CHALMERS

On the Use of Assembly Code Metrics for Error Coverage Prediction

Master of Science Thesis in Networks and Distributed Systems

FATEMEH AYATOLAHI

BEHROOZ SANGCHOOLIE

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, December 2011

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

On the Use of Assembly Code Metrics for Error Coverage Prediction

FATEMEH AYATOLAHI
BEHROOZ SANGCHOOLIE

© FATEMEH AYATOLAHI, December 2011.
© BEHROOZ SANGCHOOLIE, December 2011.

Examiner: JOHAN KARLSSON

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden December 2011

ABSTRACT

In this thesis we present two prediction techniques for estimating the error coverage of target programs stimulated with different inputs. Preliminarily, we investigate the effects of the inputs on the failure distribution of the target programs using fault injection experiments. From this study, we could find a linear correlation between the length of the input and the error coverage. This result allows us to develop a linear regression model which is one of the prediction techniques that we adopt. As this correlation may not exist in other target programs, in the second technique called instruction-based prediction we propose an approach to predict the error coverage for an input using fault injection results of other inputs known as *base points*. In order to choose the base points, instruction-based prediction technique profiles the program through a set of metrics defined at the assembly code. Those metrics are used to feed a statistical technique that helps us select the more suitable inputs for the prediction. We also investigate the failure distributions of programs enhanced with the *triple time redundancy execution with forward recovery (TTR-FR)*. From the results of the failure distributions, we observe that the non-covered failure is reduced to on the average around 1.2% for all TTR-FR execution flows which has a minor correlation to input length as analyzed by linear regression equation.

Keywords: error coverage prediction, fault injection, assembly code, dependability assessment, software implemented fault tolerance, multivariate analysis, failure mode distributions.

Acknowledgments

It is a pleasure to thank all the people who made this thesis possible.

First of all, we want to express our sincere gratitude to our advisor Professor Johan Karlsson, whose supervisions and supports have been crucial to finish this thesis.

We would also like to especially thank Domenico Di Leo for his magnificent helps and guidance through all the phases of this thesis. Thank you for all the excellent ideas, discussions and encouragements you gave us during this time.

Special thanks to Dr. Daniel Skarin for his helps and advices with the Goofi-2 tool.

At last but definitely not least, we want to thank all our friends and our wonderful families. Their love and patience were the best support during these two years of our master studies.

Table of Contents

Table of Figures

List of Tables

1 Introduction

Today, the usage of computer systems in safety-critical applications is tremendously increased. Such systems must be equipped with proper fault-tolerance mechanism to be able to detect or correct transient, intermittent, and permanent faults. In addition, the scaling of integrated circuits makes them less reliable according to the hardware failures presented in [1]. This implies that hardware fault rates for transient, intermittent, or even permanent faults is going to increase in future computer systems. Even though these systems are expected to have robust hardware error detection mechanisms, their error coverage should be evaluated using analytical or/and experimental techniques. Fault injection, one of the most popular experimental techniques, is used to estimate system's error coverage in the presence of faults. However, performing fault-injection experiments is a time consuming process which inspire researchers to look for less expensive alternatives to estimate computer system's error coverage. Even though the error coverage is directly affected by the effectiveness of the error detection mechanism, the program under assessment along with its different possibilities of input sequences can influence the calculated coverage to a great extent. Despite the notable researches that have been accomplished on the influence of error detection mechanisms on the error coverage [2] [3] [4] [5] , a little investigation has been done regarding target program variations with respect to different inputs [6] [7]. To the best of our knowledge, only little literature has investigated the effect of different inputs on error coverage when a target program is enhanced with a fault tolerance mechanism.

In this thesis we aim at performing fault injections on two target programs using different input sequences to see whether there is any correlation between the input and the failure distribution. Moreover we investigate the possibility of using assembly code metrics to predict error coverage distributions without performing fault injection. Furthermore, the effect of different inputs on error coverage is analyzed when the target programs are enhanced with temporal redundancy fault tolerance mechanism.

By analyzing the outcomes of the fault injection experiments, we proposed two prediction techniques called *linear regression model* and *instruction-based prediction*. The former technique proves that in our target programs, there is a linear correlation between the length of input and the error coverage. This can then be used to predict the error coverage of the target programs using only the length of the input sequence. The outcome of the prediction from the linear regression model has a high accuracy, while it is not useful when there is no correlation between the input of a target program and its error coverage. On the other hand, the instruction-based prediction technique uses assembly level signature of a target program along with statistical techniques for multivariate analysis to predict the coverage of an input sequence using the fault injection results of another input sequence (base input). Though the accuracy of instruction-based technique lies to a great extent to the selection of the base input, it can be ideally adopted for all target programs.

The remainder of the thesis is organized as follows. In Chapter 2 we present the required taxonomy and a brief background on the fault injection technique. Chapter 3 describes our methodology which is a detailed description of the instruction-based prediction technique along with assembly code metrics. The implementation of our target programs along with the experimental setup of the fault injection mechanism are presented in Chapter 4. Moreover, in this Chapter we describe the *Assembly-Level Signature Creator (ASC)* tool which is used to calculate the values of different metrics. In Chapter 5 we present the results of our study on the two target programs with/without software implemented fault tolerance. In addition, the results of two different predictors are compared with each other. We conclude with the limitations of this thesis along with future research in Chapter 6.

2 Background

2.1 Terminology

Common concepts of dependable and secure computing are defined in [8] which are summarized in this section based on their relevance to the thesis topic.

2.1.1 Dependability

A system is called *safety-critical* if its failure to deliver a correct service will endanger human life or the environment. Thus, the safety-critical system needs to provide *dependability* which is the ability of the system to avoid service failures that are more frequent and severe than acceptable. The generic concept of dependability includes three basic elements; attributes, threads and means.

2.1.1.1 Dependability Attributes

The main attributes for designing a dependable computer system are *reliability*, *availability* and *safety*. The reliability of a system is the probability that it provides a correct service for a specific period of time, when there are possibilities for permanent failures. The availability is the probability of the system to provide a correct service at a certain point in time, when the system is repairable. From the other point of view, safety provides the probability that the system failure does not result in any catastrophic consequences. The significance of each attribute in a system depends on the application usage. Furthermore, another attribute called *maintainability* should be considered in the implementation phase of fault tolerant computer systems which is the ability to perform modification and repair a system.

2.1.1.2 Dependability Threats

The dependability threats include *faults*, *errors* and *failures*. The deviation of the system's delivered service from the correct service is called the system failure. This deviation is triggered by an error which is an incorrect state in the system. The cause of an incorrect state is called a fault which is classified into three classes of *development faults, interaction faults, and physical faults*.

Development faults are introduced in the development phase of the system. These faults are usually caused by human during software or hardware specification, design, and implementation. Interaction faults on the other hand, are introduced by human interacting with the system in the usage phase, e.g., a user typing a wrong input value. This category also accounts for the faults that take place during the exchange of information between computer systems. The third group of faults is physical faults which are either introduced in development or usage phases of the system. They can cause *permanent*, *transient*, and *intermittent* hardware failures in which the latter can be caused by high energy cosmic neutrons.

Nowadays, computer hardware and processors are getting smaller and transistors are becoming less reliable. Bokar in [1], characterizes the main causes of failure in VLSI circuits as *process variations*, *ionizing particle radiation*, and *aging effects*. Hazucha et al. in [9] also believe that in each technology generation, single event upsets rate, caused by ionizing particle radiation, per bit is increased by about 8 percent. In addition, *negative bias temperature instability (NBTI)* and *Hot Carrier Injection (HCI)* can over time, cause failures in integrated circuits. Therefore, computer systems should be equipped with fault tolerant mechanisms in order to increase the availability and reliability of the system.

2.1.1.3 Dependability Means

The dependability means include *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting*.

Fault prevention techniques are related to development phase of the system. Using these techniques, faults can be prevented in software or hardware. Moreover, design rules and formal verification can to some extent avoid fault occurrences in software and hardware. Fault removal techniques on the other hand, are involved in development and operational phases. During the development phase, validation and verification techniques can be used to reduce the number of faults. In addition, while the system is operational, maintenance and repair techniques can reduce faults and their consequences. Fault forecasting, as another dependability mean, tries to estimate system's behavior in the presence of faults. With the help of quantitative and qualitative evaluation techniques such as *markov modeling* and *failure modes and effects analysis (FMEA)*, it is possible to estimate dependability attributes. Fault tolerance techniques try to avoid system failure in the presence of faults. They require *error detection* and *error recovery* mechanisms to detect an active error and return the system to a nonfaulty state.

2.1.2 Fault Tolerance Techniques

Fault tolerance techniques imply built-in redundancy which can be formed as *hardware redundancy*, *software redundancy*, *temporal redundancy* or *information redundancy*.

2.1.2.1 Hardware Redundancy

There are three hardware redundancy techniques, namely *voting*, *standby*, and *active* redundancy. As an example, *triple modular redundancy (TMR)* is one of the most common voting redundancy configurations in which three redundant modules are configured to perform the same functionality using the same set of inputs and applications. Modules outputs are then delivered to a voter which applies majority voting to deliver the result. Therefore in TMR, one module failure can be masked by the other two redundant modules, while the error can only be detected, not masked, in case of having two erroneous modules.

2.1.2.2 Software Redundancy

Software redundancy techniques can be divided into two different classes of *with diversity* and *without diversity*. The former corresponds to diversity in data, software design, and development process, while the latter provides error detection and/or recovery by using redundant data and instructions as mentioned in [10] for transient physical faults. *N-version programming* [11] is a well-known software redundancy technique which uses the diversity in development teams, program developers, programing languages, and/or program designs. All outputs generated by different program versions should be identical which is identified by a majority voter. Another example of software redundancy techniques is *recovery blocks*. As explained in [12], in this technique there are two primary and alternative software modules which are indeed two versions of the same program. An acceptance test is constantly applied to the outcome of the primary module so that in case of any failure, the output of the alternative module can be selected.

2.1.2.3 Information Redundancy

Information redundancy aimed to protect data stored in memories or the data transferred via networks. *Systematic* and *non-systematic* codes are two commonly used techniques. As an example, the parity check is a systematic code which generates some redundant bits attached to the original data to detect or correct the error in data. In the non-systematic technique, new sequence of data is created using the mapping of the original data. Thus it is designed in a way to detect or correct errors in the original data.

2.1.2.4 Temporal Redundancy

As mentioned before, the transient faults may cause a temporary erroneous result, so if the system is restored to its correct state and the program is executed again, it is possible to detect the error by double execution of the program. This can be done using result comparison of the two executions. The error can also be masked by executing the code three times and making use of the majority voter to decide on the correct result. This technique is more explained in [13] and is used as the fault tolerance technique in this thesis.

2.2 Fault Injection

Fault injection has been around since 1970s. Fault removal and fault forecasting are among the main purposes of using fault injection. It can also be used as a technique to measure the error coverage by fault introduction. The error coverage can then be used to calculate the availability and reliability of a computer system. Moreover, fault injection is a technique for testing programs' fault tolerant mechanisms. One of the most realistic ways of measuring the effectiveness of a fault tolerant mechanism is by injecting artificial faults to the systems and assessing the robustness of the system. The injected fault can be propagated throughout the program and resulted in an error in sections which were not protected properly by any fault tolerant mechanisms. Some properties of fault-injection are as follows [14]:

- Repeatability, the ability of injecting the same fault and achieving the same result.

- Controllability, the ability of controlling the time and location of injecting a fault.
- Observability, the ability of observing the effect of an injected fault.
- Reachability, the ability of reaching possible fault locations in a processor.

Fault injection techniques can be categorized in three different groups of *software-implemented*, *hardware-implemented*, and *radiation-based* injections as presented in [15]. Throughout the remaining of this section, we will present these techniques and a number of tools using them. N.B. faults can also be injected into a simulated model of a system that includes injections in device and logical levels along with injections in the system, functional block, and instruction set architecture.

2.2.1 Software-Implemented Fault Injection

Software-implemented fault injection (SWIFI) can be divided in two different categories, pre-runtime (compile-time) injection and runtime injection. In the pre-runtime SWIFI, program's data or source code is altered to inject simulated faults. Code insertion and mutation testing are the two commonly used pre-runtime SWIFI techniques. In mutation testing, an existing line of code is modified which corresponds to the programmers' unintentionally made mistakes. Whereas in code insertion, extra line(s) of code is inserted into the source code. Pre-runtime SWIFI was used in a distributed real-time system enhanced with a fault tolerance mechanism [16] [17]. Furthermore in runtime SWIFI, faults are injected into the system while it is running (execution-time). The system is equipped with additional software responsible for injecting faults which is either time-triggered or interrupt-triggered. Examples of tools which used runtime SWIFI include FIAT [18], FERRARI [2], and Exhaustif [19].

2.2.2 Radiation-Based Fault Injections

In this technique, the system is exposed to particle radiation and *electromagnetic inferences (EMI)*. The main obstacles facing this technique are the *controllability* and *repeatability*. This is due to the fact that fault locations cannot be selected easily and also it is difficult to inject a previously injected fault and achieve the same result. In [20], heavy-ion from a Californium-252 source is radiated to a microprocessor in order to validate a fault-handling mechanism.

2.2.3 Hardware-Implemented Fault Injection

Faults can also be injected into systems' hardware using *pin-level* and *test port-based* injections along with *power supply disturbances*. In pin-level and power supply disturbance techniques, faults are directly injected into the Integrated Circuit and microprocessor's pins using additional signal or short voltage drops, respectively. An example of a tool using pin-level injection is MESSALINE [21]. In test port-based injections on the other hand, faults are mostly injected into the registers and memory words of a processor using test ports like *BDM* and *Nexus*.

All experimental setups presented in this thesis use GOOFI-2 [22] tool equipped with exception-based, instrumentation-based, and Nexus-based fault-injection techniques. However, we only use its Nexus-based fault-injection; more details can be found in subsection 4.2. N.B. exception-based and instrumentation-based injections are actually using SWIFI technique.

2.3 Related Work

Error coverage estimation is performed by evaluating the system with respect to fault occurrence or fault activation [8]. Since the high complexity of modern computer systems makes it difficult to estimate error coverage analytically, different techniques such as fault injection are used to be able to estimate the error coverage. The authors in [23] and [24], estimated the error coverage through analytical models. In particular, in [23] they evaluate two sampling techniques for error coverage estimation, while in [24] they adopt statistical of extreme, i.e., statistical techniques to estimate rare events. In [25] and [26] coverage is calculated with respect to all possible inputs, fault location, and fault injection time. The work [27] surveys a variety of coverage models, from simple phase-type models to stochastic Petri net in order to predict coverage.

Fault injection is an expensive time-consuming technique due to its mentioned properties. It is also notable that different program inputs result in different *execution flows* which means each injection setup corresponds only to a specific program input. Therefore, other techniques such as fault prediction have been introduced trying to validate fault-tolerance mechanisms in a more cost-efficient way.

Program profiling [28] [29] is a method used in fault prediction. It is commonly used for software maintenance and testing [30] and also in literature analyzing fault localization, such as [31] [32]. In [31], different spectra (path, branch, data-dependence, etc.) of a target program are introduced and the correlation between various spectra types and program failure is analyzed. The main weakness of this method is that it is working on programs' source code. In [33] [7], a path-based fault coverage prediction technique is proposed which uses the injection results of an input sequence (base input) to predict the error coverage for another input sequence. The path-based predictor is built upon the assumption that different inputs might form different execution flows which cause different code blocks to be executed different number of times. Based on the weight of each code block and the injection results of the base input sequence, the error coverage can be predicted for another input.

The main weakness of the path-based prediction is the arbitrarily selection of the base input. In this thesis, we introduce an instruction-based prediction technique which profiles assembly level code instead of source code. We believe that assembly code is a better representative of target programs, see chapter 3. Moreover, in instruction-base estimation, not only more than one input can be selected as the base input, but also the base inputs are chosen wisely.

3 Methodology

In this thesis we investigate the effect of different inputs on programs' failure distributions. It is accomplished by performing fault injection experiments on different target programs. We believe that inputs with similar characteristics result in similar failure distribution for the same target program. So, if we appropriately characterize an execution flow associated to an input, the failure distribution of another execution flow with similar characterizations can be estimated. In order to investigate this, different inputs are chosen for two different target programs, *workloads*, and their execution flows are compared and analyzed at the assembly code level. In addition, an assembly level signature is computed for each execution flow to be used in the prediction of failure mode distribution.

3.1 Assembly Code Specifications

As mentioned above, the assembly code of two workloads are examined to be able to characterize their execution flows. There are a number of reasons behind selecting the assembly code:

- Assembly code is a proper representative of a program which is executed on the hardware exactly as it is without further optimization or interpretation by compilers or optimizers. Furthermore, since the fault model used in this thesis includes hardware transient faults, it will be more realistic to analyze target programs using their assembly code. Moreover, assembly code shows exactly which instructions, registers or memory locations are involved in the workload and indicates the potential places for failures during the execution flow.

- Even though, the assembly instruction set is different for different microprocessors, the general concepts used in this thesis to design proper signature, from assembly-level *metrics*, is applicable to different kinds of hardware architectures. In other words, our proposed metrics are not dependent on special hardware architecture and they can be calculated for other hardware architectures with minor changes in the implementation.
- In addition to hardware flexibility, the target program can also be written in different programing languages, e.g., java/C/C++. This is due to the fact that programs are translated to assembly code before they could be executed on the hardware. Therefore, our approach is not dependent on the programing language.

3.2 Assembly Code Metrics

The assembly codes of two workloads are characterized in this thesis to estimate the failure distribution of their execution flows. In order to be able to characterize the fault-free execution of a workload, we classified the assembly code in three general categories; instructions, registers, and memory sections. For each of these categories, a number of metrics are designed, see Table 3.2. In this table, the first two metrics are calculated according to general characteristics of an execution flow:

- (*NEI*), the total number of different instructions executed in an execution flow, regardless of the number of times each of them was executed.
- (*NE*), the length of the execution flow in terms of the number of executed instructions. In other words, the number of times the *program counter register (PCR)* is updated.

3.2.1 Instruction Metrics

The workload instructions are categorized into six groups of instructions namely, load, store, arithmetic, branch, logical, and processor, see Table 3.1. The summary of the instruction set is available in [34]. For instance, *add, sub, div, etc.* instructions are included in arithmetic category, while *and, or, xor, etc.* are included in logical category. In this way, the metrics for each category of instructions are studied in terms of:

- The number of times instructions of each category are executed, e.g., the total number of load instructions (*lwz, lis, lbz, etc.*) executed in an execution flow.
- The percentage of the executed instructions for each category, out of the total number of instructions executed in an execution flow (*NE*).
- The average distance between two consecutive executions of instructions in a specific group. For instance, if instructions are executed in the order shown in Figure 3.1, the first consecutive arithmetic instructions, *addi*, will have a distance of 3, while the second consecutive instance, *addi* and *subf*, will have a distance of 1. Finally, there is a distance of 3 between the last consecutive arithmetic instructions, *subf* and *addi*. The average distance metric for the arithmetic category can then be calculated using the average of these values which is 2.33.

Table 3.1. A sample list of different instruction categories

Categories	Instructions
LOAD	*lbz, li, lwi, lmw, lswi, …*
STORE	*stb, stub, sth, sthx, stw, stwbrx, …*
ARITHMETIC	*add, addo, subf, divw, mulhw, …*
BRANCH	*b, bl, bc, bclr, …*
LOGICAL	*and, or, xor, cmp, rlwimi, …*
PROCESSOR	*mcrf, mftb, sc, rfi, …*

3.2.2 Register Metrics

These metrics refer to registers used in each execution flow. The most important registers used in each execution flow are studied and grouped into three different categories: *condition register (CR)*, *stack pointer register (SP)*, and *general purpose registers (GPR)*. The following metrics are studied:

- The total number of different general purpose registers accessed in an execution flow.
- The number of times registers of each category are read in an execution flow.
- The number of times registers of each category are written in an execution flow.
- The average distance between two consecutive "read" from registers of each category.
- The average distance between two consecutive "write" into registers of each category.

3.2.3 Memory Metrics

The memory metrics and register metrics are designed similarly, i.e., the number of read/write and the average distance between two consecutive "read/write" are calculated in the same way with the exception that in memory metrics, sections of memory are considered instead of registers. The notable memory sections of our workloads are *text*, *stack*, *bss/sbss*, and *data/sdata*.

The complete list of the mentioned assembly code metrics is shown in Table 3.2. A combination of these metrics is used as the signature for each execution flow to see whether there is any correlation between different inputs and their failure mode distribution; they can then be used in error coverage prediction.

```
2548    38 09 26 30    Addi    r0,r9,9776
254c    90 1f 00 0c    Stw     r0,12(r31)
2550    3d 20 00 3f    Lis     r9,63
2554    38 09 70 00    Addi    r0,r9,28672
261c    7c 6a 18 50    Subf    r3,r10,r3
2620    83 c1 00 08    Lwz     r30,8(r1)
2624    7c 08 03 a6    Mtlr    r0
2628    38 21 00 10    Addi    r1,r1,16
```

Figure 3.1. Sample of workload's execution flow

Table 3.2. Assembly metrics - Signature of an execution flow

Metric Number	Metric Name	Description
General Metrics		
1	NEI	Number of different Executed Instructions, the total number of different instructions in the assembly code.
2	NE	Number of Executed instructions, i.e., the number of times that the PCR has been updated.
Instruction Metrics		
3	NLI	Number of Load Instructions.
4	NSI	Number of Store Instructions.
5	NAI	Number of Arithmetic Instructions.
6	NBI	Number of Branch Instructions.
7	NLGI	Number of Logical Instructions.
8	NPI	Number of Processor Instructions.
9	PLI	Percentage of Load Instructions. (NLI/NE)
10	PSI	Percentage of Store Instructions. (NSI/NE)
11	PAI	Percentage of Arithmetic Instructions. (NAI/NE)
12	PBI	Percentage of Branch Instructions. (NBI/NE)
13	PLGI	Percentage of Logical Instructions. (NLGI/NE)
14	PPI	Percentage of Processor Instructions. (NPI/NE)
15	LAD	Load Distance, the average distance between two consecutive executions of load instructions.
16	SD	Store Distance, the average distance between two consecutive executions of store instructions.
17	AD	Arithmetic Distance, the average distance between two consecutive executions of arithmetic instructions.
18	BD	Branch Distance, the average distance between two consecutive executions of branch instructions.
19	LGD	Logical Distance, the average distance between two consecutive executions of logical instructions.
20	PD	Processor Distance, the average distance between two consecutive executions of processor instructions.
Register Metrics		
21	NGPR	Total number of different GPRs accessed.
22	NRCR	Number of access in read mode to condition register.
23	NWCR	Number of access in write mode to condition register.
24	NRSP	Number of access in read mode to the stack pointer.
25	NWSP	Number of access in write mode to the Stack pointer.
26	NRGPR	Number of access in read mode to GPRs (all GPRs except r1, that has been counted in NRSP)
27	NWGPR	Number of access in write mode to GPRs (all GPRs except r1, that has been counted in NWSP)
28	NRXER	Number of access in read mode to the XER.
29	RDCR	The average distance between two consecutive read operations from the CR.
30	WDCR	The average distance between two consecutive write operations into the CR.
31	RDSP	The average distance between two consecutive read operations from the SP.
32	WDSP	The average distance between two consecutive write operations into the SP.
33	RDGPR	The average distance between two consecutive read operations from the GPRs.
34	WDGPR	The average distance between two consecutive write operations into the GPRs.
35	RDXER	The average distance between two consecutive read operations from the XER.
Memory Metrics		
36	NRTXT	Number of times the program reads from the text section.
37	NRAS	Number of times the program reads from the Stack section.
38	NWAS	Number of times the program writes into the Stack section.
39	NRAB	Number of times the program reads from the bss/sbss section.
40	NWAB	Number of times the program writes into the bss/sbss section.
41	NRAD	Number of times that the program read from data/sdata section.
42	NWAD	Number of times the program writes into the data/sdata section.
43	RSD	The average distance between two consecutive read operations from the stack section in terms of PC executions.
44	WSD	The average distance between two consecutive write operations into the stack section in terms of PC executions.
45	RBD	The average distance between two consecutive read operations from the bss/sbss section in terms of PC executions.
46	WBD	The average distance between two consecutive write operations into the bss/sbss section in terms of PC executions.
47	RDD	The average distance between two consecutive read operations from the data/sdata section in terms of PC executions.
48	WDD	The average distance between two consecutive write operations into the data/sdata section in terms of PC executions.

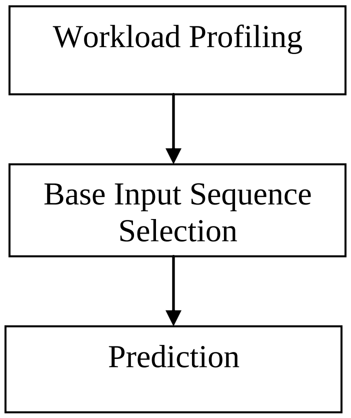


Figure 3.2. The instruction-based prediction process

3.3 Instruction-based Prediction

In this prediction technique, we profile different execution flows using assembly level metrics, discussed in Section 3.2, calculated from the fault-free run of each execution flow. The output of the profiling step allows us to select or generate the best base input sequences which can then be used to predict the error coverage of a new input sequence, see Figure 3.2.

The instruction-based prediction technique is built upon the fact that different inputs cause different number of instruction categories to be executed. As mentioned in Section 3.2.1, we defined six different instruction category metrics corresponding to the number of executed instructions (metric numbers 3 to 8 in Table 3.2). The weighted sum of these six metrics is used along with the failure distribution outcome of an input sequence (*base input*) to predict the failure distribution of a target input sequence.

Let $E_{c,e}$ denote the percentage of error classification c (N.B. error classifications are discussed in Section 5.1.2) for the target input sequence e. For every error classification and input sequence, $E_{c,e}$ is predicted using the equation (3.1):

$$E_{c,e} = \sum_{i=1}^{n} P_{c,i} * W_{e,i} \tag{3.1}$$

Here i corresponds to the instruction categories; and $P_{c,i}$ is calculated according to the fault injection results of the base input sequence, see equation (3.2). In this equation, $n_{c,i}$ refers to the number of injected faults in instruction category i that resulted in error classification c, while n_i corresponds to the total number of faults injected in instruction category i.

$$P_{c,i} = \frac{n_{c,i}}{n_i} \tag{3.2}$$

The term $W_{e,i}$ in equation (3.1) is an estimated weight for each instruction category i, which corresponds to the percentage of instruction category i that leads to error classification c, see equation (3.3).

$$W_{e,i} = \frac{n_{e,i}}{n_e} \tag{3.3}$$

In the estimation of the weight factor, n_e refers to the estimated number of injections for the target input sequence e which can be calculated using equation (3.4), while $n_{e,i}$, corresponds to the estimated total number of faults which are injected into the instruction category i in case of using the target input sequence e, see equation (3.5).

$$n_e = \sum_{j=1}^{n} \frac{n_j}{x_j} * x_{e,j} \tag{3.4}$$

$$n_{e,i} = \frac{n_i}{x_i} * x_{e,i} \tag{3.5}$$

Here n_i represents the number of faults injected in instruction category i for the *base* input sequence; x_i shows the total number of executed instruction category i for the *base* input sequence, and $x_{e,i}$ is the total number of executed instruction category i for the target input sequence e.

Now that we have all the elements of equation (3.1), the percentage of error classification c for the target input sequence e can be predicted using equation (3.6).

$$E_{c,e} = \sum_{i=1}^{n} \frac{n_{c,i}}{x_i} * \frac{x_{e,i}}{\sum_{j=1}^{n} \frac{n_j}{x_j} * x_{e,i}} \tag{3.6}$$

It is clear that the instruction-based predictor is composed of two parts, i.e., data $(n_{c,i}, x_i)$ regarding the input sequence in which a fault injection campaign has been conducted for and data $(x_{e,i})$ that refers to the input for which different error classification prediction is required.

In this way, the error coverage of input sequence e can be predicted using equation (3.7) where vf represents the value failure classification. N.B. value failures are errors that lead to the production of wrong results.

$$C_e = 1 - \sum_{i=1}^{n} P_{vf,i} * W_{e,i} \tag{3.7}$$

The effectiveness of the instruction-based error coverage prediction technique lies in the proper selection of the *base* input, i.e., not every *base* input result in an accurate prediction. Therefore, we need to enhance the instruction-based technique with a proper way of selecting the *base* input. In the next section, a well-known mathematical technique called *principal component analysis (PCA)* [35] is presented. This technique, along with the proposed assembly level metrics, guides us in selecting the best *base* input from a list of already analyzed execution flows.

3.3.1 Using Principal Component Analysis (PCA) for Base Input Selection

PCA, invented by Karl Pearson in 1901, is a mathematical technique that converts a set of possibly correlated variables into a set of uncorrelated variables known as principal components, i.e., the PCA can reduce the initial dimensions in a smaller space. Moreover, components are sorted based on variability in the data, i.e., the first component has the highest variance. The number of components should be chosen in a way that the sum of their variance accounts for the 80% of the initial variance. Thus by using the first few components, PCA can generate an N-dimensional representative of a higher-dimensional data space. This means that the metrics we proposed in Section 3.2 can now be used as the initial multivariate dataset and with the PCA it is possible to reduce them in a two dimensional space. The results of the analysis are enhanced and tend to be clearer in case of using the multivariate “normal” distribution of the initial dataset. Therefore, we also normalize each calculated metric using equation (3.8) and then provide the normalized dataset to the PCA.

$$Normalized\ value\ of\ X = \frac{X - \mu}{\sigma} \tag{3.8}$$

Here X is the metric value which should be normalized, μ is the arithmetic mean of the distribution, and σ is the standard deviation of the distribution.

Our main objective is to find a proper *base* input which can be used in the instruction-based predictor. Therefore we need to find a way to compare the *base* input and the target input sequence e, which is the input of the execution flow that we want to predict its failure distributions.

Each of our target programs has nine different execution flows each corresponding to an input sequence. After applying PCA to the normalized results of the metrics, input sequences can be presented using their (x, y) coordinates, for example Figure 3.3 corresponds to PCA-generated inputs for SHA execution flows. All metrics are involved in the calculation of the principal components, i.e., in order to calculate the value of each principle component, its eigenvector should be multiplied to the normalized results of the metrics.

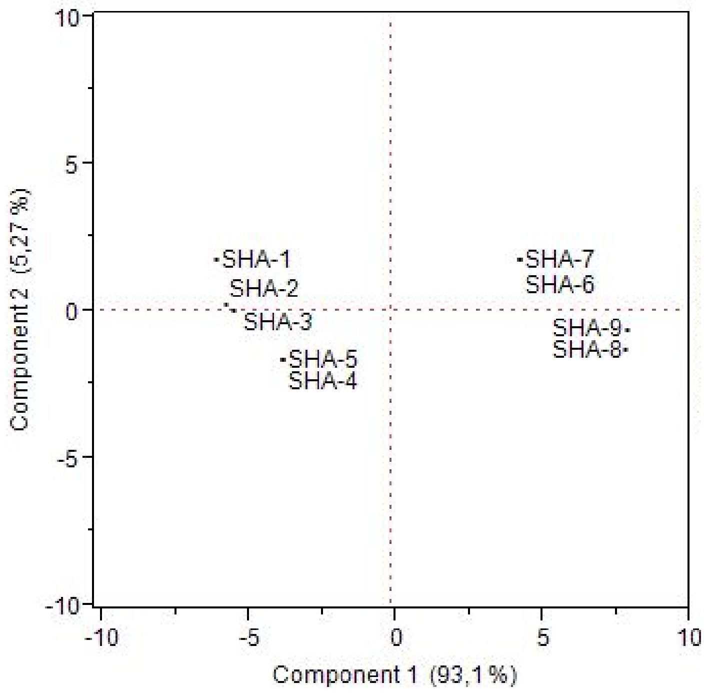


Figure 3.3. PCA-generated input sequences of SHA execution flows

Several metrics also have the same correlation with each other, e.g., an increase in the number of execution of an instruction might also increase the number of access to a specific register. Therefore, principle components can also be calculated using arbitrary number of metrics from all non-correlated groups of metrics.

Table 3.3 shows the calculation of the first two principle components using eight “normalized” metrics selected from four non-correlated groups of metrics. N.B. metric names have been taken from Table 3.2 and the multiplicands correspond to each principle component’s eigenvector. These eight metrics are selected from all the three main metric categories described in Table 3.2. However, principle components could also be calculated using other combinations of metrics.

Table 3.3. Principle Component calculation using optimized number of metrics

SHA	PCA1 = 0.35159 * NLGI + 0.35937 * PLI - 0.35608 * PAI + 0.35829 * NRCR + 0.35328 * RDSP + 0.35036 * RDGPR - 0.34654 * WSD + 0.35273 * WBD
	PCA2 = 0.40282 * NLGI - 0.14066 * PLI + 0.28238 * PAI + 0.19339 * NRCR + 0.36322 * RDSP - 0.42618 * RDGPR + 0.49539 * WSD + 0.37663 * WBD
CRC	PCA1 = 0.35499 * NAI + 0.34155 * PLI - 0.34155 * PBI + 0.36295 * SD + 0.35499 * NRCR + 0.35499 * RDSP + 0.35499 * NRXER + 0.36178 * WSD
	PCA2 = 0.34778 * NAI - 0.42043 * PLI + 0.42043 * PBI - 0.27901 * SD + 0.34778 * NRCR + 0.34778 * RDSP + 0.34778 * NRXER - 0.29126 * WSD

By analyzing the injection results of each execution flow and the coordinates of other PCA points, we conclude that closer *base* inputs to the target input e in the Cartesian coordinate system result in a better prediction. Therefore the best *base* input will be a point in the two-dimensional space closest to the target input sequence e. This is one of the strength of instruction-based technique when compared to path-based technique where the *base* input is selected arbitrarily. For example, as can be seen in Figure 3.3, in case we have already estimated failure distribution of SHA-7 using fault injection, we can predict the failure distribution of SHA-6 using SHA-7 as the *base* input.

As mentioned before, the effectiveness of the instruction-based technique lies to a great extent on the selection of the *base* input i.e., it is useful when we predict different error classifications of an execution flow with an input sequence e so close to the *base* in the two-dimensional PCA representation. Throughout the remaining of this section, we introduce an algorithm which can be used to predict both “adjacent” and “distant” points. In our proposed algorithm, we either select or generate the best possible *base* input which divides the algorithm into two parts.

In the first part of the algorithm, distances from all PCA-generated points to the target input point e are calculated. These distances will then be sorted and put in a list called *BaseDistances*. As can be guessed, an input sequence with the smallest distance in the *BaseDistances* gives us the best prediction when compared to other input sequences. The main weakness of this part is that the closest PCA-generated point might still be distant from e. Therefore, in the second part of the algorithm, we use two points instead of one to generate a better *base* for our prediction.

In the second part of the algorithm, all combinations of the PCA-generated points should be coupled and analyzed. The idea with this part is to find possibly a better point on the line connecting the elements of each couple, i.e., to generate a point closer to the target input sequence e.

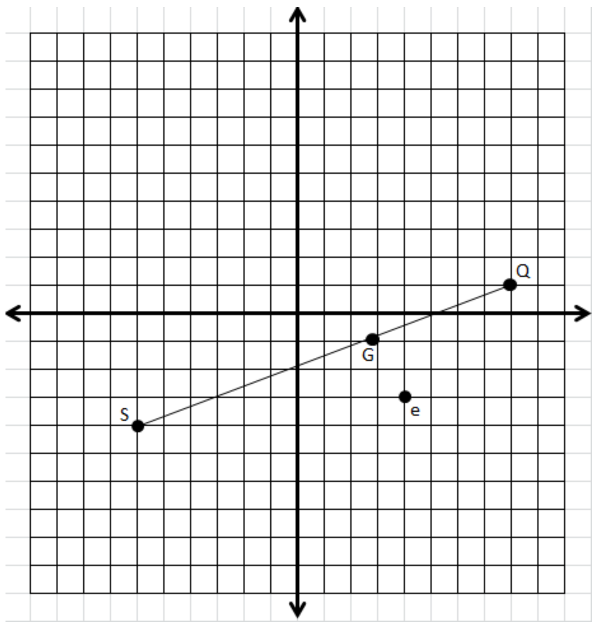


Figure 3.4. An example of generating a third point (*G*) using two other points (*S*,*Q*)

Table 3.4. Calculated distances of the potential *base* inputs to the target input *e*

	Q – e	**S – e**	**G – e**
Distance	5.65	10.04	2.43

Figure 3.4 illustrates an example of generating a third point, *G*, using the couple *(S, Q)*. This point is constructed using coordinates and weights of *S* and *Q;* each of these points receives its weight based on its distance to e. This leads to the case in which a point with a smaller distance to *e* (*Q* in this example) causes *G* to be generated closer to *e* compare to the more distant point (*S* in this example). As can be seen in Table 3.4, *G* is closer to the target input e, which indicates that it is a better candidate to be the *base* input. Equation (3.9) and (3.10) show how to calculate the coordinates of the new generated point, *G*.

$$x_G = x_S - \Delta x * W_{Q,e} \tag{3.9}$$

$$y_G = y_S - \Delta y * W_{Q,e} \tag{3.10}$$

Here Δx and Δy are calculated as

$$\Delta x = x_S - x_Q \,, \Delta y = y_S - y_Q \tag{3.11}$$

Furthermore, $W_{Q,e}$ and $W_{S,e}$ correspond to the given weights of *S* and *Q* based on their distances to *e*;

$$W_{S,e} = \frac{D_{Q,e}}{D_{S,e} + D_{Q,e}} \tag{3.12}$$

$$W_{Q,e} = \frac{D_{S,e}}{D_{S,e} + D_{Q,e}} \tag{3.13}$$

Here $D_{S,e}$ and $D_{Q,e}$ are the distances between *e* to *S* and *Q* respectively. Now that we have another point in the Cartesian coordinate system, its distance to e will be put in the *BaseDistances* list.

Since it is not always the case to find a better point by using any two points, this procedure should be performed for all different combinations of the previously injected execution flows to discover the best possible *base* input. The ultimate *BaseDistances* list contains all possible base inputs selected or generated from the initial PCA-generated points. By selecting the *base* input with the shortest distance to e from the *BaseDistances*, we can predict a target program's failure distribution without performing any injection setups for the input sequence e. In case the selected *base* input is generated from two PCA-generated points, for example *S* and *Q* in Figure 3.4, we need to also estimate the error classifications ($n_{c,i}$ and n_j) and metric values (x_j and x_i) of the new generated input *G* to be used in equation (3.6). This can simply be done using the assigned weights of *S* and *Q* along with their error classifications and metric values. Therefore, instead of using equation (3.1) to predict the percentage of error classification c for input sequence e, equation (3.14) can be used in case the selected *base* input is generated from two PCA-generated points. The results of the instruction-based prediction are presented in chapter 5.

$$E_{c,e,G} = \sum_{i=1}^{n} P_{c,i,G} * W_{e,i,G} \tag{3.14}$$

Here $E_{c,e,G}$ is the predicted percentage of error classification c for input sequence e using the generated point *G*, and $P_{c,i,G}$ is the estimated percentage of instruction category *i* classified as c for the execution flow corresponding to the generated point *G*.

$$P_{c,i,G} = P_{c,i,S} * W_{S,e} + P_{c,i,Q} * W_{Q,e} \tag{3.15}$$

$$W_{e,i,G} = \frac{n_{e,i,G}}{n_e} \tag{3.16}$$

Here $P_{c,i,S}$ and $P_{c,i,Q}$ are the percentage of instruction category *i* classified as *c* for the *S* and *Q* execution flows, respectively, and $W_{S,e}$ and $W_{Q,e}$ can be calculated using equations (3.12) and (3.13). Moreover, in equation (3.16), $n_{e,i,G}$ and n_e refer to the estimated number of experiments with injection in instruction category *i*, and the estimated total number of experiments for the *e* execution flow, respectively. They can be calculated using equations (3.17) and (3.18).

$$n_{e,i,G} = x_{i,e} * \frac{n_{i,G}}{x_{i,G}} \tag{3.17}$$

$$n_e = \sum_{j=1}^{n} x_{j,e} * \frac{n_{j,G}}{x_{j,G}} \tag{3.18}$$

Here $x_{i,e}$ and $x_{i,G}$ are the observed and estimated metric values (*NLI, NSI, NAI, NBI, NLGI, and NPI*) for the *e* execution flow and the execution flow corresponding to the generated point *G*, respectively, and $n_{i,G}$ is the estimated number of experiments with injection in instruction category *i* for the execution flow corresponding to the generated point *G*. $x_{i,G}$ and $n_{i,G}$ are calculated in equations (3.19) and (3.20).

$$x_{i,G} = M_{i,S} * W_{S,e} + M_{i,Q} * W_{Q,e} \tag{3.19}$$

$$n_{i,G} = n_{i,S} * W_{S,e} + n_{i,Q} * W_{Q,e} \tag{3.20}$$

Here $M_{i,S}$ and $M_{i,Q}$ are the metric values (*NLI*, *NSI*, *NAI*, *NBI*, *NLGI*, *and NPI*) of the *S* and *Q* execution flows, respectively. Moreover, $n_{i,S}$ and $n_{i,Q}$ refer to the total number of experiments with injection in instruction category *i* for the *S* and *Q* execution flows, respectively.

4 Implementation

4.1 Target Programs

In this section, we present the two target programs used in the fault injection setups; *secure hash algorithm (SHA-1)* and *cyclic redundancy check (CRC-32)*. SHA-1 is a cryptographic hash function which generates a 160-bit message digest. It is used in many security protocols and applications such as SSL, TLS, SSH and IPsec. CRC-32 on the other hand, is a popular error-detecting code mostly used in networks to detect undesirable changes to data. Both SHA-1 and CRC-32 take as an input a string of arbitrary length. Even though the implementation of our both workloads can be found in the MiBench suite [36], we only take CRC from this suite. The MiBench implementation of SHA uses dynamic memory allocation which is not necessary for an embedded system. Thus, we adopt another implementation of SHA[1]. The choice of these two target programs is due to the following reasons:

- Their acceptance as representative applications for characterizing instruction set architectures [37].
- They work on the same input type (a string of chars). Therefore we might investigate whether there is a correlation between the failure distribution, the applications, and the input e.g., the length of the input.
- They have different structures, in particular with regards to; *lines of source code (LOC)*, *max Cyclomatic complexity (MCC)*, and size of the program, see Table 4.1. Moreover, CRC-32 has a

[1] http://www.dil.univ-mrs.fr/~morin/DIL/tp-crypto/sha1-c

higher percentage of *branch* instructions when compared to SHA-1, while SHA-1 has a higher percentage of *arithmetic* instructions. Thus, CRC-32 can be considered as control flow intensive, while SHA-1 is arithmetic intensive.

Table 4.1. The structure of the two target programs

	Lines Of source Code (LOC)	**Max Cyclomatic Complexity (MCC)**	**Size (bytes)**
CRC-32	16	2	2560
SHA-1	125	7	13340

Throughout the rest of this thesis, these two target programs are used in our fault injection setups (campaigns) and fault prediction. Moreover, they will be equipped with a proper software implemented fault tolerant to get validated on the effectiveness of the implemented fault tolerance mechanism. For simplicity throughout the remaining of this thesis, we use SHA and CRC instead of SHA-1 and CRC-32.

4.1.1 SHA and CRC Input Sets

Nine different inputs are selected for each workload. In order to be able to evaluate the behavior of each execution flow more accurately, the chosen set of inputs should be a proper representative of real applications. The input set is composed of sequences of alphanumerical characters generated randomly with a priori knowledge of input length. On the basis of the length of the inputs, we group them into the following three categories:

- Small inputs, with length between 0 to 2 characters.
- Medium inputs, with length between 3 to 60 characters.
- Large inputs, their length between 61 to 99 characters.

The three categories are chosen to represent input lengths that are common in real applications. Indeed, small inputs denote data produced by sensors to which is applied the CRC, medium inputs are representative of passwords hashed by SHA and large inputs are packets or messages to which have been appended a checksum or a digest as in TCP/IP networks. The input sets of the execution flows are shown in Table 4.2.

Table 4.2. The input set for different execution flows

Workload	Category	Number of inputs per category	Input length	Input name
CRC	Small inputs	3	One input with length 0	CRC-1
			One input with length 1	CRC-2
			One input with length 2	CRC-3
	Medium inputs	4	Two inputs with length 10	CRC-4 & CRC-5
			Two inputs with length 46	CRC-6 & CRC-7
	Large inputs	2	Two inputs with length 99	CRC-8 & CRC-9
SHA	small inputs	3	One input with length 0	SHA-1
			One input with length 1	SHA-2
			One input with length 2	SHA-3
	Medium inputs	4	Two inputs with length 10	SHA-4 & SHA-5
			Two inputs with length 60	SHA-6 & SHA-7
	Large inputs	2	Two inputs with length 99	SHA-8 & SHA-9

Since each input corresponds to an execution flow, the input name signifies both the workload and its input. For each input we perform two different campaigns; with and without software implemented fault tolerance. The result of the fault injection campaigns will be presented in chapter 5.

4.1.2 Triple Time-Redundant Implementation with Forward Recovery (TTR-FR)

TTR-FR [38] is implemented for both SHA and CRC to be used in our second set of campaigns. In this fault tolerance mechanism, each execution flow is being executed three times and the result of each run of the program is compared with the other runs. In case all runs of the program produces the same output, the injected fault has either been masked or had no impact on the generated sub-outputs of the execution flow; whereas different sub-outputs trigger the software-implemented voter to decide on the output of the execution flow. If only one run of the program generates a different output, output of the other two runs will be elected; this technique is known as forward recovery since the state of the faulty run moves forward to a fault-free point. Moreover, an unrecoverable error is signaled, if none of the program runs generate the same output.

```
int main(void){
    crc32file(input_1, &Output_1, &input_1_length));
    crc32file(input_2, &Output_2, &input_2_length));
    crc32file(input_3, &Output_3, &input_3_length));
    if(Output_1 == Output_2){
      Output = Output_1;
      if(Output_1 != Output_3){
         errorOccured = 85;
         //Recover of Output-3 is done here.
      }
    }
    else{
      errorOccured = 85;
      if(Output_1 != Output_3){
         if(Output_2 != Output_3){
            errorOccured = 195;
         }
         else{//Recover Output-1 is done here.
         }
      }
      else{//Recover Output-2 is done here.
      }
      if(errorOccured != 195){
         Output = Output_3;
      }
    }
}
```

Figure 4.1. TTR-FR implementation of CRC

Figure 4.1 shows the TTR-FR implementation of the CRC target program. It is shown that *crc32file*, the function responsible for calculating CRC output, has been called three times each with a fresh copy of the input variable. The outputs of each program run are placed in *Output-1*, *Output-2* and *Output-3* variables, while the *Output* variable holds the elected value. The other important variable used in the TTR-FR implementation is *errorOccured* which is used to classify the effect of the injected fault in the system; as an example, in case *errorOccured* is equal to 85 at the end of a program run, the injected fault is classified as corrected by software (more on error classification in Section 4.2). *ErrorOccured* can have three different values of (0, 85, and 195). This is due to the fact that we use a single-bit-flip fault model and there is a four-bit difference between each pair of *errorOccured* values, which makes a more robust fault tolerant implementation.

4.2 Experimental Set-up

As mentioned in chapter 2.2, Goofi-2 fault injection tool is used to evaluate the results of our instruction-based prediction technique. In particular, the transient faults are emulated by single bit-flips in Goofi-2 which is the fault model of the injections. In this thesis, as illustrated in Figure 4.2, Nexus-based fault injection technique is used with the support of iSYSTEM iC3000 Active Emulator [39]. The workloads are executed on the MPC565 microcontroller which is placed on a single-board computer phyCORE-MPC565 [40]. The program which is written in C programming language is compiled with gcc 4.2.2 compiler without any optimizations, and then it is downloaded to the board using WinIDEA [41]; an integrated development environment. Thus, in the first step, the program should be compiled, linked and executed in the WinIDEA interface to make sure it is correctly engaged on the board and ready to run the fault injection campaigns.

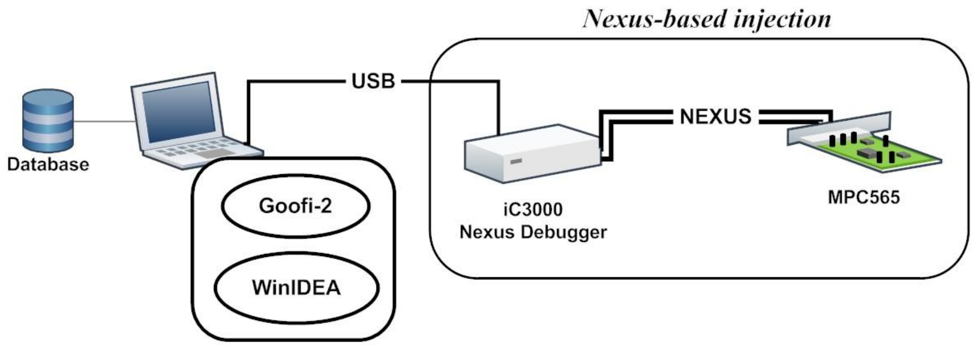


Figure 4.2. Experimental setup

Now, the fault injection campaign can be set up using Goofi-2 graphical user interface. Each campaign represents a set of fault injection *experiments* with a given execution flow. The injections are performed on registers and volatile memory words. Table 4.3 shows the used target registers and memory sections. The floating point registers are not among the target registers since they are not used in our target programs. Furthermore, since each workload's machine code is located in the nonvolatile memory, it is excluded from the fault space. The fault injection space is then optimized by a pre-injection analysis [42] which reduces the fault space by excluding all redundant faults that have the same effect on the system and also the faults which are not going to be activated by the program execution. In addition, the fault injection is performed just before the target register or memory location is read by the execution flow.

Table 4.3. Fault injection space

Target registers	Target memory sections
General Purpose Registers(GPR)	Stack
Program Counter Register (PCR)	Data
Link Register (LR)	Sdata
Condition Register (CR)	Bss
Integer Exception Register (XER)	Sbss

Since fault injection is a time-consuming process, an exhaustive fault injection is not feasible for the 36 different campaigns that we are supposed to run, 18 for each target program (9 campaigns for when there is no fault tolerance mechanism and 9 campaigns for when there is TTR-FR). Therefore, in this thesis, each SHA and CRC campaign runs 25,000 and 12,000 injection experiments, respectively, with randomly selected fault locations, i.e., for each experiment, one fault location and one of its bits are randomly chosen. The results of one performed exhaustive injection have approved the proper choice of randomly chosen experiments. The next step is to set-up a *timeout* value for each campaign; this is defined according to the execution time of the target program to help Goofi-2 detect experiments that are stuck, e.g., in an infinite loop. The timeout is pessimistically selected to be 5 seconds for all our campaigns. This is due to the fact that the execution time of the fault-free program with/without TTR-FR is one order of magnitude less than this timeout.

Finally when the setup configurations are completed and the campaign starts running, Goofi-2 performs a single fault-free run of the program, called the *golden run*, to store the measured results as reference data. The measured reference data will later on be compared to the results of the fault injection experiments to classify the outcome of every experiment. All measurements including the content of registers and memory locations will be stored in a database which is explained in the next section.

4.2.1 Goofi Database

Goofi-2 provides an optimized internal relational database to store data efficiently. A simplified representation of this database with tables' relations is shown in Figure 4.3. The campaign's configuration given in the interface of Goofi-2 is stored in the *campaign* table. Fault locations and injection space boundaries help Goofi-2 generate the elements of *faultlist* table. Furthermore, the faultlist table is connected to *fault* and *registerinfo* tables which store more detailed information about the fault injection locations and outcomes.

The fault injection outcomes and measurements are stored in the *experiment* table which apart from the fault table, is connected to *loggedmemorydata*, *loggedregisterdata*, and *pctrace*. The values of registers and memory locations are stored in loggedregisterdata and loggedmemorydata, respectively for each experiment. Indeed, more detailed measurements of general purpose registers for each instruction execution are stored in *gprvector* table. On the other hand, the execution flow of the workload can be extracted from the pctrace table which includes program counter register values each indicating the address of an instruction.

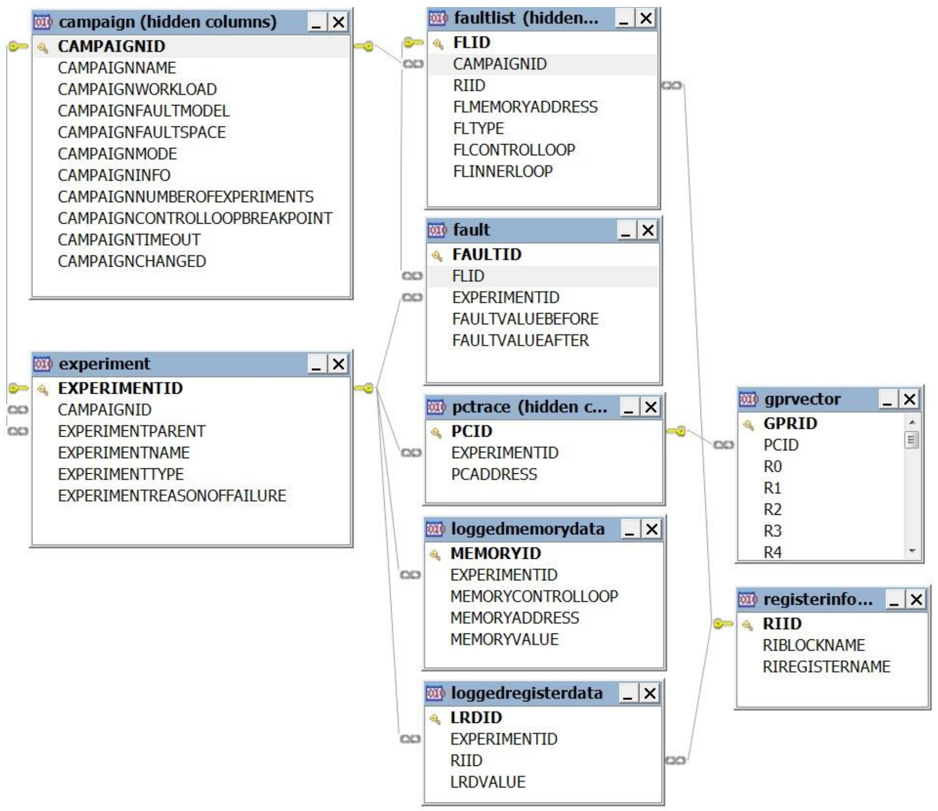


Figure 4.3. Goofi-2 Database schema

4.3 Assembly-Level Signature Creator (ASC)

ASC is an application written in this thesis for analyzing the assembly code of the workloads and calculate the metric values explained in Section 3.2 in order to be able to create an assembly-level signature for each execution flow.

In its first step, ASC parse the assembly code of a workload to make distinction between instructions' *address*, *opcode*, and *operands*. The address corresponds to where the instruction is stored and is represented in hexadecimal format. The opcode is an assembly instruction from one of the instruction categories explained in Section 3.2.1, and operands are where the data is placed, i.e., different registers or memory locations. A part of an assembly code is shown in Figure 4.4.

```
                                Add       r5,r5,r4              /* correct to real pointer */
200c:      7c a5 22 14          Add       r5,r5,r4
                                Lwz       r4,.Ltable(r5)        /* get linker's idea of where .Laddr is */
2010:      80 85 80 00          Lwz       r4,-32768(r5)
```

Figure 4.4. A sample piece of an assembly code

In order to analyze an execution flow, program counter register values should be traced for the golden run. This is due to the fact that program's signature is needed on the fault-free run of the program and the program counter register at any time contains the address of the executing instruction. Following program counter register values helps us find the execution flow of the program, e.g., different input sequences might lead to different number of while loop executions. Program counter register value updates of the golden run can be followed using Goofi-2 database specifically the pctrace table as illustrated in Figure 4.3. N.B. ASC does not use any data corresponding to the fault injection experiments and the only reason for using Goofi-2's database is to retrieve the program counter register values of the golden run. There are other utilities which can be used for this purpose even though working with Goofi-2 pctrace table is more convenient. The next step is to calculate metric values.

```
201c:          80 e5 80 10      Lwz          r7,-32752(r5)
```

Figure 4.5. Example of a load instruction

The instruction metrics which were explained in Section 3.2.1 can now be extracted using the pctrace, instruction's opcodes, and the "category file" corresponding to the instruction categories. A part of this file contains the different instructions existed in the 6 defined instruction categories, see Table 3.1.

To calculate the values of the registers metrics, additional information is needed to decide on how the registers are accessed, i.e., in the write mode or in the read mode. This is due to the fact that the operands of an instruction are accessed differently according to the instruction's opcode. For example, *add r4,r7* results in two accesses in the read mode and one access in the write mode to general purpose registers.

A list of opcodes with their operands access modes is provided as Assembly-level definitions for the MPC565 (PowerPC) in [43] which is also used in ASC with some modifications. Thus, the values of the register metrics are obtained by the following steps:

- Using the pctrace table to find the current instruction in the execution flow.
- Placing the found instruction's opcode into its instruction category.
- Deciding on the access mode of instruction's operands based on the instruction's opcode
- Calculating the values of the number of access and distance metrics.

N.B. even though *r1* is among the general purpose registers, it is analyzed in the metrics corresponding to the stack pointer.

Since the memory metrics investigate the access to different memory sections in an execution flow, ASC needs to find out the memory addresses which are accessed during the execution of the program. The only instruction categories that access the memory are load and store which read/write form/to the memory, respectively. To find target memory locations of the instructions in these categories, operands of these instructions should be explored. For instance, consider the instruction in Figure 4.5 which loads from memory location -*32752+(r5)* to register *r7*. Thus, the value of *r5* should be added to the constant value -32752 to calculate the target memory location.

This means that, sometimes the content of the general purpose registers are required to be able to calculate accessed memory locations. Therefore, ASC makes use of another Goofi-2 table called gprvector. This table holds values of general purpose registers for the Golden run of each execution flow. The next step is to find the memory section of the calculated memory location. This can be done by parsing the "memory section file" extracted for each execution flow using objdump utility. Memory section names and boundaries are placed in this file; therefore the calculated memory locations can be looked up in this file to be able to decide on the accessed memory section. N.B. since stack section is allocated dynamically, only its start address can be calculated using the “memory section file”. Its end address is extracted automatically by ASC from the “linker file” of the workload.

To conclude, the following files are parsed by ASC in order to calculate the values of different metrics of each execution flow:

- The categories
- The assembly-level definitions for the MPC565 (PowerPC)
- The assembly code
- The linker
- The memory sections

It should be mentioned that the “categories file” consists of instruction categories, register categories, and memory sections. Ultimately, ASC provides the values of all metrics in an output file to be used as the signature of each execution flow.

5 Result

5.1 Experimental Results for Workloads without Software implemented Fault Tolerance

5.1.1 Metrics Results

Table 5.1 shows a summary of the assembly metrics that are defined in Table 3.2. Metrics in this table consist of the percentages of different executed instruction categories, the number of read from general purpose registers, and the number of read from stack pointer register. Throughout this section, the values of these metrics are used to analyze the results in more details.

5.1.2 Experimental Results

In this section we describe the outcomes of fault injection campaigns conducted on the two workloads, CRC and SHA. For each SHA and CRC execution flow we inject 25,000 and 12,000 faults, respectively, hence we have 9 fault injection campaigns for each workload that result in 225,000 and 108,000 of injected faults. The error classification scheme of each injection experiment is as follows:

- No Impact (NI), errors that do not affect the output of the execution flow.
- Detected by Hardware (DHW), errors that are detected by the hardware exceptions.
- Time Out (TO), errors that cause violation of the timeout[2].

[2] Timeout is an order of magnitude larger than the worst-case execution time of a workload.

- Value Failure (VF), erroneous output with no indication of failure (silent failure).
- Detected by Software (DSW), errors that are detected by the software detection mechanisms.
- Corrected by Software (CSW), errors that are corrected by the software correction mechanisms.

When presenting the results, we also define coverage as the probability that an error does not cause value failure, thus the coverage (COV) of each execution flow can be calculated using equation (5.1) :

$$COV = 1 - \#VF/N \tag{5.1}$$

Where N is the total number of experiments, and #VF is the total number of experiments that resulted in value failure. This equation also includes experiments classified as no impact and timeout. No impact experiments can be regarded as internal robustness of the workload; therefore they contribute to the overall coverage of the system. Timeout experiments on the other hand, are detected Goofi-2. In a real life application where there is no fault injection tool to be used for timeout detection, watchdog timers are used to detect these errors.

Table 5.1. Summary of assembly metrics for CRC and SHA execution flows

Execution Flow	PLI	PSI	PAI	PBI	PLGI	PPI	NRGPR	NRSP
SHA-1	34.99%	11.29%	30.68%	3.18%	19.41%	0.44%	10650	63
SHA-2	35.07%	11.29%	30.56%	3.24%	19.40%	0.44%	10697	63
SHA-3	35.12%	11.29%	30.48%	3.28%	19.39%	0.44%	10730	63
SHA-4	35.53%	11.34%	29.96%	3.48%	19.26%	0.43%	10992	63
SHA-5	35.53%	11.34%	29.96%	3.48%	19.26%	0.43%	10992	63
SHA-6	36.57%	11.35%	29.05%	3.80%	19.02%	0.21%	23027	69
SHA-7	36.57%	11.35%	29.05%	3.80%	19.02%	0.21%	23027	69
SHA-8	37.38%	11.46%	27.99%	4.21%	18.77%	0.20%	24302	69
SHA-9	37.39%	11.46%	27.98%	4.21%	18.76%	0.20%	24334	69
AVERAGE	36.02%	11.35%	29.52%	3.63%	19.14%	0.33%	16527.89	65.66
CRC-1	33.93%	26.79%	7.14%	10.71%	5.36%	16.07%	51	14
CRC-2	37.21%	20.93%	9.30%	8.14%	12.79%	11.63%	87	14
CRC-3	38.79%	18.10%	10.34%	6.90%	16.38%	9.48%	123	14
CRC-4	41.85%	12.64%	12.36%	4.49%	23.31%	5.34%	411	14
CRC-5	41.85%	12.64%	12.36%	4.49%	23.31%	5.34%	411	14
CRC-6	42.97%	10.65%	13.09%	3.62%	25.84%	3.83%	1707	14
CRC-7	42.97%	10.65%	13.09%	3.62%	25.84%	3.83%	1707	14
CRC-8	43.16%	10.31%	13.22%	3.47%	26.27%	3.57%	3615	14
CRC-9	43.16%	10.31%	13.22%	3.47%	26.27%	3.57%	3615	14
AVERAGE	41.08%	14.78%	11.92%	5.44%	20.60%	6.96%	1303	14

5.1.2.1 Workloads Comparison

We investigate the experiments outcomes with respect to both the fault locations (e.g., general purpose registers) and the instructions being executed when the fault is injected. Table 5.2 summarizes the results of the fault injection campaigns for the two workloads. In this table, the *ERRORS* column shows to the sum of faults for the nine campaigns corresponding to the mentioned fault locations. Besides, throughout the remaining of the thesis, we refer to the Link register, Condition register, and Integer Exception register as *Miscellaneous registers (Misc)*.

Table 5.2. Summary of CRC (*left table*) and SHA (*right table*) failure distributions

Fault Location	ERRORS	NI	VF	DHW	TO
	#		%		
GPRs	48,163	27.45	39.66	30.95	1.93
PCR	42,371	13.58	11.46	74.58	0.38
MISC	3,981	61.47	28.49	9.62	0.43
MEM	13,485	28.35	63.65	7.69	0.31
Total	108,000	23.38	31.18	44.38	1.06

Fault Location	ERRORS	NI	VF	DHW	TO
	#		%		
GPRs	105,032	13.86	55.82	28.04	2.28
PCR	85,271	13.73	16.86	68.87	0.54
MISC	5,765	63.14	35.30	1.46	0.10
MEM	28,932	10.10	84.17	3.94	1.80
Total	225,000	14.59	44.18	39.73	1.50

After comparing the outcomes of the two workloads, a number of major findings are summarized as follows:

- A high percentage of the errors in the program counter register are detected by hardware exceptions for both CRC and SHA, these results are in accord with [14]. This is due to the fact that an error in the program counter register is more likely to trigger a hardware exception like data violation or execution of a not implemented instruction.
- The percentage of value failures caused by fault injections in Misc registers is higher for SHA than for CRC. The content of these registers are changed mainly by the arithmetic instructions which are executed significantly more in SHA than in CRC according to Table 5.1.
- Indeed, fault injections in GPRs and memory locations cause a higher percentage of experiments to be classified as no impact in CRC compared to SHA. However in SHA, higher percentage of injections in GPRs and memory locations are classified as value failure compared with CRC. One reason for this behavior can be explained by analyzing the two case studies in Figure 5.1. In case 1, a piece of CRC disassembly code, shows that load instructions which fetch the memory contents, are followed by rotate, shift or compare instructions. These orders of instructions along with 0-bit sequences in the memory would make the bit flipping ineffective due to the fact that the memory content is changed immediately. Since there are a great number of experiments with the above mentioned behaviour in CRC, the results of the fault injections in the memory would not be effective in 28% of the times, e.g., 30% of the experiments for the load instruction at address 21f4, and 26% of experiments at address 219c resulted in no impact. On the other hand, the faults injected into the memory for SHA, as shown in the case 2 of Figure 5.1, turn out to be more effective. This might be due to the fact that the content of the memory is not always immediately consumed by rotate or compare instructions, e.g., the fetched memory content is stored in the memory for further usages. Therefore there is a higher probability that the injected fault causes value failure. In addition, SHA is a bigger program compared to CRC and it executes load instructions more than CRC, thus SHA is more prone to value failure than CRC. The proposed metrics are not capable of fully capturing these effects. However, percentage of load instructions (PLI) (see Table 5.1) can be used to help us understand whether a workload is more sensitive than another one to faults injected in memory. Indeed, the percentage of load

instructions for CRC is greater than the percentage of load instructions of SHA, which shows that CRC workload is more likely to mask the effects of the fault injections.

```
Case 1
219c:   81 3f 00 14    lwz     r9,20(r31)
21a0:   55 20 06 3e    clrlwi  r0,r9,24
…
21f4:   80 1f 00 14    lwz     r0,20(r31)
21f8:   2f 80 00 00    cmpwi   cr7,r0,0
…
21a4:   81 7f 00 18    lwz     r11,24(r31)
21a8:   7c 00 5a 78    xor     r0,r0,r11
21ac:   54 00 06 3e    clrlwi  r0,r0,24…
```

```
Case 2
234c:   80 1f 00 10    lwz     r0,16(r31)
2350:   2f 80 00 00    cmpwi   cr7,r0,0
…
26e4:   80 1f 00 0c    lwz     r0,12(r31)
26e8:   90 1f 00 08    stw     r0,8(r31)
…
273c:   81 3f 00 10    lwz     r9,16(r31)
2740:   80 1f 00 0c    lwz     r0,12(r31)
2744:   7d 29 03 78    or      r9,r9,r0)
```

Figure 5.1. Case study of faults injected in memory for CRC (case 1) and SHA (case 2)

5.1.2.2 Execution Flows Comparison

Table 5.3 shows the experiment outcomes for different CRC and SHA execution flows. The percentage of experiments classified as value failure grows linearly as the length of the inputs increase. We can reasonably approximate the value failure to a *normal variable* due to the large number of experiments, 25000 and 12000 for each SHA and CRC execution flow. Moreover, the number of experiments is enough to give high confidence in the obtained results. The 95% confidence interval for the value failures presented in Table 5.3 varies from ±0.43% to 0.88%.

Table 5.3. CRC and SHA failure distributions for different execution flows

Execution Flow	Total	NI	VF	DHW	TO	**COV**
	#		%			
CRC-1	12000	42.72	6.06	48.22	3.01	**93.94**
CRC-2	12000	32.94	17.94	46.73	2.38	**82.06**
CRC-3	12000	28.3	24.31	45.83	1.56	**75.69**
CRC-4	12000	20.81	34.32	44.03	0.84	**65.68**
CRC-5	12000	20.33	35.5	43.61	0.57	**64.5**
CRC-6	12000	16.55	39.79	43.41	0.25	**60.21**
CRC-7	12000	17.06	39.59	43.05	0.3	**60.41**
CRC-8	12000	16.02	41.92	41.77	0.29	**58.08**
CRC-9	12000	15.68	41.19	42.75	0.38	**58.81**
SHA-1	25000	18.86	38.76	40.98	1.4	**61.24**
SHA-2	25000	17.76	40.1	40.99	1.14	**59.9**
SHA-3	25000	17.58	40.82	40.6	1	**59.18**
SHA-4	25000	15.94	43.13	39.37	1.56	**56.87**
SHA-5	25000	16.81	42.05	39.7	1.44	**57.95**
SHA-6	25000	11.53	47.06	39.5	1.9	**52.94**
SHA-7	25000	11.43	47.72	39.26	1.58	**52.28**
SHA-8	25000	10.72	48.81	38.78	1.68	**51.19**
SHA-9	25000	10.68	49.13	38.4	1.78	**50.87**

If we consider that the value failure is distributed as a normal variable with a *mean* value equals to the quote between the number of value failure experiments and total number of experiments, we can conduct one way *analysis of variance (ANOVA)* to inspect if there is a linear correlation between the length of the input and the percentage of value failure. ANOVA is performed by testing the hypothesis H0 which states "*there is no linear correlation between the length of the input and the percentage of value failure*". The results of ANOVA in Table 5.4 allow us to reject H0 with a confidence of 95%. The reason behind this correlation is that when the length of the input increases, the number of reads from registers and memory locations increase as well. This means that there are more possibilities to inject faults that result in value failure.

Table 5.4. Hypothesis test results for CRC and SHA

Hypothesis	Workload	p-value (α =0.05)	Outcome	Linear regression Equation
NO linear correlation between VF and input length.	CRC	0.0309	Reject H0	VF = 23.55 + 0.22length
	SHA	<0.001	Reject H0	VF = 40.64 + 0.09length
NO linear correlation between DHW and input length.	CRC	0.013	Reject H0	DHW = 45.78 - 0.040length
	SHA	0.034	Reject H0	DHW = 40.46 -0.019length
NO linear correlation between TO and input length.	CRC	0.046	Reject H0	TO = 1.67 - 0.017length
	SHA	0.37	Accept H0	--
NO linear correlation between COV and input length	CRC	0.0309	Reject H0	COV = 76.45 - 0.22length
	SHA	<0.001	Reject H0	COV = 59.35-0.09length

The percentage of detected by hardware experiments, as shown in Table 5.3, is always around 44 and 40 percent for the CRC and SHA execution flows, respectively. This means that there may be only a minor correlation between the percentage of detected by hardware experiments and the input length. The outcome of the hypothesis test, H0 that states "*there is no linear correlation between the results of detected by hardware experiments and the input length*" is presented in Table 5.4. Though the test reveals that for both workloads this hypothesis is rejected, the probability of errors being detected by hardware slightly decreases as the input length increases (the coefficient is -0.019 for SHA and 0.04 for CRC). Analogously, the proportion of experiments classified as timeout is almost constant for all the workloads.

Table 5.4 shows that COV is decreased as the input length increases for both CRC and SHA. This is acceptable because the percentage of value failures increases with the length of the input. The test of hypothesis in Table 5.4 reveals that coverage and value failures are linearly correlated and that the coverage decreases faster for CRC than SHA with the increase in the length of the input.

Table 5.5. CRC and SHA failure distributions w.r.t. the PCR

Execution Flow	Total	NI	VF	DHW	TO
	#	%			
CRC-1	4966	19.98	2.28	76.82	0.93
CRC-2	4811	15.9	7.3	75.81	1
CRC-3	4789	15.29	9.29	74.86	0.56
CRC-4	4742	13.05	12.13	74.42	0.4
CRC-5	4726	13.06	13.25	73.44	0.25
CRC-6	4572	11.59	14.39	73.99	0.02
CRC-7	4562	11.14	14.64	74.11	0.11
CRC-8	4578	10.66	15.4	73.9	0.04
CRC-9	4625	10.9	15.42	73.66	0.02

Execution Flow	Total	NI	VF	DHW	TO
	#	%			
SHA-1	9,533	15.05	15.26	69.26	0.42
SHA-2	9,544	14.75	16.05	68.65	0.54
SHA-3	9,473	14.56	15.92	68.78	0.74
SHA-4	9,496	13.84	16.85	68.66	0.65
SHA-5	9,516	13.69	16.6	69.21	0.49
SHA-6	9,375	12.83	17.91	68.64	0.62
SHA-7	9,494	13.46	17.73	68.32	0.5
SHA-8	9,404	12.8	17.71	69.01	0.48
SHA-9	9,436	12.57	17.78	69.27	0.38

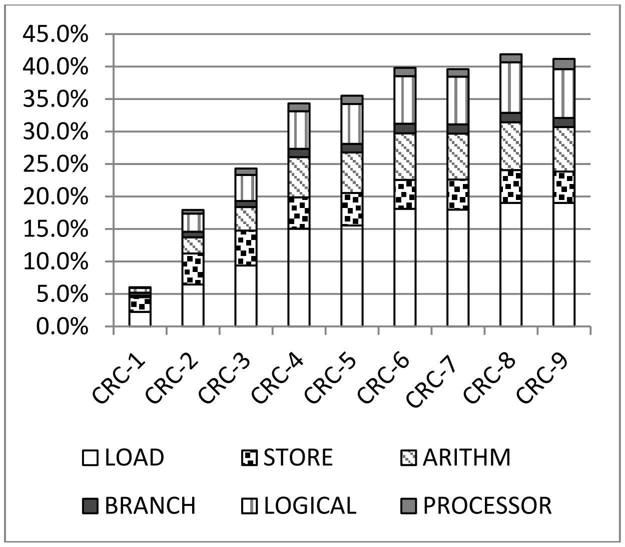
Table 5.5 shows that a high percentage of the errors in the program counter register are detected by hardware exceptions for both CRC and SHA. This is due to the fact that an error in the program counter register is more likely to trigger a hardware exception. However, around half of the remaining injections in the program counter register cause value failures. This might be due to the fact that some faults modify the program counter register in a way that the execution flow jumps to an instruction where it is still possible to finish the program execution without being detected by the hardware exceptions.

5.1.2.3 Value Failure Distribution over the Instruction Categories

Table 5.6 shows the distribution of the value failures over different instruction categories for all the execution flows, i.e., the value in each cell corresponds to the percentage of value failures for injections in different specific instruction categories. The distribution of the value failures for both workloads is in accord with the results in Table 5.1. Indeed, the most executed instructions are to a great extent responsible for value failures as it is shown in Figure 5.2. For instance, the load, logical, store, and arithmetic categories are the most executed instructions in CRC, and they are responsible for a significant part of value failures as well.

Table 5.6. Value Failure distribution over the instruction categories for CRC and SHA

	LOAD	STORE	ARITHM	BRANCH	LOGICAL	PROCESSOR
			%			
CRC-1	2.24	2.28	0.19	0.50	0.71	0.13
CRC-2	6.46	4.78	2.48	0.87	2.79	0.56
CRC-3	9.38	5.39	3.60	0.95	4.00	0.99
CRC-4	15.06	4.83	6.18	1.27	5.78	1.21
CRC-5	15.54	5.01	6.22	1.34	6.12	1.28
CRC-6	18.10	4.46	7.17	1.48	7.31	1.28
CRC-7	17.98	4.64	7.07	1.42	7.33	1.17
CRC-8	19.03	5.04	7.36	1.46	7.78	1.24
CRC-9	19.03	4.82	6.84	1.41	7.51	1.58
SHA-1	14.84	5.02	11.48	0.57	6.80	0.04
SHA-2	15.22	5.06	11.98	0.76	7.02	0.06
SHA-3	15.79	5.28	11.98	0.79	6.93	0.06
SHA-4	15.22	5.06	11.98	0.76	7.02	0.06
SHA-5	16.74	5.42	11.94	0.94	6.96	0.06
SHA-6	18.56	6.46	13.36	1.43	7.22	0.03
SHA-7	19.22	6.51	13.55	1.43	7.00	0.01
SHA-8	19.78	6.67	13.80	1.57	6.96	0.03
SHA-9	20.04	6.96	13.49	1.59	7.02	0.02



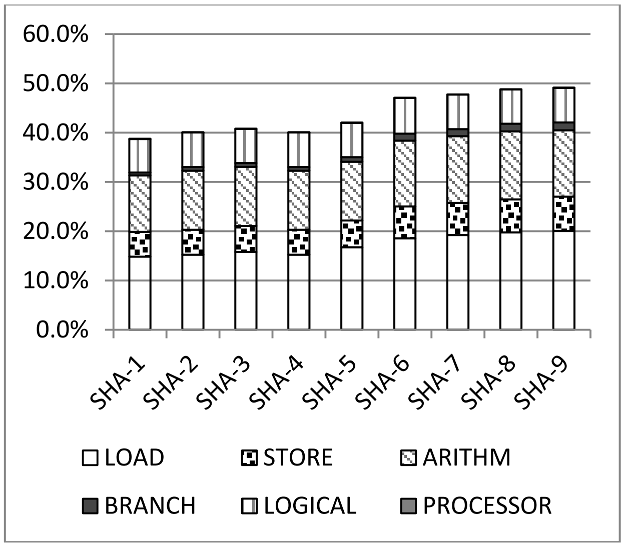


Figure 5.2. Value Failure distributions over different instruction categories for CRC (*left chart*) and SHA (*right chart*) execution flows.

5.2 Experimental Results for Workloads Equipped with TTR-FR

All the analyzed non-TTR-FR workloads consist of three major code blocks; startup, main function, and core function. In addition to these code blocks, the TTR-FR implementation also consists of the voter in its main function which performs the majority voting, see Figure 5.3. The core function, which is called three times from the main function, performs the foremost functionality of each workload. As an example, in CRC, the core function is responsible for the checksum calculations.

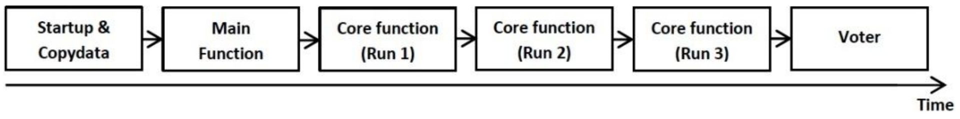


Figure 5.3. The TTR-FR code blocks

5.2.1 Workloads Comparison

This section summarizes the results of the fault injection campaigns for the workloads equipped with the TTR-FR mechanism. For simplicity, we refer to the two workloads as CRC-TTR and SHA-TTR. Table 5.7 shows the aggregated results for CRC-TTR and SHA-TTR. In this table, DSW is the percentage of errors detected by software, which is when all three runs of the program produce different outputs. Moreover, CSW is when an execution flow generates a correct output as a result of the majority voting.

Table 5.7. Summary of CRC-TTR (*left table*) and SHA-TTR (*right table*) failure distributions

Fault Location	ERRORS	NI	VF	CSW	DSW	DHW	TO
	#	%					
GPRs	48186	23.80	0.58	44.18	0.11	30.30	1.02
PCR	41782	13.22	1.77	11.24	0.30	73.24	0.23
MISC	3583	53.00	0.50	37.68	0.03	8.43	0.36
MEM	14448	24.54	5.17	60.71	0.19	8.11	1.28
Total	107999	20.77	1.65	33.43	0.19	43.22	0.73

Fault Location	ERRORS	NI	VF	CSW	DSW	DHW	TO
	#	%					
GPRs	104800	14.01	0.36	54.57	0.01	28.58	2.48
PCR	85022	13.96	1.47	16.59	0.41	66.52	1.04
MISC	5922	65.25	0.24	33.18	0.02	1.22	0.10
MEM	29254	10.92	0.33	82.93	0.00	4.05	1.77
Total	224998	14.92	0.77	43.36	0.16	39.00	1.79

The main observations are summarized as follows:

- The TTR-FR mechanism seems to be very effective for both workloads, indeed the percentage of value failure is about 0.8% for SHA and 1.6% for CRC.
- The TTR-FR mechanism proves to be a good technique to mask errors in all the fault locations and in particular for the general purpose registers. It is notable that the percentage of corrected errors in the GPRs is 54% for SHA-TTR while it is 44% for CRC-TTR. These results are expected because transient faults are masked with a new run of the program as it is done In the TTR-FR. Out of the total number of around 225000 and 108000 injected faults for SHA and CRC workloads, few experiments (1729 experiments for CRC and 1786 experiments for SHA) resulted in value failure. The proportion of value failure varies for different code blocks:
    a. With respect to the core function, the main contributor to the lack of coverage is injections in the program counter register. These faults change the control flow in such a way that the voter is incorrectly executed or not executed at all. For instance, for the core function of SHA, around 96% of the value failures were caused by faults in the program counter register.
    b. Since other parts of the source code including the voter are not protected with the TTR-FR mechanism, the injections in different registers and memory words are more likely to contribute to the value failure. For this reason, we performed exhaustive fault injection (i.e., we inject all possible faults) in the voter to evaluate its robustness for each workload. It is notable that about 14% of the injections in the voter resulted in value failures. Therefore, even though TTR-FR mechanism decreases the percentage of value failure, the voter is one of the main contributors to the remained percentage of value failure.
- The percentage of TTR-FR experiments classified as corrected by software is approximately equal to the percentage of value failure in Table 5.2. This means that the TTR-FR can tolerate almost all the previous cases of injections classified as value failures, although it is not totally immune to faults that affect program’s voter, startup code, or control flow.
- The percentages of experiments classified as detected by hardware in CRC and SHA are analogous to their enhanced versions, CRC-TTR and SHA-TTR. The reason might be that, the enhanced version of the original workload can be considered as a program that executes the original workload three times, along with few more lines of codes, e.g. the voting function. The faults are then more likely to be injected during one of the three executions, since the new lines of codes run for much shorter time. Therefore the percentage of experiments classified as detected by hardware for the enhanced version of a workload is more likely to be similar to the original workload.
- Another reason for observing analogous percentages of experiments classified as detected by hardware for CRC and SHA might be the similar percentages of injected faults in program counter register and general purpose registers for the original and its enhanced versions, see Table 5.2 and Table 5.7. Since experiments classified as detected by hardware are mainly as a result of injected faults in program counter register and GPR, the outcomes of detected by hardware for the original workloads are similar to the enhanced versions. Indeed, if an injected fault is classified as detected by hardware in one of the three runs, execution of the program will be terminated before it can be detected by TTR-FR.

- The percentages of experiments classified as no impact do not change significantly between the original workloads and their enhanced versions.

5.2.2 Execution Flows Comparison

Table 5.8 illustrates the outcome distribution of different execution flows in CRC-TTR and SHA-TTR. It is notable that the percentage of experiments classified as corrected by software increases with the growth of the input length. We can test it in a similar way that we did for SHA and CRC execution flows by using hypothesis H0 that states "*there is no linear correlation between the percentage of errors corrected by software and the length of input*". The 95% confidence interval for the corrected by software experiments varies for about ± 0.6% for SHA-TTR and from ±0.56% to ± 0.88% for CRC-TTR. The results in Table 5.9 show that we can reject H0 for SHA, but accept it for CRC even though the p-value is 0.065. Moreover, the slope of the linear regression model for SHA value failures in Table 5.4 and the slope of the model for corrected by software experiments in Table 5.9 are fairly close. This is a further enforcement of the observation that the TTR-FR can successfully cope with the errors classified as value failures with the increase of the input length.

Table 5.8. CRC-TTR and SHA-TTR failure distributions for different execution flows

Execution Flow	Total	NI	VF	CSW	DSW	DHW	TO	COV
	#				%			
CRC-1	12000	38.87	0.89	10.81	0	47.96	1.48	**12000**
CRC-2	12000	26.3	2.27	24.33	0.19	45.97	0.94	**12000**
CRC-3	12000	22.65	2.35	29.34	0.34	44.67	0.65	**12000**
CRC-4	12000	17.73	1.77	37.46	0.12	42.36	0.57	**12000**
CRC-5	12000	17.97	1.84	36.89	0.12	42.67	0.51	**12000**
CRC-6	12000	15.62	1.56	40.43	0.15	41.9	0.34	**12000**
CRC-7	11999	16.09	1.66	40.45	0.15	41.44	0.22	**11999**
CRC-8	12000	15.84	1.2	40.9	0.38	40.78	0.9	**12000**
CRC-9	12000	15.89	1.35	40.27	0.29	41.23	0.97	**12000**

Execution Flow	Total	NI	VF	CSW	DSW	DHW	TO	COV
	#				%			
SHA-1	25000	18.91	0.73	38.42	0.16	39.82	1.96	**99.27**
SHA-2	25000	18.63	0.84	39.53	0.18	38.92	1.9	**99.16**
SHA-3	24999	18.28	0.81	39.79	0.18	39.18	1.76	**99.19**
SHA-4	24999	17.32	0.91	40.71	0.15	39.23	1.67	**99.09**
SHA-5	25000	17.74	0.83	40.3	0.16	39.58	1.4	**99.17**
SHA-6	25000	11.28	0.68	47.08	0.19	39	1.78	**99.32**
SHA-7	25000	11.07	0.77	47.44	0.18	38.84	1.71	**99.23**
SHA-8	25000	10.47	0.65	48.59	0.09	38.22	1.98	**99.35**
SHA-9	25000	10.6	0.69	48.41	0.13	38.24	1.93	**99.31**

Furthermore, we can investigate if there is a linear correlation between the value failures of the extended versions and the length of input. The limits of confidence interval of the experiments classified as value failures are ± 0.01% for SHA-TTR and ±0.27% for CRC-TTR. According to Table 5.9, such hypothesis still holds for SHA. However it is interesting to notice that value failures decreases with the length of input for SHA-TTR. A possible explanation for this behavior is that the relative size of the core function in SHA-TTR is bigger than CRC-TTR. By increasing the length of the input, the core function will be lengthier and more faults would be injected in it. These faults are less likely to cause value failure due to the redundancy in the core function. Moreover, the percentage of experiments classified as corrected by software increases with the increase of input length which causes the value failures to be decreased for SHA-TTR.

Table 5.9. Hypothesis test results for CRC-TTR and SHA-TTR

Hypothesis	**Workload**	**p-value** (**α =0.05**)	**Outcome**	**Linear regression Equation**
NO linear correlation between CSW and input length	CRC-TTR	0.065	Accept H0	--
	SHA-TTR	<0.001	Reject H0	CSW = 39.43 + 0.10length
NO linear correlation between VF and input length	CRC-TTR	0.1891	Accept H0	--
	SHA-TTR	0.015	Reject H0	VF = 0.0082 - 0.0016length
NO linear correlation between DHW and input length	CRC-TTR	0.022	Reject H0	DHW = 44,78 - 0.0448length
	SHA-TTR	0.030	Reject H0	DHW = 39.42 - 0.011length
NO linear correlation between COV and input length	CRC-TTR	0. 1891	Accept H0	--
	SHA-TTR	0.015	Reject H0	COV = 99.17 + 0.001length

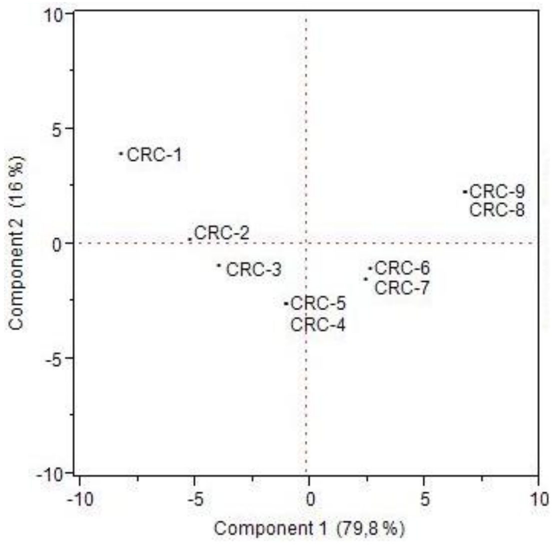
In both workloads there is a linear correlation between the percentage of experiments classified as detected by hardware and the length of input. This result is similar to the one obtained for non TTR-FR workloads.

The coverage, COV, defined in equation (5.1) is linearly correlated with the length of input for SHA, but it is not correlated for CRC. However, we can consider that the coverage is approximately constant, since the coefficient is very small for SHA and varies slightly in CRC.

5.3 Instruction-based Prediction Results for CRC and SHA Workloads

In this section, the instruction-based prediction technique is evaluated by investigating 9 different input sequences for each target program. The assembly level signatures of all fault-free execution flows are analyzed using PCA, discussed in Section 3.3.1. Moreover, the results of a number of failure distribution predictions are presented.

The instruction-based prediction technique, as discussed in Section 3.3, consists of two parts. In the first part, only one input sequence is selected as the base input, while in the second part, two input sequences are used to generate a better base input. Figure 5.4 illustrates the PCA analysis results of different execution flows on the metric values generated by the assembly signature creator (ASC). As can be seen, execution flows with same input lengths generate adjacent point in the Cartesian coordinate system. This observation will later on be used in the failure distribution predictions.



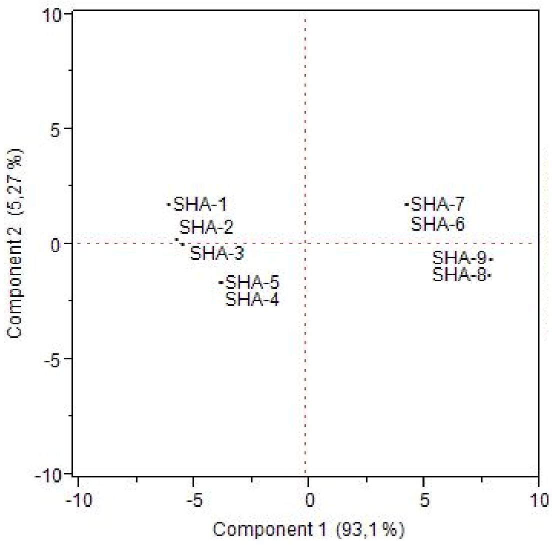


Figure 5.4. PCA-generated points for different CRC and SHA execution flows

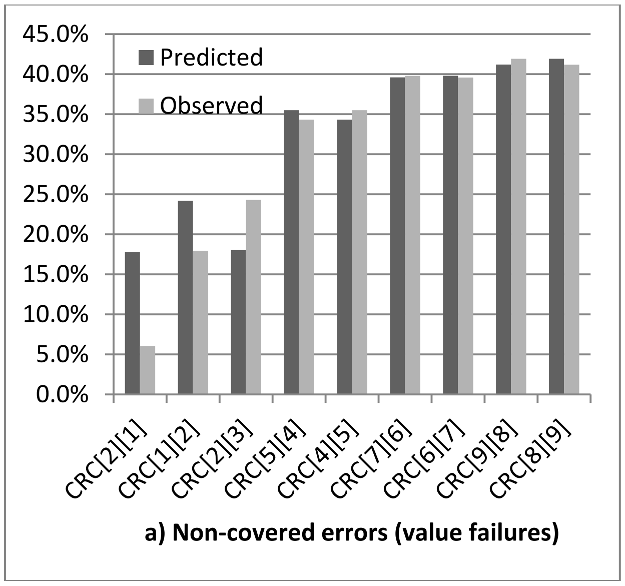
5.3.1 Using One Input Sequence as the Base

In order to predict the failure distribution of an execution flow, its closest point in the Cartesian coordinate system should be selected as its base input. Then by using the equation (3.6), the failure distribution of the target input can be predicted. As an example, as shown in Figure 5.4, in case we have already estimated the failure distribution of CRC-6 using fault injection, the failure distribution of CRC-7 can be predicted using CRC-6 as the base input. The same scenario is applicable to CRC-8 and CRC-5 in which CRC-9 and CRC-4, respectively, will be selected as the best base input sequences.

The results of using instruction-based prediction technique to predict non-covered errors (value failures) are shown in Figure 5.5. Subfigure *a* refers to the comparison of the predicted and observed value failures for CRC execution flows, while subfigure *b* corresponds to SHA. CRC shows higher fluctuation of value failure with regards to different input sequences.

It can also be seen in Figure 5.5 that each predicted value failure leans towards the value failure of its selected base input (except for when CRC-1 is used to predict CRC-2). That is, the predicted value failure is a value between the observed value failures of every two inputs. As an example, the higher percentage of predicted value failure for SHA-8 (compared to its observed value failure percentage) using SHA-9 is due to the fact that SHA-9 has a higher observed value failure when compared to SHA-8.

The weakness of the first part of the instruction-based prediction technique is that it only works fine when there is an adjacent point to the target point. For example, in case there was no point in Figure 5.4, corresponding to SHA-7, the selected base sequence for predicting the value failure of SHA-6 would be SHA-9. The value failure prediction of SHA-6 using SHA-9 as the base input shows the deviation of around 2% (between the predicted and observed values). It can also be seen in the subfigure *a* of Figure 5.5 that the results of the predictions for the first three CRC execution flows are not satisfactory. This is as a result of the high distance between the PCA generated base and target inputs, see Figure 5.4. Therefore, in the second part, two input sequences are used to generate a possibly better base input.



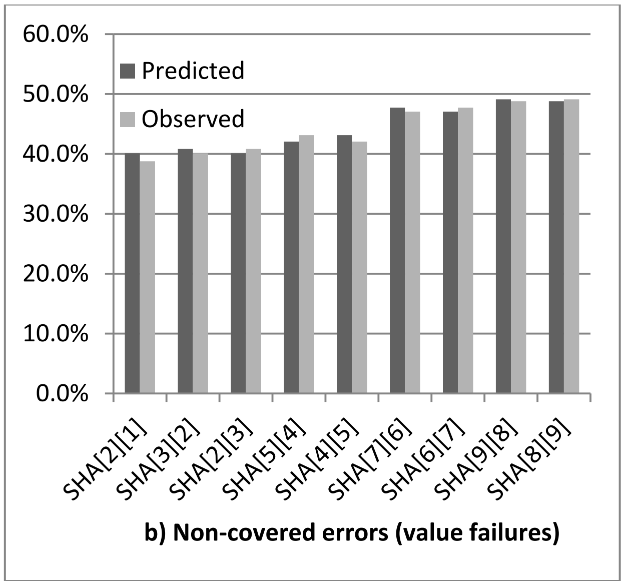


Figure 5.5. Predicted vs. observed non-covered errors (value failures) using one input as the base
N.B. the value inside the left square brackets corresponds to the base input while the right square bracket contain the target input.

5.3.2 Using Two Input Sequences to Generate the Base Input

In the second part of the instruction-based prediction technique, we use two input sequences to predict the failure distribution of another execution flow. The idea behind this mechanism is presented in Section 3.3.1 where all different combinations of two input sequences should be selected to generate another input which can also be represented in the PCA diagram. Finally, the generated input with the shortest distance to the target input is selected as the base input. It is then used in equation (3.14) to predict the failure distribution of another input sequence.

In order to evaluate the second part of the instruction-based technique using the presented execution flows, we assume that the closest base input is generated using two input sequences instead of one. For example, assume that in Figure 5.4, there is no point corresponding to CRC-7. Then the closest base input sequence to predict the failure distribution of CRC-6 is generated using CRC-8 and CRC-4. The reason for making this assumption is that for example CRC-7 is so close to CRC-6 that no other two input sequences can generate a better base input. Therefore, in order to evaluate the second part, we remove one input of each two adjacent points to let another two input sequences generate a better base input, e.g., when CRC-7 is removed, all combinations of the remained seven input sequences compete to generate a better base input for the prediction of CRC-6.

Figure 5.6 shows the results of applying instruction-based prediction technique using two input sequences to predict the percentage of experiments resulted in value failure, detected by hardware, and time out classifications. Only the input sequences of 3, 4, and 6 of each workload are predicted with the left column corresponding to CRC and the right column corresponding to SHA.

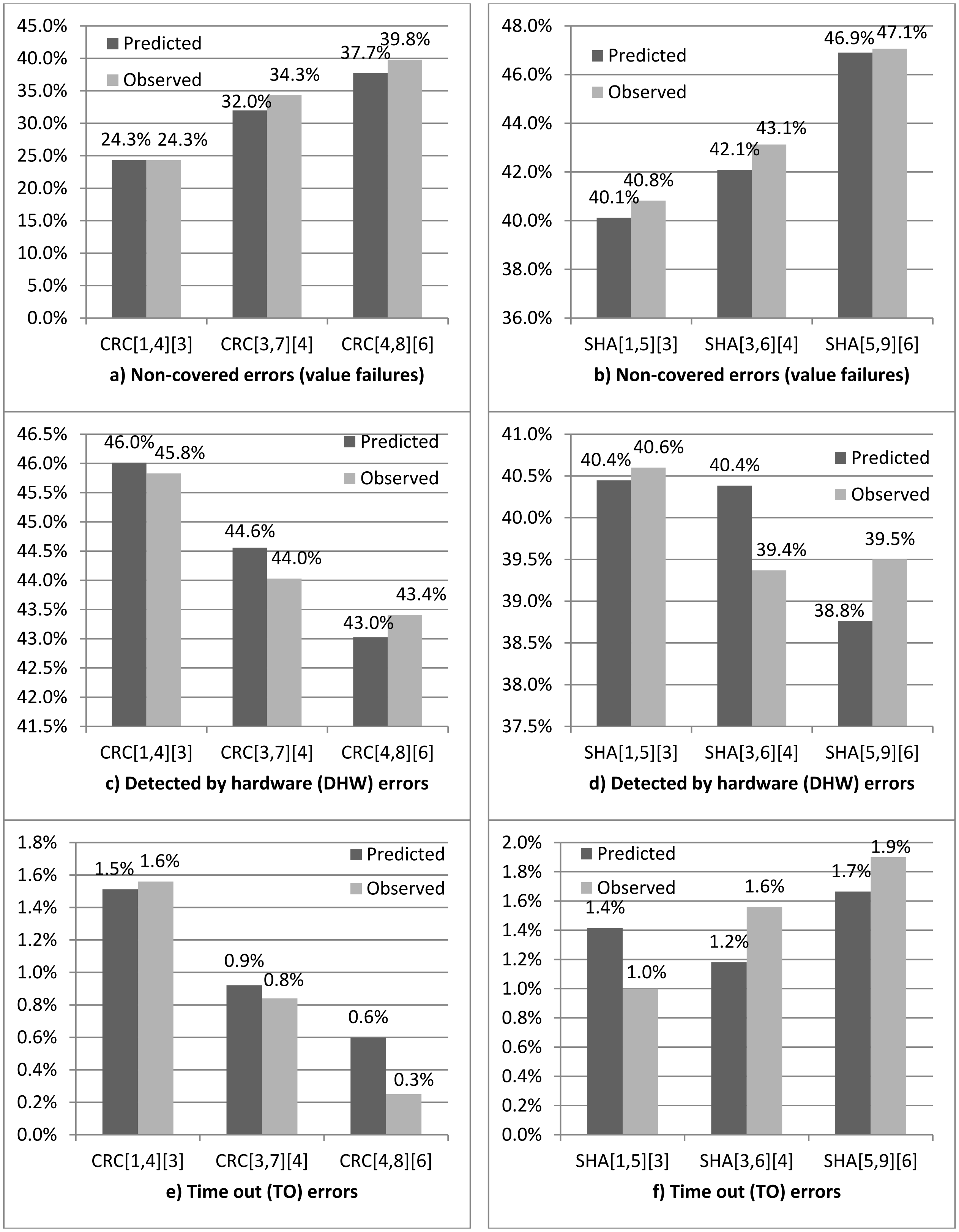


Figure 5.6. Predicted vs. observed failure distributions using two base inputs
N.B. the values inside the left square brackets corespond to the base inputs while the right square bracket contain the target input.

In the value failure predictions (subfigures *a* and *b*), the highest divergence from the observed percentage of value failure is 2.3 percentage points which corresponds to the case in which CRC-3 and CRC-7 are used to predict the value failure of CRC-4. This divergence is 24.5 percentage points in case of using only CRC-3 as the base input which means that the input generated by CRC-3 and CRC-7 produced a better base input. Even though there is a divergence of 2.3 percentage points for the value failure prediction of CRC-4, subfigure *e* shows that its time out is predicted with only 0.1 percentage points of deviation.

In Table 5.10, we used *mean squared error (MSE)* in order to quantify the difference between the predicted and the observed percentages of value failures. In this table we compare the results of value failure prediction using instruction-based technique and linear regression equation presented in Table 5.4. The errors shown in the first two columns of Table 5.10 correspond to nine sample predictions, while the ones in the second two columns refer to the three sample predictions shown in Figure 5.6.

Table 5.10. Error comparison of value failure prediction using instruction-based predictor and linear regression equation

	Instruction-based prediction using one input	Linear regression equation	Instruction-based prediction using two input	Linear regression equation
CRC	24.38	67.75	3.25	12.33
SHA	0.69	1.25	0.53	0.39

For SHA, it can be seen that instruction-based prediction using one input has less amount of error when compared to the value failure predicted by linear regression equation. However, this is vice versa when two input sequences are used in the instruction-based prediction. The explanation for these results lies on the nature of the two predictors. The instruction based predictor picks the “closest” PCA-generated points to the target point, while the linear regression model is based on the minimization of the MSE for all the points existed in the data set. Therefore, the linear regression model is more robust compared to the instruction based when there is no close point.

For CRC, the errors of both prediction techniques are almost an order of magnitude greater than the calculated MSE for SHA. This is mainly due to the fact that for CRC the failure distributions of the first three execution flows are outliers (CRC-1, CRC-2, CRC-3). This can also be seen in the signatures of the fault-free run of CRC execution flows, see Figure 5.4. However, the predicted value failures using instruction-based predictor have less amount of error when compared to the value failures predicted by linear regression equation. We also calculate the MSE by excluding the three outlier execution flows. This results in mean squared errors of 0.65 and 1.12 for instruction-based prediction using one point and linear regression equation for the remaining six execution flows, respectively.

The effectiveness of the instruction-based technique lies to proper selection of the base input. The more input sequences we have, the higher the probability that we can select or generate a useful base input which results in a more accurate prediction. On the other hand, the linear regression equation only considers the length of the input in order to predict its failure distributions which in the case of our selected target programs seem to make accurate predictions. However, the failure distribution of other target programs like *Quicksort* might not be correlated to the size of the input, while it seems their failure distributions can still be predicted using their assembly code and instruction-based prediction technique.

6 Conclusions

In this thesis we studied a series of fault injection campaigns conducted on two different target programs, SHA-1 and CRC-32. Each target program was analyzed using 18 different execution flows operating with different inputs and subsequently extended with a time redundant fault tolerant mechanism. The effect of different inputs on the failure distribution of each execution flow shows that there is a correlation between the length of the input and the error coverage. In fact inputs of the same length can be considered equivalent in terms of error coverage. This helps us reduce the number of fault injection campaigns and consequently to save resources such as time. For the non-fault-tolerant implementation of the target programs, execution flows with longer input sequences resulted in fewer covered errors when compared to shorter input sequences. Moreover, the error coverage of CRC execution flows varies between 93.94% and 58.08% while for SHA it is between 50.87% and 61.24%.

The percentage of covered errors increased significantly (to around 99%) with the adoption of the TTR-FR. Moreover, this mechanism is effective in detecting and correcting the value failures as the input changes. However, there are still faults that escape the TTR-FR and cause system failure.

The results of our two proposed prediction techniques (linear regression equation and instruction-based prediction) are quite satisfactory. Foremost advantage of linear regression equation is its simple calculation. However, it can only be used when there is a correlation between the error coverage and input length of a target program. On the other hand, the instruction-based predictor can be considered as a more general approach which does not require any linear correlation between the size of input and error coverage. By comparing the observed percentage of experiments classified as value failures and its predicted percentage, we conclude that for SHA execution flows, the linear regression equation gives us a better prediction with the mean square error (MSE) of 0.39. Whereas, the instruction-based predictor produces a better prediction for CRC execution flows with the MSE of 3.25.

The percentage of experiments classified as detected by hardware (DHW) changes slightly for the execution flows of each target program. It is yet notable that the result of detected by hardware experiments does not change considerably in the TTR-FR version of each target program when compared to the non-fault-tolerant implementation. This means that the hardware detection mechanism operates independently from the TTR-FR.

Future Work

In this thesis we have shown that there is a linear correlation between the selected input and the failure distribution of SHA-1 and CRC-32 target programs. We have also proposed two prediction techniques, but more work is needed to determine whether the methodology is applicable to other target programs. Furthermore, the prediction techniques could be improved, and other statistical methods could be employed to achieve a more accurate prediction. Our selected target programs may or may not be representative for many other target programs, but the overall conclusion must be that it is worth investigating the correlation of different inputs to the failure distribution using assembly level signature of the target programs. Moreover, our target programs had low complexities, thus more complicated programs can be the next target of our technique. Larger programs typically consist of a number of subprograms, so error coverage for each of these subprograms could probably be used to estimate the total error coverage. We would also like to encourage other researchers to use other assembly level metrics to evaluate other target programs equipped with different fault tolerance mechanisms. In addition to the single bit flip fault model that we used in our fault injection campaigns, the effect of multiple bit flips can also be investigated in the future.

Limitations

The main limitation of this study is the random selection of fault injection locations instead of performing an exhaustive injection. This is a threat to the validity of results, i.e., the results of 25000 fault injection experiments for SHA and 12000 experiments for CRC might not be a good representative of the total failure distribution of each target program. Furthermore, we only used two target programs in our investigations which do not have high complexities.

Bibliography

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," in *Micro, IEEE* , 2005, pp. 10-16.

[2] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: a tool for the validation of system dependability properties," in *Twenty-Second International Symposium on Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers.*, 1992, pp. 336-344.

[3] Henrique Madeira, Mário Zenha Rela, Francisco Moreira, and João Gabriel Silva, "RIFLE: A General Purpose Pin-level Fault Injector," in *The First European Dependable Computing Conference on Dependable Computing*, 1994, pp. 199-216.

[4] J. Arlat et al., "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers* , vol. 52, no. 9, pp. 1115-1133, September 2003.

[5] M. Sonza Reorda, M. Rebaudengo, "Evaluating the fault tolerance capabilities of embedded systems via BDM ," in *17th IEEE VLSI Test Symposium*, 1999, pp. 452-457.

[6] Z. Segall et al., "FIAT-fault injection based automated testing environment," in *Eighteenth International Symposium on Fault-Tolerant Computing*, 1988, pp. 102-107.

[7] Peter Folkesson and Johan Karlsson, "Considering Workload Input Variations in Error Coverage Estimation," in *Third European Dependable Computing Conference on Dependable Computing*, 1999, pp. 171-190.

[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, jan.-Mar. 2004.

[9] P. Hazucha et al., "Neutron Soft Error Rate Measurements in a 90-nm CMOS Process and Scaling Trends in SRAM from 0.25-μs to 90-nm Generation," in *IEEE International Electron Devices Meeting*, 2003, pp. 21.5.1-21.5.4.

[10] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August, "SWIFT: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, 2005, pp. 243-254.

[11] A. Avizienis, "The N-version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, pp. 1491-1501, 1985.

[12] James J. Horning, Hugh C. Lauer, P. M. Melliar-Smith, and Brian Randell, "A program structure for error detection and recovery," in *International Symposium on Operating Systems*, 1974, pp. 171-187.

[13] J. Aidemark, P. Folkesson, and J. Karlsson, "A framework for node-level fault tolerance in distributed real-time systems ," in *International Conference on Dependable Systems and Networks*, 2005, pp. 656-665.

[14] Daniel Skarin, "On fault injection-based assessment of safety-critical systems," Chalmers University

of Technology, Gothenburg, PhD Thesis ISSN: 0346-718X, 2010.

[15] "AMBER, State of the Art," deliverable no.d2.2, 2010.

[16] H. Kopetz et al., "Distributed fault-tolerant real-time systems the MARS approach," in *Micro, IEEE*, 2002, pp. 25 - 40.

[17] Emmerich Fuchs, "An Evaluation of the Error Detection Mechanisms in MARS using Software Implemented Fault Injection," in *Dependable Computing - EDCC-2, Proceedings of the Second European Dependable Computing Conference*, 1996, pp. 73-90.

[18] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek, "Fault injection experiments using FIAT," in *Computers, IEEE Transactions on*, 1990, pp. 575-582.

[19] Antonio Dasilva, José F Martínez, Lourdes López, Ana B García, and Luis Redondo, "Exhaustif®: A fault injection tool for distributed heterogeneous embedded systems," in *Euro American conference on Telematics and information systems* , 2007, pp. 1-8.

[20] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," in *Micro, IEEE*, 1994, pp. 8-23.

[21] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *Nineteenth International Symposium on Fault-Tolerant Computing*, 1989, pp. 348-355.

[22] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 557-562.

[23] D. Powell, E. Martins, J. Arlat, and Y. Crouzet, "Estimators for fault tolerance coverage evaluation," in *The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993, pp. 228-237.

[24] L.M. Kaufman, B.W. Johnson, and J.B. Dugan, "Coverage estimation using statistics of the extremes for when testing reveals no failures," *IEEE Transactions on Computers*, vol. 51, no. 1, pp. 3-12, January 2002.

[25] C Constantinescu, "Estimation of coverage probabilities for dependability validation of fault-tolerant computing systems," in *Ninth Annual Conference on Computer Assurance*, 1994, pp. 101-106.

[26] C. Constantinescu, "Using multi-stage and stratified sampling for inferring fault-coverage probabilities," *IEEE Transactions on Reliability* , vol. 44, no. 4, pp. 632-639, December 1995.

[27] J.B. Trivedi, K.S. Dugan, "Coverage modeling for dependability analysis of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 38, no. 6, pp. 775-787, June 1989.

[28] Thomas Ball, Peter Mataga, and Mooly Sagiv, "Edge profiling versus path profiling: the showdown," in *25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 134-148.

[29] Thomas Ball and James R Larus, "Efficient path profiling," in *29th annual ACM/IEEE international symposium on Microarchitecture*, 1996, pp. 46-57.

[30] Yih-Farn Chen, D.S. Rosenblum, and Kiem-Phong Vo, "TESTTUBE: a system for selective regression testing ," in *16th International Conference on Software Engineering*, 1994, pp. 211-220.

[31] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi, "An empirical investigation of program spectra," in *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 1998, pp. 83-90.

[32] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering* , 1997, pp. 432-449.

[33] Peter Folkesson, Joakim Aidemark, and Johan Karlsson, "Path-Based Error Coverage Prediction," *Electronic Testing*, vol. 18, no. 3, pp. 343-349, June 2002.

[34] (2011, December) MPC565 Reference. Manual.

[35] Karl Pearson, "On Lines and Planes of Closest Fit to Systems of Points in Space," *Philosophical magazine*, vol. 2, no. 6, pp. 559–572, 1901.

[36] (2011, December) MiBench version 1. [Online]. http://www.eecs.umich.edu/mibench/

[37] M.R. Guthaus et al., "MiBench: A freee commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization WWC-4.* , 2001, pp. 3-14.

[38] Ruben Alexandersson and Johan Karlsson, "Fault injection-based assessment of aspect-oriented fault tolerance," Chalmers University of Technology, Gothenburg, Technical report ISSN : 1652-926x, 2010.

[39] (2011, December) iSYSTEM AG. [Online]. http://www.isystem.com/component/content/article/4-articles/18-ic3000-activeemulator.html

[40] (2011, Dec. 1st) PHYTEC America. [Online]. http://www.phytec.com/products/som/PowerPC/phyCORE-MPC565.html

[41] (2011, December) iSYSTEM AG. [Online]. http://www.isystem.com/downloads/sw-updates/winidea-2010.html

[42] R., Vinter, J., Folkesson, P., Karlsson, J. Barbosa, "Assembly-level preinjection analysis for improving fault injection efficiency," in *Proceedings of the Fifth European Dependable Computing Conference (EDCC-5)*, Budapest, Hungry, 2005, pp. 246 – 262.

[43] Raul Barbosa, "Fault Injection Optimization through Assembly-Level Pre-injection Analysis," CHALMERS UNIVERSITY OF TECHNOLOGY, Göteborg, Master's Thesis 2004.

[44] D. Skarin, "On Fault Injection-Based Assessment of Safety-Critical Systems," Chalmers University

of Technology, Göteborg, Ph.D Dissertation 2010.

[45] "Assessing, Measuring, and Benchmarking Resilience, State of the Art," deliverable no.d2.2, 2010.