Fault Tolerance in Hardware Spiking Neural Networks

Sarah Ali El Sayed

▶ To cite this version:

Sarah Ali El Sayed. Fault Tolerance in Hardware Spiking Neural Networks. Micro and nanotechnologies/Microelectronics. Sorbonne Université, 2021. English. NNT : . tel-03681910v1

HAL Id: tel-03681910
https://hal.science/tel-03681910v1

Submitted on 6 Jan 2022 (v1), last revised 30 May 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT
DE SORBONNE UNIVERSITÉ

FAULT TOLERANCE IN HARDWARE SPIKING NEURAL NETWORKS

présentée par

SARAH ALI ELSAYED

École Doctorale Informatique, Télécommunications et Électronique

réalisée au

Laboratoire d'Informatique Paris 6

soutenue le 28 Octobre 2021

devant le jury composé de :

M.	Olivier SENTIEYS	Prof., INRIA-IRISA, Rennes	Président
M.	Alberto BOSIO	Prof., INL, Lyon	Rapporteur
M.	Ernesto SANCHEZ	Assoc. Prof., Politecnico di Torino, Italy	Rapporteur
M.	Ihsen ALOUANI	MCF, IEMN, Lille	Examinateur
Mme.	Ioana VATAJELU	CR CNRS, TIMA, Grenoble	Examinatrice
M.	Luis CAMUÑAS MESA	Research Assoc., IMSE-CNM, Spain	Invité
M.	Haralampos STRATIGOPOULOS	DR CNRS, LIP6, Paris	Directeur de Thèse

Sarah ALI ELSAYED: *Fault Tolerance in Hardware Spiking Neural Networks,*
© Juillet 2021

To my father
who always believed in me,
my mother
who did everything to make sure I succeed in life,
and my sister
who has been with me through it all...

ABSTRACT

Artificial Intelligence (AI) and machine learning algorithms are taking up the lion's share of the technology market nowadays, and hardware AI accelerators are foreseen to play an increasing role in numerous applications, many of which are mission-critical and safety-critical. This requires assessing their reliability and developing cost-effective fault tolerance techniques; an issue that remains largely unexplored for neuromorphic chips and Spiking Neural Networks (SNNs).

A tacit assumption is often made that reliability and error-resiliency in Artificial Neural Networks (ANNs) are inherently achieved thanks to the high parallelism, structural redundancy, and the resemblance to biological neural networks. However, prior work in the literature unraveled the falsity of this assumption and exposed the vulnerability of ANNs to faults, proving that without adequate designs and proper protection measures, ANNs remain at risk of performance failure due to faults.

In this thesis, we tackle the subject of testing and fault tolerance in hardware SNNs. We start by addressing the issue of post-manufacturing test and behavior-oriented self-test of hardware neural networks and propose a self-testable version of an analog biologically plausible spiking neuron circuit at transistor-level. Then we move to defect-oriented testing, where transistor-level fault simulations on a large scale are usually prohibitive in time and cost. Therefore, we follow a bottom-up approach starting from transistor-level fault injection into a single analog Integrate-and-Fire (I&F) neuron and propose a behavioural-level fault model that is specific to SNNs, yet agnostic to the circuit design and architecture. We demonstrate the acceleration offered by this fault model by performing fault injection experiments that pinpoint the critical fault types and locations on two SNNs that we designed for two neuromorphic datasets, namely the N-MNIST and IBM's DVS-gesture datasets. By leveraging observations from these experiments, we propose a neuron fault tolerance strategy for SNNs, optimized for low area and power overhead.

Finally, we present a hardware-in-the-loop case-study which would be used as a platform for demonstrating fault-injection experiments and fault-tolerance capabilities.

RÉSUMÉ

L'intelligence artificielle (IA) et les algorithmes d'apprentissage automatique sont au sommet du marché de la technologie de nos jours. Dans ce contexte, les accélérateurs matériels d'IA devraient jouer un rôle de plus en plus primordial pour de nombreuses applications, surtout ceux ayant une mission critique et un haut niveau de sécurité. Cela nécessite d'évaluer leur fiabilité et de développer des techniques peu coûteuses de tolérance aux fautes ; un problème qui reste largement inexploré pour les puces neuromorphiques et les réseaux de neurones impulsionnels (Spiking Neural Networks, SNNs).

Il est souvent présumé que la fiabilité et la résilience aux erreurs dans les Réseaux de Neurones Artificiels (ANN) sont intrinsèquement obtenues grâce au parallélisme, à la redondance structurelle et à la ressemblance avec les réseaux de neurones biologiques. Cependant, des travaux antérieurs dans la littérature ont révélé le non-fondement de cette hypothèse et ont exposé la vulnérabilité des ANN aux fautes, prouvant que sans des conceptions adéquates et des mesures de protection appropriées, les ANNs restent à risque de manque de performance en raison des fautes.

Dans cette thèse, nous abordons le sujet de test et de la tolérance aux fautes pour les SNNs matériels. Nous abordons tout d'abord la question du test de post-fabrication des réseaux de neurones matériels et de leur autotest orienté sur le comportement. À l'issue de cette phase, nous proposons une version auto-testable d'un d'un neurone impulsionnel biologiquement plausible au niveau transistor. Ensuite, nous abordons le sujet de simulations de fautes à grande échelle au niveau des transistors qui est généralement contraignant en temps. Par conséquent, nous suivons une approche ascendante partant de l'injection de fautes au niveau transistor dans un seul neurone analogique du type Integrate-and-Fire (I&F) afin de proposer un modèle de fautes au niveau comportemental. Ce modèle est spécifique aux SNN et indépendant de la conception et l'architecture du circuit. Nous démontrons l'accélération offerte par ce modèle de fautes en effectuant des expériences d'injection de fautes. Ces expériences identifient les types et les emplacements de fautes critiques sur deux SNN que nous avons conçus pour la classification des basses de données N-MNIST et DVS-gesture d'IBM. En exploitant les observations de ces expériences, nous proposons une stratégie de tolérance aux fautes des neurones pour les SNNs qui a été optimisée afin de minimiser les surcoûts en surface et puissance du circuit.

Enfin, nous présentons une étude de cas "matériel dans la boucle" qui serait utilisée comme plateforme pour démontrer les expériences d'injection de fautes et les capacités de tolérance aux fautes.

PUBLICATIONS

As a result of this thesis the following publications have appeared:

[1] S. A. El-Sayed, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Self-testing analog spiking neuron circuit," in *International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design*, Lausanne, Switzerland, 2019.

[2] S. A. El-Sayed, T. Spyrou, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Spiking neuron hardware-level fault modeling," in *IEEE International Symposium on On-Line Testing and Robust System Design*, Naples (virtual), Italy, 2020.

[3] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Neuron Fault Tolerance in Spiking Neural Networks," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble (virtual), France, 2021.

ACKNOWLEDGMENTS

"An honorable person won't tell people how great they are,
people will experience their greatness just by being with them."

– Imam Al-Ghazali

Pursuing a PhD is a life-changing decision that should not be taken lightly. This is a lesson I have learned the hard way. As I look back over the last four years, I cannot help but feel thankful for the life-changing experience, grateful for all the people that have shared the journey with me, and relieved that I am finally done!

Perhaps the thing I am grateful for the most is meeting **Haralampos Startigopoulos** and having him as my supervisor. I will quote my colleague **Julian** here: there are only a few PhD students who like their supervisors, but in our case, it was much more than just a student-professor relationship. Besides being a very professional, attentive, and involved mentor to us all, you became a friend with whom we enjoy chatting and spending time and a supportive confidant who is always willing to solve whatever problems we face, administrative or personal, just to make sure that we are comfortable enough to focus on our work. Thank you, **Haralampe**[1], for your patience, persistence, and support, and for showing me how a good academic should be. Thank you for constantly boosting my confidence and pushing me to do my best and for teaching me to always have the bigger picture in mind. I feel privileged to have had the chance to work with you and learn from you. I could not have asked for a better supervisor. And I will be waiting for you and your lovely family in Egypt someday.

A very special thank you is owed to **Alhassan Hadia**, the one who got me in contact with **Haralampos** in the first place and one of the main reasons I am here today. I owe a huge chunk of my success to him and his support over the years. And of course, **Hassan Abou Shady**, who was kind enough to recommend me and support my application to come to LIP6 and start my PhD journey.

An important factor in the advancement of this work is owed to **Luis Camuñas-Mesa** from the Instituto de Microelectrónica de Sevilla, Spain. I

1 please note how I used the proper Greek way to address you!

am very grateful to him for giving us many of his team's IPs to experiment with and for always being available and patient enough to help us and answer all our questions.

People always say you do not make friends in the workplace, but that was definitely not true here at office 415. From the very first day, the overall atmosphere and the people were energetic and friendly. Our office was always full of conversations, jokes, food, coffee, and music. Over the years we have come to know each other pretty well and we exchanged cultures, food habits, and more desserts than I can count!

I would, however, have to start with the latest person to join the office. My one-day-apart twin (albeit a few years younger) and the breath of fresh air that my PhD so desperately needed; **Theofilos Spyrou**. An amazing friend with whom I enjoyed many conversations, debates, and lessons of life, language, and history. I am very thankful for your mix of enthusiasm and optimism without which I could not have made it through the past couple of years. I hope you know how much I appreciate everything you have done to help me, little brother. And now that I have come to realize that the Greek language and mythology are probably the origin of everything, Greece is my favorite country outside of my beloved home, Egypt.

It was not always pleasant to be the only girl in the office, but fortunately, I was blessed with some amazing friends who made it very easy for me with their comprehension and respect. **Julian**, who was very enthusiastic about me coming even before I arrived at the lab, and **Antonis**, whose first question to me when we met was *"how old are you?"*! And of course, **Gabriel**; the mature one. Thank you for sharing the journey with me from the very beginning. You guys are the best.

I am also deeply grateful for getting to know many other amazing colleagues at LIP6: **Ilias**, **Alan**, **Ning**, **Doaa**, and **Islam**. I am especially grateful to **Tamer Badran**, who was a huge help to me from day one in trying to figure out my way around the Préfecture and how exactly things work in France. I would like as well to thank **Engin Afacan** and **Spyros Raptis** for sharing the last year of my PhD journey and collaborating with me and **Theofilos** to try and get something meaningful and worthy out of this project.

As for life outside the lab, the was a story with many chapters. I have been so lucky to make many amazing friends who shaped my experience and eventually became like a second family to me.

The very first friend I made in France was **Joana Oliveira**, whom I met trying to learn how to say two sentences in French. I am so happy you took the initiative and asked me to do something that weekend in 2018.

We ended up seeing all of Paris together and spending so many days exploring Disneyland, "the happiest place on earth"! You made Paris a less lonely place for me, and for that I will be eternally grateful.

There is an Arabic proverb that goes: (a coincidence is sometimes better than a thousand appointments), which loses a lot of meaning in translation I'm afraid, but it is a perfect explanation of the wonderful day I met **Amel Raboudi**. Your decision to come and talk to me during that apéro three years ago literally changed my life. You are a wonderful and understanding friend and one of the kindest souls I will ever meet. I want to thank you even more for introducing me to **Sohaila** who became a little sister to me. Her and **Abdallah**'s home became my "little Egypt" here in Paris, full of food, movies, and all sorts of Egyptian traditions. Thank you guys for opening your homes to me and allowing me to share various aspects of your lives. And thank you for having my back and creating a sanctuary that I could run to whenever I needed to feel safe.

Nonetheless, the lion share of my gratitude belongs to the most important people in my life, my family. **Prof. Nagiba Shoker**, my mum who is a university professor herself, has been my biggest motivation for all I have accomplished in my life to this day. Thank you for supporting me and relentlessly pushing me to make something of myself. I am what I am today because of you. My sister **Wesam** who is my best friend, my partner in crime, and my biggest fan. The only one I can totally be myself with and feel absolutely safe. Thank you for putting up with me and for all that you have sacrificed for me. You are, and always have been, my backbone in this life and I cannot even begin to explain how much I appreciate your existence and support.

And last but not least, my new husband, **Tamer**. You have always been someone I could count on, even back when we were just friends from work. Thank you for your incredible patience and emotional support throughout this journey. And now, we can start our own story in peace.

Finally, I would like to thank my dad **Prof. Ali ElSayed** who, although no longer with us, continues to inspire me to follow in his footsteps. Thank you for being my role model. I hope you are proud of me...

Sarah

CONTENTS

1 INTRODUCTION 1
- 1.1 Preface . . . 1
  - 1.1.1 AI Hardware Accelerators . . . 2
  - 1.1.2 Neuromorphic Computing . . . 3
  - 1.1.3 Is AI Hardware Fault-Tolerant? . . . 4
- 1.2 Motivation . . . 5
- 1.3 Methodology . . . 6
  - 1.3.1 Fault Injection . . . 6
  - 1.3.2 Fault Modeling . . . 7
  - 1.3.3 Fault Tolerance . . . 7
- 1.4 Thesis Structure . . . 8

2 SPIKING NEURAL NETWORKS & THEIR FAULT TOLERANCE: A LITERATURE REVIEW 9
- 2.1 Spiking Neural Networks . . . 9
  - 2.1.1 The Spiking Neuron . . . 10
    - 2.1.1.1 The Spike . . . 10
    - 2.1.1.2 The Membrane Potential . . . 10
    - 2.1.1.3 Spike Generation . . . 11
  - 2.1.2 Neural Coding Schemes . . . 12
- 2.2 Spiking Neural Networks in Hardware . . . 13
  - 2.2.1 Neuron Models . . . 13
  - 2.2.2 Address Event Representation . . . 14
- 2.3 Neuromorphic Technology Prospects and Challenges . . . 15
- 2.4 State-of-the-Art in Testing and Fault Tolerance in Hardware Neural Networks . . . 18
  - 2.4.1 Fault, Error, and Failure . . . 18
  - 2.4.2 Fault Injection in the Literature . . . 19
  - 2.4.3 Fault Tolerance in the Literature . . . 23

3 A SELF-TESTING ANALOG SPIKING-NEURON CIRCUIT 27
- 3.1 A Biological Perspective . . . 27
- 3.2 The Biologically-Inspired Neuron Circuit . . . 29
  - 3.2.1 The Mathematical Model . . . 29
  - 3.2.2 The Neuron Circuit . . . 29
- 3.3 The Built-In Self-Test . . . 33
  - 3.3.1 BIST Architecture . . . 33
  - 3.3.2 Expected BIST Response . . . 35
  - 3.3.3 BIST Verification . . . 35

3.4 Results and Discussion 36
4 HARDWARE-LEVEL FAULT MODELING 39
4.1 Fault Simulation Framework 39
4.2 The Spiking Neuron 40
4.2.1 Behavioral Model 40
4.2.2 Transistor-Level Design 41
4.3 Spiking Neuron Faulty Behaviors 42
4.3.1 Catastrophic Faults 42
4.3.2 Parametric Faults 46
4.4 Behavioral-level Fault Model 48
5 FAULT INJECTION AND RESILIENCY ANALYSIS IN SPIKING NEURAL NETWORKS 49
5.1 Fault Models 49
5.1.1 Neuron Fault Models 50
5.1.2 Synapse Fault Models 51
5.2 Case Studies 51
5.2.1 The Spike Response Model 52
5.2.2 Case Study (1): The N-MNIST SNN 53
5.2.2.1 The N-MNIST Dataset 53
5.2.2.2 The N-MNIST SNN Architecture 53
5.2.3 Case Study (2): The DVS-gesture SNN 55
5.2.3.1 The DVS-gesture Dataset 55
5.2.3.2 The DVS-gesture SNN Architecture 56
5.3 Fault Modeling & Injection Methodology 57
5.4 Fault Injection Experiments & Results: (1) The N-MNIST SNN 59
5.4.1 Neuron Faults 59
5.4.1.1 Dead Neuron Faults 59
5.4.1.2 Saturated Neuron Faults 61
5.4.1.3 Parametric Faults 61
5.4.2 Synapse Faults 65
5.4.2.1 Dead Synapse Faults 66
5.4.2.2 Saturated Synapse Faults 67
5.5 Fault Injection Experiments & Results: (2) The DVS-gesture SNN 69
5.5.1 Neuron Faults 69
5.5.1.1 Dead Neuron Faults 69
5.5.1.2 Saturated Neuron Faults 70
5.5.1.3 Parametric Faults 70
5.6 Discussion 71
6 NEURON FAULT TOLERANCE 73
6.1 Passive Fault Tolerance Strategy 73

6.1.1 Training with Dropout 74
6.1.2 SNN Tolerance to Multiple Faults 78
6.2 Active Fault Tolerance Strategy 80
6.2.1 Active Fault Tolerance in the Output Layer 80
6.2.2 Active Fault Tolerance in the Hidden Layers 80
6.2.2.1 Offline Self-Test 81
6.2.2.2 Online Self-Test 83
6.2.2.3 Recovery Mechanisms 84
7 a spiking neural network hardware implementation 87
7.1 The Convolutional Node 87
7.1.1 The Convolutional Unit 88
7.1.1.1 Unit Parameters 89
7.1.1.2 The Convolution Operation 91
7.1.1.3 Global Leakage 91
7.1.1.4 Rate Saturation Mechanism 91
7.1.1.5 Output Event Generation & Traffic Control 92
7.1.2 The Router 93
7.1.3 The Configuration Block 94
7.2 The Experiment 95
7.2.1 The Poker-Card Symbols Dataset 95
7.2.2 The Convolutional SNN 95
7.2.3 The Experimental Setup & Results 96
7.3 Putting the Hardware in the Loop 97
8 conclusions 101
8.1 Thesis Contributions 101
8.2 Future Perspectives 103

bibliography 105

LIST OF FIGURES

Figure 2.1 A spike 11
Figure 2.2 The dynamics of spike generation. 11
Figure 2.3 Cause-Effect relationship between fault, error and failure. 19
Figure 3.1 Intrinsic Firing Patterns of Cortical Neurons . . . 28
Figure 3.2 Transistor-Level Implementation of the Spiking Neuron Circuit 31
Figure 3.3 High-Level Model of the Spiking Neuron Circuit . 31
Figure 3.4 Output Spike Train Patterns of Neuron. 32
Figure 3.5 Approximate areas in the control voltages space V_c-V_d that produce the different firing patterns. The diamond points correspond to the nominal control voltages combinations used to produce each firing pattern. 33
Figure 3.6 BIST Architecture. 34
Figure 3.7 Neuron Output Response to BIST Stimuli. 35
Figure 3.8 Monte Carlo Analysis Showing the Neuron Output Response to BIST Stimuli for 5 Runs. 37
Figure 4.1 I&F Neuron 42
Figure 4.2 Examples of catastrophic faulty behaviors. 45
Figure 4.3 Histograms of timing variations. 47
Figure 5.1 Samples of N-MNIST saccades snapshots in 2-D format. Black regions indicate no events, red indicates ON events, and blue indicates OFF events. . 53
Figure 5.2 Architecture of the SNN for the N-MNIST dataset. 54
Figure 5.3 Learning Curve of the N-MNIST SNN. 54
Figure 5.4 **TOP:** a video frame of an individual performing a few gestures from the dataset, from left to right: right-hand wave, left-hand wave, arm roll and air drums. **Bottom:** A 2-D representation of the same video frame above but in spiking form [64]. 55
Figure 5.5 Architecture of the SNN for the DVS-gesture dataset. 56
Figure 5.6 Learning Curve of the DVS-gesture SNN. 56
Figure 5.7 Fault injection methodology. 59

Figure 5.8 Effect of dead and saturated neuron faults on the N-MNIST SNN classification accuracy in the last 3 layers. 60
Figure 5.9 Effect of dead and saturated neuron faults on the N-MNIST SNN classification accuracy. 61
Figure 5.10 Effect of integration faults on the last 3 layers of the N-MNIST SNN. 62
Figure 5.11 Effect of threshold perturbation faults on the last 3 layers of the N-MNIST SNN. 63
Figure 5.12 Effect of refractory period faults on the last 3 layers of the N-MNIST SNN. 64
Figure 5.13 Synaptic weights values between SC3 and SF4. . . 65
Figure 5.14 Synaptic weights values between SF4 and SF5. . . 65
Figure 5.15 Effects of a dead synapse fault on synaptic connections between (a) layers SC3-SF4 and (b) layers SF4-SF5. 66
Figure 5.16 Effects of a negatively-saturated synapse fault on synaptic connections between (a) layers SC3-SF4 and (b) layers SF4-SF5. 67
Figure 5.17 Effects of a positively-saturated synapse fault on synaptic connections between (a) layers SC3-SF4 and (b) layers SF4-SF5. 68
Figure 5.18 Effect of dead and saturated neuron faults on the DVS-gesture SNN classification accuracy. 69
Figure 5.19 Effect of integration faults on the last 2 layers of the DVS-gesture SNN. 71
Figure 6.1 Effect of neuron faults on classification accuracy with and without dropout for: (a) N-MNIST SNN, and (b) DVS-gesture SNN. 75
Figure 6.2 Effect of neuron timing variations for the N-MNIST SNN. 76
Figure 6.3 Effect of neuron timing variations for the DVS-gesture SNN. 77
Figure 6.4 Fault tolerance for multiple fault scenarios with regular standard training vs. training with dropout. 79
Figure 6.5 TMR at the output layer. 80
Figure 6.6 Offline self-test scheme. 82
Figure 6.7 Online self-test scheme. 83
Figure 6.8 I&F neuron design showing the recovery operation at neuron-level. 85
Figure 7.1 The convolutional Node. 88
Figure 7.2 The convolutional Unit. 88

Figure 7.3 A conceptual figure showing the process of reading input events from the FIFO by the *controller block*. 90
Figure 7.4 The change in the state of the neuron with the incoming spikes. The red lines represent the effect of the leakage mechanism, and T_R represents the refractory period that stops the neuron from spiking too soon. 92
Figure 7.5 Block diagram of the destination-driven router. 94
Figure 7.6 Generation of the poker card symbols dataset: (a) Picture taken with a frame-driven camera. (b) 2-D image obtained by collecting events in a 5ms window [128]. The red square represents the 32x32 window that shows the centered used for the symbols dataset. 95
Figure 7.7 Schematic block diagram of the convolutional SNN used for this experiment. 96
Figure 7.8 Schematic of the 2-D mesh implementation of the convolutional SNN on the FPGA. Each box represents a convolutional node in the network. The numbers indicate the address of the node in the mesh (y,x) and the colors represent the different layers of the network. 97
Figure 7.9 The experimental setup used for the convolutional SNN in [127]. 98
Figure 7.10 Hardware-in-the-Loop setup of the Xilinx FPGA board. 98

LIST OF TABLES

Table 4.1 Catastrophic faulty behaviors resulting from defect simulation. 43

ACRONYMS

ADC	Analog-to-Digital Converter
AER	Address Event Representation
AI	Artificial Intelligence
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
ATE	Automated Test Equipment
BER	Bit-Error Rate
BIST	Built-In Self-Test
CMOS	Complementary MOS
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DNN	Deep Neural Network
DPR	Dynamic Partial Reconfiguration
DVS	Dynamic Vision Sensor
ECC	Error Correction Code
FIFO	First-In-First-Out
FIT	Failure-in-Time
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HIL	Hardware-in-the-Loop
I&F	Integrate-and-Fire
IC	Integrated Circuit
IoT	Internet-of-Things
MAC	Multiply-and-Accumulate
MC	Monte Carlo
NN	Neural Network

NVM Non-Volatile Memory

OpAmp Operational Amplifier

PDK Process Design Kit

PLL Phase-Locked Loop

QNN Quantized Neural Network

RRAM Resistive Random-Access Memory

RTL Register-Transfer Level

SAR Successive-Approximation Register

SAT satisfiability

SEU Single Event Upset

SLAYER Spike LAYer Error Reassignment

SNN Spiking Neural Network

SPI Serial Peripheral Interface

SRAM Static Random Access Memory

SRM Spike-Response Model

STDP Spike Timing-Dependent Plasticity

STL Software Test Libraries

TMR Triple Modular Redundancy

TPU Tensor Processing Unit

VHDL Very High-Speed Integrated Circuit Hardware Description Language

VLSI Very Large Scale Integration

1 INTRODUCTION

"If the brain were so simple we could understand it,
we would be so simple that we couldn't."

— Emerson M. Pugh

1.1 PREFACE

Understanding how the human brain works and building a machine that can match its computing power and efficiency has been a major force driving computer science research for decades. Nowadays, computers have reached remarkable performance levels in routine information processing such as basic math, and they even sometimes surpass human capacity in terms of computational speed and memory storage. However, the human brain still has an undeniable edge that is yet to be matched by a computer in non-routine tasks such as reasoning, awareness, problem solving, and learning from experience.

The first approach was a top-down strategy that looked at solving cognitive tasks algorithmically, i.e., mimicking the function but not the structure, and the von Neumann architecture quickly became the standard platform for this solution. The other approach took inspiration from biology for a bottom-up strategy and tried to imitate the structure of the biological brain in the hope of creating a machine that can think.

The earliest definitions of Artificial Intelligence (AI) were depicted in the mid-20th century when *Alan Turing* proposed the stored-program concept [1], [2]. He described an abstract machine that consists of an infinite memory strip and a scanner that moves through the memory, reads its contents, and modifies it according to a program of instructions also stored in memory, with the implication that if the machine operates on, it would modify and improve its own program. He went on to describe an "intelligent machine" as one that can learn from experience, and that the possibility of letting the machine alter its own instructions is the way to provide this mechanism.

The Turing Machine was perhaps the earliest design of a distributed-processing system, and the inspiration from which Artificial Neural

Networks (ANNs) were eventually designed. However, his ideas could not be further exploited until the 1970s when Very Large Scale Integration (VLSI) technologies were developed. With the computing revolution that followed that era, ANNs began to regain attention.

1.1.1 *AI Hardware Accelerators*

Today, AI has invaded our everyday lives affecting the way we work and spend our free time. Deep learning algorithms [3], and machine learning in general, are increasingly dominating the computer industry with applications that perform fundamental tasks like speech recognition, computer vision, and natural language processing, all the way to robotics, Internet-of-Things (IoT), autonomous self-driving vehicles, smart healthcare, etc. In all these applications, AI algorithms rely on hardware cores for performance acceleration in one form or another. AI hardware accelerators originally made use of existing technologies and traditional processors, such as general-purpose Central Processing Units (CPUs) and Graphic Processing Units (GPUs) [4], and to this day, the actual processing typically runs on giant servers in the cloud using CPU or GPU clusters. However, this hardware is too large to fit inside a portable device and it needs far more power than a device battery can provide.

Take IoT edge devices as an example; edge computing technology aims at pushing the execution of AI algorithms away from the cloud side and closer to the user side, i.e., the edge. Achieving this goal will contribute many advantages to the existing paradigm, e.g., provide availability of the application even in the absence of an internet connection, avoid latency, save network bandwidth which is overly occupied with the moving of data back and forth between the user and the cloud, and offer a much needed privacy since the data would be handled locally.

Compared to the human brain that has a neuron density of around $10,000/mm^2$ and consumes approximately 10^{-11} Joules per spike [5] -which means it runs on about 20 Watts-, CPUs are estimated to be about 10^8 less efficient in terms of size and energy consumption. In addition, several AI applications require a real-time response, as is the case with autonomous vehicles; an application running in constantly changing conditions with data coming from cameras, radars, accelerometers, sensors, etc. In the presence of this huge amounts of data, performing inference or on-line learning with a Neural Network (NN) running in software on a CPU is not an option. All these restrictions, along with other more technical challenges, such as the von Neumann bottleneck, also known as the memory-wall problem, and the approaching end of Moore's law [6], have made it crucial to find alternative architectures. Even with AI

accelerators like GPUs and Field Programmable Gate Arrays (FPGAs) [7] which perform far better than CPUs for AI related tasks, over one magnitude of cost-energy-performance improvements can be obtained with Application Specific Integrated Circuits (ASICs) [8], [9].

1.1.2 *Neuromorphic Computing*

Neuromorphic computing, a term introduced by *Carver Mead* in the early 1990s [10], is the technology that uses special purpose VLSI silicon ANN implementations that resemble -or are inspired from- biology. Unlike the traditional von Neumann architecture that adopts a central processing system where a logic core performs computations on data fetched from memory, neuromorphic computing distributes the computational load and the memory among a multitude of elements.

Mainstream neural modeling nowadays is mostly digital and done at software-level, which is convenient since digital simulations offer precise and noise-free outputs. However, this paradigm ignores the fact that neural computing in the brain, which ANNs are essentially trying to emulate, is analog and noisy in nature. Biological NNs perform tremendous high-speed parallel computations in the presence of noise. They are highly adaptable and are even capable of exploiting noise in the environment to enhance computations [11], all the while consuming very little power and occupying a small space. In contrast, even the most powerful supercomputers cannot simulate neural connectivity in real time [12], [13].

Hence, it was only natural to look for something more suited to the objective in hand, and this was when neuromorphic technology started to appear in the spotlight. Silicon implementations provide a compromise between biological NNs and digital computers in terms of power and space, in addition to being far faster than biology [14]. While conventional ANNs fall under this description as well, the term "neuromorphic" usually refers to Spiking Neural Networks (SNNs).

Today, there exist many hardware neuromorphic chips either in digital, mixed analog-digital, or purely analog form. Some famous examples include: IBM's **TrueNorth** chip [15], Furber's **SpiNNaker** chip [16], MIT's **Eyeriss** [17], and Intel's **Loihi** chip [18]. These chips offer compact designs that incorporate large numbers of neurons and synapses with a very low power consumption and have been used to demonstrate various simple AI applications. However, their user base still consists mostly of universities and industrial research groups [19]. Before neuromorphic technology can move on to the actual industry field and take over the role dominated by mature conventional ANNs, many obstacles would have to be overcome.

For example, SNNs cannot be trained using standard learning techniques like the back propagation algorithm. They also need benchmark data sets in spiking form, which are not readily available yet. The specific advantages of SNNs, as well as the challenges they face, will be further discussed in Chapter 2.

1.1.3 *Is AI Hardware Fault-Tolerant?*

A major preoccupation nowadays is the trustworthiness of AI systems. This includes privacy, avoidance of unfair bias, resilience to adversarial attacks, hardware dependability, etc. AI applications to test-related tasks have been extensively studied over the past decades [20], [21], including on-chip NNs as a Built-In Self-Test (BIST) engine for mapping low-cost measurements to chip-level pass/fail 1-bit decisions [22]. However, the "inverse" problem of testing and reliability of AI hardware has been customarily overlooked.

Biological NNs have remarkable error resilience and fault-tolerance capabilities. Not only can the brain tolerate a finite number of faults in the neurons and synapses, but it is also capable of regenerating or rewiring network elements to make up for larger damage. Modeled after the immensely parallel architecture and operation principles of biological NNs, both conventional ANNs and spike-based SNNs are assumed to be inherently as fault-tolerant as their biological counterparts. Moreover, both ANNs and SNNs usually contain more computational units than the minimum requirements of a certain cognitive task, a property known as over-provisioning [23], which allows for a certain degree of robustness [24].

However, arguing that this assumption is true based only on architecture resemblance to biology or over-provisioning is somewhat imprudent. In fact, many recent experiments have shown that AI hardware accelerators are limited by the constraints and imperfections of the VLSI technologies and are vulnerable to hardware-level faults resulting from manufacturing defects, process variations, aging, and Single Event Upsets (SEUs).

With the foreseen industrialization and high-volume production of hardware NNss in the coming years, special attention must be paid to the fault-tolerance aspect of AI hardware. Given that several targeted applications are safety-critical and mission-critical for which AI hardware accelerators must meet the functional safety standards regulated by every application domain, e.g., ISO 26262 for automotive and IEC 61508 for industrial systems, testing strategies specific to hardware NNs remains a topic that is largely unexplored [25].

But why does this need to be a special case? If a hardware NN is basically a VLSI circuit, either analog or digital, why can't standard testing and fault tolerance techniques apply directly and be enough?
The answer to this question is the motivation behind the work carried out in this thesis.

1.2 MOTIVATION

Typically, post-manufacturing testing in the VLSI industry aims at detecting manufacturing errors and is done per manufactured chip. In high-volume production, testing is performed on Automated Test Equipment (ATE) and needs to be completed in a few seconds. When it comes to safety- and mission-critical applications, testing would also need to be performed in the field concurrently with the operation or in idle times, in order to detect latent defects, aging, etc., calling for BIST capabilities to be added into the design to allow stand-alone evaluation of the health status of the chip without relying on external test instruments.

Testing strategies also vary depending on the type of the Integrated Circuit (IC), i.e., digital or analog. Testing of digital ICs is considered a mature field today [26], [27]. On the other hand, post-manufacturing testing for analog ICs still relies on measuring certain performance metrics of the circuit and comparing them to the nominal case [28], hence tests are still specific to the particular IC (e.g., Analog-to-Digital Converter (ADC), Digital-to-Analog Converter (DAC), Phase-Locked Loop (PLL), filter, Operational Amplifier (OpAmp), etc.), and specific to different architectures within each IC class (i.e., Successive-Approximation Register (SAR), pipeline, ΣΔ, etc., architectures for the ADC class). BIST for analog ICs is not widespread since analog signal paths are sensitive and BIST circuitry tapping into them loads the IC and degrades the performance.

Like any other VLSI circuit, hardware NNs are prone to hardware faults that can happen any time during the chip's lifetime, and standard post-manufacturing testing strategies can technically be sufficient. State-of-the-art deep ANNs comprise a multitude of layers of different types, i.e., convolution, pooling, fully connected, etc., tens of millions of synaptic weight parameters, and they perform a myriad of operations in a single forward pass. From a hardware perspective, this entails immensely dense designs, albeit with a certain degree of architectural modularity. Hence, intuitively, testing efficiency can be largely improved by exploiting the architectural particularities of hardware NNs and targeting only those fault scenarios that have a measurable effect on performance [29].

Most likely, post-manufacturing testing for digital NN implementations will not be any different than testing of regular digital ICs. However,

for purely analog NN implementations or implementations with analog sub-blocks, e.g., analog neurons, new post-manufacturing and BIST test strategies will need to be developed since, in the first place, it is not clear for specifications we should be testing. In addition, adapting testing strategies to each network type, architecture, or data type is an exhaustive task.

To this end, in the course of this thesis we investigated the possibility of global testing strategies and generic fault-tolerance techniques that can be applied to hardware SNNs, independent of the architecture, design, and data propagated through the network. The goal is to enable SNNs with fault tolerance capabilities against hardware-level faults.
The followed methodology is discussed next.

1.3 METHODOLOGY

To achieve our goal, we adopted a methodical approach of evaluating the robustness of a neural network against hardware-level faults, to which extent they affect the NN, and how they can be mitigated or bypassed. The approach comprises 3 main steps:

1. Simulation of faults injected at transistor level to understand how they affect the circuit.
2. Creating abstract fault models that can be used to test the network at system level.
3. Developing cost-effective fault tolerance mechanisms for SNNs.

At every step, the findings are evaluated and exploited to optimize the next.

1.3.1 *Fault Injection*

The first step towards our goal is fault injection experiments. Fault injection is the intentional introduction of faults in a system to examine their effects [30]. While not really adequate for improving the system performance or debugging design errors, fault injection and simulation is a very effective way of testing the resilience and fault-tolerance capability of a system against known faults. In general, fault injection experiments can be helpful in evaluating the functional safety of a system with respect to the standard regulations of the respective domain.

Performing hardware-level fault injection experiments in hardware NNs, carried out at transistor-level or at system-level, can be a helpful tool so as to:

- ❖ Associate behavioral errors with hardware faults.
- ❖ Determine which faults are catastrophic and which are redundant.
- ❖ Build a compact fault model that can test the network regardless of its architecture, thereby reducing the post-manufacturing test costs.
- ❖ Identify critical layers and components within the network, so that appropriate strategies for hardening, self-test, error mitigation, error recovery, fault-tolerance, etc. can be followed.
- ❖ Understand the propagation of faults across layers. For example, it is shown that the normalization layers reduce the impact of faults by averaging fault values with adjacent correct values.
- ❖ Develop effective design-for-test techniques that exploit the hardware particularities.

1.3.2 *Fault Modeling*

After performing fault-injection experiments at transistor-level, we exploit the results in order to build a fault model and a fault taxonomy that is specific to SNNs.

For the fault model to be efficient, it needs to be defined with certain characteristics. The fault model should be:

- ❖ Consistent with manufacturing defects that can affect the chip, i.e., with the transistor-level faults.
- ❖ Defined at the behavioral level to allow the acceleration of fault simulation for deep SNNs.
- ❖ Abstract, meaning that it can be used to test any SNN regardless of its architecture or implementation.

A fault model of this style allows the acceleration of large-scale fault injection experiments for deep SNNs. This way, the vulnerabilities of the network can be assessed, and measures can be taken to bypass or mitigate their effects.

1.3.3 *Fault Tolerance*

The final step in this work is proposing some fault tolerance techniques in compliance with the results of the fault simulation experiments. But before we can do that, we need to illustrate what fault tolerance actually

means, and how it differs from other terms often encountered in the literature.

A system is said to be *reliable* if it performs correctly with high probability in the presence of faults under previous stated conditions, meaning that statistics and probability theory are needed to estimate the reliability of a system. Reliability is a quality over time that is associated with failures that happen unexpectedly. Other popularly used terms are *robustness* and *error resilience*, which describe the tolerance of the system to noisy inputs and inexact computations.

Fault tolerance, on the other hand, is a property of the system that guarantees its proper operation in the event that one or more faults have manifested themselves in the system. This means that fault tolerance entails taking action to detect and bypass or counteract the effects of faults.

1.4 THESIS STRUCTURE

In this thesis, we explore the inherent resiliency aspects of hardware SNNs to hardware-level faults and propose cost-effective fault-tolerance techniques applicable to any SNN.

In Chapter 2, we review the literature on SNNs and the work done to make them fault tolerant. Then, we kick off the experiments by designing a behavior-oriented BIST for a biologically inspired analog neuron in Chapter 3. Afterwards, we move on to a more practically adopted neuron model in Chapter 4, where we perform defect-oriented testing that maps transistor-level faults in analog neuron implementations into behavioral-level fault models. In Chapter 5, these fault models are used to assess the resiliency of a complete network, demonstrating the experiment on 2 separate SNNs. We leverage the results of these experiments in Chapter 6 to develop a neuron fault tolerance strategy for SNNs, and we show the fault-tolerant SNN architecture. In Chapter 7, we show a hardware convolutional SNN designed to run on FPGAs, and which we use as a platform to validate the ideas developed in this work. Finally, Chapter 8 concludes the work.

2 SPIKING NEURAL NETWORKS & THEIR FAULT TOLERANCE: A LITERATURE REVIEW

Despite all the impressive advances achieved by conventional ANNs, the human brain still excels in the most basic cognitive tasks. Originally a topic of interest in theoretical neurobiology and biophysics research, spiking neural networks were designed as an attempt to mimic the biological neural networks to enable the analysis of elementary processes of the brain. With the technological leap in AI and deep learning, neuromorphics and SNNs have come to the spotlight as a promising new paradigm that can possibly take deep learning to places it could not reach before.

In this chapter, we offer a basic literature review of the two pillar topics of this thesis: SNNs in hardware and previous efforts to make them fault tolerant.

2.1 SPIKING NEURAL NETWORKS

A biological or an artificial neural network is made up of neurons connected in a sophisticated complex pattern through synapses. The neuron is the basic processing unit in a neural network, and according to the computational paradigm of their processing units, ANNs can be classified into three generations [31]. The first generation is based on the McCulloch-Pitts neurons, or the perceptrons. Built exclusively to give digital outputs, a multilayer perceptron is able to compute every boolean function with only one hidden layer. The second generation is based on neurons implementing an "activation function", such as the sigmoid function. Neural networks from the second generation can compute functions with analog inputs and outputs, which give them a more realistic essence than the first-generation networks if their output is thought of as representing the average firing frequency of a biological neuron. Nonetheless, functionality-wise, both generation models are very different from biological neural networks.

Neurons in the brain communicate through discrete short electrical pulses called spikes. A typical neuron fires spikes at a frequency that is less than $100MHz$, which means that a window of $20-30ms$ is needed only to compute the current firing rate [31]. However, experimental results have shown that a visual processing task can be completed in just $20-30ms$, thus it was doubtful that biological neurons actually use the firing rate as a main coding scheme [32], [33]. Instead, it seemed that the timing of spikes plays an important role in neural coding [34].

These findings made way for a third generation of neural networks that uses spiking neurons as the basic element. Spiking neurons communicate -much like their biological counterparts- through spikes and code information in a spatio-temporal manner, making them more biologically realistic compared to the previous two models. In this section, we review the basic concepts of SNNs.

2.1.1 *The Spiking Neuron*

The spiking neuron is the basic building block of neuromorphic systems and where most of the processing takes place. A spiking neuron is often described as an integrator with a threshold [35]. Instead of constantly firing, as in the case of first- and second-generation neurons, a spiking neuron accumulates inputs from preceding neurons, and if a certain number of spikes occur within a specific time frame, it generates a spike of its own.

2.1.1.1 *The Spike*

The Spike, also known as the ***action potential***, is the primary means of communication in between spiking neurons. It is a short electrical pulse of around $100mV$ amplitude and a $1-2ms$ duration, that represents an abrupt momentary change in the state of the neuron. Spikes are stereotypical events, shown in Fig. 2.1, that do not change form as they propagate from pre-synaptic to post-synaptic neurons. Consequently, the shape of the spike carries no information. Instead, the timing of each spike and the length of the inter-spike intervals are the key aspects of information coding in SNNs.

2.1.1.2 *The Membrane Potential*

The membrane potential is the voltage that describes the state of the spiking neuron at any given time. In the quiescent state with no input spikes, the membrane potential of a typical biological neuron is strongly polarized at a resting voltage of about $-65mV$. Incoming spikes evoke

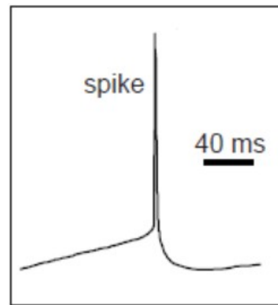


Figure 2.1: A spike

a positive or a negative change in the membrane potential, referred to as the ***post-synaptic potential***. An excitatory input will reduce the negative polarization of the membrane potential, i.e., depolarize it, while an inhibitory input will hyperpolarize the membrane voltage even further.

2.1.1.3 *Spike Generation*

The mechanism of spike generation is conceptually illustrated in Fig. 2.2. When an input spike arrives, the membrane potential increases for a moment before it starts decaying again. This decay of the membrane voltage between incoming spikes is known as the ***leakage***, which plays a significant role in the temporal correlation between spikes. Temporal correlation is a crucial concept which facilitates the possibility of exploiting the temporal information contained in real-world sensory data [31].

After the neuron spikes, the membrane potential returns to the resting value and stays there until the next input spike comes along. The neuron cannot fire again until a certain period has passed, which is known as the ***absolute refractory period***. In response to an input stimulus, spiking neurons can emit a chain of action potentials, called a spike train, that can have regular or irregular intervals.

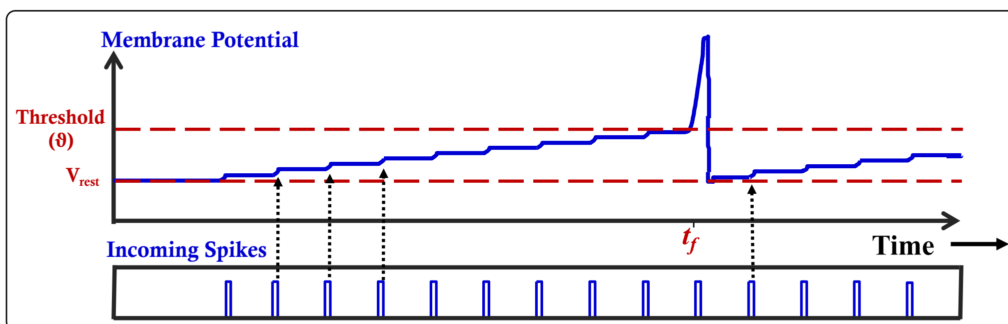


Figure 2.2: The dynamics of spike generation.

2.1.2 *Neural Coding Schemes*

The concept of mean firing rate dates back to the 1920s [36], where Adrian *et al.* proved that the firing rate of certain neuron types in the muscles is related to the strength of the applied stimulus. For so long after that, the firing rate was thought to be the principle neural coding scheme. However, neuroscience experiments have demonstrated that for high-speed neural processing, the timing characteristics of individual spikes and spike trains cannot be ignored [32].

In SNNs, Coding schemes can be classified into [37]:

1. **Rate Coding Schemes**: converting the activity level of neurons into a firing rate, which does not benefit from sparse spikes inherent to biological systems [38]. The neuron firing rate can be calculated according to:
   - ***The spike count***: averaging over time.
   - ***The spike density***: averaging over several runs.
   - ***The population activity***: averaging over the activity of several neurons that act together.
2. **Spike Coding Schemes**: Coding strategies that are based on the timing of individual spikes. There are many timing aspects through which neurons in the brain communicate information, including:
   - ***Time to first spike***: in systems that require ultra-fast information processing, such as tactile and olfactory systems, the delay between the beginning of the input stimulus and the first output spike was shown to carry enough information [39], [40].
   - ***Phase coding***: where neurons can encode information in the phase of a spike with respect to some background oscillations.
   - ***Rank order coding***: information is coded in the order of output spikes in populations of neurons [41].
   - ***Correlation and synchrony***: a coding scheme based on the observation that neurons that encode pieces of the same information object fire synchronously.

Nonetheless, the distinction between rate coding and spike coding scheme is not very sharp, i.e., some of the codes that are viewed as timing codes can also be interpreted as variations of a rate code. For example, the time-to-first-spike coding scheme is related to the firing rate scheme since a neuron that spikes early would be expected to have a high firing rate, and vice versa.

2.2 SPIKING NEURAL NETWORKS IN HARDWARE

2.2.1 *Neuron Models*

There are numerous models of spiking neurons used in neuromorphic systems [42], most of which perform the accumulation and firing function, albeit with different mechanisms. In the simplest deterministic model, the neuron is assumed to fire whenever its membrane potential reaches a threshold (ϑ). The fired spike travels along the synapse that has an efficacy, or weight (ω).

Spiking neuron models can be grouped into broad categories based on their degree of complexity and their resemblance to biology in structure or function [43]. The question of which neuron model to use usually depends on the type of problem at hand [44]. These categories are:

- **Biologically-Plausible Models**:
  These models are built to explicitly model the behaviors observed in biological neurons and are mostly used for accurate simulations of biological neural systems. One of the most famous models of this category is the classical Hodgkin-Huxley model [45], which uses four-dimensional nonlinear differential equations to describe neural behavior. Other models like the Morris-Lecar model [46] try to simplify things by reducing the dimensions of the nonlinear equations. In any case, these models are often used to simulate a small number of neurons since they are usually very expensive to implement.

- **Biologically-Inspired Models**:
  Biologically inspired models, on the other hand, try to replicate the neural behaviors without the obligation of emulating the physical activity of biological systems. Consequently, they are much simpler than the biologically plausible models in terms of computations and implementations and can therefore be more efficient in modeling large-scale systems. There are many models in this category that are very common in neuromorphic literature, including the FitzHugh–Nagumo model [47], [48] and the popular Izhikevich model [49], which can reproduce biologically accurate behavior with a set of two-dimensional ordinary differential equations.

- **Integrate-and-Fire Models**:
  Another category of spiking neuron models especially suited for minimizing computational complexity is the Integrate-and-Fire (I&F) models. These models can vary in complexity from the basic integration and firing function to something approaching the complexity level of the Izhikevich model. While not biologically realistic, I&F models are

still complex enough to incorporate spiking dynamics. Beside the basic I&F model, there are also the leaky I&F model, the resonate-and-fire model [50], and the quadratic I&F model [51]. More complex implementations also exist, such the Adaptive Exponential Integrate and Fire model [52].
Another model that can technically fall under this category, although it performs the neural behavior in the form of response kernels rather than differential equations, is the Spike-Response Model (SRM) [34]. The SRM is a generalized form of the I&F model and will be discussed in more details in Chapter 5.

2.2.2 *Address Event Representation*

In the attempt to build neural networks that resemble the human brain, the communication problem becomes evident. Compared to digital systems where each gate is connected to a relatively small number of outputs, a typical neuron can carry outputs to thousands of targets. This massive interconnection is easily built in biology using 3-D structures. However, it becomes a challenge when brought to 2-D silicon substrates [53]. On the other hand, a typical neuron fires at a rate ranging from $1-10$ kHz, which means that a cluster of neurons can have a collective firing rate in the kHz to low MHz range; a rate that modern digital systems can effortlessly hold. Therefore, in neuromorphic systems, communication is usually carried out between clusters of neurons by time-multiplexing spikes on a single channel. In neuromorphic systems, this is done using the Address Event Representation (AER) protocol [54].

AER is a communication protocol that makes use of the fact that spikes carry no information other than the fact that neuron i fired at time t_f, and sends out only this information. Each neuron in a cluster is assigned a unique address and every generated spike is encoded into a sequence that indicates the identity (or address) of the neuron that fired that spike and the time at which it fired. This way, huge synaptic connections can be avoided, and the hardware resources can be efficiently allocated where they are needed.

There are multiple approaches to the implementation of AER event-based systems that exploit the features of a hardware neural network [55]. The simplest form is a basic AER scheme where the whole network can share a single communication channel, and all events are processed through an external block that controls their traffic. Other schemes distribute this multiplexing among several points in the network. A more sophisticated scheme puts a router block within each cluster module that is responsible for directing traffic between modules. Using this scheme

can be helpful in large-scale systems because it allows a 2-D mesh architecture where each module is only connected to its immediate neighbors. The router-based scheme will be further explained in the case study presented in Chapter 7.

2.3 NEUROMORPHIC TECHNOLOGY PROSPECTS AND CHALLENGES

Judging by the rise in research activity over the past decade [43], neuromorphic computing is becoming mainstream with every passing day. This is mainly because of the many advantages that this technology promises, which could potentially take neuromorphic technology far beyond what has been accomplished by conventional ANNs. These advantages can be summed up into:

1. **Power efficiency.**
   Much like biological neural networks, SNNs are built for event-driven information processing. Information from the outside world is usually sparse, which means it can be processed in a way much more efficient than the frame-based approach of conventional ANNs. ANNs perform data sampling at predefined time steps that ignores the speed of incoming information. SNNs on the other hand process spikes as they come, meaning that the network is only working when there is something to be processed, otherwise, no computations are taking place. This event-driven operation is the basis of the huge power savings promised by neuromorphic computing and could make neuromorphic systems the go-to solution for applications such as IoT at the edge and autonomous vehicles. For example, IBM's TrueNorth Chip is a Complementary MOS (CMOS) ASIC made up of 5.4 billion transistors that make up 1 million neurons and 256 million synapses, all the while consuming only 73 mW.

2. **Speed of computation.**
   Information in the brain is transmitted in the form of spikes propagating from layer to layer as soon as they are generated by a neuron. In contrast, conventional ANNs compute the output of neural layers sequentially, introducing significant delays while each layer waits for the output of the previous one to be computed. SNNs incorporate time into their model of operation together with the state of neurons and synaptic weights, thus information flows in the form of spike trains propagating between neurons asynchronously. When combined with event-based sensors, this leads to *pseudo-simultaneous* information processing, where a first estimation of the

output can be calculated at the output layer almost immediately after the introduction of the first input spikes [56].

3. **Predisposition to bio-inspired unsupervised learning.**
While supervised learning techniques have become the standard for ANNs, to this day, exact mechanisms of supervised learning in biology are still a mystery and presumably not the primary source of learning. The Hebbian process of learning proposed by Hebb in 1949 was the first attempt at explaining how learning occurs in the brain. The Hebbian plasticity principle, and the more comprehensive version known as Spike Timing-Dependent Plasticity (STDP) [58], [59], entail that the change in the weight of a synapse is proportional to the timing between pre- and post-synaptic spikes. Since SNNs are designed to use spike timing to code information, it is due to make full use of these unsupervised biologically inspired learning algorithms.

4. **Inherent robustness to noise.**
As discussed in Section 2.1.1.1, spikes have a standard shape that holds no information, making the presence of a spike the most important aspect. Moreover, every neuron generates its own set of spikes instead of just passing the same signal along, hence there is a kind of regeneration of signals at every neuron [60]. This gives SNNs an advantage similar to that observed in digital systems, that is their robustness to noise.

But with great promise comes great challenges. Despite all its promising prospects, neuromorphic technology still faces many obstacles that need to be addressed before it can be implemented in real world applications. The main obstacles that face neuromorphic technology today and stop it from realizing its full potential are discussed next.

1. **Benchmark datasets in spiking form.**
While it is counted as a drawback of SNNs that they cannot achieve high accuracy levels on typical benchmark datasets such as the MNIST handwritten digits dataset [61] or the CIFAR image dataset [62], the real issue is with the nature of these datasets. These datasets were created in a frame-based format which is fundamentally different from the way SNNs operate. The conversion of typical datasets into a spiking format is usually inefficient, since frame-based format holds a huge amount of redundant data that is simply not present in event-based formats. In the recent years, a new direction is being followed of directly collecting datasets in spiking format using Dynamic Vision Sensors (DVSs), such as the N-MNIST

dataset (a neuromorphic version of the MNIST dataset) [63], the Dvs-Gesture dataset [64], along with other benchmark datasets that are optimized for visual recognition [65], [66].

2. **Efficient spike-based learning algorithms.**
   Another major limitation of SNNs is the lack of efficient training algorithms that leverage the full potential of spike coding capabilities. Conventional ANNs have reached amazing performance levels using supervised learning techniques that are based on gradient-descent, such as the ubiquitous back propagation algorithm [67]. However, applying back propagation directly to ANNs is infeasible since spikes are non-differentiable. Over the years, many approaches to training SNNs emerged. The first approach was the conversion of trained conventional ANNs into SNNs by adapting the weights and parameters, which offered SNNs the opportunity to benefit from the strength of deep learning but at the cost of less efficiency. On the other hand, multiple approaches have been proposed to use variants of the back propagation algorithm for direct training of SNNs [68]–[70]. Nonetheless, due to the complexity and asynchronicity of spike-based computing, the issue of SNN training algorithms remains a huge challenge for neuromorphic technology.

3. **Efficient architectures.**
   There are various neuromorphic chips that exist today from multiple manufacturers and research labs, e.g., the **TrueNorth** chip from IBM [15], the **SpiNNaker** chip from the University of Manchester [16], **Eyeriss** from MIT [17], and Intel's **Loihi** chip [18]. With the approaching end of Moore's law, the scaling down of VLSI chips needs more innovative solution to allow the scalability of neuromorphic chips and ANNs to incorporate more neurons and synapses without compromising the power consumption. In a neural network, the number of synapses is usually much higher than that of neurons, which is why synaptic implementations are usually kept simple and a lot of effort is being directed into improving synaptic implementations and finding better materials and techniques to achieve the synaptic function.

   One of the biggest bottlenecks in ASIC implementations for neuromorphic systems is the on-chip memory that holds the parameters and the synaptic weights. The standard technique used today is the on-chip or off-chip Static Random Access Memory (SRAM). However, they do not provide sufficient capacity to cope with the large number of parameters and synaptic weights in ANNs. The typical SRAM density is $100 - 200\ F^2$ per bit cell, where F is the technology node

[71] with a capacity of a few megabytes on-chip. Emerging technologies such as Non-Volatile Memory (NVM) devices, e.g., memristor crossbars, are attracting so much interest these days because of their analog nature and the capability of doing in-memory computations [72]–[74].

4. **Robust and reliable hardware.**
In Chapter 1 we discussed the appeal of neuromorphic technology for safety-critical applications and how this made their testing a must. However, there is still a gap in the literature concerning testing techniques and fault-tolerance strategies applied for SNNs. This issue will be discussed in more detail in the rest of this chapter.

2.4 STATE-OF-THE-ART IN TESTING AND FAULT TOLERANCE IN HARDWARE NEURAL NETWORKS

Despite all the major breakthroughs achieved by ANNs, their transfer into hardware unavoidably poses multiple factors that can degrade their performance. Perhaps the most important risk of hardware platforms is their susceptibility to faults either during manufacturing, such as physical defects and process-induced variations, or in the field due to environmental factors and aging. Consequently, fault detection and fault tolerance are essential to achieve better performance levels.

In this section, we review the concepts of fault tolerance and then we explore the body of work that has been carried out in the field of testing and fault tolerance for hardware neural networks.

2.4.1 *Fault, Error, and Failure*

While sometimes used interchangeably, fault, error, and failure are distinct concepts in fault-tolerant systems. A fault is the anomaly in the physical condition of a circuit arising from a defect in the manufacturing process. This fault can reflect as an error at the functional level of the system, e.g., deviation of the output from the expected value. If this deviation is large enough, this error can eventually lead to failure to meet the intended functionality purposes. Fig. 2.3 shows the conceptual relationship between the three terms.

Like all mass-produced ICs, hardware neural network circuits are predestined to have faults. These faults can be **hard faults** leading to fatal errors and eventual failure of the circuit, or they can be **soft faults** which manifest as non-fatal errors that could simply disrupt or degrade the circuit performance.

Faults can also be classified according to their temporal characteristics into permanent and transient faults. **Permanent faults**, as the name implies, are irreversible, e.g., shorts and opens, bridging and stuck-at faults. They are usually a manifest of a physical damage either during manufacturing or after employment. **Transient faults** on the other hand are temporary and they are often triggered by external factors.

2.4.2 *Fault Injection in the Literature*

As stated in Section 1.3.1, fault injection experiments are done with various goals in mind. They can be used to showcase the tolerance of an ANN to certain faults, to identify the critical faults in order to decrease testing time and cost, or to build cost-effective fault tolerance techniques.

Faults in a neural network can be observed at the signals in the communication channels, the synaptic weights, or at the neurons. There have been many attempts in the literature to find fault models suited for neural networks and to perform fault injection experiments accordingly. Fault injection experiments for different ANN architectures have been recently performed using fault models at different levels of abstraction, some of which we discuss in this section.

In [75], Bolt provided a framework for developing behavioral-level fault models for multi-layer perceptrons based on the fault location and characteristics to facilitate the investigation of their inherent fault tolerance. Another attempt was made by Chandra *et al.* [76], where a fault model was proposed for feedforward activation-based ANNs that covered external noise which can introduce faults to the input of the network, and structural faults at the layers, neurons, and synapses. They also introduced some fault and parameter-sensitivity measures for these networks.

While these fault models are meant to be abstract and hence usable on any ANN regardless of its design or architecture, they are not accountable in the study of defect tolerance of hardware ANNs. Several fault injection

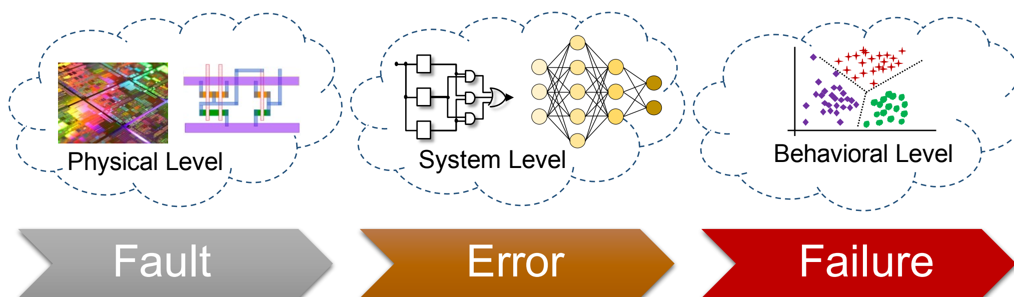


Figure 2.3: Cause-Effect relationship between fault, error and failure.

experiments for different ANN architectures have been recently performed using fault models at different levels of abstraction, which exposed the vulnerabilities of AI hardware accelerators to hardware-level faults. Faults have been injected in the form of:

- **Transistor-level faults:**
  In [77], Temam investigated this issue by analyzing the differences between gate-level defects (stuck-at) and transistor-level defects (shorts and opens in transistors) in the fundamental logic operators of a multi-layer perceptrons, i.e., adders and multipliers. The author then created a gate-level fault model from faults injected at transistor level and used this model to inject faults into a 2-layer multi-layer perceptron designed in Verilog and synthesized with Synopsys Design Compiler. The performance of the network for different tasks was shown to drop with the number of injected defects and the amplitude of the errors they introduce.

- **Stuck-at faults:**
  Nonetheless, the stuck-at fault model has become very popular in modeling faults in neurons and synapses, following the footsteps of digital-circuits testing, because of the wide range of faults it can cover [78]. Using the stuck-at model for an ANN, faults in neurons can be modeled as stuck-at-1, stuck-at-0 or stuck-at-(-1) depending on the neuron model, and in synapses as stuck-at-*value*, where *value* falls in the range of $[\omega_{min}, \omega_{max}]$.

  In [79], stuck-at faults were used to verify the test coverage capability of on-line test strategies based on Software Test Libraries (STL) proposed for embedded systems running ANN applications. The strategies are evaluated on Convolutional Neural Networks (CNNs) running on an open-source RISC-V platform. Then the different testing strategies are evaluated based on the inference time penalty and fault detection time trade-off, where fault detection time is the worst-case time to detect a fault from the moment of occurrence.

  Gate-level stuck-at and transition faults were used in [80] for testing First-In-First-Out (FIFO) based and Scratchpad based neural network hardware accelerators. Stuck-at faults were also considered in [81] for a systolic-array based neural network accelerator, i.e., the Tensor Processing Unit (TPU), where the network was shown to be severely sensitive to permanent faults in its Multiply-and-Accumulate (MAC) units, with the classification accuracy dropping from 74.13% to 39.96% when only 0.005% of the MAC units were faulty.

  In [82], Gebregiorgis *et al.* proposed the conversion of neuromorphic circuit testing into a Boolean satisfiability (SAT) problem that can gener-

ate test patterns tailored to the important faults that can affect an ANN, defining an "important" fault as one that drops the inference accuracy of the circuit beyond an acceptable level. The authors demonstrated their test flows on the Register-Transfer Level (RTL)-level hardware implementation of two ANNs and showed that by choosing an appropriate margin, the number of important faults can be significantly reduced and that testing costs can be equally reduced by focusing only on these faults and ignoring others that were found non important.

Permanent faults causing stuck-at activations at a quantized value were investigated in [83] for convolutional Quantized Neural Networks (QNNs). The experiments were carried out on FPGA-based hardware accelerated to enable fast evaluation, and the robustness of the network was assessed and optimized as a result.

- **Stuck-at faults in the conductance of memristors in memristor crossbars:**

  Fault models and injection experiments specific to memristor crossbars have also been studied. In [84], fault injection is performed for a shallow 2-layer SNN with a memristor-crossbar connecting the two layers. An in-house simulator was developed for the fault injection campaign and all simulations were performed with the CPU of a PC. Although a fault taxonomy was proposed, fault injection experiments considered only "dead" neurons, i.e., neurons that do not fire, and "dead" synapses, i.e., synapses that do not allow signal transfer from one neuron to another.

  Liu *et al.* in [85] showed that for 20% affected memristors in the crossbar, the classification accuracy can drop from 92.64% to 39.4%. In [86] and [87], stuck-at faults in the conductance of memristor crossbars were used to qualify techniques to boost the fault tolerance of Resistive Random-Access Memory (RRAM) crossbar-based accelerators.

- **Bit flips in data-paths, buffers, and memory:**

  Another form of faults considered for testing all- (or partly-) digital ANNs hardware is the static and/or transient bit flips. Li *et al.* [88] considered transient SEUs in data paths and buffers, demonstrating their experiments on four convolutional neural network accelerators written in C++ and implemented using an open-source simulator framework. Faults were randomly injected in the hardware components and the error propagation through the network was studied. Their results show that the sensitivity of the network to faults depends upon the architecture and topology of the network, the used data type, and the position of the flipped bit. The authors concluded that the projected Failure-in-Time (FIT) rate of faulty Deep Neural Networks (DNNs) can

exceed the safety standards, e.g., ISO 26262 for automotive, by orders of magnitude.

In [89], single bit flips were injected into a randomly chosen word from the activation function computations or the weights of a small multi-layer perceptron with 2 hidden layers. Bit flips were again used in fault injection experiments at software layer in [90], where Bosio *et al.* evaluated the effects of these faults on the performance of two CNNs and classified errors into masked and observed faults, which were further classified into safe and unsafe faults according to their impact on the classification accuracy. They used this fault classification to assess the criticality of faults per layer and per bit category. The analysis was further extended in [91] with additional data representations in the form of five additional fixed-point formats with sizes ranging from 32 bits down to 6 bits. Bit flips were introduced to observe the resulting percentage of unsafe faults. The authors concluded that the convolutional layers are more reliable than fully-connected layers, and that the criticality of the bit flips is linked to the type of bits and data precision.

In [92], the correlation between fault rate and classification accuracy of DNNs was investigated through bit flips injection into the memories of weights, activations, and hidden states, and the variance of fault tolerance levels was compared across six SNNs models, layers, and structures. Their analysis showed that there is a Bit-Error Rate (BER) threshold beyond which the error increases exponentially. Across the DNN models, this threshold varies from 4×10^{-9} to 6×10^{-6}, while for a particular DNN model the error across different layers varies up to 2781x. They also concluded that the bit position as well as the dynamic range of values that the data type offers play an important role in fault sensitivity.

This observation was further examined in [93], where Santos *et al.* performed fault injection, in the form of bit flips, to evaluate the reliability of a DNN implemented in three different precisions (half, single, and double) on NVIDIA mixed precision GPUs. They studied the propagation of faults through the network and the effect of mixed-precision data on the criticality of faults and showed that reducing the precision on GPUs can reduce the error rate resulting from faults and improve the performance.

Another experiment in the form of bit flips in the weight memory was carried out by Neggaz *et al.* in [94]. Neggaz *et al.* developed and used a fault injection engine implemented in Python to analyze the reliability of classification CNNs accelerated on a GPU. They considered

two data types, the 32-bit float and 8-bit fixed point, and incrementally injected random bit flips and measured the classification accuracy. They observed less vulnerability to bit flips in the case of the 8-bit representation as opposed to the 32-bit representation, and hence reached the conclusion that weight quantization has a positive impact on CNN's resilience to SEUs.

These experiments elucidated the vulnerabilities of AI hardware accelerators to hardware-level faults. A common conclusion across many of these studies is that fault sensitivity varies across network architectures, types of layers, as well as across low-level components, i.e., activation functions and synapses. However, the vast majority of the aforementioned fault injection experiments concern level-based DNNs and memristor crossbars, while few experiments have focused on SNNs.

2.4.3 *Fault Tolerance in the Literature*

According to the way it is achieved, fault tolerance can be classified into active and passive [29]. In this section, we provide a review of the most common fault tolerance techniques applied to ANNs by presenting a handful of works representing each technique.

1. ***Passive Fault Tolerance:*** Passive fault-tolerance schemes entail no special response to the occurrence of a fault, i.e., no fault detection or localization, no retraining after fault occurrence, and no reconfiguration of the system. Instead, the intrinsic characteristics of the system are leveraged to mask the known effects of faults and ensure the correct operation of the system in case the fault happens. The two main categories often practiced as passive fault tolerance techniques are:

   ❖ **Explicit Redundancy:**
   This method starts with a trained network that has a minimal architecture necessary to implement the given task, then extra neurons or synapses are added to split the burden with the original ones. Contrary to intuition, the blind addition of hidden units in a neural network before training does not improve its fault tolerance [95]. Instead, the redundancy should be added with proper adjustments and after careful considerations of the role each unit plays in achieving the task at hand.

   In [96], the hidden neurons in a 2-layer NN are replicated after the network is trained, then the inputs and the biases are appropriately scaled. In [97], Chiu *et al.* developed a quantification method for the sensitivity of neurons and synapses in a feedforward ANN, and based on that measure, unnecessary units are cut

from the network. Afterwards, they retrain the network based on the new architecture and dynamically add extra units until there are no critical points in the network, resulting in an order of magnitude less performance degradation than a randomly trained feedforward network of the same size.

The same approach was followed by Dias *et al.* [98], where parts of the network are duplicated, and the network architecture is rearranged so that the functionality is maintained without the need for retraining. Critical synapses are split, and the weight is divided among them to weaken their impact, and critical neurons are duplicated so each one can do half the work, which increases the fault-tolerance of the network up to 46%.

- **Modified Learning:**
  This category goes to the training phase of the network and adds extra options or enforces constraints on the learning algorithm to ensure the network's tolerance to possible faults.

  Faults can be injected to the network during training, as in the case of [99] where Arad *et al.* modified the training experience of the network by performing the training with the increasing probability that several hidden neurons are faulty. They demonstrated that using this method provided the network with the ability to tolerate two simultaneous faults, and that this tolerance is linked to the size of the dataset used for training.

  Another direction is adding some regularization terms to evenly distribute the weights and computations across the elements in the network and hence force the algorithm to produce a network with better fault tolerance. In [100], Bernier *et al.* proposed a modified version of the backpropagation algorithm that has an extra term proportional to the error caused by weight deviations. Their algorithm successfully reduced the sensitivity of the network to possible faults in weight values without degrading the network performance or increasing the training time. A similar idea was developed in [101] by developing a training algorithm suited for a special fault model they assumed.

  Uniformity of weight distribution can also be achieved by constraining the weights of neurons in a specific layer in the network to a specified range before training [102].

The passive fault tolerance techniques discussed above are effective in terms of improving the inherent fault tolerance of ANNs. However, they usually come at a cost. This cost can be in terms of size, as in the case of explicit redundancy, and the network can become too large

with too many hidden neurons or parameters than necessary. The cost can also be in terms of computations complexity and training time when the action is taking at the training level.

2. ***Active Fault Tolerance:*** On the other hand, in the active fault tolerance scheme, explicit resources are added to test the performance of the system, detect the presence of faults, and compensate for their effects. Active fault tolerance equips the system with the ability to recover from faults when they happen or completely bypass faulty elements in the network. There are many techniques that fall under this fault-tolerance class, namely:

   - **Weight shifting:**
     Fault tolerance can be realized by a technique known as weight-shifting. Khunasaraphan *et al.* [103] proposed a self-recovery mechanism for feedforward networks where the weights of faulty synapses are shifted to other fault-free ones, and links to a faulty neuron are considered faulty as well and get shifted as well.

   - **Retraining:**
     It can also be realized by retraining of the network after faults happen so that the network can adapt to the faulty architecture. Hashmi *et al.* [104] studied the inherent fault tolerance of a biologically-plausible computational model of cortical perceptual maps used for vision recognition tasks and implemented on a general purpose GPU. They demonstrate that when a fault happens, the network does not need to be reprogrammed or recompiled to use non faulty units in the GPU. Instead, a simple retraining can adapt the network to the modified architecture even without explicitly excluding the faulty parts.

     Retraining was also considered by Deng *et al.* [105] for mitigating timing errors in a multilayer perceptron modeled at RTL level. Retraining is done to redistribute weights and avoid the effects of timing errors.

     Pandey *et al.* [89] proposed the use of a checker neuron for real-time resilience against soft errors in feed-forward neural network architectures. The checker neuron estimates the output of a layer, compares it against the output of the actual neuron layer, and when the difference is outside a specified limit, it triggers an error signal. In this case, the error can be corrected by bypassing the whole layer and subsequently re-training.

     In [106], a fault-tolerant design of Google's TPU hardware accelerator is proposed. If a MAC unit is detected to be faulty, then it is

pruned, i.e., bypassed, using multiplexers that are added into the design.

Liu *et al.* [85] studied the fault tolerance of memristor-based neural network by identifying the significant weights and developing a retraining algorithm that can compensate for 20% random single bit failures. Another technique based on on-line detection of faults and retraining to tolerate them is proposed in [107] for RRAM-based neuromorphic systems.

- **Biologically-inspired self repair:**
  A self-repair mechanism adopted from biology is explored by Naeem *et al.* [108], and further expanded in [109] and [110], that is based on astrocyte cells. In biological NNs, an astrocyte is a subtype of glial cells abundantly available in the brain, which provide feedback to synapses and regulate their transmission. In the presence of faults in the synapses which cause a drop in their probability of release, neurons can become silent. The authors demonstrate that astrocyte cells can repair this fault by increasing the probability of release of the non-faulty synapses, and hence the network can self-repair in the presence of up to 80% fault distribution.

  Another idea was presented in [111] where the authors designed an SNN used for controlling the motion of a robotic car and implemented it in FPGA. Fault tolerance is achieved using Dynamic Partial Reconfiguration (DPR), a technique specific to FPGAs that enable the modification of the circuit mapped on the board without having to turn it off. The authors use adjustable thresholds and clock rates to compensate for faults that can affect the neuron firing rates.

In the end, it is worth noting that most of the fault injection and fault tolerance work in the literature concerns conventional ANNs and their multiple implementations. Hence, there is a huge gap when it comes to global fault models and fault tolerance techniques that can be applied to a SNN of any depth, any architecture, and any implementation. For the integration of SNNs in critical applications to materialize, dependability and resilience become of utter importance. Therefore, the thesis addresses a timely problem that has been little studied as of today.

3 A SELF-TESTING ANALOG SPIKING-NEURON CIRCUIT

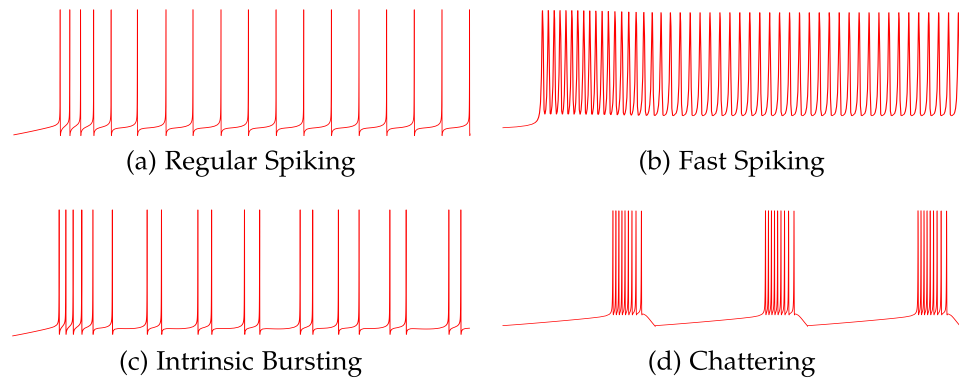
In the first step of this work, we addressed the problem of post manufacturing test and self-test of hardware-implemented spiking neurons [112]. To keep matters simple, we tried to find a small analog circuit that captures the most important features of spiking neurons, and we studied the applicability of self-test to it. Among the countless number of hardware implementations of spiking neurons, we chose a compact implementation of a biologically inspired spiking neuron model. In this chapter, we present the circuit design and features, then we propose a self-testable version of it.

3.1 A BIOLOGICAL PERSPECTIVE

Although all biological neural networks communicate through the temporal patterns of their output spike trains, not all neurons behave in the same manner. Neurobiological studies have shown that cortical neurons are not morphologically or physiologically homogeneous [113]. They differ in their chemical properties as well as their electrical fingerprint, which is determined by the intrinsic membrane properties.

From a behavioral point of view, the different intrinsic membrane properties of cortical neurons determine the way a neuron translates an input from a synapse into a spike output. These differences appear as variations in the shapes of the output firing patterns of the neuron in response to post-synaptic stimuli, examples of which are shown in Fig. 3.1. While there are many subcategories and many variations, the neuronal intrinsic firing patterns in response to a prolonged stimulus can be grouped into 4 main classes [113], [114]:

1. ***Regular Spiking (RS) Neurons:*** Regular spiking neurons are the most common in the brain. In response to a long-duration stimulus of constant amplitude, the neuron fires repeatedly, exhibiting gradual adaptation of its firing frequency, i.e., the inter-spike intervals



(a) Regular Spiking

(b) Fast Spiking

(c) Intrinsic Bursting

(d) Chattering

Figure 3.1: Intrinsic Firing Patterns of Cortical Neurons

increase gradually until it settles to a stable frequency, as shown in Fig.3.1a. The output firing frequency increases with the increase in the amplitude of the input stimulus, but the basic behavior stays the same.

2. ***Fast Spiking (FS) Neurons:*** Neurons in this class have faster rates of hyperpolarization and depolarization. As a result, they can respond to a constant input stimulus with relatively high-frequency periodic spike trains that undergo little or no noticeable adaptation, as shown in Fig.3.1b.

3. ***Intrinsically-Bursting (IB) Neurons:*** The term "burst" refers to a cluster of high frequency spikes which are followed by a period of silence. While any neuron can produce a cluster of spikes in response to repetitive phasic synaptic input, the intrinsic bursting behavior is distinguished only when it is due to the intrinsic membrane properties of the neuron and independent of the synaptic input. Intrinsically bursting neurons responds to a prolonged input stimulus with a single burst followed by repetitive smaller clusters or single spikes, as shown in Fig. 3.1c.

4. ***Chattering (CH) Neurons:*** A chattering cell responds to a synaptic stimulus with repeated bursts of spikes, shown in Fig. 3.1d, with an inter-burst interval that depends on the strength of the input stimulus. Although a single burst is similar to that produced by an IB neuron, the chattering pattern is strikingly different. Hence, it has been presumed as a distinct class of neurons, primarily present in the visual cortex.

3.2 THE BIOLOGICALLY-INSPIRED NEURON CIRCUIT

3.2.1 *The Mathematical Model*

In Chapter 2, we discussed the various types of spiking neuron models that exist in the literature and that the choice of the model to use depends upon the application. For the purpose of demonstrating a behavior oriented built-in self-test, we picked an analog VLSI implementation of a biologically inspired neuron model capable of generating the 4 firing patterns discussed above.

The designed neuron is inspired by the Izhikevich model [49]; a simple model that captures many biological aspects of spiking neurons, e.g., realistic spike shape and firing patterns, using a two-dimensional system of ordinary differential equations of the form

$$v' = 0.04v^2 + 5v + 140 - u + I \quad (3.1)$$

$$u' = a(bv - u) \quad (3.2)$$

and an adjunct resetting mechanism

$$\text{if } v \geqslant 30 \text{ mV, then} \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \quad (3.3)$$

where v is the membrane potential, u is a membrane recovery variable that can slow down changes in v, I is the input synaptic currents, and a, b, c and d are dimensionless parameters. The choice of the 30 mV and the parameter values depend on the fitting criteria of the neuron to be modeled, which is beyond the scope of this work. Detailed analysis and possible applications of this model can be found in [49].

3.2.2 *The Neuron Circuit*

For the purpose of our work, we are interested in the hardware implementation of the model [115], shown in Fig. 3.2. This circuit implements the Izhikevich model using a simple 14-transistor and 2-capacitor CMOS design, where the basic form of the mathematical model is preserved but the non-linearity is introduced through the underlying characteristics of the MOSFETs. By tuning two control variables, V_c and V_d, the circuit is capable of producing the 4 basic firing patterns of biological neurons with a spike shape that resembles that of real neurons.

The mathematical model of the circuit is of the form [115]:

$$\dot{V} = \begin{cases} \frac{k}{C_v}\Big\{\alpha \left[\frac{1}{2}\left(\frac{W}{L}\right)_{M_1} (V - V_T)^2\right] \\ \qquad - \beta \left[\frac{1}{2}\left(\frac{W}{L}\right)_{M_4} (U - V_T)^2\right] + \frac{I}{k}\Big\}, \text{ when } V \geqslant U - V_t \\ \frac{k}{C_v}\Big\{\left(\frac{W}{L}\right)_{M_4}\left((U - V_t)V - \frac{1}{2}V^2\right) + \frac{I}{k}\Big\}, \text{ when } V < U - V_t \end{cases} \tag{3.4}$$

$$\begin{aligned} \dot{U} = \frac{k}{C_v}\Big\{\alpha & \left[\frac{1}{2}\left(\frac{W}{L}\right)_{M_1}\left(\frac{L}{W}\right)_{M_2}\left(\frac{W}{L}\right)_{M_7}(V - V_T)^2\right] \\ & - \gamma\left[\frac{1}{2}\left(\frac{W}{L}\right)_{M_6}(U - V_T)^2\right]\Big\} \end{aligned} \tag{3.5}$$

where V is the membrane potential, U is a slow variable, C_v and C_u are the membrane and slow variable capacitance values respectively, and I is the post-synaptic current. $(W/L)_{M_x}$ is the gate width-to-length ratio of transistor M_x, V_T is the transistor threshold voltage, $k = \mu \times C_{ox}$ of the MOSFETs, and α, β and γ values depend upon V_T, V and U.

Equations (3.4) and (3.5) explain the changes in signals V and U with the input synaptic current. When V reaches an external set threshold value, V_{th}, an output spike is produced and V and U are reset, as follows

$$\text{if } V \geqslant V_{th}, \text{ then } \begin{cases} V \leftarrow V_c \\ U \leftarrow U + f(V_d) \end{cases} \tag{3.6}$$

Fig. 3.3 explains the most important aspects of the circuit's operation. The spiking behavior of the circuit is represented by the two variables, V and U, which are the voltages accumulated on capacitors C_V and C_U, respectively. Capacitor C_V integrates the post-synaptic input current I in addition to a positive feedback current I_V. A leakage current I_l is generated by transistor M_4, and the net sum of these voltages is integrated on capacitor C_V as

$$C_V \frac{dV}{dt} = I + I_V - I_l \tag{3.7}$$

When V reaches the threshold value, Eq. (3.6), the comparator generates two short-duration pulses, namely V_A and V_B, and the circuit spikes.

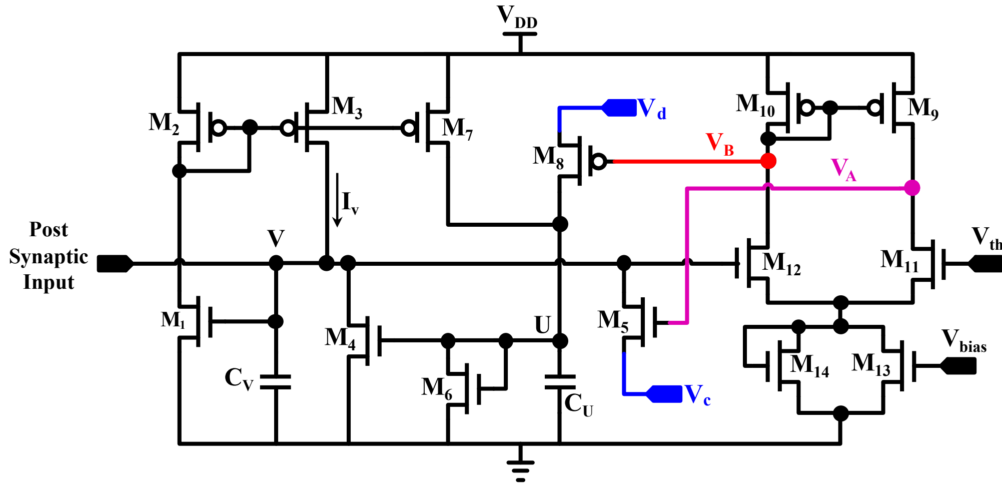


Figure 3.2: Transistor-Level Implementation of the Spiking Neuron Circuit

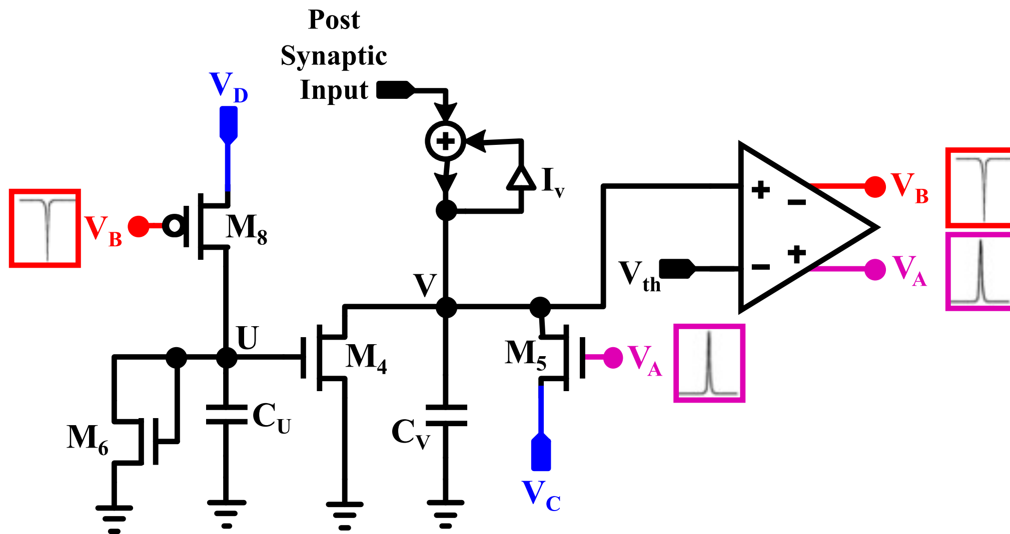
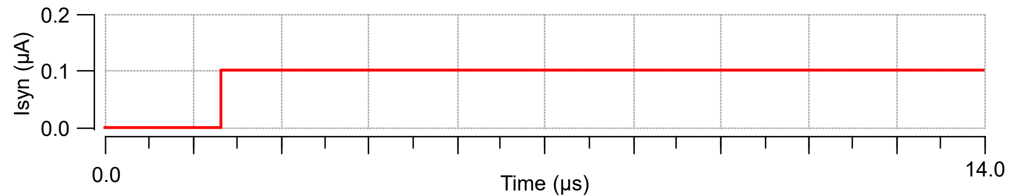


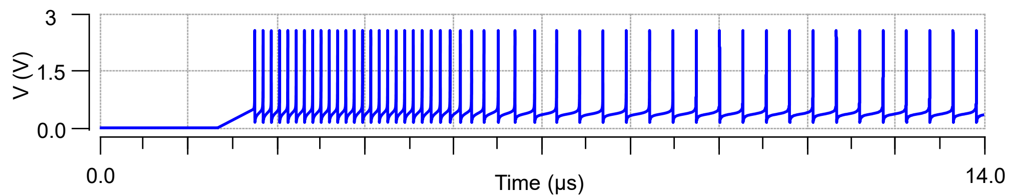
Figure 3.3: High-Level Model of the Spiking Neuron Circuit

Pulse V_A activates transistor M_5, discharging capacitor C_V and hyperpolarizing V to a predetermined value V_c. Pulse V_B turns on transistor M_8, which has a narrow channel, so that only a small amount of charge from V_d passes to capacitor C_U. The two capacitors are sized so that C_U charges more slowly than C_V. With every spike, voltage on C_U increases a little, thus increasing the leakage current through M_4 and the current through M_6, which is diode-connected to act like a non-linear resistor that discharges C_U. Leakage slows down the charging of C_V, allowing the refractory period between spikes and enabling the adaptation of the output spike train. By varying control voltages V_c and V_d, the relative speeds of V and U can be controlled and the four basic firing patterns can be obtained.

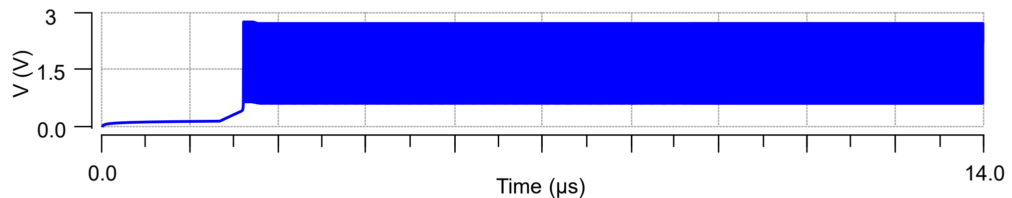
The neuron circuit is designed in the AMS 0.35 µm HV CMOS H35B4D3 technology. The input stimulus is a constant current pulse of a relatively long duration, and the output of the circuit is a pulse train whose pattern is determined by the values of the control voltages V_c and V_d. The 4 patterns are shown in Fig. 3.4 using combinations of V_c and V_d shown in Fig. 3.5 as diamond points. Fig. 3.5 also shows approximate areas in the V_c-V_d space that produce each firing pattern.



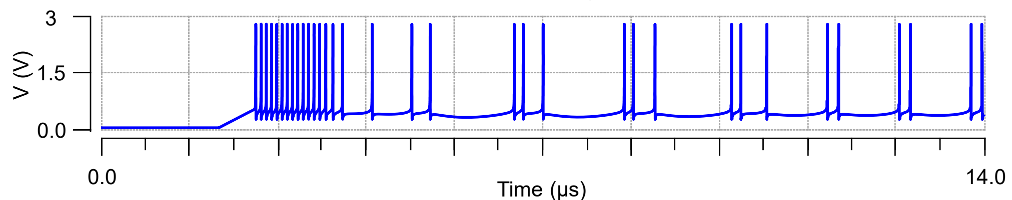
(a) Post-synaptic input current stimulus.



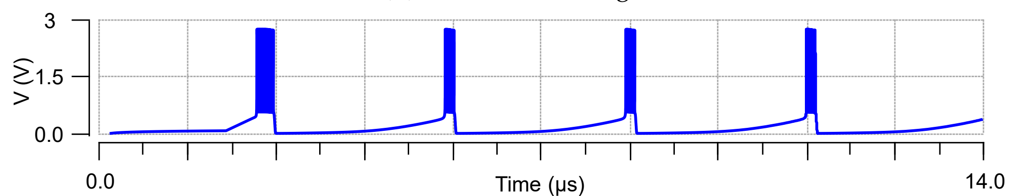
(b) Regular Spiking.



(c) Fast Spiking.



(d) Intrinsic Bursting.



(e) Chattering.

Figure 3.4: Output Spike Train Patterns of Neuron.

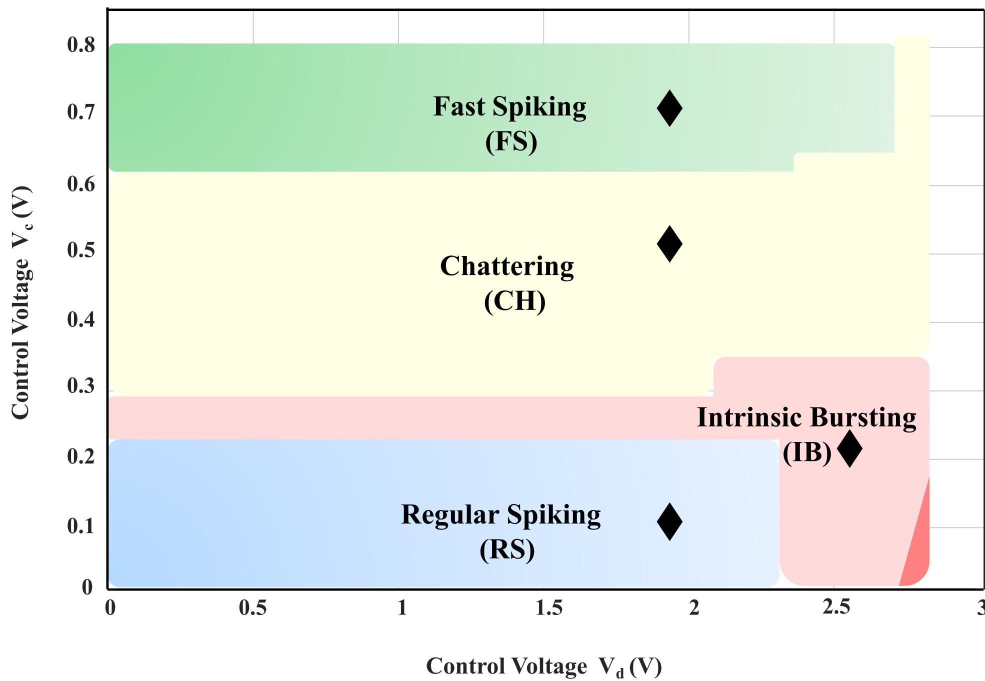


Figure 3.5: Approximate areas in the control voltages space V_c-V_d that produce the different firing patterns. The diamond points correspond to the nominal control voltages combinations used to produce each firing pattern.

3.3 THE BUILT-IN SELF-TEST

3.3.1 *BIST Architecture*

The proposed BIST architecture for the spiking analog neuron is illustrated in Fig. 3.6. The BIST wrapper includes a ramp generator block that applies ramps at the two control voltages of the neuron aiming to excite the neuron across its different operation modes and produce the four firing patterns. Voltage V_d is ramped from 1.9 V to 2.8 V, and during this duration V_c is ramped once from 0.1 V to full scale and then ramped again from 0.1 V to half the full scale, as shown with the saw-tooth stimulus in Fig. 3.6. These values were chosen based on the approximated firing areas in Fig. 3.5. A healthy neuron should produce the RS, CH, and FS firing patterns during the first full V_c ramp, and the IB firing pattern during the second half V_c ramp. The ramps do not have any stringent requirements, hence low-precision stepwise ramp generators can be used in this context.

The following step would be a digital block connected to the output of the neuron to digitize the analog output, store the digital signature, and try to identify within this digital signature the four firing patterns, i.e., to match excerpts of the digital signature to the desired firing pat-

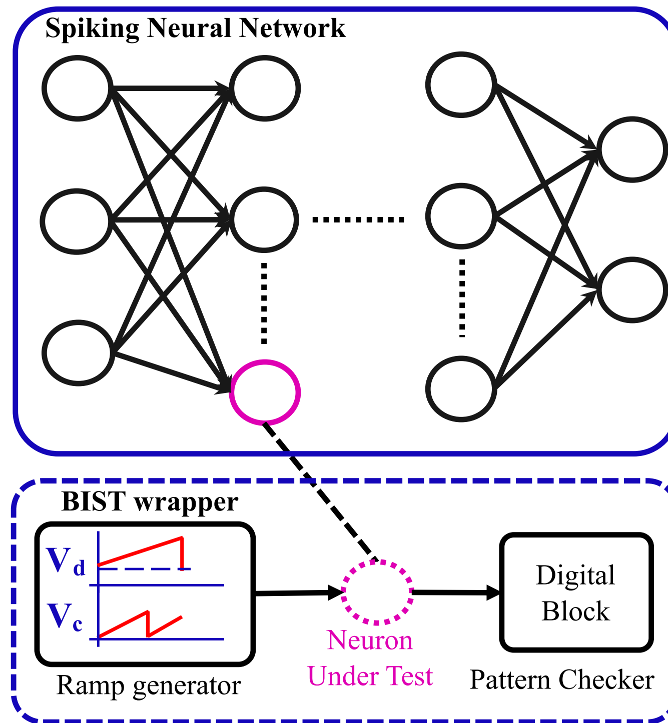


Figure 3.6: BIST Architecture.

terns. The assumption here is that a functional neuron should be able to generate all intended firing patterns. If one or more firing patterns are missing, the neuron is declared faulty. In essence, the proposed BIST architecture is behaviour-oriented, i.e., it targets verifying one of the functional specifications of the neuron, which is its ability to provide all firing patterns.

There are various advantages to this BIST architecture. First, a single BIST wrapper is sufficient to test the complete spiking neural network where the neurons connected sequentially to the BIST wrapper. Moreover, the proposed BIST architecture tests the neurons themselves independently of the application and of the data that is processed through the neural network for training or inference. This dissociates the test procedure from the underlying training algorithm and the cognitive task that the neural network is performing. In other words, this BIST architecture is suitable for versatile use as it looks solely at the hardware.

The proposed BIST can be used for both post-manufacturing testing to catch manufacturing faults, and self-testing in idle times to catch faults that occur in the field that may affect the network inference. Applying the BIST on-line can help detect failures and trigger error-correction actions, i.e., neutralizing the faulty neuron and replacing it with a fresh spare neuron, and then attempting to retrain.

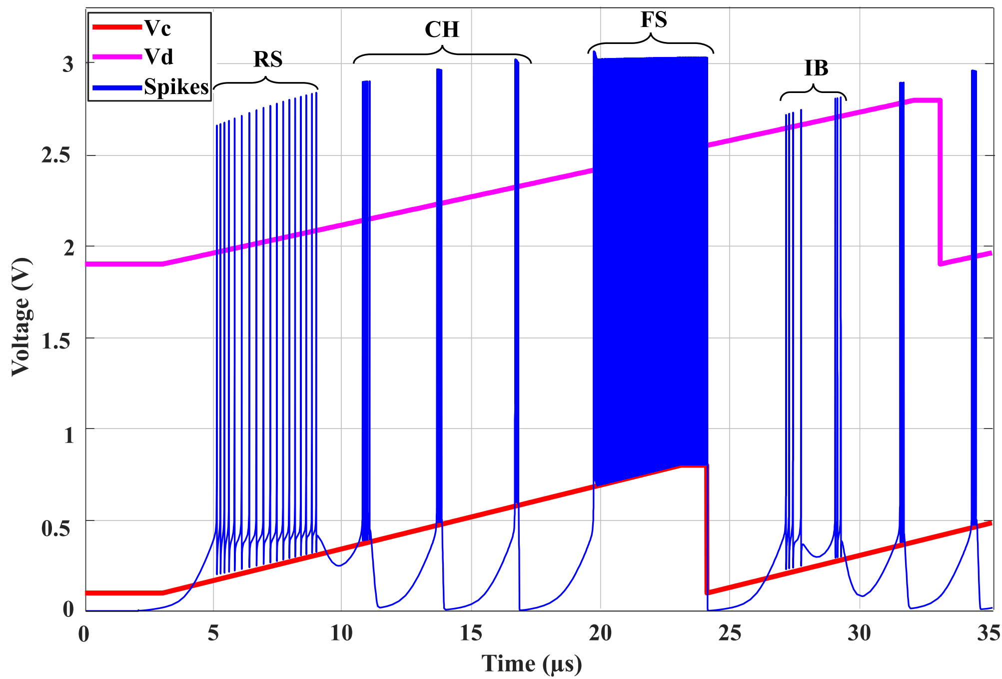


Figure 3.7: Neuron Output Response to BIST Stimuli.

3.3.2 *Expected BIST Response*

Fig. 3.7 shows the simulation of the functional neuron by applying the BIST stimuli, i.e., by ramping the two control voltages. As expected, the four firing patterns appear at the output. Specifically, RS appears first between 5 and 10 µs, CH follows between 10 and 20 µs. FS comes next between 20 and 25 µs, and IB appears last between 25 and 30 µs. A stimulus time of few µseconds is enough to excite the neuron in all four operation modes, thus the test is very fast.

3.3.3 *BIST Verification*

The proposed BIST principle is that a faulty neuron loses its ability to produce one or more firing patterns. A neuron could be faulty due to process variations or due to physical defects, i.e., random spots or voids on the die surface that may occur due to errors during the manufacturing steps and that translate into short- and open-circuits or extreme variation. In the field, a neuron can become faulty due to aging (time-dependent dielectric breakdown, hot carrier injection, etc.), and latent defects that go undetected at time-zero during manufacturing testing and manifest themselves later in the field of operation. They could also be provoked by environmental stress, i.e., humidity, temperature, etc. However, both aging and latent defects translate in the end to process variations or defects.

For our experiments, we verified our test viability for both causes. Neurons with process variations were generated by performing a Monte Carlo (MC) analysis with mismatch and inter-die variations using the statistical Process Design Kit (PDK) of the technology. Physical defects, on the other hand, were manually introduced into the circuit since the design is compact enough, comprising only 14 transistors and 2 capacitors. Defective neurons were generated by assuming a classical defect-modeling approach [116]. We consider 2 faults per transistor; a drain-to-source short and an open-gate. Drain-to-source shorts are straight forward. Open-gate defects are introduced following the simulation method in [117], where instead of an ideal open circuit which cannot be handled by the simulator, a gate-open is implemented with a weak pull-up or pull-down gate voltage. More specifically, the gate-to-source voltage is controlled by the drain-to-source voltage with a gain coefficient set to a default value of 0.5. Considering additional shorts across other terminals and opens in other terminals are shown to be redundant in practice and only increase defect simulation time. As for the capacitors, we consider 3 types of faults; short- and open-circuits and a ±50% variation.

3.4 RESULTS AND DISCUSSION

Using the proposed BIST setup, we performed 36 physical-defect simulations and a MC analysis with 100 runs. Fig. 3.8 shows the response of the neurons in the first 5 runs of the MC analysis. In this excerpt, it is evident that neurons 1, 3 and 4 are behaving correctly, i.e., the four firing patterns appear in the output response. On the other hand, neurons 2 and 5 are clearly not producing all four firing patterns, thus they are detected by the BIST and considered faulty.

In total, the test yield by the BIST is around 70%, showing that analog neuron circuits can suffer a lot from process variations, thus requiring a thorough and comprehensive post-manufacturing test procedure.

As for the physical defects, the circuit is simulated with only one defect manually injected at a time. As expected, physical defects in such a compact analog design are very significant. All transistor faults and severe capacitor faults resulted in a failure in the neuron functionality. The proposed BIST approach was capable of achieving 90.6% defect coverage. The escaped defects not detected by the BIST were the ±50% variation in C_U and the +50% variation in C_V. Since these capacitor values control the speed of the neuron response, i.e. the rate of charging and discharging, a 50% variation proved to be tolerable and the neuron can eventually be tuned to accommodate these variations.

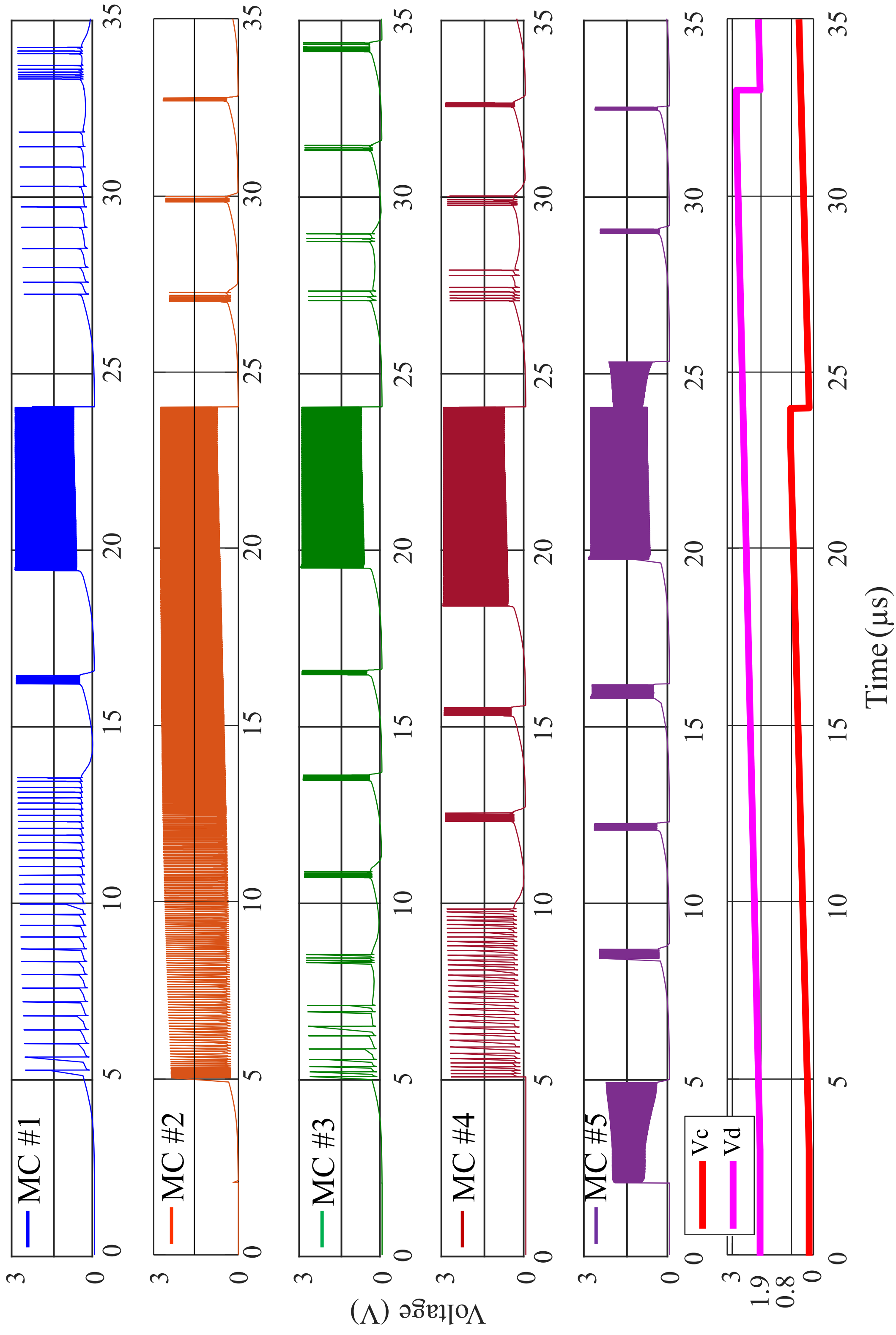


Figure 3.8: Monte Carlo Analysis Showing the Neuron Output Response to BIST Stimuli for 5 Runs.

4 HARDWARE-LEVEL FAULT MODELING

In the previous chapter, we applied a behaviour-oriented self-test that targeted the failure of the circuit to meet the required functionality. Nevertheless, this test is still circuit specific and needs to be adapted for every neuron. An important step towards global reliable-design solutions of AI hardware accelerators is understanding the effect of hardware-level faults on the performance. However, transistor-level fault simulations on a large-scale prove to be time and resource consuming, and hence, a need arises for a higher-level abstraction. In this chapter, we investigate the effects of transistor-level faults on the performance of an analog neuron [118]. We follow a bottom-up approach starting from transistor-level simulations to develop a neuron behavioral-level fault model that can be readily employed for performing behavioral-level fault simulation of deep SNNs.

4.1 FAULT SIMULATION FRAMEWORK

To serve our purpose of building a taxonomy of neuron faulty behaviours, and eventually a behavioral-level fault model for the neuron, we perform:

1. Monte Carlo simulation with 1000 runs using the technology PDK, considering both global and local process variations to model the soft faults.

2. Structural defect-oriented simulation in an automated workflow using the mixed-signal defect simulator Tessent®DefectSim by Mentor®, A Siemens Business [116].

In analog VLSI circuits, hard faults refer to physical defects caused by foreign particles on the wafer surface, wafer mishandling (e.g., scratching, and over- or under-itching), mask misalignment, etc. Soft faults, on the other hand, happen due to the inherent variability of the VLSI manufacturing process, e.g., local geometric deformations (i.e., variation in the

effective channel length and width), doping concentration variations, etc. To enable the efficient simulation, detection, and effective mitigation of these faults, they need to be modeled into the transistor-level. The fault model should be able to sufficiently quantify the dominant faults that affect the circuit.

We consider a standard defect model for the transistors that includes stuck-on and stuck-off behaviors. Stuck-on is modeled with a short-circuit across the drain and short terminals implemented with a default small resistance of 10Ω. Stuck-off is modeled with an open-circuit at the gate terminal. Since the simulator cannot handle ideal opens and since a very high series-resistance would have no effect, a gate open is implemented with a weak pull-up or pull-down gate voltage. In particular, the gate-to-source voltage is controlled by the drain-to-source voltage with a gain coefficient set to a default value of 0.5 [117]. Finally, for passive elements, i.e., resistors and capacitors, the defect model includes large variations of $\pm 50\%$. For our neuron, the defect model size is $N_{defects} = 46$.

4.2 THE SPIKING NEURON

4.2.1 *Behavioral Model*

The I&F model is nowadays one of the most dominant and widely used for describing spiking neurons [119]. This is because it offers enough complexity to capture the characteristics of biological neural processing, along with the capability of simple mathematical analysis and intuitive understanding of the model dynamics.

The I&F model explains the dynamics of a neuron through its membrane potential, V_m:

$$C_m \cdot \frac{dV_m}{dt} = I_{syn} + I_{inj}, \tag{4.1}$$

where C_m is the membrane capacitance, I_{syn} is the post-synaptic current fed to the neuron, and I_{inj} is the current injected into the neuron either externally or through a positive feedback path.

The simplicity of the I&F model comes from the separation of the sub-threshold integration dynamics from the spike generation mechanism. Because a spike is a momentary surge in voltage, and hence a stereotyped event whose form holds no information, spike generation is not regarded as an intrinsic part of the model and is not formally stated. Instead, the model usually focuses on the evolution of the sub-threshold membrane potential and then add the spike generation. This distinction allows for

better understanding of the information processing capabilities of the neurons [119].

The spike generation behavior is characterized by a firing time t^f and a threshold criterion, i.e., the neuron produces a spike at time t^f when V_m reaches the threshold value, V_{ref}:

$$t^f : V_m(t^f) = V_{ref}. \tag{4.2}$$

As soon as the neuron fires, the membrane potential is reset to a value $V_{reset} < V_{ref}$:

$$\lim_{t \to t^f; t > t^f} V_m(t) = V_{reset}. \tag{4.3}$$

For $t > t^f$, the neuron dynamics again follow Eq. (4.1) until the next time V_m reaches V_{ref}.

4.2.2 *Transistor-Level Design*

Fig. 4.1 shows the transistor-level design of the I&F neuron used in this work. It is designed in the AMS 0.35μm technology and was originally part of a neuromorphic cortical-layer processing chip for spike-based processing systems [120].

The neuron takes the input current spikes I_{syn} coming from the synapses, integrates them on capacitor C_m, and fires a spike at the output V_{spike} when the capacitor voltage V_m reaches a certain threshold V_{ref}. The circuit has an extra set of input/output nodes, namely the $\overline{Ack}$ and *Rqst* nodes, which are used by the AER communication protocol.

The main blocks are a comparator and a set of inverters that control the signal flow. During the charging time of the capacitor the circuit is inactive, transistors M_{p1} and M_{n4} are off, and transistor M_{p2} is on. Since the comparator is constantly following V_m and comparing it to V_{ref}, its bias current is kept low through transistor M_{n1} to minimize power consumption. As V_m increases towards V_{ref}, node n_1 starts changing state and switches on two transistors: (i) transistor M_{p1}, which slowly introduces a positive feedback current that accelerates the charging of the capacitor, and (ii) transistor M_{n3} through node n_2, which offers a brief surge in the comparator bias current. Combined, these actions speed up the transition time of the comparator output.

Once the transition is complete, i.e., node n_1 is low and node n_2 is high, node n_3 goes low and an output request signal is sent to the AER communication block by pulling up line *Rqst*. After a few nanoseconds, the AER block acknowledges back the request and the $\overline{Ack}$ input pulls

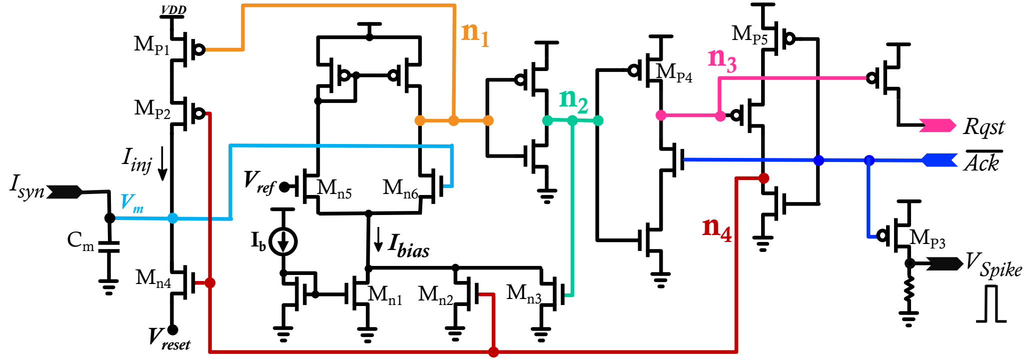


Figure 4.1: I&F Neuron

node n_4 up and turns on transistor M_{p3} which produces the output spike of the neuron. Node n_4 has three main effects on the neuron circuit: (i) it turns transistor M_{n2} on to keep the comparator bias current high during the back transitioning, (ii) it turns off transistor M_{p2} which cuts off the positive feedback path to the capacitor, and (iii) it turns transistor M_{n4} on to reset V_m to V_{reset} so the capacitor is able to charge again.

4.3 SPIKING NEURON FAULTY BEHAVIORS

To stimulate the neuron, an input current pulse of 10µs width was used, shown in Fig. 4.2a. In a fault-free scenario, the neuron should start spiking at regular intervals after the input stimulus begins and stop spiking once the input stimulus is over, as shown in Fig. 4.2b.

Fault simulation experiments revealed that there are two categories of faults in the neuron behavior that can result from a fault in the circuit. Some faults are catastrophic, i.e., the behavior of the neuron is faulty, and the neuron is no longer functional. Other faults are parametric, i.e., the output of the neuron deviates slightly from the nominal output, but the neuron produces an output spike train showing variations in timing parameters with respect to the nominal response. Catastrophic faulty behaviors were observed in 31 defect simulations, while parametric faulty behaviors were observed across the 1000 Monte Carlo runs and in the remaining 15 defect simulations.

4.3.1 *Catastrophic Faults*

We observe six different behaviors that fall under this category. These faults are considered fatal to the circuit operation. The neuron is not spiking correctly in response to the input stimulus, and it is considered defective. These faults are observed only as a result of physical defects in the circuit elements, and they are listed next, along with an example of

Table 4.1: Catastrophic faulty behaviors resulting from defect simulation.

Catastrophic faulty behavior	Number of defect simulations producing it ($N_{defects} = 46$)	Example neuron response
Saturated	5	Fig. 4.2c
Dead	12	Fig. 4.2d
Stuck-at-X	1	Fig. 4.2d
Stuck-at-1	10	Fig. 4.2e
Ghost-spike firing	1	Fig. 4.2f
Long-duration spike firing	2	Fig. 4.2g

a root-cause defect. Table 4.1 provides a summary of catastrophic faulty behaviors and shows the number of defects that produce them.

1. ***Saturated Output***
   A state where the neuron is constantly firing regardless of the presence of an input stimulus. Fig. 4.2c shows a saturated output caused by a stuck-on transistor M_{p1}. This defect triggers a constant high feedback current to the capacitor, so the capacitor is always charging even without a current from the synapse.

2. ***Dead Output***
   A state where the neuron output is stuck-at-0 when it should be spiking. The red curve in Fig. 4.2d shows a dead output caused by a stuck-on transistor M_{n4}. The capacitor cannot charge since it is constantly held at its reset value and, thereby, the neuron is incapable of spiking.

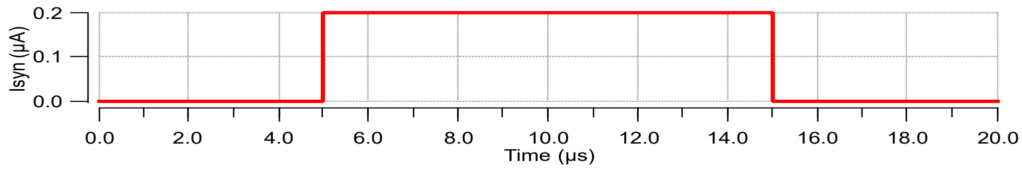
3. ***Stuck-at-X output***
   A state where the neuron output gets stuck at an arbitrary DC value between the supply voltage V_{dd} and ground. The blue curve in Fig. 4.2d shows such a faulty behavior caused by a stuck-off transistor M_{p3}. This defect isolates the neuron output from the $\overline{Ack}$ signal and, in the case of an ideal stuck-off, it turns the output node into a floating node which can settle to any DC value. Given our modeling of stuck-off transistor, the neuron node ends up settling at 1.1 V.

4. ***Stuck-at-1 output***
   A state where the neuron output gets stuck at V_{dd}. This faulty behavior can be produced even in the absence of an input stimulus, or it gets triggered once an input stimulus comes along. An example

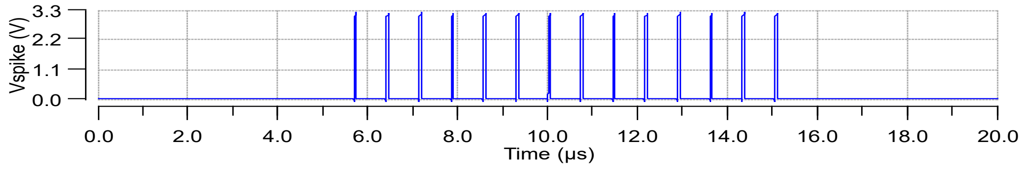
root-cause defect for the former case is a stuck-on transistor M_{n6}, as shown with the dotted red curve in Fig. 4.2e, while an example root-cause defect for the latter case is a stuck-off transistor M_{n5}, as shown with the blue curve in Fig. 4.2e. A stuck-on transistor M_{n6} forces node n_1 to be permanently low and, thereby, node n_2 to be permanently high. In the start-up, $\overline{Ack}$ is high, thus n_3 enables the *Rqst* and $\overline{Ack}$ goes low producing a spike. When $\overline{Ack}$ goes low, node n_3 is floating but retains its low value, thus $\overline{Ack}$ is permanently set low and the output gets stuck-at-1. On the other hand, a stuck-off transistor M_{n5} cuts off V_{ref} from the comparator input. According to our modeling of stuck-off transistor, the gate voltage of M_{n5} varies with time and is set equal to $V_{G,M_{n5}}(t) = (V_{D,M_{n5}}(t) + V_{S,M_{n5}}(t))/2$, where $V_{D,M_{n5}}(t)$ and $V_{S,M_{n5}}(t)$ are the drain and source voltages of M_{n5}, respectively. Initially the comparator output is high, and the capacitor keeps charging. At some point $t = t_s$, V_m exceeds $V_{G,M_{n5}}$ and the neuron eventually spikes. At the time of spiking $V_{G,M_{n5}}(t_s)$ is lower than V_{reset}, thus node n_1 gets permanently stuck at a low value and the output gets stuck-at-1 as explained above for the stuck-on transistor M_{n6}.

5. ***Ghost-spike firing***
A state where the neuron generates extra spike(s) that is(are) not a result of the membrane potential exceeding the reference voltage. We refer to these spikes as "ghost" spikes. Fig. 4.2f shows such a faulty behavior caused by a stuck-off transistor M_{p4}. When the neuron spikes, the path from node n_3 to ground gets cut-off. Node n_3 is floating since the defect isolates node n_3 from node n_2. Because of our defect model, node n_3 will eventually be weakly pulled up to V_{dd}. Simulations show that it first gets weakly pulled up to V_{dd} stopping spiking and then again it is weakly pulled down to ground producing a second ghost spike before it is finally stabilized bringing the neuron to its resting state.

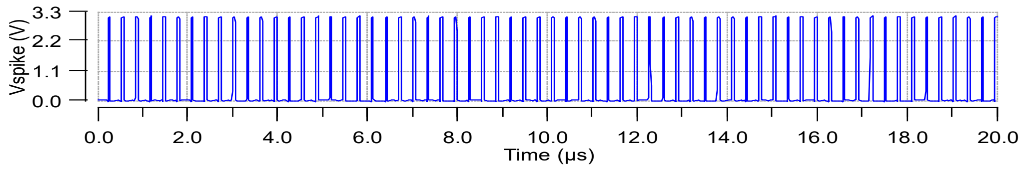
6. ***Long-duration spike firing***
A state where the neuron produces spikes of longer duration. Fig. 4.2g shows such a faulty behavior caused by a stuck-off transistor M_{p5}. When signal $\overline{Ack}$ goes low and the neuron spikes, node n_4 does not go immediately high to instantaneously reset the membrane potential and restart the integration. Instead, node n_4 is initially weakly pulled up to V_{dd} and gradually increases. As a result, the capacitor starts resetting but at a slow rate, thus extending the duration of the output spike.



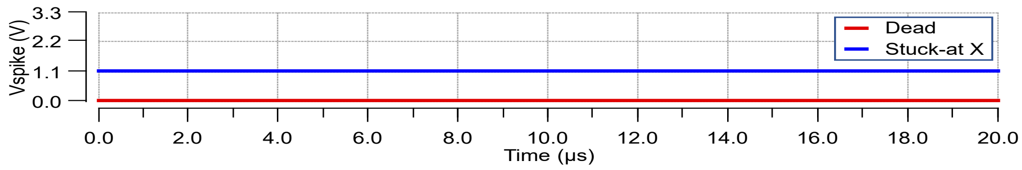
(a) Post-synaptic input current stimulus.



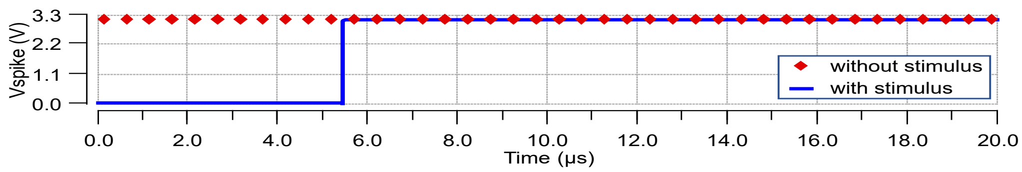
(b) Nominal neuron output.



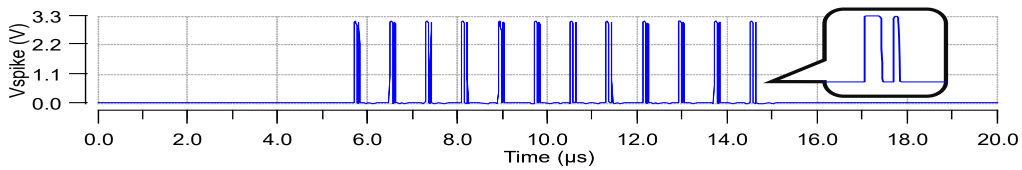
(c) Saturated output caused by a stuck-on M_{p1} transistor.



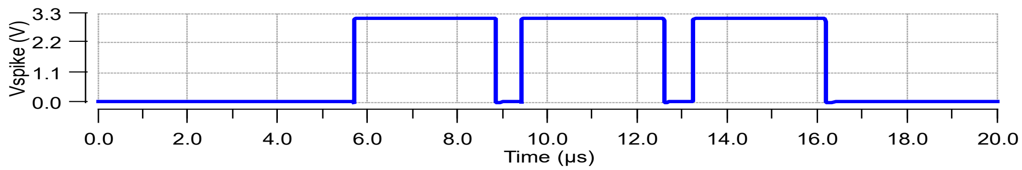
(d) Dead (or stuck-at-0) output caused by a stuck-on M_{n4} transistor and stuck-at-X output caused by a stuck-off M_{p3} transistor.



(e) Stuck-at-1 output caused by a stuck-on M_{n6} transistor (without requiring input stimulus) and by a stuck-off M_{n5} transistor (triggered with input stimulus).



(f) Output with ghost spikes caused by a stuck-off M_{p4} transistor.



(g) Long-duration spikes caused by a stuck-off M_{p5} transistor.

Figure 4.2: Examples of catastrophic faulty behaviors.

4.3.2 *Parametric Faults*

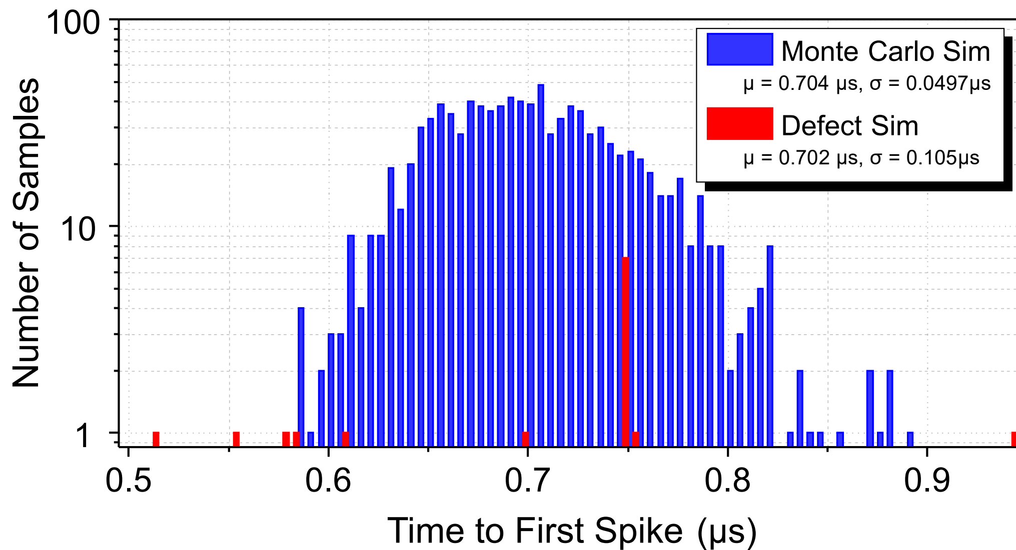
As for the parametric faulty behaviors, we consider two types of timing parameters, namely the time-to-first-spike and the firing rate. It should be noticed that such timing variations may not be problematic at network-level, i.e., they may be accommodated during training. Fig. 4.3 shows the histograms of the time-to-first-spike and the firing rates observed across the Monte Carlo runs and the defect simulations, excluding the simulations that led to a catastrophic faulty behavior as discussed above. Results suggest a correlation between the time it takes for the neuron to fire its first spike and the firing rate, i.e., a neuron that produces a first spike faster has a higher firing rate, and vice versa. This neuron is implemented with no adaptation mechanism and simulated with the initial condition for the capacitor voltage set equal to the reset value, hence the time-to-first-spike is equal to the inter-spike interval, which is the inverse of the firing frequency.

This data was collected from 1000 Monte Carlo runs and 15 defects that result in timing variations. As evident from the figure, the timing parameters of the circuit are very sensitive to process variations and mismatch. Time-to-first-spike values are distributed around a mean value of $0.702\mu s$ with a standard deviation of $0.1\mu s$, Fig. 4.3a. Similarly, firing frequencies are also normally distributed around a mean value of $1.38MHz$ and vary with a standard deviation of $94kHz$, Fig. 4.3b.

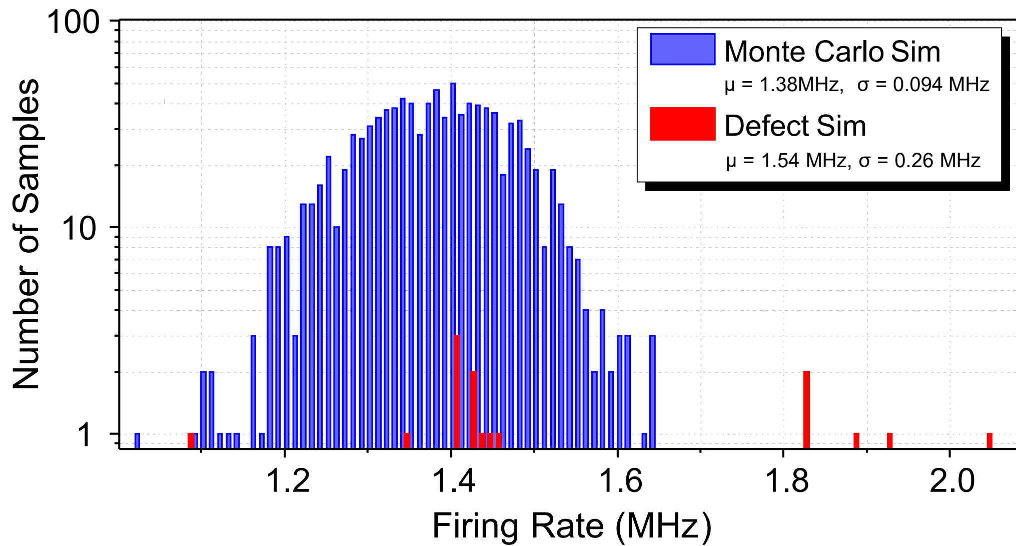
On the other hand, out of 46 physical defects, only 15 result in timing variations, as shown in red in both parts of Fig. 4.3. Some of these variations are barely noticeable, i.e., they cause a change in the time-to-spike that is so small that the firing frequency is not affected. An example is a stuck-off transistor M_{p2}. This transistor is on in the idle state of the circuit, and once the neuron spikes, it is responsible for cutting off the positive feedback path to the capacitor. When it is stuck-off, the feedback path is cut from the start and the capacitor charges only through the synaptic input. Since this feedback current is applied to accelerate the charging rate of the capacitor, its absence has very small effect on the actual circuit operation, and the neuron spikes with a frequency almost equal to the nominal value.

Other defects can lead to a clear change in the firing frequency, albeit without affecting the functionality of the neuron. For example, a 50% decrease in the membrane capacitance results in a similar decrease of the integration time constant, i.e., the charging speed of the capacitor. This entails that the capacitor reaches the reference voltage faster than the nominal case, thus producing an earlier first spike and by definition, spike at a higher firing rate of over 2 MHz. Other defects that can lead to an

apparent variation in the circuit timing are stuck-off defects in transistors M_{n1}, M_{n2} and M_{n3}. As explained in section 4.2.2, these three transistors form a dynamic biasing circuit for the comparator that momentarily change its bias current to control the transition rate. Consequently, when one of them gets stuck-off, the transition rate of the comparator is affected and ends up altering the firing rate of the neuron.



(a)



(b)

Figure 4.3: Histograms of timing variations.

4.4 BEHAVIORAL-LEVEL FAULT MODEL

Based on the experiments conducted in this work, we propose a fault model according to the different faulty behaviors observed in section 4.3. This model can be used to test spiking neurons based on the I&F model in a complete network.

At the behavioral level, these faults can be recreated depending on their types. Parametric faulty behaviors are emulated by manipulating the model parameters described in section 4.2.1. For example, timing variations can be represented as changes in either the membrane capacitance C_m in Eq. (4.1), the reference voltage V_{ref} in Eq. (4.2) or the reset voltage V_{reset} in Eq. (4.3). Catastrophic faults on the other hand are modeled a little differently. For example, a dead output or an output that is stuck-at-1 or stuck-at-X can be simulated just by forcing the neuron output to take the respective value. A saturated output is obtained by forcing a high constant input current applied form the start, so the neuron is spiking all the time. A delay in the resetting mechanism of the neuron would produce output spikes with long duration. Finally, the ghost-spike firing can be recreated by simply adding ghost spikes to the nominal spike train after decreasing their widths.

5 FAULT INJECTION AND RESILIENCY ANALYSIS IN SPIKING NEURAL NETWORKS

Now that we have investigated the effects of hardware faults on a single neuron circuit, it was time to look at the bigger picture. In this chapter, we translate the hardware-level neuron faults into an abstract behavioral-level fault model, consistent with manufacturing defects, that can be used to perform fault-injection experiments for deep SNNs [121].

To demonstrate, we design two deep convolutional spiking neural networks that perform the classification of the N-MNIST dataset [63], which is a neuromorphic version of the MNIST dataset [61], and IBM's DVS-gesture dataset [64]. We propose a framework for accelerating fault injection for deep SNNs, and we perform a large-scale fault injection experiment to grade the criticality of different faults for the cognitive task.

Fault criticality is investigated across the different components, i.e., neurons and synapses, across the layers, as well as within-layers and synaptic matrices depending on the location of the affected component. Our experiment shows that certain fault types and fault locations can drastically reduce the SNN performance. This type of experiment helps assess reliability, pinpoints the reliability-critical parts in the architecture, and offers valuable insights and guidelines for developing hardware-level self-test and error-tolerance techniques optimized for low overhead.

5.1 FAULT MODELS

As discussed in Chapter 4, fault simulations at transistor-level for a large-scale SNN can be tedious and demanding in terms of time and resources. However, to get a comprehensive look into the way hardware faults influence SNNs and to what extent they disrupt their intended tasks, it is inevitable to perform some sort of fault injection experiments. In this part of the work, we propose a taxonomy of faults for neurons and synapses modeled at behavioral level. In other words, we model faults as variations

or errors in behavioral parameters of the neurons and synapses without the need to specify their root-causes, which can be process variations, defects, aging, transient noise, voltage variations, temperature variations, etc. In this way, the fault model becomes independent of the hardware implementation and independent of the learning algorithm or the data processed through the network. This classification paves the way for an abstract technique to perform fault injection campaigns for deep SNNs in which we treat the network as a distributed system where neurons and synapses are discrete entities that can fail independently.

5.1.1 *Neuron Fault Models*

Starting off with neurons, we use the neuron fault model extracted from fault simulations in Chapter 4 to define three main fault types according to their effect on the network. The first two fault types explicitly act on the output spike train, while the third type acts on internal parameters of the neuron and implicitly affect the output spike train.

1. ***Dead Neuron Fault:*** A fault in the neuron that leads to a halt in its computations and a zero-spike output.
2. ***Saturated Neuron Fault:*** A fault that causes the neuron to be firing all the time, even without any external stimuli.
3. ***Timing-Variation Fault:*** A fault that results in timing variations in the output spike train, i.e., time-to-first-spike or the firing rate. Many parametric faults can give rise to timing variations in a spiking neuron, for example:
    a) ***Integration Fault:*** A fault that affects the integration process of incoming spikes. This fault changes the response time of the neuron and can subsequently influence the rate at which the neuron spikes.
    b) ***Threshold Perturbation Fault:*** A fault in the value of the threshold at which the neuron spikes, which can change the frequency of spiking or eventually cause the neuron to be stuck either at a saturated or a dead state.
    c) ***Refractory Period Fault:*** A fault that can influence the refractory mechanism of a spiking neuron and eventually restrain the output of the neuron or completely stop it from firing.

Other faulty behaviors that were observed from the previous experiment in Chapter 4 were considered specific to the spiking neuron circuit used for the experiment. Hence, they were not considered at the behavioral level.

5.1.2 *Synapse Fault Models*

As discussed in Chapter 2, synapses represent the second building block of neural networks. Synapses transfer signals from neurons in layer l to neurons in layer l+1, and they determine the significance of each signal on the recipient neuron through the weights. Building on this, weight errors can be used to model faults in the synapses. Here we define two synaptic fault types:

1. ***Dead Synapse:*** A fault interpreted as a cut in the synapse circuit that obstructs the transmission of signal from neuron i in layer l to neuron j in layer l+1.

2. ***Saturated Synapse:*** A fault that causes the synapse to transfer the signal from neuron i in layer l to neuron j in layer l+1 with a weight of a relatively high value, be it positive or negative. In the implementation of the SNN used in this work, weight values are unconstrained, adding a level of freedom in the learning process since weights can take any value in the positive or the negative range. A saturation fault can push this weight towards the positive or negative maximum.

5.2 CASE STUDIES

To actually model the faults defined before, we needed a spiking neural network that we can experiment with. There exist many frameworks for facilitating the behavioral modeling and simulation of ANNs, such as TensorFlow [122], Keras [123], and PyTorch [124]. Each framework implements its own approach for solving AI problems while providing different trade-offs between simplicity and modularity. However, none of these frameworks have integrated support for SNNs. To this end, we designed two deep convolutional SNNs for the classification of the N-MNIST and IBM's DVS-gesture datasets. Both SNNs are modeled using primitives in the open-source Spike LAYer Error Reassignment (SLAYER) framework [68], which is a framework completely built on PyTorch, inheriting all its benefits and capabilities. The networks are trained using batch learning with a variation of the back-propagation algorithm, and the winning class is selected by observing the neuron which is triggered the most, i.e., produces the highest number of spikes.

The SLAYER framework uses the Spike Response Model [34] (SRM) to model spiking neurons in the network. Discussed in Chapter 2; the SRM is a generalized form of the ubiquitous I&F model. The model equations

and parameters are essential for the fault-simulation experiment, and they are presented next.

5.2.1 *The Spike Response Model*

In the SRM, the state of the neuron at any given time is determined by the value of its membrane potential, $u(t)$, which must reach a certain threshold value, ϑ, for the neuron to produce an output spike. The membrane potential of a neuron j in layer l is calculated as:

$$u_j^l(t) = \sum_i \omega_{i,j}^{l-1,l}(\varepsilon * s_i^{l-1})(t) + (\nu * s_j^l)(t) \tag{5.1}$$

where $s_i^{l-1}(t)$ is the pre-synaptic spike train coming from neuron i in the previous layer $l-1$, $s_j^l(t)$ is the output spike train of the neuron, $\omega_{i,j}^{l-1,l}$ is the synaptic weight between the neuron and the neuron i in the previous layer $l-1$, $\varepsilon(t)$ is the *synaptic kernel*, and $\nu(t)$ is the *refractory kernel*.

In Eq. (5.1), the spiking action of the neuron is described in terms of the neuron's response to the input pre-synaptic spike train and the neuron's own output spikes. The incoming spikes by the neurons in the previous layer are scaled by their respective synaptic weights and fed into the post-synaptic neuron. The response of the neuron to the input spikes is defined by the synaptic kernel $\varepsilon(t)$ which distributes the effect of the most recent incoming spikes on future output spike values, hence introducing temporal dependency. Theoretically speaking, $\varepsilon(t)$ can take many forms according to the performed experiments [34]. For our experiments, we use the form [68]:

$$\varepsilon(t) = \frac{t}{\tau_s} \cdot e^{(1-\frac{t}{\tau_s})} \cdot H(t) \tag{5.2}$$

where $H(t)$ is the unit step function and τ_s is the time constant of the synaptic kernel. The second term in Eq. (5.1) incorporates the refractory effect of the neuron's own output spike train onto its membrane potential through the refractory kernel. The form used here is:

$$\nu(t) = -2\vartheta \frac{t}{\tau_{ref}} \cdot e^{(1-\frac{t}{\tau_{ref}})} \cdot H(t) \tag{5.3}$$

where τ_{ref} is the time constant of the refractory kernel.

When $u(t) > \vartheta$ the neuron fires a spike, e.g., $s_j^l(t) = 1$. If $u(t) < \vartheta$, then the neuron remains silent, e.g., $s_j^l(t) = 0$.

Figure 5.1: Samples of N-MNIST saccades snapshots in 2-D format. Black regions indicate no events, red indicates ON events, and blue indicates OFF events.

5.2.2 *Case Study (1): The N-MNIST SNN*

5.2.2.1 *The N-MNIST Dataset*

The N-MNIST, or the Neuromorphic-MNIST dataset [63], is a spiking, version of the famous MNIST dataset. The MNIST (Modified National Institute of Standards and Technology) dataset [61] is a large collection of handwritten digits that has become a benchmark dataset used for training and testing image processing systems. It consists of 70,000 grey-scale images that were normalized to fit a 28x28 pixel size.

The N-MNIST dataset also comprises 70,000 sample images generated from the saccadic motion of a DVS in front of the original images of the MNIST dataset. The motion of the sensor generates two types of events: a pixel switching from low brightness to high brightness and vice versa. As a result, the images have a third dimension of 2 channels to carry information about the events sign. Fig. 5.1 shows three snapshots of different sample saccades. The movement of DVS camera in front of the images leads to an increase in their dimensions both horizontally and vertically, hence the resulting images are of size 34x34x2. In addition, the samples in the N-MNIST dataset are not static; each sample has duration of 300 ms.

5.2.2.2 *The N-MNIST SNN Architecture*

The SNN architecture is inspired from the LeNet-5 network [125] and is shown in Fig. 5.2. It is composed of 3 convolution layers, SC1, SC2 and SC3, that extract features from the dataset images, and 2 fully connected ones, SF4 and SF5, that perform the classification of the derived features. Progressing through the layers, the number of kernels increase from 6 in layer SC1 to 120 in layer SC3 that is then flattened into a 1x1 data

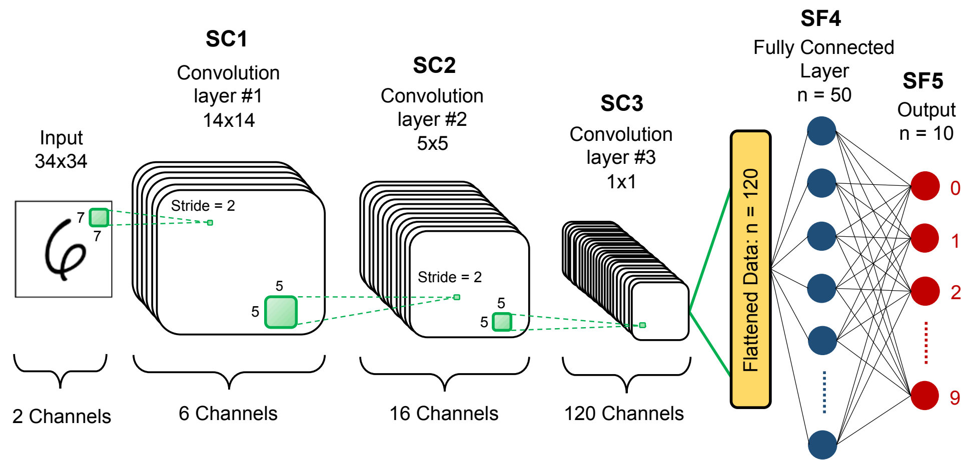


Figure 5.2: Architecture of the SNN for the N-MNIST dataset.

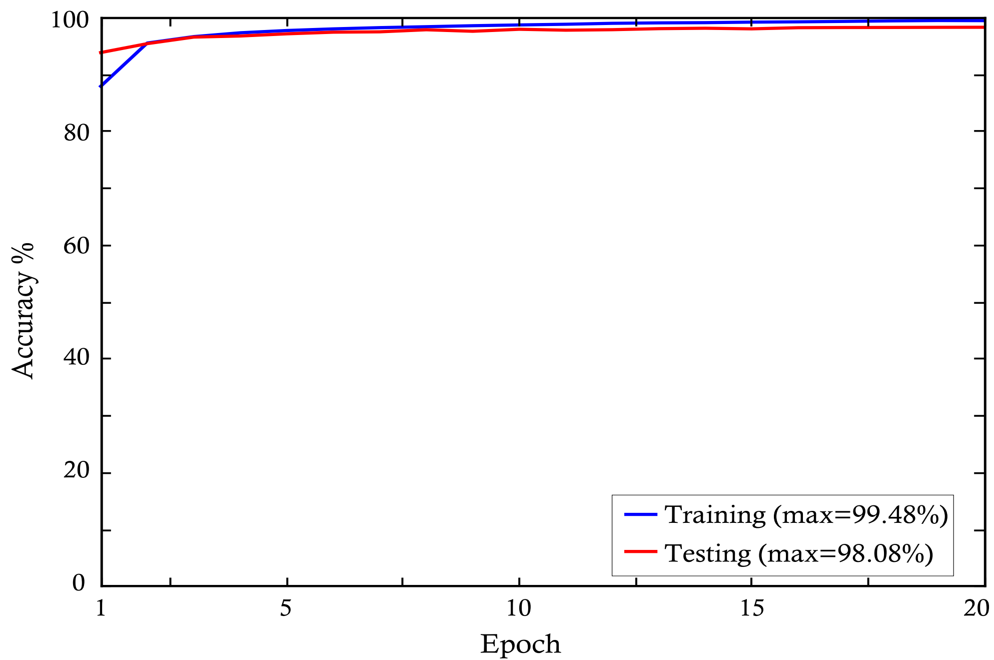


Figure 5.3: Learning Curve of the N-MNIST SNN.

representation. A stride of 2 was used as the aggregation technique instead of pooling layers, i.e., the image dimensions are halved after every convolution layer. Layers SF4 comprises 50 neurons, and the output layer SF5 comprises 10 neurons corresponding to the digit classes from 0 to 9.

To train the network, the dataset is split into a training set of 60,000 samples and a testing set of 10,000 samples. The network achieved a classification accuracy of 98.08% on the testing set, which is comparable to the performance of state-of-the-art level based DNNs. Fig. 5.3 shows the learning curve for 20 epochs with a size-12 batch learning.

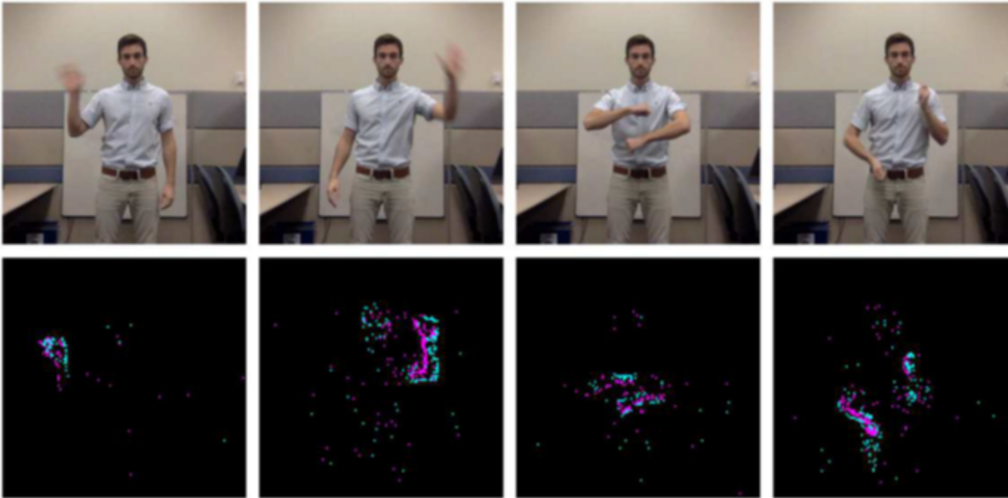


Figure 5.4: **TOP:** a video frame of an individual performing a few gestures from the dataset, from left to right: right-hand wave, left-hand wave, arm roll and air drums. **Bottom:** A 2-D representation of the same video frame above but in spiking form [64].

5.2.3 *Case Study (2): The DVS-gesture SNN*

5.2.3.1 *The DVS-gesture Dataset*

Contrary to the N-MNIST dataset that was converted from frame-based to event-based by a DVS camera, IBM's DVS-gesture dataset [64] was directly created in spiking form for event-based systems. The dataset consists of 1,342 samples that represent 11 hand and arm gestures grouped in 122 trials. The gestures are performed by 29 individuals standing against a still background in front of a 128x128 dynamic vision sensor, DVS128. Each individual performs each gesture under 3 different lighting conditions for approximately 6 seconds. Fig. 5.4 shows a few samples of the DVS-gesture dataset in addition to a picture of the actual person performing the gesture in real time.

Generally speaking, hand gestures provide a means for communication that is independent of language, age, or culture. They are indispensable as well in applications requiring human-computer interaction, such as sign-language recognition and gaming. Real-time hand-gesture recognition is considered a realistic problem that is well equipped for event-based systems. They require low-latency responses that cannot be accommodated by a frame-based system.

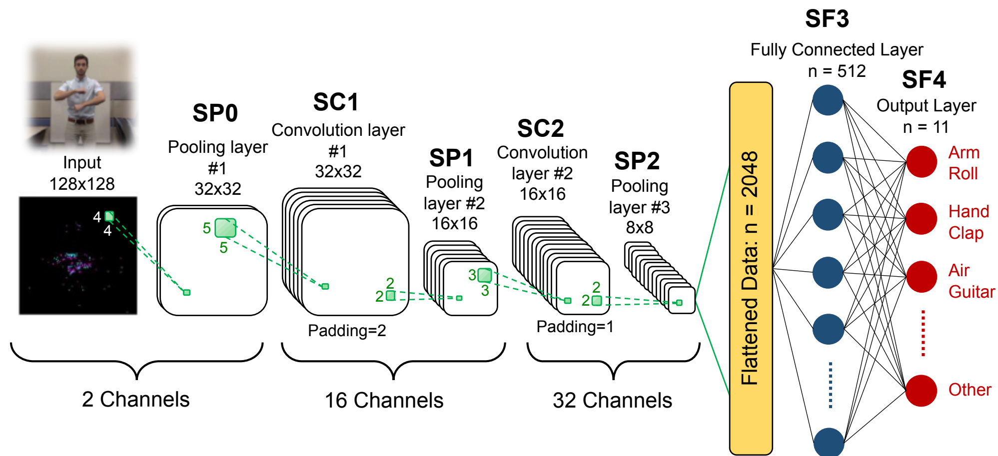


Figure 5.5: Architecture of the SNN for the DVS-gesture dataset.

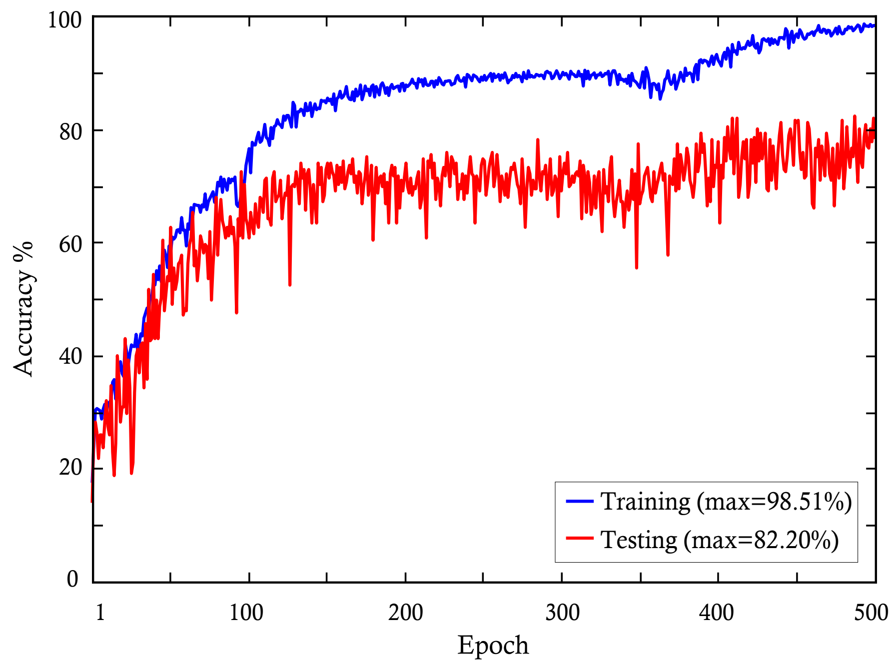


Figure 5.6: Learning Curve of the DVS-gesture SNN.

5.2.3.2 *The DVS-gesture SNN Architecture*

The architecture that we used is proposed in [68] and is shown in Fig. 5.5. the network consists of 2 convolutional layers, SC1 and SC2, that extract features, 3 pooling layers, SP0, SP1 and SP2, that compress the data, and 2 fully connected layers, SF1 and SF2, that perform the classification. The number of kernels increase from 2 at the input layer to 32 at the last pooling layer. The data is then flattened to a 2084 1x1 representation. The last hidden layer has 512 neurons and the output layer has 11 neurons, each classifying one gesture.

To train the network, samples from the first 23 subjects are used for training and samples from the last 6 subjects are used for testing. Due to computation limitations of the neuromorphic simulation, we used a trimmed version of the dataset samples of 1.5s. The network performs with an 82.2% accuracy on the testing set, which is acceptable considering the shortened samples of the dataset and the shallower architecture than of the one proposed in the original work [64]. The learning curves are shown in Fig. 5.6.

5.3 FAULT MODELING & INJECTION METHODOLOGY

The SLAYER framework used to design the networks in our case studies does not inherently support the injection of faults in the SNN. Therefore, to implement the fault injection and failure analysis necessary for our work, we had to figure out how to introduce these faults into the behavioral model, and then customize the flow of computations both at the spiking part and at the core level of PyTorch.

The hardware fault models described in Section 5.1 are introduced into the code as follows:

- **Neuron Faults**

  1. ***Dead Neuron Fault:*** A dead neuron is modeled by forcing the output spike train of the neuron to be always low.

  2. ***Saturated Neuron Fault:*** Modeled by skipping the computations and forcing constant output spiking activity at the neuron in question.

  3. ***Timing-Variation Fault:*** We refer to them as parametric faults, since they are modeled by manipulating the parameters of the neuron model in Section 5.2.1 as:

     a) ***Integration Fault:*** Modeled as a deviation in the value of τ_s in Eq. (5.2).

     b) ***Threshold Perturbation Fault:*** Modeled as a perturbation in the value of ϑ.

     c) ***Refractory Period Fault:*** This fault is modeled as a variation in the value of τ_{ref} in Eq. (5.3).

- **Synapse Faults**

  1. ***Dead Synapse:*** This fault is simply modeled as a zero weight, i.e., giving a zero value to $\omega_{i,j}$ in Eq. (4.1).

  2. ***Saturated Synapse:*** We consider both positive and negative saturation for this fault, modeled by assuming a relatively large absolute value of the weight, i.e., $\omega_{i,j}$ in Eq. (4.1).

Fig. 5.7 illustrates how neuron faults are injected in layer l and synapse faults are injected in the synaptic matrix connecting layer l with layer $l+1$. In particular, if neuron n_x in layer l is dead, then its spike train output calculation is bypassed and its output is forced permanently to 0, e.g., $s_x^l = 0$. If neuron n_y in layer l is saturated, then its spike train output calculation is bypassed and its output is forced permanently to 1, e.g., $s_y^l = 1$. For a fault in the synapse connecting neuron i in layer l to neuron j in layer $l+1$, we modify the synapse weight, e.g., $\omega_{i,j}^{l,l+1} = \bar{\omega}_{i,j}^{l,l+1}$, where $\bar{\omega}_{i,j}^{l,l+1}$ is 0 for a dead synapse fault, has a relatively high positive value for a positive saturation fault, or has a relatively low negative value for a negative saturation fault.

As for timing-variation faults in neurons, that we refer to as parametric faults, a simple modification of the parameters of a single neuron is not possible because these parameters are set at the beginning of the simulation and shared among all neurons in the network. Our approach, as illustrated in Fig. 5.7, is to create a dummy faulty layer (dfl) identical to layer l with the exception that all neurons have the target parametric fault. The neurons in dfl are driven by the incoming spike trains from the neurons in layer $l-1$. Then, the output spike train of the neuron n_z in layer l where the parametric fault is to be injected is replaced by the output spike train of the corresponding neuron $n_z^{'}$ in dfl, e.g., $s_z^l = s_z^{dfl}$.

Throughout most of our experiments, unless stated otherwise, we consider a single fault assumption where one fault is injected at a time affecting one element at a time. While time consuming, this approach helps paint a clear and thorough picture of the specific fault effects.

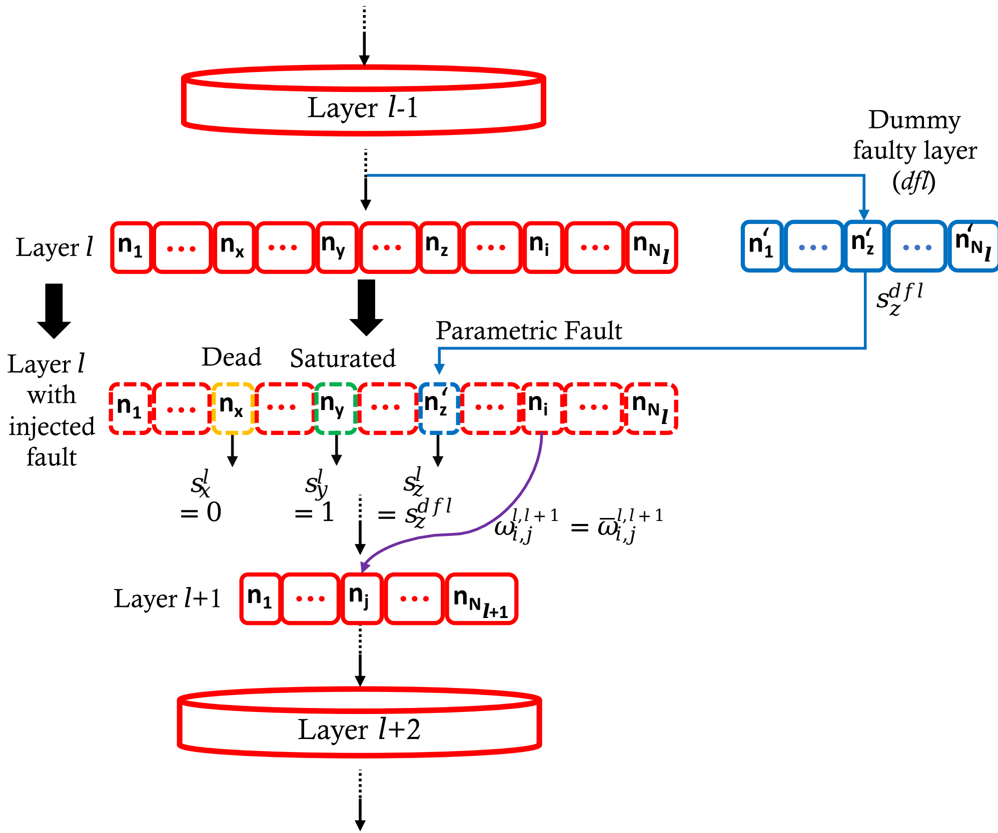


Figure 5.7: Fault injection methodology.

5.4 FAULT INJECTION EXPERIMENTS & RESULTS: (1) THE N-MNIST SNN

5.4.1 *Neuron Faults*

Examining the N-MNIST SNN architecture shown in Fig. 5.2, layers SC3, SF4 and SF5 appear to be relatively small in size comprising 120, 50 and 10 neurons, respectively. Taking this into account, we kick off the fault injection experiments with these layers, presenting the results in detail. Afterwards, we extend the experiment to include layers SC1 and SC2 as well. The results for each fault type are presented next.

5.4.1.1 *Dead Neuron Faults*

Fig. 5.8 shows the effect of a dead neuron in the last three layers on the classification accuracy of the network. The effect is shown in the form of a heat map, with the neuron number on the x-axis and the layers SC3, SF4 and SF5 on the y-axis. Each box corresponds to a specific neuron in

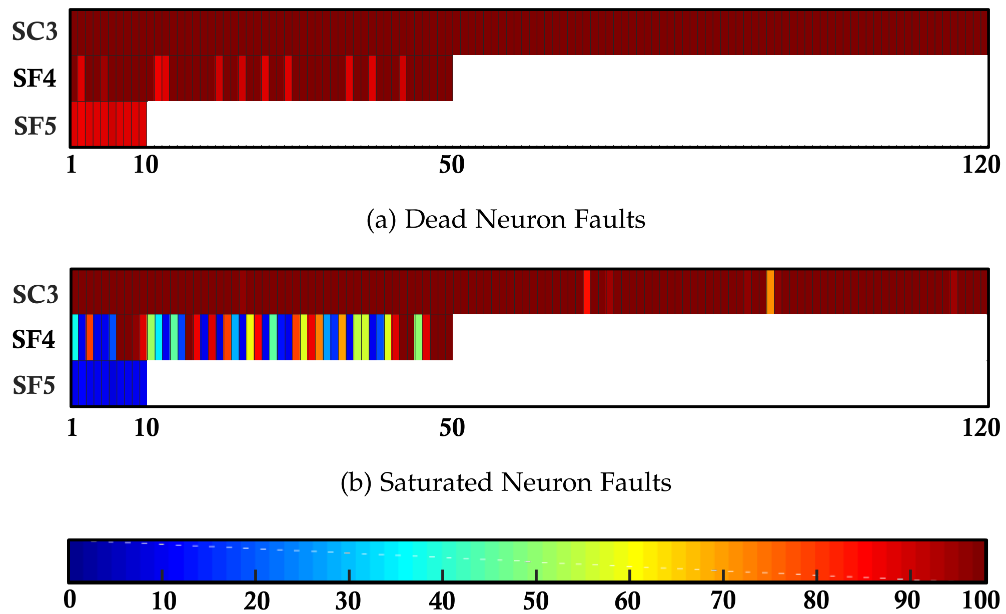


Figure 5.8: Effect of dead and saturated neuron faults on the N-MNIST SNN classification accuracy in the last 3 layers.

the layer, and the box color refers to the network classification accuracy achieved in the presence of a fault in set neuron. The color map is shown at the bottom of the figure.

Each neuron at the output layer is responsible for an output class, hence as expected, a dead neuron in layer SF5 immediately drops the classification accuracy to $(1 - \frac{1}{\#classes}) * 100\% = 90\%$ since one class is always misclassified. As for the last hidden layer, SF4, only a few dead neurons cause the classification accuracy to fall to about 90%. On the other hand, a dead neuron in layer SC3 has no effect on the overall classification accuracy of the network.

As for the first 2 convolutional layers, SC1 and SC2, visualizing the results on a per neuron basis would be a bit hard since they are much bigger in size. Instead, we present the results of these faults in terms of percentage of neurons, as shown in the columns labeled "dead" in Fig. 5.9. In this representation, the x-axis shows the layer of the network, and the y-axis shows the percentage of neurons that, when faulty, cause the classification to drop to the value indicated by the color. The value of the classification accuracy is elaborated in the color map at the bottom of the figure. The figure shows that a dead neuron in layers SC1 and SC2 has no effect on the classification accuracy.

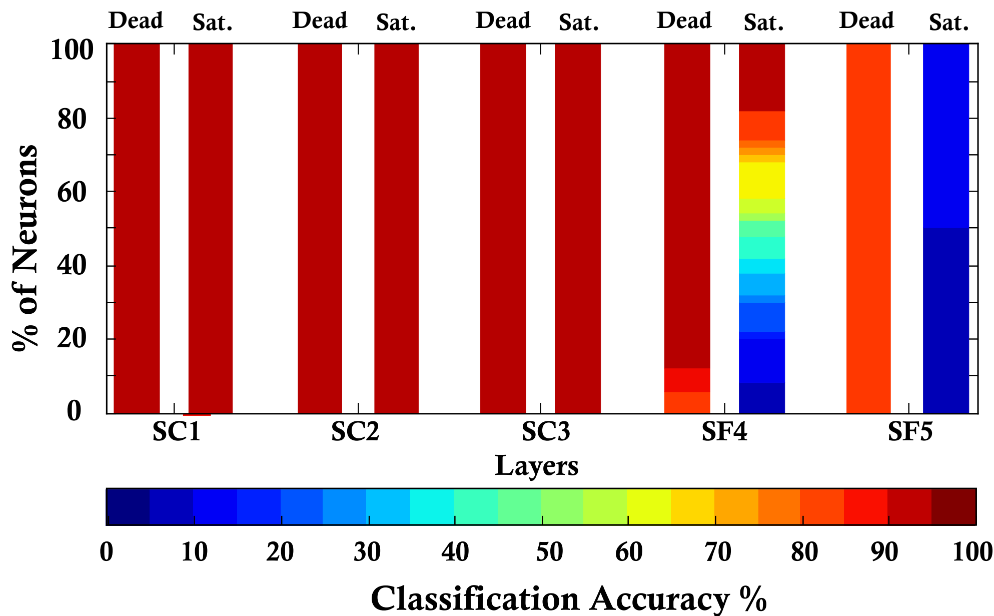


Figure 5.9: Effect of dead and saturated neuron faults on the N-MNIST SNN classification accuracy.

5.4.1.2 *Saturated Neuron Faults*

The effect of a saturated neuron fault is shown in the heat map of Fig. 5.8b. When a neuron in the output layer SF5 saturates, this neuron is always the one with the greatest number of output spikes and its class is always the winner. This means that only one digit is correctly classified at all times regardless of the input and the accuracy drops to a value of $\frac{1}{\#\text{classes}} * 100\% = 10\%$. Layer SF4 is also seriously affected by this fault type with the classification accuracy dropping to 10% for some neurons, suggesting that a saturation error is crucial in a SNN. However, the effect of a saturated neuron in layer SC3 is barely visible, and only two neurons cause the accuracy to drop no lower than 70%. The same goes for a saturated neuron in layers SC1 and SC2, that apparently have no effect on the classification accuracy of the network, as seen in the columns labeled "Sat." in Fig. 5.9.

5.4.1.3 *Parametric Faults*

As discussed in Section 5.3, parametric faults are modeled by manipulating the parameters of the neuron model. For each fault type discussed below, the corresponding parameter of every neuron is varied around its nominal value and the classification accuracy is evaluated. After the fault has been simulated for every neuron in the layer, the average of the resulting classification accuracies is computed to represent the effect of a

parametric fault on this particular layer. The maximum and minimum classification accuracies are also shown for comparison. Moreover, results are shown only for the last three layers since the first two convolutional layers were not affected by parametric faults.

a) Integration Faults:

Fig. 5.10 demonstrates the effect of neuron integration faults for each of the three layers SC3, SF4 and SF5. On the x-axis, the value of τ_s is expressed as a percentage of the nominal value used during training, while the value of the classification accuracy is put on the y-axis. A smaller τ_s in Eq. (5.2) implies a narrower synaptic kernel, i.e., a decreased integration time window and hence a reduced maximum possible value for the membrane potential. Consequently, the neuron spiking probability is reduced, and, in the extreme case, the neuron could end up as a dead neuron. Similarly, it can be argued that a higher τ_s increases the spiking probability and in the extreme the neuron could end up as a saturated neuron.

As evident from the results in Fig. 5.10, only layer SF5 appears to be severely affected by integration faults. For τ_s values below 30% of the original value, the accuracy drops only to 90%, while values of τ_s larger than 150% cause it to drop to about 70%. Layer SF4 is only slightly affected by this fault as the average accuracy decreases to no less than 96% and only by very small values of τ_s below 1%. On the other hand, layer SC3 is immune.

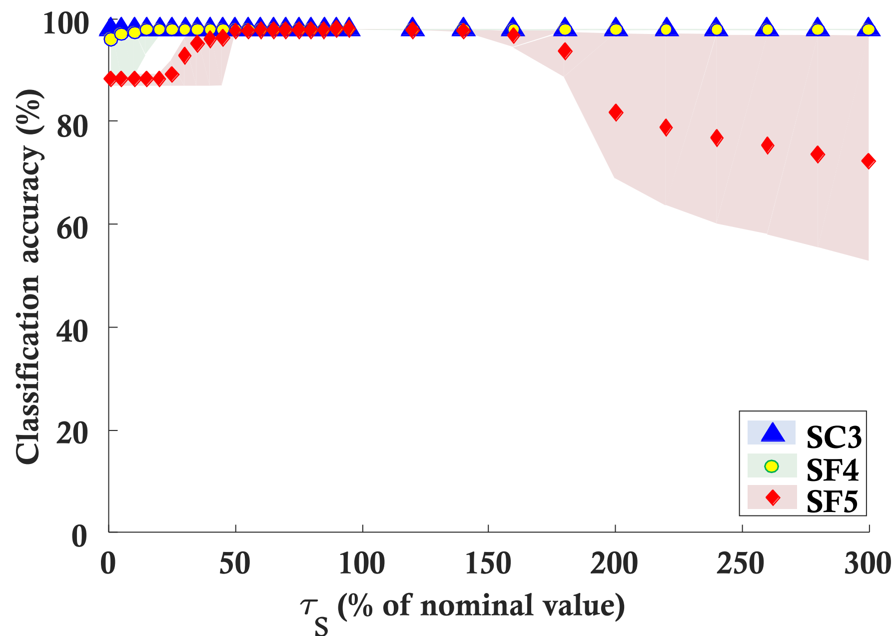


Figure 5.10: Effect of integration faults on the last 3 layers of the N-MNIST SNN.

b) Threshold Perturbation Faults:

The effect of a threshold perturbation fault in a neuron in layers SC3, SF4 and SF5, i.e., a perturbation in the value of threshold ϑ, is shown in Fig. 5.11. Small thresholds trigger spiking at lower values of the membrane potential and the neuron spikes more than usual, which can lead to misclassification of some outputs and a drop in the overall accuracy. In the extreme case, the neuron could end up saturated. On the other hand, a high threshold requires higher values of the membrane potential for spiking, thus causing the neurons to spike less and less, until eventually the neuron is not capable of spiking anymore and behaves as a dead neuron.

In the output layer SF5, it is obvious that when the threshold value is less than 50% of the original value, it results in severe drops in the average classification accuracy all the way down to 45%, whereas higher values of ϑ above 200% cause the accuracy to drop ever so slightly, with values no lower than 92%. As for layers SC3 and SF4, threshold perturbation faults have no visible effect on the network performance.

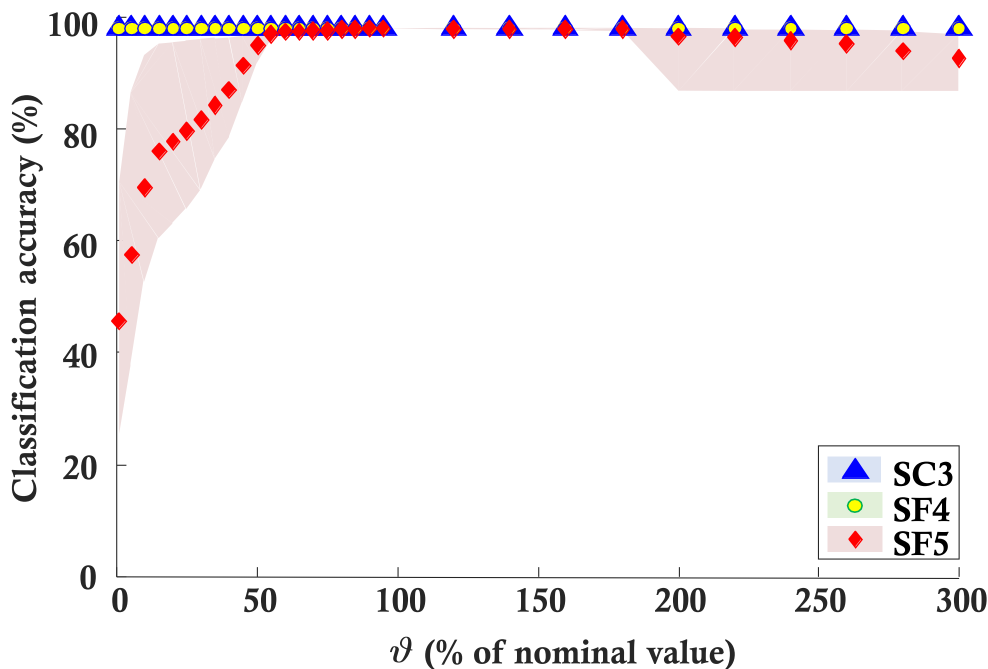


Figure 5.11: Effect of threshold perturbation faults on the last 3 layers of the N-MNIST SNN.

c) Refractory Period Faults:

Biological spiking neurons have an inherent minimum interval between two consecutive output spikes known as the refractory period, explained in Chapter 2. This value can be programmed into an artificial spiking neuron model to control the maximum spiking frequency of the neuron. A fault that causes the refractory period to be too high will make it difficult for the neuron to spike, and the neuron could end up with a dead output. On the other hand, if this period gets too short, the maximum spiking frequency can no longer be controlled and will be left to the model, which might not be critical.

Results presented in Fig. 5.12 show that for layer SF5, τ_{ref} must drop below 30% of the nominal value for the classification accuracy to start decreasing. In addition, τ_{ref} values below 15% result in severe classification accuracy drop down to 10%. In the meanwhile, large values of τ_{ref}, up to 300%, have no effect on the accuracy. Moreover, layers SC3 and SF4 are not affected by refractory period faults.

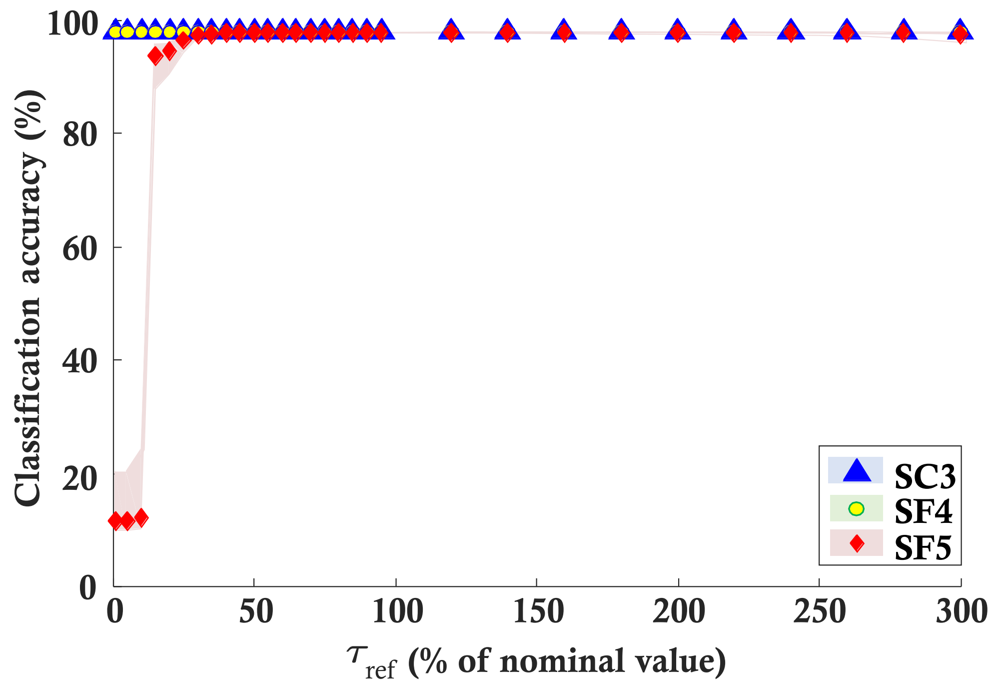


Figure 5.12: Effect of refractory period faults on the last 3 layers of the N-MNIST SNN.

5.4.2 *Synapse Faults*

Compared to neuron faults that propagate to many neurons in the next layer, synapse faults play a less important role since they affect only one neuron in the next layer. In this part of the experiment, we consider faults only in the synaptic connections between layers SC3 and SF4 and layers SF4 and SF5. The number of synaptic connections in the first layers is huge and proved to be time and resource consuming.

To select the values of weights suitable for simulating each fault type, the actual weight values resulting from the network training are examined. Figs. 5.13 and 5.14 display the weights of the synapses. Each box represents a synaptic connection between a neuron in layer l on the x-axis and a neuron in layer $l+1$ on the y-axis, with the color of the box indicating the approximate weight value according to the color maps at the bottom of each figure. The results for synaptic faults are presented next.

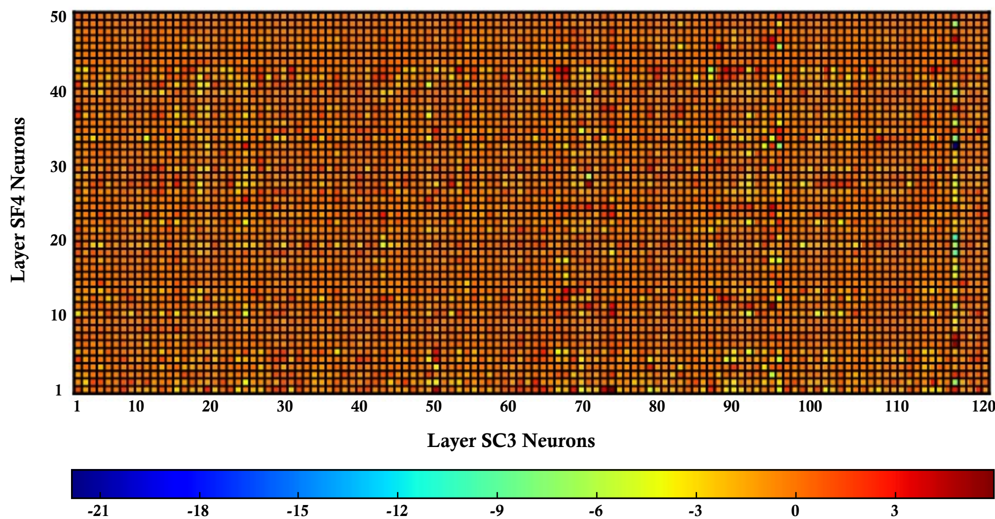


Figure 5.13: Synaptic weights values between SC3 and SF4.

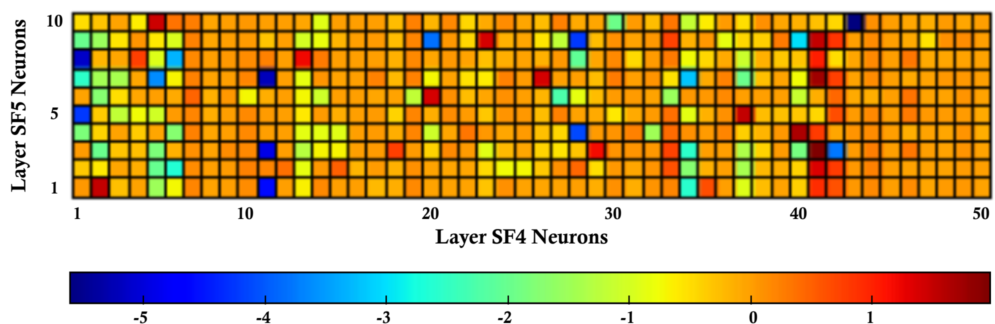


Figure 5.14: Synaptic weights values between SF4 and SF5.

5.4.2.1 *Dead Synapse Faults*

Figs. 5.13 and 5.14 show that most of the weight values rendered by the training are relatively small, with a mean value of approximately zero. Therefore, a dead synapse is not expected to be of much impact on the classification accuracy of the network. The results of the dead-synapse fault simulation are shown in Fig. 5.15, where again the x- and y-axes indicate neurons in different layers and boxes indicate a synaptic connection between 2 neurons. The color of the box in this case represents the classification accuracy according to the color map at the bottom of the figure.

A dead synapse in the synaptic matrix between layers SF4-SF5 results in a classification accuracy no less than 85%, and only about 10 synapses can cause this drop when faulty as seen in Fig. 5.15b. Meanwhile, dead synapses between layers SC3-SF4 have no visible effect on the performance, as shown in Fig. 5.15a.

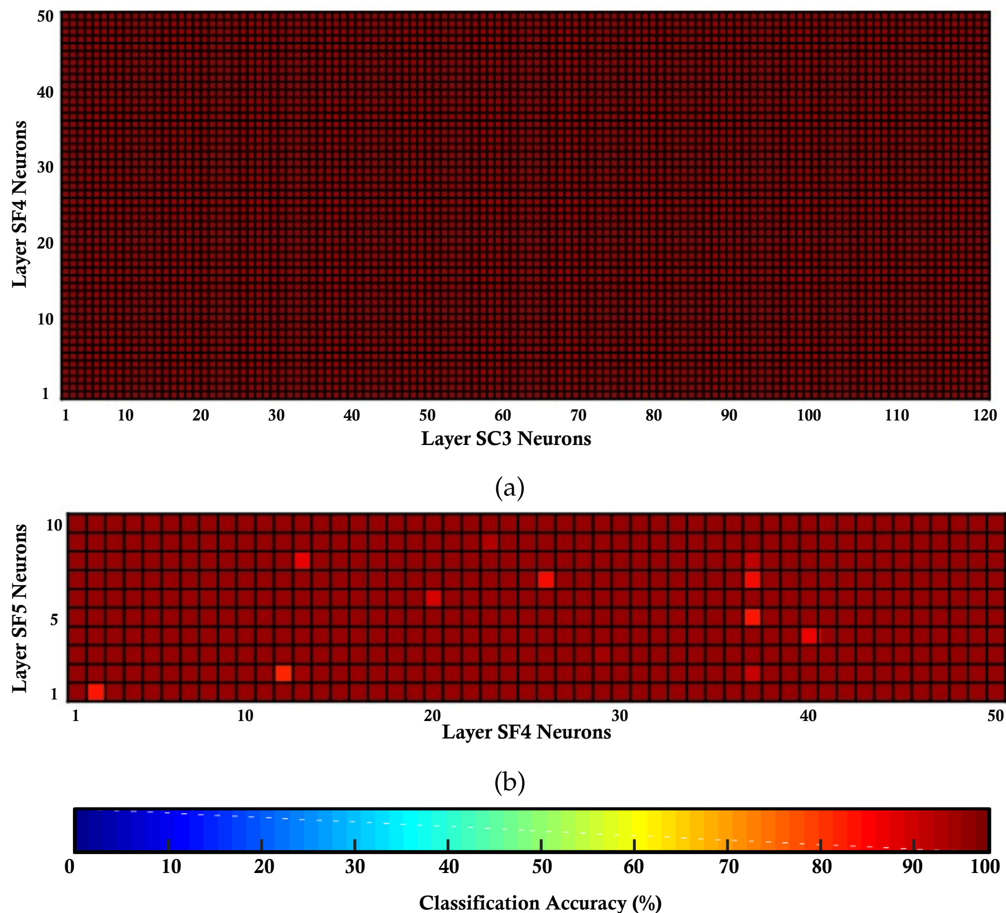


Figure 5.15: Effects of a dead synapse fault on synaptic connections between (a) layers SC3-SF4 and (b) layers SF4-SF5.

5.4.2.2 *Saturated Synapse Faults*

The values chosen to simulate negative and positive saturation faults were again chosen relative to the synapse weight distribution in Figs 5.13 and 5.14. Effects of negative and positive saturation are shown in Fig. 5.16 and 5.17, respectively.

a) Negative Saturation:
A negatively saturated synapse is of no effect between layer SC3-SF4, and hardly significant between layers SF4-SF5, as shown in Figs. 5.16a and 5.16b, respectively. A synapse that is saturated to a negative maximum will discharge the membrane potential of the post-synaptic neuron, thus reducing its spiking frequency. At the extreme case, the neuron will behave as dead and dead neurons have a little effect on the overall classification accuracy of the network.

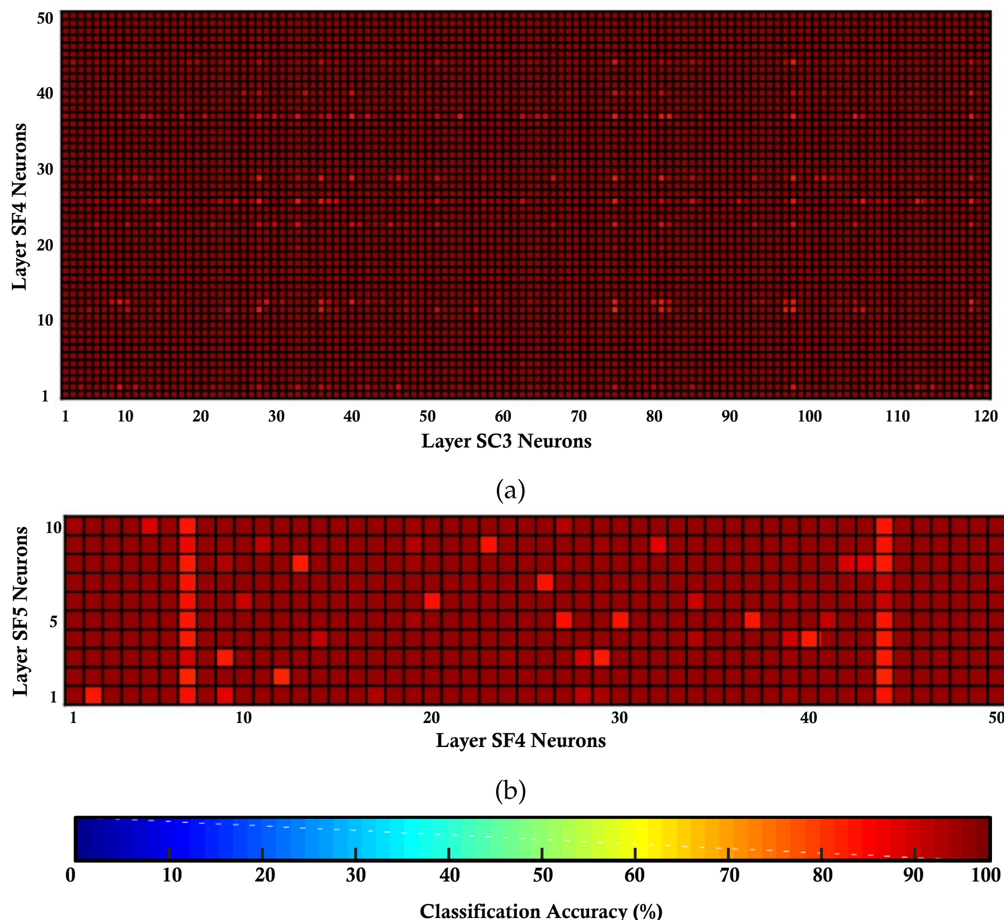


Figure 5.16: Effects of a negatively-saturated synapse fault on synaptic connections between (a) layers SC3-SF4 and (b) layers SF4-SF5.

b) Positive Saturation:

In contrast, a positively saturated synapse in both sets of synaptic connection can cause a drastic effect on the network classification accuracy, as shown in Figs. 5.17a and 5.17b. A fault of this kind in a single synapse can cause the accuracy to drop to a value as low as 10%, shown by the dark blue boxes in Fig. 5.17.

The reason that positive saturation appears to be more prominent is that a positively-saturated synapse magnifies the effect of the pre-synaptic neuron on the post-synaptic one, granting it with a higher membrane potential overall as shown from Eq. (5.1), which increases the chance that the post-synaptic neuron will fire, and in turn creating a domino effect on the following layers until the output layer where the classification takes place. In the extreme case, a positively saturated synapse may cause the post-synaptic neuron to saturate and, as deducted from the neuron fault simulation experiments, saturated neurons can have a severe effect on the classification accuracy.

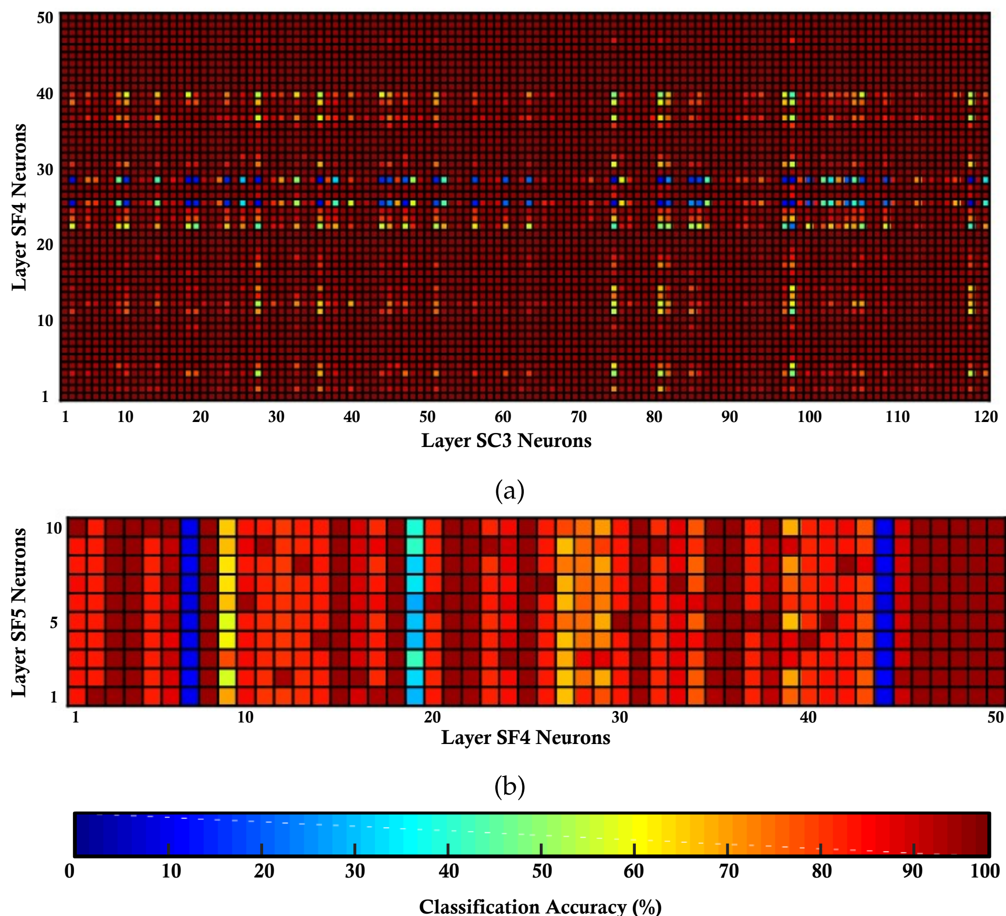


Figure 5.17: Effects of a positively-saturated synapse fault on synaptic connections between (a) layers SC3-SF4 and (b) layers SF4-SF5.

5.5 FAULT INJECTION EXPERIMENTS & RESULTS: (2) THE DVS-GESTURE SNN

Since the second case study, i.e., the DVS-gesture SNN, is much larger in size than the N-MNIST SNN, we choose to repeat only some of the fault injection experiments. For the sake of proving the validity of our fault models, we considered the most significant neuron faults, the results of which are presented next.

5.5.1 *Neuron Faults*

5.5.1.1 *Dead Neuron Faults*

The effect of a dead neuron in different layers of the DVS-gesture Network is shown in the columns labeled "Dead" in Fig. 5.18. With 82.2% maximum classification accuracy achieved by this network, it is clear that a dead neuron in either layer SC1 or SC2 has no effect on the accuracy. On the other hand, about 80% of neurons in the output layer, SF4 cause a drop in the accuracy to about 70%. Dead neurons in layer SF3 also seem to have a significant effect on the network in this case. To quantify some examples, about 5% of neurons in this layer, when dead, cause the accuracy to drop to 10%, and another 10% of the neurons drop the accuracy to around 20% when dead.

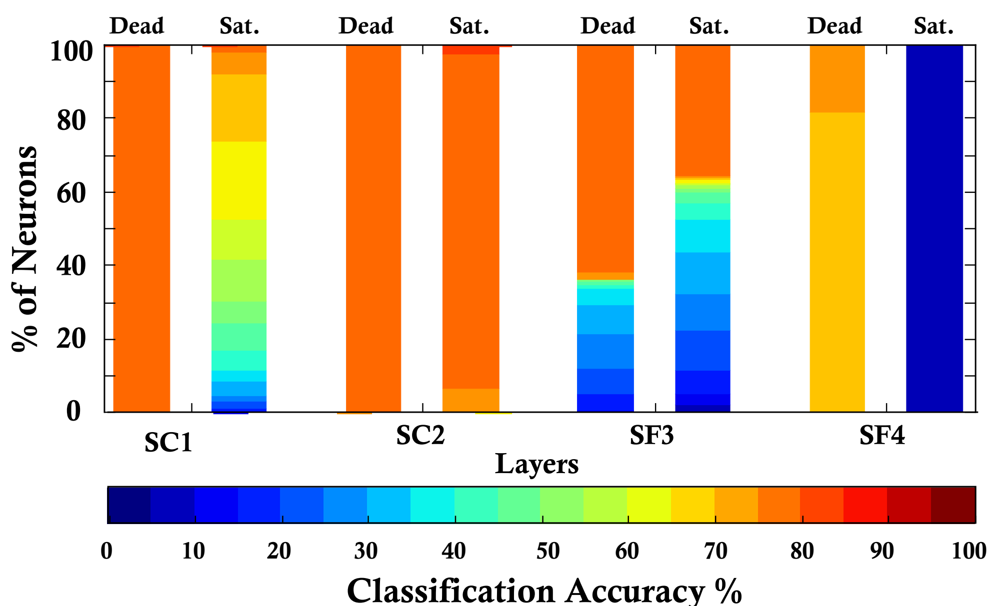


Figure 5.18: Effect of dead and saturated neuron faults on the DVS-gesture SNN classification accuracy.

In short, dead neurons seem to be a little more significant in the DVS-gesture SNN case study than in the case of the N-MNIST SNN. This could be due to the inherent difference between the complexity of the datasets.

5.5.1.2 *Saturated Neuron Faults*

Similar to the N-MNIST SNN case, saturated neurons in the case of the DVS-gesture SNN always have a significant influence on the network classification accuracy, as shown in the columns labeled "Sat." in Fig. 5.18.

As expected, a saturated neuron in the output layer, SF4, instantly drops the classification accuracy to less than 10%. A saturated neuron in the rest of the layers can also lead to significant drops in the classification accuracy, evident by the blue-ish segments in Fig. 5.18. When looking at the results of layers SC1, SC2 and SF3, an inverse relationship is observed between the number of outgoing synapses from a layer and the effect of saturated neuron faults on neurons in this layer. I.e., synapses connecting SC1 and SC2 are much less than of those connecting SC2 and SF3, while the effect of a saturated neuron in layer SC1 is notably larger than in layer SC2.

5.5.1.3 *Parametric Faults*

In the case of the N-MNIST SNN, we considered 3 forms of parametric faults in a neuron. All 3 fault types gave remarkably similar results; the last hidden layer and the output layer are practically the only ones affected, and the classification usually drops at very low or very high values of the examined parameter. Hence, for the case of the DVS-gesture SNN, we considered only 1 fault type to represent all parametric faults that can affect a spiking neuron.

Shown in Fig. 5.19, the results of an integration fault in a spiking neuron is shown for layers SF3 and SF5. The value of τ_s in Eq. (5.2) is again changed around its nominal value for one neuron at a time and then the classification accuracy is calculated. Fig. 5.19 shows the average of all the accuracies for a given layer in addition to the span between the maximum and the minimum values that arise from this error.

Results show that this fault type can seriously affect the output layer SF4, dropping the accuracy to about 70% for τ_s values less than the nominal, and a sweeping drop to less than 20% when τ_s is more than 200% of its nominal value. The last hidden layer, SF3 contributes to significant classification accuracy drop only when τ_s is reduced to less than 50%, while large values of τ_s do not seem to disturb the network.

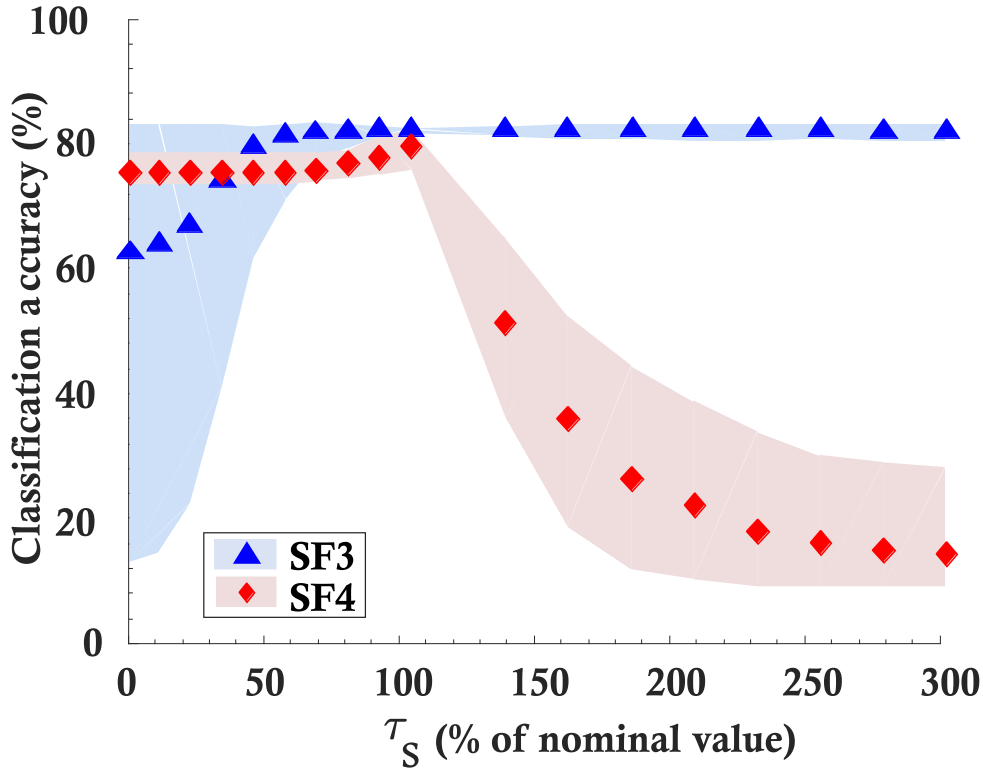


Figure 5.19: Effect of integration faults on the last 2 layers of the DVS-gesture SNN.

5.6 DISCUSSION

Experiments conducted in this chapter offer an insight into the various factors that contribute to the vulnerability of spiking neural networks to hardware faults. These deductions will be the starting point on which we build the fault tolerance strategy in the next part of the work.

Besides the obvious factor that is the type of the fault; the complexity of the network architecture, the type of layer, and the location of the fault within the network hierarchy play an important role in the severity with which faults affect the performance of the SNN.

By default, convolutional layers are not fully connected, which means they have less outgoing synaptic connections. Therefore, a fault in a neuron in a convolutional layer is less likely to have a drastic effect on the network performance. In contrast, faults affecting neurons in fully connected layers had far more serious effects regardless of the fault type.

Fault type, on the other hand, is perhaps the most significant factor of all. Fault simulation experiments have conveyed that saturation faults are much more severe than any other fault type that can affect a spiking neuron. However, fault location within the network is of great influence

as well. All performed experiments show that the closer the fault is to the output layer, the more severe its effect will be. The output layer is always the most vulnerable since there is usually one neuron for every output class, which means that special attention should be paid to making this layer more robust and tolerant against hardware errors. The last hidden layer is second in line with notably less fault aftermath.

As for the single entities of the network, neuron faults were found to be more critical than synapse faults. This makes sense because while a synapse links 2 neurons together, a neuron in layer (l) is usually connected to many neurons in layer $(l + 1)$. This means that a fault in a neuron will spread out as it propagates through the network and eventually cause more damage.

6 NEURON FAULT TOLERANCE

After performing large-scale fault injection experiments and pinpointing the critical fault types and locations; in this chapter we leverage these observations and propose a neuron fault-tolerance strategy for SNNs that is optimized for low area and power overhead [121]. Generally speaking, standard fault-tolerance techniques for regular VLSI circuits can be employed, such as Triple Modular Redundancy (TMR) and Error Correction Codes (ECCs) for memories. However, efficiency can be largely improved by exploiting the architectural particularities of AI hardware accelerators and targeting only those fault scenarios that have a measurable effect on performance.

As discussed in Section 2.4.3, fault tolerance can be classified into active and passive depending on how it is achieved and at what level is applied. In this chapter, we exploit the findings from our fault injection experiments to sketch some possible fault tolerance solutions that can be implemented in a hardware neural network. We propose cost-effective fault tolerance strategies tailored to address severe faults in the SNN and consisting in multiple layers of protection. Firstly, we propose passive fault tolerance based on dropout to nullify the effect of certain faults, then we project several active fault tolerance techniques to detect and recover from the remaining critical faults.

6.1 PASSIVE FAULT TOLERANCE STRATEGY

As a first step, we aimed at implementing passive fault tolerance such that the SNN is by construction capable of withstanding some faults without any area and power overheads. In this fault tolerance paradigm, the network is trained using a technique known as dropout [126]. In this section, a brief introduction to dropout is provided along with the specific parameters used to train our SNN case studies. After that, for verification purposes, a multiple fault simulation experiment is carried out to prove that dropout indeed helps nullify the effect of dead neuron faults and

neuron timing variations in all hidden layers in addition to increasing the tolerance of the network to multiple simultaneous faults. Consequently, active fault tolerance, which implies area and power overheads, gets simplified since it only needs to focus on saturation neuron faults in the hidden layers and on all fault types only for the output layer.

6.1.1 *Training with Dropout*

The dropout training technique was originally proposed to prevent overfitting and reduce the generalization error on unseen data. The idea is to temporarily remove neurons during training with some probability p along with their incoming and outgoing connections. At test time, the final outgoing synapse weights of a neuron are multiplied by p. For a network with n neurons, there are 2^n "thinned" scaled-down networks, and training with dropout combines exponentially many thinned network models. The motivation is that model combination nearly always improves performance, and dropout achieves this efficiently in one training session.

For the N-MNIST SNN we used $p = 10\%$ in the input and SC1 layers, 20% in layers SC2 and SC3, and 50% in layer SF4. Training with dropout resulted in a slight improvement in the classification accuracy from 98.08% to 98.31%. For the DVS-gesture SNN we used $p = 50\%$ only in layer SF3. In this case, dropout significantly increased the classification accuracy from 82.2% to 87.88%.

The beneficial effect of dropout on passively nullifying the effect of dead neuron faults is shown for each layer in Figs. 6.1a and 6.1b for the N-MNIST and DVS-gesture SNNs, respectively. For instance, for layer SF4 of the No-MNIST SNN, dead neurons have no effect on the SNN trained with dropout. Moreover, the difference between the percentage of neurons that -when saturated- cause a dramatic drop in the classification accuracy of the network is huge and the classification rate does not fall below 50%. This is likewise evident in the case of layer SF3 of the DVS-gesture SNN.

Training with dropout is also rewarding in the case of timing variation faults in the last hidden layers as shown in Figs. 6.2b and 6.3b for the N-MNIST and DVS-gesture SNNs, respectively. Compared to the non-dropout cases in Figs. 6.2a and 6.3a, variations in τ_s from 1% to 300% have now no effect. The reason behind this result is that dropout essentially equalizes the importance of neurons across the network, resulting in more uniform and sparse spiking activity across the network. Therefore, if a neuron in a hidden layer becomes dead or shows excessive timing variations, this turns out to have no effect on the overall classification

accuracy. On the contrary, dropout may magnify the effect of saturation faults in neuron, e.g., layer $SF3$ of the DVS-gesture SNN.

Finally, we observe that dropout does not compensate for faults in the output layer since in this layer there is one neuron per class and any fault will either overshadow this class or cause it to dominate the other classes.

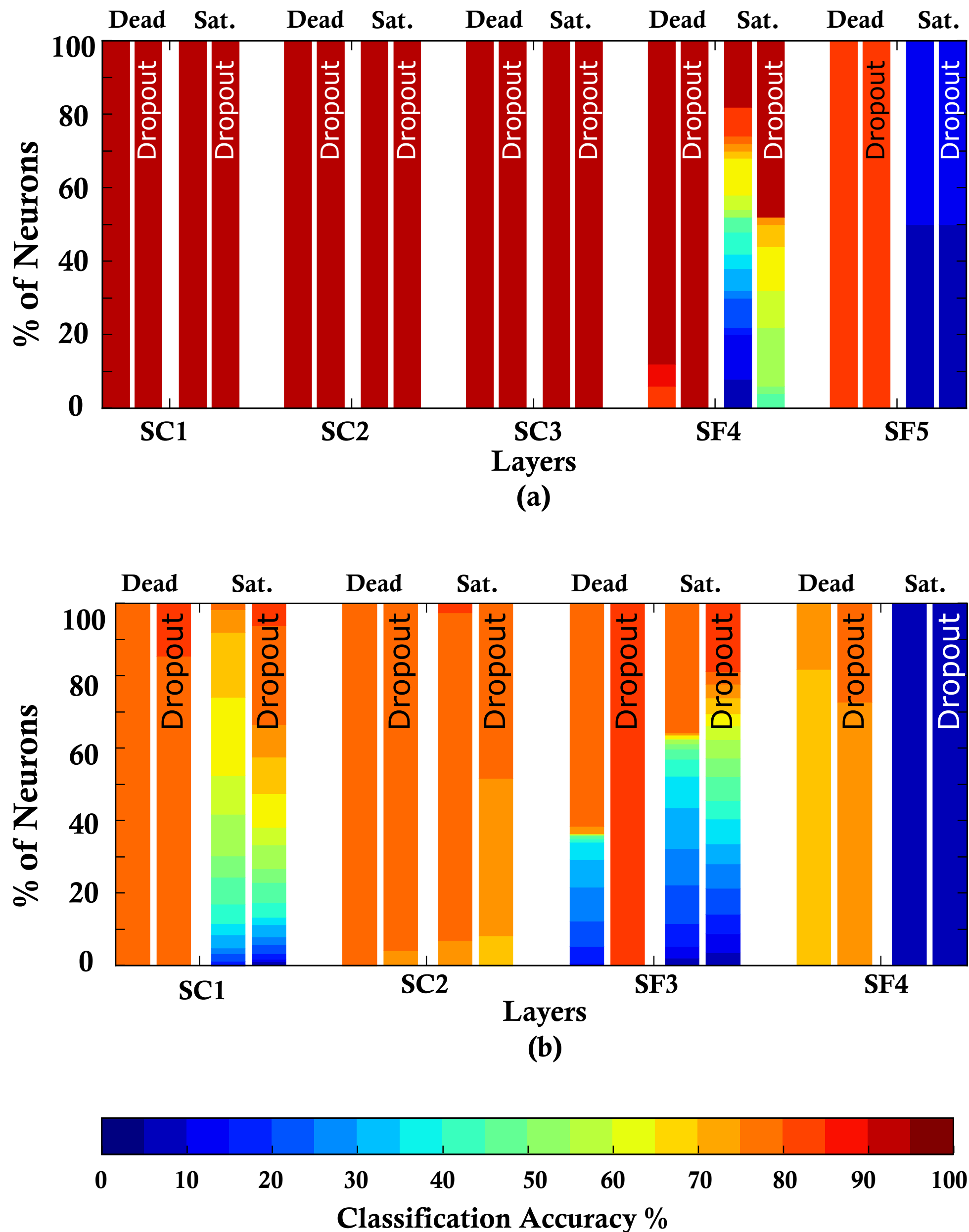
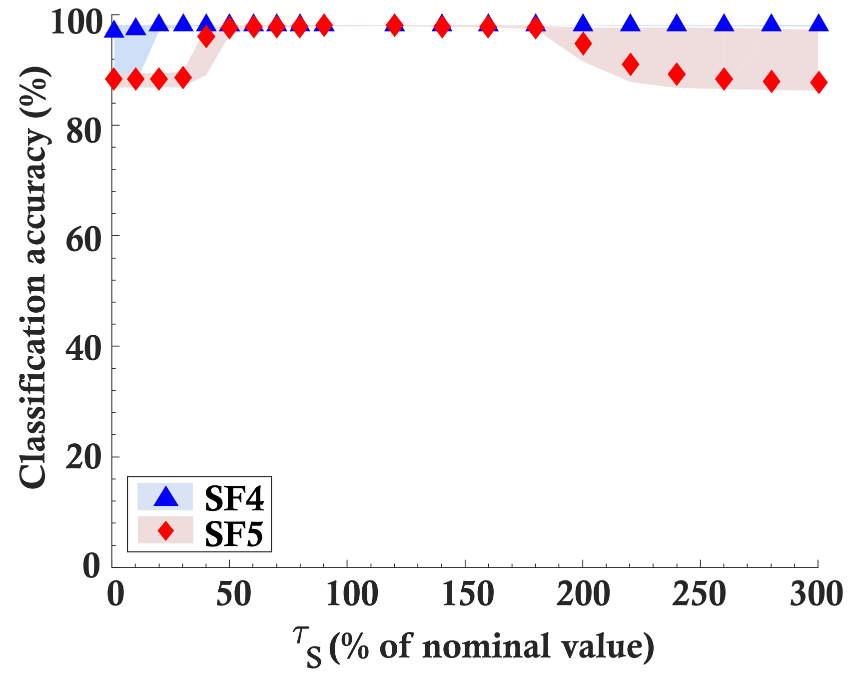
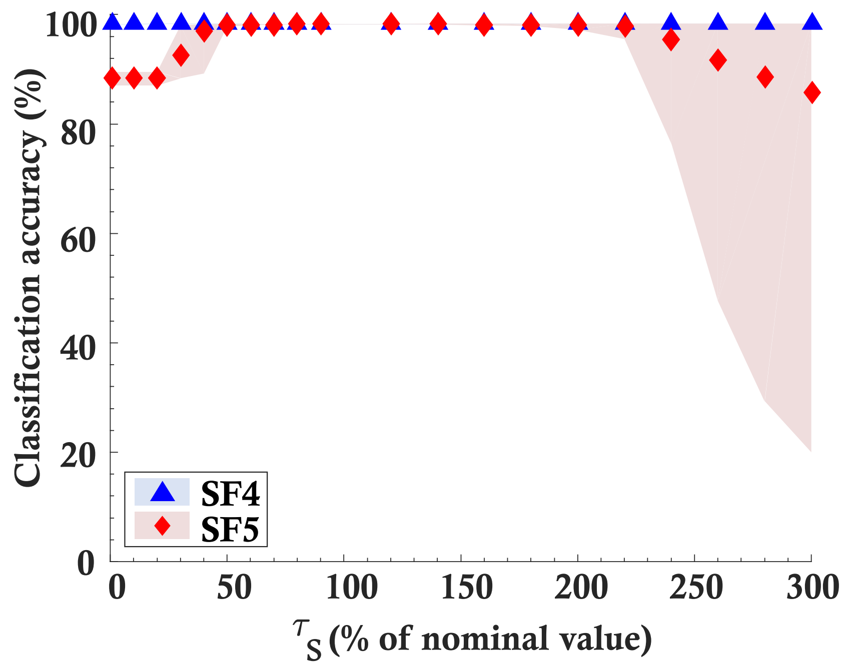


Figure 6.1: Effect of neuron faults on classification accuracy with and without dropout for: (a) N-MNIST SNN, and (b) DVS-gesture SNN.

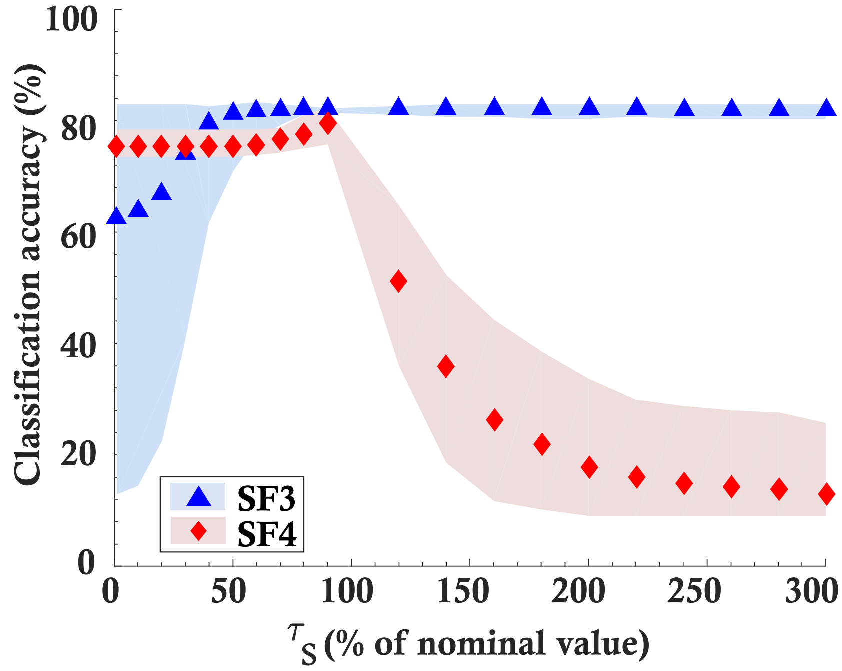


(a) Without Dropout

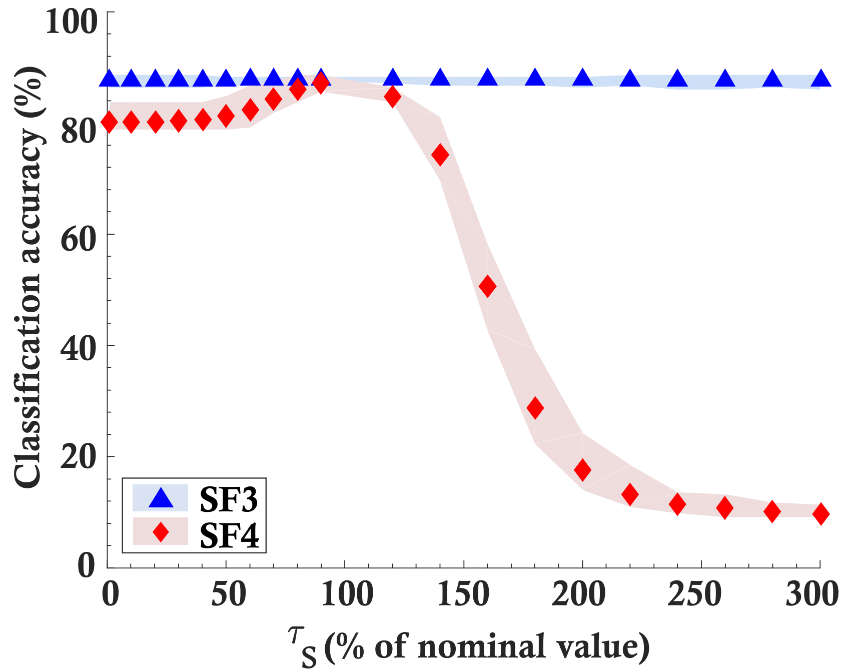


(b) With Dropout

Figure 6.2: Effect of neuron timing variations for the N-MNIST SNN.



(a) Without Dropout



(b) With Dropout

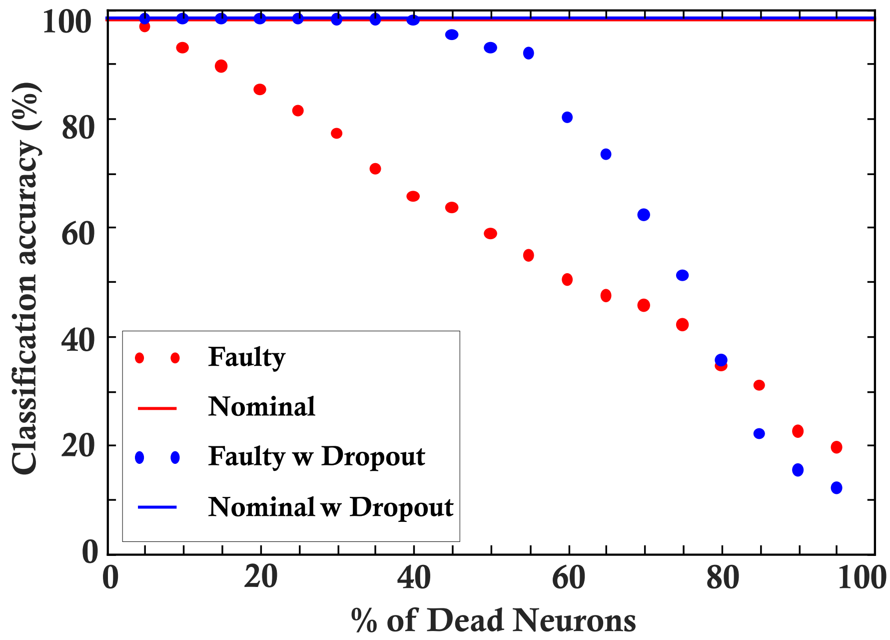
Figure 6.3: Effect of neuron timing variations for the DVS-gesture SNN.

6.1.2 *SNN Tolerance to Multiple Faults*

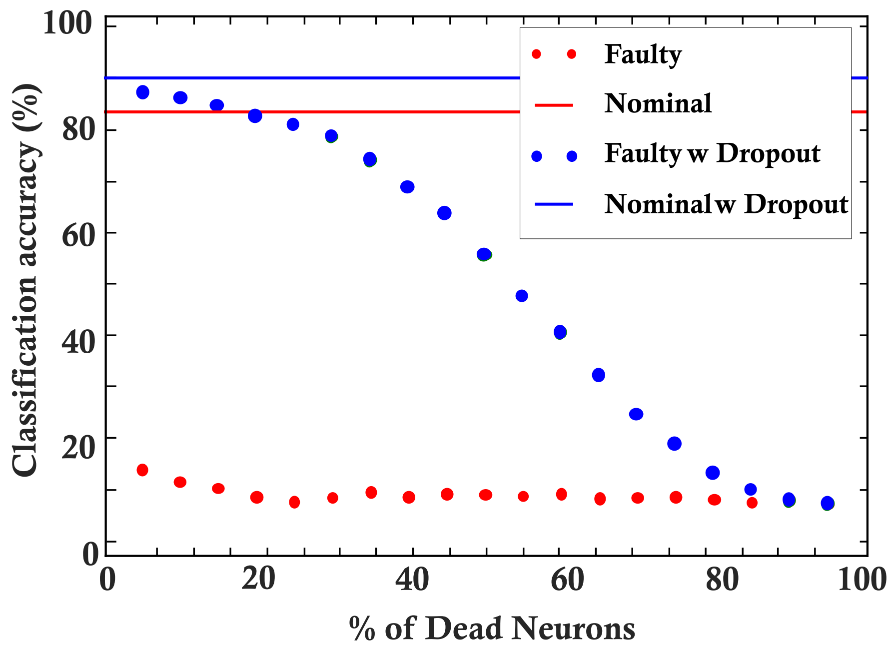
So far, we have proved that dropout can effectively diminish the effect of a dead neuron on the functionality of an SNN. But what happens if more than one neuron becomes faulty at the same time? In other words, what percentage of dead neuron faults can dropout help the network withstand? To answer this question and, hence, further verify the gains that can come from using dropout in the training phase, multiple faults are injected into the network simultaneously to assess its tolerance.

The results of this multiple fault injection campaign are presented in Fig. 6.4 where the classification accuracy is plotted as a function of the percentage of dead neuron faults introduced into the last hidden layer of the N-MNIST SNN (Fig. 6.4a) and the DVS-gesture SNN (Fig. 6.4b). The baseline classification accuracies with and without dropout are shown as straight lines for reference. The experiment starts with the injection of dead faults into 5% randomly chosen neurons from the last hidden layer, then with every step, faults are injected into an extra 5% of the neurons. This process is repeated until 90% of the neurons of the layer are simultaneously dead. Moreover, in order to properly generalize the results, the experiment is repeated 10 times and the calculations are averaged at every step.

A first glance at Fig. 6.4 clearly shows that with the increase of the percentage of dead neurons, the network performance degrades much faster in the case of standard training than in the case trained using dropout. Moreover, the SNNs employing dropout can withstand larger rates of dead neurons at once. More specifically, the N-MNIST SNN trained using dropout does not lose any classification accuracy with a dead neuron rate of up to 40%, compared to a 10% limit without dropout. As for the DVS-gesture SNN, the classification accuracy drops a little faster, but the network is still able to perform with over 80% classification accuracy at a dead neuron rate of 20% which corresponds to 102 dead neurons in the last hidden layer.

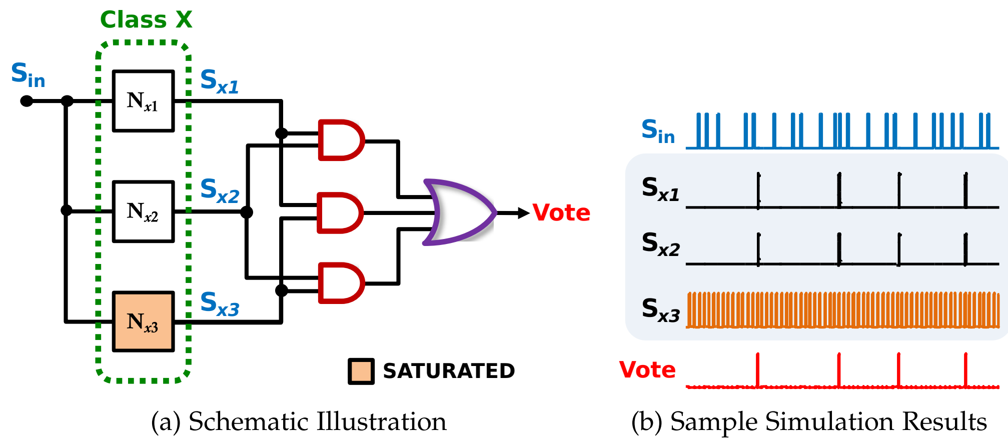


(a) N-MNIST SNN



(b) DVS-gesture SNN

Figure 6.4: Fault tolerance for multiple fault scenarios with regular standard training vs. training with dropout.



(a) Schematic Illustration (b) Sample Simulation Results

Figure 6.5: TMR at the output layer.

6.2 ACTIVE FAULT TOLERANCE STRATEGY

6.2.1 *Active Fault Tolerance in the Output Layer*

As for the most critical output layer, we propose to directly use TMR for a seamless recovery solution from any single fault type. In particular, a group of three identical neurons vote for the decision of a certain class, as shown in Fig. 6.5. The voter is a simple 4-gate structure that propagates the output upon which the majority of neurons agree. This means that a faulty neuron in the group, be it dead, saturated, or showing excessive timing variations, is outvoted and bypassed. Performing TMR only in the last layer will result in negligible increase in the area and power overhead and a reasonable overhead to pay to ensure strong fault tolerance. The reason is that the number of neurons in the output layer is typically small compared to the size of the whole network. For example, in the N-MNIST SNN, the output layer neurons account for about 0.57% of the neurons in the whole network. This percentage gets even less for the more complicated DVS-gesture SNN, where the output layer represents around 0.04% of the total number of neurons.

6.2.2 *Active Fault Tolerance in the Hidden Layers*

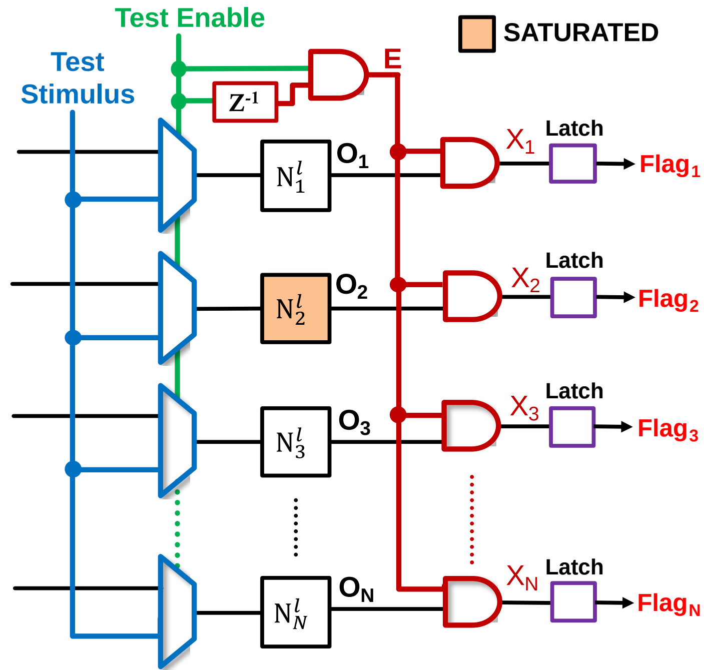
Following a training that employs dropout, the network is left with the vulnerability of the output-layer neurons and the risk posed on the overall performance by a saturated neuron in a hidden layer. This means that a cost-effective strategy to implement fault-tolerance into a neural network would be to solely focus on these two cases. We propose two self-test schemes for neuron saturation detection, namely an offline scheme

that can run during idle times of operation and an online scheme that can run concurrently with the operation. Regarding the faulty recovery mechanism, we propose the "fault hopping" concept to simplify the hardware implementation, and we propose two recovery mechanisms at neuron-level and system-level.

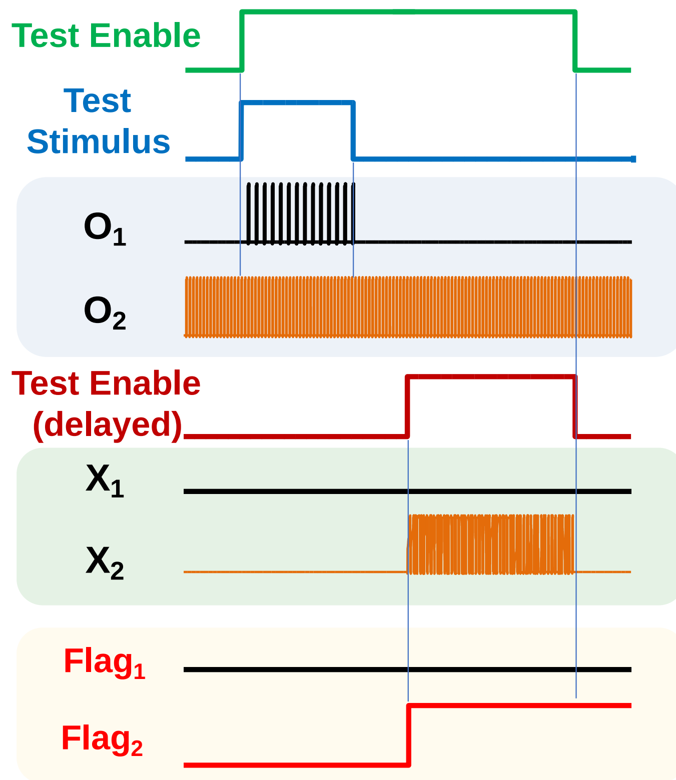
6.2.2.1 *Offline Self-Test*

An offline saturation detection scheme is depicted in Fig. 6.6. In this approach, neuron saturation is declared based on the neuron's activity in the absence of an input. The operation of the circuit is divided into two modes: self-test mode and operation mode, and a multiplexer is assigned to every neuron to switch between modes. During normal operation, the neurons are receiving inputs from the previous layer through synapses, processing them and propagating them to the next layer. When the circuit is switched to the self-test mode through the test enable signal, an ad hoc internally generated test stimulus is applied to all the neurons simultaneously. The neuron outputs are then paired with a saturation detection signal through AND gates. There is a delay between the start of the detection signal and the end of the test stimulus. This delay is important to ensure that any activity detected is uncorrelated with the input of the neuron and is indeed a result of saturation. The output of an AND gate going high indicates neuron saturation. This is captured by the latch which raises an error flag signal. A simulation is shown in Fig. 6.6 using the I&F neuron shown in Fig. 6.8 which is designed in the AMS 0.35μm technology.

There are two main advantages to this detection paradigm. Firstly, it only adds a multiplexer, an AND gate, and a latch per neuron, thus the area overhead of the test circuitry is relatively small compared to a single neuron in the original network. Secondly, the approach is fast since it tests all neurons in parallel and it can also be applied multiple times during the chip's lifetime, which allows it to detect aging-induced errors, possibly with some latency.

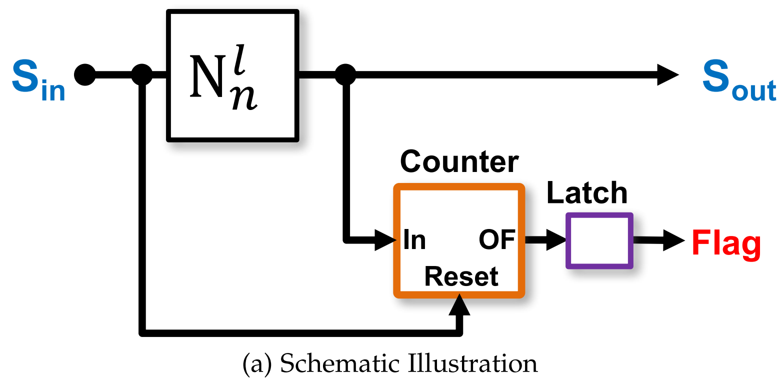


(a) Schematic Illustration

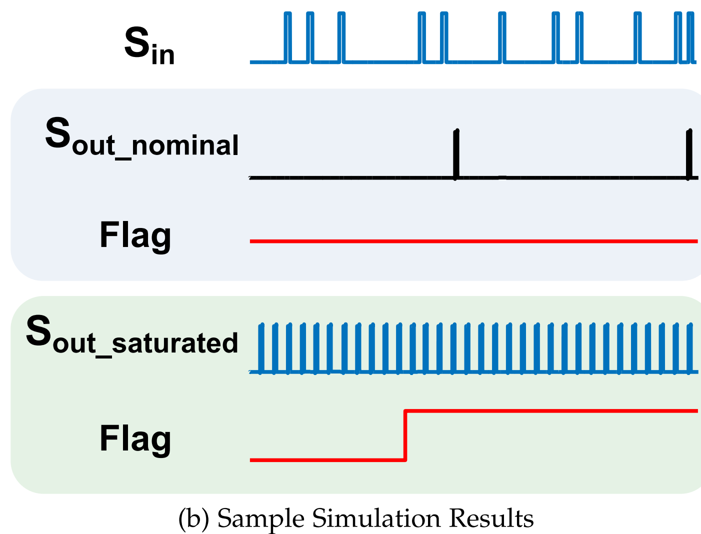


(b) Sample Simulation Results

Figure 6.6: Offline self-test scheme.



(a) Schematic Illustration



(b) Sample Simulation Results

Figure 6.7: Online self-test scheme.

6.2.2.2 *Online Self-Test*

The second fault tolerance scheme is a mechanism that can catch a saturated neuron during normal operation, i.e., online detection, as shown in Fig. 6.7. This approach is applied on a per-neuron basis and takes advantage of the temporal dependency between the input and output of a spiking neuron. In other words, we count the number of spikes a neuron produces after every single input spike using a counter whose reset port is connected to the input of the neuron. In fault-free operation, the neuron needs to integrate multiple input spikes before it can produce a spike of its own, hence the counter is always reset, and the flag signal stays at zero. On the other hand, a saturated neuron will produce more spikes than usual causing the counter's output to overflow before an input spike resets it again. In this case, A latch is set, and an error flag is raised, suggesting that this neuron is saturated, and a recovery mechanism is activated.

Based on our simulations, 2^3 uncorrelated spikes clearly indicate saturation; thus it suffices to use a 3-bit counter. Fig. 6.7 shows a simulation using the I&F neuron of Fig. 6.8. This online self-test scheme entails an area overhead comprised of a counter and a latch per neuron. All neurons are monitored individually, and neuron saturation is detected in real-time.

While the proposed scheme entails an area overhead comprised of a counter and a latch per neuron, it offers some advantages to the resilience of the system to faults, and hence its robustness. Firstly, the detection runs seamlessly during normal operation without causing any disruptions, which saves time that is usually spent on testing. Moreover, the test is automatically valid throughout the lifespan of the chip without the need for external interventions.

6.2.2.3 *Recovery Mechanisms*

Behavioral-level fault experiments performed earlier, indicated that eliminating saturated neurons provides considerable improvement in the accuracy. Therefore, we propose the concept of "fault hopping" where the critical saturation neuron fault is artificially translated to a dead neuron fault. By applying this idea, the network performance is restored since a dead neuron has essentially no effect after training with dropout. This approach leads to an elegant hardware implementation and saves significant costs as opposed to the standard approach, which is to duplicate or triplicate neurons, or provision the SNN with spare neurons that are kept "fresh" and switch the connections of a detected saturated neuron to a spare neuron [29]. By addressing this idea, we propose two types of recovery operations: transistor-level and system-level.

- ***Neuron-Level Recovery***

  Neuron-level recovery is implemented by switching-off the saturated neuron. For example, for the I&F neuron in Fig. 6.8, this can be achieved by connecting a single extra transistor M_C in the tail part of the comparator inside the neuron. This transistor is controlled by the neuron error flag signal. If a neuron gets saturated, M_C becomes open-circuit and the biasing connection of the comparator is suddenly ceased, which ultimately deactivates the neuron, and its output becomes zero.

  The area overhead of this approach is just one additional transistor per neuron. On the other hand, the power consumption is reduced since the saturated neuron, once deactivated, does not consume power anymore.

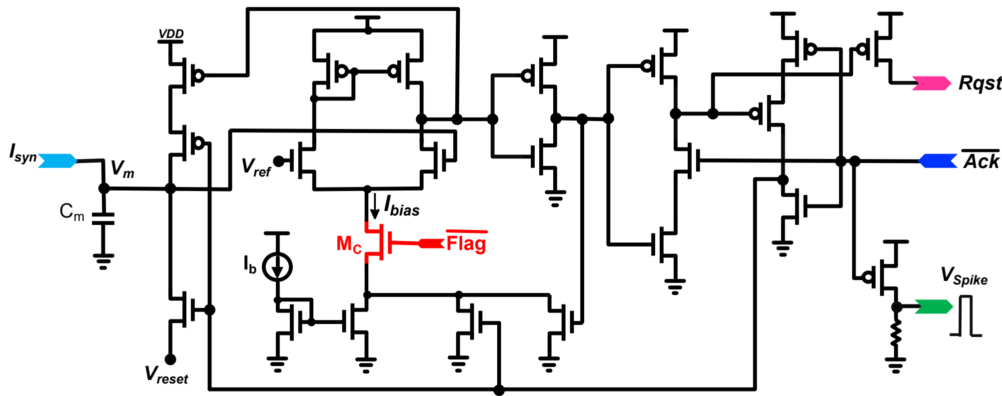


Figure 6.8: I&F neuron design showing the recovery operation at neuron-level.

❖ ***System-Level Recovery:***

System-level recovery operation is based on the manipulation of the weights linked to the saturated neurons. Typically, the communication between neurons and synapses in the SNN is performed by a controller, particularly AER-based controller [53]. AER controllers perform multiplexing/demultiplexing of spikes generated from or delivered to all neurons in a layer onto a single communication channel. Rather than delivering the actual spike, the controller encodes the address of the neuron that spiked and translates it into the addresses of the destination neurons, and then the weights corresponding to every synaptic connection are loaded accordingly.

By leveraging this operation, the proposed system-level recovery approach is based on equipping the controller with the ability to recognize the neuron error flag and update the outgoing synaptic weights to zero. There are two ways to manipulate the weights: we can either set all weights of the saturated neuron to zero, or we can create and address a null row in the memory that replaces the actual weights. In both approaches, the weights of the saturated neuron are nullified, and the saturated output is trapped unable to propagate further.

The system level approach has nearly no additional cost in terms of area overhead since it is reused across all neurons. However, the power consumption is higher compared to the transistor-level solution since saturated neurons, which are still operating, consume power.

7 A SPIKING NEURAL NETWORK HARDWARE IMPLEMENTATION

Throughout this thesis, we investigated the effects of hardware faults on SNNs and proposed strategies for passive and active fault tolerance that can help the network maintain proper operation under faulty conditions. Hardware faults were translated into their behavior-level effects that enabled us to perform experiments at the software simulation layer. However, to fully validate these experiments and solidify the obtained conclusions, there was a need for a hardware neural network implementation flexible enough to allow the execution of multiple experiments.

In this chapter, we present an event-driven configurable convolutional node designed in Very High-Speed Integrated Circuit Hardware Description Language (VHDL) for FPGAs and that can be used to construct large-scale convolutional SNNs. The node was designed by Camuñas-Mesa *et al.* [127] and demonstrated as a building block of a convolutional SNN for poker card symbol recognition with various biologically inspired capabilities. We outline the strategy followed to transform it into an autonomous hardware-in-the-loop platform where we can construct arbitrary networks and use direct spike-based training algorithms in an accelerated manner.

7.1 THE CONVOLUTIONAL NODE

The convolutional node used in this work is designed as a generic block that can be used to build multi-layer feature maps for convolutional SNNs. The ports of the node are optimized for a 2-D layout, which is an efficiently adopted structure in hardware convolutional neural networks since it optimizes the use of on-chip space. Each node has four bidirectional ports connecting it to its immediate neighbors to the *North*, *South*, *East* and *west*.

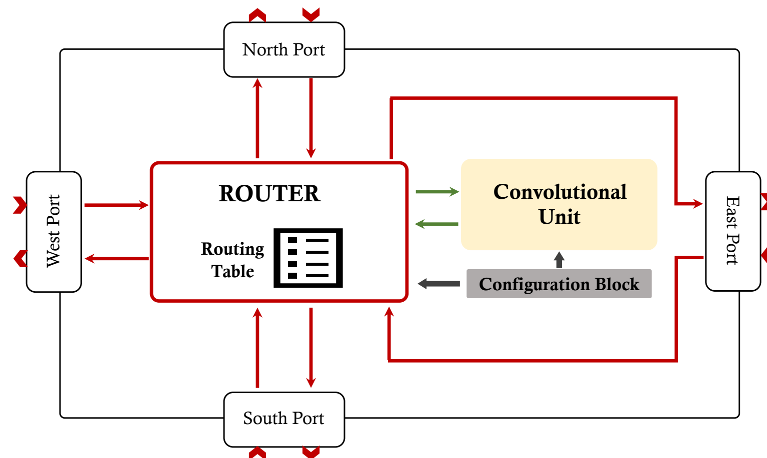


Figure 7.1: The convolutional Node.

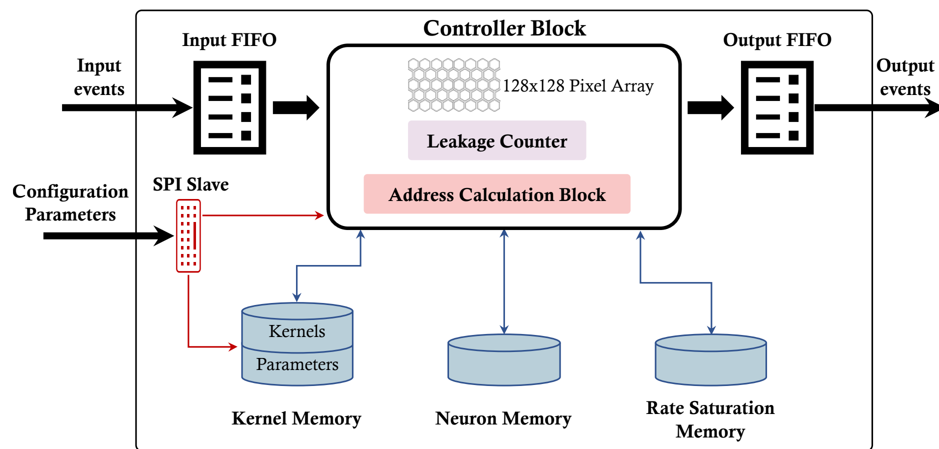


Figure 7.2: The convolutional Unit.

The node consists of three main blocks; a convolutional unit, an internal configuration block, and a router, as shown in Fig. 7.1. Each block has its own designated function that will be highlighted in this section.

7.1.1 *The Convolutional Unit*

The convolutional unit is where the transformation of input events into output events takes place. It computes the convolution between an input event $ev_{in}(t, x, y, p, k)$ and a kernel $w_k(x, y)$ to produce an output event $ev_{out}(t, x, y, p)$, where t is time, x and y are the address coordinates, p is the polarity of the event, and k is the kernel ID in the kernel memory. Shown in Fig. 7.2, the convolutional unit consists of a 128×128 I&F neuron (pixel) array, a few memory blocks, FIFO input and output registers, a controller block and an Serial Peripheral Interface (SPI) block.

To understand what each block is responsible for, we must look at the features of the node. A brief description of these features is given next.

7.1.1.1 *Unit Parameters*

The convolutional unit has a set of adjustable parameters, some of which need to be assigned before implementation, while others can be set dynamically after. By tuning these parameters, the unit can be configured according to the needs of the application. These parameters can be divided into two groups:

1. **Pre-implementation Parameters**
   These parameter need to be set before the hardware implementation of the network. They are:

   a) ***Address range of input events:*** Since the unit communicates through an AER protocol, it is very important to specify the address space of the events. For input events, the address range is identified by x_{in}^{max} and y_{in}^{max}, which are the maximum values of x and y that the input event can have.

   b) ***Address range of output events:*** As for the output events, the address range is defined by the number of pixels used out of the 128×128 array. This corresponds to x_{out}^{max} and y_{out}^{max}, i.e., the maximum values x and y the output event can take.

   c) ***Neuron memory size:*** The neuron memory is where the state of each pixel is stored after every convolution operation. The size is assigned though the number of pixels used, $n_x \times n_y$ and the number of bits needed to store the state, n_{bits}.

   d) ***Kernel memory size:*** The memory assigned to the kernels of the convolution is divided into two parts. The first part holds the kernel weights, and its size is assigned by the number of kernels N_k and the maximum acceptable size of a kernel, $x_k^{max} \times y_k^{max}$. The second part of the memory holds the kernel parameters, i.e., the size of the kernel and the center shift of the kernel position.

   e) ***Rate saturation memory size:*** The rate saturation mechanism imposes a maximum frequency on the output of the convolutional unit, essentially implementing the refractory period feature of biological neurons. This refractory period is stored in a designated memory block whose size is specified by the number of pixels in the block, $n_x \times n_y$, and the number of bits needed to represent the period.

   f) ***Rate saturation period range:*** In addition to the actual refractory period, the range of values it can take is also specified before implementation, T_R^{min} and T_R^{max}.

2. **Post-implementation Parameters**
   In contrast, these parameters are sent to the convolutional unit through the ***SPI block***, which means that they can be modified after the hardware implementation. The post-implementation parameters are:

   a) ***Threshold:*** The pixels are I&F neurons that fire when the threshold, Th is passed. As stated in Section 2.1.1.2, events in a biological NN can be either excitatory or inhibitory, i.e., cause the membrane potential to increase or decrease, respectively. Following the same paradigm, in this implementation, the events can be positive (excitatory) or negative (inhibitory), hence there is an upper and lower threshold. However, since only positive numbers are used in the FPGA implementation, the actual positive and negative thresholds inside the implementation are 0 and $2 \times Th$, and Th marks the reset value of the pixels.

   b) ***Leakage parameters:*** Leakage is the decay of the membrane potential in between incoming spikes. In this implementation, leakage is introduced to the network as a pulse of period T_{leak} and amplitude N_{leak} that is added to or subtracted from the value of the neuron state. The global leakage mechanism is further discussed in Section 7.1.1.3.

   c) ***Kernel values and parameters:*** The weights of the kernels and the kernel parameters that will be written in the kernel memory.

   d) ***Rate saturation interval:*** The actual value of the refractory period T_R inside the pre-defined range.

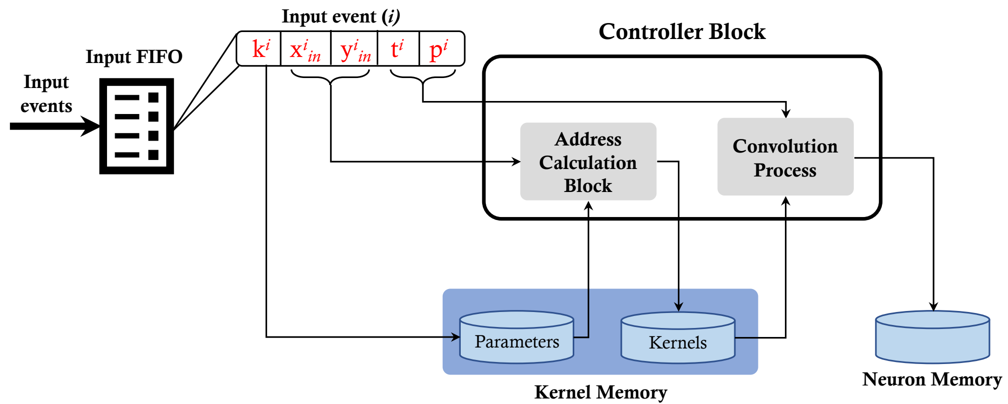


Figure 7.3: A conceptual figure showing the process of reading input events from the FIFO by the *controller block*.

7.1.1.2 *The Convolution Operation*

The input FIFO register receives input events and stores them until they are read by the ***controller block***. When an input event arrives, the controller reads its parameters, i.e., the address coordinates (x_{in}, y_{in}), the polarity (p), the time stamp (t), and the kernel ID (k), and two operations are executed. First, the coordinates of the pixels affected by the incoming event are calculated by the ***address calculation block***, using the address of the input and the kernel parameters corresponding to kernel k^i. Afterwards, the kernel weights are retrieved from the kernel memory and the pixels are updated with the results of the multiplication of these kernels with the input event. This process is conceptually illustrated in Fig. 7.3.

7.1.1.3 *Global Leakage*

The ***controller block*** houses a synchronous ***global counter*** that counts clock cycles and is responsible for implementing the leakage feature. When the counter reaches the value of the leakage period T_{leak}, a pulse of amplitude N_{Leak} is sent to all the pixels of the convolutional unit. If the value of a pixel is greater than the reset value Th, N_{leak} is subtracted from that value, and if the pixel value is less than Th, on the other hand, the value N_{leak} is added to the value in the memory. I.e., depending on the value stored in each pixel memory relative to the reset voltage Th, the value of N_{leak} is either added or subtracted to bring the value closer to Th but without actually crossing it, as shown by the red lines in Fig. 7.4. This process ensures that all neurons converge towards the reset value, preserving the temporal correlation property between successive input events.

7.1.1.4 *Rate Saturation Mechanism*

Another important feature of this convolutional unit is the imposition of the refractory period property found in biological neural networks. As explained in Chapter 2, the refractory period is the minimum interval between two consecutive spikes. To implement this function in the convolutional unit, the time at which a pixel produces a spike, t_0, is read from the global counter. When the threshold is crossed again by the same pixel at time t_1, this time is compared to $(t_{lim} = t_0 + T_R)$ to check if the refractory period has passed. If the period has indeed passed, the pixel is allowed to produce an output event, and the register holding the t_{lim} value is cleared. If not, the threshold value is held in the register until the next input event comes and the pixel is allowed to spike again. This process is illustrated in Fig. 7.4.

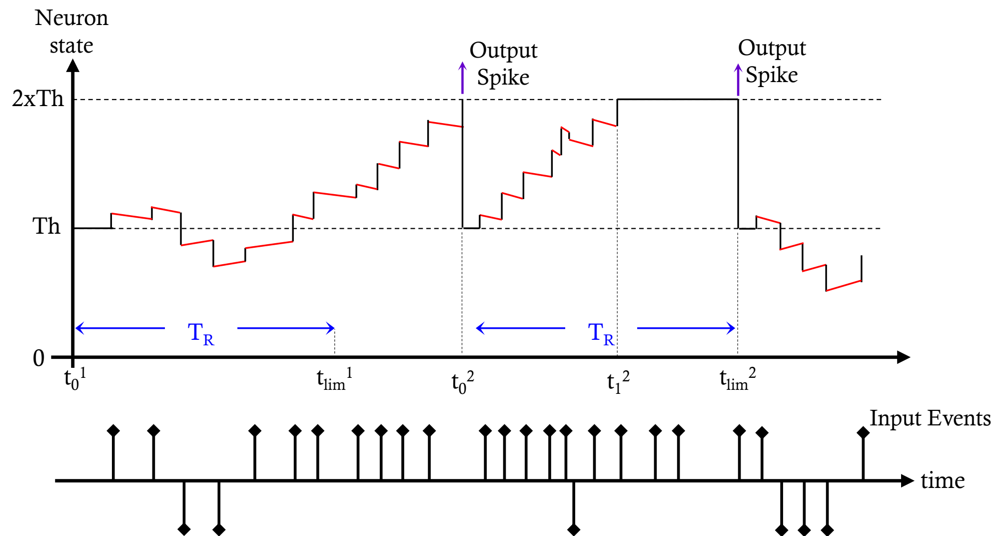


Figure 7.4: The change in the state of the neuron with the incoming spikes. The red lines represent the effect of the leakage mechanism, and T_R represents the refractory period that stops the neuron from spiking too soon.

7.1.1.5 *Output Event Generation & Traffic Control*

With every incoming event, the values of the corresponding pixels are updated and compared to the positive and negative thresholds. If the value of a threshold is reached by a pixel and the condition imposed by the rate saturation mechanism is fulfilled, i.e., the refractory period has passed, the pixel produces an output event with address (x_{out}, y_{out}) and polarity p_{out} and write it in the output FIFO register.

Because of the AER communication protocol, the convolutional unit receives a flow of input events and sends out another flow of output events through the FIFO registers. Using a $full_{FIFO}$ signal, a dropping mechanism is implemented to control the flow of these events. When the register is full, the $full_{FIFO}$ signal is activated, and the incoming events are discarded until the events in the FIFO are read by the controller and there is room for more events. While this implies that the output events would get down-sampled and some information will eventually be lost, the spatio-temporal correlation of the passing events is preserved, keeping the integrity of the information carried in these events.

7.1.2 *The Router*

The second block in a convolutional node is the router. In hardware implementation of neural networks, the highly dense connectivity required between neurons poses a challenge in terms of on-chip area. Routers provide a practical solution to this problem by acting as a network layer between the convolutional units and the physical implementation. They are responsible for handling the transmission of the events from their origin to their destination according to the routing scheme [55].

In this design, the destination-driven addressing scheme is adopted. This means that for every event, there is a routing header that carries the x and y coordinates of the destination node in the mesh distribution of nodes, which will be illustrated later in Section 7.2.2. The router receives events from two main sources, either the local convolutional unit within the node, or a neighboring node through one the four ports. For external events, the router analyzes the header address and compares it to the address of the local node. If they match, then the event is directed to the local convolutional unit. If not, the router locates the destination node with respect to the local one and decides the port through which the event should be passed. In the case of a local event, i.e., coming from the local convolutional unit, the router adds the appropriate node address in the routing header according to the pre-programmed routing table, and sends the event through the corresponding output port.

The block diagram of the destination-driven router is shown in Fig. 7.5, featuring a highly parallel architecture where every port, in addition to the local unit, has its own set of blocks that can handle its traffic. The basic building blocks of the router are:

1. **RouterIN:**
   There are four *RouterIN* blocks in the router, each one is responsible for receiving external events from a certain port. The *RouterIN* performs the comparison between the incoming event address and the local address of the node and decides whether to send the event to the local unit through the local arbiter, or to pass it on to one of the other three ports through their respective arbiter blocks.

2. **Arbiter:**
   The *Arbiter* handles the output path of events coming from the *RouterIN* blocks or the local convolutional unit. Moreover, it regularizes the traffic of outgoing events by a prioritizing algorithm that does not allow two successive outputs from the same interface. The destination-driven router has five *Arbiters*, four of which are assigned to the four

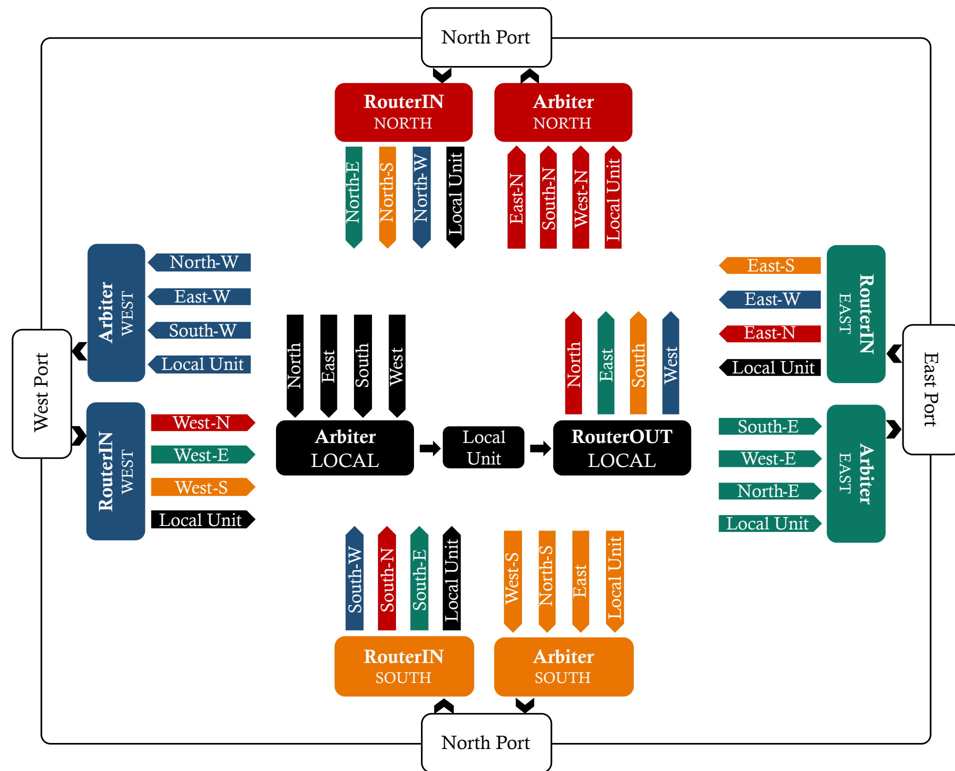


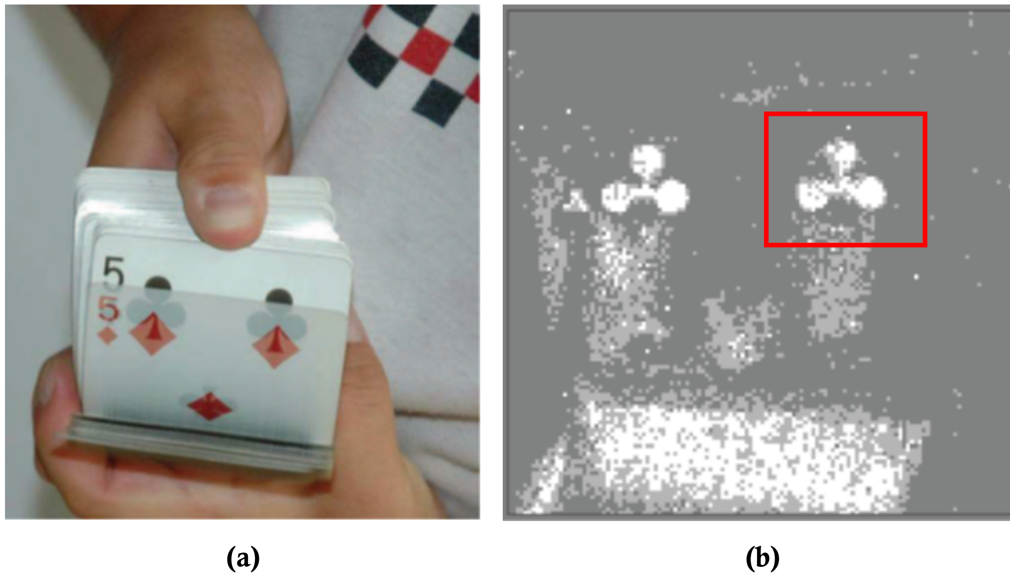
Figure 7.5: Block diagram of the destination-driven router.

ports of the node, and the fifth is assigned to carry events to the local convolutional unit.

3. **RouterOUT:**
   The router is equipped with a single *RouterOUT* block responsible for preparing events coming from the local convolutional unit to be sent out to their final destinations. Through the routing table, which is configured before the network operation, the appropriate header is added to the events, and they are forwarded to one of the four *Arbiters* to be sent through the respective port.

7.1.3 *The Configuration Block*

The configuration data of the convolutional node is sent through an SPI. Each parameter value is sent with an index indicating its ID so that it can be correctly interpreted by the controller block. In addition to the unit parameters and the kernel weights, the router configuration is also done via the SPI. The router needs two important settings, the local address of the node and the routing table information necessary for redirecting events through the ports of the convolutional node.



(a) (b)

Figure 7.6: Generation of the poker card symbols dataset: (a) Picture taken with a frame-driven camera. (b) 2-D image obtained by collecting events in a 5ms window [128]. The red square represents the 32x32 window that shows the centered used for the symbols dataset.

7.2 THE EXPERIMENT

7.2.1 *The Poker-Card Symbols Dataset*

To test the network, a dataset of poker card symbols was used, as shown in Fig. 7.6. A deck of 40 poker cards was presented in front of a DVS sensor during a 1s period. The events were recorded and processed to present a 32×32 pixel window that shows only the centered symbol as shown by the red square in Fig. 7.6. The stimulus has 174,644 events in total with a duration of 950ms and an average speed of 184 k events per second.

To verify the traffic control feature and its effects on the network behavior, the dataset was presented in different slowed down versions, ranging from 100% to 1% of the original speed.

7.2.2 *The Convolutional SNN*

The convolutional SNN used was originally designed by Pérez-Carrasco *et al.* [128], trained in software in a frame-based format and then the weights and parameters were mapped into spiking form by transforming frame-based values into spiking rates. Afterwards, the weights and parameters were scaled and rounded, and then tuned to make up for the discrepancies between hardware and software implementations using simulated annealing as an optimization algorithm.

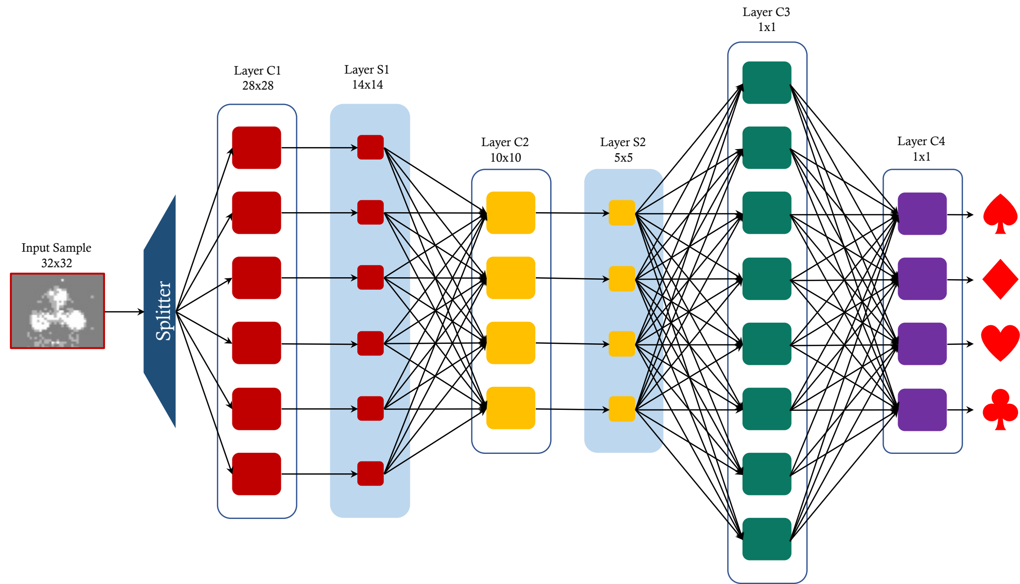


Figure 7.7: Schematic block diagram of the convolutional SNN used for this experiment.

The network is shown in Fig. 7.7, which consists of 4 convolutional layers and 2 sub-sampling layers that reduce the address space between layers by a factor of 2. The 2-D layout of the FPGA implementation is shown in Fig. 7.8, where the nodes are arranged in a 6×4 mesh with bidirectional connections between the routers of each block and its immediate neighbors. The network contains 22 nodes, every node carries an identification number corresponding to its y and x address in the mesh, and its color indicates the respective layer in the network. Nodes $(5, 4)$ and $(6, 4)$ are not part of the original network, they are added just for routing purposes, and they do not perform any processing. There is an extra *splitter* block at the input side which receives signals coming to the network and sends them to the nodes of the first layer, and a *merger* block at the output side of the network which sends signals out of the network. As explained in Section 7.1.2, the internal routers are configured so that events can propagate to their destination address through the nodes via the shortest path possible.

7.2.3 *The Experimental Setup & Results*

For the hardware implementation of this network on FPGA in [127], no simulation test bench was constructed. The experimental setup used is shown in Fig. 7.9, where the FPGA board is connected to several interfaces. The configuration of the network is accomplished through a microcontroller board that delivers the parameters and the kernels from the computer to the FPGA through the SPI interface. An AER data player

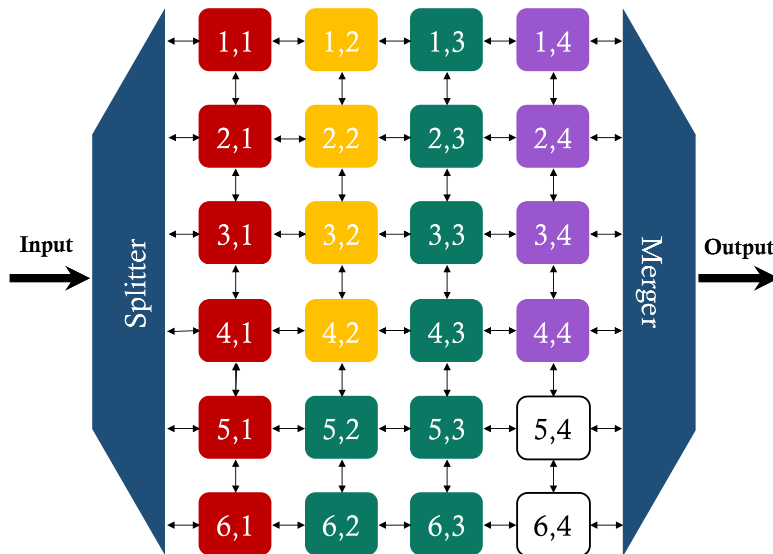


Figure 7.8: Schematic of the 2-D mesh implementation of the convolutional SNN on the FPGA. Each box represents a convolutional node in the network. The numbers indicate the address of the node in the mesh (y,x) and the colors represent the different layers of the network.

board [129] receives AER events and delivers them to the FPGA to be processed by the network. Afterwards, the events are sent out to the computer through a USB port on another board.

Using the above setup, the network was stimulated using the different versions of the dataset. At 1% of the original speed, all the events in the dataset are processed by the network and a classification rate of 97.5% is achieved. When the real-time version of the network was used, the traffic control mechanism discarded over 80% of the events. However, the network managed to reach a classification rate of 70%.

7.3 PUTTING THE HARDWARE IN THE LOOP

Hardware-in-the-Loop (HIL) simulation is a technique used to develop and test complex real-time systems. It provides an efficient platform that incorporates the specifics and complexity of a hardware implementation into a real-time simulation framework. The main idea is to see how the system behaves in the real world without having to wait until it is actually integrated in the application. HIL is very popular in automotive industry where every electronic system deals with tons of data coming from real world sensors and ever changing conditions.

The case study presented in this chapter is based on a configurable event-driven convolutional node whose architecture allowed the implementation of large convolutional SNNs on cheap commercial FPGAs.

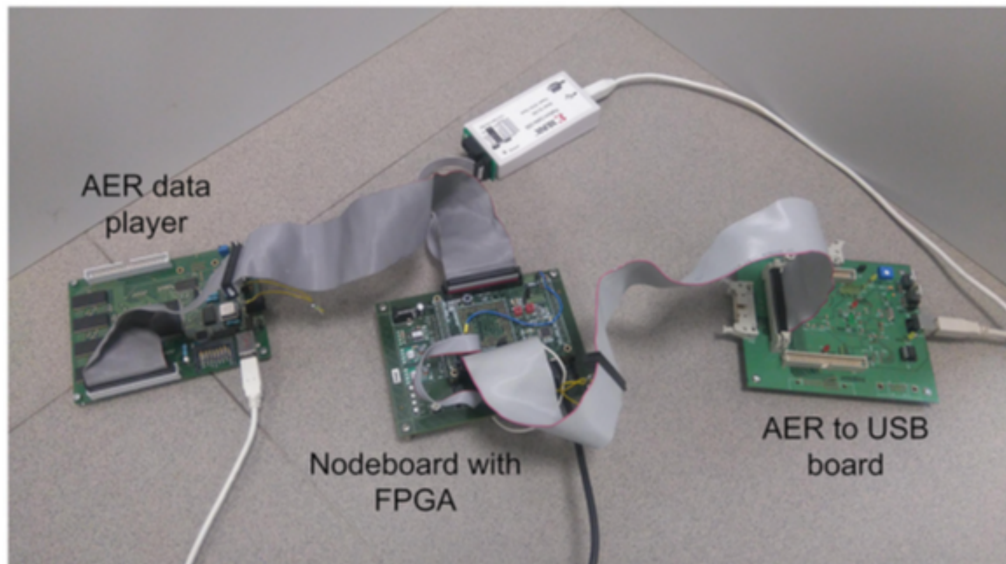


Figure 7.9: The experimental setup used for the convolutional SNN in [127].

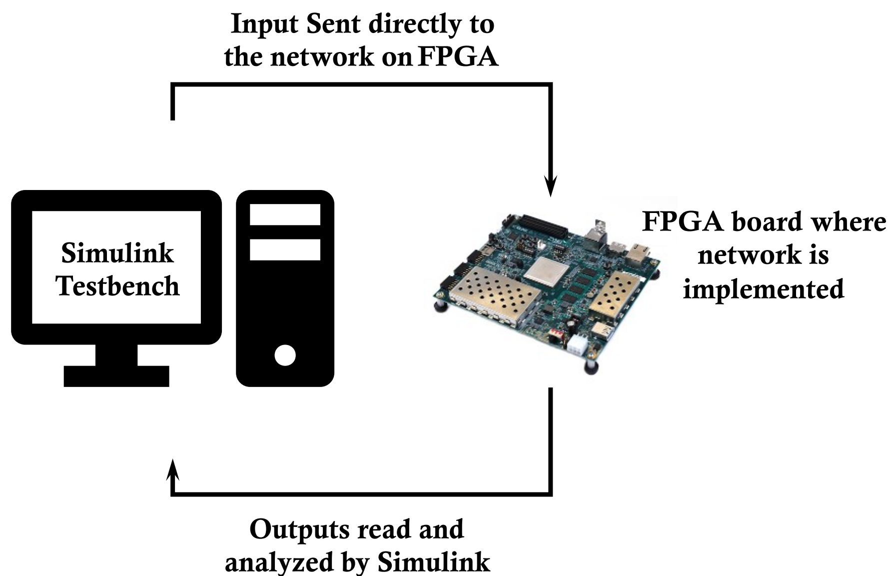


Figure 7.10: Hardware-in-the-Loop setup of the Xilinx FPGA board.

However, to communicate with the board for configuration and measurements, a complex setup had to be used in [127] as shown in Fig. 7.9. The extra boards were custom-created for spike-based systems by Serrano-Gotarredona *et al.* [129], and hence not available for commercial use like the FPGA part. Hence, the setup in [127] cannot realize its full potential as a generic platform.

In this work, we turn the FPGA implementation of this design into a standalone entity without the need for any extra custom hardware. Using Hardware Description Language (HDL) Coder™ in Matlab, the **Zynq® UltraScale+™ MPSoC ZCU**104 FPGA board was put into the simulation

loop and directly controlled from the computer. The HIL setup is shown in Fig. 7.10.

HDL Coder™[130] is a Matlab toolbox that can primarily generate VHDL code from Matlab functions and Simulink Models, automating the programming of various FPGAs using third party synthesis tools, such as Xilinx ISE and Vivado for Xilinx boards. It can also generate test benches for verifying VHDL designs along with the HDL Verifier™toolbox and third-party simulation tools such as ModelSim. This tool provides a comprehensive environment that can run HDL simulations using the FPGA-in-the-Loop feature that tests the design of the actual hardware.

Such a setup has many advantages for this design in general, and for our work in particular. As discussed in Section 7.2.2, the network demonstrated on the convolutional node presented above has transformed from a conventional ANN into spiking form. The network weights and parameters needed to be adjusted and optimized for use on the FPGA. The new HIL setup can skip all these steps by performing training directly with the FPGA in the loop. That way, the resulting values would be ready for use with no further processing. In addition, it will be possible to experiment with spiking datasets, such as the N-MNIST and the DVS-gesture datasets, in order to realize the full potential of the event-based node.

Using the new HIL setup will also be of great benefit in fault injection and simulation experiments. In Chapter 2, we presented a body of work concerned with fault injection and fault tolerance of ANNs, the majority of which were carried out at the software layer. With the scaling-up of neural networks, comprehensive testing that covers multiple fault scenarios becomes prohibitive, in terms of time and cost. Meanwhile, the same operation carried on in hardware would be much faster, allowing more extensive fault injection campaigns and thorough analysis of reliability and fault resilience capabilities.

8 CONCLUSIONS

The deployment of AI hardware accelerators in a variety of applications including safety-critical ones, requires assessing their inherent reliability and developing cost-effective fault tolerance techniques. Because the main inspiration has always been the human brain, research has not stopped at the immense progress achieved by conventional ANNs and deep learning. The trend is moving more and more towards biologically inspired architectures, materials, mechanisms, and learning techniques, hoping that if we can create circuits that behave like the brain, one day we will be able to achieve its performance levels. While we have not come close, neuromorphic systems and spike-based neural networks are certainly a step in the right direction.

With their massively parallel architectures and distributed computations, ANNs can tolerate a certain degree of inaccuracy in their calculations or noise in their inputs. However, the myth of total immunity to faults of any kind is far from true. In the literature, several fault-injection experiments have been carried out to evaluate the tolerance of ANNs to various fault types and to take measures towards ensuring their reliability. However, most of the work is focused on conventional level-based ANNs since they are predominant platforms for AI and deep learning.

In this thesis, we explore the inherent fault tolerance of SNNs at different levels of abstraction. We perform fault injection experiments and try to link transistor-level and behavioral-level together towards generic and comprehensive fault tolerance techniques.

8.1 THESIS CONTRIBUTIONS

Thesis contributions can be summarized as follows:

IN CHAPTER 3 we proposed a compact behavior-oriented BIST architecture for an analog biologically inspired spiking neuron that can be shared among all neurons in the neural network. This architecture

consists of a ramp generator block that controls two bias voltages to produce all four different firing patterns at the output in the fault-free case, and a digital block that checks if any of these patterns is missing, in which case it flags a fault detection. The BIST achieves a test yield of 70% in the case of process variations and over 90% defect coverage.

IN CHAPTER 4 we performed Monte Carlo analysis and defect simulation for an analog I&F neuron at transistor level. Then we observed and categorized the resulting faulty behaviors into catastrophic ones that destroy the neuron to translate transistor-level faults into behavioral-level faults and errors. We then used our observations to form a comprehensive behavioral-level fault model that can be used to test spiking neurons regardless of their implementation.

IN CHAPTER 5 we presented a taxonomy of faults and an abstract fault model based on SNNs, in addition to a fault-injection framework that allows an accelerated large-scale analysis of fault effects on the classification accuracy. We performed fault-injection experiments on two SNNs specifically designed for the classification of the NMNIST dataset and the DVS-gesture dataset. The experiments demonstrate that the criticality of faults and the severity of their effect on the circuit performance depend on a combination of multiple factors: the fault type, fault location in the hierarchy of the SNN, and the relative depth of the network architecture.

IN CHAPTER 6 we leveraged the findings from our large-scale fault injection experiments to develop some possible fault tolerance solutions that can be implemented in a hardware SNN. We proposed cost-effective fault tolerance strategies tailored to address critical faults in the SNN consisting in multiple layers of protection. Firstly, we proposed passive fault tolerance based on dropout to nullify the effect of certain faults, then we projected several active fault tolerance techniques to detect and recover from the remaining critical faults.

IN CHAPTER 7 we prepare to demonstrate the ideas and experiments presented in this thesis on an actual hardware convolutional neural network. We presented the configurable event-driven network designed to run on an FPGA platform and highlighted the strategy we followed to transform it into an autonomous hardware-in-the-loop platform in an effort to accelerate simulations, training and fault-injection experiments.

8.2 FUTURE PERSPECTIVES

As extensive as the work carried out in this thesis was, it can never be complete. There are many axes along which this work can be extended.

To complete the picture and validate all the ideas presented in this work, the hardware platform presented in Chapter 7 needs some preparations to be used to demonstrate the fault injection experiments and the proposed fault tolerance ideas. On top of the original network that classifies poker card symbols, we plan on using the convolutional node to build a completely configurable network that can be used to implement any SNN architecture. Putting the hardware-in-the-loop will allow spike-based training with the FPGA incorporated directly in the training loop to avoid any extra adaptation or optimization of the network parameters, in addition to the use of benchmark spiking datasets such as the N-MNIST and the DVS-gesture datasets. Afterwards, the hardware implementation should be ready as a platform for future experimentation.

Throughout the thesis, most of the fault injection experiments we carried out focused on neurons, since they are the main computational nodes of SNNs. However, to comprehensively assess the fault tolerance capabilities of SNNs, synapses would have to be considered as well. In Chapter 5, dead and saturated faults in the synaptic weights were considered for the rather small N-MNIST SNN. In the future part of this experiment, we plan on studying synaptic fault models that can accurately represent possible hardware faults in SNN implementations. Given that the number of synapses is usually much higher than that of neurons, we would need to optimize these fault models to allow for cost-effective large-scale synaptic fault injection experiments on deep SNNs. Moreover, we are planning on extending the study to cover the resilience of SNNs to latent faults caused by natural aging of the hardware components.

Another future perspective would be to work on improving the fault tolerance ideas presented in Chapter 6. As far as fault tolerance goes, the ideas presented in this work focused only on neuron faults. We plan to continue with the optimization of fault tolerance capabilities for neurons in deep SNNs and extend them to include synaptic connections as well.

BIBLIOGRAPHY

[1] A. M. Turing, “On computable numbers, with an application to the entscheidungs problem,” *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937 (cit. on p. 1).

[2] ——, “Intelligent machinery,” *NPL Math. Div.*, 1948 (cit. on p. 1).

[3] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, 2. MIT press Cambridge, 2016, vol. 1 (cit. on p. 2).

[4] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “Cudnn: efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014 (cit. on p. 2).

[5] V. Braitenberg and A. Schüz, “Cortex: statistics and geometry of neuronal connectivity. springer,” *New York*, 1998 (cit. on p. 2).

[6] T. N. Theis and H.-S. P. Wong, “The end of moore’s law: a new beginning for information technology,” *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017 (cit. on p. 2).

[7] G. Lacey, G. W. Taylor, and S. Areibi, “Deep learning on FPGAs: past, present, and future,” *arXiv preprint arXiv:1602.04283*, 2016 (cit. on p. 3).

[8] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12 (cit. on p. 3).

[9] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016 (cit. on p. 3).

[10] C. Mead, “Neuromorphic electronic systems,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990 (cit. on p. 3).

[11] W. Maass, “Noise as a resource for computation and learning in networks of spiking neurons,” *Proceedings of the IEEE*, vol. 102, no. 5, pp. 860–880, 2014 (cit. on p. 3).

[12] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–12 (cit. on p. 3).

[13] E. M. Izhikevich and G. M. Edelman, "Large-scale model of mammalian thalamocortical systems," *Proceedings of the national academy of sciences*, vol. 105, no. 9, pp. 3593–3598, 2008 (cit. on p. 3).

[14] C.-S. Poon and K. Zhou, "Neuromorphic silicon neurons and large-scale neural networks: challenges and opportunities," *Frontiers in Neuroscience*, vol. 5, 2011, Article 108 (cit. on p. 3).

[15] J. Hsu, "IBM's new brain [news]," *IEEE Spectrum*, vol. 51, no. 10, pp. 17–19, 2014. DOI: 10.1109/MSPEC.2014.6905473 (cit. on pp. 3, 17).

[16] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014. DOI: 10.1109/JPROC.2014.2304638 (cit. on pp. 3, 17).

[17] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016 (cit. on pp. 3, 17).

[18] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, *et al.*, "Loihi: a neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018 (cit. on pp. 3, 17).

[19] L. Gomes, "Special report : can we copy the brain? - the neuromorphic chip's make-or-break moment," *IEEE Spectrum*, vol. 54, no. 6, pp. 52–57, 2017. DOI: 10.1109/MSPEC.2017.7934233 (cit. on p. 3).

[20] L.-C. Wang, "Experience of data analytics in EDA and test—principles, promises, and challenges," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 6, pp. 885–898, 2017 (cit. on p. 4).

[21] H.-G. Stratigopoulos, "Machine learning applications in IC testing," in *Proc. IEEE European Test Symposium*, 2018 (cit. on p. 4).

[22] D. Maliuk, H.-G. Stratigopoulos, H. Huang, and Y. Makris, "Analog neural network design for RF built-in self-test," in *Proc. IEEE International Test Conference*, Paper 23.2, 2010 (cit. on p. 4).

[23] S. Lawrence, C. L. Giles, and A. C. Tsoi, "What size neural network gives optimal generalization? convergence properties of backpropagation," 1998, Technical report (cit. on p. 4).

[24] P. Kerlirzin and F. Vallet, “Robustness in multilayer perceptrons,” *Neural Computation*, vol. 5, no. 3, pp. 473–482, 1993 (cit. on p. 4).

[25] L. Anghel, D. Ly, G. D. Natale, B. Miramond, E. I. Vatajelu, and E. Vianello, “Neuromorphic computing - from robust hardware architectures to testing strategies,” in *Proc. IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018, pp. 176–179 (cit. on p. 4).

[26] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2000 (cit. on p. 5).

[27] S. Mitra and E. J. McCluskey, “Which concurrent error detection scheme to choose ?” In *Proc. IEEE International Test Conference*, 2000, pp. 985–994 (cit. on p. 5).

[28] M. Burns, G. W. Roberts, *et al.*, *An introduction to mixed-signal IC test and measurement*. Oxford University Press, 2001 (cit. on p. 5).

[29] C. Torres-Huitzil and B. Girau, “Fault and error tolerance in neural networks: a review,” *IEEE Access*, vol. 5, pp. 17 322–17 341, 2017 (cit. on pp. 5, 23, 84).

[30] R. Slater, “Fault injection,” *URL: http://www. cs. cmu. edu/˜ koopman/des_s99/fault_injection/index. html*, 1998 (cit. on p. 6).

[31] W. Maass, “Networks of spiking neurons: the third generation of neural network models,” *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997 (cit. on pp. 9–11).

[32] S. Thorpe, D. Fize, and C. Marlot, “Speed of processing in the human visual system,” *nature*, vol. 381, no. 6582, pp. 520–522, 1996 (cit. on pp. 10, 12).

[33] E. Rolls and M. Tovée, “Processing speed in the cerebral cortex and the neurophysiology of visual masking,” *Proceedings: Biological Sciences*, vol. 257, no. 1348, pp. 9–15, 1994 (cit. on p. 10).

[34] W. Gerstner, “Time structure of the activity in neural network models,” *Phys. Rev. E*, vol. 51, pp. 738–758, 1 1995 (cit. on pp. 10, 14, 51, 52).

[35] E. M. Izhikevich, *Dynamical systems in neuroscience*. MIT press, 2007 (cit. on p. 10).

[36] E. D. Adrian and Y. Zotterman, “The impulses produced by sensory nerve endings,” *The Journal of physiology*, vol. 61, no. 4, pp. 465–483, 1926 (cit. on p. 12).

[37] W. Gerstner and W. M. Kistler, *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge University Press, 2002 (cit. on p. 12).

[38] S. Thorpe, A. Delorme, and R. Van Rullen, "Spike-based strategies for rapid processing," *Neural networks*, vol. 14, no. 6-7, pp. 715–725, 2001 (cit. on p. 12).

[39] R. S. Johansson and I. Birznieks, "First spikes in ensembles of human tactile afferents code complex spatial fingertip events," *Nature neuroscience*, vol. 7, no. 2, pp. 170–177, 2004 (cit. on p. 12).

[40] H. T. Chen, K. T. Ng, A. Bermak, M. K. Law, and D. Martinez, "Spike latency coding in biologically inspired microelectronic nose," *IEEE transactions on biomedical circuits and systems*, vol. 5, no. 2, pp. 160–168, 2011 (cit. on p. 12).

[41] R. Van Rullen and S. Thorpe, "Rate coding versus temporal order coding: what the retinal ganglion cells tell the visual cortex," *Neural Computation*, vol. 13, no. 6, pp. 1255–1283, 2001 (cit. on p. 12).

[42] G. Indiveri, B. Linares-Barranco, T. Hamilton, *et al.*, "Neuromorphic silicon neuron circuits," *Frontiers in Neuroscience*, vol. 5, 2011, Article 73 (cit. on p. 13).

[43] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *CoRR*, vol. abs/1705.06963, 2017. arXiv: 1705.06963 (cit. on pp. 13, 15).

[44] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004 (cit. on p. 13).

[45] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952 (cit. on p. 13).

[46] C. Morris and H. Lecar, "Voltage oscillations in the barnacle giant muscle fiber," *Biophysical Journal*, vol. 35, pp. 193–213, 1981 (cit. on p. 13).

[47] R. FitzHugh, "Impulses and physiological states in models of nerve membrane," *Biophysical Journal*, vol. 1, pp. 445–466, 1961 (cit. on p. 13).

[48] J. Nagumo, S. Arimoto, and S. Yoshizawa, "An active pulse transmission line simulating nerve axon," *Proceedings of the IRE*, vol. 50, no. 10, pp. 2061–2070, 1962 (cit. on p. 13).

[49] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003 (cit. on pp. 13, 29).

[50] ——, "Resonate-and-fire neurons," *Neural networks*, vol. 14, no. 6-7, pp. 883–894, 2001 (cit. on p. 14).

[51] P. E. Latham, B. Richmond, P. Nelson, and S. Nirenberg, "Intrinsic dynamics in neuronal networks. i. theory," *Journal of neurophysiology*, vol. 83, no. 2, pp. 808–827, 2000 (cit. on p. 14).

[52] R. Brette and W. Gerstner, "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity," *Journal of Physiology*, vol. 94, pp. 3637–3642, 2005 (cit. on p. 14).

[53] S.-C. Liu, T. Delbruck, G. Indiveri, A. Whatley, and R. Douglas, *Event-based neuromorphic systems*. John Wiley & Sons, 2014 (cit. on pp. 14, 85).

[54] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, 2000 (cit. on p. 14).

[55] C. Zamarreño-Ramos, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco, "Multicasting mesh aer: a scalable assembly approach for reconfigurable neuromorphic structured aer systems. application to convnets," *IEEE transactions on biomedical circuits and systems*, vol. 7, no. 1, pp. 82–102, 2012 (cit. on pp. 14, 93).

[56] L. A. Camuñas-Mesa, T. Serrano-Gotarredona, and B. Linares-Barranco, "Event-driven sensing and processing for high-speed robotic vision," in *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*, IEEE, 2014, pp. 516–519 (cit. on p. 16).

[57] D. O. Hebb, "The organization of behavior; a neuropsychological theory," *A Wiley Book in Clinical Psychology*, vol. 62, p. 78, 1949 (cit. on p. 16).

[58] H. Markram, J. Lübke, M. Frotscher, and B. Sakmann, "Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps," *Science*, vol. 275, no. 5297, pp. 213–215, 1997 (cit. on p. 16).

[59] G.-q. Bi and M.-m. Poo, "Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type," *Journal of neuroscience*, vol. 18, no. 24, pp. 10 464–10 472, 1998 (cit. on p. 16).

[60] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware spiking neurons design: analog or digital?" In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2012, pp. 1–5 (cit. on p. 16).

[61] Y. LeCun, "The mnist database of handwritten digits," `http://yann.lecun.com/exdb/mnist/`, 1998 (cit. on pp. 16, 49, 53).

[62] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009 (cit. on p. 16).

[63] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in neuroscience*, vol. 9, p. 437, 2015 (cit. on pp. 17, 49, 53).

[64] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, *et al.*, "A low power, fully event-based gesture recognition system," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 7243–7252 (cit. on pp. 17, 49, 55, 57).

[65] Y. Hu, H. Liu, M. Pfeiffer, and T. Delbruck, "Dvs benchmark datasets for object tracking, action recognition, and object recognition," *Frontiers in neuroscience*, vol. 10, p. 405, 2016 (cit. on p. 17).

[66] T. Serrano-Gotarredona and B. Linares-Barranco, "Poker-dvs and mnist-dvs. their history, how they were made, and other details," *Frontiers in neuroscience*, vol. 9, p. 481, 2015 (cit. on p. 17).

[67] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986 (cit. on p. 17).

[68] S. B. Shrestha and G. Orchard, "SLAYER: spike layer error reassignment in time," in *Advances in Neural Information Processing Systems*, 2018, pp. 1412–1421 (cit. on pp. 17, 51, 52, 56).

[69] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in neuroscience*, vol. 10, p. 508, 2016 (cit. on p. 17).

[70] Y. Jin, W. Zhang, and P. Li, "Hybrid macro/micro level backpropagation for training deep spiking neural networks," *arXiv preprint arXiv:1805.07866*, 2018 (cit. on p. 17).

[71] S. Yu, "Neuro-inspired computing with emerging nonvolatile memories," *Proceedings of the IEEE*, vol. 106, no. 2, pp. 260–285, 2018 (cit. on p. 18).

[72] S. Yu, *Neuro-inspired computing using resistive synaptic devices*. Springer, 2017 (cit. on p. 18).

[73] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, 2015 (cit. on p. 18).

[74] L. A. Camuñas-Mesa, B. Linares-Barranco, and T. Serrano-Gotarredona, "Neuromorphic spiking neural networks and their memristor-CMOS hardware implementations," *Materials*, vol. 12, no. 17, p. 2745, 2019 (cit. on p. 18).

[75] G. Bolt, "Fault models for artificial neural networks," in *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, IEEE, 1991, pp. 1373–1378 (cit. on p. 19).

[76] P. Chandra and Y. Singh, "Fault tolerance of feedforward artificial neural networks-a framework of study," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, IEEE, vol. 1, 2003, pp. 489–494 (cit. on p. 19).

[77] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2012, pp. 356–367 (cit. on p. 20).

[78] Y. Tohma and Y. Koyanagi, "Fault-tolerant design of neural networks for solving optimization problems," *IEEE transactions on computers*, vol. 45, no. 12, pp. 1450–1455, 1996 (cit. on p. 20).

[79] A. Ruospo, D. Piumatti, A. Floridia, and E. Sanchez, "A suitability analysis of software based testing strategies for the on-line testing of artificial neural networks applications in embedded devices," *IEEE International Symposium on On-Line Testing and Robust System Design*, 2021 (cit. on p. 20).

[80] S. Motaman, S. Ghosh, and J. Park, "A perspective on test methodologies for supervised machine learning accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 562–569, 2019 (cit. on p. 20).

[81] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, IEEE, 2018, pp. 1–6 (cit. on p. 20).

[82] A. Gebregiorgis and M. B. Tahoori, "Testing of neuromorphic circuits: structural vs functional," in *Proc. IEEE International Test Conference*, Paper 3.2, 2019 (cit. on p. 20).

[83] G. Gambardella, J. Kappauf, M. Blott, C. Doehring, M. Kumm, P. Zipf, and K. Vissers, "Efficient error-tolerant quantized neural network accelerators," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, 2019, pp. 1–6 (cit. on p. 21).

[84] E.-I. Vatajelu, G. Di Natale, and L. Anghel, "Special session: reliability of hardware-implemented spiking neural networks (snn)," in *2019 IEEE 37th VLSI Test Symposium (VTS)*, IEEE, 2019, pp. 1–8 (cit. on p. 21).

[85] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2017, pp. 1–6 (cit. on pp. 21, 26).

[86] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang, "Accelerator-friendly neural-network training: learning variations and defects in rram crossbar," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 19–24 (cit. on p. 21).

[87] L. Xia, W. Huangfu, T. Tang, X. Yin, K. Chakrabarty, Y. Xie, Y. Wang, and H. Yang, "Stuck-at fault tolerance in rram computing systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 102–115, 2017 (cit. on p. 21).

[88] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12 (cit. on p. 21).

[89] S. Pandey, S. Banerjee, and A. Chatterjee, "Error resilient neuromorphic networks using checker neurons," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, IEEE, 2018, pp. 135–138 (cit. on pp. 22, 25).

[90] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*, IEEE, 2019, pp. 1–6 (cit. on p. 22).

[91] A. Ruospo, A. Bosio, A. Ianne, and E. Sanchez, "Evaluating convolutional neural networks reliability depending on their data representation," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, IEEE, 2020, pp. 672–679 (cit. on p. 22).

[92] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: a framework for quantifying the resilience of deep neural networks," in *ACM/ESDA/IEEE Design Automation Conference*, 2018 (cit. on p. 22).

[93] F. F. dos Santos, P. Navaux, L. Carro, and P. Rech, "Impact of reduced precision in the reliability of deep neural networks for object detection," in *2019 IEEE European Test Symposium (ETS)*, IEEE, 2019, pp. 1–6 (cit. on p. 22).

[94] M. A. Neggaz, I. Alouani, S. Niar, and F. Kurdahi, "Are cnns reliable enough for critical applications? an exploratory study," *IEEE Design & Test*, vol. 37, no. 2, pp. 76–83, 2019 (cit. on p. 22).

[95] M. D. Emmerson and R. I. Damper, "Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application," *IEEE transactions on neural networks*, vol. 4, no. 5, pp. 788–793, 1993 (cit. on p. 23).

[96] D. S. Phatak and I. Koren, "Complete and partial fault tolerance of feedforward neural nets," *IEEE Transactions on Neural Networks*, vol. 6, no. 2, pp. 446–456, 1995 (cit. on p. 23).

[97] C.-T. Chiu, K. Mehrotra, C. K. Mohan, and S. Ranka, "Robustness of feedforward neural networks," in *IEEE International Conference on Neural Networks*, IEEE, 1993, pp. 783–788 (cit. on p. 23).

[98] F. M. Dias and A. Antunes, "Fault tolerance improvement through architecture change in artificial neural networks," in *International Symposium on Intelligence Computation and Applications*, Springer, 2008, pp. 248–257 (cit. on p. 24).

[99] B. S. Arad and A. El-Amawy, "On fault tolerant training of feedforward neural networks," *Neural Networks*, vol. 10, no. 3, pp. 539–553, 1997 (cit. on p. 24).

[100] J. L. Bernier, J. Ortega, I. Rojas, and A. Prieto, "Improving the tolerance of multilayer perceptrons by minimizing the statistical sensitivity to weight deviations," *Neurocomputing*, vol. 31, no. 1-4, pp. 87–103, 2000 (cit. on p. 24).

[101] Y. Xiao, R.-B. Feng, C.-S. Leung, and J. Sum, "Objective function and learning algorithm for the general node fault situation," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 4, pp. 863–874, 2015 (cit. on p. 24).

[102] N. Wei, S. Yang, and S. Tong, "A modified learning algorithm for improving the fault tolerance of bp networks," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, IEEE, vol. 1, 1996, pp. 247–252 (cit. on p. 24).

[103] C. Khunasaraphan, K. Vanapipat, and C. Lursinsap, “Weight shifting techniques for self-recovery neural networks,” *IEEE transactions on neural networks*, vol. 5, no. 4, pp. 651–658, 1994 (cit. on p. 25).

[104] A. Hashmi, H. Berry, O. Temam, and M. Lipasti, “Automatic abstraction and fault tolerance in cortical microachitectures,” *ACM SIGARCH computer architecture news*, vol. 39, no. 3, pp. 1–10, 2011 (cit. on p. 25).

[105] J. Deng, Y. Rang, Z. Du, Y. Wang, H. Li, O. Temam, P. Ienne, D. Novo, X. Li, Y. Chen, *et al.*, “Retraining-based timing error mitigation for hardware neural networks,” in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2015, pp. 593–596 (cit. on p. 25).

[106] J. J. Zhang, T. Gu, K. Basu, and S. Garg, “Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator,” in *Proc. IEEE VLSI Test Symposium*, 2018 (cit. on p. 25).

[107] M. Liu, L. Xia, Y. Wang, and K. Chakrabarty, “Design of fault-tolerant neuromorphic computing systems,” in *2018 IEEE 23rd European Test Symposium (ETS)*, IEEE, 2018, pp. 1–9 (cit. on p. 26).

[108] M. Naeem, L. J. McDaid, J. Harkin, J. J. Wade, and J. Marsland, “On the role of astroglial syncytia in self-repairing spiking neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 10, pp. 2370–2380, 2015 (cit. on p. 26).

[109] J. Liu, J. Harkin, L. Maguire, L. McDaid, J. Wade, and M. McElholm, “Self-repairing hardware with astrocyte-neuron networks,” in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2016, pp. 1350–1353 (cit. on p. 26).

[110] J. Liu, J. Harkin, L. P. Maguire, L. J. McDaid, and J. J. Wade, “Spanner: a self-repairing spiking neural network hardware architecture,” *IEEE transactions on neural networks and learning systems*, vol. 29, no. 4, pp. 1287–1300, 2017 (cit. on p. 26).

[111] A. P. Johnson, J. Liu, A. G. Millard, S. Karim, A. M. Tyrrell, J. Harkin, J. Timmis, L. J. McDaid, and D. M. Halliday, “Homeostatic fault tolerance in spiking neural networks: a dynamic hardware perspective,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 2, pp. 687–699, 2017 (cit. on p. 26).

[112] S. A. El-Sayed, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Self-testing analog spiking neuron circuit," in *International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design*, Lausanne, Switzerland, 2019 (cit. on p. 27).

[113] B. W. Connors and M. J. Gutnik, "Intrinsic firing patterns of diverse neocortical neurons," *Trends Neurosciences*, vol. 13, no. 3, pp. 99–104, 1990 (cit. on p. 27).

[114] C. M. Gray and D. A. McCormick, "Chattering cells: superficial pyramidal neurons contributing to the generation of synchronous oscillations in the visual cortex," *Science*, vol. 274(5284), pp. 109–113, 1996 (cit. on p. 27).

[115] J. H. B. Wijekoon and P. Dudek, "Compact silicon neuron circuit with spiking and burtsing behaviour," *Neural Networks*, vol. 21, no. 4, pp. 524–534, 2008 (cit. on pp. 29, 30).

[116] S. Sunter, K. Jurga, and A. Laidler, "Using mixed-signal defect simulation to close the loop between design and test," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 12, pp. 2313–2322, 2016 (cit. on pp. 36, 39).

[117] B. Esen, A. Coyette, G. Gielen, W. Dobbelaere, and R. Vanhooren, "Effective DC fault models and testing approach for open defects in analog circuits," in *IEEE International Test Conference*, Paper 3.2, 2016 (cit. on pp. 36, 40).

[118] S. A. El-Sayed, T. Spyrou, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Spiking neuron hardware-level fault modeling," in *IEEE International Symposium on On-Line Testing and Robust System Design*, Naples (virtual), Italy, 2020 (cit. on p. 39).

[119] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological Cybernetics*, vol. 95, no. 1, pp. 1–19, 2006 (cit. on pp. 40, 41).

[120] R. Serrano-Gotarredona, T. Serrano-Gotarredona, A. Acosta-Jimenez, and B. Linares-Barranco, "A neuromorphic cortical-layer microchip for spike-based event processing vision systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 12, pp. 2548–2566, 2006 (cit. on p. 41).

[121] T. Spyrou, S. A. El-Sayed, E. Afacan, L. A. Camuñas-Mesa, B. Linares-Barranco, and H.-G. Stratigopoulos, "Neuron Fault Tolerance in Spiking Neural Networks," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble (virtual), France, 2021 (cit. on pp. 49, 73).

[122] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, *TensorFlow: large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/ (cit. on p. 51).

[123] F. Chollet et al., *Keras*, https://keras.io, 2015 (cit. on p. 51).

[124] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: an imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. W. et al., Ed., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 51).

[125] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998 (cit. on p. 53).

[126] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014 (cit. on p. 73).

[127] L. A. Camuñas-Mesa, Y. L. Domínguez-Cordero, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco, "A configurable event-driven convolutional node with rate saturation mechanism for modular ConvNet systems implementation," *Frontiers in neuroscience*, vol. 12, p. 63, 2018 (cit. on pp. 87, 96, 98).

[128] J. A. Pérez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. Linares-Barranco, "Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feed-forward convnets," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 11, pp. 2706–2719, 2013 (cit. on p. 95).

[129] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodríguez, L. Camuñas-Mesa, R. Berner, M. Rivas-Pérez, T. Delbruck, *et al.*, "CAVIAR: a 45k neuron, 5m synapse, 12g connects/s aer hardware sensory–processing–learning–actuating system for high-speed visual object recognition and tracking," *IEEE Transactions on Neural networks*, vol. 20, no. 9, pp. 1417–1438, 2009 (cit. on pp. 97, 98).

[130] *HDL Coder, Generate VHDL and Verilog code for FPGA and ASIC designs*, https://www.mathworks.com/products/hdl-coder.html (cit. on p. 99).