POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

On the analysis of DRAM faults on ANN applications

Supervisors

Prof. Ernesto SANCHEZ

PhD Annachiara RUOSPO

Prof. Alberto BOSIO

Candidate

Mattia DI IORIO

Academic year 2020/2021

Alla mia famiglia,
a Federica,
a tutti coloro che mi sono stati vicino.
Un particolare pensiero lo dedico alla mia bisnonna Domenica Pasqua,
venuta a mancare il 25 marzo 2021.

Acknowledgements

First and foremost, I would like to thank my supervisor Ernesto Sanchez and PhD student Annachiara Ruospo for their guidance through each stage of the process.
I would also like to show gratitude to PhD student Andrea Floridia. He assists me during the learning stage about gem5.
Special greetings to my beloved Federica for her love and encouragement during the last years despite being far away from me.
I'm also grateful to my family Dario, Beatrice and my sister Giulia for their support and love. Without them, I would never reach this goal and I would not be the person I am. They have been supportive, even during the hard times. Finally, I also want to wish my mum a speedy and full recovery from her disease.
For many memorable evenings out and in, I must thank all my old and new housemates, in particular Riccardo, William and Stefano along with the nearest friends, like Mattia and Bruno above all.
I also place on record, my sense of gratitude to one and all, who directly or indirectly, have lent their hand in this venture.

Abstract

Testing hardware faults is a crucial step for the companies, since they can lead to failures that may impact on the dependability of their systems. Indeed, the companies invest money and time in order to test their hardware components, especially in safety-critical domain, such as automotive, robotic and avionic. Furthermore the companies must be comply with various standards like the ISO 26262, an international standard for functional safety of electrical and/or electronic systems in serial production road vehicles. So there is a growing interest towards the development of new strategies for evaluating the dependability of safety-critical systems.

In particular, the main focus of this thesis is to develop a framework that evaluates faults that may occur at the memory level. These faults can be of different nature, such as Single Bit Upsets (SBUs), stuck-at bits, and block errors. The framework mimics the behaviour of a radiation beam in order to investigate the correlation between this radiation and the failures occurring within the memory device.

It has been validated by using an Artificial Neural Network running on a hardware device which is simulated on gem5. Gem5 is a simulator used for computer system research at both architecture and micro-architecture levels. In particular, in this thesis I developed two DRAMs, similar to the ones used in the radiation beam testing. These two memory devices contain different memory sections in order to isolate the source of error. Gem5 is configured using the System-call Emulation (SE) mode and a bare-metal approach, in order to be OS independent. The benefits of this model are that it can be easily modified for simulating other systems, such as changing specific components or settings and that is easily reproducible.

Table of Contents

List of Tables x

List of Figures xii

Acronyms xiv

1 Introduction 2

2 Background 5
2.1 The gem5 simulator . . . 5
2.2 System configuration . . . 6
2.2.1 System Call Emulation (SE) Mode . . . 6
2.2.2 CPU model . . . 7
2.2.3 Memory model . . . 7
2.2.4 File configurations . . . 11

3 Proposed approach 15

4 Experimental results 20
4.1 Golden values . . . 20
4.2 Fault injection . . . 21
4.3 Fault analysis . . . 22
4.3.1 32-bit ANN floating-point application fault analysis . . . 22
4.3.2 16-bit ANN integer application fault analysis . . . 24

5 Conclusions 28
5.1 Future works . . . 29

Bibliography 31

List of Tables

2.1 DRAMs main characteristics 8
2.2 DRAMs timing characteristics 9
2.3 DRAMs power characteristics 9
2.4 System memory configuration 11

4.1 Golden output values for ANN floating-point 32-bit application 21
4.2 Golden output values for ANN integer 16-bit application 22
4.3 fault injection main characteristics 22
4.4 Fault types 23
4.5 Faults leading to detected errors 24
4.6 Faults leading to detected errors 26

List of Figures

2.1 System top-level diagram 6
2.2 TimingSimpleCPU structure 8
2.3 A general page table structure 10
2.4 System structure for a simpleCPU model 13

3.1 Example of output obtained through MyMonitor for the 32-bit floating-point ANN application 16
3.2 18

4.1 Floating-point 32-bit diagram 23
4.2 Floating-point 32-bit diagram 24
4.3 Integer 16 bit diagram 25
4.4 Integer 16 bit diagram 26

Acronyms

DRAM
Dynamic Random Access Memory

ANN
Artificial Neural Network

OS
Operating System

ISA
Instruction Set Architecture

SoC
System on Chip

FPGA
Field Programmable Gate Array

EABI
Embedded Application Binary Interface

SDC
Silent Data Corruption

Chapter 1

Introduction

Nowadays, the testing of multiple faults is becoming more and more important, difficult and the necessity of new strategies for this goal is crucial. As high-performance computing systems continue to grow in scale and complexity, the study of faults and errors is critical to the design of future systems and mitigation schemes. If current predictions hold, future exascale systems expected in the early 2020s will see a hundred-fold increase in the amount of main memory (DRAM) and cache memory (SRAM) compared to current levels.[1]

Fortunately, the growing interest towards the ANN, since they are structurally capable of emulating biological neural networks. An artificial neural network is a mathematical model inspired by biological neural networks. An ANN is based on a collection of connected nodes called artificial neurons. The ones used for engineering purposes consists of inputs that are multiplied by weights. These weights are then computed by a mathematical function that determines the activation of a neuron.[2] Due to the growing use of ANN in safety-critical applications, it is important to analyze the ANN behavior under a fault injection campaign. We can notice that "upon proper analysis of the data, we designed the preprocessors which reduced the hardware complexity to a minimum and eliminated the use of external fault identifiers. Unlike the other schemes reported in the literature, real-time fault detection is feasible using the neural processor presented in the paper."[3]

In this thesis, I follow a particular radiation campaign managed that aims to investigate the impact of radiation-induced soft errors on the reliability of approximate computing systems.[4] This test campaign was carried out at the Rutherford Appleton Laboratories, UK, where an atmospheric-like neutron spectrum is delivered in the ChipIR beamline at the second target station of the ISIS neutron source. The neutron flux in the beamline is approximately 109 times larger than the atmospheric neutron flux. The ChipIr facility provides a neutron flux of about 5 $\times$ 106 n/cm2/s for energies above 10 MeV.[4]

Since ground-level radiation experiments are very costly, there is the need to replicate these experiments without this experimental setup. So, this thesis aims to reproduce this radiation campaign through the use of a framework developed on the `gem5` simulator. This framework, in particular, evaluates faults that may occur at the memory level.

This thesis is organized as follows. Chapter 2 gives background notions about gem5, focusing on the components used during this work. Hence, it explains the design choices to have a faithful depiction of the system setup used during the radiation campaign. Chapter

3 describes the proposed approach, as general as possible, to replicate this work into other possible configurations. Then, chapter 4 describes the analysis of the experimental results. The analysis is conducted both on a 16-bit integer ANN either a 32-bit float ANN, showing the behaviours of the faults in different neural networks applications. Finally, Chapter 5 concludes with suggestions and advice for future works, stressing the importance of DRAM testing.

Chapter 2

Background

The following chapter aims to explain some fundamental and preliminary concepts concerning the various areas covered by the whole project. Initially, the gem5 simulator is described, highlighting the flexibility and the advantages of this simulator. Subsequently is explained the experimental test setup used in the radiation campaign. Finally, the last part highlights the gem5 configuration steps to replicate this setup.

2.1 The gem5 simulator

Gem5 is a modular simulator platform for computer-system research. Its infrastructure is the merger of M5 and GEMS simulators, providing either a highly configurable simulation framework, either a detailed and flexible memory system.[5] The gem5 simulator provides a flexible, modular simulation system that is capable of evaluating a broad range of systems and is widely available to all researchers.[5]

Gem5 provides four interpretation-based CPU models: a simple one-CPI CPU, a detailed model of an in-order CPU and a detailed model of an out-of-order CPU. These CPU models use a common high-level ISA description. Gem5 features a detailed, event-driven memory system including caches, crossbars, snoop filters, and a fast and accurate DRAM controller model, for capturing the impact of current and emerging memories, e.g. LPDDR3/4/5, DDR3/4, HBM1/2/3, HMC, WideIO1/2. The components can be arranged flexibly, e.g., to model complex multi-level non-uniform cache hierarchies with heterogeneous memories. Furthermore, gem5 decouples Instruction Set Architecture (ISA) semantics from its CPU models, enabling effective support of multiple ISAs. Currently, gem5 supports the Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86 ISAs.[6]

In particular, gem5 with the ARM ISA can model up to 64 heterogeneous cores of a Realview ARM platform and boot unmodified *Linux* and *Android* with a combination of in-order and out-of-order CPUs. The ARM implementation supports 32 or 64-bit kernels and applications.[6]

2.2 System configuration

The device used during the radiation campaign is a Xilinx Zynq-7000 based system. This device is a System-on-Chip (SoC), which provides an ARM Cortex™ A9 processor attached to a 28 nm Artix©7 FPGA.[4] The ANN application was ported to this embedded system using two external memories:

- two units of the MT41K128M16JT-125, a 2 Gb SDRAM DDR3L from Micron Technologies[7];
- one S27KS0642GABHI020, a 64 Mib HyperRAM™ SelfRefresh DRAM manufactured by Cypress Semiconductor.[8]

In figure 2.1 the top-level diagram of all the system structure is presented. This division has made to isolate the source of errors, where the affected portion of the application are the weights and the processed image data.

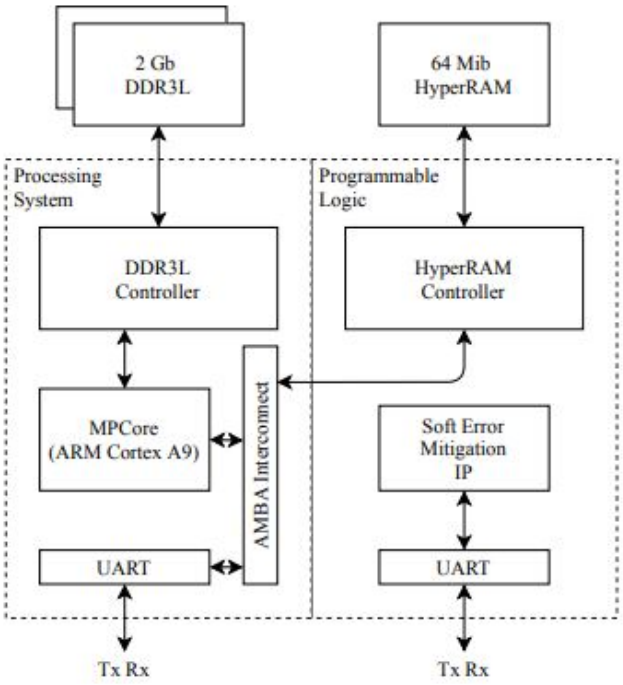

Figure 2.1: System top-level diagram

2.2.1 System Call Emulation (SE) Mode

Gem5 has two different simulation modes:

- System Emulation (SE) mode;

- Full System (FS) mode.

The former focuses on the CPU and memory system and does not emulate the entire system.[9] In the latter, the complete system can be modelled in an OS-based simulation environment.[9] Since I want to implement a bare-metal system, I choose to run the ANN application in the SE mode. Whenever the program executes a system call, gem5 traps and emulates the system call, often passing it to the host operating systems.

The System-call Emulation Mode avoids the need to model devices or an OS by emulating most system-level services.[5] So, there is just the need to specify a binary file that must be executed. Furthermore, the SE mode is faster to run application code because there is no kernel running in the background. There is also no system noise neither driver support. In this way, I have full control over the system, in particular on the memory model, which is the main focus of this thesis.

2.2.2 CPU model

The system core follows a gem5 *TimingSimpleCPU*[10], which is a purely functional in-order processor model. The TimingSimpleCPU is the version of BaseSimpleCPU that uses *timing memory accesses*. It defines the port that is used to hook up to memory and connects the CPU to the cache. It also defines the necessary functions for handling the response from memory to the accesses sent out. It stalls on cache accesses and waits for the memory system to respond before proceeding. In figure 2.2 is shown the main structure of a TimingSimpleCPU.

The timing access is the most realistic access method and is used for approximately-timed simulation, which considers the realistic timing, and models the queuing delay and the resource contention.[9]

This CPU model is chosen among the others because it is possible to avoid caches. In this way, all the data is read or write to the main memory, described in subsection 2.2.3, while keeping the timing memory access mode. This design choice leads to an unrealistic structure of an embedded system. Nowadays, every embedded system CPU has some cache structure to reduce the average cost to access data from the main memory.

2.2.3 Memory model

I decide to replicate only one unit of the device MT41K128M16JT-125, described in section 2.2. Two units of this device would create a crash of the system due to a page table fault. The page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. However, it does not impact the ANN application either the analysis of the results.

The DRAM characteristics are stored in the file `DRAMCtrl.py`, located in the folder *gem5/src/mem/*. Therefore I created two new DRAMs, named *DDR3L* and *HyperRAM*, inserting the values related to the timings, power consumption according to the ones provided in the manuals. [7] [8]. The main characteristics of both memories are described in table 2.1.

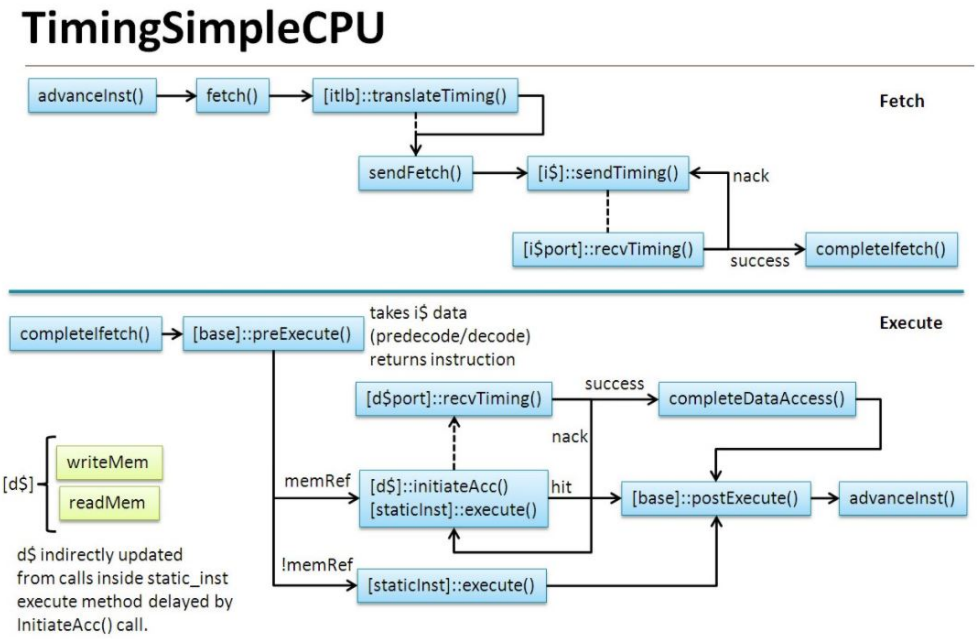

Figure 2.2: TimingSimpleCPU structure

	DDR3L	HyperRAM
device size	256 MB	256 MB
device bus width	8	8
burst length	8	8
device rowbuffer size	16 kB	16 kB
devices per rank	16	16
ranks per channel	2	2
banks per rank	8	8
activation limit	4	2

Table 2.1: DRAMs main characteristics

The timing characteristics of the two memory units are described in table 2.2.

	DDR3L	HyperRAM
tCK	1.25 ns	5 ns
tBURST	5 ns	20 ns
tRCD	13.75 ns	18 ns
tCL	13.75 ns	18 ns
tRP	13.75 ns	18 ns
tRAS	35 ns	42 ns
tRRD	6 ns	10 ns
tXAW	30 ns	50 ns
tRFC	160 ns	210 ns
tWR	15 ns	15 ns
tWTR	7.5 ns	15 ns
tRTP	7.5 ns	20 ns
tRTW	7.5 ns	10 ns
tCS	2.5 ns	10 ns
tREFI	7.8 µs	3.9 µs
tXP	6ns	N.A
tXS	170ns	N.A

Table 2.2: DRAMs timing characteristics

The power characteristics of the two memory units are described in table 2.3

	DDR3L	HyperRAM	HyperRAM2[1]
IDD0	46 mA	35 mA	35 mA
IDD2N	21 mA	35 mA	35 mA
IDD3N	34 mA	0.08 mA	0.09 mA
IDD4W	138 mA	25 mA	30 mA
IDD4R	128 mA	25 mA	30 mA
IDD5	180 mA	1mA	1 mA
IDD3P1	21 mA	8 mA	8 mA
IDD2P1	15 mA	8 mA	8 mA
IDD6	12 mA	1 mA	1 mA
VDD	1.35 V	1.8 V	3.0 V

Table 2.3: DRAMs power characteristics

[1]HyperRAM2 is introduced since HyperRAM has two different operational voltages

Along with these memory models, gem5 has a page table, a data structure used to store the mapping between *virtual* addresses and *physical* ones. Virtual addresses are used by the program executed by the accessing process, while physical addresses are used by the hardware, i.e. the two DRAMs. The relationship between physical and virtual addresses is shown in figure 2.3.

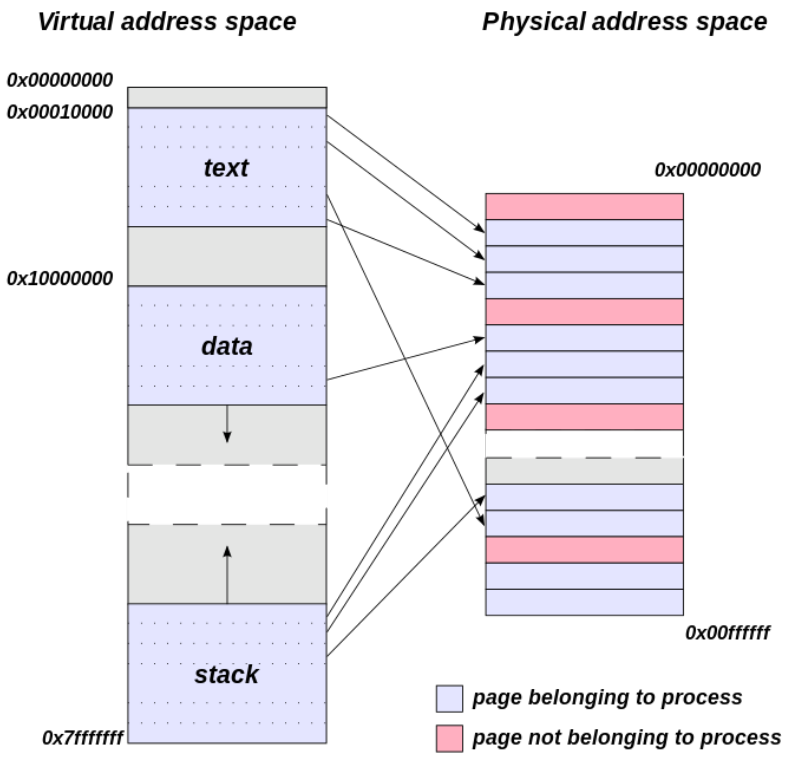

Figure 2.3: A general page table structure

Since the two memories must have a specified physical address, I slightly modified the code contained in the `page_table.cc` file, located in the folder *gem5/src/mem/*. The page table allocation algorithm is:

Algorithm 1 page table translation

```
1: procedure BOOL EMULATIONPAGETABLE::TRANSLATE(vaddr, &paddr)
2:     Define pointer entry of type Entry
3:     if !entry then
4:         return false
5:     end if
6:     if virtualaddress ≤ 2214592512 then      ▷ 2214592512 the last physical address
7:         Physical address is set to an offset due to the virtual address plus the physical
       address of the entry;
8:     else
9:         Physical address is set to the virtual one
10:    end if
11:    return true
12: end procedure
```

In this way, the correct memory section can be selected, especially if there is a need for a particular memory address.

2.2.4 File configurations

The memories sections are split into the two memory devices presented in 2.2.3, divided into:

- `text`: executable instructions;
- `data`: constants and statically allocated variables;
- `heap`: dynamically allocated variables;
- `stack`: store parameters for function calls, return addresses, and local variables.

These sections are configured into the linker file `linker.ld` file. In table 2.4 is displayed the memory segments partition.

memory type	segments
DDR3L	text bss stack
HyperRAM	data heap

Table 2.4: System memory configuration

This startup.s file initialize the *.bss* the *.data* and the *.stack* section and then it finally calls the main program. Along with the linker file, there is a `startup.s` file: this file contains:

- the reset handler which is executed after CPU reset;
- the setup values for the Main Stack Pointer (MSP);
- interrupt vectors that are device-specific with weak functions that implement default routines;
- the actual jump to the main program.

Furthermore, there is a `syscall.c` file. This file contains all the implementation of system calls that are used in the system to correctly compile and boot the program. The most important function is *_sbrk*.

The _sbrk function is used to change the space allocated for the calling process. The change is made by adding incr bytes to the process's break value and allocating the appropriate amount of space. The amount of allocated space increases when incr is positive and decreases when incr is negative. My implementation is:

Algorithm 2 _sbrk function

```
1: procedure _SBRK(incr)
2:     Variables initialization
3:     if heap_end = False then
4:         Assign the pointer to the last element of the heap to heap_end
5:     end if
6:     Update the prev_heap_end to heap_end
7:     if heap_end + incr > heap_limit then
8:         return -1                          ▷ Not enough memory available
9:     end if
10:    Increment heap_end of incr
11:    Return prev_heap_end
12: end procedure
```

In this way, the system does not have any OS, keeping it as easy as possible. This solution can represent, with some degree of fidelity, a general embedded system.

Then I created two monitors, called `MyMonitor` and `MyMonitor1`, to track the data traffic of the two DRAMs. These monitors are placed between the system *membus* and each *mem_ctrls*. In figure 2.4 can be seen the system structure of a SimpleCPU.[11]

All these files are cross-compiled from a Makefile. In personal computers, GNU GCC is a compiler that compiles an application written for LINUX X86 PC. When the host and target architectures are different, the toolchain is called *cross compiler*. Toolchains have a loose name convention like `arch-[vendor]-[os]-[eabi]`, where:

- `arch` refers to a target architecture, e.g. ARM;
- `vendor` refers to toolchain supplier;
- `os` refers to the target OS;

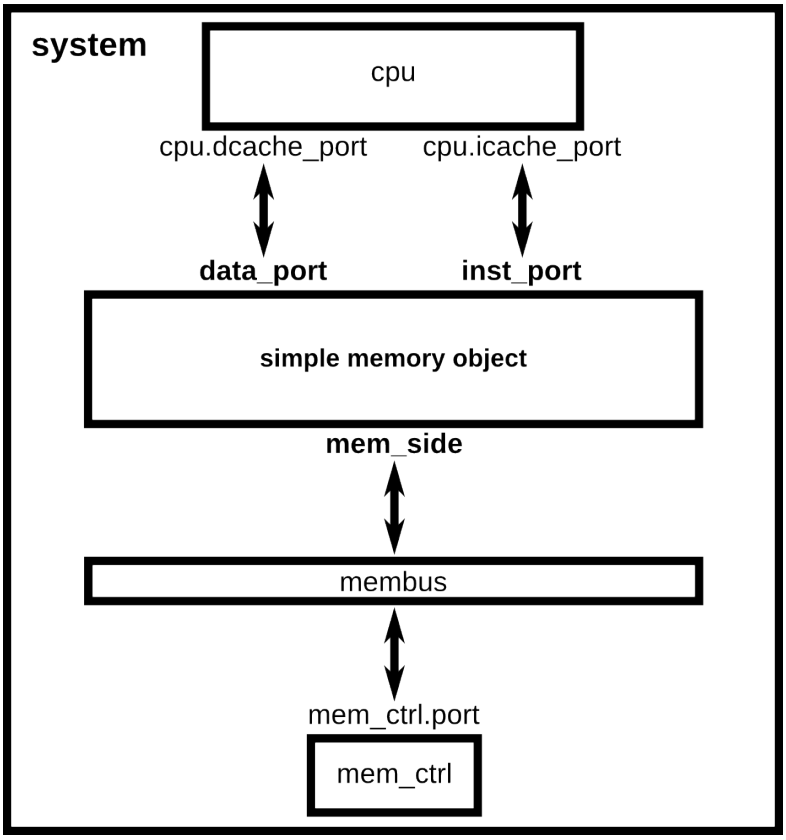

Figure 2.4: System structure for a simpleCPU model

- `eabi` refers to an Embedded Application Binary Interface (EABI).

For the aim of this thesis, I decided to use arm-none-eabi. This toolchain targets for ARM architecture, has no vendor, does not target an operating system and complies with the ARM EABI. Then, the Makefile specifies the name of the target ARM architecture. In this case, the target architecture is *armv7-a*, to be compliant with the one used in the setup of the radiation campaign. Finally, I use the least optimization possible through the *-O0* option flag. Otherwise, the variables that should go into the *.stack* section are stored in the internal register instead.

Chapter 3

Proposed approach

This chapter aims to describe the approach used to perform fault injection on DRAM of ANN applications on gem5. This methodology describes a procedure for inserting random faults at the software level on an ANN application. This procedure is applied either on a 16-bit integer ANN application either on a 32-bit floating-point ANN one. In this way, there is also space to comment on their differences and the analogies.

In my thesis, I follow the stuck-at fault model. This model is selected since it is a fault model based on the assumption that a circuit signal has a fixed value of 0 (stuck-at-0) or 1 (stuck-at-1). This model is not a physical model and hence its strength. It is far better to model such networks as abstract logic networks and then use ssf on these models than go down to the level of transistors.[11]

The training of a neural network is usually conducted by determining the difference between the processed output of the network and the target output. The network then adjusts its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output that is increasingly similar to the target output. After a sufficient number of these adjustments, the training can be terminated, based upon some criteria. So, in our case, the ANN has multiple outputs:

- one output as a standard classifier, i.e. the estimated output;
- ten outputs to indicate the correspondent single-digit number and its confidence that this network has in its classification of the input signal.

In other words, the ANN estimates its confidence of each one-digit number. Therefore, it chooses the one on which it relies the most. In case there are two or more numbers with the same grade of confidence, the ANN selects the first one in cardinal order. An example of these behaviours, related to a 32-bit floating-point ANN application, is shown in figure 3.1.

The first twenty-five lines describe the outputs of the first image. In line 5, at the address *0x80000000*, the monitor stores the expected result of the ANN, which is five. In line 17, which corresponds to output five, there is a value greater than the others since it stores a confidence value of `1.52702`.

```
1       0: system.monitor: Created monitor system.monitor with sample period 1000000000 ticks (1 ms)
2       0: system.monitor2: Created monitor system.monitor2 with sample period 1000000000 ticks (1 ms)
3 1638000: system.monitor2: Forwarded write request at address 0x80000008 of size 4 and data 0
4 1657000: system.monitor2: Latency: 19000
5 2936797409000: system.monitor2: Forwarded write request at address 0x80000000 of size 4 and data 5
6 2936797428000: system.monitor2: Latency: 19000
7 2936799198000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00678451
8 2936799217000: system.monitor2: Latency: 19000
9 2936801876000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00372169
10 2936801895000: system.monitor2: Latency: 19000
11 2936804704000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00433318
12 2936804723000: system.monitor2: Latency: 19000
13 2936807382000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00488683
14 2936807401000: system.monitor2: Latency: 19000
15 2936810225000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.0086848
16 2936810244000: system.monitor2: Latency: 19000
17 2936812903000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 1.52702
18 2936812922000: system.monitor2: Latency: 19000
19 2936815581000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.0108646
20 2936815600000: system.monitor2: Latency: 19000
21 2936818404000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00423347
22 2936818423000: system.monitor2: Latency: 19000
23 2936821082000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00969567
24 2936821101000: system.monitor2: Latency: 19000
25 2936823760000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.0150891
26 2936823779000: system.monitor2: Latency: 19000
27 5827787852000: system.monitor2: Forwarded write request at address 0x80000000 of size 4 and data 5
28 5827787871000: system.monitor2: Latency: 19000
29 5827789699000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.0526477
30 5827789718000: system.monitor2: Latency: 19000
31 5827792615000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00532987
32 5827792634000: system.monitor2: Latency: 19000
33 5827795394000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.0251464
34 5827795413000: system.monitor2: Latency: 19000
35 5827798329000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.0674277
36 5827798348000: system.monitor2: Latency: 19000
37 5827801094000: system.monitor2: Forwarded write request at address 0x80000004 of size 4 and data 0.00926839
```

Figure 3.1: Example of output obtained through MyMonitor for the 32-bit floating-point ANN application

One problem is the impossibility of providing images to the application via files. Since there is no operating system, I can't use any system call that implies the OS. Indeed, there is no possibility to manipulate files, print some variables, pass command arguments and on and on. To solve the problem, I convert the images into matrixes of floating-point type for the 32-bit floating-point ANN application and matrixes of integer type for the 16-bit integer ANN application. Therefore I stored them as a static array of matrixes.

To evaluate the outputs of these ANNs, and hence their predictions, I applied these images to the ANNs. In this way, I get the golden results, which are the outputs obtained without any fault. To monitor the system, I take some particular memory addresses and then visualize their values through MyMonitor, collecting all the data that I need for my study. They are:

- the predicted single-digit number, stored in *0x80000000*;
- the confidence rating, in cardinal order, stored in *0x80000004*;

- the system time;
- the latency of the memory store.

One example of the data obtained through Mymonitor is shown in figure 3.1.

Before the analysis, I create a fault list, using an array of type fault_t, a struct in which we have the main characteristics of a fault. They are:

- the layer of the ANN in which I inject this fault;
- the position of this defect;
- the channel;
- the width;
- the height;
- the bit position;
- the type of stuck-at-fault.

The first five attributes are related to the position of the target weight. On the other hand, the last two ones are related to the fault injection mechanism. The first is the type of stuck-at (either 0 or 1). The latter is the bit position, which has a value between 0 and *n-1*, where n is the number of bits of the weights. Hence, these faults are inserted in DRAM through a software function that takes as inputs the fault peculiarities, changing bits according to the stuck-at-fault type. It is done by following these fault characteristics, already described, using the following algorithm:

Algorithm 3 soft_error_injection

```
1: procedure SOFT_ERROR_INJECTION(index)
2:     Variables initialization
3:     Get the target weight, through reading the layer
4:     Inject the stuck_at fault at the target weight
5:     Write the faulty weight to the target weigth
6: end procedure
```

Finally, the idea is to inject faults one by one, evaluating the behaviour accordingly to Silent Data Corruption (SDC) detection mechanism. [12]

A fault is *detected* whether one of the following situations occur:

- `SDC-1:` a Silent Data Corruption (SDC) failure is a deviation of the network output from the golden network result, leading to a misprediction. Hence, the fault causes the image to be wrongly classified;
- `SDC-10%:` the faulty network correctly predicts the result but assigns a confidence score which varies by more than +/-10% of its fault-free execution;

- `SDC-20%:` the faulty network correctly predicts the result but assigns a confidence score which varies by more than +/-20% of its fault-free execution;
- `Hang:` the fault causes the system to hang and the HDL simulation never finishes;
- `Crash:` it is the opposite situation of the previous one. The HDL simulation immediately stops as a consequence of the fault.

Overall, the methodology proposed in this thesis proposes a single simulation without any error injected in the application. Hence, a fault in the DRAM memory is applied, so the result of the ANN application, track by MyMonitor, are stored in a faulty result file. Finally, there is the comparison of the golden results and the faulty ones, in order to classify them according to the cited SDC detection mechanism. Figure 3.2 shows the architecture of the method used and highlights the various blocks that compose the system.

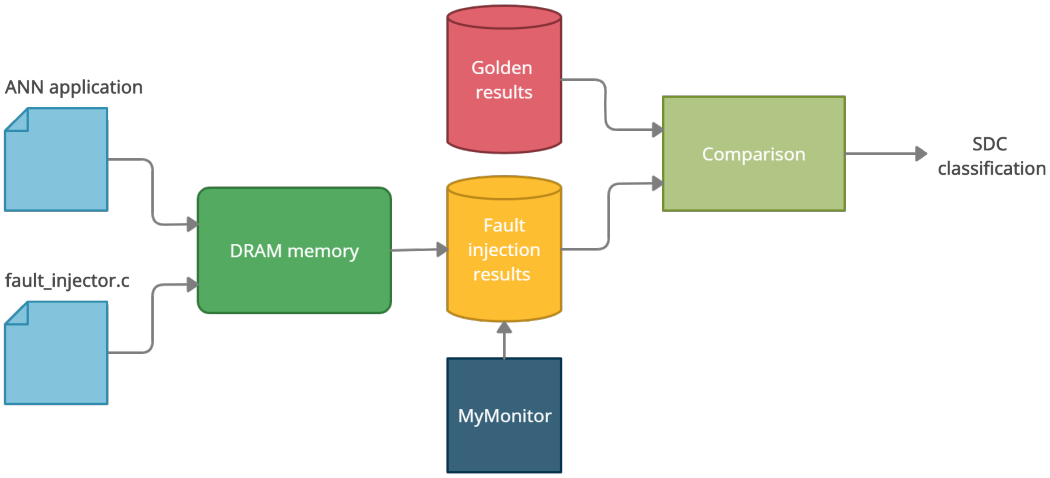

Figure 3.2

Results of the described approach are presented in chapter 4, *Experimental results*.

Chapter 4

Experimental results

The purpose of this chapter is to analyse some random fault injections to the system, using the proposed approach described in Chapter 3. These faults are injected either on a 16-bit integer ANN application and a 32-bit floating-point ANN one. These applications are written in C language, composing a total of twenty-two files each. The first part of this chapter deals with a little degression of this ANN application. To evaluate the approach, the system described in Chapter 2 is targeted, using a 32-bit ARM Cortex A9, two DRAM memories and the TimingSimpleCPU.

Overall, the purpose of this approach is to evaluate the behaviour of these DRAM faults among different ANN layers and hence the differences and similarities of these two kinds of applications.

4.1 Golden values

In this section, golden values are presented. These values are the ones that the ANN applications evaluate in a fault-free scenario. As already discussed in chapter 3, each ANN estimates its confidence of each one-digit number. Therefore, it chooses the one on which it relies the most. In case there are two or more numbers with the same grade of confidence, the ANN selects the first one in cardinal order. In table 4.1 the ANN floating-point 32-bit application outputs are shown. Each image has a prediction related to a floating-point number which expresses the confidence in this value.

Similarly, in table 4.2, the ANN integer 16-bit application ones are displayed.

Seeing this table, there is an odd behaviour on the 16-bit integer ANN function. Almost all the expected output has a confidence score of `32767`, which is the highest number that can be reached using a signed 16-bit integer. Furthermore, this value is present even related to other one-digit numbers in the same image. Hence, this comportment introduces some errors after the injection of a fault, discussed in 4.3.2.

Image 0 5 1.52702	Image 1 5 0.790647	Image 2 0 1.80845	Image 3 8 1.73504	Image 4 5 1.67788
Image 5 5 1.63971	Image 6 9 1.53866	Image 7 6 1.64181	Image 8 8 1.31346	Image 9 1 1.73478
Image 10 2 1.66945	Image 11 5 1.59841	Image 12 5 1.5512	Image 13 4 1.88076	Image 14 1 1.70697
Image 15 6 1.67884	Image 16 4 1.59015	Image 17 0 1.73323	Image 18 4 1.94977	Image 19 3 1.76429
Image 20 2 1.70951	Image 21 2 1.49161	Image 22 1 1.69595	Image 23 5 1.75724	Image 24 8 1.71161
Image 25 1 1.69072	Image 26 0 1.95635	Image 27 0 1.79684	Image 28 5 1.31981	Image 29 9 1.64174
Image 30 0 1.8645	Image 31 2 1.78268	Image 32 9 1.6116	Image 33 5 1.70745	Image 34 5 1.45337
Image 35 0 1.82736	Image 36 1 1.69603	Image 37 3 1.85674	Image 38 9 1.41799	Image 39 9 1.84334
Image 40 4 1.69482	Image 41 6 1.87303	Image 42 7 1.90492	Image 43 1 1.8138	Image 44 9 1.71095
Image 45 9 1.76778	Image 46 7 1.76624	Image 47 0 1.87437	Image 48 6 1.77714	Image 49 6 1.45596

Table 4.1: Golden output values for ANN floating-point 32-bit application

4.2 Fault injection

The fault injection main characteristics and timings are presented in table 4.3. Fifty images are provided to each application. In this way, we can examine the effects of the error supplied while keeping a reasonable simulation time. In fact, for the floating-point application, I estimate an average time of 9h and 12 minutes for each fault, while for the integer application, I evaluate an average time of 1h and 42 minutes for each error. The two timings are evaluated on two different laptops. In this way, I launched the floating-point application on a laptop and the integer one on the other. This configuration may help to obtain different evaluation times, as described in table 4.3. Furthermore, the number of pictures is kept low to avoid memory repercussions, since they are stored in memory, as discussed in chapter 3.

The number of faults is chosen to guarantee a tradeoff between timings and the amount of data.

Then, some random faults are generated among all the neural network. All errors are classified depending on the layer level. A schematic view is presented in table 4.4. As we can see, they are spread among all the different layers. It leads to an idea of the impact that a fault has on each layer of the ANN.

Image 0 3 32767	Image 1 6 32767	Image 2 7 32767	Image 3 5 32767	Image 4 5 32767
Image 5 4 32767	Image 6 1 32767	Image 7 7 32767	Image 8 5 32767	Image 9 2 32767
Image 10 1 32767	Image 11 0 32767	Image 12 2 32767	Image 13 5 32767	Image 14 2 32767
Image 15 1 32767	Image 16 1 32767	Image 17 0 32767	Image 18 5 32767	Image 19 0 32767
Image 20 0 32767	Image 21 4 32767	Image 22 2 32767	Image 23 1 32767	Image 24 2 32767
Image 25 0 32767	Image 26 1 32767	Image 27 3 32767	Image 28 7 32767	Image 29 2 32767
Image 30 9 15273	Image 31 0 32767	Image 32 3 32767	Image 33 0 32767	Image 34 2 32767
Image 35 0 32767	Image 36 0 32767	Image 37 3 32767	Image 38 3 32767	Image 39 0 32767
Image 40 2 32767	Image 41 2 32767	Image 42 6 32767	Image 43 7 32767	Image 44 3 32767
Image 45 2 32767	Image 46 1 32767	Image 47 1 32767	Image 48 0 32767	Image 49 4 32767

Table 4.2: Golden output values for ANN integer 16-bit application

ANN application	Layers	Images	Number of faults	Total time
float 32 bit	5	50	70	~640 h
integer 16 bit	5	50	70	~120 h

Table 4.3: fault injection main characteristics

4.3 Fault analysis

4.3.1 32-bit ANN floating-point application fault analysis

Firstly, I decided to evaluate the number of faults that introduce some errors in one or more images. As shown in figure 4.1, four faults (6%) lead to at least one misprediction of some pictures into the 32-bit floating-point ANN application. Fifteen errors (21%) have at least one image with a confidence level different from the golden one, but they are not detected. Finally, fifty-one faults (73%) are correct.

This classification is done by following the SDC-analysis previously described in chapter 3. Some faults are not identified, since the SDC is less than 10%. They can be seen in table 4.5.

ANN type	layer level	number of faults
16 bit integer	conv 1	12
	conv 2	15
	conv 3	19
	fc1	14
	fc2	10
32 bit floating point	conv 1	15
	conv 2	16
	conv 3	14
	fc1	14
	fc2	11

Table 4.4: Fault types

This result is reasonable since the injected errors are more or less effective accordingly to the characteristics of the position of the fault, i.e. ANN layer, bit position, width, height and stuck-at type.

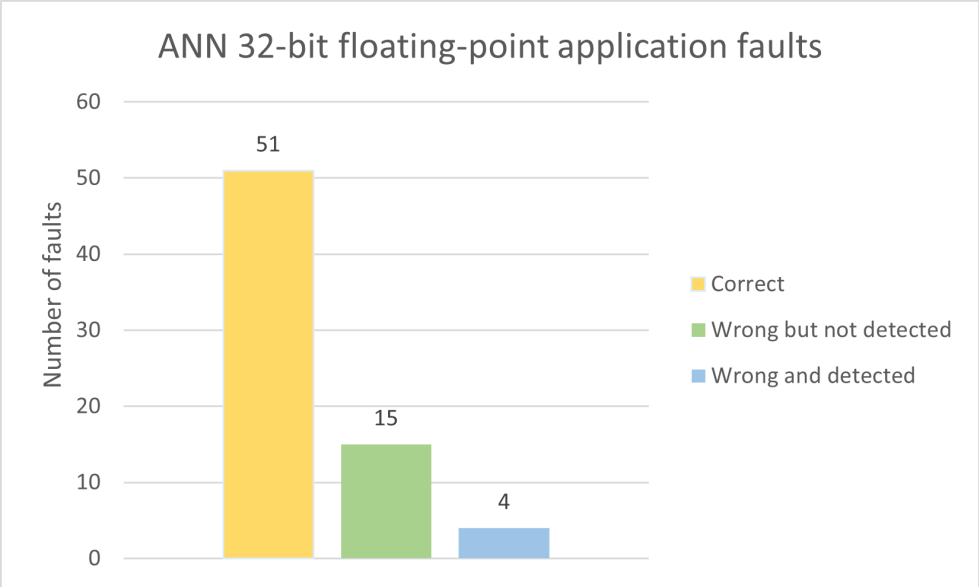

Figure 4.1: Floating-point 32-bit diagram

Therefore, I classify the faults according to the SDC data analysis. These results can be seen in figure 4.2. The injected faults do not cause any hang as well as any crash. The number of faulty images is more or less balanced among type *SDC-1*, type *SDC-10%*, type *SDC-20%*, even though this last one is less common. It is interesting to report that all the *SDC-1* errors refer to `fault 42`. These errors lead to wrongly evaluate the confidence level of thirty-six pictures, with values of the order of magnitude of 10^{34} and in some cases

10^{35}. Furthermore, surprisingly all these images are predicted as *2* by the ANN. Hence, thirty-three pictures are rated as *SDC-1*, while three are evaluated as *SDC-20%*.

Focusing on the fault characteristics, as expected the errors on higher bits have more influence on the ANN predictions. In particular, faults on bit thirty-one refers to the ones that modify the sign of the value, while bit thirty has a significant impact on the value. Finally, another consideration may be done basing on the impact of layers on the results. *Conv1* is the layer more present in table 4.5, a sign that this layer is significant during the ANN evaluation, although the greatest errors stem from `fault 42`, of layer *conv3*.

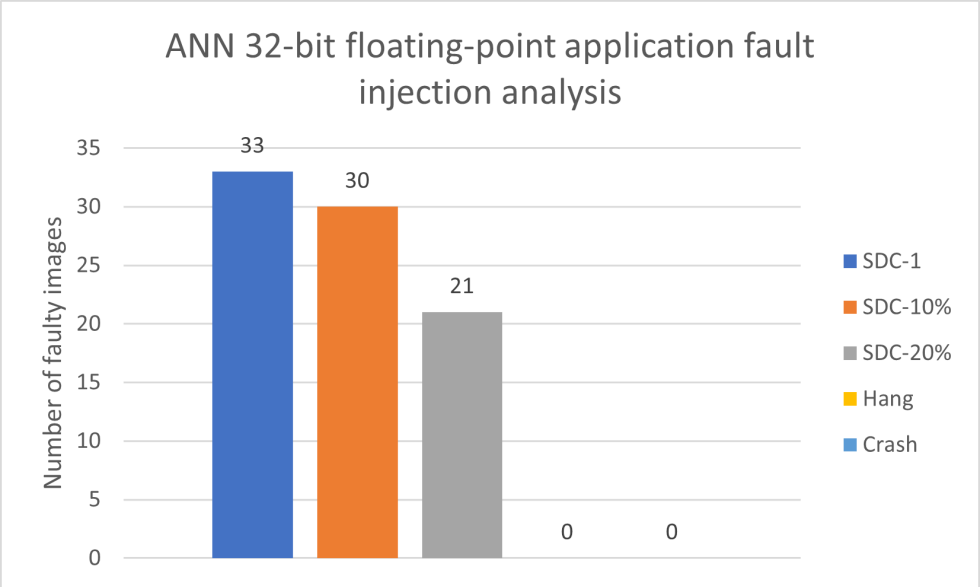

Figure 4.2: Floating-point 32-bit diagram

Fault number	Layer	Output	Channel	Width	Height	Bit	SA type
14	conv1	3	0	0	0	31	1
30	conv1	3	0	4	1	31	1
34	conv1	4	0	0	1	30	1
42	conv3	8	3	4	3	30	1

Table 4.5: Faults leading to detected errors

4.3.2 16-bit ANN integer application fault analysis

As shown in figure 4.3, twenty faults (29%) lead to at least one misprediction of some pictures into the 32-bit floating-point ANN application. Seven errors (10%) have at least one image with a confidence level different from the golden one, but they are not detected. Finally, forty-three faults (61%) are correct. This classification is done by following the

SDC-analysis previously described in chapter 3. Some faults are not identified, since the SDC is less than 10%. They can be seen in table 4.6.

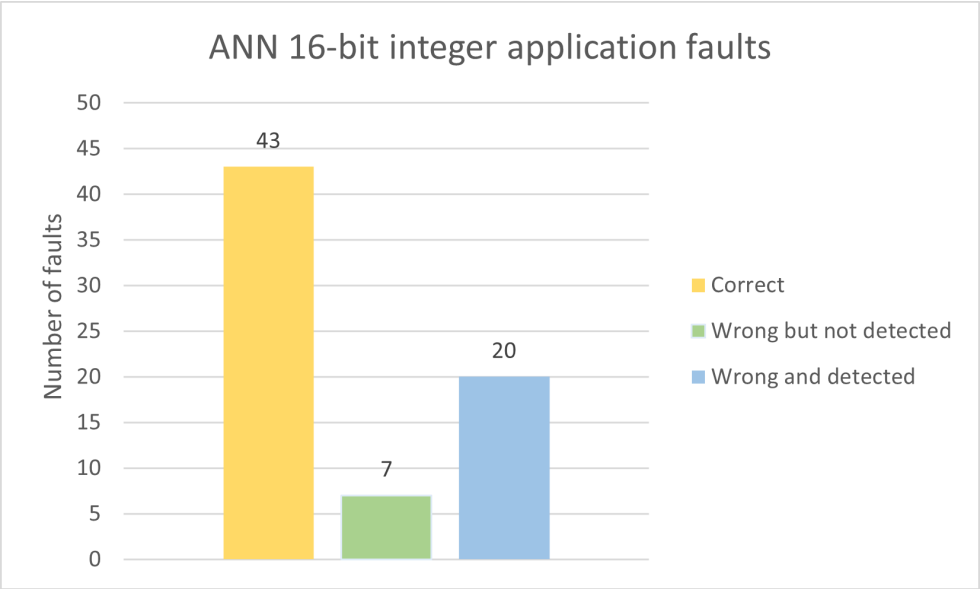

Figure 4.3: Integer 16 bit diagram

This higher percentage of detected faults is due to the different structure of the system. As already described in table 4.2, almost all the predictions output is relatable to the value `32767`, which is the maximum possible value that can be achieved with a signed integer in 16-bit. Furthermore, in some images, this value is present also in other outputs. Hence, in case there are some changes in the prediction, this situation probably leads to an *SDC-1*, more than in other cases. The prevalent type of faulty images is *SDC-1*, as shown in figure 4.4. For the same reason, the faults of class *SDC-10%* and *SDC-20%* are rare.

Similarly to the 32-bit floating-point application, as expected, the errors on higher bits have more influence on the ANN predictions. In particular, faults on bit fifteen modify the sign of the value and faults on bit fourteen has a significant impact on the ANN evaluation. Nevertheless, in some case, a stuck-at fault on a lower bit leads to some error about ANN evaluation. It is shown in table 4.6.

Then, another consideration may be done basing on the impact of layers on the results. In contrast with the analysis done in 4.3.1, *conv2* is not the layer more present in table 4.6. However, *conv1* has still a significant impact on the ANN evaluation. Furthermore, *conv3* is present twice in the table, a sign that is a layer that may lead to some errors, even if they are less common.

Finally, I want to do a last consideration about the two applications. As shown in 4.5 and 4.6, no faults in layers *fc1* and *fc2* are detected. This may lead to the conclusion that these two layers are rarely involved in ANN evaluation, or that their impact is negligible. But these conclusions may be drawn with the support of more data related to these two layers, in future works.

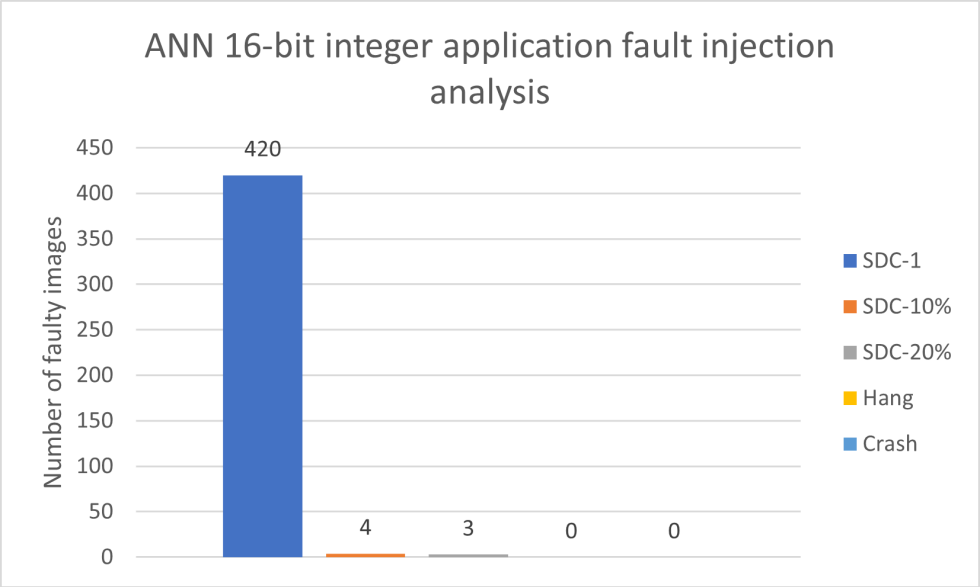

Figure 4.4: Integer 16 bit diagram

Fault number	Layer	Output	Channel	Width	Height	Bit	SA type
0	conv3	13	4	4	3	14	0
4	conv2	0	1	2	2	9	1
6	conv2	3	3	4	0	15	1
7	conv2	3	5	0	4	12	1
9	conv1	2	0	0	0	6	0
10	conv2	13	5	3	3	15	0
13	conv1	3	0	0	1	9	0
14	conv1	4	0	4	0	5	1
21	conv1	0	0	0	0	15	1
22	conv1	5	0	0	2	11	1
29	conv1	4	0	0	0	15	1
30	conv1	4	0	2	3	10	0
35	conv2	7	1	1	1	14	0
36	conv2	11	0	3	2	11	0
40	conv2	15	5	4	4	7	0
41	conv3	32	5	3	2	12	1
53	conv2	11	1	1	1	11	0
58	conv1	3	0	0	3	7	0
59	conv2	8	5	3	0	8	1
64	conv2	6	2	3	1	10	1

Table 4.6: Faults leading to detected errors

Chapter 5

Conclusions

The problem of testing embedded memories has become more complex in recent years [13]. So, this thesis describes a methodology on how to perform fault injection on DRAM of ANN applications on gem5. The approach employs stuck-at-faults inserted through software. As high-performance computing systems continue to grow in scale and complexity, the study of faults and errors is critical to the design of future systems and mitigation schemes. Fault modes in system DRAM are a frequently-investigated key aspect of memory reliability.[1] Hence, there is the need to investigate the impact of soft errors through a particular radiation campaign, like the one described in [4].

But this kind of study is costly. Thus, this thesis proposes a framework that evaluates faults that may occur at the DRAM level, simulating the radiation behaviour of a radiation beam to stimulate failures occurring within the memory device. In fact, simulating a system has always carried the advantage of increased insight and flexibility, at a cost in execution speed and timing fidelitycompared to the real machine. However, until recently, the use of simulation technology for large-scale embedded systems software development and testing has been fairly limited.[14]

Thus, the framework is built on gem5, a flexible, modular simulation system, capable of evaluating a broad range of systems and is widely available to all researchers.[5] In particular, the main focus towards the development of this framework is posed on the replication of two DRAMs memories, the ones used in the radiation campaign. Then, two monitors are created and hence inserted into the system. These monitors are based on a structure, called MyMonitor, developed by myself. MyMonitor arises from the need to track some data from the two DRAMs. In fact, this framework emulates a bare-metal system. Since there is no operating system, it is impossible to use any system calls that involve the OS itself. A bare-metal system provides more flexibility on the memory access, at the cost of having no access to system calls the implies I/O mechanisms. However, experiments clearly show that using this method is possible to insert and then evaluate faults that may occur at the memory level.

However, experiments clearly show that using this method it is possibile to insert and then evaluates faults that may occur at the memory level.

5.1 Future works

The major advantage of this project regards its portability. The approach proposed can be extended to other ISA as well as other system configurations. Possible studies may be conducted on the improvement of this framework, with the introduction of a more complicated and realistic CPU model, such as MinorCPU. This new configuration introduces also some caches, the aim is to reduce the simulation time and have a more realistic behaviour. However, some precautions have to be taken towards the caching system.

Furthermore, an analysis of some layers, like fc1 and fc2, may be done. Fault injection on these two layers during my work does not lead to detection. This may be the subject of more studies, in order to investigate more on the behaviour of some errors. Finally, an aspect that arises from this thesis is the high simulation time.

Hence, another future work may be the focus on the implementation of some multi-threading, to decrease the amount of time to apply the fault injection to the ANN and to evaluate the prediction of the ANN application.

Bibliography

[1] Elisabeth Baseman, Nathan DeBardeleben, Kurt Ferreira, Scott Levy, Steven Raasch, Vilas Sridharan, Taniya Siddiqua, and Qiang Guan. «Improving DRAM Fault Characterization Through Machine Learning». In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. June 2016 (cit. on pp. 2, 28).

[2] N. Gupta. «Artificial neural network». In: *Inter national Conference on Recent Trends in Applied Sciences with Engineering Applications*. Vol. 03. 2013, pp. 24–28 (cit. on p. 2).

[3] Ramachandran Venkatesan and d Balasubramanian Balamurugan. «A Real-Time Hardware Fault Detector Using an Artificial Neural Network for Distance Protection». In: *IEEE TRANSACTIONS ON POWER DELIVERY* 16.1 (Jan. 2001), pp. 75–82 (cit. on p. 2).

[4] Lucas Matana Luza, Daniel Söderström, Georgios Tsiligiannis, Helmut Puchner, Carlo Cazzaniga, Ernesto Sanchez, Alberto Bosio, and Luigi Dilillo. «Investigating the Impact of Radiation-Induced Soft Errors on the Reliability of Approximate Computing Systems». In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Frascati, Italy, Oct. 2020 (cit. on pp. 2, 6, 28).

[5] Nathan Binkert et al. «The gem5 Simulator». In: *ACM SIGARCH Computer Architecture News* 39.2 (May 2011), pp. 1–7 (cit. on pp. 5, 7, 28).

[6] *ABOUT*. gem5. URL: `https://www.gem5.org/about/` (cit. on p. 5).

[7] *DDR3L SDRAM MT41K128M16*. Micron. 2015 (cit. on pp. 6, 7).

[8] *Cypress HyperRAM Self-Refresh DRAM 3.0/1.8V, 64Mb (8MB)*. Infineon. 2015 (cit. on pp. 6, 7).

[9] Ashkan Tousi and Chuan Zhu. *Arm Research Starter Kit: System Modeling using gem5*. 2017. URL: `https://github.com/arm-university/arm-gem5-rsk/blob/master/gem5_rsk.pdf` (cit. on p. 7).

[10] *TimingSimpleCPU Model*. gem5. URL: `https://www.gem5.org/documentation/general_docs/cpu_models/SimpleCPU` (cit. on p. 7).

[11] J.H. Patel. «Stuck-at fault: a fault model for the next millennium». In: *Proc. IEEE International Test Conference 1998)*. Washington, DC, USA, Oct. 1998, p. 1166 (cit. on p. 15).

[12] A. Ruospo, A. Balaara, A. Bosio, and E. Sanchez. «A Pipelined Multi-Level Fault Injector for Deep Neural Networks». In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Frascati, Italy, Oct. 2020 (cit. on p. 17).

[13] S. K. Jain and C. E. Stroud. «Built-in self testing of embedded memories». In: *IEEE Design Test of Computers*. Oct. 1986, pp. 27–37 (cit. on p. 28).

[14] Jacob Engblom, Guillaume Girard, and Bengt Werner. «Testing Embedded Software using Simulated Hardware». In: *Conference ERTS06*. Toulouse, France, Jan. 2006 (cit. on p. 28).