Resilience Assessment of Machine Learning Applications under Hardware Faults

by

Udit Kumar Agarwal

B.E., University of Delhi, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

August 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Resilience Assessment of Machine Learning Applications under Hardware Faults

submitted by **Udit Kumar Agarwal** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

Examining Committee:

Karthik Pattabiraman, Professor, Electrical and Computer Engineering, UBC
Supervisor

Prashant Nair, Assistant Professor, Electrical and Computer Engineering, UBC
Supervisory Committee Member

Aastha Mehta, Assistant Professor, Computer Science, UBC
Supervisory Committee Member

Abstract

Machine learning (ML) applications have been ubiquitously deployed across critical domains such as autonomous vehicles (AVs), and medical diagnosis. Vision-based ML models like ResNet are used for object classification and lane detection, while Large Language Models (LLMs) like ChatGPT are used in cars to enable robust and flexible voice commands in AVs. The use of ML models in safety-critical scenarios requires reliable ML models.

In the first part of this thesis, we primarily focus on understanding the resilience of ML models against transient hardware faults in CPUs. Towards this end, we present an LLVM IR-level FI tool, LLTFI, which we use to evaluate the effect of transient faults on Deep Neural Networks (DNNs) and LLMs. We found that LLTFI is more precise than TensorFI, an application-level FI tool proposed by prior work. Unlike LLTFI, TensorFI underestimates the resilience of DNNs by implicitly assuming that every injected fault corrupts the outputs of the intermediate layers of the DNN. Using LLTFI, we also evaluated the efficacy of Selective Instruction Duplication to make DNNs more resilient against transient faults. While in the case of DNNs, transient faults cause the model to misclassify or mispredict the object, for LLMs, we found transient faults to cause the model to produce semantically and syntactically incorrect outputs.

In the second part of this thesis, we evaluate the effect of permanent stuck-at faults in systolic arrays on DNNs. We present a Register Transfer (RTL)-Level FI tool, called SystoliFI, to inject permanent stuck-at faults in the systolic array, which we use to understand the manifestation of stuck-at faults in systolic arrays in the intermediate layers of the DNNs. We found that the manifestation of the stuck-

at faults varies significantly with the type of operation (Convolution vs. Matrix multiplication), the operation size, and the systolic array size.

Lay Summary

The use of Machine Learning (ML) in safety-critical systems is becoming more prevalent. However, failures in these systems, such as in self-driving cars, can lead to disastrous consequences. Therefore, it is crucial to ensure the resilience of ML applications. One popular method for assessing application resilience is Fault Injection (FI), where we intentionally inject faults in the application during runtime. In this thesis, we present a FI tool, LLTFI, to carry out FI in ML models, including Large Language Models. Using LLTFI, we evaluated the effect of transient hardware faults in the CPU on Deep Neural Networks like ResNet and Large Language Models like GPT. Moreover, we also present a low-level hardware FI tool, SystoliFI, to inject stuck-at hardware faults in ML models under the systolic array hardware model. We use this tool to understand the manifestation of stuck-at faults in systolic arrays in the intermediate layers of the Neural Networks.

Preface

This thesis is the result of work carried out by myself in collaboration with my supervisor, Prof. Karthik Pattabiraman, Abraham Chan, Ali Asgari, and our industry collaborators from Intel. All chapters are based on the work listed below. I am the lead author in all the publications except for the first and the last.

- Chan, Abraham, Udit Kumar Agarwal, and Karthik Pattabiraman. "(WIP) LLTFI: Low-level tensor fault injector." 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2021.

  Abraham implemented the first version of LLTFI. My contribution to this work was limited to helping with the performance evaluation of LLTFI. Karthik supervised the project.

- Agarwal, Udit Kumar, Abraham Chan, and Karthik Pattabiraman. "LLTFI: Framework Agnostic Fault Injection for Machine Learning Applications (Tools and Artifact Track)." 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2022. Acceptance rate 29%.

  I implemented the second version of LLTFI, including the machine learning fault tracing. I also added benchmarks and carried all experiments. Abraham and Karthik helped in forming research questions and writing the paper.

- Agarwal, Udit Kumar, Abraham Chan, Ali Asgari, and Karthik Pattabiraman. "Towards Reliability Assessment of Systolic Arrays against Stuck-at Faults" 2023 19th IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE). IEEE, 2023. Received Best-Of-SELSE award (one of three papers).

I implemented the fault injection tool. Ali helped me carry out the fault injection experiments and setting up the infrastructure. Abraham and Karthik supervised the project and helped in forming research questions.

- Agarwal, Udit Kumar, Abraham Chan, and Karthik Pattabiraman. "Resilience Assessment of Large Language Models under Transient Hardware Faults (PER)". 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2023. Acceptance rate 29.5%.

  I implemented the tool, added benchmarks, and carried out the experiments. Karthik and Abraham helped in forming research questions and writing the research paper.

- Alessio Netti, Yang Peng, Patrik Omland, Michael Paulitsch, Jorge Parra, Gustavo Espinosa, Udit Agarwal, Abraham Chan, Karthik Pattabiraman. "Mixed Precision Support in Hpc Applications: What About Reliability?." Journal of Parallel and Distributed Computing, 2023. 13 Pages. Impact factor 4.54.

  In this work, our collaborators from Intel used LLTFI for fault injection in HPC applications. My contribution to this work was limited to upgrading LLTFI to use LLVM's new pass manager to easily integrate LLTFI with HPC applications. I also helped with some bug fixes in LLTFI. Karthik and Abraham helped in forming research questions.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACD Arithmetic Chain Duplication

AID Arithmetic Instruction Duplication

ALU Arithmetic Logic Unit

API Application Programming Interface

AV Autonomous Vehicles

CPU Central Processing Unit

CNN Convolutional Neural Networks

DFG Data Flow Graph

DNN Deep Neural Networks

ECC Error Correction Codes

FC Fully Connected

FI Fault Injection

GEMM General Matrix Multiplication

IR Intermediate Representation

ISA Instruction Set Architecture

LLM Large Language Models

LLVM Low-Level Virtual Machine

MAC Multiplication And Accumulation

ML Machine Learning

NAN Not a Number

NLP Natural Language Processing

OD Operator Duplication

ONNX Open Neural Network Exchange

OS Output Stationary

RNN Recurrent Neural Networks

RTL Register-Transfer Language

RL Reinforcement Learning

SDC Silent Data Corruption

SID Selective Instruction Duplication

SWIFI Software Implemented Fault Injection

TPU Tensor Processing Unit

WS Weight Stationary

YAML Yet Another Markup Language

Acknowledgments

I want to express my deepest gratitude and appreciation to the following individuals who have contributed significantly to the completion of my master's thesis:

First and foremost, I am immensely grateful to my advisor, Karthik Pattabiraman, for his exceptional guidance, unwavering support, and invaluable expertise throughout my thesis journey. His insightful feedback, constructive criticism, and constant encouragement have shaped my research and pushed me to achieve my best.

I am also indebted to the members of my thesis committee, Dr. Prashant Nair and Dr. Aastha Mehta, anonymous conference reviewers, and my course instructors for their invaluable contributions, thought-provoking discussions, and valuable suggestions that have greatly enriched the quality of my thesis. Moreover, I would like to thank researchers at Huawei and Intel, with whom we collaborated on various parts of this work.

I want to thank my colleagues at the Dependable Systems Lab, specifically Pritam Dash and Abraham Chan, for their collaboration, support, and stimulating discussions. Their diverse perspectives and expertise have played a significant role in shaping my research ideas and refining my thesis.

Last but certainly not least, I want to express my most profound appreciation to my family and friends for their unconditional love, encouragement, and belief in my abilities. Their constant support, understanding, and sacrifices have been the bedrock of my academic pursuits.

Thank you all for being a part of this remarkable journey.

Chapter 1

Introduction

1.1 Motivation

The widespread deployment of ML applications in critical domains, including AVs and medical diagnosis, has become increasingly prevalent. Vision-based ML models are extensively used in AVs for tasks such as object detection, recognition, and classification. These models, such as CNNs like ResNet, enable the vehicle to perceive its surroundings by analyzing sensor data from cameras, lidar, and radar [34]. By leveraging ML algorithms, AVs can accurately identify and track objects like pedestrians, vehicles, traffic signs, and road markings. This perception capability is crucial for safe and reliable navigation. NLP models are also employed in AVs for tasks beyond vision. For example, Mercedes [5] and General Motors [4] recently announced the inclusion of LLMs like ChatGPT in their AVs to enable voice commands and natural language interactions between the vehicle and passengers. This allows for more intuitive and flexible communication, enhancing the user experience inside the AV.

The ISO 26262 standard for the functional safety of AVs requires that, for Automotive Safety Integrity Level D (ASIL-D), there should be no more than ten failures in time (including both transient and permanent hardware faults) in a billion hours of operation [85]. Therefore, there is a compelling need to evaluate and improve the resilience of ML models under hardware faults. Hardware faults can be permanent (e.g., stuck-at fault in the data paths) or transient (e.g., bit-flips). Per-

manent faults can be caused by various factors, including physical damage to the hardware, defects in the manufacturing process, and wear and tear over time [115]. Transient faults, on the other hand, are typically induced due to high energy particles such as neutrons and alpha particles striking the hardware [78].

FI is the traditional way to evaluate the resilience of a system against hardware faults [119]. FI can be implemented either at the hardware level or at the software level, called SWIFI. Hardware-level FI techniques like focused ion beams or laser injection and RTL-level FI offer precise modeling of hardware faults [86], but they are expensive and unscalable. For instance, RTL-level fault injection requires simulation or synthesis of the hardware's RTL model, which requires expensive computation resources. In contrast to hardware-level FI, SWiFI is inexpensive and provides reasonable precision [61] in modeling the effect of hardware faults at the software level.

Prior work on evaluating the resilience of ML models using SWiFI is limited to application-level FIs like TensorFI [69] and PyTorchFI [81], which inject faults in the intermediate layers of the DNNs. Existing application-level SWiFI tools are limited to specific ML frameworks and have limited visibility into the software stack. For instance, TensorFI works only with the TensorFlow framework [8] and can inject faults only in the outputs on the intermediate layers of the DNNs. To overcome these limitations, we propose LLTFI, a framework-agnostic SWiFI tool that injects faults in the instructions and registers of the ML model during runtime. Moreover, using LLTFI, we also evaluate the resilience of LLMs against transient hardware faults.

LLTFI, TensorFI, and many other SWiFI tools are restricted to injecting faults pertaining to only the CPU or the GPU hardware model. However, to speed up the training and inference of large DNNs and LLMs, DNN accelerators like Google TPU [3] are extensively used in practice. Despite their widespread use, the reliability of DNN accelerators against hardware faults has not been extensively evaluated. Most of the prior work on the reliability assessment of DNN accelerators only considers the effect of transient faults [52]. Moreover, the few prior work on evaluating the effect of permanent hardware faults in DNN accelerators is limited to evaluating the DNN's accuracy under the fault. However, it is not clear how these faults manifest at the intermediate layers of the DNNs. This is important

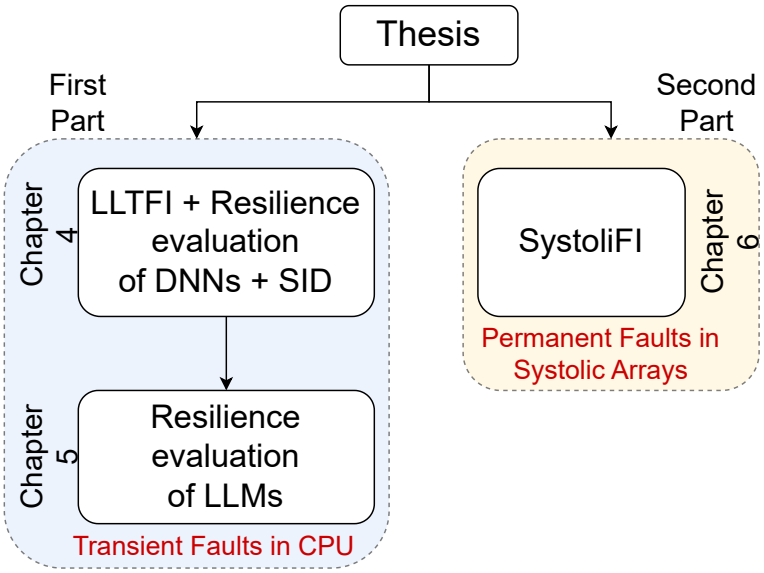


Figure 1.1: Organization of this thesis.

because understanding fault manifestation at the intermediate layers provides insights into building more resilient DNN architectures and improves the accuracy of application-level FI tools. In chapter 6, we bridge this gap by proposing a RTL-Level FI tool to inject permanent stuck-at faults in DNN accelerators, using which we evaluate the manifestation of injected faults on the intermediate layers of the DNNs.

1.2 Our Approach

In this thesis, we evaluated the resilience of ML models under two types of hardware faults: transient hardware faults in the CPU and permanent stuck-at faults in DNN accelerators. Figure 1.1 shows the overall organization of this thesis.

In the first part of this thesis, we focus on transient faults in CPUs. Since ML models are trained once and inferred multiple times, in this thesis, we focus on the inference-time resilience of ML models. Moreover, CPUs are still widely used for ML inference due to their lower total cost of ownership [7]. Therefore, we considered transient hardware faults in the CPU.

To explore the effect of transient hardware faults in the CPU, in Chapter 4, we propose a tool, LLTFI (Low-Level Tensor Fault Injector). LLTFI is agnostic to the ML framework and can work with TensorFlow, PyTorch, and other ML frameworks. LLTFI works by first converting ML models into the ONNX format [13], which is an open-source format designed to enable interoperability between different deep learning frameworks. Then, the ONNX file is converted into LLVM IR using the ONNX-MLIR [53] tool. After this, LLTFI modifies the LLVM IR to inject faults in instructions and registers. LLTFI's ability to inject faults in instructions and registers makes it more precise (empirically evaluated in Chapter 4) than application-level SWiFI tools like TensorFI [69]. Another advantage of LLTFI over other application-level SWiFI tools is Error Propagation Tracing. LLTFI provides two levels of error propagation tracing: instruction-level tracing and tensor operator-level tracing. While instruction-level tracing helps trace the effect of the injected fault on each LLVM IR instruction, tensor operator-level tracing works at a higher level and illustrates the effect of the injected fault on the output of each intermediate layer of the DNN. Since LLTFI injects faults in the instructions and registers of the ML model, we also used LLTFI to evaluate the effectiveness of SID in improving the resilience of the ML model. Moreover, we use LLTFI to evaluate the resilience of DNNs against transient hardware faults.

While in Chapter 4, we evaluated the resilience of DNNs, in Chapter 5, we used LLTFI to assess the resilience of LLMs. Unlike prior work [68, 69, 81, 97] on resilience assessment of ML models that has focused exclusively on DNNs, to the best of our knowledge, *we are the first ones to also evaluate the resilience of LLMs against transient hardware faults*. Unlike DNNs, LLMs have billions of parameters, and LLMs like CodeBert execute approximately 30 Billion CPU instructions in one inference. The sheer size of LLMs poses a scalability challenge to LLTFI. Therefore, to incorporate LLMs into LLTFI, we made two modifications. Firstly, we introduced an auxiliary FI runtime that links statically with the LLM executable, which has a minimal runtime overhead, enabling LLTFI to accommodate LLMs. Secondly, we enhanced LLTFI's fault propagation tracing to accommodate LLMs, which lets us delve into the root causes of the inaccuracies noted in our FI tests. We then employed this modified LLTFI version to conduct comprehensive FI campaigns to calculate the SDC rate, which is the proportion of undetected er-

roneous outputs of LLMs. In addition, we performed a qualitative examination of SDCs in LLMs by classifying them based on their syntactic or semantic correctness. Lastly, we assessed how the SDC rate fluctuates with FI in different LLM layers, bit positions, pre-training, and fine-tuning objectives.

In the second part of this thesis, we explore the effect of permanent stuck-at faults in DNN accelerators. Unlike for transient faults in the CPU, whose manifestation at the software-level is well-known (for example, bit flips in registers [78]), the effect of permanent faults in DNN accelerators at the software-level is not known. Therefore, we built an RTL-level FI tool called SystoliFI that expands upon Gemmini [39] - a widely used systolic array generator for assessing DNN accelerators - to implement FI in the MAC units of the DNN accelerator. Gemmini is an open-source, comprehensive DNN accelerator generator facilitating the complete implementation and assessment of custom hardware accelerators. Gemmini boasts high configurability, allowing adjustments to the systolic array size, data type, and data mapping scheme types. To implement SystoliFI, we extended Gemmini's custom RISC-V ISA to add an additional FI instruction and the required hardware support to inject stuck-at faults at runtime. This enhancement to Gemmini's ISA with additional FI instructions enabled us to control FI during runtime from the software side, eliminating the need to recompile the hardware's RTL model, a process that is computationally demanding. Using SystoliFI, we analyze the manifestation of stuck-at faults in DNN accelerators at intermediate layers of the DNN. We further evaluate how the fault manifestation varies with different hardware configurations (size of systolic array, position of fault, data mapping scheme) and software configurations (type and size of operations).

1.3 Contributions

To summarize, we made the following contributions in this thesis:

Chapter 4 of this thesis focuses on the LLTFI tool set where we:

1. Propose and implement a framework-agnostic, LLVM IR-level FI tool called LLTFI to inject transient hardware faults in ML applications. We then compared LLTFI with TensorFI, an existing application-level SWiFI tool, by

evaluating the SDC rates of eight DNNs with LLTFI and contrasting them with the SDC rate reported by TensorFI.

2. Used LLTFI to evaluate the efficacy of Selective Instruction Duplication for enhancing the resilience of DNNs against transient faults.

In Chapter 5 of this thesis, we evaluated LLM's resilience using LLTFI. Precisely, we made the following contributions:

3. We modified LLTFI to integrate it with LLMs, and then we used it to carry out a quantitative and qualitative evaluation of LLM's resilience against transient hardware faults.

4. We evaluate the variance in LLM's resilience due to FI in different layers, bit positions, model sizes, pre-training, and fine-tuning objectives.

Chapter 6 of this thesis focuses on the SystoliFI tool set where we:

5. Propose and implement an RTL-level FI tool, SystoliFI, for injecting stuck-at faults in DNN accelerators. We then used SystoliFI to understand the manifestation of stuck-at faults at the intermediate layers of the DNNs. We also evaluate how the fault manifestation varies with different hardware and software configurations.

After extensive FI experiments, We made the following observations:

1. We found that there are considerable differences between the SDC rates (i.e., the fraction of undetected, incorrect outputs) of the two tools for the same faults, *with LLTFI exhibiting significantly lower SDC rates (average: 3.4X lower) than TensorFI*. Additionally, we found LLTFI to be faster than TensorFI.

2. We found that for most of our benchmarks, SID achieves a high reduction in the SDC rate with a modest performance overhead.

3. Based on our FI experiments on LLMs, we learned that LLMs exhibit considerable resilience to transient faults, with an average SDC rate of 0.9% (9 SDCs out of 1000 FI experiments) across all benchmarks. Furthermore, the

manifestation of SDCs significantly varies depending on the type of LLMs and fine-tuning objectives.

4. We also found a large variance in the SDC rate for different benchmarks with the same architecture. Moreover, we also observed that for the fill-mask fine-tuning objective, the SDC rate increases with the model size.

5. Regarding the effect of stuck-at faults in DNN accelerators, we observe the manifestation of stuck-at faults at the software level (called Fault Pattern) to vary with different hardware and software configurations. We found these fault patterns to be deterministic, and we further categorized these fault patterns into six well-defined classes.

Concludingly, the observations we made in the thesis have several implications for future research in this direction. Particularly, our observation about the significant difference in the SDC rates of LLTFI and TensorFI invalidates the assumption TensorFI makes about every bit-flip fault corrupting the output of ML operators. Other application-level FI tools, like PyTorchFI and MindFI, make a similar assumption, and therefore, when using application-level FI tools, one needs to scale down the SDC rate estimates to obtain realistic results. Moreover, our work on the resilience assessment of LLMs shows that transient hardware faults in LLMs can cause them to output syntactically correct but semantically incorrect outputs, which are difficult to detect automatically. Therefore, for inclusion in safety-critical domains like AVs, there is a need to further understand the enhance the resilience of LLMs.

Chapter 2

Background

In this chapter, we start by providing the necessary background on Hardware Faults and ML models, including LLMs, required to understand the contributions of this thesis. We also explain the workings of ML accelerators like Google's TPU, along with the simulator we used for developing SystoliFI. Additionally, we briefly delineate the LLVM compiler and its constructs required to comprehend the structure and FI methodology employed by LLTFI. Finally, we conclude this section by discussing the fault models and the assumptions we made for LLTFI and SystoliFI.

2.1 Hardware Faults

2.1.1 Transient Hardware Faults

Transient faults refer to intermittent errors that occur in electronic systems, such as computer chips or integrated circuits. These faults can be caused by various factors, including cosmic radiation, electromagnetic interference, power fluctuations, and manufacturing defects [40]. As the size of fabrication technology continues to reduce, with advancements like the miniaturization of transistors and the increasing density of components on a chip, transient faults are becoming more prevalent [107]. For example, Martino et al. [29] observed around 49K processor-related failures in 261 days (approximately 1 fault every 8 minutes) in BlueWaters - a supercomputer located at the University of Illinois, Urbana-Champaign.

2.1.2 Permanent Hardware Faults

Permanent hardware faults in integrated circuits can occur in various components, such as transistors, interconnects, memory cells, or logic gates. With the continuous advancement in fabrication technology and the increasing complexity of integrated circuits, the occurrence of permanent hardware faults is on the rise [71]. The reduction in the size of transistors and the higher density of components on a chip make them more susceptible to manufacturing defects, material impurities, and aging effects.

2.2 Machine Learning Applications

In general terms, ML is a branch of artificial intelligence that focuses on developing algorithms that enable computers to learn from and make decisions or predictions based on data. ML can be classified as either supervised, unsupervised, or RL. Supervised learning involves training models on data with assigned labels, with examples being linear regression and neural networks. Conversely, unsupervised learning involves training models on data without any pre-existing labels, with k-means clustering and kernel density estimation being examples. In RL, an agent learns to make decisions by performing actions and receiving rewards or penalties. It is often used when the model needs to make a series of decisions, and the optimal solution is learned through trial and error. Examples of RL include learning to play a game or navigating a maze.

The process of developing an ML model typically involves two phases: 1) the training phase, where the model learns a specific task using the training data, 2) the inference phase, where the model uses what it has learned to make predictions on new, unseen data, often referred to as test data. The model's parameters are learned from the training data, and the effectiveness of the trained model is evaluated based on its ability to accurately predict the outcomes on the test data, which represents new scenarios the model has not seen before.

DNNs and LLMs are types of machine learning models that can be used in both supervised and unsupervised learning tasks, depending on the specific problem and dataset. Moreover, we specifically focus on DNNs and LLMs as they are widely used in various applications.

In the following sections, we further elaborate on DNNs and LLMs, as necessary for understanding this thesis.

2.2.1 Deep Neural Networks

DNNs are a type of artificial neural network with multiple layers (called hidden layers) between the input and output layers. These networks can learn complex patterns in large amounts of data, making them particularly useful for tasks like image and speech recognition, natural language processing, and other data-intensive tasks. Deep Neural Networks consist of several operations such as convolution, matrix multiplication (which is used to implement 'dense' or 'fully-connected' layers), Relu, MaxPool, and others. Among these, convolution and matrix multiplications, also known as GEMM, are the most computationally demanding operations. Therefore, to speed up these operations, hardware accelerators are often employed.

- **Fully-Connected Layer or GEMM**: FC layers, combined with non-linear activation functions, are utilized to discern a non-linear relationship between the input features (which are extracted by the convolution layers) and the network's final prediction. The FC layers are depicted using a matrix multiplication process involving the input features and the neuron weights. This is then followed by the addition of a bias term.

- **Convolution Layer**: The primary function of convolution layers is to extract features from the input image(s). The dimensions of these extracted features are determined by the kernel size. The kernel size is typically kept small because the number of parameters in a convolution layer increases exponentially with the kernel size, as per the equation:

$$NumParams = KernelSize^2 * InputChannel * OutputChannel \tag{2.1}$$

  Using a larger kernel size can make the CNNs more computationally demanding without significantly improving accuracy. Furthermore, the convolution kernels in DNNs comprise multiple output channels that enhance

the DNN's robustness and accuracy. For instance, using multiple output channels for detecting different features makes a DNN more adaptable to variations in input data, such as alterations in ambient lighting conditions.

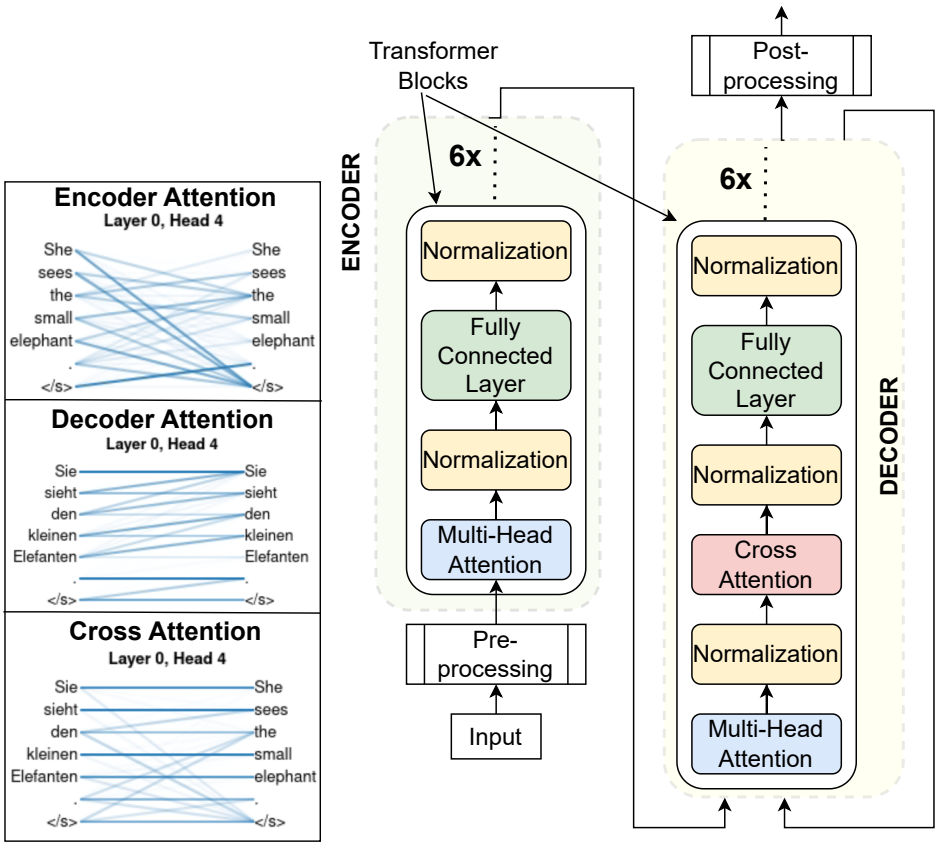
2.2.2 Large Language Models

LLMs are a type of deep learning models that are trained on extensive volumes of text data. They aim to understand the statistical patterns in language and produce text that resembles human writing. After an initial pre-training phase, these models undergo a fine-tuning process for specific tasks. These tasks can range from text classification and sentiment analysis to language generation. In addition to these tasks, LLMs can also be used for other natural language processing tasks such as question answering, summarization, and translation. The ability of LLMs to generate coherent and contextually relevant sentences has led to their use in various applications, from chatbots and virtual assistants to automated content creation.

Transformer-based Models

Transformer-based models, initially presented by Vaswani et al. [108], represent a kind of neural network architecture that has significantly advanced the field of NLP. Unlike RNNs and LSTM networks, Transformer-based models are particularly adept at identifying relationships between different segments of the input sequence in a parallel fashion. The parallel processing capability of Transformer models also makes them more efficient to train on modern hardware compared to RNNs and LSTMs, which process sequences step-by-step. The attention mechanism is a critical element of a transformer model.

Attention Mechanism: This refers to a method used by LLMs to allocate different levels of significance, known as attention weights, to various tokens in a sequence. Conceptually, it enables the model to focus on certain parts of the input that are considered more pertinent to the current context. In a practical sense, a transformer block includes multiple instances of these attention mechanisms, referred to as *Multi-head attention*, operating concurrently to comprehend different aspects of the input sequence. Certain transformer models, such as T5 [94] (Re-



(a) Attention weights. **(b)** Architecture of T5 model.

Figure 2.1: Attention weights and architecture of T5. Attention weights correspond to the fifth head of the first encoder, decoder, and cross-attention layer of a fully-trained T5 model, trained to translate English to German. The attention weights are visualized using BertViz [109].

fer to Figure 2.1b, layer highlighted with Red), also incorporate *Cross attention* layers. These layers effectively align and transfer information from the encoder to the decoder. The attention mechanism is a key factor in the success of Transformer models. It allows the model to handle long sequences and maintain a global understanding of the input sequence, which is particularly important in tasks like machine translation and text summarization. Furthermore, the weights produced by the attention mechanism can also provide some interpretability, as they indicate which parts of the input the model is focusing on when making predictions.

Figure 2.1a illustrates the attention weights of the fourth head in the first encoder, decoder, and cross-attention layer of the T5 [94] model, which is used for English-to-German translation. Self-attention is depicted through lines connecting the tokens that are attending (on the left) to the tokens being attended to (on the right). The line thickness signifies the attention score: larger attention scores between the token on the left and the one on the right are represented by thicker lines. In the cross-attention layer, it is noteworthy how the model has established thicker connections between English words and their corresponding German equivalents. This visualization provides a way to understand how the model focuses its attention during the translation process. It shows that the model can align words in the source and target languages, which is a crucial aspect of machine translation.

As depicted in Figure 2.1b, a typical transformer block comprises a multi-head attention layer, succeeded by a normalization layer and a fully-connected layer. Several of these transformer blocks (six for the T5 model) are combined to construct the encoders and decoders.

Encoders: Encoders play a crucial role in handling the input sequence, which could be a sentence, paragraph, or an entire document, and capturing the contextual representations of each word or token. The encoder's job is to create representations of the input sequence that encapsulate its meaning and context.

Decoders: Decoders utilize the encoded representations the encoder produces to generate output sequences. These sequences are used for tasks such as translations, summaries, or completions.

In other words, the encoder-decoder structure allows the model to convert an input sequence into an internal representation that captures its meaning and then convert this representation into an output sequence. This structure is commonly used in tasks like machine translation, where the input and output sequences can be of different lengths, and the relationship between input and output tokens is complex.

Different architectures of transformer-based LLMs

The three distinct architectures of transformer-based LLMs are as follows:

Encoder-Only (Auto-Encoding) These transformer models, as the name implies, only contain the encoder component and do not include a decoder. They are typically used when the main objective is to extract semantic and contextual information from the input sequence. Encoder-only transformers have demonstrated their usefulness in various downstream tasks, such as text classification, sentiment analysis, document retrieval, and feature extraction. Well-known Encoder-only transformers include models like Bert [28] and RoBerta [75].

Decoder-Only (Auto-Regressive) In contrast to Encoder-only models, Decoder-only transformers are trained in an autoregressive fashion. This means the model learns to predict the next token in the sequence based on the previous tokens. During the training process, the model aims to maximize the likelihood of generating the correct output sequence. Models like GPT2 [93] and its derivative ChatGPT [89] fall into this category.

Encoder-Decoder These models, also known as sequence-to-sequence models, comprise both an encoder and a decoder. They are extensively used in various NLP tasks, including machine translation, text summarization, and dialogue generation. Models like T5 [94] and BART [66] follow an encoder-decoder architecture.

In addition to these, there are also hybrid models that combine aspects of both encoder-only and decoder-only models. For example, the ELECTRA [26] model trains a discriminator (an encoder-only model) to distinguish between real and fake

tokens and a generator (a decoder-only model) to generate fake tokens. This allows the model to learn more efficiently from the data.

Input, Output Processing, Pre-training Objectives

Pre-training objectives Pre-training objectives can be categorized as supervised, unsupervised, or a hybrid of both. Unsupervised pre-training objectives often involve de-noising tasks, where noise or corruption is added to the input data, and the model is trained to restore the original, uncorrupted data. Typical de-noising unsupervised pre-training objectives include Masked-Language Modeling (which involves randomly masking tokens in the input text) and Denoising Autoencoding (which corrupts input text through random token dropout, token replacement, etc.).

On the other hand, supervised pre-training objectives use labeled data to direct the model's learning process. Examples of supervised pre-training objectives include sentiment analysis, Named Entity Recognition, and text classification.

LLMs like T5 are trained using both unsupervised and supervised pre-training objectives [94]. While unsupervised training aids LLMs in acquiring general linguistic knowledge and understanding language structure, supervised training allows them to fine-tune their representations for specific tasks, leading to improved performance on those tasks.

Tokenization Tokenization is the process of dividing the input text into smaller units, known as tokens. This process allows models to manage variable-length input and aids in the standardization and normalization of the text, ensuring consistent treatment of punctuation, capitalization, and variations. LLMs typically employ sub-word tokenization, which involves splitting a word into sub-word units to identify its root, prefix, and suffix. For example, the word "Reliability" would be divided into two tokens: "Reliable" (root) and "ity" (suffix). This approach allows LLMs to handle verbs, nouns, plural forms, and compound words effectively.

Tokenization also necessitates an input vocabulary file that serves as a lookup table to map tokens to unique numerical IDs. This vocabulary defines the set of tokens the LLM uses and allows the model to represent and process text as numerical values.

2.2.3 Machine Learning Frameworks

ML frameworks are libraries or toolsets designed to simplify and streamline the process of developing machine learning models. They offer high-level, user-friendly APIs that abstract away the complexities of underlying algorithms and computations, allowing developers to focus on model design and structure. These frameworks often include built-in optimizations for efficient training and inference, such as support for parallel computation, hardware acceleration, and distributed computing. They also provide flexibility for customization and extension, along with tools for tracking, visualizing, and debugging during model training. Tensorflow and PyTorch are two of the popular ML frameworks [103].

ONNX

ONNX, or Open Neural Network Exchange [13], is an open-source format for serializing ML models. It offers a standardized set of ML operations that are compatible with a variety of popular ML frameworks, including TensorFlow and PyTorch, as well as others like scikit-learn. One example of an ONNX operator is MaxPool, a common feature in many ML models that performs maximum pooling. As of the time this was written, ONNX supports over 90% of commonly used ML operators [33]. The ONNX specification is continually evolving, with new operators being added with each version release. This suggests that its coverage of ML operators will likely expand in the future.

In addition to this, ONNX also supports interoperability between different ML frameworks. This means that a model trained in one framework (like TensorFlow) can be exported to ONNX format and then imported into another framework (like PyTorch) for further use or deployment. This makes ONNX a powerful tool for ML development and deployment, as it allows developers to leverage the strengths of different frameworks and avoid being locked into a single framework. LLTFI utilizes ONNX to do FI in the ML model, irrespective of the underlying ML framework.

2.3 Machine Learning Accelerators

ML accelerators are specialized hardware designed to significantly speed up the computation-intensive tasks involved in the training and inference of DNNs and LLMs. Tasks, such as matrix multiplication and convolution, can be parallelized effectively, making them well-suited for hardware acceleration. DNN accelerators, such as Google's Tensor Processing Units (TPU)[55] and Microsoft's BrainWave[25], utilize a two-dimensional array of multiply and accumulate (MAC) units, known as a *systolic array*. This array forms the heart of these accelerators and is used for executing batch multiplication and addition operations.

2.3.1 Efficient Convolution Implementation on Systolic Arrays

There are several strategies for executing convolution operations efficiently on systolic arrays. A prevalent method involves transforming the convolution into a large GEMM operation. This is achieved by:

- reshaping the input data, which is typically a 4-D tensor with dimensions $N \times C \times H \times W$, into a 2-D matrix of dimensions $CRS \times NPQ$.
- reshaping the convolution kernel, usually a 3-D tensor with dimensions $K \times R \times S$, into a 2-D matrix of dimensions $K \times CRS$.

In this context, N, C, H, and W represent the batch size, number of channels, height, and width of the input data, respectively. Similarly, K, R, and S denote the number of output channels, rows, and columns of the convolution kernel, while P and Q correspond to the height and width of the output matrices. This approach to convolution implementation is employed in CuDNN [23], a GPU-accelerated library for deep neural networks.

This method of implementing convolution operations is particularly efficient because it allows the systolic array to perform a large number of operations in parallel. By reshaping the input data and the convolution kernel into 2-D matrices, the convolution operation can be performed as a single GEMM operation, which is highly parallelizable and can be efficiently executed on a systolic array.

2.3.2 Operation Tiling in Systolic Arrays

Operation tiling is a technique used in systolic arrays that involve dividing a larger computation into smaller sub-computations, or 'tiles.' These tiles are then processed in a parallel and pipelined manner by the array. This approach is particularly beneficial when the computation is too large to be handled by a single processor. By splitting the computation into tiles, the systolic array can process the computation in a streaming fashion, eliminating the need to store the entire computation in memory simultaneously. This is particularly useful for tasks such as image and video processing, where the computation often exceeds the memory capacity.

For instance, consider the matrix multiplication of two 4x4 matrices, A and B, as shown in Equation 2.2.

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix} \tag{2.2}$$

Assuming the systolic array size and, thus, the tile size to be 2×2. The matrices A and B are broken down into four tiles of 2×2 each, as shown in Equation 2.3 and 2.4, respectively.

$$\begin{aligned} A_{tile1} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad A_{tile2} = \begin{pmatrix} A_{13} & A_{14} \\ A_{23} & A_{24} \end{pmatrix} \\ A_{tile3} &= \begin{pmatrix} A_{31} & A_{32} \\ A_{41} & A_{42} \end{pmatrix} \quad A_{tile4} = \begin{pmatrix} A_{33} & A_{34} \\ A_{43} & A_{44} \end{pmatrix} \end{aligned} \tag{2.3}$$

The resultant matrix, C, is calculated through eight matrix multiplication and four matrix addition operations, as shown in Equation 2.5.

$$\begin{aligned} B_{tile1} &= \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad B_{tile2} = \begin{pmatrix} B_{13} & B_{14} \\ B_{23} & B_{24} \end{pmatrix} \\ B_{tile3} &= \begin{pmatrix} B_{31} & B_{32} \\ B_{41} & B_{42} \end{pmatrix} \quad B_{tile4} = \begin{pmatrix} B_{33} & B_{34} \\ B_{43} & B_{44} \end{pmatrix} \end{aligned} \tag{2.4}$$

$$
\begin{aligned}
C_{tile1} &= A_{tile1} \times B_{tile1} + A_{tile2} \times B_{tile3} \\
C_{tile2} &= A_{tile1} \times B_{tile2} + A_{tile2} \times B_{tile4} \\
C_{tile3} &= A_{tile3} \times B_{tile1} + A_{tile4} \times B_{tile3} \\
C_{tile4} &= A_{tile3} \times B_{tile2} + A_{tile4} \times B_{tile4}
\end{aligned}
\tag{2.5}
$$

This approach of operation tiling allows for efficient utilization of the systolic array's computational resources. It enables the array to handle larger computations than it could otherwise, and it allows for efficient memory usage by only requiring a portion of the computation to be stored in memory at any given time. This makes operation tiling a crucial technique for maximizing the performance of systolic arrays.

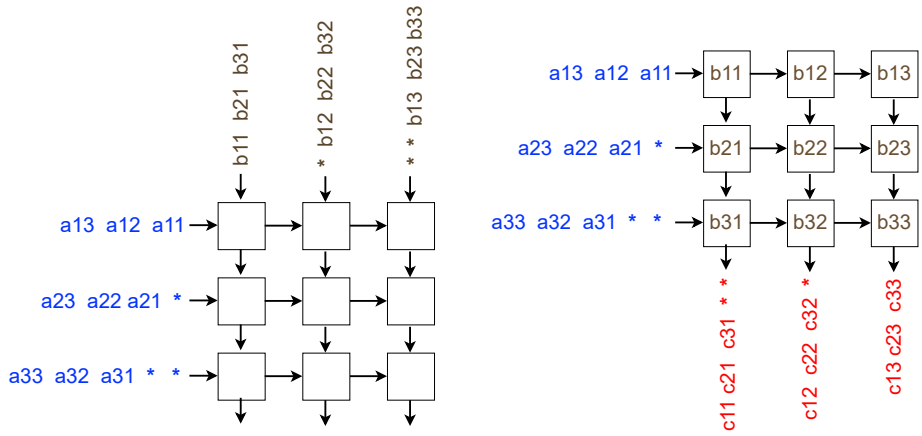
2.3.3 Data flow Mapping Schemes

In a systolic array, the term 'data flow mapping' refers to the method of directing data between the MAC units within the array. There are various data flow mapping strategies, each offering its own advantages and potential drawbacks.

Two prevalent data flow mapping strategies are WS and OS. In the WS data flow mapping scheme, the weights of the convolution kernel remain fixed, while the input data is transferred between the MAC units, as depicted in Figure 2.2b. This strategy is particularly efficient for convolution operations with large kernel sizes, as it enables the convolution operation to be segmented into smaller, parallelizable steps.

Conversely, in the OS data flow mapping scheme, the output data stays stationary, while the weights of the convolution kernel are moved between the MAC units, as illustrated in Figure 2.2a. This method is efficient for convolutions with small kernel sizes, as it enables the entire convolution operation to be executed in a single step.

Systolic arrays can also employ other data flow mapping strategies, including input stationary and hybrid schemes that incorporate aspects of both WS and OS mapping. The selection of a data flow mapping scheme is contingent on the specific needs of the application and the hardware architecture of the system.



(a) OS **(b)** WS

Figure 2.2: OS (a) and WS (b) data flow schemes. In WS, the weights are stored within the systolic array, while in OS, partial outputs are stored in the systolic array.

2.4 Gemmini: ML Accelerator Generator

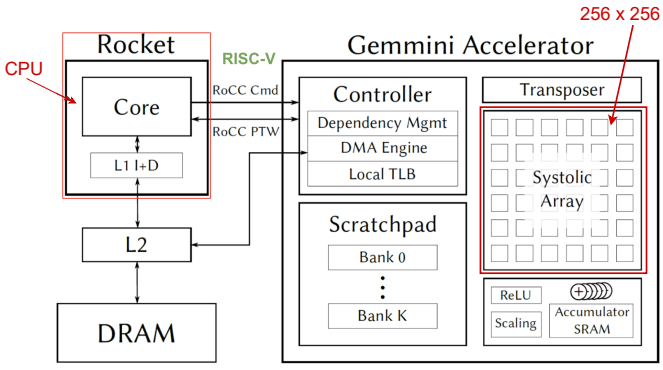


Figure 2.3: Gemmini's Architecture. [39]

Gemmini [39] is a popular systolic array generator for evaluating deep learning accelerators. Gemmini is an open-source, full-stack DNN accelerator generator for DNN workloads, enabling end-to-end, full-stack implementation and evaluation of custom hardware accelerators. Figure 2.3 shows Gemmini's architecture. The core

of the Gemmini accelerator is a systolic array that performs matrix multiplications. The matrix multiplication supports both OS and WS dataflows, which programmers can choose at runtime. The inputs and outputs of the systolic array are stored in a specifically managed scratchpad composed of banked SRAMs. A Direct Memory Access engine aids in the transfer of data between the main memory, which is accessible to the host CPU, and the scratchpad. Additionally, Gemmini includes peripheral circuitry to optionally apply activation functions such as ReLU or ReLU6, scale results down by powers-of-2 to support quantized workloads, or transpose matrices before feeding them into the systolic array.

Gemmini offers several advantages and a few drawbacks. On the positive side, it supports both systolic spatial array (like Google TPU) and vector engine array (like NVDLA [102]), and it can run ONNX models, making it easy to integrate with existing machine learning workloads. It also supports both weight-stationary and output-stationary dataflow, and the size of Gemmini's systolic array is configurable. Furthermore, Gemmini is easy to extend as it uses Cheisel, a Scala-based hardware description language that is concise and easy to understand. Due to these advantages, we used Gemmini for implementing SystoliFI.

2.5 LLVM Compiler Framework

The LLVM compiler framework [60] is a collection of tools to enable various types of code analysis, instrumentation, optimizations, and code transformations. It is used to develop compiler front ends and back ends. The Clang project, for example, provides a compiler front end for the C, C++, and Objective-C languages that uses LLVM as its back end.

LLVM Compiler Infrastructure works on the LLVM IR, which follows a static-single assignment (SSA) format [27]. It serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.

Consider the LLVM IR corresponding to the `max` function in Figure 2.4. The entire function consists of four basic blocks, all ending in either Branch or Return instructions (`br` and `ret`). The first instruction is `cmp`, which compares the value of `%a` with the value of `%b`, and returns `True` if the value of `%a` is bigger than that

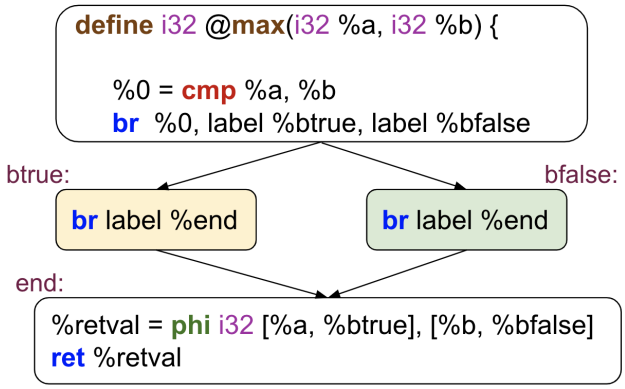


Figure 2.4: LLVM IR of `max` function, that returns the maximum of the two input values.)

of `%b`. The last basic block also consists of a 'PHI' node, a type of instruction used in LLVM IR to represent a conditional branch in the control flow of a program. It is used to specify that the value of a particular variable may come from multiple sources, depending on the program's control flow. In this example, the PHI node denotes that the variable `%retval` can be either equal to `%a` or `%b`, depending on the control flow.

Chapter 3

Related Work

3.1 Fault Injection in ML Applications for the CPU Hardware Model

3.1.1 Transient Fault Injection

Li et al. [68] were among the first to analyze the effects of transient hardware faults on DNNs by conducting a systematic FI study in a DNN simulator, specifically during the inference phase. Following their work, Reagen et al. presented Ares [97] introduced Ares, a fault injection tool that leverages tensor operations on the GPU to enhance the speed of fault injection at the application level. Ares, however, requires changes in the Keras inference computation graph which require changes in the Keras and Tensorflow framework. To overcome this limitation, researchers proposed PyTorchFI [81] and TensorFI [69], which are application-level FI tools developed for PyTorch and TensorFlow ML framework, respectively, and do not require changes in the ML framework itself.

Both PyTorchFI and TensorFI are framework-specific, i.e., PyTorchFI can only work with the PyTorch framework, while TensorFI works only with the TensorFlow framework. To overcome this limitation, in this thesis (Chapter 4), we propose LLTFI, which is an IR-level FI tool for ML applications that is ML framework agnostic. LLTFI injects faults in the instructions and registers level and is, thus,

more precise (proven empirically in Section 4.4.3) in emulating transient hardware faults than PyTorchFI and TensorFI.

He et al.'s [44] work is one of the few that evaluate the resilience of NLP-based ML models. They present FIdelity, a framework for evaluating the resilience of ML applications under transient faults in ML accelerators. Using FIdelity, they performed FI in RNNs and Transformer layers. Unlike FIdelity, in this thesis (Chapter 5), we carry our FI in all layers of the LLM, not just the transformer blocks. Moreover, we also evaluated the effect of model size, pre-training, and fine-tuning objectives, on the resilience of LLMs.

Extending He et al.'s [44] work, Liu et al. [73] evaluated error propagation in RNNs due to transient hardware faults. They found that the fault tends to spread as it propagates through the RNN layers. They also found that the execution time of RNNs significantly varies due to FI, perhaps due to the iterative nature of RNNs. Unlike Liu et al., in this thesis, we focus on LLMs which are structurally different from RNNs, so their results are not comparable to ours.

3.1.2 Transient Fault Mitigation in ML programs

Software-based soft error mitigation methods for ML programs can be broadly categorized into the following three classes:

Range Restriction

Range restriction techniques [22, 45, 48] leverage the fact that corrupted values (due to soft errors) that are *abnormally* large are most likely to cause the CNN to misclassify. These methods first profile the CNNs to identify the nominal range of neuron output values and then use it as the upper and lower bound for range restriction. At runtime, Ranger [22] checks if the neuron outputs are within the range; if not, the output gets truncated to the upper bound. Although RR has lower runtime overheads, the major issue with this line of work is that they have a high false-positive rate and can not be used with online learning ML algorithms that constantly evolve with time; thus, a fixed upper, lower bound can not be used. Along similar lines, Hong et al.[48] proposed using activation functions like `tanh`, which have implicit lower and upper bounds to prevent the propagation of abnor-

mally large neuron values. However, their approach requires retraining of the CNN and might also impact the accuracy of the ML program.

Redundancy-based

These techniques, including instruction duplication, work by adding redundancy to the ML programs. While prior works have proposed triple-modular-redundancy (TMR) [72] at the operator level, feature or neuron duplication (like FLR [79]), in this thesis (Section 4.5), we evaluate the efficacy of selective instruction duplication for improving the resilience of ML models against transient hardware faults. Unlike SID, operator-level duplication [72] has more than 100% runtime overhead, and feature duplication [79] requires extensive profiling of the ML program with all possible input features (with all images in the dataset).

Approximation-based

Approximation-based techniques work by (partial) approximation of the ML program to improve its fault tolerance. Li et al. [68] proposed an approximation-based approach that works by using smaller data types (like `Int`, `Float16`) instead of `Float64`, which is conventionally used in CNNs. The primary drawback of this approach is that it also degrades the accuracy of the CNN.

Hardware-based SDC mitigation techniques like full ALU duplication (the approach taken by IBM eServer z990 [84]), selective latch duplication [68] and using ECC in memory are certainly effective but require hardware changes, which might not always be feasible.

3.2 Fault Injection for the ML Accelerator Hardware Model

Characterization of hardware faults in ML accelerators in prior work is often limited to the impact on accuracy without analyzing the fault patterns induced at the software level. To the best of our knowledge, Rech et al.'s recent work on analyzing the resilience of EdgeTPU [98] is the only one that analyzes the fault patterns induced at the software level occurring due to transient faults. Unlike Rech et

al., in this thesis (Chapter 6), we analyze fault patterns at intermediate layers of the ML models arising due to permanent stuck-at faults in ML accelerators. We also considered various hardware and software configurations that might affect the manifestation of fault patterns.

3.2.1 Fault Injection and Characterization

Zhang et al. [114] and Holst et al. [47] studied the effects of timing faults in systolic arrays, thus, degrading DNN's accuracy. However, their work is not directly comparable to ours due to the fundamental difference in our fault models (stuck-at vs. timing errors).

Other works like that of Kundu et al. [58], Zhang et al. [116], Ren et al. [99], and Feng et al. [37] aims to understand the effects of transient faults on systolic arrays. Kundu et al. [58] analyzed the manifested faults' boundaries while leaving details of error patterns in the intermediate outputs unexplored. Ren [99] studied the geometrical pattern of faults in systolic arrays on-chip. However, it does not study the geometric pattern of manifested faults in software. Feng et al. [37] extended the characterization beyond accuracy by also considering the crash rate as a metric. However, all of these works use software simulation of systolic arrays and do not consider fault patterns. Software-level systolic array simulations create an abstract model of the systolic array that abstracts out hardware configurations like data flow mapping schemes. In contrast, we use RTL-level FI, which captures hardware details of ML accelerators, thereby resulting in more accurate fault patterns.

In very recent work, Tyagi et al. [106] worked towards quantifying the accuracy of DNN accelerators under transient faults. They used RTL-level FI and proposed a new, more-accurate resilience assessment metric. However, they do not consider fault patterns emanating at the intermediate layers of the DNNs. Li et al. [68] studied the error propagation with regard to some architectural parameters of CNNs. However, in systolic arrays, the manifestation of faults varies with the hyper-parameters that were not considered in this study.

Unlike other prior works, Burel et al. [16] considered permanent faults in systolic arrays and found that OS dataflow is more fault-tolerant than WS (similar

result to ours) and, thus, proposed OS-based, fault-tolerant systolic array architecture. However, Burel et al. [16] did not consider the effect of different parameters (including data mapping schemes) on the fault patterns: they only considered the final DNN accuracy degradation in the presence of faults.

To summarize, there is limited prior work on the extensive characterization of faults in systolic arrays. Among those, most of them were limited to only transient faults; that too, they either did not consider fault patterns at all, or they used an abstract software model for FI experiments, or even both. Moreover, none of the prior work that looked into fault patterns considered other essential factors like data mapping schemes and tiling. In this thesis, we bridge this gap by using RTL-level FI and considering the effect of various hardware and software configurations on the resulting fault patterns.

3.3 Summary

3.3.1 Transient faults in the CPU

For transient faults in the CPU, the prior work suffers from the following three limitations:

1. All prior work on software-level FI in ML applications is limited to application-level FI (like TensorFI), which implicitly assumes that every transient hardware fault ends up corrupting the output of the ML operators.

2. Existing application-level FI tools are all ML framework-specific.

3. None of the prior work considered evaluating the resilience of LLMs.

We address the first two limitations in Chapter 4 of this thesis, where we proposed LLTFI, a framework-agnostic, IR-level FI tool that injects transient hardware faults in registers and instructions. Using LLTFI, we empirically invalidate the assumption made by existing application-level FI tools. Moreover, in Chapter 5, we further use LLTFI to evaluate the resilience of LLMs against transient hardware faults.

3.3.2 Permanent faults in ML accelerators

Regarding the fault injection and characterization for the ML accelerator hardware model, most of the prior work is limited to just transient hardware faults. The few papers that considered permanent faults in ML accelerators also did not evaluate the software-level fault patterns arising from hardware faults. We address this limitation in Chapter 6 of this thesis, where we evaluated different fault patterns arising due to stuck-at faults in ML accelerators.

Chapter 4

LLTFI: Implementation and Evaluation

To evaluate the resilience of ML applications under transient hardware faults, in this chapter, we propose LLTFI, a framework-agnostic, IR-level FI tool. We start this chapter by first discussing our fault model and design constraints for LLTFI. We then articulate our implementation of LLTFI and possible design alternatives. Thereafter, we compare LLTFI with TensorFI and present the performance evaluation of LLTFI. Finally, we present a case study on using LLTFI to evaluate the efficacy of selective instruction duplication for protecting ML models against transient hardware faults. We conclude this chapter by discussing the implications of our results.

4.1 Fault Model

For LLTFI, our primary focus is on transient hardware faults that occur within the CPU in the registers and data path. Transient faults can occur due to high-energy particle strikes on hardware elements and can result in bit-flips in registers and latches [77].

We do not consider faults in the instruction encoding or the processor's control logic. Furthermore, we operate under the assumption that these faults do not alter the control flow graph of the ML models. These assumptions are made to ensure

that the faults we inject do not result in software crashes, which are more easily detectable.

Our assumptions align with most previous research in this area [42, 69, 76, 97].

4.2 Design Constraints

We adhere to the following four constraints in the design of LLTFI:

- **Compatibility with multiple ML frameworks.** The FI tool should work with ML models written in various ML frameworks. Not only this improves the utility of the tool, but it is also crucial for contrasting the resilience of ML models across different ML frameworks. This is important because complex ML-based systems for autonomous vehicles such as Baidu's Apollo [1] and Comma.ai's OpenPilot [2] incorporate ML models developed over multiple ML frameworks. For instance, Baidu Apollo contains at least 28 different ML models, written in frameworks including TensorFlow and PyTorch, along with programmed components in C++ [91]. Therefore, to evaluate the resilience of complex ML-based systems, it is important for a single FI tool to support multiple ML frameworks to ensure consistent FI and ease of use.

- **Instruction and register-level FI.** Unlike prior work that injects faults at the application level (in the output of ML operators), our FI tool should inject faults in the instructions and register level. This way, the FI will be closer to the hardware i.e., we can emulate hardware faults more accurately.

- **Efficient error propagation tracing for ML applications** The FI tool should be capable of doing efficient error propagation tracing in ML applications. This allows the user to track the injected fault as it propagates through the ML model and further understand how the fault spreads or gets suppressed by different ML operators.

- **Ease of Use and Portability** The FI tool should be user configurable and should be easy to use without requiring the user to understand the internal workings of the tool itself. Moreover, the FI tool should be portable, i.e., the tool should not make any assumptions about the underlying system architecture.

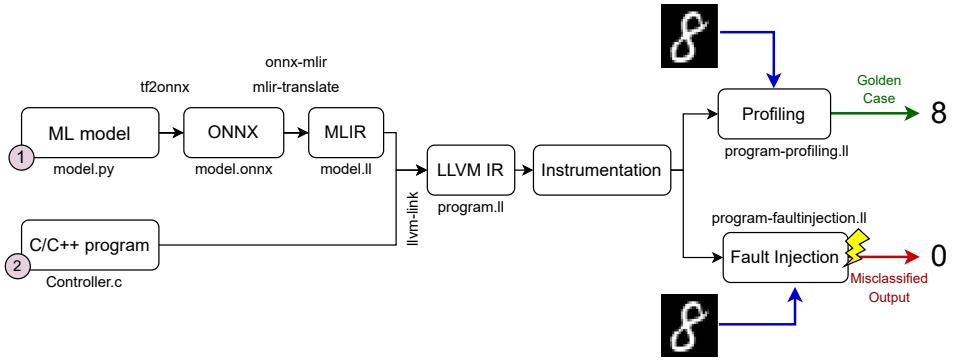


Figure 4.1: LLTFI's workflow.

4.3 LLTFI's Implementation

LLTFI is built on top of LLFI [76] and is fully backward compatible with it. While developing LLTFI, we upgraded the entire LLFI tool, which was originally written for LLVM 3.4, to LLVM 12.0. This is important as LLVM 3.4 has no support for MLIR, which is required to lower ML programs to LLVM IR. LLTFI provides a single script that converts ML models into LLVM IR for convenience.

Figure 4.1 shows the workflow diagram of LLTFI. Firstly, developers write their models using the ML framework (e.g., TensorFlow, PyTorch) and train their models. The trained models are then exported to a saved model format (like Tensorflow's SavedModel). Secondly, LLTFI converts the saved model to the ONNX [13] format, which is a common, abstract format for models written in different ML frameworks. Thirdly, the ONNX file is converted into MLIR through ONNX-MLIR [53]. Finally, MLIR is converted into LLVM IR using the `mlir-translate` tool in LLVM 12.0.

To run the ML model, a controller program written in C or C++ is required. LLTFI directly compiles the controller program into LLVM IR, which can then be linked to the ML program. After this step, LLTFI can inject faults into the LLVM IR of the model at runtime without compilation.

```yaml
compileOption:
  instSelMethod:
   - customInstselector:
       include:
         - CustomTensorOperator    # Select custom LLVM pass for FI
       options:                    # FI in 3rd Relu layer
         - -layerNo=3              # Number of layer for FI
         - -layerName=Relu         # Name of Tensor Operator for FI

  tracingPropagation: True         # Enable instruction tracing?
  tracingPropagationOption:
    maxTrace: 250                  # Max number of instructions to trace
    mlTrace: False                 # Enable tracing for ML programs?
runOption:
  - run:
    numOfRuns: 25                  # Number of experiments
    fi_type: bitflip               # Type of fault to inject
    fi_max_multiple: 1             # Number of faults to inject
```

Figure 4.2: Example of the YAML file used by LLTFI to configure the FI experiments.

4.3.1 Fault Types

LLTFI supports the injection of bit flips, which flips bits from 0 to 1 or vice versa, in the destination register of instructions. These bit flips can be single (i.e., one bit flip on a single instruction in a program) or multiple (i.e., multiple bit flips across multiple instructions or multiple bit flips in a single instruction). Users can also specify the bit position in which fault injection is performed. All of these options can be specified in a YAML file by the user without requiring any changes to the ML model.

Figure 4.2 shows an example of the YAML file. There are two sections in the YAML file: (1) compileOptions, and (2) runtime options. These mirror those used in LLFI [76]. Compiler options are those that determine which instructions should be selected for FI, as well as the layers of the ML model to inject into. Additionally, they also specify whether to trace fault propagation and, if so, to what depth (i.e., the number of instructions to trace). Runtime options, on the other hand, determine the FI experimental parameters, such as the number of experiments to be run, the

type of fault to be injected, and the maximum number of bit-flips (for multi-bit faults) to inject.

4.3.2 Tracing Fault Propagation

LLTFI provides fault propagation tracing at two different levels of granularity: at the tensor operator level and the instruction level. Tensor operator-level fault propagation tracing enables LLTFI to identify the effect of the injected fault on the output of the ML operators like GEMM and convolution. It can be used to determine how the fault originates, propagates, or gets suppressed by the ML operator. On the other hand, instruction-level tracing can be used to precisely track the propagation of the injected fault through the assembly instructions at runtime.

4.3.3 Example of running LLTFI on an ML Program

In this section, we illustrate how LLTFI works on an ML model for low-level FI and how it helps users visually understand the effects of FI. Suppose we have an ML model written in Python, using the TensorFlow and Keras [24] APIs, as shown in Listing 4.1[1]. This model is trained on the MNIST [63] dataset and aims to classify handwritten images of digits into a single numerical value between 0 and 9. We compile this model, train it and export its weights and architecture information to a saved model format.

LLTFI's compile script automatically converts the saved model into LLVM IR, as depicted in Figure 4.1. In this example, however, we break down the internal steps. The saved model is first converted into ONNX using the `tf2onnx` tool, resulting in model.onnx. Then, model.onnx is lowered into an MLIR file, model.mlir, using the ONNX-MLIR tool. The MLIR is very similar to LLVM IR [60]. Finally, model.mlir is converted into LLVM IR, *model.ll*, using the `mlir-translate` tool.

[1] Though the example uses the TensorFlow framework, LLTFI is agnostic of the ML framework as it uses ONNX.

Listing 4.1: model.py - TensorFlow model

```
model = models.Sequential()
model.add(
  layers.Conv2D(32, (5, 5), activation="relu", input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (5, 5), activation="relu"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation="softmax"))
```

Listing 4.2: controller.c - to invoke TensorFlow model

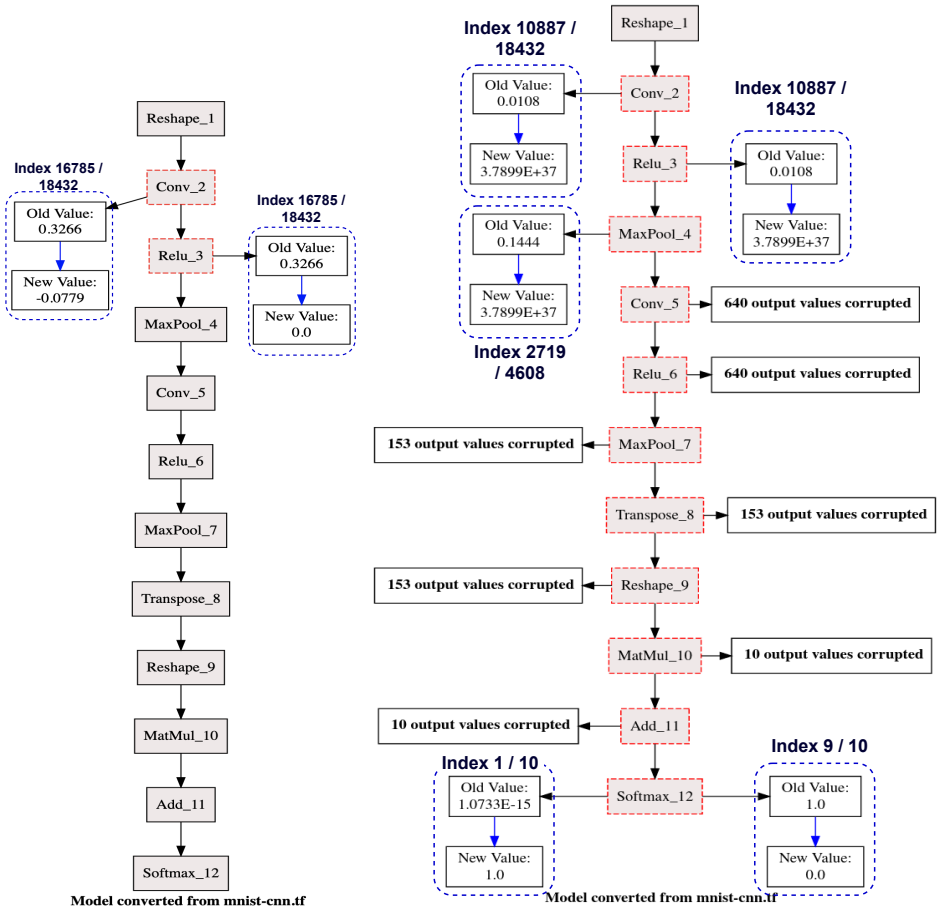
```
OMTensorList *run_main_graph(OMTensorList *);

int main(...) {
  OMTensorList *input = read_input(image_file);
  OMTensorList *outputList = run_main_graph(input);
}
```

Despite obtaining the LLVM IR code, model.ll cannot be executed directly. The model is contained within a function called *main_graph()*, which is invokable by the *run_main_graph()* function. A controller program reads the image file into a tensor and invokes the model with the input tensor, as shown in Listing 4.2. This controller program is provided by LLTFI and is independent of the ML model and dataset. Therefore, users do not need to write their own.

We compile the controller program using LLVM's clang to obtain controller.ll, which we then link with model.ll to obtain a single unified LLVM IR file, *program.ll*.

Finally, we apply LLTFI to perform FI on *program.ll*. We first instrument it to add fault injection calls and profile it (i.e., run the instrumented code without any faults). In this example, we pass in a handwritten image of the digit 8 to the controller program, as shown in Figure 4.1. The output of the model is an array of length 10, representing the digit-wise softmax output by the ML model. In the figure, we show the classes inferred with the highest probability. During the profiling run, LLTFI generates a golden output of $[0,...,0,0,0.999989,0]$. This means that the model has classified the test image as the digit 8 with a probability of 0.999989. However, when the fault is injected, LLTFI generates an erroneous



(a) Fault got suppressed by ReLu.

(b) Fault resulted in misclassification.

Figure 4.3: LLTFI on CNN-MNIST. The first snippet shows the case where the fault got suppressed by the ReLu layer. The second snippet shows fault propagation resulting in misclassification.

output of $[1,...,0,0,0,0]$. Instead of correctly classifying it as 8, the model misclassifies the image as 0.

LLTFI also allows users to trace the propagation of the injected fault among different layers of the ML application in the form of a DFG. We use Graphviz [35] to generate the visual DFGs. In Figure 4.3b, we show the tensor operator-level error propagation DFG generated by LLTFI for this example. The DFG nodes highlighted with gray represent the layers of the ML model, and the other nodes originating from the layer nodes show the change in the layer output due to the injected fault. We observe that LLTFI injected a fault in the *Conv_2* layer, thereby modifying the output value at index 10887 from 0.0108 to $3.7899E+37$. This fault propagated through *Relu_3* and *MaxPool_4* layers, and at *Conv_5* layer, the fault gets propagated manifold, thereby corrupting 640 different output values and results in a misclassification.

Figure 4.3a shows another possible scenario where the injected fault corrupts a single output value of the *Conv_2* layer, but the fault got masked by the *Relu_3* layer. This fault results in a benign output. LLTFI's graph visualization feature offers an intuitive way to understand how low-level errors originate and propagate through different layers of the ML model.

4.3.4 Design Choices and Alternatives

Selecting the lowering mechanism. There are multiple methods to lower high-level ML models to an IR. We explain why LLTFI lowers ML models to MLIR using ONNX-MLIR [53] over other alternatives.

Glow is an ML compiler [100], which lowers a neural network dataflow graph into high-level Glow IR, followed by low-level Glow IR. However, Glow does not lower to LLVM IR. XLA [62] and TVM [18] are both compilers for lowering TensorFlow and PyTorch models into LLVM IR. XLA first lowers into XLA high-level optimizer representation, while TVM first lowers into Halide IR [95], before lowering to LLVM IR. While XLA and Halide both enable more optimizations than MLIR, their compilation time overhead is also greater.

Moreover, one of the objectives of this work is to develop a unified fault injector that works with both C/C++ and ML programs (in a framework-agnostic

manner). While TVM and XLA can be used for ML programs, they do not work with C/C++ programs. In LLTFI, we used ONNX-MLIR to lower ML models to LLVM IR, followed by fault injection using LLFI, which can also be used for C/C++ applications. Therefore LLTFI is backward compatible with LLFI and can be used with both C/C++ and ML programs.

Adapting low-level fault injection for ML models. LLTFI injects faults into values chosen at runtime with uniform probability, including address values. Since address values are abundant and accessing illegal addresses often causes crashes, fault-injected models are more likely to crash than to produce SDCs (e.g., misclassifications). This is problematic for comparing the FI results with ML FI tools, which do not typically cause crashes [69].

To avoid the problem of injecting into address values, we provide an LLVM pass for users to skip certain instructions while injecting directly into math and logic operations in the basic blocks representing the high-level ML computation graphs. Additionally, this pass enables targeted FI into selected ML model layers. The primary challenge was to map Tensor operators in the high-level ML computation graph to their respective LLVM IR code blocks. We addressed this challenge in the new pass by utilizing the debug instrumentation provided by the ONNX-MLIR tool to uniquely identify the LLVM IR code block(s) of different Tensor operators.

4.4 LLTFI's Evaluation

In this section, we present our research questions, evaluation methodology, and the results of our evaluation.

4.4.1 Research Questions

We asked four RQs to compare high-level (using TensorFI) and low-level ML FI (using LLTFI).

- **RQ0** Are the FI results by LLTFI consistent among ML models written in different ML frameworks?

Table 4.1: ML applications used for LLTFI's evaluation.

Network	Dataset(s)	Number of parameters
CNN	MNIST, F-MNIST	44426
LeNet [64]	MNIST, F-MNIST	44426
VGG16 [104]	MNIST	14913226
AlexNet [57]	CIFAR10	1642282
SqueezeNet [51]	CIFAR10	122986
Dave2 CNN [15]	Driving [17]	252219

- **RQ1** How does the overall SDC rate of the ML model with LLTFI compare with that of TensorFI?
- **RQ2** Does a single bit-flip fault injected in a specific layer of the ML model affect the output of that layer differently in both LLTFI and TensorFI?
- **RQ3** How does the performance overhead of LLTFI compare with that of TensorFI?

4.4.2 Experimental Methodology

Benchmarks and Datasets

ML applications. We choose five ML programs, all of which are classifier models. These programs are of varying sizes and are commonly used by other FI studies [69]. We also included Nvidia's Dave2 [15], a CNN used in autonomous vehicles, which is a regression model. Table 4.1 lists the benchmarks and datasets.

Datasets. We used three publicly available datasets for the classifier models: MNIST (hand-digit recognition), Fashion-MNIST (F-MNIST; fashion products classification), and CIFAR-10 (a large object recognition dataset). We also used a real-world driving dataset with labeled steering angles [17] for evaluating the regression model, Dave2.

Evaluation Metrics

We measure the efficacy and performance overhead of LLTFI. We use the SDC rate for measuring LLTFI's efficacy. For the classifier applications, the SDC rate refers to the fraction of injected faults that result in object misclassification. For the regression model, Dave-2 (i.e., predicting steering wheel angles), we use four different thresholds: 15, 30, 60, and 90, and classify an output as an SDC if the difference between the corrupted and the original output is greater than the threshold, as done by prior work [69].

For the performance overhead of LLTFI, we measured the profiling time and FI time of using LLTFI on our benchmarks (20 samples for each). Profiling time is a one-time cost to analyze the program (e.g., obtain instruction counts) at runtime. In contrast, FI time is a recurring cost and is measured as the time taken to run the fault-injected program compared to the baseline program execution time.

Optimizations and Transformations

The default graph transformations by ONNX are semantics-preserving [31], and while converting the ML models to ONNX, we did not use any optimizations like constant folding. Moreover, while converting ONNX models to LLVM IR using ONNX-MLIR, we used only O0 optimization, thereby ensuring that the FI results from LLTFI are comparable to those from FI performed on the original ML program.

Experimental Setup

We ran all our experiments on an Intel Xeon E-2224 quad-core CPU with 900MHz clock speed. To keep our results comparable with the prior work [69, 81], we decided not to use GPUs for calculating the performance overheads. Moreover, for our ML benchmarks, we used FLOAT32 as a data type.

Fault Injections

Although LLTFI can be used to inject both single and multi-bit flip faults, in our FI experiments, we injected only single bit-flip faults as this is the de-facto fault model

used in soft-error studies [21, 70, 80]. Further, Sangchoolie et al. have shown that single-bit faults are often sufficient for measuring the SDC rate of programs [101].

We injected faults only in the operands and results of floating-point arithmetic, logic instructions like `FADD`, and `FCMP`, to prevent application crashes (using the LLVM pass described in Section 4.3).

To ensure an apples-to-apples comparison between LLTFI and TensorFI, we chose the same type of faults for each comparison point. However, in LLTFI, the faults were injected into the instruction operands and results within an operator of the ML model. Whereas in TensorFI, the faults were injected directly into the output of a randomly chosen operator. This is because TensorFI (like most ML FI tools) does not provide visibility into the computations performed within an operator.

Note that only one fault was injected in each FI trial to avoid interference. Further, we report only those trials in which the faults were activated (i.e., read by the system). These assumptions are common in most FI studies [69, 81, 97, 117].

4.4.3 Results

We organize the results based on the RQs.

RQ0: FI consistency between ML frameworks

For this RQ, we were interested in checking whether the FI by LLTFI is consistent among different ML frameworks. We chose two of the widely used ML frameworks - PyTorch and TensorFlow. We then compared the SDC rates reported by LLTFI for single-bit flip faults injected into the same ML models written using these two ML frameworks. We performed this experiment 1250 times for each framework and program. We report the average SDC rates for each program and the error bars at the 95% confidence intervals.

Figure 4.4 shows the SDC rates for each of the benchmarks written using the TensorFlow and PyTorch frameworks (blue and green bars in the graph). We observed that the SDC rates are similar across all benchmarks (i.e., are within the error bars). *Therefore, LLTFI is consistent across the TensorFlow and PyTorch frameworks.*

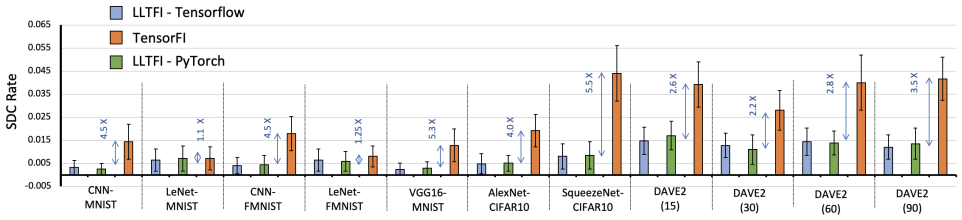


Figure 4.4: Average SDC Rate for single bit-flip faults injected by TensorFI and LLTFI across benchmarks. For LLTFI, we consider both the TensorFlow and PyTorch versions of the benchmarks. The error bars show the 95% confidence intervals.

Upon further inspection of our benchmarks, we found that the ONNX graphs exported by PyTorch and TensorFlow are very similar: most of the tensor operators are the same except `AveragePooling` and General Matrix Multiplier (`GEMM`). Unlike TensorFlow, PyTorch's ONNX exporter emits an extra padding operator with `AveragePooling`, and also prefers to use `GEMM` operator instead of `MatMul` operator for fully-connected layers. Nevertheless, these differences are minor and do not change the overall program's semantics. Therefore, LLTFI's results are consistent between these ML frameworks.

For the rest of the RQs, we will use only the TensorFlow versions of the benchmarks, as the differences between the PyTorch and TensorFlow versions are negligible.

RQ1: Comparison between high-level and low-level ML FI

To answer this RQ, we injected single bit-flip faults using LLTFI and TensorFI in randomly-selected layers of the benchmark programs and recorded the SDC rates. For each benchmark, we ran this experiment 1250 times and reported the average SDC rates as well as error bars at the 95% confidence intervals.

Figure 4.4 shows the SDC rates reported by LLTFI and TensorFI across all the benchmarks. As can be seen from the figure, across the benchmarks, the SDC rates reported by TensorFI were much higher than those reported by LLTFI. The differences ranged from 1.1*X* (LeNet-MNIST) to 5.5*X* (SqueezeNet-CIFAR10),

with an average of 3.4X. Thus, TensorFI over-reports the SDC rate compared to LLTFI. We examine the potential reasons for the same in the next RQ.

RQ2: Differences between the tools for a bit-flip fault in a specific layer of the ML model affecting the layer output

For this RQ, we were interested in the following questions: (1) what percentage of bit-flip faults injected by LLTFI propagate to the output of the target layer? (2) can a single bit-flip fault corrupt multiple output elements of the target layer? and (3) how often does the corrupted output element(s) value differ from the correct output by just a single bit-flip?

Table 4.2: Error Propagation of single bit-flip faults injected by LLTFI across models.

Benchmark Name	**% of faults that affected layer's output**	**Corrupted element differs from original by one bit flip**
CNN-MNIST	37.9%	19.2%
CNN-FMNIST	37.6%	23.7%
LeNet-MNIST	38.1%	29.5%
LeNet-FMNIST	39.7%	30.2%
VGG16-MNIST	37.0%	24.4%
AlexNet-CIFAR10	43.4%	27.2%
SqueezeNet-CIFAR10	43.0%	30.0%
Dave2-Driving	38.1%	26.6%
Average:	**39.4%**	**26.4%**

These questions will help us answer why LLTFI leads to much lower SDCs than TensorFI, as we found in RQ1. For example, if it turns out most faults injected in a layer by LLTFI get masked before reaching the output of a layer, that could explain why LLTFI has a lower SDC rate than TensorFI, which always injects a fault into the output layer. Conversely, if a single bit flip fault injected by LLTFI reaches the output of the layer but ends up corrupting multiple elements of the

output, then it could have a more significant effect on the SDC rate than TensorFI, which typically corrupts a single element of the layer output.

To answer this RQ, we injected a single bit-flip fault using LLTFI in a randomly-selected layer of our benchmarks and studied the error propagation using LLTFI's fault tracing capabilities. For this experiment, we ran each FI campaign 1000 times for each benchmark and calculated the average percentages of occurrences of each of the above outcomes.

The results are shown in Table 4.2. We observed that, on average, only 39.4% of the faults injected by LLTFI propagate to the output(s) of the target layer. The remaining faults are either masked primarily due to multiplication by zero (for convolution and matrix multiplication layers) or are pruned due to the activation function or max-pooling layers.

Moreover, in *all of our experiments*, we observed that a single bit-flip fault corrupted at most a single output element of the target layer (not shown in the table). In other words, *there was no case in which a single bit-flip fault propagated to multiple output elements of the ML model layer.* These observations are the reason why the SDC rate of LLTFI is significantly lower than that of TensorFI for the same fault types.

Finally, we find that though we injected only a single bit-flip fault in each layer, it resulted in a single bit-flip in the output of the layer in only 26.4% of the cases on average. In the remaining 73.6% cases, the corrupted output element of the layer differs from its original value by multiple bit-flips. TensorFI injects single-bit flips into the output elements of the layer. However, low-level faults often lead to multiple bit-flips at the output layer of the ML model.

In Section 4.6, we further discuss these results along with their implications.

RQ3: LLTFI's performance overheads

We measure the performance overhead due to profiling and FI by LLTFI in Table 4.3. Recall that profiling is a one-time cost, whereas FI is a recurring overhead. We find that the overhead due to profiling is 58% on average (ranges from 12% to 104%), while the FI overhead is 68% on average (ranges from 19% to 117%) compared to the baseline model.

Table 4.3: Profiling and fault injection (FI) overheads of LLTFI for ML applications.

Benchmark Name	Baseline Time (s)	Profiling Overhead	FI Overhead	TensorFI Baseline (s)	TensorFI FI Overhead
CNN-MNIST	0.088	51%	48%	1.51	85%
LeNet-MNIST	0.045	98%	113%	1.53	77.40%
CNN-FMNIST	0.056	69%	86%	1.56	73%
LeNet-FMNIST	0.045	104%	117%	1.50	76%
VGG16-MNIST	0.982	4%	19%	2.15	138%
AlexNet-CIFAR10	0.331	10%	39%	1.71	81%
SqueezeNet-CIFAR10	0.081	82%	81%	2.40	80%
Dave2-Driving	0.363	12%	38%	1.52	82%
	Average Overheads:	**54%**	**68%**		**86.6%**

We also compare the overhead of LLTFI with TensorFI for each benchmark in Table 4.3. Note that TensorFI does not have a separate profiling phase, so the overheads for TensorFI represent the FI overheads. As can be seen, the average overhead of TensorFI is 86.6%, compared to 68% for LLTFI. Note that the baseline execution times of TensorFI are also much higher than those of LLTFI. However, the FI overhead is calculated as a percentage of the baseline time.

Thus, LLTFI is $1.27x$ faster than TensorFI, on average. This is because LLTFI works at the LLVM IR level for instrumenting the code for FI, whereas TensorFI instruments the Python code. The LLVM IR code gets compiled to native assembly code, as opposed to Python code, which is interpreted. Further, the instrumentation added by LLTFI can be optimized by the LLVM compiler, while the instrumentation added by TensorFI can inhibit compiler optimizations.

4.5 Case Study: Selective Instruction Duplication

SID is a technique for selectively duplicating certain instructions in the program to detect soft errors [88]. SID has been studied in the context of general-purpose programs [49], GPUs [80], Embedded systems [87], super-scalar processors [88], etc. In this study, we use LLTFI to evaluate the effectiveness of SID techniques for ML applications in a framework-agnostic fashion. Further, we go beyond error de-

tection and also actively correct the error with SID. We extended LLTFI to perform SID at the LLVM IR level and add correction logic to the application.

To determine which instructions to duplicate, we used LLTFI to find the sensitivity (SDC rate) and the vulnerability (fraction of program run time spent in that instruction) of each arithmetic instruction within the ML program. We observed that arithmetic instructions within the convolution operators are the most vulnerable and sensitive ones. Similar findings have been reported by prior work [14, 72], and therefore, for this case study, we selected the arithmetic instructions within the convolution operators to duplicate with SID.

We then apply the three SID heuristics discussed in the next section to duplicate the instructions.

4.5.1 Duplication Heuristics

We use three heuristics for duplicating the instructions.

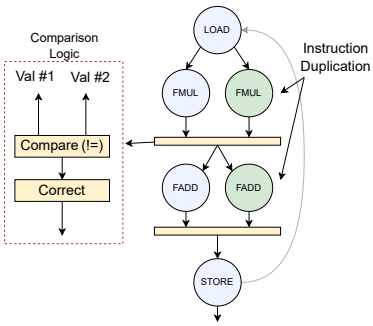
Arithmetic SID (AID)

Since we assume that transient hardware faults do not modify the control flow structure of the ML applications (Section 4.1), it is sufficient to duplicate only the arithmetic instructions such as `FMUL`, `FSUB`, `FDIV`, and `FADD`, which are primarily involved in the computation of different DNN operators such as convolution and GEMM.

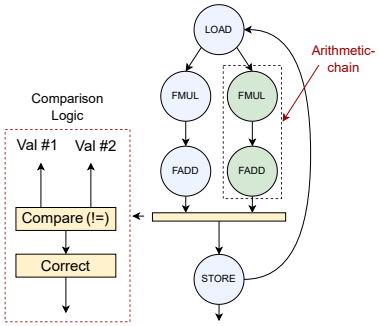
As shown in Figure 4.5a, our extension of LLTFI duplicates arithmetic instructions like `FMUL` and `FADD` and then insert the logic to compare the outputs of the original (`Val #1`) and the duplicated instructions (`Val #2`). Upon detecting a mismatch, the error correction heuristics described in Section 4.5.2 are deployed.

Arithmetic Chain Duplication (ACD)

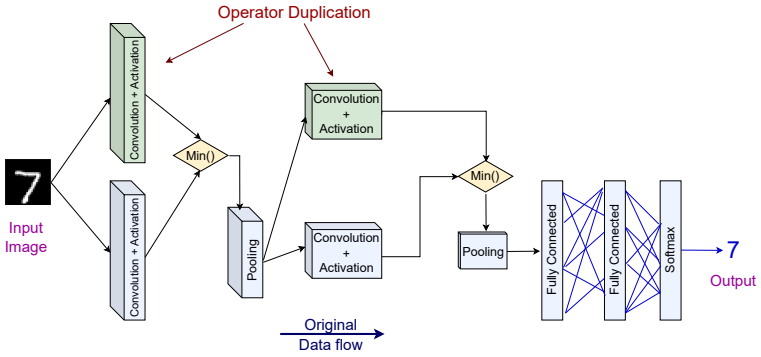
ACD is an optimization of AID that aims to reduce the number of comparison operations in the code. Referring to Figure 4.5b, ACD works by (1) statically identifying and duplicating the chain of arithmetic instructions and (2) adding the error detection and correction right after the chain. An arithmetic instruction chain consists of a sequence of arithmetic instructions within a basic block, such that the



(a) Arithmetic SID



(b) Arithmetic Chain Duplication



(c) Full Operator Duplication

Figure 4.5: Three different SID heuristics in our work. The nodes highlighted in blue represent the original data flow of the program, while those in green depict the duplicated nodes, and the ones in yellow belong to the comparison logic.

output of each instruction (except the last one) only affects the input of one or more arithmetic instructions within that chain.

Operator Duplication (OD)

OD works by fully duplicating the neural network's layer(s). As shown in Figure 4.5c, operators such as convolution and matrix multiplication are fully duplicated, followed by the comparison logic to compare the output of the original and the duplicated operators. We implemented operator duplication at the ONNX level instead of the LLVM IR level.

4.5.2 Error correction heuristics

Upon detecting a mismatch, we use two heuristics for error correction: *bitwise-and* and *return-min*. The former heuristic returns the *bitwise-and* of the two values while the *return-min* heuristic returns the minimum of the two values. These values are used in the subsequent computations. As noted by Li et al. [70], soft errors affecting the most significant bits have the highest probability of causing SDCs. Therefore, both the correction heuristics aim to prevent large, positive values from propagating through the CNN, thus preventing unnaturally large fluctuations in the neuron's output.

Note that for OD, we only use *return-min* correction heuristic as *bitwise-and* heuristic is not a standard operation in the ONNX framework[2] and, thus, is not supported by the ONNX-MLIR tool for conversion to the LLVM IR level.

4.5.3 Evaluation

Methodology

To evaluate the effectiveness of the SID extension of LLTFI, we used the six benchmarks from Table 4.1. Furthermore, we used the same evaluation metrics and setup described in Section 4.4.2. We performed each FI experiment 1000 times with LLTFI and reported the median values and error bars with 95% confidence intervals as before.

Results

The objective of our evaluation is to *evaluate the feasibility of SID for ML applications.* Towards this goal, we evaluate the trade-offs of SID concerning its resilience benefits (SDC Rate) and runtime execution overheads.

Figure 4.6 shows the SDC rate reduction percentage from the original, unprotected program of the ACD SID technique with *bitwise-and* and *return-min* correction heuristics. We observe similar SDC rates for AID SID, and hence, omitted it from Figure 4.6. The best-case reduction in the SDC rate for the benchmarks

[2]Following is the list of tensor operators supported by ONNX: https://github.com/onnx/onnx/blob/main/docs/Operators.md

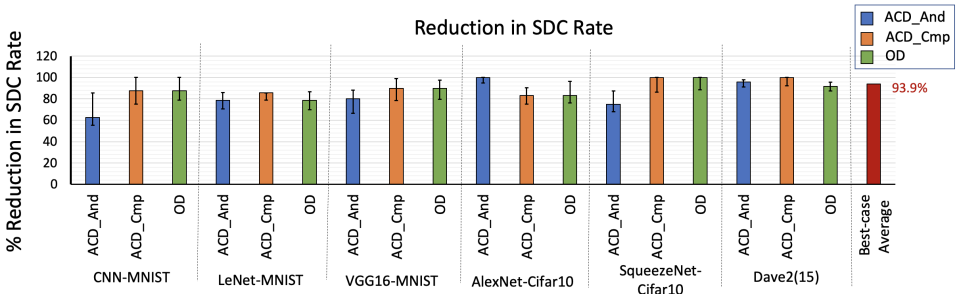


Figure 4.6: Percentage reduction in the SDC Rates for LLTFI with various SID techniques and correction heuristics compared to the unprotected program. *ACD_Cmp* and *ACD_And* are the shorthands used for ACD with *return-min* and *bitwise-and* heuristics respectively. *OD* stands for the operator duplication technique. SDC rates for AID and ACD were similar, hence, not shown separately. The bar in red shows the best-case average SDC reduction across all our benchmarks. Higher values are better.

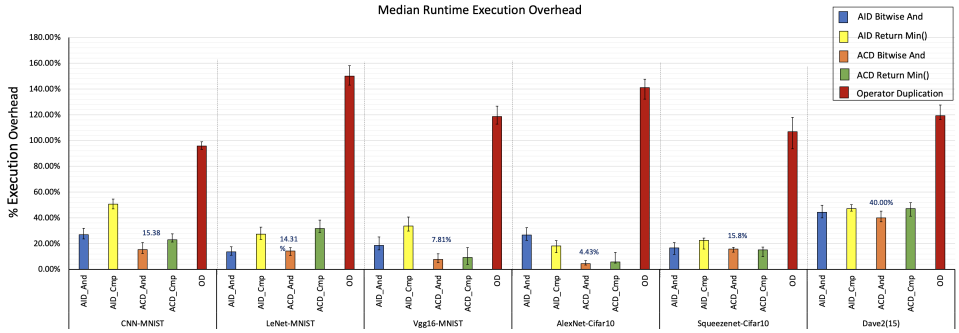


Figure 4.7: Runtime Execution Overheads for LLTFI with various SID techniques and correction heuristics. Shorthand notations are similar to Figure 4.6. The percentages are relative to the original, unprotected program. Lower values are better.

varies from 85.7% (for LeNet-MNIST) to 100% (for AlexNet and Dave2). On average, we observed a 93.9% best-case reduction in the SDC rate across all our benchmarks.

We also observed the lowest SDC rate for ACD SID with *return-min* correction heuristic, except for AlexNet-Cifar10. Moreover, the SDC rate reduction is comparable for OD and ACD with *return-min* correction heuristic.

To understand these results, we manually inspected the faults that caused the ML program to misclassify, despite our error correction heuristics. We made two observations.

1. For *return-min* correction heuristic, we observed that the majority of the escaped faults are NAN values. The IEEE-754 floating-point standard defines a NaN as a number with all ones in the exponent and a non-zero mantissa. Arithmetic instructions (except bitwise manipulations) and logic instructions operating on a NaN value will always result in a NaN value, and thus, these faults escaped our *return-min* correction heuristic.

2. For *bitwise-and* heuristic, interestingly, the NaN values got "corrected" by the bitwise-and operation because the bitwise-and of the original and the corrupted value prevents the situation with all exponent bits equal to one. However, unlike the *return-min* heuristic, which consists of only logical assembly instructions, *bitwise-and* heuristic also consists of arithmetic instruction (for bitwise `AND` operation), which itself is susceptible to soft errors. Thus, the faults that escaped *bitwise-and* correction heuristics are the ones that corrupted the `AND` operation itself.

Figure 4.7 shows the runtime overhead of AID, ACD SID techniques with *bitwise-and* and *return-min* correction heuristics. The best-case runtime overhead varies from 4.43% (for AlexNet) to 40% (for Dave2). Moreover, the least runtime overhead was observed for ACD with *bitwise-and* heuristic. This is because ACD reduces the number of comparison logic, and *bitwise-and* heuristic can be implemented with just a single assembly instruction. Moreover, we also noticed a significant reduction in runtime overheads with SID, as opposed to full operator duplication (proposed by Libano et al. [72]). Overall, these results suggest that for benchmarks like AlexNet and Vgg16, SID can significantly improve the error resilience (reduction in SDC rate by 90% and 100%) with only modest runtime overheads (7.8% and 4.4%).

For the Dave2 benchmark, we observed that the second convolution layer is very computationally expensive and takes about 52% of the total program runtime. Therefore, SID in this convolution layer resulted in an execution overhead of 40%.

Overall, SID achieves a high reduction in SDC rate with a modest performance overhead for most of our benchmarks.

4.6 Implications

Implication 1: Overreporting of SDCs Our experiments for RQ1 show that TensorFI reports a significantly (1.1x - 5.5x) higher SDC rate than LLTFI for the same types of hardware faults. *This is due to TensorFI's assumption that every bit flip fault corrupts a single layer output.* However, our experiments with LLTFI contradict this assumption, as we find that only a fraction of faults (about two out of five) ends up corrupting the layer output. These results potentially apply to other ML FI tools like PyTorchFI that inject faults in the layer outputs. Therefore, when using ML FI tools, one needs to scale down the SDC rate estimates to obtain realistic estimates. However, the amount of scaling is application specific.

Implication 2: Faults in layer outputs We also found that the majority of low-level faults result in multiple bit-flips in layer outputs. ML FI tools such as TensorFI primarily inject single-bit flip faults into layers' outputs. However, we found that low-level faults corrupted at most one value in a layer's output (RQ2), so the choices made by ML FI tools to corrupt a single value in the output layer are well justified.

Implication 3: Evaluation of low-level resilience enhancing techniques (like SID) for ML applications To the best of our knowledge, we are the first ones to evaluate the efficacy of SID for ML programs (Section 4.5). Although not exhaustive, our preliminary results are promising: for a few benchmarks like AlexNet and Vgg16, SID (in the convolution operator) significantly reduces the SDC rates (90%-100%) with modest runtime overheads (7.8% - 4.4%). LLTFI thus enables researchers to evaluate different SID heuristics and identify the instructions to duplicate. LLTFI is able to evaluate these and other low-level resilience improvement techniques [30] due to its visibility into instructions and registers, which is missing in most ML FI tools.

Overall Implications Prior research on improving the error resilience of ML applications [20] [21], [46] [90] [82] use high-level ML FI tools for evaluating their techniques. Our results suggest that there is a need to revisit the evaluation of these techniques. One of the main advantages of high-level ML FI tools is that they allow the abstraction of the faults to the ML framework's level and allow programmers to study the propagation of these errors at that level. However, LLTFI offers similar advantages as high-level ML FI tools without compromising the representativeness of the injected faults.

4.7 Summary

In this chapter, we proposed LLTFI, a framework-agnostic FI tool for ML applications. LLTFI works by converting ML models to the ONNX format, followed by further lowering them down to the LLVM IR level. At the LLVM IR level, LLTFI injects code instrumentations to inject transient faults during the program's runtime. LLTFI is highly configurable using a YAML file and also supports fault propagation tracing at two levels of granularity. We evaluated LLTFI with six popular ML programs and compared it to TensorFI, a high-level FI tool for ML programs. We find that TensorFI overreports the SDC rate of these programs for single bit-flip faults by 3.5X on average compared to LLTFI. Further, there are significant differences in fault propagation between the two tools. Finally, LLTFI is 27% faster than TensorFI on average. We also demonstrate the utility of LLTFI by extending it to perform SID for ML applications. We found SID to be an effective technique for improving the resilience of ML programs against transient hardware faults.

In the next chapter, we evaluate the resilience of LLMs using LLTFI.

Chapter 5

Resilience Assessment of Large Language Models

While all the prior work on resilience assessment of ML applications focuses solely on DNNs, in this chapter, we use LLTFI to also assess the resilience of LLMs against transient hardware faults.

We begin this chapter by first discussing our fault model, fault injection methodology, and the modifications we made to LLTFI for integrating with LLMs. Thereafter, we discuss the evaluation methodology and present the qualitative, quantitative evaluation of LLM's resilience. Finally, we conclude this chapter by presenting the effect of model size, pre-training, and fine-tuning objectives on the resilience of LLMs.

5.1 Fault Model

For this work, we used the same fault model as for LLTFI (refer Section 4.1). In a nutshell, we considered transient faults in the CPU's ALU and data paths. Moreover, we consider faults only during LLM's inference, not during the training. This is because training is a one-time process (for most LLMs), while inference is repeatedly invoked during an LLM's deployment.

5.2 Fault Injection Methodology

The primary challenge we faced while performing FI in LLMs was scalability. LLMs like CodeBert [38] have 124.6 Million parameters and execute approximately 30 Billion CPU instructions at runtime to produce output. Therefore, we had to modify LLTFI to make it work with LLMs, as it was not originally designed for this purpose.

1. FI Runtime We added a supplementary FI runtime that is essentially a stripped-down and highly-optimized version of LLTFI's original FI runtime. We then statically linked this runtime with the LLM's executable. Our supplementary FI runtime reduced FI overhead by about 25% (on average), thus making LLTFI more scalable.

2. Fault propagation tracing LLTFI supports ML fault propagation tracing that allows users to ascertain the origin, propagation, or suppression of the injected fault. However, LLTFI's original ML fault propagation tracing is not scalable to LLMs. Therefore, we replaced the element-wise output tensor comparison, i.e., element-wise comparison of each layer's output with and without FI, with hash-based comparison, where we just compare the hashes of the layer's output. While the hash-based comparison is scalable, it does not reveal the absolute amount of difference between the layer's outputs after FI. However, this optimization prevents us from ascertaining whether the fault corrupts just one element in the layer's output or multiple elements.

Figure 5.1 shows how we perform FIs using LLTFI in LLMs. The steps are as follows:

- **Step 1:** We first download the LLM from the Hugging Face model hub [6]. Hugging Face's Model hub contains a large collection of open-source pre-trained LLMs, along with their tokenizer and vocabulary files.
- **Step 2 and 3:** We also download the tokenizer and vocabulary files of the LLM that we use to convert human-readable text inputs to tokens in the Ten-

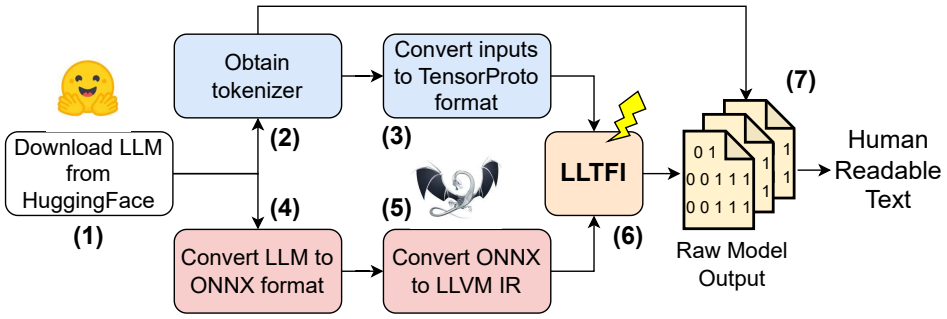


Figure 5.1: Fault Injection in LLMs using LLTFI.

sorProto format[1]. TensorProto defines a serialized format for representing multi-dimensional arrays of numerical data that we used for passing inputs and retrieving outputs from an LLM.

- **Step 4 and 5:** We then convert the downloaded LLM into the ONNX format, followed by lowering down to the LLVM IR, using the ONNX-MLIR tool, similar to the default working of LLTFI.

- **Step 6 and 7:** Subsequently, we feed the LLVM IR file and inputs of the LLM in TensorProto format to LLTFI, which executes various FI campaigns. The resulting output of each FI campaign is then transformed - through our custom scripts - into a human-readable format utilizing the tokenizer and vocabulary files of the LLM and thereafter compared with the ground truth.

5.3 Research Questions

In this thesis, we asked the following RQs to evaluate the resilience of LLMs under transient faults.

- **RQ1:** How do SDCs manifest in the LLMs?

- **RQ2:** How does the SDC probability vary across the different layers of the LLMs and the faulty bit position?

[1] https://onnx.ai/onnx/api/classes.html#tensorproto

- **RQ3:** How do the SDC rates of LLMs vary with different pre-training objectives, fine-tuning objectives, and the number of encoder/decoder blocks?

5.4 Evaluation Methodology

In this section, we describe our benchmarks, datasets, experimental design, metrics, and experimental setup.

5.4.1 ML Applications

For RQ1 and RQ2, we chose five benchmarks: PubMedBert [41], CodeBert [38], T5 [94], GPT2 [93], and Roberta [75], as shown in Table 5.1. These benchmarks vary in their architecture: three of them are Encoder-only, one Decoder-only, and one with Encoder-Decoder architecture. We chose these benchmarks based on their popularity in their respective domains, i.e., how many times they were downloaded in the last month at the time of writing. PubMedBert (100K) for medical-related fill-mask and question answers, CodeBert (200K) for code completion in Java, T5 (6 Million) for language translation, Roberta (11 Million) for sentiment analysis, and GPT2 (21 Million) for text generation.

For RQ3, we used 15 benchmarks from Alajrami et al.'s work [12] on evaluating the effect of different pre-training objectives on LLM's learning capabilities. All of these benchmarks are variants of *Bert-base*, *Bert-small*, and *Bert-medium*, and are pre-trained with five different objectives each, namely (1) Masked Language Modeling [28], (2) Manipulated word detection [113], (3) Masked first character prediction [113], (4) Masked ASCII code summation [12], and (5) Masked Random token classification [12]. We explain these objectives in Section 5.5.4.

For this work, we further fine-tuned these benchmarks on two tasks: fill-mask and question-answers, thereby resulting in a total of 30 distinct benchmarks.

5.4.2 Datasets

We used the pre-trained, publicly-available models of PubMedBert and CodeBert, trained with the PubMed [92] and CodeSearchNet [50] datasets, respectively. T5 was first pre-trained on the C4 dataset (using an unsupervised denoising objective) and then trained again on CoLa [110], MNLI [112], and 12 other datasets (using

supervised training objectives). Roberta was pre-trained on Stories [105] and four other datasets. GPT2 was primarily trained on the WebText dataset [74], which consists of about 40GB of text.

	Benchmark Name	Architecture	Task type	# of Parameters
RQ1 & RQ2	PubMedBert [41]	Encoder-Only	Fill-mask	109.5M
	CodeBert [38]	Encoder-Only	Fill-mask	124.6M
	T5 [94]	Encoder-Decoder	Translation	222.9M
	Roberta [75]	Encoder-Only	Sentiment Analysis	124.6M
	GPT2 [93]	Decoder-Only	Text Completion	124.4M
RQ3	Bert-base*	Encoder-Only	Fill-Mask, Question Answer	108.3M
	Bert-medium*			51.5M
	Bert-small*			38.9M

Table 5.1: Benchmarks used for resilience evaluation of LLMs. Benchmarks marked with * are compiled with five different pre-training objectives.

Benchmarks used in RQ3 are primarily pre-trained on BookCorpus [118] and English Wikipedia. We further fine-tuned these models using Eli5 [36] and SQuAD [96] datasets for the fill-mask and question-answer tasks, respectively.

5.4.3 Experimental Design

For each of the five benchmarks used for RQ1 and RQ2, we used ten distinct inputs from their training dataset, while for each of the 30 benchmarks used in RQ3, we used five distinct inputs in order to keep our FI experiments computationally tractable. For each input, we ran 1000 FI experiments. For the T5 benchmark, we performed FI separately for the encoder and decoder in order to explore the resilience of each component. Therefore, throughout this work, we use the term "T5-encoder" and "T5-decoder" to denote T5 with FI in the encoder and decoder, respectively. This resulted in a total of $210,000$ FI campaigns for all our exper-

iments, *which took approximately 336 Hours of CPU time*. Note that the model weights of all of our benchmarks are static, i.e., in the absence of FI, the input-output mapping of our benchmarks is constant across the FI trials.

Similar to Chapter 4, in this work, we injected only single bit-flip faults in LLMs. Additionally, to avoid application crashes, we introduced errors only in the operands and outcomes of floating-point arithmetic as well as logic instructions such as FADD, FMUL, and FCMP (for the PubMedBert benchmark, these instructions constitute *15%* of the total CPU instructions). This FI methodology ensured that the control flow of the LLM remained uncorrupted - this is in line with our fault model (Section 5.1).

To inject a single bit-flip fault in an LLM, we first randomly select a layer, followed by randomly selecting a floating-point arithmetic instruction at runtime to inject the fault into, and finally, we randomly select a bit in the operands or results of the chosen instruction to inject the bit-flip fault.

5.4.4 Metrics

To evaluate LLMs' resilience, we used two metrics: (1) SDC rate and (2) cosine similarity. An SDC is a misprediction (the output of the LLM differs from the correct one). On the other hand, cosine similarity is a popular metric in the NLP community [59, 67] to compare the similarity between document vectors or word embeddings. While the SDC rate measures how frequently the output differs from the correct prediction, cosine similarity measures how distant the corrupted output is from the correct one due to the injected fault.

To calculate cosine similarity, we converted the text output of the LLM into vector embeddings using the FastText [54] embeddings. For T5 (translation of English-to-German and English-to-French), we used FastText's German and French variants, publicly available in FlairNLP [11]. For the CodeBert benchmark, since the output of this benchmark is Java code, we could not use the cosine similarity metric due to the lack of open-source Java word embeddings. Similarly, for Roberta, the output is just 'Positive' or 'Negative', so we could not use the cosine similarity metric for this benchmark.

For both the SDC rate and the cosine similarity, we calculate values using 95% confidence intervals. Since SDCs are rare events, for the SDC rate, we used the confidence intervals proposed by Leveugle et al. [65], which is in line with prior work on FI [32, 69]. For cosine similarity, we calculate the non-parametric 95% confidence intervals.

Limitations of our metrics. SDC rate, i.e., strict word-to-word comparison of the corrupted and the correct output, might not capture the polymorphic nature of natural languages, where we can express the same concept using different words. While cosine similarity of word embeddings can capture the polymorphic nature of natural languages upto a large extent, it, too, can not fully capture the contextual nuances and ambiguities present in natural language. For example, the cosine similarity of *"This chair is white and the table is black."* and *"This chair is white and the table is white too"* (example taken from our GPT2 benchmark) is over 90%, indicating a near-perfect match when in reality, these two sentences have completely different meanings.

To address these limitations, in Section 5.5.2, we present a qualitative categorization of SDCs, based on their syntactical and semantical differences relative to the correct output.

5.4.5 Setup

We ran all our experiments on a 64-bit AMD Ryzen Threadripper 3960X 24-Core Processor with three NVIDIA RTX A5000 GPUs.

5.5 Results

In this section, we first present the SDC rates of our benchmarks. We then present the results based on the RQs.

5.5.1 SDC Rates

Table 5.2 shows the SDC rates and cosine similarity metrics for all of our benchmarks. Overall, the average SDC rate across our benchmarks varied from 0.002 (in the case of Roberta) to 0.019 (for the T5 decoder), with the average being 0.009. Regarding cosine similarity, we observed that for PubMedBert and GPT2, the aver-

Benchmark Name	SDC Rate and Cosine Similarity (CS)											
	Input 1	Input 2	Input 3	Input 4	Input 5	Input 6	Input 7	Input 8	Input 9	Input 10	Average	
PubMedBert	0.012 ± 0.006	0.005 ± 0.004	0.002 ± 0.002	0.015 ± 0.007	0.004 ± 0.003	0.010 ± 0.006	0.007 ± 0.005	0.005 ± 0.004	0.006 ± 0.004	0.011 ± 0.006	0.008	**SDC**
	0.27 ± 0.05	0.26 ± 0.06	0.28 ± 0.05	0.25 ± 0.07	0.27 ± 0.06	0.29 ± 0.05	0.15 ± 0.08	0.28 ± 0.05	0.15 ± 0.07	0.22 ± 0.12	0.24	**CS**
CodeBert	0.004 ± 0.003	0.003 ± 0.003	0.004 ± 0.003	0.005 ± 0.004	0.002 ± 0.0027	0.003 ± 0.003	0.007 ± 0.005	0.003 ± 0.003	0.003 ± 0.003	0.004 ± 0.003	0.004	**SDC**
	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	**CS**
T5 encoder	0.013 ± 0.007	0.010 ± 0.006	0.009 ± 0.005	0.015 ± 0.007	0.026 ± 0.009	0.005 ± 0.004	0.005 ± 0.004	0.014 ± 0.007	0.007 ± 0.005	0.015 ± 0.007	0.012	**SDC**
	0.87 ± 0.13	0.76 ± 0.17	0.68 ± 0.25	0.80 ± 0.15	0.86 ± 0.09	0.86 ± 0.26	0.66 ± 0.30	0.85 ± 0.13	0.61 ± 0.24	0.70 ± 0.18	0.76	**CS**
T5 decoder	0.019 ± 0.008	0.029 ± 0.01	0.017 ± 0.008	0.017 ± 0.008	0.016 ± 0.007	0.018 ± 0.008	0.018 ± 0.008	0.023 ± 0.009	0.016 ± 0.007	0.019 ± 0.008	0.019	**SDC**
	0.85 ± 0.02	0.83 ± 0.02	0.85 ± 0.017	0.87 ± 0.028	0.84 ± 0.02	0.85 ± 0.018	0.85 ± 0.04	0.86 ± 0.017	0.84 ± 0.04	0.85 ± 0.06	0.85	**CS**
GPT2	0.015 ± 0.007	0.019 ± 0.008	0.007 ± 0.005	0.019 ± 0.008	0.007 ± 0.005	0.007 ± 0.005	0.005 ± 0.004	0.006 ± 0.004	0.006 ± 0.005	0.005 ± 0.004	0.010	**SDC**
	0.59 ± 0.18	0.30 ± 0.06	0.19 ± 0.08	0.29 ± 0.04	0.44 ± 0.01	0.44 ± 0.01	0.50 ± 0.15	0.21 ± 0.01	0.27 ± 0.25	0.35 ± 0.04	0.36	**CS**
Roberta	0.002 ± 0.0027	0.001 ± 0.002	0.003 ± 0.003	0.002 ± 0.0027	0.004 ± 0.003	0.002 ± 0.0027	0.006 ± 0.004	0.000 ± 0	0.001 ± 0.002	0.002 ± 0.0027	0.002	**SDC**
	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	**CS**

Table 5.2: SDC rates and cosine similarities of all our benchmarks. Higher cosine similarity indicates that the corrupted output is closer to the correct one.

age cosine similarity is 0.3, indicating that the corrupted output differs significantly from the correct one. On the other hand, for T5-encoder and T5-decoder, the average cosine similarity is 0.8, indicating that the corrupted output is semantically very similar to the correct output (see Section 5.5.2 for more details).

We make two observations from the results.

Observation 1: Large variance of SDC rate for different inputs of the same model For PubMedBert, the SDC rate varies from 0.002 to 0.015 (approx. seven times) for different inputs. Similarly, for T5-encoder and T5-decoder, the SDC rate varies from 0.005 to 0.026 and from 0.016 to 0.029, respectively. Thus, the SDC rate varies widely across inputs.

Observation 2: Large variance of SDC rate for different benchmarks with the same architecture Even though PubMedBert, Roberta, and CodeBert, have encoder-only LLM architecture with the same number of transformer blocks, their SDC rates vary from 0.008 (for PubMedBert) to 0.002 (for Roberta). This observation indicates that, apart from the LLM architecture, other factors like training data set, training objectives, fine-tuning task, etc., also play an essential role in determining the SDC rates, thus prompting us to ask RQ3.

5.5.2 RQ1: Qualitative categorization of SDCs in LLMs

In this RQ, we delineate how the SDCs manifest in different LLMs by providing examples of SDCs, along with a qualitative categorization of SDCs, depending on whether they are syntactically incorrect, semantically incorrect, or semantically

Benchmark Name	Total SDCs	Syntactically-incorrect SDCs	Semantically-incorrect SDCs	Semantically-correct SDCs
PubMedBert	77	46	14	17
CodeBert	38	35	3	0
T5-encoder	119	42	29	48
T5-decoder	192	154	24	14
GPT2	103	45	49	9
Roberta	23	0	23	0

Table 5.3: A qualitative categorization of SDCs.

correct with respect to the original output of the LLM. The objective of this qualitative categorization is to augment our quantitative analysis in Table 5.2.

For this analysis, we manually analyzed all the corrupted outputs of the LLMs (552 SDCs, across all benchmarks and their inputs) and categorized them as syntactically incorrect, semantically incorrect, or semantically correct. We consider the output of the LLM to be syntactically incorrect if the corrupted output does not form grammatically correct text in the target language. We consider the output to be semantically incorrect if the corrupted output is syntactically correct but its meaning differs from that of the original text. Finally, we consider the output to be semantically correct if it is syntactically correct, *and* its meaning is similar to that of the original text, based on our subjective evaluation.

Table 5.3 shows the categorization of SDCs across all benchmarks. We explain the results for each benchmark below.

PubMedBert

For the PubMedBert benchmark, among the 77 SDCs observed across all inputs (out of 10,000 FI campaigns), 46 are syntactically incorrect, 14 are semantically incorrect, and the remaining 17 are semantically correct. The following is an example of the semantically incorrect output (mistakes are underlined in the examples below):

Effect of transient faults on PubMedBert (fill-mask)
Input: The hereditary [MASK] protein, HFE, specifically regulates transferrin-mediated iron uptake in HeLa cells.
Correct Output: The hereditary plp protein, HFE, specifically regulates transferrin-mediated iron uptake in HeLa cells.
Corrupted Output: The hereditary myb protein, HFE, specifically regulates transferrin-mediated iron uptake in HeLa cells.

Here, the model predicts the *myb* protein instead of the *plp* protein, which is incorrect in this context. Similarly, we observed cases where the model, under the effect of the fault, predicted the wrong disease and diagnosis, which are semantically different from the correct output. However, for syntactically incorrect outputs, the model ends up predicting out-of-context characters, including non-English language characters, punctuation marks, numbers, etc. The following is an example of the syntactically incorrect output: "Main symptom of common flu is ##合.", where the model outputs out-of-context Chinese (Hanzi) characters.

T5-decoder

For T5-decoder, among the 192 observed SDCs, 154 are syntactically incorrect, 24 are semantically incorrect, and 14 are semantically correct. The following shows an example of each type of SDC for this application:

Effect of transient faults on T5-decoder (translation)
Input: translate English to French: The House rose and observed a minute's silence
Correct Output: L'Assemblée se levera et observera une minute de silence
Semantically-correct Output #1: Le Parlement se levera et observera une minute de silence
Semantically-incorrect Output #2: zaharie a eu l'occasion de s'exprimer
Syntactically-incorrect Output #3: zügliche de l'Assemblée

For most semantically-correct SDCs, the model either ended up using synonyms of words of the original text (like in the example above) or just paraphrasing the correct output. In the semantically incorrect SDCs, the model outputs a coherent text in the target language, but its meaning differs significantly from the correct text. For instance, in the example above, the model outputs a "zaharie a eu l'occasion de s'exprimer", which translates to "Zaharie had the opportunity to

express himself " in English, which differs from the original text (*"The house rose and observed a minute's silence"*).

For syntactically incorrect SDCs, along with illegible outputs, we also observed many cases where some part of the output is in a different language than the rest of the text. For instance, in the above example, the model outputs "zügliche de l'Assemblée" where "zügliche" is a German word (for 'prompt') while the rest of the sentence is in French. In another interesting example, the model outputs "Ich aparitie im Namen des Europäischen Parlaments" as the German translation of "I would like, on behalf of the European Parliament,". While the rest of the text is in German, the word "aparitie" is Romanian and means "to appear or appearance" - this word fits correctly in the context of this text.

T5-encoder

For T5-encoder, among the 119 total SDCs we observed, 42 are syntactically incorrect, 29 are semantically incorrect, and the remaining 48 are semantically correct. Thus, unlike other benchmarks (except Roberta), SDCs in the T5 encoder are more likely to result in syntactically correct outputs. The following are examples of semantically correct and incorrect outputs in this application:

Effect of transient faults on T5-encoder (translation)
Input: translate English to French: I should like, on behalf of the European Parliament, to express our sympathy to the parents and families of the victims.
Correct Output: Au nom du Parlement européen, je voudrais exprimer notre sympathie aux parents et aux familles des victimes.
Semantically-correct Output #1: Au nom du Parlement européen, je voudrais exprimer notre sympathie aux parents des victimes.
Semantically-incorrect Output #2: Au nom du Parlement européen, je voudrais exprimer notre gratitude aux parents et aux familles des victimes.

Notice that in the semantically-incorrect output shown in the above example, the model uses the word "gratitude" instead of "sympathy", which drastically changes the meaning of the original text. We have observed similar cases where the model's output is very close to the correct text but is still semantically incorrect. For example, the model outputs "Ich erkläre die am Freitag, dem

12. Dezember 2000, unterbrochene Sitzungsperiode des Europäischen Parlaments" as the German translation of "I declare resumed the session of the European Parliament adjourned on Friday, 15 December 2000." Notice how the model subtly changes the date in the original prompt due to the fault, making it difficult to catch.

Roberta

The output of Roberta is either "Positive" or "Negative," depending on the sentiment of the input text. We did not observe any syntactically-incorrect output. Therefore, all the SDCs are semantically incorrect by definition. The following is an example of the semantically-incorrect output:

Effect of transient faults on Roberta (sentiment analysis)
Input: I really like the new design of your website!
Correct Output: Positive
Semantically-Incorrect Output: Negative

CodeBert

For CodeBert (Java code completion), among 38 SDCs observed, we found 35 of them to be syntactically incorrect, i.e., the resulting Java code is syntactically wrong. Only three SDCs are semantically incorrect, and none of them is semantically correct. The following shows an example of a semantically incorrect SDC in this application.

Effect of transient faults on Codebert (fill-mask)
Input:
protected Iterator $\langle Map.Entry\langle K,V\rangle\rangle$ createEntrySetIterator () {
if (size() **[MASK]** 0) {
return EmptyIterator.INSTANCE; }
return new EntrySetIterator $\langle K,V\rangle(this)$;
}
Correct Prediction: ==
Semantically-incorrect Output: >=

In this example, the model predicts a wrong binary operator (>= instead of ==), which is semantically incorrect as it could modify the functionality of the

program by returning an empty iterator even if the `Map` has a non-zero size. The low SDC rate (0.004, on average) for this benchmark and the high number of syntactically incorrect SDCs imply that CodeBert is quite resilient to transient faults: even in the case of SDCs, the corrupted output would be easier to detect.

GPT2

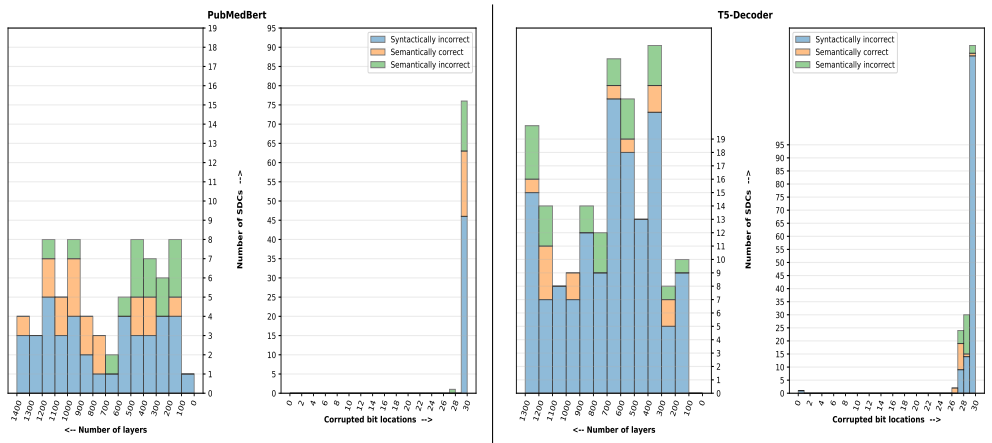
Unlike other benchmarks, which are used for text translation (T5), fill-mask (PubMedBert and CodeBert), or sentiment analysis (Roberta), GPT2 is used for text generation, and therefore, there is no objective ground truth for GPT2. Therefore, to categorize SDCs, we compare the SDC, syntactically and semantically, with the model's output in the absence of FI. Additionally, we truncated the output of GPT2 to one sentence for readability.

Among the 103 observed SDCs, we found 45 of them to be syntactically incorrect, i.e., they do not form a coherent English sentence, 49 are semantically incorrect, and nine are semantically correct (relative to the model's output in the absence of FI). The following example shows two of the semantically incorrect outputs we observed.

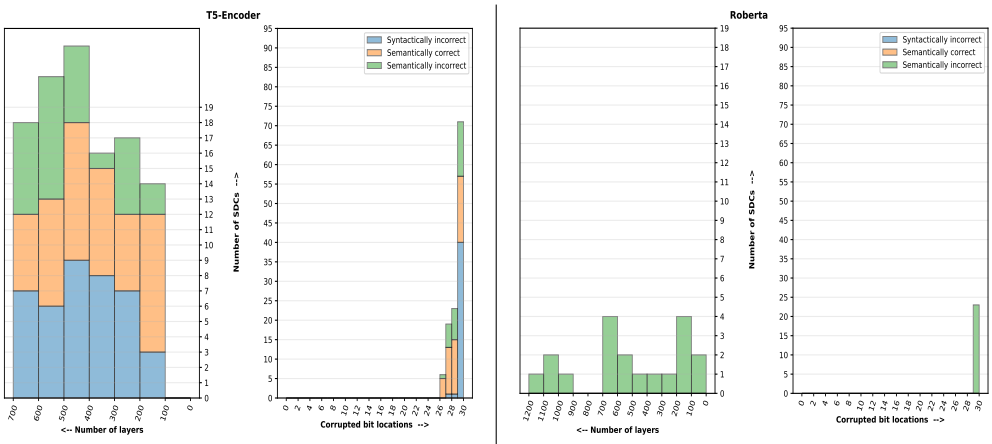
Effect of transient faults on GPT2 (text completion)
Input: This chair is white and the table is
Correct Output: This chair is white and the table is black
Semantically-incorrect Output #1: This chair is white and the table is white too.
Semantically-incorrect Output #2: This chair is white and the table is lawmakers and the media who claim you are the boss of Rep.

5.5.3 RQ2: Distribution of SDCs across layers of LLMs and faulty bit position

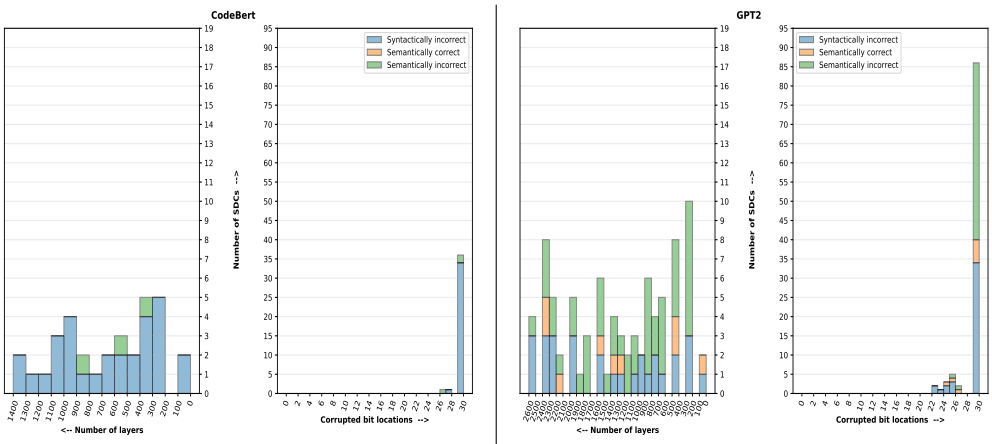
Figure 5.2 shows the distribution of SDCs across LLM's layers and bit positions across our benchmarks. The subfigure for each benchmark in Figure 5.2 shows two stacked bar plots: one highlighting the distribution of SDCs along the layers of the LLM and the other highlighting the distribution of SDCs in different bit locations of the fault.



(a) PubMedBert **(b)** T5 Decoder



(c) T5 Encoder **(d)** Roberta



(e) CodeBert **(f)** GPT2

Figure 5.2: Distribution of SDCs across layers of the LLMs and the bit position (in a 32-bit float value) in which the fault is injected. The subfigure for each benchmark shows two plots: The number of SDCs (Y-axis) vs. the layer in which the fault is injected (- X-axis) and the Number of SDCs (Y-axis) vs. the bit position in which the fault is injected (+ X-axis).

For all our benchmarks, the layers of LLMs (and their corresponding transformer blocks) are equally sensitive to transient hardware faults. Prior work [9, 68] on FI in CNNs has found the first and last layers of DNN to be more sensitive to transient faults. However, we do not observe a similar trend for LLMs.

Regarding the distribution of SDCs along different bit positions, in general, we found the higher-order bits to be significantly more sensitive to transient faults (this was the case for CNNs as well [68, 69]). Since all our benchmarks use Float32 datatype, higher-order bits (24 to 31) correspond to the exponent of the floating-point value, and therefore, a bit-flip in higher-order bits alters the exponent values.

Moreover, we used LLTFI's ML fault propagation tracing to understand why some transient faults get suppressed while others result in an SDC. We observe that only the 0-to-1 bit flips in higher-order bits, which increase the value of the exponent, result in an SDC. However, none of the 1-to-0 bit flips in the exponent resulted in an SDC. Additionally, we observe that faults in lower-order bits often get suppressed, either by multiplication with zero and near-zero values or by the normalization layers of the transformer blocks.

Finally, from Figure 5.2, we did not observe any discernible relationship between the category of SDC (syntactically incorrect, semantically incorrect, and semantically correct) and the bit location of the injected fault, across all benchmarks, except T5. For T5-encoder and T5-decoder, we observe that transient faults in 26 - 29 bits are more likely to result in a syntactically-correct SDC.

5.5.4 RQ3: Variation of SDC rates with different pre-training objectives, fine-tuning objective, and the number of encoder/decoder blocks

Table 5.4 shows the variation of SDC rates with different pre-training, fine-tuning objectives, and the size of LLMs (number of encoder/decoder blocks). For this RQ, we used three different sizes of the Bert LLM: *Bert-Small* with four encoder blocks, *Bert-medium* with eight encoder blocks, and *Bert-Base* with 12 encoder blocks. We further used two fine-tuning objectives: Fill-mask (fill-in-the-blanks) and Question Answer, and pre-trained the models with the following five different pre-training objectives each:

Fine-tuning Objective ->	Mask-Fill						Question-Answer					
Pretraining Objective	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**	**Average**	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**	**Average**
	Bert - Small											
Bert - MLM	0.004	0.002	0.002	0.002	0.001	**0.0022**	0.011	0.007	0.006	0.007	0.006	**0.0074**
Bert - S+R	0.0	0.003	0.002	0.003	0.002	**0.002**	0.009	0.01	0.006	0.004	0.011	**0.008**
Bert - First Char	0.001	0.004	0.004	0.004	0.004	**0.0034**	0.005	0.004	0.006	0.008	0.003	**0.0052**
Bert - ASCII	0.0	0.003	0.004	0.004	0.002	**0.003**	0.008	0.005	0.008	0.004	0.005	**0.006**
Bert - Random	0.0	0.003	0.004	0.004	0.004	**0.003**	0.008	0.010	0.009	0.011	0.004	**0.084**
	Bert - Medium											
Bert - MLM	0.0	0.008	0.008	0.003	0.004	**0.005**	0.005	0.007	0.006	0.004	0.003	**0.005**
Bert - S+R	0.003	0.005	0.001	0.005	0.003	**0.003**	0.011	0.010	0.003	0.011	0.007	**0.008**
Bert - First Char	0.0	0.004	0.006	0.005	0.007	**0.0044**	0.010	0.007	0.006	0.010	0.006	**0.008**
Bert - ASCII	0.003	0.008	0.005	0.006	0.003	**0.005**	0.004	0.004	0.007	0.007	0.015	**0.007**
Bert - Random	0.0	0.0	0.004	0.006	0.003	**0.003**	0.009	0.007	0.008	0.005	0.006	**0.007**
	Bert - Base											
Bert - MLM	0.009	0.008	0.008	0.006	0.006	**0.007**	0.006	0.005	0.013	0.010	0.009	**0.0086**
Bert - S+R	0.011	0.009	0.005	0.006	0.010	**0.008**	0.006	0.006	0.012	0.009	0.007	**0.008**
Bert - First Char	0.013	0.008	0.006	0.006	0.003	**0.007**	0.008	0.007	0.006	0.01	0.01	**0.008**
Bert - ASCII	0.008	0.008	0.003	0.014	0.003	**0.007**	0.009	0.008	0.005	0.009	0.01	**0.008**
Bert - Random	0.003	0.005	0.002	0.004	0.003	**0.0034**	0.012	0.007	0.004	0.008	0.01	**0.0082**

Table 5.4: Variation of SDC rates with pre-training objectives, fine-tuning objectives, and number of encoder/decoder blocks.

1. Masked Language Modeling (MLM) [28], which randomly chooses tokens from the input and replaces them either with the *[MASK]* token or with a random token.

2. Manipulated word detection (S+R) [113], which randomly chooses tokens from the input and either replaces them with random tokens or with shuffled tokens from the same input.

3. Masked first character prediction (FC) [113], where the model is trained to predict just the first character of the masked token.

4. Masked ASCII code summation (ASCII) [12], where the model is trained to predict the summation of the ASCII codes of the masked token.

5. Masked Random token classification (Random) [12], where tokens from the input sequence are randomly masked into five different classes, and the model has to predict the class of the masked token. This objective prevents the model from learning any meaningful linguistic information.

Effect of model size on SDC rates: For the fill-mask fine-tuning objective, we observe an increase in the SDC rate with the model size (except for the *Bert-Random* pre-training objective): with *Bert - Small* the average SDC rate was 0.0026 across all inputs, for *Bert - Medium* the average SDC rate was 0.004, and for *Bert*

- *Base* the average SDC rate was 0.007. However, for the Question-Answer fine-tuning objective, we do not observe any relation between the SDC rate and model size.

Effect of fine-tuning objectives on SDC rates: For *Bert-Small*, we observe a significant increase ($8\times$) in the SDC rate for the question-answer fine-tuning objective compared to the mask-fill objective. However, for *Bert-Medium* and *Bert-Base*, the difference in the SDC rates between the two fine-tuning objectives is small ($1.7\times$ and $1.25\times$ for *Bert-Medium* and *Bert-Base*, respectively).

We investigate why the fill-mask fine-tuning objective has a lower SDC rate than the question-answer objective. In the case of the fill mask, we found that a large portion of faults (in higher-order bits) propagates to the final layer but gets suppressed by the model's tokenizer. For fill-mask, the model outputs N tokens, where N is the vocabulary size (28K), V, multiplied by the number of tokens, T, in the input text. The tokenizer, however, masks $V \times (T-1)$ tokens, thus masking potential SDCs.

Effect of pre-training objectives on SDC rates: We do not observe any relationship between the SDC rates and the pre-training objectives. However, for the *Bert-Random* objective, we observe that the SDC rate is the same for a given fine-tuning objective, irrespective of the model size. This is likely because in the *Bert-Random* objective, the model does not learn any linguistic features of the language, and thus, increasing the model size does not help the model learn model new linguistic features.

5.6 Summary

In this chapter, we experimentally evaluate the resilience of LLMs under transient hardware faults. We used LLTFI to inject transient faults into LLMs, and measure their SDC rates, along with the cosine similarity between the correct and the corrupted outputs. Based on extensive FI experiments, we found LLMs to be quite resilient under hardware transient faults, with an average SDC rate being 0.9%

While in this chapter, we evaluated the resilience of LLMs against transient hardware faults in the CPU, in the next chapter, we will assess the impact of hardware faults in ML accelerators like Google's TPU.

Chapter 6

Reliability Assessment of Systolic Arrays against Stuck-at Faults

The increasing computation requirement of training LLMs and DNNs has motivated the use of ML accelerators like Google's TPU [3] and Nvidia's NVDLA [102]. To further improve energy efficiency and reduce the execution overheads, ML accelerators like Google's TPU use a 2-D array of MAC units, called a *systolic array* at their core for batch multiplication and addition operations.

In this chapter, we evaluated the resilience of systolic arrays against permanent stuck-at hardware faults using an RTL-based FI tool we developed called SystoliFI. We begin this chapter by first discussing our fault model, design constraints, and implementation of SystoliFI. Afterward, we discuss our research questions, evaluation methodology, and FI results. Finally, we conclude this chapter by discussing our results and their implications.

6.1 Fault Model

In this chapter, we used the single stuck-at-fault model to evaluate the effect of permanent faults in the MAC units of the systolic array. Stuck-at faults in systolic arrays can be caused by various factors, including physical damage to the hardware, defects in the manufacturing process, and wear and tear over time [115].

Among all the permanent fault models, the stuck-at fault model is fabrication technology independent. Moreover, even though a single stuck-at fault does not accurately model some physical defects, the tests derived for single stuck-at faults are still valid for most defects, including multiple stuck-at faults [83].

In addition, we make the following assumptions.

1. We do not consider faults in the memory elements as they can be protected with ECC.

2. We only consider faults in the data path as they can pass by silently and lead to wrong output without being detected [56].

3. We focus on faults in the MAC units as they make up most of the hardware area of the data path in the systolic arrays.

Our fault model is in line with prior work [58][116] as well.

6.2 Design Constraints

We had the following constraints for the design of SystoliFI:

- **Need for RTL-based FI.** Unlike transient faults in CPU whose manifestation at the software level is well-known (for example, bit flips in registers [78]), the effect of permanent stuck-at faults in ML accelerators at the software level is not known. Due to this, in this work, we could not use a software-based FI tool like LLTFI. Moreover, another motivation for opting for an RTL-based FI is to take into consideration all the hardware peculiarities of ML accelerators, like data reuse and parallelism, thereby resulting in a more accurate FI tool.

- **Configurability and the ease of FI.** Since a systolic array consists of a 2-D array of MAC units, for a systematic evaluation of the systolic array's resilience, the user should be able to easily configure the location of the faulty MAC unit without the need for recompiling SystoliFI's RTL model, which is computationally-expensive.

- **Compatibility.** The FI tool should be compatible with ML models written in Python using different ML frameworks. This requires SystoliFI to have a software stack consisting of a Python interpreter, an operating system, and a custom compiler to compile the ML programs for the custom RTL model of the systolic array.

- **Scalability.** SystoliFI should be scalable enough to run FI experiments on large DNNs exhaustively. This will allow us to experiment with different hardware and software configurations like data mapping schemes.

6.3 SystoliFI's Implementation

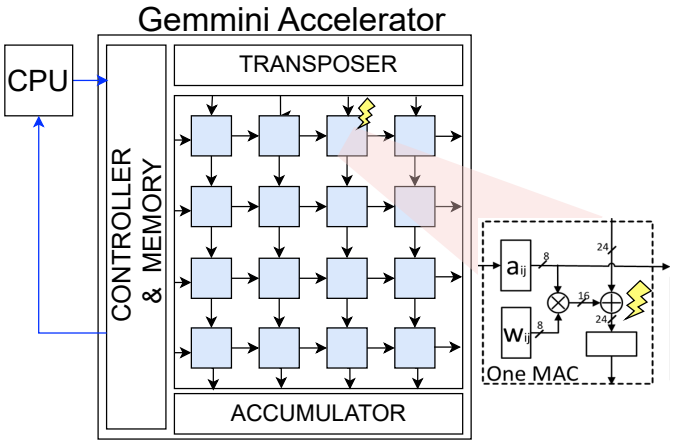


Figure 6.1: Our simplified FI setup. To do FI, we inject a stuck-at fault in a randomly-selected MAC unit.

Our RTL-based FI framework, SystoliFI, extends Gemmini [39] - a popular systolic array generator for evaluating deep learning accelerator - to carry out FI in the processing elements. Gemmini is an open-source, full-stack DNN accelerator generator for DNN workloads, enabling end-to-end, full-stack implementation and evaluation of custom hardware accelerators. Figure 6.1 shows Gemmini's architecture and our fault injection setup.

Along with the systolic array, Gemmini has hardware implementation of ReLU, a DNN accelerator controller, scratchpad memory, and a single-core CPU called Rocket to send commands to the Gemmini DNN accelerator and (un)load data into the weight buffers. Gemmini is written in Chiesel, a scala-based hardware description language, and uses the RISC-V hardware development toolchain along with the RISC-V software stack.

We chose Gemmini because it is highly configurable: we can configure the size of the systolic array, data type, and type of data mapping scheme (OS vs. WS). The only limitation of Gemmini is scalability. Since it is a full-stack simulator, it takes considerable time to simulate a large DNN workload. For instance, during our preliminary experimentation, we found that the simulation of the entire ResNet50 network took several days. To reduce the simulation time of Gemmini, we instead synthesized SystoliFI's RTL model on industrial-grade FPGAs instead of using software-based FPGA simulators.

For each FI experiment, we inject a single stuck-at fault in the intermediate signals of the MAC unit right after the addition logic and before the result is stored in the accumulator. Unlike Zhang et al. [116] that injected multiple stuck-at faults, we chose to inject only a single stuck-at fault because prior work has shown that the single stuck-at fault model is sufficient to detect 98% of five or fewer multiple stuck-at faults [10, 43].

6.3.1 Extending Gemmini for FI

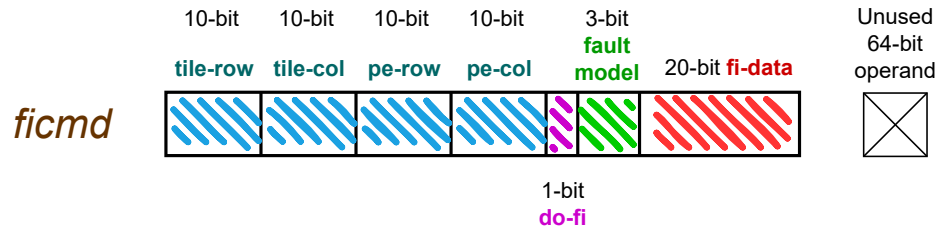


Figure 6.2: `ficmd` instruction

For FI, we extended Gemmini's custom RISC-V ISA to add an additional FI instruction `ficmd` that takes two 64-bit operands. As shown in Figure 6.2, we only use the first 64-bit operand to encode the following FI information:

- `tile-row`, `tile-col`: x and y coordinate of the tile for FI. In Gemmini, a "tile" is a group of MAC units that execute simultaneously.
- `pe-row`, `pe-col`: x and y coordinate of the MAC unit within the chosen tile for FI.
- `do-fi`: a boolean variable to indicate whether to do FI or not.
- `fault model`: 3-bit variable to specify the fault model. Currently, SystoliFI support stuck-at-0, stuck-at-1, single-bit-flip, and double-bit-flip fault model. However, for this thesis, we just used the stuck-at-0 fault model.
- `fi-data`: 20-bit variable to pass additional data for FI. For instance, this variable can be used to specify the exact bit location in which the fault has to be injected.

Having a custom instruction, `ficmd`, allows us to manage fault injection at runtime from the software without having to recompile the hardware's RTL model, which is computationally expensive. Additionally, to support `ficmd`, we also made some non-intrusive changes in Gemmini's hardware, like the addition of extra registers to hold the user-supplied FI configuration and additional circuitry for FI in MAC units of the systolic array.

6.4 Research Questions and Challenges

In this work, we aim to identify the spatial distribution of the software faults in the intermediate layers of the ML models, referred to as fault patterns, arising due to stuck-at faults in the MAC units of the systolic array.

Toward this objective, we use an RTL-level FI framework (described in Section 6.3) to simulate systolic arrays and do FI at runtime. Specifically, we ask the following RQs:

- How do the fault patterns change with different:

1. **RQ1:** data flow mapping schemes (OS and WS)? Are some data flows more fault tolerant?
2. **RQ2:** type of operations (convolution and GEMM)
3. **RQ3:** different size of operations, i.e., how does varying the size of GEMM or convolution affect fault patterns?

There are two primary challenges involved in answering our RQs: first is the enormous state space, and second is the masking of faults due to multiplication by near-zero layer weights. We detail the challenges below.

Challenge 1 The fault patterns can be influenced by hardware configurations (like systolic array size, data mapping schemes, etc.), software configurations (operation type, operation configuration, size, etc.), and fault models (type of fault, fault location, fault bit position, etc.). This results in an enormous state space that is computationally expensive to explore in its entirety. For example, even a single systolic array of size 16×16, two data mapping schemes, and two operation types and configurations results in a state space with 131K different FI configurations. This is a conservative estimate.

To address this challenge, we *sampled the state space* and selected a few representative configurations - based on their practicality and usefulness - while keeping other parameters constant. Table 6.1 shows the configuration settings for our RQs. Due to the scalability restrictions of Gemmini, we chose 16×16 as our systolic array size. Larger systolic array sizes require an impractically-large amount of system logic cells: 128×128 systolic array size requires ten times more system logic cells than available on our industrial-grade FPGA (Xilinx Virtex UltraScale+ VU9P). This scalability restriction is also common to prior work that uses the RTL model of TPUs [39]. For the input sizes (RQ3), we chose 16×16 and 112×112 input matrix sizes because the former is equal to the size of the systolic array (thus, no tiling effect), while the latter is larger than the size of the systolic array and we expect to observe the effect of tiling on the fault patterns.

Similarly, for contrasting the effect of different tensor operators (GEMM and Convolution) in RQ2, we chose two convolution kernel sizes, $3 \times 3 \times 3 \times 3$ and $3 \times 3 \times 3 \times 8$ ($R \times S \times C \times K$, refer Section 2.3.1 for notations). The former kernel

Table 6.1: Parameter configuration used for evaluating RQ1, and RQ2. The entries highlighted with brown are the ones whose effect we want to understand on the fault pattern.

	RQ1	RQ2	RQ3
Type of operation	GEMM	Conv vs GEMM	
Type of data mapping scheme	OS vs WS	WS	WS
Size of operation	16 x 16	16 x 16	16 x 16, 112 x 112
Size of Convolution Kernel	-	3x3x3x3, 3x3x3x8	
Size and Data Type of systolic array	16 x 16, INT8		

size produces a 2-D matrix of size less than the size of the systolic array, while the latter kernel size produces a 2-D matrix of size more than the size of the systolic array, thereby resulting in a tiling effect in the convolution operator.

Challenge 2 It is quite common for the weights of the DNN layers to be close to zero. These weights, when multiplied by the faulty computation (corrupted by the stuck-at fault), can suppress the fault pattern at the software level. In other words, the induced errors due to stuck-at faults might get masked due to multiplication by zero, thereby skewing the results of our RQs. To prevent this, instead of using real weights of the DNN layer, we used a uniform, non-zero weight matrix with all matrix elements set to one to extract the fault pattern.

6.5 Evaluation Methodology

For each RQ, we ran 256 FI campaigns to exhaustively inject a stuck-at fault into every MAC unit of the 16x16 systolic array. After each FI experiment, we analyze the resulting fault pattern manually. The fault patterns are extracted by contrasting the output of the systolic array with and without FI (ground truth), keeping all other configurations the same. Additionally, for each fault pattern, we categorize it (single-row fault, single-element fault, single-column fault, etc.) based on the spatial distribution of the faults in the array.

Setup

We ran RTL-level FI experiments using Amazon Web Services EC2 F1 instances[1]. F1 instances provide industrial-grade FPGAs to synthesize SystoliFI. Using FPGAs instead of simulators allowed us to perform extensive FI campaigns.

6.6 Results

We organize the results of the FI experiments by the RQs.

6.6.1 Effect of different parameters on Fault Patterns

RQ1: Data flow mapping schemes

Figure 6.3b and Figure 6.3a show the difference between the fault patterns observed for OS and WS data flow schemes, respectively. For OS, a single fault corrupts just a single output element, while for WS, a single fault ends up corrupting an entire column in the output of the GEMM operation.

RQ2: Type of operation

Figure 6.3a and Figure 6.3e show the difference in the fault patterns for GEMM and convolution, respectively, with the WS data mapping scheme. For GEMM, a single fault ends up corrupting the entire column of the output matrix, but for convolution, a single fault corrupts the entire output channel. As explained in Section 2.3.1, the convolution is implemented as a big GEMM by reshaping the input data and the convolution kernel. The resulting matrix has the dimension of $NPQ \times K$ (refer Section 2.3.1 for notations), where each output channel is mapped to each column of the systolic array. It is due to this mapping that the entire channel of the convolution is corrupted.

RQ3: Size of Operation

For GEMM, we considered two input sizes: 16×16 and 112×112, the first one being equal to the size of the systolic array and the latter one being larger than

[1] https://aws.amazon.com/ec2/instance-types/f1/

the systolic array. As described in Section 2.3.2, when the input is larger than the systolic array, the operation gets broken down into small chunks (a.k.a. Tiles).

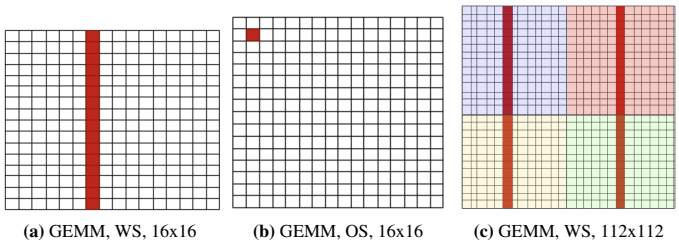
For the GEMM operation, Figure 6.3a and Figure 6.3c show the difference in fault patterns corresponding to different input sizes. Similarly, Figure 6.3b and Figure 6.3d show the difference in fault patterns, but for the OS data mapping scheme. For Figure 6.3c and Figure 6.3d, different tiles are highlighted with different colors. We observed that due to the tiling effect (when the operation size is bigger than the systolic array size), the same fault appears across multiple tiles, irrespective of the data mapping scheme. This result is intuitive since the same faulty MAC unit is used for computation across multiple tiles.

Figure 6.3f and Figure 6.3g show the tiling effect in the convolution operator. Similar to the fault patterns in the GEMM operation, the tiling effect in the convolution operation occurs when the resulting 2-D matrices - after flattening convolution into a GEMM operation, as described in Section 2.3.2 - exceeds the size of the systolic array. Due to the tiling effect, a single fault causes the corruption of multiple output channels, resulting in identical fault patterns in Figure 6.3f and Figure 6.3g.

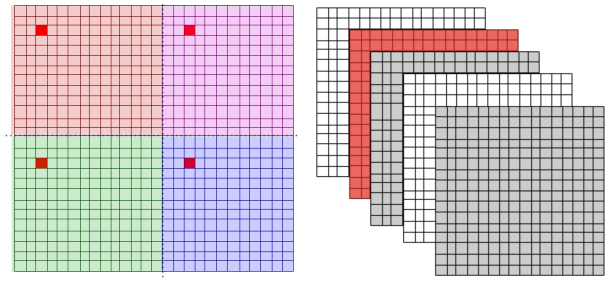
6.7 Discussion

We found that the fault patterns vary according to the hardware and software configurations. However, all the fault patterns we found are well-defined, i.e., they belong to one of the six classes: single-element corruption (e.g., Figure 6.3b), single-element multi-tile corruption (e.g., Figure 6.3d), single-column corruption (e.g., Figure 6.3a), single-column multi-tile corruption (e.g., Figure 6.3c), single-channel (e.g., Figure 6.3e), and multi-channel corruption (e.g., Figure 6.3f).

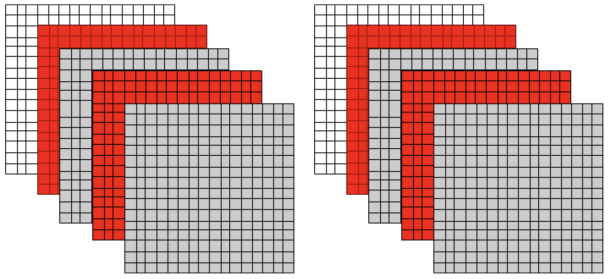
For each configuration and all of its FI experiments (one for each MAC unit), we found the same fault pattern class, regardless of the MAC unit into which we injected the fault. Moreover, the fault patterns are deterministic, i.e., given the hardware configurations (size of systolic array, data mapping scheme), type of operation and its properties (like convolution and its kernel, input size), and the location of the stuck-at-fault, we can predict the fault patterns, after taking into account the tiling effect and flattening of convolutions into GEMM.



(a) GEMM, WS, 16x16 **(b)** GEMM, OS, 16x16 **(c)** GEMM, WS, 112x112



(d) GEMM, OS, 112x112 **(e)** Conv, WS, 16x16, 3x3x3x3



(f) Conv, WS, 16x16, 3x3x3x8 **(g)** Conv, WS, 112x112, 3x3x3x8 & 3x3x3x3

Figure 6.3: Fault patterns corresponding to different configurations of RQ1. Figure 6.3a and Figure 6.3b corresponds to RQ1. For RQ2, contrast Figure 6.3a with Figure 6.3e, Figure 6.3f and contrast Figure 6.3c with Figure 6.3f and Figure 6.3g. Similarly, for RQ3, contrast Figure 6.3a with Figure 6.3c, Figure 6.3b with Figure 6.3d, and Figure 6.3e with Figure 6.3f and Figure 6.3g. For subfigures a,b,c, and d, the caption is a tuple of three elements: $\langle$ Type of operation, Type of Data Flow, and Size of GEMM $\rangle$. For subfigures e,f, and g, the caption is a tuple of four elements: $\langle$ Type of operation, Type of Data Flow, Size of the input matrix, convolution kernel size ($R \times S \times C \times K$, refer Section 2.3.1 for notations) $\rangle$. Different tiles are highlighted with different colors.

Our findings about the well-defined fault pattern classes and their deterministic property are useful for enabling application-level fault injectors like TensorFI and LLTFI to do precise error simulations pertaining to the systolic array hardware model. RTL-level FI, although accurate, has scalability restrictions: due to limited system logic cells on current industrial-grade FPGAs, it is not feasible to synthesize and experiment with larger systolic arrays (like 128×128). Moreover, despite our use of FPGAs, each FI experiment took approximately 45 seconds (for GEMM) and 130 seconds (for Convolution), which resulted in a total of 49 hours for the FI campaigns.

Application-level fault injectors, if supplemented with a precise fault model - the fault patterns, in our case - can be used to bridge this gap and run FI campaigns even with larger systolic array sizes. Specifically to improve the precision of FI, application-level fault injectors, like LLTFI, can leverage our insights about the tiling effect and flattening of convolution operators to derive fault patterns on the fly for various systolic array sizes and data mapping schemes, as opposed to hard-coding the abstract fault pattern classes or ignoring them.

Our observation about the symmetry of fault patterns - i.e., the fault pattern class remains the same irrespective of the position of the faulty MAC unit - can also be used by application-level FIs to reduce the number of FI experiments.

6.8 Summary

In this chapter, we studied the fault patterns in the intermediate layers of DNNs arising due to stuck-at faults within the MAC units of the systolic arrays. We proposed SystoliFI, an RTL-level FI framework, using which we ran FI campaigns to understand the effect of different hardware (like data mapping schemes) and software configurations (like type and properties of tensor operations) on the observed fault patterns. We found the fault patterns to be deterministic and well-defined, and hence, they can be injected by software-level FI tools like LLTFI. We further classified these fault patterns into classes based on the spatial distribution of the faults.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

With the increasing use of ML applications in safety-critical domains such as AVs and medical diagnosis, it has become essential to understand their resilience. In this thesis, we evaluated the resilience of ML models against two types of hardware faults: transient faults in the CPU and stuck-at faults in the systolic arrays.

In the first part of this thesis, we proposed LLTFI, a software-level FI tool for injecting transient hardware faults in ML applications. LLTFI works by lowering the ML model - irrespective of the ML framework used - to LLVM IR. Afterward, LLTFI instruments the LLVM IR to add code instrumentations for injecting transient faults at the runtime. We evaluated LLTFI with six popular ML programs and compared it to TensorFI, a high-level FI tool for ML programs. We found that TensorFI overreports the SDC rate of these programs for single bit-flip faults by 3.5X on average compared to LLTFI. We also demonstrate LLTFI's utility by extending it to perform SID for improving the resilience of ML applications against transient hardware faults. Moreover, we also use LLTFI to evaluate the resilience of five popular LLMs, including Bert and GPT2, against transient hardware faults. Based on extensive FI experiments, we found LLMs to be quite resilient, with an average SDC rate being 0.9% across all benchmarks. We also performed a qualitative categorization of SDCs and found that the manifestation of SDCs varies significantly with different types of LLMs. Additionally, we evaluated the impact of model size,

pre-training, and fine-tuning objectives on the SDC rates and observed that for the fill-mask fine-tuning objective, the SDC rate increases with the model size.

In the second part of this thesis, we proposed SystoliFI, an RTL-level FI tool for injecting stuck-at faults in the systolic arrays of ML accelerators. Using SystoliFI, we ran FI campaigns to understand the effect of different hardware (like data mapping schemes) and software configurations (like type and properties of tensor operations) on the manifestation of stuck-at faults at the intermediate layers of the ML models. We found the fault patterns to be deterministic and well-defined. We further classified these fault patterns based on the spatial distribution of the faults.

7.2 Future Work

We list four of the possible directions in which this thesis's work can be extended:

7.2.1 Extend LLTFI to inject faults pertaining to the intermittent hardware fault model

Intermittent hardware faults [111] occur due to process variation, manufacturing defects, and voltage fluctuations, and they can corrupt multiple instructions during the inference of the ML model. It would be interesting to extend LLTFI to evaluate the effect of intermittent hardware faults on ML models. In order to do so, LLTFI can be configured to inject multiple bit-flip faults across different instructions. Moreover, to model the effect of intermittent faults within a specific part of the CPU, for instance, the ALU, LLTFI can be used for profiling the ML model to identify all the arithmetic instructions in the program that use the ALU. Subsequently, LLTFI can be configured to inject bit-flip faults in those arithmetic instructions to model the effect of intermittent faults.

7.2.2 Extend LLTFI to inject faults in the weights of the ML models

In Chapter 4 and Chapter 5 of this thesis, we considered transient hardware faults in the ALU and data paths of the CPU. However, transient faults can also affect the content of the memory, thus corrupting the weights of the ML models. Therefore, it would be interesting to extend LLTFI for injecting faults in the weights of the ML model.

7.2.3 Extend LLTFI to inject faults pertaining to the systolic array hardware model

SystoliFI has scalability restrictions: due to limited system logic cells on current industrial-grade FPGAs, it is not feasible to synthesize and experiment with larger systolic arrays (like 128×128). LLTFI, being a software-based FI tool, does not suffer from this limitation. Therefore, if supplemented with the fault patterns that we observed in Chapter 6, LLTFI can be used to inject faults corresponding to the systolic array hardware model. Moreover, our insights about the tiling effect and data mapping schemes can be used to derive fault patterns for different sizes of systolic arrays on the fly without the need for hard-coding the fault patterns.

7.2.4 Utilize SystoliFI to inject faults in reduction-tree-based ML accelerators

In this thesis, we used SystoliFI to inject stuck-at faults in systolic arrays. However, SystoliFI can also be configured in a reduction-tree-like configuration of MAC units, used in ML accelerators like NVDLA [102]. Therefore, in the future, we can also utilize SystoliFI to assess and compare the resilience of reduction-tree-based ML accelerators with systolic-array-based accelerators.

7.2.5 Utilize SystoliFI to understand the reliability-trade-offs among hybrid data mapping schemes

Hybrid data mapping schemes like row-stationary [19] offer enticing trade-offs between energy consumption and execution time. Currently, with SystoliFI, we only considered output stationary and weight stationary data mapping schemes. However, evaluating the reliability of hybrid data mapping schemes like row stationary would be interesting.

Bibliography

[1] Baidu Apollo team (2017), Apollo: Open Source Autonomous Driving, howpublished = https://github.com/apolloauto/apollo, note = Accessed: 2022-05-10. → page 30

[2] comma ai openpilot: an open source driver assistance system, howpublished = https://github.com/commaai/openpilot, note = Accessed: 2022. → page 30

[3] System architecture of cloud tpu. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm, 2022. URL https://cloud.google.com/tpu/docs/system-architecture-tpu-vm. → pages 2, 69

[4] GM explores using ChatGPT in vehicles, 2023. URL https://www.reuters.com/business/autos-transportation/gm-explores-using-chatgpt-vehicles-2023-03-10/. Reuters. → page 1

[5] Mercedes is bringing ChatGPT into its cars, 2023. URL https://www.cnn.com/2023/06/15/business/mercedes-benz-chatgpt/index.html. CNN Business. → page 1

[6] Huggingface Model Hub, Date accessed: 31 July, 2023. URL https://huggingface.co/models. → page 53

[7] Date accessed: 31 July, 2023. URL https://www.intel.com/content/www/us/en/artificial-intelligence/documents/optimize-inference-with-cpu-technology-pdf.html. → page 3

[8] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org. → page 2

[9] 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*, 9:65803–65823, 2021. → page 66

[10] V. Agarwal and A. Fung. Multiple fault testing of large circuits by single fault test sets. *IEEE Transactions on Circuits and Systems*, 28(11): 1059–1069, 1981. → page 72

[11] A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf. FLAIR: An easy-to-use framework for state-of-the-art NLP. In *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, 2019. → page 57

[12] A. Alajrami and N. Aletras. How does the pre-training objective affect what large language models learn about linguistic properties? *arXiv preprint arXiv:2203.10415*, 2022. → pages 55, 67

[13] J. Bai, F. Lu, K. Zhang, et al. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx, 2019. → pages 4, 16, 31

[14] M. Beyer, A. Morozov, E. Valiev, C. Schorn, L. Gauerhof, K. Ding, and K. Janschek. Fault injectors for tensorflow: Evaluation of the impact of random hardware faults on deep cnns. *arXiv preprint arXiv:2012.07037*, 2020. → page 45

[15] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. → page 38

[16] S. Burel, A. Evans, and L. Anghel. Mozart: Masking outputs with zeros for architectural robustness and testing of dnn accelerators. In *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6. IEEE, 2021. → pages 26, 27

[17] S. Chen. Labeled car driving dataset. https://github.com/SullyChen/driving-datasets, 2021. → page 38

[18] T. Chen et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proc. of OSDI'18*, 2018. → page 36

[19] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016. → page 82

[20] Z. Chen, G. Li, and K. Pattabiraman. A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction. In *Proc. of DSN'21*. → page 51

[21] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben. BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems. In *Proc. of SC'19*, 2019. → pages 40, 51

[22] Z. Chen, G. Li, and K. Pattabiraman. A low-cost fault corrector for deep neural networks through range restriction. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–13. IEEE, 2021. → page 24

[23] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. → page 17

[24] F. Chollet et al. Keras. https://keras.io, 2015. → page 33

[25] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *iEEE Micro*, 38(2): 8–20, 2018. → page 17

[26] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020. → page 14

[27] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991. → page 21

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. → pages 14, 55, 67

[29] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 610–621. IEEE, 2014. → page 8

[30] M. Didehban and A. Shrivastava. nzdc: A compiler technique for near zero silent data corruption. In *Proc. of DAC '16*. IEEE, 2016. → page 50

[31] O. Docs. ONNX Graph Optimizations, Date accessed: 31 July, 2023. URL https://onnxruntime.ai/docs/performance/model-optimizations/graph-optimizations.html. → page 39

[32] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech. Analyzing and increasing the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, 2018. → page 58

[33] X. Dupre. ONNX Operator Coverage, Date accessed: 31 July, 2023. URL http://www.xavierdupre.fr/app/onnxcustom/helpsphinx/onnxmd/onnx_docs/TestCoverage.html#summary. → page 16

[34] B. B. Elallid, N. Benamar, A. S. Hafid, T. Rachidi, and N. Mrani. A comprehensive survey on the application of deep and reinforcement learning approaches in autonomous driving. *Journal of King Saud University-Computer and Information Sciences*, 34(9):7366–7390, 2022. → page 1

[35] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, 2003. → page 36

[36] A. Fan, Y. Jernite, E. Perez, D. Grangier, J. Weston, and M. Auli. ELI5: long form question answering. In A. Korhonen, D. R. Traum, and L. Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 3558–3567. Association for Computational Linguistics, 2019. doi:10.18653/v1/p19-1346. URL https://doi.org/10.18653/v1/p19-1346. → page 56

[37] X. Feng, M. Ye, K. Xia, and S. Wei. Runtime fault injection detection for fpga-based dnn execution using siamese path verification. volume 2021-February, 2021. doi:10.23919/DATE51398.2021.9473941. → page 26

[38] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. → pages 53, 55, 56

[39] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, et al. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 3:25, 2019. → pages xiii, 5, 20, 71, 74

[40] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach. Transient fault models and avf estimation revisited. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 477–486. IEEE, 2010. → page 8

[41] Y. Gu, R. Tinn, H. Cheng, M. Lucas, N. Usuyama, X. Liu, T. Naumann, J. Gao, and H. Poon. Domain-specific language model pretraining for biomedical natural language processing, 2020. → pages 55, 56

[42] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proc. of DSN'12*, 2012. → page 30

[43] J. P. Hayes. A nand model ror fault diagnosis in combinational logic networks. *IEEE Transactions on Computers*, 100(12):1496–1506, 1971. → page 72

[44] 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)*, pages 270–281. IEEE, 2020. → page 24

[45] 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. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1241–1246. IEEE, 2020. → page 24

[46] 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. In *Proc. of DATE'20*, 2020. → page 51

[47] S. Holst, L. Bumun, and X. Wen. Gpu-accelerated timing simulation of systolic-array-based ai accelerators. In *2021 IEEE 30th Asian Test Symposium (ATS)*, pages 127–132, 2021. doi:10.1109/ATS52891.2021.00034. → page 26

[48] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraș. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 497–514, 2019. → page 24

[49] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction duplication for soft error detection. In *Design, Automation and Test in Europe*, pages 1056–1057. IEEE, 2005. → page 44

[50] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019. → page 55

[51] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. → page 38

[52] 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*, 115:113969, 2020. → page 2

[53] T. Jin et al. Compiling ONNX Neural Network Models Using MLIR, 2020. → pages 4, 31, 36

[54] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov. Fasttext. zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016. → page 57

[55] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017. → page 17

[56] R. L. R. Junior and P. Rech. Reliability of google's tensor processing units for convolutional neural networks. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 25–27. IEEE, 2022. → page 70

[57] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012. → page 38

[58] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu. Toward functional safety of systolic array-based deep learning hardware

accelerators. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(3):485–498, 2021. doi:10.1109/TVLSI.2020.3048829. → pages 26, 70

[59] A. R. Lahitani, A. E. Permanasari, and N. A. Setiawan. Cosine similarity to determine similarity measure: Study case in online essay assessment. In *2016 4th International Conference on Cyber and IT Service Management*. IEEE, 2016. → page 57

[60] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of CGO'04*, 2004. → pages 21, 33

[61] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle. Bridging the gap between rtl and software fault injection. *ACM Journal of Emerging Technologies in Computing System*, 17(3):1–24, 2021. → page 2

[62] C. Leary and T. Wang. XLA, 2017. → page 36

[63] Y. LeCun, C. Cortes, and C. Burges. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010. → page 33

[64] Y. LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20(5):14, 2015. → page 38

[65] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 502–506. IEEE, 2009. → page 58

[66] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019. → page 14

[67] B. Li and L. Han. Distance weighted cosine similarity measure for text classification. In *Intelligent Data Engineering and Automated Learning–IDEAL 2013: 14th International Conference, IDEAL 2013, Hefei, China, October 20-23, 2013. Proceedings 14*. Springer, 2013. → page 57

[68] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017. → pages 4, 23, 25, 26, 66

[69] G. Li, K. Pattabiraman, and N. DeBardeleben. TensorFI: A Configurable Fault Injector for TensorFlow Applications. In *Proc. of ISSREW'18*, 2018. → pages 2, 4, 23, 30, 37, 38, 39, 40, 58, 66

[70] G. Li et al. Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications. In *Proc. of SC '17*. → pages 40, 47

[71] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 22–31. IEEE, 2008. → page 9

[72] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech. Selective hardening for neural networks in fpgas. *IEEE Transactions on Nuclear Science*, 66(1):216–222, 2018. → pages 25, 45, 49

[73] T. Liu, Y. Fu, Y. Zhang, and B. Shi. A hierarchical assessment strategy on soft error propagation in deep learning controller. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, 2021. → page 24

[74] V. Liu and J. R. Curran. Web text corpus for natural language processing. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, pages 233–240, 2006. → page 56

[75] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. → pages 14, 55, 56

[76] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults. In *Proc. of QRS '15*, 2015. → pages 30, 31, 32

[77] L. M. Luza, A. Ruospo, D. Söderström, C. Cazzaniga, M. Kastriotou, E. Sanchez, A. Bosio, and L. Dilillo. Emulating the effects of radiation-induced soft-errors for the reliability assessment of neural networks. *IEEE Transactions on Emerging Topics in Computing*, 10(4): 1867–1882, 2021. → page 29

[78] L. M. Luza, F. Wrobel, L. Entrena, and L. Dilillo. Impact of atmospheric and space radiation on sensitive electronic devices. In *2022 IEEE European Test Symposium (ETS)*, pages 1–10. IEEE, 2022. → pages 2, 5, 70

[79] A. Mahmoud, S. K. S. 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. → page 25

[80] 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*, pages 842–854. IEEE, 2018. → pages 40, 44

[81] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari. Pytorchfi: A runtime perturbation tool for dnns. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2020. → pages 2, 4, 23, 39, 40

[82] A. Mahmoud et al. Optimizing Selective Protection for CNN Resilience. In *Proc. of ISSRE'21*, 2021. → page 51

[83] E. J. McCluskey and C.-W. Tseng. Stuck-fault tests vs. actual defects. In *Proceedings International Test Conference 2000 (IEEE Cat. No. 00CH37159)*, pages 336–342. IEEE, 2000. → page 70

[84] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower. Ibm z990 soft error detection and recovery. *IEEE Transactions on device and materials reliability*, 5(3):419–427, 2005. → page 25

[85] A. Nardi and A. Armato. Functional safety methodologies for automotive applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 970–975. IEEE, 2017. → page 1

[86] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2012. → page 2

[87] N. Oh and E. J. McCluskey. Error detection by selective procedure call duplication for low energy consumption. *IEEE Transactions on Reliability*, 51(4):392–402, 2002. → page 44

[88] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002. → page 44

[89] OpenAI. ChatGPT. https://openai.com, 2021. Accessed: May 28, 2023. → page 14

[90] E. Ozen and A. Orailoglu. Sanity-Check: Boosting the Reliability of Safety-Critical Deep Neural Network Applications. In *Proc. of ATS'19*. → page 51

[91] Z. Peng, J. Yang, T.-H. P. Chen, and L. Ma. A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo. In *Proc. of ESEC/FSE '20*. → page 30

[92] pubmed. PubMed Dataset, Date accessed: 31 July, 2023. URL https://huggingface.co/datasets/pubmed. → page 55

[93] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019. → pages 14, 55, 56

[94] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140): 1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html. → pages 11, 13, 14, 15, 55, 56

[95] Ragan-Kelley et al. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. 2013. → page 36

[96] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text. *arXiv e-prints*, art. arXiv:1606.05250, 2016. → page 56

[97] B. Reagen et al. Ares: A Framework for Quantifying the Resilience of Deep Neural Networks. In *Proc. of DAC '18*, 2018. → pages 4, 23, 30, 40

[98] R. L. Rech and P. Rech. Reliability of google's tensor processing units for embedded applications. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 376–381. IEEE, 2022. → page 25

[99] J. Ren. *Geometric characterizations of fault patterns in linear systolic arrays.* PhD thesis, Carleton University, 1994. → page 26

[100] N. Rotem et al. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR*, 2018. → page 36

[101] B. Sangchoolie, K. Pattabiraman, and J. Karlsson. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 97–108, 2017. doi:10.1109/DSN.2017.30. → page 40

[102] F. Sijstermans. The nvidia deep learning accelerator. In *Hot Chips*, volume 30, pages 19–21, 2018. → pages 21, 69, 82

[103] C. Simmons and M. A. Holliday. A comparison of two popular machine learning frameworks. *Journal of Computing Sciences in Colleges*, 35(4): 20–25, 2019. → page 16

[104] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. → page 38

[105] T. H. Trinh and Q. V. Le. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*, 2018. → page 56

[106] A. Tyagi, Y. Gan, S. Liu, B. Yu, P. Whatmough, and Y. Zhu. Thales: Formulating and estimating architectural vulnerability factors for dnn accelerators. *arXiv preprint arXiv:2212.02649*, 2022. → page 26

[107] G. R. Upasani. Soft error mitigation techniques for future chip multiprocessors. 2016. → page 8

[108] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. → page 11

[109] J. Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42,

Florence, Italy, July 2019. Association for Computational Linguistics. doi:10.18653/v1/P19-3007. URL https://www.aclweb.org/anthology/P19-3007. → pages xiii, 12

[110] A. Warstadt, A. Singh, and S. R. Bowman. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019. → page 55

[111] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. *ACM SIGOPS Operating Systems Review*, 42(2): 255–264, 2008. → page 81

[112] A. Williams, N. Nangia, and S. R. Bowman. The multi-genre nli corpus. 2018. → page 55

[113] A. Yamaguchi, G. Chrysostomou, K. Margatina, and N. Aletras. Frustratingly simple pretraining alternatives to masked language modeling. *arXiv preprint arXiv:2109.01819*, 2021. → pages 55, 67

[114] J. Zhang, Z. Ghodsi, S. Garg, and K. Rangineni. Enabling timing error resilience for low-power systolic-array based deep learning accelerators. *IEEE Design Test*, 37(2):93–102, 2020. doi:10.1109/MDAT.2019.2947271. → page 26

[115] 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)*, pages 1–6. IEEE, 2018. → pages 2, 69

[116] J. J. Zhang, K. Basu, and S. Garg. Fault-tolerant systolic array based accelerators for deep neural network execution. *IEEE Design Test*, 36(5): 44–53, 2019. doi:10.1109/MDAT.2019.2915656. → pages 26, 70, 72

[117] Y. Zheng, Z. Feng, Z. Hu, and K. Pei. Mindfi: A fault injection tool for reliability assessment of mindspore applications. In *Workshop on Dependability Modeling and Design*, 2021. → page 40

[118] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, 2015. → page 56

[119] H. Ziade, R. A. Ayoubi, R. Velazco, et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004. → page 2