



**NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS**

**SCHOOL OF SCIENCE  
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

**THESIS**

**Architectural Vulnerability Factor (AVF) Assessment  
of x86 CPUs using Architectural Correct Execution (ACE)  
analysis in the Gem5 Simulator**

**Sofia Dionisios Alevizopoulos**

**Advisor: Dimitris Gizopoulos, Professor**

**ATHENS**

**May 2016**



**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ  
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

**Υπολογισμός του Architectural Vulnerability Factor (AVF)  
μικροεπεξεργαστών x86 με χρήση της Architectural Correct  
Execution (ACE) ανάλυσης στον προσομοιωτή Gem5**

**Σοφία Διονύσιος Αλεβιζοπούλου**

**Επιβλέπων: Δημήτρης Γκιζόπουλος, Καθηγητής**

**ΑΘΗΝΑ**

**Μάιος 2016**

# **THESIS**

Architectural Vulnerability Factor (AVF) Assessment  
of x86 CPUs using Architectural Correct Execution (ACE) analysis in the Gem5  
Simulator

**Sofia D. Alevizopoulos**

**A.M.:** 1115201000033

**Advisor: Dimitris Gizopoulos, Professor**

## **ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ**

Υπολογισμός του Architectural Vulnerability Factor (AVF) μικροεπεξεργαστών x86 με χρήση της Architectural Correct Execution (ACE) ανάλυσης στον προσομοιωτή Gem5

**Σοφία Δ. Αλεβιζοπούλου**

**A.M.: 1115201000033**

**Επιβλέπων: Δημήτρης Γκιζόπουλος, Καθηγητής**

## **ABSTRACT**

Despite the improvement of integrated circuits and microprocessor technologies, they become more vulnerable to external factors like cosmic radiation and alpha particles. These are the main reason causes of hardware faults. The cost for protecting all these structures in order not to result in hardware faults with diagnostic and protection methods is big enough. AVF (Architectural Vulnerability Factors) is a method for computing the vulnerability of a structure. AVF estimates the probability of a hardware fault to result in a wrong outcome for a program executing. This method can be done in the early stage of design and as a consequence many faults can pass over.

There are several methods for estimating AVF. In this study AVF is estimated using ACE analysis (Architectural Correct Analysis). This method is really fast as characterizes the bits in the structure as ACE or un-ACE bits but it has one disadvantage, it overestimates its vulnerability. ACE bits are those bits that influence the vulnerability of a structure. The experimental vehicle of this analysis is the microarchitecture simulator Gem5 for ISA x86-64. In this study, we computed AVF for ten different benchmarks in two different microarchitectural modules, the integer physical integer register file and the L1 Data Cache of Gem5. For each benchmark statistics about its runtime and ACE interval time are reported.

**SUBJECT AREA:** computer architecture, hardware, reliability, fault tolerance.

**KEYWORDS:** vulnerability factors, register file, cache memories, microarchitectural simulator, transient faults, AVF estimation, ACE analysis, Gem5.

## ΠΕΡΙΛΗΨΗ

Όσο η τεχνολογία κατασκευής ολοκληρωμένων κυκλωμάτων και μικροεπεξεραστών βελτιώνεται με το πέρασμα του χρόνου τόσο πιο ευάλωτα γίνονται όλα αυτά τα κυκλώματα σε εξωτερικούς παράγοντες όπως η κοσμική ακτινοβολία και τα σωματίδια άλφα. Πρόκειται για τη βασική πηγή που προκαλεί λάθη στο υλικό των επεξεργαστών. Το κόστος για την προστασία όλων αυτών των κυκλωμάτων με διαγνωστικές μεθόδους λάθους και μεθόδους προστασίας είναι πολύ μεγάλο. Ο συντελεστής AVF (συντελεστής αρχιτεκτονικής ευπάθειας) είναι μια μέθοδος υπολογισμού της ευπάθειας ενός συστήματος. Η μέθοδος AVF υπολογίζει την πιθανότητα ένα λάθος υλικού να οδηγήσει σε λανθασμένο αποτέλεσμα κατά την εκτέλεση ενός προγράμματος. Αυτή η μέθοδος μπορεί να εφαρμοστεί σε πρώιμο στάδιο κατά τη σχεδίαση του υπολογιστικού συστήματος και έτσι πολλά ενδεχόμενα λάθη να παραλειφθούν.

Υπάρχουν πολλές μέθοδοι για τον υπολογισμό του AVF. Στην παρούσα εργασία θα ασχοληθούμε με την ανάλυση ACE (Architectural Correct Execution). Πρόκειται για μια πολύ γρήγορη μέθοδος στην οποία τα bit ενός συστήματος χαρακτηρίζονται ως ACE ή un-ACE. ACE ονομάζονται τα bits τα οποία συμβάλουν στο συντελεστή ευπάθειας ενός συστήματος. Βασικό μειονέκτημα της μεθόδου είναι ότι υπερτιμά το συντελεστή ευπάθειας του συστήματος. Ο μηχανισμός που θα χρησιμοποιηθεί για τους διάφορους υπολογισμούς είναι ο προσομοιωτής Gem5 με αρχιτεκτονική συνόλου εντολών X86-64. Ο συντελεστής AVF έχει υπολογιστεί για 10 διαφορετικά προγράμματα τόσο για το αρχείο φυσικών ακέραιων καταχωρητών του Gem5 όσο και για τη μνήμη δεδομένων πρώτου επιπέδου. Για κάθε ένα από αυτά έχουν υπολογιστεί οι χρόνοι εκτέλεσής τους καθώς και τα ευάλωτα διαστήματά τους.

**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ:** αρχιτεκτονική υπολογιστών, υλικό υπολογιστή, αξιοπιστία, ανεκτικότητα σφαλμάτων.

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ:** συντελεστής αρχιτεκτονικής ευπάθειας, αρχείο καταχωρητών, μνήμη δεδομένων, προσομοιωτής μικροαρχιτεκτονικής, σφάλματα υλικού, υπολογισμός AVF, ανάλυση ACE, Gem5.

*For the completion of the current thesis, I would like to thank my advisor Professor, Dimitris Gizopoulos and the department's PhD candidates: Manolis Kaliorakis, Athanasios Chatzidimitriou and Sotiris Tselonis for their cooperation, advice and their valuable contribution to the successful completion of this study.*

# TABLE OF CONTENTS

<b>1. INTRODUCTION .....</b>	<b>14</b>
1.1 Subject .....	14
<b>2. HARDWARE FAULTS AND SOFT ERRORS.....</b>	<b>15</b>
2.1 Soft errors.....	15
2.2 Intermittent faults .....	17
2.3 Permanent faults .....	18
2.4 Soft Error Background and Terminology .....	18
<b>3. VULNERABILITY FACTORS.....</b>	<b>21</b>
3.1 PVF.....	22
3.2 H-AVF.....	22
3.3 IVF.....	22
3.4 AVF .....	23
3.4.1 ACE analysis .....	25
3.4.2 AVF Equation.....	27
3.5 Diagnostic and Protection mechanisms for hardware fault .....	27
<b>4. GEM5 SIMULATOR OVERVIEW .....</b>	<b>29</b>
4.1 M5 .....	29
4.2 GEMS .....	30
4.3 Fundamental requirements of Gem5 .....	31
4.4 Memory System in Gem5.....	31
4.4.1 Memory System Models .....	32
4.4.2 Port system .....	33
4.4.3 Packets .....	34



4.4.4	Requests.....	34
4.4.5	Atomic, functional, Timing access.....	34
<b>4.5</b>	<b>CPU models.....</b>	<b>35</b>
4.5.1	SimpleCPU.....	36
4.5.2	O3CPU.....	38
4.5.3	InOrder.....	40
<b>4.6</b>	<b>Extensive – object oriented design.....</b>	<b>41</b>
<b>4.7</b>	<b>Python .....</b>	<b>41</b>
<b>4.8</b>	<b>Physical Integer Register File.....</b>	<b>42</b>
<b>4.9</b>	<b>Gem5 Configuration.....</b>	<b>42</b>
<b>4.10</b>	<b>Inside the Gem5 .....</b>	<b>43</b>
<b>5.</b>	<b>SIMULATED SYSTEM .....</b>	<b>45</b>
5.1	The Linux Kernel.....	45
5.2	Clock cycle.....	45
5.3	Computation of simulation’s clock cycle .....	46
<b>6.</b>	<b>CACHE MEMORY .....</b>	<b>47</b>
6.1	Instruction Cache .....	48
6.2	Data Cache .....	49
6.3	Cache Mapping and Associativity.....	49
6.4	Cache Size .....	51
<b>7.</b>	<b>IMPLEMENTATION OF AVF ANALYSIS ASSESSMENT IN PHYSICAL INTEGER REGISTER FILE AND L1 DATA CACHE .....</b>	<b>52</b>
7.1	Start and End Tick of the Simulation .....	52
7.2	Dynamic Instruction .....	52
7.3	AVF for Physical Integer Register File .....	54
7.3.1	Structure for holding information about a read/write at a register .....	55

7.3.2	Methods for setting and loading a value from a register index.....	57
7.3.3	Instruction committed for an integer register in Physical Integer Register File .....	58
7.3.4	The computation of AVF .....	58
<b>7.4</b>	<b>AVF for Data Cache .....</b>	<b>59</b>
7.4.1	Structure for holding information about a read/write at a word in a block .....	60
7.4.2	Methods for setting and loading a word in L1 data cache.....	64
7.4.3	Packets transfer data to dynamic instruction Object.....	71
7.4.4	Instruction related with a word in L1 data cache commits.....	72
7.4.5	The Computation of AVF.....	72
<b>8.</b>	<b>RESULTS .....</b>	<b>74</b>
8.1	Conclusions .....	86
<b>9.</b>	<b>APPENDIX .....</b>	<b>87</b>
9.1	Commands for running a benchmark: .....	87
9.2	Hardware and Software configuration .....	88
	<b>ABBREVIATIONS .....</b>	<b>89</b>
	<b>SOURCE FILES AND HEADER FILES .....</b>	<b>90</b>
	<b>BIBLIOGRAPHY – REFERENCES.....</b>	<b>91</b>

## FIGURES' INDEX

Figure 1: Soft errors that do not influence the outcome of the program.....	16
Figure 2: Soft errors that result in a wrong outcome of the program.....	17
Figure 3: Intermittent error not always result in wrong outcome of a program .....	18
Figure 4: Permanent faults result in wrong outcome of a program .....	18
Figure 5: Ruby Simulator overview .....	32
Figure 6: Gem5 high level overview .....	33
Figure 7: CPU Model AtomicSimpleCPU.....	36
Figure 8: CPU Model TimingSimpleCPU.....	37
Figure 9: O3CPU pipeline.....	39
Figure 10: InOrderCPU pipeline .....	40
Figure 11: Source code Tree organization for Gem5.....	43
Figure 12: L1 & L2 cache memories.....	48
Figure 13: Divisions of the address for cache use .....	50
Figure 14: Fully associative Cache Memory .....	51
Figure 15: Class BaseO3DyInst .....	53
Figure 16: Class packet in DyInstPtr .....	53
Figure 17: Class InfoForCacheAccess .....	54
Figure 18: Class element.....	55
Figure 19: Class regi .....	55
Figure 20: Declare and initialize "class regi".....	56
Figure 21: Vector Entry for Physical Integer Register File .....	56
Figure 22: Functions readIntReg() and setIntReg() .....	57
Figure 23: Pointer to Physical Integer Register File.....	58
Figure 24: Function about simulator termination.....	59
Figure 25: Class elementForCache .....	61
Figure 26: Class CachesImplementation .....	62

Figure 27: The architecture of sets, blocks and words in L1 data cache. ....	62
Figure 28: Class cache .....	63
Figure 29: Class CacheBlkpointer .....	63
Figure 30: Class CacheBlk .....	64
Figure 31: Constructor of class CacheBlk() .....	64
Figure 32: Function cmpAndSwap() .....	65
Figure 33: Function satisfyCpuSideRequest() .....	66
Figure 34: Function writebackBlk() .....	67
Figure 35: Function handleSnoop() .....	68
Figure 36: Function access() .....	69
Figure 37: Function handleFill() .....	70
Figure 38: Class Packet.....	71
Figure 39: Dynamic Instruction committed in L1 data cache .....	72
Figure 40: AVF for the physical integer register file .....	75
Figure 41: AVF in physical register file using ACE analysis in comparison with fault injection .....	76
Figure 42: AVF for the L1 data cache with 100% vulnerable written back blocks.....	78
Figure 43: AVF for L1 data cache with 50% vulnerable written back blocks.....	79
Figure 44: AVF for L1 data cache with 0% vulnerable written back blocks.....	79
Figure 45: AVF in L1 DCache using ACE analysis in comparison with fault injection....	80
Figure 46: FIT in physical integer register File.....	84
Figure 47: FIT in L1 DCache with 100% vulnerable written back blocks .....	85
Figure 48: FIT in L1 DCache with 50% vulnerable written back blocks .....	85
Figure 49: FIT in L1 DCache with 0% vulnerable written back blocks .....	86

## TABLES' INDEX

Table 1: Hardware Faults and Soft Errors .....	15
Table 2: CPU Models and Memory system .....	35
Table 3: Simulator Configuration .....	42
Table 4: Benchmarks and Instruction count.....	74
Table 5: Execution and ACE cycles for each benchmark .....	77
Table 6: Execution and ACE cycles for each benchmark - Data Cache size: 16 KB .....	81
Table 7: Execution and ACE cycles for each benchmark - Data Cache size: 32 KB .....	82
Table 8: Execution and ACE cycles for each benchmark - Data Cache size: 64 KB .....	83

# 1. INTRODUCTION

## 1.1 Subject

The current thesis focuses on the AVF equation (Average Vulnerability factor) via ACE analysis in Gem5 simulator. My model was developed based on Gem5 (a full-system cycle-accurate simulator), the bibliography about AVF estimation and other various object-oriented programming practices. The AVF estimated for two different structures of the simulator, the physical integer register file for integer registers and the first level data cache.

AVF according Mukherjee is the probability that a fault in a structure will result in an error in a program's output [19]. There are several reasons that a bit of the integrated system can be destroyed and as a consequence change its value. The most common reason is the cosmic radiation. The rate at which these changes at the bit values occur depends on the electric potential at which this device operates, the size of transistor, the manufacturing technology of the device as long as the environment of the operating system. There are several ways to detect these errors but each of them has a big cost and is not always efficient. As a result, the AVF indicates how vulnerable a structure is in order to take it in mind during the design phase of the computing system. For instance, if it is known early enough during the design stage which structure of a chip is the most vulnerable, designers could protect it avoiding the extra cost of protecting all the structures of the chip. It should be noticed that the AVF also depends on the application as different applications use different parts of the integrated system.

## 2. HARDWARE FAULTS AND SOFT ERRORS

There are many effects related to hardware or soft errors. For example, an error bit at a microprocessor may have no effect, may change the expecting output of the running program or may cause an error able to terminate the operation of the computing system.

There are three types of faults than can affect several computing systems. The first category is the soft errors known as transient faults. The second is the intermittent faults and the third one is the hard (or permanent) faults. In the next subsections, we are going to describe in depth these three fault categories as shown in Table 1.

**Table 1: Hardware Faults and Soft Errors**

<b>Faults</b>	<b>Occurrence</b>	<b>Sources</b>
<b>Soft error</b>	instant bit fault, disappears on next, write at the bit	cosmic radiation, voltage fluctuation, transistor variability
<b>Intermittent fault</b>	remains for some executing cycles, repeated after a period of time	wear-out, oxide relegation, process differentials, industrialization residuals
<b>Permanent fault</b>	always wrong output	The age of devices, materials wear-out, manufacturing characteristics

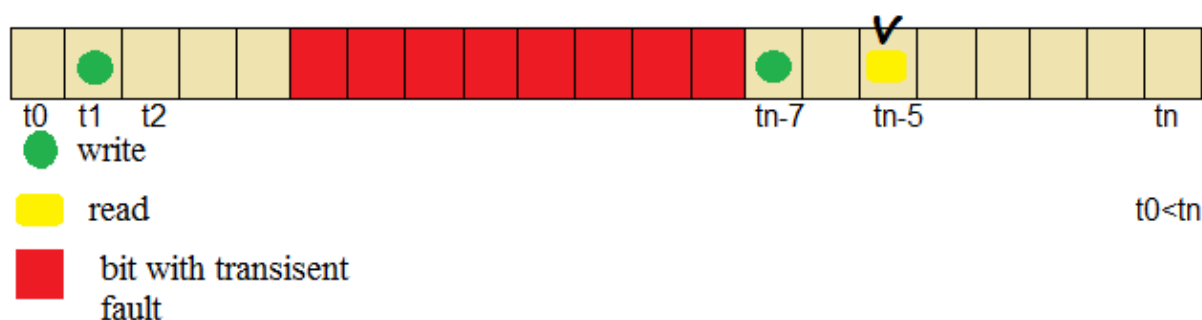
### 2.1 Soft errors

The first category is soft errors, the type of faults that will be studied in my implementation model. One of the main reasons that cause soft errors in computer applications is the induced radiation. Other factors that provoked transient faults are alpha particles, cosmic rays and transistor's variability. There are three prevalent radiation mechanisms which can lead to soft errors: cosmic neutrons which transfer

high-energy interact with silicon and other device materials, cosmic neutrons which transfer low-energy interact with high concentrations of  $^{10}\text{B}^1$  in the device and alpha particles vented from trace radioactive corruptions in the device materials.

The soft error leads to a bit flip, so the fault effect remains after the fault disappearance. In contrast, intermittent fault leads to a stuck at logic 0 or 1 of a bit value for an amount of cycles, but after its disappearance the bit takes the values of the normal operation. The increment of soft error rates (SER) has triggered computer architecture research to provide solutions in order to moderate soft errors.

The reliability of the program's outcome at this case depends on the sequence that several events execute. For instance if a fault event upset happens between two continuously write operations and a read operation occurs after the 2<sup>nd</sup> write, then the read operation's outcome is correct (Figure 1). From the other hand if the read operation occurs exactly after the fault event, the wrong value propagates to the output (Figure 2).



**Figure 1: Soft errors that do not influence the outcome of the program**

<sup>1</sup> Beryllium-10 ( $^{10}\text{Be}$ ) is a radioactive isotope of beryllium. It is composed by the cosmic ray spallation of oxygen. Beryllium-10 decays by beta decay with a maximum energy of 556.2 keV. High energy galactic cosmic ray particles react with light elements. The spallation of the reaction products is the source of  $^{10}\text{Be}$ .



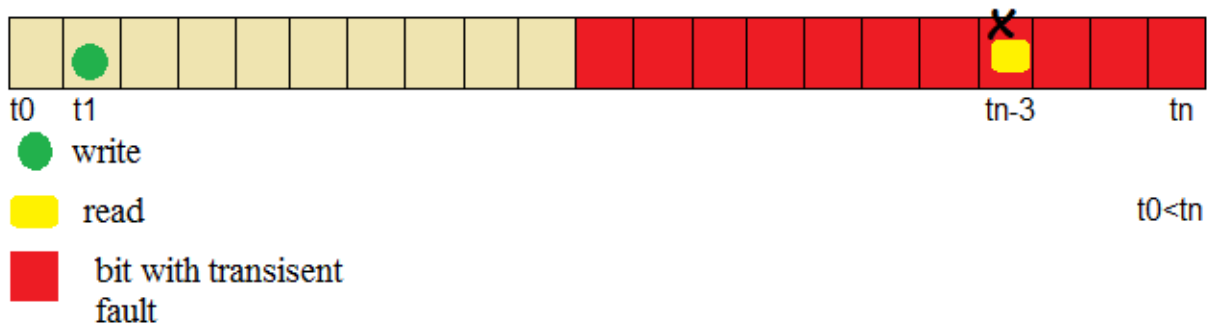


Figure 2: Soft errors that result in a wrong outcome of the program

## 2.2 Intermittent faults

The intermittent hardware faults occur very often and occasionally for a period of time. The dominant reasons that cause them are oxide relegation, process differentials, industrialization residuals and in-progress wear-out. An intermittent error is activated every time at the same place or is caused from the same module. As a result, even though this faulty component will substitute, the intermittent faults will eliminate. Also it is not sure that an intermittent fault will be activated or not during the lifetime of a chip, it can be deactivated and reactivated because of the environmental changes and the process.

The duration period of this fault varies and depends on the factor that causes the fault. For example, the duration of a fault caused by in-progress wear-out will last some days and its effect may be similar to a permanent fault, whereas an intermittent fault caused by temperature and voltage change will last at most several seconds.

An intermittent fault will take place at burst. Burst informs us about the times of activations through the appearance of the fault. As active can be considered the duration of each fault's activation and as inactive time can be considered the period between two continuous activations.

It is not certain that an intermittent error will result in a wrong output. For example, if write operations execute after an intermittent fault, then the next read operation will execute with the correct bit's value (Figure 3).



- SDC: The first category is SDC (Silent Data Corruption) at which the program generates incorrect outcome. This type of fault is the sneakiest one as the user is not aware of the bit's corruption and as result it is impossible to realize that the outcome of the process is incorrect.
- DUE: The other category of faults is DUE (Detected Unrecoverable Error). At this case an error is detected but with no ability to be corrected. The process executes but with error indications that express with ISA exceptions.
- Masked faults: An additional category is masked faults. Masked faults let the program to execute until its end with the outcome of the application and several exceptions which occur during the execution.
- Timeout faults: Timeout faults are the faults that result in Deadlock or Livelock. At a Livelock the program flow has changed and the execution of the instructions happen in random code areas. At Deadlock the program flow has corrupted and no more instructions can execute. In order to deal with these two situations, a timeout limit is used for terminating the execution after this time limit.
- Crash faults: On the other hand, crash faults consider all the cases that the execution of a program results in an unrecoverable situation. In this case, crash fault terminates the execution.
- Assert faults: The last category is the assert faults at which the simulator has reached at a condition, unable to handle it. At that point the execution stops by an assertion.

Vulnerability is measured in FIT (Failure in Time). FIT is one of the two commonly used unit for error rates. The other is MTBF (Mean Time Between Failures). The majority of the designers work with FIT which is reciprocally related to MTBF because FIT is additive. One FIT specifies one failure in a billion hours. Zero error rate equals to infinite MTBF and zero FIT. For example, 100 years MTBF equal to  $10^9 / (24 \cdot 365 \cdot 100)$  FIT. The FIT/bit of a cell typically ranges between 0.001 - 0.010 [28].

The majority of the designers work with FIT which is inversely related to MTBF because it is additive in contrast with MTBF that is more intuitive. They use multiple computer models in order to compute the FIT rate for every chip's device: latches, RAM cells, logic gates. Moreover, with several mitigation and error protection techniques it is easy to evaluate whether a chip meets its soft error budget. The overall FIT rate of a

chip is calculated by summing the effective FIT rates of all chip's structures (logic gates, RAM, latches), where the effective FIT rate for a structure is the product of structure's vulnerability factor and the raw circuit FIT rate.

It is important to be mentioned that many faults do not lead to a system corruption or an incorrect outcome. For instance, a fault in a branch predictor structure will just provoke a delay in the processor's performance. In another example, a corrupted bit may not be used in the program execution so the program outcome will be correct. At this moment the idea of AVF is introducing. AVF is computing the probability that a fault in a structure will result in an error in a program's output [19]. AVF analysis method will be explained clearly later as it is the main goal of this thesis.

### 3. Vulnerability Factors

Soft errors will become an increasingly important problem in the future computing systems, so it is crucial to deal with them. Of course there are workarounds but they are time consuming or they decrease the performance of the system, enlarge the chip size and the consuming power. As the number of transistors is expected to grow up exponentially the next years according to Moore's law, the number of soft errors will increase, too. Consequently, methods that will provide an early assessment of chip's reliability are very important.

Vulnerability factors known as soft error sensitivity factors indicate the probability that an internal fault in a device's operation will result in a visible external error. As an example, if a latch is accepting data with a bigger frequency than holding data, a fault bit may not result in an error of the program's outcome because the wrong value of the bit is likely to be overridden by another value. In other case, if a latch accepts data with a smaller frequency than holding them, then a bit flip is more likely to provoke a visible error.

There are many studies that describe several definitions of vulnerability factors. The main definition was AVF (Architecture Vulnerability Factor) that was proposed from Mukherjee et al. and it concerns the probability of a soft error to result in an error of the program visible output [19]. This vulnerability factor describes the masking probability of the entire system stack and is the metric that we are measure in this thesis. Next, Sridharan et al. proposed PVF (Program Vulnerability Factor) that was responsible to characterize the coalescent soft error masking rate of the software layer [20][18]. Bower et al. proposed the H-AVF (Hard-Fault Architectural Vulnerability Factor) that is used to compare alternative hard-fault tolerance schemes [20]. All these types focus on the masking probability of soft errors in hardware, software or system's stack. The last vulnerability equation type is IVF (Intermittent Vulnerability Factor) that expresses the probability of an intermittent error to cause an external visible error.

### 3.1 PVF

PVF is used to characterize the vulnerability of a program with no dependencies on the hardware layer. It refers to architecture level and evaluates the masking effect of soft errors in that level. Moreover, it can be used as a way to express the behavior of AVF during the program execution (runtime AVF) or such as metric to choose the appropriate algorithm or the appropriate compiler optimizations in order to reduce the vulnerability of a program due to soft errors. The type for its computation is:

$$PVF = \frac{\sum_{i=0}^I N_{A-bit}^i}{B \times I}$$

Where I is the total number of instructions in the program , B represents the total bits in the architecture structure and  $N_{A-bit}^i$  represents the number of bits type A in instruction i.

### 3.2 H-AVF

H-AVF helps designers to make comparisons between different hard-fault tolerance techniques. The main aim of H-AVF computation is to provide information about different designs and used it to compare hard-fault tolerance designs. This information will be used to compare hard-fault tolerance designs.

The equation type is:

$$H - AVF = \frac{1}{N_i} \times \frac{1}{N_f} \times \sum_{\forall fault} \sum_{\forall inst} inst_{error}$$

Where  $N_f$  is the total number of the faults sites in the structure,  $N_i$  is the total number of instructions in the program and  $inst_{error}$  is the number of instructions that will corrupt because of the hard-faults [20].

### 3.3 IVF

IVF (Intermittent Vulnerability Factor) measures the probability that an intermittent error will manifest an external visible error. The computation of IVF is very helpful for the designers as they can use it during the design of microprocessor in order to combine high reliability and good performance at the same time. It has been proved experimentally that IVF differs across different manufactures or workloads, so more protection can be added to the most vulnerable structures. Large percentages of IVF indicate that the structure is vulnerable to intermittent errors.

The equation type of IVF at the register file is:

$$IVF_{reg} = \frac{\sum_{e=1}^E U_{CT}^D(e)}{E}$$

Where E points to the number of entries in register file, and  $U_{CT}^D(e)$  indicates if an intermittent fault has occurred in a critical period of time or not. Numerator  $\Sigma$  adds together the total number of all affected registers during the time interval that the intermittent fault exists.

### 3.4 AVF

A crucial aspect of AVF analysis is that some single-bit faults such as those occurring in the branch predictor will not produce an error in a program's output. AVF is the probability that a bit fault in the structure will result in an error, so the final outcome of the program will be different from that one that was expected. Thus, it has to be declared that not all faults in a microarchitectural structure will affect the final output of a program. For example, any committed instruction will not be affected from a single bit fault in a branch predictor; hence, the AVF for a branch predictor is 0%. On the contrary the program's outcome will be affected for sure if a single bit fault occurs in the committed program counter. At this case the wrong instructions will be executed and as a result the AVF for the committed program counter is 100% [19].

The AVF for most of the structures is between 0% - 100%. AVF in combination with the raw fault rate induce the calculation of the overall error rate of a microarchitectural structure. Summing up, the raw fault rate that is detected by the process and the circuit technology can be mapped from a processor architect to an overall processor error rate and thus determine whether the design meets its error rate goals.

There are several methods for computing the AVF of a hardware structure.

- ACE analysis: A set of these approaches is based at ACE analysis (Architectural Correct Analysis) at which every bit can be ACE or un-ACE. A bit is un-ACE for the interval when its value can be flipped without affecting the final program outcome. Otherwise if this change affects the final outcome then the bit is ACE. Ace analysis will be presented in Section 3.4.1 with more details.

- **Statistical Fault injection:** During fault injection experiments, the output of a golden run is compared with that of a run that an injection of a fault was occurred. An injection can target the system or the application software or the hardware structures of a simulator. There is also injection in the RTL (Register Transfer Level) level. An RTL model represents the micro-architecture of a circuit injection. It is implemented in HDL (Hardware Description Language) for example, VHDL. Register variables are used to store data, whereas transformations represented by arithmetical and logical operators. RTL model composes a gate-level model of the design.
- **Probabilistic methods:** Probabilistic methods for evaluating AVF provide early and reliable estimation. They are about models with high level performance, coupled with low level information about processors' reliability. These models are available at early stage of the design.

ACE analysis identifies which bits are necessary for architecturally correct execution (ACE bits) of a program. Furthermore, it measures the percentage of ACE bits in a hardware structure. When an ACE bit is corrupted there is a visible error at the outcome of the program. This analysis originally assumes that all bits in a hardware structure are ACE bits, after that finds the bits that can be proven unnecessary for the correct execution of the program (unACE bits). It is significant that ACE model can be performed early in the design cycle by the hardware designers.

All these methods for evaluating AVF are really fast. After few runs of the benchmark and since the simulator is configured according to the requirements of the computing system you can have an AVF estimation in contrast to the fault injection that is really time consuming as needs a lot of runs for the same benchmark before providing the final assessments. Nevertheless all these approaches have one major disadvantage, they over-estimate the vulnerability of microprocessor structures in contrast to the fault injection method.

Especially ACE analysis overestimates the vulnerability of soft error 3 x times in average against the vulnerability computed using fault injection [1]. Recent studies have shown that this overestimation of the AVF can be decreased by adding more details about the RTL (Register Transfer Level) model. The accuracy of AVF is intrinsically connecting with detailed models execution. The overestimation of the AVF will be



presented in the next sections in more details, where we compare the AVF calculated in this study for different benchmarks and structures with that obtained by [1].

### 3.4.1 ACE analysis

In order to compute AVF we need to determine which bits are ACE bits, affecting the program outcome and which are un-ACE bits that do not affect the final output. Un-ACE bits divide into two categories.

- Microarchitectural un-ACE bits
- Architectural un-ACE bits.

#### 3.4.1.1 Microarchitectural un-ACE bits

An un-ACE microarchitectural bit is correlated with an idle/invalid state, a miss-speculated state, an ex-ACE state or a predictor state. In more details:

- Idle/invalid state: A status bit or data can be characterized as un-ACE when is idle or does not contain any valid information. Nevertheless control bits are always ACE-bits and a fault on a control bit may result in error.
- Miss-speculated state: The bits that used to represent a wrongly speculated operation such as branch prediction, are un-ACE. These operations are performed more and more often at modern microprocessors.
- Ex-ACE state: An ACE-bit after the last time it was used by a committed instruction becomes un-ACE (dead bit). With that state can be described both the architecturally dead value and the architecturally invisible states.
- Predictor state: All kind of microprocessors' predictors such as branch, jump, store-load dependence predictors and stack predictors consist of un-ACE bits. A fault in that structure most of the times will result in a misprediction. This misprediction will affect the performance of the program but will not result in an error at the final output.

#### 3.4.1.2 Architectural un-ACE bits

An un-ACE architectural bit can be correlated with NOP instructions, performance-enhancing instructions, predicated-false instructions, dynamically dead instructions and

logical masking. These bits affect the correct-path instruction execution, but they do not affect the output of the program. Those kinds of bits that are not used in the constitution of the ACE path are called un-ACE instruction bits. In more details:

- **NOP instructions:** NOP instructions do not affect the architectural state of the processor and also there are at the majority of the instruction sets. They are used in order to align instructions to address boundaries or to fill VLIW-style instruction templates. There are some ACE bits in the NOP instruction that distinguish it from a non-NOP (all the other bits are un-ACE bits). These bits can be the opcode or the destination register specifier, depending on the instruction set.
- **Performance-enhancing instructions:** Performance-enhancing instructions are included at the most of instruction sets. In a non-opcode field a single bit error will not affect the final outcome of the program. For example, a single bit upset at a prefetching instruction may cause the address to become invalid (the prefetching will be ignored at this case) or the wrong data will be prefetched. Nevertheless the program's output will not be changed. Thus, the non-opcode bits are un-ACE bits.
- **Predicated-false instructions:** Predicated instruction-set architectures are based on a predicate register in order to decide if an instruction will be executed. The instruction will commit only if the predicate predictor is true otherwise the instruction will be discarded. Thus, all bits in a Predicated false instruction set are un-ACE bits except the predicate register specifier bits. A fault in those bits may result in a false prediction for the instruction, so these bits are called ACE instruction bits. If the instruction is dead and there is a fault in the predicate register, it will not result to any problem at the computation of the program as this instruction is not going to be committed again. The predicate register as well as the corresponding specifier can be considered as un-ACE bits for that case.
- **Dynamically dead instructions:** Dynamically dead instructions are the instructions with unused destination registers. There are two types of dynamically dead instructions the first-level dynamically dead (FDD) and the transitively dynamically dead (TDD). FDD results are not read by any other instruction. The TDD instructions lead to FDD or other TDD instructions

- Logical masking: Logically masked are the bits that belong to operands in a chain of computation that their bit values are not used do not influence the final computation.

### 3.4.2 AVF Equation

The AVF for a storage cell is the percentage of time at which the cell contains an ACE bit. For example, if a storage cell contains ACE bits for 10000 cycles out of an execution of 100000 cycles, then the AVF for that cell is 10%. The equation of the AVF for a whole hardware structure is similar with the above equation [19]. The AVF for the whole structure equals to the average AVF for all bits' structure.

The equation for AVF of a hardware structure is equal to:

$$AVF = \frac{\text{Average number of ACE bits in a hardware structure in a cycle}}{\text{Total number of bits in the hardware structure}}$$

As at this study a simulator will be used for extracting data in order to compute the AVF, the above type can be more specified and rewritten as:

$$AVF = \frac{\sum \text{recidency (in cycles) of all ACE bits in a structure}}{\text{Total number of bits in the hardware structure} \times \text{total execution cycles}}$$

Using the aforementioned type, in this study we calculated the reliability of two hardware structures (the physical integer register file and the L1 data cache).

### 3.5 Diagnostic and Protection mechanisms for hardware fault

A memory error can influence both the performance and the reliability of the computing system, so it is important for a computing system to be designed with a way in order to be reliable. The reliability of the system can be achieved by using detection and diagnostic mechanisms for hardware faults as long as mechanisms for system recovery after the fault detection.

These tolerance mechanisms encumbrance the computing system in terms of execution time, memory capacity and the cost. The cost of a common memory detection technique for hardware errors with capabilities for recovering can be from very low to

very high. Mechanisms such as detection or correction of an error, being added to the system result to slowdown of the execution time of the benchmarks.

There are two types of tolerance mechanisms. The software level techniques as long as the hardware level techniques [13][1]. Software-level methods are responsible for detection and do not impose any area overhead. The hardware-level techniques do not use any software intervention for the detection or protection, so they increase the hardware cost of the computing system. For that reason the selection of best techniques about the diagnosis, the detection and the correction of hardware errors has to take into account all the characteristic of the computing system and to take place in an early stage of the design phase. In this way the high cost of redesign cycles on later integration will be avoided.

However is really difficult to choose the most appropriate mechanism as a lot of knowledge is missing at the early stages of the design procedures. The designer is not aware of the workload, the architecture of the system or the different hardware sizes. At this moment the usage of a microarchitecture simulator against RTL model is undoubtedly the best choice. It can give us an effective reliability estimation with much accuracy at the early stage before the design of the computing system.

The microarchitectural simulators are remarkably faster than the simulators in the RTL level and make easier the study of large and realistic benchmarks. Moreover, the fact that they can be used at the early stage of the design give architects the opportunity to configure many parameters and hardware structures of the computing system. A microarchitecture simulator is responsible for modeling all the microarchitecture components of the system, such as the arrays for storage in a chip area. As a result, it determines the vulnerability factors for structures like: register files, caches, buffers, queues. Also, they are important for multiple performance studies since they allow study on software's execution with big duration [1]. In this study, we use the GEM5, a microarchitectural simulator to measure the AVF of physical integer register file and L1 data cache.

## 4. Gem5 Simulator Overview

Gem5 simulator is a free open source software platform that supports simulation of multiple platforms as described below (Instruction Set Architectures – ISA) like ARM, Alpha, MIPS, Power PC, SPARC and X86 64-bit. All system components of the simulator are configurable. Gem5 is also a combination of two older simulators, M5 and GEMS. Specifically M5 has contributed with its functional full system simulator and GEMS with its memory modeling capabilities.

Below we present the main details of each ISA:

- ALPHA: the most used ISA on Gem5 simulator. Alpha architecture based on a DEC Tsunami system that can be extended for up to 64 cores.
- ARM: models a Cortex-A9 and offers support for Thumb, Thumb-2, VFPv3, NEON instructions set extensions.
- X86: models a X86 CPU (64 bit) which boots unmodified Linux Kernel in a SMP configuration (this ISA will be used in this study).
- SPARC: models an UltraSPARC t1 processor which boots Solaris.
- PowerPC: models a 32-bit processor based on POWER ISA v2.06 B.
- MIPS: models a 32-bit processor.

### 4.1 M5

The M5 project was started at University of Michigan as a full system simulator to simulate large networked systems and explore designs of network I/O. M5 simulator offers a configuration environment for multiple Instruction Set Architectures, ISAs and CPU models.

M5 consists of two CPU models, SimpleCPU and O3CPU. One of its characteristic is that M5 simulator can change CPU models during the runtime for example, can change between SimpleCPU to O3CPU if there is need for taking statistics and from O3CPU to Simple CPU if there is need for warm-up operation and forwarding.

Specifically the model of SimpleCPU is not a pipeline model. It is about an in-order model with only one outstanding memory operation. Its configuration can be such to

execute one or more instructions per cycle. Except from the previous operation SimpleCPU is also used to model network client systems.

On the other hand O3CPU is an out-of-order model, pipelined, concurrent, superscalar and multi-threading (SMT). It has been developed to provide timing accuracy. For that reason timing and functional modeling have integrated into a single pipeline execution (functional instructions execute at execute stage of timing pipeline). This type of model has several stages like decode or fetch that can be configured in their own attributes such as latency. Time buffers are responsible for the communication between these stages. In addition, the O3CPU model can simulate particularly an out-of-order pipeline as long as it includes branch predictors, store/load queues, instruction queues, predictors for memory dependence and functional units. O3CPU model is going to be used in this thesis.

According to memory system, M5's memory consists of two types of objects, the devices and the interconnections. As devices can be considered caches, I/O devices or memories. About the interconnection network there are two models: "Simple" and "Garnet. Simple network model is presented by default and traverse the network hop-by-hop while it abstracts out detailed modeling within the switches. On the other hand Garnet is a more detailed interconnection network model. It consists of flexible and fixed pipeline model and it uses routing tables, variable link bandwidth and multi-cast messages. Simple network model is faster than Garnet.

M5 can support caches with configurable parameters as size, associativity, replacement policy, latency etc.

## **4.2 GEMS**

The GEMS simulator was started at University of Wisconsin. GEMS features a timing simulator of a multiprocessor memory system called Ruby) which is used to model different cache coherency protocols. GEMS also supports interconnect models (network connection). This merge into Gem5 has taken the best aspects from these two simulators.

### **4.3 Fundamental requirements of Gem5**

Gem5 constructed according three fundamental requirements: flexibility, availability and collaboration.

Regarding to flexibility Gem5 has two execution models. It can run as a System-call Emulation (SE) or as a Full System (FS). In a SE mode all system calls are handled by the simulator and only the user space program is simulated. SE mode is really fast, if only memory operations need to be observed. On the other hand in a FS mode all systems calls and user space are simulated. Fs mode operation is slowly and provides Linux as the environment of the simulation. Also, it consists of two memory system models, the Classic and the Ruby. Ruby features in GEMS and implements a domain specific language called SLICC (Specification Language for Implementing Cache Coherence) which is used to model multiple cache coherency protocols. The Classic model features in M5 and provides a configurable memory system. Moreover Gem5 supports 4 multiple CPU models, each one has a different point across the speed vs. accuracy spectrum. The AtomicSimple model is a minimum model of an IPC (Instructions Per Cycle) CPU, the TimingSimple model is a similar model which is configured some timing characteristics. The InOrder model simulates a pipelined in-order CPU while the O3 model simulates a pipelined out-of-order CPU. Another proof of its flexibility is the fact that it is really easy to apply a wide range of investigations if you become familiarized on it.

Gem5 is available for both academic and corporate researchers. Its license is based on BSD license and there is no dependence on proprietary code. Gem5 as an open source software is combined effort of many people with different specialties (researches, students, engineers etc.). The community of Gem5 is really active and uses different collaborative technologies like mailing lists, wiki, a management system for the several code changes based on web and a public source repository.

### **4.4 Memory System in Gem5**

The memory system in Gem5 has been designed with modularity through several interfaces, flexibility as different cache models interconnects and an inclusive set of buildings. Memory system consists of Memobjects, ports, connections and the port proxies.

#### 4.4.1 Memory System Models

As the GEMS simulation system has primarily been used to study cache-coherent Shared memory systems (both on-chip and off-chip) and related issues, those aspects of GEMS 1.0 release are the most detailed and the most flexible.

The heart of GEMS is the Ruby memory system simulator. The other model is the classic. Classic model is fast, easy and can configure easily the memory system. All memory objects are connected via ports. This model supports fast forwarding. Atomic accesses and timing as long as are really fast at this model. Also, it is easy to be configured and to keep up with other memory models. One of its disadvantage is that is not able to model protocol contention.

Ruby as a timing model of a multiprocessor memory system responsible for modeling caches and their controllers, system interconnect and also the bank of the main memory as long as the memory controllers. Ruby is the model that can have a combination between timing simulation for modules that have no dependence with the cache coherence protocol such as interconnection network and a specification language for Implementing Cache Coherence (SLICC). The objects do not connect via ports like classic model but via RubyPort object. The disadvantage of that model is that does not support fast forwarding. Moreover, in contrast to the classic model is slower and is difficult to simply extend protocols to other level of cache. An overview about Ruby is at Figure 5.

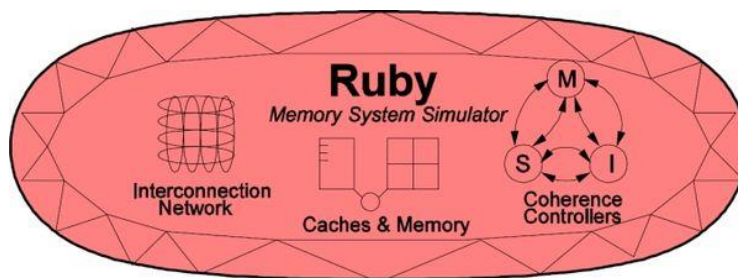


Figure 5: Ruby Simulator overview



In Figure 6 a high level overview of Gem5 shows. It concerns all modules that interact with it from the first step until the last of a program execution. To run the simulator it takes a Linux kernel, a disk image and a configuration file. To run the simulator it takes a Linux kernel, a disk image and a configuration file.

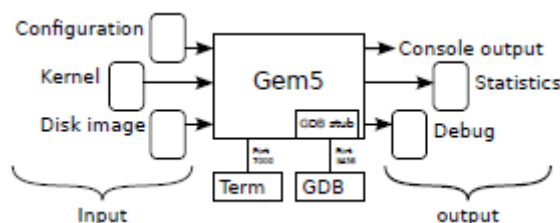


Figure 6: Gem5 high level overview

According to the Figure 6 the console output, the statistics file and the debug information are the output of the simulator. The statistics file (stats.txt at m5out folder), which contains statistics that are collected during the simulation is dumped to a file at the end of simulation. The debug output can be controlled using flags during the build.

Every program that executes in Gem5 can be accessed besides having the input and output while it is running by connecting a terminal or GDB over specified network interfaces. Attaching GDB can be used to debug the simulated system.

#### 4.4.2 Port system

Port System consists of Memobject, Ports, Connections and Port Proxies.

- **Memobjects:** Every object in a memory system is inherited from *MemObject*. The class of Memobject allows the connection of memory objects. Its functions return the name of the port that is going to be used for the connection.
- **Ports:** Every MemObject should have at least one port in order to be useful. Each port can be master or slave. The master port connects to slave. All the ports come to peer.
- **Connections:** All the entries with info about the connection are saved at a vector port.
- **Port Proxies:** There are three types of port proxies. The first one is the PortProxy and is used for setting and loading physical addresses. The other two types are

SETranslatingPortProxy and FSTranslatingPortProxy. These ports use virtual addresses.

#### 4.4.3 Packets

The encapsulated transformation among two objects in the memory system is done via packets. A packet is readable only in case of its value is valid. Its class has many information about the size, the address etc. Packet's fields are accessed by assessors in order to ensure that the data in the packet are valid. All these fields as long as some new fields that I have added will be explained with more details later in the implementation model of this study.

#### 4.4.4 Requests

The initial request issued by CPU or I/O device is encapsulated in a request object. Request's fields are accessed by assessors in order to ensure that the data are valid. A request is demanded for a request packet construction. Request fields such as physical address, its size etc. will be explained in details later.

#### 4.4.5 Atomic, functional, Timing access

The ports support three types of accesses: Atomic, Functional and Timing. *Atomic* accesses are really fast and used to forward caches. They return the expectation time that needed to complete the request without taking on mind the queue delays. Atomic and timing accesses cannot coexist in the memory system. *Functional* accesses have the same period with atomic accesses. The disadvantage of that kind of accesses is the fact that they can coexist in the memory system with atomic or timing accesses. They are used for loading binaries and changing variables in the simulated system. In the end *Timing* accesses are the accesses with the more details. They represent a real time model and take into account the queue delay in contrast to the atomic accesses.

There is a need of a flow control, because timing requests simulating a realistic memory system are not instantaneous. The flow control is responsible for guiding when a packet will be resent after the first failed sent. The packet will be sent again only in case of received a value that will ensure that the packet can be send again.

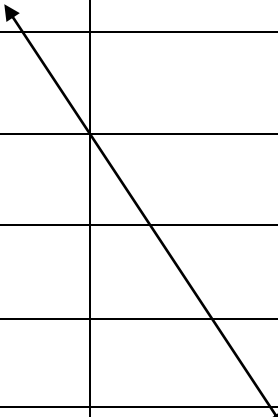
## 4.5 CPU models

The CPU models in Gem5 are the following: Simple CPU Model, Out-of-Order CPU model, In Order CPU Model and Trace CPU Model.

In Table 2 about Memory system models and Cpu Models sum up all the appropriate information about that models.

**Table 2: CPU Models and Memory system**

<u>Processor</u>		<u>Memory System</u>		
CPU Model	System Mode	Classic	Rubby	
			Simple	Garnet
Atomic Simple	SE			
	FS	Speed		
Timing Simple	SE			
	FS			
In - Order	SE			
	FS			
O3	SE			Acuracy
	FS			



### 4.5.1 SimpleCPU

Simple CPU is a functional, in-order model without representing a detailed model. It is divided into three sub models, the BaseSimpleCPU, AtomicSimpleCPU and TimingSimpleCPU. The BaseSimpleCPU model determines functions for checking interrupts, setting up fetch requests, handling pre and post executing actions and for advancing the PC to the next instruction. The AtomicSimpleCPU model uses atomic memory accesses, while the TimingSimpleCPU model uses timing memory accesses. One main difference between AtomicSimpleCPU and TimingSimpleCPU is that AtomicSimpleCPU waits until memory access returns, so the stages of fetch and memory may last more because of the fetch delay or LD/ST delay. This difference is shown at Figure 7 and Figure 8 below. Simple CPU also defines the port that is used to hook up to memory and that connects the CPU to the cache as long as determines the necessary functions for handling the response from memory to the accesses sent out.

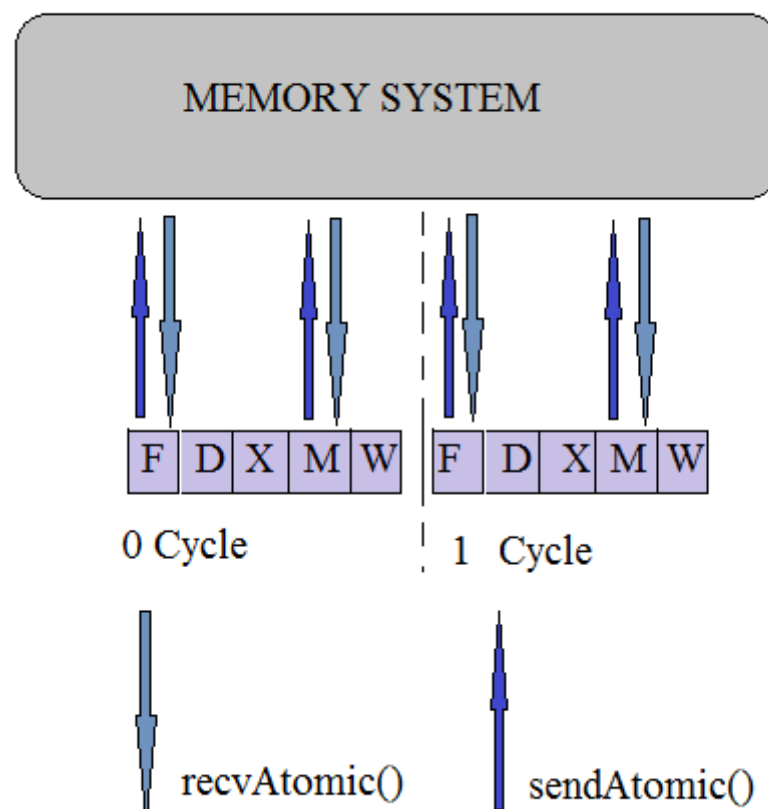


Figure 7: CPU Model AtomicSimpleCPU

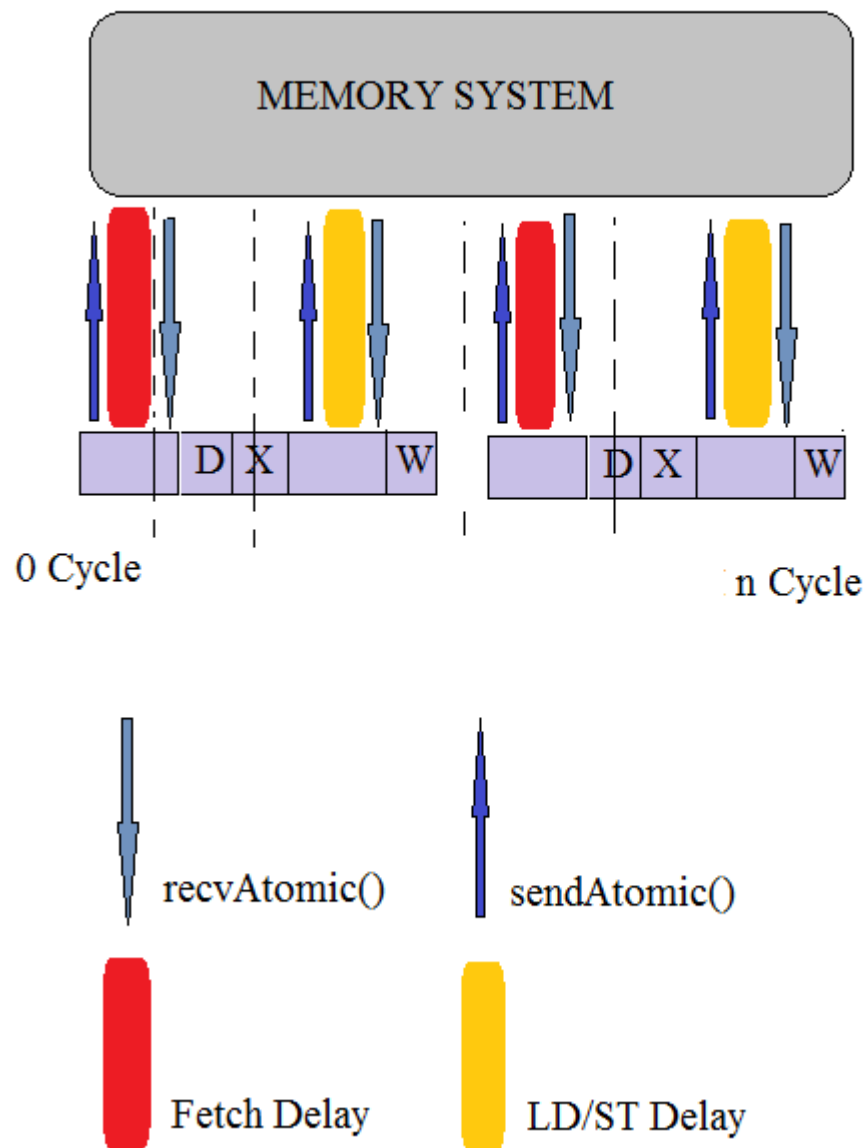


Figure 8: CPU Model TimingSimpleCPU

## 4.5.2 O3CPU

The Out-of-Order CPU (O3CPU) model that is used is an out-of-order CPU model. Next, we present all the pipeline stages and resources. Pipeline stages also are presented at Figure 9.

- Fetch: The first stage of pipeline is fetch. At this stage the dynamic instruction («*class DynInstptr*») will be created for the first time. The object of «*class DynInstptr*» represents the dynamic instruction (more details for that structure will be given in the following section). Also, it selects the thread that is going to be fetched as long as it is responsible for branch prediction.
- Decode: The other stage is Decode, and is used to handle the PC at unconditional branches. The next stage is rename. This stage uses physical integer register file and also renames architectural registers to physical registers according to the programs' needs.
- IEW stage: One of the final stage is the process of:
  - Issuing the instruction:
  - Executing the instruction
  - Writing back the instruction

This stage is a combination of processes as it can handle both execute and writeback. Also manages dispatching instruction to the instruction queue.

- Commit: The last pipeline stage is the commit. It is the stage at which an instruction is committed or not. This stage is the most important for ACE analysis, because only the instructions that are finally committed can corrupt the output. Instructions that did not arrive at this stage have not be measured in the ACE analysis presented in this study.

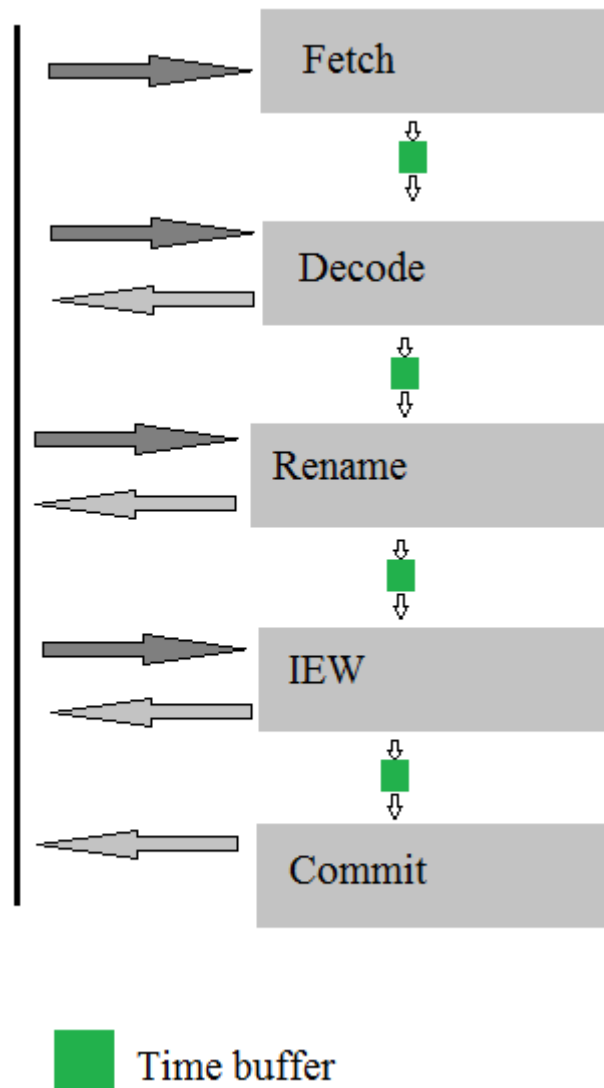


Figure 9: O3CPU pipeline

Each pipeline stage consists of several structures (queues, buffers, predictors and functional units etc.). In more details, the pipeline resources are:

- Branch predictor: Branch predictor allows the selection between a local, a global and a tournament predictor.
- Reorder buffer: Reorder buffer not only handles the instruction that are squashed but also holds all the instruction in program order.

- Instruction queue: Instruction queue acquires several dependencies among instructions and uses the memory dependence predictor for scheduling when an instruction is ready.
- Load-store queue: Load-store queue holds accesses to the memory system that have reached the back-end. When memory operations issue and start to execute then load-store queue hooks up to L1 data cache and initiates the accesses. Moreover it is used to detect memory violations, to replay memory operations in case of blocked memory system and to handle the forwarding operation between store and load actions.
- Functional units: Functional units determine which instruction can be issued at each cycle as long as the latency of the executed instruction.

### 4.5.3 InOrder

In Order CPU model provides a generic framework for in –order pipelines without a specific ISA or pipeline description. So this model provides generic pipeline stages of Fetch, Decode, Execute, Memory, Writeback. If an instruction cannot complete all its resources requests in one stage, then it blocks the pipeline. It can be shown at Figure 10.

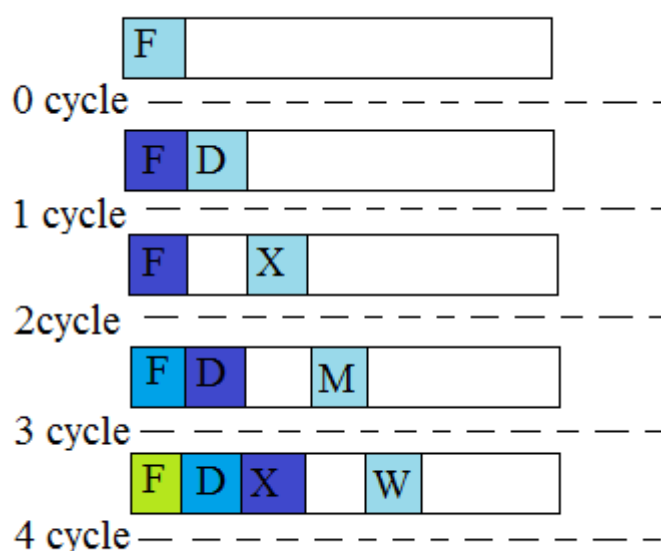


Figure 10: InOrderCPU pipeline



## 4.6 Extensive – object oriented design

The flexibility of Gem5 is a consequence of its object oriented design. The way of its design and systems' configuration with independent items lead to the modeling of multiple CPU models and systems.

All basic elements of Gem5 called SimObjects and have common behavior concerning the configuration, initialization, the collection of statistics and the serialization. The SimObjects contain independent constituents of hardware like cores of CPU, cache memories, interconnections data and devices. Furthermore there are some abstract entities, the workload and the System – Call Emulation (SE).

Every SimObject is represented by two classes, the one is written in Python and the other in C++ and is inherited by the main class SimObject that is written in both C++ and Python. The definition of the class in Python sets SimObject's parameters and is used during the configuration through a script file.

The main Python class provides consolidated mechanisms for the initialization, the definition of the parameter's value and the name of the variable. The main C++ class holds information about the SimObject, its behavior and the characteristics of the simulation process.

## 4.7 Python

Python is a scripting programming language. It is really popular because of its flexibility and easiness of use. Except from the fact that the majority of the code in Gem5 is written in C++, Python has a very big distribution. Every SimObject as already mentioned is written both in Python and C++. The script files in Python offer initialization, configuration and the control of the simulation. The main() function as long as the code for the command line process and the boot process are written in Python. At the beginning, the simulator runs Python file. All the configuration scripts used in this study for the size of physical integer register file or the L1 data cache associativity are written in python.

## 4.8 Physical Integer Register File

Three types of registers are used in CPU models of Gem5. The first one is the physical register, the next is architectural registers and the last one is Condition code registers (CC). The index of a physical register is the index that is encoded in the instruction. There are two register classes, the one for integer registers and the other for float registers. In this study, we make ACE analysis only for the integer register class. The index space for that registers start at 0. Architectural registers have been in order to avoid the managing dependencies that physical registers are not able to deal with.

## 4.9 Gem5 Configuration

Details about the configuration of the original version of the simulator which are related to this study are described in the next Table 3.

**Table 3: Simulator Configuration**

<b>Physical Integer Register File</b>	<b>256 integer registers</b>
<b>Load/Store Queue Entries</b>	<b>16 Load, 16 Store</b>
<b>L1 data cache</b>	<b>32KB, 64B line, 128 sets, 4-way, write-back</b>
<b>L1 Instruction cache</b>	<b>32KB, 64B line, 128 sets, 4-way, write-back</b>

## 4.10 Inside the Gem5

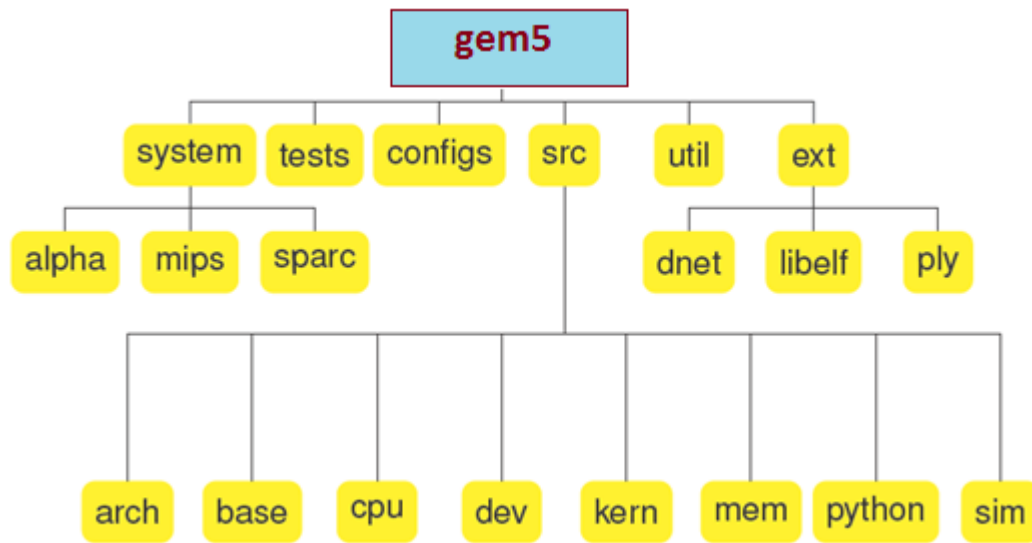


Figure 11: Source code Tree organization for Gem5

More specifically every folder at Gem5 simulator has its own functionality that is described below, see Figure 11.

- System: platform with low level software (firmware, bootloaders) – packaged separately.
- Tests: files related to regression tests.
- Configs: Configuration scripts written in Python that provide some basic prepackaged functionality. Also include some examples that can be used for your own script.
- src: the source code of the simulator
  - src/arch: ISA implementations.
  - src/base: general data structures/facilities
  - src/cpu: Specific models of CPU
  - src/dev: : Specific models
  - src/doxygen: Doxygen templates and output.
  - src/kern: Specific Operating System but architecture is independent code.
  - src/mem: Memory System models.

- `src/python`: Python configuration code.
- `src/sim`: Code for the base functionality of the simulator.
- `util`: utility programs and scripts which are not parts of the Gem5 binary but are generally useful when working on Gem5.
- `ext`: Dependencies that are really hard to find alone, not likely to be available and are generally useful when working on Gem5.

## 5. Simulated system

The simulated system is a Linux system and the release that chosen is Ubuntu 14.4.0. Ubuntu 14.4.0 is flexible and everything can be compiled from source, also is really helpful during the debugging. Gem5 developers propose Ubuntu and concerning the Kernel they recommend to fetch it directly from kernel.org. A Linux Kernel and a Linux disk image are appropriate in order to start a simulation.

### 5.1 The Linux Kernel

One of the components needed for the simulation is a compiled Linux kernel. The Gem5 wiki provides four different configurations of the Linux kernel for x86 64-bit. Newer versions of the Linux kernel are preferable as the goal of this thesis is to provide the most recent software stack. In this study the edition of Linux kernel for X86 Gem5 is `x86_64-vmlinux-2.6.22.9.smp`.

### 5.2 Clock cycle

Clock cycle is the amount of time between two pulses of an oscillator and is the parameter that determines the speed of CPU or a computer processor. It is known that the higher pulses per second, the faster the computer processor will be able to process information.

Clock speed or clock rate is the speed that the microprocessor executes each instruction or vibration of the clock. For each instruction's execution the CPU requires a number of clock ticks or cycles to be executed. The measurement for clock speed is Hz. Typically is measured in MHz or GHz. For example, a 2 GHz processor executes 2.000.000.000 clock cycles per second.

According to the type of the processor, computer processors can execute one or more instructions per clock cycle. Nowadays the modern processors can execute multiple instructions per cycle while earlier computer processors and slower CPUs can only execute one instruction per clock cycle.

A tick is an arbitrary unit for measuring internal system time. There is usually an OS-internal counter for ticks; the current time and date used by various functions of the OS

are derived from that counter. A tick is 1 pico second and in order to convert it to clock cycle is needed to be known how many seconds is a clock cycle. Consequently, a clock cycle can be any number of ticks. For example, for a 2 GHz CPU that is the CPU speed for my implementation, 1 clock cycle takes 500 pico seconds that means 500 ticks (1 tick=1 pico second).

### **5.3 Computation of simulation's clock cycle**

In order to calculate the exact clock cycles of the simulation execution, it is needed to be divided the whole ticks of the simulation by 500 (for a 2 GHz CPU that is the CPU speed for this study, 1 clock cycle takes 500 pico seconds that means 500 ticks). The same calculation has been done in order to compute how many clock cycles is the ACE interval time of the simulation.

## 6. Cache Memory

Cache memory at Gem5 simulator is related on Harvard's architecture. According to that architecture model, cache memory is separated into data cache and instruction cache.

The advantage of this architecture is that the system can fetch an instruction from the instruction cache simultaneously with data from data cache. Other important advantage of that architecture is that keeping instruction cache and data caches separated, prevents conflicts between set of instructions and data. The disadvantage of that model is that the size of lcache (Instruction cache) as the size of L1 data cache (data cache) is not fixable. It is fixed depending on the architecture. The major difference between them is that the data cache must be capable of performing both read and write operations, while instruction cache needs to provide only read operation. This type of memory is really faster than architectures like Newmann. At Newmann's architecture data and instruction cache are the same.

Figure 12 represents the hierarchy between L1 and L2 cache memory as long as the stages that an instruction follows until accesses data cache.

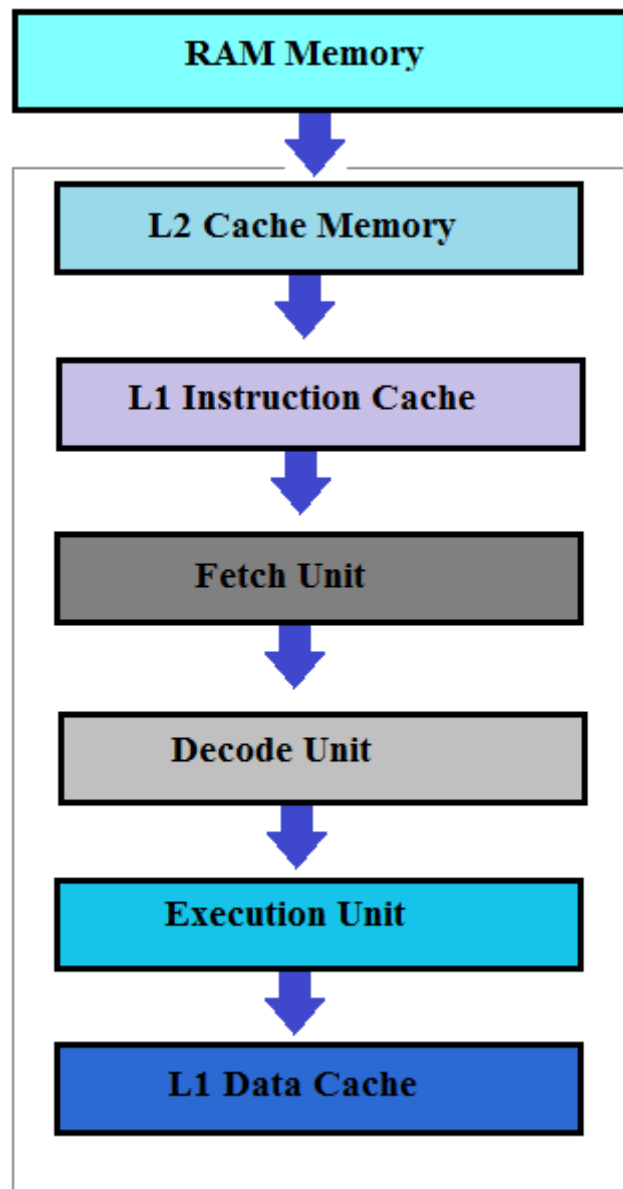


Figure 12: L1 & L2 cache memories

## 6.1 Instruction Cache

Instruction cache holds only the instructions that processor will execute. Usually its size is smaller than data cache as instructions for a program take less memory than program's data. It is very often the same instruction executed so many times during a program execution for that reason designers have decided to devote more chip area at data cache memory.



## 6.2 Data Cache

Data Cache holds temporarily data that processor uses during the program execution. From that type of memory data can be stored or loaded from the memory.

Considering the process of writing data at data cache there are two policies: Write Back and Write Through. At write through policy, the data setting happens at the same time at data cache memory and memory system. For some benchmarks, the disadvantage of the write through method is that continuously memory access will reduce the performance as it is needed to wait until the completion of the previous access in order to access memory again. Write back policy allows data writing to memory system only if some structures of the processor are available. The disadvantage of that policy is the cost and the complexity of the memory. Some writeback caches include write-buffer as temporary storage for lines that are being written back in order to avoid the big delay. Gem5 implements only the write-back policy.

## 6.3 Cache Mapping and Associativity

A very important factor that determines the effectiveness of cache memory is related with the way that is mapped to the system memory. There are many ways to allocate the storage in our cache to the memory addresses it serves. More clearly the associativity is the answer to the question “how will be divided the address lines in cache memory amongst the system memory”. Three different ways can do this mapping in the memory system.

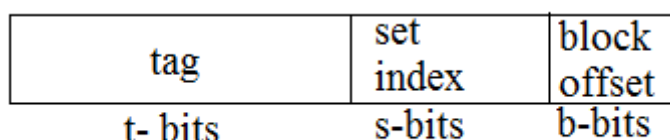
- **Direct Mapped Cache:** The simplest way for memory mapping. The memory system chopped in chunks. The number of chunks is equal to the number of address lines in cache. Then each chunk gets the use of one cache line. This is called direct mapping. Although this way is very simple, it has no flexibility about where to put the blocks in the cache.
- **Fully Associative Cache:** In a fully associative mapping a cache block can go anywhere in the cache. There is no need to allocate cache line to a specific memory location. Every tag must be compared when finding a block in the cache, but block placement is very flexible!

- n-Way Set Associative Cache: "n" is a number, at least 2 as 1-way associative is the direct mapping. It about a conciliation between the direct mapped and fully associative designs. At this design cache memory split into sets, the number of sets depends on the way of association. For example, at a 4-way set associativity, cache memory consists four sets. Each memory address is assigned a set, and can be cached in any one of those four locations within the set that it is assigned to. Generally, "n" means that there are "n" possible places that a given memory location may be in the cache. A n-way associative cache memory, with n blocks is a fully associative cache. In order to compute a set index or to select a set within the cache instead of an individual block the next equation types are used:

- $\text{Block Offset} = \text{Memory Address} \bmod 2^n$
- $\text{Block Address} = \text{Memory Address} / 2^n$
- $\text{Set Index} = \text{Block Address} \bmod 2^s$

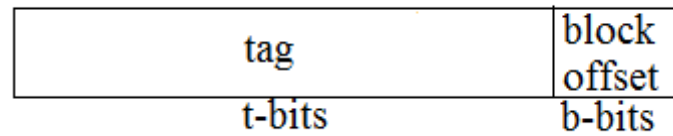
In this study, cache memory is 4-way associative and all the computation about number of sets and blocks computed with the above arithmetical types.

A memory address of m-bits shows all of the information needed to locate the data in the cache. The address consists of three parts: tag, set index and block offset. The length of these fields differs from design to design. The least significant bits are used to determine the block offset. If the block size is B then the block offset needs  $b = \log_2 B$  bits to be specified. The next highest group of bits is the set index and is used to determine which cache set we will look at. If S is the number of sets in our cache, then the set index has  $s = \log_2 S$  bits. The remaining bits are used for the tag. Tag field is used to differentiate the several regions of memory that will be mapped into a block. It is like a unique identifier for that group of data. As the length of the address in bits is m-bits, then the number of tag bits is  $t = m - b - s$ . See Figure 13.



**Figure 13: Divisions of the address for cache use**

Figure 13 also shows a memory address for a direct mapping or a n-way associative cache memory. A full associative cache memory is shown at Figure 14. Due to a cache block can go anywhere in the cache there is no need for index field in the memory address.



**Figure 14: Fully associative Cache Memory**

## 6.4 Cache Size

The capacity of a cache represents the amount of data that can be stored in the cache. For example, a cache with capacity 64KB can store 64kilobytes of data. In this study, the AVF was computed for three different data cache capacities, 64KB, 32KB and 16KB.

## 7. Implementation of AVF analysis assessment in Physical Integer Register File and L1 Data Cache

In this study vulnerability factors computed at the physical integer register file and L1 data Cache. In order to export the final results for that two modules all the ACE bits in the structure, should be recorded. The overall ACE bits consist of ACE bits in physical integer register file and ACE bits in L1 data cache, as well. I have extended the code of Gem5 by adding new fields and methods at some existing classes or by implementing new classes. AVF computation for physical integer register file will be analyzed thoroughly in Section 7.3 and about L1 data cache at Section 7.4.

### 7.1 Start and End Tick of the Simulation

For 10 benchmarks executed in Gem5, we created checkpoints for every workload. Each benchmark restore a checkpoint which is in the folder *gem5/m5out*. Because of the restore mode, the simulation does not start at Tick 0 but at the Tick which the checkpoint has taken. For that reason the initial tick of simulation has to be noted for the computation of the program duration. At the physical integer register file *simulate.hh* as long as *simulate.cc* a new variable, the variable «*Tick initial\_time*» is defined for that scope: `Tick initial_time= CurTick ();`. `CurTick ()` function returns the tick time at that moment of execution.

The end simulation Tick is taken at the file *sim\_events.cc* with the same way, using function `CurTick()` in the function «`exitSimLoop()`». The subtraction of end and initial Tick gives the program duration that is going to be used in AVF equations.

### 7.2 Dynamic Instruction

The dynamic instruction is the instruction that executed and is responsible for modifying the data of a register in the physical integer register file or a word in L1 data cache. It is declared in the header file *dyn\_inst.hh* in the «*class BaseO3DynInst*». In order to know which dynamic instruction is responsible for a change in the data field of a physical integer register or a word of the L1 data cache, three more fields were added at the class of *BaseO3DynInst*. The extra fields are shown in Figure 15.

```
template<class Impl>
class BaseO3DynInst: public BaseDynInst<Impl> {
public:
    class packet *packet1;
    vector<vector<packet>> > read_packet;
    vector<vector<InfoForCacheAccess>> > packetInfoCacheAccess;
```

Figure 15: Class BaseO3DynInst

A load dynamic instruction in the physical integer register file can access the value from one or more register at the same time. Even in data cache a load instruction can access more than one word simultaneously. So, when a load instruction is committed all the entries in vectors *Entry* (analyzed in more detail at the next Sections) that are related with that instruction updated according to that commission. *Entry* is a structure that holds information about the time that happens a change in the data field of a physical integer register or a word of the L1 data cache. The above vectors: *read\_packet* and *packetInfoCacheAccess* hold information about the registers and the words of L1 data cache which are depend on this instruction. As a result two new classes have been added. The «class *packet*» for the physical integer register file and the «class *InfoForCacheAccess*» for the L1 data cache.

«Class *packet*» as shown in Figure 16, has information about the index of the integer register and details about the exact row and column that is in the vector *Entry*. The value of *row\_of\_Entry* indicates the register ID, while the *column\_of\_Entry* indicates the load instruction that accessed that entry. All the *packets*' fields are accessed by senders and receivers.

```
class packet{
private:
    int row_of_Entry;
    int column_of_Entry;
    int reg_idx_of_Entry;
```

Figure 16: Class packet in DynInstPtr

Respectively, «class *InfoForCacheAccess*» that is presented in Figure 17, has information about the related set, block and word of L1 data cache. Also has fields about the exact row and column that is related in the vector *Entry*. The value of *row\_of\_Entry* indicates the word, while *column\_of\_Entry* indicates the load that

accessed that word. Moreover, the field of *NumSet* indicates the set, *NumBlock* refers to the block and *NumWord* indicates the word in the block. All *InfoForCacheAccess*'s fields are accessed by senders and receivers. This class has meaning only for load instruction, because write requests in L1 data cache indicate that the store instruction was committed.

```
class InfoForCacheAccess{
private:
    int row_of_Entry;
    int column_of_Entry;
    int NumSet;
    int NumBlock;
    int NumWord;
```

Figure 17: Class InfoForCacheAccess

### 7.3 AVF for Physical Integer Register File

As it is illustrated at Figure 11, Gem5 consists of several folders, each folder has its own functionality. The functionality that I am interested in is inside the folder *src* and especially the subfolder of *src/cpu*. Inside *src/cpu* there are subfolders for each model of CPU. This study will implement the classes and methods from O3CPU model (out-of-order model), so the source code files from *src/cpu/o3* will be edited. From now on, we will not refer to the accurate path in Gem5 for every header or source file because we present details about source and header files of the simulator in the last Section.

This study targets to have access and to note the accurate time that a dynamic instruction will read or write the data of a register. The methods for reading and writing at a register is implemented at header file *regfile.hh*. The functions with the specific functionality in that module are «*readIntReg()*» and «*setIntReg()*». The source code for those functions has moved to the source file *regfile.cc*, as *regfile.hh* is imported to many files in Gem5. For saving the exact Tick that becomes a read or a write at a register we created a structure that holds information about the time of reading or writing process and also information about the commission or no of the specific dynamic instruction.

### 7.3.1 Structure for holding information about a read/write at a register

A new header file *pinakas.h* is created in order to set a new class, «class element». «Class element» holds information about the Tick that an instruction accesses a register with the variable «*Tick time*» as long as a boolean variable «*Bool isSquashed*» which becomes true if the instruction that accessed the register was finally committed. All the *elements*' fields are accessed by senders and receivers. The constructor initializes time to *CurTick()* Tick and the boolean variable as false, therefore all instruction at first steps considered as not committed and so on bits are considered un-ACE bits. Below in Figure 18 we present the fields of «Class element».

```
class element{
private:
    Tick time;
    bool isSquashed;
```

Figure 18: Class element

In order to compute ACE bits during the execution of the benchmark, statistics for every register of the simulator need to be hold. In this study we used three different configurations of physical integer register file. The first one is for 256 registers, the other for 128 registers and the last one for 64 registers. The configuration about the number of physical registers is defined in a python file, *O3CPU.py*. In all of these cases we did not take under consideration the register 16 because it is implemented in a non-realistic functional way representing the zero register. In our analysis, we stored all the information needed concerning the registers in vectors. This vector is declared in the same file (*pinakas.h*) with the class «*class regi*», see Figure 19.

```
class regi{
public:
    vector<vector<element> > Entry;
```

Figure 19: Class regi

Each cell of the vector *Entry* is a «class element» object. The vector *Entry* is declared in the «class *PhysRegFile*» at the header file *regfile.hh* and is initialized with the constructor of «class *PhysRegFile*» in source file *regfile.cc*. «Class *PhysRegFile*» constructs a physical integer register file with integer and floating point registers and

also methods about handling registers. The declaration and initialization of «class *regi*» is presented in Figure 20. The variable `_numPhysicalIntRegs` is the number of physical registers of the simulator. In this study it is equal to 256, 128 or 64.

```
class regi* Level1;

Level1=new regi[_numPhysicalIntRegs];
```

Figure 20: Declare and initialize "class regi"

At this point it is important to explain the functionality of vector *Entry* and how a set operand of a register distinguishes from a read operand. A detailed diagram presents in Figure 21.

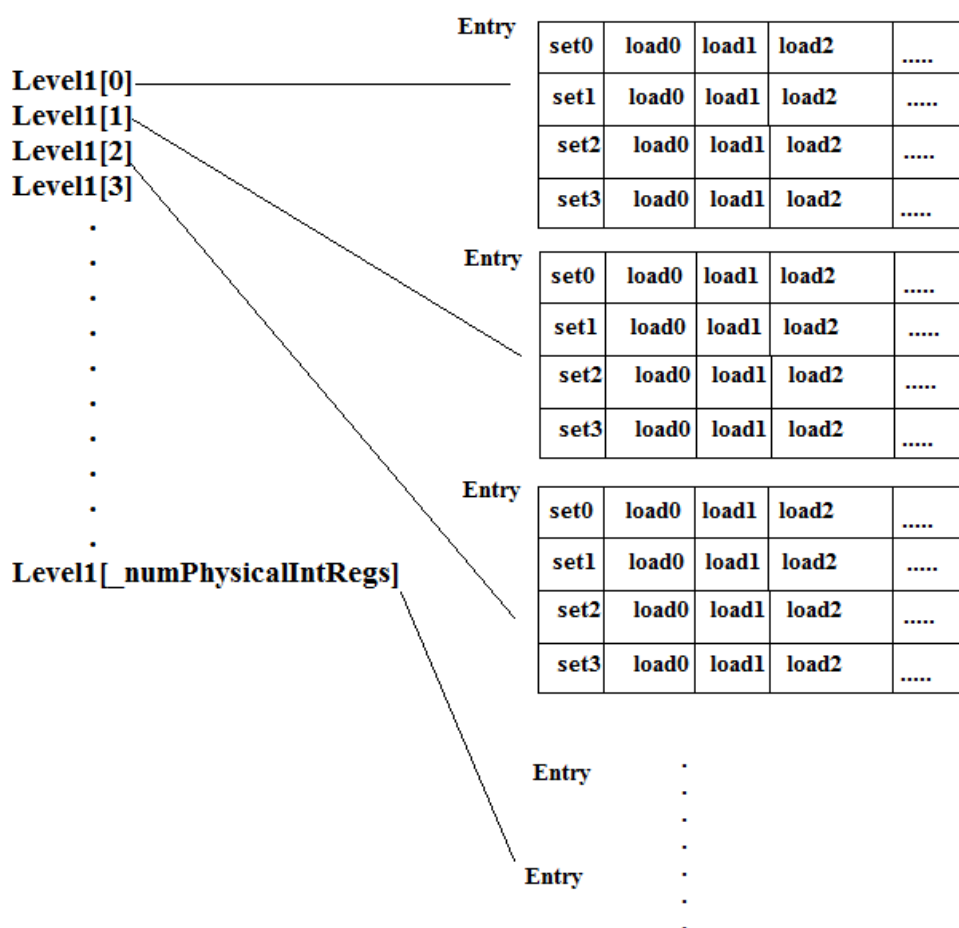


Figure 21: Vector Entry for Physical Integer Register File



The first column at the *Entry* represents the write accesses for a specific register whereas the other columns represent the read operation for the specific register. Thus, the first column of each row represents different write operations in the register. Every other column at the specific row is a read operation of the value which is written according to the write operation of the first column. For example, (Figure 21) every register has 4 different sets “set0-set3”. “Load0-Loadn” correspond to n different loads of the register. There are n loads for the 1<sup>st</sup> set, n loads for the 2<sup>nd</sup> set and so on.

### 7.3.2 Methods for setting and loading a value from a register index

In «*Class PhysRegFile*» there are the functions, «*setIntReg()*» and «*readIntReg()*» for setting a value in an integer register and the other for loading a value from it. The input parameters in both functions are different, see Figure 22. One more argument has been added, the argument of «*DynInstPtr inst*», «*class DynInstPtr*» has been explained in Section 7.2.

```
uint64_t readIntReg(PhysRegIndex reg_idx, const StaticInst *si, DynInstPtr instr);
void setIntReg(PhysRegIndex reg_idx, uint64_t val, const StaticInst *si, DynInstPtr inst);
```

Figure 22: Functions readIntReg() and setIntReg()

Inside the body of that methods, we added functionality in order to fill the vector *Entry* for the register with specific id, «*PhysRegIndex reg\_idx*». *Time* field is an *element* object that takes its value from curTick (). boolean variable *isSquashed* assigns with false (all bits before the commit stage assume as un-ACE bits). In that functions one more field of «*class DynInstPtr*» has to be assigned, that stores the details concerning the dynamic instructions that accesses the hardware entry.

«*class packet*» saves details about the index of the integer physical register, the number of row and column in the vector *Entry* where is located the element related to this access of the register. The information of «*class packet*» will be used later at commit stage in order to inform all the cells of the vector that are related to the committed instruction.

### 7.3.3 Instruction committed for an integer register in Physical Integer Register File

At first steps all bits of the hardware structures are assumed as un-ACE bits. Each *element* object initializes its boolean variable about the commit as false. When an instruction arrives at commit stage, the function «*void commitInsts()*» in *commit\_impl.hh* is called. The instruction that arrives at that stage is certain that is committed, so the bits that are related to that dynamic instruction are ACE bits. At this point the boolean variable in vector *Entry* has to be set as true. The extension of «*class DynInstPtr*» with the extra fields, see Section 7.2 help us to have information about the exact register this instruction is referred. If it is a load instruction it may refers to more than one registers. As a result all the related cells in *Entry* vectors set their boolean variable at the commit stage. Thus, this interval considered as ACE during the AVF equation.

#### 7.3.3.1 Pointer to Physical Integer Register File from commit stage

Access to CPU object is available only at the functions inside *src/cpu*, while the structures out of the core has no knowledge about CPU elements. The vector *Entry* that is used in order to hold statistics about physical integer register file has defined and takes value inside CPU, so it is not accessible out of CPU. In order to have access in the several structures of CPU a pointer has defined to a CPU object. The pointer «*regi\*myRegfileForModify*» is declared in *sim\_events.cc* and extended in other source files. The initialization becomes in «*void commitInsts ()*», see Figure 23.

```
extern regi* myRegfileForModify;

PhysRegFile * myregfile=cpu->get_reg_file();
regi* reg1=myregfile->Level1;
myRegfileForModify=reg1;
```

Figure 23: Pointer to Physical Integer Register File

### 7.3.4 The computation of AVF

Inside folder *src/sim* there is the source file *sim\_events.cc*. At this file, the simulator handles several termination event. When the benchmark comes to the end or when the user terminates the execution (using Ctrl-X, Ctrl-C), the function from Figure 24 called. At this function, the simulation ends up so the end simulation Tick that will be used for calculation program's duration is available.

```
void
exitSimLoop(const std::string &message, int exit_code, Tick when, Tick repeat,
            bool serialize)
```

**Figure 24: Function about simulator termination**

In this implementation in order to compute the overall ACE-bits of the register, all the entries of the vector Entry have to be checked. For each entry that is not committed according to the boolean variable *isSquashed*, *Time* is set equal to 0 (zero). If the instruction was committed, the period between the write and the load is considered as ACE period. Otherwise if the last load is not committed, continue with the previous load and so on. It is important to be noted that if the write operation of register is not committed then there is no ACE period for that interval.

This process is repeated for every register in the physical integer register file except *zeroReg*. At the end, ACE interval Ticks for all the registers in the register file are available. The general equation type for AVF calculation that was presented in Section 3.4.2 was used. The next equation was redefined in order to be more specified in this study:

$$AVF = \frac{ACE\ Interval\ Ticks}{Program\ duration\ in\ Ticks \times Number\ of\ physical\ integer\ registers}$$

ACE Interval Ticks are the overall ACE Ticks for the physical integer register file. Program duration is the duration in Ticks for the simulation and the number of Physical registers is the number of physical integer registers in the physical integer register file.

## 7.4 AVF for Data Cache

The source files concerning memory is in the folder *src/mem* in Gem5. At this folder there is source code about general handling of the memory. Specifically for cache memory that this study is talking about, all its files are inside the subfolder *src/mem/cache*. There is one more subfolder inside, *src/mem/cache/tag* that contains all the source and header files with information about cache characteristics and its configuration.

All the information about sets and blocks is taken from the file *base\_set\_assoc.cc*. Inside the *tags* subfolder there are all the details about the tags of the cache. Tags

contain configuration details like the block size, the number of blocks etc. A tag object passes as argument to the majority of functions in order to find if a particular word exists in the cache or not.

The L1 data cache consists of sets, each set depending on cache associativity consists of  $n$  blocks ( $n$ -way associative). Each block depends on the address size in bits (64 bit address) and the block size consists of words. In this study, the first level data cache consists of blocks of 64B size, with 8 words of 64 bit each. The size of the cache memory in combination with the way of associativity also determine the number of sets in cache.

$$\text{Number of sets} = \frac{\text{cache size}}{\text{number of words in a block} \times n - \text{way associativity}}$$

In this study all the benchmarks will be executed with several cache sizes: 16KB, 32KB and 64KB. The configuration about the cache size is defined from the running script that starts the simulation (using the option `-l1d cache=...`) and the configuration about the way associativity is set at a python file (Tags.py) in the class «class BaseSetAssoc(BaseTags)». Furthermore, the way of associativity in all our experiments is 4-way associativity meaning that every set consists of 4 blocks.

#### 7.4.1 Structure for holding information about a read/write at a word in a block

Similar to the physical integer register file, we store information at a vector concerning details about the time when an instruction accesses a hardware entry and if this instruction is finally commit. The goal is to have access and to note the accurate time that an instruction reads or writes a word in L1 data cache. The access in L1 data cache is implemented in the file *cache\_impl.hh*. Four writing accesses and four reading accesses at L1 data cache have been identified. The exact points in the source code will be presented later.

We created a structure that holds information about the moment of a read or a write operation in a word of L1 data cache about the commission or not of the specific dynamic instruction that accessed the entry. Except from that information that is similar to that of physical integer register file, the L1 data cache needs one more variable that represents if the load operation of the entry concerns a writeback or not. This difference exists due to the writeback of some dirty blocks from the first level data cache to the lower layers of the cache hierarchy. As a result our implementation cannot correlate a

committed or not instruction with a written back block. For that reason in our implementation we used a parameter (p) that defines the percentage of the intervals that were written back and are considered as ACE. Thus, if p=100%, then all the intervals that were written back are ACE. If p=50%, only the half of the intervals that were written back are considered as ACE. Finally, if p=0%, all the intervals that were written back are considered un-ACE.

At the path *src/mem/cache* a new header file *pinakas2.h* has been created in order to set a new class «class *elementForCache*». This class holds information about the time (in Ticks) that an instruction accesses a word in the data cache with the variable *Tick time*. Moreover, a boolean variable *Bool blockValidation* is used and is set when the instruction that accessed the word is committed. The field *Bool iswritebackload* determines if a block was written back or not. All the *elementForCache*'s fields are accessed by senders and receivers. The constructor initializes time using *CurTick()* function and the boolean field *blockValidation* is unset (false), therefore all instruction at first steps considered as not committed and consequently bits are considered as un-ACE. The field *iswritebackload* is false in case of the block is not written back, otherwise is true. This class is presented at Figure 25.

```
class elementForCache{
private:
    Tick time;
    bool blockValidation;
    bool iswritebackload;
```

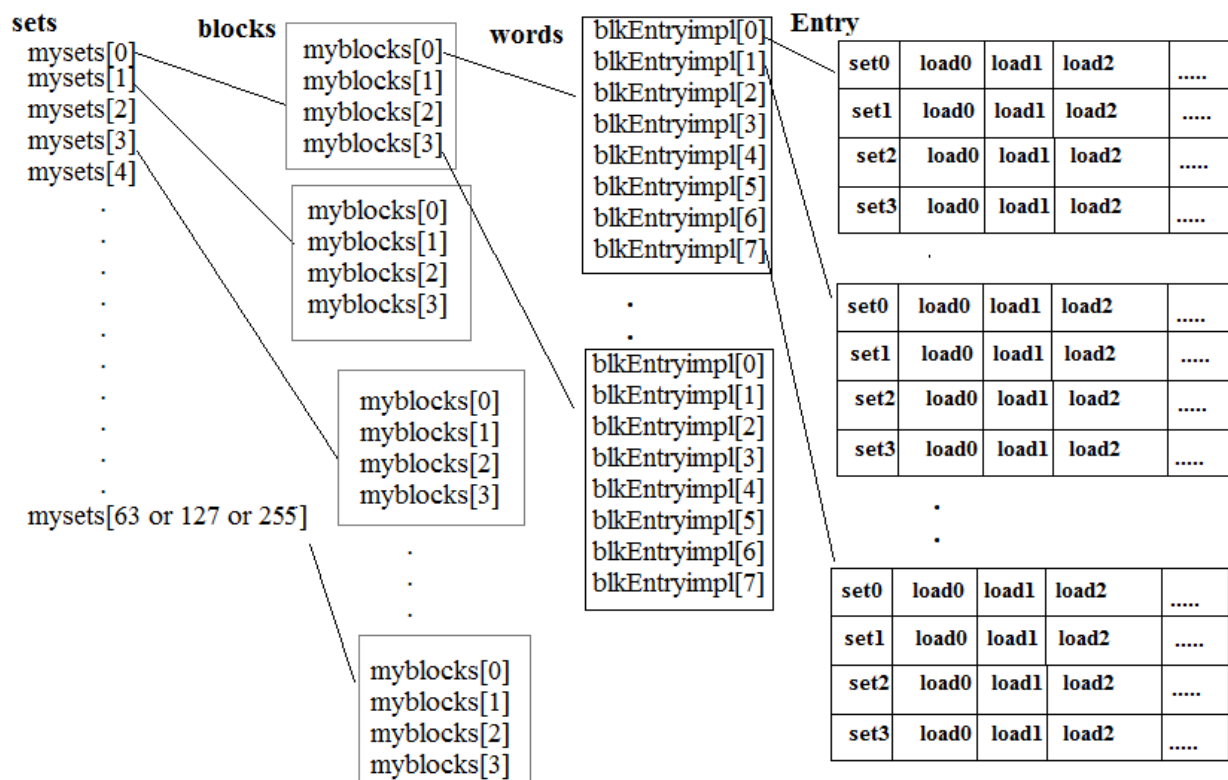
**Figure 25: Class *elementForCache***

There are three possible configurations according the cache size. The first one is for 16KB cache size, the second for 32KB cache size and the last for 64KB cache size. The cache size determines the numbers of sets in the cache as long as the number of blocks. In order to compute ACE bits for data cache during the execution of the benchmark we need to hold statistics for every word of the data cache in a vector. This vector is implemented in the same file (*pinakas2.h*) with the class «class *cachesImplementation* », see Figure 26. Every word in a block has its own vector *Entry*.

```
class cachesImplementation{
public:
    vector<vector<elementForCache> > Entry;
```

**Figure 26: Class CachesImplementation**

The first column at each row of the vector *Entry* reflects the sets of the word and the other columns correspond to the loads of this word's set. At Figure 27 except from vector *Entry* three more modules are shown. They have been constructed in order to express the cache's division in sets, blocks and words.



**Figure 27: The architecture of sets, blocks and words in L1 data cache.**

#### 7.4.1.1 The structure used to store information of sets, blocks and words

Figure 27 makes clear the idea of the model for expressing sets, blocks and words. The declaration for *mysets* becomes in the «*class cache*», for *myblocks* inside «*class CacheBlkpointer*» and for *blkEntryimpl* in «*class Cacheblk*».

In «*class cache*» apart from the variable *mysets* that is used to express the set of the L1 data cache an extra function has been added (*Cache\* get\_cache()*) to access the pointer of the L1 data cache. This pointer is used to *sim\_Events.cc* and *commit\_impl.hh* files where we need to access the structures of data cache for

computing the ACE periods and mark instruction as committed. The extra fields of «*class cache*» are illustrated at Figure 28.

```
class Cache : public BaseCache
{
public:
    class CacheBlkpointer *mysets;
    Cache *get_cache() {
        return this;
    }
}
```

Figure 28: Class cache

The initialization for the field *mysets* is in the file *cache\_impl.hh*.

```
mysets=new CacheBlkpointer[numberofSets];
```

Information concerning the blocks at each set is saved at the «*class CacheBlkpointer*». *CacheBlkpointer* is a new class constructed to identify the block an n-way associative cache memory (each set has n blocks). The declaration is in the header file *cache.hh* (Figure 29).

```
class CacheBlkpointer{
public:
    class CacheBlk* myblocks;
    CacheBlkpointer() {
        myblocks=new CacheBlk[thewayAssociation];
    }
    ~CacheBlkpointer() {
        delete [] myblocks;
    }
};
```

Figure 29: Class CacheBlkpointer

The last information about the words in data cache is also important. In «*class CacheBlk*» in header file *blk.hh* there is the declaration for the object *blkEntryimpl*, (Figure 30). *BlkEntryimpl* field is an object of the «*class cachesImplementation*», the class with the vector *Entry* that holds statistics for each word in a block.

```
class CacheBlk
{
    public:
    class cachesImplementation* blkEntryimpl;
```

Figure 30: Class CacheBlk

The initialization happens in the constructor of *CacheBlk* at the same file *blk.hh*, see Figure 31.

```
CacheBlk()
: task_id(ContextSwitchTaskId::Unknown),
  asid(-1), tag(0), data(0), size(0), status(0), whenReady(0),
  set(-1), way(-1), isTouched(false), refCount(0),
  srcMasterId(Request::invldMasterId),
  tickInserted(0)
{
    blkEntryimpl=new cachesImplementation[numberofblocksize/8];
}
```

Figure 31: Constructor of class CacheBlk()

The size of *blkEntryimpl* depends on the address size in bits. In this study the address is 64 bit, so every word is 8 bytes. Consequently *blocksize/8* gives the exact width of words in a block. Concluding all the previous classes and their functionality if it we want to access the the 3<sup>rd</sup> word of the 4<sup>th</sup> block at 134 set in data cache it should be written: *mysets[133].myblocks[3].blkEntryimpl[2]*.

#### 7.4.2 Methods for setting and loading a word in L1 data cache

The header file *commit\_impl.hh* monitors the set and load operands of the words of the L1 data cache. Four different functions are responsible for loading the data value form a word in the L1 data cache, in two of them also a set operation takes place. Two other functions are responsible for the write operation. Below is the prototype of that functions with a sample of theirs source code.



- `Void Cache::cmpAndSwap(CacheBlk *, PacketPtr )`: This function handles the comparison and swap for SPARC (a Scalable Processor Architecture) <sup>2</sup>. This function monitors read as long as set operation in the L1 data cache. In this study the benchmarks that are executed do not call this function. The exact line at the source code that a load operand took place is shown at Figure 32.

```
void
Cache::cmpAndSwap(CacheBlk *blk, PacketPtr pkt)
{
    assert(pkt->isRequest());
    uint64_t overwrite_val;
    bool overwrite_mem;
    uint64_t condition_val64;
    uint32_t condition_val32;
    int offset = tags->extractBlkOffset(pkt->getAddr());
    uint8_t *blk_data = blk->data + offset;
    assert(sizeof(uint64_t) >= pkt->getSize());
    overwrite_mem = true;
    -----READ-----
    pkt->writeData((uint8_t *)&overwrite_val);
    pkt->setData(blk_data);

    if (pkt->req->isCondSwap()) {
        if (pkt->getSize() == sizeof(uint64_t)) {
            condition_val64 = pkt->req->getExtraData();
            overwrite_mem = !std::memcmp(&condition_val64, blk_data,
                                         sizeof(uint64_t));
        } else if (pkt->getSize() == sizeof(uint32_t)) {
            condition_val32 = (uint32_t)pkt->req->getExtraData();
            .
            .
            .
        }
    }

    if (overwrite_mem) {
        std::memcpy(blk_data, &overwrite_val, pkt->getSize());
        blk->status |= BlkDirty;
    }
    -----WRITE-----
}
```

Figure 32: Function `cmpAndSwap()`

<sup>2</sup> SPARC stands for a Scalable Processor Architecture. SPARC has been implemented in processors used in a range of computers from laptops to supercomputers [33]

- `Void Cache::satisfyCpuSideRequest(PacketPtr , CacheBlk* , bool , bool ):`

This function represents cache read and write from the CPU. It signs a request for loading data using a packet from a word in the L1 data cache or write data in a word using a packet (Figure 33).

```
void
Cache::satisfyCpuSideRequest(PacketPtr pkt, CacheBlk *blk,
    bool deferred_response, bool pending_downgrade)
{
    assert(pkt->isRequest());
    assert(blk && blk->isValid());
    if (pkt->cmd == MemCmd::SwapReq) {
        cmpAndSwap(blk, pkt);
    } else if (pkt->isWrite() &&
        (!pkt->isWriteInvalidate() || isTopLevel)) {
        assert(blk->isWritable());
        if (blk->checkWrite(pkt)) {
            pkt->writeDataToBlock(blk->data, blkSize);
        }
        blk->status |= BlkDirty;
        ----WRITE----
    } else if (pkt->isRead()) {
        if (pkt->isLLSC()) {
            blk->trackLoadLocked(pkt);
        }
        ----READ----
    }
}
```

Figure 33: Function `satisfyCpuSideRequest()`

- PacketPtr Cache::writebackBlk(CacheBlk \*):

This function creates a writeback request for a given block. In this study writeback blocks are assumed as totally committed, or totally squashed or partially committed. The source code for that function is on next Figure 34.

```
PacketPtr
Cache::writebackBlk(CacheBlk *blk)
{
    assert(blk && blk->isValid() && blk->isDirty());
    writebacks[Request::wbMasterId]++;
    Request *writebackReq =
        new Request(tags->regenerateBlkAddr(blk->tag, blk->set), blkSize, 0,
                    Request::wbMasterId);
    if (blk->isSecure())
        writebackReq->setFlags(Request::SECURE);
    writebackReq->taskId(blk->task_id);
    blk->task_id = ContextSwitchTaskId::Unknown;
    blk->tickInserted = curTick();
    PacketPtr writeback = new Packet(writebackReq, MemCmd::Writeback);
    if (blk->isWritable()) {
        writeback->setSupplyExclusive();
    }
    writeback->allocate();
    ----READ----
    std::memcpy(writeback->getPtr<uint8_t>(), blk->data, blkSize);

    blk->status &= ~BlkDirty;
    return writeback;
}
```

Figure 34: Function writebackBlk()

- `Void Cache::handleSnoop(PacketPtr, CacheBlk *, bool, bool ):`

This function is executed a very few times and as a result in this implementation each load from that function is assumed as committed (the influence of the final result due to these cases is negligible). It sets the cache block that have being snooped to a new coherence state for that block. A sample from the source code of this function is in Figure 35.

```
void
Cache::handleSnoop(PacketPtr pkt, CacheBlk *blk, bool is_timing,
    bool is_deferred, bool pending_inval)
{
    .
    .
    if (forwardSnoops) {
        .
        .
    }
    .
    .
    .
    if (pkt->cmd == MemCmd::HardPFRReq) {

        pkt->setBlockCached();
        return;
    }
    if (!pkt->req->isUncacheable() && pkt->isRead() && !invalidate) {
        assert(!needs_exclusive);
        pkt->assertShared();
        int bits_to_clear = BlkWritable;
        const bool haveOwnershipState = true; // for now
        if (!haveOwnershipState) {
            bits_to_clear |= BlkDirty;
        }
        blk->status &= ~bits_to_clear;
    }
    if (respond) {
        pkt->assertMemInhibit();
        if (have_exclusive) {
            pkt->setSupplyExclusive();
        }
        ----READ----
```

Figure 35: Function `handleSnoop()`

- Bool Cache::access(PacketPtr, CacheBlk \*, Cycles &, PacketList &):

This function monitors a set operation in the L1 data cache, see Figure 36 for the source code.

```
bool
Cache::access(PacketPtr pkt, CacheBlk *&blk, Cycles &lat,
              PacketList &writebacks)
{
    .
    .
    .
    if (pkt->cmd == MemCmd::Writeback) {
        assert(blkSize == pkt->getSize());
        .
        .
        .
    }
    assert(!pkt->needsResponse());
    std::memcpy(blk->data, pkt->getConstPtr<uint8_t>(), blkSize);
    incHitCount(pkt);
    ----WRITE----
    return true;
} else if{
    .
    .
    .
}
if (blk == NULL && pkt->isLLSC() && pkt->isWrite()) {
    pkt->req->setExtraData(0);
    return true;
}
return false;
}
```

Figure 36: Function access()

- CacheBlk\* Cache::handleFill(PacketPtr, CacheBlk \*, PacketList &):

This function is responsible for cache filling. More specifically this function populates a cache block and handles all outstanding requests for this fill request (Figure 37).

```
CacheBlk*
Cache::handleFill(PacketPtr pkt, CacheBlk *blk, PacketList &writebacks)
{
    assert(pkt->isResponse() || pkt->isWriteInvalidate());
    Addr addr = pkt->getAddr();
    bool is_secure = pkt->isSecure();
    .
    .
    .
    if (blk == NULL) {
        assert(pkt->hasData());
        assert(pkt->isRead() || pkt->isWriteInvalidate());
        blk = allocateBlock(addr, is_secure, writebacks);
        .
        .
        .
    } else {
        assert(blk->tag == tags->extractTag(addr));
        assert(pkt->hasData() || blk->isValid());
    }
    .
    .

    if (pkt->isRead()) {
        assert(pkt->hasData());
        assert(pkt->getSize() == blkSize);
        std::memcpy(blk->data, pkt->getConstPtr<uint8_t>(), blkSize);
        ----WRITE----
    }

    blk->whenReady = clockEdge() + fillLatency * clockPeriod() +
        pkt->payloadDelay;

    return blk;
}
```

Figure 37: Function handleFill()

### 7.4.3 Packets transfer data to dynamic instruction Object

According to the source code for cache memory, in the header file *cache\_impl.hh* that handles the access at data cache's words there is no way to access the dynamic instruction. Nevertheless, in this header file there are the methods for writing and loading a word in the L1 data cache and the assignment to the field *packetInfoCacheAccess* has to be done in that point. This was done using another class object, the «*class packet*». «*class packet*» as shown in Figure 38, is the structure that transfers data from L1 to L2 level cache. Packets are response for each request in order to transfer data in memory hierarchy.

```
class Packet : public Printable
{
public:
    InfoForCacheAccess * infoforcacheaccess;
```

Figure 38: Class Packet

At «*class packet*» a new field has been added. The field of *infoforcacheaccess* that is an object of «*class InfoForCacheAccess*». This class has analyzed in Section 7.2 where the class of *DyInstPtr* was explored. The field *infoforcacheaccess* used to hold information about the access (set or load operation) in a word of L1 data cache. «*class packet*» is available in the header file *cache\_impl.hh* so it can transfer details that are related with a specific dynamic instruction in the header file *lsq\_unit\_impl.hh* where the object of dynamic instruction is accessible. At the function «*completeDataAccess()*» in *lsq\_unit\_impl.hh* the index of the element *infoforcacheaccess* of «*class packet*» is copied to the element *infoforcacheaccess* of «*class DyInstPtr*». Consequently, during the commit stage the dynamic instruction can inform the vector *Entry* about which of its cells have to assigned as committed. All dynamic instructions follow the same path: *lsq\_unit.hh* -> *cache\_ipml.hh* -> *lsq\_unit\_impl.hh* except from the load that are correlated with a writeback block that they do not pass from the file *lsq\_unit\_impl.hh*. In conclusion, the *packet* structure is used to pass important information to dynamic instruction as there is no access to that object inside CPU core.

#### 7.4.4 Instruction related with a word in L1 data cache commits

Like the physical integer register file (Section 7.3.3), an instruction is checked as squashed or not in the function «*commitInsts ()*» in *commit\_impl.hh* . At this point we have to identify all instructions of data cache that were committed. The access in the *Entry* vector happens via a pointer to the cache object which has initialized in the header file *cache.hh*.

```
Cache * mycopycache;
```

Using this pointer (*mycopycache*), we can correlate a specific element of the vector *Entry* with a specific dynamic committed instruction (*packetInfoCacheAccess*). Finally if the dynamic instruction was committed then the boolean variable “blockValidation” is set as presented at Figure 39. Otherwise, it remains as unset.

```
mycopycahe->mysets[head_inst->packetInfoCacheAccess[...][...].getSet()]
.myblocks[head_inst->packetInfoCacheAccess[...][...].getBlock()]
.blkEntryimpl[head_inst->packetInfoCacheAccess[...][...].getWord()]
.Entry[head_inst->packetInfoCacheAccess[...][...].getrow()][head_inst->packetInfoCacheAccess[...][...].getcolumn()].setblockValidation(true);
```

**Figure 39: Dynamic Instruction committed in L1 data cache**

#### 7.4.5 The Computation of AVF

The function «*exitSimLoop()*» according to Section 7.1 signs the end of the simulation. The overall ACE bits for the physical integer register file as long as for the L1 data cache are computed at this function. The process for the two modules is similar. It should be noticed that every set instruction in the L1 data cache is always committed so there is no need to check if the instruction commits or not in contrast to the physical integer register file.

At the end for each block of the cache, the total ACE interval is computed. Three different cases of AVF have to be computed so different total ACE Interval period is used for each equation. The general equation type for AVF in the data cache is:

$$AVF = \frac{ACE\ Interval\ Ticks}{Program\ duration\ in\ Ticks \times Number\ of\ words\ in\ L1\ Dcache}$$

Where ACE Interval Ticks are the overall ACE Ticks for the L1 data cache, Program duration is the duration in Ticks for the simulation. The number of words in L1



data cache is computed multiplying the number of sets, the number of ways and the number of words per block. Ace Interval Ticks are not the same at the three different cases due to the writeback blocks.

The first case assumes that all loads correlated to writeback blocks in the L1 data cache are committed (parameter  $p=100\%$ ), ACE Interval Ticks are computed with the same way like physical integer register file. In the loop for every set, way and finally every word we check if the last load instruction of the word is committed by checking the boolean variable *blockValidation*. If it is committed then the period between the set and the load is assumed as ACE. Otherwise if the last load is not committed, we continue with the previous load and so on.

The second case assumes that all loads correlated to writeback blocks in the data cache are not committed (parameter  $p=0\%$ ), we execute the same loop as previously mentioned with the only difference that if the last load of the word is correlated with a writeback block, then it is ignored and we continue to the next loads. The variable *iswritebackload* is used for that operation.

In the third case where we assume that partially some loads correlated to writeback blocks are ACE (parameter  $p=50\%$ ), we use a factor that defines the percentage of these loads that are finally assumed as committed.

## 8. Results

In our study we evaluated 10 benchmarks: cjpeg, djpeg, fft, qsort, sha, stringsearch, susan\_corners, susan\_edges, susan\_smoothing and rijndael\_enc from MiBench suite, using the small data sets. In Table 4 we present the exact number of committed instructions per benchmark [28]. Specifically according to the stringsearch benchmark, we run many small versions of it and also a lot of big versions from the benchmarks of fft and qsort.

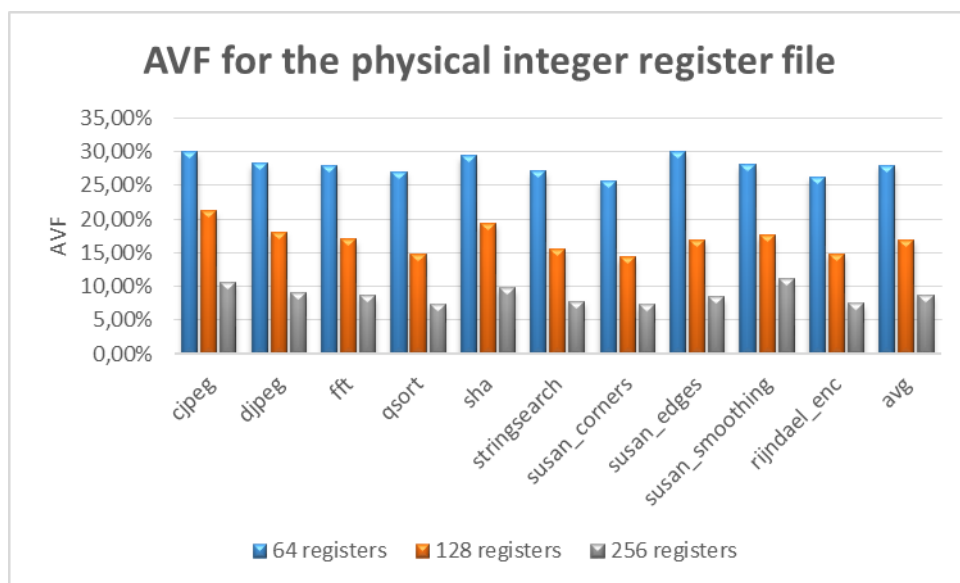
**Table 4: Benchmarks and Instruction count**

<b>Benchmark</b>	<b>Committed Instructions</b>
<b>Cjpeg</b>	28,108,471
<b>Djpeg</b>	6,677,595
<b>Fft</b>	52,625,918
<b>Qsort</b>	43,604,903
<b>Sha</b>	13,541,298
<b>Stringsearch</b>	158,646
<b>susan_corners</b>	1,062,891
<b>susan_edges</b>	1,836,965
<b>susan_smoothing</b>	24,897,492
<b>rijndael_enc</b>	28,108,471

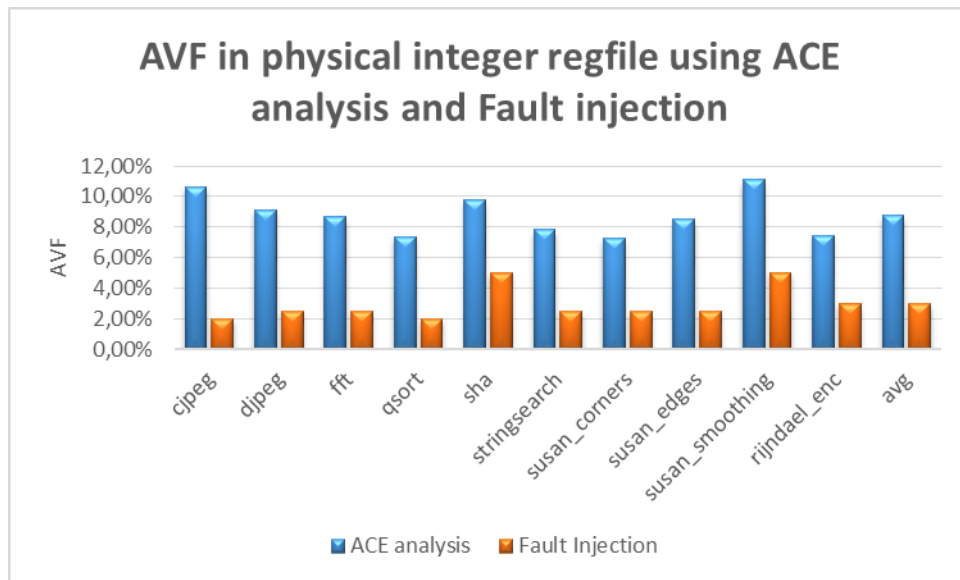
Each benchmark is executed four times. In this study we ran every benchmark for three different sizes of physical integer register file (256, 128 and 64 registers) as long as for three different sizes of L1 data cache (64 KB, 32 KB and 16KB). For each configuration there are diagrams that shown the AVF estimation and tables with the

program duration in cycles for every executed benchmark and the ACE cycles during the execution. The rightmost end of each diagram, “avg” is used for the average case of each AVF estimation.

Figure 40 shows the faulty behavior classification for the physical integer register file. For each size of the physical integer register file we assume that L1 data cache size is 32KB (Table 3). As the size of the physical integer register file is increased, the AVF is decreased. This is a logical because when there are more registers available, the workload of its one register is not so big. We observe the same trend at all benchmarks; AVF is higher using 64 registers than 128 and 256 registers respectively. If we compare these results with the results of [1] that runs the same benchmarks, where the AVF was computed with statistical fault-injection, we can observe the overestimation of ACE analysis method. This overestimation is presented in the Figure 41. As it is shown from the diagram the overestimation of AVF computation using ACE analysis is about 2,97 x times in average against the vulnerability computed using fault injection [1], this prediction about the overestimation has been done again in Section 3.4.1.



**Figure 40: AVF for the physical integer register file**



**Figure 41: AVF in physical register file using ACE analysis in comparison with fault injection**

It is remarkable that the program duration of each executed benchmark is notably shorter than the ACE period (Table 5).

**Table 5: Execution and ACE cycles for each benchmark**

		<b>64 Registers</b>	<b>128 Registers</b>	<b>256 Registers</b>
<b>cjpeg</b>	Execution cycles	26109073	24550085	24531382
	ACE cycles	50200000000000	66500000000000	669000000
<b>djpeg</b>	Execution cycles	7243352	7234256	7232260
	ACE cycles	130858000	167260000	169096000
<b>fft</b>	Execution cycles	29383619	23296147	23170500
	ACE cycles	525357000	506215000	513736000
<b>qsort</b>	Execution cycles	40156029	38146020	38115535
	ACE cycles	694974000	718016000	718227000
<b>sha</b>	Execution cycles	9671548	9846765	9857959
	ACE cycles	181904000	244828000	245914000
<b>stringsearch</b>	Execution cycles	811506	799344	800968
	ACE cycles	14102700	16000900	16049300
<b>susan_corners</b>	Execution cycles	2228972	2128850	2130583
	ACE cycles	36658900	39248600	39677700
<b>susan_edges</b>	Execution cycles	3626864	3563908	3563791
	ACE cycles	69738500	77107400	77457200
<b>susan_smoothing</b>	Execution cycles	22677032	13120004	12642338
	ACE cycles	407996000	295436000	359820000
<b>rijndael_enc</b>	Execution cycles	24861482	23545044	23310484
	ACE cycles	417967000	447784000	44520900

The next three Figure 42, Figure 43 and Figure 44 show the AVF estimation for the same ten benchmarks but for the L1 data Cache. The size of physical integer register file for the three different executions is 256 registers (Table 3). Each diagram presents the AVF for the three different sizes of L1 data cache: 16KB, 32KB and 64KB. The AVF is decreased while the size of the L1 data cache is increased. As the number of words, sets and blocks in the L1 data cache depend on its size, bigger L1 data cache corresponds to more words that are available for setting and loading value from them. The cache stores more unused data in this case.

The only difference among these three executions is the percentage of the writeback blocks that are correlated with instructions and we assume that are finally committed. The Figure 42 illustrates the case in which we assume that 100% of these written back blocks are vulnerable. At Figure 43 only 50% of the written back blocks are assumed as vulnerable and at Figure 44 all the written back blocks are assumed non-vulnerable. The overestimation of AVF with ACE analysis is also conspicuous in these cases in comparison with the statistical fault injection [1] but that time not as big as in the physical integer register file (Figure 41). In this case the overestimation of AVF using ACE analysis is about 1,23 x times in average against the vulnerability computed using fault injection [1] (Figure 45). On Figure 45, AVF using ACE analysis has estimated for L1 data cache with 100% vulnerable written blocks.

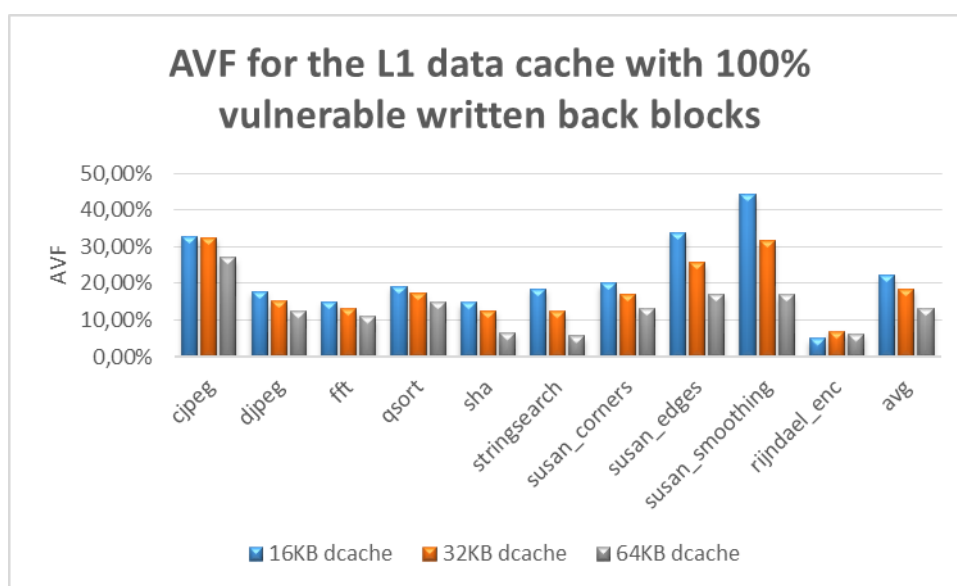
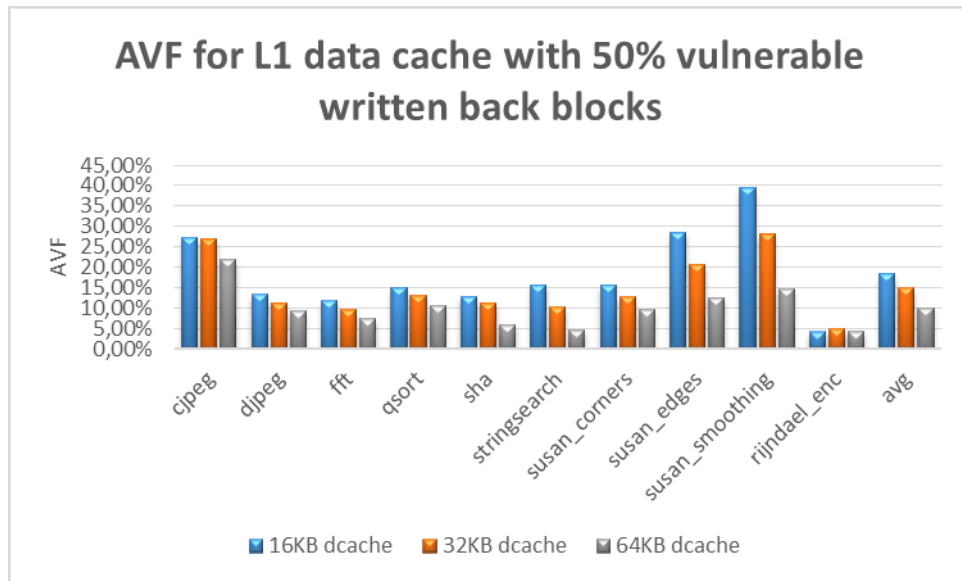
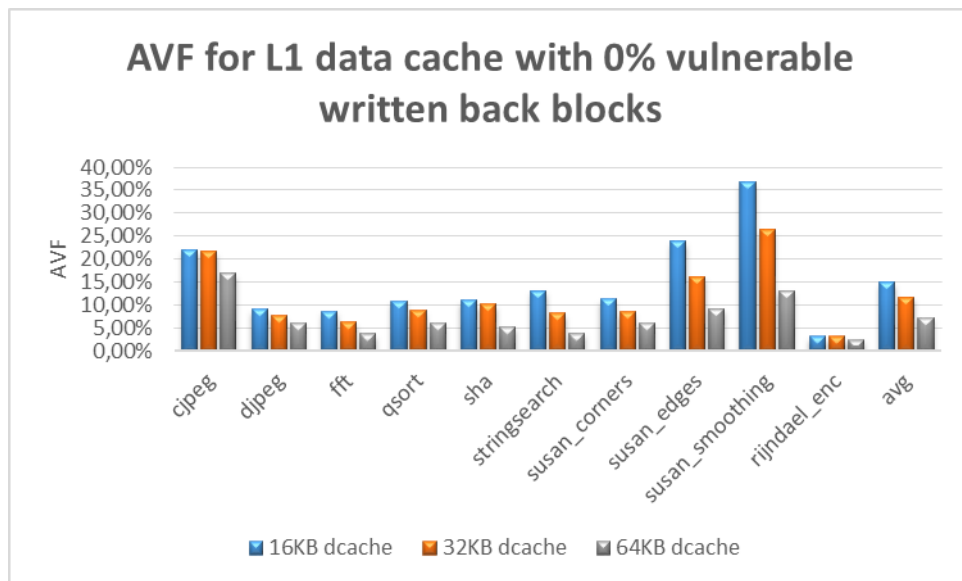


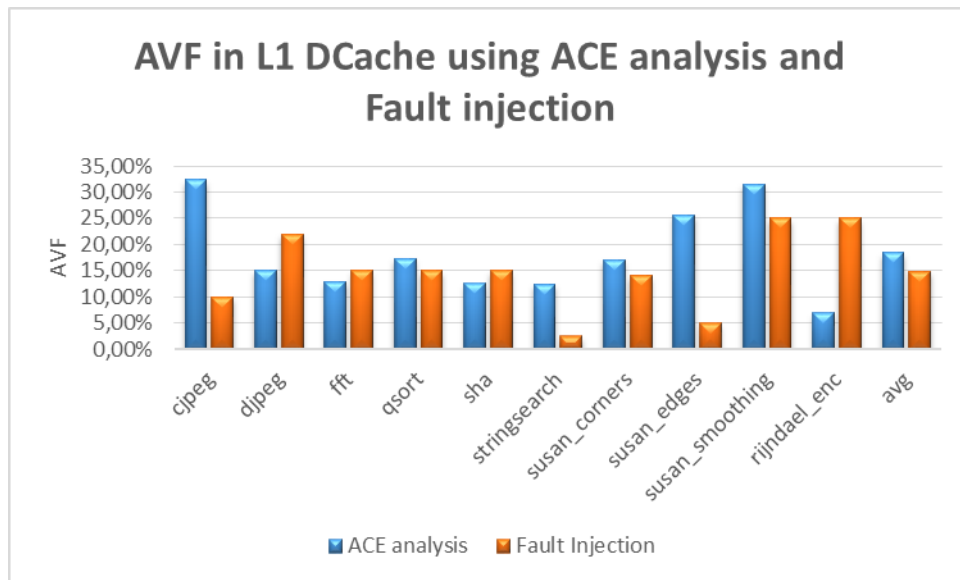
Figure 42: AVF for the L1 data cache with 100% vulnerable written back blocks



**Figure 43: AVF for L1 data cache with 50% vulnerable written back blocks**



**Figure 44: AVF for L1 data cache with 0% vulnerable written back blocks**



**Figure 45: AVF in L1 DCache using ACE analysis in comparison with fault injection**

On the other hand, in the physical integer register file there are not so much differences between the 10 benchmarks in AVF results compared to the L1 data cache. The above Figures verify the initial idea of this thesis at which AVF computation with ACE analysis should result in pessimistic AVF estimation than other methods of AVF computation such as statistical fault injection.

At the tables below presented the program duration and ACE cycles for each configuration. Program duration is significantly shorter than the intervals which considered as ACE. (Table 6, Table 7, Table 8)



**Table 6: Execution and ACE cycles for each benchmark - Data Cache size: 16 KB**

			<b>Possibility of written back blocks</b>		
			<b>100%</b>	<b>50%</b>	<b>0%</b>
<b>cjpeg</b>	Execution cycles	24769238			
	ACE cycles		16653200000	13881800000	11202400000
<b>djpeg</b>	Execution cycles	7384642			
	ACE cycles		2686690000	2015230000	1374440000
<b>fft</b>	Execution cycles	23275293			
	ACE cycles		7264800000	5606040000	4080820000
<b>qsort</b>	Execution cycles	38438355			
	ACE cycles		15053500000	11744100000	8501020000
<b>sha</b>	Execution cycles	9930908			
	ACE cycles		2994280000	2599700000	2224150000
<b>stringsearch</b>	Execution cycles	804384			
	ACE cycles		303449000	258007000	215830000
<b>susan_corners</b>	Execution cycles	2140134			
	ACE cycles		877765000	682586000	496273000
<b>susan_edges</b>	Execution cycles	3577858			
	ACE cycles		2482890000	2100310000	1745540000
<b>susan_smoothing</b>	Execution cycles	12652161			
	ACE cycles		11432100000	10236900000	9511760000
<b>rijndael_enc</b>	Execution cycles	25426260			
	ACE cycles		2631840000	2148620000	1673580000

**Table 7: Execution and ACE cycles for each benchmark - Data Cache size: 32 KB**

			<b>Possibility of written back blocks</b>		
			<b>100%</b>	<b>50%</b>	<b>0%</b>
<b>cjpeg</b>	Execution cycles	24531382			
	ACE cycles		32600000000	27100000000	21700000000
<b>djpeg</b>	Execution cycles	7232260			
	ACE cycles		4499100000	3331440000	2261040000
<b>fft</b>	Execution cycles	23170500			
	ACE cycles		12317000000	12317000000	5886210000
<b>qsort</b>	Execution cycles	38115535			
	ACE cycles		27018300000	20294100000	13695600000
<b>sha</b>	Execution cycles	9857959			
	ACE cycles		5057630000	4546450000	4082710000
<b>stringsearch</b>	Execution cycles	800968			
	ACE cycles		406528000	335073000	267157000
<b>susan_corners</b>	Execution cycles	2130583			
	ACE cycles		1493100000	1112130000	742535000
<b>susan_edges</b>	Execution cycles	3563791			
	ACE cycles		3740920000	3029750000	2363770000
<b>susan_smoothing</b>	Execution cycles	12642338			
	ACE cycles		16321900000	14545800000	13694900000
<b>rijndael_enc</b>	Execution cycles	23310484			
	ACE cycles		6601380000	4797970000	3013750000

**Table 8: Execution and ACE cycles for each benchmark - Data Cache size: 64 KB**

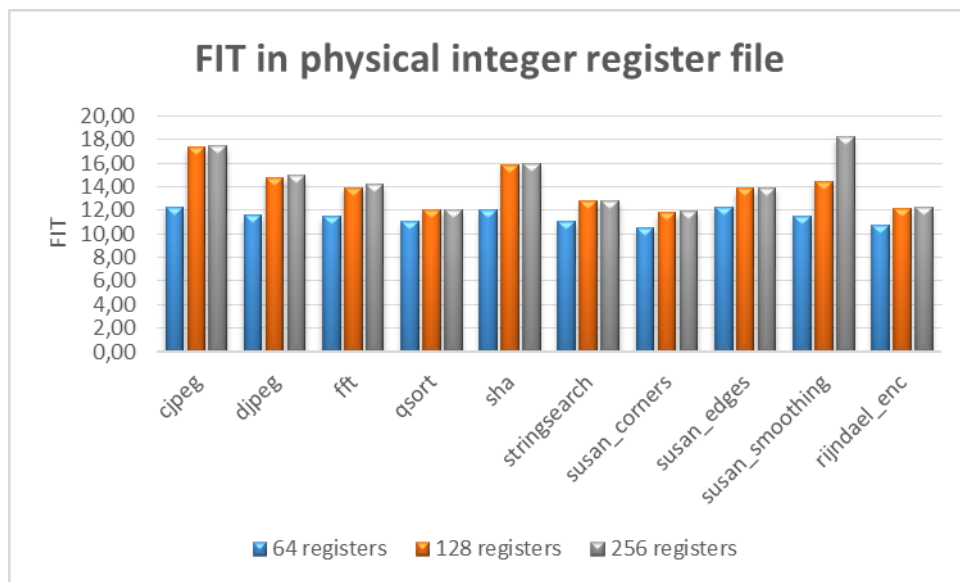
			<b>Possibility of written back blocks</b>		
			<b>100%</b>	<b>50%</b>	<b>0%</b>
<b>Cjpeg</b>	Execution cycles	24397209			
	ACE cycles		53916400000	43737700000	33721700000
<b>Djpeg</b>	Execution cycles	7133827			
	ACE cycles		7257210000	5535100000	3582450000
<b>Fft</b>	Execution cycles	23139759			
	ACE cycles		20622100000	13816100000	7087980000
<b>Qsort</b>	Execution cycles	37719873			
	ACE cycles		45806900000	32314400000	18917700000
<b>Sha</b>	Execution cycles	9833305			
	ACE cycles		5260870000	4730310000	4227450000
<b>Stringsearch</b>	Execution cycles	793112			
	ACE cycles		366489000	305387000	246845000
<b>susan_corners</b>	Execution cycles	2114984			
	ACE cycles		2284790000	1662710000	1066200000
<b>susan_edges</b>	Execution cycles	3546738			
	ACE cycles		4652960000	3635030000	2677370000
<b>susan_smoothing</b>	Execution cycles	12631586			
	ACE cycles		17457900000	15104900000	13551100000
<b>rijndael_enc</b>	Execution cycles	23126288	16702000000	8062530000	4563740000
	ACE cycles				

As analyzed in Section 2.4, FIT as a Vulnerability measurement reflects the accurate reliability of a component. FIT is estimated with the type:

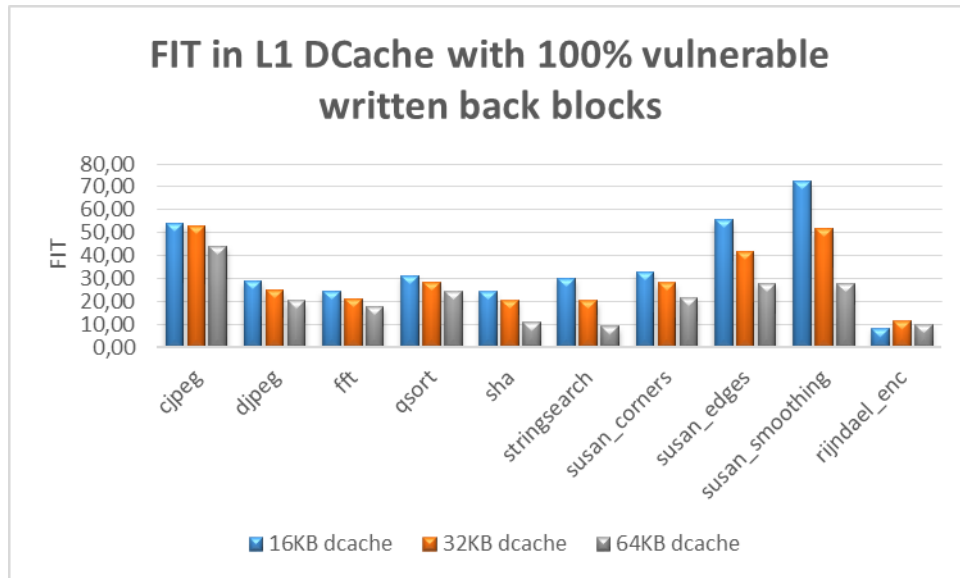
$$FIT = raw\ FIT\ rate \times number\ of\ bits \times AVF$$

Where raw FIT rate = 0.01 per bit, the number of bits is the overall number of bits in a structure. For example, for a physical integer register file with 256 registers the number of bits is: 256x64bits. AVF is the computed vulnerability factor (if AVF is 10%, for the FIT computation will be considered as 0,10).

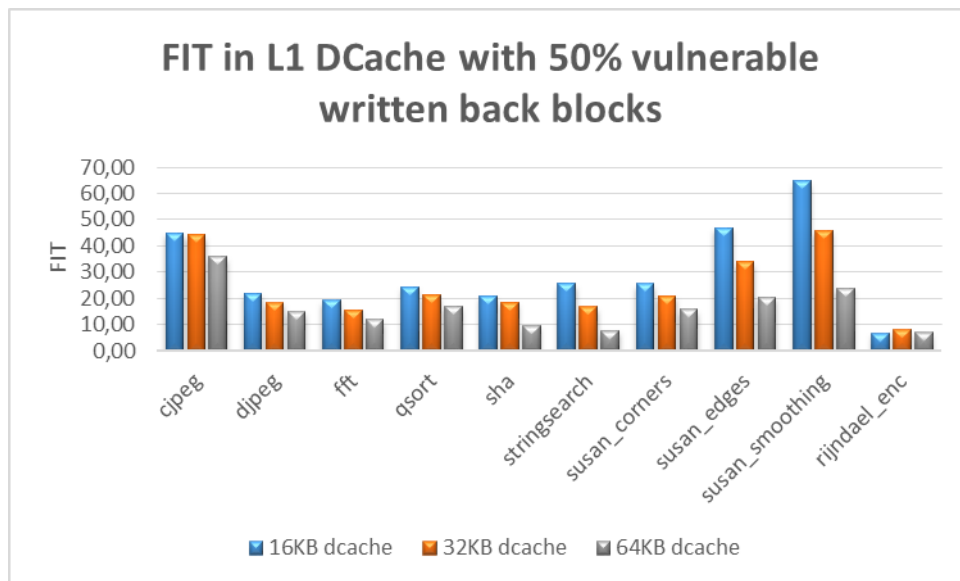
For every AVF diagram that shown before, there is a corresponding FIT diagram. In Figure 46, Figure 47, Figure 48 and Figure 49 FIT rate estimation is presented for the different configurations of physical integer register and L1 data cache.



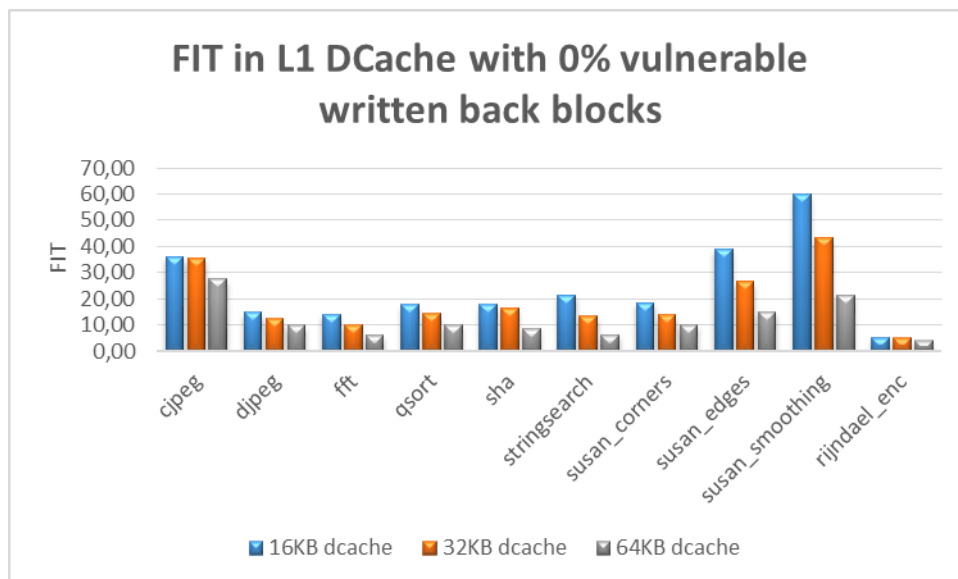
**Figure 46: FIT in physical integer register File**



**Figure 47: FIT in L1 DCache with 100% vulnerable written back blocks**



**Figure 48: FIT in L1 DCache with 50% vulnerable written back blocks**



**Figure 49: FIT in L1 DCache with 0% vulnerable written back blocks**

From Figure 46, Figure 47, Figure 48 and Figure 49 it is obvious that the benchmark with the most FITS is susan\_smoothing. The next benchmarks with the most FITS are susan\_edges and cjpeg. One FIT specifies one failure in a billion hours so for example, a benchmark with a FIT rate of 10 has  $10 \times 10^{-9}$  FITS. High FIT rate equals to big number of FITS and as a consequence it signs a not reliable component. In our study the most reliable benchmark is rijndael\_enc while the most unreliable benchmark is susan\_smoothing.

## 8.1 Conclusions

AVF estimation is a problem with many interesting aspects for a designer during the early design phase, because he can design reliable integrated circuits with smaller cost.

The problem is that at the next generation of microprocessors the hardware faults due to cosmic radiation, alpha particles is going to increase and as result more research about this topic has to be done. Until now there are many methods for computing vulnerability factors, fault injection is the most accurate method but of course with other methods like ACE analysis which is the subject of this thesis a fast but pessimistic vulnerability estimation can be offered.

## 9. APPENDIX

### 9.1 Commands for running a benchmark:

- To build, the next command should be executed:
  - `scons build/X86/gem5.opt`
- To restore a checkpoint L1 data cache size has to be set. For example, the next command for running the checkpoint has been set to 32KB. The executing checkpoint is in the subfolder *Gem5/m5out*. The commands for execution is :
  - `export M5_PATH= the path to gem5 images`
  - `./build/X86/gem5.opt --stats-file=stats.txt configs/example/fs.py --disk-image=...linux-x86-new.img --kernel=.../binaries/x86_64-vmlinux-2.6.22.9.smp -r1 --caches -l1d=32kb --l2cache --cpu-type=detailed --restore-with-cpu=detailed`
- For changing the size of physical integer register file, the python file *O3CPU.py* should modify. The next instruction has to change. For example, at this case the size of physical integer register file is 256 registers:  
`numPhysIntRegs= Param.Unsigned(256, "Number of physical integer registers")`
- For setting the associativity of cache memory, the python file *Tags.py* should modify. For example, with the next association, cache associativity is four:  
`assoc=Param.Int(4, "associativity")`

## 9.2 Hardware and Software configuration

The execution of benchmarks carried out in my personal computer. Its configuration details are shown below.

CPU: *Intel(R) Core(TM) i5 CPU M 520 @ 2.40GHz*

Cache size: 3072 KB

Disk size: 25.4 GB

OS: Ubuntu 14.04 LTS – 64 bit



## ABBREVIATIONS

AVF	Average Vulnerability Factor
ACE	Architectural Correct Analysis
DCache	Data Cache
ICache	Instruction Cache
FIT	Failure in Time
MTBF	Mean Time Between Faults
SE	System-call Emulation
FS	Full System
RTL	Register Transfer Level
O3CPU	Out of Order CPU
IEW	Issuing, Executing, Writeback

## SOURCE FILES AND HEADER FILES

regfile.hh	Gem5/src/cpu/o3
regfile.cc	Gem5/src/cpu/o3
pinakas.hh	Gem5/src/cpu/o3
dyn_inst.hh	Gem5/src/cpu/o3
commit_impl.hh	Gem5/src/cpu/o3
lsq_unit_impl.hh	Gem5/src/cpu/o3
O3CPU.py	Gem5/build/X86/cpu/o3
sim_events.cc	Gem5/src/sim
simulate.hh	Gem5/src/sim
simulate.cc	Gem5/src/sim
cache.hh	Gem5/src/mem/cache
cache_impl.hh	Gem5/src/mem/cache
blk.hh	Gem5/src/mem/cache
base_set_assoc.cc	Gem5/src/mem/cache/tags
Tags.py	Gem5/src/mem/cache/tags

## BIBLIOGRAPHY – REFERENCES

- [1] M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, N.Foutris, D.Gizopoulos, "Differential Fault Injection on Microarchitectural Simulators", IEEE International Symposium on Workload Characterization (IISWC), 2015.
- [2] R.C.Baumann, "Soft errors in advanced computer systems", IEEE Design & Test of Comp., May/June 2005.
- [3] S.Pan, Y.Hu, X.Li "IVF: Characterizing the vulnerability of microprocessor structures to intermittent faults", IEEE Transactions on VLSI Systems, May 2012.
- [4] S.Feng, S.Gupta, A.Ansari, S.Mahlke, "Shoestring: probabilistic soft error reliability on the cheap", ASPLOS 2010.
- [5] S.S.Mukherjee, C.T.Weaver, J.Emmer, S.K.Reinhardt, T.Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor, MICRO 2003.
- [6] G.Yalcin, O.S.Unsal, A.Cristal, M.Valero, "FIMSIM: A fault injection infrastructure for microarchitectural simulators", ICCD 2011.
- [7] N.J.Wang, A.Mahesri, S.J.Patel, "Examining ACE analysis reliability estimates using fault injection", ISCA 2007.
- [8] V.Sridharan, D.R.Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability", IEEE International Symposium on High Performance Computer Architecture (HPCA-15), 2009.
- [9] A.Biswas et al., "Computing architectural vulnerability factors for address-based structures", ISCA 2005.
- [10] L.Duan, B.Li, L.Peng, "Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics", HPCA 2009.
- [11] S.Feng, S.Gupta, A.Ansari, S.Mahlke, "Shoestring: probabilistic soft error reliability on the cheap", ASPLOS 2010.
- [12] N.Foutris, M.Kaliorakis, S.Tselonis, D.Gizopoulos, "Versatile architecture-level fault injection framework for reliability evaluation", IOLTS 2014.
- [13] Y.Luo et al., "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous reliability memory", DSN 2014.
- [14] N.L.Binkert et al., "The M5 simulator: modeling networked systems, IEEE Micro, July/August 2006.
- [15] M.K.Martin et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset", ACM SIGARCH Computer Arch. News, November 2005.
- [16] <http://www.m5sim.org>
- [17] <http://www.computerhope.com/jargon/c/clockcyc.htm> web
- [18] V. Sridharan and D. R. Kaeli, "Using pvf traces to accelerate avf modeling," in Proceedings of the IEEE Workshop on Silicon Errors in Logic-System Effects, 2010
- [19] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, Todd Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium.
- [20] Songjun Pan, Student Member, IEEE, YuHu, Member, IEEE, and Xiaowei Li, Senior Member, IEEE, "IVF: Characterizing the Vulnerability of Microprocessor Structures to Intermittent Faults", IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 20, NO. 5, MAY 2012.
- [21] Songjun Pan, YuHu, Xiaowei LI, Key Laboratory of Computer System and Architecture, Institute of Computing Technology, "IVF: Characterizing the Vulnerability of Microprocessor Structures to Intermittent Faults"
- [22] Chao(Saul)Wang, Zhong-Chuan Fu, Hong-Song Chen, Dong-Sheng Wang, "Characterizing the Effects of Intermittent Faults on a Processor for Dependability Enhancement Strategy", April 2014
- [23] Michail Maniatakis, Maria K. Michael, Yiorgos Makris, "Investigating the Limits of AVF Analysis in the Presence of Multiple Bit Errors", 2013 IEEE 19th International On-Line Testing Symposium (IOLTS)
- [24] Anastasiia Butko, Rafael Garibotti, Luciano Ost, Gilles Sassatelli, "Accuracy Evaluation of GEM5 Simulator System", [<http://www.lirmm.fr/ADAC>]
- [25] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sadashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, "The gem5 Simulator", ACM SIGARCH Computer Architecture News Vol. 39, No. 2, May 2011, [<http://gem5.org>]
- [26] Anders Handler, "Cycle-accurate Benchmarking of JavaScript Programs", Kongens Lyngby 2012

- [27] Brad Beckmann, Nathan Binkert, Ali Saidi, Joel Hestness,, Gabe Black, Korey Sewell, Derek Hower, "The gem5 Simulator ISCA 2011" , June 5th, 2011
- [28] M.R.Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite", IWWC 2001.
- [29] Tosaka, et al., "Impact of Cosmic Ray Neutron Induced Soft Errors, on Advanced Submicron CMOS circuits," Symposium on VLSI Technology Digest of Technical papers,1996
- [30] Nik Bessis (Edge Hill University, UK), "Technology Integration Advancements in Distributed Systems and Computing", Release Date: April, 2012. Copyright © 201
- [31] Wai-Kai Chen, "Memory, Microprocessor, and ASIC: Memory, Microprocessor and ASIC (Principles and Applications in Engineering)"
- [32] Z.Chishti, A.R.Alameldeem, C.Wilkerson, W.Wu, S.-L.Lu, "Improving cache lifetime reliability at ultra-low voltages", MICRO 2009.
- [33] California David L. Weaver,"The SPARC Architecture Manual Version 9 SPARC", International, Inc. San Jose