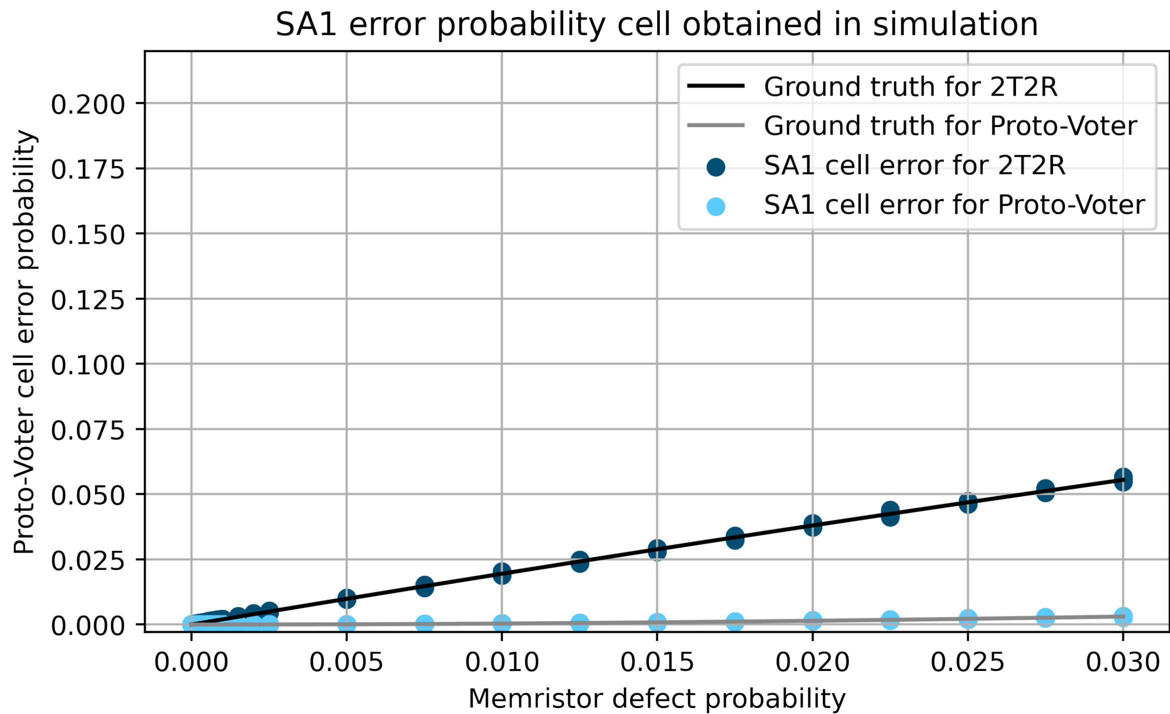


Figure 15: SA1 cell probabilities obtained for each cell versus mathematical ground truth. Source: author.

defect probabilities enable more designs to be implemented than lower ones. This may be due to the randomness of both the work done by VPR and the error generation from the defect simulation.

Going more in detail using 2T2R memory cells, as defect probability increases more routing resources are affected and unusable. The higher the device utilization from a single block type in an FPGA, the lower its flexibility during placement to avoid blocks near defect routing elements. In the 20x20 grid, the first designs to fail with defect probabilities of only 0.002% are "elliptic.blif" and "diffeq2.v" due to their high utilization of logic blocks and multipliers. It is also possible to argue that the use of memory blocks is more critical to routing as the number of implemented designs falls more rapidly in the 30x30 scenario, where designs utilize at most 43% of logic blocks and multipliers. With defect probabilities greater than 0.2% no devices can be implemented against 0.750% in the 20x20 grid scenario.

In the case where proto-voter cells are used, the robustness is clearly shown. Apart from a loss of one design at 0.065% defect probability, all of the 30x30 designs are fully implemented up to 0.09% defect probability. At the same defect rate, the 2T2R cell can only route 2 out of 10 designs. In the 20x20 grid case up to 16 out of 17 designs can be routed with defect probability up to 0.1%, where only 6 designs are still routed using

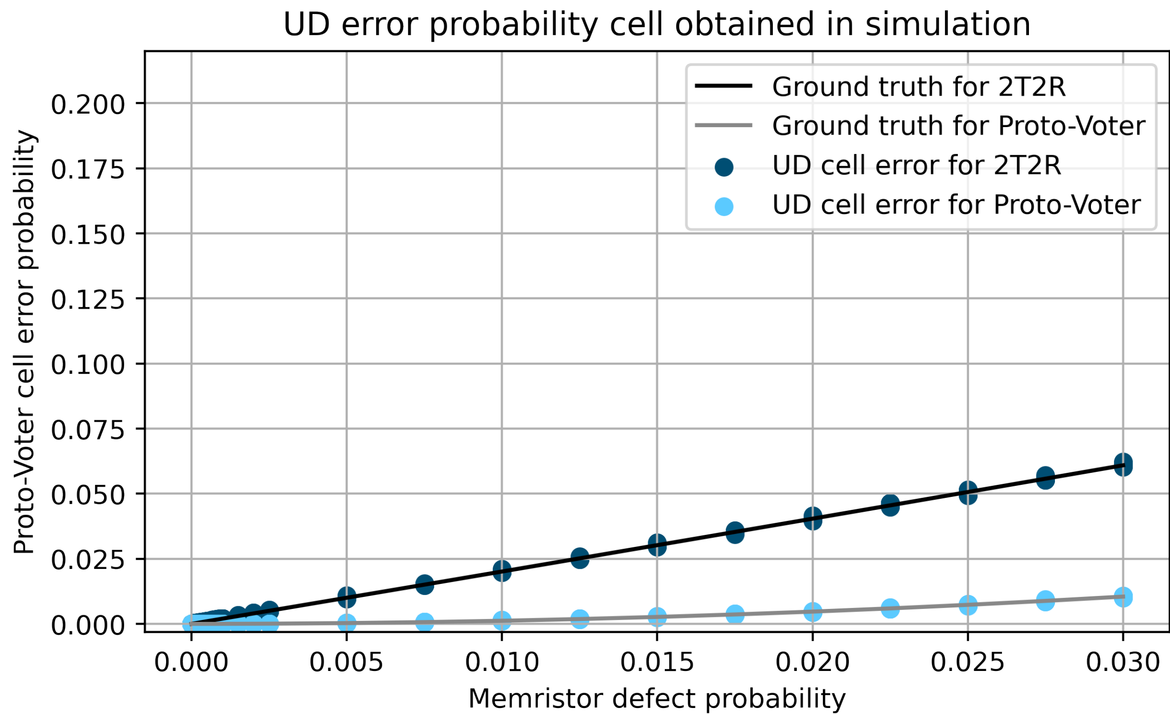


Figure 16: UD cell probabilities obtained for each cell versus mathematical ground truth. Source: author.

2T2R cells. Considering at least one design routed, the proto-voter cells can withstand up to 2.25% defect probability for each error type or 6.75% defect probability in total, representing 3 times more robustness against 2T2R cells which withstand up to 0.75% defect probability (2.25% in total). In the 30x30 grid case, the robustness is even greater: the proto-voter cell can withstand up to 2% defect probability until not being able to route any design against 0.2% maximum defect tolerance in 2T2R.

Regarding critical path delay variation, Graph 21 illustrates the results from all device grids and memory cells used. The worst-case scenario in all combinations is an approximate increase of 40% in the critical path delay. Common case scenario are variations between 10% more or less the original critical path delay. The improvements can be argued as a better effort from VTR to find a viable place and route solution as the original is not valid anymore due to defect routing resources. If a better solution is not available or if a local minimum is found, then there is the case of an increase in the critical path delay.

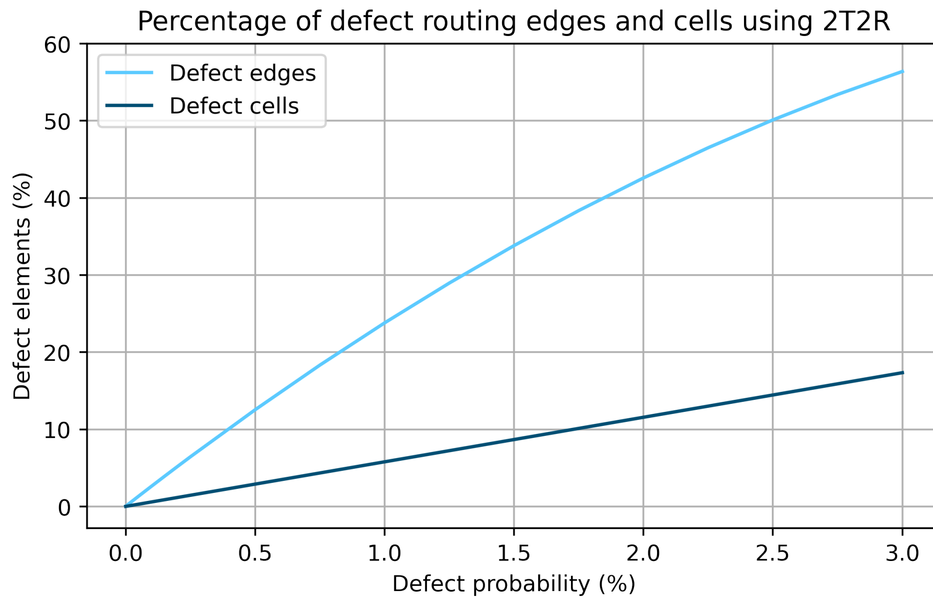


Figure 17: Percentage of defect routing edges and memory cells using 2T2R. The amount of defect edges grows quadratically due to the presence of UD errors or multiple SA1 errors in the same multiplexer. Source: author.

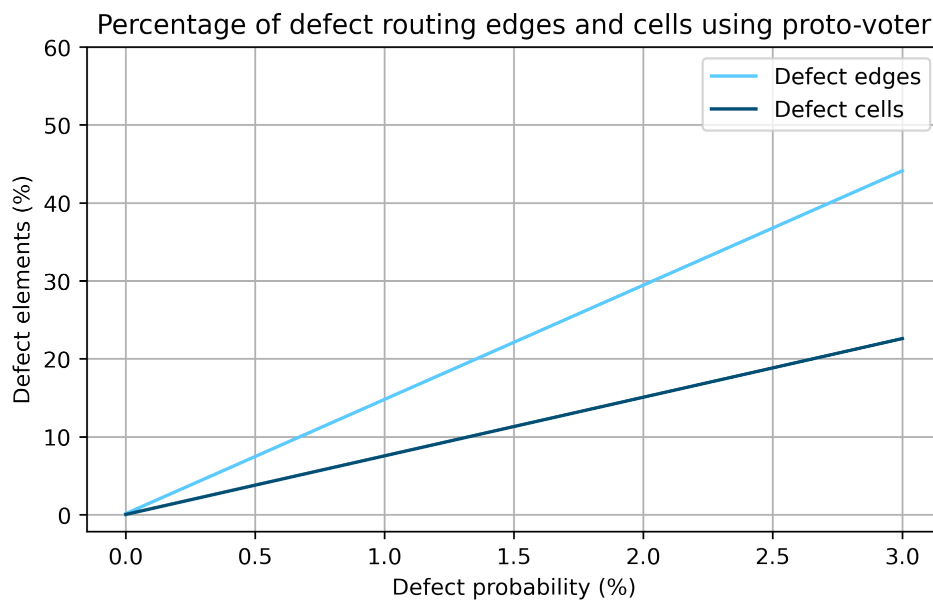


Figure 18: Percentage of defect routing edges and memory cells using proto-voter cell. The amount of defect edges grows linearly due to the mitigation of SA1 and UD error types. Source: author.

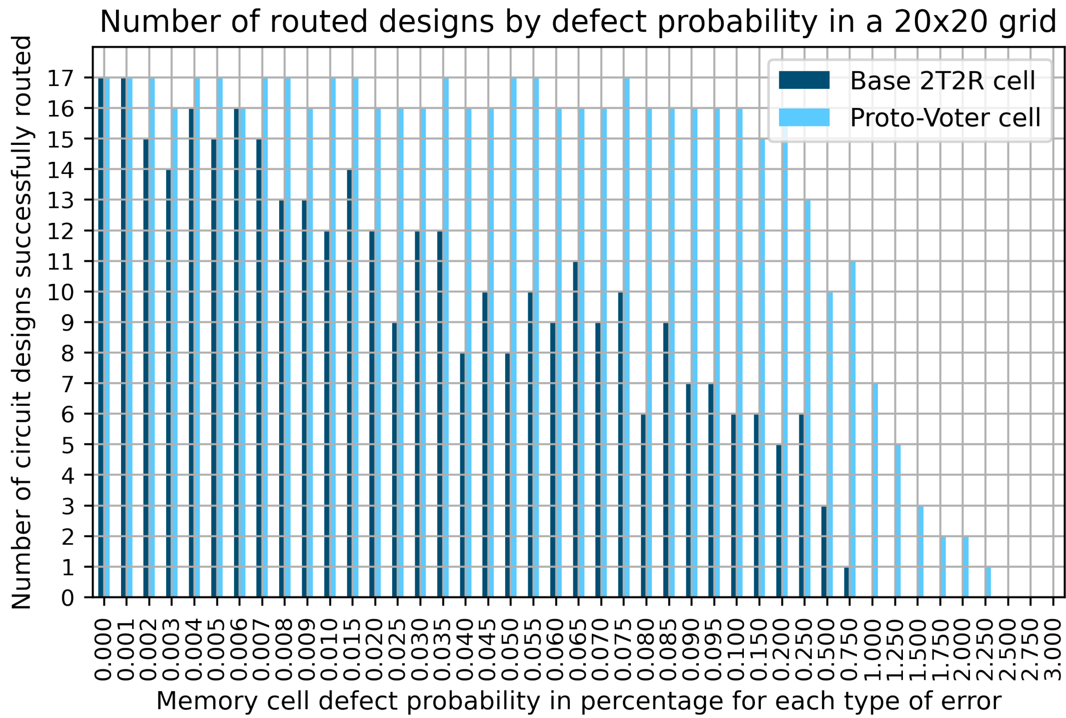


Figure 19: Number of designs routed in a 20x20 grid for each memory cell type with increasing defect probabilities. Source: author.

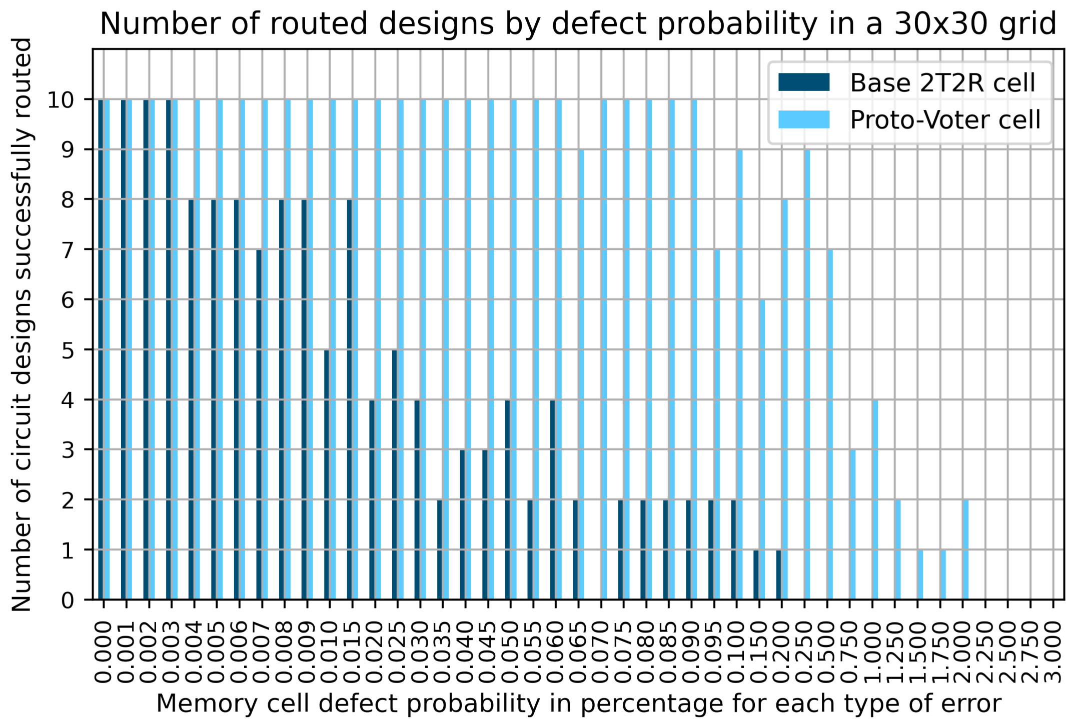
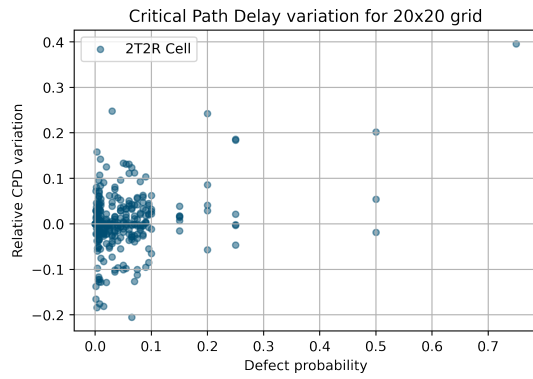
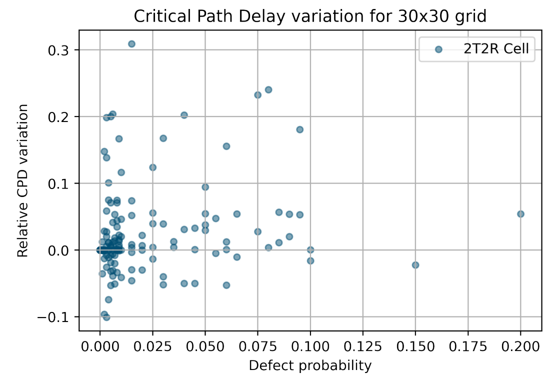


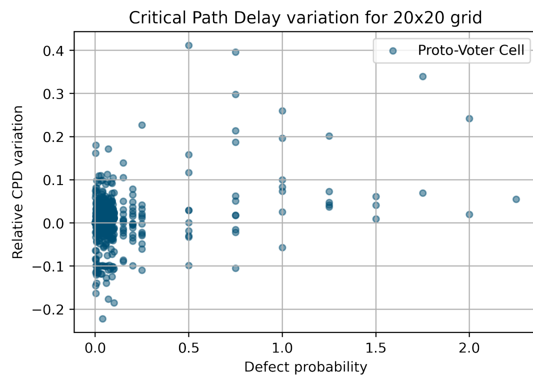
Figure 20: Number of designs routed in a 30x30 grid for each memory cell type with increasing defect probabilities. Source: author.



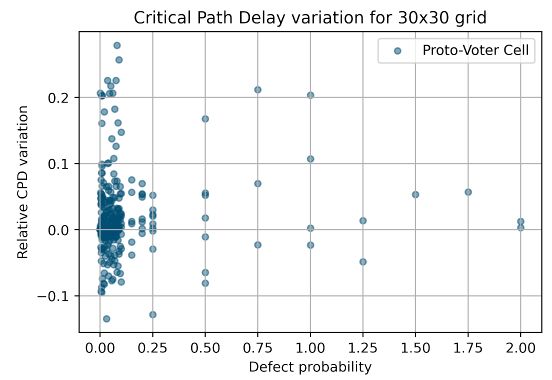
(a) CPD variation for 2T2R cell and 20x20 grid



(b) CPD variation for 2T2R cell and 30x30 grid



(c) CPD variation for proto-voter cell and 20x20 grid



(d) CPD variation for proto-voter cell and 30x30 grid

Figure 21: Critical path delay variation before and after defect simulation. Source: author.

6 CONCLUSION

In the scope of this work a novel architecture of routing elements based on non-volatile resistive memory is presented. The architecture has the same number of transistors as currently used volatile cells but may be smaller due to having one less PMOS in favor of an NMOS device. Regarding the delay, the architecture makes use of gate-boosting while using the same type of routing switch as current commercial technologies, also being supposedly faster than current devices that do not make use of gate-boosting. The robustness in the design comes from the attempt to forward a more permissive error type to the routing switch so that other switches in the routing multiplexer can still be used.

To help evaluate the proposed architecture performance, a Python module to simulate memristor defects in 2T2R memory cells or in the proposed architecture is developed. The module supports a standalone simulation of an arbitrary number of disconnected routing multiplexers, as well as a simulation based on a given routing resource graph file to better evaluate error effects on a fully connected network of routing resources. The module is optimized to load and update large RR graph files, has a complete test suite, and is openly available on GitHub.

An open-source collection of FPGA CAD tools called VTR is also used. VTR provides ready-to-use FPGA architecture files and circuit designs, as well as a complete toolflow responsible for implementing a chosen design in the desired architecture. The developed Python module is integrated into VTR together with customizable parameters over the command line. To fully evaluate the proposed routing element architecture, the defect simulation is run for multiple defect probabilities between 0% and 3% over designs from MCNC20 and the VTR benchmarks in a reasonable architecture for today's standards: "k6_frac_N10_mem32K_40nm.xml". The results showed that both the simulation is correctly implemented, supported by mathematical proof, and the proposed architecture can withstand up to 10 times more errors in comparison to the 2T2R memory cell.

A possible extension to this work is evaluating more complex designs and architectures. Possible options for such extension are the Koios benchmark [28] and 22 nm tech-

nology architectures with DSP blocks and gate-boosting support [29]. This combination would allow evaluation of the design for deep-learning circuit modules, which has been increasingly relevant in literature.

Another possible extension is to model area, power and delay of the proto-voter cell using SPICE device models for better accuracy in improvements. The greatest challenge it may prove is to find an available memristor model, as brief research by the author did not prove successful.

REFERENCES

[1] S. P. Young, "Integrated circuit having fast interconnect paths between memory elements and carry logic," 2005.

[2] M. J. Cannon, A. M. Keller, and M. J. Wirthlin, "Improving the effectiveness of tmr designs on fpgas with seu-aware incremental placement," *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 141–148, 2018.

[3] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. ElDafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "Vtr 8: High performance cad and customizable fpga architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, 2020.

[4] A. Boutros and V. Betz, "Fpga architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, pp. 4–29, 2021.

[5] "Ultrascale architecture and product data sheet: Overview," 2022.

[6] V. Betz, J. Rose, and A. Marquardt, "Architecture and cad for deep-submicron fpgas," in *The Springer International Series in Engineering and Computer Science*, 1999.

[7] E. Ahmed and J. S. Rose, "The effect of lut and cluster size on deep-submicron fpga performance and density," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 288–298, 2004.

[8] D. M. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. R. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose, "The stratix ii logic and routing architecture," in *FPGA '05*, 2005.

[9] C. Chen, R. Parsa, N. Patil, S. Chong, K. Akarvardar, J. Provine, D. M. Lewis, J. Watt, R. T. Howe, H. S. P. Wong, and S. Mitra, "Efficient fpgas using nanoelectromechanical relays," in *FPGA '10*, 2010.

[10] C. Chiasson and V. Betz, "Should fpgas abandon the pass-gate?," *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1–8, 2013.

[11] J. Rose and S. D. Brown, "Flexibility of interconnection structures for field-programmable gate arrays," *IEEE Journal of Solid-state Circuits*, vol. 26, pp. 277–282, 1991.

[12] L. O. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, pp. 507–519, 1971.

[13] L. O. Chua and S. M. Kang, "Memristive devices and systems," *Proceedings of the IEEE*, vol. 64, pp. 209–223, 1976.

[14] S. Yin, J. sun Seo, Y. Kim, X. Han, H. J. Barnaby, S. Yu, Y. Luo, W. He, X. Sun, and J.-J. Kim, "Monolithically integrated rram- and cmos-based in-memory computing optimizations for efficient deep learning," *IEEE Micro*, vol. 39, pp. 54–63, 2019.

[15] "Static random-access memory," 2022. Accessed: 2022-09-23.

[16] S. Tanachutiwat, M. Liu, and W. Wang, "Fpga based on integration of cmos and rram," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, pp. 2023–2032, 2011.

[17] L. Freiberger, "Evaluation des fpga-routings mit fehlerbehafteten elementen," 2022.

[18] J. Cong and B. Xiao, "mrfpga: A novel fpga architecture with memristor-based reconfiguration," *2011 IEEE/ACM International Symposium on Nanoscale Architectures*, pp. 1–8, 2011.

[19] B. Xiao, "Fpga-rr: A novel fpga architecture with rram-based reconfigurable interconnects," 2012.

[20] J. Cong and B. Xiao, "Fpga-rpi: A novel fpga architecture with rram-based programmable interconnects," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 864–877, 2014.

[21] "Logic compatible and low cost nvm solutions," 2022.

[22] A. J. Yu and G. G. F. Lemieux, "Defect-tolerant fpga switch block and connection block with fine-grain redundancy for yield enhancement," *International Conference on Field Programmable Logic and Applications, 2005.*, pp. 255–262, 2005.

[23] J. Rose, Z. Vranesic, and S. J. E. Wilton, "Architectures and algorithms for field-programmable gate arrays with embedded memory," 1997.

[24] Y.-W. Chang, D. F. Wong, and C.-K. Wong, "Universal switch modules for fpga design," *ACM Trans. Design Autom. Electr. Syst.*, vol. 1, pp. 80–101, 1996.

[25] I. Kuon and J. Rose, "Area and delay trade-offs in the circuit and architecture design of fpgas," in *FPGA '08*, 2008.

[26] C. Chiasson and V. Betz, "Coffe: Fully-automated transistor sizing for fpgas," *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 34–41, 2013.

[27] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," 1991.

[28] A. Arora, A. Boutros, D. Rauch, A. Rajen, A. Borda, S. A. Damghani, S. Mehta, S. Kate, P. Patel, K. B. Kent, V. Betz, and L. K. John, "Koios: A deep learning benchmark suite for fpga architecture and cad research," *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 355–362, 2021.

[29] S. Yazdanshenas and V. Betz, "Coffe 2: Automatic modelling and optimization of complex and heterogeneous fpga architectures," vol. 12, no. 1, 2019.