Individual Diploma Thesis

INTEGRITY CHECK METHODOLOGY FOR DATA ARRAYS CORRUPTION DURING VMIN TESTS

Andreas Siokouros

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2023

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

INTEGRITY CHECK METHODOLOGY FOR DATA ARRAYS CORRUPTION DURING VMIN TESTS

Andreas Siokouros

Supervisor

Yiannakis Sazeides

The Individual Diploma Thesis was submitted for partial fulfilment of the requirements for obtaining a degree in Informatics of the Department of Informatics of the University of Cyprus

May 2023

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my thesis advisor Mr Yiannakis Sazeides as well as Mrs Georgia Antoniou for their unwavering support, invaluable guidance, and patience throughout my research journey. Their expertise, insights, and constructive feedback have been instrumental in shaping my research, and I could not have completed this thesis without their invaluable assistance.

Furthermore, I would like to acknowledge the support and encouragement of my family, friends, and colleagues, who have provided me with the motivation and inspiration to persevere through the challenges of this research project.

Finally, I would like to express my gratitude to the university and department for providing me with the resources and opportunities to conduct this research.

Abstract

Integrity check in CPU voltage drop is important because it ensures the reliability and stability of the computer system. When the CPU voltage drops below the recommended level, it can cause errors and crashes, and may even damage the hardware components. The integrity check monitors the voltage levels and alerts the system if it falls below the safe threshold. This allows for timely intervention to prevent any potential damage or failure of the system.

Additionally, the integrity check can help identify any underlying issues that may be caused by the voltage drop, allowing for prompt troubleshooting and resolution. Therefore, the integrity check is crucial for maintaining the proper functioning of the CPU and ensuring the longevity of the computer system.

This thesis is specialized on expanding the integrity check of the characterization framework[6].

We want to confirm the hypothesis that the voltage drop causes unwanted changes in the machine and more specifically in the data arrays. in this research we discovered that in addition to the system crashes caused by the Vmin process, there are also other types of crashes that we tried to identify. In our attempt to create an additional level of integrity check we finally managed to identify bit flips in memory and confirm our initial hypothesis.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

There are many reasons for better integrity check. Large voltage noise is a threat to robust execution because when the supply voltage drops below a certain threshold, timing violations or bit flips may occur. This may lead to silent data corruption (SDC), application or system crashes and general system instability. Having a methodology that identifies the minimum operational voltage is crucial for the correct functionality of a By having a strong integrity check in place, it is possible to identify and troubleshoot any issues that may occur in these systems, helping to ensure that the device is operating correctly and efficiently. This can help to improve the overall performance and reliability of the device and can help to prevent problems that could impact the device's operation.

1.2 Problem definition

The characterization framework[6] is used for Vmin tests. These tests can involve monitoring voltage levels during different CPU-intensive tasks or using specialized diagnostic tools to analyze the power delivery and voltage regulation of the CPU and associated components. The current implementation of the framework has

limitations regarding problem identification. In this research I tried to prove the existence of silent data corruptions that a di/dt virus may generate, check If they do not affect the correct functionality of the system and if that is the case, how to monitor them.

1.3 Contributions

In the beginning, it was very important to understand how changes in the voltage and frequency affect power consumption. Then understand and study the hardware and software that was going to be used, especially GeST[5], Juno board[3] and the Characterization framework[6]. Furthermore, researching and studying past papers related to Integrity check and Vmin test characterization framework[6] to have a good background. And then finding out new configurations that would be interesting to test the performance and results of Vmin test. Also, finding problems that may occur throughout this process that were not identified previously. The purpose is to establish a methodology that will identify silent data corruptions during a Vmin test and the evaluate that methodology.

1.4 Thesis structure

Chapter 2:

Short review of stress test and voltage noise (di/dt) viruses and explain what the Vmin test is and its uses.

Chapter 3:

Introduction to the Characterization framework[6], data structure, functionality, how is used and its applications. Explain how integrity check is part of it.

Chapter 4:

This chapter will define the integrity check and its functionalities. In addition, explain the usage and the reason behind the extension of the current integrity check and why there is the need of memory overload when is used.

Chapter 5:

This chapter is focused on explaining how the characterization framework[6] and the integrity check are merged. To do this, the usage of signals comes in place. For the sake of better understanding the explanation of the new input parameters needs to be done and what they actually do. In case of expansion, there are certain things that need to be taken into consideration.

Chapter 6:

The combinations of the experiment's scenarios held to monitor different performance metrics and statistics with different client-side configurations. Give a brief summary of the machine characteristics and the current setup.

Chapter 7:

The results, graphs, validations, and comparison in different experiments.

Chapter 8:

State the future work needed.

Chapter 9:

General conclusions, describing some lessons learned.

Chapter 2

Background

2.1 Stress Test

A CPU (Central Processing Unit) stress test is a procedure for assessing its performance and stability under heavy workloads. It entails running specific software that strains the processor or putting the CPU to tax operations. The CPU's capacity to withstand high temperatures, maintain steady clock rates, and deliver reliable performance without faults or crashes is assessed by the stress test. This kind of test aids in evaluating the CPU's dependability, cooling effectiveness, and general capacity.

Stress testing a CPU is typically done in different situations as in System Stability Testing where Stress testing can assist guarantee that the CPU and other components are stable under high workloads before deploying a new system or after making hardware changes. It aids in the detection of any potential problems or instability, including overheating, abnormal voltage patterns, and system crashes. Additionally, stress testing

can be used for overclocking to improve performance. Overclocking entails raising the CPU's clock speed over its factory defaults. In this situation, stress testing is essential to confirming the stability and functionality of the CPU under the higher clock speeds. It aids in determining whether the overclocked settings can be maintained without leading to freezes, instability, or overheating of the machine. Furthermore, it serves as Benchmarking and Performance Evaluation: By subjecting a CPU to taxing workloads or running specialist software, stress tests can be utilized to benchmark a CPU's performance. This makes it possible to compare various CPUs or combinations, giving information about their varying performance levels. To continue with, Temperature and Cooling Assessment can be used to measure the CPU's temperature under demanding conditions. Users can assess the performance of their cooling solution and determine whether any adjustments are required to maintain ideal temperature levels by keeping an eye on the temperature throughout the test. Overall, stress testing a CPU helps to assure system stability, evaluate performance, analyze overclocking capability, and confirm the efficacy of cooling methods. For gamers, hobbyists, and professionals who need high-performance computing and want to push their computers to their limits, it is especially pertinent.

CPU stress tests are generally safe when used appropriately and with the appropriate safety measures. To avoid overheating, it's crucial to keep a constant eye on the CPU temperature throughout the test. To efficiently dissipate the heat, adequate cooling systems should be in place, such as fans or liquid cooling. In order to support the growing power needs, it is also imperative to guarantee a consistent and adequate power supply. There is a small chance of system breakdowns or instability during stress testing, which try to assess system stability. The risk of data loss can be reduced by having backup files and stored work. The safety of the CPU and entire system can be preserved by being aware of these elements and keeping an eye on the system while the test is running.

Several factors may trigger a CPU to fail a stress test. Overheating is one frequent cause, as stress tests place intense workloads on computers that produce a lot of heat. The CPU may overheat and fail the test if the cooling system is insufficient or broken. An unstable or insufficient power source can cause voltage instability, which is another factor that might affect CPU performance. Inadequate cooling, such as obstructed airflow or malfunctioning fans, might prevent heat from being dissipated and result in an abnormal rise in CPU temperature during the stress test. Additionally, instability may happen if the

CPU has been overclocked above its safe limits, leading to test failures or system crashes. Stress testing may potentially reveal hardware flaws or manufacturing problems that result in failures. To ensure a steady and effective stress test, the root reason must be found and any cooling, power supply, overclocking, or hardware issues must be resolved.

2.1.1 Applications

CPU stress testing are useful in a variety of circumstances. In order to assure the CPU's stability under severe workloads and find any potential problems or instabilities, they are frequently used for system stability testing during system assembly, configuration changes, or hardware upgrades. Stress tests are used by overclockers to verify the stability and performance of their CPUs under conditions that are more demanding than the default settings. Stress tests, which subject the CPU to heavy workloads, produce useful data for performance benchmarking and comparison, assisting in the choice of CPUs based on processing speed, multitasking ability, and power economy. It is also essential for assessing the efficacy of cooling solutions since they keep an eye on temperature levels and guarantee optimum cooling under prolonged, high workloads.

Furthermore, in software development and testing environments, stress tests aid in assessing program performance, identifying bottlenecks, and gauging CPU resource use. In conclusion, CPU stress tests are critical tools for evaluating stability, overclocking, performance, cooling, and software optimization.

2.2 Voltage noise

Voltage noise is defined as undesirable variations or disruptions in the electrical voltage level. It is distinguished by fast fluctuations or variations in the voltage signal that differ from the expected or desired level. These fluctuations can occur in a variety of electronic systems or components, such as power supply, circuitry, or communication channels. Voltage noise can be caused by a variety of factors, including electromagnetic interference, circuit noise, or insufficient power control. It can have an impact on the performance and dependability of electronic devices, perhaps resulting in signal degradation, data mistakes, or malfunctions. Shielding, correct grounding, and effective

power management techniques are used to reduce voltage noise and ensure the steady and reliable operation of electronic equipment.

2.2.1 Voltage noise virus

“Voltage-noise” or “dI/dt” viruses try to increase CPU voltage variations and they are used to characterize the worst-case voltage droop. Stress tests try to produce a quick transition from extremely low to very high current consumption rather than maintaining a continuous high current consumption. Because low voltage operation might cause malfunctions, dI/dt viruses are excellent timing-error stability tests. The lowest voltage at which a dI/dt virus works correctly can be used to determine where to set the CPU's operational voltage (for a certain operating frequency).

2.3 Vmin

2.3.1 Explain

"Vmin" typically refers to the minimum voltage level in a given system or electronic component. It represents the lowest allowable voltage that the system or component can operate at while still maintaining proper functionality. Vmin is an important parameter to consider, as operating below this minimum voltage level can lead to performance degradation, instability, or even complete failure of the device or system. It is often specified by manufacturers to ensure proper operation and reliability of electronic components. By adhering to the specified Vmin, designers and users can ensure that the system or component operates within the acceptable voltage range and functions as intended.

To utilize Vmin successfully, first obtain the Vmin requirements from the electronic component or system's manufacturer or datasheet. The minimum voltage level required for proper operation is specified in these specifications. Next, select a power supply capable of providing voltages greater than the specified Vmin and guarantee its stability and regulation. To maintain the voltage within an acceptable range, use voltage control techniques such as voltage regulators or monitoring circuits. Verify that all components are intended to operate within the specified Vmin range throughout system integration.

To ensure stability and reliability, thoroughly test and validate the system under a variety of voltage circumstances, including near or slightly below the Vmin level. Monitor the voltage levels in the system continuously and perform regular maintenance to prevent voltage decreases below the minimum threshold. Vmin can be used efficiently to assure the appropriate operation and longevity of electronic components and systems by following these methods.

2.3.2 Usage

The minimum voltage level, or Vmin, is used in a variety of settings to assure the dependable and optimal operation of electronic components and systems. It is determined throughout the design and manufacturing processes, and it specifies the minimum voltage at which the component may perform properly. Vmin is taken into account during system integration, to ensure that the power supply produces voltages over the set minimum for all components, ensuring proper functionality. Vmin is considered in power management strategies for efficient operation, particularly in battery-powered devices, by using power-saving measures and voltage scaling. Furthermore, voltage margin testing can be used to examine system reliability under different voltage situations, ensuring that the system remains stable even when it is close to or slightly below the stipulated Vmin. Following the prescribed Vmin values provided by component manufacturers are critical for preserving the dependability, longevity, and functionality of electronic systems throughout their lifecycle.

Chapter 3

Characterization Framework

3.1 Characterization Framework

The Characterization Framework[6] is a framework designed to implement the Vmin test procedure. Essentially, it is a guide for defining the actions and controls that are performed in the experiment. It also determines what the data will be and

what the results will be at the end of each experiment. Also, through this the communication between the host pc and the Juno board[3] is achieved.

It's written in python, and it uses Json as input format. It accepts as inputs parameters like frequency, Vmin steps, workloads and performs a Vmin test. It returns the Vmin of the system along with the type of errors it encounters. This framework is tested on a Juno board[3] but it can be easily extendable for other platforms.

3.1.1 Input / Output

First of all, since there is a connection between the host pc and the Juno board[3], the proper input needs to be validated to make sure that the connection can be established. For that, all the necessary information is passed through (for example the Juno board IP address, the serial port that is connected to).

Also, the experiments need to be defined in the file. For that, it contains all the necessary data to determine the aspects of the experiment. For example, it must specify on which processor and on which processor cores the experiment is to be carried out, its duration, the parameters that are used, the starting and finishing voltage and how many repetitions of the experiment are to be performed.

To talk about the output, it is saved in a mongo DB database, the column of which is determined from the Json input file and needs to be renamed after every experiment. To be able to read the output, the printPassSDCcrashes.py file needs to be run.

Example input:

Experimental setup: 1st experiment

Target hostname:10.16.20.171

Target SSH username:

Target SSH password:

Serial port: COM8

MongoDB: Juno

Mongo Col: asioko01a53_virus_5_repetitions (Save folder)

Platform: juno_a53

Cores frequency: [950, 0, 0, 950, 950, 950]

Repetitions: 5

Start voltage: 1000

End voltage: 0

Voltage decrement: 10

Virus: 107_5330_26s

Example output:

PASS 920 107_5330_26s EXEC_TIME 30.46 WORKLOAD [0] DOC 6448fdc09230876dd5152828

PASS 920 107_5330_26s EXEC_TIME 30.45 WORKLOAD [3] DOC 6448fdc09230876dd5152828

PASS 920 107_5330_26s EXEC_TIME 30.45 WORKLOAD [4] DOC 6448fdc09230876dd5152828

PASS 920 107_5330_26s EXEC_TIME 30.46 WORKLOAD [5] DOC 6448fdc09230876dd5152828

PASS 910 107_5330_26s EXEC_TIME 30.46 WORKLOAD [0] DOC 6448fe489230876dd5152829

PASS 910 107_5330_26s EXEC_TIME 30.46 WORKLOAD [3] DOC 6448fe489230876dd5152829

PASS 910 107_5330_26s EXEC_TIME 30.45 WORKLOAD [4] DOC 6448fe489230876dd5152829

PASS 910 107_5330_26s EXEC_TIME 30.45 WORKLOAD [5] DOC 6448fe489230876dd5152829

SYSTEM_CRASH 900 31

In this example of output, the information that is provided helps identify what happens in each step of the integrity check experiment. In each line information about the state, the current voltage, which is the virus that is running, the execution time and at what CPU core the virus is running are provided. Also, if a crash appears, it identifies its type and at what voltage it was caused.

3.1.2 Data Structures

The characterization framework[6] consists of several subprograms used to implement its purpose. Some of the main structures that are used are under the files Executor.py, serialHandler.py, Workload.py, printSDCcrashes.py and mongoDBhandler.py. Briefly, I am going to present some classes that are part of the Executor.py.

Executor.py

Under the Executor.py folder, many sub-classes are implemented. Some of the most important and most used ones are:

- NextVoltageGenerator: Helper class that returns the next voltage. Also, it helps in comparing between random and incremental steps.
  Parameters:
  - highVoltage
  - lowVoltage
  - voltageStep
  - method
  - voltageList[]

  Functions:
  - Cal_next
  - Get_vol

- SSHhandler: Class that handles ssh commands to the target machine
  Parameters:

- targetHostName
- targetSSHusername
- targetSSHpassword

Functions:

- lastCommandStatus
- lastCommandOut
- lastCommandErr
- hasCommandFailed
- executeCommand

- workloadHandler: Class that handles each workload

  Parameters:

  - experimentID
  - submitFromFramework
  - processIDtable[]
  - integrityPID (This project's implementation)
  - Many signals that represent different situations

SUCCESS
FAIL
CHK_ALIVE_FREQ
BIT_FLIP (This project's implementation)
WORKLOAD_SCRIPT_ABORTED
SYSTEM_CRASHED
WORKLOAD_SCRIPT_FINISHED_SUCCESFULLY
MAX_CHK_ALIVE_TRIES
UNRESPONSIVE_TIMEOUT

  Functions:

  - setVoltageOfObservation
  - getVoltageOfObservation
  - startWorkload
  - calculateCPUPowerSSH

- calculateCPUPowerSerial
- calculateMeasurementValues
- calculateErrorInfo
- calculateWorkloadStatus
- waitWorkloadToFinish: In this function, the integrity check takes place using the LinkedList.cpp program.

- Executor: Class that handles each execution

  Parameters:

  - DEF_TARG_HOST
  - DEF_USER
  - DEF_PASSWD
  - WAIT_FOR_AUTO_RESTART
  - AUTO_RESTART_LOW_THRESH
  - MAX_FREQUENCY
  - CORE_PER_PMD
  - RELATIVE_FREQ
  - CONTINUE_AFTER_APP_CRASH_SDC
  - Many signals that represent different situations

ALIVE
APP_CRASH
SYSTEM_CRASH
SDC_OCCURED
SUCCESS
FAIL

  - Execute_full_exp: In this function, the initialization of the integrity check takes place using the LinkedList.cpp program.

- convertJsonToObject: In this function, the program identifies which file to use for integrity check and what other parameters need to be initialized for this process.
- configureJunoA53
- sendRebootCommand
- printRunOutput

3.1.3 Functionality

The framework has many different functions that serve different purposes. Some of the most important ones are under the Executor.py folder. I choose to present the parts that I have edited to create the integrity check.

Execute_full_exp function under Executor class

```
if measureIntegrity:
        integrity_check_command = self.integrityCheckScript + " &"
        print(integrity_check_command)
        code = self.serial.sendCMD(integrity_check_command, waitTime=WorkloadHandler.UNRESPONSIVE_TIMEOUT);
        ser_out = self.serial.read()
        pId = ser_out.split('\n')[1].split()[1]
        self.integrityPID = pId
        while not "Initialized Complete" in str(ser_out):
            ser_out = self.serial.read()
            print(ser_out)
            time.sleep(5)
        print(ser_out)
```

In this part of the function, the program IF it must perform an integrity check, it follows the code above. Firstly, the command that is sent through the serial port has to be established. Then, the process ID must be saved in the integrity check PID. Finishing, the program waits until the initialization finish.

waitWorkloadToFinish function under workloadHandler class

```
if measureIntegrity:
                print(self.userVoltage)
                from setVoltageThroughSerial import setVoltage
                setVoltage(1, float(self.userVoltage) / 1000, JUNO_DEBUG_PORT)
                integrity_check_command_kill = "pkill -SIGUSR1 LinkedList"
                code = self.serial.sendCMD(integrity_check_command_kill,

waitTime=WorkloadHandler.UNRESPONSIVE_TIMEOUT);
                ser_out = self.serial.read()
                while not "Integrity Check Complete" in str(ser_out):
                   ser_out = self.serial.read()
                   print(ser_out)
                   time.sleep(5)
                print(ser_out)
                if "FAIL" in str(ser_out):
                   return WorkloadHandler.BIT_FLIP
```

In this part of the function, the program IF it must perform an integrity check, it follows the code above. Firstly, the voltage needs to be set to nominal to perform the integrity check. Then, the command (signal) that wakes up the LinkedList.cpp program needs to be formed and then sent along with the serial command. Finishing, the program waits

until the integrity check finishes and respond with its answer (see section 6.1.2 for more information).

convertJsonToObject function under Executor class

```
try:
        integrityCheckScript=data["integrityCheckScript"]
        userVoltage=data["userVoltage"]
    except(AttributeError,KeyError) as err:
        integrityCheckScript=None
        userVoltage=None
```

In this part of the function, the Json file parameters that matter for the integrity check get converted to object. The integrity check file name gets saved to integrityCheckScript parameter, and the user test voltage gets saved to userVoltage parameter.

3.2 Introduction to integrity check in characterization framework

3.2.1 System crash / Data crash

In the world of computer systems and data management, the terms "application crash" and "system crash" are not widely accepted or established. However, if we read them broadly, we can distinguish between data loss and system failure.

A system crash is an unexpected and sudden failure of a computer system in which the operating system or vital software components become unresponsive or encounter severe faults, causing the system to malfunction. It happens when the system's usual operation is disrupted, resulting in a total or partial system failure. System crashes may express themselves in a variety of ways, including freezing, becoming unresponsive, displaying error messages, or requiring a system reboot. System crashes can happen for a variety of causes. System crashes can be caused by software-related causes such as flaws, programming problems, or conflicts between software components. Hardware concerns,

such as broken components, overheating, or power supply issues, can also cause crashes. Furthermore, system crashes can be caused by outside causes such as incompatible device drivers, insufficient system resources (such as memory or storage), or malware infestations that disrupt the system's normal operation. System crashes can occur at any moment, although they are most often when performing resource-intensive tasks such as operating complicated software applications, playing graphics-intensive games, or conducting significant computations. They can also happen on their own, with no obvious cause. System crashes can vary in frequency and severity based on the underlying causes and system setup. In general, system crashes are disruptive occurrences that necessitate thorough investigation and resolution in order to restore system functionality and maintain a reliable computing experience.

On the other hand, when the data on a computer system or storage media becomes corrupted, illegible, or unavailable, this is referred to as a data crash. This may happen for a variety of triggers, including hardware problems, software malfunctions, power surges, incorrect shutdowns, malware infections, cosmic radiations, alpha particles or voltage droops Σ. When a data crash occurs, vital files, documents, databases, or any other type of digital information might get lost or corrupted. Data losses, operational disruptions, financial consequences, and potential legal or regulatory difficulties can all result from data crashes. Data recovery from a crash can be a complex and difficult procedure that frequently requires the use of specialized data recovery techniques or the assistance of professional data recovery services. To reduce the danger of data crashes, it is critical to keep antivirus software up to date, use dependable hardware components, and adhere to safe computing practices. Utilizing data backup systems and following to data management best practices can help limit the risks associated with data failures while also ensuring the availability and integrity of vital information. Another mitigation mechanism for data corruption is ECC, error detection correction code in DRAMS and caches.

3.2.2 Running virus

3.2.3 Reference output in different running voltages

As shown in the sample, the output of the program consists of many data. Observing column 1, the possible outputs are PASS or FAIL, and for column 2, the output is the number of voltages in which Juno runs. As the program receives PASS in status, it leads to the voltage decreasing by 10 until it reaches a crash.

A reference output represents a result that is anticipated or expected to be produced from an input or test case. It acts as a benchmark for comparing the real output that the software under development generates, assisting in the detection and correction of any differences or faults.

In this version of the integrity check, we only check the integrity of registers. We run the virus under normal settings and generate a reference output. Then for each voltage level we test and compare the reference output with the current output. If differs then that means that we have an application crash.

3.3 Application / Usage of the Characterization Framework

3.3.1 System in idle state

I recorded power consumption measurements while changing the voltage manually. All measurements were taken without any program running at the background (idle state).

Dynamic Power Consumption formula:

Given by the proportionality: P=$C\ V$^2 Af

Capacitance (C): Capacitance largely depends on the wire lengths of on-chip structures (smaller wires, smaller Capacitance)

Supply voltage (V): Depends on the power-aware design. V has dropped steadily with each technology generation

Activity factor (A): The activity factor is a fraction between 0 and 1 that refers to how often wires actually transition from 0 to 1 or 1 to 0

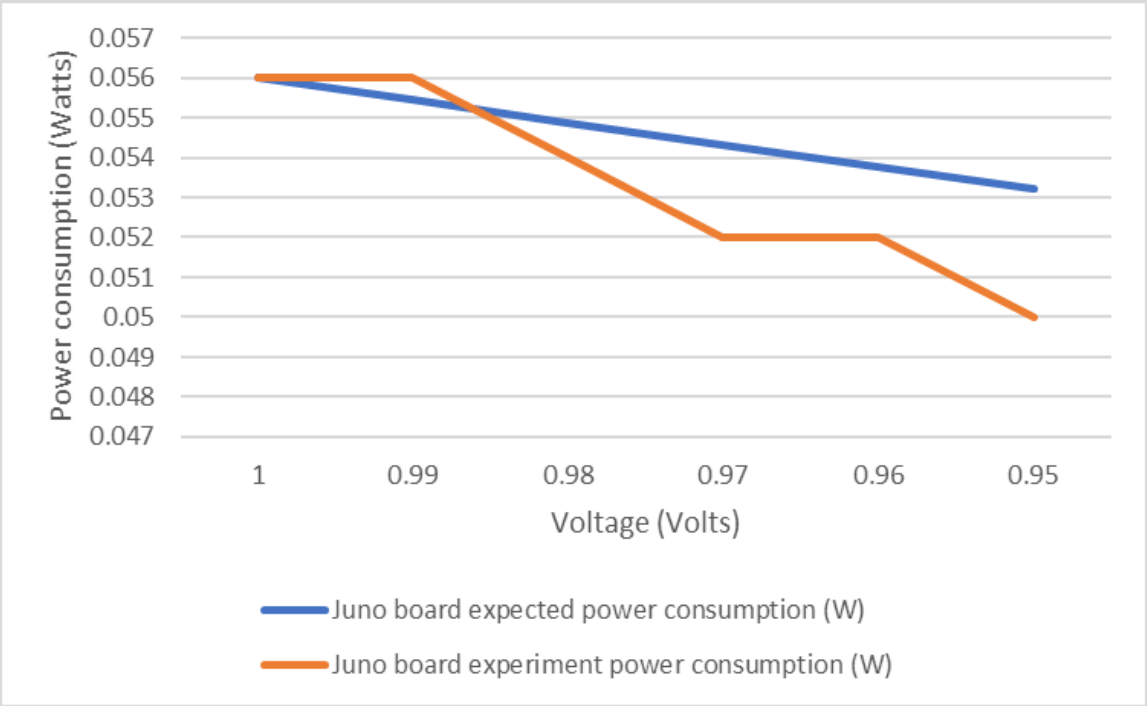
Clock Frequency (f): Maintaining higher clock frequencies may require maintaining a higher supply voltage.

Expected consumption calculation:

Because the system is idle, I only get static power consumption. The power consumption reduces linearly with voltage because of that.

Power consumption on Juno board when changing voltage.

Voltage(V)	1	0.99	0.98	0.97	0.96	0.95
Juno board expected power consumption(W)	0.056	0.05544	0.05488	0.05432	0.05376	0.0532
Juno board experiment power consumption (W)	0.056	0.056	0.054	0.052	0.052	0.05



Graph 1: Power consumption on Juno board when changing voltage

In conclusion, based on the experiment conducted in which the computer was running in an idle state, it can be observed that changes in voltage did not have a significant impact on power consumption. When a computer is idle, it operates at a lower power state, and the workload on the CPU is minimal. As a result, variations in voltage levels do not lead to substantial changes in power consumption.

3.3.2 Vmin for di/dt virus

I executed characterisation framework which uses the dI/dt virus.

Results:

PASS	920	107_5330	EXEC_TIMI	30.47	WORKLOAD	[0]	DOC	63bfdbe895f0367ff745582c
PASS	920	107_5330	EXEC_TIMI	30.45	WORKLOAD	[3]	DOC	63bfdbe895f0367ff745582c
PASS	920	107_5330	EXEC_TIMI	30.46	WORKLOAD	[4]	DOC	63bfdbe895f0367ff745582c
PASS	920	107_5330	EXEC_TIMI	30.45	WORKLOAD	[5]	DOC	63bfdbe895f0367ff745582c
PASS	910	107_5330	EXEC_TIMI	30.47	WORKLOAD	[0]	DOC	63bfdc2295f0367ff745582d
PASS	910	107_5330	EXEC_TIMI	30.45	WORKLOAD	[3]	DOC	63bfdc2295f0367ff745582d
PASS	910	107_5330	EXEC_TIMI	30.45	WORKLOAD	[4]	DOC	63bfdc2295f0367ff745582d
PASS	910	107_5330	EXEC_TIMI	30.45	WORKLOAD	[5]	DOC	63bfdc2295f0367ff745582d
PASS	900	107_5330	EXEC_TIMI	30.46	WORKLOAD	[0]	DOC	63bfdc5b95f0367ff745582e
PASS	900	107_5330	EXEC_TIMI	30.46	WORKLOAD	[3]	DOC	63bfdc5b95f0367ff745582e
PASS	900	107_5330	EXEC_TIMI	30.45	WORKLOAD	[4]	DOC	63bfdc5b95f0367ff745582e
PASS	900	107_5330	EXEC_TIMI	30.45	WORKLOAD	[5]	DOC	63bfdc5b95f0367ff745582e
SYSTEM_CRASH	890			10				

1st iteration: System crash in 0.890W

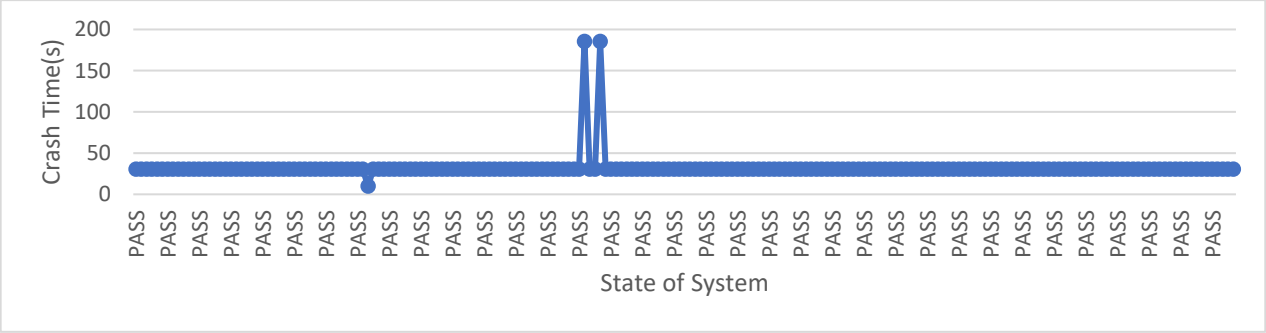
PASS	920	107_5330	EXEC_TIMI	30.45	WORKLOAD	[4]	DOC	63bfdedd95f0367ff7455838
PASS	920	107_5330	EXEC_TIMI	30.45	WORKLOAD	[5]	DOC	63bfdedd95f0367ff7455838
PASS	910	107_5330	EXEC_TIMI	30.46	WORKLOAD	[0]	DOC	63bfdf1795f0367ff7455839
PASS	910	107_5330	EXEC_TIMI	30.45	WORKLOAD	[3]	DOC	63bfdf1795f0367ff7455839
PASS	910	107_5330	EXEC_TIMI	30.46	WORKLOAD	[4]	DOC	63bfdf1795f0367ff7455839
PASS	910	107_5330	EXEC_TIMI	30.45	WORKLOAD	[5]	DOC	63bfdf1795f0367ff7455839
APP_CRASH	900	107_5330	EXEC_TIMI	185.63	WORKLOAD	[0]	DOC	63bfdff095f0367ff745583a
PASS	900	107_5330	EXEC_TIMI	30.46	WORKLOAD	[3]	DOC	63bfdff095f0367ff745583a
PASS	900	107_5330	EXEC_TIMI	30.45	WORKLOAD	[4]	DOC	63bfdff095f0367ff745583a
APP_CRASH	900	107_5330	EXEC_TIMI	185.56	WORKLOAD	[5]	DOC	63bfdff095f0367ff745583a

2nd iteration: App crash in 0.900W

PASS	930	107_5330	EXEC_TIMI	30.46	WORKLOAD	[5]	DOC	63bfe6e695f0367ff7455856
PASS	920	107_5330	EXEC_TIMI	30.46	WORKLOAD	[0]	DOC	63bfe72a95f0367ff7455857
PASS	920	107_5330	EXEC_TIMI	30.47	WORKLOAD	[3]	DOC	63bfe72a95f0367ff7455857
PASS	920	107_5330	EXEC_TIMI	30.45	WORKLOAD	[4]	DOC	63bfe72a95f0367ff7455857
PASS	920	107_5330	EXEC_TIMI	30.45	WORKLOAD	[5]	DOC	63bfe72a95f0367ff7455857
PASS	910	107_5330	EXEC_TIMI	30.46	WORKLOAD	[0]	DOC	63bfe76395f0367ff7455858
PASS	910	107_5330	EXEC_TIMI	30.47	WORKLOAD	[3]	DOC	63bfe76395f0367ff7455858
PASS	910	107_5330	EXEC_TIMI	30.45	WORKLOAD	[4]	DOC	63bfe76395f0367ff7455858
PASS	910	107_5330	EXEC_TIMI	30.45	WORKLOAD	[5]	DOC	63bfe76395f0367ff7455858

3rd, 4th and 5th iteration: After 910 voltage the iteration finishes, since the program knows that the next one is the higher voltage that was found in the current experiment.

Time VS State of the System:



Graph 2: Crash time graph

From the collected results, I can understand that for every passing state the program lasted around 30.45 secs. The SYSTEM_CRASH was found early and caused the program to continue to the next iteration since it appeared in the first 10 secs. On the other hand, APP_CRASH caused the system a delay since it wasn't responding until it understand that a crash was issued. To be able to continue, the process needs to be manually killed from the console.

An interesting observation is that application crashes (bit flips), occur at higher voltage than system crashes which suggest that data corruption has higher Vmin. The virus is stacked in an infinite loop, that's why it took longer. This might might be a result of a bit flip.

Chapter 4

Integrity Check

4.1 Introduction

Integrity checks in PC applications refer to mechanisms or processes that verify the integrity or correctness of data or files. These checks are typically performed to ensure that the data or files have not been tampered with, corrupted, or modified unintentionally. To achieve this, I created an integrity check program that is held on the Juno board[3].

Integrity checks involve comparing the current state of data or files with a known reference or expected state. This comparison can be done using various methods, such as checksums, hash functions, digital signatures, or other algorithms. In the current implementation, to ensure that the data has not been changed, the comparison is happening in a linked list data structure.

During the integrity check, the application compares the initial values with the final values. If the values match, it indicates that the data has not been altered. However, if there is a mismatch, it suggests that the data have been modified or corrupted.

4.1.1 How does it expand current integrity check

The thought between the creation of an integrity checking program is to expand the current integrity check that is minimal. As noted, the framework only checks and reports crashes that occur and divides them into two types, system crashes and app crashes. Those two categories alone are not sufficient to convince us that there is indeed no other crash that remains undetected, thus the need for a better integrity check.

The expansion has a goal to check whether any data crashes occur in the memory. To do this, the memory needs to be filled with values that are known to the program, and after each run of the virus those values need to be unchanged. With this method the memory is checked for bit flips, that previously may be undetected, and add another layer of control. To be more specific, since TAGS, TLBs, DATA and CACHES are being manifested in the memory, that means that a bit flip can be catch in any of them.

4.2 Integrity check

4.2.1 Input / Output

The integrity check is implemented in the file LinkedList.cpp that is under the Juno board[3] directory. The program to start the integrity check, it needs to be called by a signal that the characterization framework sends. Thus, the only receiving input is that. The other modification that can happen manually is the adjustment of the size of the linked list (constant N).

As an output, the program returns to the framework a message after every call, with the results of the integrity check regarding possible bit flips, in the form of a string. If a bit flip occurs, the program returns the message “Integrity Check Complete” alongside the identification message “FAIL” and if not, the identification message is not sent. Also, as the program is divided into two main functions, the initialization of the linked list and the integrity check, a message notifying that the initialization is complete is also sent (message: “Initialized Complete”). The framework is using every message accordingly to create the final report.

4.2.2 Data Structures

The data structure that is used in the integrity check program is called Node and it represents the linked list. The structure has a pointer on another node (Node* next), that achieves the connection between the elements of the linked list, and another variable (uint64_t data) that saves the value of each node.

4.2.3 Functionalities

Integrity check has only some basic functions. CreateList() that is used to create the Linked list, lfsr_init() that creates the random values that are included in the linked list, the lfsr_check() that checks for bit flips and the function printList() that is used for debugging reasons to print the values of the linked list.

CreateList() function

```
Node *createList()
{
  Node *head = nullptr;
  Node *current = nullptr;
  for (int i = 0; i < N; i++)
  {
    Node *newNode = new Node();
    newNode->data = lfsr_init();
    newNode->next = nullptr;
    if (head == nullptr)
    {
      head = newNode;
      current = newNode;
    }
```

```
  else
  {
    current->next = newNode;
    current = newNode;
  }
 }
 return head;}
```

This function is called to create the linked list, and with the help of the lfsr_init() function, fill it with certain values.

Lfsr_init() function

```
uint64_t lfsr_init()
{
 static uint64_t x = 1;
 uint64_t bit = ((x >> 0) ^ (x >> 1) ^ (x >> 3) ^ (x >> 4) ^ (x >> 64)) & 1;
 x = (x << 1) | bit;
}
```

This function creates the values of the linked list using linear-feedback SHIFT register (LFSR) algorithm. It returns a random number between 0 and 2^64-1 (inclusive) (See section [4.1] for LFSR).

Lfsr_check() function

```
bool lfsr_check(Node *head)
{
 Node *current = head;
```

```
 uint64_t bit;


 while (current != nullptr)
 {
  static uint64_t x=1;
  bit = ((x >> 0) ^ (x >> 1) ^ (x >> 3) ^ (x >> 4) ^ (x >> 64)) & 1;
  x = (x << 1) | bit;


  if (current->data != x)
    return true;


  current = current->next;
 }
 return false;
}
```

This function is used to check the linked list values for bit flips. It receives as a parameter every node of the linked list, and if there is a difference in a value, let's say the value of the node is 8 and the LFSR value is a different number, it means that a bit flip was discovered. In that case the function returns true, otherwise if there is no bit flip in the whole linked list, the function returns false.

PrintList() function

```
void printList(Node *head)
{
 Node *current = head;
```

```
  while (current != nullptr)
  {
    cout << current->data << " ";
    current = current->next;
  }
  cout << endl;
}
```

This function prints the values of the whole linked list.

4.2.3.1 LFSR

As mentioned above, the linked list values are calculated with the LFSR algorithm. The LFSR algorithm is a technique used to generate pseudorandom sequences or streams of bits. It involves using a shift register, which is a sequence of flip-flops or registers, to generate a new bit based on the current state of the register. The sequence of bits generated by the LFSR algorithm depends on the initial seed value and the configuration of the feedback function.

LFSRs can produce sequences with long periods and good statistical properties, making them useful in pseudorandom number generation. For this project I intended to create random numbers to fill the linked list. This will help to ensure that the generated sequence is diverse and free of repeating patterns. This way there is a reduced chance to have same amount of 1s and 0s bits in sequential numbers. The sequence cycle depends on the seed that I choose for the implementation.

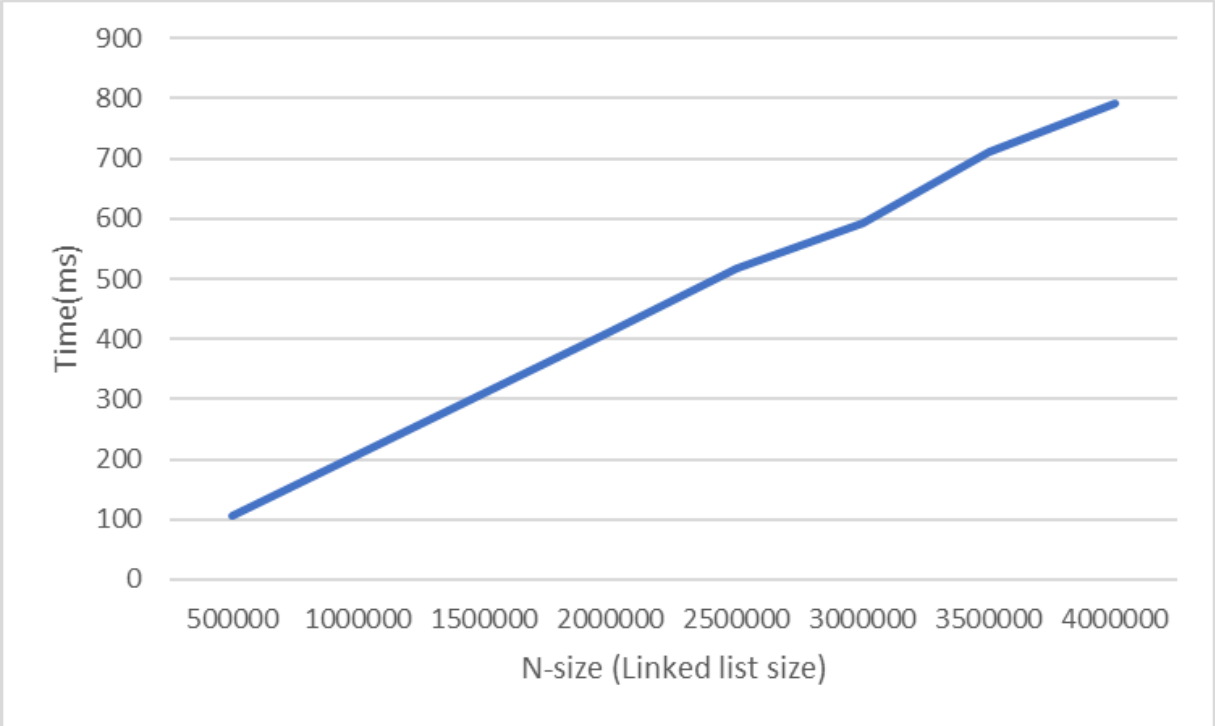
LFSR is deterministic. It will always generate the same sequence of instructions for the same seed. This is the reason I used it because I do not have to save anywhere the reference output.

4.3 Need of memory overload

In theory, the linked list must contain the appropriate number of elements to fill Juno's DRAM. That is needed because only if the memory is full of known values, the check can be valid, otherwise a bitflip may occur on a part of the memory that is not monitored, and lead to an undetected problem. Juno contains 2GB of DRAM. The bit flip is actually happening in the CPU registers and it is propagated to DRAM.

At first glance, I thought that this size would lead to a long time delay due to the fact that the initialization of the linked list happens at each iteration of the framework (at each voltage drop).

Using some N-size examples I calculated that the time needed for initialization is proportional to the N-size. I showed that approximately every 500 000 numbers, it takes 100ms.



Graph 3: Initialization time of the linked list in different linked list size values

Using the above, I managed to validate the size of the linked list. 2GB of DRAM is equivalent to 2,147,483,648 bytes, and each number I add to the Linked list (since it is of type uint64_t) takes up 8 bytes of space.

So theoretically, I would need a linked list size of around 2,147,483,648 / 8 = 268,435,456 (THEORETICAL N-size).

Based on the calculation in the previous slide, it will take 268,435,456 / 500 000 = 53000ms (53 seconds) for each initialization, which is not a large number by the standards of the framework.

After I calculated the theoretical time and saw that it satisfied me, I ran the program with the appropriate size and saw that indeed the time the initialization takes is about what I calculated.

For N=255 million (because the theoretical calculation did not take into account the size of the Linked List), it takes 50615ms, so 50,615 seconds.

Chapter 5

Merching characterization framework with integrity check

5.1 Control Flow

5.1.1 New Input Parameters

Changes were made to the framework in order to serve the purpose of the project, and some of them were related to the insertion of new input parameters.

The extra integrity check layer needed to be optional, meaning that the framework was meant to be running either with or without the new integrity check. To do this, I used a parameter that was working as a Boolean value named measureIntegrity. If the value of it is set to true, the framework is using the integrity check, and if the value is false, the framework is running without it.

Since the requirements of the program changed, I needed to ensure that the framework and the integrity check were running properly. To do this I inserted a parameter named integrityPID. This parameter keeps track of the ID of the current process. Then, whenever a problem occurs in an experiment, I have the option to manually kill it.

The characterization framework needed to have a way to call the integrity check to start working. To do this, the integrity check script is called through the serial port commands. So, the path of the script is saved on the integrityCheckScript parameter and whenever is needed, the calling command is formed and called with the self.serial.sendCMD command.

Another factor I took into consideration is the voltage where the initialization and the integrity check operations are happening. To be able to modify it, it gives user the option to pick whatever value he wants for the nominal voltage. That can happen in the asioko01junoA53_virus.json which holds the workload, by changing the parameter userVoltage.

5.1.2 Main memory initialization and integrity check in nominal voltage

As stated above, the characterization framework is running a Vmin test and the Vmin test at the end of each of its steps, force the running voltage of the Juno board to drop. Since the voltage is dropping, there is no guarantee that the voltage drop does not affect the properties of the integrity check. To make sure that none of the bit flips occur for that reason, the initialization of the main memory and the integrity check are happening at nominal voltage (which is set by the user).

5.2 Signals

The characterization framework and the integrity check are not under the same directory. In fact, the framework is located in the host pc files and the integrity check is located in the Juno board files. Because of that, I needed to find a way for those two to interact.

The most important thing to understand is when each one of them is operating. The control flow goes as follows:

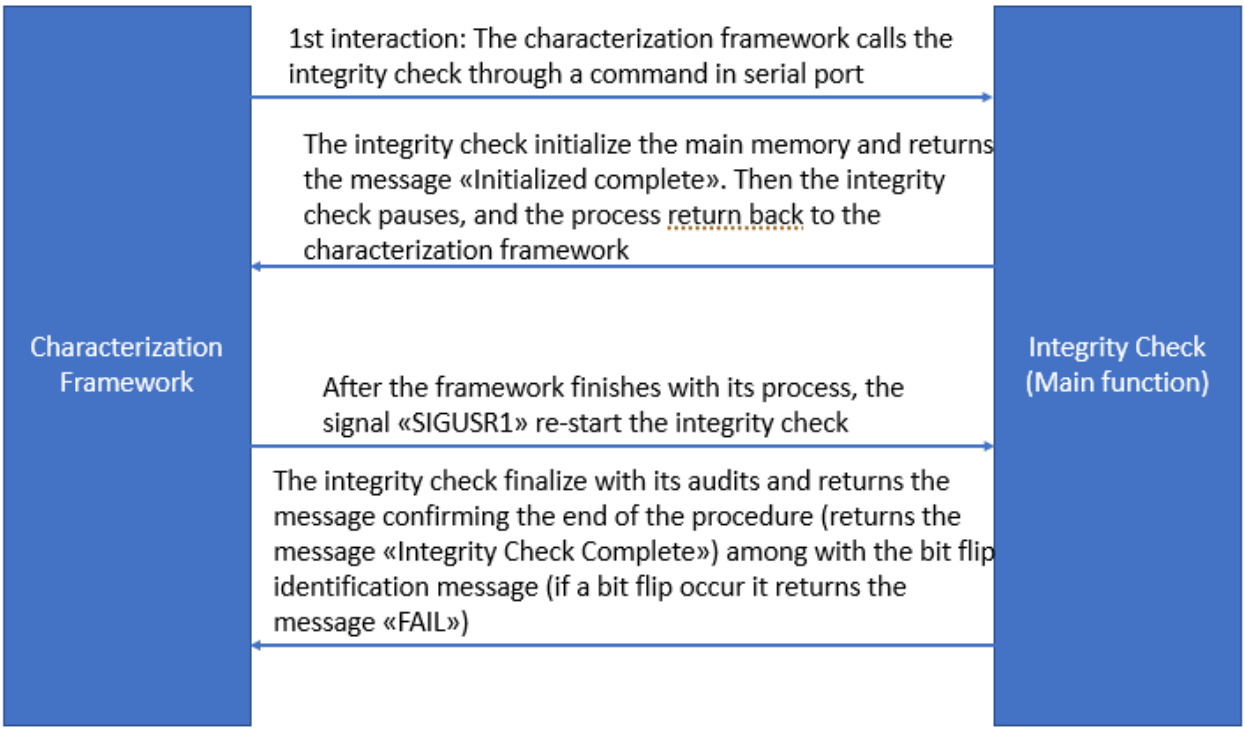


Figure 1: Connection between host and target

As mentioned in Figure 1, there are two different ways for the program to interact. The first one is throughout the serial commands. The characterization framework has the ability to call the integrity check through the serial port and more specifically with the commands:

```
integrity_check_command = self.integrityCheckScript + " &"

code = self.serial.sendCMD(integrity_check_command, waitTime=WorkloadHandler.UNRESPONSIVE_TIMEOUT);
```

In the integrityCheckScript parameter, the path of the file is located: "integrityCheckScript":"/media/oldDisk1/root/integrity_check_scripts/LinkedList"

In the second situation, the framework has to “wake up” the integrity check, which is paused since the initialization finished. To do this the following signal is sent:

```
integrity_check_command_kill = "pkill -SIGUSR1 LinkedList"
code = self.serial.sendCMD(integrity_check_command_kill, waitTime= WorkloadHandler.
UNRESPONSIVE_ TIMEOUT);
```

The signal "-SIGUSR1" is responsible to re-start the integrity check.

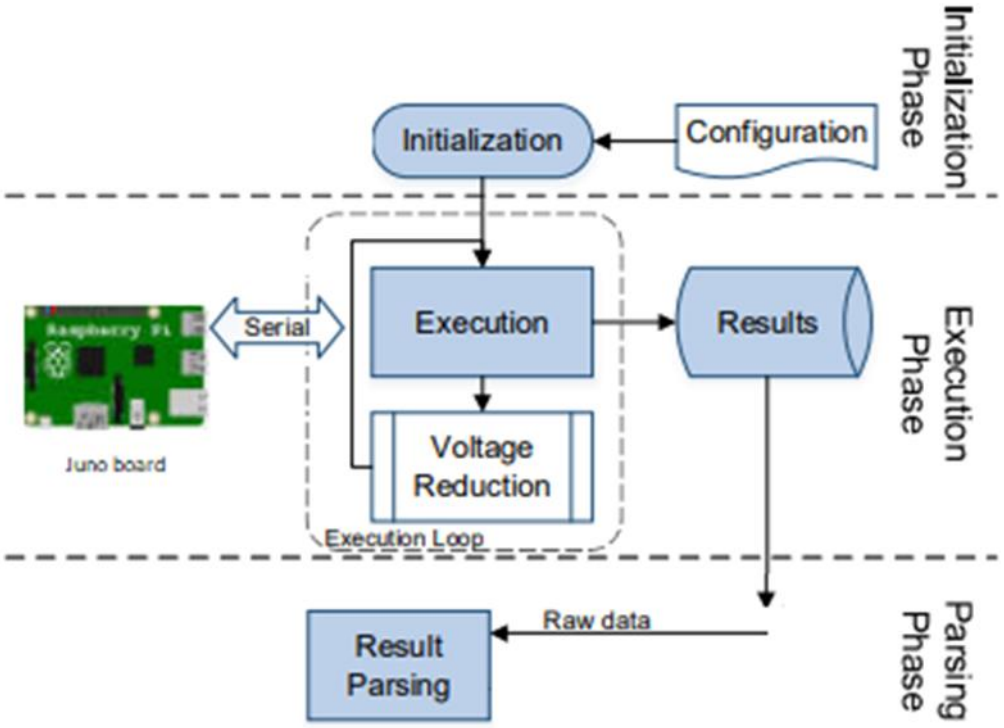


Figure 2: Control flow

Chapter 6

Methodology of Experiments

6.1 Methodology

The experiments took place on one of the two CPUs of the Juno board, cortex A53. To be able to run the framework on the Juno board, a connection needed to be established between it and the host pc. The Vmin test operation is running at the same time as a running virus. In the current project the di/dt virus that is used is the 107_5330_26s, that was created with GeST[5].

6.1.1 Machine characteristics

The ARM Juno board is a platform for development made by ARM, a well-known manufacturer of semiconductors and software. The 64-bit Cortex-A CPUs used in ARM-based systems, are the focus of the Juno board's architecture.

The Juno board offers a complete development and testing environment for ARM-targeted software and hardware solutions. It features a system-on-chip (SoC) design with multiple ARM Cortex-A processors, including big.LITTLE architecture, which combines high-performance cores with power-efficient cores for optimal performance and energy efficiency.

Juno board configurations	
Hardware	• Arm Cortex-A72 MPCore (Juno r2) • Arm Cortex-A53 • Arm big.LITTLE technology • Arm Mali graphics processor for 3D Graphics acceleration and GP-GPU compute • 4-lane Gen 2.0 PCI-Express • A SoC architecture aligned with Level 1 (Server) Base System Architecture
Software	• System initialisation, cold boot flow and controls clocks, voltage, power gating. • Delivered as binary through Linaro with public programmers interface • Application Processor Software – all delivered as source through Linaro • Arm Trusted Firmware – supporting PSCI power controls and trusted execution environments • Choice of UEFI or U-Boot firmware • Linux – support for both latest kernel and Linaro Stable Kernel which includes Mali GPU drivers and Android patch set

Table 1: Juno board configurations

As mentioned above the software is running on the cortex A53 processor.

Cortex A53 configurations	
Architecture	Armv8-A
Multicore	1-4x Symmetrical Multiprocessing (SMP) within a single processor cluster, and multiple coherent SMP processor clusters through AMBA 4 technology

Characteristics	• The Armv8-A architecture provides 64-bit data processing, extended virtual addressing, and a 64-bit general purpose registers. • The first Armv8-A processor aimed at providing power-efficient 64-bit processing. • In-order, 8-stage, dual-issue pipeline • Improved integer, Neon, Floating-Point Unit (FPU), and memory performance. • The Cortex-A53 can be implemented in two execution states: AArch32 and AArch64. The AArch64 state allows execution of 64-bit applications. The AArch32 state allows execution of existing Armv7-A applications.

Table 2: Cortex A53 configurations

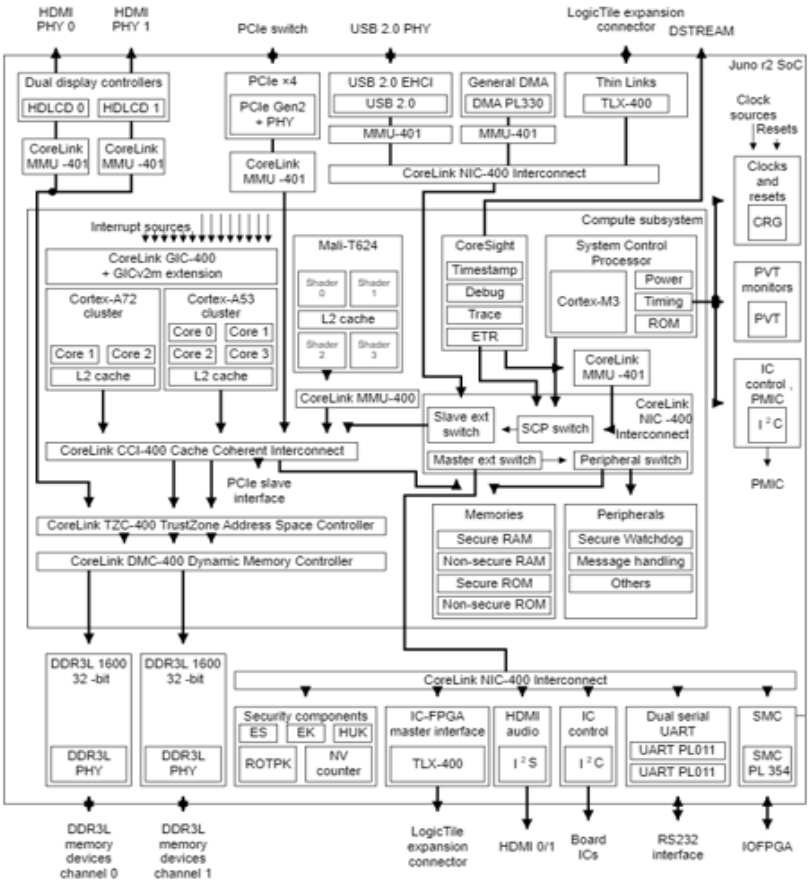


Figure 3: Juno board components

6.1.2 Setup

For the correct operation of the experiments, the connection and communication between the host pc and the board is essential. The framework software is located on the host, while the integrity check software is located on the Juno. As the experiments are initialized by the framework and then communicated to the board in order to be executed, it becomes clear how important the communication between the two is.

Figure 4: Host pc

Figure 5: Target - Juno board

The Juno board is connected to the host pc with serial connection and can be accessed through the Putty client (COM9: DOS Firmware Commands - cfg command, COM8: Linaro Distribution (Linux)).

In order to read/set power, frequency and voltage, what has to be done is the connection to the serial port with DOS environment (PORT9) and then enter debug mode.

6.1.3 Workloads

The workload is defined in a Json file named asioko01junoA53_virus.json is used that includes all the parameters that need to be initialized.

The workload that is used for the experiments of the project is the following:


```
    "targetHostName":"10.16.20.171",
    "targetSSHusername":"
    "targetSSHpassword":"

"sysLogParsingScript":"/media/oldDisk1/root/chf_scripts/printErrorsWithinTimestamps
",
    "helperScriptSetup":"/media/oldDisk1/root/chf_scripts/chf_helper_freq_taskset.sh",
    "helperScriptWorkload":"/media/oldDisk1/root/chf_scripts/chf_helper_workload.sh",

"workloadStatusOutput":"/media/oldDisk1/root/chf_scripts/chf_helper_tmp/workload_s
tatus",
    "integrityCheckScript":"/media/oldDisk1/root/integrity_check_scripts/LinkedList",
    "userVoltage": 1000,

    "measurePowerScript": null,
    "serialPort":"COM8",
    "mongoDB":"juno",
    "mongoCol":"asioko01a53_virus_Save",
```

```
  "experiments":[
    {
      "workloads":[
        {
          "work_dir":".",
"cmd_line":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/107_5330_26s",
          "toKillName":"107_5330_26s",
          "conventionName":"107_5330_26s",
          "cores":[0],
"originalOutput":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/ref.out",
          "type":"singleThread",
          "runOutput":"/media/oldDisk1/root/chf_scripts/chf_helper_tmp/0_out",
          "inputs":null,
          "VM":false,
          "stdin":null
        },
        {
          "work_dir":".",
"cmd_line":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/107_5330_26s",
          "toKillName":"107_5330_26s",
          "conventionName":"107_5330_26s",
          "cores":[3],
```

```
"originalOutput":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/ref.out",
                "type":"singleThread",
                "runOutput":"/media/oldDisk1/root/chf_scripts/chf_helper_tmp/1_out",
                "inputs":null,
                "VM":false,
                "stdin":null
            } ,

            {
                "work_dir":".",

"cmd_line":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/107_5330_26s",
                "toKillName":"107_5330_26s",
                "conventionName":"107_5330_26s",
                "cores":[4],

"originalOutput":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/ref.out",
                "type":"singleThread",
                "runOutput":"/media/oldDisk1/root/chf_scripts/chf_helper_tmp/2_out",
                "inputs":null,
                "VM":false,
                "stdin":null
            },

            {
              "work_dir":".",
```

```
"cmd_line":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/107_5330_26s",
                "toKillName":"107_5330_26s",
                "conventionName":"107_5330_26s",
                "cores":[5],

"originalOutput":"/media/oldDisk1/root/benchmarks/a53emVirusGood26s/ref.out",
                "type":"singleThread",
                "runOutput":"/media/oldDisk1/root/chf_scripts/chf_helper_tmp/3_out",
                "inputs":null,
                "VM":false,
                "stdin":null
            }
        ],
        "platform":"juno_a53",
        "core_to_freq":[950,0,0,950,950,950],
        "repetitions":1,
        "run_type":"all",
        "start_voltage":1000,
        "end_voltage":0,
        "vol_inc":10,
        "inc_wait_time":10,
        "vmin_component":"CORE"
    }
  ]
}
```

Explaining the workload from top to bottom, firstly to be able to connect to the Juno board, the IP along with the username, password and the serial port are essential to establish a connection.

The second most important thing is to initialize the parameters. In this case the path to the integrity check (integrityCheckScript), the nominal checks voltage (userVoltage) and the output file name (mongoCol).

The actual workload needs to be set up for all four cores of the cortex A53. To do this, all the specifications are applied for every single core (0,3,4,5). In the end, the nominal frequency of the A53 is set (950MHz) for every core, the number of repetitions that the experiment will execute, the start voltage and the end voltage and how much voltage to drop after every repetition.

Chapter 7

Results

7.1 Results

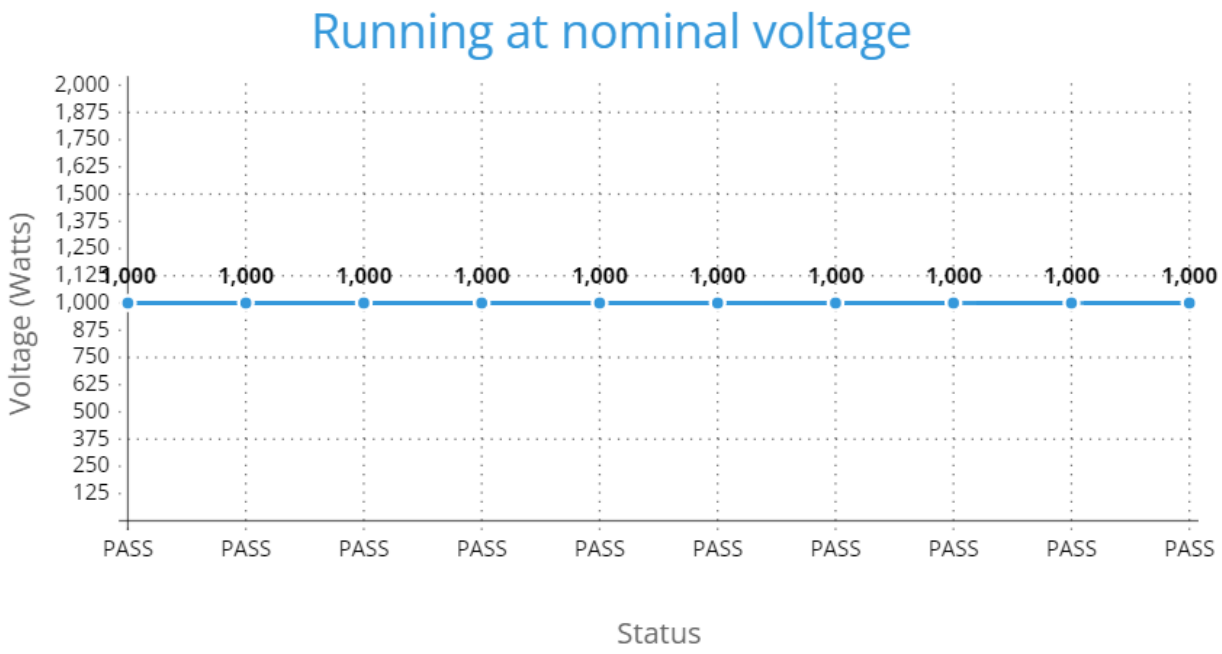
Every experiment consists of many runs, and in each run, there are many voltage steps. The reporting – the result of every run is reported as follows:

- PASS: If no crash is detected, the run continues normally.
- SYSTEM_CRASH: The run ends abnormally, and the Juno board needs to reboot to continue to the next run.
- APPLICATION_CRASH: The integrity check detects a crash due to different possible reasons (software bug, memory conflict, hardware problem). To be able to continue the run to the next voltage step, the user must manually kill the virus process.
- BIT_FLIP: The new integrity check finds different values in main memory than the initial. Again, to be able to continue the run to the next voltage step, the user must manually kill the virus process.

After Executing the Experiments stated in the Methodology chapter, I collected the following data. As predicted bit flips occur in the CPU and get propagated to DRAM during the executions, and they are identified by the implementation.

7.1.1 Validations

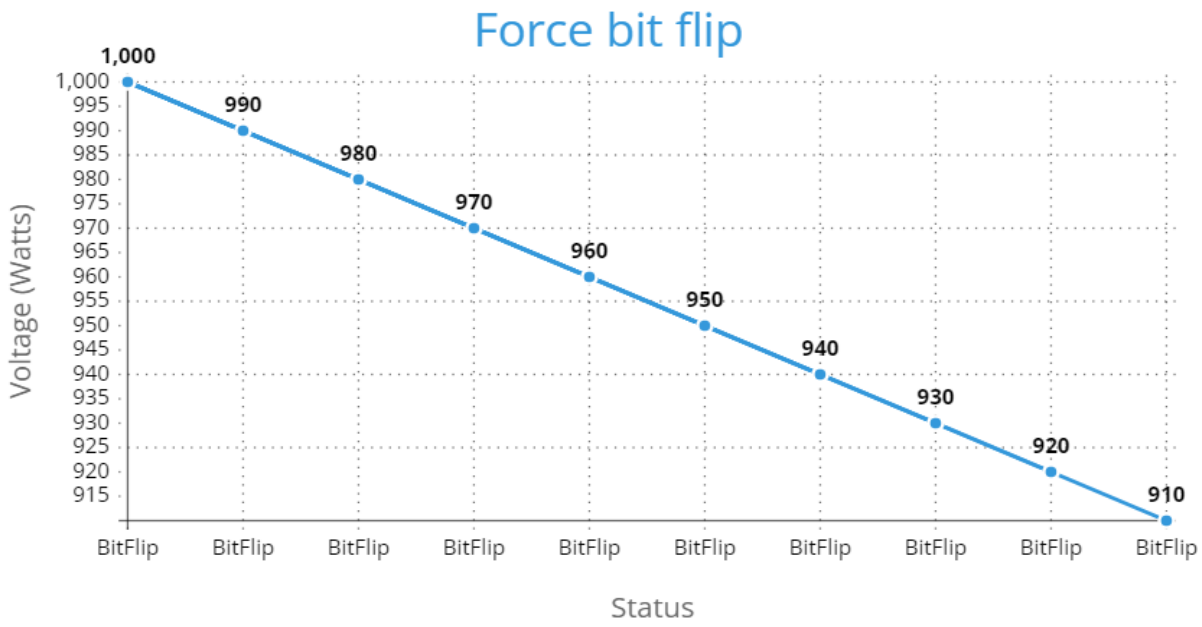
To be able to validate that the experiments are correct, I made some compulsory checks. First of all, I had to check that the characterization framework was running correctly at nominal voltage, meaning that I am expecting no crashes or bit flips while the voltage is still at nominal value.



Graph 4: Nominal voltage experiment

According to the results of the experiment shown in graph 4, when running at nominal voltage, the status of the execution running is PASS for every core that is used (0,3,4,5). That means that without the voltage dropping, no crashes or bit flips appear confirming that the experiment is not affected by other factors.

Then, I had to check that my implementation is doing what it is supposed to do, and by that meaning that it must have the ability to identify bit flips when they occur. To be able to check if this procedure is executed correctly, I forced the program to create cases that bit flips appears 100% at the end of every repetition. To be able to do this, I changed the implementation of the function lfsr_check() in the integrity check, and manually swapped the values of the compared values so that they are different from each other.



Graph 5: Force bit flips experiment

The results in Graph 5, show the results of the above experiment for one repetition. As I am swapping the values, the framework can identify the bit flip and then report it. By that, I can ensure that the implementation is correct.

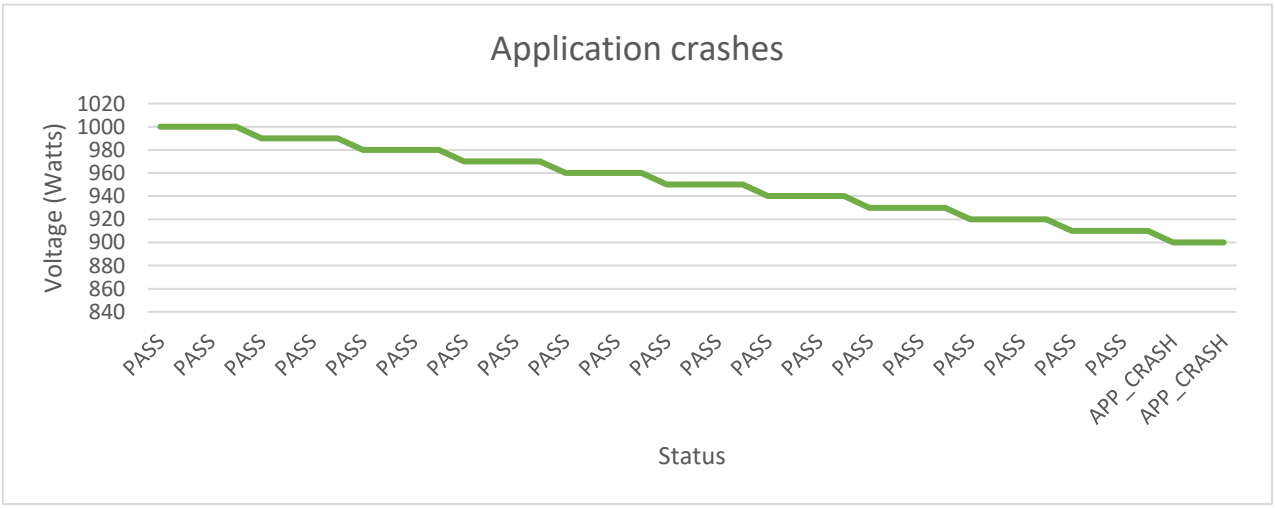
7.1.2 Graphs Description – Results without integrity check

To be able to compare the results between the previous implementation of the characterization framework and the new one with the integrity check, I needed to run some experiments in both versions. To begin with, the following graph's data are taken from experiments without the integrity check.

Graph 6: Experiments without integrity check

The experiments' results of Graph 6 are taken from 10 repetitions of the workload. Results without the integrity check, after running the framework for a few times, can be predictable. As you can observe, the framework tends to crash at 0.890 Watts, as every repetition stops the voltage drop at that point meaning that the system cannot work under that voltage barrier as every crash is a system crash.

In certain, but few situations, the type of crashes was not system, but application. An application crash occurs when a software program unexpectedly terminates or stops functioning correctly. It can be caused by various factors, including software bugs, memory issues, conflicts with other programs or system components, insufficient resources, or external factors such as hardware failures. Since the implementation of the extra integrity check was not included, the appearance of application crashes reinforces the idea that there may be crashes related to the memory that are not being properly monitored.

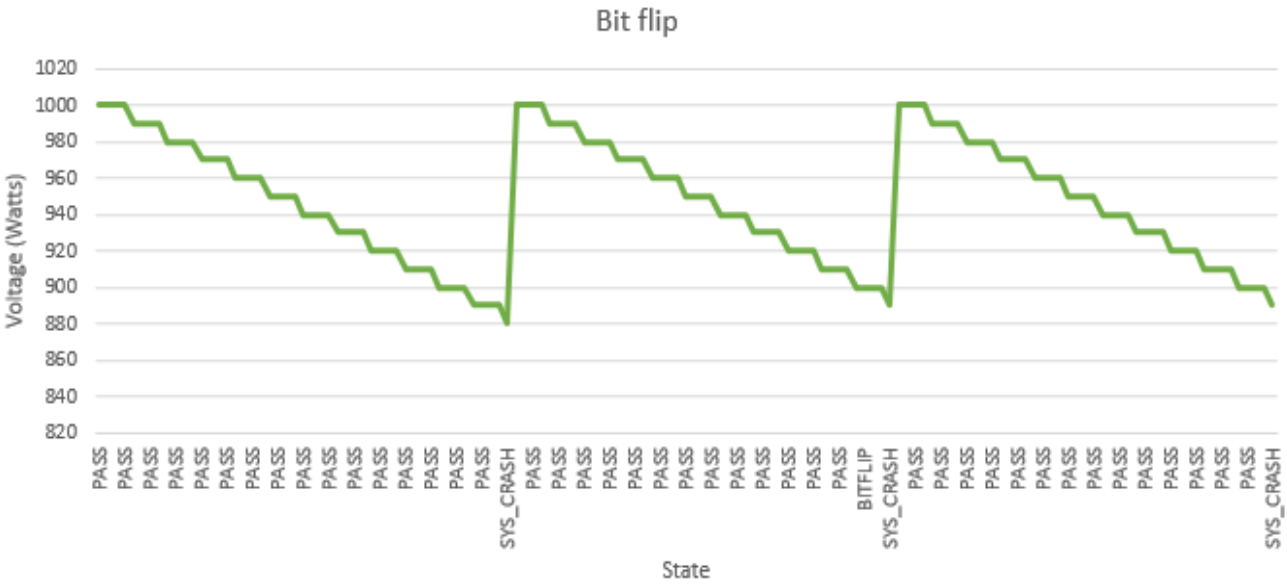


Graph 7: Application crashes repetition

As you can see in Graph 7, in that certain repetition the crashes were application crashes and occurred in two different cores (core 0 and core 4). Whenever the crash was an application crash the flow of the framework stacked at the WORKLOAD_HELPER_SCRIPT and to continue I had to manually kill the current running process manually.

7.1.3 Graphs Description – Results with integrity check

The goal of the extra layer of integrity check was to be able to identify and monitor any data array corruptions in main memory. As I mentioned, the appearance of application crashes gave me the hope that maybe bit flips can occur and get detected.



Graph 8: Bit flip occurrence

As you can see in Graph 8, a bit flip was identified after the second repetition of the program. The bitflip was found before the system crash, but at low voltage in relation to where the voltage drop starts, leading to the belief that maybe the voltage drop can be considered as one of the possible causes of the bit flip. As with application crashes, whenever the crash was a bit flip the flow of the framework stacked at the WORKLOAD_HELPER_SCRIPT and to continue I had to manually kill the current running process manually.

The voltage at which bit flip occurs is higher than the voltage of system crash, leading to the conclusion that Vmin is higher when taking into consideration silent data corruption.

7.1.4 Comparison

In comparison, there are some differences between the characterization framework with and without the extra integrity check.

First, the main difference is that now the application crashes can be identified as data corruption main memory. Since bit flips were identified, we could see the purpose behind the project.

Characterization framework with the integrity check takes much more execution time than without it, and that is because the initialization of the memory of the Juno boards takes some time. Combine the time needed with the multiple iterations that occur, those add enough extra simulation time.

Finally, there was a marked difference in the frequency of bit flip occurrences while running the framework with the new integrity check, compared to the frequency of application crash occurrences that appeared without the new version of the integrity check.

	Experiments without the new integrity check	Experiments with the new integrity check
Experiments	30	15
Runs	300	150
Amount of application crashes	7	1
Amount of bit flips	-	10
Amount of system crashes	23	14

Table 3: Crash frequency

With the new integrity check, in fewer runs, more application crashes + bit flips happen. Also, as you can see in table 3, whenever a bit flip occurs, a system crash follows.

Chapter 8

Future Work 51

As mentioned many times in this paper, this project is specified on data arrays corruption during Vmin tests. For future work it could be really interesting to expand the integrity check either by searching for other types of errors, or by expanding the current version to be able to be more specific. Maybe it can categorize the kind of memory that the bit flip occurs in, or maybe it can provide more details according to how, in what value etc. the bit flip was found.

Also, to expand the current version to be able to accept more parameters, simple changes can be made to the input file of the framework as well as the executor, to be able to receive extra information the user wants that can be helpful for him.

Another thing that can maybe be improved is the methodology used by creating multithreaded integrity check that checks the caches of each core.

Finally, changes can be made regarding the replacement policy. Linked list can be accessed in a manner so that the data are first visited in the caches before fetching something from main memory.

Chapter 9

Conclusion

9.1 Conclusions

Silent data corruption can occur in a Vmin test. I developed a methodology that can identify some of them. To be able to verify that indeed the crashes were happening because of bit flips in data arrays, many preconditions had to be respected, from characterization framework preconditions to integrity check and memory preconditions. The main memory had to be overloaded, so it could be monitored by the integrity check and to do that a proper data structure should have been chosen to fill that memory. The usage of LFSR was also important because it gave the program the randomness it needed.

9.2 Lessons Learned

In the end of this thesis, I have to say that I've learned a lot about this topic and I believe that I managed to undergo many problems that motivated me in the belief that there is nothing too difficult until you believe that you can do it. The biggest problem that I've faced was when I could not find a way to interconnect the framework with the integrity check on Juno board. After trying many different approaches that I read about, in the end the correct way to do it was way simpler than any other way I've tried before. From then and on, I try to use simpler solutions that I come up with first and don't give up after a failure.

Bibliography

[1] K. Tovletoglou et al., "Measuring and Exploiting Guardbands of Server-Grade ARMv8 CPU Cores and DRAMs," 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Luxembourg, Luxembourg, 2018, pp. 6-9, doi: 10.1109/DSN-W.2018.00013

[2] ARM ltd., Armv8-A Instruction Set Architecture, 2019-2020

[3] ARM ltd., 64 Bit Juno r2 ARM Development Platform, 2015

[4] Waylon Jepsen, Cyclic Redundancy Checks and Error Detection

[5] Zacharias Hadjilambrou, Shidhartha Das, Paul N Whatmough, David Bull, Yiannakis Sazeides, GeST: An Automatic Framework for Generating CPU Stress-Tests

[6] Zacharias Hadjilambrou, Harnessing CPU Electromagnetic Emanations for Power-Delivery Network Characterization, November,2019

[7] Software framework: https://github.com/asioko01/IntegrityCheck_IN_CharacterizationFramework.git