Universitat Autònoma
de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de Computadors
i Sistemes Operatius

ARTFUL

Deterministically Assessing the Robustness against Transient Faults of Programs

Thesis submitted by **João Artur Dias Lima Gramacho** in fulfillment of the requirements for the degree of Philosophiæ Doctor per the Universitat Autònoma de Barcelona.

Bellaterra, May 2014

ARTFUL

Deterministically Assessing the Robustness against Transient Faults of Programs

Thesis submitted by **João Artur Dias Lima Gramacho** in fulfillment of the requirements for the degree of Philosophiæ Doctor per the Universitat Autònoma de Barcelona. This work was advised by **Dr. Dolores Isabel Rexachs del Rosario**.

Bellaterra, May 2014

_____

Dr. Dolores Isabel Rexachs del Rosario
Thesis Advisor

From Oxford Online Dictionary

http://oxforddictionaries.com/definition/american_english/artful

artful

Syllabification: (art·ful)

Pronunciation: /ˈärtfəl/

Definition of **artful**

adjective

1 (of a person or action) clever or skillful, typically in a crafty or cunning way:
her artful wiles

2 showing creative skill or taste:
an artful photograph of a striking woman

From **Foundation and Earth** (by Isaac Asimov, Doubleday, 1986)

Chapter 21

The Search Ends

Perhaps it was because of the matter-of-fact way in which Daneel said it; or perhaps because a lifetime of twenty thousand years made death seem no tragedy to one doomed to live less than half a percent of that period; but, in any case, Trevize felt no stir of sympathy.

“Die? Can a machine die?”

“I can cease to exist, sir. Call it by whatever word you wish. I am old. Not one sentient being in the Galaxy that was alive when I was first given consciousness is still alive today; nothing organic; nothing robotic. Even I myself lack continuity.”

“In what way?”

"There is no physical part of my body, sir, that has escaped replacement, not only once but many times. Even my positronic brain has been replaced on five different occasions. Each time the contents of my earlier brain were etched into the newer one to the last positron. Each time, the new brain had a greater capacity and complexity than the old, so that there was room for more memories, and for faster decision and action. But-"

“But?”

“The more advanced and complex the brain, the more unstable it is, and the more quickly it deteriorates. My present brain is a hundred thousand times as sensitive as my first, and has ten million times the capacity; but whereas my first brain endured for over ten thousand years, the present one is but six hundred years old and is unmistakably senescent. With every memory of twenty thousand years perfectly recorded and with a perfect recall mechanism in place, the brain is filled. There is a rapidly declining ability to reach decisions; an even more rapidly declining ability to test and influence minds at hyperspatial distances. Nor can I design a sixth brain. Further miniaturization will run against the blank wall of the uncertainty principle, and further complexity will but assure decay almost at once.”

Acknowledgments

It was the beginning of 2006, when I first came to Europe for my first backpack trip, when my mind started to change a lot. I was well employed in a big Brazilian mobile company, with good professional opportunities in sight. To visit Europe and to visit my friends living in Europe made me start questioning if that "big city, big company, professional growth" was really what I wanted.

I took almost two years declining jobs offers to move from my home town in Brazil to larger cities, thinking about what I really wanted, to finally figure out that I wanted to fulfill my researcher needs, doing a Master, and maybe even doing a PhD. And if I had to leave my home town, to be far from my family, I would like to go to a place without many social problems Brazil has.

So, in June 2008, I left my home town, and also my home country, to move to Barcelona, Spain, to start learning how to become a researcher (and also a teacher) at CAOS (Computer Architecture and Operating Systems) department at UAB.

During the five years I was there, my life changed even more! I made new friends (my "Barcelona family") and I also met (and married) my wife!

Now it is time to finish this formal researcher formation.

I would like to thank all fellows of CAOS (always helping each other), especially my working group (always helping each other even more), and, in particular, I thank Dolores Rexachs and Emilio Luque for their trust in my work. I will always remember the researched values I learned from you... And I've started to spread these values already!

Thank you, Dona Ana, Sérgio, Gabi, Anamel, Laís and all my family and friends that, even far away, have always been by my side during this journey.

To my friends Eduardo, Fialho and Aprigio, I would like to thank the advices, the help and the patience! You know that you can count on me!

Finally, I want to thank Graziela for her love, understanding, patience and the motivation she gave me to finish this work.

Abstract

Computer chips are evolving to obtain more performance, using more transistors and becoming denser and more complex. One side effect of such a scenario is that processors are becoming less robust than ever against transient faults.

As on-chip solutions are expensive or tend to degrade processor performance, the efforts to deal with these transient faults in higher levels, such as the operating system or even by the programs, are increasing.

Software based fault tolerance approaches to deal with transient faults often use fault injection experiments to evaluate the behavior of programs with and without their fault detection proposals.

Using fault injection experiments to evaluate programs behavior in presence of transient faults require running the program under evaluation and injecting a fault (usually by flipping a single bit in a processor register) for a sufficient amount of times, always observing the program behavior and if it ended presenting the expected result. One problem with this strategy is that the fault injection space is proportional to the amount of instructions executed multiplied by the amount of bits in the processor architecture register file.

Instead of being exhaustive (it would be unfeasible), this approach consumes lots of CPU time by running or simulating the program being evaluated as many times as necessary to obtain a reasonable valid statistical approximation (usually just a few thousand times). So, the time required to evaluate how a single program would behave in presence of transient faults might be proportional to the time needed to run the program for five thousand times.

In this work we present the concept of a program's robustness against transient faults and also present a methodology named ARTFUL (from **A**ssessing the **R**obustness against **T**ransient **Fa**u**l**ts) designed to, instead of using executions with fault injections, deterministically calculate this robustness based on program's execution trace over an given processor architecture and on information about the used architecture.

Our approach was able to calculate the precise robustness of some benchmarks using just the time need to run the evaluated program for tents of times in the best cases.

Resumen

Los procesadores están evolucionando para obtener más rendimiento, utilizando más transistores y quedándose más densos y más complejos. Un efecto secundario de este escenario es que los procesadores están cada vez menos robustos frente a fallos transitorios.

Como soluciones basadas en los propios procesadores son caras o tienden a degradar el rendimiento del procesador, los esfuerzos para hacer frente a estos fallos transitorios en las capas superiores, como en el sistema operativo, o incluso en los programas, están aumentando.

Propuestas de tolerancia a fallos basada en la capa de software para hacer frente a los fallos transitorios comúnmente usan experimentos de inyección de fallos para evaluar el comportamiento de los programas con y sin sus propuestas de detección de fallas.

Utilizar experimentos de inyección de fallos para evaluar el comportamiento de los programas en presencia de fallos transitorios requiere ejecutar el programa evaluado haciendo la inyección de un fallo (por lo general cambiando un solo bit en un registro del procesador) por una cantidad suficiente de veces, siempre observando el comportamiento del programa y si este ha finalizado presentando el resultado esperado. Un problema con esta estrategia es que el espacio de inyección de fallos es proporcional a la cantidad de instrucciones ejecutadas multiplicado por la cantidad de bits en el archivo de registros de la arquitectura del procesador.

En lugar de ser exhaustivos (que sería inviable), este método consume mucho tiempo de CPU ejecutando o simulando el programa que se está evaluando tantas veces como sea necesario para obtener una aproximación estadística válida razonable (por lo general sólo unos pocos miles de veces). Así, el tiempo requerido para evaluar cómo un solo programa se comportaría en presencia de fallos transitorios podría ser proporcional al tiempo necesario para ejecutar el programa cerca de cinco mil veces.

En este trabajo se presenta el concepto de robustez de un programa frete a fallos transitorios y también presenta una metodología llamada ARTFUL diseñada para, en lugar de utilizar las ejecuciones con inyecciones de fallos, hacer el cálculo de la robustez de forma determinística, basada en una traza de ejecución del programa y en la información sobre la arquitectura utilizada.

Nuestro método fue capaz de calcular la robustez precisa de algunos *benchmarks* utilizando sólo el tiempo necesario para ejecutar el programa evaluado para decenas de veces en los mejores casos.

Contents

Chapter 1
Introduction

With the evolution of computer processors for better performance, computer chips are using smaller components, having more transistors with higher density and operating at lower voltage. All these factors turn computer processors less robust against transient faults [Wang et al., 2004].

Transient faults are those faults that might occur once and may not happen again the same way in a system lifetime. Transient faults in computer systems may occur in processors, memory, internal buses and devices, often resulting in an inversion of a bit state (i.e. single bit flip) on the faulty location [Baumann, 2005]. Transient faults in computer systems commonly are effect of cosmic radiation, high operating temperature and variations in the power supply subsystem [Constantinescu, 2005].

A transient fault may cause an application to misbehave (e.g. write into an invalid memory position; attempt to execute an inexistent instruction). Such misbehaved application will then be abruptly interrupted by the operating system fail-stop mechanism. Nevertheless, the biggest risk happens when the transient fault bit-flip causes an undetected data corruption, resulting in an incorrect application final result that might not be ever noticed [Mukherjee, Emer and Reinhardt, 2005].

In High Performance Computing (HPC), the risk of having a transient fault grows with the amount of computer processors working together [Oliner and Stearley, 2007]. So, the more computational power a HPC system has by adding more processors, the bigger is the risk of an unnoticed data corruption produced by a transient fault [Bronevetsky and Supinski, 2008].

Research about transient faults started with computers in hostile environments like outer space [Dodd and Massengill, 2003], but official reports of transient faults' effects in large computer installations became public since year 2000. Those reports evidenced the risk of having transient fault in HPC because of its large number of components working together. With the risk of having transient faults affecting computation results, researchers needed the occurrence of those faults to study its effects and also to test their work.

Since transient faults occur in a very unpredictable way, to study the effects of these faults in computers, operating systems and applications is a very common practice to use an environment with fault injection capabilities [Arlat et al., 1990].

To study the effects of transient faults in applications running into computer system, these fault injections capable environments will be used changing states of the processor registers or changing data in memory, either randomly or based on a specific design, depending on the purpose of injection. This study should result in knowledge about the sensible parts of the program and in how effective a fault detection mechanism can be.

The evaluation using executions with fault injection is an experimental approach that is very time consuming and produces approximated results based on statistical data collected. It is also known that the amount of executions to evaluate a program in a fault injection campaign will affect the precision of the results obtained [Reis et al., 2005].

Just to state an example, besides the work needed to setup an environment capable of performing fault injections, [Reis et al., 2006] had to execute a set of benchmarks for a total of 1.03 million times, injecting only one fault into each execution, to obtain their results. Even being able to distribute this huge amount of executions in multiples processing units (a multicore environment or a cluster), the cost (in terms of CPU/hour) was very high. It is important to notice that works based on executions with fault injection rarely test multi processed programs. The complexity of performing fault injections in a HPC environment (multiprocessors or clusters) discourages such approach. Often, the fault injection experimental approach uses small benchmarks or the significant fractions of the benchmarks.

So, after researching about fault injections and, more specifically, about fault injections for evaluating the behavior of programs in presence of transient faults, we started working on a possible alternative method.

This alternative method would be based on a model of the program execution: an execution trace. Using a model of the program execution as the input to our method would allow the analysis of alternatives without the need of executing the program again. Changes on the trace would be enough to perform further evaluation.

This new method would try to address some key aspects of the evaluations using executions with fault injection, like the need of performing fault injections and the non-deterministic approach using statistical approximations to verify the measured metrics.

The new approach would evaluate the whole execution of a program running over a given processor architecture, the equivalent to testing all possible fault injection points in a program execution exhaustively. The use of a model of program execution based on a trace give us useful information about the program execution: i.e. which are the most

repetitive parts of the program, and how the program uses processor architecture resources.

Also, this new method would try to address the efficiency of the process of evaluating a program behavior once in presence of transient faults. By efficiency we consider the amount of CPU/time needed to evaluate a program relative to the CPU/time needed to effectively run the program.

Based on the related work, having to run a single program for five thousand times [Reis et al., 2006] to provide an evaluation would be our baseline. We believed that improving this process efficiency could lead to another benefit: to be able to evaluate larger programs (not only the significant fractions of benchmarks, or benchmarks with small workloads) and even parallel programs.

The proposed method was divided in two tasks: the generation of the program execution trace (the generation of the model of the program execution), and the analysis of the program execution trace. Our guess was that generating a program execution trace and analyzing this trace should consume less CPU/hour than analyzing a program using executions with fault injection.

1.1 Objective

After researching about related work we elaborated an algebraic formalization of how to evaluate a program behavior in the presence of transient faults inspired in some assumptions made by the authors when trying to improve the efficiency of their experiments.

We use the concept of **robustness against transient faults** as the ability of a program running over a given processor architecture, once in presence of a transient fault, to keep running and give a correct result when finish or to stop the execution when a soft error is detected and inform about it.

During our work we used the robustness concept as a metric, allowing us to measure how a program's robustness against transient faults once running over a given processor architecture. A program's robustness as a metric would vary from zero, i.e. the minimal possible robustness, meaning that any transient fault will affect the program result) to one, i.e. the maximum possible robustness, meaning that any transient fault won't affect the program execution or will be detected, assuring that the user will know if a fault could affect the results.

Based on our algebraic formalization, and using a program execution trace as input, we present in this work a methodology to deterministically calculate a program's

robustness against transient faults over a given processor architecture as a metric without the necessity of executing the evaluated program using fault injections.

The proposed methodology is named ARTFUL (from **A**ssessing the **R**obustness against **T**ransient **Fa**u**l**ts) and have three main characteristics:

1. Exhaustiveness: We aim to evaluate the whole program execution without using statistical approximations, and doing so, we aim to present results similar to an extensive fault injection campaign;
2. Precision: Because of the deterministic nature of our methodology, the robustness of a program execution will be precisely calculated. Two evaluations of the same program execution must provide exactly the same robustness;
3. Efficiency: All the tasks needed to evaluate a program's robustness in widely used processor architecture for HPC should spend less time than performing a fault injection campaign for the program.

The accomplishment of these three characteristics will highlight the benefits of the ARTFUL methodology in comparison with evaluations using fault injection campaigns.

1.2 Organization of this dissertation

This dissertation contains eight chapters, stating with this introduction.

The state of the art of this work is distributed in three chapters: Chapter 2, Chapter 3 and Chapter 4.

In Chapter 2, we present an overview about transient faults, concepts related to transient faults research and explain why is important to study about transient faults.

Chapter 3 describes common fault injection methods, their main characteristics and the utility of such methods.

In Chapter 4 we present some related work focusing in the amount of experimentation needed to evaluate program's behavior once in presence of a transient fault. In order to provide the readers a glimpse of the whole process behind such kind of evaluation, we designed a simple program robustness evaluation using extensive fault injection campaigns. We also present a brief study about the randomness effect when not using extensive fault injections campaigns (the most common case).

Chapter 5 describes the ARTFUL methodology and its formalizations. Also, this chapter validates the methodology by comparing the evaluation made in Chapter 4 with the evaluation using the ARTFUL method.

All the evaluations in both Chapter 4 and Chapter 5 were made with a simple program with just a few instructions and non-complex processor architecture. This was necessary to accomplish the extensiveness of the fault injection campaigns and the use of the ARTFUL methodology without complex tools, but these chapters do not discuss efficiency (time needed to perform the evaluations).

In Chapter 6 we present the tools developed to help the robustness evaluation of programs for the x64 processor architecture.

Chapter 7 contains the experimental evaluation of this work and also some of our contributions. It starts with the evaluation of some benchmarks and a comparison between a robustness evaluation using ARTFUL tools and using fault injection campaigns.

Chapter 7 also presents two improvements we made in the tools to help us to reduce the time needed to perform a robustness against transient faults evaluation of a program running over the x64 processor architecture. The first one is based on the possibility of simplifying repetitive sequences of instructions during the robustness analysis. The second one uses an auxiliary performance prediction tool to predict the robustness of the evaluated program. Both improvements are tested and the results of the experimental evaluation are presented.

Finally, in Chapter 8 we state our conclusions and propose some future works.

Chapter 2
Transient Faults

Transient faults are faults that do not reflect a permanent malfunction. A permanent fault in some component will produce faults, errors or unexpected behavior every time this faulty component is used. Transient faults, on the other hand, may occur only once on the whole component lifetime because they are a result of external sources influences, such as high-energy particles that cause voltage pulses in digital circuits, or some internal sources like power supply noise and temperature variation, for example [Wang et al., 2004].

Radiation-induced transient faults, for example, arise from energetic particles (such as neutrons from the atmosphere) generating electron-hole pairs as they pass through a semiconductor device. Transistor source and diffusion nodes can collect these charges that may accumulate at an amount of charge sufficient to invert the state of a logic device, injecting a fault into the circuit's operation (i.e. inverting a bit in a memory position or in a processor register) [Mukherjee, 2008] as shown in Figure 1.

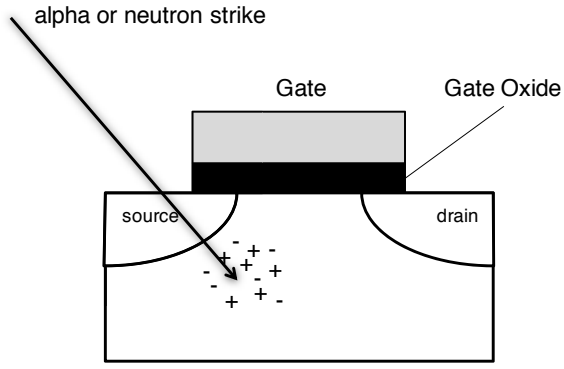

Figure 1 – Radiation particle strike

Transient faults started as a problem to those designing high-availability systems and systems for electronic-hostile environments such as outer space [Wang et al., 2004], but this situation has changed. As the process of miniaturization of components evolved,

these components become less robust against external influences. As the influence of terrestrial radiation grew, many systems started to implement extensive error detection and/or correction mainly for on-chip memories. The major problem is that protecting only memory is not enough for miniaturization scales of sub-65nm technologies and lower.

The need for protection against transient faults effects in enterprise computing and communication applications are motivating new on-chip mechanisms to protect latches and flip-flops. Eventually, even some combinatorial logic protection will be necessary in computer chips as more and more transistors are being used in future technologies [Mitra et al., 2006].

But, with the advent of multicore and manycore computer chips, the amount of transistors and buses in a computer processor is so big that the industry of computer chips expects that the higher levels of a computer system (operating system, parallel computer frameworks and even the applications) deal with the possibility of transient faults more often [Cappello et al., 2009].

2.1 Transient faults effects

A fault can generate one or more latent errors. An error is the manifestation of a fault in a system. A latent error becomes effective once the resource with the error is used by the system to do some computation. Also, an effective error often propagates from one system component to another, thereby creating new errors. A failure is the manifestation of an error on the service provided by the system. A failure occurs when the actual behavior of a system deviates from the specified one.

For example, corresponding to the states of the Figure 2:

1. If an energetic particle hits a memory cell it may produce fault;
2. Once this fault changes the state of the memory cell it generates an latent error;
3. This error remains latent until the affected memory is read by some other structure, becoming an effective error;
4. A failure occurs if the memory changed by the error is read and affects the operation of the system or application by changing its behavior.

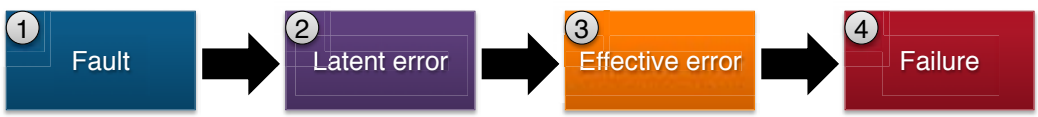

Figure 2 – Fault, error and failure states.

Faults can be characterized by its duration as permanent, transient or intermittent. A permanent fault will remain in a system until some corrective action is taken. Intermittent faults are those that keeps appearing and disappearing under some circumstances. As in permanent faults, identifying the faulty device and doing some corrective action on it (e.g.: replacing it) may prevent appearing intermittent faults. On the other hand, a transient fault appears, disappears and will probably never occur again the same way in a system lifetime [Mukherjee, 2008].

The errors produced by transient faults are called soft errors. After observing a soft error, there is no implication that the system is any less reliable than before. Soft errors change the data but not change the physical circuit itself. If the data is rewritten, the circuit will work perfectly again.

The soft error expression used in transient fault literature should not be confused with errors of software applications (software programming errors).

2.2 Metrics used in transient faults studies

Current works about transient fault and soft errors use common fault tolerance metrics, but also added some that easy the math of estimating a system possibility of failure.

Time to failure (TTF) expresses the time to a fault or an error, even though it refers specifically to failures. Mean time to failure (MTTF) of a component expresses the amount of time elapsed between the last system startup or restart and then next error of the component, as shown in Figure 3. MTTF of a component are commonly expressed in years and it is obtained based on an averaged estimative of failure prediction done by the component's supplier.

The MTTF of a whole system (a group of components) can be obtained by combining the MTTF of all its components, as shown in Equation 1 below [Mukherjee, 2008].

Equation 1 – MTTF of a given system.

$$\mathrm{MTTF}_{system} = \frac{1}{\sum_{i=0}^{n} \frac{1}{\mathrm{MTTF}_i}}$$

The use of the Failure In Time (FIT) term became more useful to engineers by its addictive property. One FIT represent an error in a billion (10^9) hours. To compute a system FIT is only necessary to add its components FIT, as shown in Equation 2 below [Mukherjee, 2008]:

Equation 2 – FIT rate of a given system.

$$\text{FITrate}_{system} = \sum_{i=0}^{n} \text{FITrate}_i$$

FIT rate and MTTF of a component are inversely related under certain conditions:

Equation 3 – Relation between FIT and MTTF.

$$\text{MTTF (in years)} = \frac{10^9}{\text{FITrate} \times 24 \text{ hours} \times 365 \text{ days}}$$

There are more two commonly used terms used in fault tolerance literature: mean time to repair (MTTR) and mean time between failures (MTBF). MTTR represent the time needed to repair an error once it is detected. MTBF represents the average time between the occurrences of two errors [Mukherjee, 2008]. The MTBF can be expressed as MTBF = MTTF + MTTR as shown in Figure 3 adapted from [Mukherjee, 2008].

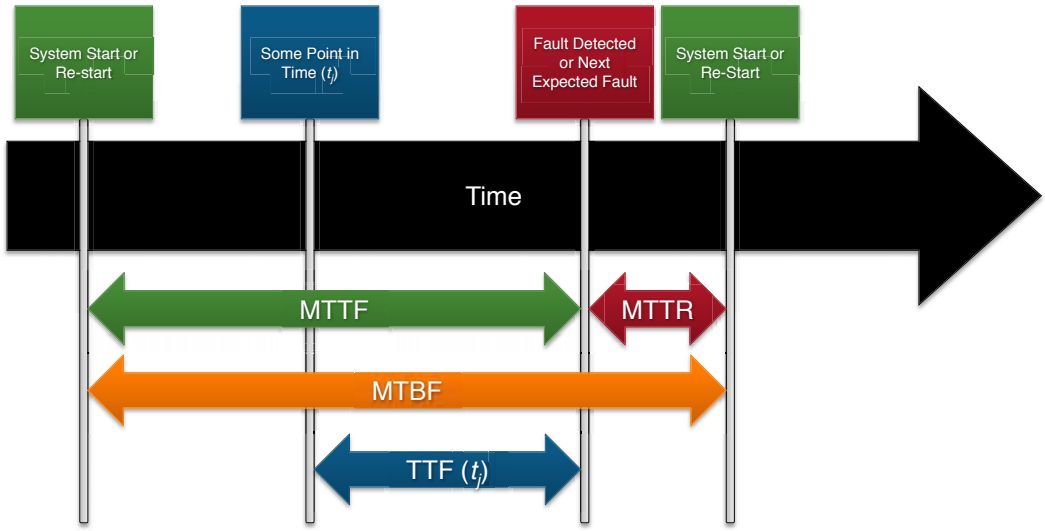

Figure 3 – Metrics and its relationships.

The estimation of FIT rate caused by soft errors are called soft error rate (SER).

Architectural Vulnerability Factor (AVF) [Mukherjee et al., 2003] is a metric that quantifies both architecture-level and program-level reliability against transient faults. It depends on both software and hardware where the program is being evaluated. So, for a software level evaluation, designers cannot use AVF to make hardware-independent statements about a program's reliability.

In an effort to design an accurate method to quantify software-level reliability, [Sridharan and Kaeli, 2008] proposed the Program Vulnerability Factor (PVF), a metric that quantifies the software-level reliability inherent in a program. PVF can be calculated for any software resource and is a property of a dynamic execution of a

program. Therefore, PVF is impacted only by changes to the binary program or to the workload used and not by changes in the hardware (considering that the processor architecture is the same). PVF can be used to reason about the reliability of a program without any knowledge of the target microarchitecture.

2.3 Evidence of soft errors

There are only few publications evidencing the occurrence of soft errors. As shown in Figure 4, the first evidences of soft errors were caused by contamination in the chips production in late 70's and 80's.

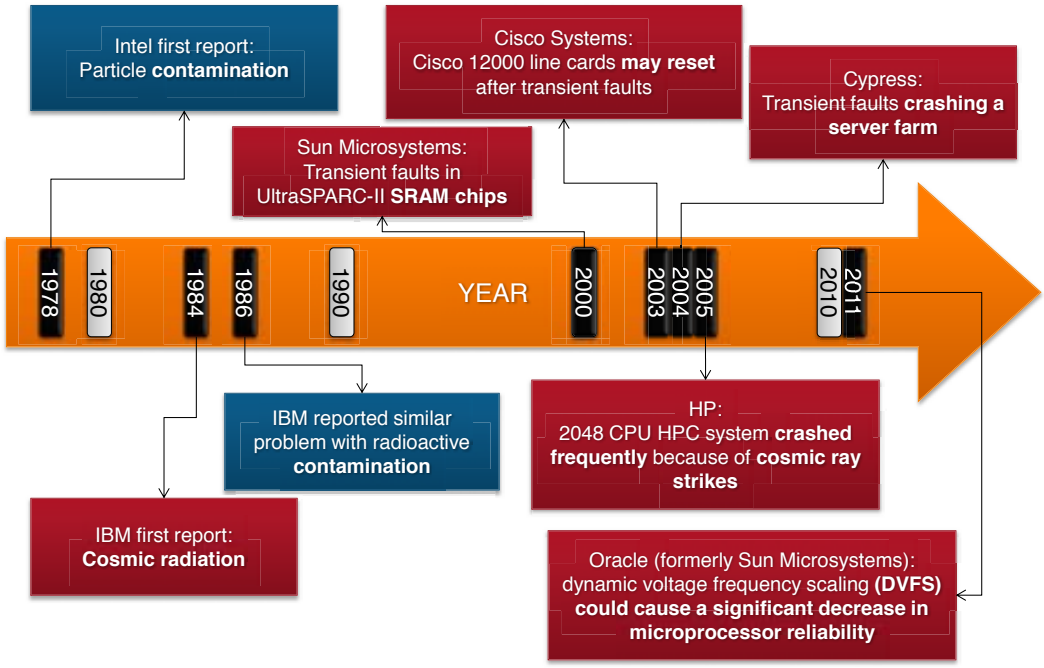

Figure 4 – Published evidences of soft errors.

Since 2000's, the reports of soft errors in large computer installations such as supercomputers and server farms are becoming more frequent. This happens because the number of components in this kind of installations is very big (thousands of CPU and terabytes of memory). Also, in this kind of installation it commonly has powerful and modern processors, with very high level of miniaturization and high density of transistors (and potentially less robust against transient faults).

For example, IBM has projected its Power 4 processor to be more robust against transient faults than usual desktop processors. When usual desktop processors are supposed to have an MTTF of 2 years, IBM Power 4 targets an MTTF of 7 years as shown in Table 1 from [Mukherjee, Emer and Reinhardt, 2005].

Table 1 – IBM Power 4 FIT per system effect.

FIT	System Effect	Transient Fault Outcome	MTTF (years)
114	Data Corruption	Silent Data Corruption (SDC)	1000
4566	System-Kill	Detectable Unrecoverable Error (DUE)	25
11415	Process-Kill	Detectable Unrecoverable Error (DUE)	10
16095	**Total transient faults with some effect**		7

Even knowing that a seven years MTTF is a low probability of failure, when we start to analyze this MTTF numbers in high performance computing, where we have hundreds or even thousands of processors working together to solve a problem, the MTTF of an hypothetic supercomputer lower as more processors are added, as show in Figure 5.

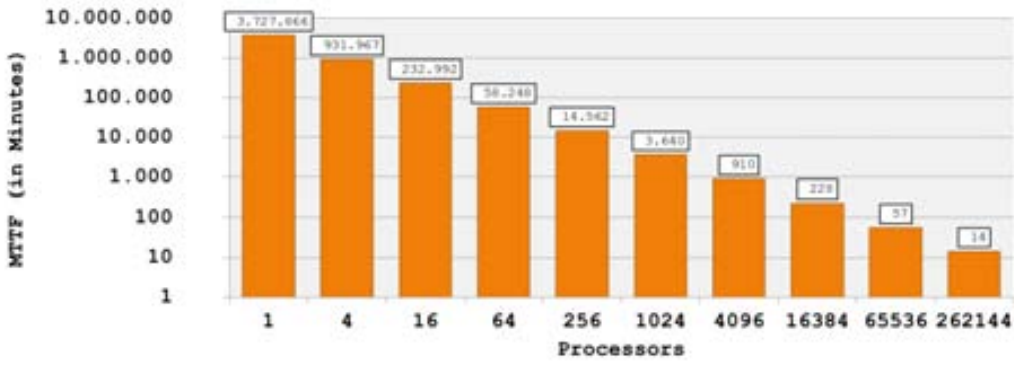

Figure 5 – MTTF of hypothetical IBM Power 4 based supercomputers.

Soft errors in microprocessor logic will soon become more common. In particular, latches, which are used in a variety of internal data structures, make up a large fraction of processor area and are a potentially vulnerable part for transient faults [Bronevetsky and Supinski, 2008].

Further, soft errors are a critical concern in the operation of real large systems. A 128k-node BlueGene/L experiences one soft error in its L1 cache every 4-6 hours due to radioactive decay in lead solder, the ASCI Q experienced 26.1 radiation induced CPU failures per week. A similarly-sized Cray XD1 supercomputer is estimated to experience 109 soft errors per week in CPUs, memory and FPGAs, if placed at the same altitude as the BlueGene/L [Bronevetsky and Supinski, 2008].

In a more recent work, driven by the concern about neutron induced soft errors from Oracle (formerly Sun Microsystems), [Dixit and Wood, 2011] stated that Dynamic Voltage Frequency Scaling (DVFS), a commonly used microprocessor energy reduction technique, could cause a significant decrease in microprocessor reliability.

2.4 Possible outcomes of a transient fault

The Figure 6 was adapted from [Mukherjee, Emer and Reinhardt, 2005] and describes the possible outcomes of an energetic particle hit in a computer processor or memory.

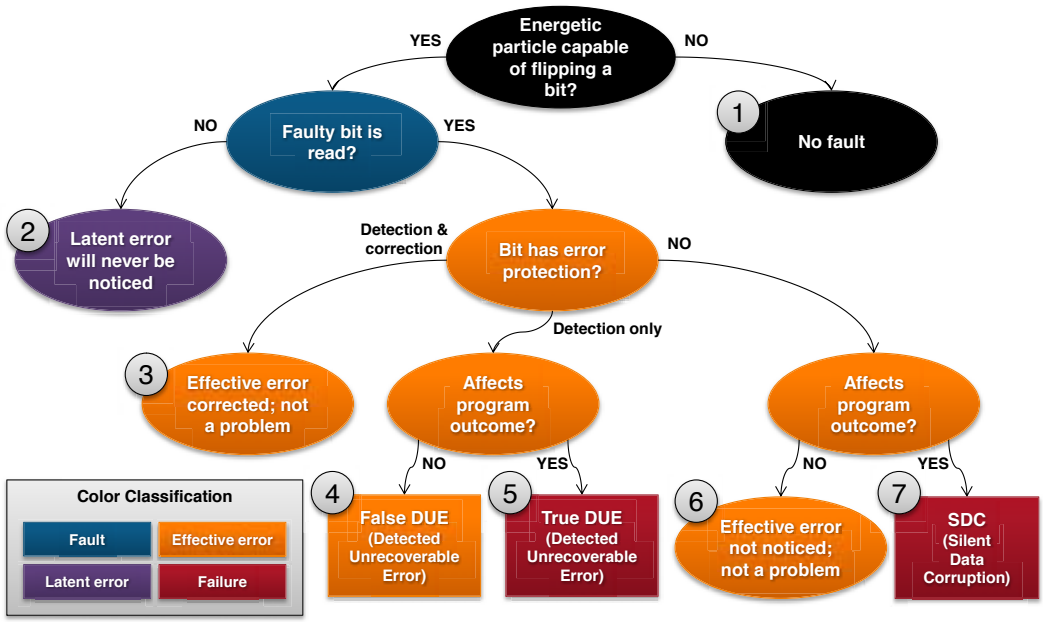

Figure 6 – Classification of possible outcomes of a transient fault.

The outcome 1 of Figure 6 indicates that the energetic particle hit was not capable of generating a fault.

The latent error happen when the energetic particle hit is capable of flipping a bit in a memory, in a processor register or even in a latch used for some purpose.

If this faulty bit isn't read by the system or it is overwritten at some point in time before using the value changed by the fault, this latent error will never be noticed (outcome 2 of Figure 6).

When the faulty bit is read by the system or one of its components, the soft error becomes effective.

The effective soft error can pass unnoticed by the upper layers of the component reading it if this bit is protected with detection and correction (outcome 3 of Figure 6). This is the case of memories with error-correcting codes (ECC) for example. A common type of memory device that uses ECC to improve its reliability is the dynamic random access memories (DRAM), more vulnerable to transient faults because of its structural

simplicity. They have extra memory bits that can be used by memory controllers to record parity of bit segments.

If this faulty bit has only error detection, it produces a state called detected unrecoverable error (DUE), avoiding the generation of incorrect outputs. In the case of a DUE, the system or component that read the faulty bit knows that it has an error and has no mechanism to correct it. The outcome 4 of Figure 6 represents a situation when the soft error doesn't affect the result generated by a program running on the system, a False DUE. If the error detected doesn't affect the program outcome, as the detection process add some overhead on the system, it might be avoided to improve systems performance.

But if the soft error affects the result generated by a program running on the system (outcome 5 of Figure 6), the system have to inform its upper layer (probably the operating system) of the effective error, avoiding the program to continue in this condition. As this error truly affects the program outcome, it is called True DUE. When the faulty bit is allocated to an application, usually the operating system produces the application interruption by an abnormal behavior (process kill), but the rest of the operating system and the other applications on it keeps running normally. In the case of this faulty bit has been allocated to the operating system, it might cause a situation where the unrecoverable part of the system affected can only be restarted by a system initialization (system kill), interrupting the operation of all applications running on it.

The major risk when a computer system is affected by a transient fault is when the soft error has occurred in a component without protection. The faulty bit is read and can be used by the program running on the system.

The outcome 6 of Figure 6 represents a situation when the undetected soft error doesn't affect the result generated by a program running on the system.

If the system uses the faulty bit in its operation without knowing that it was changed, this system will have a silent data corruption (SDC), the most dangerous outcome of a transient fault. The SDC (outcome 7 of Figure 6) will be processed by the application running on the system or by the operating system and might cause unpredicted consequences on the overall system behavior.

Currently, the industry specifies soft error rates of its components in terms of SDC and DUE numbers. The total SER of a system or component can be expressed as a sum of its SDC FIT and DUE FIT [Mukherjee, 2008] as shown in Table 1 previously explained.

2.5 Possible outcomes of a soft error

There are four main possible outcomes of soft errors in terms of DUE and SDC, as shown in Figure 7: an invalid instruction exception [Lesiak, Gawkowski and Sosnowski, 2007], a parity error during the read cycle [Lesiak, Gawkowski and Sosnowski, 2007], a memory access violation [Lesiak, Gawkowski and Sosnowski, 2007] and a change on a value produced by some system component or program calculation [Shivakumar et al., 2002].

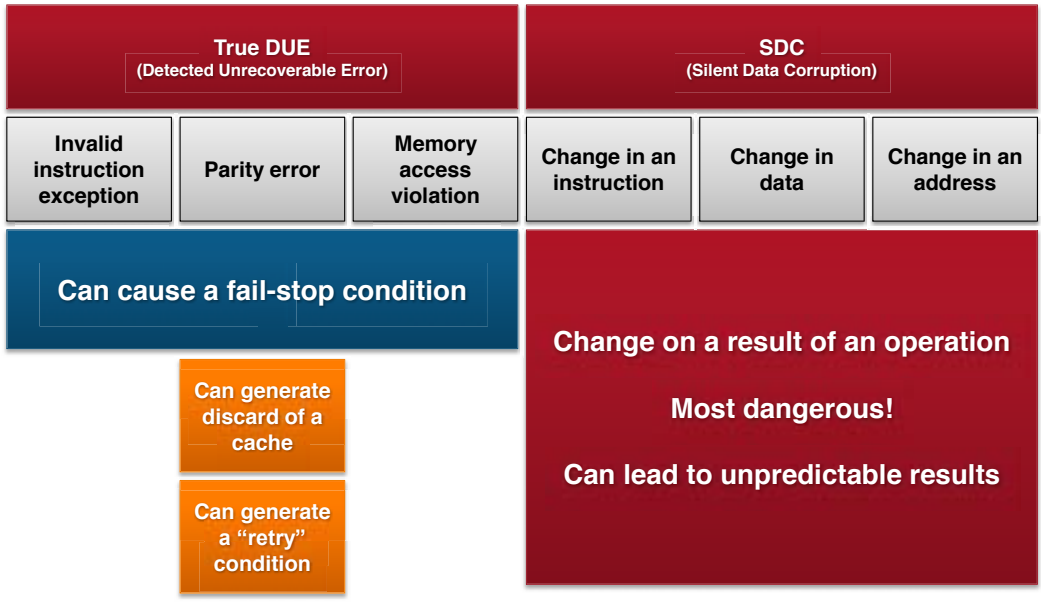

Figure 7 – Possible outcomes of soft errors.

2.5.1 Invalid instruction exception

An invalid instruction exception may occur when the processor cannot operate the data for a given instruction. For example, in a division of two numeric values, the denominator cannot be equal to zero. This situation will throw a Division By Zero exception, usually aborting the program execution.

In the example above, the denominator data might not be originally zero, but a transient fault may corrupt the denominator data, changing it to zero, throwing the invalid instruction exception.

In rare cases, a transient fault may also corrupt the instruction code to a combination that the processor architecture cannot recognize, becoming unable to execute it.

2.5.2 Parity error during read cycle

A parity error generated by a transient fault may occur on memory devices (main computer memory or caches) or in buses lines.

It is common that buses lines that implement parity check in transmissions also implement the possibility of the retransmission of the affected portion of data. The effect of this situation on computer systems is a small delay in the transmission, but the system keeps running normally.

When a soft error affects a memory position and the component affected uses parity to detect such situation, the consequences of this error depends on where in the memory hierarchy is the affected portion of memory.

If the parity error is in a portion of the main memory of the computer system it might be possible to recover it in the following cases:

1. The operating system uses the virtual memory resources of the processor and has copy of the affected portion of the memory stored in the swap file. If the portion affected didn't have been changed previously by normal computation, the operating system can recover the affected page from the swap file, restoring the previously known state of it.
2. The affected portion of the memory is a code segment of an application and there are binaries of the application in another device. In this case, is possible to the operating system read again the application's binaries on the other device and restore the affected portion of the memory.

If there is no source of a possible copy of the affected portion of the memory, the operating system has two options: stop the operation of the application or raise an error to the application, allowing the application try to recover itself.

When the parity error is detected in a cache memory, the possible scenarios are very similar to the previously explained when affected the main memory: if the portion of the cache memory affected didn't have been changed previously by normal computation, the memory controller can ask for a new copy of it to the main memory (or to the upper cache level) and restore the previously known state of it.

But, in the case of this portion of the cache memory has been changed by normal computation, the operating system may stop the operation of the application or raise an error.

2.5.3 Memory access violation

A soft error affecting a memory pointer can make an application try to get or put data, or even to jump into a memory space that isn't valid in the application's scope. Modern processors have protection against this kind of behavior, raising access violation errors to the operating system.

Once noticed that an application is trying to access a memory location that doesn't exist or is of another process, the operating system stop the application execution avoiding propagating the error to other parts of the system.

2.5.4 Change on a value

A single faulty bit is capable of generating a soft error that affect a component operation by changing its expected result into an unexpected one.

This is the case of errors affecting internal processor components, for example. Registers, pipeline, arithmetical logic unit (ALU), floating point unit (FPU), and almost all components of a modern processor have some kind of memory portion (to store intermediary results of its operations) and have some kind of buses to communicate to the other processor components. All these auxiliary memories and internal buses are possible targets of a transient fault.

A soft error in an internal component of a processor may pass unnoticed in system's operation if the component didn't have protection and if its result didn't violate the application address-space and didn't produce an invalid operation.

The SDC, the most common effect in this kind of outcome of soft errors, won't stop any application execution and might only be noticed by the users if the final result generated by the application differs significantly from the usual result.

2.6 Protection Mechanisms and Efficiency

Measuring the amount of SDC a program execution might produce once running over a given processor architecture in presence of transient faults is possible to evaluate the program's robustness against transient faults. As lower the amount of corruptions found in the program execution, more we can recognize it robustness.

However, in [Reis et al., 2005] the authors evaluated some software based fault detection/tolerance mechanisms and noticed the overhead in the program execution time could be prohibitive.

So, [Reis et al., 2005] proposed a new metric, the Mean Work to Failure (MWTF), that consider balance between the benefit of the improvement in the program robustness taking into account the overhead produced by the improvement mechanism.

Chapter 3
Fault Injection

Fault injection is mostly used for system dependability evaluation [Sosnowski et al., 2006]. It was by the need to validate dependability properties of the fault tolerant systems that the research about fault injection grew in importance [Kanawati, Kanawati and Abraham, 1995].

A dependable computer system is capable of detect errors due to hardware or software faults, isolate the errors cause (when possible) and recover from them [Kanawati, Kanawati and Abraham, 1995]. In the case of the soft errors, there is no need to isolate the errors cause because it is a transient situation that happened sometime before the detection and probably won't happen again the same way.

To obtain more confidence of the dependable properties of a system before its deployment is important to do a validation by testing the system in the presence of faults.

As the faults expected by the system dependability properties may not happen often, it is a common practice to put the system under testing (SUT) in an environment with fault injection campaigns. With fault injection, the system under test can be evaluated in faulty conditions even if the real fault doesn't happen.

A fault injection campaign consists of a set of experiments on the target system with specific workloads, injecting a specific fault (or set of faults) at specific trigger conditions. The target system behavior can be monitored and information (such as results of application execution and operating system logs) can be recorded as comprehensively as necessary and possible, to understand and evaluate the effects of the injected faults [Fidalgo, Alves and Ferreira, 2006].

3.1 Fault Injection Techniques

Fault injections could be hardware-based or software-based and there are four major techniques for fault injection as shown in Figure 8: physical fault injection

software simulation, hardware emulation and fault emulation [Yu, Garzaran and Snir, 2009].

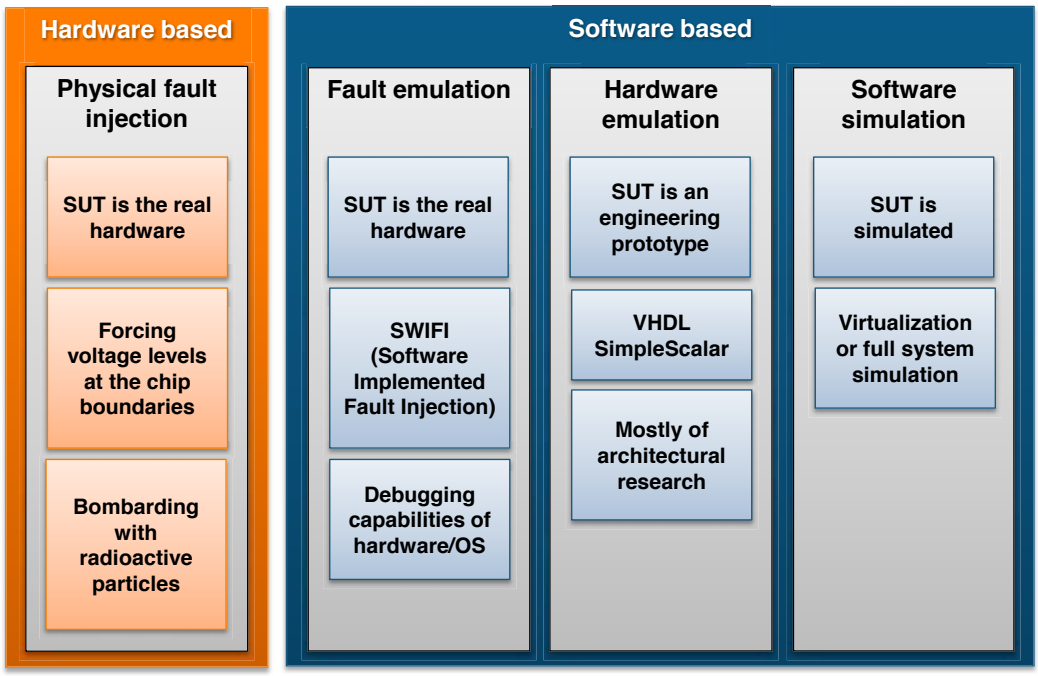

Figure 8 – Fault injection environments.

3.1.1 Physical Fault Injection

In the beginning of the studies about transient fault using fault injection, because of the nature of the problem relative to energy particles striking processor and memories, the approach to do the fault injection was using physical devices.

In the case of physical fault injection, the SUT is the actual tested fully functional computer with its operating system and applications.

The fault injection device could be an electronic device that generates disturbances into processor or memory pins or a device that exposes the SUT to radiation.

The problem with physical fault injection is that it is very unpredictable and probably untraceable. Once made an injection, it is very difficult to verify what memory or part of the processor was affected. Also, the necessary infrastructure to deal with physical fault injection (like a radiation generator device) isn't easy to obtain.

Even unusual, there is space for physical fault injection campaigns. Not only to evaluate the computer system robustness in very hostile environments, like the outer

space, but also to test how a computer system device will behave against radiation and in non-optimal environmental operation conditions.

For example, IBM has a published test with its Power 6 processor soft error tolerance using proton irradiation [Kudva et al., 2007]. In this analysis, the authors have listed the architectural characteristics of IBM Power 6 processor and how those characteristics affect the robustness of the processor against soft errors. The faults were injected using a proton beam and observing its effects using an architectural verification program.

Another example is the work performed in [Constantinescu, 2005] answering the question: "Is silent data corruption a real threat?" In their work, the author tested ten prototypes systems with operating temperature varying from -10ºC to 70ºC, and with nominal voltage from -10% to +10% of the nominal voltage. The results were impressive, with 9 of 10 of the tested systems presenting SDC.

Also, when testing the systems with electrostatic discharging [Constantinescu, 2005] the author achieved a corrupted result provided by a machine in a simple operation of on-line adding, removing or replacing a hot-swappable disk drive.

3.1.2 Fault Emulation

In the case of the fault emulation, as in the case of the physical fault injection, the SUT is a fully functional computer system.

Using processor and operating system debugging capabilities, the fault injector, usually a concurrently running software specially developed to this purpose, stops the execution of the application being tested, do the fault injection by changing processor registers or the memory of the process running the tested application and then let the process continue its operation.

One of the problems with the fault emulation approach is that the fault injector is a concurrent process of the tested application. Because of this, it indirectly affects the application being tested. As more complicated is the fault injection condition, tested frequently by the fault injector to assure the correct moment to do the fault injection, more influence into the tested application it generates.

A common approach using fault emulation is grid computing fault tolerance research, where in major cases the fault injection is done by killing a working process after some time. A simple operation (killing a process) with a simple trigger (after some time of application execution) may not influence the application behavior significantly, but limit the scope of the fault injection in transient faults studies.

For example, FAIL-FCI is a fault injection architecture designed for testing fault tolerance in grid computing [Hoarau, Tixeuil and Vauchelles, 2007]. It is composed by

three important parts: a compiler, which interprets the fault injection scenario and generates a fault injection configuration file; a library, which distributes the fault injection configuration file into the grid nodes; a daemon that interprets the fault injection configuration file and compiles a fault injection tool specific for each grid node. The fault injection scenario is described using FAIL, a language for fault description capable of expressing complex and realistic fault scenarios.

3.1.3 Hardware Emulation

Fault injection using hardware emulation is used when testing changes into processor and memory hierarchy architecture.

In this case, an emulator of the desired architecture is used, and the SUT is a partial implementation of a computer system, often without operating system, because those emulators don't perform to do full system emulation with actual operating system and actual applications.

Emulators based on VHDL models are used to propose techniques to let processors and memory more robust against transient faults by changes in their internal architectures. The experiments using those emulators and fault injection also often use precompiled benchmark applications because of the difficulty to emulate an environment capable of compile actual applications.

For example, MEPHISTO is an environment for fault tolerance experiments based on VHDL hardware description language [Jenn et al., 1994]. MEPHISTO was designed to estimate the coverage of fault tolerance mechanisms, to investigate different mechanisms for mapping results from one level of abstraction to another and to validate fault and error models applied during fault injection experiments. It uses changes into the VHDL model replacing some components to fault injectable ones and interacts with the VHDL simulator to apply the fault injection.

3.1.4 Software Simulation

The software simulation approach uses full system simulators to simulate a fully featured actual computer system with all its components executing actual operating systems and applications.

In this case, the SUT is a simulated computer system with its operating system and applications.

As those full system simulators are fast enough to allow interactive execution, it is possible to use a simulator to describe the SUT desired, install an actual operating system (like Linux or Windows) into this simulated computer, install applications and frameworks into this operating system.

As the fault injection using software simulation is done outside of the SUT, it doesn't affect the application being tested. The fault injector can stop the simulation, evaluate the trigger condition, operate a fault injection and then restart the simulation. For the simulated computer system is like the time didn't stopped.

Dealing with a full system simulator implies that the fault injector can use lots of information to evaluate fault trigger conditions: from processor registers state to memory hierarchy events, the fault injector can even do a very detailed depuration of the fault injection conditions because it won't affect the tested application at all.

A major step toward the development of fault-tolerant computer systems is the validation of the dependability properties of these systems. Fault/error injection has been recognized as a powerful approach to validate the fault tolerance mechanisms of a system and to obtain statistics on parameters such as coverages and latencies.

After describing a methodology for flexible software based fault and error injection, [Kanawati, Kanawati and Abraham, 1995] presented a tool called FERRARI, that incorporates a technique to emulate transient errors and permanent faults in software that were described in detail. The FERRARI tool was Unix based and didn't used any simulator, allowing the fault injection in actual programs running over the operating system.

Designed to be modular and portable, NFTAPE [Stott et al., 2000] provided mechanisms for fault-injection, triggering injections, producing workloads, detecting errors, and logging results, but, unfortunately, it was also very intrusive and did not scale well for large workloads.

Intending to be able to perform precise fault injections in current HPC mostly used processor architecture, [Gramacho, 2009] developed an extension of HP's and AMD joint full system simulation environment, named COTSon [Argollo et al., 2009], that allowed the injection of faults that change a single bit in processor registers and memory of a simulated computer. With the developed fault injection system the authors were able to evaluate the effects of single bit flip transient faults in an application, analyze an application robustness against single bit flip transient faults and validate some fault detection mechanism and strategies.

Chapter 4
Evaluating a Program Robustness

Experimental methods of injecting transient faults into a program during its execution were proposed to test detection and protection mechanisms against transient faults. On those methods, the program being evaluated is executed in an environment able to inject a fault in a form of a bit flip on a program architectural state (usually a bit in a processor register). At the end of the program execution, its result is evaluated to check the effect caused by the fault into the execution.

The program architectural bits changed by the fault injections on executions where the program finished correctly and presented the same result of a fault free execution were classified as unACE (unnecessary for an Architecturally Correct Execution).

On the other hand, the program architectural bits changed by the fault injections on executions where the program didn't finished correctly, or presented a result different of the fault free execution one, were classified as ACE (necessary for an Architecturally Correct Execution).

If the program being evaluated has some kind of fault detection mechanism against transient faults the program architectural bits changed may trigger the fault detection mechanism and lead the program to a fail stop avoiding the propagation of the fault effect in the program execution. On those cases, instead of being classified as ACE, as the execution finished doing a fail stop and noticed that a fault happened the program architectural bit changed is classified as DUE (Detected Unrecoverable Error).

As changes in the ACE program architectural bits lead to an abnormal program behavior and also could lead to a result different of the obtained by a fault-free execution, it is common to classify those bits as SDC (Silent Data Corruption).

To evaluate how reliable a program is in presence of transient faults with a sufficient large amount of executions with fault injection we can divide the amount of executions that didn't failed (those in which the program architectural bit changed was classified as unACE or DUE) by total amount of executions with fault injection

performed. Also, it is important to have a good distribution in which program architectural bit is changed on each execution, since it is randomly chosen.

4.1 About the Amount of Executions

The authors of [Nicolescu, Savaria and Velazco, 2003] proposed a soft error detection mechanism based on source code transformation rules. The new program (compiled with the source code transformed with the fault detection mechanism) has the same functionality as the original program but able to detect bit-flips in memory and processor registers during an execution. The transformation is done using a tool made by the authors called C2C Translator.

Evaluating programs with and without their fault detection mechanism, the authors of [Nicolescu, Savaria and Velazco, 2003] perform a set of fault injection experiments where on each execution a bit is flipped in processor registers, program code memory region or program data memory region.

Each execution can be classified as: effect-less if the injected fault didn't affect the program behavior; software detection if the fault detection mechanism was triggered by the injected fault; hardware detection if the fault injected triggered an hardware fault detection mechanism; loss sequence if the program triggered the time-out condition of an execution (the program was trapped in an infinite loop); incorrect answer if the injected fault wasn't detected and the program produced a result different of the expected.

In order to obtain realistic results, the authors of [Nicolescu, Savaria and Velazco, 2003] scaled the amount of executions with fault injection based on the total amount of processor cycles needed to execute the program chosen to be evaluated.

A total of 15208 executions with fault injection were performed with the original program (without the proposed fault detection mechanism): 8000 to evaluate fails in processor registers (0.65% of the total amount of cycles); 2208 to evaluate program instructions (two executions per program instruction); and 5000 to evaluate program data region (approximately 2.5 executions per byte used by the program).

With the program modified to detect soft errors the authors of [Nicolescu, Savaria and Velazco, 2003] performed 37520 executions with fault injection: 37520 to evaluate fails in processor registers (also 0.65% of the total amount of cycles); 8000 to evaluate program instructions (also two executions per program instruction); and 10000 to evaluate program data region (also approximately 2.5 executions per byte used by the program).

A total of 52728 executions with fault injection were performed in [Nicolescu, Savaria and Velazco, 2003] to evaluate two programs (the original one and the changed to detect soft errors), 26364 executions per program on average. The authors gave no information about how many time was spent on executions.

In Error Detection by Duplicated Instructions (EDDI) [Oh, Shirvani and McCluskey, 2002], the authors increased the robustness of programs during compilation, copying instructions but using different processor registers and adding verification for errors by comparing the value of the original processor register used by the program with the value of the processor register used in the new generated instruction. The overhead added this fault detection mechanism by the new instructions and verifications keep below the double of the time spent by the original program to execute by taking advantage on Instruction Level Parallelism (ILP) characteristics of superscalar processors. So, the time spend by the program with the fault detection mechanism is often less than two times the original program execution time.

The authors ensure that EDDI can detect errors in functional units, in control logic or in communication buses with the processor, even being designed to detect single bit flip in memory.

By using fault injection on code segment, the authors of [Oh, Shirvani and McCluskey, 2002] used a simulator to execute eight benchmarks, 500 times each. On each simulation a fault was injected at a randomly chosen time.

On the same work, the authors of [Oh, Shirvani and McCluskey, 2002] evaluated a fault detection mechanism based on duplication at the application source code using two different techniques. For each source code based fault detection mechanism, the authors had to run the evaluated benchmarks with more simulations.

Executing a total of four evaluations (the original program, the program with EDDI and the program with each of the source code based fault detection mechanism) per each of the eight benchmarks evaluated and executing 500 simulations with fault injection per evaluation, the authors of [Oh, Shirvani and McCluskey, 2002] have done a total of 16000 simulations to accomplish their work. Again, the authors gave no information about how many time was spent on simulations.

On Software-Controlled Fault Tolerance [Reis et al., 2005], the authors present a set of transient fault detection techniques based on software and also hybrid (based on software and hardware). Each of the proposed techniques has a different cost/benefit relation by improving robustness or performance.

The first technique presented by [Reis et al., 2005] if SWIFT (Software Implemented Fault Tolerance) which increases an application robustness during compilation time. SWIFT transformations of program insert redundant code using

different processor registers and also insert validations only before flow control instructions and writes on memory.

The other techniques presented on [Reis et al., 2005] are all hybrid. The set of those hybrid techniques is called CRAFT (Compiler-Assisted Fault Tolerance). In general, they increase the robustness even more than SWIFT and also improve the performance of the program in comparison with software-only fault tolerance techniques. CRAFT techniques are based on SWIFT and also redundant multi thread (RMT) fault tolerance works.

The authors of [Reis et al., 2005] noticed that it was possible to adjust the frequency of verifications used on both SWIFT and CRAFT, compromising a little of robustness by gaining on program performance. To evaluate this balance (robustness per performance), the authors started to fine grainy control the type of protection used and the amount of protection inserted and used Mean Work to Failure (MWTF) metric to compare the results. So, profiles were created to each application in order to define where and how to apply a fault tolerance technique and to activate the redundant execution. To this profiling technique they called PROFiT (Profile-Guided Fault Tolerance).

To evaluate application robustness with and without the proposed fault tolerance mechanisms, the authors of [Reis et al., 2005] executed fault injection experiments in a simulator executing all programs to the end (they didn't use partial execution and verification of intermediate program states) using functional simulator and choosing when and where to inject the fault randomly. The fault injection was done by flipping a single bit. The authors classified fault injection simulation result in three ways: unACE if the flipped bit wasn't necessary to the correct architectural execution; DUE if the flipped bit triggered a fault detection mechanism; or SDC it the flipped bit generated a silence data corruption.

The authors of [Reis et al., 2005] used one benchmark to evaluate how many fault injections should be necessary to have a significant statistical approximation of one program evaluation. They executed 5000 fault injection simulations with the selected benchmark and observed that the confidence interval of the average of the cases classified as SDC was ±2.0% after 946 simulations, ±1.5% after 1650 simulations and ±1.0% after 3875 simulations. As they noticed that the SDC average stabilized fast they assumed that 1000 fault injection simulations would be enough the evaluate a program robustness with each fault tolerance technique, and this number could be increased to achieve greater precision.

In a total of 10 sets of experiments (without fault tolerance, three variations of SWIFT, three variations of CRAFT and three variations of PROFiT), the authors of [Reis et al., 2005] evaluated the robustness of a subset of benchmarks from SPEC CPU

2000, SPEC CPUINT95 and MediaBench by simulating 5000 executions with fault injection (except for two SWIFT variations that used 1000 simulations). In each of 504000 simulated executions with fault injection a randomly chosen bit of one of 127 integer processor registers of IA64 processor architecture was flipped.

Because of the use of a simulator to execute de program with a fault injection, the authors of [Reis et al., 2005] could save some simulation time on the executions where the bit flipped was classified as unACE. On those cases, the simulation could be interrupted when the simulator observed that the flipped bit was re-written with results from processor logical unit or with a read operation before having it content used.

Continuing their research in fault tolerance for transient faults, the same authors of [Reis et al., 2005] proposed Spot [Reis et al., 2006] a technique to dynamically insert redundant instructions to detect errors generated by transient faults. This dynamically insertion was made in runtime using instrumentation.

Besides using a different architecture from previous work (in [Reis et al., 2006] they use a Intel Pentium D instead of Itanium and protect only the eight general purpose 32 bit registers of the architecture), in the new work the authors didn't use simulators anymore. All the analysis and fault injection work is done using an instrumentation tool.

By evaluating 16 benchmarks, the authors of [Reis et al., 2006] executed a total of 1.03 million fault injections to obtain their results (keeping 5000 executions with fault injection per benchmark and configuration evaluated).

In ESoftCheck [Yu, Garzaran and Snir, 2009], the authors created a fault detection mechanism based on SWIFT that analyses a program during compilation and remove those verifications (with redundant code) that it assume that are unnecessary to program robustness. In this way, they achieve with a defined method (there are no need of profiling as in PROFiT) a program with high protection level and low performance overhead.

To evaluate their results the authors of [Yu, Garzaran and Snir, 2009] have used the Intel Pentium 4 processor and executed 2000 fault injection experiments per benchmark evaluated (in a total of 12) and configuration evaluated (in a total of six). The program evaluated was executed until the end on each of 144000 program executions, changing a randomly chosen bit of a randomly chosen register at a randomly chosen execution time. By the end of each execution, the program output was evaluated to classify the result and evaluate de program robustness.

In all related work studied, the execution of a program in a transient fault injection environment was classified using basically three labels: unACE, DUE and SDC. To compute a program robustness using fault injection we only need to divide the amount of unACE cases added with the DUE cases by the amount of executions made in the

experiment. If all executions are classified as SDC, the robustness will be zero (the minimal robustness allowed). On the other hand, if all executions are classified as unACE or DUE, the robustness will be one (the maximum robustness allowed).

Figure 9 – Amount of executions with fault injections made in related work presented.

As shown in Figure 9, the robustness evaluation method using program executions with fault injection need a sufficient large amount of executions varying the fault conditions (time, register and bit) to have a representative statistical approximation of the results.

Also, we know that by using a fault injection based evaluation of robustness, the amount of executions to evaluate a program will affect the precision of robustness obtained [Reis et al., 2005].

Finally, using simulators or dynamically instrumentation to inject fault on every program execution will increase time needed on each execution in comparison with the time spent by the program running directly in the architecture without instrumentation.

The Program Vulnerability Factor (PVF) metric [Sridharan and Kaeli, 2008] was evaluated in [Sridharan and Kaeli, 2009] and the authors showed that PVF could be used as a software only based metric for measuring the effects of transient faults into program executions instead of using Architectural Vulnerability Factor (AVF) proposed in [Mukherjee et al., 2003].

PVF [Sridharan and Kaeli, 2008] and robustness against transient faults [Gramacho, Rexachs and Luque, 2011] are very similar concepts, but opposites by definition. Where PVF quantifies the amount of vulnerabilities of a program, robustness quantifies the portions of program executions that cannot be affected by a transient fault. As metrics, a program robustness is equivalent to the inverse of the same program PVF as show in Equation 4.

Equation 4 – Robustness and PVF relation.

$$\text{Robustness} = \frac{1}{\text{PVF}}$$

However, the PVF evaluation presented by the authors of [Sridharan and Kaeli, 2009] used simulators and samples of program executions, still estimating part of the PVF for the evaluated programs.

Evaluating the PVF concept, the authors of [Döbel, Schirmeier and Engel, 2013] found some limitations with the implementation of practical tools to calculate a program PVF. Because of these limitations, they only evaluated one program compiled for the x86 processor architecture with about 3.7 millions of instructions executed.

However, the comparison of the results of their limited PVF/robustness evaluation with an exhaustive fault injection campaign demonstrated that these software based concepts may serve as a starting point for estimating the vulnerability of programs against transient faults.

In [Hari et al., 2012] and [Hari, Adve and Naeimi, 2012] the authors present a comprehensive work about pruning techniques to reduce the amount of fault injections experiments needed to evaluate a program robustness against transient faults.

Although using fault injection in very specific portions of the evaluated programs, their experimental evaluation presented some limitations by not taking into account dynamic linked libraries or float point registers.

4.2 A Sample Program Evaluation

In order to provide a glimpse of what is to evaluate a program robustness against transient faults using fault injections we prepared a simple example.

The processor architecture we choose for this simple example was the 65C02, primarily designed as a CMOS replacement for the 6502 processor and best known by powering the Apple IIc and later the Apple IIe computer systems [Eyes and Lichty, 1992].

As shown in Table 2, the 65C02 processor have one 8-bit accumulator register (A), two 8-bit index registers (X and Y), seven one-bit processor flags (Carry Flag, Zero Flag, Interrupt Disable, Decimal Mode, Break Command, Overflow Flag and Negative Flag), and one 8-bit stack pointer (SP) that we consider in our evaluation.

Table 2 – 65C02 processor registers and their sizes.

Register	Accumulator	X Index	Y Index	Stack Pointer	Carry Flag	Zero Flag	Interrupt Disable	Decimal Mode	Break Command	Overflow Flag	Negative Flag	Total
Size (in bits)	8	8	8	8	1	1	1	1	1	1	1	39

4.2.1 Exponentiation Program

We developed a simple exponentiation program with only 21 program instructions in its source code shown in Figure 10.

```
1   .ORG $0200    LDX exponent    ; Load the exponent operand into X
2                 BEQ PZERO       ; If it is zero, the result will be one
3                 DEX
4                 LDA base        ; Load the base operand into accumulator
5                 BEQ ZERO        ; If it is zero, the result will be zero
6                 STA result      ; Store the accumulator into the result
7   MULT1:        STA mult        ; Store the accumulator for multiplication
8                 LDY base        ; Load the base operand into Y
9                 DEY
10  MULT2:        CLC             ; Clear carry
11                ADC mult        ; Add the mult. result to the accumulator
12                DEY
13                BNE MULT2       ; Jump if is still multiplying
14                DEX             ; Decrement X register (exponent)
15                BNE MULT1       ; Jump if is still operating the exponentiation
16                JMP FINISH
17  PZERO:        LDA #$01        ; The result is 1 (zero on exponent operand)
18                JMP FINISH
19  ZERO:         LDA #$00        ; The result is 0 (zero on base operand)
20  FINISH:       STA result      ; Store the result in byte labeled result
21                BRK             ; Finish running the program
22  base:         .DB  $05        ; Base operand
23  exponent:     .DB  $03        ; Exponent operant
24  result:       .DB  $00        ; Result of the operation
25  mult:         .DB  $00        ; Auxiliary variable to multiplication
```

Figure 10 – Exponentiation program source code.

We executed the exponentiation program in a 65C02 simulator. It executed a total of 51 instructions to perform the exponentiation with the input data of 5 as base operand and 3 as exponent operand. The expected result of 125 was correctly stored into the assigned memory location.

The fault injection space for the exponentiation program running in the 65C02 processor architecture was 1989 based on the amount of instructions executed and the amount of bits on all registers evaluated for the processor architecture.

Equation 5 – Fault injection space for the exponentiation program.

$$fault\ injection\ space = (instructions \times registers\ bits)$$
$$fault\ injection\ space = (51 \times 39) = 1989$$

With this small amount of executions with fault injection it was possible to perform an exhaustive fault injection campaign to evaluate the exponentiation program robustness against transient faults. Figure 11 presents a map with all 39 processor registers bits and all 51 program instructions traced and the respective result of the execution with fault injection.

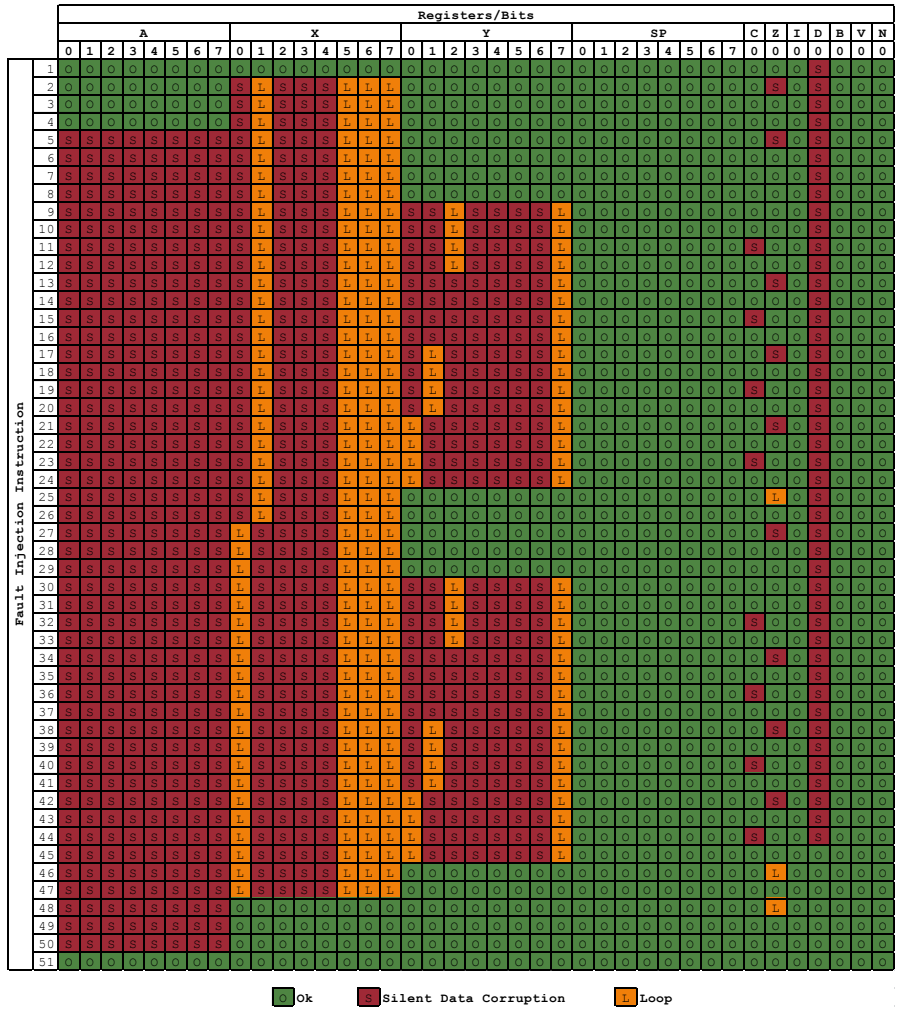

Figure 11 – Exponentiation program fault injection results map.

Figure 12 shows the robustness obtained in this exhaustive fault injection campaign. Even classifying the results as **ok** (the program finished and presented the correct result), **SDC** (the program finished but presented an incorrect result) and **loop**

(the program didn't finished after executing ten times the expected amount of instructions), for the robustness evaluation both **SDC** and **loop** are considered as no robust. In the left (Program) is the whole program (all registers) robustness. The remaining columns present the robustness of each processor register evaluated.

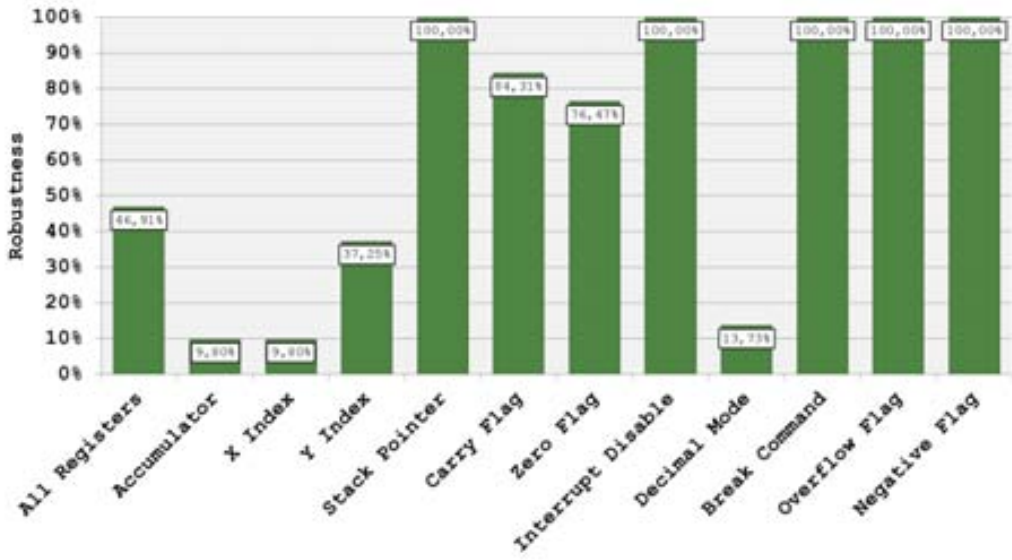

Figure 12 – Robustness of exponentiation program using fault injection.

As three of the four 8-bits registers evaluated had a low robustness, the whole program robustness scored below 50%. Also, the Decimal Mode flag, once changed by a fault injection, altered the processor's behavior in the arithmetic operations such as the one presented in line 11 of the program source code in Figure 10.

4.2.2 Improved Exponentiation Program

In order to improve our exponentiation program robustness against transient faults we changed its source code, protecting X index and Y index registers. As our protection mechanism used the processor stack, the Stack Pointer became a problem from the robustness point of view, and it was protected as well.

The protection mechanism was based on a simple software based duplication and verification proposed by [Nicolescu, Savaria and Velazco, 2003].

```
1   .ORG $0200   TSX             ; Load the stack pointer into X
2                STX spcheck     ; Store X into stack pointer check variable
3                LDX exponent    ; Load the exponent into X register
4                LDY exponent    ; Load the exponent into Y register
5                BEQ PZERO       ; If it is zero, the result will be zero
6                DEX
7                DEY
8                LDA base        ; Load the base operand into accumulator
9                BEQ ZERO        ; If it is zero, the result will be zero
10               STA result      ; Store the accumulator into the result
11  MULT1:       PHX             ; Push X register into stack
12               PHY             ; Push Y register into stack
13               STX px          ; Stores X register into px variable
14               CPY px          ; Compare Y register with px variable
15               BNE FAULT       ; Jump if X <> Y to FAULT
16               STA mult        ; Store the accumulator for multiplication
17               LDY base        ; Load the base operand into Y
18               LDX base        ; Load the base operand into X
19               DEY
20               DEX
21  MULT2:       CLC             ; Clear carry
22               ADC mult        ; Add the mult. result to the accumulator
23               DEY
24               DEX
25               BNE MULT2       ; Jump if is still multiplying
26               STX px          ; Stores X register into px variable
27               CPY px          ; Compare Y register with px variable
28               BNE FAULT       ; Jump to FAULT if X <> Y
29               PLY             ; Pull Y from stack
30               PLX             ; Pull X from stack
31               DEX             ; Decrement X register (exponent)
32               DEY             ; Decrement Y register (exponent)
33               BNE MULT1       ; Jump if is still operating the exponentiation
34               STX px          ; Stores X register into px variable
35               CPY px          ; Compare Y register with px variable
36               BNE FAULT       ; Jump to FAULT if X <> Y
37               TSX             ; Load X with the original stack pointer
38               CPX spcheck     ; Compare the original stack pointer with spcheck
39               BNE FAULT       ; Jump to FAULT if X <> spcheck
40               JMP FINISH
41  PZERO:       LDA #$01        ; The result is 1 (zero on exponent operand)
42               JMP FINISH
43  ZERO:        LDA #$00        ; The result is 0 (zero on base operand)
44  FINISH:      STA result      ; Store the result in byte labeled result
45               BRK             ; Finish running the program
46  FAULT:       LDA #$FF        ; Load the accumulator with the error code
47               STA result      ; Store the error in byte labeled result
48               BRK             ; Finish running the program by fault detection
49  base:        .DB  $05        ; Base operand
50  exponent:    .DB  $03        ; Exponent operant
51  result:      .DB  $00        ; Result of the operation
52  mult:        .DB  $00        ; Auxiliary variable to multiplication
53  px:          .DB  $00        ; Result of the operation
54  spcheck:     .DB  $00        ; Stack pointer check variable
```

Figure 13 – Improved exponentiation program source code.

We executed the improved exponentiation program in the 65C02 simulator. It executed a total of 101 instructions to perform the exponentiation with the same input data of 5 as base operand and 3 as exponent operand. The expected result of 125 was correctly stored into the assigned memory location.

The fault injection space for the improved exponentiation program running in the 65C02 processor architecture was 3939.

Equation 6 – Fault injection space for the improved exponentiation program.

$$fault\ injection\ space = (instructions \times registers\ bits)$$
$$fault\ injection\ space = (101 \times 39) = 3939$$

With this still small amount of executions with fault injection it was possible to perform an exhaustive fault injection campaign to evaluate the improved exponentiation program robustness against transient faults. Figure 14 presents a map with all 39 processor registers bits and all 101 program instructions traced and the respective result of the execution with fault injection.

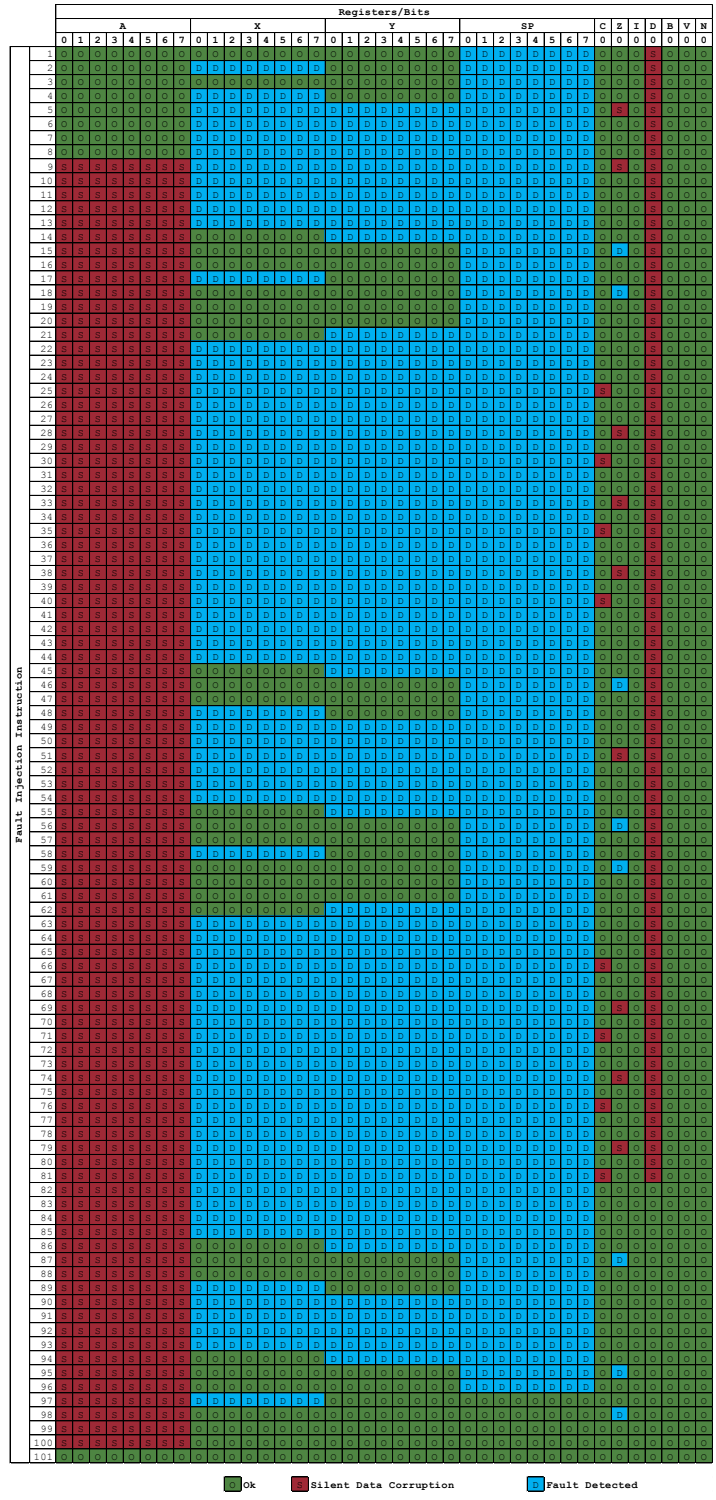

Figure 14 – Exponentiation program fault injection results map.

Figure 15 compares the robustness obtained in this exhaustive fault injection campaign of the standard program and of the improved one.

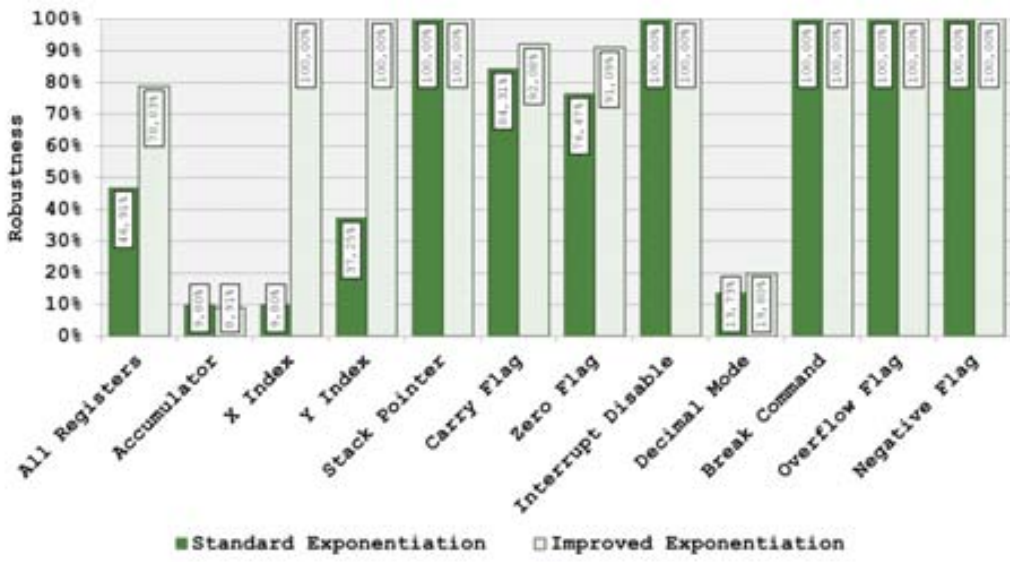

Figure 15 – Robustness of improved exponentiation program.

It is evident the robustness improvement of the program, jumping from 46,91% to 78,83%. In fact, all registers presented an improvement in their respective robustness, except for the Accumulator presenting a tiny (almost unnoticeable) decrease. All the protected registers in the improved program version (X, Y and Stack Pointer) topped 100% of robustness.

4.3 The Randomness Effect on the Fault Injection

Let's suppose now that we didn't want to evaluate the improved version exhaustively. How many executions with fault injection should be necessary to evaluate the robustness against transient faults of our program?

We used the confidence interval suggested by [Reis et al., 2005] of 2% to limit the amount of executions.

The next step was to choose a random algorithm and some random seeds to select where to inject the faults.

The algorithm selected was the Mersenne Twister [Matsumoto and Nishimura, 1998], a pseudo random number generator optimized for use with Monte Carlo simulations.

We used ten random sequences to compare themselves in respect of the amount of executions with a single fault injection until obtaining 2% of standard deviation of the evaluated robustness and also in respect of the error from the evaluated robustness exhaustively.

In Figure 16 we show that the more efficient seed we used (Seed5) was able to obtain the robustness with only 948 executions. The less efficient seed in this set of experiments (Seed3) took 3630 executions to obtain the robustness. The less efficient seed needed almost four times the amount of executions of the more efficient one. We cannot predict how efficient a random seed will be for our set of experiments before running the executions with fault injection.

The evaluated robustness for each random seed used is show in Figure 17 and the error of the evaluated robustness in comparison with the robustness obtained with an exhaustive fault injection campaign is shown in Figure 18. There is no direct relation between the amount of executions and the error obtained. We could think that a larger amount of experiments will always generate a robustness with lower error, but there is no evidence of this direct relation. The lower error we obtained (Seed1) wasn't the one that used the larger amount of executions (Seed3). Also, the larger error we obtained (Seed2) wasn't the one that used the smallest amount of executions (Seed5).

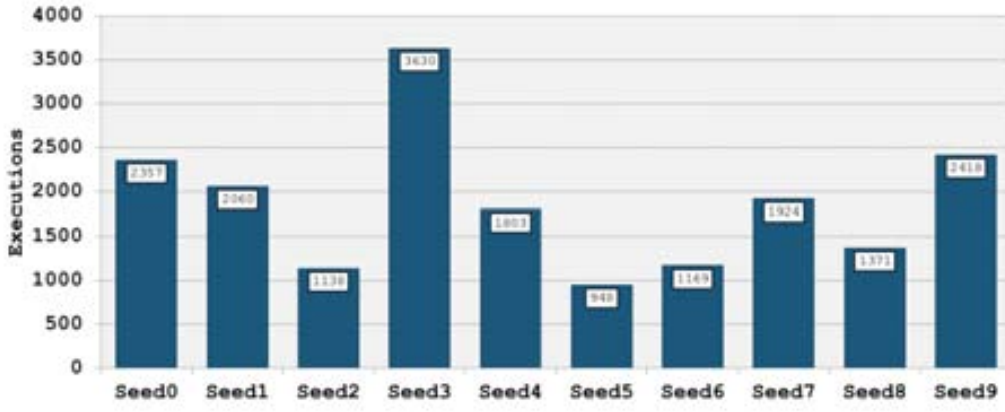

Figure 16 – Amount of execution with fault injection until 2% of standard deviation.

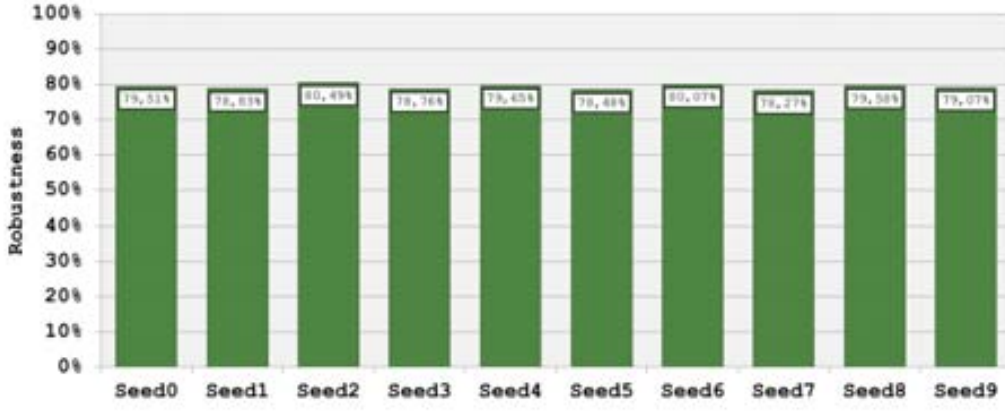

Figure 17 – Robustness evaluated until 2% of standard deviation.

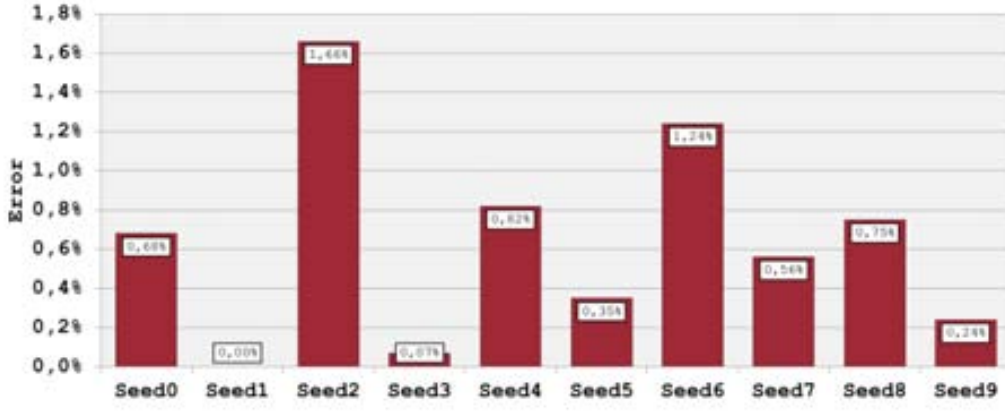

Figure 18 – Error in the evaluated robustness in comparison with the exhaustively obtained.

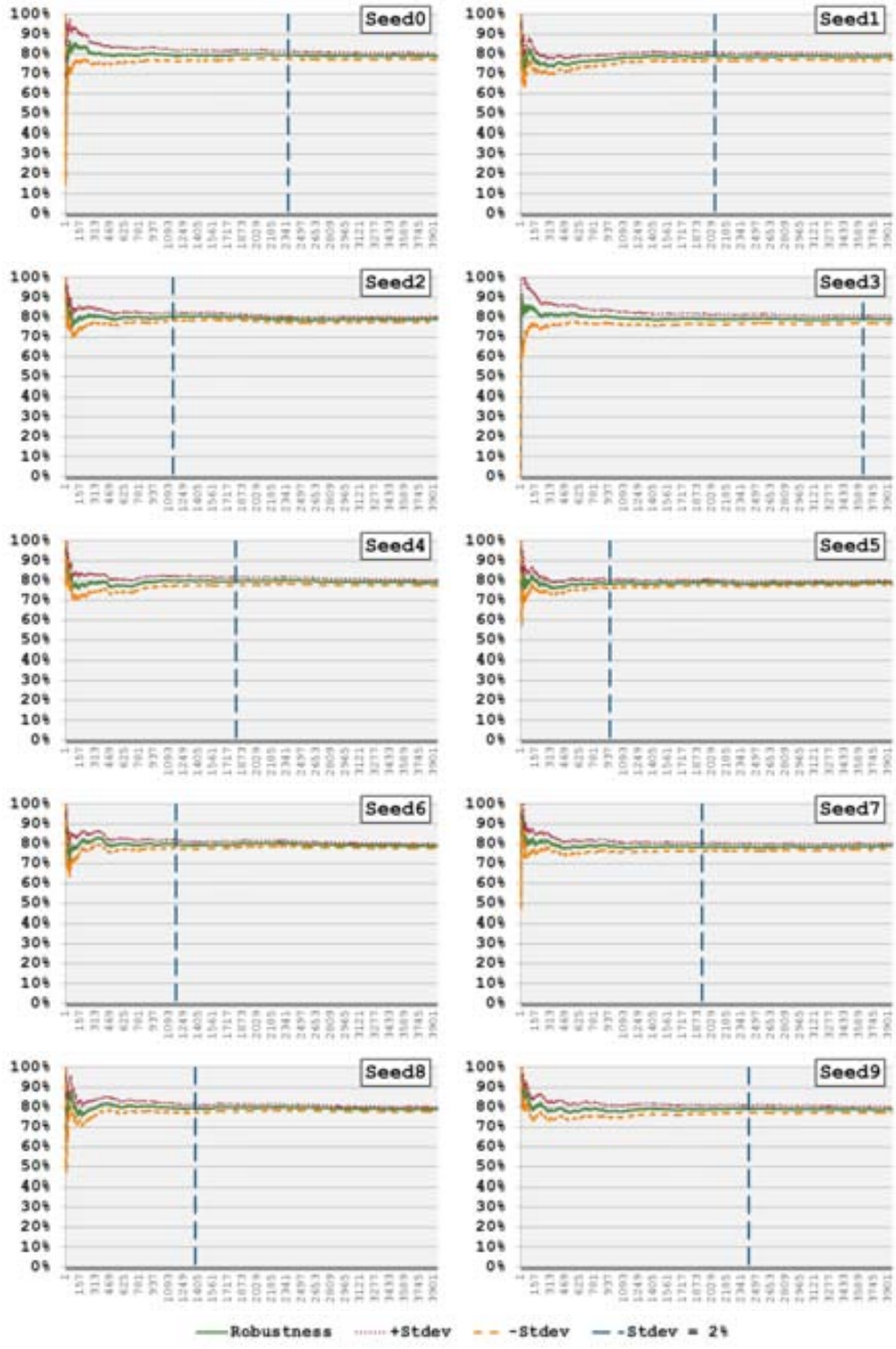

Figure 19 – Robustness average until 2% of standard deviation.

Chapter 5
The ARTFUL Methodology

To better understand the proposed methodology, it is necessary to take into account the concept of robustness against transient faults as the ability of a program, once in presence of a transient fault, to keep running and give a correct result when finish or to stop the execution when a soft error is detected and inform about it.

We consider that a program running over an determined architecture will have a robustness against transient faults represented as a number that can vary from zero (0%) to one (100%), where zero implies no robustness at all (the program fails on every possible cases) and one implies the best possible robustness (the program presents the correct result or detects the transient fault on every possible cases).

5.1 ARTFUL Methodology

The main objective of the ARTFUL methodology is to provide a deterministic way of evaluating a program robustness against transient faults when executed over a given architecture.

Designed to be used as a replacement for the fault injection campaigns, the ARTFUL methodology will provide definitions and formulas allowing a deterministic calculation of a program robustness corresponding an exhaustively evaluated robustness using fault injection campaigns.

In a robustness evaluation we classify the results of a single program execution with a single bit flip fault injection as unACE (no noticeable effects), DUE (the fault was detected by a fault detection mechanism) or SDC (the program failed to present a correct result), the Equation 7 presents the formula for the whole experiment evaluation.

Equation 7 – Total tested bits of a robustness against transient faults evaluation.

$$\frac{unACE\ bits + DUE\ bits + SDC\ bits}{amount\ of\ tested\ bits} = 1$$

The robustness can be calculated by separating the robust results (correct result or fault detected) from the non-robust results (SDC). The Equation 8 shows how the robustness can be calculated from the formula presented in Equation 7.

Equation 8 – Robustness formula based on unACE and DUE bits.

$$robustness = \frac{unACE\ bits + DUE\ bits}{amount\ of\ tested\ bits} = 1 - \frac{SDC\ bits}{amount\ of\ tested\ bits}$$

We need two things in order to know how many architectural state bits are in an evaluation (the amount of tested bits), one from the architecture and other from the program.

From the architecture we will need the amount of bits that could be changed by a fault during each processor instruction executed by the program.

From the program, we will need the amount of instructions it executes to produce its results.

Equation 9 – Generic amount of tested bits formula.

$$amount\ of\ tested\ bits = tested\ bits\ per\ instr \times amount\ of\ instr$$

As our methodology is based on Software Implemented Fault Injection (SWIFI) methods, the amount of bits that can be changed by a fault during each processor instruction executed by the program can be easily calculated by summing all processor register's size.

We define a set named $ProcReg_A$ with all processor registers that we will consider in our evaluation and a non-numerical finite sequence $RegSize_A$ representing the size in bits of each processor register in $ProcReg_A$. The relation between $ProcReg_A$ set and $RegSize_A$ sequence is defined by the function $f_{RegSize}$.

Equation 10 – Tested bits per instruction formula.

$$ProcReg_A = \{reg_1, reg_2, \ldots, reg_{nreg(A)}\}$$

$$f_{RegSize}: ProcReg_A \mapsto \mathbb{N}$$

$$RegSize_A = \left(regsize_{reg_1}, regsize_{reg_2}, \ldots, regsize_{reg_{nreg(A)}}\right)$$

$$tested\ bits\ per\ instr = \sum_{r=1}^{nreg(A)} f_{RegSize}(r)$$

In order to calculate the amount of instructions executed by the program we need at least one execution trace of the program running over the defined architecture.

A program execution trace is represented as a non-numerical finite sequence $Trace_{prog \times A}$ defined by the function f_{prog}. The f_{prog} function returns the processor

instruction executed by the program in a given point of its execution. The instruction must be a member of the $ProcIns_A$ set that contains all possible processor instructions.

Equation 11 – Amount of instructions in a program trace formula.

$$ProcIns_A = \{ins_1, ins_2, \dots, ins_{nins(A)}\}$$
$$f_{prog}: \mathbb{N} \mapsto ProcIns_A$$
$$Trace_{prog \times A} = \left(ins_1, ins_2, \dots, ins_{nins(Trace_{prog \times A})}\right)$$
$$amount\ of\ instr = nins\left(Trace_{prog \times A}\right)$$

At this point we have defined the first part of our formula to calculate a program robustness against transient faults when running over a given architecture:

Equation 12 – Preliminary robustness formula.

$$robustness_A = \frac{unACE\ bits + DUE\ bits}{nins\left(trace_{prog \times A}\right) \times \sum_{r=0}^{nreg(A)} f_{RegSize}(r)}$$

Let's consider now robust state as a property of a processor register in a given point of a program execution. This register property will be represented by a vector of logical states (true or false) with as many states as the amount of bits of the processor register, and it will be defined by the f_{rstate} function that we will explain better later.

Equation 13 – Robust state function definition.

$$f_{rstate}: ProcReg_A \times \mathbb{N} \mapsto \mathbb{B}$$

An element of a register robust state vector being true implies that the register bit represented by the element is classified as unACE in the given execution point of the program. In this way we know that any change in this register bit in this given execution point of the program won't be propagated to the final program result.

Similarly, an element of a register robust state vector being false implies that the register bit represented by the element is classified as ACE (we don't know yet if DUE or SDC) in the given execution point of the program. In this way we know that any change in this register bit in this given execution point of the program can be propagated to the final program result.

In order to know how many robust state vector elements are true (to know how many bits of a register robust state are classified as unACE in a given point of program execution) we will need the f_{abits} function. This function needs a logical states vector as input parameter and will return the amount of logical states of the vector that have its value as true.

Equation 14 – Active bits function definition.

$$f_{abits}: \mathbb{B} \mapsto \mathbb{N}$$

With the two previously presented functions we are able to present the single process version of our general robustness formula.

Equation 15 – Robustness of a single trace program execution over a given architecture.

$$robustness_{prog \times A} = \frac{\sum_{n=1}^{nins(Trace_{prog \times A})} \sum_{r=0}^{nreg(A)} f_{abits}\left(f_{rstate(r,n)}\right)}{nins\left(Trace_{prog \times A}\right) \times \sum_{r=0}^{nreg(A)} f_{RegSize}(r)}$$

Up to this point, a program's robustness against transient faults when running over a determined architecture will be the sum of the amount of bits classified as unACE of each processor register *r* of the given architecture *A* in every point *n* of the program *prog* execution present on trace $Trace_{prog \times A}$, divided by the sum of the amount of bits of each processor register multiplied by the amount of instructions present in the trace $Trace_{prog \times A}$.

5.1.1 Robust State

To define our f_{rstate} function we will use the method presented by [Reis et al., 2005] to save simulation time on those cases where the fault injection was applied in a processor register that had its value overwritten by a new one before any read of the content changed by the fault injection.

Table 3 – Basic block sample with processor instructions.

Address	Instruction	Register Use	unACE
0x401a40	mov r13, 0x3ff0000000000000	write on r13	r13, r12, r11, r10, r9
0x401a4a	mov r12, 0x3ff0000000000000	write on r12	r12, r11, r10, r9
0x401a54	mov r11, 0x3ff0000000000000	write on r11	r11, r10, r9
0x401a5e	mov r10, 0x3ff0000000000000	write on r10	r10, r9
0x401a68	mov r9, 0x3ff0000000000000	write on r9	r9
0x401a72	add ecx, 0x1	read and write on ecx	
0x401a75	mov qword ptr [rdx], r13	read on rdx and r13	
0x401a78	mov qword ptr [rdx+0x8], r12	read on rdx and r12	r13
0x401a7c	mov qword ptr [rdx+0x10], r11	read on rdx and r11	r13, r12
0x401a80	mov qword ptr [rdx+0x18], r10	read on rdx and r10	r13, r12, r11
0x401a84	mov qword ptr [rdx+0x20], r9	read on rdx and r9	r13, r12, r11, r10
0x401a88	add rdx, 0x28	read and write on rdx	r13, r12, r11, r10, r9
0x401a8c	cmp ecx, ebx	read on ecx and ebx	r13, r12, r11, r10, r9
0x401a8e	jnz 0x401a40		r13, r12, r11, r10, r9

In the example presented in Table 3 with a basic block sample of one of the programs used in our experimental evaluation, it is possible to notice that we can evaluate if a processor register's bits are important in a given point by only observing program's instruction sequence.

For example, as the at instruction in address 0x401a68 load the r9 processor register with a given value, any change done in r9 before the execution of this

instruction will be discarded. So, if a fault injection mechanism injects a fault on any of r9 bits between the executions of the instructions at address 0x401a40 and 0x401a68 the program result will not be affected and the bit changed by the fault injection will be classified as unACE.

On the other hand, after r9 be loaded by the instruction at address 0x401a40, as the loaded value will be used by the instruction at address 0x401a84, the r9 integrity must be kept in the interval between the load (write operation on register) and the use of the loaded value (read operation on register).

As the presented basic block represent a loop, while the program execution stay in the loop, the next instruction that will manipulate r9 after the execution of the instruction in address 0x401a84 will be one that will change its value (a write operation on r9, exactly the instruction at address 0x401a68) and, so, the integrity of the register value in this interval is no longer needed anymore.

By the previously analyzed situation we can assume that, once knowing that a register will have its value replaced by a new one (after a write operation on the register) the registers bits can be classified as unACE on every instruction executed before the one with the write operation, until an instruction that read the content of the register be found.

The easiest way to analyze the execution of a program in search of those relations between uses of processor registers in read or write operations is by looking the program execution trace $Trace_{prog \times A}$ in reverse order, beginning by the last executed program instruction and following the trace until the first program instruction executed.

In this way, every time we find an instruction that write content to a processor register we can turn the logical states of the register's robust state vector elements to true (classify as unACE) until find an instruction that read the register content.

On the other hand, every time we find an instruction that read content from a processor register we can turn the logical states of the register's robust state vector elements to false (classify as ACE) until find an instruction that write content on the register.

If a given processor instruction operates a register for both read and write (e.g. as an increment operation), as our analysis is done in program trace instructions backwards, we first evaluate the write operation and then the read operation.

In our methodology we also need to know how each processor instruction deals with processor register bits for read and write. So, we will use a set named $ProcInsReg_A$, which contains all ordered pairs of a processor instruction combined with a processor register.

Equation 16 – Processor instructions and registers relation set.

$$ProcInsReg_A = \{(ins_1, reg_1), \dots, (ins_{nins(A)}, reg_{nreg(A)})\}$$

For each pair in $ProcInsReg_A$ set we must have an element in two non-numerical sequences: *WrittenBits*, defined by the f_{wbits} function and *ReadBits*, defined by the f_{rbits} function.

The f_{wbits} function returns a vector of logical states with all states that represent processor register bits written by the instruction with true as value.

Equation 17 – Written bits function definition.

$$f_{wbits}: ProcIns_A \times ProcReg_A \mapsto \mathbb{B}$$

The f_{rbits} function returns a vector of logical states with all states that represent processor register bits read by the instruction with true as value.

Equation 18 – Read bits function definition.

$$f_{rbits}: ProcIns_A \times ProcReg_A \mapsto \mathbb{B}$$

Knowing how a processor instruction operated a given processor register for read and write, and also because our analysis is done by evaluating a program trace backwards, by the truth table presented in Table 4 we deduced a formula to the f_{rstate} of a given processor register in a given point of program trace.

Table 4 – Truth table of the f_{rstate} function

Previous Robust State	f_{wbits}	f_{rbits}	New Robust State	Description
TRUE	TRUE	TRUE	FALSE	Wasn't important, write on it, read from it, change to being important
TRUE	TRUE	FALSE	TRUE	Wasn't important, write on it, keep don't being important
TRUE	FALSE	TRUE	FALSE	Wasn't important, read from it, change to being important
TRUE	FALSE	FALSE	TRUE	Wasn't important, didn't operate, keep don't being important
FALSE	TRUE	TRUE	FALSE	Was important, write on it, read from it, keep being important
FALSE	TRUE	FALSE	TRUE	Was important, write on it, change to not important
FALSE	FALSE	TRUE	FALSE	Was important, read from it, keep being important
FALSE	FALSE	FALSE	FALSE	Was important, didn't operate, keep being important

For every processor register and for every program instruction except the last one, the robust state of a given register *reg* in a given point *n* of a program execution trace $Trace_{prog \times A}$ will be the result of the robust state of the next point in program execution trace (the previously analyzed instruction) operated with a logical OR with the bits written by the analyzed instruction and then operated with a local AND with the negation of the bits read by the analyzed instruction.

Equation 19 – Robust state formula for all but last trace instructions.

$$1 \leq n < nins(Trace_{prog \times A}); i = f_{prog}(n)$$
$$f_{rstate}(reg, n) = [f_{rstate}(reg, n+1) \vee f_{wbits}(i, reg)] \wedge \sim f_{rbits}(i, reg)$$

When the program finishes its execution we can assume that a change in any of processor registers won't affect the program result anymore. So, we define a function named $f_{endstate}$ that returns a vector with a robust state of a given register with all logical states as true (all register bits classified as unACE).

Equation 20 – End state function definition.

$$f_{endstate}: ProcReg_A \mapsto \mathbb{B}$$

The f_{rstate} function for each program executed instruction will need the robust state of the next executed program instruction (the previously analyzed program execution trace instruction). In the particular case of the last instruction executed by the program, the f_{rstate} will need the $f_{endstate}$.

Equation 21 – Robust state formula for the last traced instruction.

$$n = nins\left(Trace_{prog \times A}\right); i = f_{prog}(n)$$
$$f_{rstate}(reg, n) = [f_{endstate}(reg) \vee f_{wbits}(i, reg)] \wedge\sim f_{rbits}(i, reg)$$

With all the presented functions in this section it is possible to calculate a program's robustness against transient faults when executed over a determined architecture by calculating the precise amount of unACE bits of the program execution trace. This calculation, by the presented methodology, can be done in a single loop evaluating every program trace instruction backwards.

5.1.2 Going Multi Processes

After performing a good number of evaluations of serial programs, as the main field of our research is High Performance Computing (HPC) and most of the problems in HPC are solved by parallel programs (either by multiple processes or threads in a single computer or by processes distributed among distinct processing nodes), we aimed to extend our robustness concept to a program that have multiple execution threads.

In this case, each execution thread or process must generate its own trace.

For the robustness evaluation, we now take into account that the amount of instructions executed by the program is the sum of the amount of instructions executed by each thread/process.

Also, as how the threads/processes communicate each other is indifferent for our methodology (at the end it will all result in reading and writing in memory regions) we will calculate the robustness of each instruction of each trace generated by the program execution, consolidating all in a single program robustness.

So, we had to slightly change our robust state formula by adding the multiple trace possibility:

Equation 22 – Robust state formula for all but last trace instructions of a given program.

$$1 \leq n < nins\big(trace(t)\big); i = f_{prog}(t,n)$$
$$f_{rstate}(reg,t,n) = [f_{rstate}(reg,t,n+1) \vee f_{wbits}(i,reg)] \wedge\sim f_{rbits}(i,reg)$$

The last instruction of each program trace had also to be slightly changed:

Equation 23 – Robust state formula the last traced instructions.

$$n = nins\big(trace(t)\big); i = f_{prog}(t,n)$$
$$f_{rstate}(reg,t,n) = [f_{endstate}(reg) \vee f_{wbits}(i,reg)] \wedge\sim f_{rbits}(i,reg)$$

Finally, we had to change our general robustness formula to let it represent a multi threaded/processed robustness with all its components, including the possibility of multiple traces.

Equation 24 – Robustness of a multi trace program execution over a given architecture.

$$robustness_{prog\times A} = \frac{\sum_{t=1}^{ntraces(exec_{prog\times A})} \sum_{n=1}^{nins(trace(t))} \sum_{r=0}^{nreg(A)} f_{abits}(r,t,n)}{\sum_{t=1}^{ntraces(exec_{prog\times A})} nins\big(trace(t)\big) \times \sum_{r=0}^{nreg(A)} f_{RegSize}(r)}$$

For a serial program, the result of the given formula will be exactly the same as the formula presented in Equation 15.

5.2 Evaluating a Program Robustness with the ARTFUL Methodology

We prepared an evaluation to explain in details how the ARTFUL methodology can be used in practice. This evaluation will use the two programs evaluated with fault injections in section 4.2: the simple exponentiation program for the 65C02 processor architecture and the improved version against transient faults of the same program.

5.2.1 Information about the Processor Architecture

In order to evaluate a program robustness against transient faults using the ARTFUL methodology we must have some information about the processor architecture the program in running on.

The set of 65C02 processor registers is, as shown in Table 2:

Equation 25 – 65C02 processor registers set.

$$ProcReg_{65C02} = \{\mathrm{A}, \mathrm{X}, \mathrm{Y}, \mathrm{SP}, \mathrm{C}, \mathrm{Z}, \mathrm{I}, \mathrm{D}, \mathrm{B}, \mathrm{V}, \mathrm{N}\}$$

Also, as show Table 2, the 65C02 processor registers sizes are:

Equation 26 – 65C02 processor registers size sequence.

$$RegSize_{65C02} = (regsize_{A}, regsize_{X}, regsize_{Y}, regsize_{SP}, \dots, regsize_{N})$$
$$RegSize_{65C02} = (8,8,8,8,1,1,1,1,1,1,1)$$

So, for the 65C02 processor architecture, the amount of bits per instruction executed is:

Equation 27 – Bits per instruction of the 65C02 processor.

$$bits\ per\ instr = \sum_{r=1}^{nreg(65C02)} f_{RegSize}(r)$$
$$= f_{RegSize}(A) + f_{RegSize}(X) + f_{RegSize}(Y) + f_{RegSize}(SP) + f_{RegSize}(C) + f_{RegSize}(Z) + f_{RegSize}(I) + f_{RegSize}(D) + f_{RegSize}(B) + f_{RegSize}(V) + f_{RegSize}(N)$$
$$bits\ per\ instr = 8 + 8 + 8 + 8 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 39$$

The 65C02 processor architecture instruction set has 70 instructions. These instructions are described by a total of 212 distinct OpCodes depending on the registers used and the addressing mode [The Western Design Center, Inc., 1981-2003].

For our methodology purposes, we have to put in the processor instructions set the 212 distinct OpCodes, as each of them has its own way of dealing with processor registers.

Equation 28 – 65C02 processor instructions set.

$$ProcIns_{65C02} = \{BRK_{00}, \dots, BBS7_{FF}\}$$

Chapter eighteen of [The Western Design Center, Inc., 1981-2003] contains detailed information about each 65C02 OpCode. This detailed information contains, for example, the registes read by the processor to execute the instruction represented by an opCode and the registers written by the processor once executed the instruction represented by an opCode.

The above described information is enough to construct the 65C02 processor instruction set described in Equation 28.

5.2.2 The Exponentiation Program

Let's consider the robustness formula we presented in section 5.1 prepared to evaluate de robustness of our exponentiation program (*prog* = exp) running over the 65C02 processor architecture (*A* = 65C02):

We must obtain the program execution trace and some information about the processor architecture to calculate the amount of bits we will test (the same as the "fault injection space" in the fault injection experiments).

Figure 20 shows the trace generated with the simulator.

#	Address	OpCodes	Instruction
1	0x0200	AE 2F 02	LDX $022F
2	0x0203	F0 1D	BEQ $0222
3	0x0205	CA	DEX
4	0x0206	AD 2E 02	LDA $022E
5	0x0209	F0 1C	BEQ $0227
6	0x020B	8D 30 02	STA $0230
7	0x020E	8D 31 02	STA $0231
8	0x0211	AC 2E 02	LDY $022E
9	0x0214	88	DEY
10	0x0215	18	CLC
11	0x0216	6D 31 02	ADC $0231
12	0x0219	88	DEY
13	0x021A	D0 F9	BNE $0215
14	0x0215	18	CLC
15	0x0216	6D 31 02	ADC $0231
16	0x0219	88	DEY
17	0x021A	D0 F9	BNE $0215
18	0x0215	18	CLC
19	0x0216	6D 31 02	ADC $0231
20	0x0219	88	DEY
21	0x021A	D0 F9	BNE $0215
22	0x0215	18	CLC
23	0x0216	6D 31 02	ADC $0231
24	0x0219	88	DEY
25	0x021A	D0 F9	BNE $0215
26	0x021C	CA	DEX
27	0x021D	D0 EF	BNE $020E
28	0x020E	8D 31 02	STA $0231
29	0x0211	AC 2E 02	LDY $022E
30	0x0214	88	DEY
31	0x0215	18	CLC
32	0x0216	6D 31 02	ADC $0231
33	0x0219	88	DEY
34	0x021A	D0 F9	BNE $0215
35	0x0215	18	CLC
36	0x0216	6D 31 02	ADC $0231
37	0x0219	88	DEY
38	0x021A	D0 F9	BNE $0215
39	0x0215	18	CLC
40	0x0216	6D 31 02	ADC $0231
41	0x0219	88	DEY
42	0x021A	D0 F9	BNE $0215
43	0x0215	18	CLC
44	0x0216	6D 31 02	ADC $0231
45	0x0219	88	DEY
46	0x021A	D0 F9	BNE $0215
47	0x021C	CA	DEX
48	0x021D	D0 EF	BNE $020E
49	0x021F	4C 29 02	JMP $0229
50	0x0229	8D 30 02	STA $0230
51	0x022C	00	BRK

Figure 20 – Exponentiation program execution trace.

As our program execution over the given architecture produces only one execution trace, we can simplify the robustness formula to the following:

Equation 29 – Exponentiation robustness over 65C02 processor generic formula.

$$robustness_{\exp\times 65C02} = \frac{\sum_{n=1}^{nins(trace_{\exp\times 65C02})} \sum_{r=0}^{nreg(65C02)} f_{abits}(r,t,n)}{nins(trace_{\exp\times 65C02}) \times \sum_{r=0}^{nreg(65C02)} f_{RegSize}(r)}$$

Our exponentiation program execution trace has 51 instructions executed. We can now replace this information in the left side of our formula denominator:

Equation 30 – Exponentiation robustness over 65C02 processor with traced instructions.

$$robustness_{\exp\times 65C02} = \frac{\sum_{n=1}^{nins(trace_{\exp\times 65C02})} \sum_{r=0}^{nreg(65C02)} f_{abits}(r,t,n)}{51 \times \sum_{r=0}^{nreg(65C02)} f_{RegSize}(r)}$$

We also have the right side of the denominator of the formula solved from section 5.2.1 where we calculated the amount of bits that would be tested on each trace instruction. So, the robustness formula can be written as:

Equation 31 – Exponentiation robustness over 65C02 processor with the two denominator items.

$$robustness_{\exp\times 65C02} = \frac{\sum_{n=1}^{nins(trace_{\exp\times 65C02})} \sum_{r=0}^{nreg(65C02)} f_{abits}(r,t,n)}{51 \times 39}$$

That gives us an amount of bits tested of 1989.

Equation 32 – Exponentiation robustness over 65C02 processor with the denominator calculated.

$$robustness_{\exp\times 65C02} = \frac{\sum_{n=1}^{nins(trace_{\exp\times 65C02})} \sum_{r=0}^{nreg(65C02)} f_{abits}(r,t,n)}{1989}$$

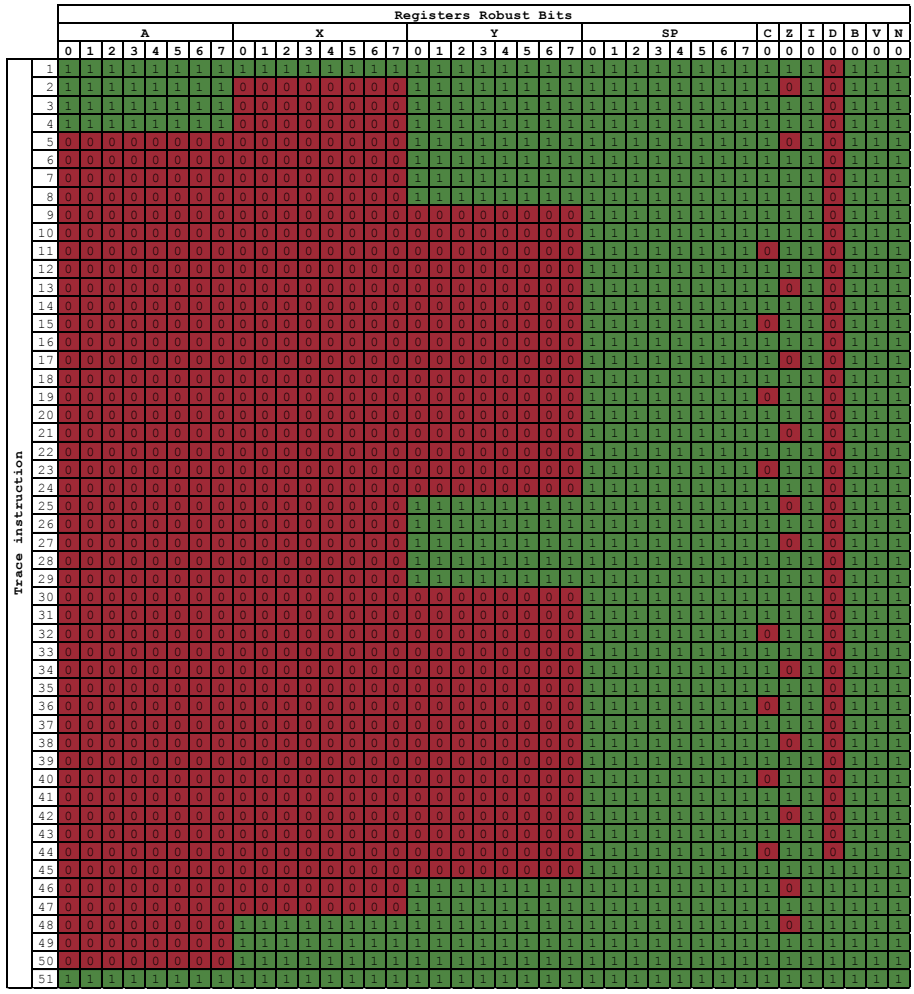

Figure 21 – Registers robust bits over the trace instructions of the original exponentiation program.

Figure 21 shows the 65C02 registers robust bits obtained using the ARTFUL methodology. If compared with Figure 12 that presents the robustness of the same program but evaluated with executions with fault injection, the two figures shows exactly the same robustness points.

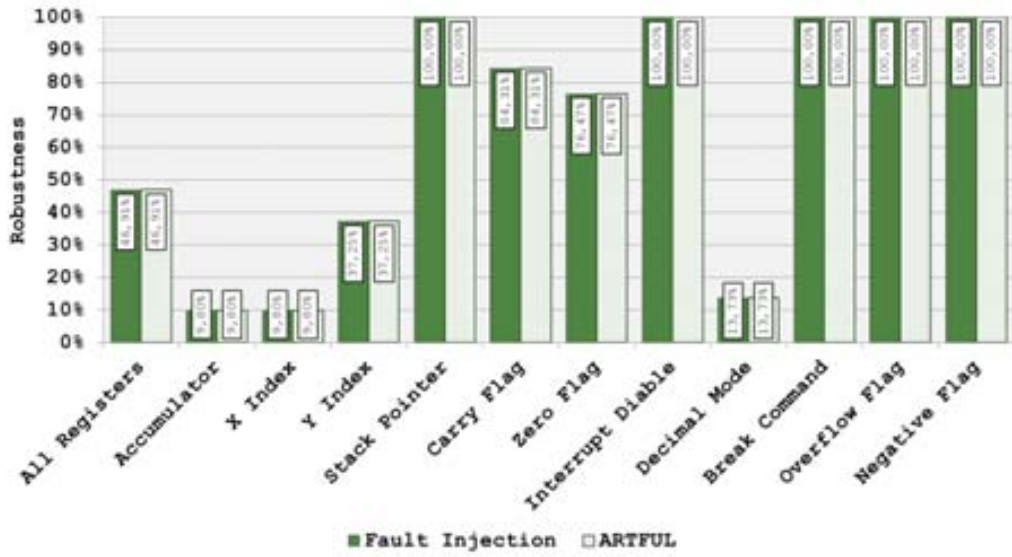

Figure 22 – Comparison between the evaluated exponentiation program robustness using fault injection and the ARTFUL methodology.

For this example program with this given input parameters the evaluated robustness using executions with fault injections and using the ARTFUL methodology was exactly the same. But this is not a common this to happen, as we will present in the next section.

5.2.3 The Improved Exponentiation Program

Without considering that the code are trying to protect the program execution from transient faults, the evaluation of the improved exponentiation program robustness against transient faults using the ARTFUL methodology will present a worse robustness than the original program. The evaluated robustness in this case was 29.80%.

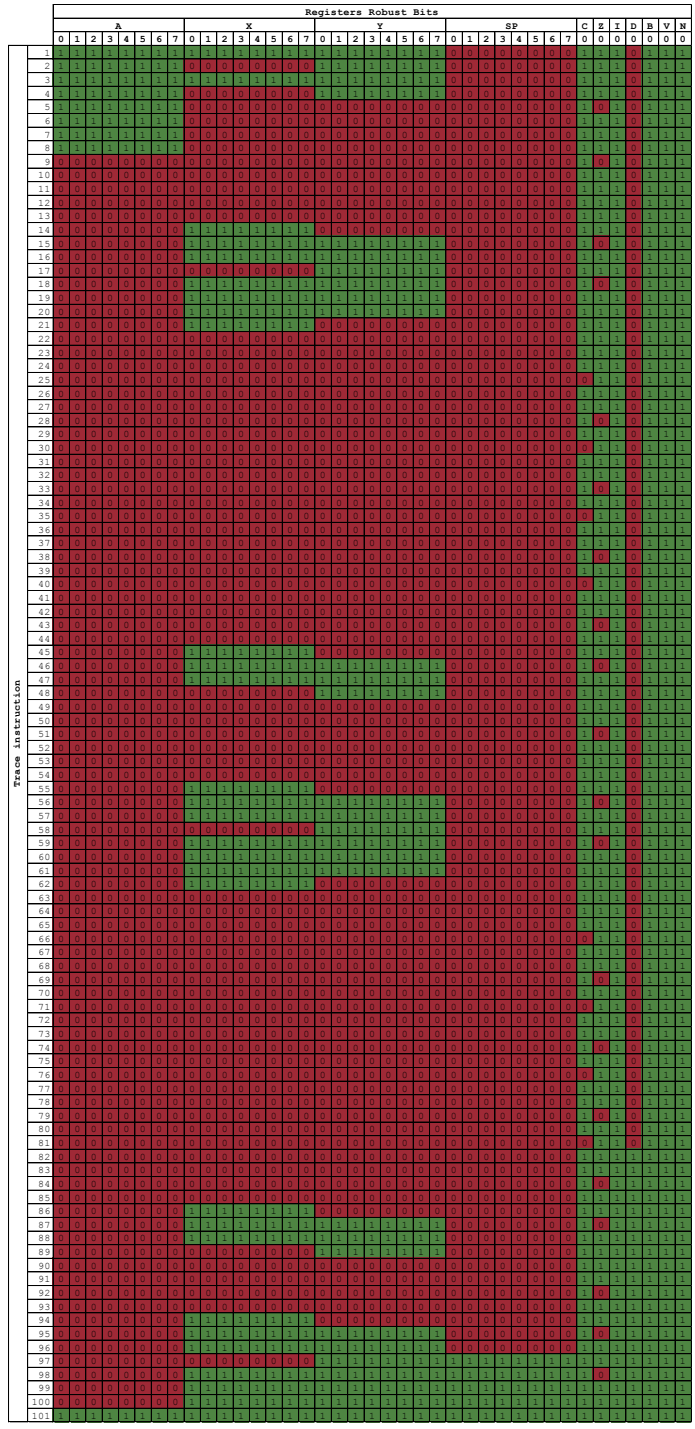

Figure 23 – Registers robust bits over the trace instructions of the improved exponentiation program using partial ARTFUL methodology.

The improved program became sensitive to faults into Stack Pointer and also have more instructions executed.

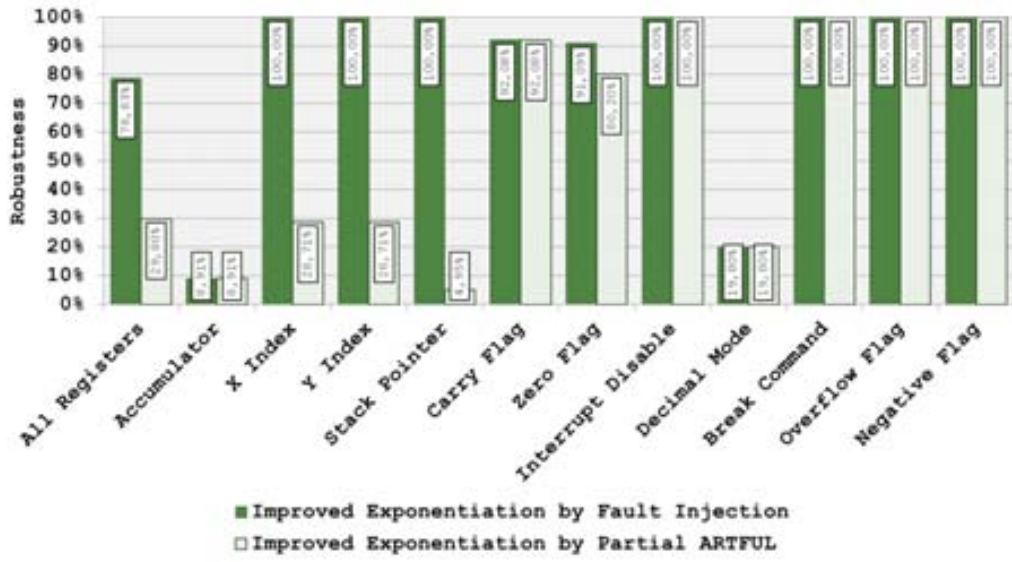

Figure 24 – Comparison between the improved exponentiation robustness evaluated with fault injections and with partial ARTFUL methodology.

However, considering the protection we coded into the improved exponentiation program we can assume that we have protected the X index, the Y index and the Stack Pointer as shown in section 4.2.2, we obtained almost the same results as the fault injection campaign.

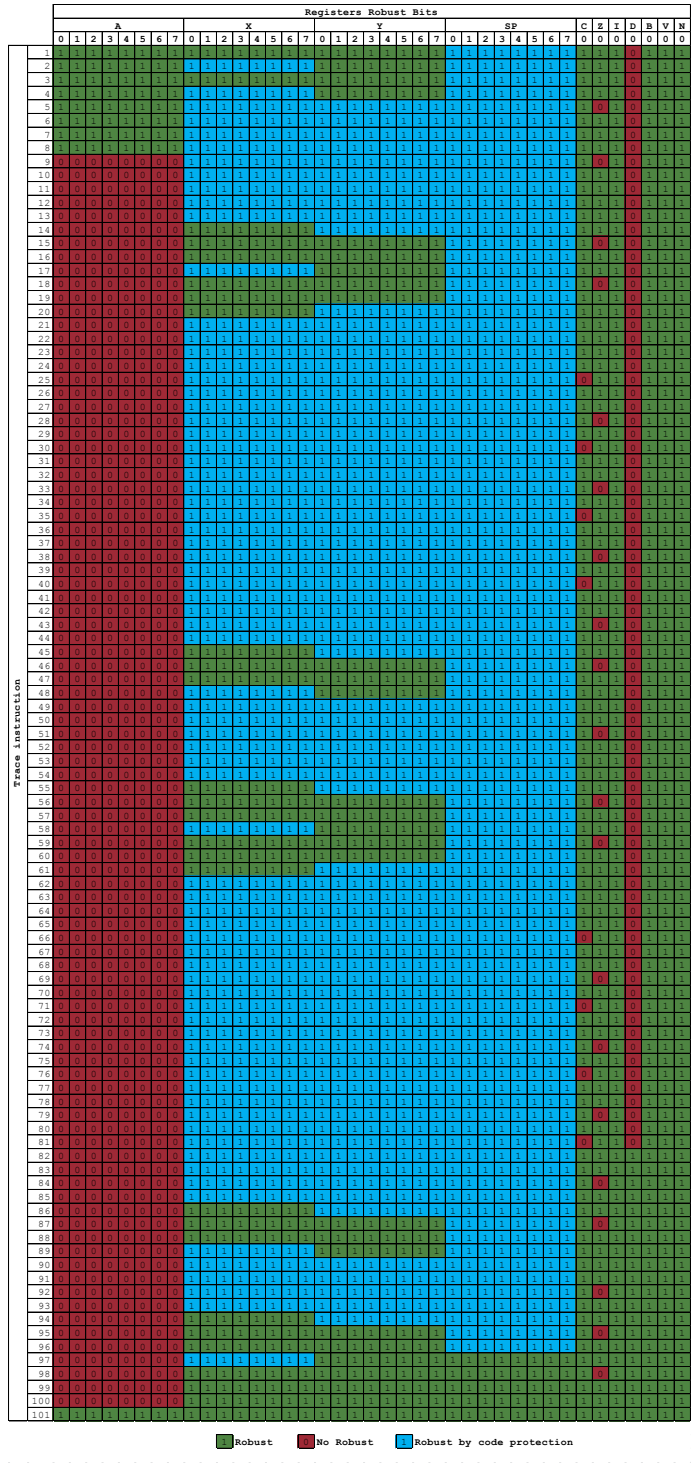

Figure 25 – Registers robust bits over the trace instructions of the improved exponentiation program using the full ARTFUL methodology.

Figure 25 only differ from Figure 14 by not being able to recognize some faults at the Zero Flag that were detected in the fault injection campaign, even assuming that we didn't protected this processor register with our detection mechanism.

Figure 26 – Comparison between the improved exponentiation robustness evaluated with fault injections and with full ARTFUL methodology.

So, in Figure 26, comparing our methodology evaluation with the exhaustive fault injection campaign, the only difference in the chart is the Zero Flag. All other processor registers had the same robustness. The ARTFUL evaluation calculated a robustness of 78,55% against 78,83% of the exhaustive fault injection campaign.

In this aspect we can state that our methodology has a pessimist approach by not being able to recognize some side effects of the fault detection mechanisms as the one that allowed the Zero Flag faults to be detected.

Chapter 6
ARTFUL Tools

One of our primary objectives was to make the process of evaluating a program robustness against transient faults using the ARTFUL methodology more efficient than evaluating with executions with fault injection from the CPU time needed to perform the evaluation.

First of all, we needed to choose a processor architecture to work on. We choose the x64 processor architecture because…

Table 5 presents the x64 processor registers we considered in our evaluations: Sixteen 64 bits general purpose integer registers and sixteen 128 bits Streaming SIMD Extensions (SSE) floating point registers [Advanced Micro Devices, Inc., 2002-2013]. With all this registers, the amount of bits per instruction for this processor architecture is 3072.

The amount of instructions and combination of OpCodes the x64 processor architecture is about 1040, almost five times the amount of OpCodes of the 65C02 processor architecture we used in Chapter 4 and in Chapter 5. Because of this large amount, we will not present here a table with all possible OpCodes and affected registers of the x64 processor architecture.

Table 5 – x64 processor architecture registers.

Register	Size (in bits)	Purpose
RAX	64	General Purpose Integer Register
RBX	64	General Purpose Integer Register
RCX	64	General Purpose Integer Register
RDX	64	General Purpose Integer Register
RSI	64	General Purpose Integer Register
RDI	64	General Purpose Integer Register
RBP	64	General Purpose Integer Register
RSP	64	General Purpose Integer Register
R8	64	General Purpose Integer Register
R9	64	General Purpose Integer Register
R10	64	General Purpose Integer Register
R11	64	General Purpose Integer Register
R12	64	General Purpose Integer Register
R13	64	General Purpose Integer Register
R14	64	General Purpose Integer Register
R15	64	General Purpose Integer Register
XMM0	128	SSE Float Point Register
XMM1	128	SSE Float Point Register
XMM2	128	SSE Float Point Register
XMM3	128	SSE Float Point Register
XMM4	128	SSE Float Point Register
XMM5	128	SSE Float Point Register
XMM6	128	SSE Float Point Register
XMM7	128	SSE Float Point Register
XMM8	128	SSE Float Point Register
XMM9	128	SSE Float Point Register
XMM10	128	SSE Float Point Register
XMM11	128	SSE Float Point Register
XMM12	128	SSE Float Point Register
XMM13	128	SSE Float Point Register
XMM14	128	SSE Float Point Register
XMM15	128	SSE Float Point Register
Total	**3072**	

Our evaluation is divided in two steps: the trace generation and the trace analysis. Up to now there is no way of doing the evaluation in only one step. This is because the second step (trace analysis) depends on all data generated in the first step (trace generation) but will evaluate its contents backwards (from the last generated data to the first generated data).

6.1 ARTFUL Tracer

The trace generation step comprehends all activities of the methodology regarding obtaining program information to perform the analysis.

From the methodology point of view, the trace generation should log all instructions executed by the program in a single trace. This trace should have all instructions executed by the program in the order they were executed.

From the implementation point of view, the trace generation logs all basic blocks executed by the program being evaluated in the order they are executed and also logs all instructions executed in each basic block.

In the implemented trace generation tool we divided the trace information in two distinct files: the basic block information (BBI) file and the packed basic block (PBB) sequences file, as shown in Figure 27.

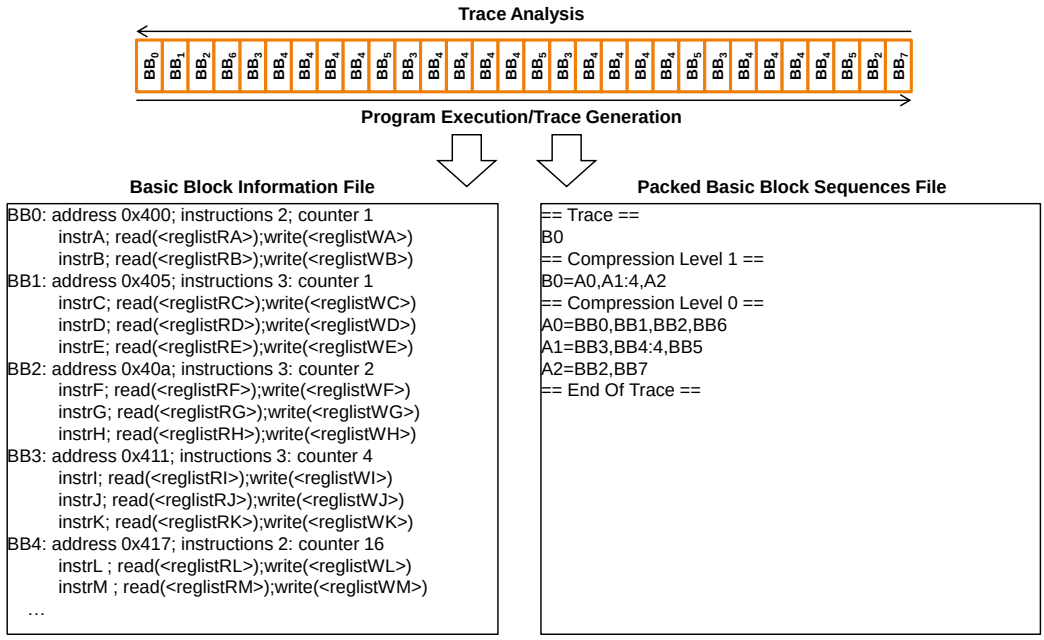

Figure 27 – Program trace generation outputs: the BBI and the PBB files.

The BBI file has detailed information about all basic blocks: the address of the basic block in memory, the amount of instructions the basic block has, how many times during trace generation the basic block was executed and detailed information about its instructions. The basic block information file size is proportional to the amount of unique basic blocks recognized during the trace generation.

The current implemented trace generation tool stores all information about how the instructions affect the registers in the BBI file instead of having this information statically in some kind of processor architecture information table. Fortunately, the dynamic instrumentation tool (PIN [Luk et al., 2005]) we used to develop our trace generation tool has in its API ways to analyze instructions and collect information about read and written registers.

Once having all instructions information in the BBI file, our tool saves all basic blocks executed by the program during the trace generation in the order they have been executed in the PBB file.

The very first implementation of the trace generation tool didn't perform any kind of compression in the PBB file. But, once we started generating traces of program executions we noticed that a very small program could have tens of millions of basic blocks executions, and small programs might have billions of basic blocks executions. These amounts grew as the evaluated programs executed for more time (with larger workloads).

We implemented an on-the-fly simple compression algorithm in our trace generation tool trying to avoid huge files with the basic block sequences executed.

The implemented compression algorithm creates buffers to store sequences of identifiers for each compression level.

At each compression level, as the basic blocks are executed, the identifier of the basic block (for the first compression level) or the identifier of the sequence of basic blocks from the lower compression level are stored in a buffer until an identifier lower than the last stored one arrives. Finding a jump to a lower sequence identifier triggers the delimitation of a single sequence.

The compression algorithm will then seek for the current buffered sequence in the list of known sequences. If it finds a match, it will store only the matched sequence identifier (and not the whole sequence information) in the upper compression level. If it didn't find a match, it will store the buffered sequence as a new sequence in the list of known sequences. Then, the compression algorithm will clean the current buffer and will let the trace generation keep running.

Also, when a sequence happen to occur repeatedly (mostly in loops at the program execution), the compression algorithm will put only one occurrence of the sequence identifier on the PBB file and will also put a counter with the amount of times that the sequences repeated itself.

In Figure 27 we have a sample of a hypothetical packed basic block sequence trace with two levels of compression. There are cases of repetition in both presented compression levels: of a sequence of basic blocks in compression level one and of basic blocks in compression level zero.

6.2 ARTFUL Analyzer

The trace analysis step uses all architecture information (about its instructions and how the instructions affects the architecture registers) and the trace of the program execution to evaluate the program robustness against transient faults.

From the methodology point of view, the trace analysis must evaluate every instruction executed by the program in the inverse order they were executed, calculating

the robust state for all architecture registers bits and storing the amount of bits considered robust to inform the whole program robustness.

In the implemented trace analysis tool we first read the BBI file and create a table with all basic blocks instructions and affected registers (both by reading and writing).

After finishing reading the BBI file, the analysis tool reads the PBB file, storing the sequences information in memory. Once finished reading the PBB file, the analysis begins by replacing recursively the sequences of the higher compression levels by sequences of lower compression levels until arriving at a basic block unit, when it performs the robustness evaluation for the basic block instructions.

Chapter 7
Experimental Evaluation

In order to realize our first experimental evaluation of the ARTFUL tools we designed a set of experiments to calculate the robustness against transient faults of five programs both using fault injection executions and using both ARTFUL Tracer and Analyzer.

The selected programs are part of the NAS Parallel Benchmark [Bailey et al., 1991] in its version 3.3. Because of the amount of executions needed to realize this experimental work we choose to evaluate the serial (non-parallel) versions of BT, CG, FT, LU and SP benchmarks with their smallest class (S) [NASA Website, 2012].

All five benchmark programs used in this experimental work were compiled using GNU C and Fortran in their version 4.4.1, with static linkage of libraries used by the programs and with maximum code optimization during compilation (O3).

The computing nodes used in the experiments have Linux Ubuntu Server operating system in version 9.10 with 64 bits kernel in version 2.6.31. The hardware of all computing nodes used have one 2 GHz AMD Athlon 64 X2 processor with 2 gigabytes of memory.

7.1 Using Executions with Fault Injection

The fault injection environment used in this part of the experimental evaluation uses a tool based on Intel PIN [Luk et al., 2005] to flip a single randomly chosen bit of a randomly chosen processor register in an also randomly chosen point of a program execution.

For each one of the evaluated programs we made 8,000 executions with fault injection. The amount of executions was defined with the objective of achieve at least 2% of confidence interval of the average cases classified as unACE.

In Table 6 we present the execution time of each benchmark program.

Table 6 – Benchmarks basic collected information.

Benchmark	Execution Time (in Seconds)	Binary Code Instructions	Instructions Executed	Amount of States to Evaluate
BT	0,19	25.337	521.847.689	1.603.116.100.608
CG	0,16	11.376	357.952.094	1.099.628.832.768
FT	0,28	11.137	666.494.276	2.047.470.415.872
LU	0,08	28.452	187.912.097	577.265.961.984
SP	0,08	21.138	212.261.609	652.067.662.848

Also, we present in Table 6 the amount of instructions executed by each program and the amount of states to evaluate in order to exhaustively cover all possible bit flips in processor registers (amount of instructions executed multiplied by the sum of the amount of bits of all processor registers took into account during the evaluation, in this particular case equal to 3072).

Figure 28 presents the average robustness calculated with the results of the fault injection executions of the selected programs.

Only one of the evaluated programs, the BT benchmark, didn't achieve 1.5% of standard deviation with 8,000 executions with fault injection.

The CPU time needed to calculate the robustness against transient fault using fault injection executions depends on the fault injection environment used to inject the faults.

The best theoretical amount of time needed can be calculated by multiplying the amount of executions the experiment intends to do (8,000 in our case) by the amount of time needed to execute de program being evaluated once.

Perhaps, the environment we used in our experimentation using fault injections uses dynamic instruction instrumentation during the program execution and adds some overhead to the program execution.

In Table 7 we present the amount of time we spent to realize all executions with fault injection in our fault injection environment and also how many executions we needed to achieve 2% of standard deviation in robustness.

Table 7 – Benchmark programs fault injection data.

Benchmark	Execution Time (in Seconds)	Execution Time Using Dynamic Instrumentation (in Seconds)	Fault Injection Time Using Dynamic Instrumentation (in Seconds)	Robustness (Amount of unACE Cases)	Stantard Deviation after 8,000 Executions	Executions to 2% of Standard Deviation
BT	0,19	21,73	87.661	55,41%	2,23%	6.346
CG	0,16	11,28	45.749	56,05%	1,31%	3.301
FT	0,28	12,12	49.602	62,34%	1,13%	2.456
LU	0,08	21,31	85.548	39,14%	1,31%	2.019
SP	0,08	16,68	67.043	44,94%	1,68%	3.350

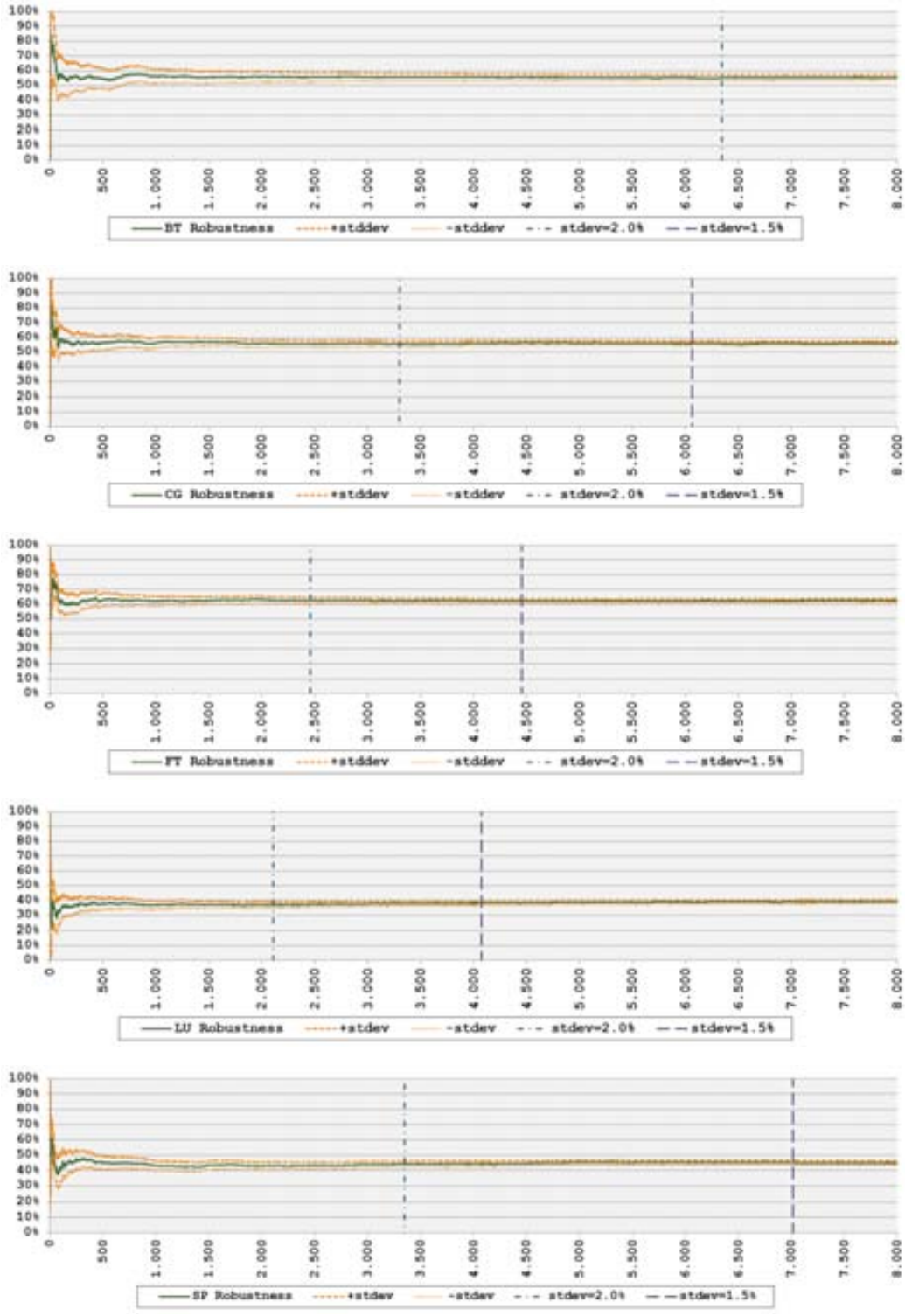

Figure 28 – Fault injection executions and the average evaluated robustness.

The amount of time needed to realize a program set of executions using our fault injection environment was calculated assuming that the program executes, on average, half of its instructions with the dynamic instrumentation overhead and the other half without any overhead. This is because, once the fault is injected, the environment let the program run until the end without any interference.

7.2 Using ARTFUL Tools

In Table 8 we present the time we spent generating the traces and the time to analyze those traces and calculate the robustness with the ARTFUL methodology tools. Also, the table presents the calculated robustness and the total time needed to calculate a program's robustness (time to generate the trace plus time to analyze the trace).

Table 8 – Benchmark programs data using ARTFUL tools.

Benchmark	Trace Generation Time (in seconds)	Robustness Analysis Time (in seconds)	Robustness	Total Robustness Evaluation Time (in seconds)
BT	6,69	413,96	29,88%	420,65
CG	13,27	237,84	57,35%	251,11
FT	6,18	442,65	49,71%	448,83
LU	4,12	139,26	28,93%	143,38
SP	6,01	151,93	39,18%	157,94

The time spent on generating a program trace depends on the program being analyzed algorithm. On the other hand, the time spent on the analysis of the program trace is proportional to the amount of instructions executed by the analyzed program.

As we already predicted, in Figure 29 we present that the calculated robustness using our methodology is always lower than the calculated using fault injection executions or it can be higher (but almost the same) depending on the amount of executions done to calculate de robustness using fault injection and the random number generator and seed used. Our methodology will score a lower robustness because the approach of using fault injection is more data dependent than our proposal and can mask possible DUE and SDC as unACE as explained previously Chapter 4.

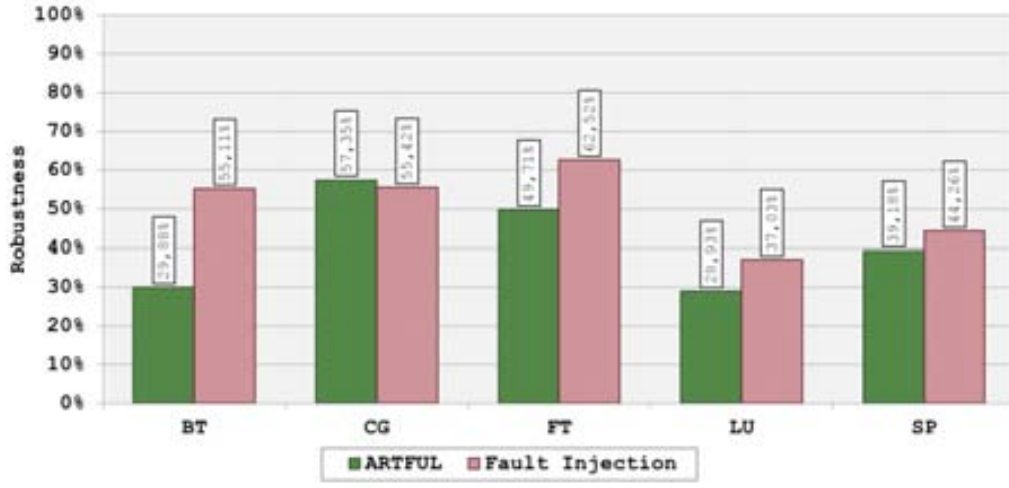

Figure 29 – ARTFUL methodology vs. fault injection robustness's.

On the analysis of the CPU time spent during the robustness calculation using the proposed methodology in Figure 30, we used on average almost 60% of the time needed to run enough experiments using the best theoretical fault injection method and achieve 2% of standard deviation in the statistical approximation. Also, comparing the CPU time spent during the robustness calculation using the proposed methodology with the real fault injection environment used based on dynamic instrumentation to inject the faults, we needed on average only 1.22% of the time needed to achieve 2% of standard deviation in the fault injection statistical approximation.

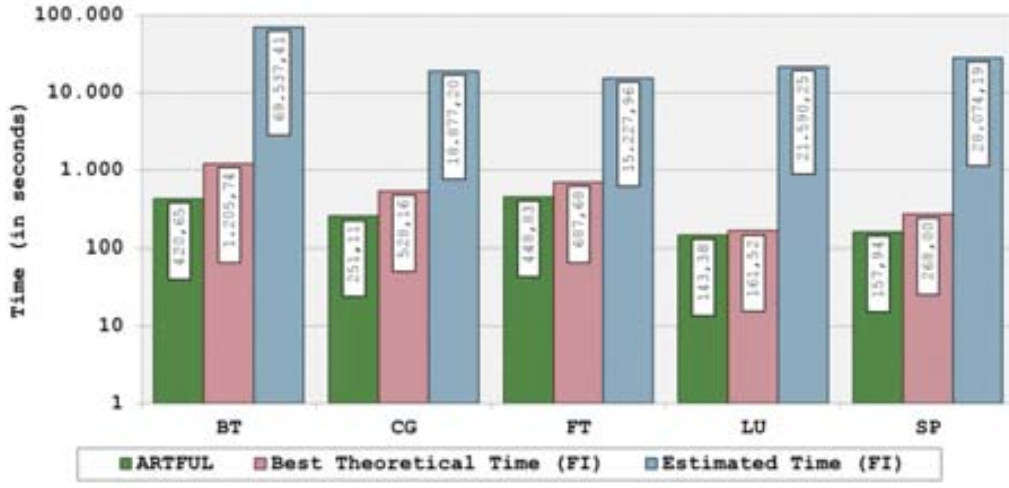

Figure 30 – Time spent on calculating robustness's.

All the times collected in our experimental evaluation took in account the use of only one CPU core to all activities without any kind of parallelism. We know that the executions with fault injection are independent and could exploit many cores in

processor nodes to minimize the total time needed to calculate a program's robustness against transient faults.

Fortunately, the calculations of each processor register's robustness in the proposed methodology are independent. In this way, we can also take benefit of parallelism to speed up our robustness against transient fault analysis (we could parallelize the analysis of this experimental evaluation in 32 independent threads as we evaluated the robustness of 32 of the processor registers of the experimented architecture).

7.3 Improving Efficiency

In order to improve the efficiency of the robustness evaluation by reducing the amount of time needed to perform the whole evaluation (program trace generation plus trace analysis) we worked in two distinct lines: one based on compression and trying to avoid the calculation of repetitive program parts; other based on using a prediction tool for parallel program executions to estimate the parallel program robustness against transient faults.

7.3.1 Simplification

The idea of using some kind of simplification during the analysis step started when we were doing proofs of concept of our methodology with architectures and programs simple enough to allow running the whole methodology (both the trace generation and the analysis step) by hand.

We noticed that, once in a loop, where a sequence of instructions are repeated a given amount of times, there were a coincidence in the robustness evaluated for the set of instruction of the sequence in all the repetitions except the first analyzed.

So we supposed that, if we could prove that this coincidence was in fact a simplification, it could be done without affecting two of our methodology characteristics (exhaustiveness and precision) reducing significantly the amount of time needed to perform the robustness evaluation.

7.3.1.1 Analytical Proof

Even noticing the potential of the coincidence in some proof of concept evaluations and trace analyses, we studied the simplification using our analytical definition of the robustness presented in Chapter 5 in order to prove that we could really simplify some parts of the robustness evaluation without affecting the precision or the exhaustiveness of the evaluation, key characteristics of our methodology.

One of the key concepts of the robustness evaluation using our methodology is (2) the robust state f_{rstate} function of an architecture A register *reg* in a given point n of a program *prog* trace execution $Trace_{prog \times A}$, shown in section 5.1.1.

This function needs three things to be evaluated. The first is the robust state of the register *reg* in the previously evaluated trace point (the next instruction in the execution trace as the analysis is performed backwards).

Also, the f_{rstate} function need to know the bits that the instruction i at the point n in the execution trace $Trace_{prog \times A}$ reads (read operation, f_{rbits}) and writes (write operation, f_{wbits}) to perform its operation.

7.3.1.1.1 Single Instruction Repetition Sequence

Consider a program execution trace over a given architecture with n executed instructions. Consider also that this trace has a sequence with the instruction i_1 that is repeated k times, with its last repetition at the trace point x. Figure 31 shows a sample of the considered trace.

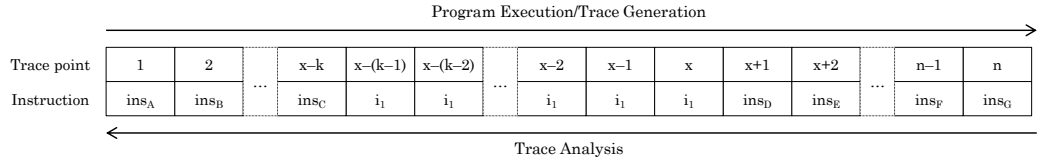

Figure 31 – Program trace with a single instruction repetition sequence.

For the instruction evaluated just before the beginning of the sequence with the repetition (trace point x+1), we have:

Equation 33 – Robust state before evaluating the instruction repetition.

$$s_{x+1} = f_{rstate}(reg, x+1)$$
$$s_{x+1} = \left(f_{rstate}(reg, x+2) \vee f_{wbits}(ins_D, reg)\right) \wedge\sim f_{rbits}(ins_D, reg)$$

So, for the first instruction i_1 evaluated (the last executed of the sequence) we can calculate its robust state based on the robust state s_{x+1}, as follows:

Equation 34 – Robust state of the first evaluated instruction in the repetition.

$$s_x = f_{rstate}(reg, x)$$
$$s_x = \left(f_{rstate}(reg, x+1) \vee f_{wbits}(i_1, reg)\right) \wedge\sim f_{rbits}(i_1, reg)$$
$$f_W = f_{wbits}(i_1, reg); f_R = f_{rbits}(i_1, reg)$$
$$s_x = (s_{x+1} \vee f_W) \wedge\sim f_R$$

Then, to the second instruction i_1 evaluated we can calculate its robust state based on the robust state s_x of the first instruction i_1 of the sequence, as follows:

Equation 35 – Robust state of the second evaluated instruction in the repetition.

$$
\begin{aligned}
s_{x-1} &= f_{rstate}(reg, x-1) \\
s_{x-1} &= \left(f_{rstate}(reg, x) \vee f_{wbits}(i_1, reg)\right) \wedge\sim f_{rbits}(i_1, reg) \\
&= (s_x \vee f_W) \wedge\sim f_R \\
&= \left(\left((s_{x+1} \vee f_W) \wedge\sim f_R\right) \vee f_W\right) \wedge\sim f_R \\
&= (s_{x+1} \vee f_W) \wedge\sim f_R && \text{by absorption} \\
s_{x-1} &= s_x
\end{aligned}
$$

The robust state s_{x-1} for the second occurrence of the instruction i_1 of the sequence, once calculated, resulted to be the same as the first evaluated one (s_x). This happened because both evaluations were done with the same instruction i_1, and so used the same f_{wbits} and f_{rbits}, what lead to an absorption of the redundant items in the equation.

Indeed, for the third instruction i_1 evaluated, as for all the rest of them until the trace point x-(k-1), all robust states will be the same as the first occurrence of i_1.

Equation 36 – Robust state of the third evaluated instruction in the repetition.

$$
\begin{aligned}
s_{x-2} &= f_{rstate}(reg, x-2) \\
s_{x-2} &= \left(f_{rstate}(reg, x-1) \vee f_{wbits}(i_1, reg)\right) \wedge\sim f_{rbits}(i_1, reg) \\
&= (s_{x-1} \vee f_W) \wedge\sim f_R \\
&= \left(\left((s_x \vee f_W) \wedge\sim f_R\right) \vee f_W\right) \wedge\sim f_R \\
&= (s_x \vee f_W) \wedge\sim f_R && \text{by absorption} \\
s_{x-2} &= s_x
\end{aligned}
$$

7.3.1.1.2 Many Distinct Instructions Repetition Sequence

Consider now another program execution trace over a given architecture with n executed instructions.

Consider also that this trace has a sequence with two distinct instructions i_1 and i_2 that are repeated k times, with the last repetition of the instruction i_1 at the trace point x. Figure 32 shows a sample of the considered trace.

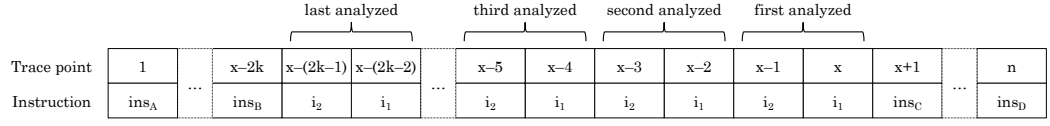

Figure 32 – Program trace with a two distinct instructions repetition sequence.

Beginning with the same assumptions of the previously evaluated scenario, for the first occurrences of i_1 and i_2 we have two write them as functions of the s_{x+1} robust state, as follows:

Equation 37 -

$$s_x = f_{rstate}(reg, x)$$
$$s_x = \left(f_{rstate}(reg, x+1) \vee f_{wbits}(i_1, reg)\right) \wedge\sim f_{rbits}(i_1, reg)$$
$$f_{W1} = f_{wbits}(i_1, reg); f_{R1} = f_{rbits}(i_1, reg)$$
$$s_x = (s_{x+1} \vee f_{W1}) \wedge\sim f_{R1}$$

Equation 38 -

$$s_{x-1} = f_{rstate}(reg, x-1)$$
$$s_{x-1} = \left(f_{rstate}(reg, x) \vee f_{wbits}(i_2, reg)\right) \wedge\sim f_{rbits}(i_2, reg)$$
$$f_{W2} = f_{wbits}(i_2, reg); f_{R2} = f_{rbits}(i_2, reg)$$
$$s_{x-1} = (s_x \vee f_{W2}) \wedge\sim f_{R2}$$
$$s_{x-1} = \Big(\big((s_{x+1} \vee f_{W1}) \wedge\sim f_{R1}\big) \vee f_{W2}\Big) \wedge\sim f_{R2}$$

For the second occurrences of i_1 and i_2 we can resolve their robust state as function of s_{x+1} as follows:

Equation 39 -

$$s_{x-2} = f_{rstate}(reg, x-2)$$
$$s_{x-2} = \left(f_{rstate}(reg, x-1) \vee f_{wbits}(i_1, reg)\right) \wedge\sim f_{rbits}(i_1, reg)$$
$$= (s_{x-1} \vee f_{W1}) \wedge\sim f_{R1}$$
$$= \Big(\Big(\Big(\big((s_{x+1} \vee f_{W1}) \wedge\sim f_{R1}\big) \vee f_{W2}\Big) \wedge\sim f_{R2}\Big) \vee f_{W1}\Big) \wedge\sim f_{R1}$$
$$= \Big(\big((s_{x+1} \vee f_{W2}) \wedge\sim f_{R2}\big) \vee f_{W1}\Big) \wedge\sim f_{R1} \qquad \text{by absorption}$$

Equation 40 -

$$s_{x-3} = f_{rstate}(reg, x-3)$$
$$s_{x-3} = \left(f_{rstate}(reg, x-2) \vee f_{wbits}(i_2, reg)\right) \wedge\sim f_{rbits}(i_2, reg)$$
$$= (s_{x-2} \vee f_{W2}) \wedge\sim f_{R2}$$
$$= \Big(\Big(\Big(\big((s_{x+1} \vee f_{W2}) \wedge\sim f_{R2}\big) \vee f_{W1}\Big) \wedge\sim f_{R1}\Big) \vee f_{W2}\Big) \wedge\sim f_{R2}$$
$$= \Big(\big((s_{x+1} \vee f_{W1}) \wedge\sim f_{R1}\big) \vee f_{W2}\Big) \wedge\sim f_{R2} \qquad \text{by absorption}$$
$$s_{x-3} = s_{x-1}$$

In the second occurrence of the first instruction evaluated i_1 (Equation 39) it wasn't possible to simplify its robust state yet because there was no way to reduce it more until finding something we've already calculated.

However, we could write the second instruction evaluated i_2 (Equation 40) as a function of the last instruction evaluated before the sequence, with the same formula as its first occurrence.

This behavior is slightly different of the repetition of only one instruction in the previously presented example, and will not allow us to simplify the robust states of the first occurrence of the repetition in the rest of the occurrences.

Starting in the third occurrence of i_1 (X) and i_2 (X), all the instructions present in the repetition can be simplified, as follows:

$$s_{x-4} = f_{rstate}(reg, x-4)$$
$$s_{x-4} = \left(f_{rstate}(reg, x-3) \vee f_{wbits}(i_1, reg)\right) \wedge\sim f_{rbits}(i_1, reg)$$
$$= (s_{x-3} \vee f_{W1}) \wedge\sim f_{R1}$$
$$= \Big(\Big(\Big(\big((s_{x+1} \vee f_{W1}) \wedge\sim f_{R1}\big) \vee f_{W2}\Big) \wedge\sim f_{R2}\Big) \vee f_{W1}\Big) \wedge\sim f_{R1}$$
$$= \Big(\big((\mathrm{s_{x+1}} \vee \mathrm{f_{W2}}) \wedge\sim \mathrm{f_{R2}}\big) \vee \mathrm{f_{W1}}\Big) \wedge\sim \mathrm{f_{R1}} \qquad \text{by absorption}$$
$$s_{x-4} = s_{x-2}$$

$$s_{x-5} = f_{rstate}(reg, x-5)$$
$$s_{x-5} = \left(f_{rstate}(reg, x-4) \vee f_{wbits}(i_2, reg)\right) \wedge\sim f_{rbits}(i_2, reg)$$
$$= (s_{x-4} \vee f_{W2}) \wedge\sim f_{R2}$$
$$= (s_{x-2} \vee f_{W2}) \wedge\sim f_{R2}$$
$$= \Big(\Big(\Big(\big((s_{x+1} \vee f_{W2}) \wedge\sim f_{R2}\big) \vee f_{W1}\Big) \wedge\sim f_{R1}\Big) \vee f_{W2}\Big) \wedge\sim f_{R2}$$
$$= \Big(\big((\mathrm{s_{x+1}} \vee \mathrm{f_{W1}}) \wedge\sim \mathrm{f_{R1}}\big) \vee \mathrm{f_{W2}}\Big) \wedge\sim \mathrm{f_{R2}} \qquad \text{by absorption}$$
$$s_{x-5} = s_{x-3} = s_{x-1}$$

With all these analyses we proved that, for a given sequence of instructions that repeats itself a given amount of times, we can use the result obtained in the calculation of the first two sequences analyzed to simplify the robust state (and so, the robustness) of all the rest of the occurrences of the sequence.

7.3.1.2 Key Factors in the Implementation

In order to exploit the potential of the simplification we needed to observe and recognize patterns where sequences of instructions repeated themselves in a program trace.

Fortunately, our trace compression algorithm performed such recognition during the trace generation step with the basic blocks. Also, our trace compression algorithm stored the repetition information in the PBB file.

In this way, our trace generation tool needed no further modification in order to collect more information to help possible simplifications during the analysis step. All the

information about repetitions of sequences of basic blocks (and their instructions) was already in the trace because of our goal of saving disk space.

The only modification we did in our robustness analysis tool was, once recognizing repetition information on the evaluated PBB trace, if the repetition was performed for more than two times, to cache all the information about the second iteration of the repetition and to multiply this cached information by the amount of times left to perform the evaluation of the whole set of iterations. Figure 33 shows an example of a trace analysis using simplification.

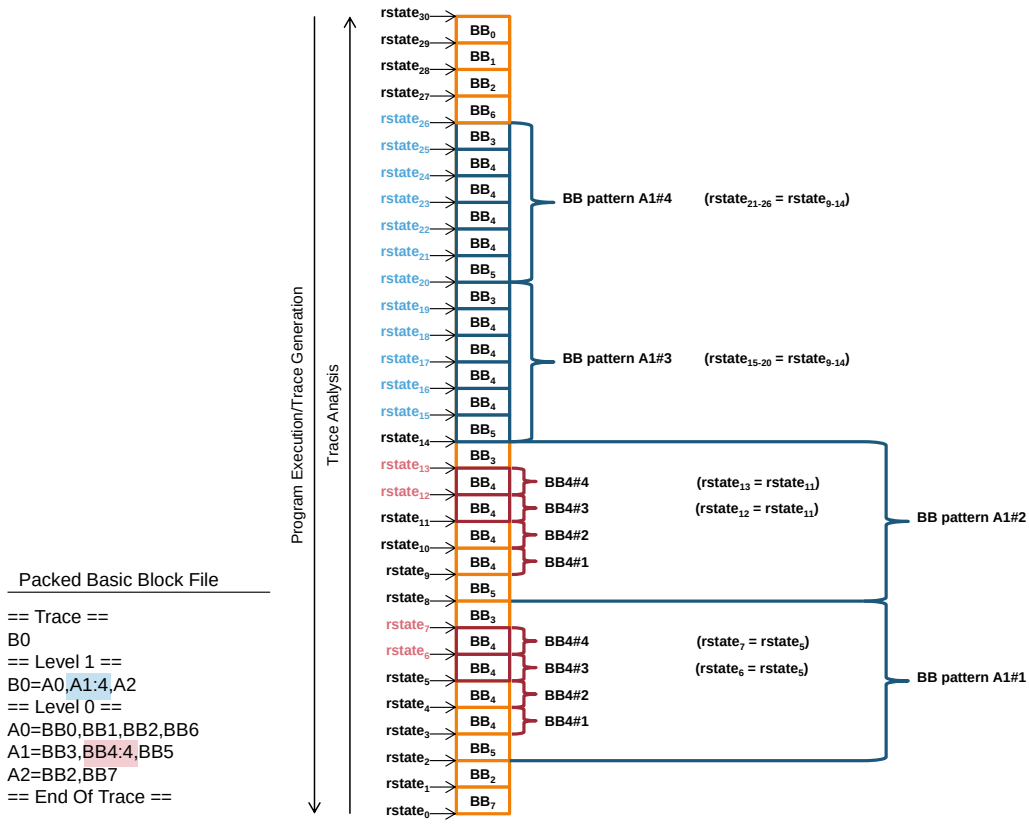

Figure 33 – Example of a PBB trace analysis with simplification.

In the example, we have a hypothetical trace with two potential simplification points. One in the basic block sequence B0 at compression level one with four repetitions of the basic block sequence A1 and other in the basic block sequence A0 with four repetitions of the basic block BB4.

As the trace analysis starts at the basic block sequence B0, the analysis program will calculate the robust state for all architecture registers at basic blocks BB7 and BB2 (basic block sequence A2).

Then, the analysis program will find the repetition pattern of the basic block sequence A1 and it will perform the first iteration of this repetition. It will calculate the robust state for the basic block BB5 and will find another repetition sequence (now for the basic block BB4). As the program performs the analysis recursively, there is no problem of having a sequence being simplified inside other simplification.

The trace analysis will perform the analysis of the first occurrence of BB4, will perform and cache the analysis of the second occurrence of BB4 and will simplify the cached result by two times.

After this, the analysis program will calculate the robust state for basic block BB3 and will finish the first occurrence of the basic block sequence A1.

For the second occurrence of the basic block sequence A1, the trace program will cache the evaluation of BB5, of BB4 (with its own simplification) and BB3. Then, the analysis program will simplify the cached result by two times and will finish the evaluation by calculating the robust state of the basic block sequence A0 (basic blocks BB6, BB2, BB1 and BB0).

This implementation (as the simplification concept presented for our methodology supposes) will generate for a given program trace the exactly same result for the robustness evaluation as the program analysis without simplification, but it may reduce significantly the time needed to perform the robustness analysis against transient fault of a given program trace.

7.3.1.3 NAS Parallel Benchmarks Evaluation

For this section we calculated the robustness against transient faults of nine programs, with five distinct workloads each, using ARTFUL methodology tools with and without simplification.

The selected programs are part of the NAS Parallel Benchmark (NPB) [Bailey et al., 1991] in its version 3.3.1. We selected to evaluate the serial versions of Block Tri-diagonal solver (BT), Conjugate Gradient (CG), Embarrassingly Parallel (EP), discrete 3D fast Fourier Transform (FT), Integer Sort (IS), Lower-Upper Gauss-Seidel solver (LU), Multi-Grid on a sequence of meshes (MG), Scalar Penta-diagonal solver (SP) and Unstructured Adaptive mesh (UA) benchmarks with their S, W, A, B and C classes [NASA Website, 2012].

In comparison with the data presented in section Chapter 7, where we evaluated only the S class of five benchmarks (BT, CG, FT, LU and SP), we are scaled our robustness analysis from programs that execute in tents of seconds (0.16 on average) to programs that execute in hundreds of seconds (683.47 on average).

All nine benchmark programs used in this section were compiled using GNU C and Fortran in their version 4.4.6, with maximum code optimization during compilation (O3).

All the computing nodes used in this set of experiments have CentOS version 6 operating system with 64 bits kernel version 2.6.32. The hardware of all computing nodes used in this work has eight 1.6 GHz AMD Opteron processors with eight cores each and 256 gigabytes of RAM.

Table 9 and Table 10 show a summary of the numbers we obtained with this set of experiments. The tables contain: the program used in the evaluation (Benchmark), the workload used in the evaluation (Class), the standard program execution time without any kind of interference (Execution Time), how much time took to generate the traces (Trace Generation Time), the size of the basic block information trace file (BBI File Size), the amount of unique basic blocks recognized by the trace generator (Unique Basic Blocks), the size of the packed basic block sequences file (PBB File Size), the total amount of instructions that were traced during the trace generation step (Instructions Executed), the percentage of the instructions that were executed by the program and not by some library (Program Influence), the robustness calculated (Robustness), the time spent on the robustness analysis without simplification (Analysis Time Without Simplification), the time spent on the robustness analysis with simplification (Analysis Time With Simplification) and the amount of instructions that the analyses could simplify (Simplified Instructions).

Table 9 – Summary of experimental evaluation results for NPB (1/2).

Benchmark	Workload (Class)	Execution Time (in seconds)	Trace Generation Time (in seconds)	BBI File Size (in Mbytes)	Unique Basic Blocks	PBB File Size (in Mbytes)	Instructions Executed by the Benchmark
BT	S	0,12	4,30	3,24	4.166	0,10	523.995.891
BT	W	4,21	55,73	3,25	4.170	0,10	17.217.054.499
BT	A	97,61	1.165,33	3,26	4.194	0,10	374.989.775.193
BT	B	413,01	4.723,00	3,28	4.175	0,10	1.615.068.999.753
BT	C	1.695,96	19.024,00	3,29	4.171	0,10	6.612.381.481.454
CG	S	0,10	9,40	1,75	3.710	1,81	379.751.162
CG	W	0,76	54,15	1,75	3.706	7,70	2.510.616.217
CG	A	3,18	266,07	1,74	3.698	27,05	10.101.436.998
CG	B	108,13	3.577,29	1,75	3.710	170,52	279.455.264.831
CG	C	253,75	10.373,00	1,75	3.711	435,66	753.648.354.488
EP	S	1,82	105,30	1,73	3.671	58,43	3.543.251.551
EP	W	3,66	207,78	1,73	3.667	113,76	7.086.152.780
EP	A	29,29	1.730,08	1,73	3.669	834,10	56.683.093.108
EP	B	117,09	7.099,00	1,86	3.917	3.179,85	226.733.672.790
EP	C	468,36	29.367,00	1,87	3.924	12.181,01	906.914.579.684
FT	S	0,20	4,20	1,72	3.639	0,08	700.790.731
FT	W	0,38	7,36	1,72	3.651	0,08	1.527.659.555
FT	A	6,25	97,06	1,72	3.642	0,08	27.508.731.558
FT	B	72,86	1.241,39	1,71	3.653	0,09	317.781.421.201
FT	C	355,54	7.267,00	1,82	3.861	0,12	1.419.089.585.765
IS	S	0,02	1,00	1,09	2.234	0,04	28.187.780
IS	W	0,33	4,03	1,09	2.233	0,04	450.210.674
IS	A	2,75	26,50	1,07	2.206	0,04	3.600.001.741
IS	B	11,54	104,93	1,07	2.210	0,04	14.399.286.913
IS	C	48,97	429,56	1,06	2.211	0,04	57.596.415.431
LU	S	0,10	2,70	3,65	4.313	0,10	183.783.055
LU	W	8,69	111,40	3,52	4.306	0,10	29.731.132.471
LU	A	65,91	727,47	3,57	4.337	0,10	195.286.219.236
LU	B	297,98	3.100,34	3,55	4.336	0,10	815.837.775.233
LU	C	1.326,01	12.511,00	3,55	4.320	0,10	3.339.256.715.227
MG	S	0,01	1,70	1,94	3.784	0,10	32.032.457
MG	W	0,45	9,22	1,94	3.793	0,11	1.791.271.812
MG	A	4,18	54,62	1,94	3.802	0,12	14.044.248.666
MG	B	14,51	128,35	1,94	3.793	0,12	49.910.756.229
MG	C	134,86	900,32	1,93	3.804	0,13	395.275.263.604
SP	S	0,10	3,73	2,92	4.264	0,10	219.096.456
SP	W	8,72	260,16	2,95	4.271	0,10	31.143.833.781
SP	A	59,01	1.539,73	2,94	4.281	0,10	187.254.308.059
SP	B	272,55	6.101,00	2,94	4.280	0,10	771.573.262.708
SP	C	1.138,18	25.128,00	2,95	4.286	0,10	3.157.545.842.543
UA	S	0,70	18,32	5,17	5.845	0,82	2.446.467.394
UA	W	4,25	97,45	5,26	5.960	7,96	14.377.904.873
UA	A	39,72	887,08	5,28	5.967	117,63	111.067.994.805
UA	B	175,39	4.603,00	5,28	5.968	559,13	470.340.024.288
UA	C	729,63	20.800,00	5,43	6.250	2.041,21	1.932.283.092.641

Table 10 – Summary of experimental evaluation results for NPB (2/2).

Benchmark	Workload (Class)	Program Influence	Robustness	Analysis Time Without Deduction (in seconds)		Analysis Time With Deduction (in seconds)	Amount of Deduced Instructions
BT	S	99,9171%	40,32%	315,00		5,50	98,38432%
BT	W	99,9972%	38,83%	10.313,00		9,16	99,91532%
BT	A	99,9998%	38,00%	225.035,00		9,10	99,99612%
BT	B	99,9999%	36,84%	969.158,48	*	9,65	99,99905%
BT	C	99,9999%	36,50%	3.967.908,25	*	9,63	99,99977%
CG	S	99,8865%	63,29%	228,00		10,40	95,66392%
CG	W	99,9815%	63,08%	1.505,81		59,13	96,20491%
CG	A	99,9954%	62,74%	6.065,00		188,59	96,99172%
CG	B	99,9995%	62,78%	167.696,90	*	2.011,00	98,83503%
CG	C	99,9998%	62,70%	452.253,03	*	4.415,08	99,05249%
EP	S	30,1605%	74,51%	2.187,50		2.001,60	8,73643%
EP	W	30,1571%	74,51%	4.873,00		4.017,00	8,73096%
EP	A	30,1554%	74,51%	35.039,00		32.065,00	8,72579%
EP	B	30,1554%	73,71%	147.519,98	*	128.140,00	8,72547%
EP	C	30,1561%	73,74%	590.066,83	*	513.007,00	8,72569%
FT	S	95,6037%	57,30%	413,40		1,22	99,74990%
FT	W	95,9872%	57,93%	901,55		1,74	99,83285%
FT	A	96,4578%	56,24%	16.158,00		2,14	99,98881%
FT	B	99,9970%	55,11%	187.499,96	*	4,72	99,99763%
FT	C	99,9973%	55,74%	837.302,68	*	15,17	99,99828%
IS	S	99,7169%	66,52%	16,82		0,20	99,51034%
IS	W	99,9807%	67,24%	266,70		0,20	99,96788%
IS	A	99,9976%	67,24%	2.409,41		0,20	99,99599%
IS	B	99,9993%	67,79%	8.561,00	*	0,19	99,99899%
IS	C	99,9997%	67,79%	34.243,55	*	0,20	99,99975%
LU	S	99,7529%	38,04%	111,10		4,30	96,49928%
LU	W	99,9983%	34,86%	17.829,00		8,75	99,95334%
LU	A	99,9997%	34,05%	116.309,00		9,09	99,99261%
LU	B	99,9999%	33,75%	491.204,53	*	9,05	99,99824%
LU	C	99,9999%	33,86%	2.010.519,82	*	9,15	99,99957%
MG	S	99,2524%	64,07%	19,22		0,73	97,19477%
MG	W	99,9866%	61,43%	1.062,64		0,98	99,93263%
MG	A	99,9982%	61,01%	8.243,00		1,04	99,99026%
MG	B	99,9994%	60,15%	29.777,01	*	2,36	99,99286%
MG	C	99,9999%	59,93%	235.823,19	*	2,64	99,99898%
SP	S	99,7982%	49,35%	132,50		2,50	98,37803%
SP	W	99,9983%	46,79%	18.584,00		3,19	99,98478%
SP	A	99,9997%	46,45%	111.944,00		3,21	99,99744%
SP	B	99,9999%	45,94%	463.490,96	*	3,21	99,99938%
SP	C	99,9999%	45,91%	1.896.766,03	*	3,23	99,99985%
UA	S	99,8481%	52,81%	1.473,80		183,70	87,65023%
UA	W	98,7710%	52,58%	8.614,00		1.103,33	87,44490%
UA	A	94,8764%	53,41%	66.454,00		11.536,00	82,99920%
UA	B	90,3040%	54,60%	282.562,47	*	67.057,00	76,75120%
UA	C	87,0591%	55,66%	1.160.842,50	*	329.929,00	72,18357%

All the evaluated programs in this set of experiments were very predictable, with exception for the EP and UA benchmarks. This "unpredictability" is reflected directly in the efficiency of the compression we obtained during the PBB file generation by recognizing repetitive sequence patterns of basic blocks.

Figure 34 to Figure 37 show the relation between workloads and the amount of instructions executed and simplified.

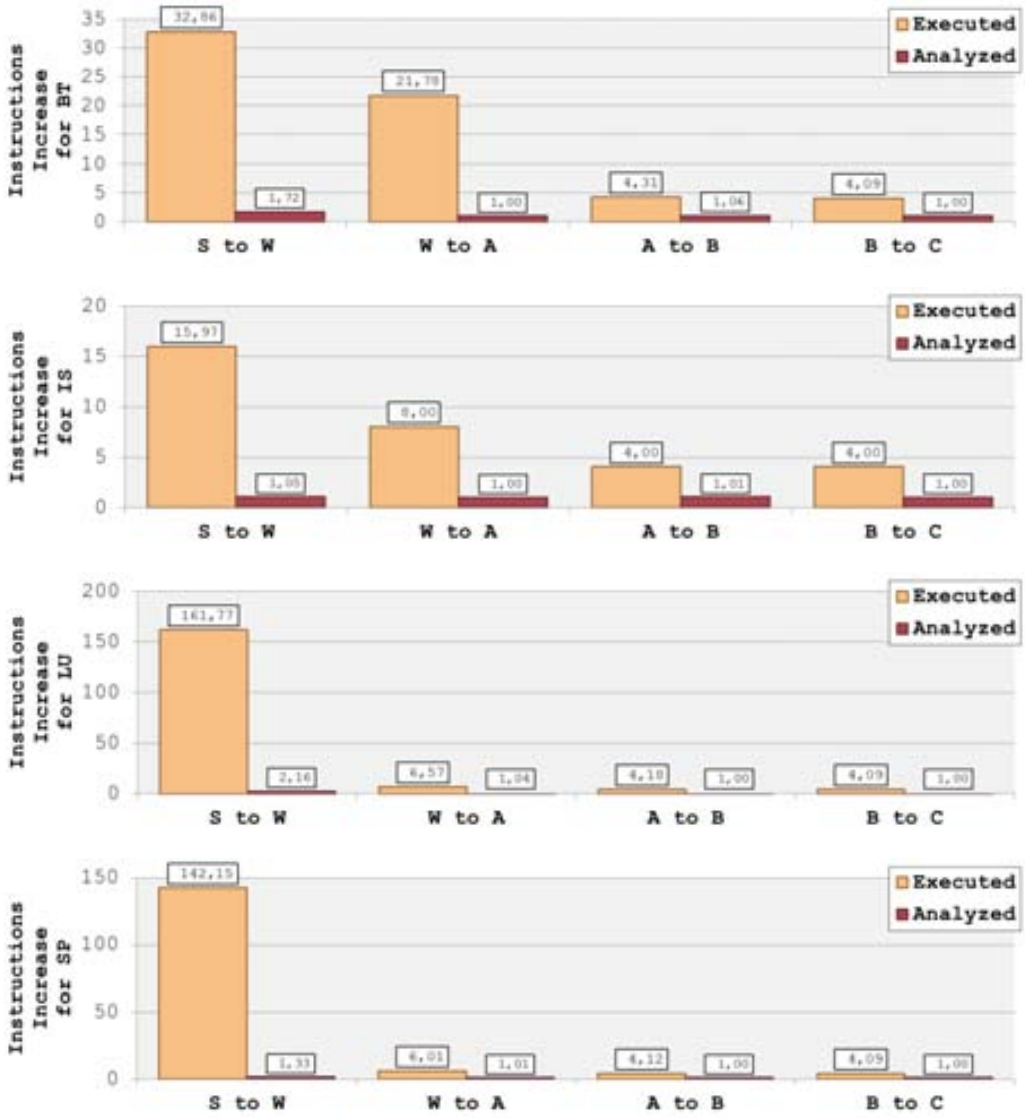

Figure 34 – Instructions executed and analyzed for NPB – the good cases.

The optimal behavior of a program for our simplification purposes are presented in BT, IS, LU and SP charts in Figure 34. These programs, once scaling out the workload, have a significant increase in the instructions executed, but the increase in the

amount of instructions analyzed keep always around one. This means that the analysis time for these programs with simplification is almost constant, even with larger workloads, as showed in Table 9 and Table 10.

A non-optimal but very good behavior of programs for our simplification purposes were obtained with CG, FT and MG programs shown in Figure 35. They kept increasing the amount of executed instructions more than increasing the amount of analyzed instructions. This means that the benefit of simplification is better for larger workloads.

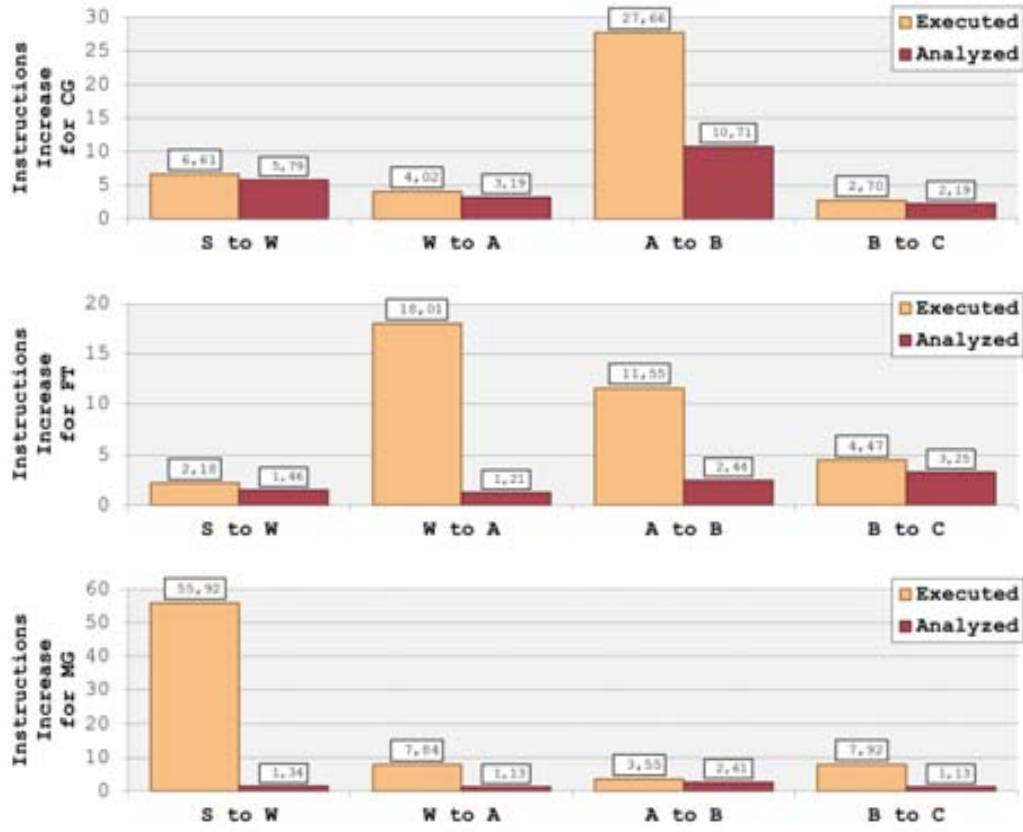

Figure 35 – Instructions executed and analyzed for NPB – the not bad cases.

The EP program shown in Figure 36 presented an equal increase in the amount of instructions executed and in the amount of instructions analyzed. In this case, the benefit of simplification has a constant proportion with the amount of instructions executed (around 8.7%) as shown in Table 9 and Table 10.

The worst scenario for our simplification is what we obtained with the UA program evaluation, shown in Figure 37. The increase of the amount of instructions analyzed was greater than the increase of the amount of instructions executed. This means that we are losing simplification capacity for larger workloads.

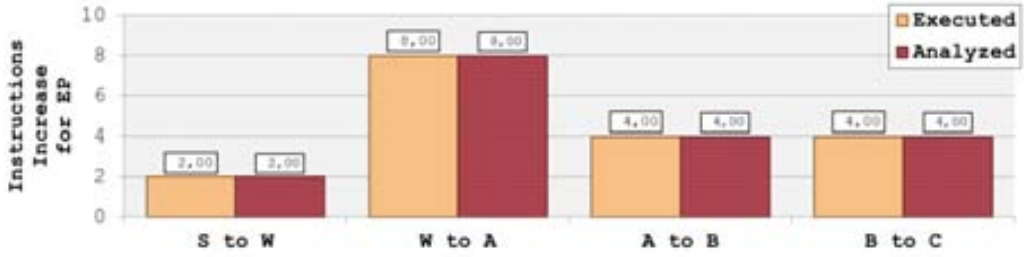

Figure 36 –

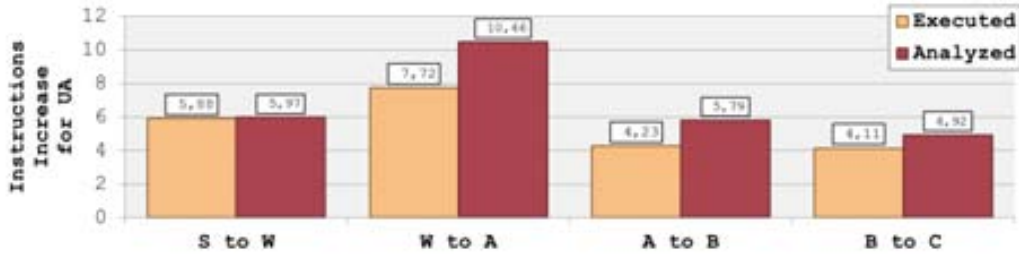

Figure 37 – Instructions executed and analyzed for NPB – the bad cases.

Because of a limitation in our experimental environment allowing a program to run only up to three days, unfortunately, the accurate numbers about Class C EP and UA programs analyses with simplification are unavailable.

However, even in the worst cases presented in this set of experiments, the analysis time using simplification will always be equal or better the analysis time without simplification.

Also, in the worst result we obtained, the time needed to generate the program trace and analyze it without any simplification was equivalent to run the evaluated program for less than 2,800 times (1,880 times on average without simplification and 170 times on average with simplification).

Our current analysis tool scored about 1.6 millions of instructions analyzed per second. This number can be used for predicting the analysis time without simplification once obtained the program trace. In fact, for all classes B and C presented in Table 9 and Table 10, the numbers of the analysis time without simplification (with an *) were estimated.

Figure 38 shows the robustness evaluated of each program of this set of experiments with almost all the workloads proposed.

It is noticeable that the smaller workloads tend to present a robustness slightly different of those with the larger workloads. However, the standard deviation found between the smaller and the larger workloads were all lower than 3%.

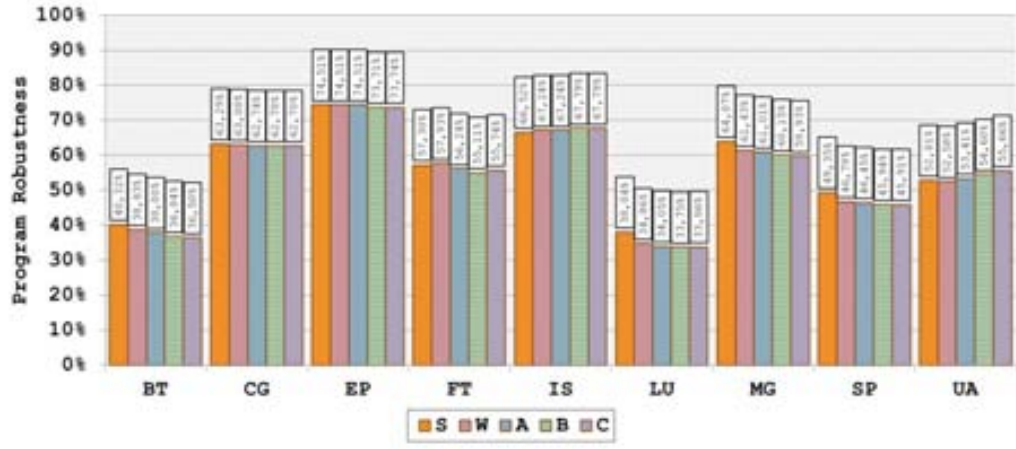

Figure 38 – Robustness for NPB.

This low deviation implies that we could evaluate the robustness of the smaller workloads to test the programs with fault detection and protection mechanisms. These evaluations with the smaller workload should test enough of the program/algorithm to the evaluation be considered valid also for the same program/algorithm with larger workloads.

7.3.1.4 SPEC CPU2000 Benchmarks Evaluation

For this section we calculated the robustness against transient faults of all SPEC CPU2000 benchmarks [Standard Performance Evaluation Corporation, 2007], with two distinct workloads each, using ARTFUL methodology tools with and without simplification.

Even been retired back in 2007, the SPEC CPU2000 is a benchmark set largely used for experiments using simulators.

All 26 benchmark programs used in this section were compiled using GNU C and Fortran in their version 4.4.6, with maximum code optimization during compilation (O3).

All the computing nodes used in this set of experiments have CentOS version 6 operating system with 64 bits kernel version 2.6.32. The hardware of all computing nodes used in this work has eight 1.6 GHz AMD Opteron processors with eight cores each and 256 gigabytes of RAM.

Table 11 and Table 12 show a summary of the numbers we obtained with this set of experiments. The tables contain: the program used in the evaluation (Benchmark), the workload used in the evaluation (“t” for a small test workload; “r” for a reference workload), the standard program execution time without any kind of interference (Execution Time), how much time took to generate the traces (Trace Generation Time), the size of the basic block information trace file (BBI File Size), the amount of unique

basic blocks recognized by the trace generator (Unique Basic Blocks), the size of the packed basic block sequences file (PBB File Size), the total amount of instructions that were traced during the trace generation step (Instructions Executed), the percentage of the instructions that were executed by the program and not by some library (Program Influence), the robustness calculated (Robustness), the time spent on the robustness analysis without simplification (Analysis Time Without Simplification), the time spent on the robustness analysis with simplification (Analysis Time With Simplification) and the amount of instructions that the analyses could simplify (Simplified Instructions).

Table 11 – Summary of experimental evaluation results for SPEC (1/2).

Benchmark	Workload	Execution Time (in seconds)	Trace Generation Time (in seconds)	BBI File Size (in Mbytes)	Unique Basic Blocks	PBB File Size (in Mbytes)	Instructions Executed by the Benchmark
164.GZIP	t	0,72	134,77	1,44	2.955	36,02	2.199.854.877
164.GZIP	r	23,31	4.764,00	1,44	2.935	984,55	67.786.449.431
168.WUPWISE	t	2,36	238,89	2,15	4.350	6,28	10.890.307.710
168.WUPWISE	r	76,06	8.518,00	2,16	4.362	6,31	375.252.013.209
171.SWIM	t	0,16	7,27	2,16	4.200	0,12	407.431.780
171.SWIM	r	67,31	1.238,02	2,28	4.364	1,77	202.678.687.528
172.MGRID	t	4,94	117,90	2,12	4.150	0,15	18.213.573.464
172.MGRID	r	123,31	2.975,87	2,22	4.261	0,79	455.332.034.734
173.APPLU	t	0,07	5,81	3,96	4.480	0,14	244.733.911
173.APPLU	r	98,56	4.526,00	3,95	4.471	0,30	309.344.484.149
175.VPR	t	0,51	90,67	2,41	5.087	58,89	1.200.306.989
175.VPR	r	44,62	5.841,00	3,17	6.436	2.367,03	66.282.330.326
176.GCC	t	0,52	124,26	15,32	39.361	33,44	1.332.080.434
176.GCC	r	8,48	1.065,91	15,42	39.640	52,59	23.127.069.920
177.MESA	t	0,64	83,98	2,35	4.351	0,60	2.783.304.645
177.MESA	r	64,13	5.646,00	2,11	4.145	69,17	234.053.132.033
178.GALGEL	t	0,74	44,15	4,77	7.856	1,10	3.418.320.099
178.GALGEL	r	71,04	1.983,62	4,77	7.843	5,63	303.217.945.968
179.ART	t	0,72	40,13	1,13	2.319	0,06	1.719.273.390
179.ART	r	24,40	2.255,07	1,25	2.586	0,21	54.485.996.407
181.MCF	t	0,07	12,23	1,37	2.926	7,66	115.342.323
181.MCF	r	61,23	4.215,00	1,37	2.921	608,81	44.750.047.001
183.EQUAKE	t	0,27	21,29	1,81	3.017	1,97	739.364.983
183.EQUAKE	r	105,68	1.050,94	1,90	3.157	6,83	90.490.804.068
186.CRAFTY	t	1,14	262,04	3,31	6.943	77,68	3.150.883.466
186.CRAFTY	r	48,33	13.552,00	3,42	7.163	2.722,14	139.353.806.057
187.FACEREC	t	3,63	102,91	2,94	5.629	9,25	4.779.027.741
187.FACEREC	r	126,09	6.204,00	2,96	5.660	309,35	259.033.771.886
188.AMMP	t	2,84	235,94	2,58	4.351	11,92	5.222.917.578
188.AMMP	r	100,46	10.294,00	2,56	4.454	445,65	318.032.699.535
189.LUCAS	t	1,27	186,99	1,87	3.278	10,49	5.815.761.632
189.LUCAS	r	65,01	4.269,00	2,67	3.717	115,37	226.779.381.088
191.FMA3D	t	-	2,87	3,38	6.814	0,26	5.732.824
191.FMA3D	r	125,36	2.665,65	4,55	8.044	45,29	225.704.844.554
197.PARSER	t	0,98	229,23	4,12	10.043	64,40	2.532.769.846
197.PARSER	r	128,21	23.319,00	4,29	10.399	1.325,04	267.830.255.714
200.SIXTRACK	t	2,71	300,66	6,32	10.446	11,25	9.518.123.545
200.SIXTRACK	r	101,37	4.627,12	6,37	10.485	12,12	549.824.854.514
252.EON	t	0,02	5,32	4,05	7.249	0,76	59.132.405
252.EON	r	14,37	2.029,95	4,12	7.359	59,03	47.239.649.567
253.PERLBMK	t	4,37	9,46	3,75	9.282	0,48	4.753.025
253.PERLBMK	r	9,02	1.964,81	5,92	14.481	61,20	28.534.480.080
254.GAP	t	0,27	64,62	4,07	9.349	13,03	723.721.055
254.GAP	r	49,65	11.346,00	4,28	9.595	357,63	182.690.386.754
255.VORTEX	t	1,89	348,67	6,66	14.072	7,72	7.864.055.624
255.VORTEX	r	24,24	4.831,00	6,67	14.059	86,22	101.226.204.184
256.BZIP2	t	1,95	163,33	1,43	2.836	85,02	8.890.045.237
256.BZIP2	r	25,96	4.566,00	1,47	2.919	1.905,13	79.505.798.714
300.TWOLF	t	0,08	11,78	3,24	6.542	4,12	203.483.885
300.TWOLF	r	139,12	13.162,00	3,44	6.917	3.606,20	297.811.272.751
301.APSI	t	1,85	53,67	3,81	6.348	1,04	5.718.539.651
301.APSI	r	118,77	2.938,82	4,01	6.633	1,77	341.547.636.231

Table 12 – Summary of experimental evaluation results for SPEC (2/2).

Benchmark	Workload	Program Influence	Robustness	Analysis Time Without Simplification (in seconds)		Analysis Time With Simplification (in seconds)	Amount of Simplified Instructions
164.GZIP	t	99,4318%	85,46%	1.343,68		774,79	43,12%
164.GZIP	r	99,0896%	85,09%	41.872,50	*	29.852,00	28,51%
168.WUPWISE	t	99,9985%	66,98%	6.577,00		294,98	95,57%
168.WUPWISE	r	99,9998%	64,78%	231.797,65	*	2.510,31	98,90%
171.SWIM	t	83,6046%	46,59%	246,61		37,23	85,45%
171.SWIM	r	99,5553%	41,86%	125.197,05	*	537,80	99,62%
172.MGRID	t	99,9563%	21,18%	10.875,00		9,98	99,91%
172.MGRID	r	99,9570%	21,17%	281.264,04	*	240,69	99,92%
173.APPLU	t	97,1514%	47,47%	147,99		15,20	90,09%
173.APPLU	r	99,9843%	44,06%	191.085,78	*	105,65	99,94%
175.VPR	t	95,3157%	84,72%	738,61		712,94	3,50%
175.VPR	r	99,0651%	78,81%	40.943,39	*	34.338,00	15,56%
176.GCC	t	94,2315%	86,96%	831,14		699,34	16,28%
176.GCC	r	78,7260%	81,44%	14.285,87	*	3.033,44	78,97%
177.MESA	t	34,3162%	79,07%	1.725,48		48,53	97,21%
177.MESA	r	94,1862%	69,64%	144.577,42	*	119.629,00	16,53%
178.GALGEL	t	70,6120%	59,49%	2.042,09		68,10	96,77%
178.GALGEL	r	67,7932%	57,55%	187.301,35	*	627,71	99,66%
179.ART	t	99,6596%	69,03%	1.023,46		4,74	99,56%
179.ART	r	99,9782%	65,26%	33.656,65	*	21,96	99,94%
181.MCF	t	93,9946%	81,93%	70,49		38,99	45,63%
181.MCF	r	99,0018%	80,43%	27.642,64	*	11.114,00	59,44%
183.EQUAKE	t	49,5511%	72,87%	507,49		171,08	62,47%
183.EQUAKE	r	95,0348%	55,34%	55.897,25	*	4.978,00	90,91%
186.CRAFTY	t	99,9690%	87,54%	1.947,18		1.876,99	3,71%
186.CRAFTY	r	99,9985%	87,85%	86.080,51	*	82.065,00	4,73%
187.FACEREC	t	67,8621%	65,91%	2.896,07		710,14	76,47%
187.FACEREC	r	85,3654%	67,68%	160.008,25	*	25.622,00	84,10%
188.AMMP	t	91,9953%	64,58%	3.163,27		620,88	80,70%
188.AMMP	r	98,4288%	54,91%	196.452,60	*	35.518,00	81,66%
189.LUCAS	t	76,0025%	61,41%	3.543,74		638,75	82,14%
189.LUCAS	r	84,5493%	49,76%	140.084,33	*	5.525,00	96,06%
191.FMA3D	t	8,8999%	86,44%	3,95		3,18	21,96%
191.FMA3D	r	93,8861%	60,20%	139.420,58	*	9.943,00	92,93%
197.PARSER	t	95,9409%	86,25%	1.550,47		1.020,75	35,35%
197.PARSER	r	97,0723%	86,32%	165.441,95	*	124.875,00	24,77%
200.SIXTRACK	t	80,9245%	62,31%	5.799,00		2.578,82	56,13%
200.SIXTRACK	r	99,6573%	37,07%	339.633,38	*	8.332,00	97,51%
252.EON	t	95,1603%	77,70%	36,60		31,56	14,27%
252.EON	r	99,7377%	77,65%	29.180,50	*	17.586,00	39,19%
253.PERLBMK	t	63,7579%	86,78%	3,35		2,78	19,57%
253.PERLBMK	r	93,4172%	88,03%	17.626,09	*	8.965,00	49,82%
254.GAP	t	92,2895%	87,04%	448,60		361,58	19,94%
254.GAP	r	99,9690%	84,00%	112.850,03	*	49.985,00	55,67%
255.VORTEX	t	98,2349%	86,92%	4.871,00		2.752,34	49,32%
255.VORTEX	r	98,7383%	86,29%	62.528,64	*	26.047,00	58,74%
256.BZIP2	t	99,9903%	80,94%	5.292,00		864,63	84,21%
256.BZIP2	r	99,9725%	84,00%	49.111,68	*	22.359,00	54,39%
300.TWOLF	t	96,0373%	83,40%	124,99		116,20	7,13%
300.TWOLF	r	99,8505%	81,77%	183.961,58	*	133.629,00	26,72%
301.APSI	t	96,0335%	60,33%	3.419,48		182,61	94,79%
301.APSI	r	99,5750%	61,03%	210.978,05	*	429,98	99,80%

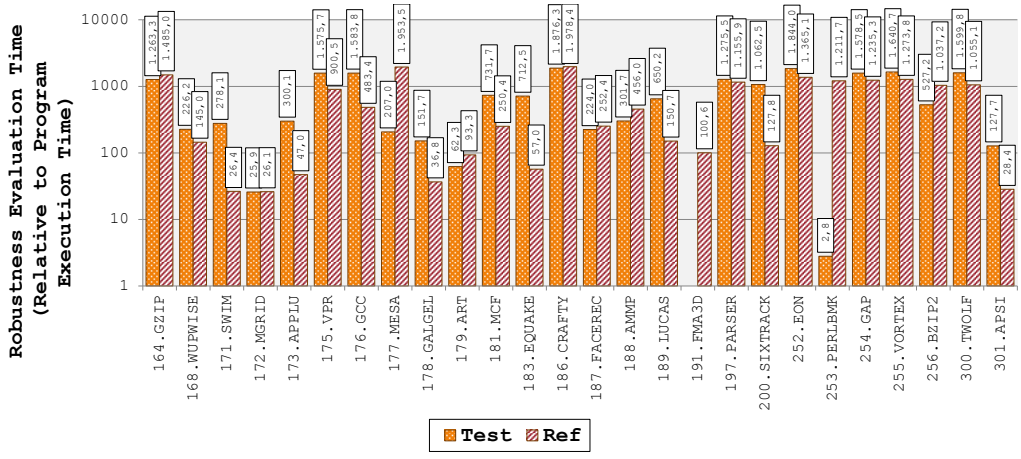

Figure 39 – Robustness evaluation time for SPEC.

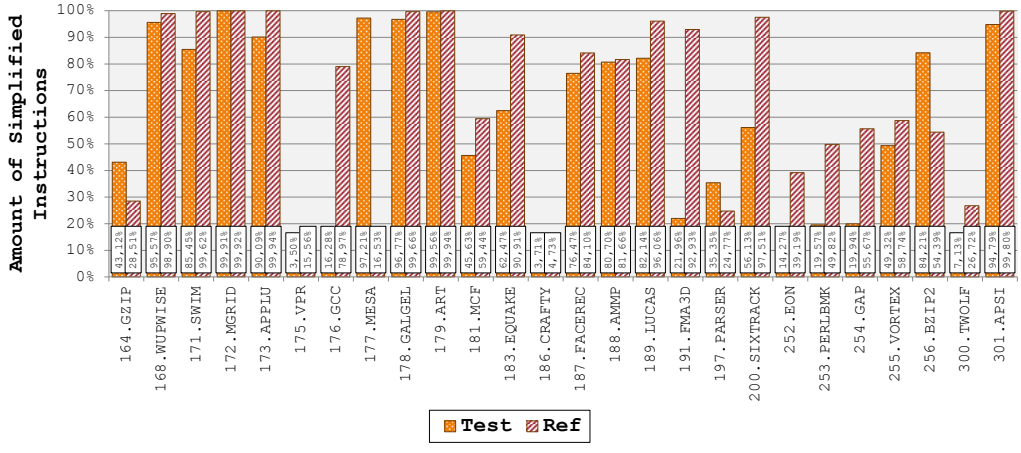

Figure 40 – Amount of simplified instructions for SPEC.

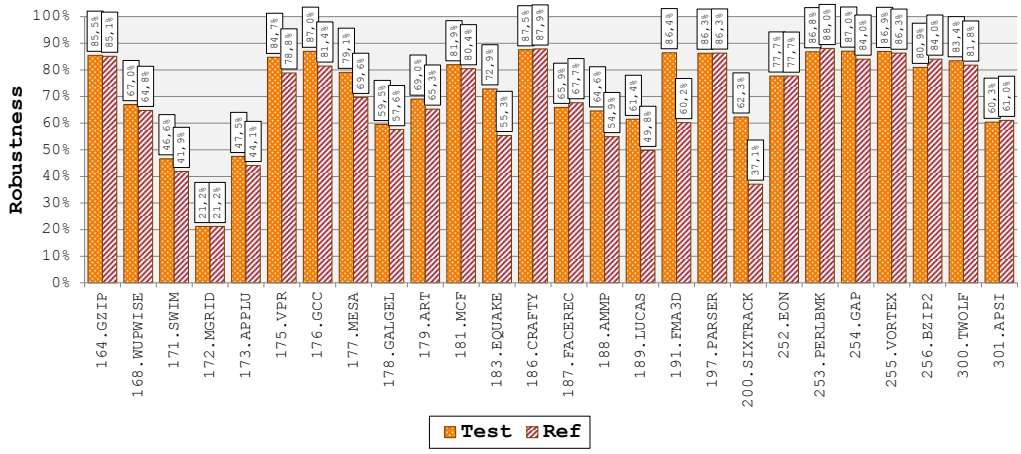

Figure 41 – Robustness for SPEC.

7.3.2 PAS2P

PAS2P [Wong, Rexachs and Luque, 2010] instruments a MPI program and executes parallel programs in a base machine, producing a trace log. The collected data is used to characterize the computation and communication behavior of the program. In order to obtain a machine-independent program model, the trace is logged using a logical global clock according to causality relations between communication events.

Once PAS2P generates the logical trace, it processes the trace using a technique that searches for similarity to identify and extract the most relevant phases and assign them a weight based on the number of times they occur. The signature will be defined by a set of phases and weights.

The execution of the signature in different target systems allows us to measure each phase execution time, and predict the program execution time in each target machine by extrapolating each phase's execution time using the obtained weights.

It is important to notice that the signature creation and execution is a two-step process:

1. The first step is the analysis of the program, the building of the model and subsequent extraction of its phases and weights.
2. The second step is the prediction method where PAS2P executes the signature in a target machine to measure the phases' execution time and predict the program execution time.

7.3.2.1 Parallel application model

To create the signature, first PAS2P build a model (Machine-Independent Model) of the application and then use that model to perform the predictions.

By instrumenting the MPI program, PAS2P obtain a program communication and computation trace that contains all the communications events between processes and computation time elapsed between MPI primitives.

In this context, an event will be a message sent or a message received. With this information we build the program model and use it to study at what point of the program the most computing time is spent (relevant phases), and how many times those phases are repeated (weights).

In the Figure 42 (a) we show an example of a hypothetic program trace generated by PAS2P, with the phases recognized as P0, P1, P2, P3, P4, P5, P6 and P7.

The result of the PAS2P trace analysis is presented on Figure 42 (b), showing all program phases ordered by its relevancy in the total program execution time. In this

example PAS2P has detected that only two of the program phases (P1 and P2) are the most relevant to the execution time prediction.

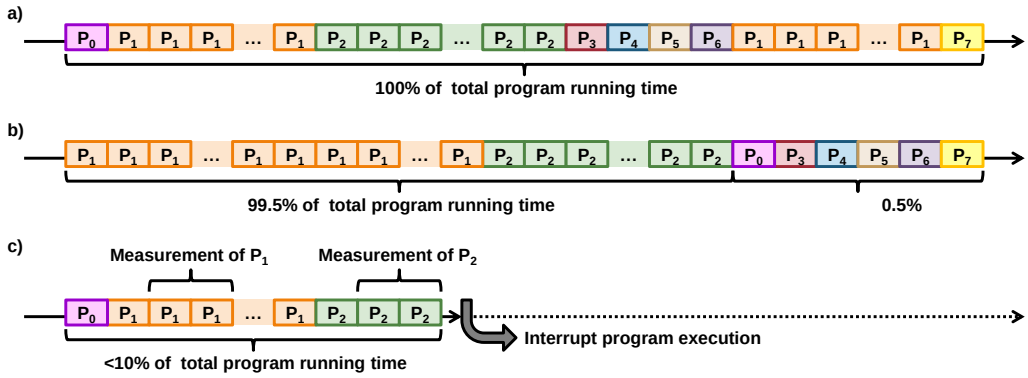

Figure 42 - PAS2P trace generation (a), analysis (b) and signature execution (c)

In the last step of its analysis PAS2P determines a way of executing the minimum fraction of the program, as shown in Figure 42 (c), which allows measuring the time needed to execute an amount of those phases assumed most relevant. PAS2P will try to determine the measurement start and finish by leaving some warming up phases before the measurement. PAS2P will also try to run a maximum of 100 phase's repetitions to be able to calculate a good average time for one phase. These measured averages will be used with the weights of each phase to predict the total program execution time.

7.3.2.2 Performance Prediction

The executable signature runs the parallel program from the beginning and measures the time spent from the point a phase begins until its end. When a phase has been measured, PAS2P continues the program execution until a new relevant phase is found.

The signature repeats this method and proceeds to execute all constituent phases. When the last phase has been measured, the signature finalizes its execution that is often just a small fraction of the whole program execution.

The prediction of the program total execution time is a matter of adding the multiplication of each phase execution time by its weight as:

$$\text{PET} = \sum_{i=1}^{k} PhaseT_i \times W_i$$

Where PET is the predicted execution time, k is the number of phases, $PhaseT_i$ is the phase i execution time and W_i is the phase i weight.

7.3.2.3 Evaluating a PAS2P Signature Robustness

In order to combine both the robustness evaluation methodology and PAS2P methodology we changed the trace generation tools (the one that generates the basic block information for the robustness analysis and the one that generates PAS2P phase's information) to cooperate during their execution.

In Figure 43 we present an example of a basic block trace activity associated to the phases analysed by PAS2P during the hypothetical program execution.

In this example, P0 to P7 are PAS2P recognized phases of the program and BB0 to BB19 are basic blocks executed by the program.

The basic blocks trace activity shown in Figure 3 will have all basic blocks executed by the complete execution of the evaluated program.

The analysis of this trace can calculate either the whole program execution robustness or a per basic block robustness. In both cases the robustness evaluated will be informed after analysing the whole basic block trace file.

Our strategy to combine both methodologies (and tools) was to make PAS2P inform the basic block tracing tool about the beginning and the end of each measured phase.

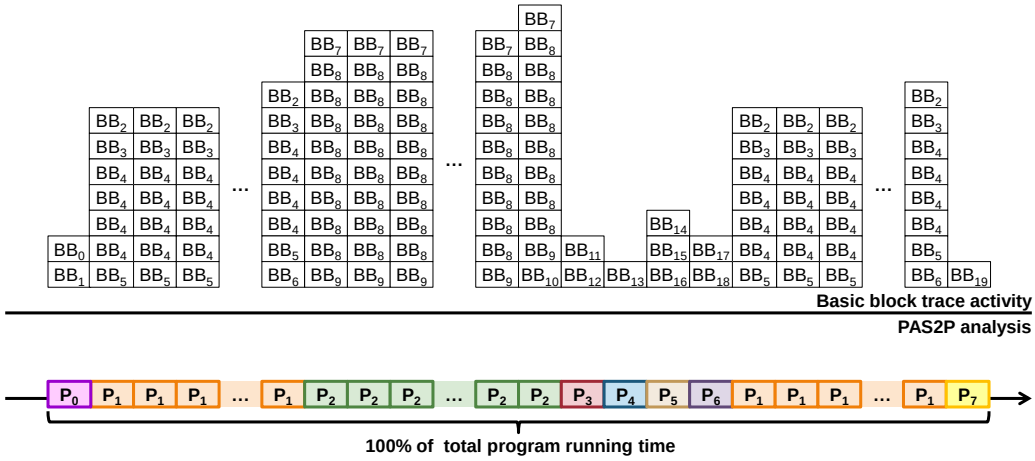

Figure 43 – Basic block trace and PAS2P analysis phase's example.

With this interaction between the tools, the trace generated with basic block information stores two new types of information:

1. A phase start tag in the beginning of every phase being measured by PAS2P;
2. A phase finish tag in the end of every phase being measured by PAS2P.

In Figure 44 we present a trace activity of a PAS2P signature execution. During this evaluation, PAS2P will inform the basic block tracing tool by shared memory the

phase that is starting or finishing its measurement. The basic block tracing tool, then, put this information as tags in the trace.

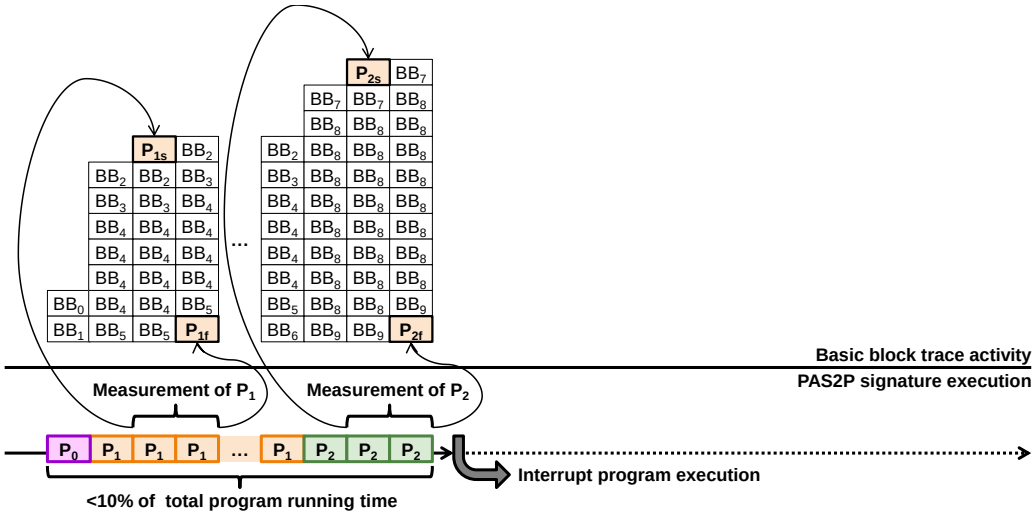

Figure 44 – Basic block trace of a PAS2P signature execution.

The tool that performs the robustness evaluation based on a basic block trace had to be changed too. The new version computes the program robustness the same way as before, but also presents a summary information of the specific robustness of each program phase tagged by PAS2P.

With this summary, and multiplying it by each phase weight informed by the PAS2P analysis, we could predict the whole program robustness with just a fraction of its real execution (the execution of the PAS2P signature). The robustness analysis is performed individually for each process of the parallel program.

The current version of the methodology treats each parallel program process as an individual program that starts, runs and finishes its execution. One basic block trace is generated per each programs processes executed.

7.3.2.4 Experimental Evaluation

In order to realize our experimental evaluation we designed a set of experiments to calculate the robustness against transient faults of five programs. The ARTFUL methodology is applied to both the standard program execution and the PAS2P signature execution of the program being evaluated.

The selected programs are part of the NAS Parallel Benchmark [Bailey et al., 1991] in its version 3.3. Because PAS2P required an MPI based parallel application, we choose to evaluate the MPI versions of BT, CG, FT, LU and SP benchmarks with their B class [NASA Website, 2012].

All five benchmark programs used in this experimental work were compiled using GNU C and Fortran in their version 4.4.1, with static linkage of libraries used by the programs and with maximum code optimization during compilation (O3). Also, all five benchmarks were compiled to run dividing they work between four computing nodes.

The four computing nodes used in the experiments have Linux Ubuntu Server operating system in version 9.10 with 64 bits kernel in version 2.6.31. The version of the OpenMPI library used was 1.4.3. The hardware of all computing nodes used have one 2 GHz AMD Athlon 64 X2 processor with 2 gigabytes of memory.

In the first step of our experimental design we've generated all PAS2P signatures of the programs being evaluated. We also tested the PAS2P prediction tool just to evaluate the prediction of the execution time of the programs.

Figure 45 – Execution time comparison between complete program and PAS2P signature.

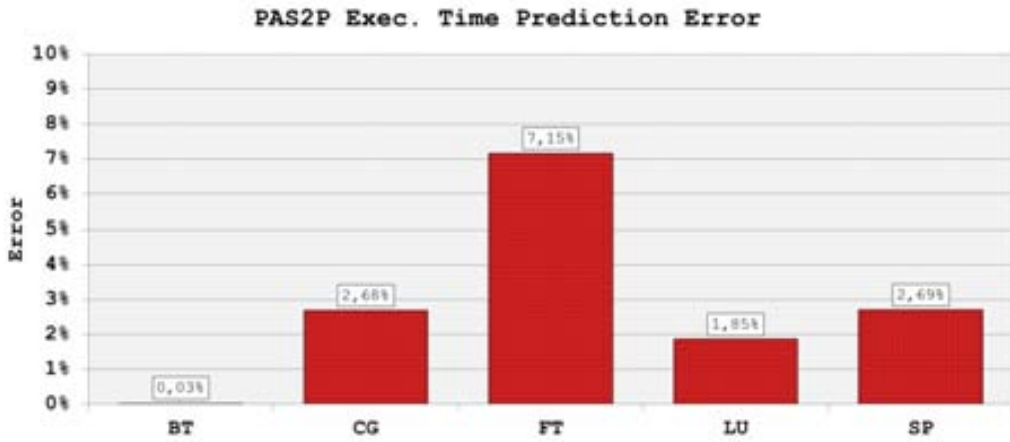

Figure 46 – PAS2P execution time prediction error.

Figure 45 shows the time (wall clock) needed (on average) to completely execute the programs (without any instrumentation) and the time needed to execute the PAS2P signature.

Figure 46 shows the error in the predicted execution time of the evaluated programs by comparing it with the standard program execution time.

Figure 47 shows the trace generation overhead (also in comparison with the standard program executions) of the evaluated programs.

The average error of the predicted execution time was 2.88% (1.81% without taking into account the FT benchmark) and the average overhead in the execution time was 9.84% (4.12% without taking into account the FT benchmark).

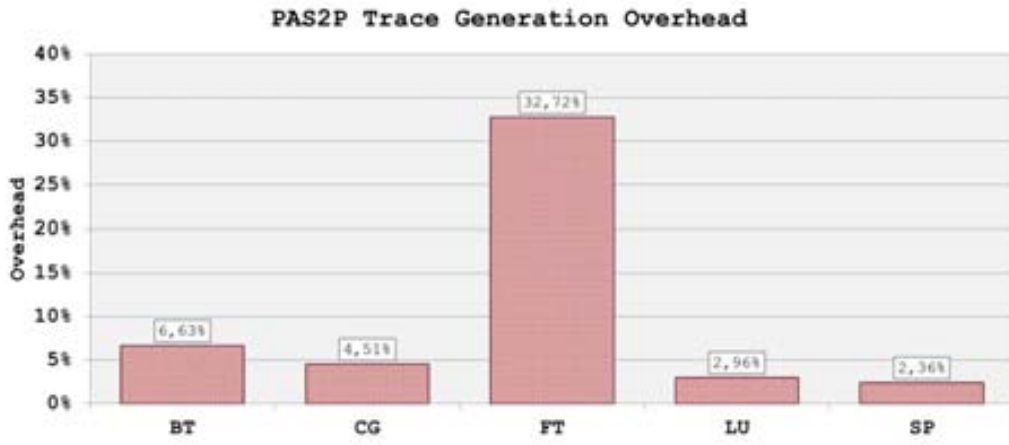

Figure 47 – PAS2P trace generation time overhead.

In both Figure 46 and Figure 47, the FT benchmark presented a particular behaviour, scoring worse than the other programs evaluated. The problem with the FT benchmark is that the workload used to evaluate the program was small enough to allow the PAS2P prediction tool to find phases with enough repetitions to have an accurate evaluation.

So, PAS2P had to execute proportionally more instructions of the program and achieved a worse prediction of the FT benchmark execution time. As we will show in section 5.3, this is not a problem to the robustness prediction.

The second step of our experimental work consisted of, once obtained the PAS2P signature, generating the basic block trace of the complete programs executions and of the PAS2P signature executions.

Figure 48 – Trace generation time for complete program and PAS2P signature.

Figure 49 – Packed basic block trace size for complete program and PAS2P signature.

Figure 48 shows that the trace generations of the PAS2P signatures were considerably faster than the trace generations of the whole programs execution.

Even spending less time by tracing the PAS2P signatures than tracing the whole programs, the size of the basic block traces (in bytes) shown in Figure 49 presents a reduction of less than 10% on average.

This occurs because of:

1. Our basic block tracing tool compresses the traced data based on basic block sequence repetitions on the fly during the trace generation (not after the trace is generated);
2. In the experiments for this work, the portion of the trace file that contains the basic block sequences (more influenced by PAS2P) represented on average 22.9% of the whole trace file, meanwhile the portion of the trace

file with the information about the architecture instructions in the basic blocks (less influenced by PAS2P) represented 77.1%.

Figure 50 – Trace analysis time for complete program and PAS2P signature.

In Figure 50 we present the time needed to calculate the robustness of the evaluated programs and its respective PAS2P signatures.

The LU benchmark achieved the best gain in time saving of the robustness analysis. It needed only 3.57% of the time spent by the standard program analysis to complete its work.

The worst case, as we could foresee, was achieved by the FT benchmark (45.3%).

On average, the traces analysis of the PAS2P signatures needed 16.25% of the time required to analyse the whole programs traces (8.99% without the FT benchmark).

As we previously mentioned, the basic block trace size of the PAS2P signatures weren't significantly smaller than the whole programs basic block traces. However, the amount of instructions analysed in each case of robustness analysis for the PAS2P signatures were significantly smaller (Figure 51).

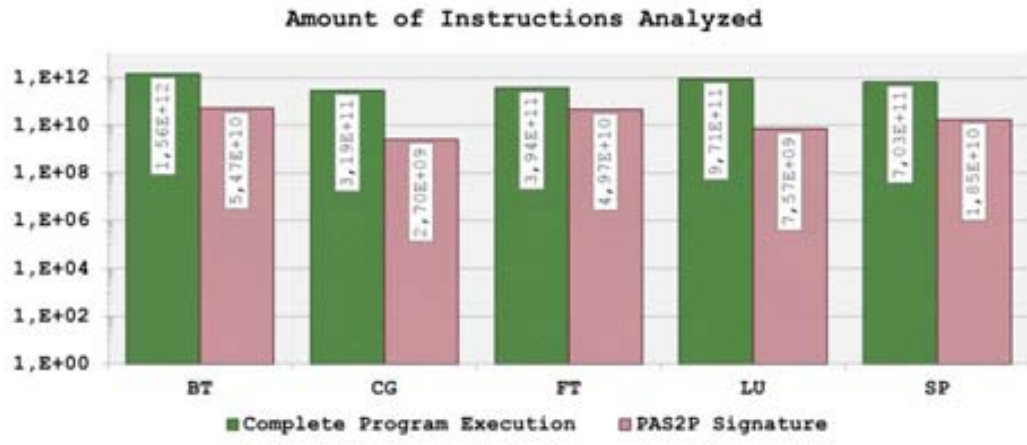

Figure 51 – Amount of instructions analyzed for complete program and PAS2P signature.

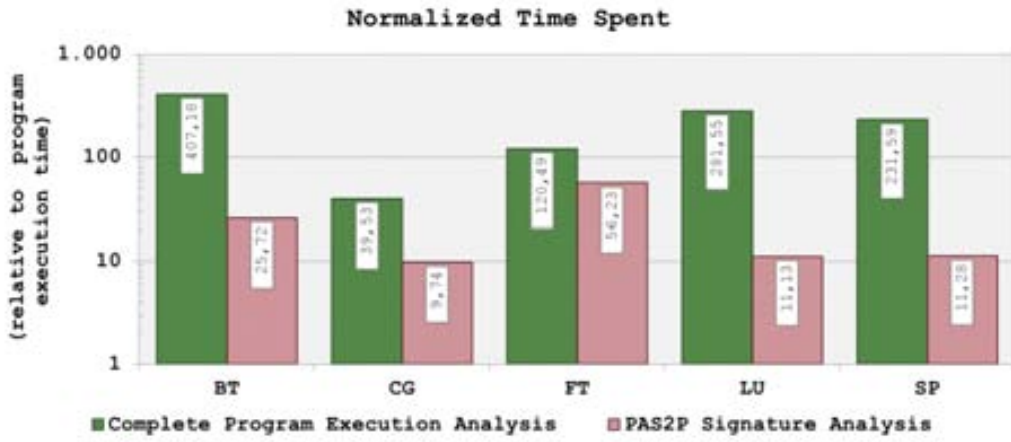

Figure 52 – Normalized time spent for complete program and PAS2P signature.

The robustness analysis of the PAS2P signatures needed, on average, 4.07% (1.94% without the FT benchmark) of the instructions of the whole program analysis to accomplish its work.

In Figure 52 we compare the time needed to perform the whole analysis of the programs' robustness: with and without PAS2P.

The PAS2P Signature Analysis time takes into account not only the robustness trace and the analysis, but also the PAS2P signature generation time and the PAS2P trace analysis time.

While the whole program robustness analysis needed 216 times the programs execution time (on average), using PAS2P and evaluating the robustness of the PAS2P signatures needed 23 times the programs execution time (14 without the FT benchmark).

After evaluating the time needed to calculate the robustness's we compared the results of the calculated robustness (Figure 53) and the robustness prediction of the PAS2P signatures (Figure 54).

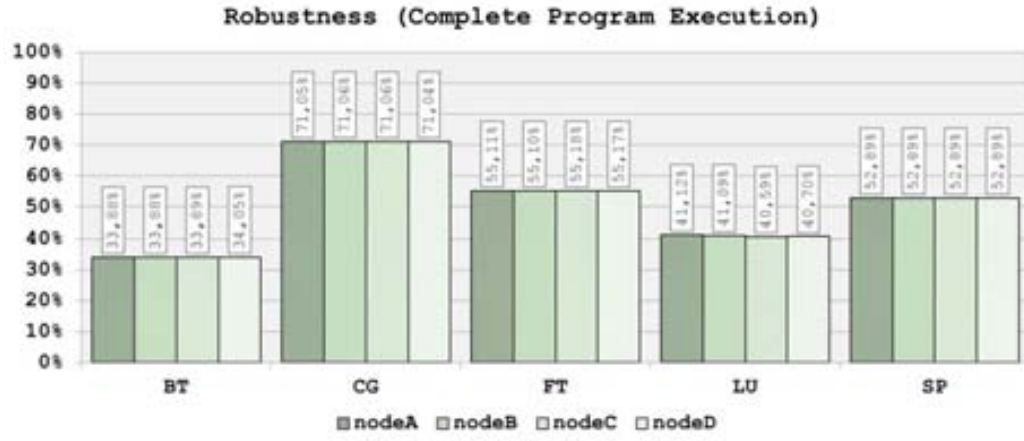

Figure 53 – Per process robustness for complete program execution.

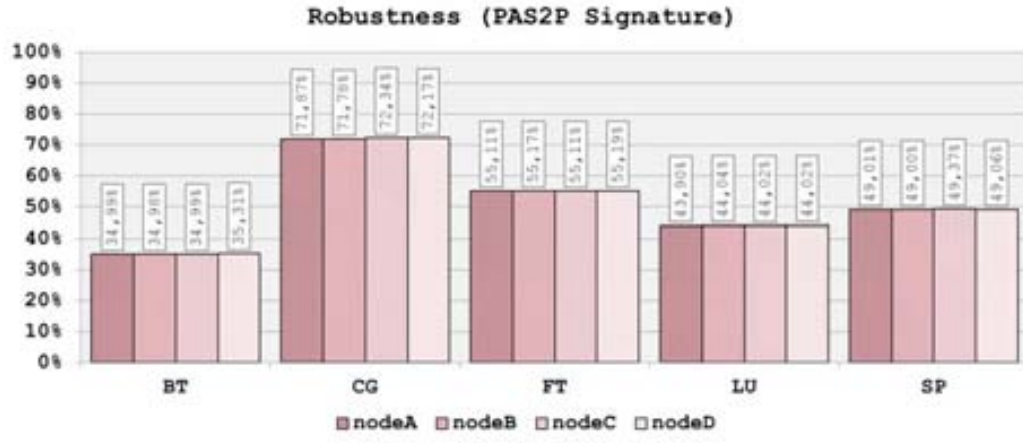

Figure 54 – Per process robustness for PAS2P signature execution.

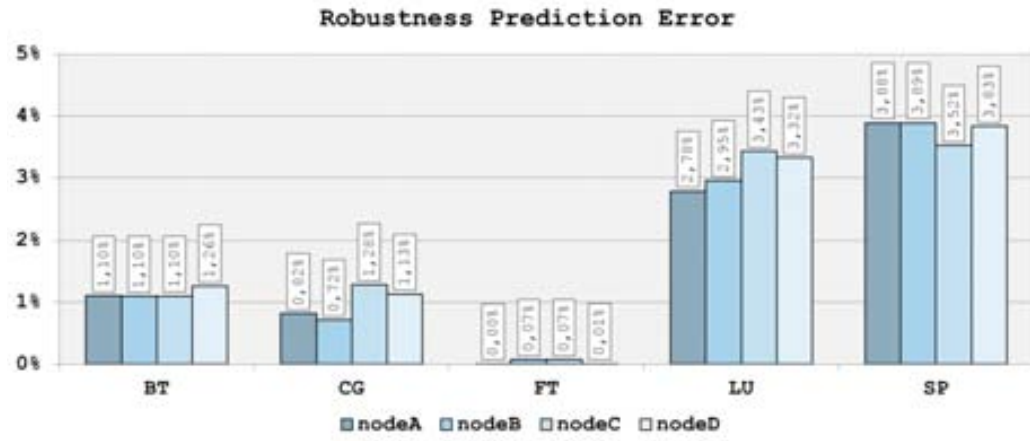

Figure 55 – Robustness prediction error.

The results obtained with the prediction of the PAS2P signatures' robustness were very accurate in comparison with the numbers obtained for the complete programs' executions.

All the PAS2P signatures robustness's predictions achieved an error lower than 4% as show in Figure 55.

As the FT benchmark was the program that needed more execution than the others when running its PAS2P signature, its robustness evaluation took into account more real information and needed less extrapolation, scoring the best robustness prediction of this set of experiments with an error lower than 0.1%.

Chapter 8
Conclusion and Future Work

8.1 Conclusion

We started researching about transient faults and how they affect a program execution about five years ago, back in 2008.

In the beginning, we studied about software based fault injections, trying to understand the effects that transient faults could cause in program executions in HPC systems.

One thing that we learned in the first years was that evaluating a program's robustness against transient faults using software based fault injection environments is a very expensive task given the amount of CPU time needed to obtain a statistical approximation of the desired result. Even using any type of parallelism!

With this limitation in our thoughts, we started the last part of this five years journey with the motivation to make the process of evaluating a program robustness against transient faults feasible for common HPC processor architectures and also for parallel programs. We wanted to be able to execute not only tiny benchmarks (of fractions of benchmarks as some fault injection campaigns suggests), but larger benchmarks, including using parallel frameworks like Messaging Passing Interface (MPI).

In our work we proposed a methodology to precisely calculate a program robustness against transient faults that was equivalent to an exhaustive fault injection campaign and performing all the evaluation with developed tools for both serial and parallel programs in a time significantly smaller than using fault injection campaigns most of the times.

The methodology allowed us to work in a simplification method, that reduced the amount of instructions needed to be evaluated without affecting the calculated robustness, keeping its precision.

8.1.1 Published Work

ARTFUL methodology main concepts [Gramacho, Rexachs and Luque, 2011] presented in Chapter 5 were published at the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA.

After publishing the main concepts of our methodology, we start working improving the methodology tools efficiency both by the parallel evaluation with prediction using PAS2P presented in 7.3.2 and by the simplification method presented at 7.3.1.

The work about the use of PAS2P to predict parallel program robustness against transient faults is accepted for be published at the International Journal of Computational Science and Engineering, Special Issue on Frontiers in Computer Science and Technology 2012.

The work about the simplification was presented at the 11th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA-13), in Melbourne, Australia.

8.2 Future Work

It is possible to improve even more the efficiency of the robustness evaluation considering that the robustness for each processor register is calculated independently. So, using a multithreaded ARTFUL Analyzer might improve significantly the amount of time needed to calculate the program robustness against transient faults.

Also in the efficiency improvement field, it is possible to prove our methodology with other tools for tracing the program execution. There are other instrumentation tools like Dyninst that might reduce the trace generation time.

Still in the efficiency improvement field, we believe that there are space to improve the compression algorithm we used in out trace generation to compress even more the trace in the “not so bad” and “bad” cases presented in section 7.3.1.3 during the NPB evaluation. This improvement will also provide gains in the analysis time, as more we compress more we can simplify the robustness calculation.

We believe that is possible to work in a way to recognize the fault detection/protection mechanisms coded in the programs and automatically use this information to differentiate a registers that is being used but is protected by the program itself. This evaluation is currently performed manually.

Finally, we believe that is possible to mix our methodology with a database of known software based fault detection/tolerance mechanisms to aid the program

developers on choosing the best way to reduce the risk of having their program results affected by a transient fault without compromising valuable resources like CPU time proving each mechanism with executions with fault injection.

References

Advanced Micro Devices, Inc. (2002-2013) *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*.

Argollo, E., Falcon, A., Faraboschi, P., Monchiero, M. and Ortega, D. (2009) 'COTSon: infrastructure for full system simulation', *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 52-61.

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., Martins, E. and Powell, D. (1990) 'Fault injection for dependability validation: a methodology and some applications', *Software Engineering, IEEE Transactions on*, vol. 16, no. 2, Feb, pp. 166-182.

Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V. and Weeratunga, S.K. (1991) *The NAS Parallel Benchmarks, Technical Report RNR-94-007*, 1994th edition, NASA Ames Research Center.

Baumann, R. (2005) 'Soft errors in advanced computer systems', *Design & Test of Computers, IEEE*, vol. 22, no. 3, May-June, pp. 258-266.

Bronevetsky, G. and Supinski, B.d. (2008) 'Soft error vulnerability of iterative linear algebra methods', ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, 155-164.

Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B. and Snir, M. (2009) 'Toward Exascale Resilience', *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374-388.

Constantinescu, C. (2005) 'Dependability benchmarking using environmental test tools', Reliability and Maintainability Symposium, 2005. Proceedings. Annual, 567-571.

Dixit, A. and Wood, A. (2011) 'The impact of new technology on soft error rates', Reliability Physics Symposium (IRPS), 2011 IEEE International, 5B.4.1-5B.4.7.

Döbel, B., Schirmeier, H. and Engel, M. (2013) 'Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment', 5th Workshop on Design for Reliability (DFR 2013).

Dodd, P.E. and Massengill, L.W. (2003) 'Basic mechanisms and modeling of single-event upset in digital microelectronics', *Nuclear Science, IEEE Transactions on*, vol. 50, no. 3, June, pp. 583-602.

Eyes, D. and Lichty, R. (1992) *Programming the 65816 (Including the 6502, 65C02 and 65802)*, The Western Design Center, Inc.

Fidalgo, A.V., Alves, G.R. and Ferreira, J.M. (2006) 'Real time fault injection using a modified debugging infrastructure', On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International, 6.

Gramacho, J. (2009) *Analyzing the effects of transient faults into applications*, Degree dissertation, Universitat Autònoma de Barcelona.

Gramacho, J., Rexachs, D. and Luque, E. (2011) 'A Methodology to Calculate a Program's Robustness against Transient Faults', International Conference on Parallel and Distributed Processing Techniques and Applications, 645-651.

Hari, S.K.S., Adve, S.V. and Naeimi, H. (2012) 'Low-cost program-level detectors for reducing silent data corruptions', *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, vol. 0, pp. 1-12.

Hari, S.K.S., Adve, S.V., Naeimi, H. and Ramachandran, P. (2012) 'Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults', Intl. Conf. on Architectural Support for Programming Languages and Operating Systems.

Hoarau, W., Tixeuil, S. and Vauchelles, F. (2007) 'FAIL-FCI: Versatile fault injection', *Future Generation Computer Systems*, vol. 23, no. 7, pp. 913-919.

Jenn, E., Arlat, J., Rimen, M., Ohlsson, J. and Karlsson, J. (1994) 'Fault injection into VHDL models: the MEFISTO tool', Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on, Austin, TX, USA, 66 - 75.

Kanawati, G.A., Kanawati, N.A. and Abraham, J.A. (1995) 'FERRARI: a flexible software-based fault and error injection system', *Computers, IEEE Transactions on*, vol. 44, no. 2, February, pp. 248-260.

Kudva, P.a.K.J.W., Sanda, P.N., McBeth, R., Schumann, J. and Kalla, R. (2007) 'Fault Injection Verification of IBM POWER6 Soft Error Resilience', Proceedings of the Workshop on Architectural Support for Gigascale Integration.

Lesiak, A., Gawkowski, P. and Sosnowski, J. (2007) 'Error Recovery Problems', Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on, 270-277.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J. and Hazelwood, K. (2005) 'Pin: building customized program analysis tools with dynamic instrumentation', Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, 190-200.

Matsumoto, M. and Nishimura, T. (1998) 'Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator', *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, January, pp. 3-30.

Mitra, S., Zhang, M., Seifert, N., Mak, T.M. and Kim, K.S. (2006) 'Soft Error Resilient System Design through Error Correction', Very Large Scale Integration, 2006 IFIP International Conference on, 332-337.

Mukherjee, S. (2008) *Architecture Design for Soft Errors*, Morgan Kaufmann.

Mukherjee, S.S., Emer, J. and Reinhardt, S.K. (2005) 'The Soft Error Problem: An Architectural Perspective', HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 243-247.

Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K. and Austin, T. (2003) 'A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor', MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, 29.

NASA Website (2012) *Problem Sizes and Parameters in NAS Parallel Benchmarks.*, 19 March, [Online], Available: http://www.nas.nasa.gov/publications/npb_problem_sizes.html.

Nicolescu, B., Savaria, Y. and Velazco, R. (2003) 'SIED: Software Implemented Error Detection', DFT '03: Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 589.

Oh, N., Shirvani, P.P. and McCluskey, E.J. (2002) 'Error detection by duplicated instructions in super-scalar processors', *Reliability, IEEE Transactions on*, vol. 51, no. 1, March, pp. 63-75.

Oliner, A. and Stearley, J. (2007) 'What Supercomputers Say: A Study of Five System Logs', Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on, 575-584.

Reis, G.A., Chang, J., August, D.I., Cohn, R. and Mukherjee, S.S. (2006) 'Configurable Transient Fault Detection via Dynamic Binary Translation', Proceedings of the 2nd Workshop on Architectural Reliability.

Reis, G.A., Chang, J., Vachharajani, N., Rangan, R. and August, D.I. (2005) 'SWIFT: software implemented fault tolerance', *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, March, pp. 243-254.

Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I. and Mukherjee, S.S. (2005) 'Design and Evaluation of Hybrid Fault-Detection Systems', ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture, 148-159.

Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I. and Mukherjee, S.S. (2005) 'Software-controlled fault tolerance', *ACM Trans. Archit. Code Optim.*, vol. 2, no. 4, pp. 366-396.

Shivakumar, P., Kistler, M., Keckler, S.W., Burger, D. and Alvisi, L. (2002) 'Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic', DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, 389-398.

Sosnowski, J., Gawkowski, P., Zygulski, P. and Tymoczko, A. (2006) 'Enhancing Fault Injection Testbench', Dependability of Computer Systems, 2006. DepCos-RELCOMEX '06. International Conference on, 78-83.

Sridharan, V. and Kaeli, D.R. (2008) 'Quantifying software vulnerability', Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies.

Sridharan, V. and Kaeli, D.R. (2009) 'Eliminating microarchitectural dependency from Architectural Vulnerability', High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on, 117-128.

Standard Performance Evaluation Corporation (2007) *SPEC CPU2000*, 07 June, [Online], Available: http://www.spec.org/cpu2000/ [12 August 2013].

Stott, D.T., Floering, B., Burke, D., Kalbarczpk, Z. and Iyer, R.K. (2000) 'NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors', Computer Performance and Dependability Symposium (IPDS 2000), Chicago, IL, 91-100.

The Western Design Center, Inc. (1981-2003) *W65C02S 8–bit Microprocessor*.

Wang, N.J., Quek, J., Rafacz, T.M. and Patel, S.J. (2004) 'Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline', DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, Washington, DC, USA, 61.

Wong, A., Rexachs, D. and Luque, E. (2010) 'Extraction of Parallel Application Signatures for Performance Prediction', High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on, 223 -230.

Yount, C.R. and Siewiorek, D.P. (1996) 'A Methodology for the Rapid Injection of Transient Hardware Errors', *IEEE Trans. Comput.*, vol. 45, no. 8, pp. 881-891.

Yu, J., Garzaran, M.J. and Snir, M. (2009) 'ESoftCheck: Removal of Non-vital Checks for Fault Tolerance', CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization, Washington, DC, USA, 35-46.