NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF GRADUATE STUDIES
SPECIALIZATION: COMPUTING SYSTEMS, SOFTWARE AND HARDWARE

&

INTERDISCIPLINARY PROGRAM OF GRADUATE STUDIES IN MICROELECTRONICS
SPECIALIZATION: DESIGN OF INTEGRATED CIRCUITS

MASTER THESIS

Microarchitecture-level reliability assessment of multi-bit upsets in processors

Christos M. Gavanas
Georgios A. Katsoridas

Supervisor: **Dimitrios Gizopoulos,** Professor

ATHENS

MAY 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΕΙΔΙΚΕΥΣΗ: ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ, ΥΛΙΚΟ ΚΑΙ ΛΟΓΙΣΜΙΚΟ

&

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΣΤΗ ΜΙΚΡΟΗΛΕΚΤΡΟΝΙΚΗ
ΕΙΔΙΚΕΥΣΗ: ΣΧΕΔΙΑΣΗ ΟΛΟΚΛΗΡΩΜΕΝΩΝ ΚΥΚΛΩΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αξιολόγηση ευπάθειας επεξεργαστών για πολλαπλά ελαττώματα σε επίπεδο μικροαρχιτεκτονικής

Χρήστος Μ. Γαβάνας
Γεώργιος Α. Κατσορίδας

Επιβλέπων: **Δημήτριος Γκιζόπουλος,** Καθηγητής

ΑΘΗΝΑ

ΜΑΪΟΣ 2019

MASTER THESIS

Microarchitecture-level reliability assessment of multi-bit upsets in processors

Christos M. Gavanas
A.M.: M1515

Georgios A. Katsoridas
A.M.: MM303

SUPERVISOR: **Dimitrios Gizopoulos,** Professor

May 2019

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αξιολόγηση ευπάθειας επεξεργαστών για πολλαπλά ελαττώματα σε επίπεδο μικροαρχιτεκτονικής

Χρήστος Μ. Γαβάνας

Α.Μ.: Μ1515

Γεώργιος Α. Κατσορίδας

Α.Μ.: ΜΜ303

ΕΠΙΒΛΕΠΩΝ: **Δημήτριος Γκιζόπουλος,** Καθηγητής

Μάϊος 2019

ABSTRACT

The continuing decrease in feature sizes for modern Integrated Circuits (ICs) leads to an ever-important role of reliability and vulnerability assessments on the core in early stages of the design (pre-silicon validation). With the increase of the lithography resolution in recent technological nodes, the radiation effects play a bigger role, leading to more severe effects in the devices and increased numbers of multi-bit faults. Therefore, it is crucial to utilize some common fault injection mechanisms to evaluate each design, using micro-architectural simulators, which provide us with flexibility and improved latency, compared to RTL (Register Transfer Level) designs.

This thesis focuses on the multi-bit faults, showing their effects on different components of a microarchitectural model of the ARM Cortex-A9 core, implemented on the Gem5 simulator. For that, the GeFIN (Gem-5 based Fault INjector) is used for the fault injection campaigns, with the addition of an improved fault mask generation tool for the creation of fault masks with some particular characteristics. The improved version of the fault mask generator includes the capability for the injection of multi-bit faults in adjacent areas of a structure, a case very common in real environments. The generator also includes the ability to insert faults in interleaved memories, a widely used technique to mitigate the effects of multiple bit upsets.

The results of this study showed that some specific components of the core under test (e.g. the Instruction Translation Lookaside Buffer) showed significant vulnerability to fault injection, with rates as low as 25% correct executions for 1000 experiments, while others like the Level 1 Data/Instruction Caches and the Level 2 Cache showed bigger vulnerability to the increasing number of faults injected, with a variation of as high as 24% between single and triple bit fault injection for the L1 D-Cache. Those numbers were related to the “theoretical” Architectural Vulnerability Factor (AVF), independent of the fabrication technology node. An extension in the calculation was done to compute the AVFs for each technology node from 250 nm to 22 nm, showing increasing AVF rates as the node decreases.

Lastly, a reliability assessment was done, using the Failures in Time (FIT) metric, which showed the highest numbers for the Level 2 Cache, primarily because of its size (4 MBits) with a FIT of 822.9 at the 130 nm. The FIT of the core showed a high of 918 at the same node, while we observed that for nodes smaller than 130 nm the FITs decreased primarily because of the decrease of the raw FIT factor of each technology.

SUBJECT AREA: Computer Architecture

KEYWORDS: fault tolerance, multiple bit faults, microarchitectural simulation, reliability & vulnerability assessment, interleaving

ΠΕΡΙΛΗΨΗ

Η συνεχιζόμενη μείωση στις διαστάσεις των μοντέρνων Ολοκληρωμένων Κυκλωμάτων (Ο.Κ.) οδηγούν στον ολοένα και πιο σημαντικό ρόλο των αξιολογήσεων αξιοπιστίας και ευπάθειας στον επεξεργαστή, σε πρόωρα στάδια της σχεδίασης (pre-silicon validation). Με την εξέλιξη των τεχνολογικών κόμβων, τα αποτελέσματα της ακτινοβολίας παίζουν μεγαλύτερο ρόλο, οδηγώντας σε πιο σημαντικά αποτελέσματα στις συσκευές, με μια επιπρόσθετη αύξηση σε σφάλματα πολλαπλών bit. Συνεπώς, είναι καθοριστική η χρησιμοποίηση κάποιων κοινών μηχανισμών εισαγωγής σφαλμάτων για την αξιολόγηση κάθε σχεδίου, χρησιμοποιώντας προσομοιωτές μικρό-αρχιτεκτονικής, που μας παρέχουν ευελιξία και βελτιωμένη ταχύτητα, σε σύγκριση με τα σχέδια Επιπέδου Μεταφοράς Καταχωρητή.

Αυτή η διπλωματική εργασία, εστιάζει στα σφάλματα πολλών bit, παρουσιάζοντας τα αποτελέσματα τους σε διαφορετικές δομές ενός μικρό-αρχιτεκτονικού μοντέλου του επεξεργαστή ARM Cortex-A9, που έχει υλοποιηθεί στον προσομοιωτή Gem5. Για αυτό τον λόγο χρησιμοποιείται για τις εκστρατείες εισαγωγής σφαλμάτων ο GeFIN (Gem-5 based Fault INjector), με την προσθήκη μιας βελτιωμένης γεννήτριας σφαλμάτων, για τη δημιουργία μασκών σφαλμάτων με κάποια πολύ συγκεκριμένα χαρακτηριστικά. Η βελτιωμένη έκδοση της γεννήτριας, περιλαμβάνει την δυνατότητα για την εισαγωγή σφαλμάτων πολλών bit σε γειτονικές περιοχές κάθε δομής, μια πολύ συνηθισμένη περίπτωση σε πραγματικά περιβάλλοντα. Η γεννήτρια περιλαμβάνει επίσης της δυνατότητα για την εισαγωγή σφαλμάτων σε διεμπλεκόμενες (interleaved) μνήμες, ένας μηχανισμός που χρησιμοποιείται για το περιορισμό των αποτελεσμάτων των σφαλμάτων πολλών bit.

Τα αποτελέσματα αυτής της διπλωματικής εργασίας, έδειξαν ότι κάποιες συγκεκριμένες δομές του επεξεργαστή-υπό-εξέταση (π.χ. ο Instruction Translation Lookaside Buffer) έδειξαν μεγάλη ευπάθεια στην εισαγωγή σφαλμάτων, με ποσοστά έως και 25% σωστών εκτελέσεων για 1000 πειράματα, ενώ άλλες δομές όπως οι Κρυφές Μνήμες Εντολών και Δεδομένων 1ου επιπέδου και η Κρυφή Μνήμη 2ου επιπέδου, έδειξαν μεγαλύτερη ευπάθεια στον αυξανόμενο αριθμό εισαγόμενων σφαλμάτων, με διακυμάνσεις μέχρι και 24% ανάμεσα στη εισαγωγή ενός και τριών σφαλμάτων στην κρυφή μνήμη 1ου επιπέδου. Αυτοί οι αριθμοί σχετιζόταν με τον θεωρητικό Architectural Vulnerability Factor (AVF) και ήταν ανεξάρτητοι από την τεχνολογία κατασκευής. Πραγματοποιήθηκε μια επέκταση στους υπολογισμούς για τον υπολογισμό των AVFs για κάθε τεχνολογικό κόμβο από 250 έως 22 nm, που έδειξε αυξημένα ποσοστά AVF όσο ο κόμβος μειωνόταν.

Τέλος, πραγματοποιήθηκε μια ανάλυση αξιοπιστίας, χρησιμοποιώντας την μετρική Failures in Time (FIT), που έδειξε του υψηλότερους αριθμούς για την Κρυφή Μνήμη 2ου επιπέδου, κυρίως λόγω του μεγέθους της (4 MBits) με ένα FIT ίσο με 822.9 στα 130 nm. Ο FIT του επεξεργαστή είχε μέγιστο το 918 στον ίδιο κόμβο, ενώ παρατηρήσαμε ότι για κόμβους μικρότερους από 130 nm οι FIT μειώνονται, κυρίως επειδή υπάρχει μείωση στον παράγοντα raw FIT κάθε τεχνολογίας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική Υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ανοχή σε σφάλματα, σφάλματα πολλών bit, μικρό-αρχιτεκτονική προσομοίωση, αξιολογήσεις αξιοπιστίας και ευπάθειας, πλέξη μνήμης

ΕΥΧΑΡΙΣΤΙΕΣ

Σε αυτό το σημείο θα θέλαμε να ευχαριστήσουμε ιδιαίτερα τον επιβλέποντα καθηγητή μας κ. Δημήτρη Γκιζόπουλο που μας παρείχε την δυνατότητα ενασχόλησης με ένα πολύ ενδιαφέρον θέμα σε άμεση συνάρτηση με τις τρέχοντες έρευνες του εργαστηρίου του. Θα θέλαμε επίσης να ευχαριστήσουμε θερμά τον υποψήφιο διδάκτορα Σάκη Χατζηδημητρίου για την αμέριστη και άμεση υποστήριξη του καθ' όλη της διάρκεια της εκπόνησης της παρούσας διπλωματικής εργασίας. Η βοήθεια του ήταν καθοριστική και θα θέλαμε να του ευχηθούμε τα καλύτερα για την συνέχεια της διδακτορικής του διατριβής.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

FOREWORD

This thesis was implemented from October 2018 through May of 2019. The experiments were conducted in the Computer Architecture Laboratory (cal.di.uoa.gr) of the Department of Informatics & Telecommunications, National & Kapodistrian University of Athens.

1. INTRODUCTION

1.1 Faults in integrated circuits

Fault tolerance on semiconductor devices has been an important issue for many years. The interest in studying fault-tolerant techniques in order to keep integrated circuits (ICs) operational in many environments (including space) is continuously increasing. A fault is defined as an undesired state change in hardware. When considering the duration or persistence, a fault can have a transient effect (transient fault) or a permanent one (permanent or stuck-at fault). A fault can also be intermittent, which means that a bit is stuck at 0 or 1 for a certain interval. If a transient fault flips a bit in a hardware structure, that bit can be overwritten to remove the fault. Faults affecting two or more bits (all of the bits, flip) are called multi-bit faults. The nature of the transient faults in a chip, for example in an SRAM (Static random-access memory) array, is often caused by high-energy particle strikes, such as neutrons from cosmic radiation. Those particles deposit an amount of charge onto transistors, as they pass from a silicon device. The required amount of charge that can invert the logic state of the device, causing a transient fault, is defined as *critical charge*. This critical charge is reduced in new technology nodes as a result of technology scaling. The changes in transistor sizes and characteristics affect the charge or current needed to store information. Consequently, devices become more vulnerable to radiation, and this means that particles with a small charge, which would once not affect a circuit, are now much more likely to produce a fault.

When considering the number and time of affected memory cells, there are three major classes of transient faults that can occur due to particle strikes: a single-bit fault (SBF) occurs given a single strike affecting a single bit, a spatial multi-bit fault (sMBF) occurs given a single strike affecting multiple bits at the same instance in time and a temporal multi-bit fault (tMBF) in which multiple strikes (spaced in time) lead to multiple flipped bits the next time a set of bits is read.

An error is defined as an incorrect result in program output. Errors can be classified based on their result on the system behavior. An undetected fault can cause silent data corruption (SDC), which causes a program to behave incorrectly without being detected by the hardware. Furthermore, faults that are detected but not corrected are called detected uncorrected errors (DUE). Detectable errors that will not cause any abnormality are called false DUEs, while the errors that will cause abnormal behavior are called true DUEs.

1.1.1 Radiation effects in integrated circuits

A single particle can strike either the combinational logic or the sequential logic in the silicon of the hardware structure. Figure 1.1 illustrates a typical circuit topology common in many sequential circuits [6]. The data from the first latch is “released” to the combinatorial logic on the falling or the rising clock edge, which enables logic operations. The output of the combinatorial logic reaches the second latch sometime before the next falling or rising clock edge. Upon this clock edge, the available data at the end of the logic circuitry is stored in the latch.

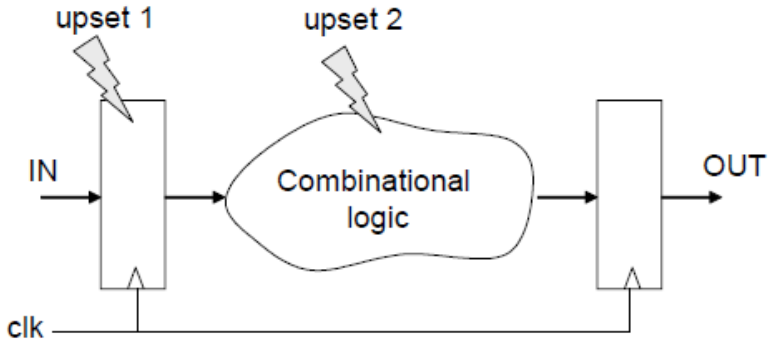


Figure 1.1: Upsets hitting combinational and sequential logic

When a charged particle strikes one of the sensitive nodes of a memory cell, such as the drain in an off-state transistor, it generates a transient current pulse that can turn on the gate of the opposite transistor thus producing an inversion in the stored value (a bit-flip in the memory cell).

Memory cells have two stable states, one that represents a stored '0' and one that represents a '1'. In each state, two transistors are turned on, and two are turned off. A bit-flip in the memory element occurs when an energetic particle causes the state of the transistors in the circuit to reverse, as seen in Figure 1.2. This is called a Single Event Upset (SEU).

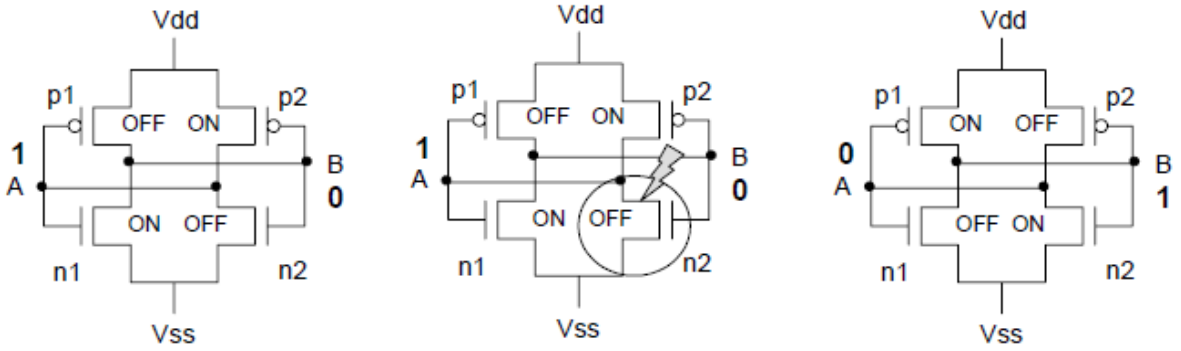


Figure 1.2: Single Event Upset (SEU) effect in an SRAM memory cell

When a charged particle hits the combinatorial logic block, it also generates a transient current pulse, on what is called a single transient effect (SET). If the logic is fast enough to propagate the induced transient pulse, then the SET will eventually appear at the input of the second latch, as shown in Figure 1.1, where it may be interpreted as a valid signal. Whether or not the SET gets stored as real data depends on the temporal relationship between its arrival time and the falling or rising edge of the clock. Thus, not all SETs become SEUs. A single SET can produce multiple transient current pulses at the output. Consequently, SETs in the logic can also invoke multiple bit upsets (MBU).

Looking at the phenomenon at transistor granularity: the charged particles interact with the silicon atoms causing excitation of atomic electrons. When a single heavy ion strikes the silicon, it loses its energy via the production of free electron-hole pairs, resulting in a dense ionized track in the local region (upper part of Figure 1.3). Protons and neutrons can cause a nuclear reaction when passing through the material (lower part of Figure 1.3). The recoil also produces ionization. The ionization generates a charge deposition

that can be modeled by a transient current pulse that can be interpreted as a signal in the circuit, causing an upset.

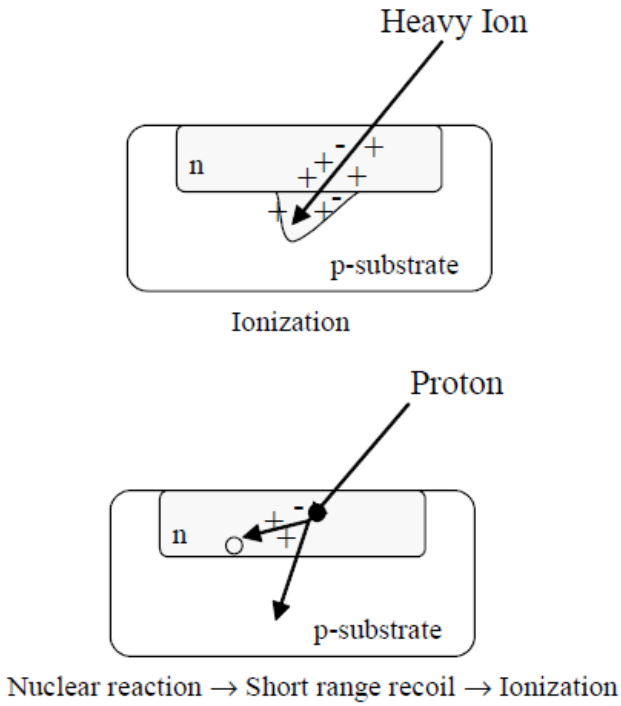


Figure 1.3: Charged particle striking the silicon surface

The detailed analysis of the effects of radiation particles in the bulk of a semiconductor is still a challenge. One of the difficulties resides in the prediction of the percentage of the electron-hole pairs collected in the area around the stored data. It's this percentage that determines the critical point at which the radiation-induced charge invokes an error in the stored data.

1.2 Reliability & Vulnerability assessment

One of the most important design constraints in modern microprocessors is its reliability. A serious challenge, when it comes to reliability, is the mitigation of the effects caused by transient faults. The mitigation process requires a great level of analysis that includes modeling of the faults and their effects. The modeling process allows the microprocessor architects to make tradeoffs in the processor's reliability. Data from recent studies, as mentioned in [1], shows the increasing number of multi-bit transient faults in SRAMs at recent technology nodes. A trend that is expected to be even more prominent as the desnity of transistors increases. These factors show the importance of the accurate fault modeling during the design process of a microprocessor.

Microprocessor vendors set a desired failure rate target during the design phase to prevent excessive failure of the device, caused by transient faults. Then the design for optimal power and performance takes place under this placed constraint. As mentioned, a significant amount of analysis is required to validate the design complies with the target constraint and at the same time improve the efficiency of protection in order to reduce unnecessary over-protection costs.

Some commonly used techniques are briefly explained below:

- *Accelerated testing:* as described in detail in [2] which described the experimental techniques which have been developed at IBM to determine the sensitivity of electronic circuits to cosmic rays at sea level. Accelerated testing to determine the chip Soft Error Rate (SER) is accomplished by placing VLSI (Very Large Scale Integration) chips in beams of nucleons with fluxes of about 10^6 nucleons/cm^{-2}-s, which accelerates the natural failure rate by a factor of 10^8.
- *Pre-Silicon Validation*: as described in [3] is generally performed at a chip, multi-chip, or system level. The objective of pre-silicon validation is to verify the correctness and sufficiency of the design. It typically requires modeling the complete system, where the model of the design under test may be RTL (Register Transfer Level), and other components of the system may be behavioral or bus functional models. The goal is to subject the DUT (design under test) to real-world-like input stimuli. The characteristics of pre-silicon validation are:
    - It validates design correctness.
    - It may be used for implementation or intent verification.
    - It does not rely on a design specification or golden reference model.
    - It uncovers unexpected system component interactions, inadequate or missing functionality in RTL.
    - Manually specifying expected results or output at a low level is difficult.
- *Statistical Fault Injection: Statistical Fault Injection* (SFI) is a type of Pre-Silicon Validation. Due to the often-impossible nature of being able to inject in a DUT -in a reasonable time- all the possible faults in all locations at each clock cycle, most of the results published in the literature are based on SFI [4]. During an SFI campaign, only a subset of the possible faults is injected. The selection of this subset is random with respect to the injection target and the injection cycle. The process of SFI, allows the designer to define the number of experiments taking place, according to the time that is available for the evaluation of the device. A common practice is to set a target population of injected faults that correspond to a high statistical confidence in the results.
- *Architectural Correct Execution* (ACE) analysis: The AVF [1] is measured by identifying whether a state in a system is required for architecturally correct execution. This state is called the ACE (architecturally-correct execution) state. A fault in this state results in an incorrect execution (an error). The other states are unACE states and they have no effect in correct execution. The process of defining those two types of state is called ACE analysis. The ACE analysis begins with the assumptions that all states are ACE and then gradually proves that some states are unACE. This results a computation of the upper bound estimate of the AVF of the system. ACE analysis -despite its limitations- is widely used and has immense practical value to the industry. The AVF of a hardware structure H containing B_H bits over a period of N cycles can be expressed as:

$$AVF_H = \frac{\sum_{n=1}^{N} [\text{ACE bits in H at cycle n}]}{B_H \times N}$$

Figure 1.4: AVF Formula

- *Post-Silicon Validation* **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.***:* Post-silicon validation is the last step in the development of a semiconductor integrated circuit. In contrary to pre-silicon validation, it occurs on actual devices running at native speed, in commercial, real-world system boards using a logic analyzer and assertion-based tools. It includes the validation part of a system after the first few silicon prototypes become available and before the product is released. While in the past most of the effort of post-silicon validation was dedicated to validating electrical aspects of the design, or diagnosing systematic manufacturing defects, today a growing portion of the effort focuses on functional system validation, which emerges as a need because traditional pre-silicon validation techniques can't verify the device adequately, due to the increasing complexity of digital systems. Because of that, a number of functional bugs that survive into manufactured silicon can only be detected by post-silicon validation methods. Those bugs are often system-level bugs and rare corner-case situations. Since these problems encompass many design modules, they are difficult to identify with pre-silicon tools. So, Post-Silicon validation, has a big advantage of high raw performance, because tests are executed directly on manufactured silicon but at the same time, it possesses several challenges to traditional validation methodologies, because of the limited internal observability and the difficulty of applying modifications to manufactured silicon chips. Two types of post-silicon validation are field-data collection and beam testing.

Apart from the aforementioned techniques linked with reliability and vulnerability assessment, the metrics that are used to express the reliability of a system are:

- *Failures in Time (FIT)* [7]**Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.**: The Failures in Time (FIT) rate of a device is the number of failures that can be expected in one billion (10^9) device-hours of operation. For example, 1000 devices for 1 million hours, or 1 million devices for 1000 hours each, or other combinations. For each structure in a device, a different FIT is computed using the formula in Figure 1.4. As one can see, the FIT of the structure is affected by three components: the FIT_{BIT} (or rawFIT) which is affected by the technology used, the AVF of the structure which is affected by the micro-architecture and the software and the number of bits in the structure. The FIT of the entire CPU is computed by adding the respective FITs of the structures (Figure 1.5).

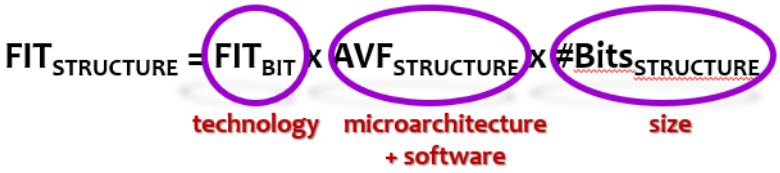


Figure 1.5: Formula of FIT of a structure

$$FIT_{CPU} = FIT_{S1} + FIT_{S2} + \ldots + FIT_{Sn}$$

Figure 1.6: Formula of FIT of the CPU

- *Mean Time to Failure (MTTF)* **Σφάλμα! Το αρχείο προέλευσης της αναφοράς δεν βρέθηκε.**: Mean Time to Failure (MTTF) is the length of time a device or other product is expected to last in operation. As a metric, MTTF represents how long a product can reasonably be expected to perform in the field based on specific testing. The relation between MTTF and FIT is: MTTF = 1/FIT and also 1 FIT ~= 114K years or 1-year MTTF ~= 114K FIT.

1.3 Micro-architecture level fault injection

Fault injection is one of the most common, among the aforementioned techniques, for the evaluation of a hardware structure's vulnerability. Fault injection calculates the number of errors or failures that occur according to a predetermined distribution of faults and is useful for evaluating the effectiveness of fault-tolerant techniques and the system's dependability. It can test fault detection, fault isolation, and the capability of reconfiguration or recovery of the system [9].

The Microarchitecture level in the pyramid of abstraction of a computer system is the level directly above the digital logic level. Its purpose is to implement in hardware the ISA (Instruction Set Architecture) level above it. The design of the microarchitecture level depends on the ISA being implemented, as well as the cost and performance goals of the computer [10]. It includes performance models, pipeline and hardware descriptions, and memory elements (arrays, flops).

The simulation tools can be grouped into three categories based on their abstraction level [11]:

- *RTL simulator:* which is the actual hardware design,
- *Microarchitecture level simulator:* which is a detailed cycle-accurate model of the microarchitecture and an,
- *Architectural Simulator:* which is a software-level emulation without hardware details.

The more abstract the model is, the faster it can be, which makes RTL models extremely slow contrary to the other two. Microarchitectural simulators have been widely used to evaluate the impact of different design choices in terms of performance, power, energy, and reliability as they offer the following advantages compares to RTL simulators:

- They have a higher throughput: faster by two or three orders of magnitude.
- They have a complete system stack modeling, allowing reliability estimation of the hardware layer (Hardware Vulnerability Factor), the software layer (Program Vulnerability factor) and the entire system stack (AVF).
- They are suitable for early reliability estimation since they are available earlier in the design chain.
- They support full system capabilities that can also include the operating system.
- They accurately model important array-based microarchitecture components: storage arrays which occupy the majority of a chip's area and thus largely determine the vulnerability to faults. For instance, on-chip caches, register files, buffers, and queues.
- They offer many advantages for accessing array-based structures, such as the ability to run large workloads.

The study's [11] findings, show however that the average difference on the vulnerability estimation between the two models is about 15% (μarch with the lowest ones for benchmarks), which translates to a difference of about 2 percentile units. However, as stated, some particular limitations don't allow for the modeling of the exact same workloads on the exact same hardware for the two methods. Table 1.1Figure 1.6 shows the average simulation throughput and time for a fault injection run for many

benchmarks, comparing the RTL and μarch (microarchitecture)(using GeFIN fault injector) solutions, respectively:

Table 1.1: Throughput of fault injection: RTL vs. μarch

Benchmark	Throughput		
	RTL	**GeFIN**	**Ratio**
FFT	7001 s/run	39.1 s/run	178.9
qsort	3157 s/run	23.9 s/run	131.8
cAES	413 s/run	30.7 s/run	30.7
sha	3421 s/run	8 s/run	427.2
stringsearch	60 s/run	2.8 s/run	21.39
susan corners	1019 s/run	3.2 s/run	315.5
susan edges	874 s/run	3.6 s/run	242.1
susan smooth	893 s/run	3.7 s/run	241.4
Average			198.6

1.3.1 Gem5 simulator, fault-injector, and the fault mask generator

The basic blocks that are used in this thesis for the fault injection campaigns in the microarchitectural level are:

- The Gem5 simulator
- The GeFIN (Gem5-based Fault Injector) Fault Injector [12]
- And an improved version of a pre-existing Fault Mask Generator [12] used to create fault masks for the fault injection campaigns through GeFIN.

The tools described are as follows:

- *The Gem5 simulator:* is one of the most widely adopted simulators in the computer architecture community. Gem5 is cycle-accurate (thus can allow per cycle fault injection at any modeled hardware component), publicly available and regularly maintained today by developers. Gem5's popularity is mainly due to its accurate support of important ISAs, its detailed and configurable model of the memory system, and check-pointing support. Lastly, Gem5 supports different reliability studies on different ISAs, like ARM and x86 among others and has a fully configurable CPU model (pipeline depth and widths, structures sizes and organizations, etc.) to facilitate a study's need. Figure 1.7 shows some of the characteristics of the Gem5 simulator, while Figure 1.8 shows how a system is built using modules to allow high level of flexibility.

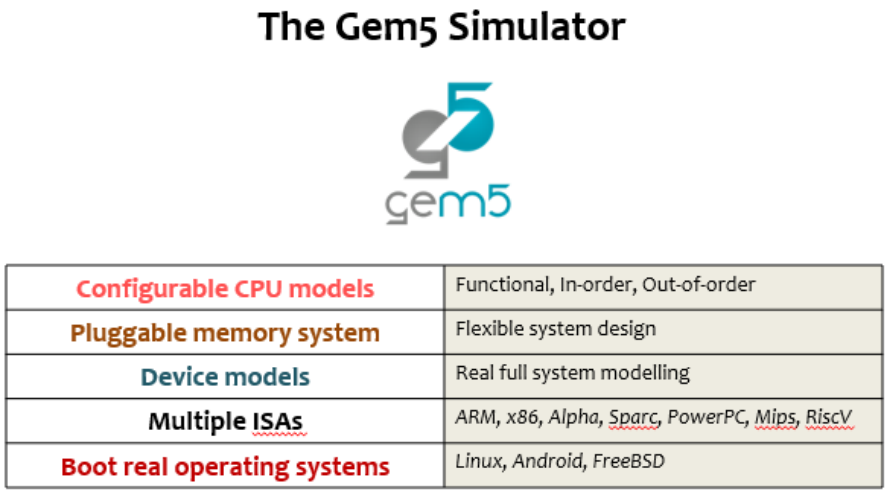


Figure 1.7: Gem5 simulator basic characteristics

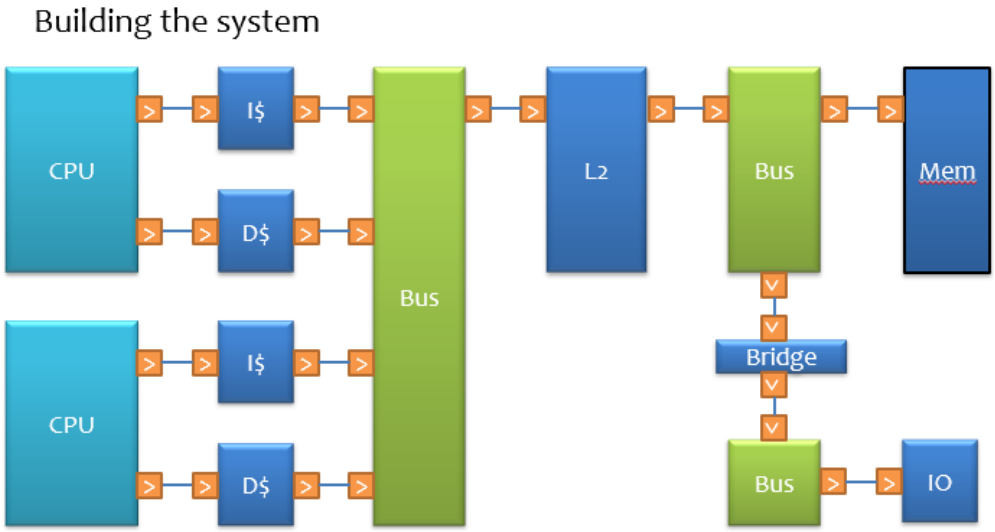


Figure 1.8: Building a highly flexible system in the Gem5 simulator

- *The GeFIN (Gem5-based Fault Injector)*: GeFIN is the micro-architecture level fault injection tool, build on top of Gem5 simulator, supporting the two popular ISAs (x86 and ARM). It consists of three modules: a fault mask generator, an injection campaign controller, and a parser of the logged information. Through GeFIN, a study in the full range of fault models: transient, intermittent, permanent, and multi-bit is possible. Faults can be injected in different bits of the same entry of a hardware structure, different entries of a structure, different hardware structures simultaneously, and all combinations of the above. The GeFIN injector classifies the outcomes of each fault simulation based on the impact of the fault on the simulated system. The fault classification is fully configurable, and the classes of the fault effects can be modified by a user by changing the parser of the injection logging information. Five classes are used for the fault effects classification: Masked, Silent Data Corruption (SDC), Timeout, Crash, and Assert (details in Chapter 4: Experimental Setup).
- *The Fault Mask Generator:* The Fault Mask Generator module produces the fault masks that are used during the injection campaign. It can produce (given user-defined parameters) a random set of fault masks for any type of fault (transient,

intermittent, permanent or a mixture of the three), for the entire simulation time of the benchmark. A fault mask contains information about: the processors core where the fault is going to be injected which can also be used in multicore architectures, the microarchitecture structure on which the fault will be injected, the exact bit position of the injection, the exact simulation cycle or exact instruction on which injection happens, the type of fault and the population of fault (single or multiple). This thesis includes two different improved implementations of the Fault Mask Generator, with the addition of the injection of adjacent multi-bit faults and the capability for different levels of interleaving. The implementations of the Fault Mask Generator are discussed in detail in Chapter 3.

Figure 1.9 shows a simplified diagram of the main components of a typical Fault Injection Campaign at the Microarchitectural level.

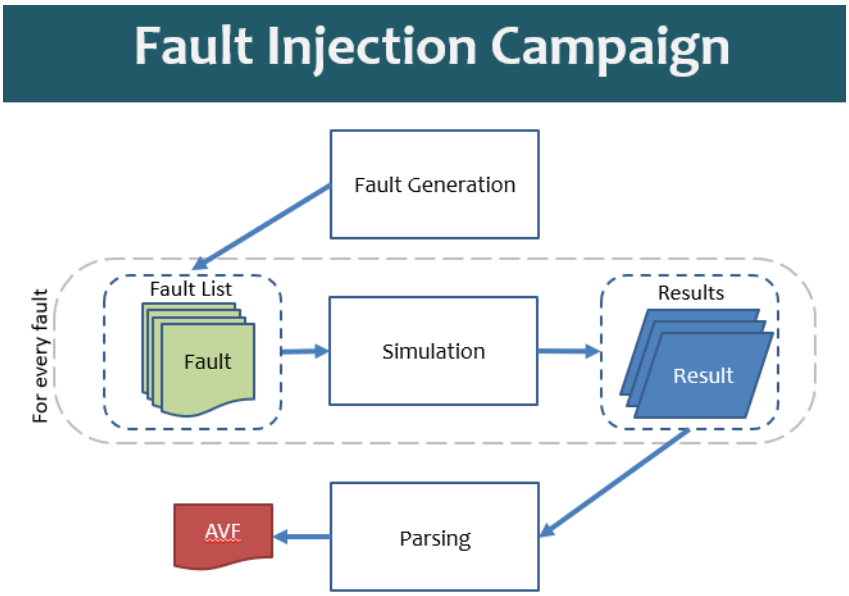


Figure 1.9: Typical steps in a Fault Injection Campaign

1.4 Thesis Structure

While Chapter 1 (Introduction) covered the basics of: faults in integrated circuits and what causes them, reliability & vulnerability methods, pros & cons, etc., micro-architecture level fault injection and specific tools, the rest of the thesis is organized as follows:

- *Chapter 2* further expands the background, related work and the problem examined, by focusing mainly on multi-bit upsets, their characteristics and their types but also discusses about protection methods including interleaving.
- *Chapter 3* presents two different implementations of a Fault Mask Generator, which include support for multi-bit adjacent faults, and interleaving. It shows and explains some parts of their code and their internal operation and some typical use and corner cases.
- *Chapter 4* presents the experimental setup for the fault injection campaign showing details for the system used, the workload, the number of faults injected, etc.

- *Chapter 5* presents and discusses the results from the fault-injection experiments, presenting stats about the vulnerability of caches, register files, etc., the AVFs of each component and also their reliability (failure rates) along with the reliability of the whole system, all for different technology nodes.
- *Conclusion* includes some thoughts obtained from the thesis and suggests future work.
- *Appendices A & B* include the code for the two implementations of the Fault Mask Generator.

2. BACKGROUND, RELATED WORK AND PROBLEM DESCRIPTION

2.1 General

In this chapter we follow the conversation from Chapter 1, looking in a closer manner about the problem we examine in this thesis. Specifically, we examine the nature of the multi-bit faults in integrated circuits, how and how often they appear, their types and effects. Then, we examine some common protection methods, such as error correction codes and interleaving and finally, we establish the problem that we try to confront considering multi-bit upsets on a microarchitecture level modeling of a system.

2.2 Multi-Bit Upsets

On the previous chapter we presented that aside from the single-bit faults (SBFs), spatial multi-bit faults (sMBFs) can also occur when a single strike affects multiple bits at the same time instance, in contrast to temporal multi-bit faults (tMBFs) in which multiple strikes (spaced in time) lead to multiple flipped bits the next time a set of bits is read. We also explained how a single SET (single transient effect) can produce multiple transient current pulses at the output; SETs in the circuit logic can invoke multiple bit upsets (MBU). An MBU can occur when a single charged particle traveling through the integrated circuit at a shallow angle, nearly parallel the surface of the die, simultaneously strikes two sensitive junctions by direct ionization or nuclear recoil.

There are three types of MBUs. The first one occurs when a single particle hits two adjacent sensitive nodes, located in two distinct memory cells. This event is classified as a second-order effect (first-order effects are the single-bit upsets). This type of MBU can be avoided by specific placement, for example, memory cells of the same register or memory data can be placed further away from each other to avoid the same charged particle strike, affect two or more adjacent cells of the same data structure. The second type of MBU occurs when a single particle, strikes two adjacent sensitive nodes located in the same memory cell. This event is classified as a third-order effect. The probability of such a multiple node strike can be minimized in circuit design by taking care in the psychical layout to separate critical node junctions by large distances, and by aligning such junctions so that the area of each junction, as viewed from other, is minimized. The third type of MBU occurs when multiple particles strike multiple sensitive nodes in the silicon, provoking upsets in multiple memory cells. This event can be analyzed like a group of SEUs, and it will represent the same immunity characteristics. The majority of multiple upsets located in adjacent cells are provoked by a single particle [6].

When it comes to exactly how often we can see multiple-bit faults in a structure, study [15] shows that in a 180 nm process for an SRAM device, fewer than 0.6% of the faults, affected more than one bit along a word line, while on the contrary as the scaling of the dimensions continues, we can see that for a 22 nm manufacturing process, 3.6% of all faults affected multiple bits along a word line. This increase in multi-bit fault rate is projected to continue despite the introduction of technologies such as FinFET transistors that reduce the overall rate of transient faults, but do not slow the trend towards multi-bit faults [1]. Table 2.1 (taken from study [15]) illustrates the effect we described as the technology node lowers.

Table 2.1: Ratio of multi-bit faults to total number of faults in percent

Design Rule (nm)	Total	Bit width of multi-bit fault			
		2	3	4-8	>8
180	0.5	0.5	0.0	0.0	<0.1
130	1.2	0.8	0.2	0.2	<0.1
90	1.9	1.5	0.2	0.2	<0.1
65	2.4	2.1	0.1	0.0	<0.1
45	2.3	1.9	0.2	0.1	<0.1
32	3.1	2.6	0.2	0.3	<0.1
22	3.9	3.0	0.2	0.3	0.1

Study [15] which was conducted using a simulator to predict the impact of devices scaling from 250 nm to 22 nm, on the soft error rate of SRAM, also showed that: (i) soft error rates per device in SRAMs will increase x6-7 from 130 nm to 22 nm process (iii) as SRAM is scaled down to a smaller size, soft-error is dominated more significantly by low-energy neutrons and (iii) that the area affected by one-nuclear reaction spreads over 1 M bits and the bit multiplicity of a Multiple Cell Upset (MCU - simultaneous errors in more than one memory cell, induced by a single event) becomes as high as 100 bits and more.

Study [1] performed an ACE analysis to measure AVFs for multi-bit faults. The study didn't observe any correlation between single-bit AVFs and multi-bit AVFs, showing that the MB-AVFs are not derivable analytically from SB-AVFs. MB-AVF analysis can reduce Silent Data Corruption (SDC) estimates relative to approximating MB-AVFs from SB-AVFs. MB-AVFs can vary independently of SB-AVFs and must be measured through simulation. Also, the study showed that MB-AVFs range from one to M times SB-AVFs, where M is the number of bits in the multi-bit fault pattern in question, and that larger fault modes have larger MB-AVFs, offsetting to some extend the reduced rate of these larger faults, as shown in Table 2.1.

Paper [16] presented a work for multiple bit errors in SRAM memories at 130 and 90 nm technology nodes using beam testing of neutron and alpha particles. The study focuses on the dependencies of multi-bit faults on voltage, stored data patterns in SRAM and the probabilities of 2, 3, 4, and 5-bit errors. Figure 2.1 shows the multi-fault probability vs. voltage (Vcc) for 2, 3, 4 and 5 adjacent bits, for a checkerboard data pattern. The probability numbers are referenced to the number of failure events. A 2E-2 probability for double bits, for example, indicates that 2% of all failure events were double bits (adjacent bits failing within a single readout). The voltage dependency for multi-bits is small for the 130nm SRAM, displaying only a small increase at low voltages. Also, the study showed similar results for different data patterns and directions of the neutron beam incidence.

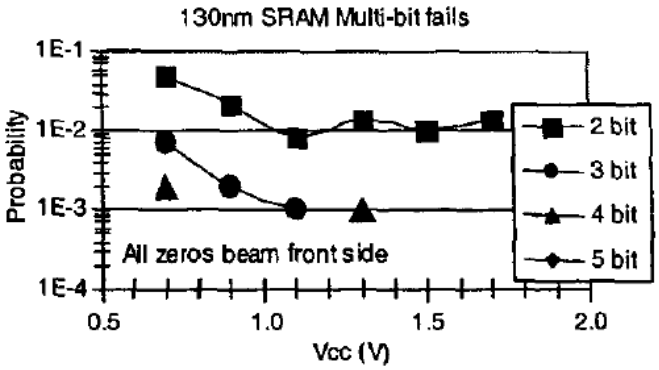


Figure 2.1: Probability of multi-bit faults for different number of bits against Voltage (Vcc) for a 130nm SRAM

The results for the 90nm SRAM from the same study appear in Figure 2.2. The double bit error rate probability is slightly higher, by about 2X, but we don't see the increase in lower voltages like in the 130 nm SRAM. On the other hand, a similar independence to data pattern or beam orientation is seen for the improved technology. While the probability for double bit errors is higher in the 90 nm nodes, there is a crossover for 3, 4 and 5 bit fails, with 130 nm showing higher probabilities compared to 90 nm (Figure 2.3). In fact, there was not a single 5-bit event for the 90 nm SRAM, while 4 such events were recorded for the 130 nm one. The study explains that this could be related to the fact that for those extremely small cells, the dimension of the burst of charge generated by the neutron event is larger than the cell dimension, leading to the distribution of charge among more cells. The transistor at 90 nm (the experiment tested 6-transistor CMOS SRAMs) also has a faster response and perhaps was able to neutralize better the injected charge.

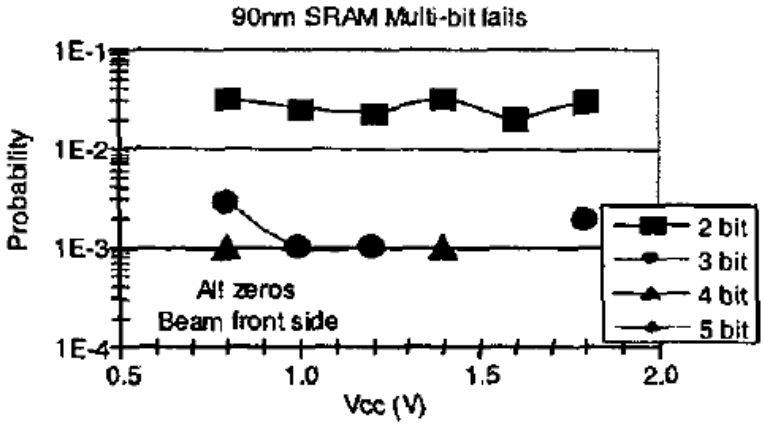


Figure 2.2: Probability of multi-bit faults for different number of bits against Voltage (Vcc) for a 90nm SRAM

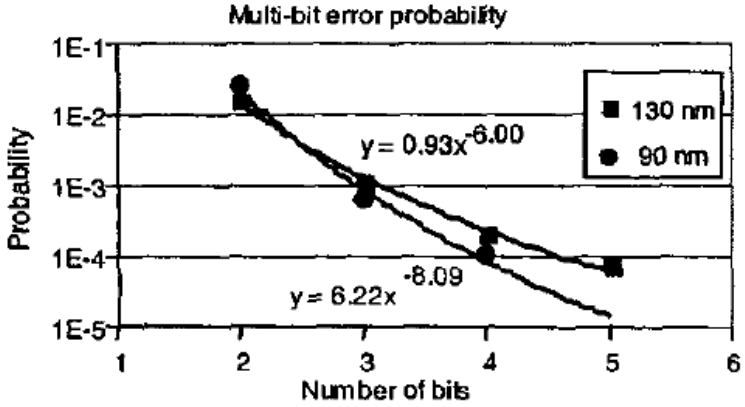


Figure 2.3: Multi-bit probability comparison between 90 & 130 nm SRAMs

The study also showed some common patterns for double and triple bit faults (Figure 2.4 and Figure 2.5 respectively). For double bit faults, we can see an expansion of fault between two columns, two rows or diagonally while for triple bit faults we can also see "L" patterns or even some adjacent faults with no clear geometry (Figure 2.6).

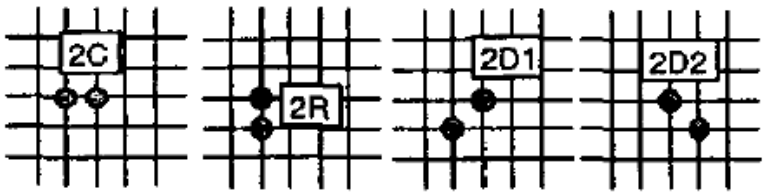


Figure 2.4: Common double bit fault patterns

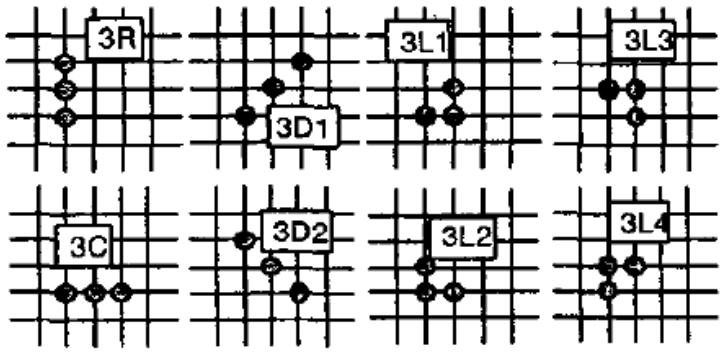


Figure 2.5: Common triple bit fault patterns

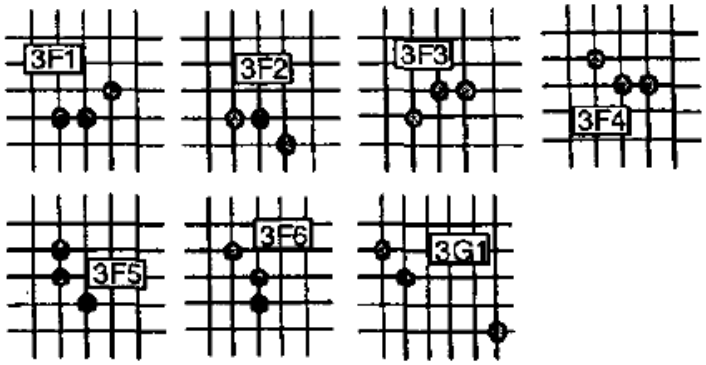


Figure 2.6: Triple bit fault patterns with no clear geometry

2.3 Protection Methods & Interleaving

Many techniques can be used by an architect in order to protect against multi-bit faults while designing ICs. Techniques such as redundant multi-threading can detect most multi-bit faults. Other techniques like stronger error-correcting codes (ECCs) such as double-error correction / triple-error detection (DEC-TED) can detect and correct multi-bit faults, while cyclic redundancy codes (CRCs) can provide robust multi-bit error detection. Also, bit interleaving is commonly used in conjunction with ECC or parity to minimize the error rate contribution of multi-bit errors. It refers to a memory layout architecture in which psychically adjacent bits belong to different logic words. The result is that from an error detection and correction standpoint, two adjacent failing bits appear as single bit errors rather than as a double bit error in the logic word. Logical interleaving increases the number of ECC words per data word and assigns physically adjacent bits to different ECC words. This increases the area overhead of ECC because each data word contains multiple ECCs. Physical interleaving, on the other hand, assigns physically adjacent bits to different data words, thus ensuring that bits are protected by different ECCs. Interleaving can be done between two data words (x2 interleaving), four data words (x4 interleaving), and so on. Bit interleaving rules are often defined as the minimum physical distance separating two bits belonging to the same logic word. The quantification of their effectiveness requires a detailed understanding of the multi-bit failure probabilities we showed earlier.

Study [1] showed that logical interleaving, in which each word is split into multiple interleaving check words, can have MB-AVFs many times lower than that of a physical interleaving, in which each data word is interleaved with other data words. Also, this study depicts that logical interleaving consistently yields MB-AVFs very close to the theoretical limit. This is because bits in the same cache line are often written and read close together in time. Therefore, these bits are more likely to be more ACE -which is necessary for the correct program execution- or both unACE, than bits from different cache lines. This property is referred to as ACE locality. Furthermore, the study showed than the MB-AVF for physical interleaving varies substantially based on the workload (benchmarks) and the style of physical interleaving (way or index physical interleaving). For the workloads used, way-physical interleaving had a 56% higher MB-AVF and index-physical interleaving had a 65% higher MB-AVF than logical interleaving for an L1 cache structure because a cache with logical interleaving shows higher ACE locality than a cache with physical interleaving. So, increasing this property in a structure, reduces the MB-AVF. As far as the fault mode is concerned the same study showed, that MB-AVF increases for larger fault modes, because a larger fault group has a higher

probability of containing an ACE bit. Specifically, the 4x1 fault mode MB-AVF was 2.74x SB-AVF with a parity protection. Also, the difference between parity and ECCs was underlined with the finding that the 8x1 fault mode MB-AVF with a single-error correction / double-error detection (SEC-DED) ECC is 2.74x SB-AVF, the same as a 4x1 fault mode MB-AVF with parity. As large multi-bit faults become more common, parity may have a detection advantage over ECC. An implication of this could make parity a better choice than ECC in systems in which detection is the primary concern, such as systems with higher-level rollback/recovery mechanisms. Also, future systems may be better off decoupling detection from correction to meet reliability targets.

However, all these techniques have power, area and performance costs. For example, implementing a DEC-TED ECC on a 128-bit data word requires 17 check bits, which translates to a 13% overhead, whereas implementing a SEC-DED ECC requires only 9 check bits -a 7% overhead. DEC-TED ECC also requires a deeper XOR tree to decode, resulting in increased latency. Similarly, a physically interleaved array requires multiple columns to be activated and multiplexed to read out a single bit, resulting in increased dynamic power consumption, an area overhead, and latency. Therefore, it is up to the computer architect to decide on the required level of multi-bit fault protection. Excessive protection increases power and area and reduces performance unnecessarily, while inadequate protection results in an unreliable design. But even though the use of parity or ECC is crucial, a large enough multi-bit fault can defeat the protection and cause an error like an SDC.

2.4 Problem Description

Estimating multi-bit architectural vulnerability factors allows architects to model the impact of multi-bit faults at design time and to deploy appropriate strategies to protect their designs. Also, as we saw in section 2.1 multi-bit faults seems to play an ever-important role in ICs of different kinds while the feature sizes shrink. Unfortunately, while multi-bit fault rates can be measured via accelerated testing, methods to allow architects to quantify the impact of multi-bit faults are lacking. The main subject of this thesis revolves around the experimental multi-bit fault injection in microarchitectural level in different structures of a core under test in order to evaluate the effect of multi-bit faults in each structure. For that, we use the Gem5 simulator to create the system under evaluation, the GeFIN framework to inject the faults (see Chapter 1) and an enhanced version of the GeFIN fault mask generator suited to exploit the areas we examined on this chapter (see Chapter 3 for an in-depth analysis), namely: multi-bit fault injection in adjacent areas of a structure, and the ability to enable interleaving rules.

The multi-bit fault injection is based around the idea of a cluster, as seen in Study [15]. A generated fault mask contains multiple faults to be injected to the system. It is the cluster that creates a sense of vicinity between the injected faults. By creating for example 3 faults to be injected inside a 3x3 cluster randomly put inside the structure, we know that those bits are placed in adjacent areas, much like is done in real beam testing where the multi-bit faults are observed in adjacent areas (Figures 2.4 - 2.6). Figure 2.7 shows some examples of MCUs. The MCU pattern is classified into three basic categories: a single line across Bit Line (BL)(category ‘b’), a single line along Word Line (WL)(category ‘w’) and a cluster (an MCU that has two or more bits along with both BL and WL directions - category ‘c’). Study [15] proposes an MCU code that can be almost uniquely relevant to a physical address pattern in an MCU. The code is: C_N_1_N_2_N_3_N_4 where C is the category (b/w/c), N_1 is the MCU size (N_3 x N_4), N_2 is the bit multiplicity in an MCU, N_3 is the width in the BL direction, N_4 is the width in the WL direction and P is the parity. This is not the code we use in our implementation of the fault mask

generator. We use parameters (the fault mask parameters file) to define the cluster size (rows X columns) and the number of parallel faults inside it (fault population), among other parameters for the fault injection (see Chapter 3 for detailed information).

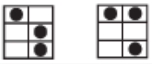
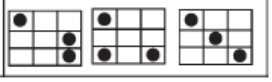
Category	Code	Error bit pattern example
On single BL	B_2_2_2_1_ any parity	
On single WL	W_2_2_1_2_ any parity	
Cluster	C_4_2_2_2_ any parity	
	C_6_2_2_3_ any parity	
	C_6_2_3_2_ any parity	
	C_6_3_3_2_ any parity	
	C_8_2_4_2_ any parity	
	C_9_3_3_3_ any parity	

Figure 2.7: Examples of MCU categories and codes from [15]

The second improvement of the fault mask generator is the addition of an interleaving scheme. By adding an interleaving scheme of some degree (powers of 2, or 1 for no interleaving) we change the selected (rowID, columnID) for any generated fault for injection inside the fault mask (under all cases of injection i.e. single-bit injection, multi-bit random injection, multi-bit adjacent injection), to correspond to the (rowID, columnID) of a structure with an interleaving degree of N. So, the fault injection we perform corresponds with the real values where the faults are to be inserted, if the structure elements were shuffled (a.k.a. interleaving). The interleaving is done programmatically with the help of some rules and is explained in detail in Chapter 3. But the main logic is that for an interleaving degree of N, the first N words of a structure are shuffled, followed by the next N etc. This is done if we consider the structure as a sum of sub-arrays with size NxN. The shuffle happens inside those sub-arrays, so bits in adjacent areas, might not be adjacent on an interleaved memory. For the shuffle to happen, each “column” of the sub-array is shifted with an offset equal to the number of the column (C0 = 0 offset, C1 = 1 offset etc.). The offsets for each column could be different (e.g. C0 = 0, C1 = 2) to avoid some cells that are still adjacent after interleaving, but this is not done in our implementation. Figure 2.8 shows the results of an interleaving process for a 4x4 structure with an interleaving scheme of 2 and 4.

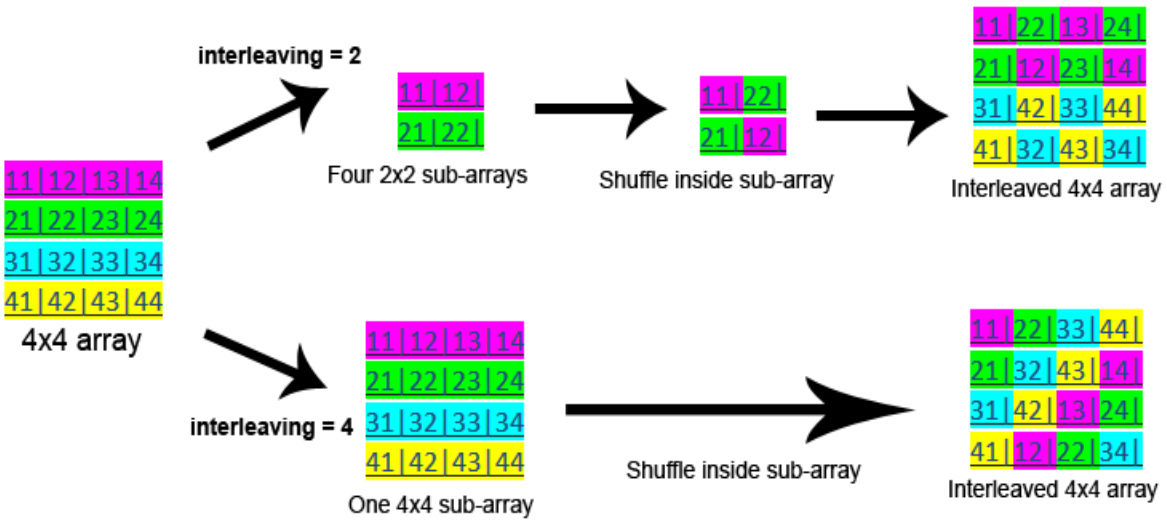


Figure 2.8: Interleaving shuffle process in a 4x4 array for an interleaving degree of 2 and 4

3. MODELLING

In this chapter, we present two different implementations of the fault mask generator used for the fault injection campaign. The fault mask generator is the first step of the entire campaign. As explained in Chapter 1, the purpose of it is to create a user-defined number of fault masks, each with some user-defined characteristics, for input to the fault injection tool GeFIN that works in association with the Gem5 simulator, which simulates the system under evaluation. An already existing fault mask generator written in Perl Scripting Language was the basis for the reconstruction of it and the addition of some new features such as, the ability to create adjacent multi-bit faults and interleaving.

Two different implementations of the fault mask generator were created, one in C++ and the other in Python, each with different internal functionality and outputs. Both of them had the same requirements and are expected to work in a similar manner for the same purposes. But due to the different implementation of each, it is practical to analyze each one in a different section, namely 3.3 and 3.4.

3.1 Requirements for the fault mask generator

As stated, the first step of the process was the re-build of a pre-existing fault mask generator written in Perl who had some specific characteristics, that the newer version had to follow. Also, some additions to facilitate new functionality had to be done. The requirements were:

- The fault mask generator has an input of a fault mask parameter, text file, which includes the basic user-defined parameters needed for each fault campaign. The very first version of the fault mask parameters file included information about: the target processor core (for multicore architectures), the microarchitecture structure on which the fault will be injected, the bit position of the injection, the exact simulation cycle or exact instruction on which injection happens, the type of the fault and the population of faults (single or multiple) among others. The fault mask parameters file specification changed during the implementation process and got additional parameters. Thus, it will be analyzed in detail in the following section. The Python implementation does not use a fault mask parameters file but instead accepts the parameters from the command line.
- The fault mask generator read the input file, analyzed the parameters and produced a user-defined amount of transient, intermittent, permanent or mixed fault masks for a specific core (or cores) and a specific module (or modules), each including a user-defined number of faults. As a result, the form of the output fault masks will be discussed separately in the two sections 3.3 and 3.4.
  - Aside from the main functionally of the generator, two feature improvements were made, namely:
    - The addition of multi-bit adjacent faults. The generator creates some parallel faults (more than 1 faults) to be added in adjacent areas of the structure.
    - The addition of interleaving. The generator assumes the system under evaluation uses an interleaving scheme for protection against multi-bit faults and changes the positions of the injected faults (rows and column) to correspond to the true rows and columns after some user-defined degree of interleaving.

3.2 Implementation A (Fault Mask Generator in C++)

3.2.1 The Fault Mask Parameters File

The fault mask generator has an internal operation of reading and processing the user-defined parameters defined in a fault mask parameters text file for the generation of the fault masks. The fault mask parameters file has the structure shown in Figure 3.1 and defines the following parameters for the fault masks generation:

- *Version:* defines the name of the system under evaluation for example “arm_small_susan_c-cortex_a15_base”.
- *Population:* defines the number of fault masks (independent files) generated per fault category (transient, intermittent, permanent, mix) for each generator execution. For example, a value of 6, will create six fault mask files under each category.
- *Module ID:* defines the module or modules where the injection will take place. The form of each module is as follows: “xxxx yyyy zzzz/b --> xxxx:memory(0)/core(1), yyyy:module ID, zzzz:sub-array ID” and each module (in case of many) is separated by a semicolon (;). For example, 256;272.
- *Rows:* defines the number of Rows of the structure (where the fault(s) will be injected).
- *Columns:* defines the number of Columns of the structure (where the fault(s) will be injected).
- *Bit vector size:* defines the size of the bit vector. It has small practical value to the current version of the generator and was included in correspondence with an older version. It’s only used in the statistical safe sample calculation. Its default value is 32.
- *Offset range:* Defines the offset range. It has no practical value to the current version of the generator and was included in correspondence with an older version of the generator. Its default value is 0.
- *Parallel faults:* if a multi-bit fault injection is required, then the number of parallel faults must be greater than 1. Then each fault mask file will include more than 1 “columns” each representing a fault. The number of parallel faults must be smaller than the multiply of rows * columns. It’s also must be smaller than the multiply of cluster_rows * cluster_columns in case of multi-bit adjacent faults, where we use a cluster to simulate vicinity.
- *Multi-Bit Cluster Rows:* Defines the number of rows in a cluster, if an injection of multi-bit adjacent faults is required. If zero, then no adjacent multi-bit injection is done.
- *Multi-Bit Cluster Columns:* Defines the number of columns in a cluster, if an injection of multi-bit adjacent faults is required. If zero, then no adjacent multi-bit injection is done.
- *Interleaving scheme:* defines the interleaving degree, if interleaving is desired. Allowed values are any power of 2 (2, 4, 8, etc.) and they must be smaller than the number of rows (similarly for the number of columns). If 1, then no interleaving occurs.

- *Total simulation time:* defines the total simulation time and bounds the activation parameter. It is applicable only for transient and intermittent faults (it's value always appears as 1 in the fault masks, for permanent faults, no matter what we put here).
- *Duration:* defines the total duration. For permanent and transient faults the value is ignored.

The fault mask parameters file includes all the parameters in the first space of each line. Any other line starts with a "#" sign and is ignored by the generator. Mixed fault mask files are only generated if the number of parallel faults is greater than one.

```
1 ################################################
2 #----                                        ----#
3 #---- Fault-mask Generator INPUT PARAMETERS ----#
4 #----                                        ----#
5 ################################################
6 #
7 #.:|Version:
8 arm_small_susan_c-cortex_a15_base
9 #
10 #.:|Population:
11 6
12 #
13 #.:|Core ID (Note: generate fault_mask for the specific coreID):
14 0
15 #
16 #.:|Module ID (Note: multiple modules separate with ';'):
17 #.:| xxxx yyyy zzzz/b --> xxxx:memory(0)/core(1), yyyy:module ID, zzzz:sub-array ID
18 #.:| Physical_register_file:256 -- Branch_predictor (Bimodal:352, Meta:353, Two-Level:354, iBTB/data:355, iBTB/tag:356, iBTB/LRU:357, iBTB/valid:358
19 #.:| dBTB/data:359, dBTB/tag:360, dBTB/LRU:361, dBTB/valid:362, RAS:363)
20 #.:| LSQ (data:272, virtaddr:273, addrvalid:274, datavalid:275, bytemask:276)
21 256;352
22 #
23 #.:|Rows:
24 128
25 #
26 #.:|Columns:
27 128
28 #.:|Bit vector size:
29 32
30 #
31 #.:|Offset range:
32 0
33 #
34 #.:|Parallel faults (must be less or equal to cluster size rows*columns in case of multi-bit faults):
35 5
36 #
37 #.:|Multi-Bit Fault Cluster Rows (leave 0 if no multi-bit fault injection is desired):
38 0
39 #
40 #.:|Multi-Bit Fault Cluster Columns (leave 0 if no multi-bit fault injection is desired):
41 0

42 #
43 #.:|Interleaving Scheme (leave 1 if no interleaving is required, else any power of 2 is allowed smaller than Number of Rows (or Number of Columns):
44 16
45 #
46 #.:| Total simulation time (Note: Bounds the activation parameter. Only for transient and intermittent faults):
47 1892025
48 #
49 #.:|Duration (NOTE: For permanent and transient faults equals to '1'):
50 #.:| Warning: "Duration" MUST be smaller than request pool size of FaultRequest
51 100000
52 #
53 #.......:|  NOTE: MIX fault masks are generated only when parallel faults parameter differs from '1'  |:......#
54 #
55 ################################################
56 #----                                        ----#
57 #                     END OF FILE              #
58 #----                                        ----#
59 ################################################
60
```

Figure 3.1: The fault mask parameters file

3.2.2 Technical details (the fault mask generator source file)

In this section, we present the internal operation of the fault mask generator by analyzing different parts of the source code. It serves as an extra description alongside the comments in the code. The first step inside the main function of the C++ source code is the reading of the input file parameters and their storage in a struct named "parameters" which includes fields that corresponds to all the parameters in the text file. The program creates a generator.log file that will include all the basic information during

the program life, mainly for debugging purposes. The generator.log file is empty unless the variable debug equals 1.

The program opens the fault_mask_parameter.txt file using an ifstream (input) object called FAULT_MASK_PARAMETERS_FILE. In case of an error, the program exits. A similar try-catch block to prevent errors is used in many cases throughout the program. A string variable called “line” is used to read -in a serial-manner- all the lines of the text file, one by one until there is nothing left to read. Using a *while (getline(FAULT_MASK_PARAMETERS_FILE, line)),* we read all lines of the text file. The first space of each line is checked (point of [0, 1]). A “#” character in this point indicates that this line has no parameters and we move on to the next one (next iteration of the while loop). For the lines that include a parameter, a case structure is used, so we can read and store them in a serial manner. Case 1 reads the version parameter and stores it in the “version” variable of the “fault_mask_parameter” struct, case 2 reads the number of faults masks parameter and stores it in the “faults” variable of the “fault_mask_parameter” struct, etc. Wherever required, the program performs sanity checks to avoid usage of invalid arguments. For example, case 2, checks for a “0” value which is of course invalid.

One interesting case during the parameters read process, is the case of the moduleId parameter, because in this one we can have more than one moduleIds separated by a semicolon, so it is crucial to store each one of them. Since we don’t know the number of moduleIds included, we create a dynamic data structure such as a vector, which saves all the moduleId’s. First, we save the whole line to the string type “moduleId” variable of the fault_mask_parameter struct. If no moduleId is included, the program exits. Else, we open a do-while loop. While a “found” variable (which indicates the existence of a semicolon in the “moduleId” variable) is not equal to -1 (a case that indicates that there is not a semicolon left in the line) we open a for-loop. The for-loop is executed from 0 to found – 1. So, in a case of a line with “353;257” we try to isolate the first number and add it to the vector. This happens by reading its character one by one inside the for loop and saving it in an incrementing position of a string type “temp” variable. When we reach the position of the semicolon, we leave the for-loop. Then the temp value is added to the “multiple_module” vector and the “moduleId” parameter of the struct, now includes all that exists to the line after the semicolon (e.g. 257). We re-enter the do-while loop and a new value for found is calculated. This time found is equal to -1, because no semicolon exists. In this case, we add the number (e.g. 257) to the next position of the “multiple_module” vector through the “temp” string and we empty the “moduleId” struct variable. We exit the do-while loop. Then we re-enter the values of the moduleIds inside the “multiple_module” vector, onto the “moduleId” string in a form like moduleId1_moduleId2_...._moduleIdX (e.g. 352_257), because this is the way we wish to have them for the output.

We mentioned that, during the reading process, we define some valid ranges for certain cases. An interesting case appears while we read the Parellel Faults value. This value can’t be 0 but at the same time, we can’t have only one fault if two or more moduleIds exist. Also, in the case of the multi-bit cluster parameters (multi-bit cluster row and multi-bit cluster column), the size of the cluster can’t be bigger than the actual size of the structure, but also the number of parallel faults injected can’t be bigger than the multiple of cluster rows * cluster columns, as mentioned.

After the reading and the storage of the parameters internally, the program creates the main directory consisting of the “version” variable of the fault_mask_parameter struct (for example “arm_small_susan_c-cortex_a15_base”). The directory is created in the folder where the program is executed. Inside the main directory a sub-directory is

created with the name of the moduleId or moduleIds (for example a 353_257 folder). Inside that folder, four more folders are created for each type of fault, namely a permanent, a transient, an intermittent, and a mix folder.

On the next part of the code, we check if multi-bit adjacent fault injection is enabled. If both parameters, cluster rows and cluster columns are different than 0, then a variable multi_bit_enabled is set to 1, indicating exactly that. In that case, a while loop is opened running for the number of fault mask files, that we wish to create inside each fault type folder. A random starting point (rowId, columnId) for the cluster is calculated using a typical random number generation function called, random_num. The random_num function uses an argument as an “upper_limit” and by using the rand() function of C++, it calculates a random value using the formula: random = rand() % (upper_limit). To verify that the cluster won’t exceed the limits of the structure we generate random starting points until the whole cluster fits inside the structure. The condition we check is: random generated row + cluster rows <= structure rows AND random generated column + cluster columns <= structure columns. We also create random values for the type of the fault, the selected module and the selected activation.

In order to make sure, that we won’t inject a fault with the same characteristics as before on the same file, we check the randomly selected parameters with a checksum function which checks if we had the same combination of parameters before on the same fault mask file. The checksum does that by checking each one of the newly selected random parameters with all the previously selected parameters (each one stored inside a vector). For example, the “checksum_row” vector includes all the rows selected before in the same fault mask file, the “checksum_column all the columns, etc. In case the combination of parameters wasn’t selected before, the checksum function adds the new parameters to the vectors and returns success. Otherwise, the function returns a 0, and the main function re-enters a while loop to pick new random values for the parameters.

Next, we generate the permanent multi-bit fault mask. We enter a loop for all the number of parallel faults we wish to inject. Each fault is a new “column” in the fault mask file. This time, while we have the starting points of our clusters, we have to find a random spot inside the cluster to inject the fault. We now use a random_num_multi function which has both an upper limit and a lower limit. In the case of the row, the function runs, with the randomly “selected_row” as the lower limit and the “selected_row + fault_mask_parameter.multi_bit_rowId – 1” as the upper limit. This way, we define a random row inside the cluster. The same goes for columns. This more sophisticated random generator now uses a different formula to calculate the random number: random = rand() % (upper_limit-lower_limit+1) + lower_limit. We also have to check if this combination has already been used before, this time in the smaller scope of a cluster, with a modified version of the checksum function, called the checksum_multi. This function works in the same manner, but uses different vectors for the storage of the randomly selected rows and columns inside the cluster. As the rows and columns are the only things that change under the cluster scope, these two are the only ones we need to check. The final step before writing the randomly generated values to a file is to check if interleaving is selected. Like we mentioned a value of 1 in the interleaving parameter, indicates no interleaving of memory and any other value N indicates an interleaving of degree N. In the case interleaving is enabled, we have to change the values of the randomly selected rows and columns to correspond to an interleaving scheme of N. That means that the actual row might be different in the actual memory because of the shuffle that occurred.

For that matter, we use an interleave_check function. This method contains appropriate code to implement interleaving in the following way: Consider a 16x16 structure. In such a structure we can have an interleaving of 1 (no interleaving), 2, 4, 8 or 16. This value indicates the NxN sub-arrays “created” inside the structure, where the shuffle of their respective elements happens. In the case of N = 2, we will have 64 sub-arrays inside the memory. Each sub-array has two columns, C0 and C1. No shuffle ever happens on C0, while in C1 the element swaps with the other element in the same column. For example, the element [1,1] moves to position [0,1] right above it, and the element [0,1] moves its way down the column to the position [1,1] because there is nothing above. This is applied in the whole structure by doing the same work on each 2x2 sub-array. In the case of interleaving 4, we now “split” the structure into 4x4 arrays. We have 16 sub-arrays in the structure. Now each column has an incrementing offset (C0 = 0, C1 = 1, C2 = 2, C3 = 3). This time no swap occurs, but each element in each column shift some positions “up” according to the column it belongs. The important part is to check the elements near the top. Those elements won’t go up to another “sub-array”. Thus, we have to move them down some positions. The same logic of incrementing the sub-array size and the column offsets goes for bigger interleaving degrees. It’s logical that we can’t have an interleaving scheme bigger than the size of the actual structure. Since the structure has to be in the form of NxN for interleaving to work (fill all the spaces of the structure), we can’t have an interleaving degree bigger than N. The interleave_check method returns the updated row to a global value interleaving_row. The column does not change, but it is used as an argument for the method because it is needed for the aforementioned calculations.

After that, we call the write_fault_mask method to write the parameters to the fault mask file. The write_fault_mask method accepts the arguments referring to: the selected coreId, the selected moduleId(s), the selected type, the selected row, the selected column, the selected offset, the model, the actual fault mask number, the first activation, the duration, two counters and the string type variable “dir”. The actual values that we put in any of those arguments depend on the place where we call the write_fault_mask method. This method is called numerous times inside the program under many different scopes.

We will give an example under the scope we already examine, which is the interleaving of a degree N. When we call the write_fault_mask method, we pass the following arguments:

- the coreId
- the randomly selected moduleId between all the moduleIds (if one is defined then we pass that one)
- the randomly selected type of fault (0 (stuck-at-0), 1 (stuck-at-1), 2 (bit-flip)). Can either be 0 or 1 for permanent and intermittent faults, and 2 for transient faults.
- the randomly selected row that corresponds to the interleaved memory
- the randomly selected column that corresponds to the interleaved memory
- the randomly selected offset (un-important to this version of the generator)
- the number of the model (in this case 0 (permanent), 1 is used for intermittent and 2 for transient)
- the fault mask number which is always 1. In this version of the generator, it has no practical usage.
- the first activation (in this case 1 for permanent fault)

- the duration (in this case 1 for permanent fault)
- the external loop counter
- the internal loop counter
- the “path/permanent”, path where the fault mask will be created

The write_fault_mask method initially creates a fault mask file. The file is created in a directory depending on the nature of the fault. In the case that we examine, in the permanent folder. The fault mask file is always opened by and fstream object FAULT_MASK_FILE in append mode (we use the arguments: fstream::in | fstream::out | fstream::app). We do these because we need to read the already existing data and re-write them along with the new ones (e.g., in the cases of multi-write operations in one file, i.e. parallel faults). The external counter passed into the function is used to name the fault mask file like fault_mask_EXT_COUNTER.txt. Next, we check the internal counter passed into the function. A counter > 1 indicates parallel faults, which means that we are revisiting (or we will revisit in our case) the file. In that case, we read each line of the file, and we append it to a fault[] string array. Then we close the file in append mode, and we re-open it in truncate mode (we use the arguments fstream::in | fstream::out | fstream::trunc) to write the previously read data plus the new data we have. Truncate in C++, empties the file at the time it opens it. Then we add the new values of the parameters to the fault[] string array in the form of: fault[i] = to_string(parameter) + “_” + fault[i] e.g 14_21 when we add row 21. If we reach the number of parallel faults (i.e. the internal counter equals the number of parallel faults) we delete the last character from every element of the fault[] string array, which is the character “_” from the previous iteration. Each element in the one-dimension fault[] string array represents a line on the fault mask file. We write each line in a serial manner, by appending each element of the fault[] array into the fstream object, FAULT_MASK_FILE. Lastly, we empty the fault[] array for the next iteration or the next file to write and we close the fstream object.

In the case of no interleaving (interleaving parameter = 1) we call the write_fault_mask directly. In any of the two cases (with or without interleaving), we call the write_fault_mask function as many times as the parallel faults we have to create in one permanent fault mask file. We do the same procedure to create one fault mask file for intermittent, transient, and mix faults. Then, we re-enter the while loop, and we repeat the same procedures for as many times as the number of fault masks we wish to create in each fault category (population parameter of the fault mask parameter file). In the case of a mix fault mask, we create different “columns” in our mixed fault mask files, were in each column the fault could be either transient, permanent, or intermittent. The type of fault in each “column” inside a fault mask file of a mixed type, is randomly generated.

In the case where the “multi_bit” variable is equal to 0, then a multi-bit fault injection can still happen if the parallel faults parameter value is greater than 1, but this time the faults won’t be adjacent although a similar series of actions to create the fault mask files, is taken. So, it is practical to point out just the differences. This time, when a random row or column is selected, it’s not under the scope of a multi-bit cluster. So, the row and the column that is selected for each “column” in a fault mask file, is the one that we will write in that file. At the case of multi-bit adjacent faults, there were many for-loops running through the internal counter (representing the number of parallel faults for each fault mask file). At this scope, there is just one for loop, right under the while loop, and inside it, are all the actions for the three (or four) types of fault mask files (transient, intermittent, permanent and/or mix). So, in that case, the process is different, but the

outcome to the user is exactly the same. The process repeats for as many fault masks are needed. The end of the main() function, happens with the closing of the FAULT_MASK_PARAMETERS_FILE, ifstream object, and the DEBUG_FILE, ofstream object.

Lastly, we explain the statistical safe sample calculation. This area of the code is typically inside comments, so it is not pre-enabled. For the statistical safe sample calculation some variables are defined or calculated (confidence, error margin, initial probability, probability, etc.). Then the safe sample is calculated using the formula: sample = ceil(initialpop / (1 + (errorMargin * errorMargin) * ((initialpop - 1)/((confidence * confidence) * prob * (1 - prob))))). The generator outputs this number to the user. It also shows the number of faults we decided to have (Population input parameter). The user decides whether to go on with the execution of the generator or not, depending on this result.

3.2.3 Output of the Fault Mask Generator, typical use-cases and corner cases

In this chapter, we examine the outputs of the fault mask generator, with some typical use-cases and a few corner cases. More specifically, we will examine the form of the fault masks generated while we present some outputs for some general cases. After the program is built (using the recommended GCC compiler or the build option from an IDE, e.g. Eclipse), we can execute it. The program was created for use in Linux operating systems. The fault_mask_parameter.txt file has to be on the same folder with the executable. Following are some of the use cases:

Use Case 1: Version = arm_small_susan_c-cortex_a15_base, Population = 6, CoreID = 0, ModuleID = 256, Rows = 128, Columns = 128, Bit-Vector Size = 32, Offset Range = 0, Parallel Faults = 1, Multi-Bit Fault Cluster Rows = 0, Multi-Bit Fault Cluster Columns = 0, Interleaving Scheme = 1, Total Simulation Time = 1892025, Duration = 100000.

We change the fault mask parameter file, with the parameters above. Those will be the base for the next use-cases, so from now on, we will only state the changes. We define: the “arm_small_susan_c-cortex_a15_base” core (the folder name), a population of 6 fault masks per fault category (transient, intermittent etc.), a CoreId of 0, a ModuleId of 256 (physical register file), a structure with 128 Rows x 128 Columns, a Bit-Vector size of 32, an Offset Range of 0, 1 Parallel fault per fault mask file (no multi-bit injection), no multi-bit adjacent injection, no interleaving, a total simulation time of 1892025 and a Duration of 100000.

By executing the program, we notice the output of the screen:

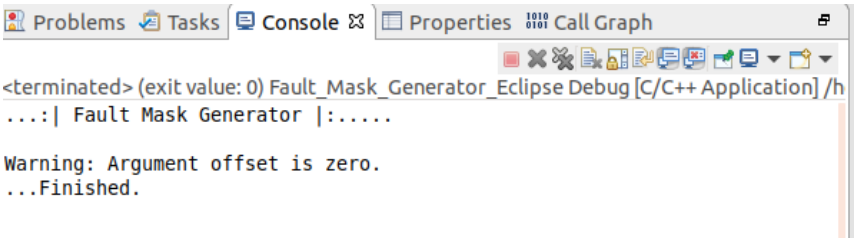


Figure 3.2: Execution message

The program is successfully executed, giving us a warning about the offset argument which is zero. Warnings, contrary to Errors, won’t affect the normal execution of the

program. We now notice that on the base folder, a new folder named arm_small_susan_c-cortex_a15_base appears (Figure 3.3). We also see the appearance of the generator.log file, which includes a log of all the internal operations of the program (Figure 3.4)

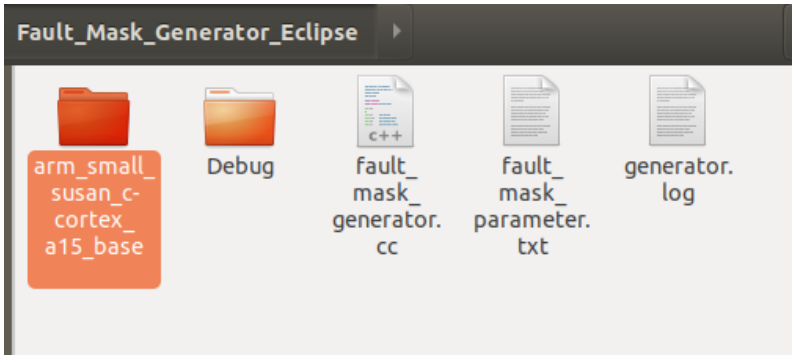


Figure 3.3: Updated File Structure

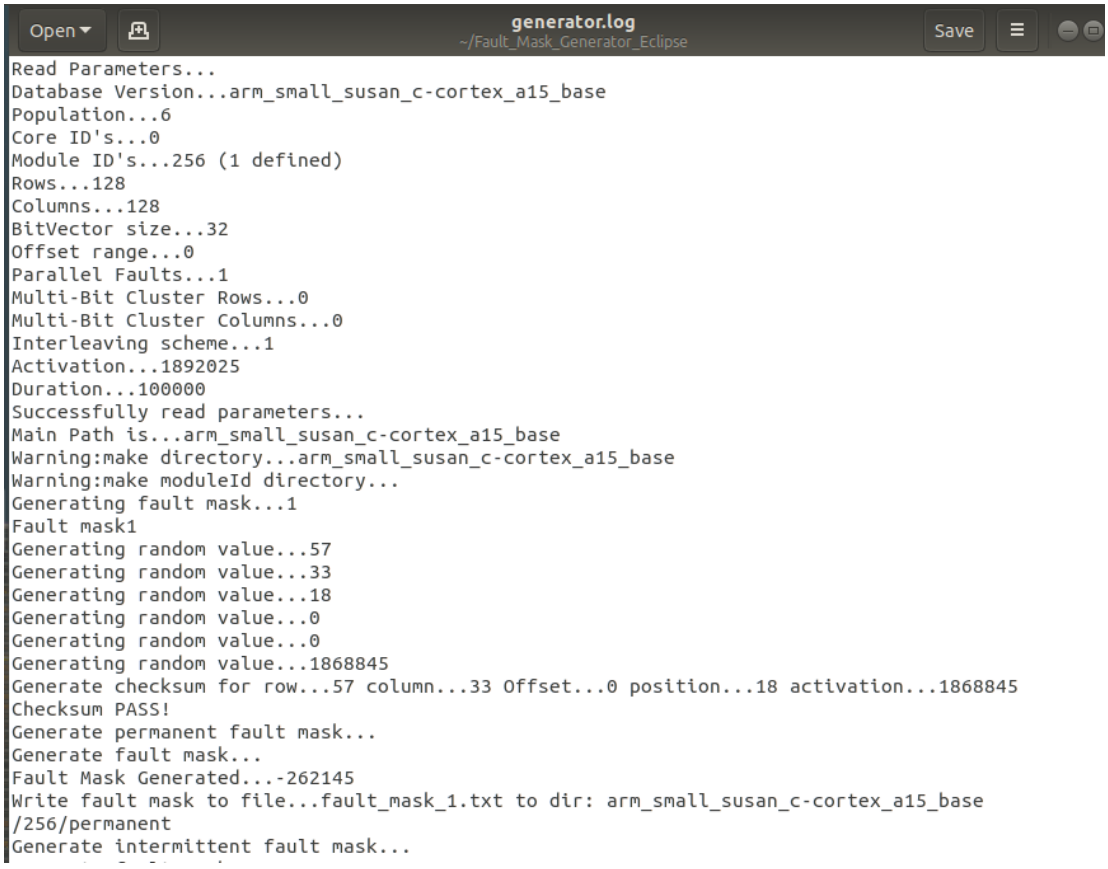


Figure 3.4: The generator.log file

Inside the core's name folder, we see a folder with the name of the ModuleId we inserted, and inside it we see four different folders, each for a fault type (transient, intermittent, permanent, mix)(Figure 3.5 and Figure 3.6).

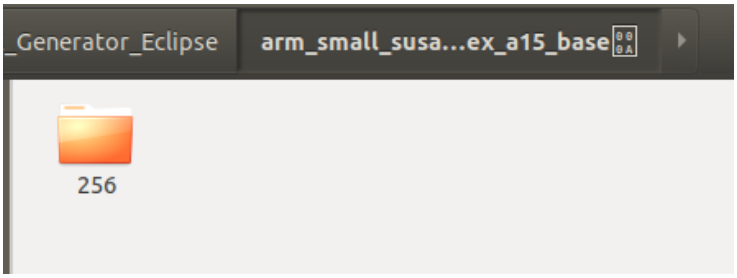


Figure 3.5: ModuleId folder

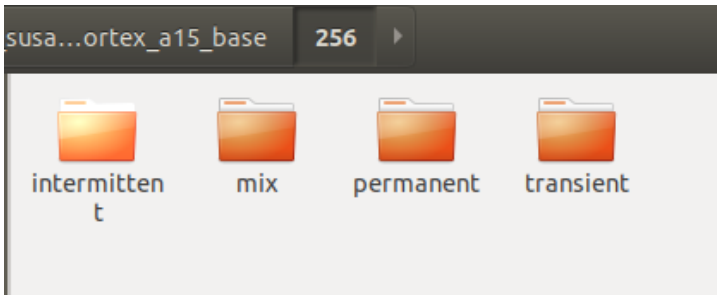


Figure 3.6: Fault Type Folders

Each folder (except in this case the mix folder, because no mix fault masks are created when we don't have multi-bit injection) has six fault mask files inside, as we requested. (Figure 3.7). The structure of a fault mask file appears in Figure 3.8. Each "column" consists of one fault and its details. In our case, we only inject one fault per fault mask, so we have no second column. Figure 3.8 shows what each element means.

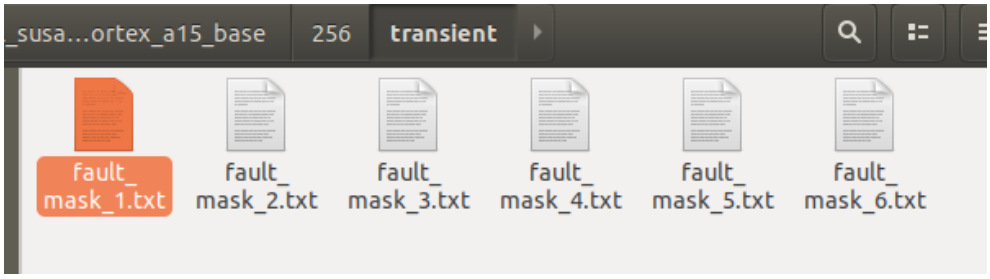


Figure 3.7: Fault Mask files inside a Transient Fault folder

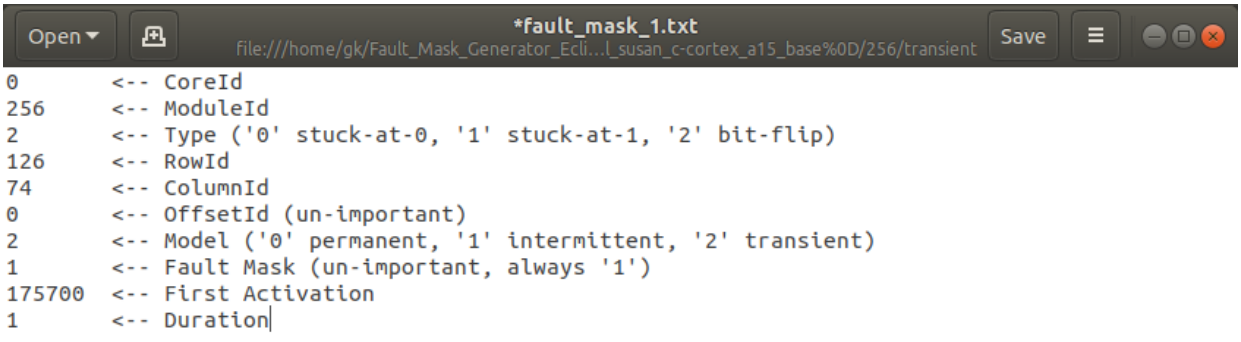


Figure 3.8: General Structure of Fault Mask File

We can see that the generator, randomly selected a fault in the position [126, 74] inside the 128x128 structure. We also see that, because we examine a transient fault mask, the Duration is always one and the Type is 2 (a bit-flip). The Model indicates the nature of the fault (2 for transient). In the case of an intermittent fault, we can see that the

Duration is 100000 as we defined, the Type is 0 (stuck-at 0 fault) and the Model is 1 (intermittent fault)(Figure 3.9). The remaining Type value is a 1 (stuck-at 1 fault), and the remaining Model value is 0 (a permanent fault).

Figure 3.9: Intermittent Fault Mask

Use Case 2: ModuleId 256;352, Parallel Faults = 5.

In this case, we add a second ModuleID (Bimodal Branch Predictor, 352) and also, we create a multi-bit fault injection with 5 faults per fault mask file. If we try to re-execute the program, we will see that the execution is abnormally terminated because the folder already exists (Figure 3.10). In order to execute the program correctly, we first have to delete or rename any folder with the same name (in our case arm_small_susan_c-cortex_a15_base).

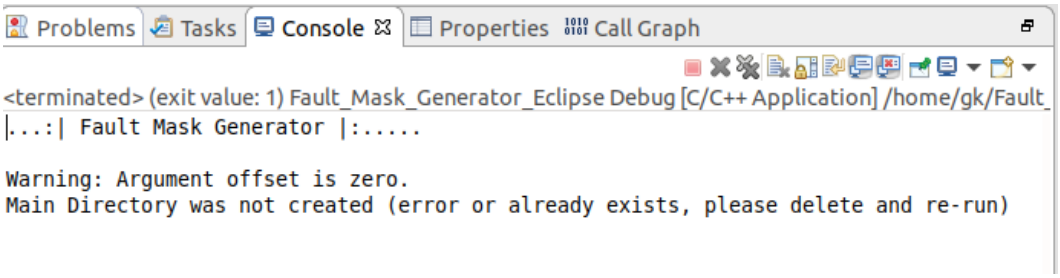


Figure 3.10: Execution Error

When we execute the program with the new parameters, we now see a folder with the two ModuleIDs (Figure 3.11). Since we have a multi-bit injection, we expect to see multiple “columns” inside the fault masks, each for a fault and also mix fault masks. By examining one (Figure 3.12), we see that five parallel faults are created, one in each “column”. The ModuleIds for each are randomly selected between the two predefined (256, 352). Same goes for the type of fault (0, 1 or 2), the rows and columns ([75, 70], [41, 60] etc.), the model etc. The duration depends on the type of fault with transient and permanent fault having a duration of 1.

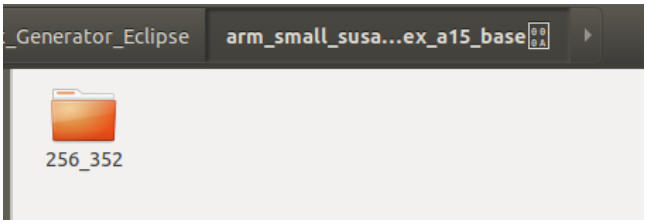


Figure 3.11: A folder with the two ModuleIDs

Open
fault_mask_2.txt
file:///home/gk/Fault_Mask_Generator_Ecli...all_susan_c-cortex_a15_base%0D/256_352/mix

```
0_0_0_0_0
256_256_256_352_256
1_0_2_1_1
75_41_34_87_10
70_60_71_8_62
0_0_0_0_0
0_1_2_1_1
1_1_1_1_1
1_178255_141522_1074904_854638
1_100000_1_100000_100000
```

Figure 3.12: A mix fault mask with five parallel faults

Use Case 3: Multi-Bit Fault Cluster Rows = 3, Multi-Bit Fault Cluster Columns = 3.

In this case, we choose to do an injection in adjacent bits of the structure. That is why we create a 3x3 cluster, randomly put inside the memory, where the 5 parallel faults will be injected. We examine a transient fault mask (Figure 3.13). By looking at the row and columns of the calculated faults, we can see that all of them are injected in the area [74-76, 97-99] of memory, so they are all adjacent. Also, since the faults are adjacent, they are injected in only one of the two modules we define (here randomly selected is Module 256). This way, we model adjacent multi-bit faults, which are quite common threats in processor reliability (see Chapter 2).

Open
fault_mask_4.txt
file:///home/gk/Fault_Mask_Generator_Ecli...san_c-cortex_a15_base%0D/256_352/transient

```
0_0_0_0_0
256_256_256_256_256
2_2_2_2_2
74_76_75_76_75
99_99_99_98_97
0_0_0_0_0
2_2_2_2_2
1_1_1_1_1
1698948_1698948_1698948_1698948_1698948
1_1_1_1_1
```

Figure 3.13: Multi-Bit Adjacent Faults

By examining the generator.log file, we also see the checksum capabilities of the program. For a permanent fault mask file, the program recognizes that we've already 'injected' a fault in positions [119, 95] and [120, 95], checksum fails and the program re-tries with a new position until checksum is passed.

```
Generate checksum for cluster row...119 and column...95
Checksum PASS!
Write fault mask to file...fault_mask_1.txt to dir: arm_small_susan_c-cortex_a15_base
/256_352/transient
Generating random value...119
Generating random value...95
Generate checksum for cluster row...119 and column...95
Checksum FAIL!
Generating random value...120
Generating random value...95
Generate checksum for cluster row...120 and column...95
Checksum PASS!
Write fault mask to file...fault_mask_1.txt to dir: arm_small_susan_c-cortex_a15_base
/256_352/transient
Generating random value...119
Generating random value...95
Generate checksum for cluster row...119 and column...95
Checksum FAIL!
Generating random value...120
Generating random value...95
Generate checksum for cluster row...120 and column...95
Checksum FAIL!
Generating random value...118
Generating random value...93
Generate checksum for cluster row...118 and column...93
Checksum PASS!
Write fault mask to file...fault_mask_1.txt to dir: arm_small_susan_c-cortex_a15_base
/256_352/transient
```

Figure 3.14: Checksum check in the generator.log file

Use Case 4: Parallel Faults = 8, Interleaving Scheme = 2.

We now add more multi-bit adjacent faults, but we also add interleaving (with a degree of 2). This means that the whole memory will be “shuffled”. The memory is divided into 2x2 arrays and the shuffles happen inside them. This way, we must make sure that the computed Row corresponds to an interleaved memory. The effect of interleaving is not directly observable from the fault mask files. This time, we see the same structure as above, but we now have 8 adjacent faults (Figure 3.15). By observing the generator.log file we can see how the interleaving process happens (Figure 3.16). We can see that when the position [79, 89] is selected, the program uses the function interleaving_check to check what is the “real” position corresponding to an interleaved memory. The program returns the new position of [78, 89] changing the row to row = row – 1. For another position [80, 88] where the column is even, no change occurs.

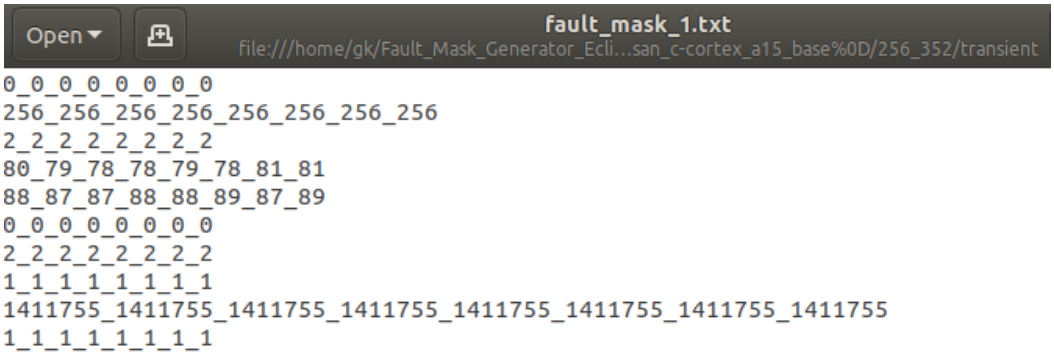


Figure 3.15: A transient fault mask with 8 parallel adjacent faults

```
Generate permanent fault mask...
Generating random value...80
Generating random value...88
Generate checksum for cluster row...80 and column...88
Checksum PASS!
Checking if randomly selected row...80 and column...88 corresponds to an interleaved memory with
N = 2.
Method returns row...80 and column...88 of the interleaved memory
Write fault mask to file...fault_mask_1.txt to dir: arm_small_susan_c-cortex_a15_base
/256_352/permanent
Generating random value...79
Generating random value...89
Generate checksum for cluster row...79 and column...89
Checksum PASS!
Checking if randomly selected row...79 and column...89 corresponds to an interleaved memory with
N = 2.
Method returns row...78 and column...89 of the interleaved memory
```

Figure 3.16: Interleaving of degree 2

Use Case 5: Interleaving Scheme = 4.

The same process happens with an interleaving scheme of 4. By examining Figure 3.17, we see that the generator randomly picks a position [24, 8]. We check to see what the real position is, for an interleaved memory. The program returns position [24, 8], so no change occurs meaning that we are on C0 of a 4x4 sub-array inside the structure. For the next element in position [24, 6], the program does the same procedure and returns [26, 6] meaning that we are on C2 of a 4x4 sub-array inside the structure. So, the row = row – 2 since 26 % 4 != 0 or 1 (e.g. 24 or 25). That would indicate that we can't move two positions 'up' because we will exceed the boundaries of the sub-array (in that case we would have had to move some positions 'down'). So, we move two positions 'up'. Same goes for the element [22,6] that is moved to position [20,6]. This element obviously belongs to a different sub-array in the structure.

```
Checking if randomly selected row...24 and column...8 corresponds to an interleaved memory with N
= 4.
Method returns row...24 and column...8 of the interleaved memory
Write fault mask to file...fault_mask_4.txt to dir: arm_small_susan_c-cortex_a15_base
/256_352/permanent
Generating random value...24
Generating random value...8
Generate checksum for cluster row...24 and column...8
Checksum FAIL!
Generating random value...24
Generating random value...6
Generate checksum for cluster row...24 and column...6
Checksum PASS!
Checking if randomly selected row...24 and column...6 corresponds to an interleaved memory with N
= 4.
Method returns row...26 and column...6 of the interleaved memory
Write fault mask to file...fault_mask_4.txt to dir: arm_small_susan_c-cortex_a15_base
/256_352/permanent
Generating random value...22
Generating random value...6
Generate checksum for cluster row...22 and column...6
Checksum PASS!
Checking if randomly selected row...22 and column...6 corresponds to an interleaved memory with N
= 4.
Method returns row...20 and column...6 of the interleaved memory
```

Figure 3.17: Interleaving of degree 4

Use Case 6: Statistical Safe Sample Calculation

At this time, we re-execute the program while we remove the comments from the code that calculates the statistical safe sample. After execution we get the following message:

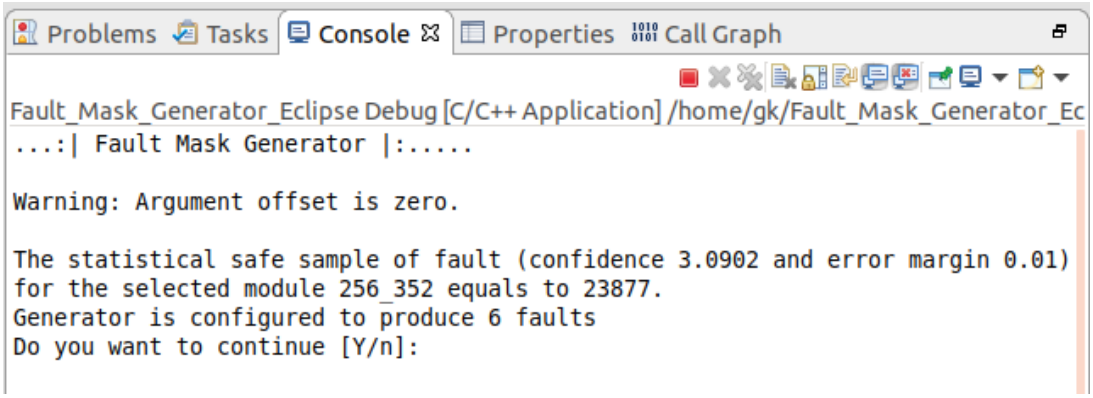


Figure 3.18: Calculation of the Statistical Safe Sample

The generator informs us that we need 23877 samples to be statistically safe, while we have 6. If we choose to continue with the execution (Figure 3.19), we type 'Y' or 'y' and the program executes normally. Else (Figure 3.20), we type 'n' and the program execution stops immediately.

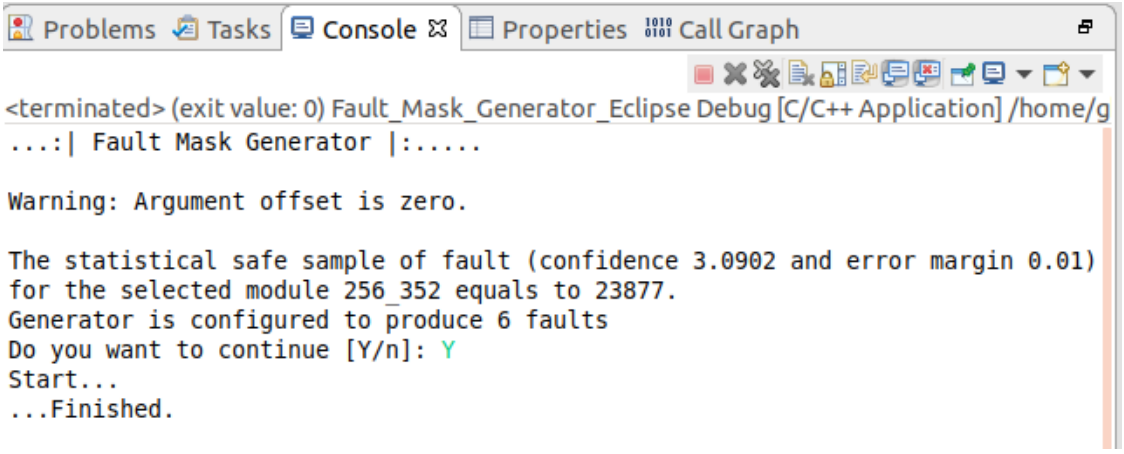


Figure 3.19: Execution continues

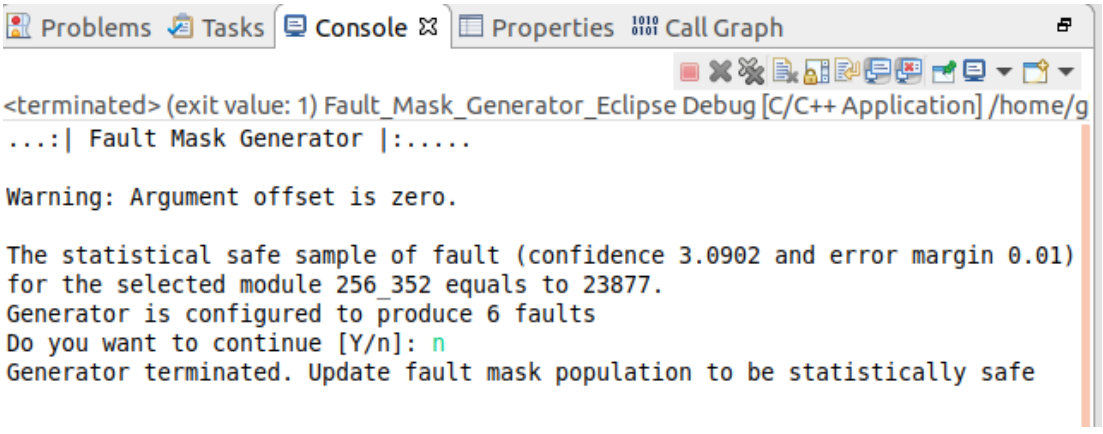


Figure 3.20: Execution stops

Corner cases:

The program also acts against some corner cases, either by throwing error messages and terminating the execution (for example if a parameter is not correct) or by acting according to the situation. One example is the case where the multi-bit cluster is very big compared to the size of the structure. This way, the cluster could get out of the structure limits and produce values for Rows and Columns that don't actually exist! We will look at the extreme case of creating a cluster of 128x128 for a 128x128 memory, which has no practical use but demonstrates the way the program prevents such cases. As one can see in the generator.log file (Figure 3.21 which demonstrates the last few lines of a quite big generator.log file) the program does many attempts to fit the cluster into the memory until it randomly hits the only position where the cluster can start, [0,0].

```
Generating random value...3
Generating random value...54
Generating random value...117
Generating random value...28
Generating random value...0
Generating random value...0
Generating random value...8
Generating random value...1
Generating random value...1
Generating random value...527492
Generate checksum for row...0 column...0 Offset...0 position...8 activation...527492
Checksum PASS!
Generate permanent fault mask...
Generating random value...30
Generating random value...88
Generate checksum for cluster row...30 and column...88
Checksum PASS!
Generate fault mask...
```

Figure 3.21: Cluster getting out of structure limits

Another case could be the one where we try to inject more errors than a cluster can have. For example, a 4x4 cluster contains 16 elements so we can inject up to 16 faults. If we try to inject 17, the program will output an error and ask as, to change the Parallel Faults parameter by informing as of the maximum number of faults we can inject for the given cluster size (Figure 3.22). The program handles many cases like these.

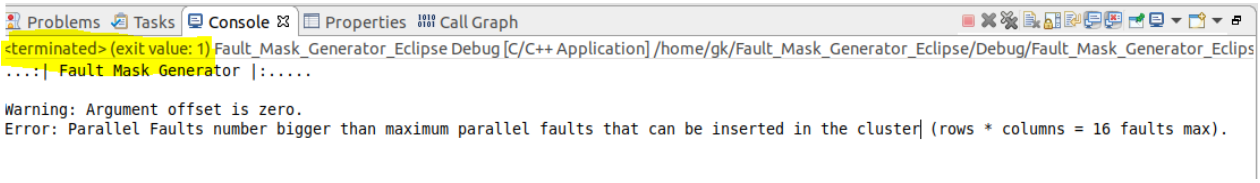


Figure 3.22: Program error; trying to add more faults than allowed

3.3 Implementation B (Fault Mask Generator in Python)

3.3.1 The Fault Mask Parameters

As previously described in the first implementation of the fault mask generator, a set of parameters is required. Those parameters are presented below:

- *--version: Version used as part of the export path* ***DEFAULT VALUE: v1***

- *--core_id: It's the CPU core id related with the current output dataset* ***DEFAULT VALUE: 0***
- *--rows: Number of rows of SRAM structure, where faults will be injected* ***DEFAULT VALUE: 32***
- *--columns: Number of columns of SRAM structure, where faults will be injected* ***DEFAULT VALUE: 32***
- *--faults: Number of faults to be injected* ***DEFAULT VALUE: 1***
- *--model: Type of model. Valid values are 0 for permanent, 1 for intermittent and 2 for transient* ***DEFAULT VALUE: 2 (Transient)***
- *--cl_rows: Number of rows of a cluster used for multi-bit injection*

  DEFAULT VALUE: 1
- *--cl_cols: Number of columns of a cluster used for multi-bit injection*

  DEFAULT VALUE: 1
- *--duration: Defines the total duration. If the model type is equal to 2 (Transient), this value is ignored.*

  DEFAULT VALUE: 10000
- *--module_id: Defines the component id of the fault injection*
- *--simtime: Duration of the simulation. It is translated to the workload clock cycles*
- *--seed: Seed used for application to allow reproducing same output datasets for a given seed*

DEFAULT VALUE: Current time in seconds since the Epoch. time() function is used for that purpose

- *--path: Specified path for storing the output datasets*

  DEFAULT VALUE: latest
- *--interleaving: Interleaving distance value between bits*

  DEFAULT VALUE: 1 (Essentially no interleaving)
- *--probability: Probability value for calculating number of datasets*

  DEFAULT VALUE: 50 (which means 50%)
- *--error_margin: Error margin value for calculating number of datasets*

  DEFAULT VALUE: 1 (which means 1%)
- *--confidence_level: Confidence level value for calculating number of datasets*

  DEFAULT VALUE: 99.8 (which means 99.8%)
- *--datasets: Dataset number to be produced*

The above arguments are passed as command arguments from the command line. For example: ***python main.py --module_id=2 --simtime=100***. If any argument is missing, its default value will be set.

3.3.2 Technical details (the fault mask generator source file)

The second implementation of the simulator is written in Python (<= 2.7 version to be compatible with the servers GeFIN is running).

Once the simulator runs, the output mask files are exported under the path specified as an argument. By running the example described before, the exported path is: **latest/v1/1**. Since no dataset size was given, its size is calculated by the statistical formula that is described in detail in [4]. In case older generated mask files exist in the specified export path, a cleanup takes place before running the fault generation.

An abstraction of the implementation and workflow of this process is described briefly below:

1. Reading of arguments.
2. Basic validations on arguments (e.g., cluster size can fit inside outer grid).
3. Calculating the fault population (if it was not provided), using statistical methods with probability, error margin, and confidence level.
4. Removing potential old mask files.
5. Iteratively calculating and storing the mask files in the specified export path, taking into account the presence (or absence) of interleaving.

3.3.3 Output of the Fault Mask Generator

The output mask files always start with a single line that contains a **V2** constant. Its purpose is to help the integration of GeFIN with different inputs of different simulators. V2 constants states that the fault-mask will have a different format, consisting of values that correspond to the following fields, separated by empty spaces:

- **core_number**: Target core number.
- **component_number:** Target component number.
- **ticks:** Snapshot that the error occurred. It's a random integer between 0 and simulation time.
- **error_row_dim:** Number of rows of SRAM structure, where faults were injected
- **error_column_dim:** Number of columns of SRAM structure, where faults were injected.
- **model:** Type of model (0 for permanent, 1 for intermittent and 2 for transient)
- **type:** Type of fault (AND, OR, XOR). If the model is transient, this field has no value. Otherwise a random type is chosen.
- **duration:** T*he total duration. If the model type is equal to 2 (Transient), this value is ignored.*

The number of the above faults printed to the output is defined from the **--faults** field given as an argument to support parallel faults.

4. EXPERIMENTAL SETUP

4.1 General

In this chapter, we present the experimental setup for all the fault injection campaigns. We define the system characteristics (the chosen core under test), the workloads (i.e., benchmarks), the components of the system that we targeted (L1 Cache, L2 Cache, etc.), the number of runs, the faults that were injected per run and their type (single bit or multi-bit faults). We also list the classification techniques of the faults (Masked, SDC, etc.) and the metrics used to make some conclusions based on the results (AVF, FIT). Lastly, we define the metrics we used in our results to convert them for actual fabrication technology nodes.

4.2 System under test

For the microarchitecture-level reliability assessment, the GeFIN fault-injection framework was used alongside the microarchitecture-level simulator Gem5 (see Chapter 1). The simulator was configured to resemble the micro-architecture of the ARM Cortex-A9 core as close as possible. Table 4.1 summarizes some major attributes of the core, that were used in the micro-architecture level configuration [11].

Table 4.1: Microarchitectural Configuration of Cortex-A9

Microarchitectural attribute	Value
ISA/Core	ARMv7 /Out-of-order
Data cache	32KB 4-way
Instruction cache	32KB 4-way
Physical Register File	56 registers
Instruction queue	32
Reorder buffer	40
Fetch/Execute/Writeback width	2/4/4

Inside the ARM Cortex-A9 processor, we target the following components:

- The Level 1 (L1) Instruction Cache
- The Level 1 (L1) Data Cache
- The Level 2 (L2) (unified) Cache
- The Instruction Translation lookaside buffer (ITLB)
- The Data Translation lookaside buffer (DTLB)
- The Register File

More specifically:

- *Level 1 (L1) Instruction Cache:* The L1 Instruction Cache is exploited by the CPU front-end, the fetch stage. Requests that come from the instruction port either follow the program flow or in case of control instructions, are predicted by the branch prediction units. In both cases, the incoming memory block contains

instructions with high probability to be used, due to locality and accurate branch prediction. So, a faulty fetched cache block is very likely to be used by the core. Study [14] shows that for all benchmarks used, 90% of corruption cases appear in less than 100,000 clock cycles.

- *Level 1 (L1) Data Cache:* Data Cache requests are guided by memory operations of the program flow, mispeculated load instructions, prefetching and unresolved load/store dependencies. There is no standard ratio among these sources of data cache requests, and the workload itself can have a severe impact on each one of them. Also, speculation can have a significant effect on the measurements contributing to the argument that Data Cache accessing is highly unpredictable.
- *Level 2 (L2) (unified) Cache:* The unified second level of cache memory has similarities with the L1 cache. It includes both data and instruction blocks and behaves accordingly, similar to the corresponding L1 cache. The data part of L2 cache is highly unpredictable similarly to the L1 data cache. However, the findings of the L1 instruction cache also apply for L2. Study [14] shows that instructions in L2 follow a similar trend line with the L1 instruction cache and are normalized above 80% on a Manifestation epoch (the time from the first access of the faulty entry in the structure) of 100,000 clock cycles.
- *Register File:* The Gem5 simulator uses an approach in its out-of-order core, consisting of a Physical register file combined with a rename map to indicate the current, up-to-date committed values for the architectural registers. Allocated resources in the physical register file are either *architectural* or *dynamic*. Typically, the physical register file has allocated at minimum the number of architectural registers at any time, and from that point on, it additionally allocates resources for the in-flight dynamic instructions. So, the term dynamic and architectural registers is used to express the nature of the allocated resource in the physical register file. Their main difference is their residency time where Dynamic registers remain active as long as the instruction lives inside the pipeline (usually a few clock cycles) while Architectural registers are part of the program state and may be used millions of cycles later, or even not used at all. This leaves for a very short window of opportunity for dynamic registers, to cause a state corruption. Study [14] shows that for an experimental sample of 40,000 injections, all of the faults that hit dynamic registers and lead to a state corruption, did reach a visible point in the program flow in less than 500 clock cycles. The same study shows that, on the other hand, the architectural registers show a mixed behavior: some are highly critical, and others are not critical at all. The vulnerability of the Register File in that study is approximately 5%.
- *Translation Lookaside Buffer:* A Translation Lookaside Buffer (TLB) is a type of cache used to speed up the virtual to physical memory translation process by storing recently accessed virtual memory page numbers and their related physical page numbers. They also store control information related to these entries, such as a bit to indicate if an entry is valid, or permission bits. A common way of implementing a TLB consists of a Content Addressable Memory (CAM) that stores the virtual page numbers and an associated Random-Access Memory (RAM) which keeps the physical page number for each CAM entry. An error can disturb either of those structures. Bit flips in the virtual page information stored in the CAM, may produce a false positive when the queried tag matches the corrupted entry. The result would be the execution of a wrong set of instructions by the program, as they are retrieved from a different (but valid) physical page. This has several possible consequences such as a hard fault, a silent data corruption, a

system freeze, etc. Another effect from the bit flip could be a false negative, where the virtual page being checked, no longer matches any entry of the table, and a miss is produced. However, an error in the CAM values stored in the TLB, does not always produce an undesired behavior. If the program does not access the specific virtual page in error, or the entry is overwritten due to the TLB replacement algorithm, then the error will be masked. An error could also occur in the RAM part of the TLB. This would modify the physical address, which may now point to a valid (but different) physical page or to a page that is not allocated to the process. A TLB for both instructions (ITLB) and data (DTLB) is used in processors, just like it does with caches [17].

These components provide a wide coverage of the core (about 95%) although each component occupies a different percentage of that space with the L1 and L2 caches dominating while the other components (Register File, ITLB, etc.) hold a much smaller percentage of that space. For each component, a campaign of 1000 simulation runs (for each case of single, double-bit or triple-bit faults) were executed per benchmark to satisfy a specific error-margin and confidence level.

4.3 Workloads

A subset of the MiBench suite was used as the target workloads for the experiments. MiBench benchmarks are widely used in reliability studies as they combine a wide range of common workloads/algorithms with relatively small datasets, which effectively translates to short execution time (shorter compared to the standard benchmarks for performance: SPEC) and a large number of fault injections. MiBench contains benchmarks from different application domains that also have similar instruction mixes with the SPEC benchmark suite. It consists of six categories including algorithms used in: Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation and Telecommunications. Table 4.2 includes the full list of the MiBench suite benchmarks. A portion of them was used in our experiments namely: CRC32, FFT, adpcm_dec, basicmath, cjpeg, dijkstra, djpeg, gsm_dec, qsort, rijndael_dec, sha, stringseach, susan_c, susan_e, susan_s [12][13][14]. Table 4.3 summarizes the used benchmarks along with their execution time (clock cycles). The complete execution of each benchmark is required per fault injection campaign, for comparison with AVF estimations [11].

Table 4.2: MiBench Benchmarks

Auto/Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rihndael dec.	GSM enc.

	tiffmedian			sha	GSM dec.
	typeset				

Table 4.3: Benchmarks execution time in Clock Cycles

Benchmark	Execution Time (Million Clock Cycles)
adpcm	53,690,367
basicmath	67,556,250
cjpeg	26,126,843
CRC32	132,195,721
dijkstra	41,643,556
djpeg	10,105,853
FFT	48,339,852
gsp_dem	12,862,888
qsort	31,326,716
rijndael_dec	33,327,494
sha	12,141,593
stringsearch	1,082,451
susan_c	2,150,961
susan_e	2,876,202
susan_s	13,750,557

4.4 Fault Effect Classification

The GeFIN injector classifies the outcomes of each fault simulation based on the impact of the fault on the simulated system. Five classes are used for the fault effects classification for AVF measurements [12][14]:

- *Masked:* Masked includes the fault injection runs in which the fault does not affect the execution of the application (which is executed through its end) or the system. The result of a simulation with a masked fault is identical to the fault-free simulation in terms of the output of the application and any exceptions generated during execution.
- *Silent Data Corruption (SDC):* Silent Data Corruption includes the fault injection runs for which the final output of the program that is written to an output file is corrupted (differs from the output of the fault-free execution) and no other indication of the fault has been recorded (an abnormal event such as an exception, etc.).
- *Timeout:* Timeout includes all the cases where the simulation did not finish within a certain amount of time, that lead to either a Deadlock (a condition in which the program flow has been trapped -due to the injected fault- and can't commit any further instructions) or a Livelock (a situation where the program flow has been redirected -due to the injected fault- and continues the execution of instructions on

random code areas). The execution timeout limit to monitor these cases is four times equal to the fault-free execution of each benchmark.

- *Crash:* Crash includes any case that results in an unrecoverable situation that stops the simulated program. Crashes involve: a process crash, where the simulated program was abnormally terminated and a system crash (kernel panic), where the simulated full-system was unable to recover.
- *Assert:* Assert includes all the cases where the simulation was unexpectedly terminated due to a simulator failure. If the simulator crashes or reaches a high-level condition that is unable to handle, it raises an assertion to stop the simulation.

4.5 Metrics, Fault Types and Fabrication Technology

We use two metrics, namely the Architectural Vulnerability Factor (AVF) and Failures-in-Time (FIT)(see Chapter 1 for definitions) for the results of the experiments. We calculate the AVF per component for different benchmarks and for a different number of transient faults (single or multi-bit fault injection) using the above fault classification for the results. The AVF is then the sum of all the non-Masked classes. For each component, an AVF for each fault case (1, 2 or 3 faults injected) is computed. We then compute different AVFs (for each component) by taking into account the fabrication technology node (using soft error rates). After that we also calculate the FIT for each component for different technology nodes, taking into account the rawFIT of each node, the size of each component and the corresponding AVF, we previously calculated. Lastly, for a technology node, we can calculate the FIT of the core but adding the respective FITs of all the components.

For the fault injection experiments, we use a single-bit fault injection and a multi-bit fault injection with 2 or 3 transient faults per fault injection campaign. In the multi-bit case, we use the adjacent fault injection capability of the fault mask generator (see Chapter 3) using the solution of clusters to inject faults in adjacent bits in the selected structure. We inject 2 or 3 faults per cluster per fault campaign in the selected structure (see Chapter 2 for the definitions of clusters and multi-bit fault injection). This way, we simulate the effects of multi-bit faults in real structures, and we examine the results.

5. RESULTS / COMPARATIVE RESULTS

5.1 General

In this chapter, we present the results from the experiments that took place. The presented results showcase the Vulnerability (AVF) of different components (Caches, Register Files, etc.) of the core under test (ARM Cortex-A9) for many technological nodes, and the Reliability of each component and the system (FIT per technological node).

5.2 Vulnerability (Architectural Vulnerability Factor)

In this section, we present the results showcasing the vulnerability of each component of the core under test, utilizing the AVF metric and the fault effect classification described in Chapter 4 (Masked, SDC, etc.). We do that for three cases: a single fault was injected during each fault campaign, two faults were injected in adjacent areas (on a 3x3 cluster inside the structure) of each structure during each fault campaign and the case where three faults were injected in adjacent areas (on a 3x3 cluster inside the structure) of each structure during each fault campaign. As we described in the previous chapter, we performed 1000 runs (fault campaigns) for each benchmark, for each fault injection case (one, two or three faults) and for each structure. Fifteen benchmarks were used for each experiment, namely: CRC32, FFT, adpcm_dec, basicmath, cjpeg, dijkstra, djpeg, gsm_dec, qsort, rijndael_dec, sha, stringseach, susan_c, susan_e, susan_s. Lastly, we present the AVFs in correlation with the fabrication technology nodes from 250 to 22 nm.

5.2.1 Level 1 Data Cache

The results for the Level 1 Data Cache of the ARM Cortex-A9, for all three cases (one, two, three faults injected per run) are:

5.2.1.1 Single-bit fault injection per run

The results from the single fault injection appear in Figure 5.1. We observe that the Masked class varies from a percentage of 53.7% for the worst case (cjpeg) to 94.5% for the best case (stringsearch) with the rest benchmarks being distributed evenly among those two corner cases. For the runs that caused a SDC we see a percentage from 39.40% for the worst case (cjpeg) to 1.10% for the best case (dijkstra). We point out a match between the best and worst cases for Masked and SDC while we notice that SDC happens to be the second-most frequent class in percentage for most of the cases, an observation seen in almost all of our results for the D-Cache. For the Crash class, the percentages are pretty low (around 2-3%) for most cases, while we see higher percentages for the cjpeg (6.70%), dijkstra (4.50%) and qsort (6 %) benchmarks. Again, the cjpeg benchmark, has the worst percentage. For the other two classes (Assert and Timeout) the percentages are extremely low, with an exception in the case of qsort where we see a percentage of 5.50% for Timeout.

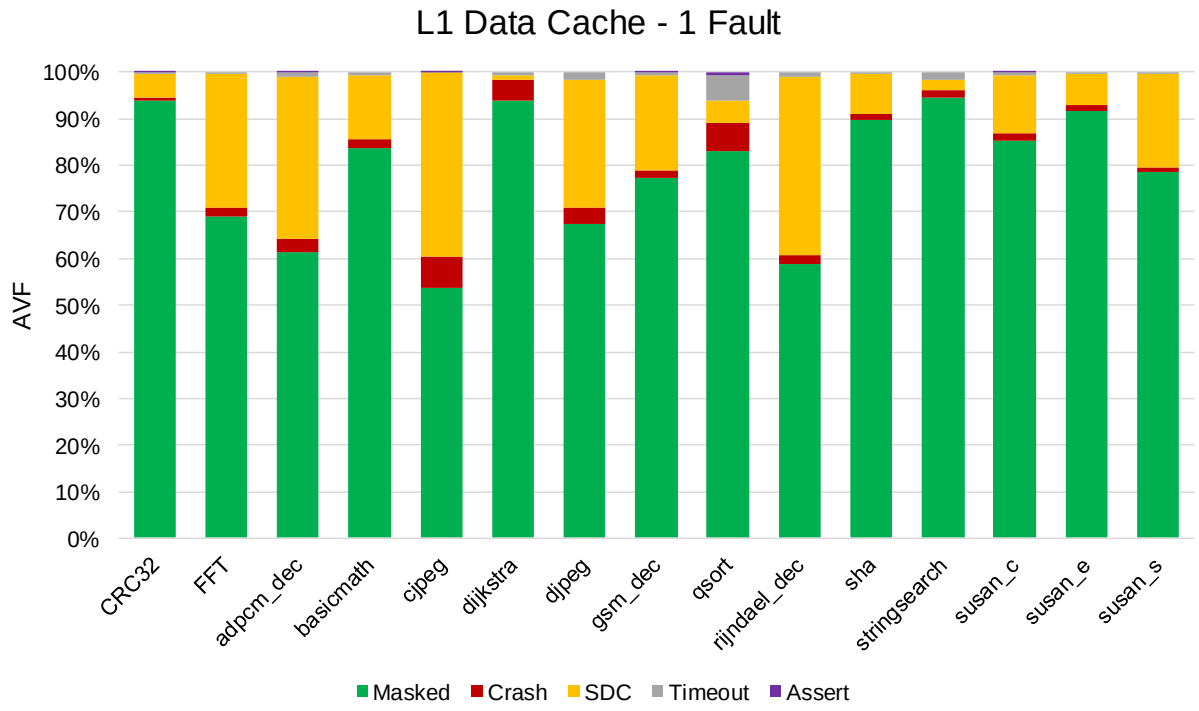


Figure 5.1: L1 D-Cache, one fault injected per run

5.2.1.2 Multi-bit (two bit) fault injection in adjacent bits, per run

The results from the double fault injection in adjacent bits appear in Figure 5.2. For the Masked class we observe a variation from 39.20% for the worst case (rijndael_dec) to 92.10% for the best case (stringsearch). Three more benchmarks have Masked percentages lower than 50% namely: adpcm_dec, cjpeg, djpeg while FFT has a percentage just above 50% (51.8%). In all the cases the benchmarks have a smaller degree of Masked faults compared to the single-bit fault injection, revealing a vulnerability of the L1 D-Cache to multi-bit fault injection in adjacent bits, just like expected. For most benchmarks the majority of the remaining percentage belongs to SDCs while in the dijkstra benchmark the Crash prevails with 7.30% compared to only 1% for SDCs and in the stringsearch case Crash prevails with 3% compared to 2.60% for the SDC. SDCs vary from 56.3% for the worst case (rijndael_dec) to 1% for the best case (dijkstra). Crashes vary from 10.6% for the worst case (qsort) to 1.30% for the best case (susan_s and CRC32). Only one more benchmark (cjpeg) has a Crash rate bigger than 10% (10.5%). We observe, that the SDCs and Crashes rates are worst for all the cases compared to the single-bit fault injection, similar to the Masked Class. For the two remaining classes the percentages are extremely low, except the case of qsort where we have a Timeout percentage of 9.30%, bigger than the SDC (7%) and slightly smaller than Crash (10.60%).

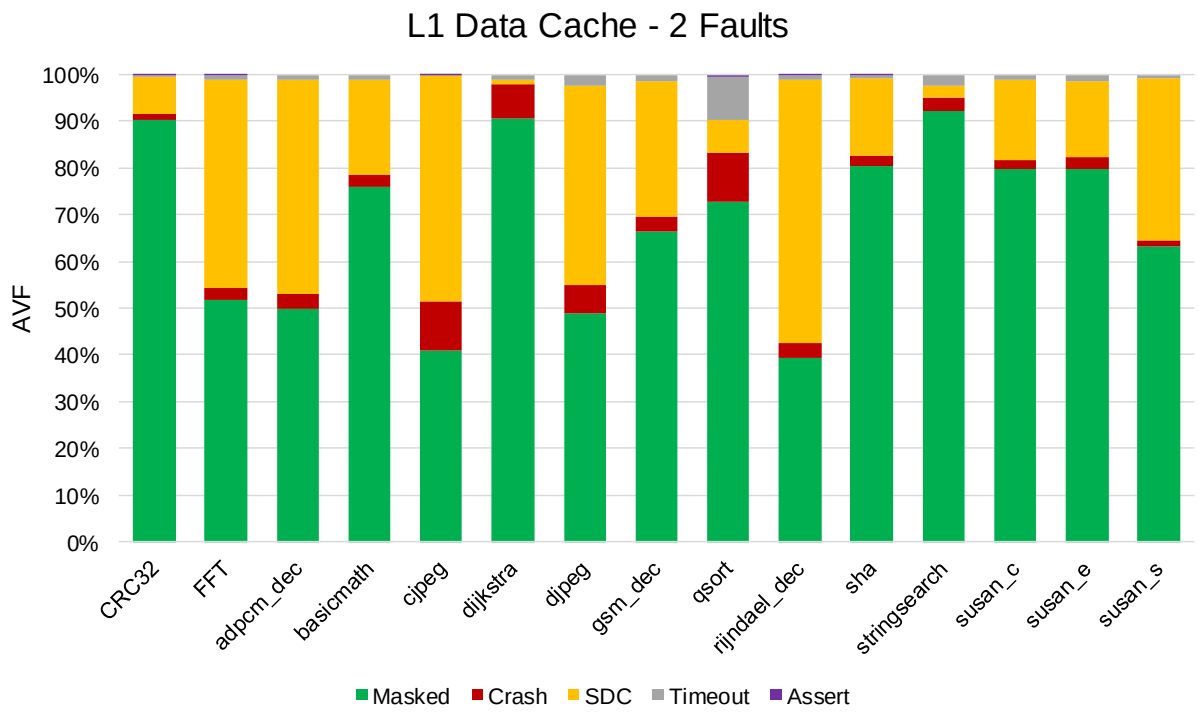


Figure 5.2: L1 D-Cache, two faults injected per run

5.2.1.3 Multi-bit (three bit) fault injection in adjacent bits, per run

The results from the triple fault injection in adjacent bits appear in Figure 5.3. For the Masked class we observe a variation from 29.70% for the worst case (rijndael_dec) to 88.90% for the best case (stringsearch). Compared to single and double fault injection, we see a 24% and a 9,5% decrease respectively for the worst case and a 5.6% and a 3.2% decrease respectively for the best case. This showcases a further decrease of the Masked faults for each benchmark as we inject more adjacent faults per run. Now five, benchmarks have Masked percentages lower than 50%. All of that, certainly strengthen the argument for the vulnerability of the L1 D-Cache to multi-bit adjacent faults. For most benchmarks the majority of the remaining percentage belongs to SDCs while in the dijkstra benchmark the Crash prevails with 10.60% compared to 1.50% for SDCs and in the stringsearch case Crash prevails with 4.10% compared to 3.60% for SDC. SDCs vary from 62.50% for the worst case (rijndael_dec) to 1.50% for the best case (dijkstra). Crashes vary from 13.4% for the worst case (qsort) to 1.80% for the best case (CRC32). We observe, that the SDCs and Crashes rates are worst for all the cases compared to the two bits fault injection, similar to the Masked Class. For the two remaining classes the percentages are extremely low, except the case of qsort where we have a Timeout percentage of 12.30%, bigger than the SDC (10%) and slightly smaller than Crash (13.40%). In most cases we see many similarities with a trend for worst results for three-bit fault injection. We also see that trend in the Assertion class which is almost non-existent in one and two-bit fault injections (from 0 to 2 simulator crashes out of 1000 runs). In the case of the three-bit fault injection we see a percentage of 0.90% Assertions for CRC32 an extraordinary 9x compared to the two-bit fault injection case for the same benchmark, and smaller increases for the rest of the benchmarks. On the other hand, we see a gradual decrease in Assertions for the qsort benchmark, as we increase the number of injected faults (0.60% to 0.40% to 0.10%) which makes it difficult to draw a conclusion for the relationship between the number of faults injected and Assertions.

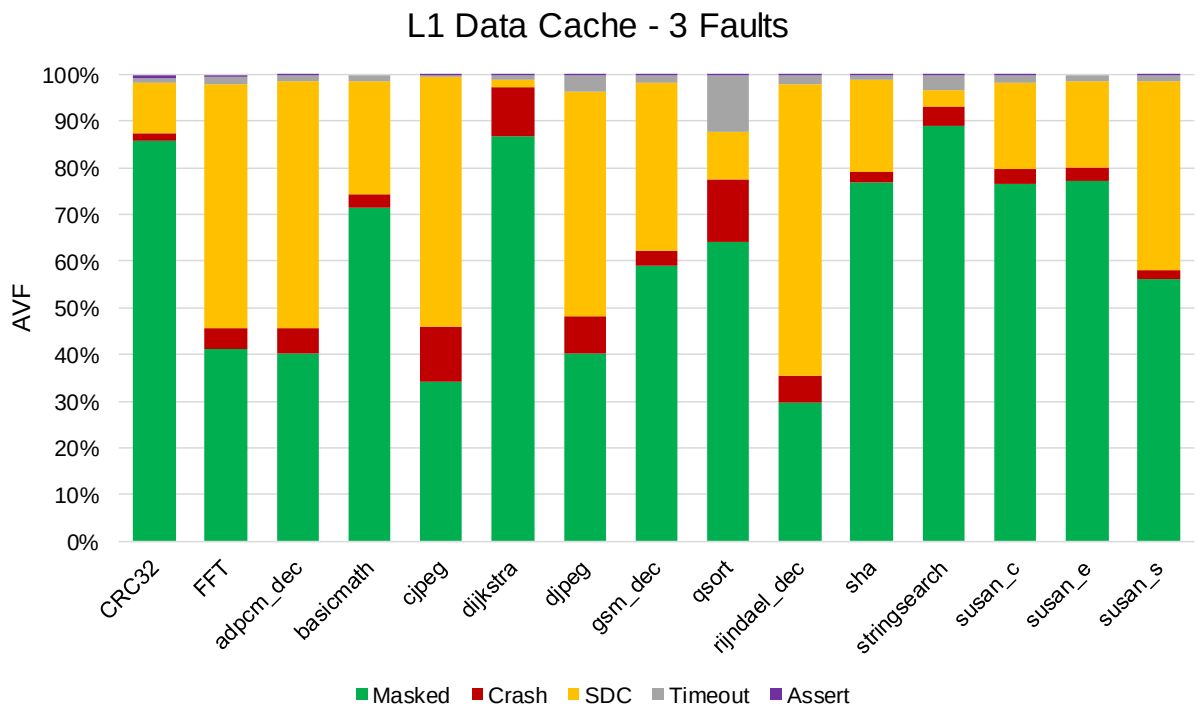


Figure 5.3: L1 D-Cache, three faults injected per run

The combined results for all the three cases (one, two, three faults injected per run) for all the benchmarks, for the L1 D-Cache appear in Figure 5.4:

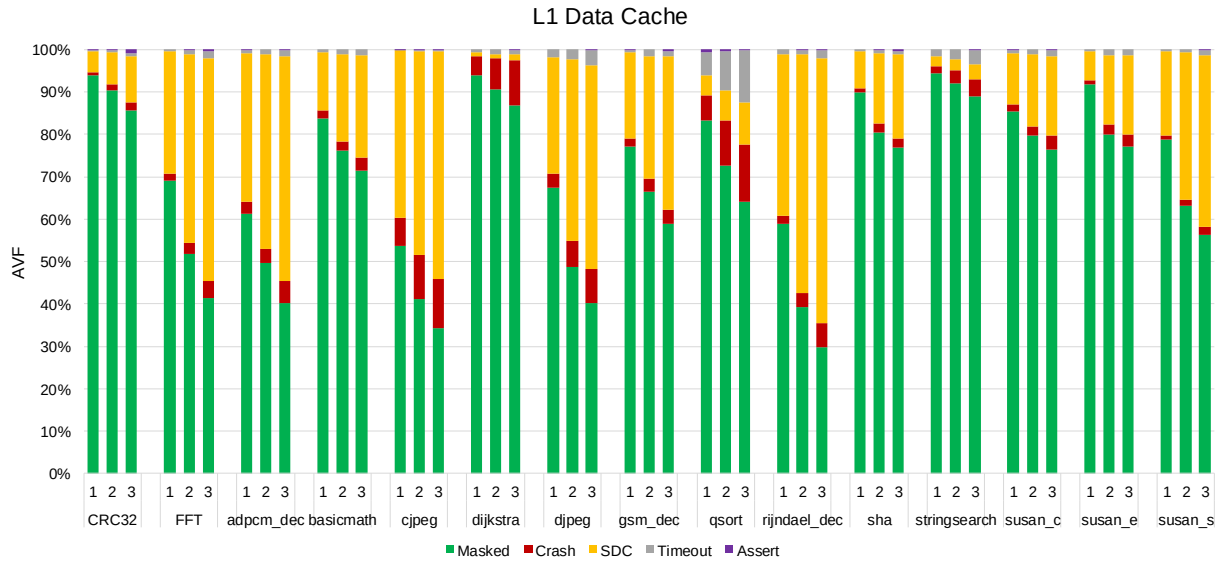


Figure 5.4: L1 D-Cache, combined results

5.2.2 Level 1 Instruction Cache

The results for the Level 1 Instruction Cache of the ARM Cortex-A9, for all three cases (one, two, three faults injected per run) are:

5.2.2.1 Single-bit fault injection per run

The results for the single-bit fault injection appear in Figure 5.5. The percentage for the Masked Class varies from 95.10% for the best case (susan_e) to 76.30% for the worst case (rijndael_dec). The main difference in the behavior between the L1 I-Cache and the L1 D-Cache appears if we examine the class with the prominent percentage after the Masked class. In the case of the L1 D-Cache the percentage remaining from the Masked class was mainly SDCs, while in the case of the I-Cache, the prominent faulty behavior seems to be a Crash. This is of course explained if we consider the type of each memory. The faults in the D-Cache tend to affect the data of each benchmark and that is why, we see data corruptions on the output, while on the other hand, on the I-Cache the faults affect the instructions which cause each benchmark to crash. The Crash percentage varies from 14.30% for the worst case (adpcm_dec) to 3.30% for the best case (susan_e) with six benchmarks having a rate bigger than 10%. For SDCs we have a percentage from 8.60% for the worst case (rijndael_dec) to 0.10% for the best case (stringsearch). The SDCs percentage are extremely low for most of the benchmarks. Likewise, the rates for the rest of the classes (Timeout & Assertion) are extremely low to non-existent in the case of the latter (Assertion).

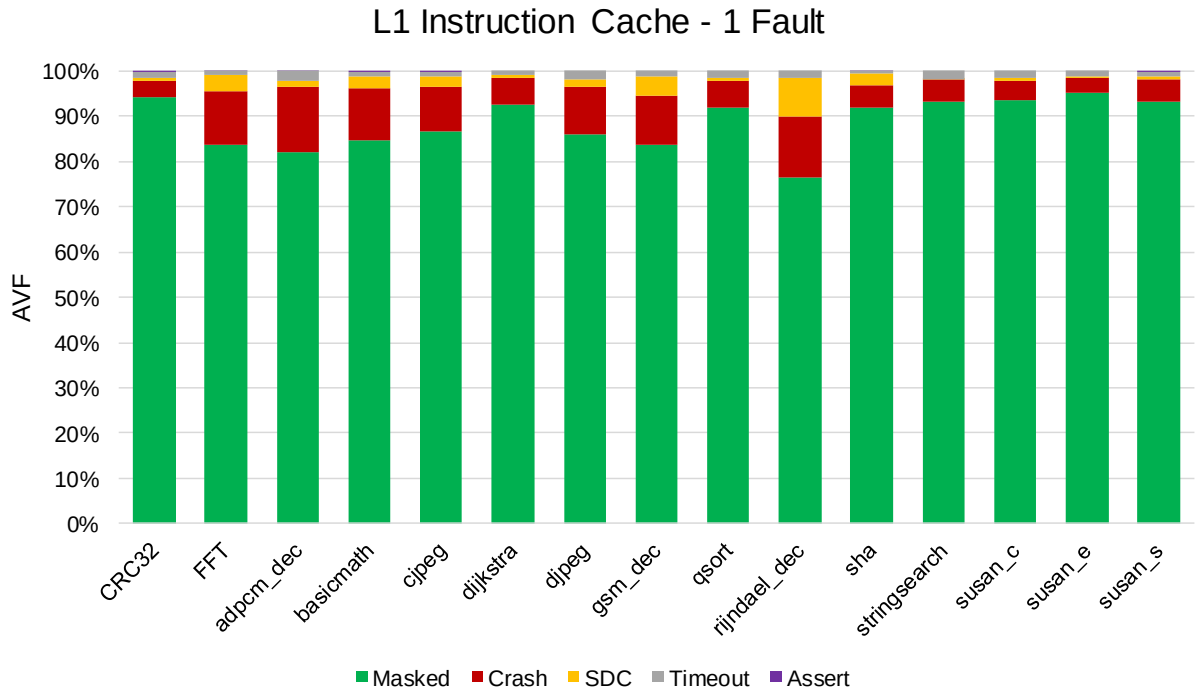


Figure 5.5: L1 I-Cache, one fault injected per run

5.2.2.2 Multi-bit (two bit) fault injection in adjacent bits, per run

The results from the double fault injection in adjacent bits appear in Figure 5.6. For the Masked class we have a percentage of 63% for the worst case (rijndael_dec) to 89.3% for the best case (susan_c). We can see that the percentage of Masked faults is much higher than the double fault injection on the D-Cache where we had four benchmarks with a percentage lower than 50% for the Masked class. Still, the rates for the Masked class are worse for the two-bit fault injection in the I-Cache for all the cases compared to the single bit fault injection, but the effect is not as severe as it was in the case of the

D-Cache. Inspite of that -like we mentioned- the I-Cache shows a bigger vulnerability to crashes. Crashes vary from a percentage of 23.40% for the worst case (rijndael_dec) to 7.20% for the best case (CRC32). SDCs have a percentage smaller than 10% for all cases except rijndael_dec with a rate of 11.10%. Assertion has extremely low values (under 1% for all cases), with highs in the case of CRC32 and cjpeg (0.90% and 0.70% assertions) while Timeout has rates smaller than 4% for all cases.

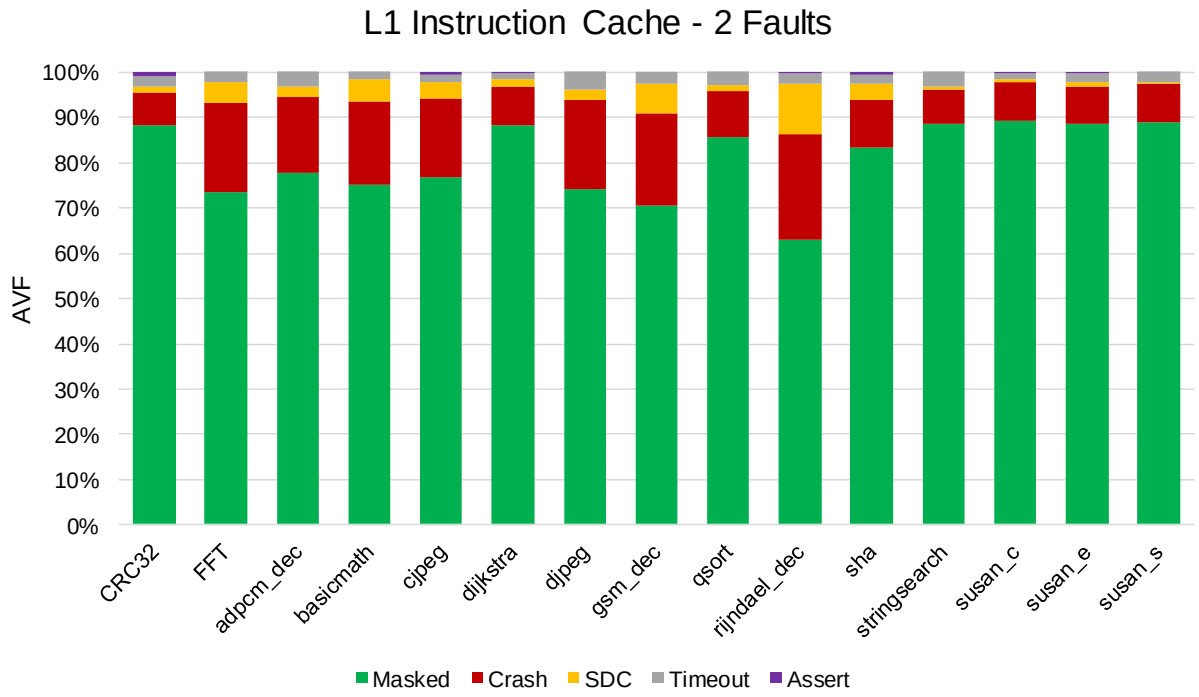


Figure 5.6: L1 I-Cache, two faults injected per run

5.2.2.3 Multi-bit (three bit) fault injection in adjacent bits, per run

The results from the triple fault injection in adjacent bits appear in Figure 5.7. For the Masked class we observe a variation from 55.40% for the worst case (rijndael_dec) to 86.70% for the best case (susan_c). Compared to single and double fault injection, we see a 20.9% and a 7.6% decrease respectively for the worst case and a 8.40% and 2.60% decrease respectively for the best case. Compared to the same scenario for the D-Cache, all the decreases are smaller, showing better behavior of the I-Cache to mask the faults. Although we still see the decrease of the Masked faults for each benchmark as we inject more adjacent faults per run, still no benchmark has a Masked rate smaller than 50%, which happened in the D-Cache even from the case of the two-bit fault injection. The Crash percentage for all benchmarks is further deteriorated in the case of three bits fault injection with a rate from 30.20% for the worst case (rijndael_dec) to 9.70% for the best case (susan_c). So, out of the 1000 simulations, at least 97 crashed for each benchmark, while in the case of rijndael_dec, 302 of them crashed. The deterioration in crashes seems to worsen with the more adjacent faults we inject. We see a deterioration of 2x for most benchmarks, compared to the single-bit fault injection, while in the benchmarks with a smaller percentage of crashes in the single-bit fault injection, we observe even a 3x worsening for their Crash rate (susan_e from 3.30% to 11.60%). SDCs vary from a 12.10% for the worst case (rijndael_dec) to 0.50% for the best case (stringsearch). All the cases except rijndael_dec are under 10%. Timeouts

reach a high of 4.80 for the worst case (adpcm_dec). Almost all the Assertions stay under 0.30%, while we observe the same behavior we observed in the D-Cache case for the CRC32 benchmark were 8 out of 1000 (0.8%) simulations lead to a simulator crash, a 4x compared to single-bit fault injection. Inspite of that we actually see a decrease for CRC32 and cjpeg Assertions compared to the double bit fault injection case (0.90% to 0.80% and 0.70% to 0.30%), which makes it difficult to make a definite assumption for the relationship between Assertions and the increasing number of faults injected per run.

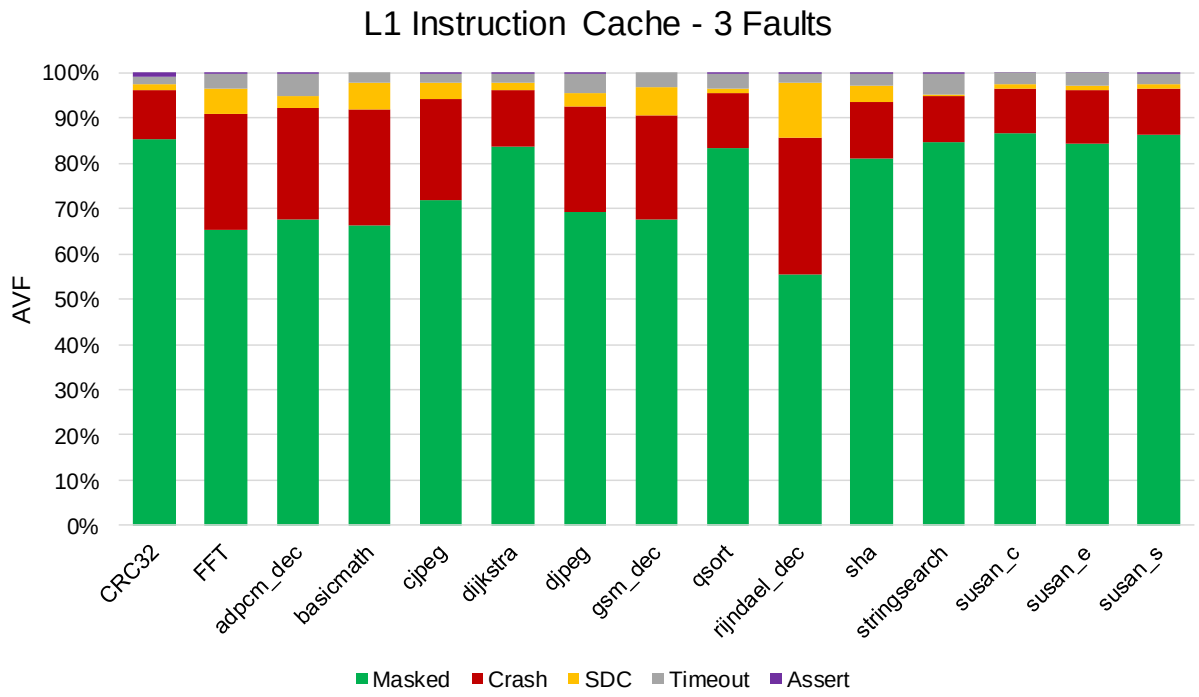


Figure 5.7: L1 I-Cache, three faults injected per run

The combined results for all the three cases (one, two, three faults injected per run) for all the benchmarks, for the L1 I-Cache appear in Figure 5.8:

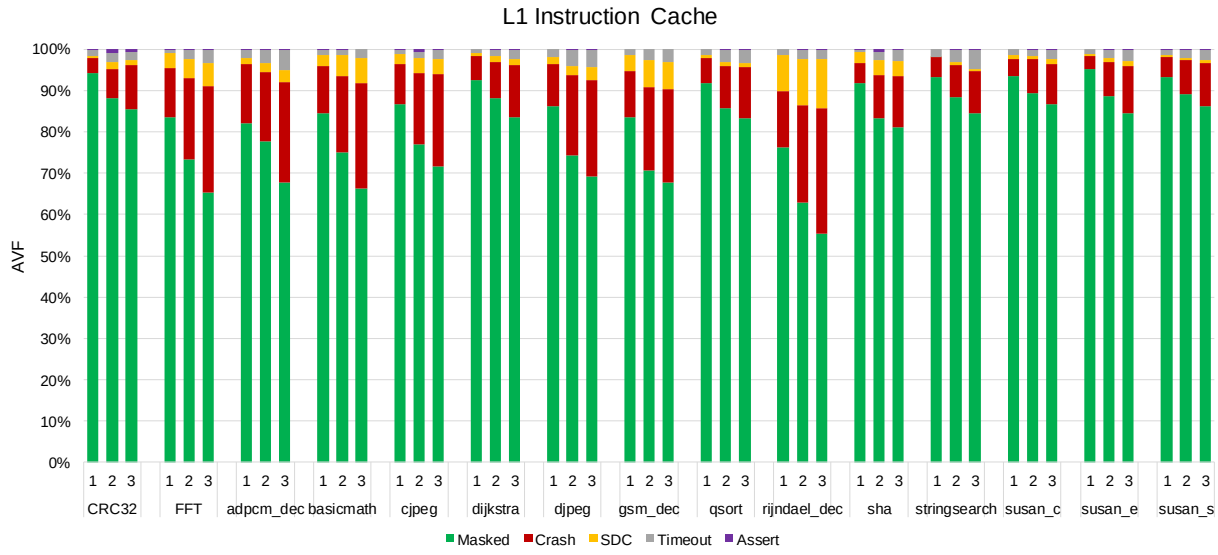


Figure 5.8: L1 I-Cache, combined results

5.2.3 Level 2 Cache

The results for the Level 2 Cache of the ARM Cortex-A9, for all three cases (one, two, three faults injected per run) are:

5.2.3.1 Single-bit fault injection per run

The results for the single-bit fault injection appear in Figure 5.9. The percentage for the Masked Class varies from 95.80% for the best case (stringsearch) to 52.20% for the worst case (adpcm_dec). The worst case is on a similar level with the D-Cache (53.7%) and much lower than the I-Cache (76.30%), for the same scenario, showing some level of correlation between the two. This argument is strengthened by the fact, that just like the D-Cache the highest fault percentages of the L2 Cache are SDCs. For the SDCs, we have a variation of 44.40% for the worst case (adpcm_dec) to just 0.10% for the best case (sha). SDCs are the main fault type (besides Masked faults) for 6 out of 15 benchmarks while for the rest the main fault is a Crash. Crashes vary from a 4.70% for the worst case (susan_s) to 1.90% for the best case (rinjndael_dec). Timeouts and Assertions all have rates under 1.60% with Assertions having rates as low as 0% for three benchmarks. FFT seems to be the only benchmark with more than 1-4 assertions, with a rate of 0.6% (6 out of 1000 runs crashed the simulator).

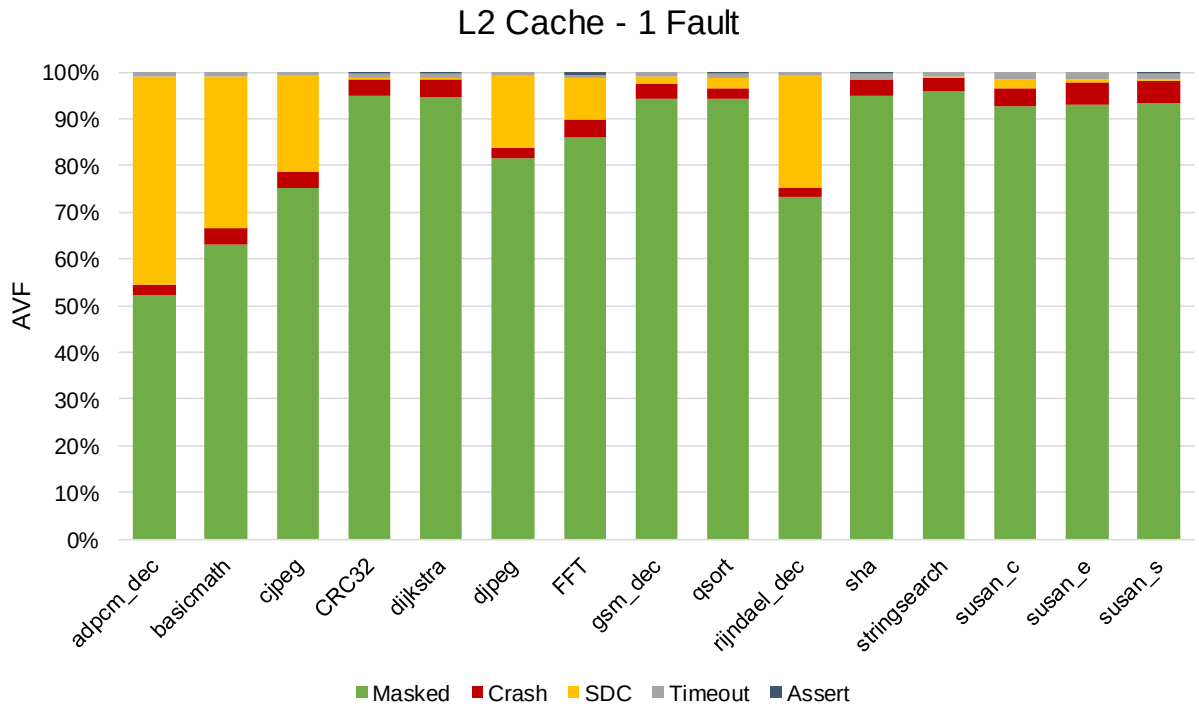


Figure 5.9: L2 Cache, one fault injected per run

5.2.3.2 Multi-bit (two bit) fault injection in adjacent bits, per run

The results from the double fault injection in adjacent bits appear in Figure 5.10. For the Masked class we have a percentage of 91.70% for the best case (sha) to 40% for the worst case (adpcm_dec), showing a variation of over 50% between those corner cases. Benchmark adpcm_dec is the only one with a masked class rate lower than 50%, much better than the D-cache where we had four benchmarks with a rate lower than 50% for the same scenario and worse than I-Cache where no benchmarks had a rate of under

50%. SDCs remain the most severe fault class, with a rate from 54.40% for the worst case (adpcm_dec) to 0% for the best case (susan_s). This actually shows a decrease in SDCs for susan_s (from 0.20% to 0%) for the two-bits fault injection, compared to the single-bit case. Crashes vary from a 9.60% for the worst case (susan_s) to 3.30% for the best case (qsort). Therefore, the complete nullification of SDCs in the case of susan_s lead to a 2x amount of crashes (from 4.70% to 9.60%). Timeouts vary from 2.00% for the worst case (qsort and susan_c) to 0.75% for the best case (cjpeg). FFT remains the worst case for Assertions with a percentage of 1.20%, a 2x from the single-bit case.

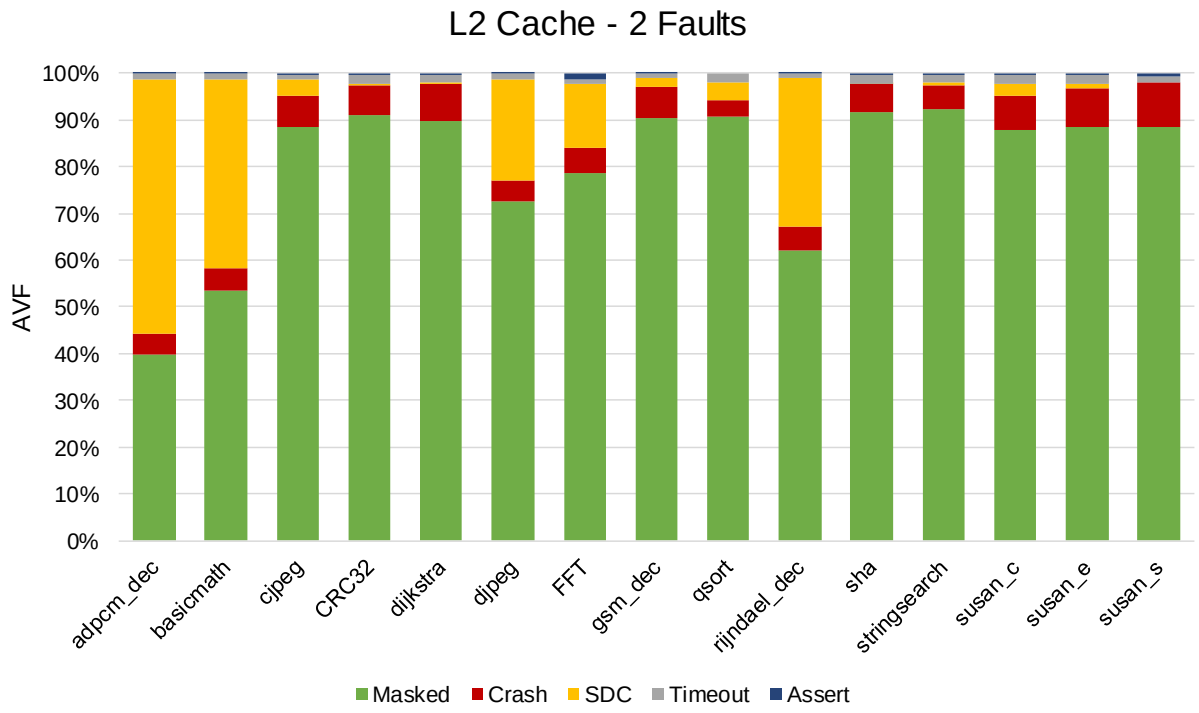


Figure 5.10: L2 Cache, two faults injected per run

5.2.3.3 Multi-bit (three bit) fault injection in adjacent bits, per run

The results from the triple fault injection in adjacent bits appear in Figure 5.11. For the Masked class we observe a variation from 30.2% for the worst case (adpcm_dec) to 89.80% for the best case (stringsearch). Compared to single and double fault injection, we see a 22% and a 9.8% decrease respectively for the worst case and a 6% and 1.9% decrease respectively for the best case. The 22% decrease between the single and triple fault injection, is the worst among all components showing that the L2 Cache has the worst response to multi-bi fault injection. Compared to the same scenario for the D-Cache and the I-Cache, we see similar results with the D-Cache making I-Cache the best one thus far when it comes to fault masking behavior. We see only 2 benchmarks out of 15, with a rate smaller than 50% for the Masked class, a far better result than the case of the D-Cache (where five benchmarks had a Masked class rate, smaller than 50%). SDCs vary from a 62.10% for the worst case (adpcm_dec) to just 0.2% for the best case (susan_s). Crashes vary from a 10.90% for the worst case (susan_s) to 4.40% for the best case (qsort). Timeouts have rates from 2.70% for the worst case (qsort) to 1% for the best case (adpcm_dec). Lastly, Assertions continue holding

negligible rates, with a maximum of 0.5% for FFT, a decrease from the double- and single-bit case.

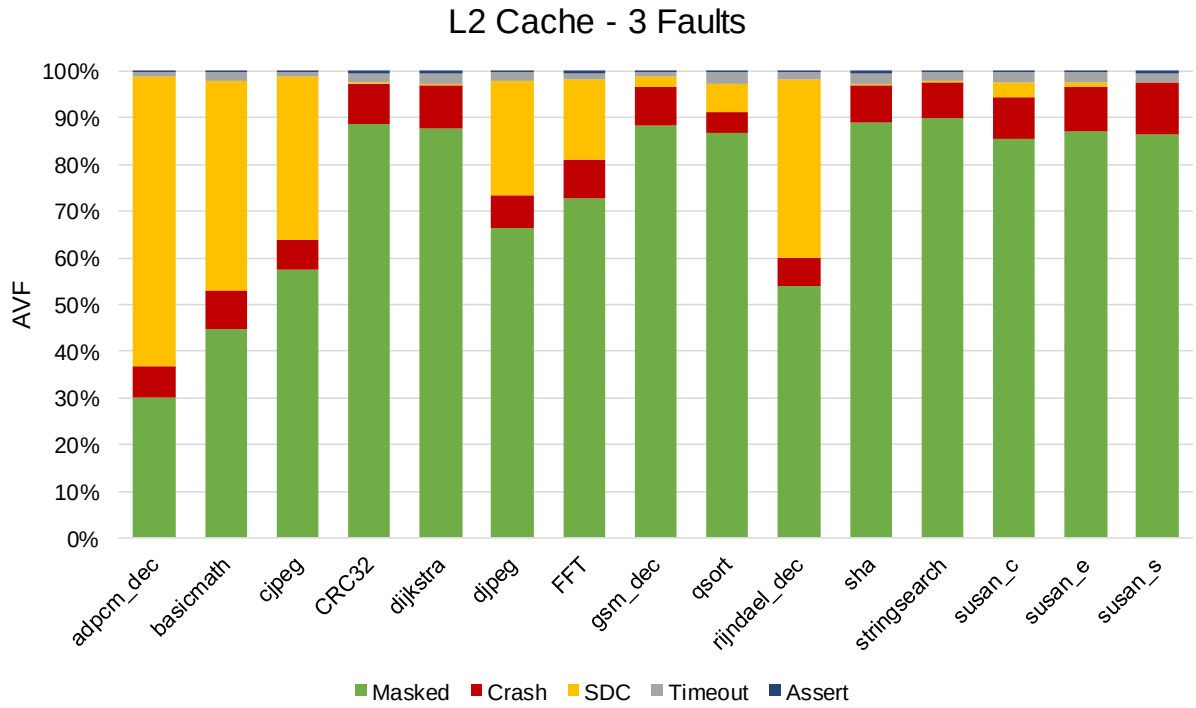


Figure 5.11: L2 Cache, three faults injected per run

The combined results for all the three cases (one, two, three faults injected per run) for all the benchmarks, for the L2 Cache appear in Figure 5.12:

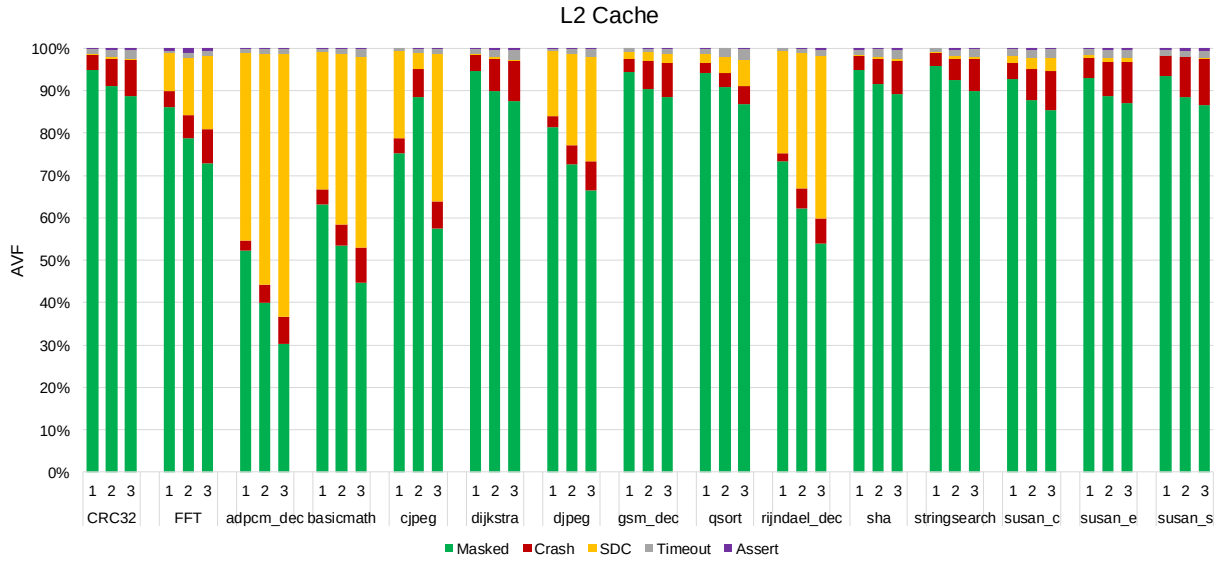


Figure 5.12: L2 Cache, combined results

5.2.4 Register File

The results for the Register File of the ARM Cortex-A9, for all three cases (one, two, three faults injected per run) are:

5.2.4.1 Single-bit fault injection per run

The results for the single-bit fault injection appear in Figure 5.13. The percentage for the Masked Class varies from 94.40% for the best case (susan_c) to 84.20% for the worst case (sha). This signifies the best “worst case” rate for masked faults compared to all the other components for the same scenario, showing that the Register File has the best capability for fault masking in the case of single bit fault injection. The second most prominent fault class in the case of the Register File, appears to be the Crash class similar to the I-Cache, for most of the cases, although unlike the I-Cache -for some cases- SDCs dominate. Crashes vary from a 9.30% for the worst case (susan_s) to 3.10% for the best case (stringsearch). SDCs vary from a 11% for the worst case (sha) to 0.5% for the best case (susan_c). The rest of the classes (Timeouts and Assertions) hold very small rates, with Timeouts reaching a high of 2.40% (stringsearch) while Assertions hold a 0% for 13 out of 15 cases.

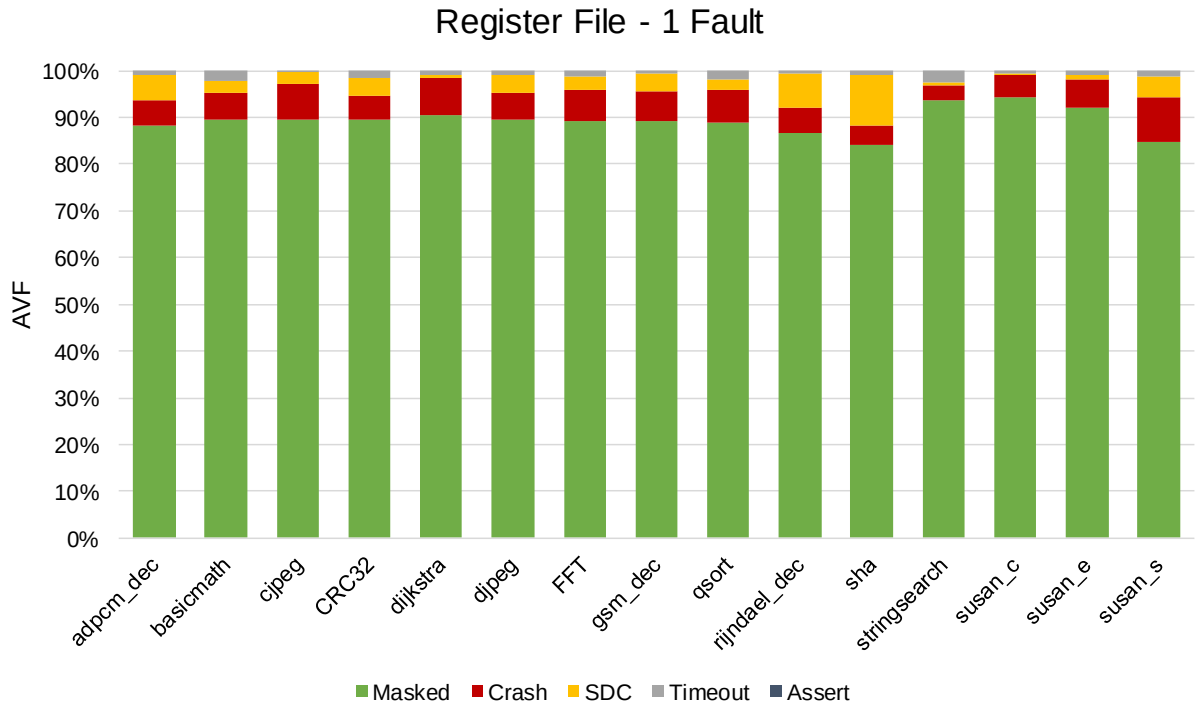


Figure 5.13: Register File, one fault injected per run

5.2.4.2 Multi-bit (two bit) fault injection in adjacent bits, per run

The results from the double fault injection in adjacent bits appear in Figure 5.14. For the Masked class we have a percentage of 89.7% for the best case (stringsearch) to 74% for the worst case (sha). No benchmark has a worst-case rate for the Masked class lower than 50% like in the case of L1 I-Cache. Still, in 13 out of 15 benchmarks, Crashes are the second most often fault type, with a worst-case percentage of 14.90% (susan_s) and a best-case percentage of 5% (stringsearch). SDCs vary from a 15.50%

for the worst-case (sha) to 0.70% for the best case (stringsearch). Timeouts vary from 4.50% for the worst case (stringsearch) to 1.30% for the best case (FFT). We also have 8 out of 15 benchmarks with no assertions (a decrease from the single-bit case) with a high of 3 (out of 1000) assertions for susan_c.

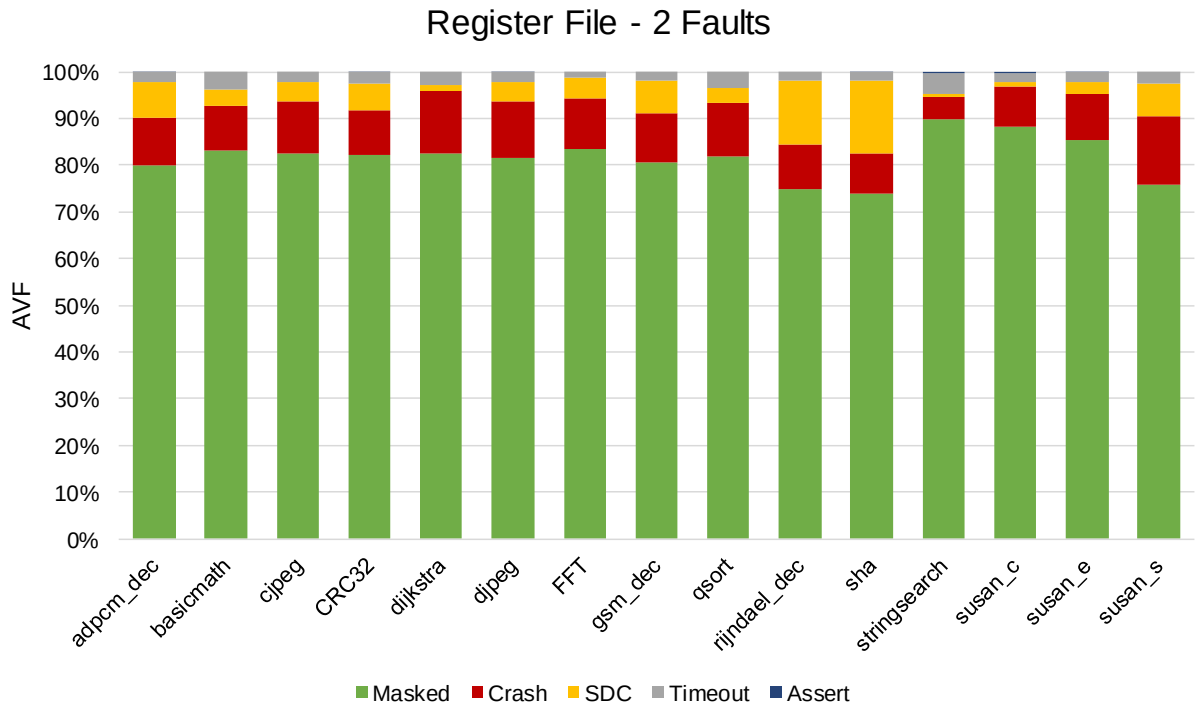


Figure 5.14: Register File, two faults injected per run

5.2.4.3 Multi-bit (three bit) fault injection in adjacent bits, per run

The results from the triple fault injection in adjacent bits appear in Figure 5.15. For the Masked class we observe a variation from 69.10% for the worst case (sha) to 85.50% for the best case (stringsearch). Compared to single and double fault injection, we see a 15.10% and a 4.90% decrease respectively for the worst case and a 8.9% and 4.20% decrease respectively for the best case. Compared to the same scenario for the D-Cache, the I-Cache and the L2 Cache, we see that the Register File has the smaller decreases for the worst case, showing the smallest effect of multi-bit fault injection between the four components. The 69.10% rate for the worst case of the Masked class is also the best among the four components for the same scenario, showing that the Register File masks the faults in the most efficient manner. The second most often fault is Crash in 12 out of 15 benchmarks. Crashes vary from a 18.40% for the worst case (susan_s) to 7.70% for the best case (stringsearch) with 14 out of 15 benchmarks having rates larger than 10%. SDCs vary from 18.50% for the worst case (sha) to 0.80% for the best case (susan_c). Sha now has a smaller rate compared to both the two-bit and single-bit case (from 1.10% and 0.50 to 0.80%). Also, we have smaller rates compared to the two-bit case for qsort (3.1% to 2.90%), susan_c (1,1% to 0.8%), susan_e (2.4% to 2.3%) and cjpeg (4.1% to 3.6) while rijndael_dec stays the same (13.8%) although we see increases in the SDCs for 10 out of 15 benchmarks compared to the two-bit case. Timeouts vary from 5.70% for the worst-case (stringsearch) to 2.1%

for the best case (gsm_dec). We also have 4 out of 15 benchmarks with no Assertions. Assertions reach a high of 0.3% for stringsearch.

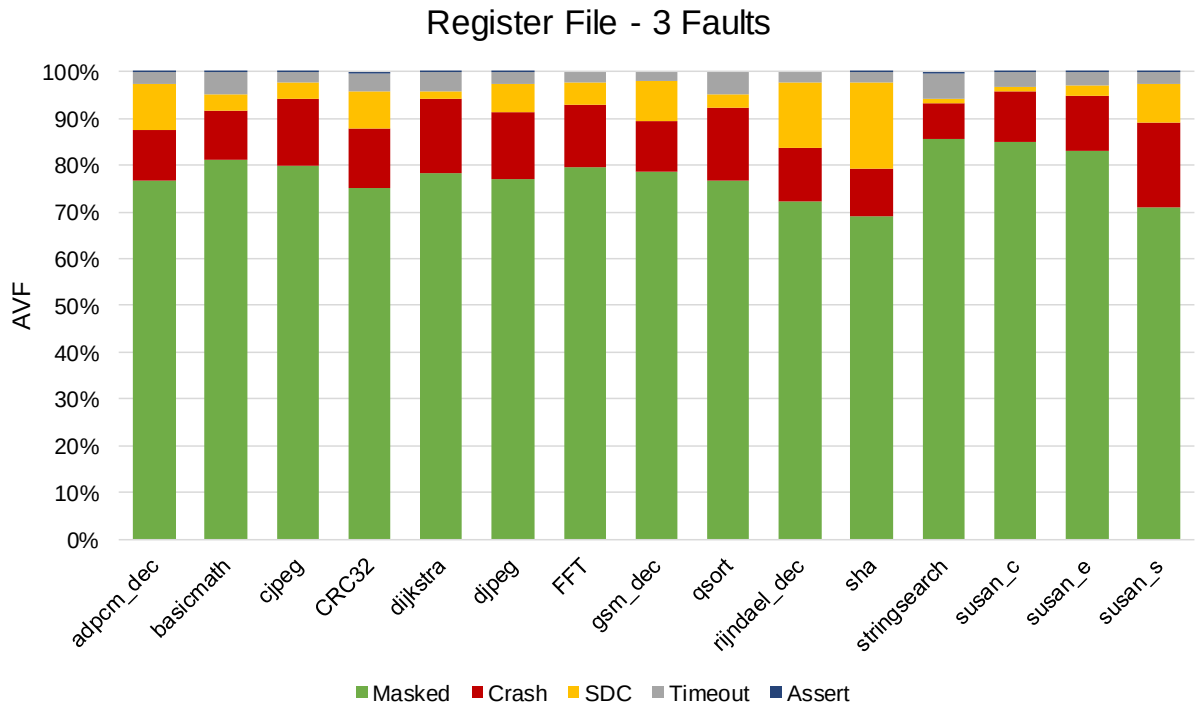


Figure 5.15: Register File, three faults injected per run

The combined results for all the three cases (one, two, three faults injected per run) for all the benchmarks, for the Register File appear in Figure 5.16:

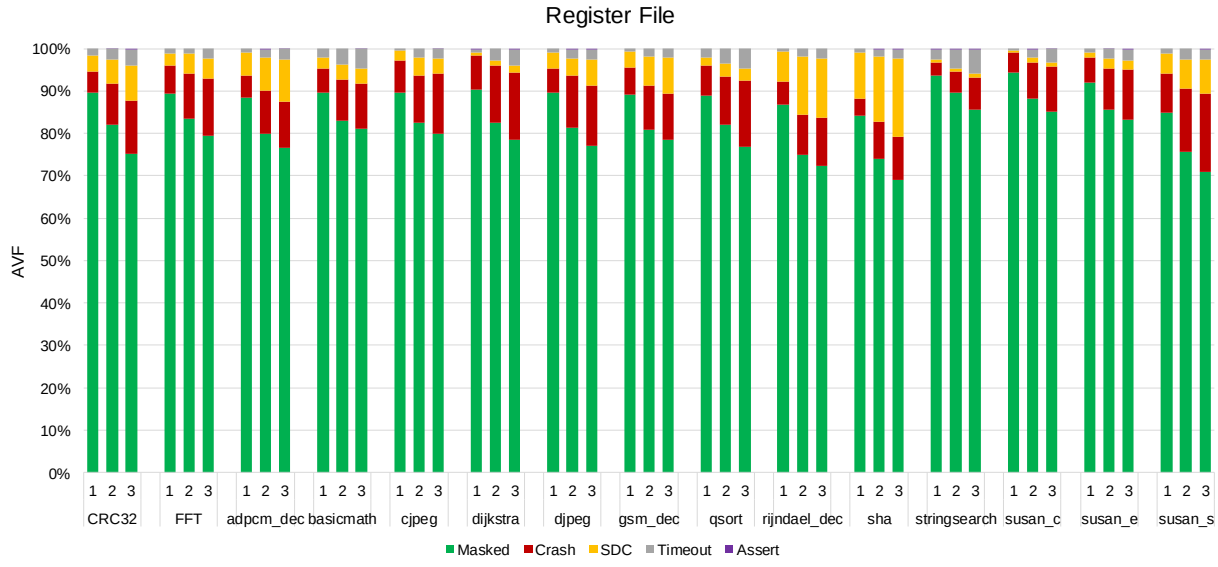


Figure 5.16: Register File, combined results

5.2.5 Data Translation Lookaside Buffer

The results for the Data Translation Lookaside Buffer (DTLB) of the ARM Cortex-A9, for all three cases (one, two, three faults injected per run) are:

5.2.5.1 Single-bit fault injection per run

The results for the single fault injection appear in Figure 5.17. For the Masked class we have a variation from 42.40% for the worst case (basicmath) to 60.80% for the best case (qsort). These signify the worst rates for the same scenario compared to all the other components, thus far. We see that 7 out of the 15 benchmarks have a Masked rate lower than 50%. The second most common fault class in the case of the TLB appears to be the Crashes for most of the cases (14 out of 15 benchmarks). Crashes vary from 32.80% for the worst case (FFT) to 16.7% for the best case (qsort). High rates for Timeouts are also observed with a worst case of 23.10% (stringsearch) and a best case of 6.30% (cjpeg). SDCs have the 4th smaller rates for most cases with a worst-case rate of 17.10% (basicmath) and a best-case rate of just 1.50% (sha). 9 out of 15 benchmarks have an SDC rate lower than 10%. For the DTLB we also see “high” rates for Assertions, the highest among all components where such events were extremely rare, with a high of 3.50% (35 out of 1000 simulations led to a simulator crash) for CRC32.

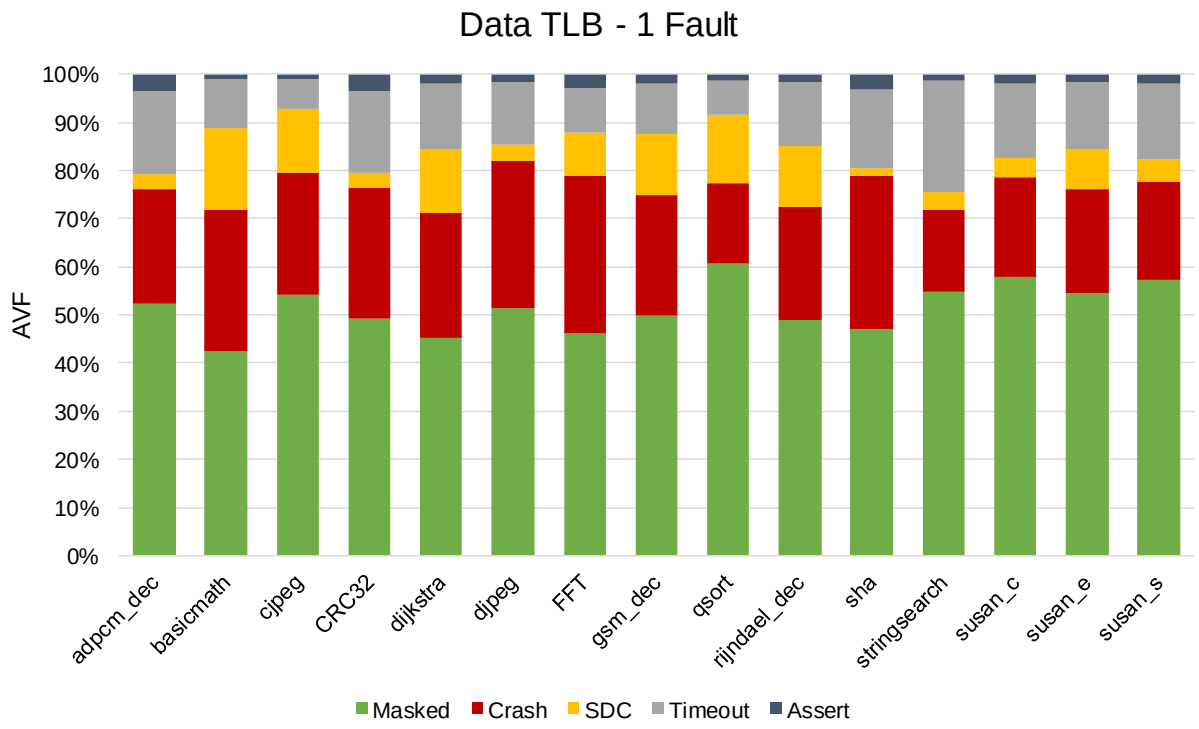


Figure 5.17: DTLB, one fault injected per run

5.2.5.2 Multi-bit (two bit) fault injection in adjacent bits, per run

The results for the double fault injection appear in Figure 5.18. For the Masked class we see a variation from 32.10% for the worst case (basicmath) to 49.70% for the best case (qsort). We observe a deterioration with the increasing number of faults injected like in all the other components but most importantly we observe that all the benchmarks (15

out of 15) have a Masked rate lower than 50%, something that didn't happen for any other component even for a bigger number (three) of faults injected per run. So, DTLB appears to have the worst effect to fault injecting, thus far. For the Crash class, a variation of 37.20% (djpeg) for the worst case to 16.20% for the best case (stringsearch), is observed. Timeouts vary from 32.80% for the worst case (stringsearch) to 10.20% for the best case (cjpeg). SDCs vary from 14.10% for the worst case (basimath and qsort) to 0.60% for the best case (sha). SDCs have a better behavior for the increasing number of faults injected, compared to the single fault case with most of the benchmarks (14 out of 15) showing a decrease in SDC rates. Assertions vary from 8.60% for the worst case (CRC32) to 2.30% for the best case (stringsearch), showing a doubling or even more, for most of the cases, compared to the single fault injection.

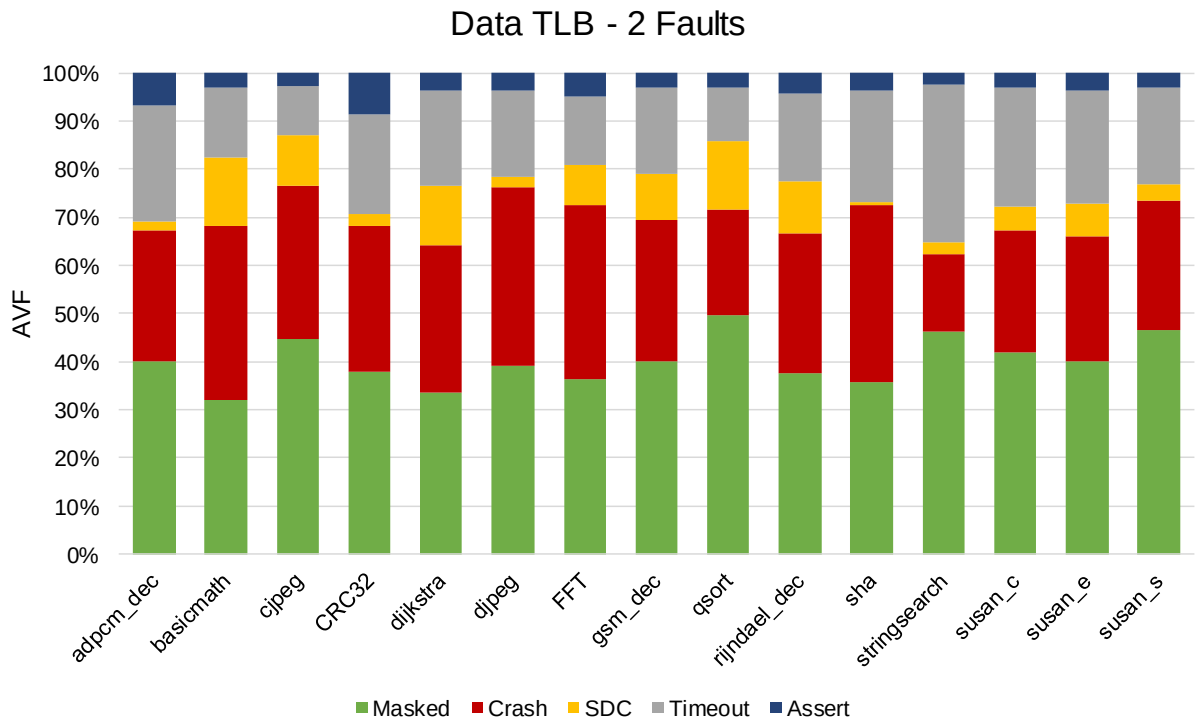


Figure 5.18: DTLB, two faults injected per run

5.2.5.3 Multi-bit (three bit) fault injection in adjacent bits, per run

The results from the triple fault injection in adjacent bits appear in Figure 5.19. For the Masked class we observe a variation from 28.50% for the worst case (basimath) to 44.10% for the best case (qsort). Compared to single and double fault injection, we see a 13.90% and a 3.6% decrease respectively for the worst case and a 16.7% and 5.6% decrease respectively for the best case. This signifies the smaller drop between single and double/triple fault injection for the worst case, compared to the other components, thus far. Therefore, the DTLB seems to have the worst Masked rates even from the first phase of the injection (single fault), but the effect of the multi-bit injection is not that strong. Crashes vary from 41.80% for the worst-case (basicmath) to 19.80% for the best case (stringsearch). Timeouts vary from 37.50% for the worst case (stringsearch) to 14.80% for the best case (cjpeg). SDCs vary from 13.40% for the worst-case (qsort) to 0.70% for the best case (sha). We observe a similar trend for improved SDC rates as

we increase the faults injected, with 11 out of 15 benchmarks having smaller rates compared to the double fault case. Like we mentioned, the DTLB shows by far the worst rates for Assertions compared to all the other components. In this case we see a variation from 8.90% for the worst-case (CRC32) to 2.50% for the best case (cjpeg). So, Assertions have a bigger rate than SDCs for 6 out of 15 benchmarks. Inspite of that, the effect of the transition from two to three faults is not strong, with only a small increase in most of the cases.

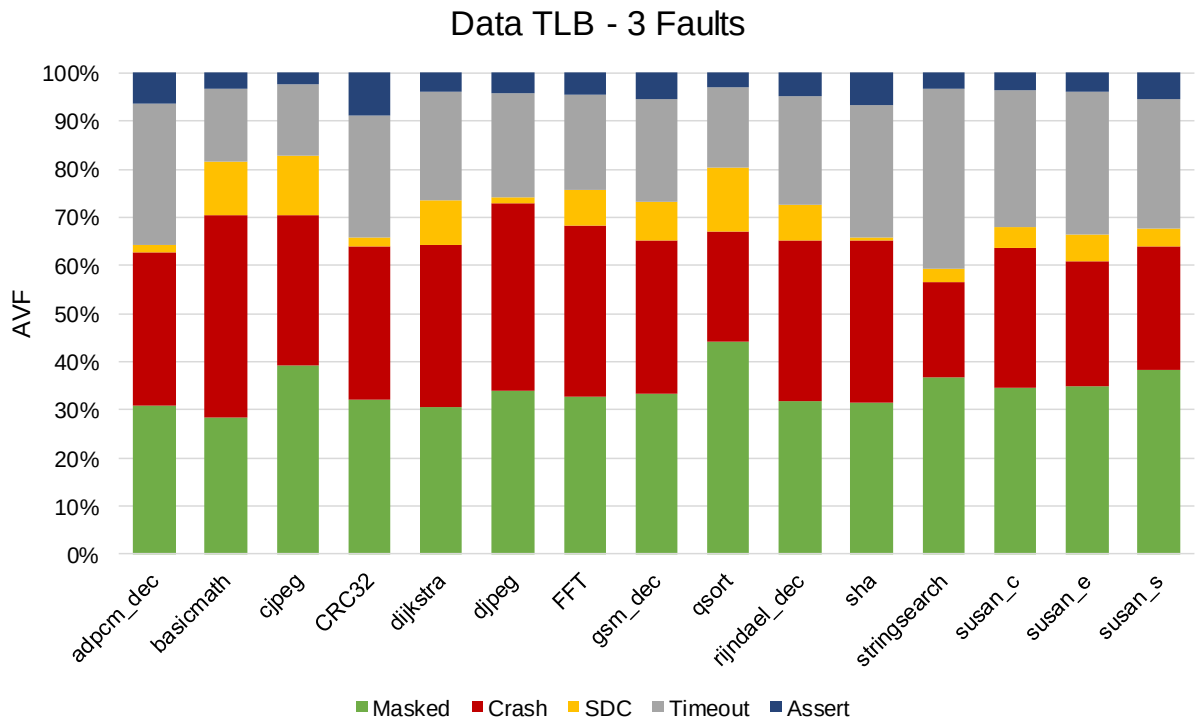


Figure 5.19: DTLB, three faults injected per run

The combined results for all the three cases (one, two, three faults injected per run) for all the benchmarks, for the DTLB appear in Figure 5.20:

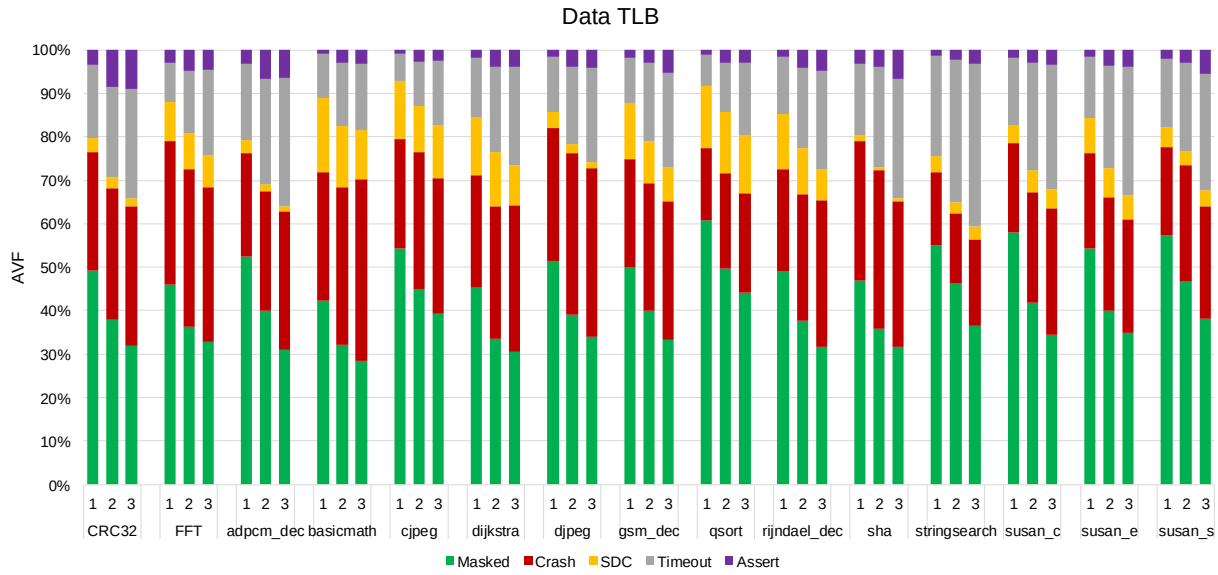


Figure 5.20: DTLB, combined results

5.2.6 Instruction Translation Lookaside Buffer

The results for the Instruction Translation Lookaside Buffer (ITLB) of the ARM Cortex-A9, for all three cases (one, two, three faults injected per run) are:

5.2.6.1 Single-bit fault injection per run

The results for the single fault injection appear in Figure 5.21. For the Masked class we see a variation from 58.1% for the best case (rijndael_dec) to 38% for the worst case (basicmath). We observe that similarly to the DTLB, 6 out of 12 benchmarks have a Masked rate lower than 50%. The second most common fault class appears to be the Crash for most of the cases. A rate of 43.10% for the worst case (djpeg) and just 1.50% (stringsearch) for the best case is observed. But for this benchmark we can find the worst case for the Timeout class with a rate of 44.20% (the best case is 16.20% for qsort). So, the faults, lead to either a crash of the benchmark because of a virtual address in the TLB that does not correspond to a physical address or to a Timeout, possibly in the case where a valid but irrelevant (to the program) physical page is fetched from the virtual memory. Because ITLB is relevant to instructions, we can observe non-existent rates for the SDC class. Most of the benchmarks (10 out of 15) have zero SDCs, while the maximum rate is just 0.40% (4 out of 1000 runs led to a SDC) for sha. Assertions also hold small rates, with a maximum of 1.30% for CRC32.

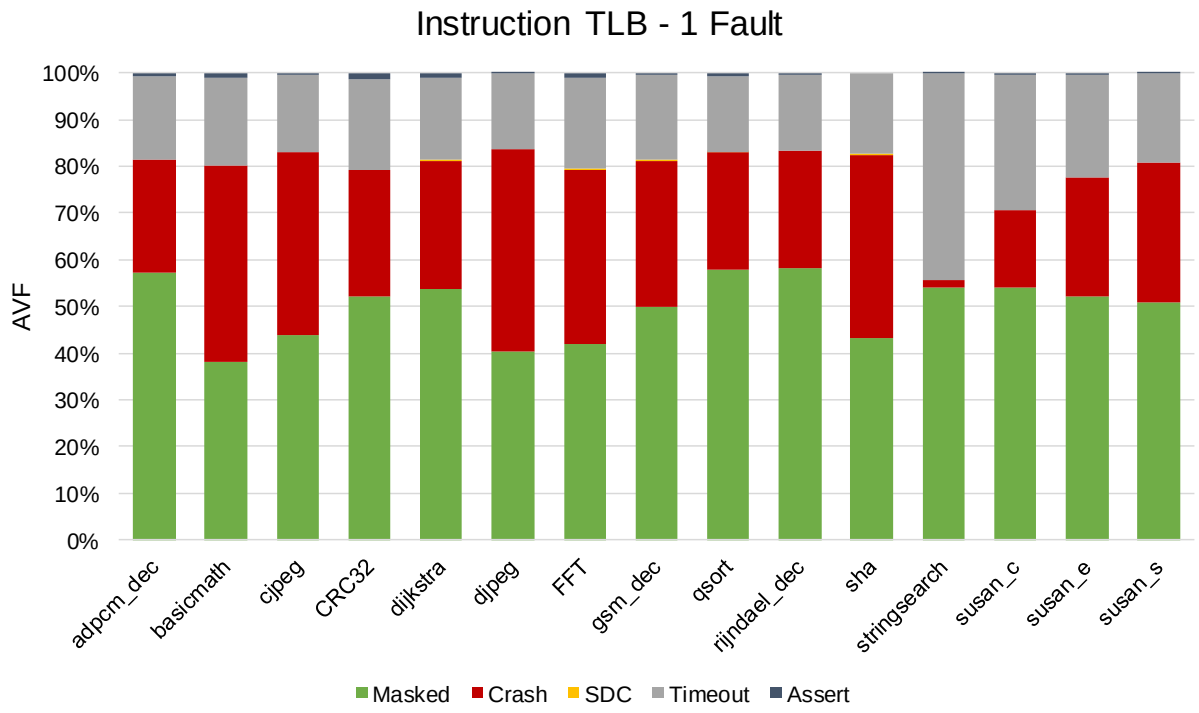


Figure 5.21: ITLB, one fault injected per run

5.2.6.2 Multi-bit (two bit) fault injection in adjacent bits, per run

The results for the double fault injection appear in Figure 5.22. Just like we observed for the DTLB case, all the benchmarks have a Masked class rate lower than 50% with the worst one being 30.20% (basicmath) and the best one being 46% (rijndael_dec). This rate signifies the worst Masked class rate among all other components for the same scenario, showing the significant effects of fault injection in the ITLB. Crashes vary from 44.30% for the worst case (djpeg) to 1.50% for the best case (stringsearch), while Timeouts vary from 55.50% for the worst case (stringsearch) to 22.10% for the best case (djpeg). Therefore, we observe that the best case of Crashes is the worst case for Timeouts and vice-versa. SDCs, still hold negligible rates while Assertions reach a high of 2.10% for CRC32.

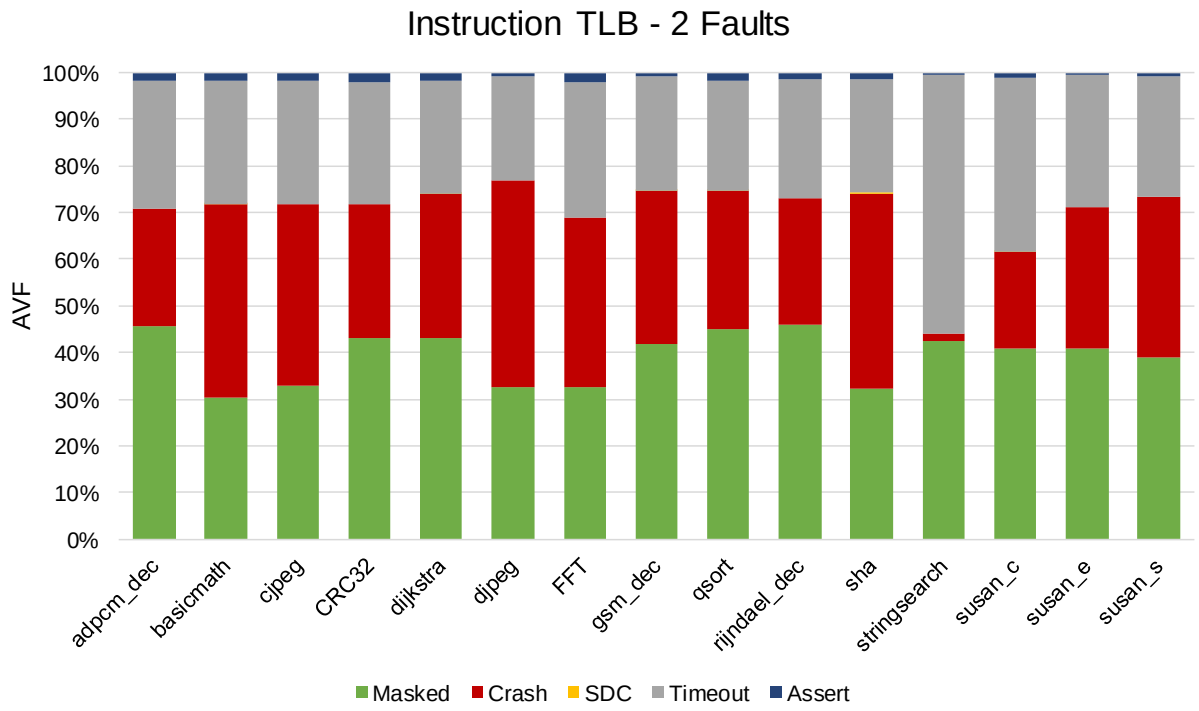


Figure 5.22: ITLB, two faults injected per run

5.2.6.3 Multi-bit (three bit) fault injection in adjacent bits, per run

The results for the triple fault injection appear in Figure 5.23. For the Masked class we observe a percentage of 25.10% for the worst case (basimath), to 40.80% for the best case (adpcm_dec). All the rates are smaller than 41% while 25.10% for basicmath is the worst Masked Class rate for all the experiments we observed for all components, meaning that in this case only ¼ of the benchmark runs were executed correctly. Compared to single and double fault injection, we see a 12.90% and a 5.10% decrease respectively for the worst case and a 17.30% and 5.2% decrease respectively for the best case. This signifies the smallest one-to-three faults decrease between all components, showing -just like in the case of the DTLB- that this type of component is affected very quickly and significantly (with just one fault) by fault injection but the effect of the multi-bit injection is not as remarkable as other components were we observed higher Masked fault rates at the first stages of fault injection. Crash rates vary from 46.40% for the worst case (basimath) to 1.40% for the best case (stringsearch), while Timeouts vary from 62.80% for the worst case (stringsearch) to 25.40% for the best case (basimath). SDCs are non-existent (only 1 SDC for one benchmark amongst all in 15.000 runs!) showing (almost) complete immunity of the ITLB to this kind of errors. Assertions appear higher for this case, with a high of 3.40% for CRC32 (an almost 3x compared to the single fault case).

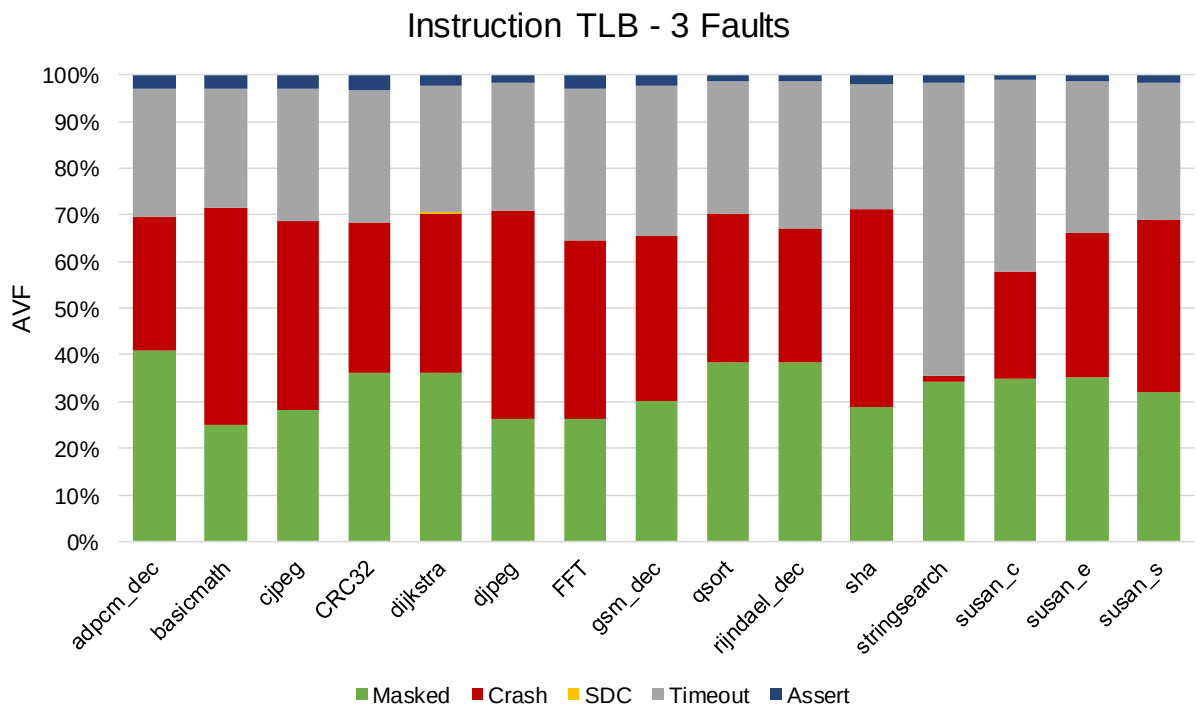


Figure 5.23: ITLB, three faults injected per run

The combined results for all the three cases (one, two, three faults injected per run) for all the benchmarks, for the ITLB appear in Figure 5.24:

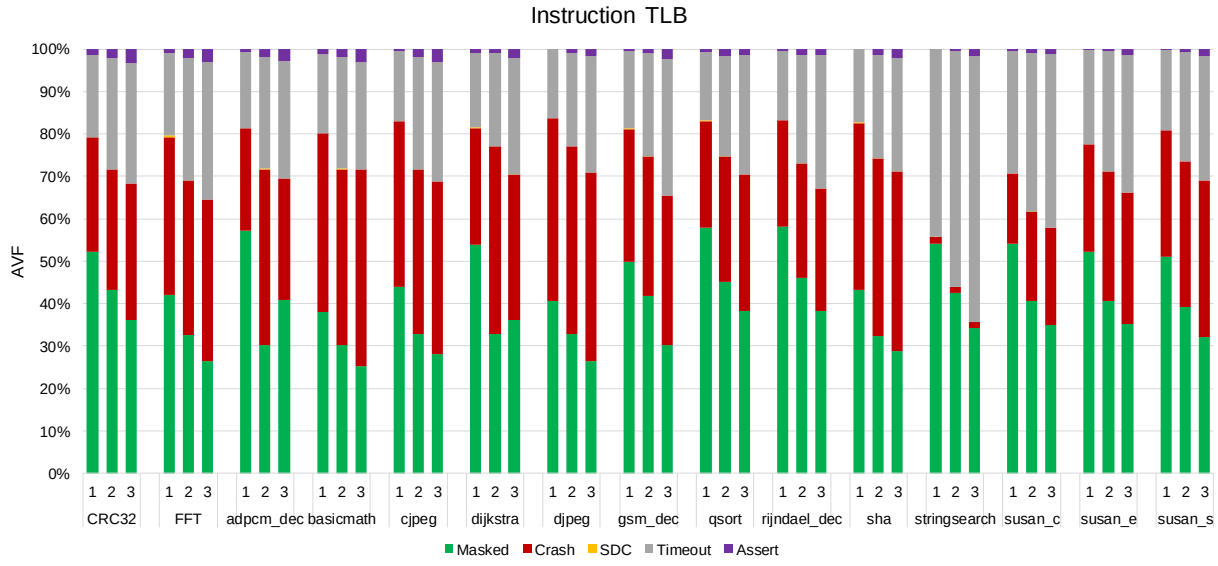


Figure 5.24: ITLB, combined results

5.2.7 Total AVFs per component and technology node

In the previous few sections, we observed how each of the six components react to fault injection in a few cases namely: a single fault injection and a two- or three-bit fault injection in adjacent areas (a 3x3 cluster) of each structure. We observed that the faults

affect each structure differently depending on their nature and cause different kinds of upsets. A staggering 25% of correct executions of a benchmark among 1000 executions was observed in one case showing the tremendous effects a few adjacent faults can have on a structure. These rates we found for each benchmark correspond to an Architectural Vulnerability Factor (see Chapter 1). It is the sum of the rates of the non-Masked classes that lead to the AVF of a benchmark, for a single fault injection case (one, two- or three-bits injection) for a specific component [18]. The AVF for each fault injection case, for each benchmark of each component was calculated from the results. The average of the AVFs for one injection case (of one component) was calculated by summing the AVFs of all benchmarks and dividing by the number of the benchmarks:

$$AVF_{\text{of (1,2,3) faults for X component}} = [AVF_{\text{(1,2,3) faults/benchmark1}} + \ldots + AVF_{\text{(1,2,3) faults/benchmark15}}] / 15$$

Those independent of technology node AVFs for 1,2,3 faults for each component appear in Table 5.1 and in Figure 5.25. On those, one can also see the percentage change (increase) in the AVF of a component between the 1-fault case and the 2-fault case and between the 2-fault and 3-fault cases:

Table 5.1: AVF per component for 1, 2, 3 faults injected

Component	Number of Faults Injected	AVF	*Percentage Change*
L1 D Cache	1	21.17%	-
	2	31.85%	*+10.68%*
	3	38.11%	*+6.27%*
L1 I Cache	1	11.47%	-
	2	19.22%	*+7.75%*
	3	24.11%	*+4.89%*
L2 Cache	1	14.68%	-
	2	21.01%	*+6.33%*
	3	25.66%	*+4.64%*
Register File	1	10.63%	-
	2	18.28%	*+7.65%*
	3	22.04%	*+3.77%*
ITLB	1	50.11%	-
	2	62.51%	*+12.40%*
	3	67.28%	*+4.77%*
DTLB	1	48.50%	-
	2	59.89%	*+11.39%*
	3	65.81%	*+5.92%*

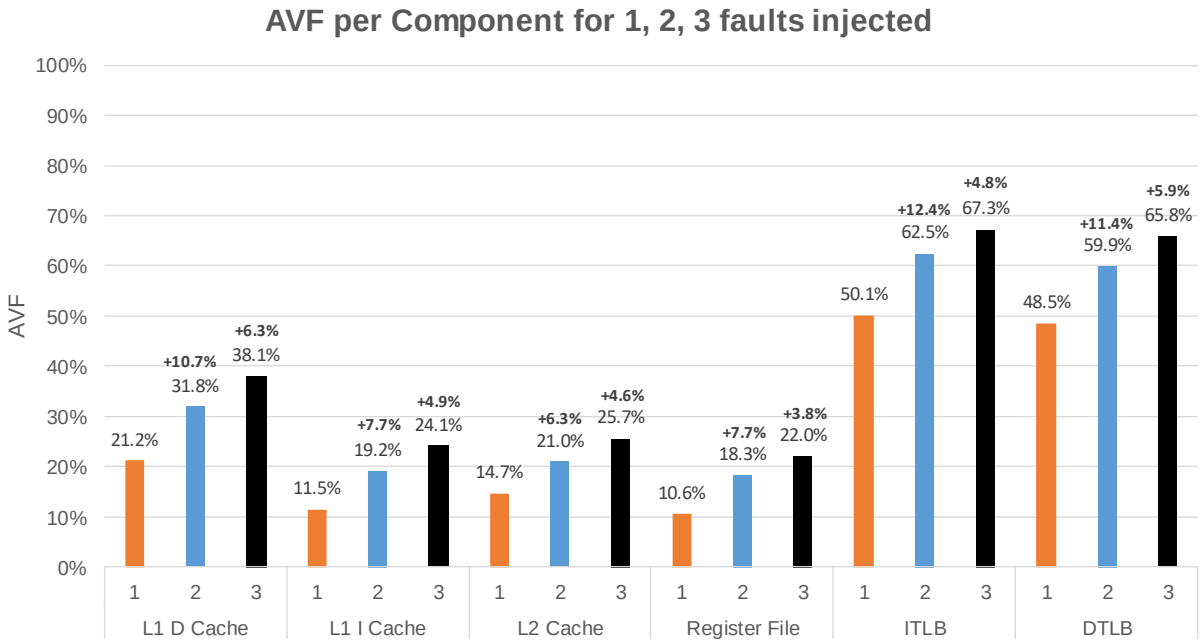


Figure 5.25: AVF per component for 1, 2, 3 faults injected

For these, we can observe that the TLBs hold the biggest AVF rates. These rates are increased as we increase the faults injected, leading to a 68% AVF for ITLB for 3 faults injected. The TLBs have larger AVF rates than the other components, even for single fault injection. So, we can conclude that the possibility of a fault in the TLB creating an error is always greater than (almost) 50%. The Register File has the smallest AVF rates among all components, while in the case of the cache memories we can see that the L1 D Cache has the worst AVF rates. As far as the percentage increases are concerned, we observe that for every component there is an increase in the AVF as the number of faults we inject increases but most importantly we see that the increase between the 1 and 2 faults injected is bigger for all components compared to the case of 2 to 3 faults. A high of 12.40% AVF increase is observed for the ITLB case for 2-faults injected, compared to single fault injection. We also observe a low of a 3.77% AVF increase on the Register File for 3 fault injections compared to 2 faults injections.

The above calculation of the AVF of a component for each fault injection case, assumes that each benchmark's execution time is the same. Although, this leads to adequate results for many cases, we make the same calculations for a weighted AVF that takes into consideration the execution time (measured in clock cycles) of each of the 15 benchmarks. This weighted average of the AVFs for one injection case (of one component) was calculated by summing the AVFs of all benchmarks each multiplied by the execution time of the corresponding benchmark and dividing them by the sum of the execution time of all the benchmarks:

$$\text{Weighted AVF}_{\text{of (1,2,3) faults for X component}} = [(\text{AVF}_{\text{(1,2,3) faults/benchmark 1}} * \text{ExecutionTime}_{\text{benchmark1}}) + \ldots + (\text{AVF}_{\text{(1,2,3) faults/benchmark15}} * \text{ExecutionTime}_{\text{benchmark15}}] / (\text{ExecutionTime}_{\text{benchmark1}} + \ldots + \text{ExecutionTime}_{\text{benchmark1}})$$

Those independent of technology node weighted AVFs for 1,2,3 faults for each component appear in Table 5.2 and in Figure 5.26. Just like the case of the non-weighted AVF, one can see the percentage change (increase) in the AVF of a

component between the 1-fault case and the 2-fault case and between the 2-fault and 3-fault cases:

Table 5.2: Weighted AVF per component for 1, 2, 3 faults injected

Component	Number of Faults Injected	AVF	*Percentage Change*
L1 D Cache	1	20.32%	-
	2	29.70%	*+9.38%*
	3	36.28%	*+6.58%*
L1 I Cache	1	12.01%	-
	2	19.57%	*+7.56%*
	3	25.14%	*+5.56%*
L2 Cache	1	17.94%	-
	2	24.83%	*+6.89%*
	3	30.13%	*+5.30%*
Register File	1	10.95%	-
	2	18.65%	*+7.69%*
	3	23.01%	*+4.37%*
ITLB	1	50.31%	-
	2	62.91%	*+12.60%*
	3	66.67%	*+3.76%*
DTLB	1	50.66%	-
	2	61.77%	*+11.11%*
	3	67.22%	*+5.45%*

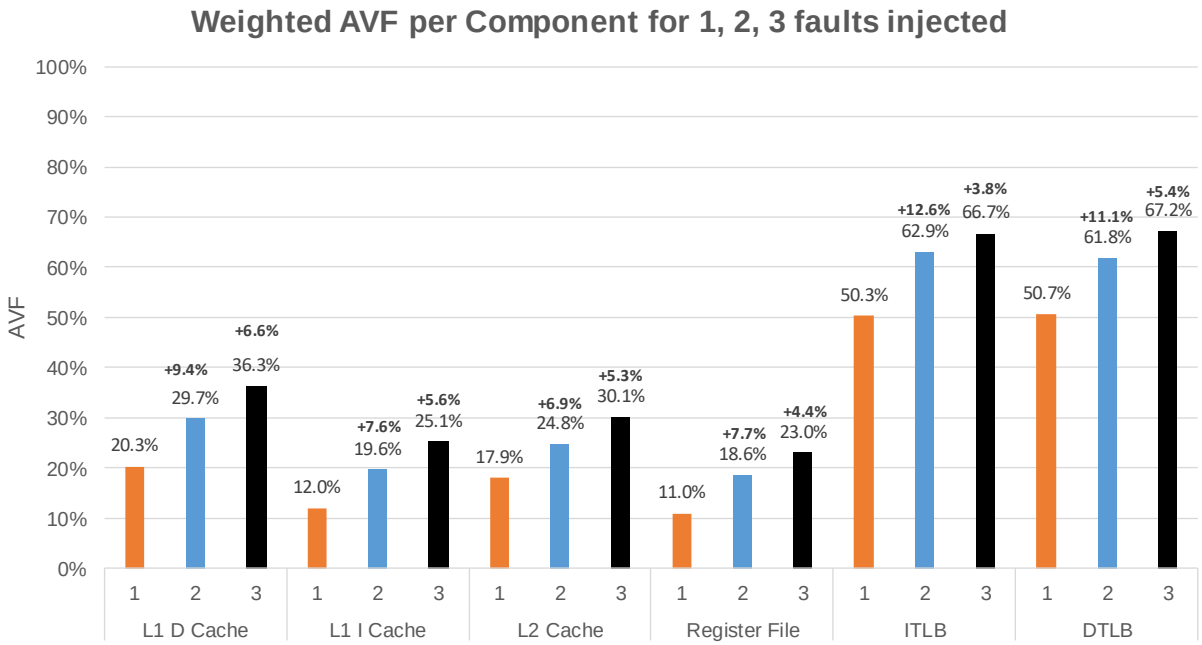


Figure 5.26: Weighted AVF per component for 1, 2, 3 faults injected

The same conclusions as in the case of the non-weighted AVF are also drawn in this case, with a few minor changes. First, we see that the worst AVF is also around 68%, but this time for the DTLB for 3 faults injected. This time, the probability of a fault in the TLB creating an error is always greater than 50%. A high of 12.60% AVF increase is observed on the ITLB-case for 2-faults injected, compared to single fault injection. A low of 4.4% AVF increase is observed on the Register File for 3 fault injections compared to 2 faults injections.

Those numbers for the components of the ARM Cortex-A9 core are independent of the manufacturing process. So, in the next step, we try to combine the AVFs with the technology node. For each technology node, there is a probability of a certain amount of fault appearing, as seen on Table 5.3 (see also Chapter 2). This is derived from Table II & III of [15] which presents the MCU ratio for each technology. Since we don't know how many of the MCUs are 2-bit fault faults, how many are 3-bit faults etc., we use the ratios of Table V of [15] (which presents the ratio of 2-bit faults to 3-bit faults etc.) because Table V does not include all MCUs but only those that appear on the same SRAM line. So, the 3+ -bit faults are reduced to 3-bit faults, since the rates for 4+ -bit faults are extremely low. For example, at 22 nm the MCU ratio is 44.7% as Table III in [15] shows. Table V on the other hand shows that at 22 nm, 2-bit faults are 3 out of the 3.9 total, so ~0.77 of the total. Then this is multiplied with the 22 nm MCU ratio (x44.7%) and we get the multi-bit rate for 2-bit faults at 22 nm which is 34.4%. The rest (44.7% - 34.4% = 10.3%) is the multi-bit rate for 3-bit faults at 22 nm, since we ignored the 3+ -bit faults. The 100% - MCU ratio (44.7%) = 55.3% is the 1-bit fault rate at 22 nm.

Table 5.3: Multi-bit rates per node

Technology Node	1 bit faults	2 bit faults	3 bit faults
250nm	100.00%	0.00%	0.00%
180nm	96.40%	3.60%	0.00%
130nm	93.40%	4.40%	2.20%
90nm	87.80%	9.60%	2.60%
65nm	81.60%	16.10%	2.30%
45nm	72.20%	23.00%	4.80%
32nm	65.30%	29.10%	5.60%
22nm	55.30%	34.40%	10.30%

As one can see, the bigger the technology node, the smaller the chances for the appearance of a multi-bit fault. But as the sizes shrink, the chances of multi-bit faults become non-negligible. By combining the probabilities for each fault case with the appropriate AVFs, for each component, we can find the AVF of each component for each technology node. We use the formula:

$$AVF_{\text{of component X, for Y nm}} = (AVF_{\text{of 1 fault for X component}} * \text{1 bit fault rate}_{\text{for Y nm}}) + (AVF_{\text{of 2 faults for X component}} * \text{2 bit fault rate}_{\text{for Y nm}}) + (AVF_{\text{of 3 faults for X component}} * \text{3 bit fault rate}_{\text{for Y nm}})$$

The results of this procedure for each component appear in Table 5.4 and Figure 5.27. The same results for the weighted AVF case appear in Table 5.5 and Figure 5.28:

Table 5.4: AVF per component for different technology nodes

Node	L1 D Cache	L1 I Cache	L2 Cache	Register File	ITLB	DTLB
250 nm	21.17%	11.47%	14.68%	10.63%	50.11%	48.50%
180 nm	21.55%	11.75%	14.91%	10.90%	50.56%	48.91%
130 nm	22.01%	12.09%	15.20%	11.21%	51.04%	49.38%
90 nm	22.63%	12.55%	15.57%	11.66%	51.75%	50.04%
65 nm	23.28%	13.01%	15.95%	12.12%	52.50%	50.73%
45 nm	24.44%	13.86%	16.66%	12.93%	53.79%	51.95%
32 nm	25.22%	14.44%	17.14%	13.49%	54.68%	52.79%
22 nm	26.59%	15.44%	17.99%	14.43%	56.15%	54.20%

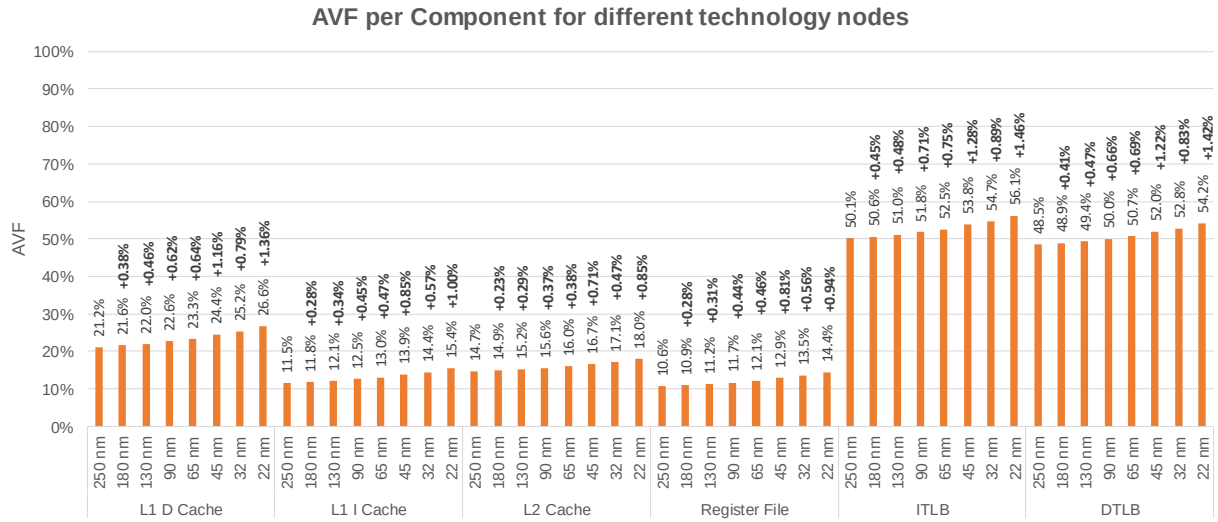


Figure 5.27: AVF per component for different technology nodes

We can observe that as the node decreases, multi-bit faults play a more significant role leading to larger AVFs. The trends for each component are the same as before. ITLB reaches a high of 56.15% AVF for the 22 nm node. This is smaller by 11,13 percentile units compared to the largest percent we observed on the AVFs without the technology node consideration, leading to the conclusion that because of the not so often appearance of two and three bit faults (compared to single bit faults) the real AVF is actually smaller than the one we calculated for no manufacturing process. We would actually need a big increase in 2- and 3-bit fault occurrences in nodes smaller than 22 nm, to reach those theoretical AVF numbers. This increase in AVF rates for all components as the node decreases, seems to also be steady as seen on the bold rates of Figure 5.27 which point out the percentage increase compared to the previous node. We can see that the rate of increase of the AVF is always around 0.5-1% as the node decreases. A high of 1.46% increase is observed on the ITLB case from 32 to 22 nm while a low of 0.23% increase is observed for the L2 Cache from 250 nm to 180 nm. We also observe that the rate increase is always larger than the "previous" one for all components (e.g. the rate of increase from 180 nm to 130 nm is larger than the rate of increase from 250 nm to 180 nm). This trend stops momentarily for all components on the 45 to 32 nm case where the rates are always smaller than the 65 to 45 nm case.

Table 5.5: Weighted AVF per component for different technology nodes

Node	L1 D Cache	L1 I Cache	L2 Cache	Register File	ITLB	DTLB
250 nm	20.32%	12.01%	17.94%	10.95%	50.31%	50.66%
180 nm	20.65%	12.28%	18.19%	11.23%	50.76%	51.06%
130 nm	21.08%	12.63%	18.51%	11.56%	51.23%	51.52%
90 nm	21.63%	13.08%	18.92%	12.01%	51.95%	52.16%
65 nm	22.19%	13.53%	19.33%	12.47%	52.72%	52.83%
45 nm	23.24%	14.38%	20.11%	13.30%	53.99%	54.01%
32 nm	23.94%	14.95%	20.63%	13.87%	54.89%	54.82%
22 nm	25.19%	15.96%	21.56%	14.84%	56.33%	56.19%

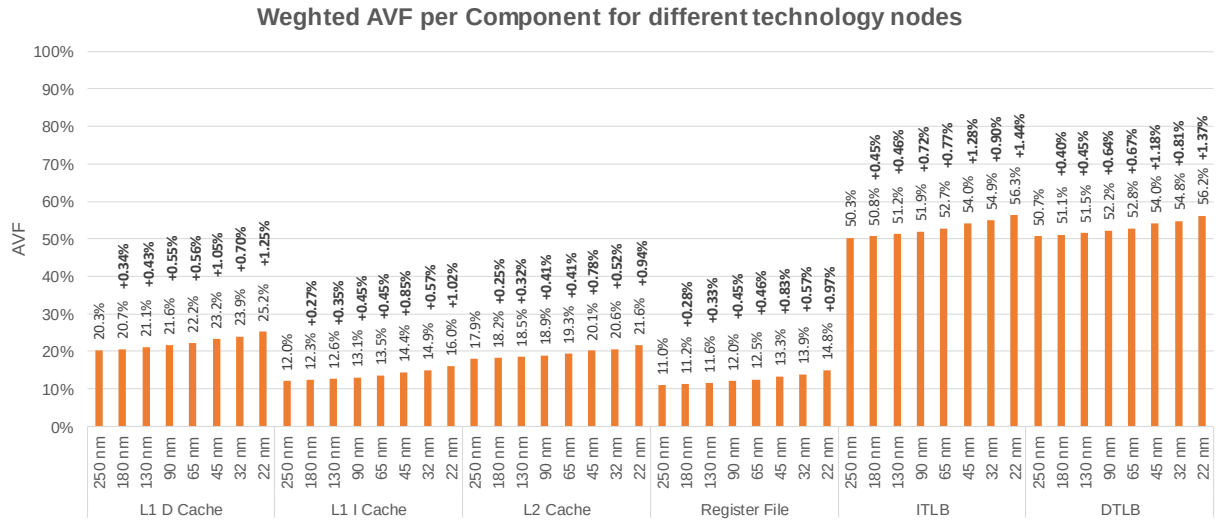


Figure 5.28: Weighted AVF per component for different technology nodes

The same trends are observed in the case of the weighted AVF. This time, ITLB reaches a high of 56.33% AVF at the 22 nm node. The DTLB AVF for the same node is 56.19%. This is smaller by 11,03 percentile units compared to the largest percent we observed on the weighted AVFs without the technology node consideration. This time a high of 1.44% increase is observed on the ITLB case from 32 to 22 nm while a low of 0.25% increase is observed for the L2 Cache from 250 nm to 180 nm.

5.3 Reliability (Failures in Time)

On this section, we continue the measurements by studying the Reliability of the core under test using the metric of Failures in Time (FIT)(see Chapter 1). For the calculation of the FIT of a component for a specific technology node, we need the rawFIT (or FIT_{BIT}) -which is relevant to the technology node, the size of the component in bits and the AVF of that component for that technology node. We extract the rawFIT per bit for each technology node, from Table III in [15] where the SER per MBit is presented for different technology nodes (we divide by 10^6 to get the SER rate for one bit). The results appear in Table 5.6. The size in bits of each of the six components we tested appears in Table 5.7.

Table 5.6: Raw FIT for nodes 250 to 22 nm

Node	Raw FIT per bit	Raw FIT per bit (scientific notation)
250 nm	0.00000047	$47x10^{-8}$
180 nm	0.00000085	$85x10^{-8}$
130 nm	0.00000106	$106x10^{-8}$
90 nm	0.000001	$100x10^{-8}$
65 nm	0.00000085	$85x10^{-8}$
45 nm	0.00000058	$58x10^{-8}$
32 nm	0.00000038	$38x10^{-8}$
22 nm	0.00000023	$23x10^{-8}$

Table 5.7: Component sizes in bits

Component	Size (in bits)
L1 D Cache	262144
L1 I Cache	262144
L2 Cache	4194304
Register File	2112
ITLB	1024
DTLB	1024

Then as we know the FIT for each component for one technology node is calculated by the formula:

$$FIT_{\text{of component X for Y nm}} = \text{rawFIT} * AVF_{\text{of component X, for Y nm}} * \text{Size of structure in bits}$$

This is computed for the case of AVF and weighted AVF. Then once we have the FITs of each of the six components for a technology node, we add them to compute the FIT of the core, FIT_{core}. Table 5.8 shows the FIT of each component for each technology node along with the FIT_{core} (Figure 5.29 and Figure 5.30 illustrate the results) for the non-weighted AVF case. From those, we can observe that the FIT for each component is increasing with the decreasing node up until the point of 130 nm. At 90 nm, a decrease starts, reaching the lowest FIT values at 22 nm for all components. This is mainly because at 130 nm, the rawFIT value starts decreasing, therefore decreasing the FIT of each component. The L2 Cache holds the larger values of FIT with a maximum of 676 failures in time at 130 nm, while the Register File, ITLB and DTLB hold negligible failures. So, we observe a contradiction between the AVFs and FIT. The TLBs which had large AVFs have non-existent FITs, mainly because of their extremely small sizes (just 1024 bits). On the other hand, the L2 Cache with a size of 4 MBits dominates. Same trends can also be seen for the accumulative FIT_{core} with a maximum of 772 failures at the 130 nm node.

Table 5.8: FIT per Component for different technology nodes

Node	*L1 D Cache*	*L1 I Cache*	*L2 Cache*	*Register File*	*ITLB*	*DTLB*	**FIT_{core}**
250 nm	26.1	14.1	289.4	0.1	0.2	0.2	**330.2**
180 nm	48	26.2	531.5	0.2	0.4	0.4	**606.8**
130 nm	61.2	33.7	675.8	0.3	0.6	0.5	**771.9**
90 nm	59.3	32.9	653.2	0.2	0.5	0.5	**746.7**
65 nm	51.9	28.9	568.8	0.2	0.5	0.4	**650.7**
45 nm	37.1	21.1	405.4	0.2	0.3	0.3	**464.4**
32 nm	25.1	14.4	273.1	0.1	0.2	0.2	**313.2**
22 nm	16	9.3	173,5	0.07	0.1	0.1	**199.2**

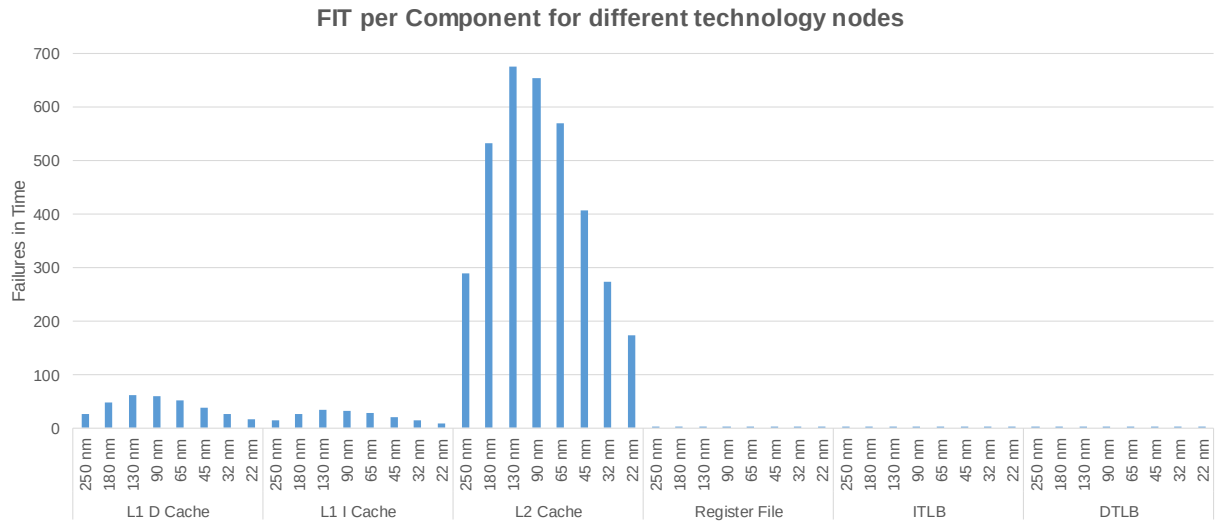


Figure 5.29: FIT per Component for different technology nodes

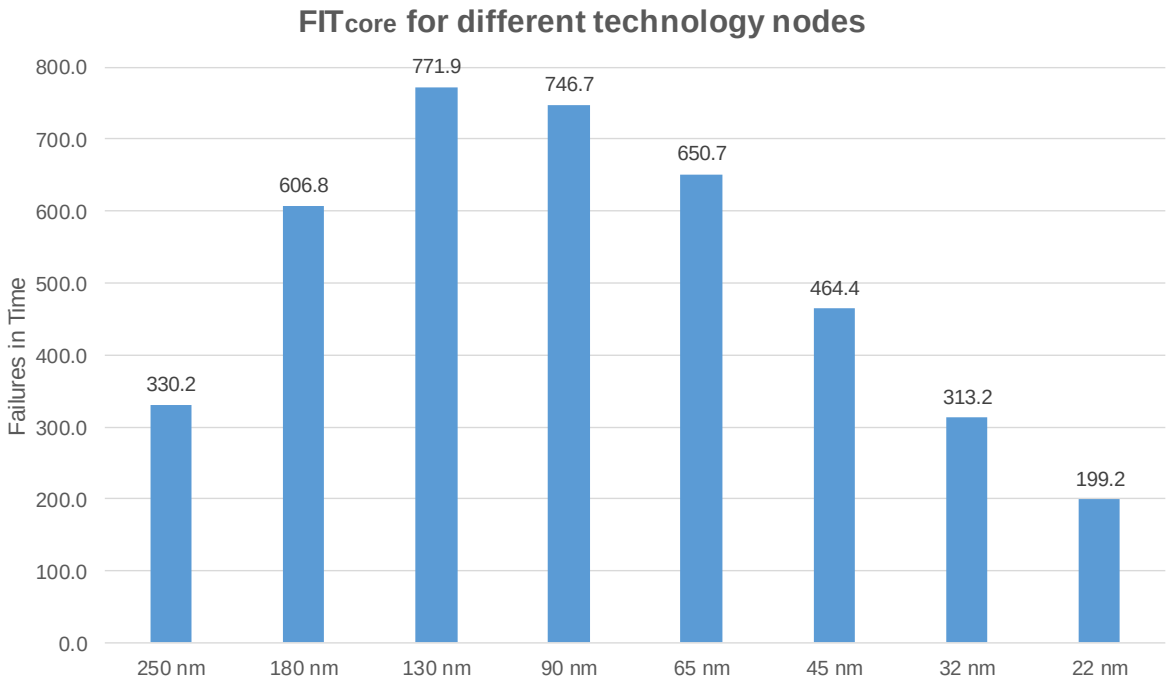


Figure 5.30: FITcore for different technology nodes

The same results for the weighted AVF case, appear in Table 5.9, Figure 5.31 and Figure 5.32. We observe the same trends compared to the non-weighted AVF case. The L2 Cache holds the larger values of FIT with a maximum of 823 failures in time at 130 nm. The FIT_{core} reaches a maximum value of 918 failures in time at 130 nm.

Table 5.9: FIT per Component for different technology nodes (weighted AVF)

Node	*L1 D Cache*	*L1 I Cache*	*L2 Cache*	*Register File*	*ITLB*	*DTLB*	**FIT_{core}**
250 nm	25	14.8	353.6	0.1	0.2	0.2	**394**
180 nm	46	27.4	648.4	0.2	0.4	0.4	**722.8**
130 nm	58.6	35.1	822.9	0.3	0.6	0.6	**918**
90 nm	56.7	34.3	793.4	0.3	0.5	0.5	**885.7**
65 nm	49.5	30.2	689.1	0.2	0.5	0.5	**769.8**
45 nm	35.3	21.9	489.2	0.2	0.3	0.3	**547.2**
32 nm	23.8	14.9	328.7	0.1	0.2	0.2	**368**
22 nm	15.2	9.6	208	0.1	0.1	0.1	**233.2**

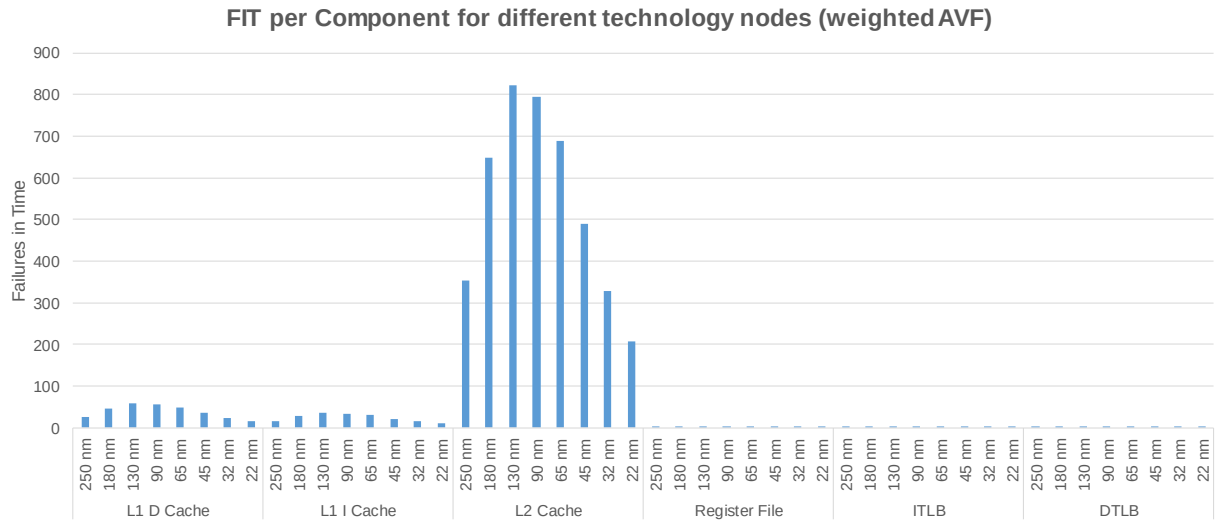


Figure 5.31: FIT per Component for different technology nodes (weighted AVF)

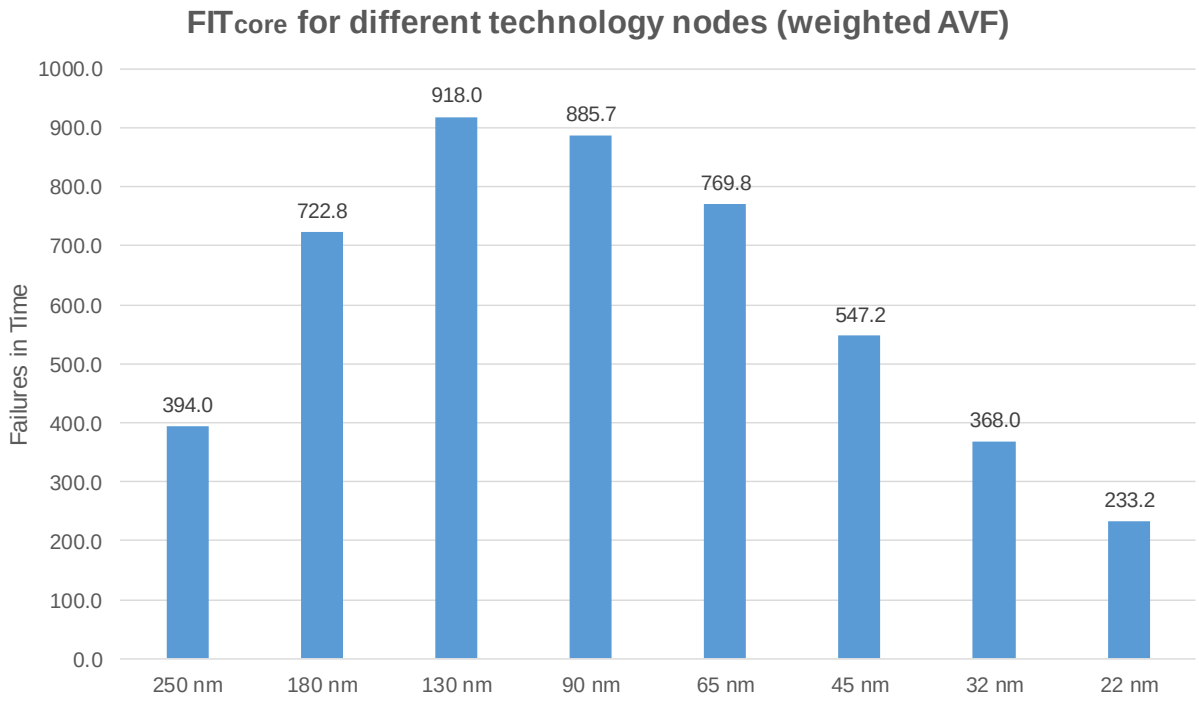


Figure 5.32: FITcore for different technology nodes (weighted AVF)

CONCLUSION

This thesis focused mainly on the effects of multi-bit fault injection in adjacent areas of several structures of a core under test (AMR Cortex-A9) and in that core in general. Its primary focus was a reliability and vulnerability assessment using the metrics of Architectural Vulnerability Factor (AVF) and Failures in Time (FIT). The results showed some very interesting points. From each component perspective, we observed that all the components are affected by the increasing number of adjacent faults injected, observing the highest vulnerability in structures like the TLBs and then the L1 D-Cache. For the case of the ITLB, we observed a record-low of 25% correct executions of a benchmark for 1000 runs when we injected three faults, while structures like the Register File seem to have the best fault masking behavior.

When the AVFs for each component were computed, we saw a similar trend with the biggest rates reaching a maximum of 67.28% in the case of the ITLB for the three-bit fault injection. Then, we combined those “theoretical” AVFs with different fabrication technological nodes from 250 to 22 nm, to see how this number could be affected under different manufacturing processes of the same core. The results showed increasing AVF rates for all components as the node decreases because on smaller nodes, multi-bit faults tend to have a bigger effect and appear more frequently. A high of 56.33% AVF for the 22 nm process was observed.

Lastly, the reliability of the core under test was examined utilizing the FIT metric. Because of its correlation with the raw FIT factor and the size of each structure we observed completely different results compared to the AVFs, showing almost non-existent values for components with high AVFs but very small sizes (ITLB, DTLB), with roughly 1 FIT for most technological nodes. On the other hand, the L2 cache because of its size (4 Mbit) dominated with a high of 822.9 FIT at the 130 nm node. Despite that we didn’t see increasing FITs as the node decreases. FITs reach a high at 130 nm for all components and then the decrease of the rawFIT factor leads to a decrease of the FIT for smaller nodes. The maximum FIT of the core was observed to be 918 at 130 nm.

This study provided an advanced capability for the injection of multi-bit faults in adjacent structure areas and the utilization of interleaving through the updated fault mask generator of the GeFIN framework. As future work, one could utilize those capabilities by either performing studies for more than 3 adjacent faults injections per run or perform tests on a core with an interleaving degree of N, to observe how this countermeasure against multi-bit faults affects the reliability and vulnerability of the core under test.

ABBREVIATIONS - ACRONYMS

ACE	Architecturally-Correct Execution
AVF	Architectural Vulnerability Factor
CAM	Content Addressable Memory
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Codes
DEC-TED	Double Error Correction / Triple Error Detection
DTLB	Data Translation Lookaside Buffer
DUE	Detected Uncorrected Error
DUT	Design Under Test
ECC	Error Correction Code
FinFET	Fin Field-Effect Transistor
FIT	Failures in Time
GeFIN	Gem5-based Fault Injector
IC	Integrated Circuit
IDE	Integrated Development Environment
ISA	Instruction Set Architecture
ITLB	Instruction Translation Lookaside Buffer
MB-AVF	Multiple Bit Architectural Vulnerability Factor
MBU	Multiple Bit Upsets
MCU	Multiple Cell Upset
MTTF	Mean Time to Failure
RAM	Random-Access Memory
ROB	Re-Order Buffer
RTL	Register Transfer Level
SB-AVF	Single Bit Architectural Vulnerability Factor
SBF	Single-Bit Fault
SDC	Silent Data Corruption
SEC-DED	Single Error Correction/Double Error Detection
SER	Soft Error Rate
SET	Single Transient Effect

SEU	Single Event Upset
SFI	Statistical Fault Injection
sMBF	Spatial Multi-Bit Fault
SPEC	Standard Performance Evaluation Corporation
SRAM	Static Random-Access Memory
tMBF	Temporal Multi-Bit Fault
VLSI	Very Large Scale Integration
XOR	Exclusive OR
μarch	Microarchitecture

ANNEX I

This Annex includes the code of the fault mask generator implementation in C++ and the corresponding fault mask parameters text file, described in Section 3.3.

---

fault_mask_parameter.txt

```
1.  ##################################################
2.  #----                                        ----#
3.  #---- Fault-mask Generator INPUT PARAMETERS ----#
4.  #----                                        ----#
5.  ##################################################
6.  #
7.  #.:|Version:
8.  arm_small_susan_c-cortex_a15_base
9.  #
10. #.:|Population:
11. 6
12. #
13. #.:|Core ID (Note: generate fault_mask for the specific coreID):
14. 0
15. #
16. #.:|Module ID (Note: multiple modules separate with ';'):
17. #.:| xxxx yyyy zzzz/b --> xxxx:memory(0)/core(1), yyyy:module ID, zzzz:sub-array ID
18. #.:| Physical_register_file:256 -- Branch_predictor (Bimodal:352, Meta:353, Two-
    Level:354, iBTB/data:355, iBTB/tag:356, iBTB/LRU:357, iBTB/valid:358
19. #.:| dBTB/data:359, dBTB/tag:360, dBTB/LRU:361, dBTB/valid:362, RAS:363)
20. #.:| LSQ (data:272, virtaddr:273, addrvalid:274, datavalid:275, bytemask:276)
21. 256;352
22. #
23. #.:|Rows:
24. 128
25. #
26. #.:|Columns:
27. 128
28. #.:|Bit vector size:
29. 32
30. #
31. #.:|Offset range:
32. 0
33. #
34. #.:|Parallel faults (must be less or equal to cluster size rows*columns in case of mult
    i-bit faults):
35. 8
36. #
37. #.:|Multi-Bit Fault Cluster Rows (leave 0 if no multi-
    bit fault injection is desired):
38. 3
39. #
40. #.:|Multi-Bit Fault Cluster Columns (leave 0 if no multi-
    bit fault injection is desired):
41. 3
42. #
43. #.:|Interleaving Scheme (leave 1 if no interleaving is required, else any power of 2 is
     allowed smaller than Number of Rows (or Number of Columns):
44. 4
45. #
46. #.:| Total simulation time (Note: Bounds the activation parameter. Only for transient a
    nd intermittent faults):
47. 1892025
48. #
49. #.:|Duration (NOTE: For permanent and transient faults equals to '1'):
```

```
#.:| Warning: "Duration" MUST be smaller than request pool size of FaultRequest
100000
#
#.......:|  NOTE: MIX fault masks are generated only when parallel faults parameter dif
fers from '1'  |:......#
#
#####################################################
#----                                           ----#
#                       END OF FILE                 #
#----                                           ----#
#####################################################
```

---

fault_mask_generator.cc

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <cstdlib>
#include <stdio.h>
#include <string.h>
#include <sys/io.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <cmath>

using namespace std;

//Global Variables
int debug = 1; //logging enabled

struct parameters {
    string version;
    int faults;
    int coreId;
    string moduleId;
    int rowId;
    int columnId;
    int bitvector;
    int offsetId;
    int conc_faults;
    int multi_bit_rowId;
    int multi_bit_columnId;
    int interleaving;
    int firstActivation;
    int duration;
};

parameters fault_mask_parameter; //update stucture with fault_mask_paramter.txt file co
ntents

vector<char> checksum_row, checksum_column, checksum_offset, checksum_module, checksum_
position, checksum_activation;
int checksum_count = 0; //aux variable to track random number distribution
vector<char> checksum_row_multi, checksum_column_multi;
int checksum_count_multi = 0; //aux variable to track random number distribution (multi
-bit faults)
vector<int> multiple_module;

int dummy_debug_flag = 0; //check if valid entries exist on apps.txt file

ofstream DEBUG_FILE; //create debug file object
```

```
47. fstream FAULT_MASK_FILE; //create fault mask file object
48. ifstream FAULT_MASK_PARAMETERS_FILE; //create fault mask parameters file object
49.
50. int dummy_i = 1; //dummy counter to parse struct elements
51. int dummy_j = 1; //dummy counter to parse multiple module vector elements
52.
53. int found = 0;
54.
55. string fault[10];
56.
57. int multi_bit_enabled = 0; //defines if multi-bit fault injection is active or not
58.
59. int interleaving_row, interleaving_column = 0;
60.
61. //Method declaration
62. int random_num(int upper_limit);
63. int random_num_multi(int lower_limit, int upper_limit);
64. int checksum(int row, int column, int offset, int position, int module, int activation)
    ;
65. int checksum_multi(int row, int column);
66. int interleave_check(int row, int column, int interleaving_scheme);
67. void write_fault_mask(int coreID, int moduleID, int type, int rowID, int columnID, int
    offsetID, int model, int fault_mask, int firstActivation, int duration, int counter, in
    t counter2, string dir);
68.
69.
70. int main() {
71.
72.     // Seed random number generator
73.     srand(time(NULL));
74.
75.     cout << "...:| Fault Mask Generator |:.....\n\n";
76.
77.     try { //open debug file
78.         DEBUG_FILE.open("generator.log");
79.     }
80.     catch (const exception& e)
81.     {
82.         cout << "Error while opening file";
83.         exit(EXIT_FAILURE);
84.     }
85.
86.     try { //open fault mask parameters file
87.         FAULT_MASK_PARAMETERS_FILE.open("fault_mask_parameter.txt");
88.     }
89.     catch (const exception& e)
90.     {
91.         cout << "Error while opening file";
92.         exit(EXIT_FAILURE);
93.     }
94.
95.     dummy_debug_flag = 0;
96.
97.     if (debug == 1) {
98.         DEBUG_FILE << "Read Parameters...\n";
99.     }
100.
101.    string line;
102.
103.    while (getline(FAULT_MASK_PARAMETERS_FILE, line)) {
104.
105.        string sub_str = line.substr(0, 1);
106.
107.        if (sub_str.compare("#") != 0) {
108.
109.            dummy_debug_flag = 1;
110.
111.            switch (dummy_i) {
```

```
112.
113.             case 1:
114.                 fault_mask_parameter.version = line;
115.                 if (debug == 1) {
116.                     DEBUG_FILE << "Database Version..." << fault_mask_parameter.version
     << endl;
117.                 }
118.                 break;
119.             case 2:
120.                 fault_mask_parameter.faults = stoi(line);
121.                 if (!(fault_mask_parameter.faults)) {
122.                     cout << "Error: Argument population is zero. Generation stops abnor
    mally.\n";
123.                     exit(EXIT_FAILURE);
124.                 }
125.                 else
126.                 {
127.                     if (debug == 1) {
128.                         DEBUG_FILE << "Population..." << fault_mask_parameter.faults <<
     endl;
129.                     }
130.                 }
131.                 break;
132.             case 3:
133.                 fault_mask_parameter.coreId = stoi(line);
134.                 if (debug == 1) {
135.                     DEBUG_FILE << "Core ID's..." << fault_mask_parameter.coreId << endl
    ;
136.                 }
137.                 break;
138.             case 4:
139.                 fault_mask_parameter.moduleId = line;
140.                 if (fault_mask_parameter.moduleId == "") {
141.                     cout << "Error: ModuleID parameter is null. Experiment stops abnorm
    ally.\n";
142.                     exit(EXIT_FAILURE);
143.                 }
144.                 else {
145.                     do { //modules separated with ';', split output
146.                         found = fault_mask_parameter.moduleId.find(";");
147.                         if (found != -1) {
148.                             string temp;
149.                             for (int i = 0; i < found; i++){
150.                                 temp[i] = fault_mask_parameter.moduleId[i];
151.                             }
152.                             multiple_module.insert(multiple_module.end(), stoi(temp));

153.                             fault_mask_parameter.moduleId = fault_mask_parameter.module
    Id.erase(0, found+1);
154.                         }
155.                         else if (found == -1) {
156.                             multiple_module.insert(multiple_module.end(), stoi(fault_ma
    sk_parameter.moduleId));
157.                             fault_mask_parameter.moduleId = "";
158.                         }
159.                     } while (found != -1); //no moduleId left
160.                     if (multiple_module.size() > 1) { //moduleId has more than 2 module
    s inside
161.                         fault_mask_parameter.moduleId = to_string(multiple_module[0]);

162.                         for (dummy_j = 1; dummy_j < (int)multiple_module.size(); dummy_
    j++) {
163.                             fault_mask_parameter.moduleId = fault_mask_parameter.module
    Id + "_" + to_string(multiple_module[dummy_j]);
164.                         }
165.                     }
166.                     else {
```

```
167.                         fault_mask_parameter.moduleId = to_string(multiple_module[0]);

168.                     }
169.                     if (debug == 1) {
170.                         DEBUG_FILE << "Module ID's..." << fault_mask_parameter.moduleId
    << " (" << multiple_module.size() << " defined) " << endl;
171.                     }
172.                 }
173.                 break;
174.             case 5:
175.                 fault_mask_parameter.rowId = stoi(line);
176.                 if (!(fault_mask_parameter.rowId)) {
177.                     cout << "Error: Argument rows is zero. Generation stops abnormally.
    \n";
178.                     exit(EXIT_FAILURE);
179.                 }
180.                 else {
181.                     if (debug == 1) {
182.                         DEBUG_FILE << "Rows..." << fault_mask_parameter.rowId << endl;

183.                     }
184.                 }
185.                 break;
186.             case 6:
187.                 fault_mask_parameter.columnId = stoi(line);
188.                 if (debug == 1) {
189.                     DEBUG_FILE << "Columns..." << fault_mask_parameter.columnId << endl
    ;
190.                 }
191.                 if (!(fault_mask_parameter.columnId)) {
192.                     cout << "Warning: Argument columns is zero.\n";
193.                 }
194.                 break;
195.             case 7:
196.                 fault_mask_parameter.bitvector = stoi(line);
197.                 if (debug == 1) {
198.                     DEBUG_FILE << "BitVector size..." << fault_mask_parameter.bitvector
    << endl;
199.                 }
200.                 if (!(fault_mask_parameter.bitvector)) {
201.                     cout << "Error: Argument bit vector is zero. Generation stops abnor
    mally.\n";
202.                     exit(EXIT_FAILURE);
203.                 }
204.                 break;
205.             case 8:
206.                 fault_mask_parameter.offsetId = stoi(line);
207.                 if (debug == 1) {
208.                     DEBUG_FILE << "Offset range..." << fault_mask_parameter.offsetId <<
    endl;
209.                 }
210.                 if (!(fault_mask_parameter.offsetId)) {
211.                     cout << "Warning: Argument offset is zero.\n";
212.                 }
213.                 break;
214.             case 9:
215.                 fault_mask_parameter.conc_faults = stoi(line);
216.                 if (debug == 1) {
217.                     DEBUG_FILE << "Parallel Faults..." << fault_mask_parameter.conc_fau
    lts << endl;
218.                 }
219.                 if (!(fault_mask_parameter.conc_faults)) {
220.                     cout << "Error: Argument parallel faults is zero. Generation stops
    abnormally.\n";
221.                     exit(EXIT_FAILURE);
222.                 }
223.                 if ((multiple_module.size() > 1) && (fault_mask_parameter.conc_faults =
    = 1)) {
```

```
224.                     cout << "Error: Multiple modules defined for single fault models.\n
    ";
225.                     exit(EXIT_FAILURE);
226.                 }
227.                 break;
228.             case 10:
229.                 fault_mask_parameter.multi_bit_rowId = stoi(line);
230.                 if (debug == 1){
231.                     DEBUG_FILE << "Multi-
    Bit Cluster Rows..." << fault_mask_parameter.multi_bit_rowId << endl;
232.                 }
233.                 break;
234.             case 11:
235.                 fault_mask_parameter.multi_bit_columnId = stoi(line);
236.                 if (debug == 1){
237.                     DEBUG_FILE << "Multi-
    Bit Cluster Columns..." << fault_mask_parameter.multi_bit_columnId << endl;
238.                 }
239.                 if (fault_mask_parameter.multi_bit_rowId != 0 && fault_mask_parameter.m
    ulti_bit_columnId == 0){
240.                     cout << "Error: Multi Bit Cluster Rows greater than 1, but Multi Bi
    t CLuster Columns is zero.\n";
241.                     exit(EXIT_FAILURE);
242.                 }
243.                 if (fault_mask_parameter.multi_bit_columnId != 0 && fault_mask_paramete
    r.multi_bit_rowId == 0){
244.                     cout << "Error: Multi Bit Cluster Columns greater than 1, but Multi
     Bit CLuster Rows is zero.\n";
245.                     exit(EXIT_FAILURE);
246.                 }
247.                 if (fault_mask_parameter.multi_bit_rowId > fault_mask_parameter.rowId |
    | fault_mask_parameter.multi_bit_columnId > fault_mask_parameter.columnId){
248.                     cout << "Error: Multi Bit Cluster Rows greater than Rows, or Multi
    Bit Cluster Columns greater than Columns.\n";
249.                     exit(EXIT_FAILURE);
250.                 }
251.                 if (fault_mask_parameter.multi_bit_rowId > 0 && fault_mask_parameter.mu
    lti_bit_columnId > 0){
252.                     multi_bit_enabled = 1; //multi bit fault injection is enabled
253.                     if (debug == 1){
254.                         DEBUG_FILE << "Multi-Bit Fault Injection enabled..." << endl;
255.                     }
256.                 }
257.                 if ((fault_mask_parameter.conc_faults > (fault_mask_parameter.multi_bit
    _rowId * fault_mask_parameter.multi_bit_columnId) && multi_bit_enabled == 1)){
258.                     cout << "Error: Parallel Faults number bigger than maximum parallel
     faults that can be inserted in the cluster (rows * columns = " << fault_mask_parameter
    .multi_bit_rowId * fault_mask_parameter.multi_bit_columnId << " faults max).\n";
259.                     exit(EXIT_FAILURE);
260.                 }
261.                 break;
262.             case 12:
263.                 fault_mask_parameter.interleaving = stoi(line);
264.                 if (fault_mask_parameter.interleaving == 0) {
265.                     cout << "Error: Argument interleaving is zero. Generation stops abn
    ormally.\n";
266.                     exit(EXIT_FAILURE);
267.                 }
268.                 if (((fault_mask_parameter.interleaving % 2) != 0) && (fault_mask_param
    eter.interleaving != 1)) {
269.                     cout << "Error: Argument interleaving must be 1 or a power of 2.\n"
    ;
270.                     exit(EXIT_FAILURE);
271.                 }
272.                 if ((fault_mask_parameter.interleaving > fault_mask_parameter.rowId) ||
     (fault_mask_parameter.interleaving > fault_mask_parameter.columnId))
273.                     cout << "Error: Argument interleavng must be smaller than Number of
     Rows (Number of Columns).\n";
```

```
274.                 if (debug == 1) {
275.                     DEBUG_FILE << "Interleaving scheme..." << fault_mask_parameter.inte
    rleaving << endl;
276.                 }
277.                 break;
278.             case 13:
279.                 fault_mask_parameter.firstActivation = stoi(line);
280.                 if (debug == 1) {
281.                     DEBUG_FILE << "Activation..." << fault_mask_parameter.firstActivati
    on << endl;
282.                 }
283.                 break;
284.             case 14:
285.                 fault_mask_parameter.duration = stoi(line);
286.                 if (debug == 1) {
287.                     DEBUG_FILE << "Duration..." << fault_mask_parameter.duration << end
    l;
288.                 }
289.                 if (!(fault_mask_parameter.duration)) {
290.                     cout << "Warning: Argument duration faults is zero. None fault will
     be injected.\n";
291.                 }
292.                 break;
293.             } //end_switch
294.             dummy_i++;
295.         } //end_if
296.     } //end_while
297.
298.     if (!(dummy_debug_flag)) {
299.         cout << "Error: None valid entries exist on fault_mask_parameter.txt file\n";
300.         exit(EXIT_FAILURE);
301.     }
302.     else {
303.         if (debug == 1) {
304.             DEBUG_FILE << "Successfully read parameters...\n";
305.         }
306.     }
307.
308.     string path;
309.     path = fault_mask_parameter.version;
310.
311.     if (debug == 1) {
312.         DEBUG_FILE << "Main Path is..." << path << endl;
313.     }
314.
315.     if (mkdir(path.c_str(), 0777) != 0) //make main directory (if it exists, execution
    stops)
316.     {
317.         cout << "Main Directory was not created (error or already exists, please delete
     and re-run)" << endl;
318.         exit(EXIT_FAILURE);
319.     }
320.     else
321.     {
322.         if (debug == 1) {
323.             DEBUG_FILE << "Warning:make directory..." + path << endl;
324.         }
325.     }
326.
327.     string path2;
328.     path2 = path + "/" + fault_mask_parameter.moduleId;
329.
330.     //make moduleId directory inside the main directory
331.     if (mkdir(path2.c_str(), 0777) != 0)
332.     {
333.         cout << "ModuleId directory was not created" << endl;
334.         exit(EXIT_FAILURE);
335.     }
```

```
336.    else
337.    {
338.        if (debug == 1) {
339.            DEBUG_FILE << "Warning:make moduleId directory..." << endl;
340.        }
341.    }
342.
343.    //make different folders inside the moduleId directory
344.    if (mkdir((path2 + "/permanent").c_str(), 0777) != 0)
345.    {
346.        cout << "Error:make /permanent directory..." << endl;
347.        exit(EXIT_FAILURE);
348.    }
349.    if (mkdir((path2 + "/intermittent").c_str(), 0777) != 0)
350.    {
351.        cout << "Error:make /intermittent directory..." << endl;
352.        exit(EXIT_FAILURE);
353.    }
354.    if (mkdir((path2 + "/transient").c_str(), 0777) != 0)
355.    {
356.        cout << "Error:make /transient directory..." << endl;
357.        exit(EXIT_FAILURE);
358.    }
359.    if (mkdir((path2 + "/mix").c_str(), 0777) != 0)
360.    {
361.        cout << "Error:make /mix directory..." << endl;
362.        exit(EXIT_FAILURE);
363.    }
364.
365.    //-------------MAIN() part of code---------------------------------------
366.    //Generate fault masks
367.    int ext_counter = 1;
368.    int int_counter = 1; //dummy_loop_counter
369.    int selected_row = -1;
370.    int selected_column = -1;
371.    int selected_offset = -1;
372.    int selected_position = -1;
373.    int selected_type = -1;
374.    int selected_module = -1;
375.    int selected_activation = -1;
376.    int selected_row_cluster = -1;
377.    int selected_column_cluster = -1; //store random numbers generated from subs
378.
379.    //-------Statistical safe sample calculation-----
380.//    float confidence = 3.0902; //99.8%
381.//    float errorMargin = 0.01; //1%
382.//    int initialpop = fault_mask_parameter.firstActivation * fault_mask_parameter.rowI
    d * fault_mask_parameter.columnId * fault_mask_parameter.bitvector;
383.//    float prob = 0.5; //probability X event to happen
384.//    int sample = 0; //stat safe. sample
385.//
386.//    sample = ceil(initialpop / (1 + (errorMargin * errorMargin) * ((initialpop -
     1)/((confidence * confidence) * prob * (1 - prob)))));
387.//    cout << "\nThe statistical safe sample of fault (confidence " << confidence << "
    and error margin " << errorMargin << ") \nfor the selected module " << fault_mask_param
    eter.moduleId << " equals to " << sample << ".\n";
388.//    cout << "Generator is configured to produce " << fault_mask_parameter.faults << "
     faults\n";
389.//    cout << "Do you want to continue [Y/n]: ";
390.//    char YN;
391.//    cin >> YN;
392.//    if (YN == 'Y' || YN == 'y'){
393.//        cout << "Start...\n";
394.//    }
395.//    else{
396.//        cout << "Generator terminated. Update fault mask population to be statistical
    ly safe\n";
397.//        exit(EXIT_FAILURE);
```

```
398.//    }//end of estimation
399.
400.    if (multi_bit_enabled == 1){//if multi-bit injection is enabled
401.        while (ext_counter <= fault_mask_parameter.faults){
402.                if (debug == 1) {
403.                    DEBUG_FILE << "Generating fault mask..." << int_counter << endl;
404.                }
405.                int ret_val = 1; //checksum() return value
406.
407.                do {
408.                    if (debug == 1) {
409.                        DEBUG_FILE << "Fault mask" << ext_counter << endl;
410.                    }
411.                    do {
412.                        selected_row = random_num(fault_mask_parameter.rowId); //starti
    ng (0,0) rowId of multi-bit cluster
413.                        selected_column = random_num(fault_mask_parameter.columnId); //
    starting (0,0) columnId of multi-bit cluster
414.                    } while ((selected_row + fault_mask_parameter.multi_bit_rowId > fau
    lt_mask_parameter.rowId) ||  (selected_column + fault_mask_parameter.multi_bit_columnId
     > fault_mask_parameter.columnId));
415.                    if (fault_mask_parameter.offsetId == 0){//avoid division by 0
416.                        selected_offset = 0;
417.                    }
418.                    else{
419.                        selected_offset = random_num(fault_mask_parameter.offsetId);
420.                    }
421.                    selected_position = random_num(fault_mask_parameter.bitvector);
422.                    selected_type = random_num(2);
423.                    selected_module = multiple_module[random_num(multiple_module.size()
    )];
424.                    selected_activation = random_num(fault_mask_parameter.firstActivati
    on);
425.                    ret_val = checksum(selected_row, selected_column, selected_offset,
    selected_position, selected_module, selected_activation);
426.                } while (ret_val == 0);
427.
428.                if (debug == 1) {
429.                    DEBUG_FILE << "Generate permanent fault mask...\n";
430.                }
431.
432.                //generate permanent multi bit fault mask
433.                for (int_counter = 1; int_counter <= fault_mask_parameter.conc_faults;
    int_counter++) {
434.                    do {
435.                        selected_row_cluster = random_num_multi(selected_row, selected_
    row + fault_mask_parameter.multi_bit_rowId - 1);
436.                        selected_column_cluster = random_num_multi(selected_column, sel
    ected_column + fault_mask_parameter.multi_bit_columnId - 1);
437.                        ret_val = checksum_multi(selected_row_cluster, selected_column_
    cluster);
438.                    } while (ret_val == 0);
439.                    //in case of interleaving change selected row and column to correns
    pond to their positions in the interleaved memory before the write of fault mask
440.                    if (fault_mask_parameter.interleaving != 1){ //interleaving
441.                        interleave_check(selected_row_cluster, selected_column_cluster,
     fault_mask_parameter.interleaving);
442.                        write_fault_mask(fault_mask_parameter.coreId, selected_module,
    selected_type, interleaving_row, selected_column_cluster, selected_offset, 0, 1, 1, 1,
    ext_counter, int_counter, path2 + "/permanent");
443.                    }
444.                    else { //no interleaving
445.                        write_fault_mask(fault_mask_parameter.coreId, selected_module,
    selected_type, selected_row_cluster, selected_column_cluster, selected_offset, 0, 1, 1,
     1, ext_counter, int_counter, path2 + "/permanent");
446.                    }
447.                }
448.
```

```
449.                //clear checksum_multi row and column vectors
450.                checksum_row_multi.clear();
451.                checksum_column_multi.clear();
452.                checksum_count_multi = 0;
453.
454.
455.                //duplicate fault mask location
456.                if (debug == 1) {
457.                    DEBUG_FILE << "Generate intermittent fault mask...\n";
458.                }
459.
460.                //generate intermittent multi bit fault mask
461.                for (int_counter = 1; int_counter <= fault_mask_parameter.conc_faults;
    int_counter++) {
462.                    do {
463.                        selected_row_cluster = random_num_multi(selected_row, selected_
    row + fault_mask_parameter.multi_bit_rowId - 1);
464.                        selected_column_cluster = random_num_multi(selected_column, sel
    ected_column + fault_mask_parameter.multi_bit_columnId - 1);
465.                        ret_val = checksum_multi(selected_row_cluster, selected_column_
    cluster);
466.                    } while (ret_val == 0);
467.                    //in case of interleaving change selected row and column to correns
    pond to their positions in the interleaved memory before the write of fault mask
468.                    if (fault_mask_parameter.interleaving != 1){ //interleaving
469.                        interleave_check(selected_row_cluster, selected_column_cluster,
     fault_mask_parameter.interleaving);
470.                        write_fault_mask(fault_mask_parameter.coreId, selected_module,
    selected_type, interleaving_row, selected_column_cluster, selected_offset, 1, 1, select
    ed_activation, fault_mask_parameter.duration, ext_counter, int_counter, path2 + "/inter
    mittent"); //intermittent
471.                    }
472.                    else { //no interleaving
473.                        write_fault_mask(fault_mask_parameter.coreId, selected_module,
    selected_type, selected_row_cluster, selected_column_cluster, selected_offset, 1, 1, se
    lected_activation, fault_mask_parameter.duration, ext_counter, int_counter, path2 + "/i
    ntermittent"); //intermittent
474.                    }
475.                }
476.
477.                //clear checksum_multi row and column vectors
478.                checksum_row_multi.clear();
479.                checksum_column_multi.clear();
480.                checksum_count_multi = 0;
481.
482.                if (debug == 1) {
483.                    DEBUG_FILE << "Generate transient fault mask...\n";
484.                }
485.
486.                //generate transient multi bit fault mask
487.                for (int_counter = 1; int_counter <= fault_mask_parameter.conc_faults;
    int_counter++) {
488.                    do {
489.                        selected_row_cluster = random_num_multi(selected_row, selected_
    row + fault_mask_parameter.multi_bit_rowId - 1);
490.                        selected_column_cluster = random_num_multi(selected_column, sel
    ected_column + fault_mask_parameter.multi_bit_columnId - 1);
491.                        ret_val = checksum_multi(selected_row_cluster, selected_column_
    cluster);
492.                    } while (ret_val == 0);
493.                    //in case of interleaving change selected row and column to correns
    pond to their positions in the interleaved memory before the write of fault mask
494.                    if (fault_mask_parameter.interleaving != 1){ //interleaving
495.                        interleave_check(selected_row_cluster, selected_column_cluster,
     fault_mask_parameter.interleaving);
496.                        write_fault_mask(fault_mask_parameter.coreId, selected_module,
    2, interleaving_row, selected_column_cluster, selected_offset, 2, 1, selected_activatio
    n, 1, ext_counter, int_counter, path2 + "/transient"); //transient
```

```
497.                     }
498.                     else { //no interleaving
499.                         write_fault_mask(fault_mask_parameter.coreId, selected_module,
    2, selected_row_cluster, selected_column_cluster, selected_offset, 2, 1, selected_activ
    ation, 1, ext_counter, int_counter, path2 + "/transient"); //transient
500.                     }
501.                 }
502.
503.                 //clear checksum_multi row and column vectors
504.                 checksum_row_multi.clear();
505.                 checksum_column_multi.clear();
506.                 checksum_count_multi = 0;
507.
508.
509.                 //generate a fault model mixture (multi_bit)
510.                 for (int_counter = 1; int_counter <= fault_mask_parameter.conc_faults;
    int_counter++) {
511.                     selected_row_cluster = random_num_multi(selected_row, selected_row
    + fault_mask_parameter.multi_bit_rowId - 1);
512.                     selected_column_cluster = random_num_multi(selected_column, selecte
    d_column + fault_mask_parameter.multi_bit_columnId - 1);
513.                     ret_val = checksum_multi(selected_row_cluster, selected_column_clus
    ter);
514.                     int mix_fault = random_num(3);
515.                     if (debug == 1) {
516.                         DEBUG_FILE << "Generate multi-
    bit cluster mixed fault model..." << mix_fault << endl;
517.                     }
518.                     //in case of interleaving change selected row and column to correns
    pond to their positions in the interleaved memory before the write of fault mask
519.                     if (fault_mask_parameter.interleaving != 1){ // interleaving
520.                         interleave_check(selected_row_cluster, selected_column_cluster,
     fault_mask_parameter.interleaving);
521.                     }
522.                     if (mix_fault == 0) {
523.                         if (fault_mask_parameter.interleaving != 1){ //interleaving is
    enabled
524.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, interleaving_row, selected_column_cluster, selected_offset, 0, 1, 1,
     1, ext_counter, int_counter, path2 + "/mix"); //permanent
525.                         }
526.                         else { //no interleaving
527.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, selected_row_cluster, selected_column_cluster, selected_offset, 0, 1
    , 1, 1, ext_counter, int_counter, path2 + "/mix"); //permanent
528.                         }
529.                     }
530.                     else if (mix_fault == 1) {
531.                         if (fault_mask_parameter.interleaving != 1){ //interleaving is
    enabled
532.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, interleaving_row, selected_column_cluster, selected_offset, 1, 1, se
    lected_activation, fault_mask_parameter.duration, ext_counter, int_counter, path2 + "/m
    ix"); //intermittent
533.                         }
534.                         else { //no interleaving
535.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, selected_row_cluster, selected_column_cluster, selected_offset, 1, 1
    , selected_activation, fault_mask_parameter.duration, ext_counter, int_counter, path2 +
     "/mix"); //intermittent
536.                         }
537.                     }
538.                     else {
539.                         if (fault_mask_parameter.interleaving != 1){ //interleaving is
    enabled
540.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, 2, interleaving_row, selected_column_cluster, selected_offset, 2, 1, selected_activ
    ation, 1, ext_counter, int_counter, path2 + "/mix"); //transient
```

```
                        }
                        else { //no interleaving
                            write_fault_mask(fault_mask_parameter.coreId, selected_module, 2, selected_row_cluster, selected_column_cluster, selected_offset, 2, 1, selected_activation, 1, ext_counter, int_counter, path2 + "/mix"); //transient
                        }
                    }
                }

                //clear checksum_multi row and column vectors
                checksum_row_multi.clear();
                checksum_column_multi.clear();
                checksum_count_multi = 0;

                ext_counter++;

                if (debug == 1) {
                    DEBUG_FILE << "Fault mask generated...\n";
                }
            }
        }
    else{//if multi_bit injection is disabled
        while (ext_counter <= fault_mask_parameter.faults) {
            for (int_counter = 1; int_counter <= fault_mask_parameter.conc_faults; int_counter++) {
                if (debug == 1) {
                    DEBUG_FILE << "Generating fault mask..." << int_counter << endl;
                }
                int ret_val = 1; //checksum() return value

                do {
                    if (debug == 1) {
                        DEBUG_FILE << "Fault mask" << ext_counter << endl;
                    }
                    selected_row = random_num(fault_mask_parameter.rowId);
                    selected_column = random_num(fault_mask_parameter.columnId);
                    if (fault_mask_parameter.offsetId == 0){//avoid division by 0
                        selected_offset = 0;
                    }
                    else{
                        selected_offset = random_num(fault_mask_parameter.offsetId);
                    }
                    selected_position = random_num(fault_mask_parameter.bitvector);
                    selected_type = random_num(2);
                    selected_module = multiple_module[random_num(multiple_module.size())];
                    selected_activation = random_num(fault_mask_parameter.firstActivation);
                    ret_val = checksum(selected_row, selected_column, selected_offset, selected_position, selected_module, selected_activation);
                } while (ret_val == 0);

                if (debug == 1) {
                    DEBUG_FILE << "Generate permanent fault mask...\n";
                }

                //in case of interleaving change selected row and column to correnspond to their positions in the interleaved memory before the write of fault mask
                if (fault_mask_parameter.interleaving != 1){
                    interleave_check(selected_row, selected_column, fault_mask_parameter.interleaving);
                }

                //generate single fault model
                if (fault_mask_parameter.interleaving != 1){ //if interleaving is enabled
```

```
598.                     write_fault_mask(fault_mask_parameter.coreId, selected_module, sele
    cted_type, interleaving_row, selected_column, selected_offset, 0, 1, 1, 1, ext_counter,
     int_counter, path2 + "/permanent");
599.                 }
600.                 else { //no interleaving
601.                     write_fault_mask(fault_mask_parameter.coreId, selected_module, sele
    cted_type, selected_row, selected_column, selected_offset, 0, 1, 1, 1, ext_counter, int
    _counter, path2 + "/permanent");
602.                 }
603.
604.                 //duplicate fault mask location
605.                 if (debug == 1) {
606.                     DEBUG_FILE << "Generate intermittent fault mask...\n";
607.                 }
608.                 if (fault_mask_parameter.interleaving != 1){ //if interleaving is enabl
    ed
609.                     write_fault_mask(fault_mask_parameter.coreId, selected_module, sele
    cted_type, interleaving_row, selected_column, selected_offset, 1, 1, selected_activatio
    n, fault_mask_parameter.duration, ext_counter, int_counter, path2 + "/intermittent"); /
    /intermittent
610.                 }
611.                 else { //no interleaving
612.                     write_fault_mask(fault_mask_parameter.coreId, selected_module, sele
    cted_type, selected_row, selected_column, selected_offset, 1, 1, selected_activation, f
    ault_mask_parameter.duration, ext_counter, int_counter, path2 + "/intermittent"); //int
    ermittent
613.                 }
614.
615.                 if (debug == 1) {
616.                     DEBUG_FILE << "Generate transient fault mask...\n";
617.                 }
618.                 if (fault_mask_parameter.interleaving != 1){ //if interleaving is enabl
    ed
619.                     write_fault_mask(fault_mask_parameter.coreId, selected_module, 2, i
    nterleaving_row, selected_column, selected_offset, 2, 1, selected_activation, 1, ext_co
    unter, int_counter, path2 + "/transient"); //transient
620.                 }
621.                 else { //no interleaving
622.                     write_fault_mask(fault_mask_parameter.coreId, selected_module, 2, s
    elected_row, selected_column, selected_offset, 2, 1, selected_activation, 1, ext_counte
    r, int_counter, path2 + "/transient"); //transient
623.                 }
624.
625.                 //generate a fault model mixture in concurrent fault injection
626.                 if (fault_mask_parameter.conc_faults != 1) {
627.                     int mix_fault = random_num(3);
628.                     if (debug == 1) {
629.                         DEBUG_FILE << "Generate multi-
    bit mixed fault model..." << mix_fault << endl;
630.                     }
631.                     if (mix_fault == 0) {
632.                         if (fault_mask_parameter.interleaving != 1){ //if interleaving
    is enabled
633.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, interleaving_row, selected_column, selected_offset, 0, 1, 1, 1, ext_
    counter, int_counter, path2 + "/mix"); //permanent
634.                         }
635.                         else { //no interleaving
636.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, selected_row, selected_column, selected_offset, 0, 1, 1, 1, ext_coun
    ter, int_counter, path2 + "/mix"); //permanent
637.                         }
638.                     }
639.                     else if (mix_fault == 1) {
640.                         if (fault_mask_parameter.interleaving != 1){ //if interleaving
    is enabled
641.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, interleaving_row, selected_column, selected_offset, 1, 1, selected_a
```

```
    ctivation, fault_mask_parameter.duration, ext_counter, int_counter, path2 + "/mix"); //
    intermittent
642.                         }
643.                         else { //no interleaving
644.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, selected_type, selected_row, selected_column, selected_offset, 1, 1, selected_activ
    ation, fault_mask_parameter.duration, ext_counter, int_counter, path2 + "/mix"); //inte
    rmittent
645.                         }
646.                     }
647.                     else {
648.                         if (fault_mask_parameter.interleaving != 1){ //if interleaving
    is enabled
649.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, 2, interleaving_row, selected_column, selected_offset, 2, 1, selected_activation, 1
    , ext_counter, int_counter, path2 + "/mix"); //transient
650.                         }
651.                         else { //no interleaving
652.                             write_fault_mask(fault_mask_parameter.coreId, selected_modu
    le, 2, selected_row, selected_column, selected_offset, 2, 1, selected_activation, 1, ex
    t_counter, int_counter, path2 + "/mix"); //transient
653.                         }
654.                     }
655.                 }
656.             } //end_of_for
657.
658.             ext_counter++;
659.
660.             if (debug == 1) {
661.                 DEBUG_FILE << "Fault mask generated...\n";
662.             }
663.         }
664.     }
665.
666.     if (debug == 1) {
667.         DEBUG_FILE << "Successfully generate fault masks!\n";
668.     }
669.
670.     cout << "...Finished.\n";
671.
672.     //close open files
673.     FAULT_MASK_PARAMETERS_FILE.close();
674.     DEBUG_FILE.close();
675.     return 0;
676. }
677.
678. //generate random number
679. int random_num(int upper_limit) {
680.
681.     int random = 0;
682.     random = rand() % (upper_limit);
683.
684.     if (debug == 1)
685.     {
686.         DEBUG_FILE << "Generating random value..." << random << endl;
687.     }
688.
689.     return random;
690.
691. }
692.
693. //generate random number with lower and upper limits
694. int random_num_multi(int lower_limit, int upper_limit) {
695.
696.     int random = 0;
697.     random = rand() % (upper_limit-lower_limit+1) + lower_limit;
698.
699.     if (debug == 1)
```

```
700.    {
701.        DEBUG_FILE << "Generating random value..." << random << endl;
702.    }
703.
704.    return random;
705.
706.}
707.
708.int checksum(int row, int column, int offset, int position, int module, int activation)
     {
709.
710.    if (debug == 1) {
711.        DEBUG_FILE << "Generate checksum for row..." << row << " column..." << column <
    < " Offset..." << offset << " position..." << position << " activation..." << activatio
    n << endl;
712.
713.    }
714.
715.    for (dummy_i = 0; dummy_i < checksum_count; dummy_i++) {
716.        if ((checksum_row[dummy_i] == row) && (checksum_column[dummy_i] == column) && (
    checksum_offset[dummy_i] == offset) && (checksum_position[dummy_i] == position) && (che
    cksum_module[dummy_i] == module) && (checksum_activation[dummy_i] = activation)) {
717.            if (debug == 1) {
718.                DEBUG_FILE << "Checksum FAIL! \n";
719.            }
720.            return 0; //entry, way, position and bit already selected
721.        }
722.    }
723.
724.    checksum_row.insert(checksum_row.begin() + dummy_i, row);
725.    checksum_column.insert(checksum_column.begin() + dummy_i, column);
726.    checksum_offset.insert(checksum_offset.begin() + dummy_i, offset);
727.    checksum_position.insert(checksum_position.begin() + dummy_i, position);
728.    checksum_module.insert(checksum_module.begin() + dummy_i, module);
729.    checksum_activation.insert(checksum_activation.begin() + dummy_i, activation);
730.    checksum_count++;
731.
732.    if (debug == 1) {
733.        DEBUG_FILE << "Checksum PASS! \n";
734.    }
735.
736.    return 1;
737.
738.}
739.
740.int checksum_multi(int row, int column) {
741.
742.    if (debug == 1) {
743.        DEBUG_FILE << "Generate checksum for cluster row..." << row << " and column..."
     << column << endl;
744.    }
745.
746.    for (dummy_i = 0; dummy_i < checksum_count_multi; dummy_i++) {
747.        if ((checksum_row_multi[dummy_i] == row) && (checksum_column_multi[dummy_i] ==
    column)) {
748.            if (debug == 1) {
749.                DEBUG_FILE << "Checksum FAIL! \n";
750.            }
751.            return 0; //entry, way, position and bit already selected
752.        }
753.    }
754.
755.    checksum_row_multi.insert(checksum_row_multi.begin() + dummy_i, row);
756.    checksum_column_multi.insert(checksum_column_multi.begin() + dummy_i, column);
757.    checksum_count_multi++;
758.
759.    if (debug == 1) {
760.        DEBUG_FILE << "Checksum PASS! \n";
```

```
761.     }
762.
763.     return 1;
764.
765. }
766.
767. //interleave method, check if selected element position in memory corresponds to interl
    eaved memory
768. int interleave_check(int row, int column, int interleaving_scheme){
769.
770.     if (debug == 1){
771.         DEBUG_FILE << "Checking if randomly selected row..." << row << " and column..."
     << column << " corresponds to an interleaved memory with N = " << interleaving_scheme
    << ".\n";
772.     }
773.
774.     bool condition;
775.
776.     for (int i = 1; i <= (interleaving_scheme -
     1); i++){//moving through columns of the sub-
    array sized interleaving_schemeXinterleaving_scheme
777.         if (column % interleaving_scheme == i){
778.             condition = true;
779.             int j = 0;
780.             while (condition && (j < i)){
781.                 condition = condition && ((row % interleaving_scheme) != j);
782.                 j++;
783.             }
784.             if (condition)
785.             {
786.                 row = row - (column % interleaving_scheme);}
787.             else
788.             {
789.                 row = row + (interleaving_scheme - (column % interleaving_scheme));
790.             }
791.         }
792.     }
793.
794.     if (debug == 1){
795.         DEBUG_FILE << "Method returns row..." << row << " and column..." << column << "
     of the interleaved memory\n";
796.     }
797.
798.     interleaving_row = row;
799.
800.     return 1;
801. }
802.
803.
804. void write_fault_mask(int coreID, int moduleID, int type, int rowID, int columnID, int
    offsetID, int model, int fault_mask, int firstActivation, int duration, int counter, in
    t counter2, string dir) {
805.
806.
807.     if (debug == 1) {
808.         DEBUG_FILE << "Write fault mask to file...fault_mask_" << counter << ".txt" <<
    " to dir: " << dir << endl;
809.     }
810.
811.     try { //open fault mask file in append mode to read it's data (needed for multi wri
    te operations in one file e.g. parallel faults)
812.         FAULT_MASK_FILE.open(dir + "/fault_mask_" + to_string(counter) + ".txt", fstrea
    m::in | fstream::out | fstream::app);
813.     }
814.     catch (const exception& e)
815.     {
816.         cout << "Error while opening file";
817.         exit(EXIT_FAILURE);
```

```
818.     }
819.
820.     //get existing file data (in case of revisiting file for parallel faults)
821.     if (counter2 > 1){
822.         for (int i = 0; i <= 9; i++){
823.             getline(FAULT_MASK_FILE, fault[i]);
824.         }
825.     }
826.
827.     //close file in append mode
828.     FAULT_MASK_FILE.close();
829.
830.     try { //open fault mask file in truncate mode to write read data + new data
831.         FAULT_MASK_FILE.open(dir + "/fault_mask_" + to_string(counter) + ".txt", fstrea
    m::in | fstream::out | fstream::trunc);
832.     }
833.     catch (const exception& e)
834.     {
835.         cout << "Error while opening file";
836.         exit(EXIT_FAILURE);
837.     }
838.
839.     //add new values to fault tmp array
840.     fault[0] = to_string(coreID) + "_" + fault[0];
841.     fault[1] = to_string(moduleID) + "_" + fault[1];
842.     fault[2] = to_string(type) + "_" + fault[2];
843.     fault[3] = to_string(rowID) + "_" + fault[3];
844.     fault[4] = to_string(columnID) + "_" + fault[4];
845.     fault[5] = to_string(offsetID) + "_" + fault[5];
846.     fault[6] = to_string(model) + "_" + fault[6];
847.     fault[7] = to_string(fault_mask) + "_" + fault[7];
848.     fault[8] = to_string(firstActivation) + "_" + fault[8];
849.     fault[9] = to_string(duration) + "_" + fault[9];
850.
851.     //remove last character if no more faults are to be inserted
852.     if (counter2 == fault_mask_parameter.conc_faults) {
853.         for (int i = 0; i <= 9; i++){
854.             fault[i] = fault[i].substr(0, strlen(fault[i].c_str()) - 1);
855.         }
856.     }
857.
858.     //write fault tmp array values to fault mask output file
859.     for (int i = 0; i <= 9; i++){
860.         FAULT_MASK_FILE << fault[i] << endl;
861.     }
862.
863.     //empty fault tmp array for next iteration/next file to write
864.     for (int i = 0; i <= 9; i++){
865.         fault[i] = "";
866.     }
867.
868.     //close file
869.     FAULT_MASK_FILE.close();
870.
871. }
```

ANEX II

This Annex includes the code of the fault mask generator implementation in Python described in Section 3.4.

---

fault_mask_generator.py

```python
import optparse
import os
import random
import shutil
import time

from scipy import stats, math

DEFAULT_VALUES = {
    'rows': 32,
    'columns': 32,
    'duration': 10000,
    'version': 'v1',
    'model': 2,  # Transient,
    'core_id': 0,
    'faults': 1,
    'cluster_rows': 1,
    'cluster_columns': 1,
    'interleaving': 1,
    'probability': 50,
    'error_margin': 1,
    'confidence_level': 99.8
}


# perm kai interm AND i OR (0 i 1)
def main():
    parser = optparse.OptionParser()
    # Arguments
    parser.add_option("--core_id", type="int", help="CPU core id", default=DEFAULT_VALUES['core_id'])
    parser.add_option("--rows", type="int", help="SRAM rows", default=DEFAULT_VALUES['rows'])
    parser.add_option("--columns", type="int", help="SRAM columns", default=DEFAULT_VALUES['columns'])
    parser.add_option("--faults", type="int", help="Number of faults", default=DEFAULT_VALUES['faults'])
    parser.add_option("--model", type="int", help="Fault model", default=DEFAULT_VALUES['model'])
    parser.add_option("--cl_rows", type="int", help="Cluster rows", default=DEFAULT_VALUES['cluster_rows'])
    parser.add_option("--cl_cols", type="int", help="Cluster columns", default=DEFAULT_VALUES['cluster_columns'])
    parser.add_option("--duration", type="int", help="Duration of int faults", default=DEFAULT_VALUES['duration'])
    parser.add_option("--module_id", type="int", help="Component id")
    parser.add_option("--simtime", type="int", help="Workload clock cycles")
    parser.add_option("--seed", type="float", help="Rand seed", default=time.time())
    parser.add_option("--version", type="string", action="store", help="Mask version",
                      default=DEFAULT_VALUES['version'])
    parser.add_option("--path", type="string", action="store", help="Store path", default='latest')
```

```
    parser.add_option("--interleaving", type="int", help="Interleaving size", default=DEFAULT_VALUES['interleaving'])
    parser.add_option("--probability", type="float", help="Probability value", default=DEFAULT_VALUES['probability'])
    parser.add_option("--error_margin", type="float", help="Error margin in % format",
                      default=DEFAULT_VALUES['error_margin'])
    parser.add_option("--confidence_level", type="float", help="Confidence level in % format",
                      default=DEFAULT_VALUES['confidence_level'])
    parser.add_option("--datasets", type="int", help="Number of datasets")

    # Get args
    (options, args) = parser.parse_args()

    # Map to variables
    rows = options.rows
    path = options.path
    model = options.model
    core_id = options.core_id
    columns = options.columns
    version = options.version
    datasets = options.datasets
    duration = options.duration
    module_id = options.module_id
    time_seed = options.seed
    parallel_faults = options.faults
    simulation_time = options.simtime
    cluster_rows_size = options.cl_rows
    cluster_columns_size = options.cl_cols
    interleaving = options.interleaving
    probability = options.probability
    error_margin = options.error_margin
    confidence_level = options.confidence_level

    # TODO Terminate if any mandatory argument is missing

    if model == 2:
        duration = ""

    random.seed(time_seed)

    if datasets is None:
        population = calculate_population(rows, columns, cluster_rows_size, cluster_columns_size,
                                          parallel_faults, simulation_time)
        t_score = calculate_t_score(confidence_level, population)
        datasets = calculate_sample(population, error_margin, t_score, probability)

    output_format = '{core_number} {component_number} {ticks} {error_row_dim} {error_column_dim} {model} ' \
                    '{type} {duration}\n'
    file_output_format = './{path}/{version}/{module_id}/{mask_name}.txt'
    ensure_cluster_size(rows, columns, cluster_rows_size, cluster_columns_size)

    cluster_positions = []
    for x in range(0, rows - cluster_rows_size + 1):
        for y in range(0, columns - cluster_columns_size + 1):
            cluster_positions.append((x, y))

    clean_path(path)
    mappings = calculate_interleaving_mappings(rows, columns, interleaving)

    for i in range(0, datasets):
        output = "V2\n"
        random_cluster = random.choice(cluster_positions)
```

```
104.         random_faults = random.sample(range(0, cluster_rows_size * cluster_columns_size
    ), parallel_faults)
105.         for fault in random_faults:
106.             error_row_dim = (fault // cluster_columns_size) + random_cluster[0]
107.             error_column_dim = random_cluster[1] + (fault % cluster_columns_size) % (co
    lumns - random_cluster[1])
108.
109.             error_row_dim, error_column_dim = mappings[(error_row_dim, error_column_dim
    )]
110.
111.             if model != 2:
112.                 op_type = random.randint(0, 1)
113.             else:
114.                 op_type = ""
115.
116.             output += output_format.format(core_number=core_id, component_number=module
    _id,
117.                                            ticks=random.randint(0, simulation_time),
118.                                            duration=duration, error_row_dim=error_row_d
    im,
119.                                            error_column_dim=error_column_dim, model=mod
    el, type=op_type)
120.
121.         save_mask(file_output_format.format(
122.             path=path, version=version, module_id=module_id, mask_name="fault_mask_" +
    str(i + 1)), output
123.         )
124.
125.
126. def ensure_cluster_size(rows, columns, cluster_rows_size, cluster_columns_size):
127.     if rows < cluster_rows_size or columns < cluster_columns_size:
128.         raise ValueError("Cluster's size cannot fit to grid size")
129.
130.
131. def create_file_path_if_not_exists(file_path):
132.     if not os.path.exists(os.path.dirname(file_path)):
133.         os.makedirs(os.path.dirname(file_path))
134.
135.
136. def clean_path(file_path):
137.     if os._exists(file_path):
138.         shutil.rmtree(file_path)
139.
140.
141. def save_mask(file_path, mask):
142.     create_file_path_if_not_exists(file_path)
143.     with open(file_path, "w") as f:
144.         f.write(mask)
145.
146.
147. def calculate_interleaving_mappings(rows, columns, interleaving):
148.     sub_arrays = []
149.
150.     for x in range(0, rows, interleaving):
151.         for y in range(0, columns, interleaving):
152.             sub_arrays.append((x, y))
153.
154.     grid = []
155.     for arr_start in sub_arrays:
156.         grid_columns = [[((column, row), [column, row]) for column in range(arr_start[0
    ], arr_start[0] + interleaving)]
157.                         for row in range(arr_start[1], arr_start[1] + interleaving)]
158.
159.         shift = 0
160.         for column in grid_columns:
161.             for real, mapped in column:
162.                 mapped[0] = (mapped[0] - shift) % interleaving + arr_start[0]
163.             shift += 1
```

```
164.
165.        grid.append(grid_columns)
166.
167.    mappings = {}
168.    for interleaving_size_columns in grid:
169.        for interleaving_column in interleaving_size_columns:
170.            for mapping in interleaving_column:
171.                mappings[mapping[0]] = mapping[1]
172.
173.    return mappings
174.
175.
176.def calculate_population(rows, columns, cluster_rows, cluster_columns, faults, simulati
    on_time):
177.    cluster_size = cluster_rows * cluster_columns
178.    cluster_population = (rows - cluster_rows) * (columns - cluster_columns)
179.    fault_combinations = math.factorial(cluster_size) // (
180.            math.factorial(faults) * math.factorial(cluster_size - faults))
181.    return simulation_time * cluster_population * fault_combinations
182.
183.
184.def calculate_t_score(confidence_level, population):
185.    return stats.t.isf((100 - confidence_level) / 100 / 2, population)
186.
187.
188.def calculate_sample(population, error_margin, t_score, p):
189.    return int(
190.        population / (1 + (error_margin / 100 ** 2) * ((population -
     1) / ((t_score ** 2) * p / 100 * (1 - p / 100)))))
191.
192.
193.if __name__ == "__main__":
194.    main()
```

REFERENCES

[1] Mark Wilkening, Vilas Sridharan, Si Li, Fritz Previlon, Sudhanva Gurumurthi and David R. Kaeli, “Calculating Architectural Vulnerability Factors for Spatial Multi-bit Transient Faults”, 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2014), 2014.

[2] J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman and J. M. Ross, "Accelerated testing for cosmic soft-error rate", *IBM Journal of Research and Development*, vol. 40, no. 1, Jan. 1996, p. 51-72.

[3] “Verification Architecture for Pre-Silicon Validation”, 2002; **www10.edacafe.com/book/parse_book.php?article=transeda%2Fch-10.html**. [Προσπελάστηκε 16/3/19]

[4] R. Leveugle, A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence", 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, 2009, p. 502-506.

[5] S. Mitra, S. A. Seshia and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances", Design Automation Conference, Anaheim, CA, 2010, p. 12-17.

[6] Fernanda Lima Kastensmidt, Ricardo Reis, *Fault-Tolerance Techniques for SRAM-Based FPGAs*, Springer, 2006, p. 1-27.

[7] M. Kaliorakis, A. Chatzidimitriou and D. Gizopoulos, “Tutorial: Microarchitecture Level Reliability Assessment Throughput and Accuracy”, The 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2017), Boston, MA, USA, October 2017.

[8] Shubu Mukherjee, *Architecture Design for Soft Errors 1st Edition*, Morgan Kaufmann, 2008, p. 9-11.

[9] Mei-Chen Hsueh, T. K. Tsai and R. K. Iyer, "Fault injection techniques and tools”, *Computer*, vol. 30, no. 4, April 1997, p. 75-82.

[10] «Microarchitecture Level - Principles of Computer Architecture”, *Slides for Advanced Computer Architecture,* Amity Business School; **www.docsity.com/en/microarchitecture-level-principles-of-computer-architecture-lecture-slides/313537/**, [Προσπελάστηκε 16/3/19]

[11] A.Chatzidimitriou, M.Kaliorakis, D.Gizopoulos, M.Iacaruso, M.Pipponzi, R.Mariani and S.Di Carlo, “RT Level vs. Microarchitecture Level Reliability Assessment: Case Study on ARM Cortex-A9 CPU”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017), Denver, CO, USA, June 2017.

[12] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, D. Gizopoulos, “Differential Fault Injection on Microarchitectural Simulators”, IEEE International Symposium on Workload Characterization (IISWC 2015), Atlanta, GA, USA, October 2015.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4, Austin, TX, USA, 2001, p. 3-14.

[14] A. Chatzidimitriou, D. Gizopoulos, “Anatomy of Microarchitecture-Level Reliability Assessment: Throughput and Accuracy”, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2016), Uppsala, Sweden, April, 2016.

[15] E. Ibe, H. Taniguchi, Y. Yahagi, K.-I. Shimbo, , and T. Toba, “Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule”, *IEEE Transactions on Electron Devices*, p. 1527–1538, Jul 2010.

[16] J. Maiz, S. Hareland, K. Zhang and P. Armstrong, "Characterization of multi-bit soft error events in advanced SRAMs", IEEE International Electron Devices Meeting 2003, Washington, DC, USA, 2003, p. 21.4.1-21.4.4.

[17] A. Sánchez-Macián, L. A. Aranda, P. Reviriego, V. Kiani and J. A. Maestro, "Enhancing Instruction TLB Resilience to Soft Errors", *IEEE Transactions on Computers*, vol. 68, no. 2, Feb. 2019, p. 214-224.

[18] A. Chatzidimitriou, M. Kaliorakis, S. Tselonis and D. Gizopoulos, "Performance-aware reliability assessment of heterogeneous chips", 2017 IEEE 35th VLSI Test Symposium (VTS), Las Vegas, NV, 2017, p. 1-6.