Individual Diploma Thesis

INVESTAGATING SILENT DATA CORRUPTIONS AND THEIR IMPACT IN LARGE LANGUAGE MODLES

Theodoros Siokouros

UNIVERSITY OF CYPRUS

DEPARMENT OF COMPUTER SCIENCE

May 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Investigating Silent Data Corruptions – Occurrence, Manifestation, And Impact

Theodoros Siokouros

Supervisor

Haris Volos

The Individual Diploma Thesis was submitted for partial fulfilment of the requirements for the degree of Computer Science of the Department of Computer Science of the University of Cyprus

May 2024

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr Haris Volos, Professor at the Department of Computer Science at the University of Cyprus, for trusting me on this topic and for the guidance and support throughout the entire process of my Diploma Thesis.

I would also like to thank my friends and family for their unwavering support and encouragement.

Abstract

Silent data corruptions (SDCs) pose a significant challenge in modern computational structures, such as cores in computing systems. These errors occur without triggering immediate error detection mechanisms, making them particularly insidious. SDCs can manifest at various levels of the computing stack within these structures, including memory, storage devices, and communication channels. Their occurrence threatens the integrity and reliability of data-driven systems within computational structures, potentially leading to erroneous results, system instability, and compromised security. Understanding the nature and impact of SDCs in computational structures is crucial for ensuring the robustness and dependability of critical applications.

This thesis aims to delve into the impact of silent data corruptions (SDCs) within computational structures, such as cores, and propose effective strategies for their detection. Through meticulous examination and empirical analysis, the research seeks to elucidate the ramifications of SDCs on system functionality and data integrity specifically within these structures. By scrutinizing real-world scenarios and experimental data, the thesis endeavors to quantify the extent of SDC-induced disruptions in computational environments, focusing on their impact within cores and similar computational structures.

Table of contents

Chapter 1

Introduction

1.1 Motivation

The motivation for this study stems from the increasing reliance on computing systems across various domains, where data integrity is paramount. As modern systems grow in complexity and scale, the susceptibility to silent data corruptions (SDCs) becomes a pressing concern. Unlike traditional errors, SDCs occur stealthily, evading detection mechanisms and potentially leading to erroneous outcomes with severe consequences. Understanding the occurrence and impact of SDCs is crucial for safeguarding the reliability and trustworthiness of critical applications such as financial transactions, healthcare systems, and scientific simulations. Moreover, as data volumes continue to surge and computing architectures evolve, the need for robust mechanisms to detect and mitigate SDCs becomes ever more imperative.

SDCs are not a new problem, as they have been studied extensively in the context of memory and storage. However, the emergence of SDCs in computational structures presents a novel challenge. With shrinking transistor sizes, cores and computational units are becoming increasingly vulnerable to silent failures. These failures can occur within cores without triggering immediate detection, potentially compromising the accuracy and reliability of computational processes. Understanding and addressing SDCs within computational structures is essential for ensuring the continued reliability and performance of modern computing systems. By investigating this new frontier of SDCs, this study aims to contribute to the advancement of fault-tolerant computing and data management techniques, ultimately enhancing the resilience of critical applications in today's computing landscape.

1.2 Goals of the study

The primary focus of this study is to delve into the nature of silent data corruptions (SDCs) and investigate their impact on Large Language Model (LLM) systems. By closely examining the characteristics and manifestations of SDCs, the research seeks to gain a deeper understanding of their behaviour within the context of LLM frameworks. Through empirical analysis and theoretical exploration, the study aims to uncover how SDCs manifest and propagate within LLM systems, potentially affecting model accuracy, training convergence, and overall system performance. By shedding light on the relationship between SDCs and LLM, the research aims to provide insights that can inform the development of more robust and resilient machine learning algorithms and systems.

In addition to exploring the nature of silent data corruptions (SDCs) and their impact on Large Language Model (LLM) systems, this study also evaluates various tools for detecting such issues. By evaluating detection tools alongside the investigation of SDCs and their impact on LLM, the research aims to contribute to the development of comprehensive strategies for enhancing the resilience and reliability of machine learning systems in the face of silent data corruptions.

1.3 Methodology

The methodology employed in this study involved a multi-faceted approach to comprehensively investigate silent data corruptions (SDCs) and their impact on Large Language Model (LLM) systems.

Initially, an extensive review of literature and existing research papers was conducted to gain insight into the nature and characteristics of SDCs. This phase provided a foundational understanding of SDC phenomena through various examples and case studies.

Subsequently, attention shifted towards a meticulous review of tools designed for SDC detection. The focus was on understanding the methodologies employed by these tools, particularly the routines through which processors were tested. This enabled a deeper comprehension of the tests conducted and contributed to the overall understanding of the problem.

Following this theoretical scrutiny, CloudLab[6] servers emerged as the testing ground for the next phase. Here, a fleet of processors and cores provided the environment for testing the tools for SDC detection and their routines in depth. Despite the thorough examination, the search for a faulty processor within the CloudLab[6] infrastructure yielded negative results for our research as a faulty processor was not found, thus highlighting the challenges inherent in detecting SDCs within real-world systems.

Undeterred by this outcome, the methodology pivoted towards a novel approach utilizing the capabilities of Pin[9]. With Pin[9] used as a fault injection tool, deliberate faults were induced into an LLM system, simulating the subtle disruptions characteristic of SDCs. Through meticulous experimentation and documentation, this phase facilitated a granular analysis of the impacts of various fault types on LLM performance.

This methodology allowed for a comprehensive exploration of SDCs, encompassing both theoretical understanding and practical experimentation to provide valuable insights into their detection and mitigation within the context of LLM systems. In the following chapters the methodology used for each step of the research is explained further.

1.4 Document Organization

The rest of this thesis is split into six chapters. **Table 1.1** reports the content of each chapter.

Chapter Number	**Chapter Description**
2	This chapter provides a comprehensive introduction to silent data corruptions (SDCs), exploring their definition,

	characteristics, and significance in computing systems.
3	This chapter examines various tools for detecting silent data corruptions (SDCs), including the OpendcDiagram and Intel diagnostic tool.
4	Here, the focus shifts to CloudLab, where tests were conducted to evaluate the performance of servers. The chapter covers CloudLab tests, automations employed, and the results obtained.
5	This chapter details the process of infecting Large Language Model (LLM) systems with faults. It discusses the Llama framework, the Pin tool used for fault injection, and various methods of injecting faults into LLM systems.
6	This chapter presents the findings and results obtained from the experiments conducted throughout the study, including insights into SDC detection, CloudLab server performance, and the impact of injected faults on LLM systems.
7	The discussion chapter summarizes the key findings of the study and provides insights into their implications. It also explores avenues for future research and enhancements in the field of SDC detection and mitigation.

Table 1.1 Document Organization

Chapter 2

An Overview of SDC’s

2.1 What are SDC’s

Silent data corruptions (SDCs) are defined as errors that occur within computing systems without triggering any immediate error detection mechanisms. Unlike traditional errors, which are typically detected and corrected through error-checking processes, SDCs remain latent, posing a significant challenge to the integrity and reliability of data-driven applications. These corruptions can manifest at various levels of the computing stack, including memory, storage devices, and communication channels and more recently computational structures, which is the focus of this thesis. SDCs may arise from a multitude of sources, such as cosmic radiation, electromagnetic interference, or inherent flaws in hardware components. The insidious nature of SDCs makes them particularly worrisome, as they can silently propagate throughout the system, potentially leading to erroneous results, system instability, and compromised security. Understanding the characteristics and manifestations of SDCs is paramount for developing robust strategies to detect, mitigate, and prevent their occurrence, thereby ensuring the dependability of computing systems in critical applications.

2.2 Basic example of SDC’s

One of the most notable SDC (silent data corruption) known is the one described by Meta in their article [1]. The article discusses the pervasive issue of silent data corruption (SDC) in large-scale infrastructure systems, emphasizing its potential to cause application-level problems and data loss. It provides insights into a real-world example of SDC within a data center application, where missing rows in a database resulted from a silent error during file decompression. Specifically, the error was attributed to a single line of code: "Int (1.1^53)" executed on core 59 of the CPU, yielding the unexpected result of 0 (figure

2.1). This anomaly disrupted the file compression process, leading to missing lines in the database. The article also outlines the complexity of detecting and reproducing such scenarios in a large-scale environment and describes the investigative process to identify the root cause (figure 2.2). Through meticulous debugging and analysis, the article reveals that silent errors can affect computations, leading to incorrect results and data inconsistencies. It emphasizes the importance of robust detection mechanisms and fault-tolerant software architectures to mitigate SDCs in large-scale infrastructures. The article underscores the significance of understanding and addressing SDCs to enhance the reliability and fault tolerance of software architectures, ultimately contributing to the development of more reliable infrastructure computing systems.

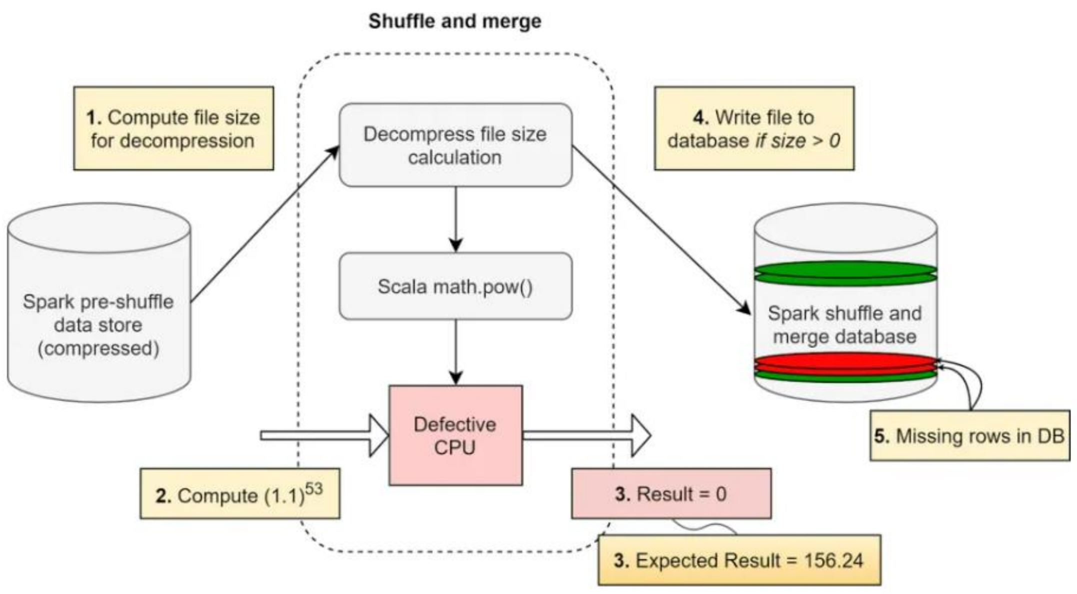

Shuffle and merge
1. Compute file size for decompression
Decompress file size calculation
4. Write file to database if size > 0
Spark pre-shuffle data store (compressed)
Scala math.pow()
Spark shuffle and merge database
Defective CPU
5. Missing rows in DB
2. Compute (1.1)^53
3. Result = 0
3. Expected Result = 156.24

Figure 2.1 – Decompression Problem

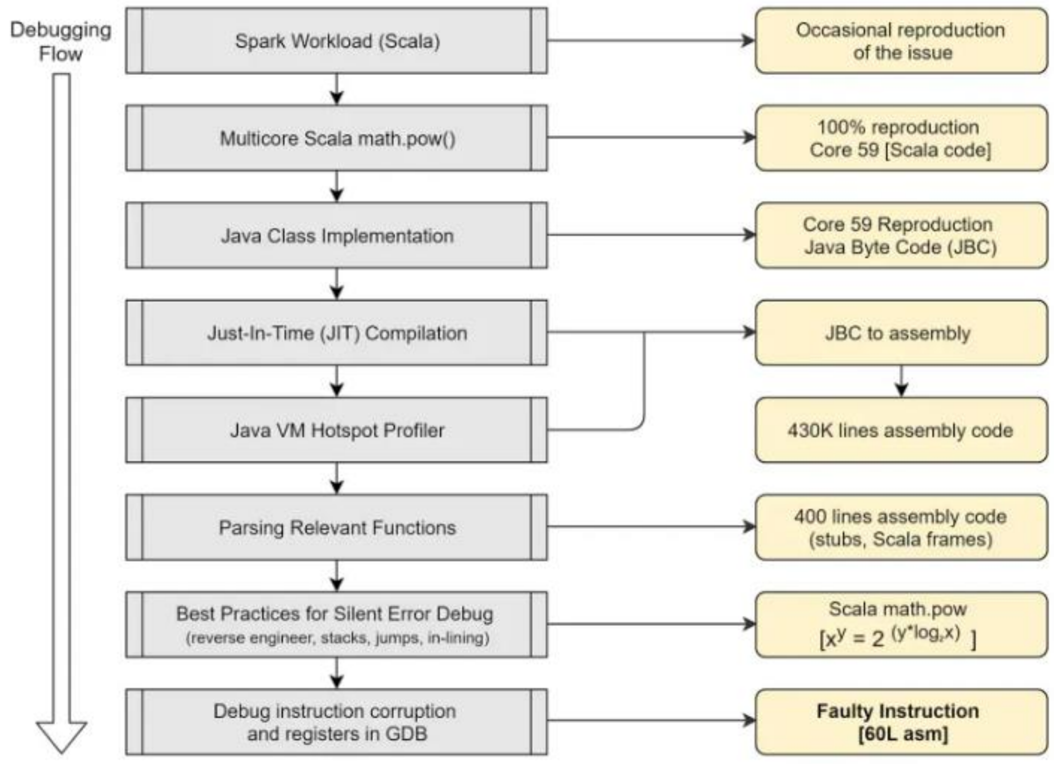

Debugging Flow
Spark Workload (Scala)
Multicore Scala math.pow()
Java Class Implementation
Just-In-Time (JIT) Compilation
Java VM Hotspot Profiler
Parsing Relevant Functions
Best Practices for Silent Error Debug (reverse engineer, stacks, jumps, in-lining)
Debug instruction corruption and registers in GDB
Occasional reproduction of the issue
100% reproduction Core 59 [Scala code]
Core 59 Reproduction Java Byte Code (JBC)
JBC to assembly
430K lines assembly code
400 lines assembly code (stubs, Scala frames)
Scala math.pow [x^y = 2^(y*log2x)]
Faulty Instruction [60L asm]

Figure 2.2 – Debugging Flow

2.3 Impact of SDC's and Challenges in SDC Detection and Mitigation

SDCs are indeed a critical concern, as evidenced by the findings outlined in the paper "Understanding Silent Data Corruptions in a Large Production CPU Population" [2]. The research delves deep into the intricacies of silent data corruptions (SDCs) within the expansive CPU ecosystem of Alibaba's datacenter, offering invaluable insights into their pervasive nature and impact. The study meticulously quantifies the frequency of SDC occurrences, revealing a wide spectrum ranging from as low as 0.01 times per minute to occurrences in the hundreds per minute across different settings. Notably, the paper identifies temperature as a significant contributing factor to SDCs, showcasing how the occurrence frequency of these errors exhibits exponential growth with increasing temperatures, high workload intensity and heavy system workload. Moreover, the researchers developed a sophisticated tool named Farron to mitigate the impact of SDCs efficiently. Farron stands out as an innovative solution, leveraging insights gleaned from the study's observations to enhance existing mitigation strategies. It prioritizes efficiency-focused SDC testing, utilizing adaptive temperature control mechanisms to complement traditional testing approaches. Through a fine-grained processor decommissioning process, Farron effectively identifies and isolates defective cores, thereby minimizing the impact on system reliability and performance.

SDCs represent a pressing concern in modern computing environments, a reality also underscored by the seminal work "Cores That Don't Count" [12]. This research delves into the intricate realm of mercurial cores and their disruptive potential within CPU ecosystems. By exploring the nuanced dynamics of these cores, the study illuminates the complex interplay between hardware complexity, manufacturing risks, and environmental factors. The findings reveal that mercurial cores can manifest unexpectedly, posing challenges to traditional detection and mitigation approaches. Moreover, the paper highlights the significance of early-warning systems and robust verification mechanisms to address the evolving landscape of CPU defects.

Chapter 3

Detection Tools

3.1 Intoduction

Chapter 3 of this thesis delves into the exploration of tools crucial for identifying and mitigating silent data corruptions (SDCs) within computing systems. As highlighted in previous chapters, SDCs pose significant challenges to data integrity and system reliability, necessitating robust detection mechanisms to safeguard against potential disruptions.

In this chapter, we embark on a comprehensive examination of a diverse array of tools designed to uncover and analyze instances of SDCs across various computing environments. These tools encompass a range of methodologies. Each tool offers unique insights into the integrity of data processing systems, providing valuable information for enhancing resilience and fortifying defenses against data corruption.

Through detailed discussions and empirical evaluations, we aim to shed light on the underlying principles and practical applications of these detection tools. By elucidating their capabilities and limitations, we seek to empower practitioners and researchers with the knowledge necessary to effectively assess and address the risks posed by SDCs.

The subsequent sections of this chapter will offer in-depth insights into the functionalities and methodologies employed by each tool, offering a comprehensive overview of their contributions to the field of data integrity and system reliability.

3.2 OpendcDiagram

Overview

OpenDCDiag[4], a vital resource within CPU testing, is an open-source project hosted on GitHub. It offers a comprehensive suite of tests built around a sophisticated CPU testing framework, catering primarily to Data Center CPUs while also extending its utility to a broader range of CPU architectures. With an emphasis on robustness and accuracy, OpenDCDiag[4] enables users to assess the functionality and reliability of CPU cores through rigorous testing procedures, scrutinizing various aspects of CPU performance and behavior.

Key Features

- Test Suite: OpenDCDiag[4] boasts a rich repository of tests, covering a broad spectrum of functionalities and scenarios relevant to CPU testing. These tests are meticulously crafted to address different aspects of CPU behaviour, ranging from floating-point arithmetic to memory management and system interface operations.

- Supported Architectures: While primarily tailored for Data Centre CPUs, OpenDCDiag[4] offers compatibility with a range of CPU architectures, providing versatility and adaptability across different hardware platforms.

- Ease of Use: The framework is designed with simplicity and accessibility in mind, facilitating the creation of new CPU tests with minimal overhead. Test authors can leverage the built-in functionality of the OpenDCDiag[4] framework to focus on specific test functionalities without getting bogged down by implementation details.

- Fast Execution: Tests within OpenDCDiag[4] typically run within a short timeframe of 1-2 minutes, enabling efficient validation of CPU cores without significant time investment.

- Selective Test Execution: The framework provides the flexibility to run specific tests individually, offering users the ability to tailor their testing procedures according to their requirements instead of executing the entire test suite.

Test Types

The tests within OpenDCDiag[4] cover a wide range of functionalities and scenarios pertinent to CPU validation. The main ones are shown in table 3.1:

Test	Explanation
Eigen_gemm	This test aims to stress test the execution (in particular FMA) by repetitively solving general matrix multiplication. The multiplication function is from the 3rd party library Eigen. A pair of random matrices are generated as inputs and then multiplied using Eigen's gemm. The multiplication result is compared against a golden result that is computed during initialixation. This test goes for double precision, complex and float input matrices. This variation adds extra copies and consistency checks.
Eigen_sparse	This test aims to stress test the execution (in particular FMA) by repetitively solving the set of linear equations represented by Ax=b where A is a sparse real symmetric matrix using Cholskey method. The decomposition function is from the 3rd party library Eigen. A random double precision sparse real symmetric matrix (A) and a random vector (b) are generated as inputs and then Eigen::SimplicialCholesky is used to solve the problem Ax=b. The result vector is compared against a golden result that is computed during initialization.
Eigen_svd	This piece of code aims to stress test the FMA execution units of the CPU, among others, by repetitively solving the singular value decomposition problem on given input matrices, which involve a lot of matrix multiplication operations underneath.

	The logic comes from the 3rd party library Eigen. The first thread that gets to run computes a "golden value" of the results, those being output matrices "U" and "V". Subsequent runs have results contrasted to those golden values and, whenever they differ, an error is flagged.
Eigen_svd_jacobi	This particular version of the Eigen SVD test go for single precision input matrices and the "Jacobi" SVD algorithm.
Simple_add	simple_add repeatedly adds the same two random numbers on each thread and checks all threads produce the same result.
Vector_add	Vector_add repeatedly adds two arrays of random numbers together using AVX-512 instructions and checks that their output is correct.
Openssl	The test calculates 3 different checksums (sha256, sha384 and sha512) for a given random-generated buffer and compares the results against pre-calculated golden values.
Zlib	This test performs GZIP compression and decompression on buffers of varying compressibility. Some buffers are random data, and therefore not very compressible. Other buffers are still random but are limited to uppercase ASCII characters [A-Z], and therefore more compressible. For this test, a random compression level is used.
Zstd	This test performs ZStandard compression and decompression on random data. Because random data is not very compressible, it emphasizes different codepaths compared to the other ZStandard tests.

Table 3.1 – OpenDCDiag main tests

Conclusion

OpenDCDiag[4] stands as a pivotal tool in the arsenal of CPU validation and verification, offering a robust framework for identifying and addressing defects within CPU architectures. With its extensive test suite, versatile architecture support, fast execution times, and selective test execution capabilities, OpenDCDiag[4] provides an efficient and effective solution for ensuring the reliability and performance of Data Centre CPUs. It should also be mentioned that Alibaba reports that OpenDCDiag is equally effective as the home-brew tests they run for detecting SDCs.

3.3 Intel Data Centre diagnostic tool

Overview

The Intel Data Center Diagnostic Tool[5] is a robust solution designed specifically for verifying the health and performance of Intel Xeon processors deployed in Data Center environments. With its comprehensive suite of tests and user-friendly interface, this tool enables IT managers to conduct periodic maintenance and screening of their Data Center fleets, ensuring the highest level of ongoing quality and availability in the dynamic landscape of hyperscale cloud providers and enterprise Data Centers. By systematically assessing the functionality of each individual microprocessor core, it offers insights into potential faults or issues that could compromise system reliability and performance.

Key Features

- Comprehensive Testing: It conducts a suite of tests designed to methodically check most aspects of the system-on-chip (SoC) functionality, including core functionality, caches, core-to-core communications, and processor instructions. By verifying the correctness of processor operations with multiple tests, it can detect various types of faults, including Silent Data Errors.

- Easy-to-Interpret Results: Upon completion of the diagnostic tests, it provides clear and concise test results, indicating whether the test completed successfully, detected machine check errors, or encountered issues with processor support or outdated microcode. This enables system administrators to quickly identify any potential issues and take appropriate action.

- Periodic Maintenance: The tool can be used for periodic fleet screening, either in foreground mode with a runtime of approximately 45 minutes or in background mode. By running periodic system maintenance, IT managers can proactively identify and address potential issues before they impact system performance or availability.

How Does It Work?

The Tool operates by executing multiple loops of code or running the same code on all cores, then comparing the results to ensure consistency. Random data sets are used to test instructions and cores, with the tool providing a straightforward pass/fail assessment based on the test outcomes.

System Requirements

- Supported Processors: The tool is compatible with a range of Intel Xeon processors, including various generations of Xeon Scalable Processors and specific processor families such as Broadwell and Skylake.

- Operating Systems: The tool is available for both Linux and Windows operating systems, with support for current Linux and Windows distributions. It is recommended to run the application in the root system of a server for optimal coverage.

Conclusion

The Intel Data Centre Diagnostic Tool[5] stands as a valuable resource for IT managers tasked with maintaining the health and performance of data centre fleets. By leveraging a comprehensive suite of tests and providing easy-to-interpret results, it empowers organizations to ensure the highest level of quality and availability in their data centre environments.

3.4 Tool Comparison

In this section we take a closer look at the differences and the similarities between the two tools described above in order to get a better understanding of their advantages and disadvantages.

Scope and Purpose:

OpenDCDiag[4] is an open-source project hosted on GitHub, focusing on CPU validation and verification. It offers a comprehensive suite of tests for assessing the functionality and reliability of CPU cores, with a primary emphasis on Data Centre CPUs. On the other hand, the Intel Data Centre Diagnostic Tool[5] is specifically designed for verifying the health and performance of Intel Xeon processors deployed in Data Centre environments. While both tools aim to ensure CPU reliability and performance, OpenDCDiag[4] caters to a broader range of CPU architectures beyond Intel Xeon processors.

Test Suite:

OpenDCDiag[4] boasts a rich repository of tests covering various CPU behaviours, including floating-point arithmetic, memory management, and system interface operations. The tests are meticulously crafted to address different aspects of CPU behaviour. In contrast, the Intel Data Centre Diagnostic Tool[5] conducts a suite of tests methodically checking most aspects of the system-on-chip (SoC) functionality, including core functionality, caches, core-to-core communications, and processor instructions.

Architecture Support:

OpenDCDiag[4] offers compatibility with a range of CPU architectures, providing versatility and adaptability across different hardware platforms beyond Intel Xeon processors. On the other hand, the Intel Data Centre Diagnostic Tool[5] is specifically designed for Intel Xeon processors, including various generations of Xeon Scalable Processors and specific processor families such as Broadwell and Skylake.

Ease of Use:

OpenDCDiag[4] is designed with simplicity and accessibility in mind, facilitating the creation of new CPU tests with minimal overhead. It provides built-in functionality for test authors to focus on specific test functionalities. Similarly, the Intel Data Centre Diagnostic Tool[5] offers an easy-to-use interface with clear and concise test results, enabling quick identification of potential issues.

Execution Time:

Tests within OpenDCDiag[4] typically run within a short timeframe of 1-2 minutes, enabling efficient validation of CPU cores without significant time investment. In comparison, the Intel Data Centre Diagnostic Tool[5] requires approximately 45 minutes to complete a full suite of tests.

Selective Test Execution:

OpenDCDiag[4] provides the flexibility to run specific tests individually, allowing users to tailor their testing procedures according to their requirements. Similarly, the Intel Data Centre Diagnostic Tool[5] can be used for periodic fleet screening in foreground or background mode.

Effectiveness in Detecting SDCs:

It's worth noting that Alibaba reports OpenDCDiag[4] as equally effective as their home-brew tests for detecting Single-Event Upsets (SDCs). However, there's no specific mention of SDC detection capabilities in the description of the Intel Data Centre Diagnostic Tool[5].

In conclusion, both OpenDCDiag[4] and the Intel Data Centre Diagnostic Tool[5] offer valuable resources for assessing CPU reliability and performance in Data Centre environments. While OpenDCDiag[4] provides versatility across different CPU architectures and boasts fast execution times, the Intel Data Centre Diagnostic Tool[5] offers comprehensive testing specifically tailored for Intel Xeon processors with easy-to-interpret results. The choice between the two tools may depend on the specific requirements and architecture of the Data Centre environment. For the next part of the thesis where we took the tools to CloudLab[3] for testing, we mostly used OpenDCDiag[4] and the main reason was the time it takes to execute which is significantly lower in comparison to the Intel Data Centre Diagnostic Tool[5].

Chapter 4

CloudLab Tests

4.1 CloudLab

CloudLab[3] is a cloud computing testbed designed to support research and experimentation in distributed systems and networking. It provides researchers and developers with access to a scalable and customizable infrastructure for testing novel ideas and evaluating new technologies. CloudLab offers a diverse range of resources, including compute nodes, storage, and networking components, allowing users to create complex experimental environments tailored to their specific research requirements. With its flexible and programmable architecture, CloudLab enables researchers to explore a wide variety of distributed systems and networking scenarios, ranging from cloud computing and edge computing to Internet of Things (IoT) and wireless networks. Through its user-friendly interface and extensive documentation, CloudLab empowers researchers to conduct cutting-edge experiments and advance the state-of-the-art in distributed systems and networking.

4.2 Automated Tests

In my research on CloudLab[3], I conducted automated tests focusing on the performance and reliability of individual nodes using the OpenDC[4] framework. To streamline the testing process, I developed a command-line bash script, based on a tool[6], that automated the execution of tests across different nodes within the CloudLab[3] environment. The primary objective of these tests was to assess the stability and robustness of the individual nodes under various workload scenarios and to identify a problematic CPU or core that could be used for further research in SDC's.

The automated test workflow involved deploying instances of the OpenDC[4] framework on multiple nodes within the CloudLab[3] infrastructure. The script configured each node with the necessary dependencies and parameters required to run the experiments. Once the setup was complete, the script initiated the execution of the OpenDC[4] framework on each node.

During the execution phase, the OpenDC[4] framework simulated various workload scenarios, emulating real-world conditions that stress the system's capabilities. The script monitored the execution of tests in real-time and recorded the outcome of each test run, whether it passed or failed.

Upon completion of the experiments, the script compiled the results from each node, summarizing the overall test outcomes. The collected data included information about the success or failure of each test case, providing valuable insights into the system's performance and reliability.

By automating the testing process and focusing on the pass/fail outcomes of the tests, I was able to efficiently assess the stability and robustness of the distributed system under investigation. This approach enabled me to gather crucial data on the system's behaviour and identify any potential issues or weaknesses that required further investigation. Overall, the automated testing framework facilitated rigorous experimentation and provided valuable insights into the reliability of individual nodes in CloudLab[3].

It is noteworthy that OpenDC[4] was used instead of the Intel Diagnostic Tool[5] do to the fact that it has much less runtime and it made contracting a large amount of tests much easier.

4.3 Type of CPU Tested

During the tests I checked nodes of type c220g5 in CloudLab[3] which corresponds to the Intel Xeon Silver 4114 processor[7]. This processor represents a powerful and versatile computing solution tailored for a wide range of enterprise applications. Built on Intel's scalable Xeon architecture, the Silver 4114 offers a balance of performance, efficiency, and reliability suitable for demanding workloads in cloud computing, data analytics, virtualization, and more. With x86_64 architecture and a base clock speed of 2.20 GHz and Turbo Boost technology delivering up to 3.00 GHz, the processor delivers robust computing performance to handle complex tasks efficiently. Featuring 10 cores and 20 threads, along with support for DDR4 memory and advanced security features, the Xeon Silver 4114 is well-equipped to meet the demands of modern data centres and cloud environments. Its combination of high core count, multi-threading capabilities, and hardware-level security make it an ideal choice for applications requiring scalability, reliability, and performance optimization. Overall, the Intel Xeon Silver 4114 processor [7] offers a compelling balance of performance, efficiency, and reliability, making it a suitable choice for powering critical workloads in enterprise computing environments.

4.4 Results

The results of the testing process yielded a highly favorable outcome for CloudLab[3], as no faulty nodes were identified. However, from a research perspective, the absence of a problematic node hindered the advancement of the investigation.

Comparing these results to existing literature, such as Alibaba's reports indicating a processor failure rate of 0.3%, suggests that a considerable number of nodes would need to be tested to locate a problematic processor. With approximately 227 nodes in total, testing 150 nodes (each with 2 processors) would likely uncover one problematic processor, assuming a similar failure rate.

However, achieving full coverage across all nodes is challenging due to resource allocation constraints. The nature of borrowing resources from CloudLab[3] means that specific nodes cannot be requested, making it difficult to systematically test every single node. Resources are assigned based on availability, and the inability to select specific nodes further complicates the testing process.

Despite these challenges, the comprehensive evaluation conducted across a total of 81 unique nodes underscores the reliability and stability of the distributed system. The consistent success of the tests across all nodes validates the effectiveness of the system's design and implementation, affirming its capability to deliver reliable and consistent performance in real-world deployments. These results provide confidence in the system's ability to meet the demands of enterprise-scale applications and underscore its suitability for mission-critical workloads in cloud computing and distributed systems environments.

Chapter 5

Injecting SDC’s in LLMs

5.1 Introduction

In this section, we delve into the critical endeavour of injecting silent data corruptions (SDCs) into Large Language Models (LLMs) to comprehend their impact on system performance and reliability. Our focus extends to simulating various types of faults within the LLM architecture, aiming to emulate potential scenarios of data corruption that may occur unnoticed. By injecting these faults, we aim to scrutinize the resilience of LLMs against such adversities and discern their repercussions on the model's output. Through meticulous experimentation, we intend to shed light on how these injected SDCs manifest within the LLM, influencing its behaviour and altering the accuracy and coherence of the generated output. The insights garnered from these endeavours will not only elucidate the vulnerabilities of LLMs to silent data corruptions but also pave the way for devising robust mitigation strategies to fortify their integrity and reliability in real-world applications.

Our decision to move towards injecting faults into the LLM stems from the inability to identify a faulty node within the CloudLab[3] infrastructure. As we aimed to conduct experiments on potentially flawed processors to assess the impact on LLM performance, the absence of a faulty node posed a significant challenge. Consequently, we redirected our approach towards artificially introducing faults into the LLM environment. By simulating these faults, we aim to replicate real-world scenarios of system instability and evaluate the LLM's response under adverse conditions. This shift in methodology underscores the adaptive nature of our research approach, allowing us to explore alternative avenues for investigating the resilience of LLMs against silent data corruptions.

5.2 What are LLM’s

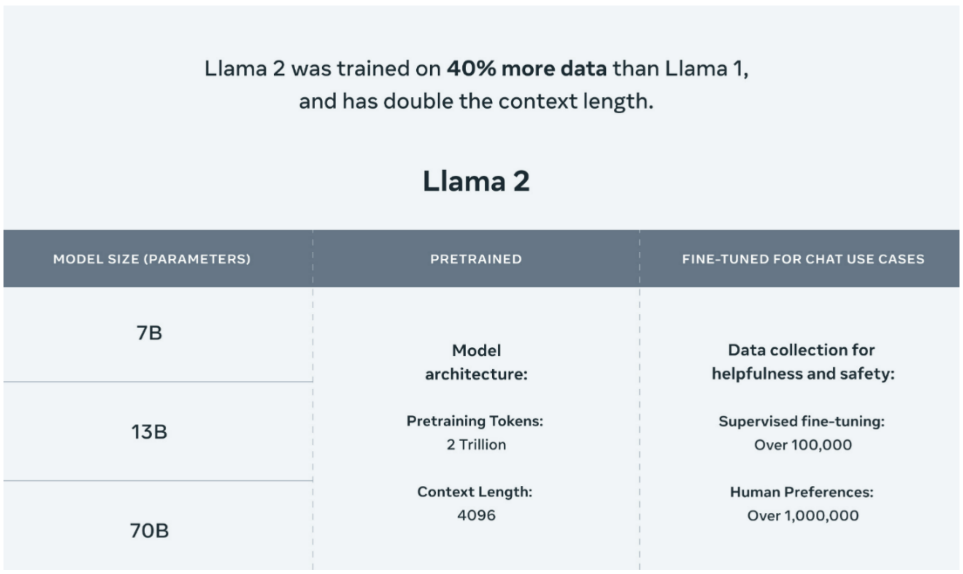
Large Language Models (LLMs) represent a pivotal advancement in natural language processing (NLP) and artificial intelligence, revolutionizing various fields including text generation, translation, and sentiment analysis. LLMs are sophisticated neural network architectures trained on vast amounts of textual data, enabling them to comprehend and generate human-like text with remarkable fluency and coherence. These models leverage deep learning techniques, such as transformers, to capture intricate linguistic patterns and relationships within the input data, allowing them to generate contextually relevant and semantically accurate text. LLMs have garnered immense attention and acclaim for their ability to perform a wide range of language-related tasks, from autocomplete suggestions to document summarization, with unprecedented accuracy and efficiency. Examples of popular LLMs include OpenAI's GPT (Generative Pre-trained Transformer) series, Google's BERT (Bidirectional Encoder Representations from Transformers) and more recently Gemini, and Facebook's RoBERTa (Robustly optimized BERT approach). It is notable that Gemini’s technical paper “Gemini: A Family of Highly Capable Multimodal Models”[13] mentions SDC’s. The technical paper mentions Silent Data Corruptions (SDC’s) as one of the challenges encountered during the training process. Specifically, it highlights the need to address SDC’s due to the unprecedented scale at which Gemini models are trained. The paper describes how the team developed new techniques, including deterministic replay and proactive SDC scanners, to detect and mitigate SDC events, ensuring stable training of the models. This demonstrates the importance of addressing SDC in large-scale training scenarios, as discussed in the Gemini paper.

As the cornerstone of modern NLP systems, LLMs play a pivotal role in advancing the capabilities of AI-powered applications and facilitating human-computer interaction in diverse domains. Understanding the architecture and capabilities of LLMs is crucial for comprehending their vulnerabilities to silent data corruptions and devising effective mitigation strategies to ensure their reliability and integrity in real-world deployment scenarios.

5.3 The LLM used – Llama

For this research, the Llama[8] implementation of LLMs is utilized. Llama[8] is a lightweight, pure C implementation designed for both training and inference of LLMs. The repository provides a comprehensive solution for training the Llama 2 LLM architecture using PyTorch and subsequently performing inference with a single C file. The primary focus of the Llama implementation is on simplicity and minimalism, providing researchers and developers with a straightforward approach to working with LLMs. By leveraging the Llama framework, users can train and inference LLMs with ease, even on resource-constrained environments. Additionally, Llama supports the loading and inference of Meta's Llama 2 models, extending its utility beyond the models trained specifically within the repository.

LLaMA-2 as described in the article "Everything you need to know about the best open-source LLM on the market..."[14] is a suite of open-source language models, ranging from 7 to 70 billion parameters, that stands out in the field of large language models (LLMs). It improves upon its predecessor, LLaMA-1, by pre-training over more data, utilizing longer context lengths, and optimizing its architecture for faster inference. Notably, LLaMA-2 invests heavily in the alignment process to create models optimized for dialogue applications, such as LLaMA-2-Chat, which rivals top proprietary LLMs like ChatGPT and GPT-4 in certain areas. More information about the model can be seen in figures 5.1 and 5.2.



Llama 2 was trained on **40% more data** than Llama 1, and has double the context length.

Llama 2

MODEL SIZE (PARAMETERS)	PRETRAINED	FINE-TUNED FOR CHAT USE CASES
7B	**Model architecture:**	**Data collection for helpfulness and safety:**
13B	**Pretraining Tokens:** 2 Trillion	**Supervised fine-tuning:** Over 100,000
70B	**Context Length:** 4096	**Human Preferences:** Over 1,000,000

Figure 5.1 – Model Architecture and Pre-Training

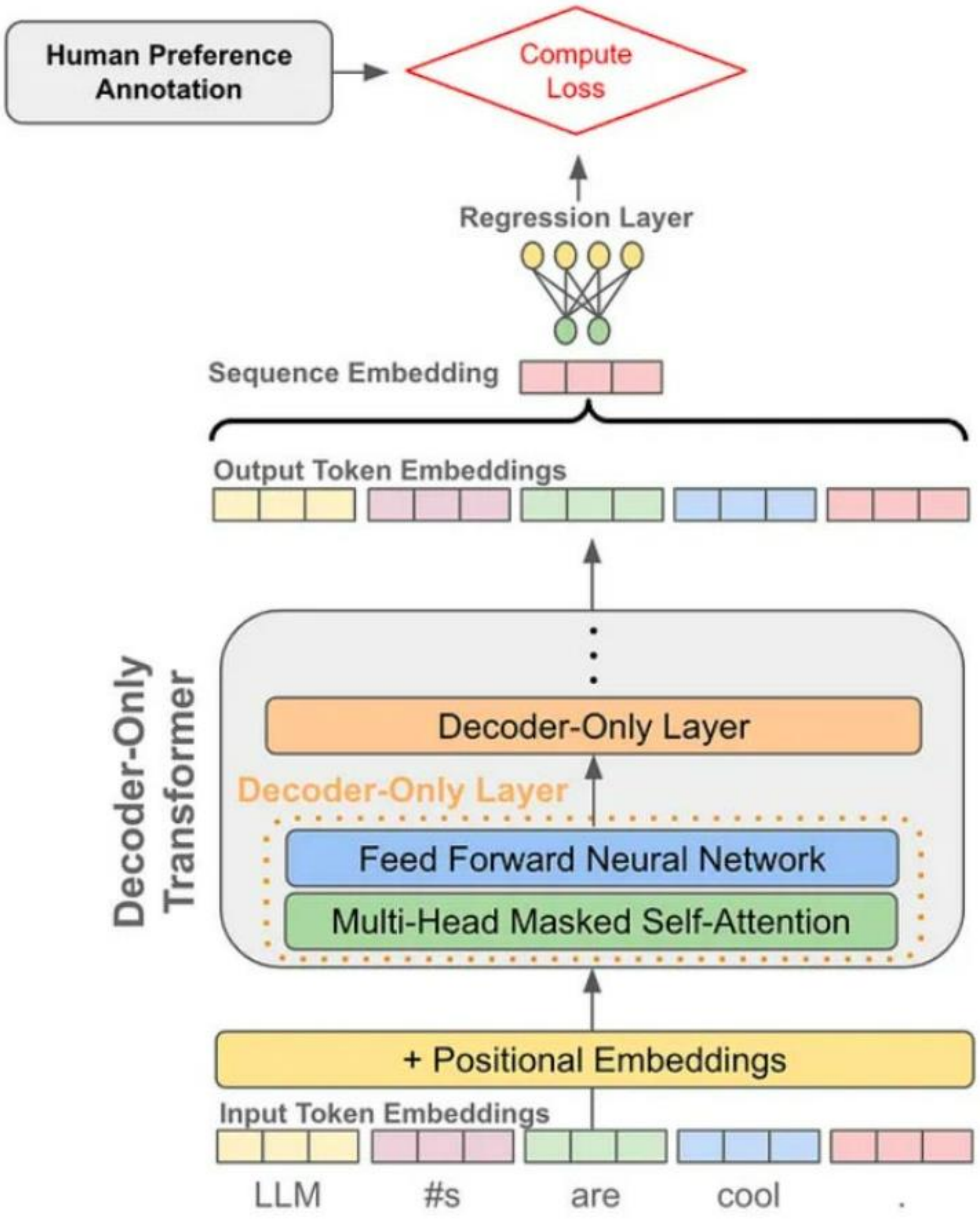


Figure 5.2 – Llama2 from the Ground Up

In our fault injection analysis, we concentrated on the Feed Forward Neural Network (FFNN) component within the decoder transformer of the model. Specifically, our focus was on the forward function of the program, where we targeted instructions related to matrix multiplication and vectors. By strategically infecting these instructions, we aimed to simulate potential faults or errors that could occur during the inference phase of the model's operation. It's important to note that we deliberately limited our fault injection to the inference part of the model and did not affect the training phase. This approach allowed us to isolate and assess the resilience of the model's FFNN component during inference, which is crucial for real-world deployment scenarios where the model is used for making predictions or generating outputs. By specifically targeting the inference process, we were able to evaluate the model's robustness and performance under potential fault conditions, providing insights into its reliability and suitability for practical applications.

5.4 The fault injection tool - Pin Tool

In the pursuit of understanding the behavior of Large Language Models (LLMs) under various fault conditions, a fault injection tool is indispensable. The Pin Tool[9] stands out as a powerful and versatile instrumentation framework that facilitates dynamic binary instrumentation of software applications. Developed by Intel, Pin[9] provides a robust platform for analyzing program behavior at the instruction level, enabling researchers to inject faults and observe their effects on program execution.

The Pin Tool[9] operates by dynamically instrumenting the binary code of the target application, allowing for the insertion of custom analysis routines or fault injection mechanisms. This capability is particularly valuable when studying complex and opaque systems like LLMs, where traditional debugging or profiling techniques may fall short.

By leveraging the Pin[9] Tool's flexibility, researchers can design fault injection experiments tailored to their specific research objectives. Whether simulating hardware faults, injecting software bugs, or perturbing system parameters, the Pin[9] Tool provides a reliable and efficient means of conducting fault injection studies.

In the context of this research, Pin[9] version 3.20 serves as the primary mechanism for injecting silent data corruptions (SDCs) into the Llama 2 LLM. Since the Llama 2 LLM is implemented as a single C program, integrating Pin[9] alongside it was straightforward, allowing for seamless instrumentation and fault injection. Through carefully crafted instrumentation scripts, faults can be introduced at various points in the LLM's execution flow, allowing researchers to observe how these corruptions manifest in the model's output.

The Pin Tool's[9] ability to seamlessly integrate with existing software systems, coupled with its extensive documentation and support, makes it an invaluable asset for fault injection research. As such, it plays a pivotal role in the experimental methodology employed to investigate the impact of SDCs on LLM behavior.

5.5 Injection Scenarios

In this section, we delve into the heart of our fault injection experiments, where we explore a myriad of scenarios to understand the behavior of the Llama 2 Large Language Model (LLM) under various fault conditions. Our aim is to elucidate the impact of silent data corruptions (SDCs) on the LLM's output and performance.

By systematically examining these injection scenarios, we aim to uncover insights into how SDCs manifest within the LLM and their repercussions on the model's output quality and stability. Through rigorous experimentation and analysis, we endeavor to contribute to a deeper understanding of the robustness of LLMs in the face of potential data corruptions.

In our investigation, particular attention was devoted to the forward function within the Llama[8] codebase, given its pivotal role in orchestrating the core operations of the Large Language Model (LLM). This function serves as the nexus where crucial computations are executed, including the invocation of the matmul routine responsible for performing essential matrix multiplications integral to the LLM's functionality. By scrutinizing this critical segment of the code, we aimed to gain a nuanced understanding of how silent data corruptions (SDCs) could potentially manifest and propagate within the LLM's computational pipeline.

Scenario 1:

In the initial scenario of our investigation, the focus was on identifying the assembly command `mulps xmm0, xmm4` within the forward function of the Llama codebase. This specific instruction is crucial as it signifies the point where floating-point multiplication operations are performed using the registers xmm0 and xmm4. Our objective was to pinpoint the occurrences of this command within the codebase and analyze its impact on the model's behavior when subjected to fault injection.

To initiate this test, we employed a targeted modification approach by deliberately altering the value stored in one of the two registers involved in the `mulps` operation. Specifically, we augmented the value by adding a constant (in this case, 1) to one of the registers. This deliberate manipulation aimed to simulate a silent data corruption scenario where a subtle alteration in the data being processed could potentially lead to unforeseen consequences in the model's output. This was performed on a fixed number of instructions (10000).

By executing this modified version of the forward function and observing the resultant output, we sought to discern any discernible changes or anomalies induced by the injected fault. This initial test served as a foundational step in our exploration of how silent data corruptions could influence the behavior of the Llama large language model.

Scenario 2:
In the second scenario, our investigation delved into the influence of error frequency on the behaviour of the Llama large language model. Unlike the previous scenario, where we focused on deliberate modifications to specific registers, this test aimed to explore the broader impact of varying error rates on the model's output.

Our approach involved systematically adjusting the frequency at which errors were introduced during the execution of the mulps xmm0, xmm4 instruction within the forward function. By incrementally increasing the error frequency, we sought to ascertain the threshold at which the model's behaviour became notably affected. This threshold represented the minimum error frequency at which the model exhibited problematic or erratic output.

For instance, we experimented with error frequencies ranging from 1 error per 10000 mulps xmm0, xmm4 instructions to higher frequencies, progressively increasing the likelihood of errors occurring during computation. By observing the model's response to varying error rates, we aimed to discern any discernible patterns or correlations between error frequency and the severity of output deviations. This analysis provided valuable insights into how the frequency of silent data corruptions influenced the stability and reliability of the Llama large language model's output.

Furthermore, it's noteworthy to mention that the error injected in this scenario remained consistent with the previous test, involving the deliberate alteration of one of the two registers by adding 1 to its value. This uniformity in the error injection methodology ensured that our comparative analysis focused solely on the impact of varying error frequencies, eliminating confounding variables and facilitating a clearer understanding of their effects on the model's behaviour.

Scenario 3:
In the third scenario, we undertook a more granular approach to error injection, aiming to explore the impact of individual bit flips within the 128-bit registers utilized by the mulps xmm0, xmm4 instruction. These registers, designed to store four floating-point numbers, each comprising 32 bits, provided a rich landscape for error analysis. To enhance the realism of our simulations, we devised a mechanism capable of introducing bit flips into a single bit of one of these numbers. Leveraging this mechanism, we meticulously tested each bit's significance, commencing with the most significant bit of the exponent and progressing through to the least significant bits of the mantissa. Our methodology ensured a systematic examination of each bit's influence on the output when flipped. Importantly, we maintained a constant error frequency while testing each bit, ensuring consistency in our experimental conditions. Upon identifying a bit with no discernible impact on the output, we augmented the error frequency to ascertain whether it would provoke any observable effects. This meticulous approach enabled us to unravel the nuanced effects of individual bit errors on the model's behavior, shedding light on the intricacies of error propagation within the system. We were influenced to test the effects of individual bits flips from Alibaba's paper "Understanding Silent Data Corruptions in a Large Production CPU Population"[2] were they show that different bit flips have different effects.

Scenario 4:

In the fourth scenario, our investigation pivoted towards discerning potential disparities arising from errors inserted into the xmm0 and xmm4 registers. Through an in-depth analysis of the llama2 code and its corresponding assembly instructions, we uncovered that xmm0 represents the input of the multiplication operation, while xmm4 corresponds to the weight involved in the computation. Armed with this understanding, we meticulously crafted experiments to gauge the impact of error insertion on each register. Employing varying error frequencies, we meticulously compared the model's output when modifying each register. To ensure a fair comparison, we maintained consistency in both the frequency and the specific bit flipped across tests involving each register. For instance, we conducted experiments with identical error frequency (1 error per 100 instructions) and flipped bit 8 in both xmm0 and xmm4 registers, subsequently analyzing and contrasting the resulting outputs. Furthermore, we broadened our investigation by testing diverse error frequencies and flipping different bits within each register, allowing us to glean insights into the nuanced effects of error propagation within the system. Through this rigorous comparative analysis, we aimed to elucidate any discernible discrepancies in the model's behavior stemming from error insertion into distinct registers, shedding light on the intricacies of error propagation mechanisms within the llama2 architecture.

Scenario 5:

In the fifth scenario, our investigation delved deeper into the structure of the llama2 code, particularly focusing on its composition into six distinct layers within the machine learning algorithm. Armed with this insight, we devised a sophisticated mechanism leveraging both Pin and modifications to the original program to precisely delineate and isolate each layer during execution. Subsequently, we leveraged this mechanism to selectively introduce errors into each layer individually, facilitating a granular assessment of their respective impacts on the model's output. Like previous experiments, we subjected each layer to tests encompassing a spectrum of error frequencies, maintaining consistency in the error frequency applied across all layers for comparative analysis. Through these meticulously designed experiments, we sought to elucidate any discernible variations in the model's behaviour resulting from error injection at different layers, thereby unravelling potential disparities in error propagation dynamics across the various stages of the llama2 algorithm.

Scenario 6:

In the final scenario of our research, we turned our attention to a different assembly command within the matmul function: "addss xmm0, xmm1". Like the previously examined "mulps xmm0, xmm4" command, this instruction is integral to the matrix multiplication computations performed within the machine learning algorithm. Our objective was to ascertain whether modifying this command would yield outcomes akin to those observed with the previous command. To evaluate this, we adhered to the same methodological approach employed in earlier scenarios. We conducted tests across various error frequencies and manipulated different bits, subsequently comparing the resultant outputs with those obtained from the "mulps" command. Through this comparative analysis, we aimed to glean insights into the consistency or divergence of error propagation effects between the two distinct assembly instructions utilized within the matmul function.

Scenario	**Description**
Scenario 1	Inserted a fixed number of faults, precisely 10,000, at the beginning of the execution. The primary objective was to observe the reaction of the algorithm when subjected to these injected faults. By introducing faults at the outset of execution, we aimed to gauge the algorithm's resilience and performance under such conditions, providing valuable insights into its robustness and potential vulnerabilities.
Scenario 2	Investigated the impact of varying error frequencies. Errors were introduced during the execution of the "mulps xmm0, xmm4" instruction within the forward function, with frequencies ranging from 1 error per 10,000 instructions to higher frequencies. The objective was to identify the threshold at which the model's behavior became notably affected and observe any patterns between error frequency and output deviations. The error injection methodology remained consistent with scenario 1, ensuring a focused analysis on the influence of error frequency.

Scenario 3	Examined the influence of individual bit flips within the 128-bit registers utilized by the "mulps xmm0, xmm4" instruction. Our objective was to understand how different bit flips affect the model's behavior. Employing a systematic approach, we introduced bit flips into each bit, from the most significant to the least significant, while maintaining a constant error frequency. This detailed analysis provided insights into the nuanced effects of individual bit errors on the model's output, offering a deeper understanding of error propagation within the system.
Scenario 4	Investigated the impact of errors inserted into xmm0 (input) and xmm4(wheight) registers within the llama2 architecture. We meticulously compared the model's output when modifying each register, maintaining consistency in error frequency and flipped bits. Our experiments aimed to reveal any differences in the model's behavior resulting from error injection into distinct registers, shedding light on error propagation mechanisms within the system.
Scenario 5	Explored the llama2 code's structure, divided into six layers within the machine learning algorithm. Using Pin and program modifications, we isolated each layer during execution, allowing us to introduce errors individually. By testing with varying error frequencies across all layers, we aimed to understand how errors at different stages impact the model's behavior. Through these experiments, we sought to uncover variations in error propagation dynamics across the layers of the llama2 algorithm.
Scenario 6	Investigated a different assembly command within the matmul function: "addss xmm0, xmm1". Like our previous examination, we aimed to understand how modifying this command would affect the model's behavior. Using the same methodological approach, we conducted tests across various error frequencies and manipulated different bits. We then compared the outputs with those obtained from the "mulps" command. This analysis aimed to reveal any consistency or divergence in error propagation effects between the two assembly instructions.

Table 5.1 – Injection Scenarios Summarized

Chapter 6

Results

6.1 Introduction

In this chapter, we delve into the results obtained from the series of fault injection experiments conducted on the LLAMA2 machine learning algorithm. The primary objective of these experiments was to assess the resilience and robustness of the algorithm in the face of various fault injection scenarios. By systematically introducing faults into critical components of the LLAMA2 codebase and observing the resulting effects on output behavior, we aimed to gain insights into the algorithm's susceptibility to errors and its ability to maintain functionality under adverse conditions.

Throughout this chapter, we present and analyze the outcomes of each scenario tested, providing detailed descriptions of the fault injection methods employed and their impact on algorithm performance. From examining the effects of modifying specific assembly commands to exploring the influence of error frequency and bit flips on output behavior, each scenario sheds light on different aspects of LLAMA2's fault tolerance capabilities.

Moreover, we compare and contrast the results obtained from different fault injection scenarios, identifying patterns, trends, and potential areas of vulnerability within the algorithm. By comprehensively analyzing the experimental data, we aim to contribute to a deeper understanding of the factors that influence the reliability and resilience of machine learning algorithms in real-world deployment scenarios.

Before we move to the results let’s look at the typical output of Llama2. The following output is taken when we give as input the stories15M.bin to the algorithm, we used this file as input for all scenarios.

Output:
Once upon a time, there was a little bunny named Benny. Benny loved to hop and play all day long. One day, Benny found a pretty rock in the forest. He picked it up and said, "This is so soft and pretty!"

Suddenly, a nosy squirrel named Sammy appeared. "What are you doing?" he asked. "I found a pretty rock!" Benny replied. "Why are you nosy?" Sammy questioned. "I just wanted to see it," Benny said.

Sammy was impressed by Benny's find and asked, "Can I see it?" Benny happily handed the rock to Sammy, but as Sammy touched the rock, he accidentally dropped it and it broke into many pieces. Benny was sad and scared that Sammy would be mad. But then, something unexpected happened. Sammy said, "I'm sorry, I didn't mean to break it. I just wanted to see it." Benny was relieved and happy that his friend wasn't mad. They both went on their way, looking for more pretty things to look at together.

6.2 Evaluation Methodology

To comprehensively assess the impact of fault injection scenarios on the LLAMA2 machine learning algorithm, we employed a multi-faceted evaluation approach encompassing various qualitative metrics. These metrics, namely Coherence, Correctness, Creativity, Naturalness, and Engagement, were selected to capture different aspects of output behavior and provide a holistic understanding of algorithm performance under different conditions.

Coherence: This metric evaluates the logical consistency and flow of the algorithm's output. A higher coherence score indicates output that is more logically connected and structured, reflecting the algorithm's ability to maintain coherence despite injected faults.

Correctness: Correctness measures the accuracy and fidelity of the algorithm's output compared to expected or ground truth results. A higher correctness score indicates output that closely aligns with expected outcomes, demonstrating the algorithm's ability to produce accurate results even in the presence of faults.

Creativity: Creativity assesses the algorithm's ability to generate novel and innovative outputs. A higher creativity score indicates output that exhibits originality and diversity, showcasing the algorithm's capacity to adapt and explore new solutions.

Naturalness: Naturalness evaluates the degree to which the algorithm's output resembles natural language patterns and expressions. A higher naturalness score indicates output that is more fluent, coherent, and human-like in its language usage, enhancing the overall user experience.

Engagement: Engagement measures the algorithm's ability to captivate and maintain the interest of the audience or end-users. A higher engagement score indicates output that is more compelling, immersive, and interactive, fostering deeper user engagement and satisfaction.

For each fault injection scenario, multiple tests were conducted, and the outputs generated were evaluated based on these metrics. Each output was assigned a score from 1 to 10 for each metric, with higher scores indicating better performance. Subsequently, an average score was computed for each metric across all tests within a scenario, providing a quantitative assessment of algorithm performance under different fault conditions.

For the evaluation process, we automated the assessment using ChatGPT[10], which scored outputs based on the metrics. An output waw given to ChatGPT for scoring, and an average score for the entire output was computed. This process was repeated 10 times for each test, ensuring robustness and reliability in the evaluation. Finally, a total average score was calculated, providing a comprehensive measure of the model's performance across multiple iterations. This automated approach ensured consistency, objectivity, and efficient handling of large volumes of experimental data, enhancing the reliability of our findings.

Based on this methodology we first evaluate the output of llama without inserting any faults. Here are the results:

LLAMA2 Output Evaluation (No Fault Injection)

Coherence: 8 - The story follows a clear sequence of events and maintains logical flow, ensuring readability and comprehension for the audience.

Correctness: 10 - There are no factual errors detected in the output, and the actions of the characters align with the established narrative, demonstrating the algorithm's precision and accuracy.

Creativity: 8 - While the story introduces characters and a conflict, it lacks significant innovation or unexpected twists, resulting in a moderately creative output that adheres to conventional storytelling conventions.

Naturalness: 8 - The dialogue feels natural and appropriate for the characters, contributing to the overall fluency of the story and enhancing the reader's immersion in the narrative world.

Engagement: 8 - The story effectively engages its intended audience, maintaining interest through character interaction and resolution of the conflict, albeit without introducing highly captivating or suspenseful elements.

Average Score: 8.4

These evaluation results serve as a crucial baseline against which we will compare the outcomes obtained after fault injection in subsequent experiments. By establishing the performance metrics of the LLAMA2 algorithm in its fault-free state, we gain a clear understanding of its inherent capabilities in terms of coherence, correctness, creativity, naturalness, and engagement. This baseline provides a reference point for assessing the impact of fault injection on the algorithm's output quality. By comparing the post-fault injection results to these baseline scores, we can identify any deviations or improvements in narrative quality caused by the introduced faults. This comparative analysis will enable us to evaluate the effectiveness of fault injection techniques in simulating real-world scenarios and their potential implications for the reliability and robustness of machine-generated narratives.

6.3 Results for each scenario

Scenario 1:
In the first scenario, where the value of a register was increased by one a fixed number of times, we aimed to gain insights into the effects of such faults on the LLAMA2 algorithm's output. Notably, errors were injected at the beginning of the execution, 10,000 errors to be exact, rather than being distributed periodically throughout the program's runtime. This approach allowed us to observe the immediate impact of faults on the narrative generation process. The evaluation results indicate a slight decrease in coherence and engagement compared to the fault-free baseline, suggesting that the injected errors may have disrupted the logical flow of the narrative to some extent. However, the correctness score remains relatively high, indicating that the essential factual accuracy of the narratives was largely preserved despite the injected faults. Creativity and naturalness scores also show minor fluctuations, suggesting that the injected errors did not significantly alter the overall narrative style or introduce major inconsistencies. Overall, this scenario provides valuable insights into the initial effects of fault injection on the LLAMA2 algorithm's performance, laying the groundwork for more comprehensive evaluations in subsequent experiments.

Metric	Score
Coherence	7.9
Correctness	8.7
Creativity	6.8
Naturalness	7.7
Engagement	7.3
Average Score	7.68

Table 6.1 – Metric scores for scenario 1

In analyzing the outputs of this scenario, a noticeable trend emerged: narratives often began with a coherent storyline but abruptly shifted to a different narrative halfway through. This phenomenon is likely attributable to the uniform injection of errors at the beginning of the program execution and to the fact that errors were injected in the same manner.

Example output:

“Once upon a time, there was a boy named Tim. Tim was a healthy boy who loved to eat. He liked to eat many things, but his favorite food was spaghetti.

One day, Tim's mom said, "Tim, you must complete your nap." Tim did not want to nap, but he knew he had to listen to his mom. So, Tim closed his eyes and took a long nap.

When Tim woke up, he felt good and ready to play. His mom said, "Good job, Tim! You had your long nap today." Tim smiled and said, "Thank you, Mom!" Then, he played and had fun with his friends.”

Conclusion:

As we can see the quality of the output deteriorates, looking at the metric scores the downgrade is not that high likely to the fact that the fix number of errors that were inserted were not that high, but as we can see the total effect it has on the outputs is very significant.

Scenario 2:

Scenario 2 focused on testing the impact of different error frequencies on the output of the program. Errors were introduced periodically throughout the execution of the program, aiming to simulate a more realistic fault injection scenario. In this experiment, the error injection method remained consistent, with the addition of the value 1 to one of the two registers. By varying the frequency of error injection, from low to high, the goal was to observe how different error rates affect the coherence, correctness, creativity, naturalness, and engagement of the output. This scenario aimed to provide insights into the relationship between error frequency and the quality of the program's output, shedding light on the optimal frequency range for fault injection.

We tested six different frequencies 10000, 5000, 1000, 500, 200, 100.

Note: Error frequency 10000 means 1 error is inserted every 10000 instructions.

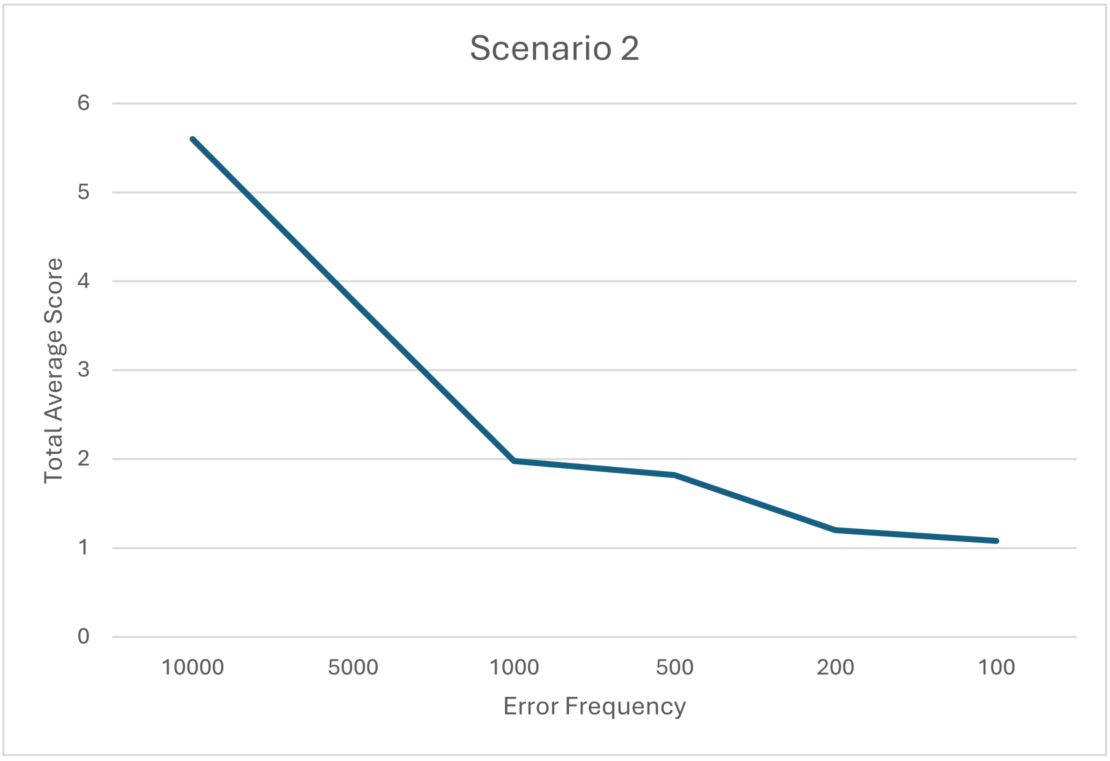


Figure 6.1 – Total Average Score per Error Frequency

Conclusion:
Scenario 2 demonstrated a clear correlation between error frequency and the quality of the program's output. As expected, as the error frequency increased, the coherence, correctness, creativity, naturalness, and engagement of the output deteriorated. This observation aligns with the anticipated outcome, indicating that higher error frequencies have a detrimental effect on the overall performance of the program. The results underscore the importance of carefully managing error rates in fault injection testing, as excessively high frequencies can significantly degrade the output quality, potentially leading to undesirable behavior or unreliable results. Thus, maintaining an optimal balance in error frequency is crucial to ensure the integrity and effectiveness of fault injection testing methodologies.

Scenario 3:

In this scenario bits of the registers involved in the multiplications were individually tested. Note that we used error frequency 2000 for all tests as we found from the previous scenario that a frequency in this range would insert notable errors in output. Initially we thought that 5000 would be the best frequency but after further testing we realized that when performing bit flips on bits with lower significance in the exponent part of the floating point number[11] or on bits in the mantissa part the effects of the bit flips were much lower than the change we had before (where we added 1 to the register), so we tried a higher frequency.

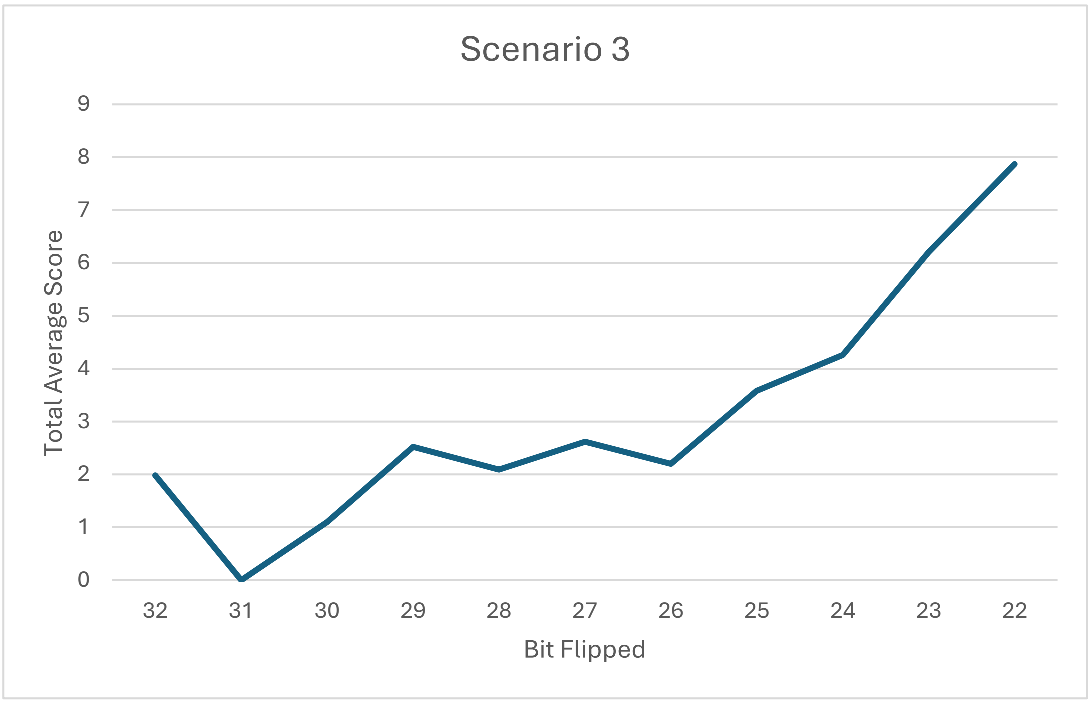


Figure 6.2 – Total Average Score per bit flipped

Note: We show the scores for bits 32-22 because the rest of the bits didn't show any significant fluctuation on the average score when flipped. For those bits to have any significant impact the frequency of fault injection needed to be unrealistically high. Bits 32-25 are the exponent part of the number and the rest 24-22 belong to the fraction part.

Conclusion:

As we can see from the graph, for the most part as we move down to less significant bits the total average score of the outputs increases, something that was expected from the nature of floating point numbers[11]. I should also comment on the effects when flipping bits 32 and 31. Bit 32 holds the value that determines the sign of the number (whether the number is positive or negative), changing the sign of the number seems to have a significant impact on the output. Bit 31 is the most significant bit of the number, flipping this bit seems to have very destructive effects on the outputs as they become completely unreadable, therefore the total score value is 0.

Scenario 4:

In this scenario, we delved into examining the difference between modifying the input (xmm0) and the weight (xmm4) registers within the multiplications. By focusing on these distinct registers, which respectively correspond to the input and weight values used in the machine learning algorithm, we aimed to understand how errors introduced at these different stages affect the program's output. Specifically, we conducted tests with four different error frequencies (500, 1000, 2000, and 5000), maintaining consistency by targeting bit 25 for error injection. This bit was selected based on its promising results from the previous scenario, providing a reliable starting point for our comparative analysis as this bit introduced faults that were noticeable and at the same time not completely destructive. Through this investigation, we sought to elucidate any disparities in output quality arising from errors introduced at the input and weight stages, shedding light on the relative impact of these modifications on the program's behavior and performance.

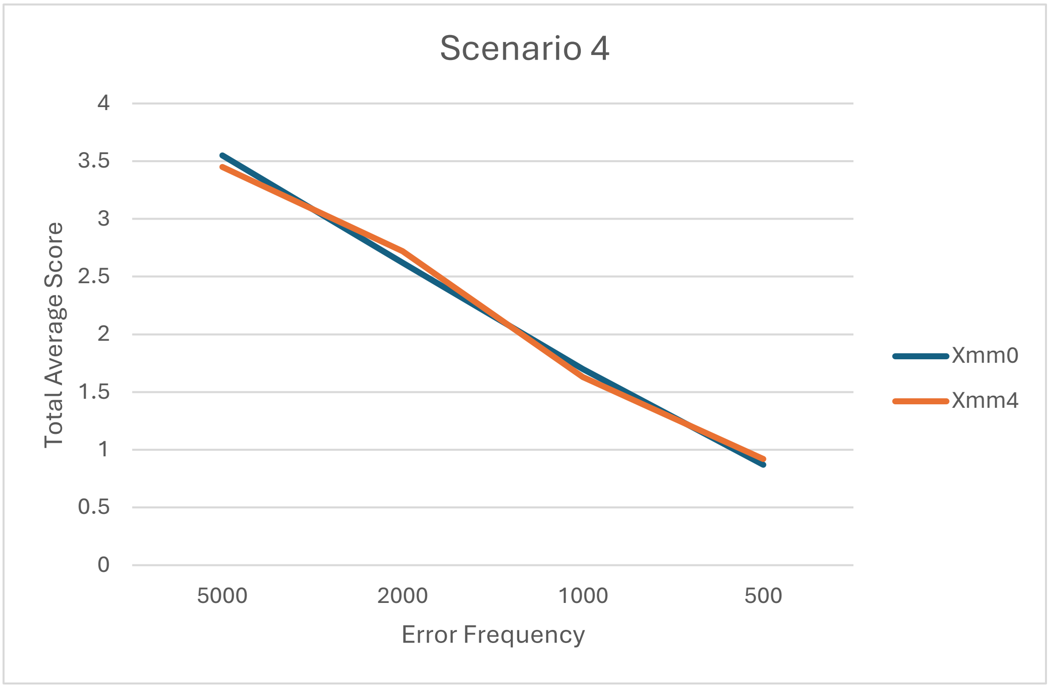


Figure 6.3 – Total Average Score per Error Frequency for xmm0 and xmm4

Conclusion:

This scenario revealed that there is no discernible difference in output quality when modifying the xmm0 (input) and xmm4 (weight) registers within the llama2 codebase. Despite varying error frequencies (500, 1000, 2000, and 5000) and consistent error injection at bit 25, our evaluations consistently demonstrated comparable results. This finding suggests that, at least within the parameters tested, errors introduced at either the input or weight stage do not significantly impact the program's output quality. Therefore, in terms of error injection effects, the distinction between modifying xmm0 and xmm4 registers appears negligible, highlighting a degree of robustness or insensitivity in the llama2 algorithm to such alterations.

Scenario 5:

In this scenario, we explored the impact of inserting bit flips into all six different layers of the llama algorithm. This comprehensive test aimed to assess whether the layers exhibited differential sensitivities to error injection and how such errors influenced the overall output quality. Similar to previous scenarios, we maintained consistency by testing the same frequencies (500, 1000, 2000, and 5000) for each layer and focusing on bit 25 for error insertion. By systematically evaluating the output quality across multiple layers under varying error frequencies, we sought to uncover potential insights into the algorithm's resilience and vulnerability to fault injection at different stages of computation.

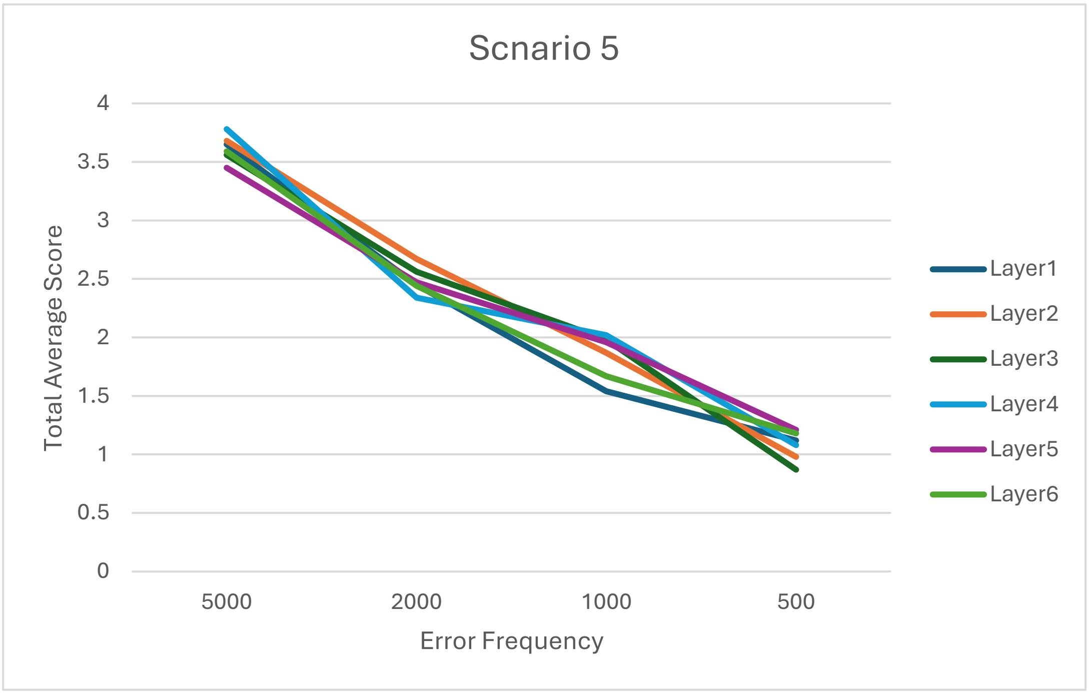


Figure 6.4 – Total Average Score for each Layer

Conclusion:

The analysis of inserting errors into different layers of the llama algorithm reveals that there are no significant differences in the output quality across the layers. Despite varying error frequencies and consistent error insertion at bit 25, the performance metrics remained largely consistent across all layers. This suggests that the llama algorithm may exhibit a uniform susceptibility to faults across its computational stages. The absence of notable variations implies that fault resilience may be evenly distributed throughout the algorithm, with no single layer significantly dominating in terms of error impact. Therefore, from the standpoint of error injection and output quality, the layers of the llama algorithm appear to demonstrate consistent behavior, emphasizing the algorithm's robustness under fault conditions across its computational hierarchy.

Scenario 6:

In the following scenario, we shift our focus from modifying the registers involved in the "mulps" command to altering those associated with the "addss" command within the matmul function called by the llama algorithm's forward function. While the "mulps" command handles matrix multiplications crucial to the algorithm's computations, the "addss" command performs addition operations, equally integral to the algorithm's functionality. By exploring the effects of modifying the "addss" command, we aim to discern whether similar outcomes to those observed in the "mulps" scenario emerge. This investigation provides insight into the robustness of the llama algorithm concerning different types of assembly commands and sheds light on potential vulnerabilities or consistencies in error propagation across distinct computational operations.

For this scenario we tested the error frequencies 100, 200, 500, 1000, 5000, 10000 here are the results:

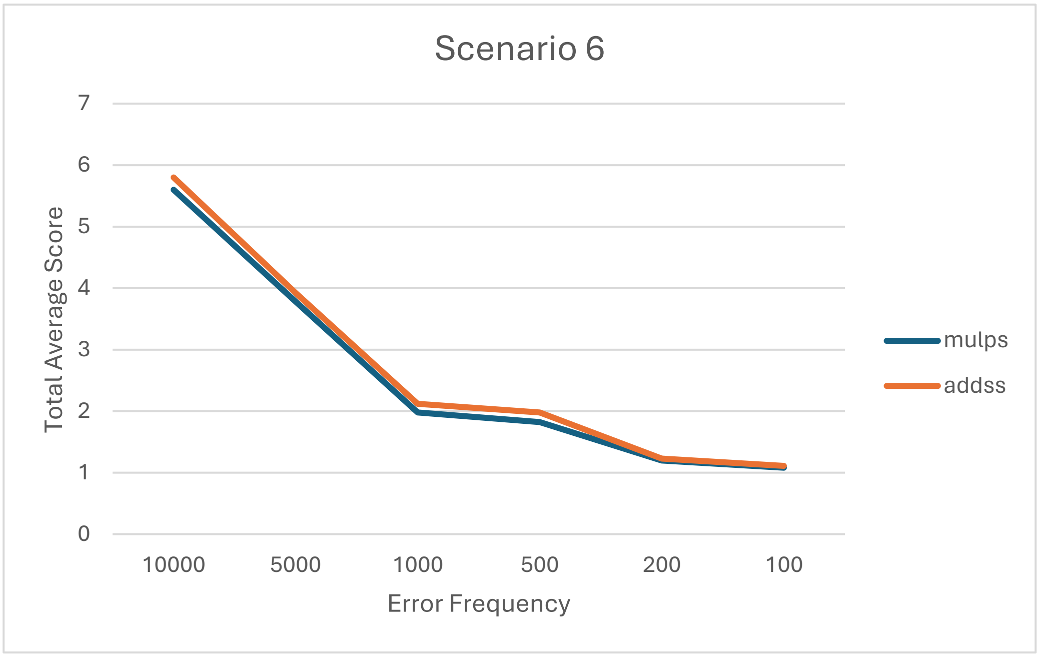


Figure 6.5 – Total Average Score for mupls and addss

Conclusion:

In conclusion, our exploration into modifying the "addss" command within the matmul function yielded results consistent with our findings from the "mulps" scenario. The outcomes suggest that injecting faults into the "addss" command does not produce discernible differences compared to modifying the "mulps" command. This observation underscores the algorithm's resilience to errors introduced at different stages of computation, highlighting a consistent behavior regardless of the specific assembly command targeted. Consequently, our study indicates that fault injection in either the "mulps" or "addss" commands elicits similar effects on the llama algorithm's output, emphasizing its robustness to computational errors across different arithmetic operations.

Chapter 7

Discussion

7.1 Summary

In this thesis, we embarked on a comprehensive exploration of fault injection techniques, reminiscing of silent data corruptions and their implications for machine learning algorithms, with a specific focus on the llama2 algorithm. The journey began with a motivation to study SDC's due to their unforeseen nature. Building upon this foundation, we delved into an overview of Silent Data Corruptions (SDC) and their potential impact on computational systems, highlighting the critical need for effective fault detection and mitigation strategies.

Our investigation then transitioned to an examination of various detection tools, including OpenDCDiag and the Intel Data Center Diagnostic Tool, each offering unique capabilities for assessing CPU health and detecting potential faults. Following this, we conducted experiments using the CloudLab infrastructure to simulate real-world scenarios and test different nodes using the detection tools. Then we evaluated the performance of the llama2 algorithm under different fault conditions.

The heart of the thesis lay in the injection scenarios, where we meticulously designed and executed a series of experiments to understand how the algorithm responds to injected faults. From modifying assembly commands to altering error frequencies and flipping individual bits, each scenario provided valuable insights into the algorithm's behavior and resilience. Through systematic evaluation based on metrics such as coherence, correctness, creativity, naturalness, and engagement, we were able to assess the impact of injected faults on the algorithm's output quality.

Our findings revealed intriguing patterns and trends, shedding light on the algorithm's susceptibility to different types of errors and providing valuable insights into fault tolerance mechanisms. From observing the degradation of output quality with increasing error frequencies to exploring the nuances of error propagation across different layers of the algorithm, each scenario contributed to a deeper understanding of the algorithm's behavior in fault-prone environments.

In conclusion, this thesis represents a comprehensive investigation into fault injections, simulating silent data corruptions, and their implications for machine learning algorithms, with a specific focus on the llama2 algorithm. By performing practical experimentation, we have gained valuable insights into the algorithm's resilience and identified avenues for future research in fault tolerance and reliability. Overall, this study contributes to the broader discourse on system reliability and lays the groundwork for developing more robust machine learning algorithms capable of withstanding the challenges posed by silent data corruptions.

7.2 Future

In looking towards the future, several avenues for further research and development emerge from the findings and insights gained in this study. One promising direction involves the refinement and enhancement of fault injection techniques to more accurately simulate real-world fault scenarios. By incorporating a broader range of fault types and patterns, researchers can gain a more comprehensive understanding of algorithmic behavior under diverse fault conditions.

Additionally, there is a pressing need for the development of more sophisticated fault detection and mitigation strategies tailored specifically to machine learning algorithms. This could involve the integration of machine learning techniques themselves, leveraging anomaly detection algorithms to identify and mitigate the effects of silent data corruptions in real-time.

Furthermore, the exploration of fault tolerance mechanisms within machine learning algorithms represents a rich area for future investigation. By designing algorithms with built-in resilience to computational errors, researchers can mitigate the impact of faults on algorithmic performance and ensure reliable operation in critical applications.

Moreover, there is a growing interest in exploring the intersection of fault tolerance and interpretability in machine learning algorithms. By understanding how faults manifest in algorithmic outputs, researchers can develop methods for identifying and explaining the underlying causes of errors, enhancing trust and confidence in algorithmic decision-making processes.

In addition to the outlined avenues for future research, a particularly intriguing prospect lies in investigating fault injection within the training phase of LLAMA2, especially considering the findings about Google's Gemini[13], which indicated that training processes can indeed be influenced by silent data corruptions. Exploring fault injection within the training phase of LLAMA2 not only aligns with the overarching objective of comprehensively assessing algorithmic resilience but also offers valuable insights into the vulnerability of machine learning models during their developmental stages. By scrutinizing how SDC's affects the training dynamics, researchers can gain deeper insights into the robustness of the learning process and devise strategies to fortify algorithms against such adversities. However, it's crucial to acknowledge the considerable computational and temporal demands associated with this line of inquiry. Conducting extensive fault injection experiments during training necessitates substantial computational resources and meticulous planning to ensure the reliability and reproducibility of results. Nonetheless, the potential insights garnered from such endeavors can significantly advance our understanding of fault tolerance mechanisms within machine learning algorithms, ultimately contributing to the creation of more resilient and dependable systems.

Finally, as machine learning algorithms continue to be deployed in increasingly complex and critical applications, there is a need for greater collaboration and knowledge sharing across disciplines. By bringing together experts in machine learning, fault tolerance, system reliability, and related fields, researchers can leverage complementary expertise to tackle the multifaceted challenges posed by silent data corruptions and ensure the continued advancement of reliable and robust machine learning algorithms.

In summary, the future holds exciting opportunities for further research and development in the field of fault tolerance for machine learning algorithms. By addressing the challenges posed by silent data corruptions and advancing the state-of-the-art in fault detection, mitigation, and resilience, researchers can pave the way for the continued growth and adoption of machine learning technologies in critical applications.

References

[1] Mitigating the effects of silent data corruptions at scale by Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, Sriram Sankar. Available: https://engineering.fb.com/2021/02/23/data-infrastructure/silent-data-corruption/

[2] SOSP 2023: Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding Silent Data Corruptions in a Large Production CPU Population. In Proceedings of the 29th Symposium on Operating Systems Principles. 2023 Available: https://dl.acm.org/doi/10.1145/3600006.3613149

[3] CloudLab: https://www.cloudlab.us/user-dashboard.php#profiles

[4] opendc diagram: https://github.com/opendcdiag/opendcdiag

[5] Intel Data Center diagnostic tool: https://www.intel.com/content/www/us/en/support/articles/000058107/processors/intel-xeon-processors.html

[6] CloudLab automations tool: https://gitlab.flux.utah.edu/jacob/portal-tools

[7] Processor Tested in CloudLab: https://www.intel.com/content/www/us/en/products/sku/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz/specifications.html

[8] Llama2 llm: https://github.com/karpathy/llama2.c/tree/master

[9] Pin: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[10] ChatGPT: https://chatgpt.com/auth/login?sso=&oai-dm=1

[11] Floating point numbers: https://www.log2base2.com/storage/how-float-values-are-stored-in-memory.html

[12] Proceedings of the 20th Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA: Hochschild, J., Turner, Y., Mogul, J. C., Govindaraju, M., Ranganathan, P., Culler, D., & Vahdat, A. (2021). Cores that don't count. Available: https://dl.acm.org/doi/10.1145/3458336.3465297

[13] Gemini Team Google, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul R. Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Jack Krawczyk, Cosmo Du, Ed Chi, Heng-Tze Cheng, Eric Ni, Purvi Shah, Patrick Kane, Betty Chan, Manaal Faruqui, Aliaksei Severyn, Hanzhao Lin, YaGuang Li, Yong Cheng, Abe Ittycheriah, Mahdis Mahdieh, Mia Chen, Pei Sun, Dustin Tran, Sumit Bagri, Balaji Lakshminarayanan, Jeremiah Liu, Andras Orban, Fabian Güra, Hao Zhou, Xinying Song, Aurelien Boffy, Harish Ganapathy, Steven Zheng, HyunJeong Choe, Ágoston Weisz, Tao Zhu, Yifeng Lu, Siddharth Gopal, Jarrod Kahn, Maciej Kula, Jeff Pitman, Rushin Shah, Emanuel Taropa, Majd Al Merey, Martin Baeuml, Zhifeng Chen, Laurent El Shafey, Yujing Zhang, Olcan Sercinoglu, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, Alexandre Frechette, Charlotte Smith, Laura Culp, Lev Proleev, Yi Luan, and Xi Chen et al. (1245 additional authors not shown). "Gemini: A Family of Highly Capable Multimodal Models." arXiv:2312.11805 (cs.CL), April 2, 2024, version 2.
Available: https://doi.org/10.48550/arXiv.2312.11805.

[14] Wolfe, C. R. (2023, August 14). LLaMA-2 from the Ground Up: Everything you need to know about the best open-source LLM on the market...
Available: LLaMA-2 from the Ground Up - by Cameron R. Wolfe, Ph.D. (substack.com)