THE UNIVERSITY OF CHICAGO

RESILIENT DEEP LEARNING ACCELERATORS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
YI HE

CHICAGO, ILLINOIS
JUNE 2023

Copyright © 2023 by Yi He

All Rights Reserved

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF TABLES vi

ACKNOWLEDGMENTS vii

ABSTRACT viii

1 INTRODUCTION 1

2 EFFICIENT FUNCTIONAL IN-FIELD SELF-TEST FOR DEEP LEARNING ACCELERATORS 8
- 2.1 Introduction 9
- 2.2 Background and Related Work 12
  - 2.2.1 Deep Learning Accelerators 12
  - 2.2.2 In-Field Self-Test 13
- 2.3 Functional In-Field Self-Test Pattern Generation Technique 15
  - 2.3.1 Functional In-Field Self-Test Generation Technique for Compute Units 15
  - 2.3.2 Functional In-Field Self-Test Generation Technique for Control Units 16
- 2.4 Evaluation Methodology and Results 22
  - 2.4.1 Stuck-at Faults in Compute Units 22
  - 2.4.2 Transition Faults in Compute Units 30
  - 2.4.3 Mapping CMAC_compute's transition ATPG patterns to DNNs . . . 31
  - 2.4.4 Mapping CACC_compute's transition ATPG patterns to DNNs . . . 33
  - 2.4.5 Mapping SDP_compute's transition ATPG patterns to functional test programs 36
  - 2.4.6 Mapping PDP_compute's transition ATPG patterns to a DNN . . . 39
  - 2.4.7 Control Units 41
  - 2.4.8 Evaluation Results and Discussions 42
  - 2.4.9 In-Field Self-Test Coverage 42
  - 2.4.10 In-Field Self-Test Time Results 43
  - 2.4.11 Test storage results 46

3 FIDELITY: EFFICIENT RESILIENCE ANALYSIS FRAMEWORK FOR DEEP LEARNING ACCELERATORS 48
- 3.1 Introduction 49
- 3.2 The FIdelity Framework 51
  - 3.2.1 Insights and Novelty 52
  - 3.2.2 Reuse Factor Analysis 54
  - 3.2.3 Deriving Faulty Output Neuron Values 61
  - 3.2.4 FIdelity's Fault Injection Flow 62
  - 3.2.5 Broader applicability 66

3.3 FIdelity Framework Validation . . . 68
3.3.1 Accurate Software Fault Models for NVDLA . . . 68
3.3.2 Validation Methodology Details . . . 68
3.3.3 Results and Discussions . . . 69
3.4 Large-Scale Resilience Study . . . 70
3.5 Related Work . . . 75

4 UNDERSTANDING AND MITIGATING HARDWARE FAILURES IN DEEP LEARNING TRAINING SYSTEMS . . . 77
4.1 Introduction . . . 78
4.2 Background . . . 81
4.3 Methodology and Framework . . . 82
4.3.1 Accelerator Architecture for DNN Training . . . 83
4.3.2 Fault Injection Framework . . . 83
4.3.3 Experiment Setup . . . 85
4.4 Results . . . 87
4.4.1 Characterization of Hardware Failure Effects . . . 87
4.4.2 Detailed Analysis . . . 90
4.4.3 Other Results and Discussions . . . 99
4.5 Techniques to Tackle Hardware Failures in DNN Training Systems . . . 101
4.5.1 Detection . . . 102
4.5.2 Recovery . . . 102
4.5.3 Implementation and Evaluation . . . 104
4.6 Related Work . . . 105

5 UNDERSTANDING PERMANENT HARDWARE FAILURE EFFECTS IN DEEP LEARNING TRAINING SYSTEMS . . . 108
5.1 Introduction . . . 108
5.2 Related Work . . . 110
5.3 Methodology and Framework . . . 111
5.3.1 Fault Injection Framework . . . 111
5.3.2 Experiment setup . . . 112
5.4 Key Results . . . 113
5.4.1 Analysis on the Unexpected Training Outcomes . . . 115
5.4.2 Summary and Generalization . . . 117
5.5 New Mitigation Techniques . . . 118

6 CONCLUSION . . . 121

REFERENCES . . . 122

LIST OF FIGURES

2.1 NVDLA high level diagram. 14
2.2 Illustration of Algorithm 1. 25
2.3 Illustration of CACC_compute's operations for the DNN test programs generated by Algorithms 2 and 1. 28
2.4 Illustration on why applying the DNN generated by Algorithm 5 is equivalent to applying the corresponding transition ATPG pattern for CMAC_compute. . . . 34
2.5 Illustration on why applying the DNN generated by Algorithms 6 and 7 is equivalent to applying the corresponding transition ATPG pattern for CACC_compute. 36
2.6 Illustration on why applying the DNN generated by Algorithm 9 is equivalent to applying all transition ATPG patterns to all pooling_max units. 41
2.7 Illustration of Algorithm 9. 42

3.1 Datapath FF reuse factor analysis for (a) A NVDLA-like accelerator; (b) An Eyeriss-like accelerator. 58
3.2 Overview of the FIdelity framework. 62
3.3 Accelerator_FIT_Rate Values for Inception, Resnet, and Mobilenet. 72
3.4 Accelerator_FIT_Rate Values for Transformer & Yolo. 73
3.5 Accelerator_FIT_Rate Values Assuming that Global Control FFs are Protected. 74

4.1 Hardware failure examples in DNN training. 82
4.2 Four new unexpected latent outcomes observed from our experiments. 89
4.3 Percentages of training outcomes, normalized to the total number of experiments for each workload. 91
4.4 Characterization of fault propagation paths and effects. 92
4.5 Explanation of the three phases in the convergence trends of SlowDegrade and SharpSlowDegrade. The math symbols are defined in Eq. 1. 94
4.6 Contributions to unexpected outcomes for bit-flips in various FF groups. 99

5.1 The training/test accuracy trends of SharpDegrade. 115
5.2 Percentages of unexpected training outcomes, normalized to the total number of experiments for each software fault model and each network model. 115
5.3 Percentages of training outcomes for different bit positions where hardware failures occur. 116

LIST OF TABLES

2.1 Test coverage results for NVDLA. Fault models: single stuck-at faults and transition faults for compute units; single-variable control faults for control units. . . . 43
2.2 Functional in-field self-test time results for the compute units in NVDLA (clock frequency = 2.5GHz). . . . 45
2.3 Functional in-field self-test time results for the control units and total test time results (frequency = 2.5GHz). All results are in milliseconds. . . . 46
2.4 Test storage results for NVDLA. All results are in MBytes. . . . 47

3.1 Summary of reuse factor analysis for datapath FFs in DL accelerators. . . . 54
3.2 NVDLA software fault models for convolution (Conv), fully Connected (FC), & matrix multiplication (MatMul) layers. . . . 65
3.3 Workloads (DNN Layers) Used for FIdelity Validation. Precision: FP16. . . . 67
3.4 FIdelity's Fault Injection Experiment Setup. . . . 71

4.1 Fault injection framework and methodology. . . . 86
4.2 DNN training workloads. Optimizer: Adam (except for Resnet_SGD). Momentum value in batch normalization (BatchNorm) layers: 0.9 (except for Resnet_LargeDecay). 88
4.3 Unexpected outcomes in DNN training workloads. . . . 90
4.4 Necessary conditions for short-term/latent unexpected outcomes. iter. t is the iteration during which a fault is injected. . . . 97
4.5 Bounds derived from Algorithm 11 for target DNN training workloads . . . 104
4.6 Resilience properties of inference vs. training. . . . 105

5.1 Fault injection framework and methodology. . . . 113
5.2 DNN training workloads. . . . 114

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Prof. Yanjing Li. Without her support and guidance, this thesis would not have been possible. I would also like to thank my committee members, Prof. Michael Maire and Prof. Shan Lu for their assistance and comments. I would also like to thank all of my collaborators for their generous help, including Mike Hutton, Rama Govindaraju, Nishant Patil, Steven Chan, Robert de Gruijl of Google, Prasanna Balaprakash from Argonne National Laboratory, and Takumi Uezono from Hitachi. I also want to thank all my PhD fellows and friends for their help in every aspect of my life.

ABSTRACT

Deep learning (DL) accelerators have been widely deployed in a broad range of application domains, ranging from edge computing, self-driving cars, to cloud services. Resilience against hardware failures is a top priority for these accelerators as hardware failures can lead to various undesirable consequences. For inference workloads, hardware failures can generate crashes and significant degradation in inference accuracy. For training workloads, hardware failures can result in failure to converge, low training/test accuracy, numerical errors (e.g., INF/NaNs), and crashes.

In this thesis, we establish the new knowledge on how major classes of hardware failures propagate and affect deep neural network (DNN) inference and training workloads, and advance state-of-art resilient DL systems by devising lightweight, effective detection and recovery techniques to mitigate hardware failures. There are three main contributions in this thesis.

First, we devise a novel technique for generating high-quality test programs to detect permanent hardware failures (i.e., hardware failures that impose persistent effects, e.g., early-life failures, circuit aging) in the field for all hardware units in a DL inference accelerator. For hardware units that perform computations, we take advantage of the architectural characteristics of a given DL accelerator to first adopt the ATPG (Automatic Test Pattern Generation) technology that is routinely used in manufacturing testing to generate high-quality test patterns, and then reverse-engineer the dataflow/reuse algorithm of the accelerator to map the test inputs to equivalent DNNs that can be directly executed on the accelerator. For hardware units that are responsible for controlling data movement and computation, our key observation is that typically only one or a few fixed DNNs are deployed at a time, which allows us to target only the hardware failures that can directly affect these DNNs by executing different layers of the DNNs using carefully-crafted weight and input tensors. We demonstrate the efficacy of our technique using Nvidia's open-source accelerator, NVDLA,

and show that our technique successfully detects $> 99.9\%$ of stuck-at faults and $> 99.0\%$ of transition faults (two representative fault models for permanent hardware failures) for the entire NVDLA design, compared to only $< 80\%$ if random test programs are used. Moreover, our technique requires minimal test time and test storage and no hardware modification.

Second, we provide (1) an open-source resilience analysis framework targeting transient hardware failures (i.e., hardware failures that impose temporary effects, e.g., soft errors, dynamic variations) in a DL inference accelerator, called FIdelity, as well as (2) the first in-depth study, using this framework, on the impacts of transient hardware failures on all hardware units of a DL inference accelerator through fault injection experiments. FIdelity enables accurate and quick resilience analysis by modeling hardware failures in software with high fidelity. This is achieved by taking account of both the spatial and temporal reuse effects of hardware signals to map the effects of a given faulty hardware signal to a set of faulty output neurons in the current DNN layer, for which the corresponding faulty values can be obtained through software simulation. Moreover, these accurate software fault models can be obtained using minimal hardware information, without requiring access to the detailed hardware implementation (e.g. RTL, which may not be easily attainable). We implement and thoroughly validate the FIdelity framework using NVDLA as the baseline DL accelerator. Using FIdelity, we perform 46M fault injection experiments running various representative deep neural network inference workloads. We thoroughly analyze the experiment results, fundamentally understand how the injected faults propagate, and obtain several new insights that can be used in designing efficient, resilient DL inference accelerators.

Lastly, we focus on DL training workloads, and provide (1) the first study that reveals the fundamental understanding on how both transient and permanent hardware failures affect DL training workloads, and (2) new, light-weight hardware failure mitigation techniques. We extend the FIdelity framework to perform large-scale fault injection experiments targeting DL training workloads, and conduct $> 2.9M$ experiments using a diverse set of DL

training workloads. We characterize the outcomes of these experiments, thoroughly analyze the fault propagation paths, and derive the necessary conditions that must be satisfied for hardware failures to eventually cause unexpected training outcomes. Based on the necessary conditions, we develop ultra-lightweight software techniques to detect hardware failures and recover the workloads, which only require 24-32 lines of code change, and introduce $0.003 - 0.025\%$ performance overhead for various representative neural networks, which is 25x-500x more efficient than checkpointing techniques.

CHAPTER 1

INTRODUCTION

Hardware failures pose a growing challenge for deep learning (DL) accelerators that run both Deep Neural Network (DNN) inference and training workloads. For inference workloads, these failures can generate crashes and significant degradation in inference accuracy. For training workloads, hardware failures can not only introduce numerical errors (e.g., INFs/NaNs) and crashes, but also result in abnormal convergence trends, such as failure to converge and low training/test accuracy.

These unexpected outcomes not only pose significant reliability threats to DNN systems deployed in safety-critical applications (e.g., self-driving cars) but also affect DNNs in other application domains. This is because these applications also require certain reliability and output quality standards to be met.

The impact of hardware failures on DNN workloads have drawn industrial concerns, as evidenced by the increasing number of hardware failures that have recently been reported by Google, Facebook, and more [1, 2, 3, 4, 5, 6]. The hardware failure rate that industry has observed is as high as a few cores per several thousand server machines in datacenters [2, 3]. For example, Google has experienced "mysterious, difficult to identify problems" in their TPU training systems that were later identified to be hardware failures [1]. Although these problems were subsequently corrected through significant efforts, they have raised the urgency of addressing the growing challenges of hardware failures impacting many DNN systems.

The hardware failures that have previously been reported by the industry share a wide variety of origins, which include both transient failures and permanent failures [2, 3, 4, 5, 6]. Permanent failures such as early life failures and aging, are a major reliability concern. These failures are likely to show up in the field [7, 8, 9]. Therefore, guaranteeing functional safety in-the-field is extremely important, especially for DL accelerators running safety critical ap-

plications such as self-driving cars. In-field self-test is the primary solution for permanent failures, as it enables concurrent detection and localization of faults during normal operations. There are two types of in-field self-test: structural test and functional test. Structural testing involves making changes to the hardware structures to enable testing of hardware modules. On the other hand, functional testing focuses on verifying that the system functions correctly according to its intended design without the need for any hardware modifications. Existing works focus on structural tests, which require hardware supports that are not yet available to all DL accelerators. Transient failures is another major reliability concern, which includes soft errors and dynamic variations. They pose critical reliability risks to DL accelerators running both DNN inference and training workloads. However, previous works only focus on memory errors. There is *no* logic transient error analysis framework that targets DL accelerators. This gap in research is a significant limitation that needs to be addressed to ensure the reliability of DL accelerators. For example, based on our analysis, for a DL inference accelerator in which resilience protection of sequential elements is not provided, the FIT (failure in time) rate of these sequential elements (>9.5) is significantly higher than the stringent automotive safety requirement (<0.2), even just as a result of random transient errors that occur infrequently (more results reported in Chapter 3). Moreover, the impact is expected to be even higher in the presence of permanent hardware errors.

Hardware failures can impact both DL inference and training accelerators, as well as both DL inference and training workloads. With the growing prevalence of deep neural network (DNN) training workloads in data centers, they are becoming more vulnerable to hardware failures. This has been confirmed by previous industry studies [1, 2, 3, 4, 5, 6]. Understanding and mitigating the various error effects for hardware failures in both DL inference and training accelerators is particularly challenging. Prior to our work, there was no existing resilience analysis frameworks to enable the fundamental understanding and characterization of logic errors in DL accelerators. Previous resilience analysis frameworks

for general-purpose systems all exhibit limitations: RTL/mixed-mode fault injections are accurate, but RTL is not likely to be available during early design phases, and the simulation time is prohibitively long [10, 11, 12, 13, 14, 15, 16]. Software-level fault injection techniques are quick, but their accuracy can not be guaranteed [17, 18, 19]. Moreover, existing error mitigation techniques, such as redundancy-based approaches for permanent failures, and circuit level FF-hardening techniques for transient failures, are all costly in performance and energy (as evidenced by our results in Chapter 3).

Thesis statement: The works in this thesis address the pressing need for understanding how hardware failures propagate and affect DNN inference and training workloads, and advance the state-of-the-art in resilient DL systems by devising lightweight and effective detection and recovery techniques that can mitigate the impact of hardware failures. To achieve this, we have conducted extensive research and experimentation to investigate the impact of hardware failures on DNNs and the challenges they pose to the reliable operation of DNN systems. Our work addresses the need for reliable DNN systems that meet the reliability demands of various applications. Moreover, our work uniquely combines the knowledge of testing, hardware error resilience, computer architecture, and machine learning to address both permanent and transient hardware errors for both inference and training workloads. Our key contributions are:

(1) We present a novel technique that generates high-quality functional in-field self-tests specifically targeting DL accelerators to detect permanent failures. These functional tests can be applied in the field during normal operation, which is crucial to ensure that the safety and/or reliability requirements are met for any given application, including safety-critical applications such as self-driving cars, robotics, and more.

Our technique takes advantage of special architectural characteristics and application properties to achieve high functional test coverage, while incurring minimal system-level costs. To account for the different architectural properties of different hardware units, we

devise different strategies for the compute units (which support computation operations) and the control units (which control data movement).

For the compute units, we first use combinational ATPG to generate test patterns with high test coverage, which is possible because these units do not contain complex sequential logic. Next, we map the ATPG patterns to one or more equivalent deep neural networks (DNNs) that can be directly executed on the accelerator, which is possible given the well-defined dataflow/reuse algorithm of a DL accelerator. This is possible because we leverage the following properties: (1) these units generally do not contain complex sequential logic, so combinational ATPG yields high test coverage; (2) given that the dataflow/reuse algorithms of a DL accelerator are well defined, it is always possible to map the inputs of individual compute units to the primary inputs of the accelerator.

For the control units, we leverage the property that typically only one or a few fixed DNNs are deployed at a time in many application domains. Thus, it is sufficient to target only the faults that can directly affect the correctness of the DNNs that are currently deployed, which can be tested by executing different layers of each target DNN using input or weight tensors with linearly-independent columns to maximize test coverage while minimizing test time.

We apply our technique using Nvidia's open-sourced accelerator NVDLA as a case study to demonstrate its efficacy. Our results show that our technique achieves extremely high test coverage. For the compute units, we achieve 99.9% / 99.0% coverage for single stuck-at / transition faults, which are among the most prominent metrics for permanent hardware failures.

For the control units, we are able to mathematically prove that our technique achieves 100% test coverage for a large class of single and multiple fault models, out of all control faults that can affect the correctness of the target DNN.

For single stuck-at faults / transition faults, the in-field functional self-test time ranges

from 1.13-16.84 ms / 1.57-17.28 ms for various representative DNNs (including GoogleNet, Yolo, DenseNet and EfficientNet), and the test storage is <600MB / 650MB, respectively. Such minimal cost enables practical adoption of this technique in various application scenarios. Our technique can be applied during boot-up, reset, or concurrently with normal operation by executing DNN test programs directly on the accelerator, without any hardware test support.

(2) We present FIdelity, our resilience analysis framework, which accurately and quickly analyzes the behavior of hardware failures in DL accelerators. FIdelity ensures that the reliability requirements can be met starting from the very beginning of the hardware design process, such that the final product can be safely and reliably deployed for a wide range of applications, including safety-critical applications such as self-driving cars.

Our framework only requires a minimal amount of high-level design information, which can be obtained from architectural descriptions/block diagrams, or estimated and varied for sensitivity analysis. FIdelity leverages unique architectural properties of DL accelerators to systematically model transient hardware failures in software with high fidelity. This is achieved through a novel algorithm called "Reuse Factor Analysis", which takes in high-level design information and generates software fault models. With these software fault models, FIdelity enables quick and accurate software fault injections, and does not require access to RTL.

We thoroughly validate FIdelity using NVDLA, which shows that the results are highly accurate – out of 60K fault injection experiments, the software fault models derived using FIdelity closely match the behaviors observed from RTL simulations. Also, as a software framework, FIdelity achieves >10000X/>440X simulation time reduction compared with conventional hardware/mixed mode fault injection techniques.

Using the validated FIdelity framework, we perform the first large-scale resilience study targeting both logic and memory transient hardware failures on NVDLA, which includes

46M fault injection experiments running various representative DNN workloads. For the first time, accurate results and observations for various representative DNN workloads are reported. This study enhances our understanding on the resilience properties of DL accelerators/workloads, and can be used to guide the design of future accelerators.

(3) We present the first in-depth study on hardware failures in DNN training systems, which is enabled by two new fault injection frameworks that accurately models the behaviors of both transient and permanent hardware failures in DNN training accelerators.

The frameworks are modified based on our FIdelity framework to add support for DNN training workloads and permanent failures, and are validated through $>$ 60K RTL fault injections following a similar methodology used for FIdelity.

Using these two frameworks, we perform $>$ 2.9M fault injection (FI) experiments ($>$ 490K node-hours) for transient failures and $>$ 100K FI experiments for permanent failures in a distributed DNN training environment. Based on the experiment results, we characterize the failure behaviors for transient and permanent failures. In addition to known effects (e.g., a failure is masked, or generates NaNs/INFs [1, 6]), we identify four new outcomes that have not been found in industry reports or in the literature. These new cases do not generate immediate, visible anomalies such as NaN values, but can result in abnormal convergence trends that persistently affect the training workloads (e.g., thousands of training iterations or more). We also observe one of the new outcomes (SlowDegrade) in real datacenter DNN training accelerator systems.

We then perform in-depth analysis of these outcomes, and find that large absolute gradient history values in optimizers, or large absolute moving variance values in normalization layers, are the necessary conditions for transient hardware failures to generate unexpected training outcomes in DL accelerator training systems, and a large increase in training loss values is the necessary condition for permanent hardware failures to generate unexpected outcomes. Moreover, these conditions always occur within two training iterations after hardware

failures occur.

Based on the necessary conditions, we devise (a) a new hardware failure detection technique that checks the absolute gradient history values, the absolute moving variance values, and the training loss values against their respective bounds, where the bounds can be mathematically derived based on properties of a given DNN training workload; coupled with (b) light-weight re-execution of the two most recent training iterations, which is sufficient to recover training workloads from hardware failures.

We evaluate our technique using Google Cloud TPUs, and the result shows that our detection and re-execution techniques together require $24 - 32$ lines of code change and introduce only $0.003 - 0.025\%$ performance impact for various DNN training workloads.

This thesis is organized as follows. Our novel functional in-field self-test technique is presented in Chapter 2. Our FIdelity framework, and our resilience study targeting DNN inference accelerators and inference workloads are discussed in Chapter 3. The first in-depth resilience studies targeting DNN training accelerator and training workloads, along with our light-weight techniques to detect hardware failures are discussed in Chapter 4 and Chapter 5. Chapter 4 focuses on transient hardware failures, while Chapter 5 focuses on permanent failures.

CHAPTER 2

EFFICIENT FUNCTIONAL IN-FIELD SELF-TEST FOR DEEP LEARNING ACCELERATORS

The content of this chapter was previously published in our paper titled "Efficient Functional In-Field Self-Test for Deep Learning Accelerators" in ITC21[20], and our paper titled "Achieving Automotive Safety Requirements through Functional In-Field Self-Test for Deep Learning Accelerators" in ITC22.[21].

We present a technique that generates high-quality functional in-field self-tests specifically targeting deep learning (DL) accelerators. These functional tests can be applied in the field during normal operation of a DL accelerator, which is crucial to ensure that the safety and/or reliability requirements are met for any given application, including safety-critical applications such as self-driving cars, robotics, and more.

Our technique takes advantage of special architectural characteristics and application properties to achieve high functional test coverage while incurring minimal system-level costs. Moreover, we devise different strategies for the compute units (which support computation operations) and the control units (which control data movement) because these two types of units exhibit different properties. For the compute units of a DL accelerator, we first use combinational ATPG to generate test patterns with high test coverage, which is possible because these units do not contain complex sequential logic. Next, we map the ATPG patterns to one or more equivalent deep neural networks (DNNs) that can be directly executed on the accelerator, which is possible given the well-defined dataflow/reuse algorithm of a DL accelerator. For the control units, we leverage the property that typically only one or a few fixed DNNs are deployed at a time in many application domains (e.g., self-driving cars). Thus, it is sufficient to target only the faults that can directly affect the correctness

of the DNNs that are currently deployed. This is done by executing different layers of each target DNN using carefully-crafted input and weight values to maximize test coverage while minimizing test time.

We apply our technique using Nvidia's open-source accelerator as a case study to demonstrate its efficacy. Our results show that our technique achieves high test coverage. For the compute units, 99.9% (98.8%) functional test coverage is achieved for stuck-at (transition) faults. For the control units, we are able to prove that, given any target DNN, 100% coverage can be achieved for a large class of single and multiple fault models. The in-field functional self-test time is also very low, < 17.28 ms for various representative DNNs. These functional tests can be applied during boot-up, reset, and even concurrently with normal operation by executing DNN test programs directly on the accelerator, without requiring any test support in the hardware.

2.1 Introduction

Deep learning (DL) accelerators are widely deployed in a wide variety of application domains, including safety-critical applications such as self-driving cars, robotics, and more[22]. For example, DL accelerators are commonly used in automotive systems to perform object detection and classification tasks, which can affect mission-critical decisions such as emergency braking, lane change assistance, and adaptive cruise control [23]. Thus, it is mandatory for these accelerators to meet the stringent automotive safety standards (e.g., ISO26262). However, previous work has shown that, without any reliability features, DL accelerators would fail to meet such safety requirements (as shown in Chapter 3). Therefore, it is a top priority as well as a critical challenge to ensure that DL accelerators are able to meet the required levels of safety/reliability for *all* applications.

In this chapter, we present a technique that generates high-quality functional tests for DL accelerators, which can be applied during normal operation in the field. Our technique

is capable of detecting permanent faults – for example, early-life failures, circuit aging, and manufacturing defects and variations [24] – which are a major reliability concern. Compared to structural tests, the main advantage of functional tests is that they do not require structural test support (e.g., CASP [25, 26] or Logic BIST [23]) in the hardware. Even though structural test features are becoming more prominent for in-field self-test applications ranging from datacenters to automotive systems [7, 8, 9, 23, 27, 28], they are not yet available in all DL accelerators. Therefore, in-field functional tests are essential for DL accelerators that are not equipped with in-field structural test support.

A well-known challenge of functional tests is that they generally achieve lower test coverage than structural tests due to both observability and controllability constraints [25, 26]. Fortunately, for in-field self-tests that are applied for the purpose of detecting permanent faults, it is sufficient to target only the faults that can affect application correctness. In other words, it is sufficient to achieve high *functional test coverage*, which is defined in this work as the percentage of detected faults out of the total number of faults that are detectable using functional tests. Achieving high functional test coverage, however, is still a great challenge on its own due to the massive circuit complexity in today's DL accelerator designs.

To tackle this challenge, we create a novel functional test generation technique that specifically targets DL accelerators. Our technique achieves high functional test coverage by taking advantage of special architectural characteristics and application properties of DL accelerators. The hardware units in a DL accelerator can be classified into two categories: compute units and control units. *Compute units* are defined as the hardware modules that perform computations to transform data (e.g., multiply-accumulate units and element-wise units), while *control units* are the modules that are solely responsible for data movement (e.g., input/weight fetch units and sequencing units that dispatch different {input, weight} pairs to different compute units). We devise different strategies for these two types of units because they exhibit different properties.

First, for the compute units of a DL accelerator, since these units generally do not contain complex sequential logic, high-coverage functional test patterns can be generated using combinational ATPG. Moreover, since DL accelerators implement well-defined dataflow/reuse algorithms, the ATPG patterns generated for each compute unit can be mapped to one or more equivalent DNNs based on the specific dataflow algorithm. The response of each ATPG pattern can then be observed from memory and/or buffer structures throughout the execution of the equivalent DNN test programs.

Second, for the control units, we leverage the following property: in many application domains (e.g., self-driving cars), only one or a few DNNs are deployed at a time. Thus, it is sufficient to only target all faults that can affect the correctness of the DNNs currently deployed. We are able to mathematically prove that, by executing different layers of a given DNN using carefully-crafted input and weight values, 100% test coverage is achieved for all *single-variable-type control fault models*, out of all control faults that can affect the correctness of the given DNN. Here, a single-variable-type control fault model is defined as single or multiple faults that only affect the control logic associated with a single variable type in DNN applications (e.g., inputs or weights).

To demonstrate the effectiveness of our technique, we use Nvidia's open-source DL accelerator as a case study. The key results are:

1. Our technique achieves extremely high functional test coverage. 99.9% (98.8%) single stuck-at (transition) functional test coverage is obtained for the compute units, and 100% test coverage is achieved for all single-variable-type control faults with respect to the correctness of any given DNN.

2. The in-field self-test time for applying the functional tests generated by our technique is $<$ 17.28 ms for various representative DNNs. Given such short test time and therefore minimal performance/power impact at the system level, it is practical to periodically apply these tests in the field, during boot-up, reset, and even concurrently with normal operation.

Moreover, since our high-coverage and low-cost functional tests can be applied by executing various DNN test programs on a DL accelerator, no hardware modification or test support is required.

In summary, the major contributions of this work are:

1. A novel functional test generation technique that specifically targets DL accelerators.

2. The detailed algorithms of our technique using Nvidia's DL accelerator (NVDLA) as a case study.

3. Thorough evaluation of our technique to demonstrate that it is highly effective and practical.

This chapter is organized as follows. We discuss background and related work in Sec. 2.2. We present the details of our technique in Sec. 2.3. Evaluation methodology and results are presented in Sec. 2.4.

2.2 Background and Related Work

2.2.1 Deep Learning Accelerators

We focus on digital electronic inference DL accelerators, which typically share a common dataflow architecture that consists of the following stages:

(1) **Input/weight pre-processing**, which fetches input and weight values from off-chip memory, performs optional data processing such as padding and compression, and then stores the values in on-chip buffers.

(2) **Input/weight sequencing**, which dispatches each pair of input and weight values to a compute unit. The sequencing logic is implemented based on a specific dataflow/reuse algorithm [29].

(3) **Multiply-accumulate (MAC) processing**, which performs MAC operations, the core computation primitive for all DNN workloads, using multiple compute units.

(4) **MAC output processing**, which applies the activation function and other optional operations (e.g., bias addition) to the outputs of each MAC compute unit, and stores the partial or final results in memory.

We use Nvidia's open-source accelerator, NVDLA, to illustrate a detailed example of the DL accelerator architecture, which is shown in Fig. 2.1. DNN inputs and weights are fetched by the CDMA unit and stored in CBUF. The sequence controller unit (CSC) schedules and distributes inputs and weights to 16 parallel MAC units (CMAC). In terms of the reuse algorithm, each CMAC unit calculates the products of 64 {input, weight} pairs (for 16-bit floating-point or integer values), and outputs the sum of the 64 products. Different CMAC units operate on different weight kernels, and the weight kernels are re-used for 16 cycles. On the other hand, a new set of DNN inputs are fetched each cycle, and the same input values are dispatched to all 16 CMAC units. The partial sums generated by each CMAC unit across multiple clock cycles are next added up by the accumulation unit (CACC). The final sums are then sent to the single data processing unit (SDP), which applies the activation function and performs data precision conversion. SDP also performs optional element-wise operations such as bias-addition and batch normalization. The planar data processing unit (PDP) is responsible for performing pooling operations. The compute and control units in NVDLA are also labeled in Fig. 2.1.

2.2.2 In-Field Self-Test

Permanent faults or hard failures, such as those caused by early-life failures, circuit aging, variations, and manufacturing defects, pose major reliability challenges in advanced CMOS technologies. Their impact is expected to become even more severe in the future as system complexity increases and device geometry miniaturization further diminishes [24, 30, 31, 32]. Therefore, it is essential to enable robust systems with built-in tolerance to permanent faults in the field, concurrently with normal system operation, to ensure that the required levels

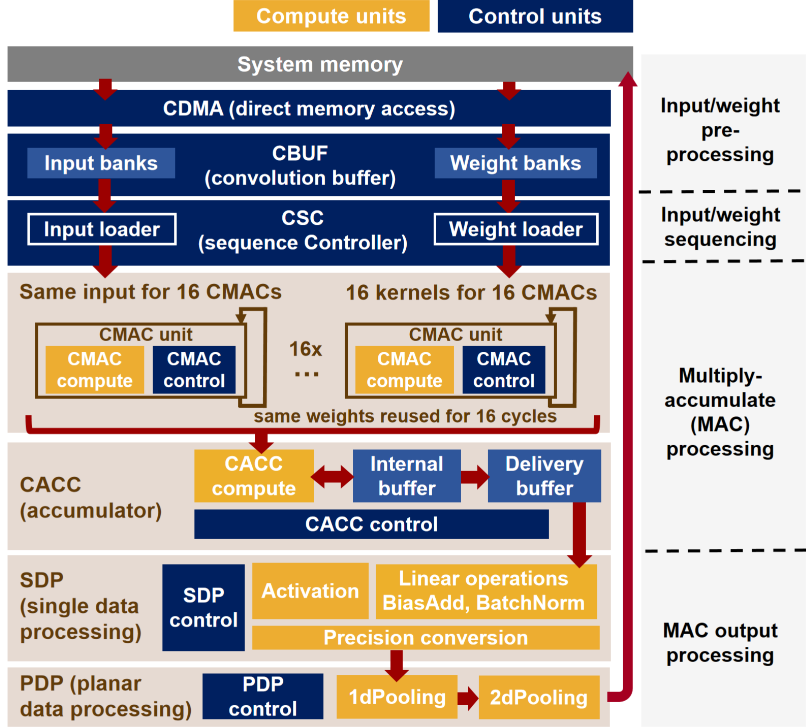


Figure 2.1: NVDLA high level diagram.

of safety and/or reliability requirements are achieved for any application, including safety-critical applications such as self-driving cars.

Many in-field self-test techniques exist in the literature. These techniques can be classified into two categories: structural and functional. Structural in-field self-test techniques [7, 8, 9, 23, 25, 27, 28] generally achieve higher test coverage than functional tests. However, they require structural (i.e., scan) test support in the field, which in turn requires modifications to the hardware. Even though in-field structural test features are becoming more prominent, they are not yet available in all DL accelerators.

Functional tests, on the other hand, are executed as software programs without requiring hardware support. Extensive research has been done on generating functional test patterns for general-purpose processors. However, high test coverage is difficult to achieve with functional tests alone. In some cases, a fault that can be detected using structural tests is simply not detectable using functional tests due to circuit constraints (e.g., constant values). In

other cases, if a hardware module implements complex sequential logic, it is difficult for functional tests to cover the large number of sequential states. These reasons are well-known for functional tests targeting general-purpose processors, but they also apply to DL accelerators, and we will discuss examples of these cases in Sec. 2.4.8. However, as discussed in Sec. 2.1, our focus is to utilize functional in-field self-test for detecting permanent faults. Therefore, it is sufficient to generate functional tests with high *functional test coverage* for DL accelerators.

2.3 Functional In-Field Self-Test Pattern Generation Technique

Our technique takes advantage of the key observation that DL accelerators and applications exhibit unique characteristics and properties to generate functional tests with high functional test coverage and low test time. We devise different strategies for the compute and control units of a DL accelerator according to their distinct architectural properties, which we will discuss in details below.

2.3.1 Functional In-Field Self-Test Generation Technique for Compute Units

We identify two key properties associated with the compute units of DL accelerators, which are utilized in our technique to achieve high functional test coverage for these units.

Architecture Property (1): Compute units generally do not contain complex sequential logic due to the nature of dataflow architectures.

For example, in NVDLA, each CMAC_compute unit consists of 64 multipliers and a 5-level CSA adder tree. The inputs of a CMAC_compute unit (64 DNN {input, weight} pairs) simply streamline through the multipliers and adders, without any complex control loops or finite state machines. Such a property is representative for other compute units and other DL accelerators as well.

By leveraging Architecture Property (1), advanced combinational ATPG algorithms can

be used to generate functional test patterns for various fault models (such as single stuck-at faults, transition faults, and more) to achieve high functional test coverage.

Architecture Property (2): The dataflow/reuse algorithms of a DL accelerator (e.g., weight stationary, row stationary, output stationary, etc.) are well-defined in the architectural specification.

Given the knowledge of the dataflow algorithm, ATPG patterns of each compute unit can be mapped to one or more equivalent DNN test programs by reversing the dataflow algorithm. Our technique also ensures that the outputs of the different DNN layers correspond to the ATPG test responses. Therefore, they can be observed either from memory or buffer structures throughout the execution of the DNNs, and compared against the golden responses. If all test responses match the golden responses, it means that no permanent fault is detected, and normal operation can be resumed. Otherwise, appropriate diagnosis and self-repair actions will need to be performed.

Putting these two architecture properties together, for a given compute unit, our functional in-field self-test generation technique follows two steps:

- Generate test patterns for the compute unit using combinational ATPG.
- Map the ATPG test patterns to one or more equivalent DNN test programs.

The mapping algorithms are specific to individual compute units and dataflow algorithms. As a case study, we develop the mapping algorithms for all major compute units in NVDLA, and the details are in Sec. 2.4.

2.3.2 Functional In-Field Self-Test Generation Technique for Control Units

In contrast to compute units, control units consist of many finite state machines and complex sequential logic to generate appropriate control signals for various datapath functions. Without structural test support, it is expected that even the most advanced ATPG would yield

low test coverage. Thus, the functional test generation approach described in the previous section is inadequate for control units, and a new approach is required.

For control units, we also leverage special architectural/application properties in our technique. Due to space limits, we focus on presenting our technique for the control units that are used to control MAC operations (e.g., CDMA, CBUF, CSC, CMAC_control, and CACC_control in NVDLA as shown in Fig. 2.1), referred to as *MAC control units*. This is because MAC operations dominate overall DNN computations [33], and MAC control units are more complex than others. Generating functional test patterns for other control units is similar in principle but much simpler.

One of our key observations is that, although it is challenging to generate functional test patterns to screen out all faults in the control units, in practice, it is sufficient to only target a subset of control faults that directly affect the output correctness of one or a few DNNs due to the application property shown below.

Application Property: In many application domains, only one or a few fixed DNNs are deployed on a DL accelerator at a time.

To ensure that MAC control units function correctly with respect to a given DNN, we categorize the functionalities of the signals in these units, and identify five requirements that must be satisfied. Note that, this list is complete based on the general architecture of DL accelerators.

(1) There are no visible anomalies such as accelerator time-outs.

(2) All computations are done according to the data precision defined by the DNN. For example, if a DNN uses 16-bit integers to represent both its inputs and weights, then the MAC operations should be performed on 16-bit integers.

(3) Accelerator pipeline configuration signals must correctly indicate if certain optional operations (e.g., batch normalization, bias addition, and pooling) should be performed, as well as how these operations should be performed.

(4) The dimensions of the output tensors of each DNN layer must be correct.

(5) The values of all output neurons must be correct.

Given these requirements, it is natural to adopt the set of DNNs currently deployed as functional test programs. The goal of our test generation technique is to determine the values of the inputs/weights of the DNNs to maximize coverage while minimizing test time. To be more concrete, our technique is responsible for generating appropriate input/weight patterns of a given DNN to ensure that, for each control fault that can affect the output correctness of the DNN, there exists at least one output neuron in one of the DNN layers such that its value differs from the golden output neuron value.

It is relatively straight-forward to generate test patterns to cover requirements (1)-(4). First, accelerator time-outs can be directly observed. Second, to determine if the output tensor dimension in a DNN layer is correct, an input/weight pattern that yields all non-zero output neurons without any overflow conditions would suffice, if we initialize all elements in the output memory regions to 0. Finally, it is extremely unlikely for the effects of erroneous data precision or pipeline configurations to be completely masked. For example, to verify that the correct data precision is used in all computations, it is sufficient to ensure that each output neuron, when computed using the correct data precision, differs from all other data precision settings allowed in the accelerator. This can be satisfied with random patterns.

Requirement (5), on the other hand, is less straight-forward. Recall that control units are solely responsible for data movement. Therefore, faults in MAC control units can only result in an incorrect output neuron value through one of two ways: (a) the number of {input, weight} pairs used to calculate the output neuron is larger or smaller; or (b) the weight/input values used for computation are incorrect (e.g., if they are fetched from incorrect memory locations).

To test against requirement (5), we leverage another key property associated with the dataflow architecture of DL accelerators.

Architecture Property (3): The control logic for DNN inputs and weights are independent. In other words, a fault in the weight control logic will not cause errors in the input control logic, and vice versa. (This is also true for input and weight datapath.)

Therefore, given an output neuron y which is calculated as shown in Eq. (1) in the fault-free scenario, one or more permanent faults in the weight or input control logic would cause the same output neuron to be calculated as y' in Eq. (2) or (3), respectively, if the number of {input, weight} pairs used to compute the faulty neuron is $\leq n$ (the size of the weight kernel). Note that, the case where this number is less than n simply means that some w'_i or x'_i values are zero. If this number is greater than the correct value, then at least one more non-zero {input, weight} products will be added to y' in Eq. (2) and (3).

$$\begin{gathered} y = f(\sum_{i}^{n} x_i \times w_i) \\ f : \text{activation function; } n : \text{the size of one weight kernel;} \\ x_i/w_i : \text{input/weight values.} \end{gathered} \tag{2.1}$$

$$\begin{gathered} y' = f(\sum_{i}^{n} x_i \times w'_i) \\ w'_i : \text{differs from } w_i \text{ in Eq. (1) for at least one } i. \end{gathered} \tag{2.2}$$

$$\begin{gathered} y' = f(\sum_{i}^{n} x'_i \times w_i) \\ x'_i : \text{differs from } x_i \text{ in Eq. (1) for at least one } i. \end{gathered} \tag{2.3}$$

Leveraging Architecture Property (3), we prove that, given a DNN layer, as long as the input/weight values of this layer satisfy the *n-linearly-independent property* specified in Theorems 1 and 2, then all MAC control logic faults that can affect the output correctness of this DNN layer will be detected.

Definition 1. InVec(y): a column vector of all input values used to calculate the output neuron y, organized following the order of width, height, and channel indices.

Definition 2. WtVec(y): a column vector of all weight values used to calculate the output neuron y, organized following the order of width, height, and channel indices.

Theorem 1. *Given a DNN layer where the size of a weight kernel is n, construct an input feature map with all non-zero values, such that for any given output neuron y_0, $\exists$ at least $n-1$ output neurons y_1, ..., y_{n-1} where InVec(y_0), ..., InVec(y_{n-1}) are all linearly independent of each other. Such an input feature map is said to satisfy the n-linearly-independent property. If the activation function is invertible, then executing the given DNN layer with such an input feature map can detect all single and multiple permanent faults that affect the weight control logic.*

Proof. Form a linear system with n equations and n unknowns, where the equations are y_0, ..., y_{n-1} as shown in Eq. (1), and the n weight values are treated as unknowns. Since the parameters of these equations, i.e., InVec(y_0), ..., InVec(y_{n-1}), are linearly independent of each other and the activation function is invertible, there is a unique solution for the n weight values. Also, for the case where the number of {input, weight} pairs is greater than n, all weight values outside of the range of n must be 0 to satisfy all n equations. Thus, when applying an input feature map that satisfies the n-linearly-independent property, if all output neurons match the golden values, WtVec(y) must be equal to the golden WtVec(y) $\forall$ y. Therefore, the condition required in Eq. (2) is violated, which means that the weight control logic functions correctly with respect to the given DNN. $\square$

Similarly, we can construct weight kernels that satisfy the n-linearly-independent condition to cover all input control faults, as stated in Theorem 2.

Theorem 2. *Given a DNN layer where the size of the weight kernel is n, construct n weight kernels with all non-zero values, such that when expressing these weight kernels as column vectors, they are all linearly independent of each other. If the activation function is invertible, then executing the given layer with these n weight kernels, which requires the DNN layer to be*

executed ceil(weight kernel size / number of weight kernels) times, can detect all single and multiple permanent faults that affect the input control logic.

Combining Theorems 1 and 2, DNN inputs and weights that both satisfy the n-linearly-independent property can be used to detect weight and input control faults simultaneously for any given DNN layer.

As we will show in our experiment results in Sec. 2.4.7, the n-linearly-independent property can be satisfied using random patterns for various representative DNNs. Moreover, to minimize in-field self-test time of these control units, the following test optimization can be applied: given a DNN, if the dimension of a layer L1 subsumes that of another layer L2 – that is, the height, width, and number of channels of both the weights and inputs of L1 are greater than L2 – then it is only necessary to execute L1 using a functional test pattern that satisfies the n-linearly-independent property to ensure that the input/weight values for each output neuron in both L1 and L2 are correct, because WtVec(y)/InVec(y) in L2 is a subspace of the corresponding vector in L1 $\forall y$.

In summary, given a DNN, the steps for generating and applying MAC control unit functional tests include:

- Select all unique layers in the DNN. Moreover, for each of these layers, its dimension must not be subsumed by another layer.

- For all layers selected in step 1, create one or more input/weight patterns to cover the follow conditions: (a) satisfy the n-linearly-independent property; (b) yield all positive output neurons without any overflow conditions; and (c) are capable of verifying that Requirements (2) and (3) are satisfied for all output neurons.

- For all unique DNN layers that are not selected in step 1, create one or more test pattern that cover the following conditions: (a) yield all positive output neurons without any overflow conditions; and (b) are capable of verifying that Requirements (2) and (3) are

satisfied for all output neurons.

- Execute all unique layers using the {input, weight} patterns generated in previous steps. A time-out detection mechanism should also be applied in this step.
- Check that all output neuron values match the golden values, and that the output dimensions of all layers are correct.

2.4 Evaluation Methodology and Results

We use NVDLA along with various representative DNNs as a case study to demonstrate the effectiveness of our functional test generation approach. Since digital DL accelerators share many common features, we expect that our technique is applicable and effective for other DL accelerators as well.

2.4.1 Stuck-at Faults in Compute Units

For all the compute units in NVDLA as shown in Fig. 2.1, we use Synopsys TestMax to generate single stuck-at fault test patterns using the combinational ATPG setting. To minimize ATPG run-time, we first convert these compute units into purely combinational logic by turning all pipeline registers into buffers. This conversion does not affect test coverage because it preserves all faults given Architecture Property (1) discussed in Sec. 2.3. In addition, we hardwire all 'valid' input signals to high to activate the compute units. We focus on the 16-bit floating point (FP16) data precision.

Next, we provide the detailed algorithms that map ATPG patterns generated for each compute unit to equivalent DNN test programs.

Mapping CMAC_compute's ATPG patterns to DNNs

Recall from Sec. 2.2 that there are 16 parallel CMAC_compute units in NVDLA. The mapping algorithm for these units is given in Algorithm 1, which is derived by reverse-engineering NVDLA's dataflow algorithm. Given t ATPG patterns, each of which corresponds to 64 {input,weight} pairs, we form t identical DNNs (line 1). Each DNN corresponds to a different ATPG pattern. The input tensor of each DNN is taken directly from the 64 input values in the corresponding ATPG test pattern (line 4). There are 16 weight kernels per DNN (line 5). Each weight kernel has the shape $1 \times 1 \times 64$ (line 3) and share the same values that correspond to the 64 weight values in the ATPG pattern (line 6).

In Fig. 2.2, we demonstrate that executing one of these DNNs is equivalent to applying the corresponding ATPG pattern to all 16 CMAC_compute units. According to the dataflow algorithm, when executing this DNN, its input vector is dispatched to all 16 CMAC_compute units. At the same time, each of the 16 weight kernels is sent to one of the 16 CMAC_compute units. Since all 16 weight kernels are identical, all 16 CMAC_compute units simultaneously receive the same input and weight values from the corresponding ATPG pattern, which is the desirable behavior. The output neurons of DNN l, which corresponds to the test responses of all 16 CMAC units for ATPG pattern l, can be observed through the internal buffer of CACC, which is discussed in the next section. Once retrieved, the actual test responses are then compared with the golden responses.

Mapping CACC_compute's ATPG patterns to DNNs

CACC receives the outputs of the 16 CMAC units (each of which is a 44-bit FP value) and performs addition operations to accumulate subsequent CMAC outputs into 48-bit FP partial sums. In NVDLA, multiple output neurons can be outstanding at the same time, and the final sum of each neuron may take multiple cycles to generate. Therefore, CACC implements an internal buffer to hold the partial sums that belong to multiple output neurons. When

Algorithm 1: Mapping CMAC_compute ATPG patterns to equivalent DNNs.

```
   Input 1: t, the number of ATPG patterns.
   Input 2: ins, a list of the t input patterns generated by ATPG (64 input values per
    pattern).
   Input 3: wts, a list of the t weight patterns generated by ATPG (64 weight values per
    pattern).
1  for l in range(t) do
2  |   input_tensor_l = zeros(1, 1, 64);
3  |   wt_tensor_l = zeros(1,1,64,16);
   |   // Prepare the input tensor.
4  |   input_tensor_l[0,0,:] = ins[l];
   |   // Prepare the weight tensor.
5  |   for k in range(16) do
6  |   |   wt_tensor_l[0,0,:,k] = wts[l];
   Output: t DNNs, where input_tensor_l and wt_tensor_l are the inputs and weights of
    the lth DNN.
```

the final sum is calculated for a given output neuron, it is then converted to a 32-bit FP (FP32) value, and stored in CACC's delivery buffer to be delivered to the next pipeline stage in the accelerator.

Here we assume that CACC's internal buffer and delivery buffer are both directly accessible by software programs through debug features, since such features are commonly supported in modern designs. Therefore, combinational ATPG can be used to generate test patterns for CACC_compute. If this debug feature is not supported, the desired internal buffer contents can be generated through accumulation operations over one or more cycles.

Using combinational ATPG, each test pattern for CACC_compute consists of 16 44-bit values, corresponding to the outputs of 16 CMAC units, and 16 48-bit values, corresponding to the internal buffer values to be added to each of the CMAC outputs. Since the internal buffer is accessible by a DNN test program, the 16 48-bit values in each ATPG pattern can be directly specified accordingly. To map the 16 44-bit values from each ATPG pattern to the inputs of CACC_compute through a DNN test program, our technique first applies Algorithm 2 to generate appropriate inputs to all 16 CMAC_compute units so that their outputs match the values specified in the ATPG pattern. Afterwards, the inputs of the CMAC_compute units can be mapped to equivalent DNNs following Algorithm 1 with one

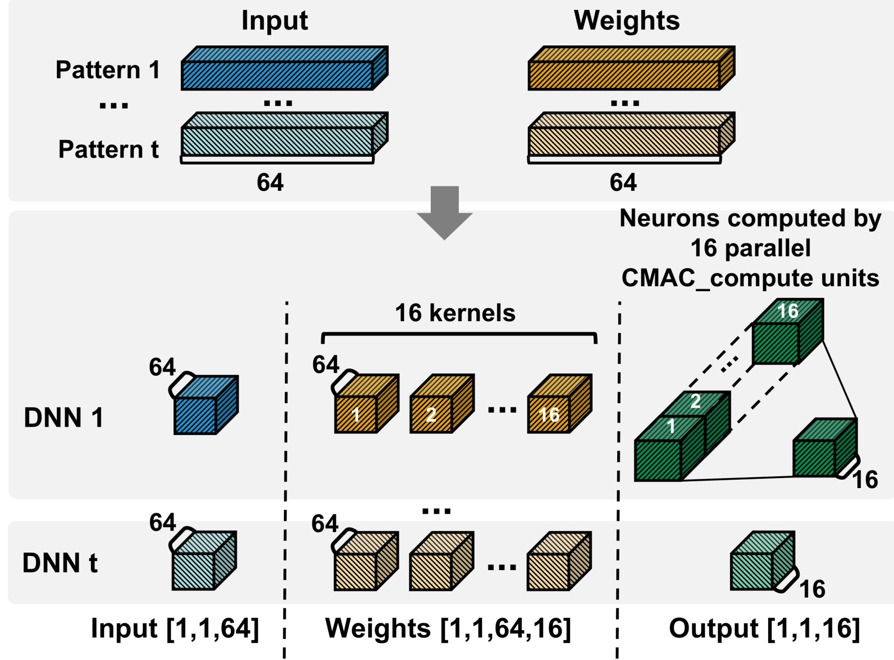


Figure 2.2: Illustration of Algorithm 1.

slight modification: the 16 weight kernels in each DNN are not required to be the same, which allows each CMAC_compute unit to produce different outputs to match the values in each CACC_compute's ATPG pattern.

Algorithm 2 first expresses each 44-bit value ($f0$,...,$f15$) as a sum of power-of-2 terms according to the definition of the 44-bit FP datatype, i.e., $f0 = sign0*mant0_0*2^{e0_0}+sign0*mant0_1*2^{e0_1}+...+sign0*mant0_{38}*2^{e0_{38}}, ..., f15 = sign15*mant15_0*2^{e15_0}+sign15*mant15_1*2^{e15_1}+...+sign15*mant15_{38}*2^{e15_{38}}$, where $sign0, ..., sign15$ are the sign bits of $f0, ..., f15$, $mant0_i, ..., mant15_i$ are the ith mantissa bits of $f0, ..., f15$, and $e0_i, ..., e15_i$ are the exponents corresponding to the ith mantissa bits of $f0, ..., f15$. This is done by storing the value of ej_i in $exp_array[j, i]$ if the corresponding mantissa bit is not zero, and all the sign bits in the $sign_array$ (lines 6-12).

Next, since the number of mantissa bits (39, including the mantissa's implicit leading bit) of a 44-bit FP value is smaller than 64 (i.e., the number of {input, weight} pairs in the input of a CMAC_compute unit), we express each power-of-2 term ($sign0*mant0_i*2^{e0_i}, ..., sign15*mant15_i*2^{e15_i}$) as the product of a FP16 input value and a FP16 weight

value, such that $sign0 * mant0_i * 2^{e0_i} = sign0 * 2^{p0_i} * 2^{w0_i}, ..., sign15 * mant15_i * 2^{e15_i} = sign15 * 2^{p15_i} * 2^{w15_i}$. Lines 13-26 in Algorithm 2 show how the exact values of $p0_i, ...p15_i$ and $w0_i, ..., w15_i$ are determined. Two considerations in our algorithm are worth noting. First, given the dataflow algorithm that in the same cycle, the same inputs are sent to all 16 CMAC units, $p0_i, ..., p15_i$ must be equal for all i. The shared input exponent value is represented using sft in the algorithm. Second, due to space limitations, we do not show the cases where (1) a single power-of-2 term exceeds the dynamic range of the product of two FP16 values, and (2) there exist two exponents in the same row of exp_array whose difference exceeds the difference between the minimal and maximal exponents of FP16 (lines 17-18), which require a more complex iterative process. Actually, by taking advantage of the Application Property discussed in Sec. 2.3.2, these cases may be safely omitted since they correspond to very large output neuron values or very large input/weight values that are extremely unlikely to appear in today's DNN workloads.

Combining Algorithms 2 and 1, a set of DNN test programs, one per ATPG pattern, are generated for CACC_compute. When a test DNN is executed, the 16 44-bit FP values in the corresponding ATPG pattern are applied to the corresponding input ports of the CACC_compute unit. Moreover, at the first cycle when each DNN is executed, the 16 48-bit FP values in the same ATPG pattern are also loaded to the appropriate internal buffer address (the first entry in internal buffer bank 0 given the structures of the test DNNs) so that the accumulation operations are performed as intended, as shown in Fig. 2.3.

To observe the test responses produced by CACC_compute, we retrieve the outputs of CACC_compute directly from CACC's delivery buffer. Otherwise, the outputs of CACC will be converted from FP32 to FP16 in SDP. Although such precision conversion does not affect functional test coverage, avoiding this conversion improves observability. For the same reason, we obtain the test responses of the CMAC_compute units directly from the CACC's internal buffer to maximize observability, which is possible by initializing the internal buffer

contents to all 0's when executing the DNN test programs for the CMAC_compute units.

Algorithm 2: Mapping CACC_compute's ATPG patterns to CMAC_compute inputs.

```
   Input: cacc_data, one ATPG pattern for CACC_compute, which consists of 16 44-bit
     values.
1  exp_fp44 = 6, mant_fp44 = 38;
2  min_exp_fp16 = -24, max_exp_fp16 = 15;
3  o_input = zeros(64); o_weight = zeros(16, 64);
4  exp_array = Nones(mant_fp44+1, 16);
5  sign_array = zeros(16);
6  for i in range(16) do
       // Derive the correct sign bit, exponent bits and mantissa bits for a 44-bit floating
         point value
7      sign_array[i], exp, mants = DERIVE_SIGN_EXP_MANTS(cacc_data[i]);
8      mants.INSERT_FRONT(ZERO(exp)? 0 : 1);
9      e = exp − (2^(exp_fp44−1) − 1) − ZERO(exp)?1 : 0;
       // Specify the power-of-2 exponent for all non-zero mantissa bits
10     for j in range(mant_fp44+1) do
11         if mants[j] != 0 then
12             exp_array[j,i] = e - j;
   // Represent each power-of-2 term in each FP44 value as the product of two FP16 values
13 for j in range(mant_fp44+1) do
       // The exponent of a FP16 input value shared by 16 CMAC units
14     sft = None;
15     if ALL_NONE(exp_array[j,:]) or (min(exp_array[j,:]) ≥ min_exp_fp16 and
         max(exp_array[j,:]) ≤ max_exp_fp16) then
16         sft = 0;
17     else if max(exp_array[j,:]) - min(exp_array[j,:]) > max_exp_fp16 - min_exp_fp16
         or min(exp_array[j,:]) < 2 * min_exp_fp16 or max(exp_array[j,:]) > 2 *
         max_exp_fp16 then
18         EXIT();
19     else if min(exp_array[j,:]) < min_exp_fp16 then
20         sft = min(exp_array[j,:]) - min_exp_fp16;
21     else
22         sft = max(exp_array[j,:]) - max_exp_fp16;
23     o_input[j] = FP16(2^sft);
24     for i in range(16) do
25         if exp_array[j,i] != None then
26             o_weight[i, j] = FP16((sign_array[i]? − 1 : 1) ∗ 2^(exp_array[j,i]−sft));
   Output 1: o_input, a list of 64 FP16 values used as the DNN inputs for all CMAC units.
   Output 2: o_weight, 16 lists of 64 FP16 values, each of which is used as the DNN weights
     for each CMAC unit.
```

Mapping SDP_compute's ATPG patterns to a DNN

For SDP, we perform ATPG for four different sub-units separately: precision conversion unit (which converts each CACC output from FP32 to FP16), ReLU unit (which implements the ReLU activation function), addition unit (for bias addition, element-wise matrix addition, and batch normalization), and multiply unit (for element-wise matrix multiplication and batch normalization).

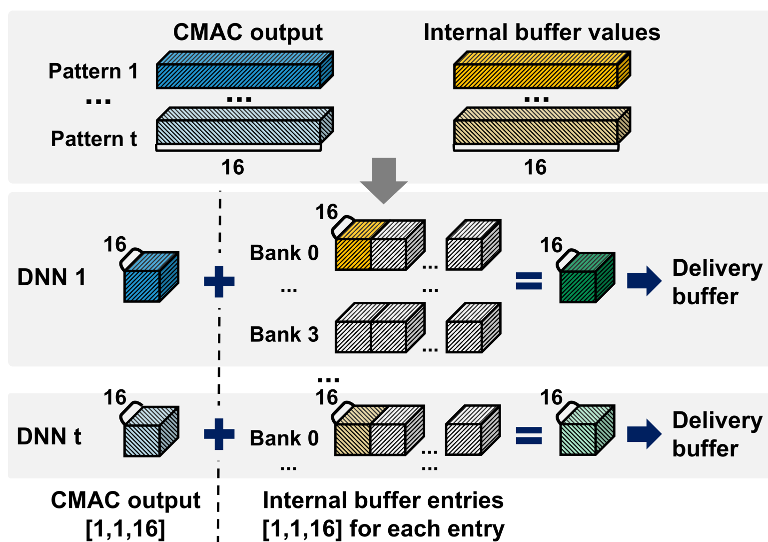


Figure 2.3: Illustration of CACC_compute's operations for the DNN test programs generated by Algorithms 2 and 1.

Each SDP sub-unit takes 16 FP32 values from the outputs of CACC as its inputs. Therefore, each ATPG pattern for a SDP sub-unit includes 16 32-bit values. In NVDLA, the accelerator pipeline can be configured so that the outputs of CACC can be directly connected to any SDP sub-unit. As such, the same mapping algorithm is applicable to all SDP sub-units, which requires the 16 values in each ATPG pattern to be mapped as the outputs of CACC. However, this cannot be done by simply writing the values to CACC's delivery buffer, which holds the outputs of CACC. This is because, in order for CACC to actually deliver the outputs to a SDP sub-unit, the control signals in CACC must indicate that the outputs are ready, but a software program cannot specify the values in these control signals

directly. Instead, we define a dummy DNN where all inputs and outputs are 0. During the execution of this dummy DNN, the delivery buffer of CACC is also initialized with the values specified in each ATPG pattern generated for a SDP sub-unit, which is equivalent to applying the ATPG pattern to this SDP sub-unit.

The detailed algorithm is shown in Algorithm 3. Given t total ATPG patterns for a SDP sub-unit, the algorithm generates a dummy DNN that follows the same structure as the DNNs generated in Algorithm 1: there is one layer in the dummy DNN. The input tensor shape is $1 \times 1 \times 64$, and the weight tensor shape is $1 \times 1 \times 64 \times 16$ (lines 3-4). The dummy DNN is executed t times to apply the t ATPG patterns. At the first cycle when the dummy DNN is executed to apply one pattern, we load the appropriate entry in CACC's delivery buffer (which is the first entry in bank 0 given the structure of the dummy DNN) with the 16 32-bit values obtained from the corresponding ATPG pattern (line 6).

The test responses from all SDP sub-units are observed from on-chip memory through SDP's output interface. Note that, the outputs of the ReLU unit, addition unit, and multiply unit all need to undergo FP32 to FP16 conversion before they are stored in memory, and our ATPG results reflect any loss of observability due to this conversion.

Algorithm 3: Mapping ATPG patterns of the sub-units in SDP_compute to equivalent test programs.

```
   Input: sdp_ins, a set of t ATPG patterns for a SDP sub-unit.
1  buffer_seq = [];
2  addr_bank = 0, addr_entry = 0;
3  input_tensor = zeros(1, 1, 64);
4  wt_tensor = zeros(1, 1, 64, 16);
5  for l in range(t) do
6      buffer_seq.PUSH_BACK([l, sdp_ins[l], [addr_bank, addr_entry]]);
   Output 1: a dummy DNN, where input_tensor and wt_tensor are the inputs and weights.
   Output 2: buffer_seq, a sequence of [l, val, addr] to indicate that when executing the
     dummy DNN to apply the lth ATPG pattern, the delivery buffer address addr should be
     loaded with value val.
```

Mapping PDP_compute's ATPG patterns to a DNN

The PDP unit implements pooling operations, and it supports standalone pooling layers. In our case study, we generate ATPG patterns for the max pooling compute unit (called pooling_MAX), which implements the max function given two operands. Each operand contains four FP16 values belonging to four consecutive input channels, so the max values for 4 channels are obtained at the same time. There are multiple instances of the pooling_MAX unit in NVDLA, which can be configured in different ways to support different pooling layer sizes. Due to space constraints and presentation clarity, we use 2x2 max pooling with stride=2, a common pooling layer in representative DNNs, to demonstrate our algorithm. The idea of our algorithm can be extended to other pooling configurations.

To support 2x2 max pooling with stride=2, two pooling_MAX units are used: one computes the row-wise pooling result for each row in two clock cycles, and is referred to as the *1d pooling_MAX unit*; and the other, called the *2d pooling_MAX unit*, receives the pooling result in each row and returns the max of the two in the third cycle. Our algorithm (Algorithm 4) maps the ATPG patterns to one DNN layer with dimensions $2h_in \times 4w_in \times 4$, where h_in and w_in can be any values that satisfy $h_in \times w_in \geq t$, and t is the total number of ATPG patterns. The exact choice of h_in and w_in values does not affect the performance of this DNN. In our algorithm, we use the square root of t to derive h_in and w_in (lines 1-2). For each ATPG pattern, the algorithm uses a $2 \times 4 \times 4$ region in the input tensor to test both the 1d and 2d pooling_MAX units (lines 11-14), so that the same set of ATPG patterns are applied to both units. The test responses can be directly observed as the output of this pooling layer. We illustrate this algorithm in Fig. 4.

2.4.2 Transition Faults in Compute Units

The high-level functional test generation flow for transition faults is similar to that for stuck-at faults. The difference between stuck-at and transition tests is that, for stuck-at tests, each

Algorithm 4: Mapping PDP_compute ATPG patterns to a DNN.

```
   Input: pdp_ins, a set of t ATPG patterns for the 2x2 max pooling function with stride=2.
1  h_in = ceil(sqrt(t));
2  w_in = ceil(t / h_in);
3  input_tensor = zeros(2* h_in, 4* w_in, 4);
4  for i in range(h_in) do
5      for j in range(w_in) do
6          if i * w_in + j ≥ t then
7              break;
8          else
9              dat0 = pdp_ins[i * w_in + j][0];
10             dat1 = pdp_ins[i * w_in + j][1];
               // Test the row-wise pooling_MAX unit
11             input_tensor[2 * i, 4 * j, :] = dat0;
12             input_tensor[2 * i, 4 * j + 1, :] = dat1;
               // Test the pooling_MAX unit across rows
13             input_tensor[2 * i, 4 * j + 2, :] = dat0;
14             input_tensor[2 * i + 1, 4 * j + 2, :] = dat1;
   Output: a DNN with one 2x2 pooling layer with stride=2. The input of the DNN is
    input_tensor.
```

ATPG pattern requires one cycle to launch the pattern, and the test response is captured at the end of the cycle. In contrast, for transition tests, test application spans two consecutive cycles. In each cycle, different values are assigned to all input signals of the corresponding compute unit to trigger 1-to-0 or 0-to-1 transitions in various circuit nodes. To check if a test pattern passes, the primary outputs of the compute unit is compared against the golden response at the end of the second cycle.

To extend our functional in-field self-test generation approach to support transition tests, we develop the detailed algorithms that map the two-cycle transition ATPG patterns for each compute unit to equivalent DNN test programs, as discussed below.

2.4.3 Mapping CMAC_compute's transition ATPG patterns to DNNs

The mapping algorithm for these units is derived by reverse-engineering NVDLA's dataflow/reuse algorithm (shown in Fig. 2.1): each CMAC_compute unit calculates the products of 64 {input, weight} pairs, and outputs the sum of the 64 products. Each CMAC_compute unit

operates on a different weight kernels, and each weight kernel is re-used for 16 cycles in each CMAC_compute unit. On the other hand, the same DNN inputs are dispatched to all 16 CMAC units, but a new set of input values are fetched each cycle.

While the input values can change once every cycle, changes in the weight values can only occur once every 16 cycles because each weight kernel is re-used for 16 cycles. Our algorithm (shown in Algorithm 5) takes this restriction into account.

Given t transition test patterns, each of which consists of two sets of 64 {input,weight} pairs to be applied across two consecutive cycles, the algorithm forms t DNNs (line 1), each of which corresponds to one ATPG pattern. The size of the input tensor of each DNN is $4 \times 5 \times 64$ (line 2). Assuming that there is no padding, this is the smallest input tensor size that requires more than 16 cycles of computation to allow the weight values to transition at least once. As shown in Fig. 2.4 (lines 4-5 in Algorithm 5), the 64 input values from cycle 1 of the ATPG pattern are placed in row 3, column 3 of the input tensor, corresponding to the positions that will be referenced in cycle 16 of the DNN execution[1]. For the input values from cycle 2 of the ATPG pattern, they are placed in row 0 and column 1, so these values will be used in cycle 17 of the DNN execution. All other values in the input tensor are 0. In terms of weights, there are 16 identical weight kernels in the DNN. The size of each weight kernel is $1 \times 2 \times 64$ to hold the two-cycle weight values in the ATPG pattern (lines 3, 7-8). As shown in Fig. 2.4, during cycles 1-16 of the DNN execution, the first $4 \times 4 \times 64$ input tensor is convoluted with the first $1 \times 1 \times 64$ weight tensor for all 16 weight kernels through parallel execution of the 16 CMAC_compute units (following NVIDIA's reuse algorithm). Specifically, at cycle 16, the 64 {input, weight} pairs specified in the first cycle of the ATPG pattern are deployed to all 16 CMAC_compute units. At the 17th cycle, changes in both input and weight values occur, which is equivalent to applying the 64 {input, weight} pairs specified in the second cycle of the ATPG pattern to all 16 CMAC_compute units.

1. For clarity, the cycle numbers of DNN execution are specified with respect to the individual compute unit, i.e., ignoring prior pipeline stages in the accelerator, in this section.

To check if this test passes, we just need to retrieve the partial sums generated at cycle 17 (in positions $[0, 0, 0 : 15]$ of the output tensor), which correspond to the actual test response from all 16 CMAC_compute units, because the partial sums in these output positions are all 0 prior to cycle 17. These values can be observed through CACC's internal buffer[2], and then compared with the golden responses.

Algorithm 5: Mapping CMAC_compute's transition ATPG patterns to equivalent DNNs.

```
   Input 1: t, the number of ATPG patterns.
   Input 2: ins, a list of the t input patterns generated by ATPG.
   Input 3: wts, a list of the t weight patterns generated by ATPG.
1  for l in range(t) do
2      input_tensor_l = zeros(4, 5, 64);
3      wt_tensor_l = zeros(1,2,64,16);
       // Prepare the input tensor.
4      input_tensor_l[3,3,:] = ins[l][0];
5      input_tensor_l[0,1,:] = ins[l][1];
       // Prepare the weight tensor.
6      for k in range(16) do
7          wt_tensor_l[0,0,:,k] = wts[l][0];
8          wt_tensor_l[0,1,:,k] = wts[l][1];
   Output: t DNNs, where input_tensor_l and wt_tensor_l are the inputs and weights of
     the lth DNN. Each DNN performs convolution operations for input_tensor_l and
     wt_tensor_l.
```

2.4.4 Mapping CACC_compute's transition ATPG patterns to DNNs

Each transition ATPG pattern for CACC_compute consists of two inputs for each of the two cycles: (1) 16 44-bit FP values, corresponding to the outputs of the 16 CMAC units, and 16 48-bit FP values, corresponding to the internal buffer values to be added to each of the CMAC outputs. The element-wise addition of these two inputs are computed by CACC_compute each cycle.

For each ATPG pattern, the corresponding DNN test program for CACC_compute is

2. We follow the same assumption as [34] that the buffers in DL accelerators are accessible by test programs.

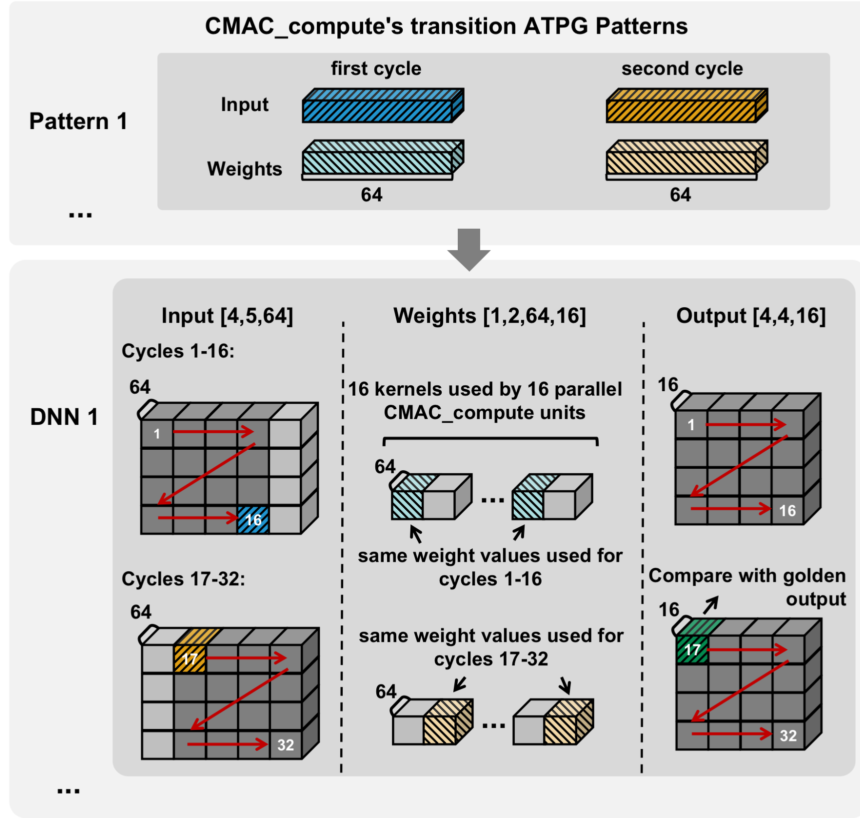


Figure 2.4: Illustration on why applying the DNN generated by Algorithm 5 is equivalent to applying the corresponding transition ATPG pattern for CMAC_compute.

the smallest DNN that takes 2 cycles to execute, as illustrated in Figure 2.5. Since the internal buffer of CACC_compute is accessible by test programs, the 48-bit FP values can be directly loaded to the internal buffer. During cycle 1 (2), the 16 values in {bank 0 (1), address 0} of the internal buffer are accumulated with the 16 CMAC outputs. Therefore, we assign the 48-bit FP values from the first (second) cycle of a transition ATPG pattern into {bank 0 (1), address 0} of the internal buffer.

For the outputs of CMAC_compute to match the 44-bit FP values specified in the ATPG pattern for two consecutive cycles, our technique first applies Algorithm 2. For the cycle-1 values, Algorithm 2 reversely map each of the 16x 44-bit FP values into 64x {input, weight} pairs, which serve as the inputs to one CMAC_compute unit (lines 6-25). This part of the algorithm is exactly as the same as Algorithm 2: each 44-bit FP value is first expressed as

a sum of power-of-2 terms according to the definition of the 44-bit FP datatype, which can then be expressed as the product of two 16-bit floating point values (following lines 13-25 in Algorithm 2).

The same process also applies to the 16x 44-bit floating values from cycle 2 of the ATPG pattern (that's why lines 6-12 in Algorithm 2 are looped twice, and lines 26-28 are added, compared to Algorithm 2). However, For the second cycle, we fix the weight values to be exactly the same as the ones in the first cycle, and then determine the corresponding input values accordingly (lines 26-28). The reason why we fix the weight values across the two cycles is because, when mapping CMAC_compute inputs to equivalent DNNs, we must take NVDLA's dataflow restriction into account, which only allows the weight values of each CMAC_compute unit to change at most once every 16 cycles. Thus, fixing the weight values minimizes test time. Given the fixed weight values, it is possible that we cannot generate input values such that the corresponding CMAC_compute outputs match the 44-bit FP values specified in the second cycle of the ATPG pattern (line 26). However, these cases can only occur if the output neuron values or input/weight values are extremely large, which is extremely unlikely to appear in today's DNN workloads.

After applying Algorithm 2, Algorithm 7 is used to map the two sets of 64x {input, weight} pairs (with the same weight values) for all 16 CMAC_compute units into an equivalent DNN. The high-level idea of Algorithm 7 is the same as Algorithm 5, except that: (1) the weight values for each CMAC_compute unit can be different; (2) the size of the input tensor of the DNN is $1 \times 2 \times 64$ (instead of $4 \times 5 \times 64$), because the weight values in each CMAC_compute unit do not need to change.

Combining Algorithms 2 and 7, a set of DNN test programs, one per transition ATPG pattern, are generated for CACC_compute, as illustrated in Fig. 2.5. When a DNN test program generated by these two algorithms is executed, the 16x 44-bit FP values in the corresponding ATPG pattern are applied to the corresponding input ports of the CACC_compute

unit in two consecutive cycles. Moreover, the two sets of 16x 48-bit FP values in the same ATPG pattern are also loaded into the appropriate internal buffer addresses (lines 11-12 in Algorithm 7), so that the accumulation operations are performed as intended. To observe the test responses, we retrieve the outputs of CACC_compute directly from CACC's delivery buffer.

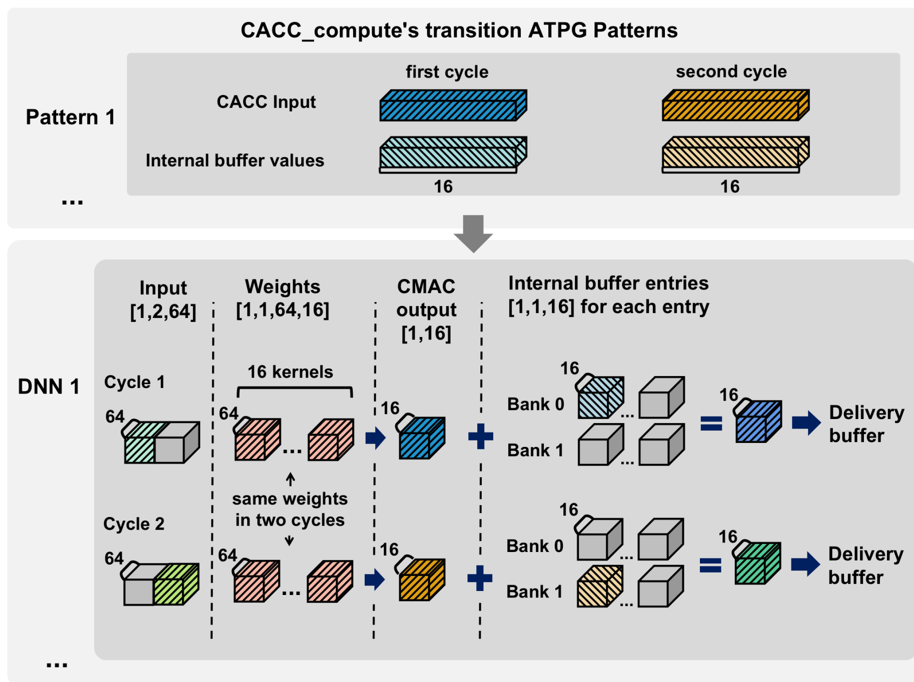


Figure 2.5: Illustration on why applying the DNN generated by Algorithms 6 and 7 is equivalent to applying the corresponding transition ATPG pattern for CACC_compute.

2.4.5 Mapping SDP_compute's transition ATPG patterns to functional test programs

We focus on four major sub-units in SDP_compute, similar to Algorithm 3: (1) the precision conversion unit; (2) the ReLU activation function unit; (3) the addition unit (which is used to perform bias addition, element-wise matrix addition, and batch normalization); and (4)

Algorithm 6: Mapping CACC_compute's transition ATPG pattern to CMAC_compute inputs.

```
   Input: cacc_data, one ATPG pattern for CACC_compute
1  exp_fp44 = 6, mant_fp44 = 38;
2  min_exp_fp16 = -24, max_exp_fp16 = 15;
3  o_input = zeros(2, 64); o_weight = zeros(16, 64);
4  exp_array = Nones(2, mant_fp44+1, 16); sign_array = zeros(2, 16);
5  for i in range(2) do
6      for j in range(16) do
           // Derive the correct sign bit, exponent bits and mantissa bits for a 44-bit floating
             point value
7          sign_array[i][j], exp, mants = DERIVE_SIGN_EXP_MANTS(cacc_data[i][j]);
8          mants.INSERT_FRONT(ZERO(exp)? 0 : 1);
9          e = exp − (2^(exp_fp44−1) − 1) − ZERO(exp)?1 : 0;
           // Specify the power-of-2 exponent for all non-zero mantissa bits
10         for k in range(mant_fp44+1) do
11             if mants[k] != 0 then
12                 exp_array[i,k,j] = e - k;

   // Represent each power-of-2 term in each FP44 value as the product of two FP16 values
13 for j in range(mant_fp44+1) do
       // The exponent of a FP16 input value shared by 16 CMAC units
14     sft = None;
15     if ALL_NONE(exp_array[0, j,:]) or (min(exp_array[0, j,:]) ≥ min_exp_fp16
         and max(exp_array[0, j,:]) ≤ max_exp_fp16) then
16         sft = 0;
17     else if max(exp_array[0, j,:]) - min(exp_array[0, j,:]) > max_exp_fp16 -
        min_exp_fp16 or min(exp_array[0, j,:]) < 2 * min_exp_fp16 or
        max(exp_array[0, j,:]) > 2 * max_exp_fp16 then
18         EXIT();
19     else if min(exp_array[0, j,:]) < min_exp_fp16 then
20         sft = min(exp_array[0, j,:]) - min_exp_fp16;
21     else
22         sft = max(exp_array[0, j,:]) - max_exp_fp16;
23     o_input[0, j] = FP16(2^sft);
24     for i in range(16) do
25         if exp_array[0,j,i] != None then
26             o_weight[i, j] = FP16((sign_array[0, i]? − 1 : 1) ∗ 2^(exp_array[0,j,i]−sft));
27             if exp_array[1,j,i] - exp_array[0,j,i] + sft > max_exp_fp16 or
                 exp_array[1,j,i] - exp_array[0,j,i] + sft < min_exp_fp16 then
28                 EXIT();
29             o_input[1, j] = FP16((sign_array[1, i] XOR sign_array[0, i]? − 1 :
                 1) ∗ 2^(exp_array[1,j,i]−exp_array[0,j,i]+sft));

   Output 1: o_input, FP16 values used as CMAC_compute inputs with shape 2 × 64.
     Positions [0, 0 : 63] are used in the first cycle, positions [1, 0 : 63] are used in the second
     cycle.
   Output 2: o_weight, FP16 values used as CMAC weights with shape 16 × 64. Each 64x
     values are used by each CMAC_compute units in both cycles.
```

Algorithm 7: Mapping the outputs of Algorithm 7 to equivalent DNNs.

```
  Input 1: t, the number of ATPG patterns.
  Input 2: ins, a list of the t input patterns generated by Algorithm 6.
  Input 3: wts, a list of the t weight patterns generated by Algorithm 6.
  Input 4: buffer_ins, a list of t internal buffer patterns generated by ATPG.
1 buffer_seq = [];
2 addr_bank_0 = 0, addr_entry_0 = 0;
3 addr_bank_1 = 1, addr_entry_1 = 0;
4 for l in range(t) do
5     input_tensor_l = zeros(1, 2, 64);
6     wt_tensor_l = zeros(1,1,64,16);
      // Prepare the input tensor.
7     input_tensor_l[0,0,:] = ins[l][0];
8     input_tensor_l[0,1,:] = ins[l][1];
      // Prepare the weight tensor.
9     for k in range(16) do
10        wt_tensor_l[0,0,:,:] = transpose(wts);
      // Prepare internal buffer values.
11    buffer_seq.PUSH_BACK([l, buffer_ins[l][0], [addr_bank_0, addr_entry_0]]);
12    buffer_seq.PUSH_BACK([l, buffer_ins[l][1], [addr_bank_1, addr_entry_1]]);
  Output 1: t DNNs, where input_tensor_l and wt_tensor_l are the inputs and weights of
    the lth DNN. Convolution operations are performed using input_tensor_l and
    wt_tensor_l for each DNN.
  Output 2: buffer_seq, a sequence of [l, val, addr] to indicate that when executing the lth
    DNN for the lth ATPG pattern, the internal buffer address addr should be loaded with
    value val.
```

the multiplication unit (which is used to perform element-wise matrix multiplication and batch normalization). Combinational ATPG patterns are generated for each sub-unit.

In NVDLA, the outputs of CACC (16x FP32 values) can be configured to directly connect to any SDP sub-unit. As such, the same mapping algorithm is applicable to all SDP sub-units, which requires two sets of 16x FP32 values in each ATPG pattern to be mapped as the outputs of CACC across two cycles.

The algorithm that maps one transition ATPG pattern is equivalent to mapping two stuck-at ATPG patterns in two consecutive cycles (following Algorithm 8): we define a dummy DNN where all inputs and outputs are 0. The purpose of the dummy DNN is to activate the SDP such that at least two cycles of operations in a SDP sub-unit can be performed, while preventing the system from scheduling other workloads into the accelerator.

The smallest dummy DNN that can fulfill this purpose consists of one layer, and the size of the input/weight tensor is $1 \times 2 \times 64$ and $1 \times 1 \times 64 \times 16$, respectively (lines 4-5). During the execution of this dummy DNN, the appropriate entries in CACC's delivery buffer are loaded with the values specified in each ATPG pattern generated for a SDP sub-unit (lines 7-8 in Algorithm 8), which is equivalent to applying the ATPG pattern to the SDP sub-unit. The test responses from all SDP sub-units are observed from on-chip memories.

Algorithm 8: Mapping transition ATPG patterns of the sub-units in SDP_compute to equivalent test programs.

```
  Input: sdp_ins, a set of t transition ATPG patterns for a SDP sub-unit.
1 buffer_seq = [];
2 addr_bank_0 = 0, addr_entry_0 = 0;
3 addr_bank_1 = 1, addr_entry_1 = 0;
4 input_tensor = zeros(1, 2, 64);
5 wt_tensor = zeros(1, 1, 64, 16);
6 for l in range(t) do
7     buffer_seq.PUSH_BACK([l, sdp_ins[l][0], [addr_bank_0, addr_entry_0]]);
8     buffer_seq.PUSH_BACK([l, sdp_ins[l][1], [addr_bank_1, addr_entry_1]]);
  Output 1: a dummy DNN, where input_tensor and wt_tensor are the inputs and weights.
  Output 2: buffer_seq, a sequence of [l, val, addr] to indicate that when executing the
    dummy DNN to apply the lth ATPG pattern, the delivery buffer address addr should be
    loaded with value val.
```

2.4.6 Mapping PDP_compute's transition ATPG patterns to a DNN

In our case study, we focus on 2x2 max pooling with stride=2, which is commonly used in various representative DNNs (our algorithm can be extended to other pooling operations). Three pooling_MAX units are used in this case: two of them (referred to as the 1d pooling_MAX units) performs the row-wise max pooling operations for two rows of the pooling layer input in one clock cycle; and the third unit (referred to as the 2d pooling_MAX unit) receives the pooling result in each row and returns the max of the two in the second cycle. Each pooling_MAX unit takes two consecutive operands in the input tensor as inputs, where each operand contains four 16-bit floating-point values from four consecutive input channels.

Algorithm 9 shows how all transition ATPG patterns for the pooling_MAX unit can be mapped to one DNN max pooling layer, which allows the patterns to be applied to all three pooling_MAX units. The size of this layer is $2h_in \times 12w_in \times 4$, where h_in and w_in can be any values that satisfy $h_in \times w_in \geq t$, and t is the total number of ATPG patterns. The exact choice of h_in and w_in values does not affect the performance of this DNN. In our algorithm, we use the square root of t to derive h_in and w_in (lines 1-2).

For each ATPG pattern, the algorithm uses two $2 \times 4 \times 4$ regions in the input tensor to test the two 1d pooling_MAX units (lines 11-14), as shown in Fig. 2.6, across four cycles. The next $2 \times 4 \times 4$ region of the DNN is used to test the 2d pooling_MAX unit (lines 15-16). Such a $2 \times 12 \times 4$ region is repeated t time to cover all ATPG patterns.

Algorithm 9: Mapping PDP_compute's transition ATPG patterns to a DNN.

```
   Input: pdp_ins, a set of t ATPG patterns for the pooling_MAX unit.
1  h_in = ceil(sqrt(t));
2  w_in = ceil(t / h_in);
3  input_tensor = zeros(2 * h_in, 12 * w_in, 4);
4  for i in range(h_in) do
5      for j in range(w_in) do
6          if i * w_in + j ≥ t then
7              break;
8          else
9              cycle0_dat = pdp_ins[i * w_in + j][0,:];
10             cycle1_dat = pdp_ins[i * w_in + j][1,:];
               // Test the first 1d pooling_MAX unit
11             input_tensor[2*i, 4*j : 4*j+2, :] = cycle0_dat;
12             input_tensor[2*i, 4*j+2 : 4*j+4, :] = cycle1_dat;
               // Test the second 1d pooling_MAX unit
13             input_tensor[2*i+1, 4*j+3 : 4*j+5, :] = cycle0_dat;
14             input_tensor[2*i+1, 4*j+5 : 4*j+7, :] = cycle1_dat;
               // Test the 2d pooling_MAX unit
15             input_tensor[2*i : 2*i+2, 4*j+8, :] = cycle0_dat;
16             input_tensor[2*i : 2*i+2, 4*j+10, :] = cycle1_dat;
   Output: a DNN with one 2x2 pooling layer with stride=2. The input of the DNN is
     input_tensor.
```

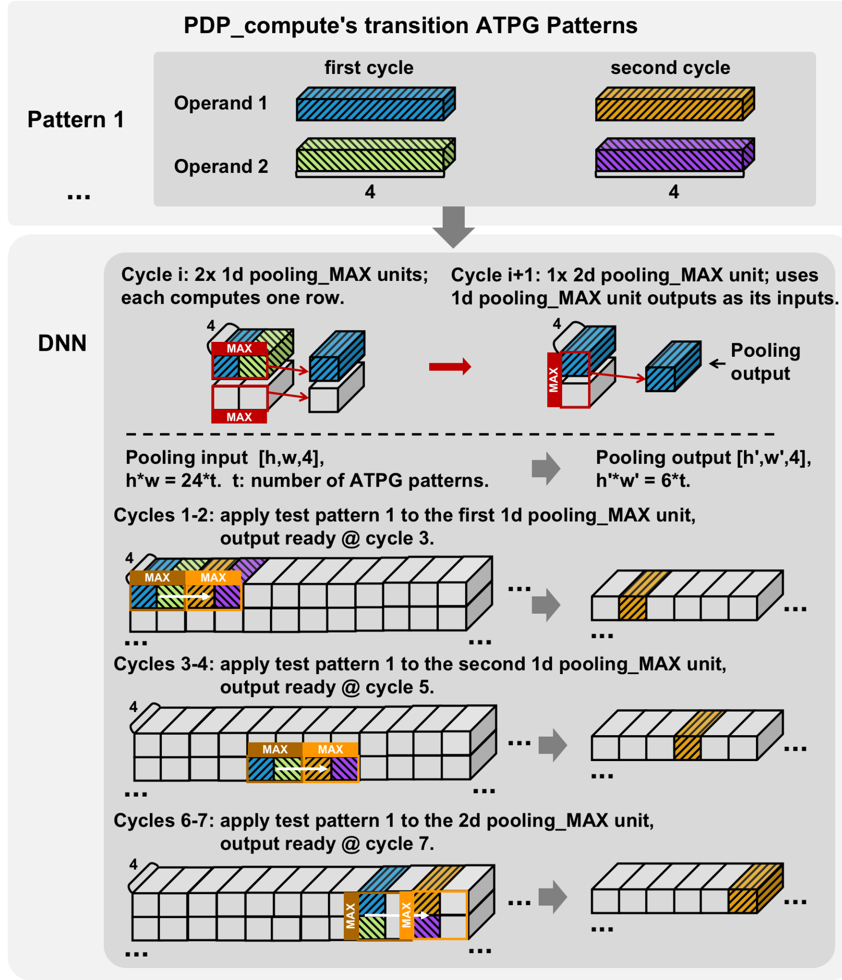


Figure 2.6: Illustration on why applying the DNN generated by Algorithm 9 is equivalent to applying all transition ATPG patterns to all pooling_max units.

2.4.7 Control Units

To generate the input/weight patterns that satisfy the n-linearly-independent property for a given DNN layer, our methodology is to generate random input and weight values, and then check if this condition is met. If not, we may discard the current values and try again. Alternatively, we may manually tweak the pattern such that the n-linearly-independent condition is satisfied.

Note that, in Theorems 1 and 2, the activation function must be invertible. However, ReLU, which is commonly used as the activation function in many state-of-the-art DNNs, is not invertible. Fortunately, ReLU is invertible for all positive input values. Therefore, we

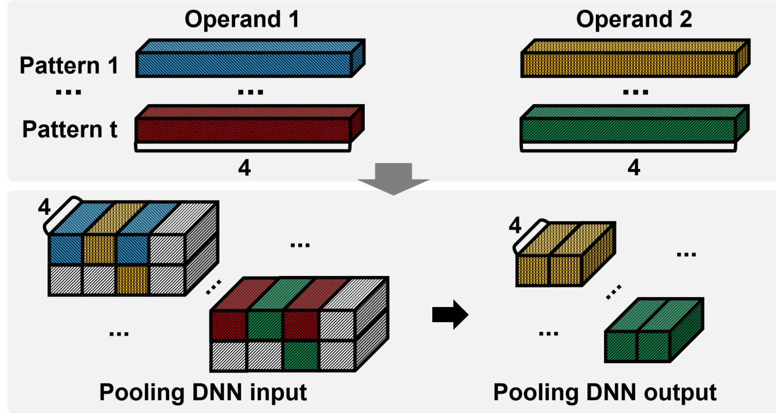


Figure 2.7: Illustration of Algorithm 9.

only use positive input and weight values. We also carefully choose a range for inputs/weights such that the maximum output neuron value does not exceed the dynamic range of the DNN's data precision type.

We evaluate the methodology discussed above using the largest layer of two representative DNNs: GoogleNet and Yolo[35, 36]. We run 100 Matlab experiments to randomly generate the inputs/weights of each workload. All input/weight values satisfy the n-linearly-independent condition check on the first try. Moreover, we verify that these patterns can also be used to test other conditions such as correct data precision and pipeline configurations.

2.4.8 Evaluation Results and Discussions

In this section, we present the test coverage, test time, and test storage results to demonstrate the effectiveness and practicality of our functional test generation approach.

2.4.9 In-Field Self-Test Coverage

We perform combinational ATPG on various compute units in NVDLA using the Synopsys TestMax tool. The transition test coverage results for the compute units are presented in Table 2.1. We report both the test coverage obtained from the ATPG tool, as well as functional test coverage, which is defined as the percentage of detected faults out of the

total number of faults that are detectable using functional tests. We also include the stuck-at test coverage results. The test coverage is very high: 99.8% for stuck-at, and 98.8% for transition. Combined with the test coverage results for control units (100% for both stuck-at and transition), overall >99.9% stuck-at and >99.0% transition coverage is achieved.

Table 2.1: Test coverage results for NVDLA. Fault models: single stuck-at faults and transition faults for compute units; single-variable control faults for control units.

Modules	**Num. of faults**	**Test coverage**		**Functional test coverage**	
		Transition	Stuck-at	Transition	Stuck-at
CMAC_compute	9187872	99.0%	99.8%	99.0%	99.8%
CACC_compute	286272	98.2%	98.9%	99.1%	99.9%
SDP_compute	625662	91.6%	96.3%	95.1%	99.9%
PDP_compute	1944	100%	100%	100%	100%
Overall (compute units)	10101750	98.5%	99.6%	98.8%	99.8%
MAC control units	Functional test coverage = 100%				

2.4.10 In-Field Self-Test Time Results

To obtain the in-field self-test time for the compute units in NVDLA, we first apply Algorithms 1-5 to create DNN test programs. Next, for the DNN test programs generated for CMAC_compute, CACC_compute, and SDP_compute, we obtain the run-time of each DNN through RTL simulation, and apply a 3-cycle configuration time per DNN. For CMAC_compute, we also stop the execution of each DNN as soon as the 17th partial sum in each output channel (which corresponds to the actual test response) is generated and stored

in CACC's internal buffer to minimize test time. For the DNN test programs generated for PDP_compute, since its run time is dominated by memory access time, we use NVDLA's performance evaluation tool [37] to estimate its total execution time.

In Table 2.2, we summarize the key characteristics of the DNN test programs generated for each compute unit, and report their run time results (assuming a 2.5GHz clock). The total in-field self-test time is 0.44ms for the transition functional tests. We also include the test time results for the stuck-at tests in Table 2.2, and the total run time for the stuck-at tests is 0.14ms.

For all compute units, the test time of transition tests is higher than that of stuck-at tests, because it takes at least two clock cycles to detect transition faults, compared to only one cycle for stuck-at faults. Specifically, for the CMAC_compute unit, it takes 17 cycles to induce a transition in each test pattern. As discussed in Sec. 3, this is because varying input and weight values across two consecutive cycles can only occur once every 16 cycles, as constrained by NVDLA's reuse algorithm. If NVDLA allows both input and weight values to change every cycle, the transition functional test time for CMAC_compute can be reduced to 0.1ms.

We also consider the test time for NVDLA's control units (the control units are depicted in Fig. 2.1). The time to apply functional tests on control units depends on the DNNs that are currently deployed, because different layers in the DNNs themselves are used as the functional test programs. These tests can detect all single-variable-type fault models (including the single stuck-at and transition fault models). Therefore, the transition test time for the control units is the same as the stuck-at test time (reproduced in Table 2.3).

In Table 2.3, we also summarize the total in-field self-test time (i.e., compute unit test time + control unit test time), assuming that the DNN currently deployed is one of four representative DNNs: GoogleNet, YOLOv3, DenseNet, and EfficientNet B2. The total transition test time ranges from 2.36ms-17.27ms across the different DNNs. The total test

Table 2.2: Functional in-field self-test time results for the compute units in NVDLA (clock frequency = 2.5GHz).

Compute units	Num. ATPG patterns	Num. equivalent DNNs	DNN layer dimensions	Test time (ms)
Transition tests				
CMAC_ compute	3010	3010	Input: $4 \times 5 \times 64$ Weight: $1 \times 2 \times 64 \times 16$	0.39
CACC_ compute	549	549	Input: $1 \times 2 \times 64$ Weight: $1 \times 1 \times 64 \times 16$	0.02
SDP_ compute	3243	1		0.02
PDP_ compute	61	1	Input: $16 \times 96 \times 4$	0.01
Total	6863	N/A	N/A	0.44
Stuck-at tests				
CMAC_ compute	5802	5802	Input: $1 \times 1 \times 64$ Weight: $1 \times 1 \times 64 \times 16$	0.093
CACC_ compute	457	457		0.007
SDP_ compute	2293	1		0.037
PDP_ compute	56	1	Input: $14 \times 32 \times 4$	1e-5
Total	8608	N/A	N/A	0.137

time for the transition tests is similar to that for the stuck-at tests, because test time for the control units dominates overall test time and is the same for both stuck-at and transition tests.

If a 0-to-1 (or 1-to-0) transition fault is detected, then the corresponding stuck-at-0 (or stuck-at-1) fault is detected as well. Thus, by applying transition tests, high stuck-at test coverage is also achieved. The coverage gap between stuck-at and transition tests can be filled by a small top-off stuck-at tests that incur negligible test time.

Table 2.3: Functional in-field self-test time results for the control units and total test time results (frequency = 2.5GHz). All results are in milliseconds.

Network	Control unit test time	Total transition test time	Total stuck-at test time
GoogleNet	1.92	2.36	2.06
YOLOv3	10.37	10.81	10.51
DenseNet	16.84	17.28	16.98
EfficientNet B2	1.13	1.57	1.27

2.4.11 Test storage results

The total storage space required to store our functional in-field self-tests is obtained by adding the following terms of each DNN test program (for both compute and control units): (1) the size of all inputs, weights, and outputs of each DNN test program; (2) the size of golden test responses; and (3) the space required to store configuration information of each DNN test program, such as the input/weights/output addresses. Table 2.4 reports the test storage results.

For compute units, the storage requirement for transition tests is higher than that for stuck-at tests, because the inputs and weights of the transition DNN test programs must be made bigger than those in the stuck-at DNN test programs to allow both the launch and capture operations in order to detect transition faults.

The DNN test programs for control units detect both transition and stuck-at faults, and their storage requirement depends on the DNNs currently being deployed (as shown in table 2.4). The total test storage required to detect the transition faults in both compute and control units is less than $600MB$ among the different DNNs used in our evaluation. When combined with stuck-at tests, the total storage requirement is less than $650MB$ among all DNNs, which is practical given the off-chip storage capacities available today.

Table 2.4: Test storage results for NVDLA. All results are in MBytes.

Compute units	CMAC_ compute	CACC_ compute	SDP_ compute	PDP_ compute	Total compute
Transition test storage	34.99	4.62	1.29	0.84	41.73
Stuck-at test storage	13.45	1.12	0.30	0.01	14.88
Total	48.44	5.74	1.59	0.61	56.37

Network	**Control unit test storage**	**Transition test storage**[1]	**Stuck-at test storage**[2]	**Total**[3]
GoogleNet	100.99	142.72	115.87	157.60
YOLOv3	569.18	610.91	584.06	625.79
DenseNet	581.79	623.52	596.67	638.40
EfficientNet B2	99.97	141.70	114.85	156.58

1 = control unit test storage + transition test storage for compute units;

2 = control unit test storage + stuck-at test storage for compute units;

3 = total transition test coverage + total stuck-at test storage - control unit test storage.

CHAPTER 3

FIDELITY: EFFICIENT RESILIENCE ANALYSIS FRAMEWORK FOR DEEP LEARNING ACCELERATORS

The content of this chapter was previously published in our paper titled "FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators" in MICRO20 [38].

We present a resilience analysis framework, called FIdelity, to accurately and quickly analyze the behavior of hardware errors in deep learning accelerators. Our framework enables resilience analysis starting from the very beginning of the design process to ensure that the reliability requirements are met, so that these accelerators can be safely deployed for a wide range of applications, including safety-critical applications such as self-driving cars.

Existing resilience analysis techniques suffer from the following limitations: 1. general-purpose hardware techniques can achieve accurate results, but they require access to RTL to perform time-consuming RTL simulations, which is not feasible for early design exploration; 2. general-purpose software techniques can produce results quickly, but they are highly inaccurate; 3. techniques targeting deep learning accelerators only focus on memory errors.

Our FIdelity framework overcomes these limitations. FIdelity only requires a minimal amount of high-level design information that can be obtained from architectural descriptions/block diagrams, or estimated and varied for sensitivity analysis. By leveraging unique architectural properties of deep learning accelerators, we are able to systematically model a major class of hardware errors – transient errors in logic components – in software with high fidelity. Therefore, FIdelity is both quick and accurate, and does not require access to RTL.

We thoroughly validate our FIdelity framework using Nvidia's open-source accelerator called NVDLA, which shows that the results are highly accurate – out of 60K fault injection experiments, the software fault models derived using FIdelity closely match the behaviors

observed from RTL simulations. Using the validated FIdelity framework, we perform a large-scale resilience study on NVDLA, which consists of 46M fault injection experiments running various representative deep neural network applications. We report the key findings and architectural insights, which can be used to guide the design of future accelerators.

3.1 Introduction

Deep learning (DL) accelerators have been deployed in a wide range of application domains, from edge computing, self-driving cars, to cloud servers [33, 39]. Hardware error resilience is a top priority for these accelerators. The importance of resilience for safety-critical applications such as self-driving cars has already been pointed out by Nvidia[22], Tesla[40], and many others. Furthermore, in general, resilience analysis provides better understanding of application/design requirements, and enables efficient architectural exploration to achieve optimal tradeoffs between power, performance, area, and reliability. It also provides a means to quantitatively compare resilience properties of different designs/applications (e.g., for benchmarking purposes). Resilience analysis can even be used to assess the impact of fault attacks (e.g., using hardware trojans, injecting optical/electromagnetic disturbances, exploiting variations, and so on, for malicious purposes), and to guide the design of secure architectures. Because of its importance, resilience analysis should be performed starting from the very beginning of the design process.

There exist several resilience analysis studies targeting DL accelerators [17, 41, 42, 43], but they only focus on memory errors. These studies are not sufficient, because transient errors in logic components, referred to as *logic transient errors* for short, are a major reliability concern. Our results show that, for a DL accelerator in which resilience protection of sequential elements is not provided, the FIT (failure in time) rate of these sequential elements (> 9.5) is significantly higher than the stringent automotive safety requirement (< 0.2), even just as a result of random transient errors that occur infrequently. Therefore,

in this chapter, we focus on a new resilience analysis framework that targets logic transient errors in DL accelerators.

A well-known resilience analysis approach is to perform large-scale fault injection experiments. However, existing fault injection techniques suffer from several key limitations. In general, RTL-level fault injection techniques (or mixed-mode techniques that combine RTL and software simulations) can achieve accurate results. However, RTL is not likely to be available (or easily modified to test out different implementations) during the early phases of the design process. Even if RTL is available, simulation time would be prohibitively long. On the other hand, software-level fault injection techniques are quick, but their accuracy cannot be guaranteed [10]. Various optimizations for fault injection techniques have been developed for CPUs [14, 15, 16], but they cannot be directly applied to DL accelerators.

To overcome the above challenges, we present FIdelity, which models hardware logic transient errors in software with high fidelity. The accurate mapping is possible because of the following insights: as a special-purpose design, the majority of the hardware operations in DL accelerators closely match software operations. Moreover, because of the well-defined dataflow and scheduling algorithms, all operations that are affected by a given hardware error, and how they are affected by the error, can be systematically derived. **The novel contribution of this work is that, using just a minimal amount of hardware information, our framework is able to generate accurate software fault models for both datapath and control components, without the need to access RTL.** As such, the behavior of a logic transient error can be modeled in software to enable quick and accurate software fault injection.

To validate our framework, we apply it to Nvidia's open-source DL accelerator, NVDLA [39], and obtain its software fault models. We then perform 60K RTL fault injection experiments using various representative DNN workloads. By manually analyzing all RTL fault injection cases that lead to non-masked outcomes, we confirm that, for the datapath, the

software fault models derived using FIdelity capture the *exact* fault behaviors obtained from RTL simulations. For the control portion, FIdelity's models closely match with RTL results.

Using the validated FIdelity framework, we perform a large-scale resilience study on NVDLA, which consists of 46M fault injection experiments. Our workloads include various representative deep neural networks: CNNs used for image classification tasks (Inception, Resnet, Mobilenet), the Yolo CNN used for object detection tasks, and the Transformer network for language translation tasks. The key findings are:

1. Our results quantitatively demonstrate the crucial need for resilience analysis and protection solutions for DL accelerators. Although DL workloads exhibit certain tolerance to errors, such tolerance alone cannot guarantee that a DL accelerator will meet the resilience requirement of a target application.

2. Our results reveal how hardware design choices, data precision, and correctness metrics affect the overall resilience of the design, as well as fault properties that can be leveraged to develop new resilience techniques.

In summary, the major contributions of this work are:

1. We create the FIdelity resilience analysis framework for DL accelerators.

2. We thoroughly validate this framework.

3. We use FIdelity to perform large-scale (46M) fault injection experiments, which reveals new resilience knowledge and architectural insights.

This chapter is organized as follows. We present the FIdelity framework in Sec. 3.2, and validation methodology and results in Sec. 3.3. Large-scale fault injection experiments and results are discussed in Sec. 3.4, followed by related work in Sec. 3.5.

3.2 The FIdelity Framework

FIdelity is a new logic transient error analysis framework targeting digital DNN inference accelerators. FIdelity provides high-fidelity software models for all single-cycle, single FF

bit-flip errors (or multiple single-cycle bit-flips in a single register), which are the most prominent abstraction for transient errors including soft errors [10] and voltage variations [11]. Thus, it achieves quick and accurate results without relying on the availability of RTL. FIdelity is broadly applicable to a wide variety of DL accelerators because it leverages their common architectural properties. An overview of the framework is shown in Fig. 3.2.

3.2.1 Insights and Novelty

Existing software fault injection techniques model a logic transient error as a single-cycle bit-flip in a single architectural (i.e., software-visible) state, which is highly inaccurate because in reality logic transient errors can result in various architectural effects, including single or multiple bit-flips in one or more architectural states, system time-outs, and so on. For complex general-purpose designs such as CPUs, it is very challenging to model the effects of these errors without detailed RTL models. However, for DL accelerators, we identify four key properties pertaining to resilience analysis, which suggest that the architectural effects of an FF bit-flip can be accurately and systematically derived.

Accelerator Property (1): For an FF bit-flip to cause errors in the final output of a DL application, it must first cause errors in the output neurons of the current DNN layer. Moreover, the bit-flip cannot directly affect the computation in other layers. Thus, to capture the effects of an FF bit-flip, it is *equivalent* to first obtaining the list of faulty output neurons in the current layer, as well as their faulty values, and then determining how these faulty neurons affect the final application output through software simulation.

Accelerator Property (2): As a special-purpose design, the datapath FFs in a DL accelerator closely match the variables in a DNN software framework. For example, in NVDLA, datapath FFs only store the following information: a DNN's inputs, weights, bias values, partial sums, and output values, all of which are visible in software.

Accelerator Property (3): Due to the regular structure and precisely-defined dataflow

architecture, in every cycle, the value stored in a datapath FF only affects a deterministic set of output neurons in the current DNN layer. Note that, multiple neurons can be affected by a single-cycle FF value because an important design principle of DL accelerators is to *reuse* the input/weight/partial sum values effectively to optimize energy efficiency[29].

Accelerator Property (4): Control FFs in DL accelerators are classified into two categories: local and global. A local control FF directly interacts with a deterministic set of datapath FFs, so its effects on the output neurons can be derived based on the corresponding datapath FFs. On the other hand, global FFs (e.g., FFs that store the number of kernels or data precision in the current DNN layer) control the computations of a large number of, or even all, output neurons.

Therefore, given a *fault site*, which specifies both the FF where a fault is injected and the cycle during which the fault is injected, *software fault models* that accurately capture hardware logic transient errors can be obtained by answering two questions:

1. How to obtain the set of *faulty output neurons*, i.e., output neurons that are affected by this fault?
2. How to change the values of faulty output neurons to reflect the effects of the fault?

To answer the first question, we create a systematic approach called *Reuse Factor Analysis*, presented in Sec. 3.2.2. The answer to the second question follows the information provided by Reuse Factor Analysis, as discussed in Sec. 3.2.3.

The novelty of our approach is that, using just a few pieces of information – that can be obtained from design plans, block diagrams, architectural descriptions, and estimated values (that can be varied to perform sensitive analysis) – Reuse Factor Analysis can generate accurate software fault models, without the need to access RTL, or the presence of RTL at all.

Table 3.1: Summary of reuse factor analysis for datapath FFs in DL accelerators.

Faulty FF positions	Possible variable types	Properties	RF, and computation order / relative locations of faulty neurons
Before each level of on-chip memory	Input, weight, bias	A transient fault manifests as one incorrect value in memory	Specified by scheduling/reuse algorithm
Between L1 on-chip memory & MAC units, and inside MAC units	Input, weight, bias	Datapath RF properties (3), (4)	Obtained by using Algorithm 1
Inside and after MAC units	Partial sum, output	RF = 1	Specified by scheduling/reuse algorithm
After MAC units	Bias	Affect neurons that use the bias	Obtained by using Algorithm 1

3.2.2 Reuse Factor Analysis

Given a target FF in a DL accelerator design, our Reuse Factor Analysis technique provides information on: 1. the maximum number of faulty neurons that can be generated if this FF experiences a single cycle bit-flip, which is defined as the *reuse factor (RF)* of the FF; 2. the relative location(s) of all possible faulty neuron(s); as well as 3. the order in which these faulty neurons are calculated.

Reuse Factor Analysis for Datapath FFs

For the general accelerator architecture, the RF of a datapath FF depends on various design parameters such as the scheduling/reuse algorithm, memory organization, and connections from the FF to various compute units. To take these factors into account, we separate datapath FFs in the following partitions, where each partition consists of one or more pipeline stages: 1. before each level of the on-chip memories; 2. between the first-level (L1) on-chip memory and MAC units; 3. inside MAC units; and 4. after MAC units. We also categorize these FFs into different types of variables (inputs, weights, partial sums, and outputs[1]), and refer to them as input FFs, weight FFs, and so on. The pipeline stage and variable type together determine a datapath FF's *category*.

Based on our analysis, we derive the following unique properties in different datapath FF categories:

1. Output FFs store output values after accumulation is done, but can precede element-wise operations such as ReLU or bias addition.

Datapath RF Property (1): For datapath FFs before each level of the on-chip memories, a single-cycle bit-flip manifests as one incorrect value stored in the corresponding memory. Therefore, their RF values (as well as the relative locations and computation order of faulty neurons) can be determined by the scheduling/reuse algorithm since these on-chip memories are software-managed caches. For example, in NVDLA, there is one level of on-chip memory, and it stores both inputs and weights. The scheduling/reuse algorithm specifies that values stored on-chip should be reused for all MAC operations that involve these values. Therefore, any error occurring on the datapath preceding the on-chip memory can affect all output neurons that use the corresponding input/weight value in the current DNN layer.

Datapath RF Property (2): For output FFs inside and after the MAC units, since each FF corresponds to exactly one output neuron, the RF of all of these FFs equals to 1. Moreover, since the mapping from MAC units to output neurons are defined by the scheduling/reuse algorithm, the locations of the faulty neurons can be obtained as well.

Datapath RF Property (3): All datapath FFs in the same category (i.e., they have the same variable type and belong to the same pipeline stage) have the same RF. For example, in NVDLA, special NaN and zero values are signified using extra FFs in addition to the normal floating point values, and all of these FFs that belong to the same pipelines stage are reused in the same way to produce the same set of output neurons. Therefore, their RF values are the same. This means that we only need to perform Reuse Factor Analysis for each datapath FF category, which is a tractable problem.

Datapath RF Property (4): There are three independent datapath flows: weight FFs or input FFs or bias FFs→partial sum FFs→output FFs. Moreover, for each independent datapath flow, the RF of an FF in pipeline stage i must be greater than or equal to the RF of an FF in stage k, $\forall k > i$, because datapath FFs in later stages are driven by those in earlier stages.

As summarized in Table 3.1, given the above properties, the Reuse Factor Analysis

Algorithm 10: Reuse factor analysis

Definition: *compute units* are multipliers for input FFs and weight FFs, and accumulators/adders for partial sum FFs and bias FFs.

Inputs:

1. Variable type and pipeline stage of a target FF;
2. FF_value_cycles: the maximum number of cycles for which the target FF holds the same output value;
3. M_l, $\forall l = 0, ..., FF_value_cycles - 1$: the set of compute units that use the target FF's value for its computation at the target FF's lth loop (i.e., the lth cycle after the target FF last updates its output value); 4. $in_effect_cycles(m)$, $\forall m \in M_l$, and $\forall l = 0, ..., FF_value_cycles - 1$: the number of cycles during which a single-cycle value in the target FF is in effect (i.e., is used) by compute unit m;
5. $neurons(m)_{y,l}$, $\forall m \in M_l$, $\forall y = 0, ..., in_effect_cycles(m) - 1$, and $\forall l = 0, ..., FF_value_cycles - 1$: the set of relative (batch, height, width, channel) indices of the output neurons that are computed in the yth cycle by compute unit m since m starts to use target FF's value at the lth loop. The first neuron in $neurons(M_0[0])_{0,0}$ serves as the reference neuron.

```
1  FaultyNeurons = [];
2  for l ← 0 to FF_value_cycles − 1 do
3      for m ∈ M_l do
4          for cycle ← 0 to in_effect_cycles(m) − 1 do
5              for neuron ∈ neurons(m)_{cycle,l} do
6                  insert((neuron, l), FaultyNeurons);
7              end
8          end
9      end
10 end
11 RF = length(FaultyNeurons);
12 return RF, FaultyNeurons;
```

algorithm (Algorithm 1) focuses on input, weight, and bias FFs that are positioned after the L1 on-chip memory, while the RF values and corresponding faulty neuron information of other datapath FFs can be directly obtained from a DL accelerator's scheduling/reuse algorithm.

Algorithm 1 uses inputs 2-4 to account for how many compute units (line 3, capturing the spatial reuse aspect) and how many cycles of computation in each compute unit (line 4, capturing the temporal reuse aspect) can be affected by a single-cycle bit-flip in a target FF[2]. Our algorithm also considers the possibility that a faulty value may stay in the target

2. Without loss of generality, we assume that a compute unit is responsible for the computation of one output neuron per cycle, and can produce a new output per cycle. We also assume that partial sums are

FF for multiple cycles (line 2). As such, the relationship between the target FF and the compute units is established. Next, the relationship between each compute unit and each output neuron (specified in input 5) is taken into account to link the effects of the target FF to output neurons (lines 5-6).

The inputs of Algorithm 1 are both minimal and sufficient, and they can all be obtained from a high-level block diagram or microarchitectural description of the design, or from the scheduling/reuse algorithm. Specifically, in the hardware level, only the relationship between the target FF and the compute units is required. This is because, based on Datapath RF Property (4), an FF cannot drive another FF with a higher RF value. Thus, the compute units that have a connection to the target FF already include all the compute units that the target FF affects, which means that a detailed hardware model (e.g., one that specifies the connectivity between different datapath FFs) is not needed.

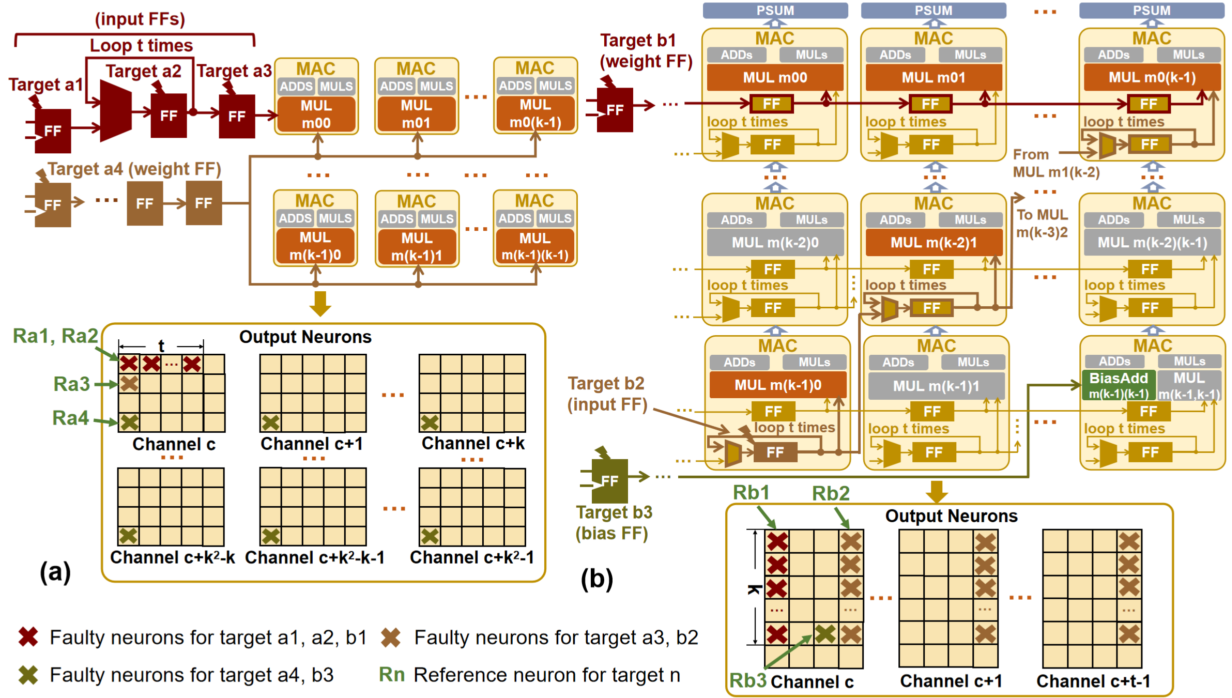
The algorithm returns the RF value of the target FF, the corresponding set of unique faulty output neurons, and the order that they are generated. The order is indicated by a time stamp (l in line 6 of Algorithm 1) attached to each faulty neuron. The faulty neurons are represented in relative indices, since the actual set of faulty output neurons (in absolute indices) depends on which cycle the fault is injected. To model a random fault injection cycle, we randomly choose one set of faulty neurons from all possible sets. Moreover, if the target FF holds its output for more than 1 cycle, we further randomly choose an integer p between 0 and $FF_value_cycles - 1$, and only consider the subset of $FaultyNeurons$ whose time stamp l is greater than or equal to p.

Datapath FF Reuse Factor Analysis Examples

Figure 3.1 demonstrates how Algorithm 1 is applied to various datapath FFs in DL accelerators. In all examples, the workload is a convolution layer and the computations are done

not reused by multiple output neurons. Algorithm 1 can be adapted easily without these assumptions.

for a single batch.



Input 1:	Target a1	Target a2	Target a3	Target a4	Target b1	Target b2	Target b3
Input 2:	1	t	1	1	1	t	1
Input 3:	$M_0=[m_{00}]$	$M_l=[m_{00}]$, $l \in [0, t)$	$M_0=[m_{00}]$	$M_0=[m_{00}, m_{01}, \dots, m_{(k-1)(k-1)}]$	$M_0=[m_{00}, m_{01}, \dots, m_{0(k-1)}]$	$M_l=[m_{(k-1)0}, m_{(k-2)1}, \dots, m_{0(k-1)}]$, $l \in [0, t)$	$M_0=[m_{(k-1)(k-1)}]$
Input 4:	t	1	1	$1, \forall\, m \in M_0$	$1, \forall\, m \in M_0$	$1, \forall\, m \in M_l, l \in [0, t)$	1
Input 5: Ref. position: [h, w, c]	neurons $(m_{00})_{y,0}=\{[h, w+y, c] \mid y \in [0,t)\}$	neurons $(m_{00})_{0,l}=\{[h, w+l, c] \mid l \in [0, t)\}$	neurons $(m_{00})_{0,0}=\{[h, w, c]\}$	neurons$(m_{ab})_{0,0}=\{[h, w, c+ak+b] \mid a \in [0, k), b \in [0, k)\}$	neurons$(m_{0a})_{0,0}=\{[h+a, w, c] \mid a \in [0,k)\}$	neurons$(m_{ab})_{0,l}=\{[h+b, w, c+l] \mid l \in [0, t), a+b=k-1\}$	neurons $(m_{(k-1)(k-1)})_{0,0}=\{[h, w, c]\}$
Output: RF	t	t	1	k^2	k	$k \times t$	1
Output: FaultyNeurons	[([h, w, c],0), ([h, w+1, c],0), ..., ([h, w+t-1, c],0)]	[([h, w, c],0), ([h, w+1,c],1), ..., ([h, w+t-1, c],t-1)]	[([h, w, c],0)]	[([h, w, c],0), ([h, w, c+1],0), ..., ([h, w, c+k²-1],0)]	[([h, w, c],0), ([h+1, w, c],0), ..., ([h+k-1, w, c],0)]	[([h, w, c],0), ([h+1, w, c],0), ..., ([h+k-1, w, c],0), ..., ([h+k-1, w, c+t-1],t-1)]	[([h, w, c],0)]

Figure 3.1: Datapath FF reuse factor analysis for (a) A NVDLA-like accelerator; (b) An Eyeriss-like accelerator.

In Fig. 3.1(a), we demonstrate the datapath of a NVDLA-like accelerator. There are k^2 parallel MAC units. The same inputs are sent to all MAC units, but the weights are different. In each MAC unit, the weights are reused for multiple operations, but a new input (shared by all MAC units) is fetched for each operation. The MAC units perform computations in the row-major order, and they compute the output neurons with the same (height, width) position in k^2 consecutive channels at the same time. We show four examples in Fig. 3.1(a):

targets a1-a3 are weight FFs, and target a4 is an input FF. The output of target a1 is only sent to one multiplier (m_{00}). However, downstream in the weight FFs→partial sum FFs→output FFs datapath flow, the max FF_value_cycles is t, i.e., the output of a1 is eventually sent to another FF (target a2 in this example) that keeps the same output for t cycles before it reaches multiplier m_{00}. Moreover, a2's $in_effect_cycles(m_{00})$ value is 1. Thus, a1's $in_effect_cycles(m_{00})$ is t, and a fault in target a1 affects t consecutive neurons in one output channel. Target a2 affects the same set of faulty neurons as a1. The difference is that a2's $FF_value_cycles = t$, while $in_effect_cycles(m_{00}) = 1$. Thus, a fault in a2 affects a random number of neurons between 1 to t. In the case of target a3, the only difference from target a2's case is that the RF is 1 instead of t because the faulty value only lasts for 1 cycle. For target a4, its value is sent to k^2 multipliers which produce k^2 output neurons each cycle, so its RF is k^2. All of a4's faulty neurons belong to the same 2D matrix position and span k^2 consecutive output channels.

Figure 3.1(b) shows the datapath of an Eyeriss-like accelerator, where target b1 is a weight FF, b2 is an input FF, and b3 is a bias FF. In this design, MAC units are arranged as a $k \times k$ systolic array. The MAC units belonging to the same column perform the MAC operations for one row of the output neurons in consecutive cycles, and consecutive MAC columns compute consecutive rows of the output neurons. Each MAC unit computes a row of partial sum values using one row of inputs and one row of weights, and the corresponding row of output neurons are obtained by accumulating the partial sum values of all MAC units in each column. In each cycle, a weight value is passed from one MAC unit to its neighbor in the next column so that it can be reused in the computations of different output rows. Therefore, the RF of b1 is k, and the set of faulty output neurons occupy k consecutive rows in the same column. Meanwhile, an input value is reused by the MAC units diagonally, and it is also reused inside each MAC unit to compute output neurons in consecutive channels and consecutive columns. Here we assume that, at the fault site of target b2, the FF stores

an input that is only needed for computing the last output column, so it is only reused across t output channels. Therefore, the RF of b2 is $k \times t$. The set of faulty output neurons for b2 are located in t consecutive channels. Within each channel, they also occupy k consecutive rows in the last column. In the case of target b3, since it is connected to only one compute unit (BiasAdd) and is not reused temporally, its RF is 1.

Reuse Factor Analysis for Control FFs

Control FFs in a DL accelerator can be broadly classified into two categories, *global* and *local*.

Global control FFs include those that store configuration information for the execution of an entire DNN layer (e.g., the size of inputs and weights, base addresses, and data types), and FFs that are responsible for sequencing data to/from on-chip memories (e.g., counters for advancing the address of a memory to read input/weight values). These FFs affect a large number of, if not all, output neurons. Moreover, one observation based on our own experiments is that, if the number of faulty neurons is large, it is very likely that an application output error or a system anomaly (e.g., time-out) would occur. For example, when $\sim 10\%$ of all output neurons in a layer are faulty, the probability that the final application output is correct is only $\sim 15\%$. Therefore, in FIdelity, we model a fault injected to an active global control FF as one that always results in application error or system anomaly.

Local control FFs, on the other hand, are closely coupled with certain datapath FFs. For example, the control FF that indicates whether the output of a multiplier is valid and ready to be passed to the next pipeline stage is only coupled with the output FFs of the corresponding multiplier. Therefore, the RF value and the set of faulty output neurons of the valid signal are the same as those derived for the output FFs in the same multiplier. Also, it is possible for a local control FF to affect multiple datapath FFs – for example, if the valid signal controls the outputs of multiple parallel multipliers. In this case, we take

the sum of the RF values and the union of $FaultyNeurons$ from all the datapath FFs it is coupled with. Another example of a local control FF is the select signal of a datapath multiplexer, which only affects the output of the multiplexer.

3.2.3 Deriving Faulty Output Neuron Values

After performing Reuse Factor Analysis on an accelerator design, our next task is to determine how to change the values of the faulty output neurons, and we consider datapath and local control FFs separately.

For datapath FFs, recall Accelerator Property (2) in Sec. 3.2.1, i.e., each datapath FF already corresponds to a software-visible state. In other words, for each datapath FF bit-flip, there exists an *equivalent* bit-flip in software, which can be used to calculate the values of the corresponding faulty neurons. For example, if the faulty FF corresponds to a bit of a weight value, then the whole set of faulty neurons are computed by using the faulty weight value.

For local control FFs, the effects of a fault on the corresponding datapath value(s) are not deterministic. Recall the valid signal example presented in the previous session, which is a local control FF that is used to indicate whether the output of a multiplier is valid or not. If a fault flips its value from 'valid' to 'not valid', effectively it means that the result generated by the multiplier at this cycle is dropped, so its value will be replaced by the next output generated by the same multiplier. However, if the value is flipped from 'not valid' to 'valid', then a non-deterministic intermediate value will be incorrectly interpreted as a valid output. As another example, if a multiplexer's select signal encounters a fault, then any input of the multiplexer (including inputs that are not currently driven) may be passed to the output, resulting in a non-deterministic output value. Therefore, for each neuron in the $FaultyNeurons$ set of a local control FF, we replace its value with a random value.

3.2.4 FIdelity's Fault Injection Flow

The accurate software fault models, derived based on our Reuse Factor Analysis algorithm, is the key component in the FIdelity framework. Using the software fault models, FIdelity's entire fault injection process is shown in Fig. 3.2.

The inputs of FIdelity, summarized in Fig. 3.2, include: 1. a DNN workload (e.g., its layer type, kernel size, etc.); 2. a raw FF FIT rate, which is the probability that a transient error occurs in one FF, and its value depends on the technology node of the design; 3. the scheduling/reuse algorithm and hardware configuration parameters of the accelerator design, which are available from architectural descriptions; and 4. high-level microarchitecture information. The required microarchitecture information can be estimated from design plans/sketches, block diagrams, or previous design generations, and the estimated values can be varied for sensitivity analysis to obtain resilience bounds. As the design process evolves and more detailed hardware information becomes available, FIdelity's inputs are more accurate, which in turn improves its accuracy.

The output of FIdelity is the accelerator FIT (failure in time) rate, a standard resilience metric, which denotes the number of system failures in 1 billion device hours, where system failure in the context of DL accelerators is either DNN application output error or system anomaly (e.g., time-out).

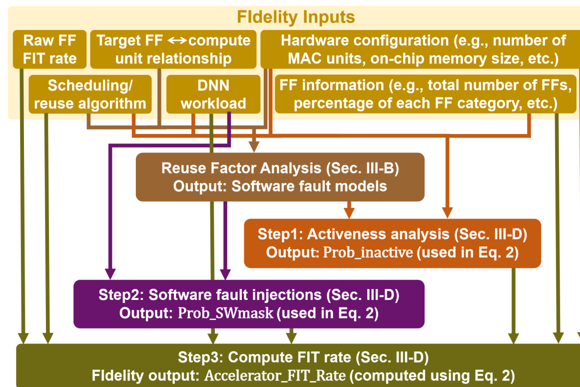


Figure 3.2: Overview of the FIdelity framework.

The fault injection flow of FIdelity consists of three steps. In **step 1**, we perform FF activeness analysis to account for error masking scenarios, since a fault injected to an inactive FF will always be masked.

The probability that an FF is inactive can be estimated by simply assuming a reasonable range. Given a few pieces of high-level microarchitecture information, a more detailed analysis can be performed by considering three classes of inactive FFs during the execution of a DNN layer:

Class 1. *Component not used*: an FF belongs to a hardware component that remains idle for the entire execution of the workload. For example, if the weights are not compressed, then all FFs in the decompression unit are idle.

Class 2. *Signal not used*: an FF belongs to an active hardware component, but the FF itself stays inactive the entire time. For example, FFs responsible for floating point calculations remain inactive if the current workload uses an integer representation.

Class 3. *Temporally not used*: a hardware component – and therefore, all FFs in this component – is inactive for a portion of the time. For example, the MAC units are inactive when they are waiting for data to be fetched from memory.

These classes are mutually exclusive and complete. The first two classes are determined by the DNN layer's characteristics (e.g., whether it uses integer or floating point representation, or whether its weight values are compressed). For the "temporally not used" class, the percentage of time for which a hardware component is inactive can be estimated using high-level architectural information. For example, in NVDLA, a performance tool, which uses information solely obtained from the scheduling/reuse algorithm and hardware configuration parameters such as the number of MAC units, is available[37]. Given a workload (a DNN layer), the tool breaks down the time required to fetch data and to perform MAC/linear/non-linear/etc. operations. This breakdown indicates how long an FF positioned before CBUF (NVDLA's on-chip memory) is inactive, or how long an FF inside a MAC unit is inactive.

Note that, it is possible for an FF to be inactive when the corresponding hardware component is active. However, based on our study, we found that this case constitutes a very small fraction of all cases, so it is not included in our analysis.

Given a DNN layer r, we define $\text{Perc_inactive}(cat, cl, r)$ for FFs that are mapped to software fault model cat and inactive class cl, which equals 1 if $cl \in Class\ 1, 2$, and equals to the percentage of inactive time of the corresponding component if $cl \in Class\ 3$. Let $\text{FF_Perc}(cat, cl)$ represent the percentage of FFs in cl out of all FFs in cat. Then, the average probability that an FF belonging to cat is inactive during the execution of r, denoted as $\text{Prob_inactive}(cat, r)$, is calculated as shown in Eq. 3.1.

$$
\begin{aligned}
&\text{Prob_inactive}(cat, r) = \\
&\sum_{cl} \text{FF_Perc}(cat, cl) \times \text{Perc_inactive}(cat, cl, r)
\end{aligned}
\tag{3.1}
$$

In **step 2**, given the accurate software fault models based on our Reuse Factor Analysis, we perform large-scale software fault injection experiments for a statistically significant number of samples using each software fault model, and record the outcome of each experiment run.

For DNN applications, the outcome of logic transient errors can be classified into two categories: 1. *masked*, which means that the final output generated in the presence of a fault is sufficiently similar to the golden output, such that the effects of the fault can be considered negligible; and 2. *system failure*, which captures all non-masked cases, including application output error and system anomaly. The outputs of step2 are the probabilities of masked cases for all software fault models in each layer of a given DNN application ($Prob_SWmask(cat, r)\ \forall r$). Note that, $Prob_SWmask(\text{global control FFs}, r) = 0\ \forall r$, since this is how FIdelity models faults in active global control FFs.

In **step 3**, we calculate $Accelerator_FIT_rate$ using Eq. 3.2. Let FIT_raw be the raw FF FIT rate, N_{ff} be the number of FFs in the accelerator, $FF_Perc(cat)$ be the percentage of FFs whose faulty behavior is modeled by software fault model cat, and $exec_time(r)$ be

Table 3.2: NVDLA software fault models for convolution (Conv), fully Connected (FC), & matrix multiplication (MatMul) layers.

Datapath positions / Control type	Variables	Num. of FFs / % FFs	RF	Faulty neuron information	Software fault model
Before CBUF	Input	54569/ 8.2%	Total number of neurons using the target FF value	**Conv:** All neurons that use the input value are faulty. The locations depend on layer parameters such as stride, dilution, and the size of the kernels. **FC:** All neurons are faulty. **MatMul:** All neurons in the output rows that this input value maps to are faulty.	One random bit-flip at one randomly chosen input, affecting all neurons that use the input value.
	Weight	63949/ 9.6%		**Conv:** All neurons in the one output channel that this weight value maps to are faulty. **FC:** One neuron in each batch uses the weight value and thus is faulty. **MatMul:** All neurons in the output columns that this weight value maps to are faulty.	One random bit-flip at one randomly chosen weight, affecting all neurons that use the weight value.
Between CBUF & MAC units, and inside MAC units	Input	83084/ 12.5%	16	**Conv:** 16 neurons in the same 2D matrix position and spanning 16 consecutive output channels use the same faulty value. See target a4 in Fig. 3.1(a) as an example. **FC:** 16 consecutive output neurons use the same faulty value. **MatMul:** 16 consecutive neurons in the output rows that this input value maps to are faulty.	One random bit-flip at one randomly chosen input, affecting the corresponding 16 faulty neurons.
	Weight	68217/ 10.3%	16	**Conv:** All or a subset of 16 consecutive neurons (in row major order) belonging to the same output channel use the same faulty value. See target a1/a2 in Fig. 3.1(a) as an example. **FC:** One out of 16 output neurons are faulty, for a total number of <= 16 faulty neurons. **MatMul:** All or a subset of the 16 consecutive neurons in the output columns that this weight value maps to are faulty.	One random bit-flip at one randomly chosen weight, affecting the corresponding <= 16 neurons.
Inside and after MAC units	Output, partial sum	320487/ 48.2%	1	The one neuron that uses the target FF value is faulty.	One random bit-flip at one randomly chosen output neuron or partial sum.
Local control	N/A	26738/ 4.1%	Same RF as that of the datapath FF that this FF controls	Same faulty neurons as those of the datapath FF that this FF controls	Random faulty value at one randomly chosen output neuron.
Global control	N/A	48478/ 7.3%	ALL	A large number of output neurons are faulty.	System failure.

the execution time of layer r in a given DNN application, which can be estimated or obtained based on high-level architectural information. Accelerator_FIT_Rate is the product of the probability that a fault occurs in an FF (FIT_raw), and the probability of an application error or a system anomaly, given that a fault has occurred in a single FF:

$$\begin{aligned} &\text{Accelerator_FIT_rate} = FIT_raw \times N_{ff} \times \sum_{r} [\ exec_time(r) \\ &\times \sum_{cat} \text{FF_Perc}(cat) \times (1 - \text{Prob_inactive}(cat, r)) \\ &\times (1 - \text{Prob_SWmask}(cat, r))\ ] / \sum_{r} exec_time(r) \end{aligned} \tag{3.2}$$

3.2.5 Broader applicability

Although we focus on logic transient errors, FIdelity can be used to model memory errors as well, based on Datapath RF property (1) in Sec.3.2.2. Reuse factor analysis for errors occurring in one memory word is the same as that for the datapath FFs that serve as input data to the memory (Table 3.1, row2). For multiple memory errors, the set of faulty neurons are the union of the faulty output neurons of each error. After the memory software fault models are established, the fault injection flow can be carried out exactly as shown in Fig. 3.2.

Moreover, even though FIdelity is developed to model single-cycle single-FF bit-flips, it can be extended to cover multiple bit-flips in multiple FFs across multiple cycles.

- FIdelity can be directly applied to situations where multiple bit-flips occur in different DNN layers, and there is at most one bit-flip per layer. This is because, according to Accelerator Property (1), a hardware fault in one layer can only affect the subsequent layer through the layer's output (details see 3.2.1). In each layer where a bit-flip occurs, FIdelity models the faulty neuron positions and values correctly, while propagating the faulty effects from previous layers to the input of the current layer. The current layer then propagates the combined effects of all bit-flips that have occurred to subsequent layers.

- For cases where multiple bit-flips occur in one layer, FIdelity can also be applied if the faulty effects for each bit-flip are independent. The combined faulty effects are the union of the faulty effects of each bit-flip. For example, one bit-flip might affect 16 neurons computed in one batch, while another bit-flip affects one neuron located at a different position within the same batch. In this case, the combined faulty effects are modeled based on 17 faulty neurons, with 16 being modeled based on the first bit-flip and 1 being modeled based on the second bit-flip. Another example is that one bit-flip might affect the input feature map datapath, while another bit-flip affects

the weight datapath. In this case, the combined faulty effects can be modeled as faulty positions and values in the corresponding software variables (i.e., input feature map and weights), and then the error effects can be propagated in software by simply performing the layer operation.

- For cases where the faulty effects of multiple bit-flips are dependent, more detailed hardware information is required to model the combined faulty effects correctly. For example, with the following two pieces of information: (1) which pipeline stages do the faulty FFs belong to, and (2) how the FFs in each pipeline stage are connected to the ones in subsequent pipeline stages, we could first use FIdelity to model the bit-flip occurring in the earliest pipeline stage. Then, we can propagate its faulty effects down to subsequent pipeline stages and use FIdelity to model bit-flips in the subsequent stages accordingly.

Table 3.3: Workloads (DNN Layers) Used for FIdelity Validation. Precision: FP16.

<table>
<tr><th>Networks</th><th>Layer</th><th>Dataset</th></tr>
<tr><td>Inception</td><td>A 3×3 Conv layer
in inception module.</td><td rowspan="2">Imagenet</td></tr>
<tr><td>Resnet50</td><td>A 3×3 Conv layer
in residual block.</td></tr>
<tr><td rowspan="2">Transformer</td><td>A FC layer in
feed-forward network.</td><td rowspan="2">IWSLT2014</td></tr>
<tr><td>A MatMul layer in attention.</td></tr>
<tr><td>RNN</td><td>A FC layer in LSTM.</td><td>UCI HAR</td></tr>
<tr><td>Yolo</td><td>A 3×3 Conv layer
in residual block.</td><td>COCO</td></tr>
</table>

3.3 FIdelity Framework Validation

We use NVDLA as a case study to demonstrate that FIdelity's software fault models are accurate. Our methodology is to first perform RTL injection to obtain golden references on the number of faulty neurons, their relative positions, as well as the order in which they are generated, and then compare the software fault models generated using Reuse Factor Analysis with the information obtained from RTL simulations.

3.3.1 Accurate Software Fault Models for NVDLA

We apply the Reuse Factor Analysis to NVDLA. The design of NVDLA is similar to our example in Fig. 3.1(a), which is configured with $k = 4$ and $t = 16$ in our case study. Its software fault models are shown in Table 3.2 for three types of representative DNN layers: convolution, fully-connected, and matrix multiplication.

3.3.2 Validation Methodology Details

RTL fault injection experiments are performed using Synopsys VCS for various representative workloads shown in Table 3.3. After a fault site is selected for a given workload, we perform RTL simulation until all output neurons of the current layer are generated, or until the simulation reaches a system time-out value. By comparing the output neuron results against the fault-free results, we obtain the set of faulty neurons and their faulty values. We ran experiments with 10K fault sites for each workload to achieve 95% confidence interval. Moreover, in order to validate FIdelity's results for global control FFs, for all experiments where faults are injected into global control FFs, we further perform mixed-mode simulations to check the correctness of the final DNN application outputs (correctness metrics are discussed in Sec. 3.4 and specified in Table 3.4).

Out of the 60K total RTL fault injection experiments, 9956 generate errors in the output

neurons, and 72 lead to system time-out (the time-out cases are all due to faults in global control FFs). For all of the 10028 cases where the injected faults are not masked, we use the same fault sites to derive the corresponding software fault models from Table 3.2. For datapath FFs and local control FFs, we perform software fault injection using FIdelity's software fault models in TensorFlow (which we have modified to support all software fault models). Each software fault injection experiment generates a set of faulty neurons and their faulty values, which are manually compared with those obtained form RTL simulations.

3.3.3 Results and Discussions

For each of the datapath FF cases (8262 total), our software fault model generates *exactly* the same set of faulty neurons and faulty values as the RTL result. Our model for global control FFs, i.e., faulty global control FFs always result in system failures, is also accurate. The RTL+TensorFlow mix-mode simulation results suggest that only ∼9.5% of faults injected to global control FFs are masked.

For the local control FF cases (138 total), RTL simulation results confirm that indeed only one output neuron is faulty (RF=1) in each case. Moreover, our analysis derives the same faulty neuron as the RTL simulation result in each case. In terms of the faulty values, although our results differ from RTL simulation results, we expect that, with a sample set that is sufficiently statistically-significant, the results derived using the FIdelity framework will be similar to RTL simulation results. This is because, as discussed in Sec. 12, the behavior of a fault in a local control FF is non-deterministic and its effects can be approximated using a random faulty value.

In summary, the accuracy of the software fault models generated by the Reuse Factor Analyses algorithm in our FIdelity framework is thoroughly validated.

3.4 Large-Scale Resilience Study

We apply the validated FIdelity framework to NVDLA, and perform large-scale software fault injection experiments in TensorFlow (using software fault models presented in Table 3.2) to obtain resilience results for four typical CNNs (Inception, Resnet, MobileNet, Yolo) and the Transformer network. The details of the experiments are shown in Table 3.4. Note that, the FP16 networks are taken from public resources, and we trained the INT16 and INT8 models with TensorFlow's support for quantization.

The correctness metric of a DNN application, which determines whether the final output is correct or not, is an important consideration. For Inception, Resnet, MobileNet, which are CNNs used for image classification tasks, we compare the top1 label of each fault injection experiment with the top1 label obtained from the fault-free execution, and the application output is considered correct if they match. For the Transformer and Yolo networks, each application output is given a quality score, and an output can be considered correct even if its score does not match the fault-free score exactly. Here we apply two metrics similar to previous work [17]: an application output of the Yolo or Transformer network is considered correct if the difference of its score is within 10% or 20% of the score obtained from the fault-free execution.

Our resilience analysis results are shown in Figs. 3.3 and 3.4, breaking down the Accelerator_FIT_rate contributions from datapath, local, and global FFs. The raw FF FIT rate used to derive our results is 600/MB for soft errors [44]. Other raw FF FIT rates (e.g., for voltage variations, or soft errors in a different technology node) can be used, and the general conclusions of our results remain the same.

We present five key results that are revealed by our study below.

Key result (1): There is a need for resilience protection in the logic portion of DL accelerators. One objective of this study is to find out if NVDLA's resilience level can meet the ASIL-D level of the ISO26262 standard (i.e., the highest level of automotive

Table 3.4: FIdelity's Fault Injection Experiment Setup.

Platform: Tensorflow (modified to support FIdelity's fault injection flow)			
Total number of Experiments: 46M			
DNN Workload	Inception, Resnet50, MobileNet	Transformer	Yolo
Dataset	Imagenet, Cifar10	IWSLT14	COCO
Correctness Metric	Top 1 label match	< 10%/20% BLEU score difference	< 10%/20% precision difference
Data Precision	FP16, INT16, INT8	FP16	FP16

safety standard) [45], if the FFs in NVDLA are left unprotected from transient errors.

According to the safety requirement, the overall FIT rate for an entire self-driving car chipset must be <10. We follow the standard approach [17] that assigns a fraction of the FIT rate requirement to each individual hardware component based on the area of the component, and estimate that the FIT rate requirement for all FFs in NVDLA must be <0.2 since the FFs in NVDLA occupies $\sim$ 2% of the total chipset area[46].

However, our results in Figs. 3.3 and 3.4 show that, without resilience protection, NVDLA's Accelerator_FIT_rate values are significantly higher than 0.2. For example, for the Yolo network, which is widely used for object detection tasks for self-driving cars, the FIT rate is 9.5 when the 10% precision difference is used as the correctness metric.

In general, various DNN applications are subject to various resilience requirements. Safety-critical applications impose high resilience standards as shown in the self-driving car example above. Moreover, even for non-safety-critical applications, the rate of hardware errors can be quite high under certain operating conditions and environments, which will result in significant degradation in inference accuracy. Thus, resilience analysis and resilience pro-

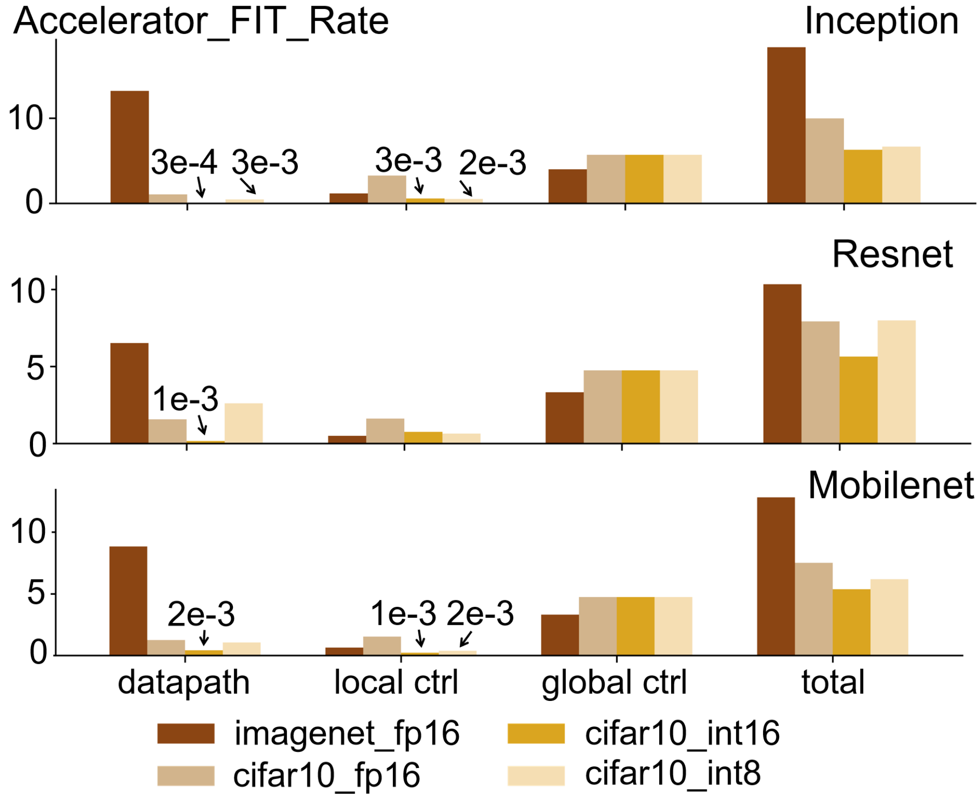
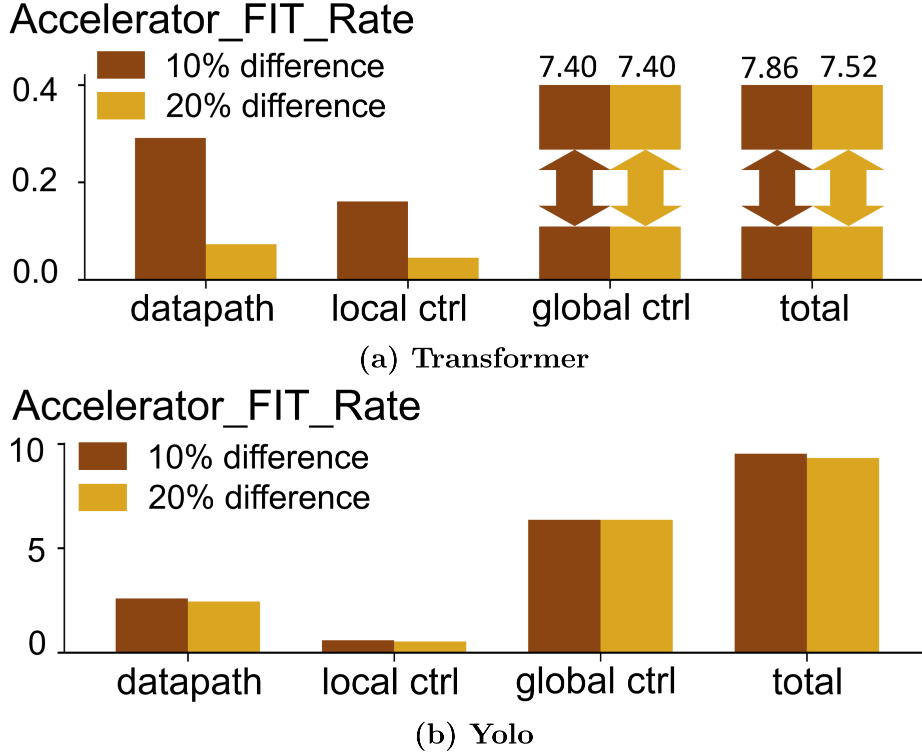


Figure 3.3: Accelerator_FIT_Rate Values for Inception, Resnet, and Mobilenet.

tection techniques are essential to ensure that the resilience target of any given application is met under all conditions.

Key result (2): There is a need for resilience analysis and protection for datapath and local control FFs in DL accelerators. Our results show that global control FFs contribute to the largest portion of the Accelerator_FIT_rate values. If all global control FFs are protected, is there still a need to perform resilience analysis and provide resilient protection for the other FFs?

The answer is yes. In Fig. 3.5, we show the Accelerator_FIT_rate values for three CNN applications assuming that the raw FIT rate of all global control FFs is 0. We can see that the FIT rates are still larger than 0.2, the largest FIT rate allowed by the automotive safety standard. Therefore, frameworks such as FIdelity is crucial to analyze the resilience properties of datapath and local control FFs in DL accelerators.



(a) **Transformer**

(b) **Yolo**

Figure 3.4: Accelerator_FIT_Rate Values for Transformer & Yolo.

Key result (3): The correctness metric can impose a large impact on Accelerator_FIT_Rate. For example, in Fig. 3.4, the FIT rates for datapath and local control FFs are very different when different correctness metrics (10% vs. 20% BLEU score difference) are used. Therefore, application characteristics and requirements must be taken into consideration when resilience analysis is performed.

Key result (4): Data precision also influences Accelerator_FIT_rate. From Fig. 3.3, we can see that the FP16 networks result in higher Accelerator_FIT_rate values compared to their INT16 and INT8 counterparts in most cases. One possible reason is that the actual dynamic range of FP16 is much larger than INT16 and INT8, which allows bigger perturbations in the presence of errors, and thus results in a higher probability of application output errors according to Key result (5). We can also see that the FIT rates of the INT8 networks are in general higher than those of the INT16 networks. One hypothesis is that, due to the precision loss, if INT8 and INT16 networks are quantized with similar min and

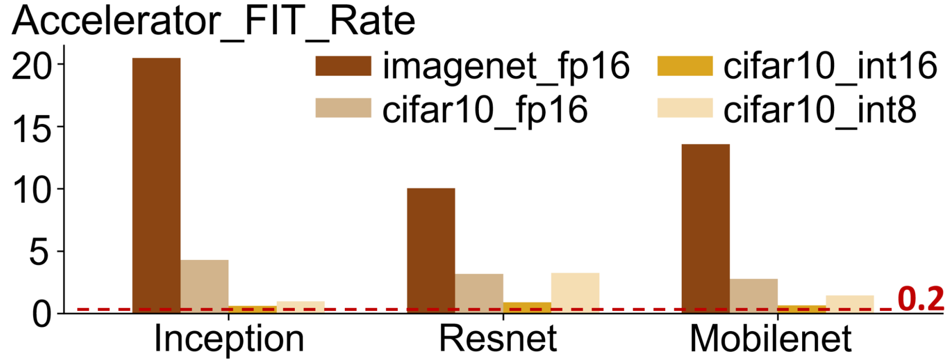


Figure 3.5: Accelerator_FIT_Rate Values Assuming that Global Control FFs are Protected.

max values, then the same absolute perturbation that a fault brings to a neuron will result in a bigger effect in INT8 networks than INT16 networks, resulting in a higher number of Top 1 label mismatches.

Key result (5): Large perturbations in the output neurons are more likely to cause application output errors than smaller perturbations. We separate a subset of the fault injection experiments, the ones that result in one faulty output neuron from the FP16 Inception, Resnet, and Mobilenet networks, into two cases: 1. the difference between the faulty and fault-free values is $\leq$100; and 2. the difference is $>$100. In the former case, there is only a $< 4\%$ probability that a fault results in an incorrect application output; while in the latter, this probability is $> 45\%$. These results suggest that large perturbations in the values of faulty output neurons are more likely to cause application output errors.

Architectural Insights: The key results revealed by FIdelity provide architectural insights to guide resilience designs, and can inspire new resilience techniques. As an example, if errors in certain FF categories contribute more to Accelerator_FIT_rate than others, the design may be optimized by minimizing the number of FFs in these categories. Alternatively, selectively protecting only the FFs in these categories may be sufficient to achieve a given resilience target while minimizing system-level costs. As these resilience-critical FF categories are workload dependent, the selective protection scheme can be implemented in an adaptive

manner. As another example, hardware-software co-design techniques can be explored, by bounding the values of output neurons based on Key result (5), or by experimenting with different data precision based on Key result (4), to achieve higher resilience.

3.5 Related Work

Since resilience is a top priority in DL accelerators, various resilience studies targeting these accelerators exist. However, they mainly focus on memory errors, e.g., soft errors that occur in SRAMs [17], or errors caused by reduced memory voltage [41, 42, 43]. In previous work, transient errors in datapath FFs are modeled as single bit-flips in a single architectural state [17], which is highly inaccurate. Furthermore, previous work either assumes that control FFs have little resilience impact [17], or that a faulty control FF always causes visible system anomaly [47], which is also highly inaccurate. For example, in NVDLA, 34% of the control FFs (5.7% overall) can only affect at most one output neuron, and their resilience impact is usually small, while the rest of the control FFs have global effects (see Table 3.2). To the best of our knowledge, this is the first detailed resilience study on logic transient errors in DL accelerators.

Large-scale statistical fault injection is a well-known approach for analyzing logic transient errors [10, 11, 48, 49]. However, techniques that produce accurate results require access to a RTL model to perform detailed fault injection experiments, which takes a prohibitively long time. We quantitatively compare RTL simulation time, mixed-mode simulation time, and the time required to perform FIdelity's software fault injection experiments, for all workloads in Table 3.3. For NVDLA, FIdelity achieves >10000X and 40X-2200X speedup vs. RTL and mixed-mode simulations, respectively, while achieving similar accuracy as validated in Sec. 3.3[3].

3. It is infeasible to directly compare Accelerator_FIT_Rate calculated using FIdelity vs. RTL or mixed-mode simulations because it will take $> 100M$ CPU hours to perform RTL simulation to reach statistically significant results.

On the other hand, software fault injection techniques are quick (similar to FIdelity), but their results are highly inaccurate. Again using the workloads in Table 3.3, we compare the accurate Accelerator_FIT_Rate values obtained from FIdelity with those obtained from a naive software fault injection technique where single bit-flips are injected to the architectural states. We found that the naive technique underestimates NVDLA's Accelerator_FIT_Rate by up to 25X across different workloads. Such inaccurate results are misleading and can impose significant safety/reliability risks.

Various techniques have also been proposed to optimize the fault injection process. However, many of these techniques require access to RTL [14, 15]. Moreover, these techniques are specifically designed for CPUs, but the architectural properties of DL accelerators are fundamentally different from CPUs. For example, both MeRLiN [15] and Relyzer [16] aim to reduce the number of fault sites by identifying CPU instructions that generate equivalent/similar faults, so they are not applicable for DL accelerators. Raven [14] obtains the error probability of each hardware block through RTL simulations, which takes less time than simulating an entire design, and then it combines these probabilities to obtain the final FIT rate. Raven also cannot be applied to DL accelerators because an error that occurs in a hardware block may not lead to a DNN application output error or system anomaly. Thus, FIT rate estimations obtained using the Raven approach are inaccurate (pessimistic).

AVF (Architectural Vulnerability Factor) can be used to analyze the effects of transient errors in CPUs without performing fault injection experiments. However, it relies heavily on the architecture and microarchitecture of microprocessors. Thus, it cannot be applied to DL accelerators.

CHAPTER 4

UNDERSTANDING AND MITIGATING HARDWARE FAILURES IN DEEP LEARNING TRAINING SYSTEMS

The content of this chapter will be published in our paper titled "Understanding and Mitigating Hardware Failures in Deep Learning Training Accelerator Systems" in ISCA23.

Deep neural network (DNN) training workloads are increasingly susceptible to hardware failures in datacenters. For example, Google experienced "mysterious, difficult to identify problems" in their TPU training systems due to hardware failures [1]. Although these particular problems were subsequently corrected through significant efforts, they have raised the urgency of addressing the growing challenges emerging from hardware failures impacting many DNN training workloads.

In this paper, we present the first in-depth resilience study targeting DNN training workloads and hardware failures that occur in the logic portion of deep learning (DL) accelerator systems. We developed a fault injection framework to accurately simulate the effects of various hardware failures based on the design of an industrial DL accelerator, and conducted $> 2.9M$ experiments ($> 490K$ node-hours) using representative workloads. Based on our experiments, we present (1) a comprehensive characterization of hardware failure effects, (2) the fundamental understanding on how hardware failures propagate in training devices and interact with training workloads, and (3) the necessary conditions that must be satisfied for these failures to eventually cause unexpected training outcomes.

The insights obtained from our study enabled us to develop ultra-light-weight software techniques to mitigate hardware failures. Our techniques require 24-32 lines of code change, and introduce $0.003\% - 0.025\%$ performance overhead for various representative workloads. Our observations and techniques are generally applicable to mitigate various hardware fail-

ures in DL training accelerator systems.

4.1 Introduction

Hardware failures are a growing challenge in datacenters, as evidenced by the increasing number of hardware failures that have recently been reported by Google, Facebook, and more [1, 2, 3, 4, 5, 6]. The hardware failure rate is high – a few cores per several thousand server machines [2, 3]. Moreover, a wide variety of hardware failures have been reported, including transient failures such as soft errors and dynamic variations, and permanent failures such as early life failures, manufacturing defects that escape testing, and circuit aging/degradation [2, 3, 4, 5, 6].

As deep neural network (DNN) training workloads are becoming more and more prevalent in datacenters [50, 51, 52], they are increasingly susceptible to hardware failures. For example, through significant efforts, Google recognized and corrected multiple instances of hardware failures during the execution of DNN training workloads on TPUs, with a failure rate similar to previously reported numbers [2, 3]. These hardware failures resulted in not only easy-to-detect unexpected outcomes such as NaN values that corrupted the training process, but also "mysterious, difficult to identify problems" [1]. Due to the widespread use of ECCs (Error Correction Codes) in both on-chip and off-chip memories, these hardware failures predominantly occurred in the logic portion of these TPU systems. They mostly exhibited transient effects – some could not be reproduced at all, while others could only be reproduced intermittently (e.g., when running the same workload 10 times on a faulty machine, the unexpected outcome was only observed 3 times), and were root-caused to manufacturing defects, circuit degradation, voltage variations, environmental conditions, and soft errors, among others.

We have learned several lessons from these reported experiences. First, as many logic hardware failures that pose real threats have been found in datacenter systems, they are

not rare and cannot be ignored. Second, contrary to the common belief that hardware failures (especially those that exhibit transient effects) can largely be tolerated by training algorithms, we now have the evidence that suggests otherwise. Third, when an unexpected training outcome occurs, it is critical to determine if the issue is caused by hardware failures or software bugs. Otherwise, significant software engineering efforts would be wasted on debugging a problem engineers incorrectly perceive to be in their software systems. Last but not least, although there is rich resilience literature, in practice, no solution exists to efficiently handle unexpected DNN training outcomes caused by hardware failures (see Sec. 4.6) – these issues have been termed "bugs from hell" [1], and the industry has issued urgent call-to-action to address them [1, 2, 3].

All of these lessons point to one important realization: there is an urgent and crucial need to devise efficient and effective hardware failure mitigation techniques for DNN training workloads. In order to create new solutions, the critical first step is to thoroughly understand the impacts of logic hardware failures on DNN training workloads. However, there is no such prior study in the literature.

To bridge these important knowledge gaps, we present the first study on hardware failures in DNN training systems. We focus on logic hardware failures that exhibit transient effects, which predominantly occur in datacenters today [1] – in the rest of the paper, *hardware failure* is used to refer to this class of failures unless specified otherwise. Moreover, we focus on deep learning (DL) training accelerator systems, since they are widely used and are currently undergoing rapid growth [50, 53]. Through in-depth analysis, we now have a comprehensive characterization of the hardware failure effects. We also fundamentally understand how hardware failures propagate, as well as the necessary conditions for these failures to eventually cause unexpected outcomes. These new insights enabled us to develop efficient hardware failure mitigation solutions that are readily deployable in practice.

The major contributions of this paper are:

(1) We present the first in-depth study on hardware failures in DNN training accelerator systems, which is enabled by a new fault injection framework that accurately models the behaviors of hardware failures. Using this framework (open-sourced [54]), we performed $> 2.9M$ fault injection (FI) experiments ($> 490K$ node-hours) in a distributed DNN training environment.

(2) Based on the experiment results, we present a complete characterization of the failure behaviors. In addition to known effects (e.g., a failure generates INFs/NaNs [1, 6]), we identified four new, intricate outcomes (Sec. 4.4.1), where failures resulted in abnormal convergence trends that persist for a long time (thousands of training iterations or more), without visible anomalies. Instances of one of the new outcomes (SlowDegrade, see Sec. 4.4.1) were later observed (and corrected) in DL training accelerator systems in datacenters.

(3) Deeper analysis led to a finding that large absolute gradient history values in optimizers, or large absolute moving variance values in normalization layers, are the necessary conditions for hardware failures to generate the new unexpected training outcomes revealed by our experiments. Moreover, these conditions always occur within two training iterations after hardware failures occur.

(4) Based on the necessary conditions, we devised (a) a new hardware failure detection technique that checks the absolute gradient history values and the absolute moving variance values against their respective bounds, where the bounds can be mathematically derived based on the properties of a given DNN training workload, coupled with (b) light-weight re-execution of the two most recent training iterations, which is sufficient to recover the training workload from hardware failures. Evaluation on Google Cloud TPUs shows that our detection and re-execution techniques together require $24 - 32$ lines of code change and introduce $0.003\% - 0.025\%$ performance impact for various DNN training workloads.

This paper is organized as follows. Background information is provided in Sec. 4.2. Our

fault injection experiments and results are presented in Sec. 4.3 and Sec. 4.4. We present new mitigation techniques in Sec. 4.5, and discuss related work in Sec.4.6.

4.2 Background

DNN training workloads are typically executed in a distributed manner using many training devices. For example, in synchronous distributed training [55], every device stores a separate copy of a given DNN model, and uses one mini-batch of the training data-set to compute a training loss through a forward pass, followed by a backward pass where the gradients of the trainable parameters (e.g., weights, biases, etc.) are computed with respect to a loss function using an optimizer. After each iteration, the weight gradients generated by all training devices are averaged (e.g., by a central server). The average gradients are then propagated back to all training devices to start the next iteration.

Hardware failures can pose various effects on DNN training workloads (some examples are shown in Fig. 4.1). Although it might appear that DNN training is resilient to hardware failures, industry reports have already shown that these failures are detrimental to training and not rare, as discussed in Sec. 4.1.

Some failures may be masked by hardware logic, e.g., if faulty values are AND'ed with 0's. They may also be masked by various operations performed during the training process, e.g., if a faulty value is multiplied by a 0, or is set to 0 by the activation function. Without the above masking effects, still the final training outcome (training/test accuracy and training time) may not be affected significantly because the training process may be able to recover from the effects of hardware failures. This presents an opportunity: if we can pinpoint the hardware failures that are likely to cause unexpected outcomes, we can devise optimized mitigation solutions.

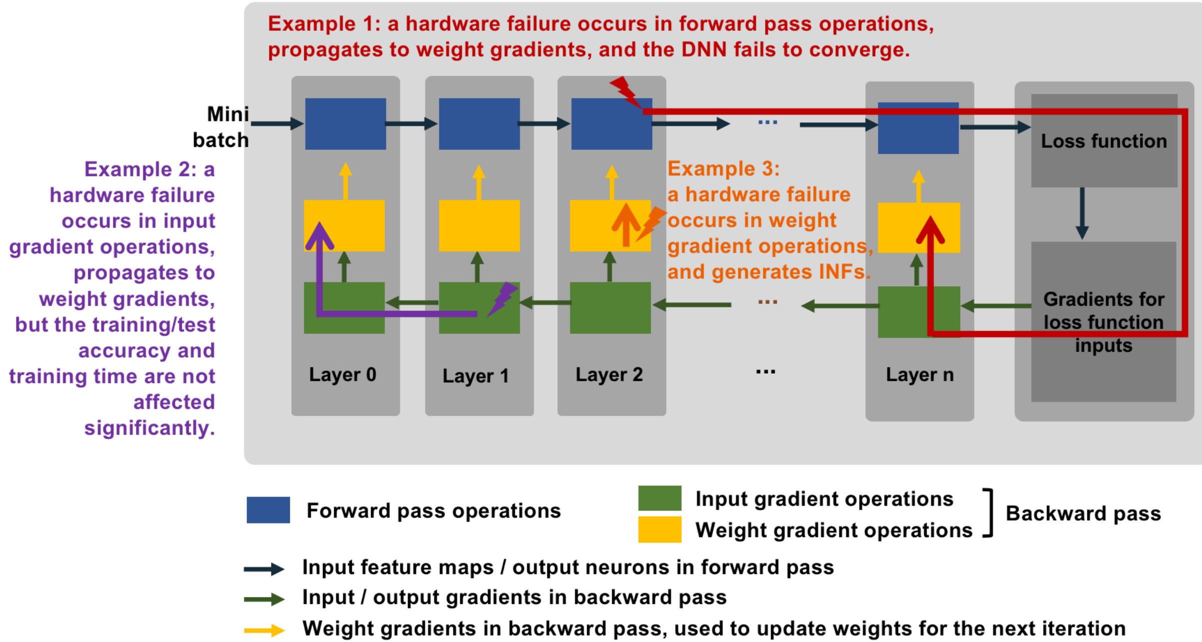


Figure 4.1: Hardware failure examples in DNN training.

4.3 Methodology and Framework

To study the resilience of DL training accelerator systems, we performed statistical fault injection (FI) experiments, the most widely-used approach for analyzing hardware failure behaviors [10, 11, 12, 13, 14, 15, 16, 17].

Existing FI methods suffer from the following limitations. Fast FI is typically achieved by injecting faults in software [17, 18, 19]. However, both previous works and Chapter 3 have found that the accuracy of software FI is low [10, 56]. To achieve high accuracy, RTL-level FI needs to be performed; however, RTL simulation time is prohibitively long, especially for the already time-consuming DNN training workloads. For example, using Resnet18 with the Cifar10 data-set as the DNN training workload, it would take 46K years with 8 threads to perform 1M RTL FI experiments to achieve high statistical significance. Our methodology and framework, discussed in this section, explain how we overcame these challenges.

4.3.1 Accelerator Architecture for DNN Training

Detailed hardware information (e.g., RTL) is required to obtain accurate FI results. Although there are no DL training accelerators with open-source access, an inference accelerator can be adopted for training because the designs of DNN training and inference accelerators are similar. For example, TPU v4 (training) and v4i (inference) share the same design [57], and the same is true for Nvidia A100 (training) and A30 (inference) [58]. The training and inference versions differ mainly in the number of cores.

In our study, we adopted NVDLA, Nvidia's DL accelerator [39], as our base architecture (to the best of our knowledge, NVDLA is the only industrial DL accelerator with open-source RTL access). However, the key findings from our work can be generalized to other accelerator designs, because DL accelerators follow a similar dataflow architecture [57, 59, 60, 61], and are expected to experience similar hardware failure effects.

The major modules in NVDLA include (1) $512KB$ on-chip buffers to store layer inputs, weights, partial sums, and layer outputs, (2) sequencing units that control the dataflow of inputs and weights, (3) 16 parallel compute units that perform MAC (Multiply-ACcumulate) operations, and (4) compute units for element-wise, activation, and pooling operations.

To use NVDLA for training, on the hardware side, we augmented the datapath so that bfloat16 and FP32 are used for MAC and element-wise operations, respectively, which is a common precision setting for training [62]. On the compilation side, we introduced extra matrix transpose and rotation operations such that the order of gradient computations, which is fixed by NVDLA's dataflow algorithm, matches that required by the training algorithm [63].

4.3.2 Fault Injection Framework

We achieve accurate and quick FI in our framework by (1) deriving a set of software fault models through a systematic analysis of NVDLA's RTL, and (2) injecting software faults

(i.e., instances of the software fault models) using Tensorflow [64], where each software fault accurately captures the behavior of a hardware failure. We have open-sourced our framework [54].

Hardware Fault Model

Hardware failures are modeled using single-cycle, single-FF (flip-flop) bit-flips. This fault model is widely used to study dynamic variations, unstable/marginal circuit behaviors, and soft errors [10, 11, 13, 15, 16, 65, 66, 67, 68, 69, 70, 71]. The observations obtained using this model may also provide insights for other failures that exhibit transient or intermittent effects. For example, the SlowDegrade outcome (Sec. 4.4.1), which was first revealed by our study, was later observed in real systems and root-caused to hardware failures that can be reproduced intermittently.

Software Fault Models

Based on the hardware fault model, we derived a set of software fault models, which can accurately represent the effects of hardware faults.

A subset of the software fault models in our framework draw similarities to those from our previous work called FIdelity, in Chapter 3, which provides software fault models for DNN inference workloads based on the same hardware fault model used in this study. These similar fault models are the ones used to represent bit-flips in a *datapath FF* (i.e., an FF in the accelerator's datapath) or a *local control FF* (i.e., an FF that controls exactly one datapath register), because the dataflow and compute operations are the same in the forward/backward pass of training, and also during inference. Therefore, given a single-cycle bit-flip in one of these FFs (following the hardware fault model), the number of faulty elements in the output tensor, their relative positions, and their faulty values are derived in exactly the same way for all of the above operations.

The key difference between the new framework and FIdelity lies in how bit-flips in *global control FFs* are modeled. Global control FFs are FFs in the control logic that affect more than one datapath registers. Thus, a bit-flip in a global control FF can result in many faulty elements in the output tensor of the current DNN layer (see Table 4.1 for some examples). For inference, it is highly likely that the final prediction will be different from the fault-free prediction, so FIdelity simply models bit-flips in global control FFs as such in Chapter 3. In contrast, for training, many faulty output elements in a single DNN layer do not necessarily lead to unexpected training outcomes, because the training process may still be able to recover from the hardware failure effects. Therefore, accurate software fault models for bit-flips in global control FFs are required for training.

To this end, we systematically studied the functionalities of all global control FFs in NVDLA ($41K$ in total, corresponding to $7,531$ unique control variables) to derive the corresponding software fault models, as summarized in Table 4.1.

Validating the New Software Fault Models

We performed $40K$ RTL FI experiments, targeting global control FFs, for five layers arbitrarily selected from five representative DNN models: GoogleNet [35], Resnet [72], Transformer [73], Yolo [74], and LSTM [75]. For each RTL experiment where the injected fault is not masked by hardware ($11K$ total), we confirmed that the faulty output elements match those obtained by simulating the corresponding software fault. Given this result, we can estimate with 99% confidence that the accuracy of our software fault models is very high, with < 1 in $1M$ faults not modeled correctly.

4.3.3 Experiment Setup

We implemented the software fault models derived for NVDLA using Tensorflow APIs. The DNN models used in our study are summarized in Table 4.2. In the fault-free runs, we

Table 4.1: Fault injection framework and methodology.

DL accelerator: NVDLA [39], adopted for training
Software fault injection platform: Tensorflow [64]
Hardware fault model: a single-cycle bit-flip in a single FF

Definitions and terminologies:

Layer_ Output: output neurons in forward pass, input gradients or weight gradients in backward pass.

Layer_ Input_ 1: input feature map in forward pass and for weight gradient operations, output gradients for input gradient operations.

Layer_ Input_ 2: weights in forward pass and for input gradient operations, output gradients for weight gradient operations.

n: an integer ≥ 1 indicating how long the effect of a fault lasts in a given DNN layer. If the FF where the fault occurs has a feedback
loop. n is randomly chosen between 1 and the max number of loop iterations. Otherwise, $n = 1$.

Layer_ Outputs computed in one cycle: they belong to 16 consecutive channels, computed by 16 MAC units in parallel.

Layer_ Outputs computed in n consecutive cycles: output elements across n cycles grow in the width dimension.

Layer_ Inputs_ 1/Layer_ Inputs_ 2 required in one cycle: they belong to 64 consecutive channels.

Layer_ Inputs_ 1/Layer_ Inputs_ 2 required in n consecutive cycles: input elements across n cycles grow in the width dimension.

Software fault models for datapath FFs and local control FFs: same as FIdelity

Accurate software fault models for global control FFs	**For which bit-flips?**	**% (Num.) of FFs***
1. Random faulty values that can span the entire data precision dynamic range are set in all Layer_Outputs computed in one cycle, for n consecutive cycles.	A bit-flip in a configuration FF, or a valid signal for Layer_Output turns from 'invalid' to 'valid', affecting all 16 MAC units.	0.24%(1723)
2. All Layer_Outputs computed in one cycle are set to 0, for n consecutive cycles.	A valid signal for Layer_Output turns from 'valid' to 'invalid', affecting all 16 MAC units.	0.25%(1795)
3. One Layer_Output element is randomly chosen, and its value is set to a random faulty value in each cycle. This effect lasts for n consecutive cycles.	Same as group 1, but the bit-flips affect only one MAC unit.	0.48%(3448)
4. All Layer_Outputs computed in one cycle are written to incorrect, randomly chosen memory locations while maintaining their relative positions, for n consecutive cycles.	Bit-flips in FFs that control the memory addresses of Layer_Outputs.	2.36%(16952)
5 / 6. All Layer_Inputs_1 / Layer_Inputs_2 required in one cycle are read from incorrect, randomly chosen memory locations while maintaining their relative positions, for n consecutive cycles (from DRAM) or one cycle (from on-chip buffers).	Bit-flips in FFs that represent the memory addresses of Layer_Inputs_1 / Layer_Inputs_2.	1.31%(9410) / 0.96%(6895)
7 / 8. All Layer_Inputs_1 / Layer_Inputs_2 required in one cycle are set to 0, for n consecutive cycles (from DRAM) or one cycle (from on-chip buffers).	A valid signal for Layer_Input_1 / Layer_Input_2 turns from 'invalid' to 'valid'.	0.09%(646) / 0.22%(1580)
9 / 10. All Layer_Inputs_1 / Layer_Inputs_2 required in one cycle use a random set of values from Layer_Input_1 / Layer_Input_2, while maintaining their relative positions, for n consecutive cycles (from DRAM) and 1 cycle (from on-chip buffers).	A valid signal for Layer_Input_1 / Layer_Input_2 turns from 'valid' to 'invalid'.	0.16%(1149) / 0.12%(862)

* Since we augmented the datapath to support bfloat16 and FP32 for MAC and element-wise operations, the % of FFs is different from the numbers reported in Table 5.1.

trained each workload for $430 - 50K$ iterations, which corresponds to $40 - 80$ epochs with 8 training devices (similar to typical training procedures in the literature [62]). For each workload, the final fault-free training/test accuracy reaches $> 95\%$ of that reported in the corresponding paper cited in Table 4.2.

We deployed our framework on Google Cloud TPUs, and conducted $> 2.9M$ ($> 490K$ node hours) FI experiments. Each FI experiment consists of the following steps: (1) randomly select an FF and a cycle to indicate where and when a bit-flip is to be injected; (2) use the corresponding software fault model to obtain the number and the positions of all faulty output elements in the current DNN layer; (3) obtain the faulty values of the faulty output elements based on the software fault model; and, (4) propagate the effects of the faulty output elements in the current DNN layer by continuing to train the DNN until either an error message (e.g., one that reports the occurrence of INFs/NaNs) is encountered, or until a predefined number of training iterations are completed.

For each workload, the upper bound of the training iterations used in our experiments is 2× the number of iterations in the fault-free run (reported in Table 4.2). In each FI experiment, we captured the convergence trend by recording the training loss and accuracy values in every training iteration, as well as the test accuracy once every 100 training iterations.

4.4 Results

4.4.1 Characterization of Hardware Failure Effects

We observed two distinct categories of training outcomes from our FI experiments. In the first category, which accounts for 82.3%−90.3% of all cases across the workloads, the injected faults did not significantly affect the final training/test accuracy for the same training time as the fault-free runs. In fact, the majority of them (65.5% − 86.3% of all cases) yielded slightly higher training/test accuracy compared to the fault-free cases, perhaps because the faults created noises that introduced certain regularization effects. The rest of the cases in this category showed slight degradations (mostly within 2%, up to 6%) in training/test accuracy for the same training time compared to the fault-free runs. These cases by and large correspond to those where faults were injected late in the training process. For these

Table 4.2: DNN training workloads.
Optimizer: Adam (except for Resnet_SGD).
Momentum value in batch normalization (BatchNorm) layers: 0.9 (except for Resnet_LargeDecay).

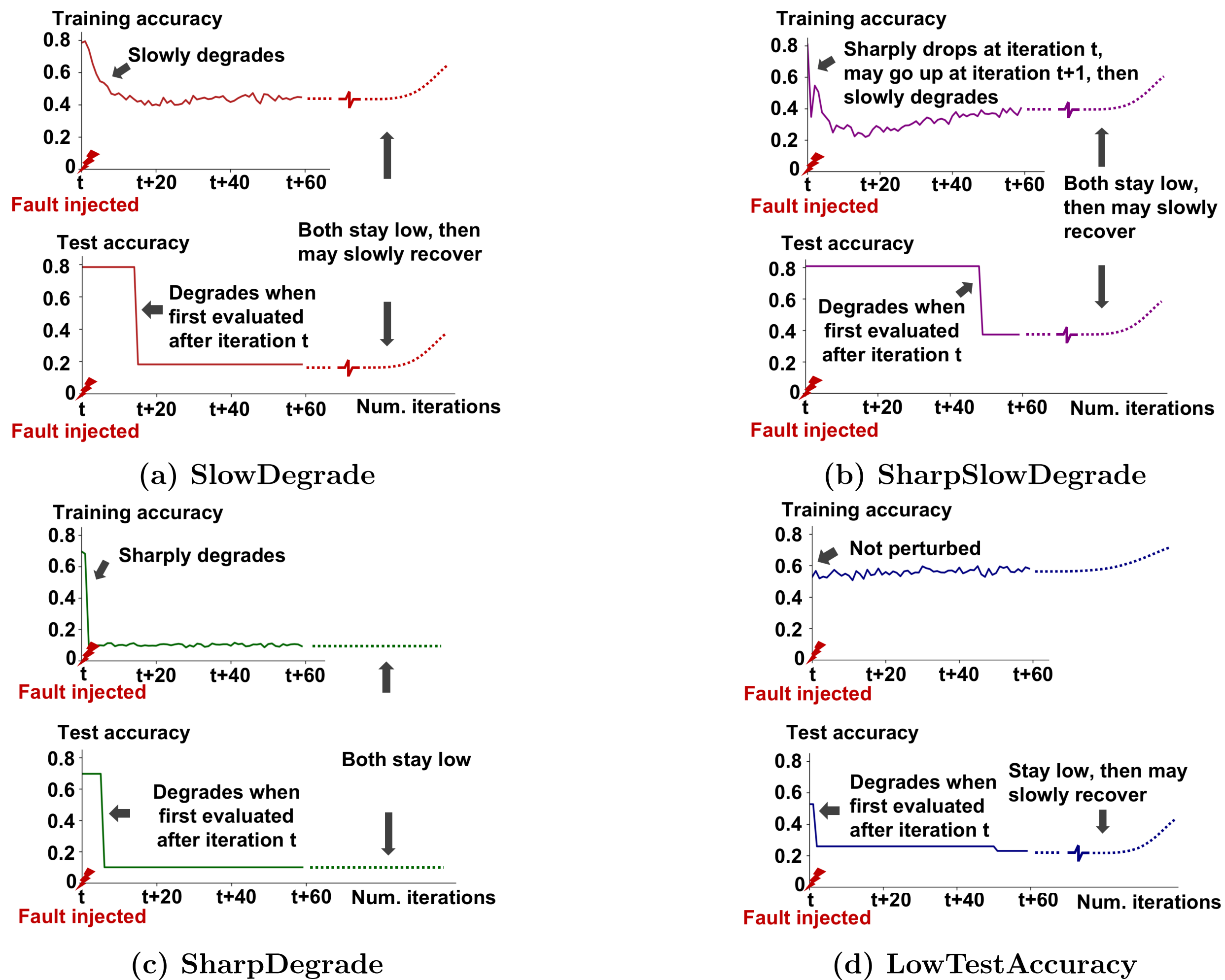
DNN models	Data-sets	Num. iterations / epochs (fault free)	Num. experiments
Resnet [72] (4 configurations*)	Cifar10[76]	1960 / 80	>900K
DenseNet [77]			>400K
Efficientnet [78]			>400K
NFNet [79]			>100K
Yolov3 [74]	VOC12[80]	430 / 40	>200K
Multi-grid neural memory [81]	25*25 maze	50000 / N/A	>400K
Transformer [73]	WMT14 EN-DE [82]	50000 / 40	>100K

* Four configuration of Resnet18: (1) Resnet, a BatchNorm layer follows every convolution layer; (2) Resnet_NoBN, no BatchNorm layers; (3) Resnet_SGD, same as Resnet, except that SGD (stochastic gradient decent) is used as the optimizer; (4) Resnet_LargeDecay, same as Resnet, except that the momentum value in BatchNorm layers is 0.99.

cases, when we increased the training time by 10% / 17% to allow the training algorithm to recover the effects of the faults, the training/test accuracy differed by only less than 2% / 0.5% from that of the corresponding fault-free runs.

The remaining 9.7% − 17.7% of the FI experiments, belonging to the second category, all exhibit certain unexpected training outcomes. We characterized these outcomes based on (1) convergence trends (i.e., training/test accuracy values throughout the training process), and (2) occurrences of visible anomalies, as shown in Table 4.3. In addition to the occurrences of INFs/NaNs which have been reported by industry, we discovered four new unexpected outcomes: (1) SlowDegrade, (2) SharpSlowDegrade, (3) SharpDegrade, and (4) LowTestAccuracy. In Fig. 4.3, we report the percentage breakdown of different training outcomes normalized to the total number of experiments for each workload.

Based on the same statistics analysis methodology used in previous resilience studies [10, 83], we have achieved a 99% confidence level that the percentage of each outcome reported in this section is within a confidence interval of 0.1%. The probability of an unexpected outcome not exposed by our experiments is $< 0.004\%$ with a 99.5% confidence level. Moreover, after observing the SlowDegrade outcome in our experiments, this outcome was later observed in datacenters when training large DNN workloads using DL training accelerator systems.



(a) SlowDegrade

(b) SharpSlowDegrade

(c) SharpDegrade

(d) LowTestAccuracy

Figure 4.2: Four new unexpected latent outcomes observed from our experiments.

Table 4.3: Unexpected outcomes in DNN training workloads.

Symptoms	Descriptions
Manifestation latency: immediate	
INFs/NaNs	A fault in the forward pass: INFs/NaNs are observed in the forward or backward pass of the current iteration. A fault in the backward pass: INFs/NaNs are observed in either the backward pass of the current iteration, or the forward pass of the next iteration.
Low hardware utilization	Hardware resources are not fully utilized, resulting in sub-optimal performance, because a faulty control FF incorrectly disables a subset of hardware modules.
Accelerator hang	The accelerator fails to notify the host server that its task is completed within a pre-specified timeframe, because a fault causes some control logic to be stuck in an infinite loop.
Manifestation latency: short-term	
INFs/NaNs	INFs/NaNs show up within a few training iterations (2 in our experiments) after a fault occurs.
Manifestation latency: latent	
SlowDegrade (Fig. 4.2a)	Training accuracy slowly degrades for $10-100$ iterations, then stays at a low level. Training/test accuracy may recover after $10K - 100M$ iterations.
SharpSlow Degrade (Fig. 4.2b)	Similar to SlowDegrade, except that an additional sharp drop in training accuracy is observed at the iteration when a fault occurs.
SharpDegrade (Fig. 4.2c)	Training accuracy drops sharply at the iteration when a fault occurs, and stays at a low level. Test accuracy follows training accuracy.
LowTest Accuracy (Fig. 4.2d)	Training accuracy appears normal, but test accuracy shows visible degradation after a fault occurs. Test accuracy may recover after $10K - 100M$ iterations.

4.4.2 Detailed Analysis

A detailed characterization of the fault propagation paths and effects are summarized in Fig. 4.4. We have also derived the necessary conditions for a fault to generate a latent unexpected

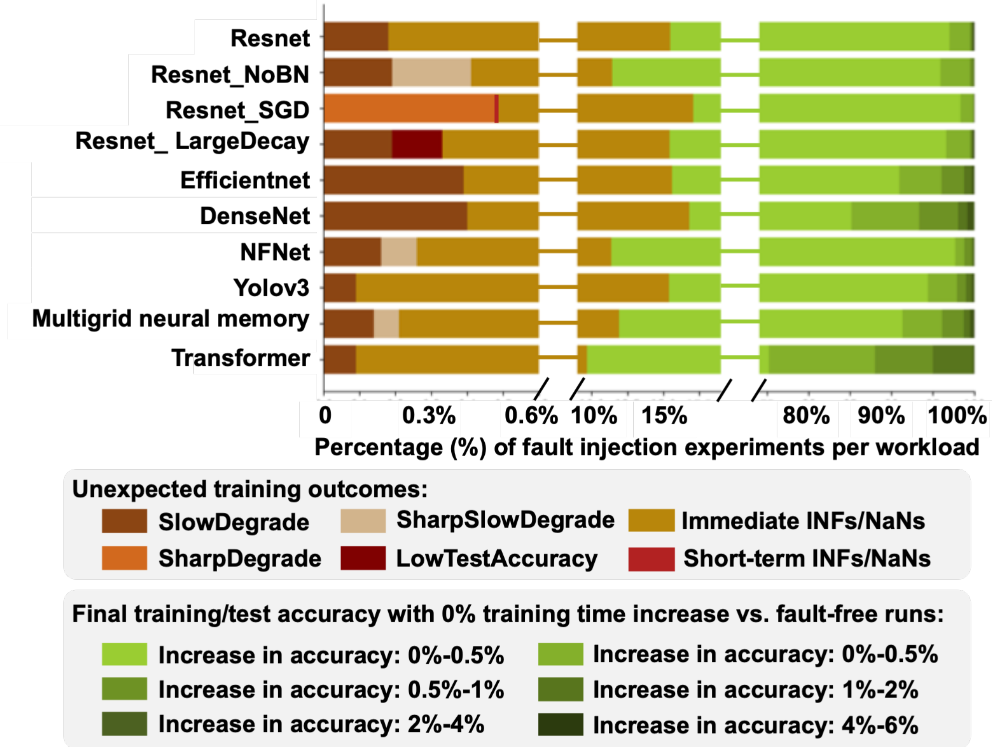


Figure 4.3: Percentages of training outcomes, normalized to the total number of experiments for each workload.

outcome.

Analysis on the Immediate Outcomes

Immediate INFs/NaNs are generated by faults in the following FFs: (1) the datapath FFs that represent the high exponent bits, (2) a subset of control FFs that configure the data precision (e.g., if a fault in one of these FFs causes int16 MAC operations to be performed instead of bfloat16 operations, the results may overflow when they are converted to FP32 to undergo element-wise operations), and (3) FFs that correspond to valid/invalid signals (a fault in one of these FFs can result in incorrect logic functions that generate arbitrary datapath values, including INFs/NaNs).

The faults that can generate the other two immediate unexpected outcomes (low hardware utilization and accelerator hang) cannot be modeled using software-visible states. We

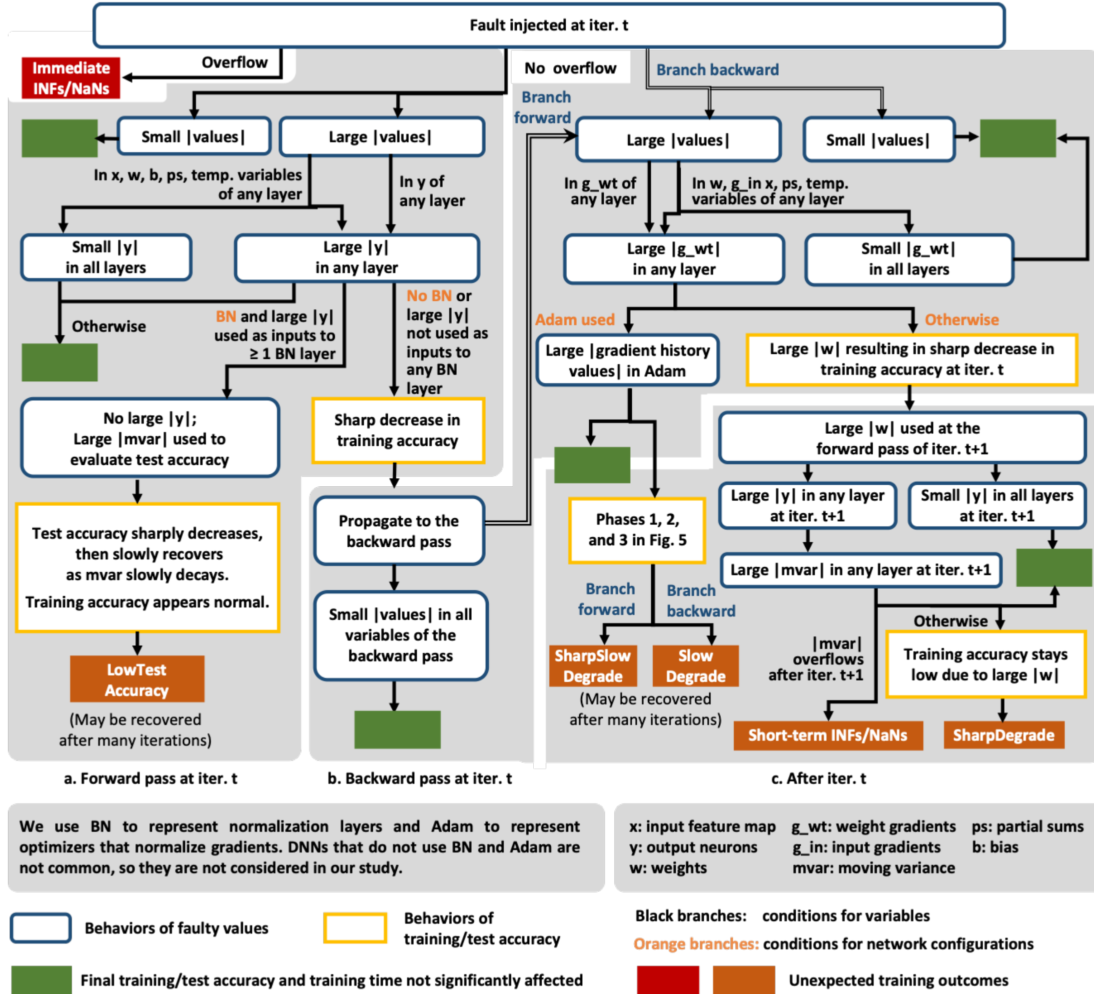


Figure 4.4: Characterization of fault propagation paths and effects.

observed these outcomes in our RTL FI experiments (see Sec. 4.3.2), and included them in Table 4.3 for completeness.

Analysis on the Short-Term INFs/NaNs Outcome

The fault propagation paths leading to this outcome are shown in Fig. 4.4. There are two major events along these paths. First, at the end of iteration t (i.e., the iteration when a fault occurs), weights with large absolute values are generated as a result of the fault and propagate to subsequent training iterations. Second, a history term combines the effects of large absolute faulty weight values across at least two training iterations to generate a value

that overflows after iteration $t+1$.

History terms are used in normalization layers that are widely adopted in DNNs [79, 84]. For example, moving variance ($mvar$) in BatchNorm is such a history term, which combines the variance of the layer's inputs with the $mvar$ obtained in the previous iteration, weighted using a decay factor: $mvar\ at\ iter\ (i+1) = decay_factor * mvar\ at\ iter\ i + (1 - decay_factor) * input\ variance$. The inputs of a BatchNorm layer are the outputs of the previous layer, which can contain large absolute neuron values because of the faulty weight values. The faulty BatchNorm layer inputs can result in a large absolute $mvar$ value, which may overflow after iteration $t+1$. For clarity, we use $mvar$ to generally denote such a history term in normalization layers.

Short-term INFs/NaNs are rare. First, if an optimizer that normalizes gradients (e.g., Adam) is used, large absolute weight values can only be generated if a fault occurs during the weight update operation (i.e., the operation that adds gradients to current weight values), which is extremely unlikely because this operation takes a very small amount of time. This is why we observe this case for Resnet_SGD only in our experiments, since SGD does not normalize gradients.

Second, the absolute value of a faulty $mvar$ across multiple iterations must lie in a specific range ($2.9e38 - 3.0e38$ from our experiments, as shown in Table 4.4) so that it does not overflow at iteration t, but overflows at a later iteration. Moreover, the overflow is expected to appear shortly after iteration $t+1$ because (1) a decay factor is applied to $mvar$, and (2) although quite slowly, the faulty weights are updated towards the correct direction (decreasing their absolute values) by the optimizer. Thus, it is not likely for INFs/NaNs to occur beyond a small number of iterations after a fault occurs. Because of this decaying effect, the magnitude of $mvar$ at iteration $t+1$ must be very close to the max floating point value that can be represented (e.g., the max value of FP32 in our study). A large absolute $mvar$ value therefore is a *necessary condition* for this outcome.

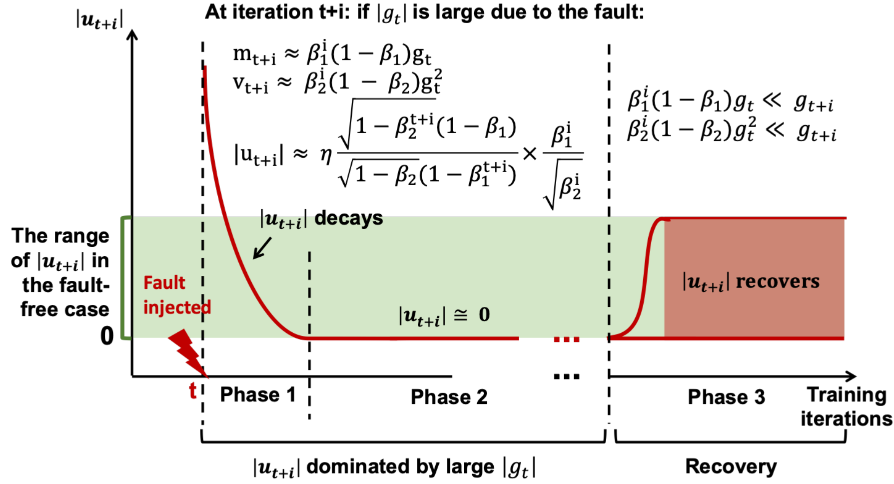


Figure 4.5: Explanation of the three phases in the convergence trends of SlowDegrade and SharpSlowDegrade. The math symbols are defined in Eq. 1.

The fault propagation paths that lead to these two outcomes are depicted in Fig. 4.4. Both outcomes are observed only if the optimizer uses gradient history values to normalize the gradients derived in the current iteration, which is common in DL training workloads (e.g., 134 such optimizers were developed out of a total of 154 between 2015 and 2021 [85]). SharpSlowDegrade can only occur if normalization layers are not present (e.g., Resnet_NoBN and NFNet) and if a fault occurs in the forward pass, while SlowDegrade can only occur if a fault occurs in the backward pass. Moreover, the convergence trends of these two outcomes exhibit three distinct phases. We mathematically explain each phase in Fig. 4.5 using Adam (Eq. 1) as an example.

From Fig. 4.4, we see that a *necessary condition* for both outcomes is that the absolute values of the faulty gradient history values of the optimizer (m_t and v_t in Eq. 1 for Adam) must be large enough to influence training accuracy, but not large enough to cause immediate or short-term INFs/NaNs (the range of faulty values obtained from our experiments is shown in Table 4.4).

However, this is a necessary condition but not a sufficient condition, because even if this

Operations performed in Adam

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

$$u_t = \eta \frac{\frac{m_t}{1-\beta_1^t}}{\sqrt{\frac{v_t}{1-\beta_2^t}} + \epsilon}, \quad w_t = w_{t-1} - u_t \tag{4.1}$$

g_t : gradient values computed in iteration t. β_1, β_2 : decay factors.

m_t : the history values of the gradients.

v_t : the history values of the square of the gradients.

u_t : the values used to update the weights. w_t : weight values.

η : learning rate. ϵ : a small value for numerical stability.

condition is met, it is possible that the final training/test accuracy would not be affected significantly. For example, if only a few gradient history values are perturbed, it may not be significant enough to perturb the overall convergence trend. Moreover, training/test accuracy can start to improve again in Phase 3 (Fig. 4.5). However, for all of our experiments in which the convergence trend is perturbed due to large absolute gradient history values except for those training the Transformer workload, the final training/test accuracy values are still low even after the numbers of training iterations are doubled with respect to the corresponding fault-free runs. The takeaway is that, although theoretically there is a recovery phase, the final training outcome is largely dependent on the interactions between the magnitudes of the faulty values, the choice of hyperparameters (e.g., decay factors), and the training dynamics. In practice, the recovery phase may never be reached, or it may require millions of iterations to fully recover from the fault. For example, the latter can happen with a decay factor of 0.9999 (used in real datacenter workloads) and a faulty absolute gradient history value in the order of $1e19$ (observed from our experiments).

Analysis on the SharpDegrade Latent Outcome

The propagation path that leads to the SharpDegrade outcome is mostly the same as that for the short-term INFs/NaNs outcome, except that faulty *mvar*'s never overflow in the case

of SharpDegrade. Instead, their absolute values must be large enough (which are generated by large absolute weight values due to a fault) for the training/test accuracy to show sharp degradations – this is thus a *necessary condition* for the SharpDegrade outcome. Afterwards, the absolute values of the faulty weights continue to stay large as they are updated very slowly by the optimizer, so the training/test accuracy stay low for a long time.

Analysis on the LowTestAccuracy Latent Outcome

From Fig. 4.4, we see that LowTestAccuracy can only occur in DNN workloads that satisfy two conditions. First, a history term is used, which is updated based on its values from previous iterations. Moreover, it is only used to evaluate test accuracy but not training accuracy. An example of such a history term is the moving variance in BatchNorm layers ($mvar$ as defined previously), which we will use to generally denote such history terms for clarity. Second, the absolute value of $mvar$ must be very large, such that it can visibly degrade the test accuracy (see Table 4.4 for the range observed in our experiments). Thus, this is a *necessary condition* for the LowTestAccuracy outcome.

Similar to the SlowDegrade and SharpSlowDegrade cases, there is typically a recovery phase for LowTestAccuracy, because a faulty absolute $mvar$ value will decay over time given the common use of a decay factor (as explained in Sec. 4.4.2). However, whether the test accuracy can be successfully recovered depends on various factors, including the magnitudes of the faulty values, the decay factor, and the training dynamics. In our experiments, LowTestAccuracy is observed for the Resnet_LargeDecay workload, because the large decay factor (0.99, vs. 0.9 in other workloads) corrects the faulty $mvar$'s too slowly.

Moreover, only faults that occur in the forward pass can lead to LowTestAccuracy, because those in the backward pass can only perturb $mvar$'s in the forward pass of the next training iteration through faulty weight values. However, in this case, large absolute weight values will dominate the overall effect and create the SharpDegrade outcome instead.

Summary

We summarize the necessary conditions for a fault to generate each latent outcome (including short-term INFs/NaNs) in Table 4.4. The necessary conditions always occur within two training iterations after a fault occurs. Note that, these necessary conditions are not sufficient. In our experiments, we observed cases in which the training process is able to recover from the effects of faulty gradient history values in optimizers or faulty *mvar*'s in normalization layers, especially if the number of faulty values is small.

Table 4.4: Necessary conditions for short-term/latent unexpected outcomes. iter. t is the iteration during which a fault is injected.

Outcomes	**Necessary conditions**	**When conditions observed**	**Ranges observed in experiments**
SlowDegrade	Large absolute gradient history values in optimizer	iter. t	3.6e9-1.1e19
Sharp SlowDegrade		iter. t	2.7e8-1.2e19
SharpDegrade	Large absolute *mvar* values in normalization layers	iter. $t+1$	6.5e16-1.2e38
LowTestAccuracy		iter. t	7.3e17-7.1e37
Short-term INFs/NaNs		iter. $t+1$	2.9e38-3.0e38

In addition to the necessary conditions, we have obtained the following three key observations from our analysis.

Observation (1) Recovery effects of DNN training workloads. If the perturbations in all software variables that are affected by a fault are small, then the training process (provided that it is implemented correctly) is highly likely to be able to recover from the effects of the fault without posing high training time overheads. Even if the perturbations are large, given a long enough training time (which may not be practical), the training process may recover from the effects of the fault, unless INFs/NaNs are generated.

Observation (2) Necessary conditions for latent unexpected outcomes. For any hardware fault to cause a latent unexpected outcome, the effects of the fault need to last across multiple

training iterations; otherwise, it is highly likely that the training process will recover. This observation is reflected in the necessary conditions, as both the *mvar*'s in normalization layers and gradient history values in optimizers can carry the effects of a fault from one iteration to the next.

The effects of faulty weight/gradient values will also last across multiple training iterations; however, they will propagate to the *mvar*'s and/or gradient history values, and subsumed in our necessary conditions. On the other hand, faulty *mvar* and gradient history values do not always imply faulty weights/gradients.

We also analyzed the behaviors of the training loss value to determine if it can serve as a necessary condition for the latent unexpected training outcomes. For faults that occur in the forward pass and subsequently generate the SharpSlowDegrade, SharpDegrade, and short-term INFs/NaNs outcomes, a sharp increase in the training loss value is observed at the iterations during which the faults first appear. However, for faults that occur in the backward pass, even if they eventually lead to latent unexpected outcomes (SlowDegrade or LowTestAccuracy), the training loss value appears normal throughout the training process.

Observation (3) Interactions between DNN configurations and hardware failures. Normalization layers in DNNs play an important role in the resilience of DNN training workloads. On the one hand, the occurrence of large absolute *mvar*'s is a necessary condition for various short-term and latent unexpected outcomes. On the other hand, the presence of normalization layers makes it more likely for a training workload to recover from the faults that occur in the forward pass. As shown in Fig. 4.4, if large absolute output neurons (large $|y|$) are generated in the forward pass, normalization layers will normalize the magnitudes of these output neurons, effectively alleviating the impacts of these faults.

The choice of optimizers and hyperparameters also play an important role. For example, the SlowDegrade and SharpSlowDegrade outcomes can only be generated if the optimizer normalizes gradients using gradient history values, while the SharpDegrade outcome can

only occur if the optimizer does not.

4.4.3 Other Results and Discussions

Contributions to Unexpected Outcomes from Different FFs

In NVDLA, global control FFs whose bit-flips belong to groups 1 and 3 defined in Table 4.1 and local control FFs are more likely to generate large absolute *mvar*'s and large absolute gradient history values. As shown in Figure 4.6, together they contribute to 55.7-68.5% of the total number of unexpected outcomes across different workloads in our experiments, even though these FFs only account for 9.8% of all FFs in the design.

For datapath FFs, bit-flips that correspond to the upper two exponents bits (5.5% of all FFs) contribute to 31.9%-44.3% of all unexpected outcomes across different workloads (see Fig. 4.6). These bit-flips are more likely to generate large absolute values that cause overflow or satisfy the necessary conditions reported in Table 4.4 than the bit-flips in other datapath FFs.

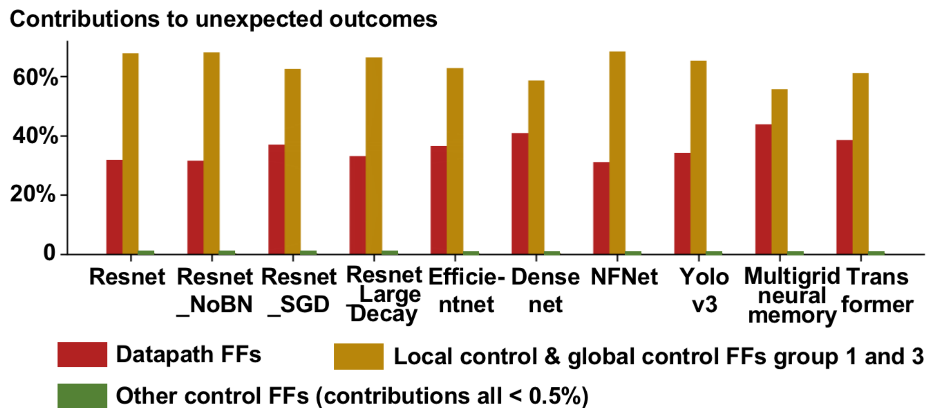


Figure 4.6: Contributions to unexpected outcomes for bit-flips in various FF groups.

Generalization to other hardware fault models

The necessary conditions discussed in Table 4.4 were derived based on the single-cycle single-FF bit-flip hardware fault model. However, based on Observation (2), the same necessary conditions are applicable to any single hardware failure, regardless of the fault model.

Furthermore, given the hardware failure rate reported by industry, it is expected that at most one hardware failure would occur during the training process of mid-sized DNNs (i.e., DNNs with $< 1GB$ parameters), which account for the majority of all DNNs deployed in datacenters today [57]. For larger DNNs, even though multiple failures may occur during the training process, they are expected to occur far enough apart such that their effects are largely independent. Therefore, the same necessary conditions are also applicable to multiple hardware failures under the reported hardware failure rate.

Discussions on the number of training devices.

We used 8 training devices in our experiments. With more training devices, our findings still apply. First, if a hardware failure occurs in a training device and generates an immediate unexpected outcome, the outcome will show up in the local device without affecting other devices, so the number of training devices is irrelevant. Second, the necessary condition for short-term INFs/NaNs, SharpDegrade and LowTestAccuracy is large absolute $mvar$ values on a single training device, which is not affected by the number of devices. Last but not least, we consider the SlowDegrade and SharpSlowDegrade outcomes, for which the necessary condition is large absolute gradient history values. On the one hand, using more training devices results in a shorter training time, which makes it less likely for a workload to reach the recovery phase, or for the recovery phase to fully recover the training/test accuracy. On the other hand, since gradients are averaged among all training devices, absolute faulty gradient values (due to a hardware failure) would be smaller if more devices are used, making it less likely to meet the necessary condition of these two outcomes. These opposing factors

balance out the sensitivity to the number of training devices.

Discussions on the sizes of DNNs and data-sets.

How hardware failures propagate and affect DNN training workloads, as shown in Fig. 4.4, does not depend on the sizes of the DNN or the training data-set. The only consideration is that the sizes may influence when the three phases in SlowDegrade/SharpSlowDegrade and the recovery phase in LowTestAccuracy occur, and how long the different phases last.

4.5 Techniques to Tackle Hardware Failures in DNN Training Systems

In datacenters, when a potential issue is detected in a DL accelerator, a standard procedure is to decommission the accelerator for further investigation, revert all affected workloads to their previous checkpoints, and execute these workloads in other healthy devices [1]. Handling immediate and short-term NaNs/INFs is easy. However, for a latent unexpected outcome, its error detection latency, i.e., the time between when a hardware failure occurs and when the unexpected outcome is observed, can be very long – spanning thousands to millions of training iterations. The long error detection latency makes it challenging to recover an affected workload. For example, even though checkpointing is routinely used in DNN training, it is not clear how one could determine which checkpoint to revert to, not to mention that the available checkpoints may all have been corrupted.

Therefore, a detection technique that guarantees a short error detection latency is required. Although there exist a plethora of resilience techniques in the literature, these techniques are inadequate because they incur high performance/energy costs even in the absence of hardware failures (more details in Sec. 4.6). To this end, we leverage the necessary conditions revealed by our study to devise new, efficient techniques to mitigate hardware

failures in DL training accelerator systems.

4.5.1 Detection

Our technique detects all hardware failures that are likely to lead to latent unexpected outcomes. The idea is to compare the gradient history values against a bound (for workloads trained by optimizers that use such history values), and also compare the *mvar*'s against a bound (for workloads with normalization layers). If any of these values is out of bound, an error message is generated. Since the necessary conditions occur within 2 training iterations after a failure occurs, the error detection latency of our technique is bounded. Further, we proved that these bounds can be mathematically derived based on the properties of a given DNN workload. As shown in Algorithm 11, the absolute gradient history values in the absence of hardware failures (and software bugs) are less than $20 \times \sqrt{n_l/m^2}$ with a probability larger than $(1 - 3 \times 10^{-89})$, where n_l is the number of partial sums used to generate one gradient value, and m is the batch size. Similarly, we derived a bound for the *mvar*'s in Algorithm 11. The bounds derived for all of our target workloads can be found in Table 4.5.

Note that, a hardware failure detected by our technique does not always lead to unexpected training outcomes. However, it is still beneficial to decommission the accelerator for further analysis because it is highly likely (based on the probability shown in Algorithm 11) that the accelerator has encountered a hardware failure.

4.5.2 Recovery

We developed a light-weight recovery technique that re-executes the two most recent iterations of a DNN training workload on all training devices, which is sufficient to mitigate all immediate, short-term, and latent unexpected outcomes when coupled with our detection technique. The following changes to a DNN training program are required to implement our

Algorithm 11: Derive bounds for gradient history values in Adam and moving variance values in BatchNorm. The bounds apply for various DNN layers including convolution, matrix multiplication, and fully-connected layers.

Without loss of generality, assume the following DNN properties [86, 87]:

(1)The mean of the outputs (before activation) and inputs of every DNN layer is 0, and the variance of all layers are approximately the same.

(2)The input data-set is normalized to zero mean and unit variance.

(3)Softmax-cross-entropy is used as the loss function, and Adam is used as the optimizer.

(4)The weight gradient values follow the Gaussian distribution.

I. Deriving the bound for absolute gradient history values in Adam.

Step 1: Let a_i and p_i := the ith inputs and outputs of the softmax layer; y_i := the ith one-hot encoded training target, $1 \leq i \leq I$; m := the number of mini-batches; L := Softmax-cross-entropy $= -\sum_i y_i log(p_i)/m$. We bound $\frac{\partial L}{\partial a_i}$, the input gradients of the last DNN layer $\forall i$.

$p_i = \frac{e^{a_i}}{\sum_{k=1}^{I} e^{a_k}}, \qquad \frac{\partial L}{\partial a_i} = (p_i - y_i)/m.$

$\because\ p_i \in [0,1], \quad y_i \in [0,1], \quad \therefore \frac{\partial L}{\partial a_i} \in [-\frac{1}{m}, \frac{1}{m}].$

Step 2: Let y_i^l := the ith element of the output tensor of layer l. We bound $\frac{\partial L}{\partial y_i^l}$ $\forall l$. Given Property 1,

$\frac{\partial L}{\partial a_i} \in [-\frac{1}{m}, \frac{1}{m}] \quad \rightarrow \quad \frac{\partial L}{\partial y_i^l} \in [-\frac{1}{m}, \frac{1}{m}]$ $\forall l$, since a_i is the output of the last layer.

Step 3: Let w_i^l := the ith element of the weight tensor of layer l; $\hat{x^l}$:= the transpose of layer l's input tensor; n_l := the number of the partial sums used to compute one gradient. We bound $\frac{\partial L}{\partial w_i^l}, \forall l$.

$\frac{\partial L}{\partial w^l} = \hat{x^l}\frac{\partial L}{\partial y^l} \rightarrow Var[\frac{\partial L}{\partial w^l}] = Var[\hat{x^l} \times \frac{\partial L}{\partial y^l}].$

Given Property 1, $\because\ \frac{\partial L}{\partial y_i^l} \in [-\frac{1}{m}, \frac{1}{m}], \quad \forall l\ \therefore$ In the worst case,

$\frac{\partial L}{\partial y_i^l} = \begin{cases} -\frac{1}{m} & x \in \hat{x^l}, x < 0 \\ \frac{1}{m} & x \in \hat{x^l}, x \geq 0 \end{cases}$ and

$Var[\frac{\partial L}{\partial w^l}] \leq \frac{n_l}{m^2} Var[x_l].$

Step 4: Based on the history value computation in Adam, shown in Eq. 1, $\because\ \beta < 1$, $\therefore$ the bound for $\frac{\partial L}{\partial w_i^l}$ can also be used for m_t. Given Properties 1, 2, and 4, $E[\frac{\partial L}{\partial w^l}] = 0$, and also given the bound for $Var[\frac{\partial L}{\partial w^l}]$ shown in Step 3, $m_t \sim \mathcal{N}(0, \frac{n_l}{m^2})$,

$\therefore\ Prob(|m_t| > 20 \times \sqrt{\frac{n_l}{m^2}}) < 3 \times 10^{-89}.$

II. Deriving the bound for absolute moving variance values in BatchNorm.

Step 1: Let η := learning rate; $g_t := \frac{\partial L}{\partial w^l}$ in iteration t; $w^{l'}$:= the weight values of layer l in iteration $t+1$; N_l := the number of partial sums used to compute one output neuron in layer l.

$\because$ Property 1, $E[w^l] = 0, Var[w^l] = \frac{1}{N_l}$. Since Adam is the optimizer, let $k := \sqrt{1-\beta_2^t}/(1-\beta_1^t)$, then we have $u_t \sim \mathcal{N}(0, \eta^2 k^2)$ based on Eq. 1.

Step 2: $\because u_t$ and w_l are independent, $\therefore E[w^{l'}] = 0$, and $Var[w^{l'}] \leq \frac{1}{N_l} + \eta^2 k^2$.

$\because Var[y^l] = N_l * Var[w^{l'}] * Var[y^{l-1}],$

$\therefore \frac{Var[y^l]}{Var[y^{l-1}]} = N_l * Var[w^{l'}] \leq 1 + N_l \eta^2 k^2.$

$\because$ Properties 1 and 2, $\therefore Var[y^l] \leq (1 + N_l \eta^2 k^2)^l$.

Step 3: Let $mvar_{l,t}$:= the moving variance of BatchNorm at layer l and iteration t; β := the decay factor. $mvar_{l,t} = \beta \times mvar_{l,t-1} + (1-\beta) \times Var[y^l]$.

$\because \beta < 1$, $\therefore$ the bound for $Var[y^l]$ can be used to bound $mvar_{l,t}$. $\therefore mvar_{l,t} \leq (1 + N_l \eta^2 k^2)^l$.

Table 4.5: Bounds derived from Algorithm 11 for target DNN training workloads

DNN models	Bounds for history values	Bounds for moving variance values
Resnet	21.2	3.2e6 (Not used for Resnet_NoBN)
Densenet	50.0	3.8e7
Efficientnet	49.4	2.1e9
NFNet	70.0	Not used
Yolov3	49.4	1.2e8
Multi-grid neural memory	84.9	Not used
Transformer	28.2	3.2e9

re-execution technique: (1) subtracting the gradients obtained in the last iteration from the current weight values to obtain the weight values used in the previous iteration; (2) reloading the mini-batch data-set used for the previous iteration; and (3) recording the seeds used to initialize random variables (if they are used) in the previous iteration, and applying them during re-execution.

4.5.3 Implementation and Evaluation

We implemented the detection and re-execution techniques in Tensorflow for the same set of workloads presented in Table 4.2, which requires only $24-32$ lines of code change to the different DNN programs. The memory overhead is negligible since our detection technique only requires two new variables to bound the gradient history and *mvar* values, and our re-execution technique only requires a few seeds to be stored (if seeds are used). We evaluated the techniques on Google Cloud TPUs, using the cloud TPU profiler [88] to obtain performance/power/memory overheads. For each workload, the bounds-checking and re-execution operations were both executed $10K$ times.

If no out-of-bound values are detected, the performance impact is $0.003\% - 0.025\%$ on average (geomean) across different DNN training workloads. If re-execution is invoked once, the average (geomean) performance impact is $0.04\% - 0.15\%$. Also, the profiler reported a similar utilization of TPU resources between the modified and original programs for each

workload, indicating that the power/memory overheads of our techniques are negligible.

Compared to the checkpointing approach where a checkpoint is saved at the end of each training epoch [62, 89], the performance/energy costs of our recovery technique are up to 500× lower (depending upon the number of iterations per epoch, which is typically $\sim 1,000$ iterations) assuming that 8 training devices are used.

4.6 Related Work

Resilience analysis on DNN workloads. There is one study that targets memory errors in DNN training workloads [19]. Memory errors in datacenters are not a critical concern because ECCs are commonly supported in both on-chip and off-chip memories. Moreover, errors in memory behave differently from those in logic. For example, all the latent unexpected outcomes revealed by our study are unique to hardware failures in logic.

Many conclusions and findings from previous work on the resilience of inference workloads [17, 41, 42, 43, 83, 90, 91, 92, 93, 94, 95, 96, 97, 98], including our work in Chapter 3 cannot be extended to training workloads due to the fundamental differences in their respective algorithms and resilience requirements, as summarized in Table 4.6.

Table 4.6: Resilience properties of inference vs. training.

Inference	Training (details in Sec. 4.4)
Normalization layers effectively mask hardware failures [17].	Normalization layers can exacerbate or reduce the impact of hardware failures. See Observation (3) in Sec. 4.4.2.
Failures that occur in early layers are more likely to generate visible anomalies [17, 18].	We observed this trend only for the failures that lead to the SlowDegrade outcome.
Hardware failures that occur in certain output feature maps or input data samples are more likely to generate visible anomalies [83].	We did not observe such correlations in training.
INFs/NaNs are not observed.	INFs/NaNs are a major class of unexpected DNN training outcomes.

Hardware Failure Mitigation Techniques. There exist a plethora of resilience techniques across various system design layers [11], spanning algorithm [99, 100], compiler/software [101, 102, 103, 104, 105, 106, 107, 108, 109], architecture [104, 105, 109, 110, 111, 112, 113,

114, 115, 116], and circuit [117]. Selectively protecting FFs using circuit-level solutions (e.g., FF hardening) is a potential resilience solution, and our results in Sec. 4.4.3 can guide which FFs to harden; however, it requires hardware modifications, which may not be possible or desirable. Existing compiler/software techniques and architecture techniques were mostly developed for CPUs or GPUs, and they rely on specific properties in CPU/GPU applications or architectures; therefore, they do not lend themselves to be used in DL accelerators.

The authors in [83] focused on inference workloads, and proposed selective duplication in the weight kernel level or the inference task level. However, it is not clear how one would apply the kernel-level technique to training, since weight values are not static during training. Regarding the task-level technique, it is also not clear how to determine the importance of each input sample because that depends on the model and various training dynamics. Without selective redundancy, detection through duplication (or other redundancy-based techniques) incur high overheads. To recover from a failure, the overhead will be even higher since additional operations need to be executed upon detection.

In the algorithm level, algorithm-based fault tolerance (ABFT) techniques have been developed for DNN inference workloads [99, 100]. We extended the idea in [99] to cover training workloads, implemented it in Tensorflow for Resnet [72], Efficientnet [78] and DenseNet [77], and obtained the performance/energy results for these workloads using Google Cloud TPUs. This ABFT technique requires non-trivial software modifications ($463 - 485$ lines of code change), and incurs large ($5\% - 7\%$) performance/energy costs even in the absence of hardware failures.

Another line of work proposed to bound the activation outputs to improve the resilience of inference workloads [17, 92, 98, 118, 119]. This approach is inadequate for training because it can only detect a small fraction (33.7% from our experiments) of all latent unexpected outcomes.

Gradient clipping techniques in DNN training. Gradient clipping techniques [120, 121, 122]

were proposed to boost test accuracy or reduce training time, without any resilience considerations. These techniques cannot be used to mitigate all unexpected training outcomes caused by hardware failures, because, as shown for the SlowDegrade, SharpDegrade, and LowTestAccuracy cases in Fig. 4.4, hardware failures can perturb gradient history / $mvar$ values without affecting gradient values. Moreover, the bounds from previous work were heuristically determined. In contrast, our bounds were derived mathematically based on DNN properties to yield high detection coverage for hardware failures that are likely to generate latent unexpected outcomes.

CHAPTER 5

UNDERSTANDING PERMANENT HARDWARE FAILURE EFFECTS IN DEEP LEARNING TRAINING SYSTEMS

The content of this chapter will be published in our paper titled "Understanding Permanent Hardware Failure Effects in Deep Learning Training Systems" in ETS23.

5.1 Introduction

In Chapter 4, we present a comprehensive study on how transient hardware failures affect DL training systems. However, it is important to note, as highlighted in Section 4.1, that permanent failures are also prevalently observed in previous industry reports. However, there is a notable dearth of research regarding the impact of permanent hardware failures on DL training accelerator systems. To bridge this gap, we present the first in-depth study on logic permanent hardware failures in DL training accelerator systems. The fundamental understanding obtained from this study advances our knowledge and leads to more efficient and cost-effective mitigation solutions. We developed a new software-level fault injection framework (Sec. 5.3), which leverages the architectural properties of a DL accelerator to model permanent hardware faults accurately in software. The permanent hardware fault models used in our study are single stuck-at-1 and single stuck-at-0 faults, which are widely used in the literature [123, 124, 125, 126, 127]. Similar to our previous studies presented in Chapter 4, we adopted NVDLA (augmented to support DNN training) as a case study. We injected faults into a subset of the datapath nets that directly represent one or more software variables, and conducted 102.8K fault injection experiments using three representative DNN models (Resnet18, YOLOv3, and Transformer). The key results are (more details in Sec. 5.4):

- Even though the effects of permanent faults are persistent, a large percentage ($93.3\%-95.2\%$ across the three DNN models) of the injected faults did not affect the final training/test accuracy and training test significantly compared to the fault-free runs.

- We observed two unexpected training outcomes: (1) INFs/NaNs that corrupt the training process; and (2) a sharp degradation (SharpDegrade) in training accuracy after the fault first occurs, and then both training accuracy and test accuracy stay low throughout the rest of the training process.

- Through our analysis, we also found that for a permanent fault to generate the SharpDegrade outcome, a sharp increase in the training loss value must occur *within two training iterations* after the fault first occurs. In other words, this is a *necessary condition* for the SharpDegrade outcome. Furthermore, if a fault generates INFs/NaNs, these INFs/NaNs always occur within one training iteration from when the fault first occurs.

Based on these results, we devised a new detection technique that compares the increase in training loss against a pre-computed bound, which can be mathematically derived based on the properties of a given DNN model. Using this technique, all SharpDegrade cases observed in our experiments are detected. We also developed a lightweight recovery technique that re-executes the two most recent training iterations after a hardware failure is detected. We implemented our techniques in Tensorflow for the three DNN models used in our fault injection experiments, which require only $15-25$ lines of code change. We also evaluated these techniques on Google Cloud TPUs, which shows that the performance overhead is minimal ($0.004\%-0.025\%$) when no hardware failures are detected. If one permanent failure is detected, the recovery overhead (in terms of performance and energy) only involves the cost for re-deploying the workload on a healthy device and re-executing two training iterations. We present the details of our techniques in Sec. 5.5.

5.2 Related Work

Resilience Studies on DNN Workloads Prior work largely focuses on transient hardware failures and inference workloads. A few studies on logic permanent hardware failures in DL accelerators [123, 124, 125] also focus on inference only. The results obtained by studying inference workloads do not apply to training workloads because of the fundamental differences in the algorithms of these two types of workloads. Moreover, in [123], netlist simulation was used to perform fault injection experiments in systolic arrays, but the authors were only able to conduct a small number of experiments due to long simulation time, so the statistical significance of the results is low. In [124], the authors first estimated the worst-case perturbations that a fault can bring to the outputs of a DNN layer by determining which fan-in logic cone of an output bit this fault belongs to, and then obtained the effects of the perturbations using software simulation. However, the worst-case analysis approach cannot be used to obtain a comprehensive understanding on how various permanent hardware failures affect training workloads. In [125], the authors used a Python-based fault injection framework to simulate single stuck-at faults only at the primary input ports of a processing engine (256 cases) for inference workloads. In contrast, we consider a broader set of faults and focus on training workloads.

Hardware Failure Mitigation Techniques Structural in-field self-test techniques that are generally applicable to any compute devices have been developed and adopted commercially [25, 26, 128]. Functional in-field self-test for DL accelerators also exists (Chapter 2). However, both functional and structural in-field self-test techniques may only be invoked infrequently to minimize system-level costs, which may lead to long error detection latencies and pose significant challenges in error recovery. Furthermore, structural techniques require special hardware test support, which may not be practical or desirable. Redundancy-based concurrent error detection techniques generally incur high costs, as discussed in Chapter 4,

Section 4.6.

5.3 Methodology and Framework

Statistical fault injection (FI) is widely used to analyze hardware failures. FI can be performed at the hardware (RTL) level; however, the simulation time is prohibitively long. Using Resnet18 with Cifar10 as an example, it would take $5K$ years with 8 threads to perform $102.8K$ experiments. Alternatively, software FI can be performed; however, its accuracy cannot be guaranteed [10, 56]. We overcome the above limitations by developing a new software FI framework that is both quick and allows us to obtain accurate results.

5.3.1 Fault Injection Framework

Hardware Fault Models

Similar to previous work (e.g., [123, 124, 125]), the single stuck-at-1 and single stuck-at-0 fault models, which are among the most representative models for logic permanent hardware failures, are used in our framework.

Software Fault Models

To achieve accurate results, the architectural properties of DL accelerators can be leveraged to derive software fault models that accurately reflect the effects of single stuck-at faults. A key challenge of this approach is that stuck-at faults can occur in all nets in the accelerator design. If we were to exactly model a stuck-at fault of an internal net using software-visible states, we would need to simulate the detailed circuit structure to propagate the fault until it reaches a sequential element that is mapped to a software variable, which would be as time-consuming as RTL simulations.

Due to this challenge, in our study, we only include a subset of datapath nets for which the stuck-at faults can be directly modeled by injecting equivalent faults in certain software variables, as shown in Table 5.1. These datapath nets, referred to as the *target datapath nets* for short, account for 14.41% of the total number of nets in NVDLA.

To model single stuck-at faults in the target datapath nets, the only hardware information required is the dataflow algorithm implemented in the accelerator design, including the datapath widths, the total number of compute units, which input variables are processed at the same cycle, and which output variables are computed at the same cycle. Based on this information, we can derive which software variables in the current DNN layer are affected by a single stuck-at fault, which is critical because *multiple* software variables can be affected by a single faulty net. Moreover, since a target datapath net directly maps to a set of corresponding software variables, for a given single stuck-at-fault, equivalent faults can be injected into these software variables directly.

We validated these software fault models by performing $25K$ RTL FI experiments using five layers randomly selected from five DNN models (GoogleNet, Resnet18, Transformer, YOLOv3, and LSTM), which achieves 99% confidence that < 1 fault in $1M$ faults is not correctly modeled.

5.3.2 Experiment setup

To perform large-scale FI experiments, we implemented the software fault models described in Table 5.1, using Tensorflow as the DNN software platform. We deployed our framework on Google Cloud TPUs, and conducted $102.8K$ FI experiments. We use three representative DNN models in our study, and the details are summarized in Table 5.2.

Each FI experiment consists of the following steps. (1) Randomly select a bit in one target datapath net and a training iteration to specify where and when to inject a stuck-at fault (the fault is always injected at the beginning of the selected training iteration). Whether

Table 5.1: **Fault injection framework and methodology.**

DL accelerator: NVDLA [39] adopted for training **Software fault injection platform**: Tensorflow [64] **Hardware fault model**: Stuck-at-0 & Stuck-at-1			
Definitions and explanations of terminologies: *Layer_Output*: output neurons in forward pass, input gradients or weight gradients in backward pass. *Layer_Input_1*: input feature map in forward pass and for weight gradient operations, output gradients for input gradient operations. *Layer_Input_2*: weights in forward pass and for input gradient operations, output gradients for weight gradient operations. *Layer_Outputs computed in one cycle*: they belong to 16 consecutive channels, computed by 16 MAC units in parallel. *Layer_Inputs_1/Layer_Inputs_2 required in one cycle*: they belong to 64 consecutive channels.			
Accurate software fault models for stuck-at faults	**Datapath nets modeled**	**Hardware units**	**% / Num. of Nets**
1. In one out of the 64 Layer_Input_1 computed in the same cycle, one randomly chosen bit is stuck at 0/1. Error effects start at the iteration & layer where failure occurs, and last till the end of the FI experiments.	Datapath nets that correspond to the first input tensor of a DNN layer.	CDMA	1.75% / 36246
		CSC	0.77% / 15948
		CBUF	0.38% / 7871
		CMAC	1.41% / 29204
		Memory interface	0.97% / 20091
		Total	5.28% / 109360
2. In one out of the 64 Layer_Input_2 computed in the same cycle, one randomly chosen bit is stuck at 0/1. Error effects start at the iteration & layer where failure occurs, and last till the end of the FI experiments.	Datapath nets that correspond to the second input tensor of a DNN layer.	CDMA	0.78% / 16155
		CSC	0.74% / 15327
		CBUF	0.38% / 7870
		CMAC	1.41% / 29204
		Memory interface	0.97% / 20090
		Total	4.27% / 88436
3. In one out of the 16 Layer_Output computed in the same cycle, one randomly chosen bit is stuck at 0/1. Error effects start at the iteration & layer where failure occurs, and last till the end of the FI experiments.	Datapath nets that correspond to the output neurons of a DNN layer.	CACC	1.38% / 28582
		SDP	2.97% / 61514
		Memory interface	0.49% / 10148
		Total	4.86% / 100654

the fault is stuck-at-0 or stuck-at-1 is also randomly chosen. (2) Use the corresponding software fault model from Table 5.1 to inject equivalent faults in software. (3) Propagate the effects of the fault until either an error message is encountered, or until a predefined number of training iterations are completed. The upper bound used in our experiments is $2x$ the training time of the fault-free run.

5.4 Key Results

Of the $\sim 100K$ FI experiments, $93.3\% - 95.2\%$ of all injected stuck-at faults did not affect the final training/test accuracy significantly. Compared to the fault-free runs across different workloads, without any increase in training time, 52.1%-67.5% of all FI cases result in an increase in accuracy by less than 0.5% (similar the results presented in Sec. 4.4, because the faults created noises that might introduced certain regularization effects), 27.7%-41.1% of

Table 5.2: DNN training workloads.

DNN models	Data-sets	Num. iterations / epochs (fault-free)	Num. FI experiments
Resnet18 [72]	Cifar10[76]	1960 / 80	50.9K
YOLOv3 [74]	VOC12[80]	430 / 40	21.4K
Transformer [73]	WMT14 EN-DE [82]	50000 / 40	30.5K

all cases result in a decrease in training/test accuracy by less than 2%, and 0.01%-0.03% of all cases result in a decrease in training/test accuracy by 2%-4%. The cases where a > 2% degradation in training/test accuracy were observed correspond to those where faults were injected to the last 5 epochs of the training processes. For the remaining $4.8\% - 6.7\%$ of all FI cases, two unexpected outcomes[1] are observed:

(1) Immediate INFs/NaNs, where INFs/NaNs are generated in the training iteration when a stuck-at fault first occurs.

(2) SharpDegrade (Fig. 5.1), where training accuracy drops sharply within two training iterations after a fault occurs and remains at a low level. If the fault is injected in the forward pass, training accuracy drops at the same iteration, right after the fault occurs. If the fault is injected in the backward pass, the accuracy drops at the next iteration. The trend of test accuracy follows that of training accuracy.

These results are statistically significant. With $100K$ experiments, we have achieved a 99% confidence level that the percentage of each outcome reported in this section is within a confidence interval of 0.1%. Moreover, the probability of an unexpected outcome not exposed by our experiments is < 0.01% with a 99% confidence level.

In Fig. 5.2, we show the percentage breakdown of each unexpected outcome for different software fault models and DNN models.

1. In NVDLA, stuck-at faults in a few nets in the control logic may result in accelerator hangs or abnormally low hardware utilization. As discussed in 4.4, these effects cannot be modeled in software but can be detected easily. Therefore, they are not included in our analysis.

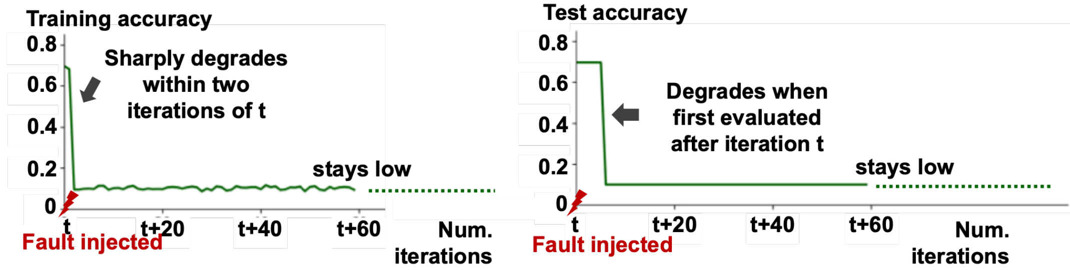


Figure 5.1: The training/test accuracy trends of SharpDegrade.

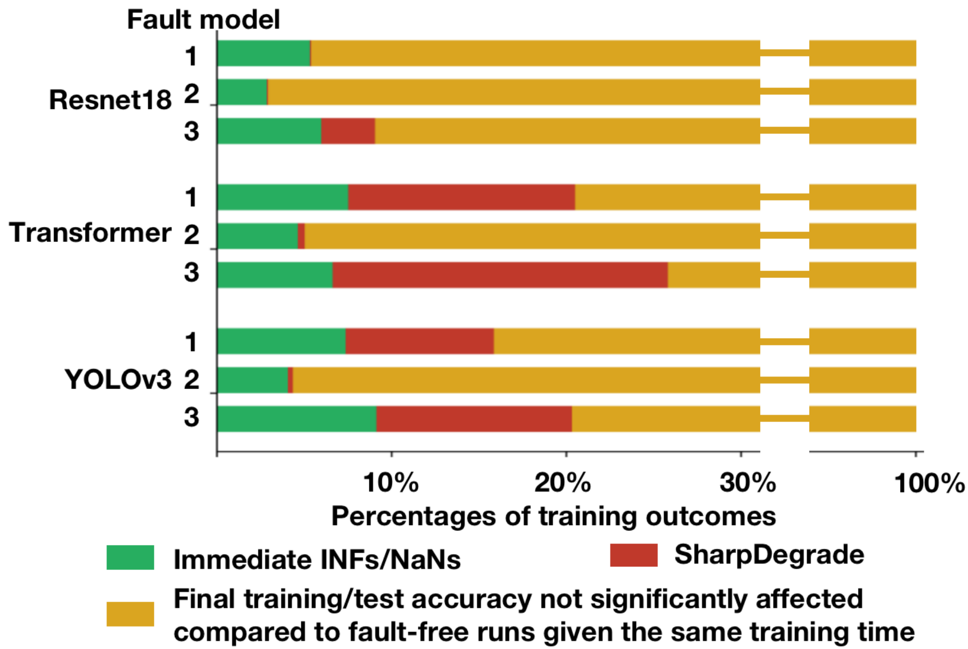


Figure 5.2: Percentages of unexpected training outcomes, normalized to the total number of experiments for each software fault model and each network model.

5.4.1 Analysis on the Unexpected Training Outcomes

Immediate INFs/NaNs. As shown in Fig. 5.3, the majority of the immediate INFs/NaNs are generated by stuck-at-1 faults in the datapath nets that represent the top two exponent bits of a software variable. This is because these faults affect every single layer in both the forward and backward passes of every training iteration (after the fault occurs), resulting in values with very large magnitudes, which then quickly turn into an overflow. For software fault models 1 and 3, all stuck-at-1 faults injected to the top two exponent bits generate

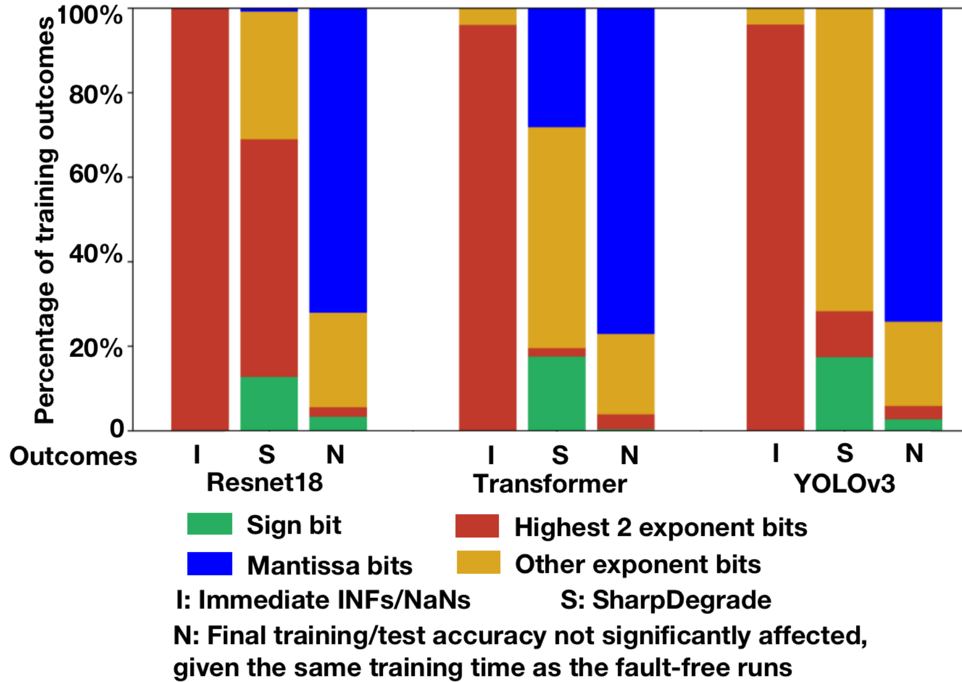


Figure 5.3: Percentages of training outcomes for different bit positions where hardware failures occur.

immediate INFs/NaNs; while for fault model 2, only those injected to the highest exponent bit generate immediate INFs/NaNs. This is because a stuck-at-1 fault on the second-highest exponent bit will generate a large value only if the original value is greater than 2.0 (meaning that the highest exponent bit is 1), which is rare for Layer_Input_2 (defined in Table 5.1 for fault model 2) in today's DNN models. On the other hand, values > 2.0 are more common in Layer_Input_1 (fault model 1) and Layer_Output (fault model 3).

SharpDegrade. Across all workloads, first consider all cases where stuck-at-1 faults generate the SharpDegrade outcome. Figure 5.3 shows that, 10.1%, 75.6%, and 14.3% of these cases are generated by faults in the sign bit, exponent bits, and mantissa bits, respectively. For all stuck-at-0 faults that generate this outcome, 97.4% of these faults occur in the datapath nets that represent the highest exponent bits (in other words, faults that convert values > 2.0 to values that are close to 0), while the rest occur in the nets that represent other exponent bits. The faults that lead to this outcome all introduce relatively

large perturbations.

When breaking down the contributions of different fault models, we found that fault model 2 rarely generates the SharpDegrade outcome. This is because, as discussed in the Immediate INFs/NaNs case, almost all the highest exponent bits in Layer_Input_2 are 0's, which means that it is not likely for either the stuck-at-0 or stuck-at-1 faults to introduce large perturbations.

Because a sharp decrease in training accuracy is always accompanied by a larger increase in the loss value, the latter is a *necessary condition* for this outcome.

5.4.2 Summary and Generalization

Our key finding is that the amount of perturbations caused by a hardware stuck-at fault largely determines the outcome. If the perturbations are small (which is the majority of the cases), the training algorithm is likely to be able to mask their effects. If the fault causes the absolute values of the affected variables to increase substantially, then immediate INFs/NaNs or SharpDegrade may be generated. If the fault causes the absolute values of the affected variables to decrease substantially, then it may lead to the SharpDegrade outcome. Overall, the amount of perturbations is determined by the type of the software variables affected by the fault, the bit position of the fault, and whether the fault is stuck-at-0 or stuck-at-1.

The above results may be generalized to other datapath and control nets. First, based on our systematic analysis of the RTL of NVDLA (and confirmed by RTL simulations), the majority of the datapath nets not in the target datapath nets belong to the input logic cone of a single target datapath net. Thus, the software variables affected by a fault in these datapath nets are already covered by one of the fault models in Table 5.1. The only inaccuracy is that we cannot obtain the exact faulty values. However, our FI experiments already cover a wide range of faulty values. Furthermore, our analysis in Sec. 5.4.1 suggests that the exact faulty values do not affect the high-level conclusions. Therefore, the key

findings of our study may apply to other datapath nets. Second, for nets that belong to the control logic, they can be classified into local control nets or global control nets. A local control net affects one datapath variable. By the same argument as above, our findings may also apply to local control nets. On the other hand, a global control net (e.g., one that configures the data precision or the dimensions of DNN layers, and so on) affects a large number of computations. We expect that stuck-at faults in global control nets are likely to generate visible system symptoms such as accelerator hangs, low hardware utilization, and Immediate INFs/NaNs.

Our results can be generalized to other DL accelerators, because these accelerators (for both training and inference) typically share a common dataflow architecture [57]. One consideration is the number of compute units in the design. For example, in NVDLA, if the number of MAC units increases from 16 to 32 or 64, the number of Layer_output variables affected by a stuck-at fault in a corresponding datapath net would decrease from 16 to 32 or 64 (see fault model 3 in Table 5.1), which may make it more likely for the training algorithm to mask the fault.

Our results may also be generalized to other DNN training workloads, because the value range and distribution of each software variable type are similar across many DNN models. Moreover, although the specific DNN training dynamics (determined by a wide range of factors including the DNN model, the optimizer algorithm, and hyperparameters) may affect how likely the SharpDegrade outcome will occur, the high-level findings and analysis remain the same.

5.5 New Mitigation Techniques

We utilize the same technique for detection as that presented in Chapter 4. This approach has been proven effective in identifying and recovering transient hardware failures in DL training systems. The difference is that for permanent failures, we leverage a different necessary

condition revealed by our study, i.e., a large increase in the loss value, to perform hardware failure detection.

In every training iteration, the increase in the training loss value is compared to a bound, and an out-of-bound value signifies the presence of a hardware failure. As this necessary condition is observed within two training iterations after a failure occurs, the error detection latency of our technique is also bounded by the same amount.

Furthermore, the bound can be mathematically derived based on the properties of a given DNN workload. As shown in Algorithm 12, with a probability higher than $1-3\times10^{-89}$, the loss value will not increase by more than $1/(m \times ln10) \times 20 \times \sqrt{2}$ (where m is the batch size) across two consecutive training iterations in the absence of hardware failures (this value is 0.095 for Resnet18, 0.19 for Yolov3, and 0.015 for Transformer). Using this bound, all faults that cause the SharpDegrade outcome in our FI experiments are successfully detected.

Algorithm 12: Bounding the Increase in Training Loss Value.

Without loss of generality, assume the following DNN properties [86, 87]:

(1)The mean of the outputs (before activation) and inputs of every DNN layer is 0, and the variance is approximately equal to that of the previous layer.

(2)The input data-set is normalized with a mean of 0 and variance of 1.

(3)Softmax-cross-entropy is used as the loss function, and Adam is used as the optimizer.

In the tth training iteration, let $a_{i,t}$ and $p_{i,t} \coloneqq$ the ith inputs and outputs of the softmax layer, $1 \leq i \leq I$; $y_{i,t} \coloneqq$ the ith one-hot encoded training target; and $L_t \coloneqq$ the loss value. Let $m \coloneqq$ num. mini-batches.

$p_{i,t} = \frac{e^{a_{i,t}}}{\sum_{k=1}^{I} e^{a_{k,t}}}$, $L_t = -\sum_i y_{i,t} log(p_{i,t})/m$.

Since one-hot encoding is used, there exists only one neuron with the label 1, while the rest are all labeled with 0. We use s and s' to represent the index of the neuron labeled with 1 in iterations t and $t+1$, respectively.

$|L_{t+1} - L_t| = \frac{1}{m} \times |log(e^{a_{s',t+1}}) - log(\sum_k e^{a_{k,t+1}}) - log(e^{a_{s,t}}) + log(\sum_k e^{a_{k,t}})|$

Given properties 1, 2 and 3,

$log(\sum_k e^{a_{k,t+1}}) \approx log(\sum_k e^{a_{k,t}})$

$\therefore \quad |L_{t+1} - L_t| \approx -\frac{1}{m} \times |log(e^{a_{s',t+1}}) - log(e^{a_{s,t}})| = \frac{1}{m \; ln10} \times |a_{s',t+1} - a_{s,t}|$.

$\because \quad a_{s',t+1}$ and $a_{s,t} \sim \mathcal{N}(0,1)$, and are independent, $a_{s',t+1} - a_{s,t} \sim \mathcal{N}(0,2)$.

$\therefore \; Prob(|L_{t+1} - L_t| > \frac{1}{m \; ln10} \times 20 \times \sqrt{2}) < 3 \times 10^{-89}$.

Implementation and Evaluation. We implemented the detection and re-execution techniques in Tensorflow for all workloads in Table 5.2, which involve $15-25$ lines of code change. We evaluated these techniques on Google Cloud TPUs, using the TPU profiler [88]

to obtain performance, power, and memory usage results. We executed each workload $10K$ times to obtain the average run time.

Our result shows that our techniques introduce $0.004\% - 0.025\%$ performance overhead on average (geomean) across the workloads. Memory and power costs are negligible, since the utilization of TPU resources when running the modified workloads is similar to that of the original workloads. Comparing our re-execution technique with the conventional check-pointing approach, the performance/energy cost of our technique is 200x lower. Our results align with those presented in Chapter 4 as the checking for all necessary conditions incurs very small performance overhead. Additionally, we employed the same replay technique as described in Chapter 4.

CHAPTER 6

CONCLUSION

In this thesis, we first present a light-weight and efficient functional in-field self-test technique targeting permanent hardware failures in DNN inference accelerators. We then present resilience analysis frameworks that accurately model both transient and permanent hardware failure behaviors in software, which cover both DNN inference and training accelerators. Based on these frameworks, we perform the first resilience study on transient failures in DNN inference accelerators, and the first in-depth resilience study on both transient and permanent hardware failures in DNN training accelerators. Our in-depth studies inspire light-weight, effective techniques that can be used to detect hardware failures and recover DNN training workloads.

Our studies are essential as hardware failures are pressing issues in large-scale datacenters nowadays. As the DNN systems continue to scale and the complexity of DNNs continues to grow, hardware failures are expected to generate more undesirable outcomes, and the costs to mitigate the undesirable consequences brought by hardware failures are expected to grow.

For future work, we plan to extend our work to cover other types of hardware failures, extend our analysis frameworks to include other fault models, and also cover other devices such as GPUs.

REFERENCES

[1] R. Bonderson, “Training in turmoil: Silent data corruption in systems at scale.” International Test Conference Silicon Lifecycle Management Workshop, Oct. 2021.

[2] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, “Cores that don’t count,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 9–16, 2021.

[3] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, “Silent data corruptions at scale,” *arXiv preprint arXiv:2102.11245*, 2021.

[4] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, “Detecting silent data corruptions in the wild,” 2022.

[5] J. Markoff, “Tiny chips, big headaches,” 2022.

[6] S. Sankar, R. Govindaraju, A. V. D. Ven, S. Hesley, and S. Mitra, “Panel: Hardware operation at scale reliability to address silent data corruptions,” Nov. 2021.

[7] V. Devanathan, V. Srinivas, and J. Prasad, “Tester-on-chip: An in-field system-test interface for heterogeneous ips,” in *International Workshop on Automotive Reliability and Test (ART)*, IEEE, 2018.

[8] N. Semiconductors, “Using the built-in self-test (bist) on the mpc5744p,”

[9] P. K. D. Jagannadha, M. Yilmaz, M. Sonawane, S. Chadalavada, S. Sarangi, B. Bhaskaran, S. Bajpai, V. A. Reddy, J. Pandey, and S. Jiang, “Special session: in-system-test (ist) architecture for nvidia drive-agx platforms,” in *2019 IEEE 37th VLSI Test Symposium (VTS)*, pp. 1–8, IEEE, 2019.

[10] H. Cho *et al.*, “Quantitative evaluation of soft error injection techniques for robust system design,” in *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–10, 2013.

[11] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, “Clear: Cross-layer exploration for architecting resilience - combining hardware and software techniques to tolerate soft errors in processor cores,” in *Proceedings of the 53rd Annual Design Automation Conference*, DAC ’16, (New York, NY, USA), Association for Computing Machinery, 2016.

[12] B. Reagen *et al.*, “Ares: A framework for quantifying the resilience of deep neural networks,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2018.

[13] E. Cheng, *Cross-layer resilience to tolerate hardware errors in digital systems*. PhD thesis, Stanford University, 2018.

[14] S. Mirkhani, S. Mitra, C. Cher, and J. Abraham, "Efficient soft error vulnerability estimation of complex designs," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 103–108, March 2015.

[15] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 241–254, June 2017.

[16] S. K. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Application resiliency analyzer for transient faults," *IEEE Micro*, vol. 33, pp. 58–66, May 2013.

[17] G. Li *et al.*, "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*, SC '17, pp. 8:1–8:12, 2017.

[18] G. Li *et al.*, "Tensorfi: A configurable fault injector for tensorflow applications," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 313–320, Oct 2018.

[19] Z. Zhang, L. Huang, R. Huang, W. Xu, and D. S. Katz, "Quantifying the impact of memory errors in deep learning," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–12, 2019.

[20] Y. He, T. Uezono, and Y. Li, "Efficient functional in-field self-test for deep learning accelerators," in *2021 IEEE International Test Conference (ITC)*, pp. 93–102, IEEE, 2021.

[21] T. Uezono, Y. He, and Y. Li, "Achieving automotive safety requirements through functional in-field self-test for deep learning accelerators," in *2022 IEEE International Test Conference (ITC)*, IEEE, 2022.

[22] S. Keckler, "High performance computing in a world of embedded intelligence." `https://www.esweek.org/sites/default/files/ES_Week_101419_Keckler.pdf`, 2019.

[23] G. Tshagharyan, G. Harutyunyan, and Y. Zorian, "An effective functional safety solution for automotive systems-on-chip," in *2017 IEEE International Test Conference (ITC)*, pp. 1–10, 2017.

[24] S. Mitra, P. Bose, E. Cheng, C.-Y. Cher, H. Cho, R. Joshi, Y. M. Kim, C. R. Lefurgy, Y. Li, K. P. Rodbell, K. Skadron, J. Stathis, and L. Szafaryn, "The resilience wall: Cross-layer solution strategies," in *Proceedings of Technical Program - 2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, pp. 1–11, 2014.

[25] Y. Li, S. Makar, and S. Mitra, "Casp: Concurrent autonomous chip self-test using stored test patterns," in *2008 Design, Automation and Test in Europe*, pp. 885–890, 2008.

[26] Y. Li, O. Mutlu, D. S. Gardner, and S. Mitra, "Concurrent autonomous self-test for uncore components in system-on-chips," in *2010 28th VLSI Test Symposium (VTS)*, pp. 232–237, 2010.

[27] S. Otani, N. Otsuki, Y. Suzuki, N. Okumura, S. Maeda, T. Yanagita, T. Koike, Y. Shimazaki, M. Ito, M. Uemura, T. Hattori, T. Yamauchi, and H. Kondo, "2.7 a 28nm 600mhz automotive flash microcontroller with virtualization-assisted processor for next-generation automotive architecture complying with iso26262 asil-d," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 54–56, 2019.

[28] N. Mukherjee, D. Tille, M. Sapati, Y. Liu, J. Mayer, S. Milewski, E. Moghaddam, J. Rajski, J. Solecki, and J. Tyszer, "Test time and area optimized brst scheme for automotive ics," in *2019 IEEE International Test Conference (ITC)*, pp. 1–10, 2019.

[29] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, vol. abs/1703.09039, 2017.

[30] S. Borkar, "Thousand core chips: A technology perspective," in *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, (New York, NY, USA), p. 746–749, Association for Computing Machinery, 2007.

[31] J. Hicks, D. Bergstrom, M. Hattendorf, J. Jopling, J. Maiz, S. Pae, C. Prasad, and J. Wiedemer, "45nm transistor reliability.," *Intel Technology Journal*, vol. 12, no. 2, 2008.

[32] S. R. Nassif, V. B. Kleeberger, and U. Schlichtmann, "Goldilocks failures: Not too soft, not too hard," in *2012 IEEE international reliability physics symposium (IRPS)*, pp. 2F–1, IEEE, 2012.

[33] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *CoRR*, vol. abs/1704.04760, 2017.

[34] Y. He, T. Uezono, and Y. Li, "Efficient functional in-field self-test for deep learning accelerators," in *2021 IEEE International Test Conference (ITC)*, pp. 93–102, 2021.

[35] C. Szegedy *et al.*, "Going deeper with convolutions," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[36] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[37] NVDLA, "Nvdla opensourced performance." `https://github.com/nvdla/hw/tree/nvdlav1/perf`, 2018.

[38] Y. He, P. Balaprakash, and Y. Li, "Fidelity: Efficient resilience analysis framework for deep learning accelerators," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 270–281, 2020.

[39] NVIDIA Corporation, "Nvdla open source project." `http://nvdla.org/primer.html`, 2018.

[40] Tesla, "How does tesla's new self-driving ai chip match up to competitors?." `https://www.electronicspoint.com/opinion/how-does-teslas-new-self-driving-ai-chip-match-up-to-competitors/`, 2019.

[41] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, June 2016.

[42] N. Chandramoorthy *et al.*, "Resilient low voltage accelerators for high energy efficiency," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 147–158, 2019.

[43] P. N. Whatmough *et al.*, "14.3 a 28nm soc with a 1.2ghz 568nj/prediction sparse deep-neural-network engine with >0.1 timing error rate tolerance for iot applications," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 242–243, Feb 2017.

[44] S. Jagannathan, T. D. Loveless, B. L. Bhuva, N. J. Gaspard, N. Mahatme, T. Assis, S. J. Wen, R. Wong, and L. W. Massengill, "Frequency dependence of alpha-particle induced soft error rates of flip-flops in 40-nm cmos technology," *IEEE Trans. Nucl. Sci.*, vol. 59, no. 6, pp. 2796–2802, 2012.

[45] International Organization for Standardization, "Iso 26262-1:2018 road vehicles – functional safety." `https://www.iso.org/standard/68383.html`, 2018.

[46] WikiChip, "Full self-driving chip - tesla." `https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip`, 2019.

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

[48] N. J. J. Wang, J. Quek, T. M. M. Rafacz, and S. J. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks*, pp. 61–70, 2004.

[49] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical fault injection," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 122–127, 2008.

[50] M. Naumov, J. Kim, D. Mudigere, S. Sridharan, X. Wang, W. Zhao, S. Yilmaz, C. Kim, H. Yuen, M. Ozdal, K. Nair, I. Gao, B. Su, J. Yang, and M. Smelyanskiy, "Deep learning training in facebook data centers: Design of scale-up and scale-out systems," *CoRR*, vol. abs/2003.09518, 2020.

[51] P. K. Gupta, “Accelerating datacenter workloads,” in *26th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2017, p. 20, 2016.

[52] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629, 2018.

[53] Google, “Cloud tpu.” `https://cloud.google.com/tpu`, 2021.

[54] Y. He, “Fault injection framework.” `https://github.com/YLab-UChicago/TrainingFI.git`, 2023.

[55] M. Zinkevich, M. Weimer, L. Li, and A. Smola, “Parallelized stochastic gradient descent,” in *Advances in Neural Information Processing Systems* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), vol. 23, Curran Associates, Inc., 2010.

[56] G. Papadimitriou and D. Gizopoulos, “Demystifying the system vulnerability stack: Transient fault effects across the layers,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 902–915, 2021.

[57] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, *et al.*, “Ten lessons from three generations shaped google’s tpuv4i: Industrial product,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, IEEE, 2021.

[58] Nvidia, “Nvidia ampere architecture.” `https://www.nvidia.com/en-us/data-center/ampere-architecture`, 2021.

[59] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, “Ai accelerator survey and trends,” in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, 2021.

[60] Y. Hu, Y. Liu, and Z. Liu, “A survey on convolutional neural network accelerators: Gpu, fpga and asic,” in *2022 14th International Conference on Computer Research and Development (ICCRD)*, pp. 100–107, 2022.

[61] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, “A survey of accelerator architectures for deep neural networks,” *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[62] MLCommons, “v1.0 results.” `https://mlcommons.org/en/training-normal-10/`, 2021.

[63] Wikipedia, “Backpropagation.” `https://en.wikipedia.org/wiki/Backpropagation`, 2022.

[64] Google, “Tensorflow.” `https://www.tensorflow.org`, 2019.

[65] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "Qed: Quick error detection tests for effective post-silicon validation," in *2010 IEEE International Test Conference*, pp. 1–10, IEEE, 2010.

[66] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 77–88, Aug 2013.

[67] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pp. 123–134, 2012.

[68] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, 2016.

[69] E. Singh, C. Barrett, and S. Mitra, "E-qed: electrical bug localization during post-silicon validation enabled by quick error detection and formal methods," in *International Conference on Computer Aided Verification*, pp. 104–125, Springer, 2017.

[70] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech, "Demystifying soft error assessment strategies on arm cpus: Microarchitectural fault injection vs. neutron beam experiments," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 26–38, 2019.

[71] Y. Ibrahim, H. Wang, J. Liu, J. Wei, L. Chen, P. Rech, K. Adam, and G. Guo, "Soft errors in dnn accelerators: A comprehensive review," *Microelectronics Reliability*, vol. 115, p. 113969, 2020.

[72] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[73] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[74] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.

[75] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[76] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[77] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018.

[78] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," 2020.

[79] A. Brock, S. De, S. L. Smith, and K. Simonyan, "High-performance large-scale image recognition without normalization," *CoRR*, vol. abs/2102.06171, 2021.

[80] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results." http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[81] T. Huynh, M. Maire, and M. R. Walter, "Multigrid neural memory," *CoRR*, vol. abs/1906.05948, 2019.

[82] O. Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C. Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, and A. Tamchyna, "Findings of the 2014 workshop on statistical machine translation," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*, (Baltimore, Maryland, USA), pp. 12–58, Association for Computational Linguistics, June 2014.

[83] A. Mahmoud, S. K. Sastry Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing selective protection for cnn resilience," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 127–138, 2021.

[84] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[85] R. M. Schmidt, F. Schneider, and P. Hennig, "Descending through a crowded valley-benchmarking deep learning optimizers," in *International Conference on Machine Learning*, pp. 9367–9376, PMLR, 2021.

[86] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.

[87] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[88] Google, "Profile your model with cloud tpu tools." `https://cloud.google.com/tpu/docs/cloud-tpu-tools`, 2021.

[89] Tensorflow, "Training checkpoints." `https://www.tensorflow.org/guide/checkpoint`, 2021.

[90] C. Schorn, A. Guntoro, and G. Ascheid, "An efficient bit-flip resilience optimization method for deep neural networks," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1507–1512, 2019.

[91] C. Schorn, A. Guntoro, and G. Ascheid, "Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 979–984, 2018.

[92] E. Ozen and A. Orailoglu, "Just say zero: Containing critical bit-error propagation in deep neural networks with anomalous feature suppression," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2020.

[93] E. Ozen and A. Orailoglu, "Boosting bit-error resilience of dnn accelerators through median feature selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3250–3262, 2020.

[94] K. Adam, I. I. Mohamed, and Y. Ibrahim, "A selective mitigation technique of soft errors for dnn models used in healthcare applications: Densenet201 case study," *IEEE Access*, vol. 9, pp. 65803–65823, 2021.

[95] S. Mittal, "A survey on modeling and improving reliability of dnn algorithms and accelerators," *Journal of Systems Architecture*, vol. 104, p. 101689, 2020.

[96] K. Adam, I. I Mohd, and Y. Ibrahim, "Analyzing the resilience of convolutional neural networks implemented on gpus: Alexnet as a case study," *International journal of electrical and computer engineering systems*, vol. 12, no. 2, pp. 91–103, 2021.

[97] B. F. Goldstein, V. C. Ferreira, S. Srinivasan, D. Das, A. S. Nery, S. Kundu, and F. M. G. França, "A lightweight error-resiliency mechanism for deep neural networks," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pp. 311–316, 2021.

[98] Z. Chen, G. Li, and K. Pattabiraman, "A low-cost fault corrector for deep neural networks through range restriction," 2020.

[99] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "Algorithm-based fault tolerance for convolutional neural networks," *IEEE Transactions on Parallel and Distributed Systems*, p. 1–1, 2021.

[100] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2546–2558, 2022.

[101] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, pp. 1–12, 2004.

[102] D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 33, no. 10, pp. 1573–1590, 2014.

[103] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in spuper-scalar processor," *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 63–75, 2002.

[104] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 150–157, 2011.

[105] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 385–396, 2010.

[106] M. N. Lovellette, K. S. Wood, D. L. Wood, J. H. Beall, P. P. Shirvani, N. Oh, and E. J. McCluskey, "Strategies for fault-tolerant, space-based computing: Lessons learned from the argos testbed," in *IEEE Aerospace Conference Proceedings*, vol. 5, pp. 2109–2119, 2002.

[107] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 111–122, 2002.

[108] S. K. Sahoo, M. L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 70–79, 2008.

[109] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 842–854, 2018.

[110] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII*, vol. 42, p. 265, 2008.

[111] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 210–222, 2007.

[112] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 169–180, 2007.

[113] N. J. Wang and S. J. Patel, "Restore: Symptom-based soft error detection in microprocessors," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 3, pp. 188–201, 2006.

[114] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 1–12, 2012.

[115] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference on – DAC13*, 2013.

[116] Y. He and Y. Li, "Time-slicing soft error resilience in microprocessors for reliable and energy-efficient execution," in *2019 IEEE International Test Conference (ITC)*, pp. 1–10, 2019.

[117] D. Blaauw, S. Kalaiselvan, K. Lai, W.-H. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull, "Razor ii: In situ error detection and correction for pvt and ser tolerance," in *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*, pp. 400–622, IEEE, 2008.

[118] Z. Chen, G. Li, and K. Pattabiraman, "Ranger: Boosting error resilience of deep neural networks through range restriction," *ArXiv*, vol. abs/2003.13874, 2020.

[119] L. H. Hoang, M. A. Hanif, and M. Shafique, "Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation," *CoRR*, vol. abs/1912.00941, 2019.

[120] J. Zhang, T. He, S. Sra, and A. Jadbabaie, "Why gradient clipping accelerates training: A theoretical justification for adaptivity," 2019.

[121] X. Chen, S. Z. Wu, and M. Hong, "Understanding gradient clipping in private sgd: A geometric perspective," *Advances in Neural Information Processing Systems*, vol. 33, pp. 13773–13782, 2020.

[122] A. K. Menon, A. S. Rawat, S. J. Reddi, and S. Kumar, "Can gradient clipping mitigate label noise?," in *International Conference on Learning Representations*, 2019.

[123] 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)*, pp. 1–6, IEEE, 2018.

[124] M. Sadi and U. Guin, "Test and yield loss reduction of ai and deep learning accelerators," *IEEE Trans. CAD*, 2022.

[125] A. Chaudhuri *et al.*, "Functional criticality analysis of structural faults in ai accelerators," *IEEE Trans. CAD*, 2022.

[126] M. Hanif and M. Shafique, "Salvagednn: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, p. 20190164, 02 2020.

[127] B. Zhang, N. Uysal, D. Fan, and R. Ewetz, "Handling stuck-at-fault defects using matrix transformation for robust inference of dnns," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2448–2460, 2019.

[128] S. Chakravarty *et al.*, "Tanatomy of an in-die tester for infield testing," in *IEEE International ART Workshop*, 2019.

[129] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[130] 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)*, pp. 1–6, 2019.

[131] M. Sadi and U. Guin, "Test and yield loss reduction of ai and deep learning accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 104–115, 2021.

[132] D. M. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[133] Tensorflow, "Batchnormalization layer." `https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization`, 2021.

[134] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (New York, NY, USA), p. 129–140, Association for Computing Machinery, 2018.

[135] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[136] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 510–520, Association for Computing Machinery, 2019.

[137] D. Adiwardana, M.-T. Luong, D. R. So, J. Hall, N. Fiedel, R. Thoppilan, Z. Yang, A. Kulshreshtha, G. Nemade, Y. Lu, and Q. V. Le, "Towards a human-like open-domain chatbot," 2020.

[138] Google, "Training checkpoints." https://www.tensorflow.org/guide/checkpoint, 2021.

[139] C. Li, "Openai's gpt-3 language model: A technical overview." https://lambdalabs.com/blog/demystifying-gpt-3/, 2020.

[140] Wikipedia, "Batch normalization." https://en.wikipedia.org/wiki/Batch_normalization, 2021.

[141] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," 2021.

[142] "Imagenet." https://www.image-net.org/, 2021.

[143] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," 2016.

[144] Xilinx, "Device reliability report," 2022.

[145] H. Zhang, H. Jiang, T. R. Assis, D. R. Ball, K. Ni, J. S. Kauppila, R. D. Schrimpf, L. Massengill, B. L. Bhuva, B. Narasimham, S. Hatami, A. Anvar, A. Lin, and J. K. Wang, "Temperature dependence of soft-error rates for ff designs in 20-nm bulk planar and 16-nm bulk finfet technologies," in *2016 IEEE International Reliability Physics Symposium (IRPS)*, pp. 5C–3–1–5C–3–5, 2016.

[146] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[147] S. Mitra, M. Zhang, N. Seifert, T. Mak, and K. S. Kim, "Built-in soft error resilience for robust system design," in *2007 IEEE International Conference on Integrated Circuit Design and Technology*, pp. 1–6, 2007.

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

[149] X. She, N. Li, and D. W. Jensen, "Seu tolerant memory using error correction code," *IEEE Transactions on Nuclear Science*, vol. 59, no. 1, pp. 205–210, 2012.

[150] Y. Li, Y. M. Kim, E. Mintarno, D. S. Gardner, and S. Mitra, "Overcoming early-life failure and aging for robust systems," *IEEE Design Test of Computers*, vol. 26, no. 6, pp. 28–39, 2009.

[151] L. Lamport, *LaTeX: A Document Preparation System.* Reading, Massachusetts: Addison-Wesley, 2nd ed., 1994.

[152] F. Lastname1 and F. Lastname2, "A very nice paper to cite," in *Proceedings of the 33rd Annual ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[153] F. Lastname1, F. Lastname2, and F. Lastname3, "Another very nice paper to cite," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2011.

[154] F. Lastname1, F. Lastname2, F. Lastname3, F. Lastname4, and F. Lastname5, "Yet another very nice paper to cite, with many author names all spelled out," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[155] Z. Liang *et al.*, "Deep learning for healthcare decision making with emrs," in *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 556–559, Nov 2014.

[156] F. Calivá *et al.*, "A deep learning approach to anomaly detection in nuclear reactors," in *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, July 2018.

[157] J. Huang *et al.*, "Speed/accuracy trade-offs for modern convolutional object detectors," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[158] B. Gill *et al.*, "Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node," in *IEEE International Reliability Physics Symposium Proceedings*, pp. 199–205, 2009.

[159] P. Nsengiyumva *et al.*, "A comparison of the seu response of planar and finfet d flip-flops at advanced technology nodes," *IEEE Trans. Nucl. Sci.*, vol. 63, no. 1, pp. 266–272, 2016.

[160] N. Seifert *et al.*, "Soft error susceptibilities of 22nm tri-gate devices," *IEEE Trans. Nucl. Sci.*, vol. 59, no. 6, pp. 2666–2673, 2012.

[161] T. M. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 196–207, 1999.

[162] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, p. 72, 2008.

[163] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R.

Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1, 2011.

[164] A. Biswas, N. Soundararajan, S. Mukherjee, and S. Gurumurthi, "Quantized avf: A means of capturing vulnerability variations over small windows of time," in *Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2009.

[165] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithmic based fault tolerance applied to high performance computing," *J. Parallel Distrib. Comput*, vol. 69, no. 4, pp. 410–416, 2008.

[166] K. A. Bowman, J. W. Tschanz, S. L. L. Lu, P. A. Aseron, M. M. Khellah, A. Raychowdhury, B. M. Geuskens, C. Tokunaga, C. B. Wilkerson, T. Karnik, and V. K. De, "A 45 nm resilient microprocessor core for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 194–208, 2011.

[167] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron cmos technology," *Nucl. Sci. IEEE Trans.*, vol. 43, no. 6, pp. 2874–2878, 1996.

[168] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. González, "End-to-end register data-flow continuous self-test," in *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '09)*, 2009.

[169] Z. Chen and J. Dongarra, "Numerically stable real-number codes based on random matrices," *Itw2004*, pp. 24–29, 2004.

[170] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw, "Razorii: In situ error detection and correction for pvt and ser tolerance," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 32–48, 2009.

[171] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. M. Harris, D. Blaauw, and D. Sylvester, "Bubble razor: Eliminating timing margins in an arm cortex-m3 processor in 45 nm cmos using architecturally independent error detection and correction," *IEEE J. Solid-State Circuits*, vol. 48, no. 1, pp. 66–81, 2013.

[172] J. Furuta, C. Hamanaka, K. Kobayashi, and H. Onodera, "A 65nm bistable cross-coupled dual modular redundancy flip-flop capable of protecting soft errors on the c-element," in *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, pp. 123–124, 2010.

[173] J. L. Henning and J. L., "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[174] H. K. Lee, K. Lilja, M. Bounasser, P. Relangi, I. R. Linscott, U. S. Inan, and S. Mitra, "Design of a sequential logic cell using leap: Layout design through error-aware transistor positioning," in *Reliability Physics Symposium (IRPS), 2010 IEEE International*, pp. 1–6, 2010.

[175] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modelling framework for multicore and manycore architectures," in *42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2009)*, pp. 469–480, 2009.

[176] D. J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 681–685, 1982.

[177] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05*, vol. 40, no. 6, p. 190, 2005.

[178] N. N. Mahatme, N. J. Gaspard, S. Jagannathan, T. D. Loveless, B. L. Bhuva, W. H. Robinson, L. W. Massengill, S.-J. Wen, and R. Wong, "Impact of supply voltage and frequency on the soft error rate of logic circuits," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 6, pp. 4200–4206, 2013.

[179] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower, "Ibm z990 soft error detection and recovery," *IEEE Trans. Device Mater. Reliab.*, vol. 5, no. 3, pp. 419–427, 2005.

[180] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, p. 99, 2002.

[181] S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, pp. 243–247, 2005.

[182] N. Xiang and L. V. Kale, "Flipback: automatic targeted protection against silent data corruption," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, 2016.

[183] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech, "Experimental and analytical study of xeon phi reliability," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*, 2017.

[184] D. Parikh, K. Skadron, Y. Zhang, and M. Stan, "Power-aware branch prediction: Characterization and design," *IEEE Trans. Comput.*, vol. 53, no. 2, pp. 168–186, 2004.

[185] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 1, pp. 44–57, 2011.

[186] R. Pawlowski, J. Crop, M. Cho, J. Tschanz, V. De, T. Fairbanks, H. Quinn, S. Borkar, P. Y. Chiang, and A. R. Work, "Characterization of radiation-induced sram and logic soft errors from 0 . 33v to 1 . 0v in 65nm cmos," in *Custom Integrated Circuits Conference (CICC), 2014 IEEE Proceedings of the*, pp. 33–36, 2014.

[187] P. Zhao, J. McNeely, W. Kuang, N. Wang, and Z. Wang, "Design of sequential elements for low power clocking system," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 5, pp. 914–918, 2010.

[188] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 319–330, 2015.

[189] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," in *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)*, 2007.

[190] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ace analysis reliability estimates using fault-injection," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 460–469, 2007.

[191] G. An, "The effects of adding noise during backpropagation training on a generalization performance," *Neural computation*, vol. 8, no. 3, pp. 643–674, 1996.

[192] C. M. Bishop, "Training with noise is equivalent to tikhonov regularization," *Neural computation*, vol. 7, no. 1, pp. 108–116, 1995.

[193] L. Holmstrom and P. Koistinen, "Using additive noise in back-propagation training," *IEEE transactions on neural networks*, vol. 3, no. 1, pp. 24–38, 1992.

[194] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Characterizing the impact of intermittent hardware faults on programs," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 297–310, 2015.

[195] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," in *2008 Annual Reliability and Maintainability Symposium*, pp. 370–374, 2008.

[196] D. C. Bossen, "Cmos soft errors and server design," *2002 IRPS Tutorial Notes - Reliability Fundamentals*, 2002.

[197] Y. He and Y. Li, "Time-slicing soft error resilience in microprocessors for reliable and energy-efficient execution," in *2019 IEEE International Test Conference (ITC)*, pp. 1–10, IEEE, 2019.

[198] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 117–128, 2009.

[199] N. Seifert *et al.*, "Soft error rate improvements in 14-nm technology featuring second-generation 3d tri-gate transistors," *IEEE Trans. Nucl. Sci.*, vol. 62, no. 6, pp. 2570–2577, 2015.

[200] I. Pomeranz, "Skewed-load tests for transition and stuck-at faults," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 10, pp. 1969–1973, 2018.

[201] A. Krizhevsky, "Convolutional deep belief networks on cifar-10," 2010.

[202] S. S. Liew *et al.*, "Bounded activation functions for enhanced training stability of deep neural networks on visual pattern recognition problems," *Neurocomput.*, vol. 216, pp. 718–734, 2016.

[203] Pytorch, "Pytorch." `https://pytorch.org`, 2019.

[204] E. Cheng *et al.*, "Cross-layer resilience: Challenges, insights, and the road ahead," in *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, 2019.

[205] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *CoRR*, vol. abs/1905.11946, 2019.

[206] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016.

[207] J. Deng *et al.*, "Retraining-based timing error mitigation for hardware neural networks," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 593–596, March 2015.

[208] M. Zhang *et al.*, "Sequential element design with built-in soft error resilience," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 1368–1378, Dec 2006.

[209] L. Yang and B. Murmann, "Sram voltage scaling for energy-efficient convolutional neural networks," in *2017 18th International Symposium on Quality Electronic Design (ISQED)*, pp. 7–12, March 2017.

[210] D. L. Tao and C. R. P. Hartmann, "A novel concurrent error detection scheme for fft networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 198–221, Feb 1993.

[211] H. Cho *et al.*, "System-level effects of soft errors in uncore components," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, pp. 1497–1510, Sep. 2017.

[212] P. Molchanov *et al.*, "Pruning convolutional neural networks for resource efficient transfer learning," *CoRR*, vol. abs/1611.06440, 2016.

[213] H. Hu *et al.*, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *CoRR*, vol. abs/1607.03250, 2016.

[214] N. a. o. Srivastava, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, Jan. 2014.

[215] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[216] Z. Chen and J. Dongarra, "Numerically stable real number codes based on random matrices," in *Proceedings of the 5th International Conference on Computational Science - Volume Part I*, ICCS'05, 2005.

[217] Y.-H. Chen *et al.*, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pp. 367–379, 2016.

[218] S. Han *et al.*, "Eie: Efficient inference engine on compressed deep neural network," *SIGARCH Comput. Archit. News*, vol. 44, pp. 243–254, June 2016.

[219] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, (Stroudsburg, PA, USA), pp. 311–318, Association for Computational Linguistics, 2002.

[220] M. Cettolo, J. Niehues, S. Stuker, L. Bentivogli, R. Cattoni, and M. Federico, "The iwslt 2016 evaluation campaign," in *The International Workshop on Spoken Language Translation*, IWSLT '16, 2016.

[221] Y. Zhao, X. Hu, S. Li, J. Ye, L. Deng, Y. Ji, J. Xu, D. Wu, and Y. Xie, "Memory trojan attack on neural network accelerators," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1415–1420, March 2019.

[222] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 18–37, May 2016.

[223] J. G. J. van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 91–99, Sep. 2011.

[224] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, "Fault attacks on rsa with crt: Concrete results and practical countermeasures," vol. 2523, pp. 260–275, 08 2002.

[225] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, WOST'99, (USA), p. 2, USENIX Association, 1999.

[226] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S. L. Lu, T. Karnik, and V. K. De, "Energy-efficient and metastability-immune timing-error detection and instruction-replay-based recovery circuits for dynamic-variation tolerance," in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pp. 402–623, 2008.

[227] J. Cao, L. Xu, B. L. Bhuva, S. . Wen, R. Wong, B. Narasimham, and L. W. Massengill, "Alpha particle soft-error rates for d-ff designs in 16-nm and 7-nm bulk finfet technologies," in *2019 IEEE International Reliability Physics Symposium (IRPS)*, pp. 1–5, 2019.