UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

UNIVERSIDAD DE ALICANTE
DEPARTAMENTO DE TECNOLOGÍA INFORMÁTICA Y COMPUTACIÓN
DOCTORADO EN INFORMÁTICA

EDUARDO CHIELLE

Selective Software-Implemented Hardware Fault Tolerance Techniques to Detect Soft Errors in Processors with Reduced Overhead

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Microelectronics

Dr. Fernanda Lima Kastensmidt
Advisor at UFRGS

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Informatics

Dr. Sergio Cuenca-Asensi
Advisor at Universidad de Alicante

April 2016.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Chielle, Eduardo

Selective Software-Implemented Hardware Fault Tolerance Techniques to Detect Soft Errors in Processors with Reduced Overheads / Eduardo Chielle. – Porto Alegre: Programa de Pós-Graduação em Microeletrônica, 2016.

221 f.:il.

Thesis (doctorate) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, RS, Brazil, 2016. Advisor: Fernanda Lima Kastensmidt. Universidad de Alicante. Doctorado en Informática. Alicante, Spain. Advisor: Sergio Cuenca-Asensi.

1. Fault tolerance. 2. Processors. 3. Soft errors. I. Kastensmidt, Fernanda G. L.; Cuenca-Asensi, S. II. Selective Software-Implemented Hardware Fault Tolerance Techniques to Detect Soft Errors in Processors with Reduced Overheads.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PGMICRO: Prof. Fernanda Lima Kastensmidt
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

BID	Basic Block Identifier
BS	Bubble Sort
CCA	Control Flow Checking using Assertions
CEDA	Control flow Error Detection through Assertions
CFCSS	Control Flow Checking by Software Signatures
CFID	Control Flow Identifier
CFT	Configurable Fault Tolerant
DA	Dijkstra's Algorithm
ECCA	Enhanced Control Flow Checking using Assertions
EDDI	Error Detection by Duplicated Instructions
HETA	Hybrid Error-Detection Technique Using Assertions
MM	Matrix Multiplication
MWTF	Mean Work To Failure
PC	Program Counter
rDFS	Recursive Depth-First Search
RLE	Run-Length Encoding
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
sDFS	Sequential Depth-First Search
SEE	Single Events Effects
SET	Single Event Transient
SETA	Software-only Error-detection Technique using Assertions
SEU	Single Event Upset
SIHFT	Software-Implemented Hardware Fault Tolerance
SUM	Summation
SWIFT	Software Implemented Fault Tolerance
SWIFT-R	SWIFT - Recovery
S-SETA	Selective SETA
S-SWIFT-R	Selective SWIFT-R
S-VAR	Selective VAR
TEA2	TETRA Encryption Algorithm 2

TETRA	Terrestrial Trunked Radio
TH	Tower of Hanoi
VAR	Variables
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
YACCA	Yet Another Control-Flow Checking using Assertions

LIST OF FIGURES

LIST OF TABLES

PREFACE

This document is the result of the jointly supervised doctoral work in the *Programa de Pós-Graduação em Microeletrônica* at *Universidade Federal do Rio Grande do Sul* (PGMICRO – UFRGS) and in the *Escuela de Doctorado* at *Universidad de Alicante* (EDUA). During the period of the doctorate work, I have been granted a scholarship from the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES), a Brazilian agency.

In this work, the problem of designing processor-based fault tolerant systems is undertaken. The research is focused on the selective application of software-based techniques to get high fault detection levels with low overhead costs.

This thesis has been performed in the context of the following research project:

- "Development of hybrid fault tolerance techniques for embedded microprocessors" (ref.:292/13), CAPES/DGU. The main goal of the project is the proposal of new hardware/software techniques to improve the tolerance to faults induced by radiation and their automatic application to embedded modern processors.

ABSTRACT

Software-based fault tolerance techniques are a low-cost way to protect processors against soft errors. However, they introduce significant overheads to the execution time and code size, which consequently increases the energy consumption. System operation with time or energy restrictions may not be able to make use of these techniques. For this reason, this work proposes software-based fault tolerance techniques with lower overheads and similar fault coverage to state-of-the-art software techniques. Once detection is less costly than correction, the work focuses on software-based detection techniques.

Firstly, a set of data-flow techniques called VAR is proposed. The techniques are based on general building rules to allow an exhaustive assessment, in terms of reliability and overheads, of different technique variations. The rules define how the technique duplicates the code and insert checkers. Each technique uses a different set of rules. Then, a control-flow technique called SETA (Software-only Error-detection Technique using Assertions) is introduced. Comparing SETA with a state-of-the-art technique, SETA is 11.0% faster and occupies 10.3% fewer memory positions. The most promising data-flow techniques are combined with the control-flow technique in order to protect both data-flow and control-flow of the target application.

To go even further with the reduction of the overheads, methods to selective apply the proposed software techniques have been developed. For the data-flow techniques, instead of protecting all registers, only a set of selected registers is protected. The set is selected based on a metric that analyzes the code and rank the registers by their criticality. For the control-flow technique, two approaches are taken: (1) removing checkers from basic blocks: all the basic blocks are protected by SETA, but only selected basic blocks have checkers inserted, and (2) selectively protecting basic blocks: only a set of basic blocks is protected. The techniques and their selective versions are evaluated in terms of execution time, code size, fault coverage, and Mean Work To Failure (MWTF), which is a metric to measure the trade-off between fault coverage and execution time. Results show that was possible to reduce the overheads without affecting the fault coverage, and for a small reduction in the fault coverage it was possible to significantly reduce the overheads. Lastly, since the evaluation of all the possible combinations for selective hardening of every application takes too much time, this work uses a method to extrapolate the results obtained by simulation in order to find the parameters for the selective combination of data and control-flow techniques that are probably the best candidates to improve the trade-off between reliability and overheads.

Keywords: SIHFT techniques, selective hardening, transient faults, soft errors, Single Event Effects, SEU, SET, processor, reliability, execution time, code size, energy consumption, lower overheads.

Técnicas Seletivas de Tolerância a Falhas em Software com Custo Reduzido para Detectar Erros Causados por Falhas Transientes em Processadores

RESUMO

A utilização de técnicas de tolerância a falhas em *software* é uma forma de baixo custo para proteger processadores contra *soft errors*. Contudo, elas causam aumento no tempo de execução e utilização de memória. Em consequência disso, o consumo de energia também aumenta. Sistemas que operam com restrição de tempo ou energia podem ficar impossibilitados de utilizar tais técnicas. Por esse motivo, este trabalho propoe técnicas de tolerância a falhas em *software* com custos no desempenho e memória reduzidos e cobertura de falhas similar a técnicas presentes na literatura. Como detecção é menos custoso que correção, este trabalho foca em técnicas de detecção.

Primeiramente, um conjunto de técnicas de dados baseadas em regras de generalização, chamada VAR, é apresentada. As técnicas são baseadas nesse conjunto generalizado de regras para permitir uma investigação exaustiva, em termos de confiabilidade e custos, de diferentes variações de técnicas. As regras definem como a técnica duplica o código e insere verificadores. Cada técnica usa um diferente conjunto de regras. Então, uma técnica de controle, chamada SETA, é introduzida. Comparando SETA com uma técnica estado-da-arte, SETA é 11.0% mais rápida e ocupa 10.3% menos posições de memória. As técnicas de dados mais promissoras são combinadas com a técnica de controle com o objetivo de proteger tanto os dados quanto o fluxo de controle da aplicação alvo.

Para reduzir ainda mais os custos, métodos para aplicar seletivamente as técnicas propostas foram desenvolvidos. Para técnica de dados, em vez de proteger todos os registradores, somente um conjunto de registradores selecionados é protegido. O conjunto é selecionado com base em uma métrica que analisa o código e classifica os registradores por sua criticalidade. Para técnicas de controle, há duas abordagens: (1) remover verificadores de blocos básicos, e (2) seletivamente proteger blocos básicos. As técnicas e suas versões seletivas são avaliadas em termos de tempo de execução, tamanho do código, cobertura de falhas, e o *Mean Work to Failure (MWTF)*, o qual é uma métrica que mede o compromisso entre cobertura de falhas e tempo de execução. Resultados mostram redução dos custos sem diminuição da cobertura de falhas, e para uma pequena redução na cobertura de falhas foi possível significativamente reduzir os custos. Por fim, uma vez que a avaliação de todas as possíveis combinações utilizando métodos seletivos toma muito tempo, este trabalho utiliza um método para extrapolar os resultados obtidos por simulação com o objetivo de encontrar os melhores parâmetros para a proteção seletiva e combinada de técnicas de dados e de controle que melhorem o compromisso entre confiabilidade e custos.

Palavras-Chave: técnicas de tolerância a falhas em software, proteção seletiva, falhas transientes, soft errors, SEU, SET, processador, confiabilidade, tempo de execução, tamanho do código, consumo de energia, diminuição de custos.

Técnicas Selectivas de Tolerancia a Fallos en Software con Gastos Generales Reducidos para Detectar Errores Causados por Fallos Transitorios en Procesadores

RESUMEN

La utilización de tecnicas de tolerancia a fallos en *software* és un método de bajo costo utilizado para la protección de procesadores contra *soft errors*. Sin embargo, causan el aumento en el tiempo de ejecución y la utilización de memoria. En consecuencia, el consumo de energia también aumenta. Sistemas que operan con restricción de tiempo o energia pueden quedarse imposibilitados de utilizar tales técnicas. Por lo tanto, este trabajo propone técnicas de tolerancia a fallos en *software* con reducción de gastos generales en desempeño y memoria; Además, incorpora la cobertura a fallos similar a técnicas de la literatura. Como detección és menos costosa que corrección, este trabajo se centra en técnicas de detección.

Primeramente, un conjunto de técnicas de dados basadas en reglas de generalización, llamada VAR, és presentada. Las técnicas son baseadas en este conjunto generalizado de reglas para permitir una investigación exhaustiva, en termos de confiabilidad y gastos generales, de diferentes variaciones de técnicas. Las reglas definen como la técnica duplica el código y inserta verificadores. Cada técnica utiliza un conjunto diferente de reglas. Después, una técnica de control, llamada SETA, és introducida. Comparando SETA com una técnica del estado del arte, SETA és 11.0% más rápida y utiliza 10.3% menos posiciones de memoria. Las técnicas de dados más prometedoras son combinadas con la técnica de control con el objetivo de proteger tanto los dados como también el flujo de control de la aplicación.

Posteriormente, fueron desarrollados métodos para aplicar selectivamente las técnicas propuestas. Para técnica de datos, en vez de proteger todos los registros, solamente un conjunto de registros és seleccionado y protegido. El conjunto és selecionado basado en una métrica que analiza el código y ordena los registros por su nivel crítico. Para técnicas de control, hay dos métodos: (1) remoción de verificadores de bloques básicos, y (2) selectivamente proteger bloques básicos. Las técnicas y sus versiones selectivas son evaluadas en términos de tiempo de ejecución, tamaño de código, cobertura de fallos, y por el *Mean Work to Failure (MWTF)*, que és una métrica para medir el compromiso entre la cobertura de fallos y el tiempo de ejecución. Los resultados muestran una reduccíon de los gastos generales sin reducir la cobertura de fallos, y, también, realizando una pequeña reducción en la cobertura de fallos fue posible reduzir significativamente los gastos generales. Finalmente, la evaluación de todas las posibles combinaciones utilizando métodos selectivos requiere mucho tiempo, este trabajo utiliza un método para extrapolar los resultados obtenidos por simulación para encontrar los mejores parámetros para la protección selectiva y combinada de técnicas de datos y de control que mejoren el compromiso entre confiabilidad y gastos generales.

Palabras-Clave: técnicas SIHFT, protección selectiva, fallos transientes, soft errors, SEU, SET, procesador, confiabilidad, tiempo de ejecución, tamaño de código, consumo de energía, reducción de gastos generales.

1 INTRODUCTION

1.1 Motivation

Aerospace applications use dozens to hundreds of processors (FERNANDEZ-LEON, 2013), (ANTHONY, 2012), which are susceptible to transient faults caused by the radiation present in the operating environment (O'BRYAN, 2015). Furthermore, processors, as other integrated circuits, have significantly improved their performance in the last decades due to the advances in the semiconductor industry. Such advances have led to the fabrication of high-density integrated circuits (ICs). We are reaching the physical limits of a couple of atoms to form the transistor's gate (KIM, 2003), (THOMPSON, 2005). On the other hand, the higher quantity of transistors per die combined with reduced voltage threshold and increased operating frequencies have made ICs more sensitive to transient faults caused by radiation (BAUMANN, 2001).

Transient faults can be caused by energized particles present in space or secondary particles, such as alpha particles, generated by the interaction of neutron and materials at flight altitude or ground level (ITRS, 2005). The interaction with an off-state transistor's drain in the PN junction may charge or discharge a node of the circuit. The consequence is a transient pulse in the circuit logic, also known as *Single Event Effect* (SEE). SEE can be potentially destructive, known as *hard errors*, or non-destructive, known as *soft errors* (O'BRYAN, 2015). In this work, we focus on soft errors.

The consequence in circuits can be seen as a *bit flip* (alteration of logical value) in memory elements, known as *Single Event Upset* (SEU), or as a transient pulse in the combinational logic, known as *Single Event Transient* (SET), which can be captured by a memory element. With regards to processor, the consequence of such faults can be seen as an error affecting the control-flow of a running application, where an unexpected branch happens, or as an error affecting the data-flow, where the result of the computation is incorrect, even if the application runs until its end.

In critical systems, errors are unaceptable. Since nowadays critical systems use dozens to hundreds of embedded processors, it is necessary to ensure reliability to these processors in order to provide reliability to the whole system. In this context, it is possible to use radiation hardened processors (*RadHard* processors). However, they present several limitations, such as:

- **Low processing speed**: the operating frequency is significantly lower than commercial processors, as can be seen in Table 1.1
- **High energy consumption**: the energy consumption per task is also higher, mainly because the *RadHard* processor needs considerably more time to perform it

- **High price**: the high cost of *RadHard* devices is becoming a determinant factor to support an increasing use of commercial devices, mainly in the less critical parts of a system (MCHALE, 2014). Besides, the necessity of extra hardware in aerospace applications has an additional cost, because the weight and the physical space that the device occupies are also important factors in this regard (E2V, 2015)
- **Commercial restrictions**: another limitation for using *RadHard* processors that makes the search for other alternatives more interesting is the existence of restrictions for the commerce of *RadHard* devices, due to international trade regulations, such as ITAR (*International Traffic in Arms Regulations*), which makes impossible to many countries to buy many of these devices (CASTRO NETO, 2010).

Table 1.1: Operating frequency and power of *RadHard* processors.

Processor	Frequency (MHz)	Power (W)
GR712RC Dual-Core	100	1.5 / core
UT699 Single-Core	66	not available
LEON3FT-RTAX	25	0.3 – 0.5

Source: (COBHAM GAISLER AB, 2015)

Commercial processors are an alternative for the use of *RadHard* processors. Furthermore, they present some advantages, such as:

- **More recent technology**: commercial processors are developed with the most recent technology, which were still not used in *RadHard* processors
- **High processing speed**: commercial processors operate in significantly higher frequencies, as shown in Table 1.2. It permits, for example, that much of the information be processed on board, before sending to Earth
- **Low power/energy**: there are ultra-low-power models of commercial processors, as one can see in Table 1.2. In general, the energy consumption of commercial processors is lower due to their higher processing speed.
- **Low price**: commercial processors are considerably cheaper, what makes their utilization very interesting to reduce costs. This approach could facilitate the developing and deployment of new systems that could not be developed due to high costs if *RadHard* devices were utilized.

Table 1.2: Operating frequency, power, and price of commercial processors.

Processor	Frequency (MHz)	Power (W)	Price (USD)
TMS320C6748	375-456	0.006 – 0.42	11
ARM Cortex-A9	800 – 2000	0.5 – 1.9	33
Intel I7 960	3200	130	414

Sources: (TEXAS INSTRUMENTS, 2015), (ARM, 2015), (EBAY, 2015), (AMAZON, 2015), and (WANG, 2011).

On the other hand, commercial processors are more sensitive to radiation. Still, it is possible to use software-based fault tolerance techniques to provide reliability to

commercial processors (OH, 2002a), (AZAMBUJA, 2011a), (REIS, 2005b), (GOLOUBEVA, 2003), (VEMU, 2011). These techniques are also known in the literature as *Software-Implemented Hardware Fault Tolerance* (SIHFT) techniques (GOLOUBEVA, 2006). They modify the source code of a target application without modifying the underlying hardware. Therefore, they are applicable to commercial processors. Software-based fault tolerance techniques are capable of detecting a high amount of errors affecting the processors by the effect they cause in the program execution.

Although software redundancy brings reliability to the system, it requires extra processing time, since more instructions are executed. As a consequence, the energy consumption is increased (YAO, 2013), (ASSAYAD, 2013). Furthermore, a reliable program will require more area in memory since software redundancy is inserted. The larger code size increases the probability of cache misses, which increases the number of accesses to the main memory and reduces the performance. Besides, the redundancy of load or store instructions also increases the number of memory accesses. All these extra memory transfers contribute to increase the power consumption (VOGELSANG, 2010), (LI, 2003).

1.2 Objectives and contributions

The aim of this thesis is to provide reliability to commercial processors, with a level of fault coverage similar to state-of-the-art SIHFT techniques, and a significant reduction of overheads in the execution time and memory. Thus, the application will run faster, occupy less space in memory, and, as a consequence, reduce the energy consumption. Furthermore, the faster an application runs, the lower is the probability that it is affected by a transient fault, simply by the fact that the application is executed in less time. Therefore, the application exposure to radiation is lower. Consequently, the reliability is higher. Also, a smaller code size reduces the probability of cache misses.

The main objective of this work consists of reducing drastically overheads in the execution time and memory, keeping the fault coverage similar to state-of-the-art SIHFT techniques. In order to do so, three steps are proposed: (1) development of SIHFT techniques with lower overheads; (2) implementation of selective hardening methods using the proposed techniques to go further in the overhead reduction; and (3) proposition of a method to indicate the best parameters for selective hardening. A detailed list is presented bellow:

1. Decomposition of SIHFT techniques in general building rules to allow an exhaustive assessment, in terms or reliability and overheads, of different technique variations. As a result, the proposal of new techniques specifically designed to combine the protection of both data and control-flow with lower overheads
2. Implementation of selective methods for the proposed data-flow and control-flow techniques. With regards to the data-flow techniques, the selectiveness is based on register selection, i.e., the most critical registers have higher priority to be protected. Concerning the control-flow technique, there are two methods for selectiveness: (a) removing checkers from the least critical basic blocks[1],

[1] A basic block is a branch-free sequence of instructions. This term is better discussed in the following chapters.

and (b) protecting only the most critical basic blocks, i.e., completely removing the protection of the least critical basic blocks

3. Proposal of a method to guide the selective application of the SIHFT techniques in order to find in a shorter time better trade-offs between reliability and performance. A method that indicates the probable best parameters for the selective hardening methods discussed in this work. The goal of this item is to avoid the need for excessive fault tolerance tests. Thus, it is possible to select the recommended parameters. Fault tolerance tests may still be needed, but the number of required tests will be considerably reduced.

All SIHFT techniques and selective methods will be evaluated regarding fault coverage, execution time, code size, and suitable metrics that provide the trade-off between reliability and overheads.

Therefore, the main contributions of this thesis work are:

- Improve the reliability of SIHFT techniques by reducing overheads and keeping similar fault coverage. Achievable with new proposed SIHFT techniques and complemented with existing and proposed selective hardening methodologies
- Provide high reliability with very low overheads using the proposed SIHFT techniques with selective hardening methodologies. It allows the protection of applications with strict performance or energy constraints
- Provide high reliability for applications with limited availability of resources for redundancy. Sometimes, there are no enough resources for applying entirely the SIHFT techniques. In such scenarios, the protection of more sensitive parts may increase significantly reliability. Therefore, it is important to identify and protect these parts
- Propose a model that can guide designers to select how much protection is needed to reach one of the following parameters:
    - Maximum reliability
    - Minimum overhead for a specific fault coverage
    - Maximum reliability for a maximum overhead.

1.3 Thesis organization

Chapter II introduces some definitions and background knowledge necessary for the understanding of the thesis. Chapter III presents the related work. It shows the main SIFHT techniques and selective hardening methods. The fault injection methodology and the metrics used in this work to evaluate the results are exposed in Chapter IV. In Chapter V, one can see the proposed data-flow and control-flow techniques. Then, the selective hardening methods using the proposed techniques is introduced in Chapter VI. Chapter VI also includes a method to estimate the probable bet parameters for selective hardening methods. Finally, Chapter VII draws conclusions, discusses future work, and lists the publications. Additional information can be found in the appendices.

2 DEFINITIONS AND BACKGROUND KNOWLEDGE

This chapter introduces the main definitions and background knowledge for the understanding of the thesis. It presents the sources of radiation and its effects on circuits and processors and shows some techniques to protect the processors against the radiation.

2.1 Sources of ionizing radiation

Ionizing radiation comes from solar flares, solar wind, and cosmic rays. In the interaction with Earth's magnetosphere, some ionizing particles are trapped. The Van Allen belts include two electron belts and one inner proton belt. The inner electron belt contains electrons with energy lower than 5 MeV, and the outer belt contains electrons with energy that may reach 7 MeV. Heavy ions may also be trapped in the magnetosphere (BOUDENOT, 2007).

Cosmic rays entering the atmosphere may interact with the atoms and molecules present. This interaction produces a cascade of lighter particles (air shower) that includes x-rays, muons, protons, alpha particles, pion, electrons, and neutrons (MORRISON, 2008), as shown in Fig. 2.1. Neutrons are typical particles produced in such events.

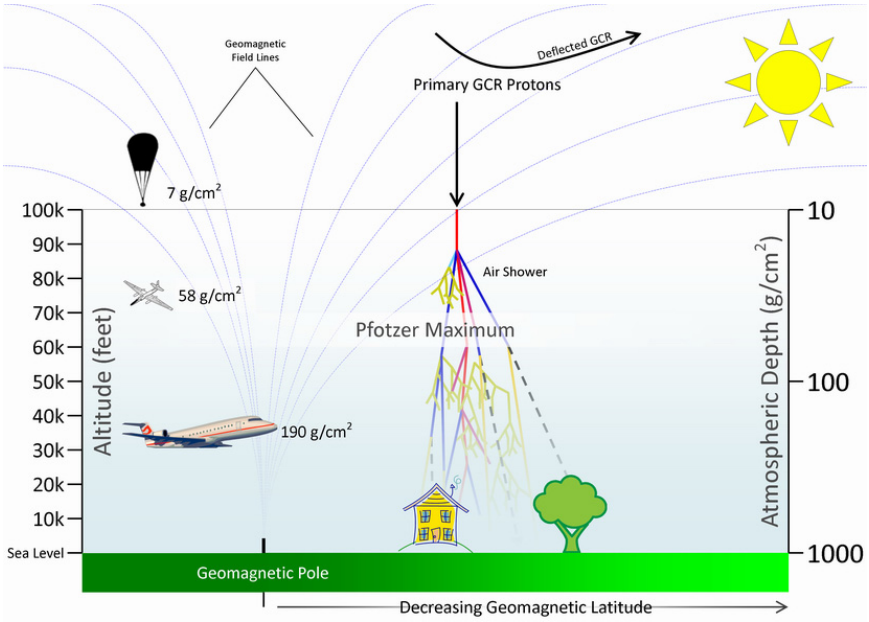


Fig. 2.1: Source of ionizing radiation.

According to the operating environment, there are different types of particles and different fluxes that may affect the integrated circuit:

- **Space**: heavy ions, protons, and electrons
- **Flight altitude**: mainly neutrons
- **Sea level**: neutrons.

Each kind of particle may produce different effects on integrated circuits. Heavy ions produce direct ionization, while protons interact with matter producing nuclear reactions and secondary ionizations. Neutrons also interact with material producing secondary particles such as alpha particles and nuclear reactions.

The transient faults are originated from the interaction of such energized or secondary particles with the silicon at the PN junctions of an off-state CMOS transistors. The electron-hole pair track formed by this interaction may charge or discharge that struck node producing a transient pulse. The phenomena is known as *Single Event Effect* (SEE). SEE can be potentially destructive, known as *hard errors*, but generally they are non-destructive, known as *soft errors* (O'BRYAN, 2015). This work focuses on soft errors.

2.2 Non-destructive Single Event Effects

According to the European Space Agency (ESA) (STURESSON, 2009), the main type of non-destructive SEEs are:

- **Single Event Upsets (SEU)**: an SEU is characterized when a transient fault affects a memory element, such as a memory cell or a register, changing the state of the element
- **Single Event Functional Interrupts (SEFI)**: it is an event that leads to temporal loss of device functionality. SEFIs are often induced from SEU in control registers. The system is recovered by reset or power cycle
- **Single Event Transients (SET)**: when the transient fault affects a gate of the combinational logic, creating a glitch, it is called *Single Event Transient*. SETs are becoming a major concern because the frequency is increasing (FERLET-CAVROIS, 2005).

Fig. 2.2 shows an example of an SEU and an SET. A particle hits a memory element, causing an SEU, changing the stored value from 0 to 1. The change of the memory element value is known as *bit flip*. Fortunately, the SEU in the example is masked by the NAND gate that follows the fault because the other NAND input is 0. If it were 1, then the output of the first NAND would be affected by the fault, and the error would propagate. Another particle hits an NOR gate causing an SET, temporarily changing the expected output from 1 to 0. In the example, the fault is not masked by any gate. Therefore, it propagates until a memory element. If the pulse hits a memory element during a clock event, a wrong value is stored in the memory element.

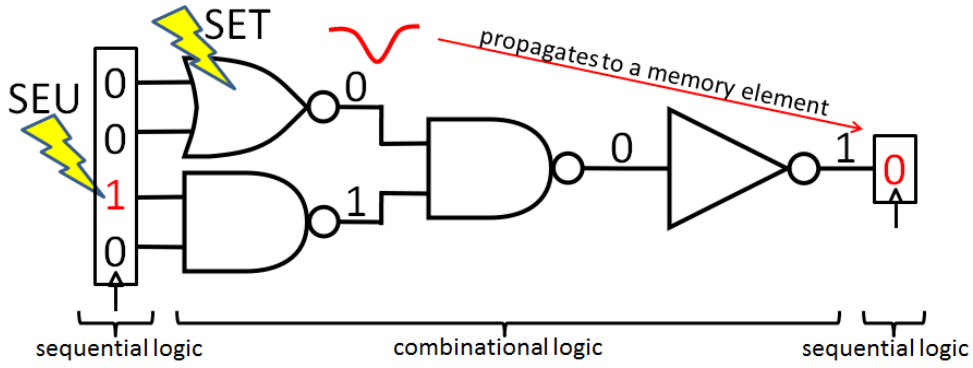


Fig. 2.2: SEU and SET in circuits.

2.3 Fault, error, and failure

Following the definitions presented by (AVIZIENIS, 2004), the concepts of fault, error, and failure can be understood as:

- **Fault**: it is the logical effect of the particle hit. The fault effect is a bit flip in a memory element. Faults can lead to errors, or they can be masked by latch window, logical, or electrical properties.
- **Error**: error is an active fault. It may propagate to the system output and causes a system failure.
- **Failure**: a failure is defined as a system malfunction. It occurs when the system produces an incorrect output.

It is important to notice that not all faults cause errors, and not all errors lead to failures.

2.4 Soft errors in processors

Soft errors affect processors by modifying values stored in memory elements (such as registers or data memory). They may lead the processor to incorrectly execute an application, producing a wrong output or even entering into a loop and never finishing the execution. They can affect the control-flow and the data-flow of a running application, as Fig. 2.3 shows.

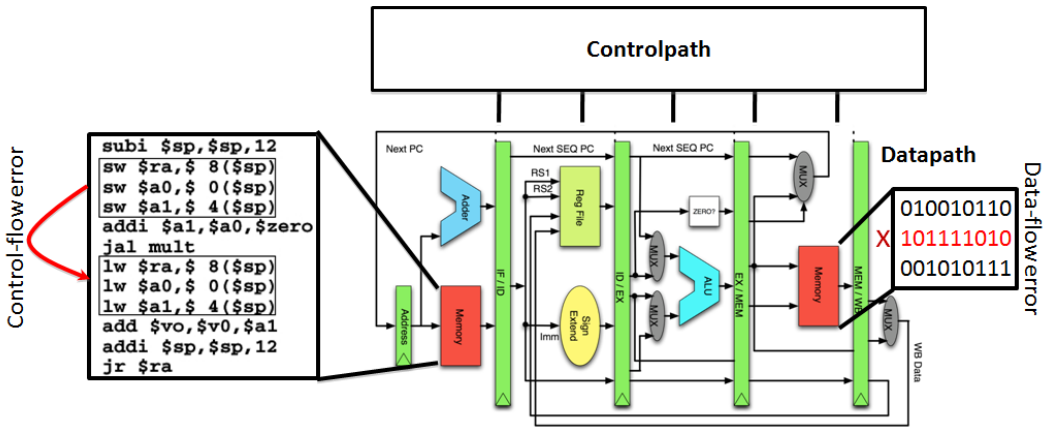


Fig. 2.3: Effect of soft errors in processors.

Data-flow error refers to soft errors caused by bit flips in storage devices, such as registers or memories. They affect the program output, but not its execution. When a fault affects the data-flow, the application runs normally, but the result, in the end, is incorrect. The data-flow errors are normally caused by:

- **Wrong operation**: the bit flip modifies the instruction, and it performs another operation, which affects a memory element, such as a register or memory cell
- **Incorrect data**: the bit flip affects directly a memory element that contains the data used by an operation. Since the operation input is wrong, it is likely that its output will also be wrong. The error may propagate to the program output.

A control-flow error occurs when the program flow is incorrectly followed, i.e., the error changes the program execution. When a fault affects the control-flow, an erroneous execution flow occurs. The possible outcomes caused by the fault are:

- **Branch creation**: a bit flip converts a non-branch instruction into a branch, and then this illegal branch changes the program flow to a wrong address
- **Branch deletion**: a branch instruction is converted into another instruction. Thus, a branch is not taken when it should be
- **Incorrect branch decision**: it happens when a branch that should go in one direction, based on a comparison, goes in the other direction, i.e., a branch is not taken when it should be, or when it is taken when it should not be
- **Incorrect target address**: the bit flip modifies the register that contains the target address of a branch instruction (for example, the one used to return from a subroutine). It will change the program execution to an incorrect address
- **Bit flip in the PC register**: it changes the next instruction to be executed. It has the same effect as branch creation.

2.5 Fault tolerance techniques

The use of fault tolerance techniques can provide reliability for processors against soft errors and significantly reduces the chances of an application being incorrectly executed. The techniques can detect, mask or correct errors. The ones detecting errors are less costly than the ones masking or correcting because less redundancy is added. Therefore, this work focuses on detection techniques. Once an error is detected, a restarting or rollback process can be performed. With regards to processors, there are two types of fault tolerance techniques: hardware-based techniques, which rely on replicating or adding hardware modules, and software-based techniques, which rely on the replication of information and instructions in the program code (UNSAL, 2002).

2.5.1 Hardware-based fault tolerance techniques

Hardware-based techniques are fault tolerant schemes based on hardware redundancy (PRADHAN, 1996). The redundancy can be applied at many different logic granularities. The Triple Modular Redundancy (TMR) is a well-known hardware-based fault tolerance technique. It triplicates the hardware and masks the error by voting the results in order to get the correct values. In some cases, the datapath and registers can be triplicated, which implies modifying the original processor design (PILOTTO, 2008).

The processors specially designed with hardware-based techniques in the internal architecture are used called RadHard processors. Examples of RadHard processors are Cobham LEON 3FT/4FT, Space Micro Proton200k SBC, etc. However, they are very expensive and are not high-end architectures, usually fabricated in old technologies. Another limitation for using RadHard processors is the restrictions on the commerce of RadHard devices, due to international trade regulations, such as ITAR (International Traffic in Arms Regulations), which makes impossible to many countries to buy many of these devices (CASTRO NETO, 2010).

Other techniques use the processor as a black box by adding redundancy or extra hardware just outside the processor. One example is when the entire processor is triplicated, and just the outputs or memory values are voted, as, for instance, the Maxwell SCS750 (MAXWELL TECHNOLOGIES, 2015), Atmel SPARC V7 ERC32 and TSC695FL (GINOSAR, 2012). For only detecting an error, the Duplication With Comparison (DWC) technique (WAKERLY, 1978) can be used. After masking or detecting an error, the processor must restart or to recompute from a safe state step.

Fig. 2.4 shows an example of a processor protected by a hardware-based technique. In the example, a copy processor, similar to the original, is utilized. The copy processor executes the same application of the original processor. A checker compares if the outputs of both processors match. If they do, the output is correct; otherwise, an error is reported.

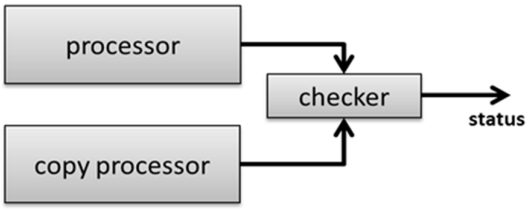


Fig. 2.4: Example of DWC hardware-based technique using a black box processors.

Many modern multi-core processors have the lockstep capability. It means that different cores run the same program, which allows error detection (if dual modular redundancy) or correction (if triple modular redundancy). Examples of the processors implementing the lockstep capability are the ARM Cortex-R processors, which implement a dual modular redundancy. Detection is done in a short time, usually few clock cycles, and the process rollback to a previous correct saved state. The rollback requires additional memory for storage the previously executed commands. Furthermore, the time consumed on saving the current processor's state for a future rollback is an important drawback (BELCASTRO, 2006).

The hardware techniques can also be based on hardware monitoring devices, called *watchdogs* (MAHMOOD, 1988), to monitor specific information. Such devices monitor the control-flow and memory accesses of applications running on the target processor, such as (AZAMBUJA, 2011b), or the information provided by the processor through the debug port, for example, as in (DU, 2015). On the downside, the information that watchdogs can access may be limited. Some registers, buses, cache memories may not be accessible. That limits the use of watchdogs depending on the architecture of the target processor.

Although the high reliability of hardware-based techniques, they introduce significant overheads, like an increase in area and power consumption. It can be critical for space

applications since they are supplied by batteries (REIS, 2005b). Furthermore, hardware-based techniques present high design and manufacture costs (ASENSI, 2011).

2.5.2 Software-based fault tolerance techniques

Software-based techniques, also referred in the literature as SIHFT (*Software-Implemented Hardware Fault Tolerance*) techniques (GOLOUBEVA, 2006), are an approach to protect processor-based systems against soft errors by modifying the program code, without having to modify the underlying hardware. They rely on adding instruction redundancy and comparison to detect errors, as exemplified in Fig 2.5. These techniques provide high flexibility and low development time and cost. In addition, they allow the use of commercial off-the-shelf (COTS) processors, since no modification of the hardware is necessary. That makes possible to use new generations of processors with no available equivalent *RadHard*.

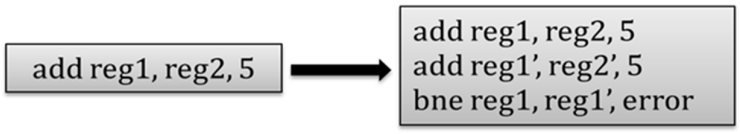


Fig. 2.5: Example of a SIHFT technique.

As stated, there are two types of soft errors that affect processors: errors in the data-flow and errors in the control-flow. In consequence, the software-based techniques present in the literature can be divided in two groups: control-flow techniques (OH, 2002b), (MCFEARIN, 1995), (ALKHALIFA, 1999) and data-flow techniques (AZAMBUJA, 2011b), (OH, 2002a). The first group aims to detect faults affecting the data. In order to do so, such techniques duplicate registers used by the application. By duplicating the registers, it is possible to compare them by adding checking instructions. It is important to notice that every operation performed in a register must also be performed in its copy to keep consistency. The second group aims to detect illegal branches in the program execution by assigning a unique signature for each block of instructions. Then, these techniques assign the block signature to one register available during the execution. Checking instructions are inserted in the code to compare the signature register with the expected signature for that block. By doing so, it is possible to detect incorrect branches in the program execution. Finally, it is important to mention that there are techniques that aim at protecting against both data-flow and control-flow errors. These techniques combine characteristics of both data-flow and control-flow techniques with some optimizations (REBAUDENGO, 1999), (CHEYNET, 2000), (REIS, 2005b). Anyhow, they can be seen as a data-flow and a control-flow technique applied together. Details about data-flow and control-flow techniques are presented as follows.

2.5.2.1 Data-flow techniques

Data-flow techniques aim to detect faults affecting the data, i.e., the values stored in registers and the memory. In order to do so, such techniques duplicate, when detecting, and triplicate, when correcting, the registers used by the application. By duplicating registers, it is possible to detect data-flow errors by comparing a register with its replica. It is important to notice that every operation performed in a register must also be performed in its replica to keep the program consistent. Fig. 2.6 shows an example of a code hardened by a data-flow technique. On the left side, one can see the original code composed of five instructions, lines 1, 4, 9, 13, and 17. And the right side shows the same code hardened. Registers \$12, \$13, \$14, and \$15 are replicas of registers \$2, \$3, \$4 and

$5, respectively. The duplicated instructions are presented at lines 2, 5, 10, and 14 (formatted as italic). In this technique, checkers are inserted before stores and branches, checking the source registers, and after any other instruction, checking the destination register. Checkers are inserted at lines 3, 6, 7, 8, 11, 12, 15, and 16 (formatted as bold).

original code	**hardened code**
1: lw $2,0($4)	1: lw $2,0($4)
	2: *lw $12,offset($14)*
	3: **bne $2,$12,error**
4: sll $4,$2,1	4: sll $4,$2,1
	5: *sll $14,$2,1*
	6: **bne $4,$12,error**
	7: **bne $2,$12,error**
	8: **bne $3,$13,error**
9: sw $2,0($3)	9: sw $2,0($3)
	10: *sw $12, offset ($13)*
	11: **bne $4,$14,error**
	12: **bne $2,$12,error**
13: sw $4,0($2)	13: sw $4,0($2)
	14: *sw $14, offset ($12)*
	15: **bne $4,$14,error**
	16: **bne $5,$15,error**
17: ble $4,$5,$L2	17: ble $4,$5,$L2

Fig. 2.6: Example of a data-flow technique.

It is possible to notice by the example that data-flow techniques introduce significant overheads. The overheads caused by data-flow techniques are higher than the ones caused by control-flow due to the duplication of data and instructions, and the insertions of checkers. Control-flow techniques only insert instructions to modify and check the value of signatures. If the application needs fault tolerance but has performance or energy constraints, data-flow techniques might not be applied. New data-flow techniques with reduced overheads are desirable in such scenarios.

2.5.2.2 Control-flow Techniques

Control-flow techniques aim to detect incorrect branches during the program execution. The code is divided into *basic blocks* (BBs), which are branch-free sequences of instructions with no branches into the basic block, except to the first instruction, and no branches out of the basic block, except possibly for the last instruction. Fig. 2.7(b) shows the basic blocks and the program flow of the code presented in Fig 2.7(a). Calls to subroutines (*jal*), branches, and jumps are ends of basic blocks. Consequently, the following instruction is a beginning of another basic block. Labels indicate the beginning of basic blocks. Thus, the instruction before a label is the end of the previous basic block.

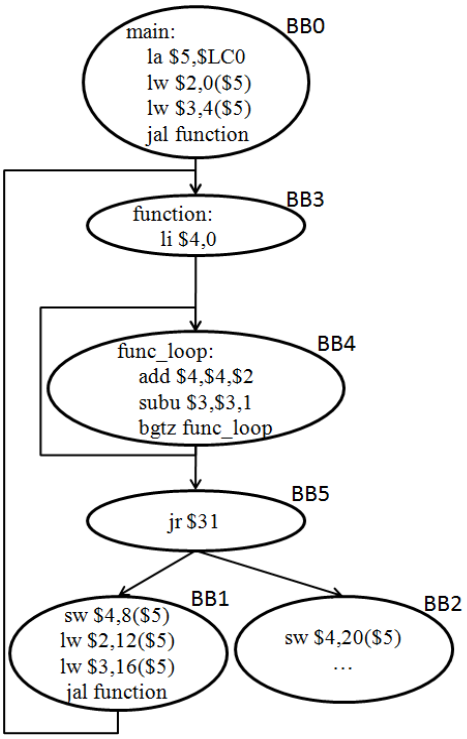
BB0	main:
	la $5,$LC0
	lw $2,0($5)
	lw $3,4($5)
	jal function #$31<-C+4

BB1	sw $4,8($5)
	lw $2,12($5)
	lw $3,16($5)
	jal function

BB2	sw $4,20($5) …
BB3	function: li $4,0

BB4	func_loop:
	add $4,$4,$2
	subu $3,$3,1
	bgtz func_loop

BB5	jr $31 #PC <- $31



(a) (b)

Fig. 2.7: Basic Blocks and program flow.

These techniques usually assign a unique signature for each basic block and, sometimes, another protection to the program flow. The signature is assigned to an available register at the beginning of the basic block, and it is usually checked at the beginning or end of the basic block. By doing so, they are able to detect illegal branches in the program execution.

The following chapter presents, in details, some of the state-of-the-art SIHFT techniques and selective hardening methods. They are important for a deep understanding SIHFT techniques and the contributions of this work.

3 RELATED WORK

Software-based fault tolerance techniques, also referred in the literature as *Software-Implemented Hardware Fault Tolerance* (SIHFT) techniques (GOLOUBEVA, 2006), are techniques implemented in software to protect processor against soft errors that may affect the program flow or the data stored in registers or memory. The techniques that aim to protect the data are called *data-flow techniques*, and the ones to protect the control-flow are the *control-flow techniques*. There are also techniques that combine features of both data-flow and control-flow techniques. They consist of code transformation rules, and can be understood as a data-flow and control-flow technique applied together.

Although software techniques bring reliability to processors, they cause performance and memory overheads, and, consequently, increase the energy consumption. To reduce those overheads, selective methods to protect processors, known as selective hardening, can be implemented. Their goal is to reduce the overheads with minimum impact in the protection. This chapter presents the state-of-the-art of data-flow techniques, control-flow techniques, techniques that combine features of data-flow and control-flow techniques, and methods for selective hardening.

3.1 Data-flow techniques

Data-flow techniques are designed to protect the data stored in registers or memory. These techniques replicate the variables, assigning copies to the original ones. When the aim is error detection, variables are duplicated, and when correction is desirable, variables are triplicated. Checkers are inserted in the code to compare variables with their copies. The points where checkers are inserted depends on the technique. Since error detection presents lower overheads than correction, due to duplication instead of triplication, this work focuses on that. Some data-flow techniques present in the literature are discussed below.

3.1.1 EDDI

EDDI (*Error Detection by Duplicated Instructions*) is a well-known data-flow technique proposed by (OH, 2002a). It duplicates all the information, i.e., all registers are duplicated, and all operations on the registers are also performed on the registers replicas. To ensure that the data is correct, instructions are inserted to compare the original register with its replica. If the values differ, an error is detected. The points where checkers are inserted by EDDI are:

- **Before storing a register value in memory**: stores are the connection between the processed information and the memory. If an error occurs during

the computation, it will probably propagate to the output. Therefore, stores are a good point to check the consistency of the information

- **Before branch instructions**: a misdirected branch can make the program execution skip stores, or execute incorrect stores. Besides, exiting a loop before or after its supposed exiting point can make the output incorrect.

Fig. 3.1 shows how EDDI is applied. As one can see, the original instructions are presented at lines 1, 3, and 8 (formatted as normal text). Their replicas are at lines 2, 4, and 9, respectively (formatted as italic). Instructions accessing the memory present an offset to duplicate the values in memory. The replicas of the registers r11, r12, and r13 are the registers r21, r22, and r23, respectively. Instructions to compare the registers used by the store at line 8 are inserted at lines 5 and 6. If an error is detected, the branch at line 7 is taken. The checking instructions are formatted as bold.

original code	**EDDI code**
1: ld r12=[GLOBAL]	1: ld r12=[GLOBAL] 2: *ld r22=[GLOBAL+offset]*
3: add r11=r12, r13	3: add r11=r12,r13 4: *add r21=r22,r23*
	5: **cmp.neq.unc p1,p0=r11,r21** 6: **cmp.neq.or p1,p0=r12,r22** 7: **(p1) br faultDetected**
8: st m[r11]=r12	8: st m[r11]=r12 9: *st m[r21+offset]=r22*

Fig. 3.1: EDDI technique (REIS, 2005b).

3.1.2 Variables 1

Variables 1 (VAR1) is a data-flow technique proposed by (AZAMBUJA, 2011a). It is based on (CHEYNET, 2000) rules that aim to protect the data. These rules are implemented in a high-level language. On the other hand, VAR1 is implemented in the assembly code. Therefore, the variables are replaced by registers. VAR1 proposes rules aiming at detecting faults affecting the data. Such rules are presented below; they comprise of instruction replication and insertion of checkers to detect incorrect values.

- **Rule 1**: every variable used in the program must be duplicated
- **Rule 2**: every write operation performed on a variable must be performed on its replica
- **Rule 3**: before each read on a variable, its value and the value of its replica must be checked for consistency.

Fig. 3.2 shows an example of how VAR1 rules are applied. The original code is presented on the left side, and the protected one is on the right side. The duplications are showed at lines 3, 7, and 11 (formatted as italic), and checkers at lines 1, 4, 5, 8, and 9 (formatted as bold). The checker at line 1 checks the base address register of the *load* operation with its replica. Checkers at lines 4 and 5 are related to the *add* instruction at line 6. Finally, the checkers at lines 8 and 9 check the registers used by the *store* operation at line 10.

original code	VAR1 code
2: ld r1, [r4]	1: **bne, r4, r4', error** 2: ld r1, [r4] 3: *ld r1', [r4' + offset]*
6: add r1, r2, r4	4: **bne r2, r2', error** 5: **bne r4, r4', error** 6: add r1, r2, r4 7: *add r1', r2', r4'*
10: st [r1], r2	8: **bne r1, r1', error** 9: **bne r2, r2', error** 10: st [r1], r2 11: *st [r1' + offset], r2'*

Fig. 3.2: VAR1 technique (AZAMBUJA, 2011a).

3.1.3 Variables 2

Variables 2 (VAR2) is an alternative data-flow technique, proposed by (AZAMBUJA, 2011a), that aims at reducing the overheads imposed by VAR1. The technique uses VAR1 rules 1 and 2, but it changes the way checkers are inserted in order to reduce the number of extra instructions. Based on that, it is expected to reduce the execution time overhead. Instead of checking the variables before read them, VAR2 checks the variables after writing a new information on them. VAR2 is also implemented in assembly level. Thus, the references to variables should be understood as references to registers. In instructions where there no new value is assigned to any variable (stores, for example), it is implemented the same checking approach of VAR1, i.e., the variables are checked before they are read. The rules are stated as:

- **Rule 1**: every variable used in the program must be duplicated
- **Rule 2**: every write operation performed on a variable must be performed on its replica
- **Rule 3**: after each write on a variable, its value and the value of its replica must be checked for consistency. If no value is assigned to a variable in the instruction, the checking is performed before reading the variable.

Fig. 3.3 shows how VAR2 is implemented. Duplications are inserted at lines 2, 5, and 10 (formatted as italic). Checkers are presented at lines 3, 6, 7, and 8 (formatted as bold). The checkers, at lines 3 and 6, check after a write operation on r1, and the ones at lines 7 and 8 perform checkings before reading the registers in the store instruction (line 9).

original code	VAR2 code
1: ld r1, [r4]	1: ld r1, [r4] 2: *ld r1', [r4' + offset]* 3: **bne, r1, r1', error**
4: add r1, r2, r4	4: add r1, r2, r4 5: *add r1', r2', r4'* 6: **bne r1, r1', error**
9: st [r1], r2	7: **bne r1, r1', error** 8: **bne r2, r2', error** 9: st [r1], r2 10: *st [r1' + offset], r2'*

Fig. 3.3: VAR2 technique (AZAMBUJA, 2011a).

3.1.4 Variables 3

Variables 3 (VAR3) is a technique proposed by (AZAMBUJA, 2011a) that aims to reduce the overheads of VAR1. It considers that inconsistency between the duplicated variable only needs to be checked before reading or writing data to memory, and before branches. The rules for VAR3 technique are:

- **Rule 1**: every variable used in the program must be duplicated
- **Rule 2**: every write operation performed on a variable must be performed on its replica
- **Rule 3**: before each read on a variable by *loads*, *stores* or *branches*, its value and the value of its replica must be checked for consistency.

Fig. 3.4 presents an example of VAR3. The original code is shown on the left side, and the one protected by VAR3 is presented on the right side. Firstly, the instructions are duplicated. The duplications can be seen at lines 3, 5, and 9 (formatted as italic). The way VAR3 duplicates the code is the same of VAR1. Checkings are also performed before reads on variables, but not in all operations. Checkers are not inserted for arithmetic instructions. They are inserted at lines 1, 6, and 7, and are related to a *load* and a *store* instruction. VAR3 presents significant overhead reduction when compared to VAR1 or VAR2.

original code	VAR3 code
2: ld r1, [r4]	1: **bne, r4, r4', error** 2: ld r1, [r4] 3: *ld r1', [r4' + offset]*
4: add r1, r2, r4	4: add r1, r2, r4 5: *add r1', r2', r4'*
8: st [r1], r2	6: **bne r1, r1', error** 7: **bne r2, r2', error** 8: st [r1], r2 9: *st [r1' + offset], r2'*

Fig. 3.4: VAR3 technique (AZAMBUJA, 2011a).

3.1.5 Drawbacks of data-flow techniques

The data-flow techniques duplicate the data and instructions in the code. Replicas are assigned to the registers used by the application. The main difference among these techniques consists of the location where checkers are inserted. Table 3.1 shows the average execution time, code size, and data error detection rate for tests with the state-of-the-art data-flow techniques using a set of applications[2] running on the miniMIPS processor[3] (HANGOUT, 2009). As one can see, the execution time varies from 1.63x (EDDI) to 2.54x (VAR2), and the code size ranges from 1.73x (EDDI) to 2.48x (VAR2). The percentage of errors affecting the data-flow detected goes from 91.7% (EDDI) to 96.3% (VAR2). One can notice that the current data-flow techniques detect most of the errors affecting the data-flow. However, the overheads they imply to the application are high. New approaches to reduce such overheads are necessary.

Table 3.1: State-of-the-art data-flow techniques

technique	execution time	code size	data error detection
EDDI	1.63x	1.73x	91.7%
VAR1	2.42x	2.35x	96.1%
VAR2	2.54x	2.48x	96.3%
VAR3	1.83x	1.90x	95.3%

3.2 Control-flow techniques

Control-flow techniques are designed to protect the program flow, i.e., to protect against incorrect branches. Such techniques divide the code into basic blocks. A *basic block* (BB) is a branch-free sequence of instructions, i.e., a portion of code that is always executed in sequence. There only can be a branch instruction at the end of the basic block. Furthermore, there are no branches to the basic block, except, possibly, to the first instruction. For each basic block, a signature is assigned. The signature is attributed to a global register at the beginning of the basic block. Checkers are inserted into the code to verify if the signature register contains the expected value. If it does not, it means there was an incorrect branch and an error is reported. The main control-flow techniques present in the literature are described below.

3.2.1 CCA

McFearin (1995) proposed a control-flow technique called *Control-Flow Checking Using Assertions* (CCA). CCA divides the code into basic blocks, referred by them as *Branch Free Interval* (BFI), and assigns two identifiers for each basic block, the *Branch Free Interval Identifier* (BID) and the *Control Flow Identifier* (CFID). CCA requires three registers to be implemented, one for BID and two for CFID. BID represents the BB, and it has a unique value for each BB. The BB's BID is attributed to the BID register at the beginning of the basic block and checked at the end. CFID is used to check if the BBs are executed in the correct order. It indicates the possible next BBs. CFID works with two

[2] The set of applications consists of bubble sort, Dijkstra's algorithm, matrix multiplication, run-length encoding, summation, TETRA encryption algorithm. Details about them in the appendix C.

[3] More details about the miniMIPS processor in the appendix B.1.

queue elements, and that is why it requires two registers. At the beginning of the program execution, the queue contains only the CFID of the first BB. When it enters in a BB, the next BB's CFID is enqueued. And before exiting the basic block, the queue is dequeued, and the dequeued element is checked with the expected CFID. Fig. 3.5 summarizes the explained above. It is important to notice that BBs B and C share a common parent; therefore, they have the same CFID. A control-flow error is detected for any of the following cases:

- **The BID register differs from the BB's BID**: an illegal branch has occurred from the previous BB to the middle of the current one. It will be detected at the end of the current basic block
- **Trying to enqueue when the queue is full**: it happens due to an illegal branch from the middle of the previous BB to the beginning of the current one
- **Trying to dequeue an empty queue**: an illegal branch from the end of the previous BB to the middle of the current BB
- **The dequeued CFID differs from the expected one**: it means there was a branch to another BB, different from the expected ones.

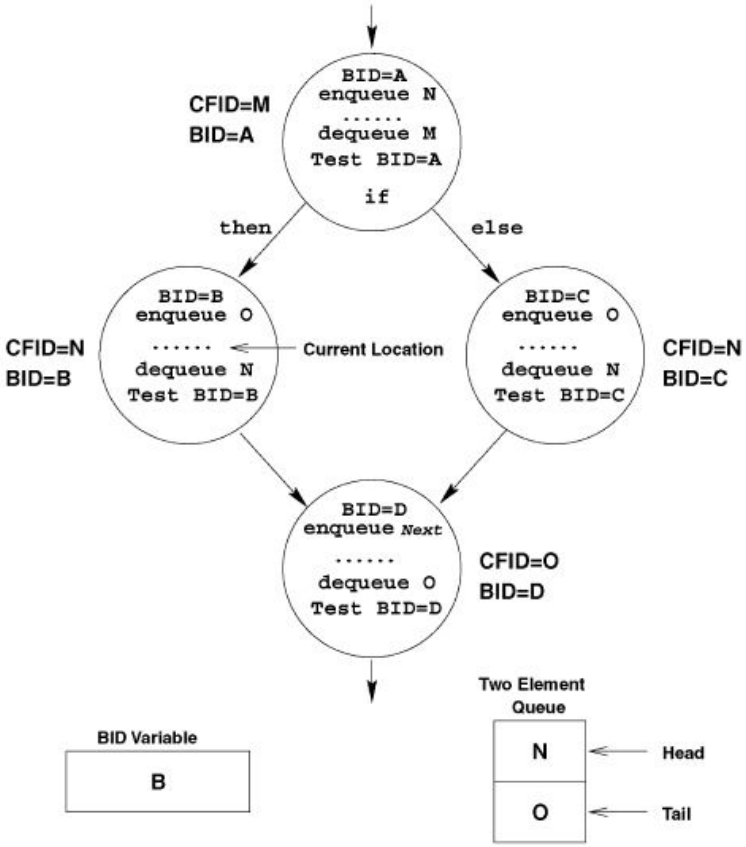


Fig. 3.5: Example of CCA technique (ALKHALIFA, 1999).

CCA can detect most control-flow errors, but it cannot detect errors affecting the branches used as checkers by the technique. Furthermore, when two parents have a common child, all the children have the same CFID. For example, if parent A has C and D as children, and parent B has D and E as children, all C, D, and E, have the same CFID.

So if an illegal branch occurs, from A to E or from B to C, it will not be detected. Besides, CCA presents high overheads. Instructions to enqueue, dequeue, check the queue integrity, check the CFID, assign and check the BID, are quite costing. Actually, it could also increase the probability of error due to the increased program size (ALKHALIFA, 1999).

3.2.2 ECCA

Enhanced Control-flow Checking Using Assertions (ECCA) is the successor of CCA (ALKHALIFA, 1999). It divides the code into blocks that are collections of consecutive basic blocks with a single entry and exit. A unique prime number is assigned for each block as the *block identifier* (BID). At the beginning of the block, a global variable *id* is updated using the following equation:

$$id \leftarrow \frac{BID}{\overline{(id \bmod BID)} \cdot (id \bmod 2)} \quad \text{(Eq. 3.1)}$$

And, at the end of the block, the variable is updated using the equation below. The *NEXT* component of the equation is the product of all possible subsequent blocks.

$$id \leftarrow NEXT + \overline{\overline{(id - BID)}} \quad \text{(Eq. 3.2)}$$

ECCA combines the CCA's BID and CFID in one signature. Although ECCA seems to present lower overheads, the equations to update *id* are quite complex, and they are implemented using several instructions at the assembly level.

3.2.3 CFCSS

Oh (2002b) proposed *Control-flow Checking by Software Signatures* (CFCSS). It is a control-flow technique that updates signature register *G* at runtime. Firstly, the program is divided into basic blocks, and for each basic block, a random signature is assigned. When the execution pass from one BB (let us call BB1) to another BB (let us call BB2), a new *G* is generated based on an XOR function *f* that uses the signatures of the BB1 and BB2, as shown in Eq. 3.3:

$$f \equiv f(G, d_d) = G \oplus d_d \quad \text{(Eq. 3.3)}$$

where *G* is the signature register that, at first, contains the BB1's signature, and d_d is a value to update *G* to the new signature (the BB2's signature). The d_d, defined in Eq. 3.4, is given by the XOR operation between the signatures of BB1 (s_s) and BB2 (s_d):

$$d_d = s_s \oplus s_d \quad \text{(Eq. 3.4)}$$

If a branch is correctly taken, *G* is updated to the correct signature. In Fig. 3.6, during the execution of BB1, *G* is equal to signature s_1. It is updated to s_2 when entering BB2 by using Eq. 3.3. If an illegal branch from BB1 to BB4 occurs, *G* is updated to an incorrect value. To detect errors, branches to check *G* with the expected signature are placed at the beginning of the basic block, right after updating *G*.

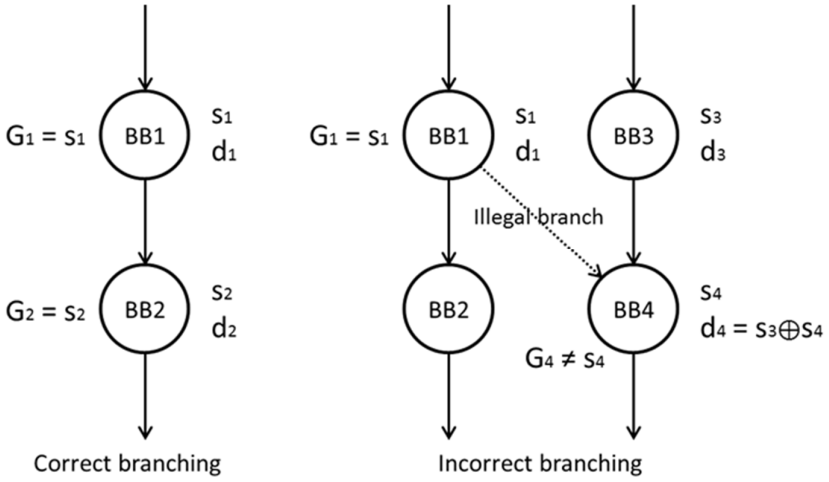


Fig. 3.6: Signature update during correct and illegal branch.

The presented method can detect most control-flow errors. However, when a basic block has more than one predecessor, the predecessors' signatures must be the same so *G* can be correctly updated to the new signature in any case. If this method is used, the same problem that occurs with CCA can happen with CFCSS, i.e., if BB1 and BB2 share a common successor (BB4), and each one has an independent successor (BB3 and BB5, respectively), as shown in Fig 3.7, an illegal branch from BB1 to BB5, or from BB2 to BB3, cannot be detected.

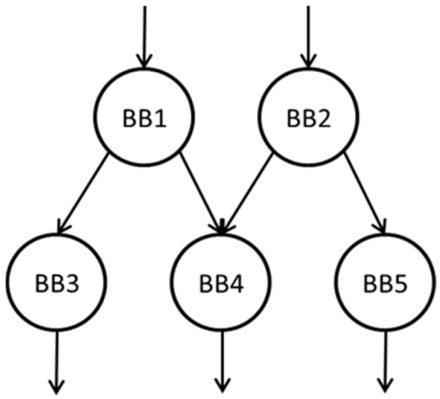


Fig. 3.7: Basic blocks sharing a common successor.

To avoid this problem, a *runtime adjusting signature D* is used. After *G* is transformed by Eq. 3.3 at the beginning of BB4, *G* is also updated by an XOR operation between the just transformed *G* and *D*. The value of *D* is defined in the predecessors BBs. Fig. 3.8 illustrates how *D* is used. The d_4 is calculated using Eq. 3.4, where s_s is randomly picked among the predecessors. In the example, $d_4 = s_1 \oplus s_4$. For this reason, *D* is set to zero in BB1 because *G* has the correct signature after the first update. On the other hand, *D*, in BB2, needs to be set to $s_1 \oplus s_2$. Thus, *G* has the correct value after the two updates: $G = (G \oplus d_4) \oplus D = (s_2 \oplus s_1 \oplus s_4) \oplus (s_1 \oplus s_2) = s_4$. The not shared successors do not need to use *D*. In this case, *G* can be updated using only Eq. 3.3.

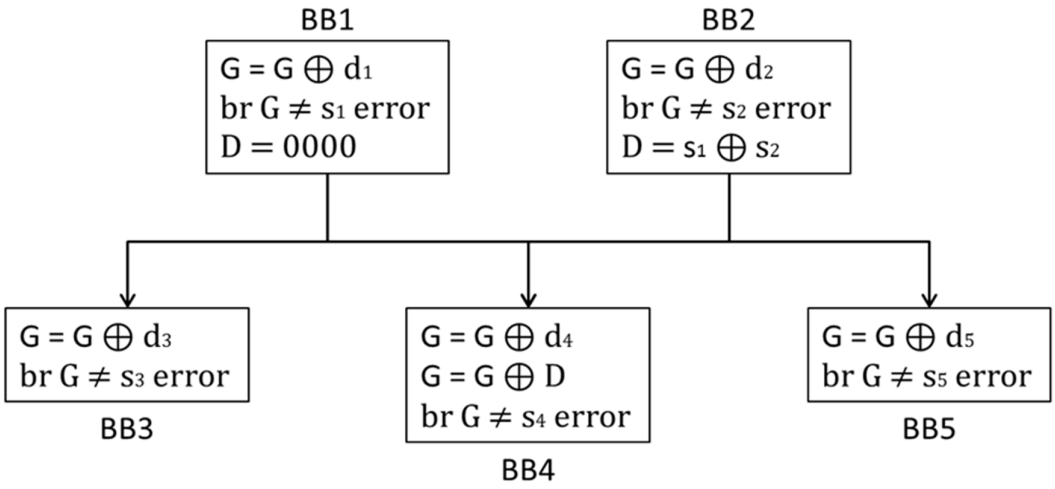


Fig. 3.8: Update of *D* when BBs share a successor.

Although that the introduction of the *runtime adjusting signature* solve the problem cited above, the same problem can still happen if multiple basic blocks share multiple successors, as shown in Fig. 3.9. BB5 has three predecessors, BB1, BB2, and BB3, and BB6 has two predecessors, BB2 and BB3. Since there is only one runtime adjusting signature *D*, both BB5 and BB6 have to use the same value for *D*. Thus, if *D* is calculated for BB5, BB6 has to use the same *D* to update *G*. Thus, if an illegal branch occurs from BB1 to BB6, it will not be detected.

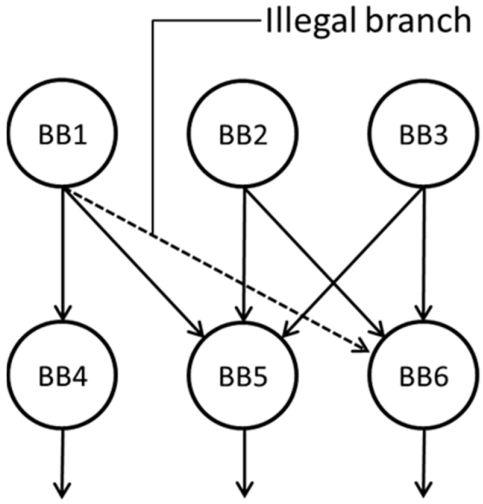


Fig. 3.9: Undetected illegal branch.

3.2.4 YACCA

Goloubeva (2003) introduced a new control-flow technique called YACCA, acronym for *Yet Another Control-Flow Checking using Assertions*. The code is divided into basic blocks, and a unique signature is assigned for each basic block. Like ECCA, a global variable is updated at the beginning and end of the basic blocks. At the beginning, the global variable is used to control if the transition from previous basic block to the current one is valid. And at the end, the global variable is updated to the BB's signature. However,

unlike ECCA the basic block transitions are not controlled by checking if the current BB signature is multiple of the previous ones. Actually, it checks if the global variable contains the signature of a predecessor BB. In order to avoid adding multiple checkers for each basic block since a basic block can have many predecessors, a global variable *ERR_CODE* is defined as follows:

(Eq. 3.5) $$ERR\_CODE = (code != BB_1) \&\& (code != BB_2) \&\& (...) \&\& (code != BB_N)$$

where BB_1, BB_2, ..., BB_N are the signatures of the previous basic blocks, and *code* is the global variable that contains the current signature. If *ERR_CODE* is 1, an incorrect transition has occurred.

The equation to update the global variable *code* is presented below (Eq. 3.6). *M1* represents a constant defined only by the signatures of the previous basic blocks. And *M2* is a constant based on the previous BBs' signatures further the current BB signature. This method avoids the aliasing effect presented by CFCSS when multiple BBs share multiple previous BBs. Furthermore, the runtime adjusting signature is not necessary.

(Eq. 3.6) $$code = (code \,\&\, M1) \; \square \; M2$$

In addition, another rule proposed by Rebaudengo (1999) integrated to the technique. It aims at detecting incorrect decisions of conditional branches. For this, the branch instruction is repeated at the beginning of both target basic blocks to detect if the same test produces different results, which means that an error has occurred. This method is better discussed in section 2.3.1 with the original technique.

YACCA is capable of detecting most of the faults affecting the control-flow. Furthermore, it avoids the aliasing presented by CFCSS when multiple basic blocks share multiple predecessors. However, the technique present a high overhead in performance, mainly due to the method to detect erroneous transitions between basic blocks. It requires as many instructions as the number of predecessors the basic block has to determine *ERR_CODE,* besides another instruction to check the *ERR_CODE* value.

3.2.5 CEDA

Control flow Error Detection through Assertions (CEDA) is an efficient control-flow technique (VEMU, 2011). It divides the code into basic blocks. For each BB, two signatures are assigned. One is the basic block's signature, called *Node Signature* (NS), and the other is a transition signature, called *Node Exit Signature* (NES). Only one signature register *S* is necessary for implementing CEDA. At the beginning of the basic block, *S* is updated to the BB's NS and at the end, before exiting the BB, *S* is updated to the BB's NES. The basic blocks are classified in two types: A and X, which are used to define the operation to update *S* to the BB's NS. A basic block is of type A if it has multiple predecessors and at least one of its predecessors has multiple successors. Otherwise, the basic block is of type X. The possible equations to update *S* when entering in a BB are:

(Eq. 3.7) $$S = S \; AND \; d1(BB_i), \text{ if } BB_i \text{ is of type A}$$

(Eq. 3.8) $$S = S \; XOR \; d1(BB_i), \text{ if } BB_i \text{ is of type X}$$

where BB_i is the current basic block, and *d1* is a constant to update *S* to the current NS. The equation to update *S* to NES, before exiting the BB, is presented bellow. The constant *d2* is used for this.

(Eq. 3.9) $$S = S \; XOR \; d2(BB_i)$$

Furthermore, the same rule proposed by Rebaudengo (1999), and used by YACCA, is implemented in CEDA. It aims at detecting incorrect decision of conditional branches. The branches are retested at both true and false destinations. However, CEDA does not implement it to the basic blocks that have multiple predecessors.

There is a new concept introduced in CEDA, the networks. A *network* is a set of basic blocks that share a common predecessor. Each basic block belongs to one and only one network. And the *network predecessors* are the predecessor basic blocks of a network. It is a group composed of all the basic blocks that are predecessors of any basic block of a network. Furthermore, there is the *related signature set*, which is a set containing all the BBs' NES of the network predecessors, plus the NS of the BBs of type A in the current network.

The signatures in CEDA are divided into upper and lower half. The upper half of NS and NES are related to the network. For each network, the signatures contained in the related signatures set have the upper half assigned the same unique value. The remaining NS and NES have a unique and independent value assigned to their upper halves. The upper half signature is never masked, so it can detect any illegal branch, except for those between basic blocks belonging to the same related signature set. In order to detect such errors, the lower half is used, and it is assigned based on the following rules, considering the signatures as binary numbers:

- Since signatures in the same related signature set have the same upper half, the lower half cannot be equal, with exception of the NES signatures that belong to basic blocks that share a common successor of type X. In these cases, the entire signature (upper and lower half) must be equal
- Let *set of zeros* of a signature be all the zeros in the lower half. The positions of the zeros must also be taken into account. Considering BB_1 as a basic block of type A, and BB_2 as being a predecessor of BB_1, the set of zeros of BB2's NES must be contained in the set of zeros of BB_1's NS, but they cannot be equal. It makes possible the transition from BB_2 to BB_1 since the set of zeros of BB_1's NS is masked by the AND operation from Eq. 3.7
- Considering BB_1 as a basic block of type A that belongs to a network *Net*, and BB_2 a basic block that is not a predecessor of BB_1, but that belongs to the network predecessors of *Net*. The set of zeros of BB_2's NES cannot be contained in the set of zeros of BB_1's NS. Thus, an illegal branch to the beginning of a basic block of type A (from BB_2 to BB_1) will not be masked.

CEDA is an efficient control-flow technique when compared to previous control-flow techniques because it presents error detection rates similar to YACCA and performance overhead similar to CFCSS. However, when a program has many small basic blocks, i.e., basic blocks with just a few instructions, the overhead introduced by CEDA is high.

3.2.6 HETA

HETA, acronym for *Hybrid Error-Detection Technique Using Assertions*, is a hybrid control-flow technique that mixes SIHFT techniques and hardware techniques (AZAMBUJA, 2013). The software part of HETA is based on CEDA. It also classifies basic blocks into types A and X and add them to networks. But instead of using two signatures, HETA uses three signatures:

- **Node Ingress Signature (NIS)**: a signature for entering the basic block

- **Node Signature (NS)**: the basic block signature. It is defined by XOR operations of all instructions contained in the basic block and the memory address of its first instruction
- **Node Exit Signature (NES)**: a signature for exiting the basic block.

The aim of NIS and NES is to detect illegal branches between different basic blocks. It follows the same idea of CEDA. The NS is used to detect illegal branches inside a basic block. A global predetermined register *S* is used to store the current signature. The update of *S* from NIS to NS and from NS to NES follows the equation 2.10. The invariant is a constant necessary to update *S* to its next value.

(Eq. 3.10) $S = S \; XOR \; invariant(BB_i)$

To update *S* from NES to NIS, the possible equations are:

(Eq. 3.11) $S = S \; AND \; invariant(BB_i)$, if BB_i is of type A

(Eq. 3.12) $S = S \; XOR \; invariant \; (BB_i)$, if BB_i is of type X

When the basic block is of type X, NIS and NS can be combined for optimization because the updating of *S* to NS is subsequent to its updating to NIS.

Another novelty of HETA is related to the upper and lower halves of the signatures. HETA reserves the maximum number of bits to the lower half, i.e., the upper half, which is related to the networks, receives the minimum number of bits possible, $log_2(\#networks)$. Thus, the probability of bits being masked is lower since the upper half is never masked and there are more bits to represent the lower half.

The NS cannot detect illegal branches inside a basic block by itself only. It needs assistance of a watchdog. The watchdog reads the memory address and data buses. When a basic block starts, the internal register *W* of the watchdog is reset. The beginning of the basic block is determined by the watchdog using the XOR operation that updates *S*. Register *W* is updated by performing XOR operation on the instructions coming from the memory. Checkers to determine if *S* contains the correct value are done by performing store operations of *S* to a predetermined memory address. The watchdog compares if *S* and *W* have different values to determine if an error has occurred. Illegal branches inside of a basic block would also be detected because, in such cases, the watchdog would perform XOR operation on some instructions twice, or it would just skip some instructions, which would result in a *W* different from *S*.

Furthermore, the rule proposed by Rebaudengo (1999) to detect incorrect decisions in conditional branches, which is implemented by CEDA, is not implemented by HETA. Azambuja implements it in a particular technique called *Inverted Branches* (AZAMBUJA, 2011c). Anyhow, the author proposes the use of both techniques together to improve the protection of the control-flow.

HETA presents very high detection rates. It is even capable of detecting illegal branches inside a basic block due to the help of a watchdog. Such errors are impossible to be detected by software-only techniques. However, the use of an extra signature increases the performance and memory overheads, when compared to CEDA, because it needs an extra instruction to update *S* in every basic block of type A. The power consumption is increased due to the additional hardware. Also, the additional hardware affects the portability of the technique to various platforms (GOLOUBEVA, 2003). And, as the author stated, the watchdog needs access to the memory buses. Some processors

that use on-chip embedded cache memories may not be accessible by the watchdog. It would make impossible the implementation of this technique.

3.2.7 Drawbacks of control-flow techniques

Control-flow techniques assign signatures and checkers to basic blocks. The main difference among these techniques is the way the signatures are attributed and updated. Table 3.2 shows the overheads for CFCC, ECCA, and YACCA. One can notice that ECCA presents significant higher overheads when compared to CFCSS and YACCA.

Table 3.2: Overheads of CFCSS, ECCA and YACCA.

Program	Code size			Execution time		
	CFCSS	ECCA	YACCA	CFCSS	ECCA	YACCA
Matrix multiplication	2.61x	4.08x	1.91x	1.35x	1.99x	1.47x
5th order elliptical wave filter	1.24x	1.53x	1.29x	1.07x	1.20x	1.10x
Kalman filter	1.64x	2.82x	2.17x	1.17x	1.68x	1.56x
LZW data compression	3.38x	6.30x	4.96x	1.85x	4.26x	3.54x

Source: (GOLOUBEVA, 2003).

Vemu (2011) compared CEDA with CFCSS and YACCA. Only three types of control-flow faults were injected: (1) branch deletion, jump instruction was replaced by *nop* instruction, (2) branch creation, the value in the PC corresponding to any instruction was corrupted, and (3) corrupting the target address of branch instruction. As can be noticed, CFCSS and CEDA present the lower performance overhead, and YACCA and CEDA present lower percentage of undetected faults. Thus, CEDA is the best state-of-the-art software-only control-flow technique.

Table 3.3: Percentage of undetected faults (%UF) and performance overhead (%PO) for CFCSS, YACCA, and CEDA.

Benchmarks	CFCSS		YACCA		CEDA	
	%UF	%PO	%UF	%PO	%UF	%PO
parser	4.6	14.36	1.0	33.9	1.1	13.79
gzip	3.4	57.7	0.7	84.32	0.6	57.8
ammp	4.7	4.45	0.3	78.97	0.2	3.15
twolf	2.8	7.5	0.6	39.8	0.6	9.8
equake	2.8	18.81	0.5	33.9	0.5	17.9

Source: (VEMU, 2011).

The overheads presented by control-flow techniques are lower than the ones presented by data-flow techniques. However, there is still room to reduce such overheads. Furthermore, the use of selective hardening is not well explored in control-flow techniques.

3.3 Combined data-flow and control-flow techniques

Some SIHFT techniques combine characteristic of both data-flow and control-flow techniques. Usually, such techniques are composed of transformation rules. Part of these

rules aim at protecting the control-flow, and the other part aim at protecting the data-flow. The combined data and control-flow techniques can be understood as an independent data-flow technique and an independent control-flow technique applied together. Some optimizations are possible when the protection of both techniques overlaps. In the following subsections, some combined data-flow and control-flow techniques are presented.

3.3.1 Transformation rules by Rebaudengo

Transformation rules in a high-level language have been proposed by Rebaudengo (1999), and, lately, implemented by Cheynet (2000). They consist of rules that modify the program code, introducing data and code redundancy. Since the rules are applied in the high-level code, they are independent of the processor architecture. However, the compiler optimization flags have to be disabled to avoid that the redundant code, inserted by the technique, is removed by the compiler.

The first three rules of Rebaudengo aim at protecting the data. They consist of duplicating the variables and inserting checkers right after the variables are read. The rules are:

- **Rule #1**: every variable *x* must be duplicated: let *x1* and *x2* be the names of the two copies
- **Rule #2**: every write operation performed on *x* must be performed on *x1* and *x2*
- **Rule #3**: after each read operation on *x*, the two copies *x1* and *x2* must be checked for consistency, and an error detection procedure should be activated if an inconsistency is detected.

Fig. 3.10 shows an example of how these rules are applied. The original code is presented at left, and the code modified by the rules is presented at right. The original instructions can be seen at lines 1 and 4. The replicas are showed at lines 2 and 5. Checkers are inserted after the variables are read, at lines 3 and 6. These rules are the same of VAR1, but in this case, they are implemented in a high-level language. The same rules discussed above must be applied to the parameters passed to a procedure. Fig. 3.11 shows an example of it. The return value is also duplicated. To do so, the return variable and its copy are passed as reference to the procedure.

original code	**modified code**

`1: a = b;`	`1: a1 = b1;`
	`2: a2 = b2;`
	`3: if (b1 != b2) error();`

`4: a = b + c;`	`4: a1 = b1 + c1;`
	`5: a2 = b2 + c2;`
	`6: if ((b1 != b2) || (c1 != c2)) error();`

Fig. 3.10: Transformation rules by Rebaudengo to protect the data.

original code	modified code

`res = search(a);`
`...`
`int search (int p)`
`{`
`  int q;`
`  ...`
`  q = p + 1;`
`  ...`
`  return(1);`
`}`
	`search(a1, a2, &res1, &res2);`
	`...`
	`int search (int p1, int p2, int *r1, int *r2)`
	`{`
	`  int q1, q2;`
	`  ...`
	`  q1 = p1 + 1;`
	`  q2 = p2 + 1;`
	`  if (p1 != p2) error();`
	`  ...`
	`  *r1 = 1;`
	`  *r2 = 1;`
	`  return;`
	`}`

Fig. 3.11: Transformation rules by Rebaudengo to protect the data in procedures.

Other five rules have been proposed to protect the control-flow. Firstly, the code is divided into basic blocks. Then the following rules are inserted to check if all the instructions in a basic block were all executed in sequence.

- **Rule #4**: an integer value k_i is associated with every basic block *i* in the code
- **Rule #5**: a global execution check flag (*ecf*) variable is defined; a statement assigning to *ecf* the value of k_i is introduced at the very beginning of every basic block *i*; a test on the value of *ecf* is also introduced at the end of the basic block.

Fig. 3.12 provides an example of how rules #4 and #5 are applied. If there is an illegal branch from one basic block to another, the different *ecf* value will signalize the error. However, if the illegal branch goes to the beginning of the other basic block, to exactly the instruction that assigns the signature, the error will not be detected.

original code	modified code

`/* basic block beginning */`
`...`
`/* basic block end */`
	`/* basic block beginning #371 */`
	`ecf = 371;`
	`...`
	`If (ecf != 371) error();`
	`/*basic block end */`

Fig. 3.12: Transformation rules by Rebaudengo to protect the control-flow.

Furthermore, to detect errors affecting the branch decisions, which cannot be detected by signatures, another rule is proposed:

- **Rule #6**: for every test statement the test is repeated at the beginning of the target basic block of both the true and false clauses. If the two versions of the test produce different results, an error is signaled.

An example of how rule #6 is applied to the code can be seen in Fig. 3.13. At the beginning of each basic block, a condition opposite to the valid one is inserted to check if the branch was correctly taken.

original code	modified code

`if (condition)`
`{`
`  /* BB 1 */`
`  ...`
`}`
`else`
`{`
`  /* BB 2 */`
`  ...`
`}`
	`if (condition)`
	`{`
	`/* BB 1 */`
	`if (!condition) error();`
	`...`
	`}`
	`else`
	`{`
	`/* BB 2 */`
	`if (condition) error();`
	`...`
	`}`

Fig. 3.13: Transformation rules by Rebaudengo to protect branch decisions.

The last two rules are used to protect procedures against control-flow errors. Such rules are presented as follows:

- **Rule #7**: an integer value k_j is associated with any procedure j in the code
- **Rule #8**: immediately before every procedure return statement, the value k_j is assigned to *ecf*; a test of *ecf* is also introduced after any call to the procedure.

These rules detect illegal branches to a procedure, erroneous target address of the function calls, and errors affecting the register that contains the return address. An example of rules #7 and #8 is presented in Fig. 3.14.

original code	modified code

`...`
`ret = my_proc(a);`
`/* procedure call */`
`...`
 	
`/* procedure definition */`
`int my_proc(int a)`
`{`
`  /* procedure body */`
`  ...`
`  return (0);`
`}`
	`...`
	`/* call of procedure #790 /*`
	`ret = my_proc(a);`
	`if (ecf != 790) error();`
	`...`
	`/* procedure definition */`
	`int my_proc(int a)`
	`{`
	`  /* procedure body */`
	`  ...`
	`  ecf = 790;`
	`  return (0);`
	`}`

Fig. 3.14: Transformation rules by Rebaudengo to protect the control in procedures.

The transformation rules by Rebaudengo present very high overheads. First of all, protecting in the high-level language produces higher overheads than protecting in the low-level (CHIELLE, 2014), (RESTREPO-CALLE, 2011). The deactivation of the flags to optimize the code during compilation significantly reduces the performance. Furthermore, besides the duplication, checkers are inserted to verify every single read for every variable. The overheads it causes are clearly very high because, for most of the instructions in the code, two other instructions are added, one to duplicate and the other to check the variables. If the instruction reads two variables, the checker needs to test both variables. This more complex checker is converted to assembly in more than one instruction, which makes the overheads even higher. Therefore, performing checkings every time a variable is read is very costly. Additionally, the signatures for control-flow protection can easily be masked because they are assigned for each basic block and not updated from one basic block to the other.

3.3.2 Transformation rules by Nicolescu

Nicolescu (2003) proposed new transformation rules to modify the high-level code, specifically in C language, and provide reliability to it. The rules are partially based on (REBAUDENGO, 1999). There are rules to protect the data-flow and rules to protect the control-flow. The rules to protect the data-flow duplicate all variables and perform, on their replicas, the same operations performed on the original variable. Checkers are inserted to verify only the final variables, which are variables that do not take part in the calculation of any other variable. For example, in Fig. 3.15 (a), variables *c* and *b* are used to compose variable *a*; therefore, *c* and *b* are considered intermediary variables. In the following instruction, *a* and *b* are used to define variable *d*. Since *a* is used to define the value of another variable, it is also an intermediary variable. On the other hand, variable *d* is not used by any other variable, so it is a final variable, and it will have its value checked with its replica. The rules proposed by Nicolescu to protect the data-flow are:

- Identification of the relations among the variables
- Classification of the variables according to their role in the program: *intermediary variable* or *final variable*
- Every variable *x* must be duplicated: let *x1* and *x2* be the names of the two copies
- Every operation performed on *x* must be performed on *x1* and *x2*
- After each write operation on the *final variables*, the two copies *x1* and *x2* must be checked for consistency, and an error detection procedure is activated if an inconsistency is detected.

One can see the application of these rules in Fig. 3.15. All variables are duplicated, and the operation is replicated to the copies. Since *d* is the only final variable in this code, a checker is inserted to verify only *d*. The value of *d1* is compared with the value of *d2* after a write operation on them. If they have the same value, the program execution continues normally. Otherwise, an error detection procedure is called.

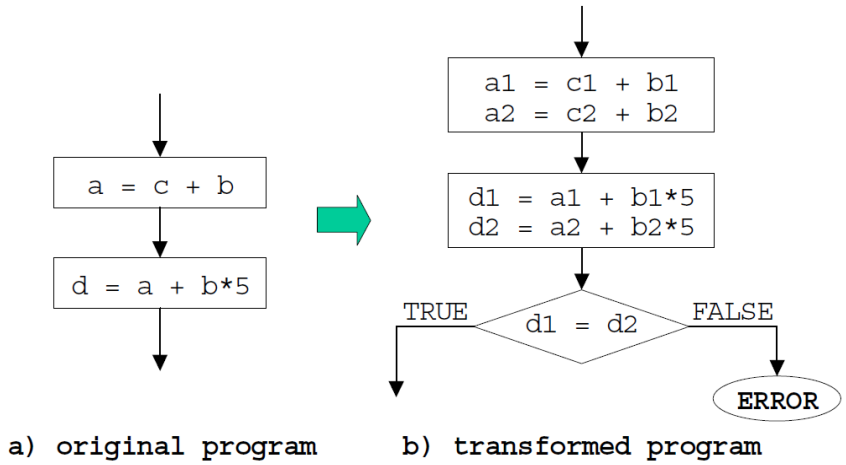


Fig. 3.15: Transformation rules to protect the data (NICOLESCU, 2003).

For the protection of the control-flow, the code is divided into basic blocks, and for each basic block, a Boolean flag is assigned. The flag takes the value zero if the BB is active, and one otherwise. It is updated by incremented modulo 2 at both beginning and end of the basic block. Thus, the value is always one or zero. Furthermore, an integer is associated with every basic block. A global signature variable is assigned based on the Boolean flag and the associated integer. The control-flow rules are presented as follows:

- A boolean flag *status_block* is associated with every basic block *i* in the code; 1 for the inactive state and 0 for the active state
- An integer value *ki* is associated with every basic block *i* in the code
- A global execution check flag (*gef*) variable is defined
- A statement assigning to *gef* the value of (*ki & (status_block = status_block + 1) mod 2*) is introduced at the beginning of every basic block *i*; a test on the value of *gef* is also introduced at the end of the basic block.

Fig. 3.16 shows an example of how the control-flow rules are applied. The *status_block* flag is updated at the beginning of the basic block, and the *gef* is updated right after. At the end, *gef* is compared with the basic block integer signature. If they are different, the error procedure is called. In theory, it should avoid masking errors when an illegal branch goes to the beginning of the basic block because, in this case, *status_block* would be updated to one, which would make *gef* different from the expected signature *i*, and it would be detected at the end of the basic block. However, when compiled, the update at the beginning of the basic block is divided in two assembly instructions, one to update *status_block* and another to update *gef*. If the illegal branch goes to the second instruction, the error would not be detected.

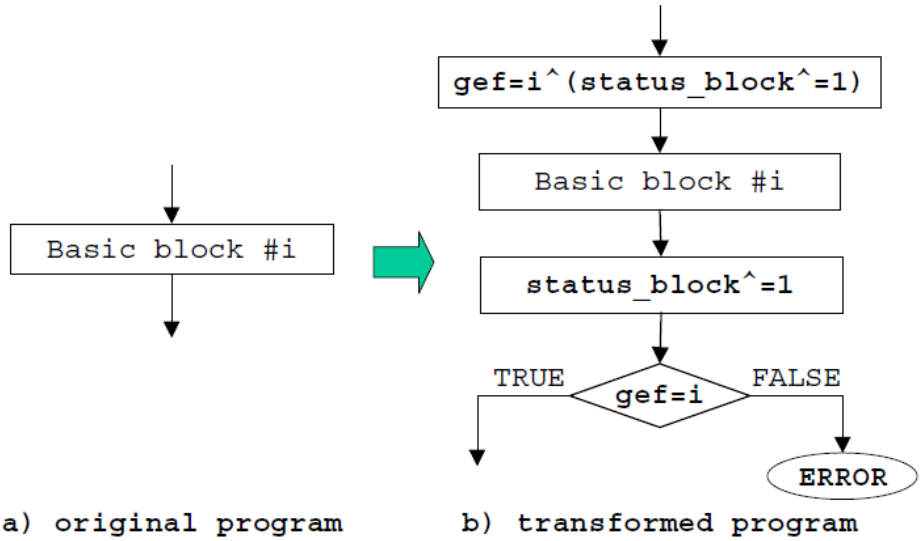


Fig. 3.16: Transformation rules to protect the control-flow (NICOLESCU, 2003).

The incorrect branch decisions cannot be detected by the control-flow rules described above. Thus, Nicolescu implements Rebaudengo's rule to protect that. The branches are retested at both true and false destinations of the branches. This transformation can be seen in Fig. 3.17.

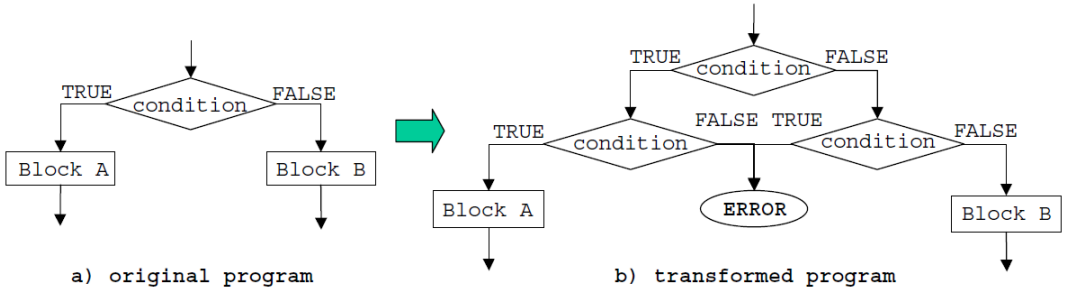


Fig. 3.17: Transformation to protect branch decisions (NICOLESCU, 2003).

Furthermore, a *ctrl_branch* flag is assigned to every procedure in the program. The value is assigned at the beginning of each procedure. Checkers are inserted before and after the procedure call. The rules are presented bellow, and their application is illustrate in Fig. 3.18.

- A flag *ctrl_branch* is defined in the program
- An integer value *kj* is associated with any procedure *j* in the code
- At the beginning of every procedure, the value *kj* is assigned to *ctrl_branch*; a test on the value of *ctrl_branch* is introduced before and after every call to the procedure.

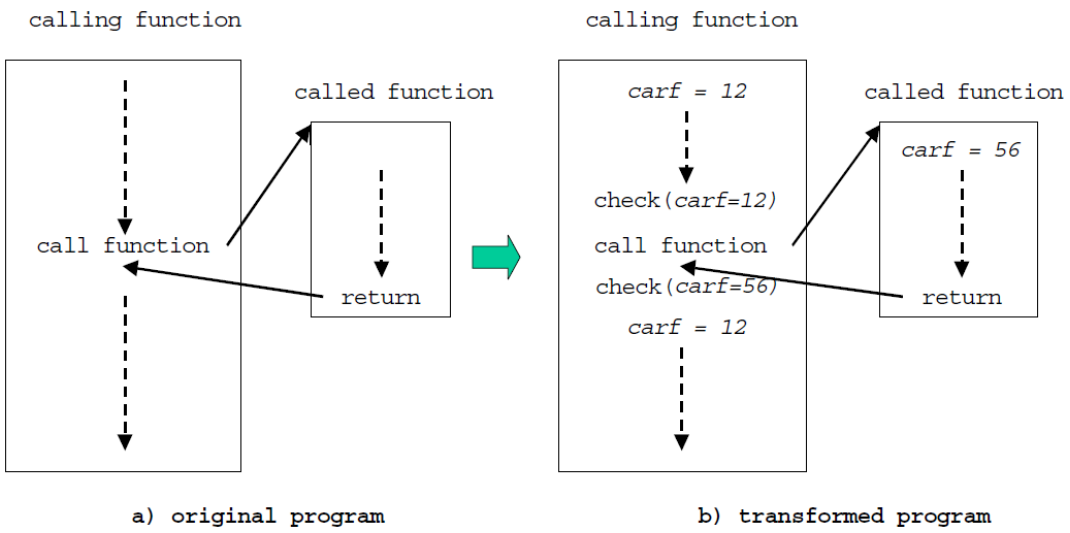


Fig. 3.18: Transformation rules to protect procedures (NICOLESCU, 2003).

The transformation rules by Nicolescu present high overheads. They are implemented in the high-level language, which makes necessary to deactivate the optimizations performed by the compiler. The rules to protect the data are less costly than Rebaudengo's since they only check the final variables. On the other hand, the control-flow techniques are more costly. The protection of procedures needs extra instructions, two checkers instead of one, and the value of *ctrl_branch* is also assigned after the return from the procedure. Furthermore, the basic blocks' signatures use two variables, one for the signature and another for the *status_flag*. When compiled, the operation to update *status_flag* and assign the new value to the signature variable *gef* is converted in two instructions. Besides being slower than Rebaudengo's by using two instructions instead of one, it will still not detect an illegal branch if it jumps to the instruction that assigns the signature variable.

3.3.3 SWIFT

Software Implemented Fault Tolerance (SWIFT) is an SIHFT technique proposed by Reis (2005b) that aims at protecting the data-flow and the control-flow of a running application. It is based on two techniques, one data-flow technique (EDDI) and one control-flow technique (CFCSS). The union of these two techniques, together with a set of optimizations that have been made, was named SWIFT.

The transformation rules to protect the data are very similar to EDDI, but there are two differences: (1) the authors assume that the memory is protected by some ECC. Thus, it is not necessary to duplicate the data in memory. Since there is no replicated data in memory, the loads for the replicas are done from the same memory position that for the originals. Furthermore, it is not necessary to duplicate store instructions. (2) Checkers to verify registers used by branch instructions are removed. The authors justify that by saying its protection is redundant with the control-flow technique. As follows, one can understand the transformation rules to protect the data:

- Every register in the program must be duplicated
- The same operation performed on a register must be performed on its replica, with the exception of stores

- Checkers are inserted before storing register values in memory. The variable is compared with its replica. If they are different, an error is signaled.

For protecting the processor against control-flow errors, CFCSS is implemented by SWIFT. However, it does not protect against incorrect branch decisions. A modification of the implemented CFCSS has been proposed to extend the fault coverage to such cases. It consists of a dynamic equivalent of a runtime adjusting signature for all basic blocks. The target basic block is assigned to the runtime adjusting signature before the branch instruction. The runtime adjusting signature is checked after the branch by comparing it with the *general signature register* (GSR). For optimization purpose, the checkings are done only in basic blocks with store instructions. If it is possible to ensure that only stores that should be executed are executed and that they write the right data, the application runs correctly.

One can see the transformation rules applied by SWIFT in Fig. 3.19. Instruction 1 and 2 are replicas applied by the transformation rules to protect the data. Instruction 3 computes the *runtime signature* (RTS) to the target basic block by performing an XOR operation using the current basic block's signature and the target basic block's signature. Instruction 4 does the same as instruction 3, but in the case the branch is not taken. Instruction 5 updates the GSR to the new basic block's signature. Instructions 6 and 7 are used to detect mismatches in the signature. They are included only in basic blocks that contain store instructions. And instructions 8-10 implement data-flow rules to check the registers before their values are stored in the memory.

```
add r11=r12,r13
cmp.lt.unc
   p11,p0=r11,r12

(p11) br L1
...

L1:

st m[r11]=r12
```

(a) Original Code

```
    add r11=r12,r13
1:  add r21=r22,r23
    cmp.lt.unc
       p11,p0=r11,r12
2:  cmp.lt.unc p21,p0=r21,r22
3:  (p21) xor RTS=sig0,sig1
    (p11) br L1
    ...
4:  xor RTS=sig0,sig1
    L1:
5:  xor GSR=GSR,RTS
6:  cmp.neq.unc p2,p0=GSR,sig1
7:  (p2) br faultDetected
8:  cmp.neq.unc p3,p0=r11,r21
9:  cmp.neq.or p3,p0=r12,r22
10: (p3) br faultDetected
    st m[r11]=r12
```

(b) EDDI+ECC+CFE Code

Fig. 3.19: Transformation rules (REIS, 2005b).

SWIFT is an efficient SIHFT technique when compared to previous software techniques to protect both data-flow and control-flow. It presents performance improvements. However, the control-flow part of SWIFT uses two signatures transformations per basic block, GSR and RTS, and it increases the overheads in time and performance.

3.3.4 Transformation rules by Azambuja

Transformation rules in software were combined with a watchdog in order to cover errors not detectable by SIHFT techniques. This hybrid technique was proposed by (AZAMBUJA, 2011b). It consists of the data-flow technique VAR1, plus two control-

flow techniques, the Inverted Branches and a modified version of CCA that works together with a watchdog, which is connected to the memory bus. The rules to protect the data are:

- **Rule #1**: variables must be duplicated
- **Rule #2**: write operations performed on a variable must also be performed on its replica
- **Rule #3**: before each read on a variable, its value and the value of its replica must be checked for consistency.

Furthermore, there is the rule for the Inverted Branches technique, which aims at detecting incorrect branch decisions. It is presented below:

- **Rule #4**: every branch instruction is replicated on both destination addresses.

A modified version of CCA was implemented to work with the watchdog. It aims at detecting two types of illegal branches: (1) to the beginning of a basic block and (2) to the same basic block. The same concepts introduced in CCA using BID to identify the basic blocks and CFID in a queue to protect the transitions between basic blocks is used here. However, the queue management, which is a task that significantly increases the overheads, is performed by the watchdog. To inform the watchdog about the BID or to tell when to enqueue or dequeue CFID, store instructions to predetermined memory addresses are used. An example of how the code changes by this modified CCA is presented in Fig. 3.20. The target address of the branch at line 1 is modified to keep the correctness of the program. Stores to send the BID to the watchdog are inserted at the beginning of the basic blocks (lines 2 and 6). CFID is enqueued at lines 3 and 7, also beginning of the basic blocks, right after sending the BID. And CFID is dequeued at the end of the basic blocks (lines 5 and 10). The enqueuing and dequeuing of CFID is also done using store instructions.

XOR operations are done in real-time by the watchdog to detect illegal branches inside the same basic blocks. The executed instructions of a basic block are xored and compared to the expected value for that basic block. If they differ, an illegal branch inside of the basic block has occurred. Fig. 3.21 shows how this technique is applied. At the beginning of each basic block, a store operation to a predetermined memory address signalizes the watchdog to reset its internal register that contains the result of the XOR operations of the executed instructions. During the execution, the executed instructions are xored and stored in the watchdog's internal register. At the end of the basic block, another store instruction to a predetermined memory address signalizes the watchdog to check its register with the expected value for that basic block.

1: beq r1, r2, 8	1: beq r1, r2, **6**
	2: **send BID**
	3: **enqueue CFID**
4: add r2, r3, 1	4: add r2, r3, 1
	5: **dequeue CFID**
	6: **send BID**
	7: **enqueue CFID**
8: add r2, r3, 4	8: add r2, r3, 4
9: st [r1], r2	9: st [r1], r2
	10: **dequeue CFID**
11: jmp end	11: jmp end

Fig. 3.20: Example of the modified CCA proposed by (AZAMBUJA, 2011b).

1: beq r1, r2, 8	1: beq r1, r2, **5**
	2: **reset XOR**
3: add r2, r3, 1	3: add r2, r3, 1
	4: **check XOR**
	5: **reset XOR**
6: add r2, r3, 4	6: add r2, r3, 4
7: st [r1], r2	7: st [r1], r2
	8: **check XOR**
9: jmp end	9: jmp end

Fig. 3.21: Technique to detect illegal branches inside basic blocks by (AZAMBUJA, 2011b).

The transformation rules by Azambuja present very high detection rates. It is capable of detecting errors inside a basic block, which is impossible to detect by software-only techniques. Although the high overhead, to enqueue and dequeue that CCA presents, was moved to the watchdog, it still presents significant overheads. The modified CCA uses two signatures, so in every basic block, three instructions are inserted: one to send BID to the watchdog; one to enqueue CFID; and one to dequeue CFID. Furthermore, other two instructions to detect illegal branches inside a basic block are also included in all basic blocks. For programs with small basic blocks, the overheads it causes are very high. Some of the flaws presented by CCA are also presented in this technique. When two basic blocks have a common successor, all the successors have the same CFID. For example, if basic block A has C and D as successors, and basic block B has D and E as successors, all C, D, and E have the same CFID. Thus, illegal branches from A to E or from B to C will not be detected. This problem was later corrected when Azambuja replaced the modified CCA by HETA. Anyhow, the overheads are the same. Furthermore, the technique to detect data errors is VAR1. This technique presents very high overheads because, before each read on a variable, a checker is inserted to check the variable with its replica. In addition, the power consumption due to the additional hardware is increased. The additional hardware also affects the portability of the technique to other platforms because it needs access to the memory buses. Some processors that use on-chip

embedded cache memories may not be accessible by the watchdog and, thus, the technique could not be implemented.

3.3.5 Drawbacks of combined data-flow and control-flow techniques

The combined data-flow and control-flow techniques are nothing more than a data-flow technique and a control-flow technique applied together. Therefore, the same comments for data-flow and control-flow techniques can be extended to the combined data-flow and control-flow techniques. With regards to the presented techniques, SWIFT is by far the one with the lower overheads. It is mainly due to its data part, which implements EDDI and uses checkers only before stores. The control-flow part is complemented by CFCSS, a control-flow technique that has low overheads, but not the highest fault coverage. The advantage of SWIFT over EDDI + CFCSS consists in the fact that the redundant protection of branches is removed. However, the overheads are still high. Another approach to reducing more the overheads is essential.

3.4 Selective hardening

A recent approach to reduce overheads caused by SIHFT techniques consists of applying them selectively. Only selected portions of the application are protected, not the entire application. Few works based on selective hardening aim to guarantee application-level correctness in multimedia applications (CONG, 2011), (SUDARAM, 2008). For multimedia applications, some errors can be tolerated since they will not be noticed by the user (YEH, 2009). However, in critical systems, correctness is required. A recent work on this field was proposed by (RESTREPO-CALLE, 2013). In this work, subset of the registers used by the application were protected by data-flow techniques and evaluated.

With regards to data-flow techniques, the selective hardening is applied to registers, i.e., the most critical subset of used registers is protected. For control-flow techniques, selected basic blocks are protected. In the literature, (VEMU, 2011) states that the number of checkers in the code can be reduced to decrease overheads. On the other hand, it increases the latency of the error detection. The selective hardening also brings flexibility to SIHFT techniques due to the new range of possibilities in which a code can be hardened. It can bring reliability given a maximum time overhead, or reduce overheads given a minimum fault coverage, for example.

Furthermore, portions of code can be selectively hardened. For example, an application that is not critical in general, but that has few critical subroutines. These critical subroutines could be hardened, while the rest of the application is left unhardened. It would increase the reliability of the application when compared to the unhardened, and it would reduce the overheads when compared to hardening the entire application. This type of selectiveness is application-dependent, so a study on the target application must be performed. It is not part of this work, which focuses on selective hardening for data-flow and control-flow techniques. A state-of-the-art selective technique is presented as follows.

3.4.1 Selective SWIFT-R

S-SWIFT-R stands for *Selective Software Implemented Fault Tolerance – Recovery.* It is a selective hardening technique proposed by Restrepo-Calle (2013). It is based on the software recovery technique SWIFT-R (REIS, 2007), which is based on the software detection technique SWIFT (REIS, 2005b). Later, Restrepo-Calle (2016) proposed a selective hybrid technique combining S-SWIFT-R with a hardware *Selective Triple*

Modular Redundancy (S-TMR). Some registers were protected by TMR in order to reduce the overheads caused by SWIFT-R. On the other hand, it was necessary to modify the underlying hardware, which increased the area and power consumption, as the author stated.

SWIFT-R triplicates the registers and instructions, and it inserts software majority voters to identify and correct errors. Its aim is to protect the register file. S-SWIFT applies SWIFT-R selectively, i.e., a selected set of registers is triplicated, instead of all used registers. Fig. 3.22 shows examples of a code hardened by S-SWIFT-R. Several versions where different sets of registers have been protected are presented. The version that all used registers are protected (*s0* and *s1*) is equivalent to SWIFT-R. In the version that *s0* is protected, copies of *s0* are created after it is loaded from the memory. Then, the *ADD* operation is replicated to the copies. Finally, a voter is inserted before *s0* is stored. In the version where only *s1* is protected, copies of *s1* are created, and two voters are inserted. The first voter (line 5) is due to the use of *s1* by the *ADD* operation (line 6) to define a new value to *s0*, which is stored after. And the second voter is due to the use of *s1* to address the memory in the *STORE* operation. No instruction is replicated since no new value is attributed to *s1*.

#	unhardened	protected: s0	protected: s1	protected: s0, s1
1	LOAD s0, 00	LOAD s0, 00	LOAD s0, 00	LOAD s0, 00
2		*Create s0 copies*		*Create s0 copies*
3	LOAD s1, 2A	LOAD s1, 2A	LOAD s1, 2A	LOAD s1, 2A
4			*Create s1 copies*	*Create s1 copies*
5			**Voter for s1**	
6	ADD s0, s1	ADD s0, s1	ADD s0, s1	ADD s0, s1
7		ADD s0', s1'		ADD s0', s1'
8		ADD s0'', s1''		ADD s0'', s1''
9		**Voter for s0**		**Voter for s0**
10			**Voter for s1**	**Voter for s1**
11	STORE s0, (s1)	STORE s0, (s1)	STORE s0, (s1)	STORE s0, (s1)

Fig. 3.22: Example of a code hardened by S-SWIFT-R (RESTREPO-CALLE, 2016).

Fig. 3.23 shows the code size and execution time overheads for a *Finite Impulse Response* (FIR) filter implemented using five registers for the PicoBlaze soft-core processor. Each possible set of registers was protected using S-SWIFT-R. The horizontal axis presents the names of the registers protected in that version. The code overhead varies from 1.01x to 2.67x, and the execution time ranges from 1.01x to 2.53x.

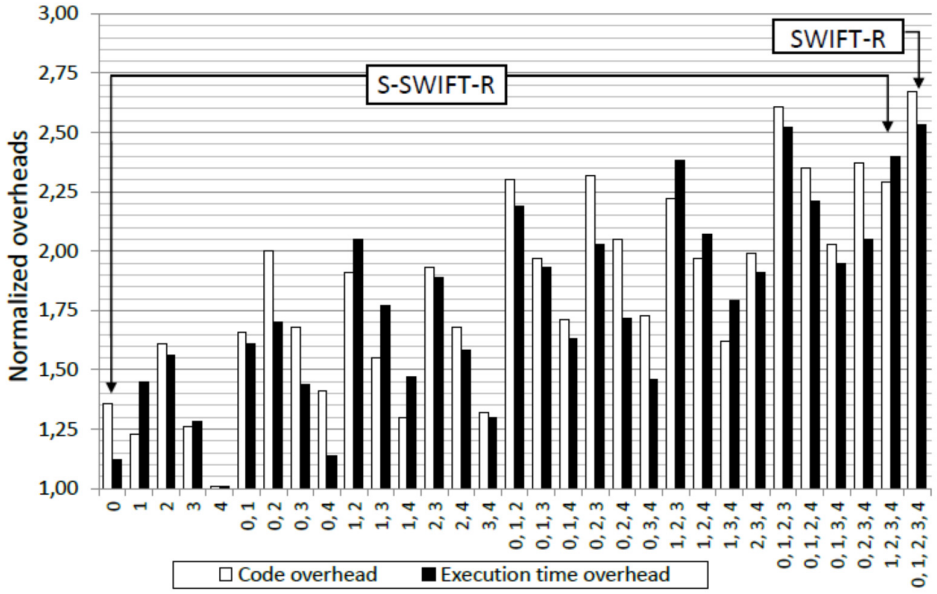


Fig. 3.23: Code size and execution time overheads for an FIR hardened by S-SWIFT-R (RESTREPO-CALLE, 2016).

The fault coverages of several versions of an FIR hardened by S-SWIFT-R are presented in Fig. 3.24. The fault coverage is indicated by *unACE*. SDC (Silent Data Corruption) indicates the faults that caused errors in the output. And *Hang* represents the faults that caused abnormal program termination or infinite loop. As one can see, the unhardened version presents around 74% of fault coverage. Meanwhile, the one with all registers hardened (SWIFT-R) reached 92% of fault coverage. The S-SWIFT-R goes from 82% to 92%. An interesting example can be seen by the protection of register 2 and 3. It achieves 89.6% of fault coverage with a cost of 1.89x in the execution time and 1.93x in the code size. Depending on the case, it could be a better solution than SWIFT-R.

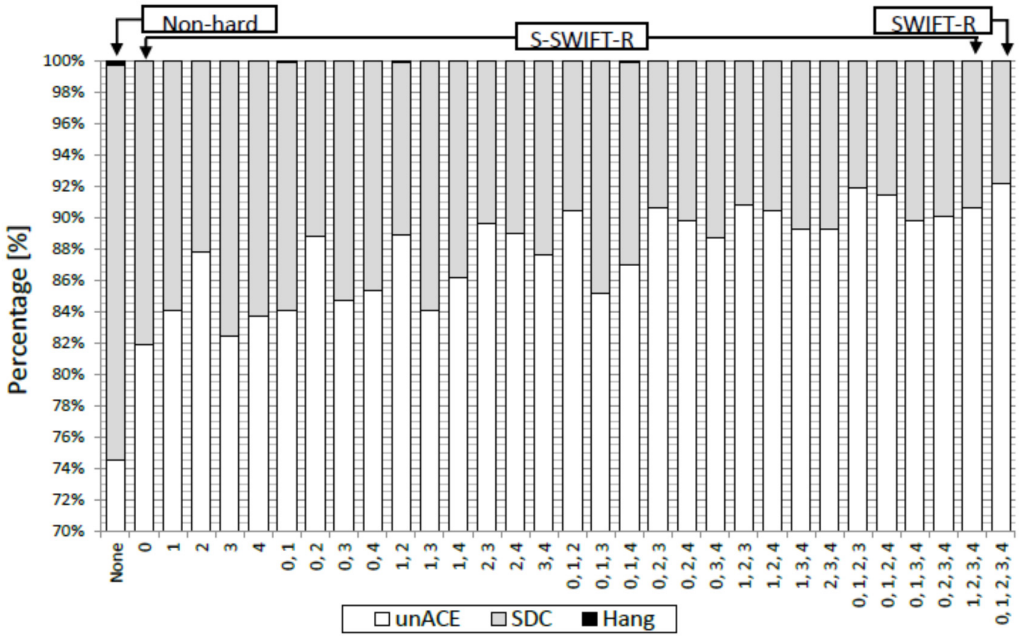


Fig. 3.24: Fault coverage for an FIR using S-SWIFT-R (RESTREPO-CALLE, 2016).

S-SWIFT-R reduces the overheads when compared to the standard SWIFT-R. However, since it triplicates instead of duplicating the registers, it causes higher overheads. Furthermore, the selective hardening is only applied to registers. Therefore, it is only a selective data-flow technique. The selective hardening for control-flow techniques is, so far, limited to removing checkers from basic blocks. A new approach for selective hardening of control-flow techniques that explores better the basic block protection is necessary.

4 METHODOLOGIES AND METRICS

The subjects of this chapter are the methodologies for hardening applications with SIHFT techniques and the fault injection utilized in this work. Furthermore, the metrics to evaluate and compare the proposed techniques are introduced.

The concept of fault, error, and failure was already defined in a previous chapter. Nevertheless, to elucidate the discussion of this chapter, Fig. 4.1 shows a fault and its possible effects on a processor hardened by SIHFT techniques. If the fault is masked, the application runs correctly. In case of error, this error can be detected or undetected. An error detected would possibly cause a failure if it were not detected, but it is also possible that this error would not produce a failure. Finally, an undetected error causes necessarily a failure; otherwise, it would be a masked fault.

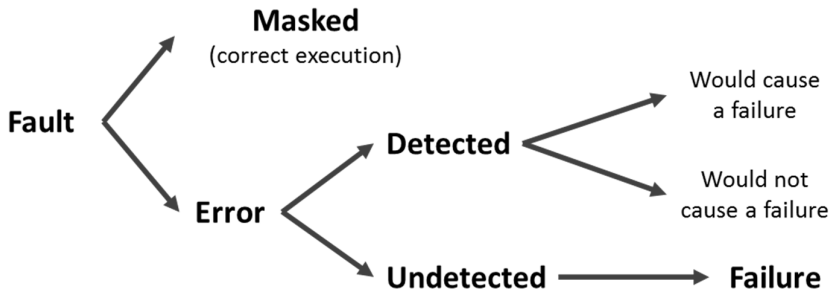


Fig. 4.1: Fault, error, and failure in processors hardened by SIHFT techniques.

4.1 Hardening methodology

The SIHFT techniques are automatically applied to the assembly code of unhardened applications using the CFT-tool (CHIELLE, 2012).[4] For this, the code in a high-level language is compiled and assembled, generating, respectively, the assembly code and the executable. Sometimes, the CFT-tool needs additional information not available in the assembly code. Thus, it is necessary to provide the disassembly file to the tool. The entire process is illustrated in Fig. 4.2. CFT-tool reads the assembly code and the disassembly file, and, based on some configuration files, creates a new assembly code hardened by the selected SIHFT techniques. The configuration files contain information about the processor architecture and organization the SIHFT techniques. Finally, the hardened assembly can be assembled to generate a hardened executable.

[4] Details about the CFT-tool are available in the appendix A.

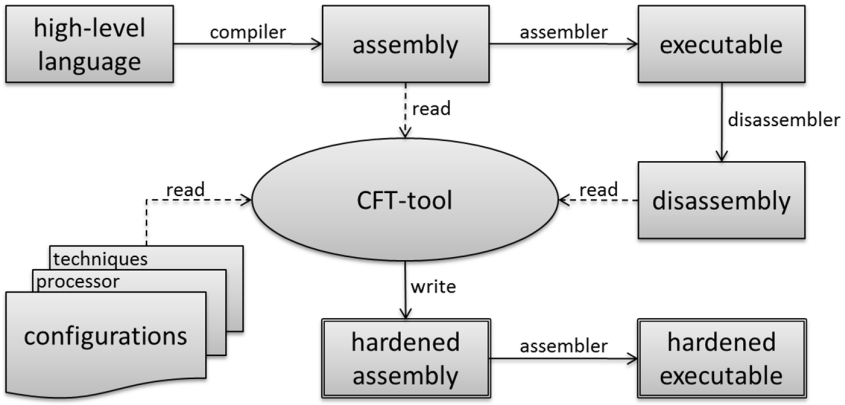


Fig. 4.2: Steps to protect an application using CFT-tool.

After the process described above, the hardened application is ready to run on the target processor. In this stage, the execution time and code size can be extracted. It is also possible to evaluate the application's fault coverage by performing a fault injection campaign. The methodology for fault injection utilized in this work is described as follows.

4.2 Fault injection methodology

It would be infeasible to acquire the large amount of information about fault coverage necessary for this work only from radiation experiments. Thus, most of the tests were performed through simulation. Anyhow, radiation tests were done in selected cases to confirm the results obtained by simulation. The simulated fault injections at logical level using the hardware description (HDL) of the processor. The radiation experiments utilized a hard-core processor embedded in an All Programmable SoC. Details about the simulated fault injection and the radiation experiments are presented below.

4.2.1 Fault injection by logical simulation

Faults are injected by forcing a bit flip at RTL level in the processor's internal signals using ModelSim (MENTOR GRAPHICS, 2012), a simulation tool. For this, it is necessary the processor's hardware description. A total of 10,000 faults is injected per version of each application[5]. Only one fault is injected per execution. It can affect any of the processor's internal signals. The fault duration is set to one clock cycle in order to force its effect to hit the clock barrier of the flip-flops and, thus, increase the probability of error. A golden execution (with no injected faults) is executed. All the PC values during the execution are saved. Also, the portion of the memory that contains the program output is saved. Then, the program is submitted to faults, and the values of the PC and the memory results of the program under test are compared with the gold results. The error is signaled when the result stored in the memory differs from the expected one. The effect of a fault can be classified as:

- **Correct**: the fault had no effect on the program output, i.e., the result is correct

[5] Applications with more than 98% of fault coverage may have up to 40,000 faults injected.

- **Data-flow error**: it occurs when the fault affects the output, but the PC is correct during the execution. It is also known as Silent Data Corruption (SDC). If the error is detected by a fault detection technique, it is classified as *detected data-flow error*. Otherwise, it is an *undetected data-flow error*
- **Control-flow error**: the output and PC are incorrect. It is a *detected control-flow error* if detected, and an *undetected control-flow error* if not.

The sum of data-flow errors and control-flow errors for a fault injection campaign gives the total of errors. The sum of detected data-flow errors and detected control-flow errors provides the total of errors detected. The processor utilized in the fault injections by logical simulation was miniMIPS (HANGOUT, 2009).[6]

4.2.2 Radiation tests with neutrons and heavy ions

Radiation tests were used to validate the fault injection campaigns by simulation. They were performed with neutrons and heavy ions. During the tests with neutrons, all the board was irradiated. During the tests with heavy ions, only the processor was irradiated. For the test with heavy ions, it was necessary to decapsulate the chip to allow the ions to hit the sensitive parts of the processor. In both cases, the board was exposed to air at room temperature.

The setup, shown in Fig. 4.3, consists of a board, computer, USB net switch, cables for communication, and cables for power supply. The computer is connected to the board by two USB cables. One is used to program the board, and the other is used to receive the output from the board. The board's power supply is connected to the USB net switch, which is connected by USB to the computer. It is used to control when the power supply is available to the board.

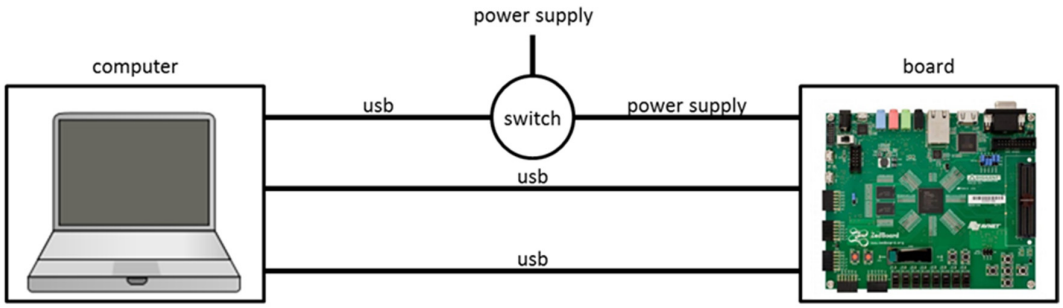


Fig. 4.3: Setup for radiation tests.

The board utilized in the tests was a ZedBoard™. It is a low-cost development board for the Xilinx Zynq®-7000 All Programmable SoC, XC7Z020-CLG484 part, which offers high configurability, stimulates strong interest in the scientific community, and is highly present in the market. The board is composed of two main parts: a Processing System (PS) that contains a dual-core ARM® Cortex-A9 processor[7], and Programmable Logic (PL) (AVNET, 2015)[8]. The PL section is ideal for implementing high-speed logic, arithmetic, and data processing subsystems, while the PS supports software routines and

[6] More information about the miniMIPS processor is available in the appendix B.1.

[7] Details about the ARM Cortex-A9 processor are available in the appendix B.2.

[8] More information about the ZedBoard™ is available in the appendix B.3.

operating systems. The proposed analysis is based only on the PS part of the board. The PL part is not used at any moment during the experiment. One only ARM core was utilized. It ran a target application that sends the output by UART to the computer and, then, restarts its execution. The computer was running a monitoring application that listens to the COM port connected to the board's UART and classifies the output as:

- **Correct**: the output of the program is correct
- **Detected**: an error was detected and reported
- **Undetected error**: the output is incorrect, and no error was reported by the fault detection technique
- **Timeout**: no output was produced until a time limit.

Depending on the output, different actions are taken:

- **No action**: for correct outputs
- **Soft reset**: the fault tolerance technique applied to the target application has detected an error. It calls the error subroutine that reports the error and restarts the application by software
- **Hard reset**: it happens when timeout, undetected errors, or consecutive detected errors are reported. The monitoring application performs a power cycle on the board and, then, reprograms it.

When consecutive detected errors are reported, they are accounted as only one detected error because it affected a region that was not corrected by the soft reset. Furthermore, the hard reset presents a delay in which the processor is still running and, possibly, producing output. For that reason, any output, since the hard reset is started until it is concluded, is ignored.

4.3 Metrics

Applications are hardened using the proposed techniques and some state-of-the-art techniques. For comparison purpose, the following parameters are utilized, in this work, to evaluate the quality of the proposed and state-of-the-art techniques:

- **Execution time**: it expresses the time that an application takes to execute. The execution time of a hardened application is presented normalized by the execution time of the equivalent unhardened application, as shown in Eq. 4.1. $T_{normalized}$ represents the normalized execution time of the hardened application, $T_{hardened}$ is the absolute execution time of the hardened application, and $T_{unhardened}$ is the absolute execution time of the unhardened application. For example, if the unhardened application runs in 5 ms and the hardened one runs in 10 ms, the hardened one is expressed as being 2x the execution time of the unhardened

  (Eq. 4.1) $$T_{normalized} = \frac{T_{hardened}}{T_{unhardened}}$$

- **Code size**: it refers to the total of bytes a program occupies in disk. As the execution time, the code size of a hardened application is also normalized by the unhardened application, as one can see in Eq. 4.2. $M_{normalized}$ represents the normalized code size of the hardened application, $M_{hardened}$ is the absolute

code size of the hardened application, and $M_{unhardened}$ is the absolute execution time of the unhardened application

(Eq. 4.2) $$M_{normalized} = \frac{M_{hardened}}{M_{unhardened}}$$

- **Error detection rate**: let $E_{undetected}$ be the number of undetected errors that led to an incorrect output (system failure), $E_{detected}$ the number of detected errors in executions with an incorrect output, and E_{total} the sum of $E_{detected}$ and $E_{undetected}$, i.e., the total number of executions with an incorrect output. The error detection rate ($E_{detectionRate}$) is the percentage of errors with incorrect output detected by the fault tolerance technique out of the total number of errors with incorrect output. It is given by the Eq. 4.3

(Eq. 4.3) $$E_{detectionRate} = \frac{E_{detected}}{E_{total}}$$

- **Fault coverage**: it is the sum of $E_{detected}$ and the number of correct executions ($X_{correct}$), divided by the total number of executions (X_{total}). The fault coverage is expressed in percentage, and it is given by Eq. 4.4. It can also be expressed as one minus $E_{undetected}$, divided by the total number of executions

(Eq. 4.4) $$F_{coverage} = \frac{E_{detected} + X_{correct}}{X_{total}} = 1 - \frac{E_{undetected}}{X_{total}}$$

- **Mean Work To Failure (MWTF) (**REIS, 2005a): the MWTF, given by Eq. 4.5, is an overall quality metric. It captures the tradeoff between reliability and performance, once that the more time an application needs to run, the higher is the probability that it is hit by a particle and, consequently, affected by a fault. Some of the parameters are the *Average Vulnerability Factor* (AVF), which is used to measure microarchitectural structure's susceptibility to transient faults (MUKHERJEE, 2003), and the *raw error rate*, which is determined by the circuit technology (MARTINEZ-ALVAREZ, 2012). The AVF can be calculated analytically, or it can be estimated statistically by fault injection campaigns. In our case, the AVF is estimated as the sum of data-flow and control-flow errors (SDCs + Hangs) out of the total of faults injected (equivalent to $1\text{-}F_{coverage}$). Furthermore, the MWTF of a hardened application is normalized by MWTF of the unhardened application, as shown in Eq. 4.6.

(Eq. 4.5) $$MWTF = \frac{amount\ of\ work\ completed}{number\ of\ errors\ encountered} = (raw\ error\ rate \times AVF \times execution\ time)^{-1}$$

(Eq. 4.6) $$MWTF_{normalized} = \frac{MWTF_{hardened}}{MWTF_{unhardened}}$$

During the evaluation, the parameters are presented per benchmark. Also, the average results are included. In order to avoid biased results, the average the data of a specific parameter of each application is divided by the highest value of that parameter for that application. Thus, the highest value for each application is always 1. Then, a harmonic mean is performed using these normalized values. Finally, in order to put the data back to the normal representation, the harmonic mean of the normalized values is multiplied

by the harmonic mean of the highest values of that parameter for each application. Eq. 4.7 shows how the mean is calculated.

$$AVG_x = \hbar armean\left(\sum(max_i)\right) \cdot \hbar armean\left(\sum\left({}^{x_i}/_{max_i}\right)\right) \quad \text{(Eq. 4.7)}$$

Where:

- x is a parameter of the evaluated SIHFT technique (or combination of SIHFT techniques)
- AVG_x is the average value of x
- x_i is the value of x for the evaluated SIHFT technique and case-study application i
- max_i is the highest value of x for case-study application i (including all SIHFT techniques)
- *harmean* is the harmonic mean.

The harmonic mean was selected instead of other averages (arithmetic or geometric mean) because the average results are not biased by extreme values. Thus, it better represents populations with outliers. In this work, the term *average* is always referring to the Eq. 4.7, except if explicitly said.

5 PROPOSED TECHNIQUES

This chapter introduces the proposed SIHFT techniques that aim at reducing overheads and keeping a similar level of fault coverage of state-of-the-art SIHFT techniques. Firstly, a set of data-flow techniques based on general building rules is presented. They evaluate the influence of checkers in the fault coverage and overheads. The aim is to reduce overheads by reducing the number of checkers without losing reliability. Then, a control-flow technique with similar fault coverage and lower overheads than state-of-the-art techniques is introduced. The aim of this control-flow technique is to complement the proposed data-flow techniques in order to protect the data-flow and the control-flow of the target application running on a COTS processor.

5.1 Data-flow techniques based on rules

Data-flow techniques are based on replicating information and verifying if the original information matches the replica. Spare registers are assigned as replicas of the used registers (for detection techniques, one spare register per used register). The replicas perform the same instructions as the original registers do. Finally, checkers are inserted in the code to compare the original register with its replica. Since the code is entirely replicated and many checkers are inserted, it is clear that the overheads introduced by data-flow techniques are high. As a first approach, what can be done to reduce such overheads is listed as follows.

- **Reducing the number of instruction duplication**: if the memory is protected by some ECC, it is possible to remove the duplication of the stores. It also removes the need to duplicate the data in memory
- **Reducing the number of checkers**: considering that the errors will probably propagate and can be detected later, it is possible to remove checkers from the hardened code.

In order to reduce the overheads caused by data-flow techniques, the proposed data-flow techniques focus on reducing duplication and number of checkers. The aim is to evaluate the trade-off between reliability and overheads and find the point that the insertion of checkers saturates the reliability and only increases the overheads.

5.1.1 Methodology and implementation

A set of rules for data-flow protection is proposed. They consist of three different types of rules: global, duplication, and checking rules, as one can see in Table 5.1. The global rule states that every register used by the application must have a spare register assigned as a replica. The global rule is implemented by all techniques. Duplication rules regard how the instructions are duplicated. They are only applicable to instructions that perform write operations in registers or memory. Therefore, branch instructions are not considered in this case.

There are two types of duplication rules: D1 and D2. Each technique must implement one and only one duplication rule. D1 duplicates all instructions, including stores, which allow the use of unhardened memories because the original value and its replica can be stored in different memory positions. D2 duplicates all instructions, except stores. The last one is adequate when the memory is hardened because the data in memory do not

need to be duplicated. Thus, the overhead caused by duplicating the code and the number of memory accesses are reduced.

Checking rules indicate when a register is compared to its replica. Thus, it is possible to verify if an error has occurred (when the register and its replica have different values). Techniques can have more than one checking rule. Theoretically, the more checkers are included in one technique, the more reliability is achieved. On the other hand, the overheads are higher. For this reason, a technique using all checking rules was not proposed. The overheads would be higher than the state-of-the-art data-flow techniques, and it would go against of this work proposal. Checking rule C1 states that a checker must be placed before a register is read by an instruction (excluding loads, stores, and branches). The checker compares the register value with the value of its replica to detect a possible error. For C2 rule, a checker is inserted right after a write operation is performed on a register. When C3 is implemented, the register that contains the address in load instructions has to be checked before the load is performed. C4 and C5 insert checkers before stores. C4 checks the register that contains the datum, and C5 checks the register that contains the address. Finally, C6 is responsible for checking the registers used by branch instructions (conditional or not).

Table 5.1: Rules for data-flow techniques

	Global Rules (valid for all techniques)
G1	every register used in the program must have a spare register assigned as replica
	Duplication Rules (perform the same operation on the register's replica)
D1	all instructions
D2	all instructions, except stores
	Checking Rules (compare the register with its replica)
C1	before every read on a register (except load/store and branch/jump instructions)
C2	after every write on a register
C3	the register that contains the address (before loads)
C4	the register that contains the datum (before stores)
C5	the register that contains the address (before stores)
C6	before branches or jumps

Based on the rules, seventeen techniques have been implemented. They are listed in Table 5.2. Each technique consists of a combination of rules. Global rule G1 and one duplicating rule (D1 or D2) are mandatory. Only one duplication rule can be used per technique. The checking rules are optional. Three techniques (VAR1, VAR2, and VAR3) belong to Azambuja (2011a), VAR4 is equivalent to EDDI, and VAR4++ is similar, but not equal, to the data part of SWIFT. The only difference is with regards to the duplication

of load instructions. VAR4++ duplicates the load instructions, and the data part of SWIFT performs a move from the original register to its copy after each load. VAR0 and VAR0+ do not implement any checking rule. Therefore, they are not capable of detecting errors. They are implemented to show the minimum overhead for each duplication rule when all the used registers are duplicated. VAR1+ and VAR1++ are variations of VAR1. They use a different duplication rule, and VAR1++ implements fewer checking rules. The same can be said about VAR2+ and VAR2++ in comparison to VAR2, and VAR3+ and VAR3++ concerning VAR3. They have a different duplication rule, and VAR2++ and VAR3++ use fewer checking rules than VAR2 and VAR3, respectively. By removing more checking rules, we get to VAR4 and VAR5, and by applying the same explanation stated above, we get techniques VAR4+, VAR4++, VAR5+, and VAR5++.

Table 5.2: Data-flow techniques and rules

Technique	Duplication Rule	Checking Rules
VAR0	D1	None
VAR0+	D2	None
VAR1	D1	C1, C3, C4, C5, C6
VAR1+	D2	C1, C3, C4, C5, C6
VAR1++	D2	C1, C3, C4, C5
VAR2	D1	C2, C4, C5, C6
VAR2+	D2	C2, C4, C5, C6
VAR2++	D2	C2, C4, C5
VAR3	D1	C3, C4, C5, C6
VAR3+	D2	C3, C4, C5, C6
VAR3++	D2	C3, C4, C5
VAR4	D1	C4, C5, C6
VAR4+	D2	C4, C5, C6
VAR4++	D2	C4, C5
VAR5	D1	C4, C6
VAR5+	D2	C4, C6
VAR5++	D2	C4

Table 5.3 exemplifies how the different techniques are applied to the unhardened code. In this regard, it was used a piece of code that permits to see the application of all rules. It consists of five instructions: two loads, one add, one store, and one branch. The original code is formatted as normal text, the duplications are in italics, and the checkers are bold. Techniques that use D1 have no plus sign in the name, and other ones, with one (+) or two (++) plus signs, use D2. Techniques that have D1 as duplication rule, such as VAR0, have all instructions that perform write operations in registers or memory (all instructions except branches) replicated to the registers replicas. Techniques using D2,

such as VAR0+, only duplicate the instructions that perform write operations in registers, i.e., all instructions but branches and stores are duplicated.

With regards to the checking rules, VAR1 and VAR1+ use almost all the checking rules, with the exception of C2. The first and the second checkers are due to C3, and the third one is due to C1. The fourth and fifth checkers are related to C4 and C5, respectively. And the sixth and the seventh are due to C6. It is important to mention that if a register is used twice by an instruction, it will only be checked once. This optimization is applied because there is no point in checking the same register twice in a row. It would only increase even more the overheads without providing more reliability. VAR1++ has the same checking rules of VAR1 and VAR1+, with the exception of C6. VAR2 and VAR2+ implement all the checking rules but C1 and C3. The first three checkers are related to C2. The fourth and the fifth are due to C4 and C5, respectively. And the last two are because of C6. VAR2++ implements the same checking rules as VAR2 and VAR2+, with the exception of C6. VAR3 and VAR3+ use C3, C4, C5, and C6. The first and second checkers are due to C3, the fourth and the fifth are due to C4 and C5, respectively, and the last two checkers are due to C6. VAR3++ implements the same checking rules of VAR3 and VAR3+, except for C6. VAR4 and VAR4+ use checking rules C4, C5, and C6, and VAR4++ uses C4 and C5. Finally, VAR5 and VAR5+ use C4 and C6, and VAR5++ uses only checking rule C4, which checks the register that contains the datum in store instructions.

Let us see an example using VAR3. Firstly, we must assign replicas to all registers used by the application. In this regard, registers $12, $13, $14, $15, and $16 are assigned as replica of registers $2, $3, $4, $5, and $6, respectively. VAR3 uses duplication rule D1. The duplications are inserted in lines 3, 7, 11, and 16 (lines in italics). And the checking rules are C3, C4, C5, and C6. Thus, checkers have to be inserted before loads, verifying the register that contains the address (C3), before stores, checking the registers that contain the datum and the address (C4 and C5), and before branches, checking the registers used by the respective branch (C6). At the first and fifth lines, there are checkers regarding C3. At lines 13 and 14, checkings are made respecting C4 and C5, respectively. C6 is applied at lines 17 and 18. Now, if we look at VAR4++, the duplication rule is D2. So all the instructions, except branches and stores are duplicated. Duplications appear at lines 3, 7, and 11 (in italics). The checking rules consist of C4 and C5. They are applied at lines 13 and 14, respectively (bold). If we compare VAR3 and VAR4++, we can see that VAR4++ clearly present a lower overhead, but at a cost of fewer checking instructions. Thus, it is important to find out if such techniques with fewer checkers can provide similar reliability than the ones with more checkers.

Table 5.3: Examples of VAR data-flow techniques for the miniMIPS processor

#	Unhardened	VAR0	VAR0+	VAR1	VAR1+	VAR1++
1				**bne $2,$12,err**	**bne $2,$12,err**	**bne $2,$12,err**
2	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)
3		*lw $14,1000($12)*	*lw $14,0($12)*	*lw $14,1000($12)*	*lw $14,0($12)*	*lw $14,0($12)*
4						
5				**bne $2,$12,err**	**bne $2,$12,err**	**bne $2,$12,err**
6	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)
7		*lw $15,1004($12)*	*lw $15,4($12)*	*lw $15,1004($12)*	*lw $15,4($12)*	*lw $15,4($12)*
8						
9				**bne $3,$13,err**	**bne $3,$13,err**	**bne $3,$13,err**
10	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1
11		*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*
12						
13				**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**
14				**bne $5,$15,err**	**bne $5,$15,err**	**bne $5,$15,err**
15	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)
16		*sw $14,1000($15)*		*sw $14, 1000($15)*		
17				**bne $3,$13,err**	**bne $3,$13,err**	
18				**bne $6,$16,err**	**bne $6,$16,err**	
19	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop

#	VAR2	VAR2+	VAR2++	VAR3	VAR3+	VAR3++
1				**bne $2,$12,err**	**bne $2,$12,err**	**bne $2,$12,err**
2	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)
3	*lw $14, 1000($12)*	*lw $14,0($12)*	*lw $14,0($12)*	*lw $14, 1000($12)*	*lw $14,0($12)*	*lw $14,0($12)*
4	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**			
5				**bne $2,$12,err**	**bne $2,$12,err**	**bne $2,$12,err**
6	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)
7	*lw $15, 1004($12)*	*lw $15,4($12)*	*lw $15,4($12)*	*lw $15, 1004($12)*	*lw $15,4($12)*	*lw $15,4($12)*
8	**bne $5,$15,err**	**bne $5,$15,err**	**bne $5,$15,err**			
9						
10	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1
11	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*
12	**bne $3,$13,err**	**bne $3,$13,err**	**bne $3,$13,err**			
13	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**
14	**bne $5,$15,err**	**bne $5,$15,err**	**bne $5,$15,err**	**bne $5,$15,err**	**bne $5,$15,err**	**bne $5,$15,err**
15	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)
16	*sw $14, 1000($15)*			*sw $14, 1000($15)*		
17	**bne $3,$13,err**	**bne $3,$13,err**		**bne $3,$13,err**	**bne $3,$13,err**	
18	**bne $6,$16,err**	**bne $6,$16,err**		**bne $6,$16,err**	**bne $6,$16,err**	
19	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop

#	VAR4	VAR4+	VAR4++	VAR5	VAR5+	VAR5++
1						
2	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)	lw $4,0($2)
3	*lw $14, 1000($12)*	*lw $14,0($12)*	*lw $14,0($12)*	*lw $14, 1000($12)*	*lw $14,0($12)*	*lw $14,0($12)*
4						
5						
6	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)	lw $5,4($2)
7	*lw $15, 1004($12)*	*lw $15,4($12)*	*lw $15,4($12)*	*lw $15, 1004($12)*	*lw $15,4($12)*	*lw $15,4($12)*
8						
9						
10	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1	add $3,$3,1
11	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*	*add $13,$13,1*
12						
13	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**	**bne $4,$14,err**
14	**bne $5,$15,err**	**bne $5,$15,err**	**bne $5,$15,err**			
15	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)	sw $4,0($5)
16	*sw $14, 1000($15)*			*sw $14, 1000($15)*		
17	**bne $3,$13,err**	**bne $3,$13,err**		**bne $3,$13,err**	**bne $3,$13,err**	
18	**bne $6,$16,err**	**bne $6,$16,err**		**bne $6,$16,err**	**bne $6,$16,err**	
19	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop	ble $3,$6,loop

5.1.2 Fault injection results in the miniMIPS processor

To evaluate how much the different techniques impact in the overheads and fault coverage, we hardened nine case-study applications with the proposed data-flow techniques, tested the overheads, and submitted them to a fault injection campaign. The case-study applications consist of a bubble sort (BS), the Dijkstra's algorithm (DA), a recursive depth-first search (rDFS), a sequential depth-first search (sDFS), a matrix multiplication (MM), the run length encoding (RLE), a summation (SUM), the TETRA encryption algorithm (TEA2), and the Tower of Hanoi (TH)[9]. A total of 162 versions was evaluated (9 unhardened and 153 hardened).

Fig. 5.1 shows the averages execution time, code size, MWTF, and fault coverage for all VAR techniques. The fault coverage is presented in percentage (right axis). The other parameters are normalized by the unhardened application (left axis). The horizontal axis identifies the data-flow technique. For example, *3*++ means that the data-flow technique VAR3++ was utilized. As one can see, the average minimum execution time is 1.32x, and the average minimum code size is 1.29x when hardening with a data-flow technique that implements duplication rule D1 (see VAR0). If D2 is used as the duplication rule, the average minimum execution time is 1.24x, and the average minimum code size is 1.23x (see VAR0+).

Techniques VAR1, VAR1+, VAR1++, VAR2, VAR2+, and VAR2++, present a high fault coverage for data-flow techniques, but very high overheads. Similar fault coverages can be obtained by techniques VAR3 and VAR3+ with the advantage of considerably lower overheads. It shows that after a certain point, checking instructions get saturated. Considering only the baseline techniques, VAR3 presents the best results since it has a similar fault coverage to VAR1 and VAR2 and lower overheads. VAR4 has lower overheads than VAR3, but it does not achieve the fault coverage. By changing the VAR3's duplication rule from D1 to D2, we get VAR3+. This change reduces the average execution time from 1.85x to 1.77x, and the average code size from 1.74x to 1.68x, and it keeps a similar fault coverage rate to VAR3. Comparing VAR4++ to VAR3, we can see a significant reduction in the overheads. The execution time went from 1.85x to 1.42x, and the code size dropped from 1.77x to 1.48x, with a loss of around 2% in the fault coverage. Although the lower fault coverage, VAR4++ can be a better solution when constraints are more restrictive or when using the technique combined with a control-flow technique. It is important to notice that the unhardened applications have nonnegligible fault coverages due to the masked faults and fault injection methodology. Actually, their fault coverage ranges from 81% to 89% depending on the application. The contribution of fault tolerance techniques is better evaluated by looking at how much the fault tolerance technique reduced the distance to 100% of fault coverage. This is one of the MWTF's parameters.

[9] More information about the case-study applications is available in appendix C.

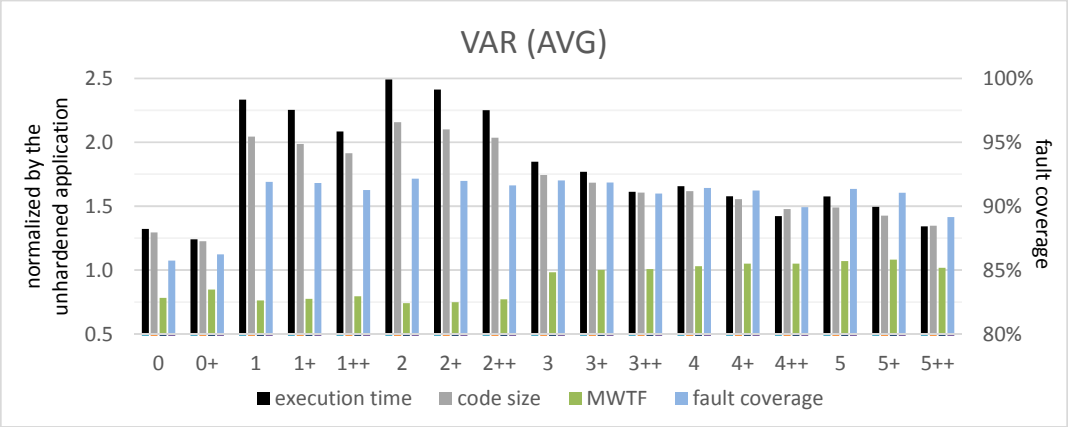


Fig. 5.1: Average results for the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened applications (left axis). The fault coverage is presented in percentage (right axis).

Fig 5.2 presents the execution time, code size, MWTF, and fault coverage for the bubble sort (BS). The duplication rule D1 increases the execution time to 1.33x and the code size to 1.37x (VAR0), while duplication rule D2 increases the execution time to 1.25x and the code size to 1.30x (VAR0+). Considering only the VAR techniques that implement some checking rule (VAR1 to VAR5++), the execution time, code size, and fault coverage range, respectively, from 1.36x to 2.62x, from 1.40x to 2.58x, and from 90.7% to 92.6%, which is similar to the average results.

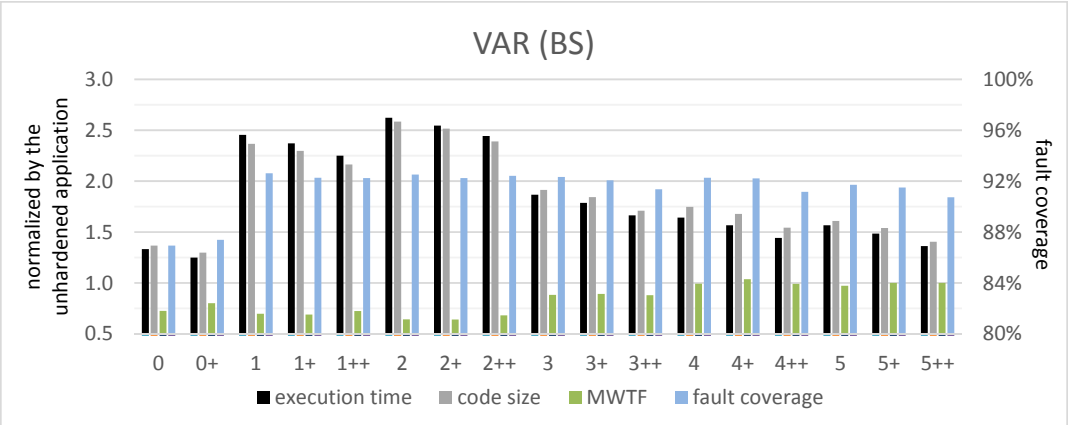


Fig. 5.2: Results for the bubble sort (BS) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 5.3 shows the execution time, code size, MWTF, and fault coverage for the Dijkstra's algorithm (DA) when protected by VAR. The execution time ranges from 1.28x to 2.52x, the code size goes from 1.46x to 2.69x, and the fault coverage ranges from 90.5% to 92.7%, when checking rules are implemented. Duplication rules D1 and D2 cause a respective execution time of 1.24x and 1.16x, and a code size of 1.38x and 1.28x.

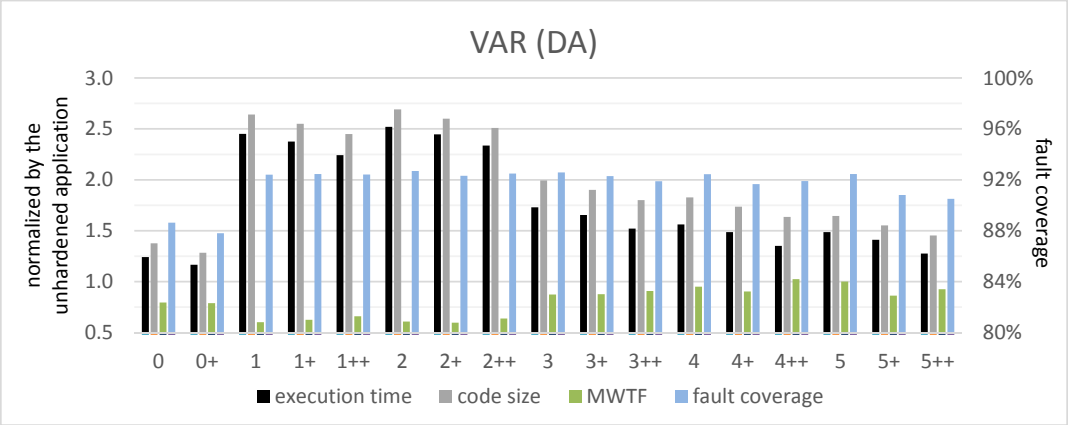


Fig. 5.3: Results for the Dijkstra's algorithm (DA) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The recursive depth-first search presents a high fault coverage when protected by VAR technique that implements checking rule C6, as shown in Fig. 5.4. On the other hand, when this rule is not applied (VAR1++, VAR2++, VAR3++, VAR4++, and VAR5++), the application suffers a significant drop in the fault coverage. In recursive applications, the number of errors causing *wrong, but legal branches* is higher due to many condition tests and returns from subroutines. An error affecting a register used by such instructions may not be detected if C6 is not implemented. The drop in fault coverage for VAR1++, VAR2++, and VAR3++ is lower than VAR4++ and VAR5++ because the first ones implement more checking rules, which increases the probability of checking the registers used by branches or returns in other points of the code. The fault coverage can reach more than 92%. The execution time and code size range from 1.26x to 2.16x, and from 1.08x to 1.25x, respectively, when checking rules are implemented. VAR0 and VAR0+ have an execution time of 1.26x and 1.20x and a code size of 1.08x and 1.07, respectively.

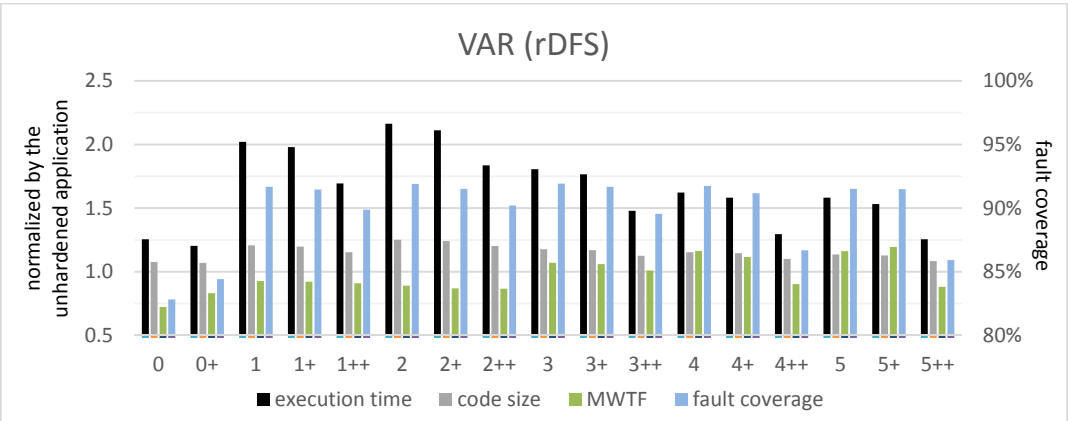


Fig. 5.4: Results for the recursive depth-first search (rDFS) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 5.5 presents the execution time, code size, MWTF, and fault coverage for the sequential depth-first search (sDFS). Although a lower execution time, code size, and fault coverage, the behavior of these parameters is similar to rDFS. Furthermore, sDFS

has a higher percentage of branch instructions[10], which explains why techniques that do not implement checking rule C6 present lower fault coverage. Considering all the techniques, the fault coverage can reach up to almost 91.6%. The execution time and code size of techniques implementing checking rules range from 1.18x to 2.03x, and from 1.05x to 1.17x, respectively. For this application, duplication rules D1 and D2 have similar execution time and code size (due to few number of store instructions), which are 1.18x and 1.05x, respectively.

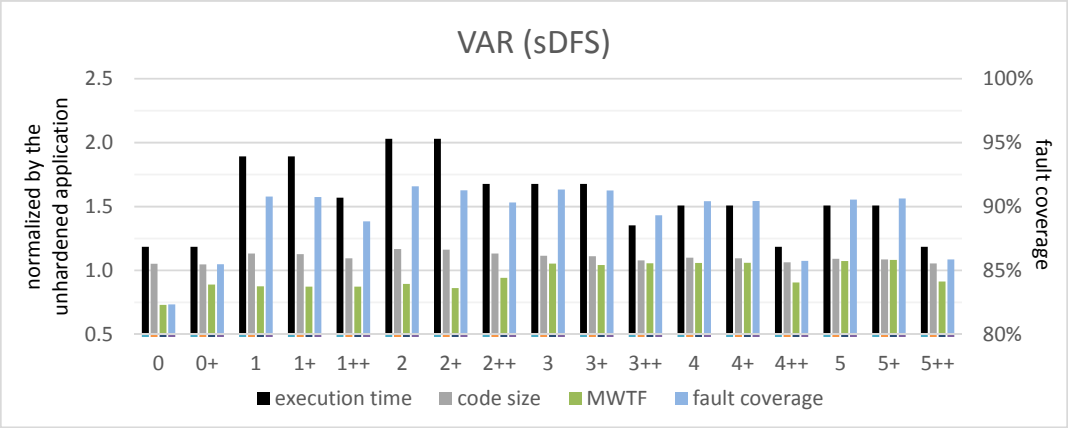


Fig. 5.5: Results for the sequential depth-first search (sDFS) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The matrix multiplication (MM) has data processing and load/store instructions. The VAR techniques that check both load and store instructions are VAR1 to VAR3++. As a result, they provide a higher fault coverage for this application, as one can see in Fig. 5.6. The execution time, code size, and fault coverage of VAR techniques that implement checking rules range, respectively, from 1.30x to 2.61x, from 1.42x to 2.63x, and from 87.7% to more than 92.5%. VAR0 and VAR0+ present an execution time of 1.29x and 1.24, and a code size of 1.40x and 1.31x, respectively.

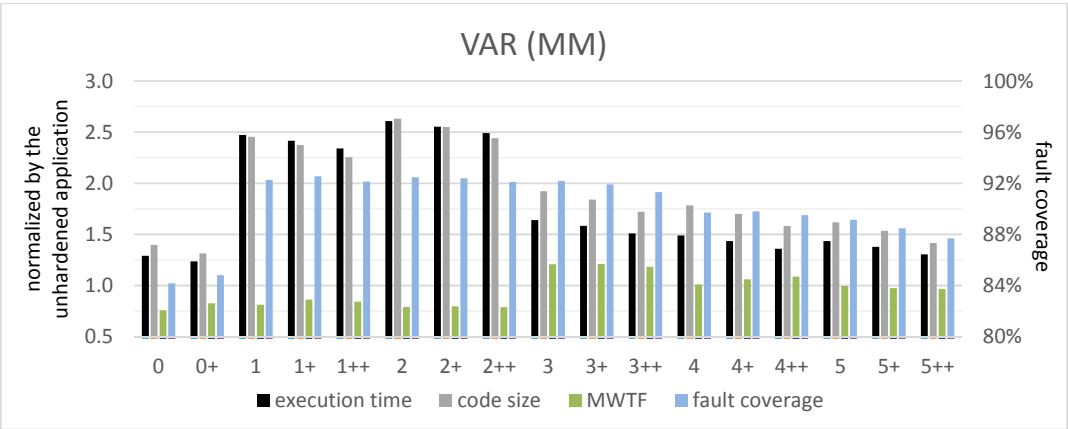


Fig. 5.6: Results for the matrix multiplication (MM) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

[10] Further information about the benchmarks is available in the appendix C.

Fig. 5.7 shows the results for the run-length encoding (RLE). RLE has the greatest number of instructions, which explains the higher code size overhead. The execution time varies from 1.29x to 2.44x, the code size ranges from 1.49x to 2.99x, and the fault coverage goes from 90.2% to almost 92%, when checking rules are implemented. The minimum execution time and code size when using duplication rule D1 are 1.27x and 1.35x, respectively. When using D2, the minimum execution time is 1.19x, and the minimum code size is 1.24x.

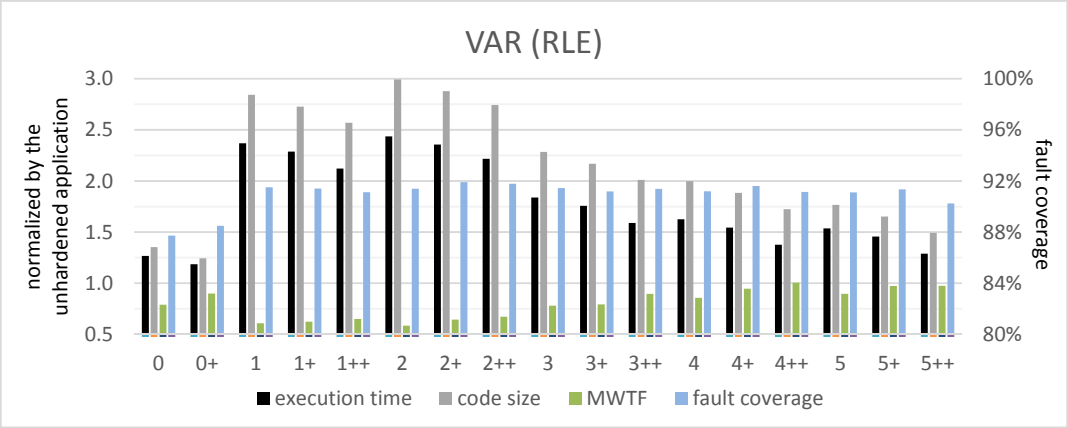


Fig. 5.7: Results for the run length encoding (RLE) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Summation (SUM) is a small and heavily loop-based application, and the single output depends on the entire program execution. Thus, an error affecting the program at any time will very likely propagate to the output, which consists of a store instruction that is checked by all techniques implementing checkers. That increases the chance of detecting an error, which is shown by the high fault coverage (from 90.8% to 92.3%) presented in Fig. 5.8. The execution time and code size range, respectively, from 1.40x to 2.47x, and from 1.15x to 1.55x, when implementing checking rules. VAR0 and VAR0+ present execution time of 1.33x and 1.20x, and code size of 1.14x and 1.10x, respectively.

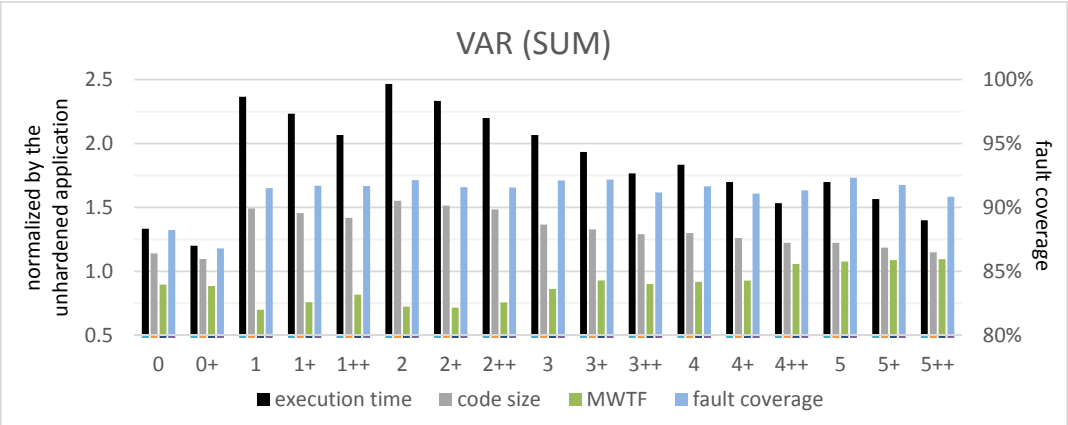


Fig. 5.8: Results for the summation (SUM) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The TETRA encryption algorithm has many load/store instructions. Once the store instructions are hardened by all VAR techniques implementing checking rules, the

probability of detecting an error increases. In addition, there are very few branches or jumps that rely on registers to get the target address. It corroborates to explain why the techniques present similar fault coverages (from 91.8% to more than 92.6%), as one can see in Fig. 5.9. The execution time ranges from 1.42x to 2.61x, and the code size varies from 1.41x to 2.43x when implementing checking rules. Duplication rule D1 causes an execution time of 1.38x and code size of 1.29x, and D2 causes an execution time of 1.29x and code size of 1.20x.

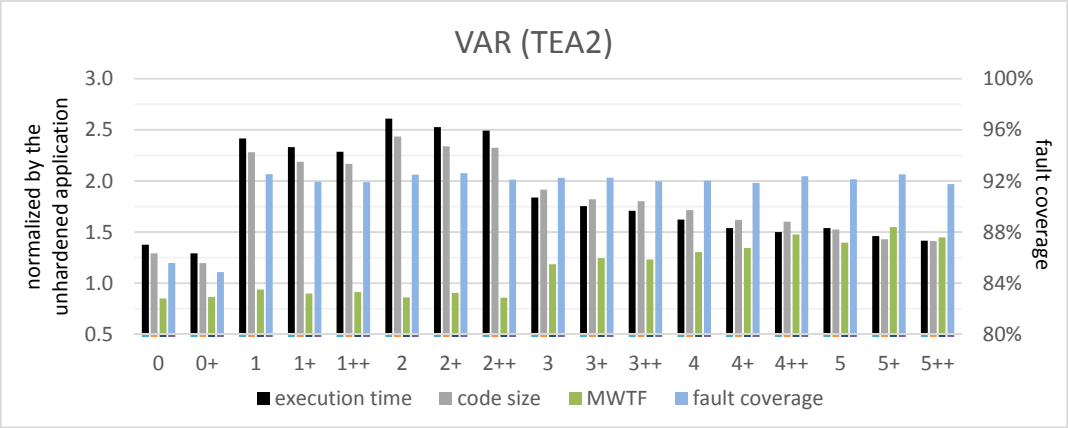


Fig. 5.9: Results for the TETRA encryption algorithm (TEA2) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The Tower of Hanoi has just a few conditional branches. On the other hand, it has many subroutine calls, and consequently many returns. If checking rule C6 is not implemented, the return register will not be checked, and an error affecting this register will not be detected. That explains why techniques that do not implement C6 have lower fault coverage, which goes from 88% to more than 92%. The execution time and code size of VAR techniques implementing checking rules range from 1.58x to 2.96x, and from 1.38x to 2.10x, respectively, as shown in Fig. 5.10.

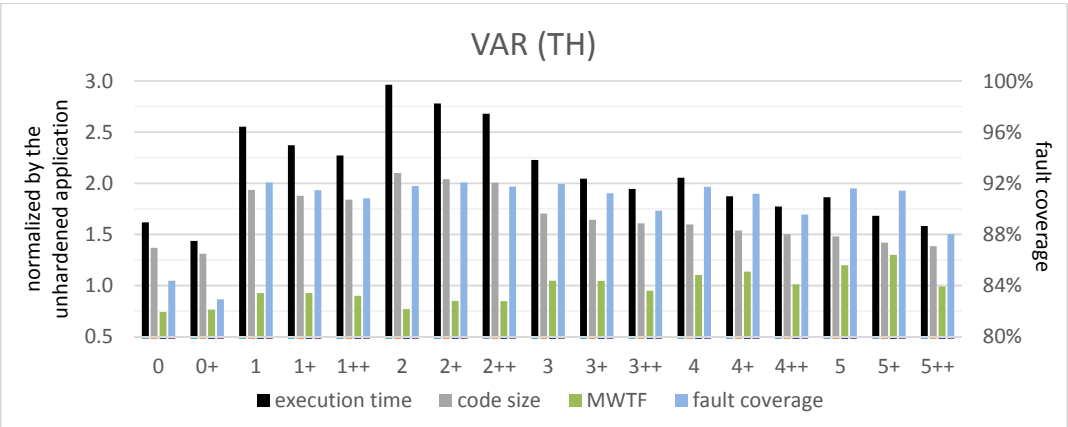


Fig. 5.10: Results for the Tower of Hanoi (TH) hardened by the VAR techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The memory accesses caused by load or store instructions are presented in Fig. 5.11. All the techniques that implement duplication rule D1 present twice the memory accesses of the unhardened application because they duplicate all load and store instructions. On

the other hand, when duplication rule D2 is used, only the load instructions are duplicate. Thus, the number of memory accesses can be define as twice the number of loads, plus the number of stores of the unhardened code. So the relation between the number of memory accesses of techniques using D2 and the number of memory accesses of the corresponding unhardened application depends on the application. Anyhow, it will always range from the same number of memory accesses of the unhardened application (when there are no load instructions) to twice the number of memory accesses (when there are loads, but there are no store instructions).

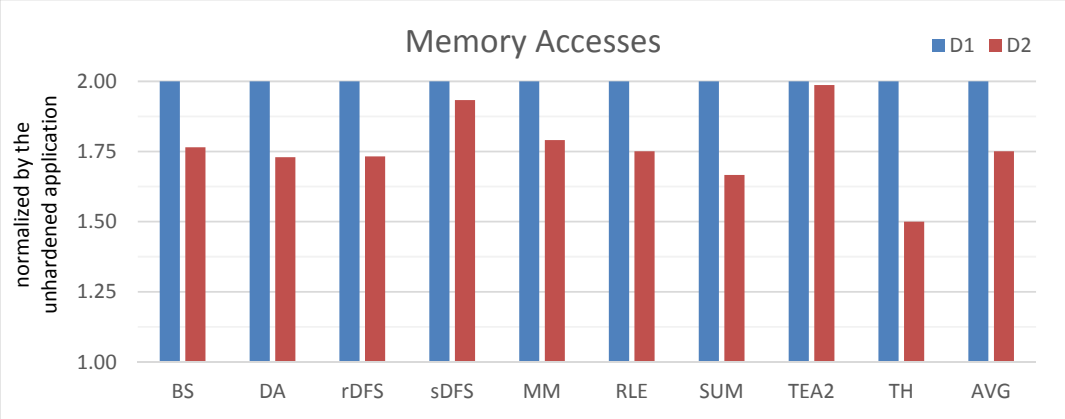


Fig. 5.11: Memory accesses for duplication rules D1 and D2 for the nine case-study applications and average (harmonic mean).

SIHFT techniques are cheaper than hardware-based ones, but they present performance and memory overheads. The proposed set of rules and VAR techniques evaluated the execution time, code size, MWTF, and fault coverage. As one can see, it was possible to reduce the overheads and keep similar fault coverage. For loop-based applications, the implementation of checking rule C6 is very implement to improve the fault coverage.[11] Based on the results, the following items were enumerated as the most points where checkers are important:

1. **Before stores (checking rules C4 and C5)**: store instructions are the final point before sending the results to the memory. The protection of the registers used by store instructions is fundamental to increase the fault coverage
2. **Before branches or jumps (checking rule C6)**: checking registers used by branches or jumps increases the fault coverage, mainly in applications in which the average number of executions of the basic blocks is high
3. **Before loads (checking rule C3)**: it was observed that checking the register used as address by loads increases, in general, a little the fault coverage. Thus, checking rule C3 could be not necessary. However, in applications that make significant use of sequences of load/stores in which there is a high data dependency, i.e., the following calculations depends on the previous calculations (which is the case of the matrix multiplication), checking rule C3 increases significantly the fault coverage. This rule could be replaced by

[11] It is possible to find the loop-based applications by the relation between the average number of times a BB is executed (Table C.4) and the execution time (Table C.5).

a rule checking the data register after the load, but it would cause a higher execution time due to data dependency in the pipeline.

The other rules increased a lot the overheads and did not increase the fault coverage when compared to applications hardened by rules C3 to C6. Furthermore, it is possible to notice that the MWTF is lower, or not much higher, than unhardened application in many cases. This is explained by the high overheads imposed by VAR techniques (mainly VAR1 to VAR2++), and because they only protect the data-flow[12]. Thus, the control-flow is left uncovered, and errors affecting it are not detected. Therefore, it may be a better option to leave a code unhardened than to protect only with a data-flow technique. However, if a control-flow is applied together with a data-flow technique, both data-flow and control-flow will be protected, and that may increase the application's MWTF. The use of data-flow techniques together with control-flow techniques is evaluated in section 5.3.

5.2 Control-flow technique

In order to complement the data-flow techniques and promote a protection of both data-flow and control-flow, we introduce a technique called SETA (*Software-only Error-detection Technique using Assertions*) to detect control-flow errors in processors with no modification or addition of extra hardware. The penalties in performance and memory caused by SETA are lower than other control-flow techniques. SETA is based on HETA and CEDA. These techniques use runtime signatures to detect errors affecting the control-flow of a running application. HETA uses an extra signature, which increases the overheads. Also, it makes use of a watchdog to help in the detection, which requires extra power. And, as the author stated, the watchdog needs access to the memory buses. Some processors that use on-chip embedded cache memories may not be accessible by the watchdog, which makes impossible to implement this technique in the target ARM processor. Furthermore, both CEDA and HETA are concerned about the error detection rate they achieve, but not about the overheads they cause. Aiming at providing similar error detection rate as CEDA with lower overheads, SETA is proposed. The technique uses signatures calculated a priori and processed during runtime. The program code is divided into basic blocks (BB), and signatures are assigned to the basic blocks.

5.2.1 Methodology and implementation

Two Basic Block Types (BBT) are defined: A and X. A basic block is of type A if it has multiple predecessors, and at least one of its predecessors has multiple successors. And it is of type X if it is not of type A. Then, the basic blocks are grouped into networks. Basic blocks sharing a common predecessor belong to the same network. An example is shown in Fig. 5.12.

[12] The next chapter presents an analysis trying to reduce even more such overheads caused by data-flow techniques by selectively hardening the registers used by the application.

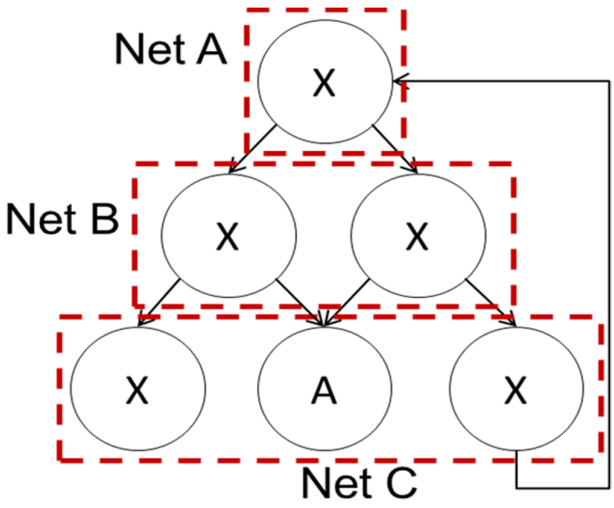


Fig. 5.12: Representation of a program flow. Basic blocks (circles) classified as of type A or X, and grouped into networks (dashed rectangles). The arrows indicate the valid directions that a basic block can take.

Every basic block has two different signatures: a *Node Ingress Signature* (NIS), for when entering the basic block, and a *Node Exit Signature* (NES), for when exiting the basic block. The NIS represents the current basic blocks, and the NES is used to identify the successor network and the valid successor basic blocks.

The signatures are divided in two parts: an upper half and a lower half, as shown in Table 5.4. The upper half identifies the network, and the lower half identifies the basic block. The NIS' upper half identifies the network that the basic block belongs to. The lower half has a random number assigned if the BB is of type X. If the BB is of type A, the lower half is calculated by the AND operation of the lower halves of all predecessor BBs' NES. The NES' upper half identifies the successor network, and the lower half has a random number. Table 5.5 summarizes it. If a BB of type X has multiple predecessors, all its predecessors must have the same NES. The size of these "halves" is, actually, variable per application in order to maximize the basic block identifier (lower half) and, thus, avoid aliasing. The upper half receives the minimum number of bits it needs to represent all the networks, i.e., the first integer greater or equal to $log_2(N+1)$, where N is the total number of networks. Let us define it as $ceil(log_2(N+1))$. The remaining bits are used by the lower half. The networks are sequentially identified, from 0 to N-1. The identifier N is reserved for what we call *ghost network*. It is used as successor network of the basic blocks that have no successors. Thus, it invalidates any transition (caused by a fault) from such BBs to another BB.

Table 5.4: Signature division.

Upper half	Lower half
01000100	01001011110101001011110 1

↔ variable per application

Table 5.5: Role of each half in the signatures

Signature	Upper half	Lower half
NIS	Identifies the BB's network	BB's signature 1
NES	Identifies the successor network	BB's signature 2

At runtime, a signature register *S* is updated according to the conditions presented in Table 5.6 to keep track of the program execution. The operation to update *S* can be an XOR or an AND. It is performed with *S* and an invariant value, as shown by Eqs. 5.1 and 5.2.

Table 5.6: Signature update

BB Type	NIS	NES
A	AND	XOR
X	XOR	XOR

(Eq. 5.1) $S \leftarrow S \text{ XOR invariant}$

(Eq. 5.2) $S \leftarrow S \text{ AND invariant}$

The invariant is a constant that will make *S* have the expected signature during a correct execution. Its calculation is described as follows. From NIS to NES (inside a BB), the NES invariant to update *S* depends only on the BB's signatures. The invariant is the result of an XOR operation of the BB's NIS and NES. From NES to NIS (BBs transition), the NIS invariant relies on the predecessor BBs' NES, and on the NIS and type of the current BB. If the BB is of type X, there are two possible ways to calculate the invariant:

- **The BB has no predecessors (starting BB)**: in this case, the NIS invariant is equal to the BB's NIS
- **The BB has predecessors**: the NIS invariant is the result of an XOR operation with any predecessor NES and the BB's NIS.

If the BB is of type A, the NIS invariant is divided into upper and lower half (like the signatures) for its calculation. The upper half is filled with ones (the equivalent in unsigned integer is $2^{ceil(log_2(N+1))}-1$). The lower half of the NIS invariant is equal to the lower half of the BB's NIS. The classification of basic blocks into types and networks ensures that there will not be invalid transitions, except for the following case: the starting BB has itself as successor. Consequently, it is also its predecessor. In this case, a constant is loaded to *S* at the beginning of the BB to keep the execution consistent.

Checkers are inserted in the basic blocks to verify if *S* contains the expected signature for that basic block. The more checkers, the lower is the latency to detect errors. On the other hand, the higher is the overhead. The maximum number of checkers in SETA matches the number of basic blocks since only one checker is needed per basic block. Table 5.7 shows an example of SETA for the miniMIPS processor. An unhardened portion of code is shown in the left side, and, in the right side, there is the same code protected by SETA. The instructions inserted by SETA are in italics (signature updates) or bold (checkers). The first XOR (*xori*) is to update the signature to the basic block's NIS. The instructions *li* and *bne* are used to compare the signature register *$7* with the expected signature for that basic block. Finally, the last XOR is used to update the

signature to the expected NES. Since new instructions are inserted, it is clear that the execution time and the code size will increase.

Table 5.7: Example of SETA control-flow technique for the miniMIPS processor

#	Unhardened code		Code hardened by SETA	
1	jal	dfs	jal	dfs
2			*xori*	*$7,$7,41407*
3	la	$2,$result	la	$2,$result
4	lw	$4,0($6)	lw	$4,0($6)
5	sw	$6,4($2)	sw	$6,4($2)
6	sw	$4,0($2)	sw	$4,0($2)
7			**li**	**$8,41407**
8			**bne**	**$7,$8,error**
9			*xori*	*$7,$7,29184*
10	j	loop	j	loop

The main differences from CEDA to SETA are:

- Removed inverted branches check. CEDA inserts branches at both possible targets of each branch to check it was taken correctly. SETA does not implement it because the fault coverage it provides is negligible compared to the overheads it causes. It only detects errors affecting the decision of a branch when the registers and the comparison are correct, but the branch takes the wrong direction.
- Removed extra instructions used to avoid aliasing. SETA does not need to insert instructions to "clear" the signature, as it is done in CEDA because the signature values are assigned in a different way. The upper half is deterministic, and the lower half is randomly determined. Thus, the signature register can always be directly updated, which reduces the overheads. SETA avoid aliasing by varying the size of the "halves", trying to maximize the size of the lower half.

5.2.2 Fault injection results in the miniMIPS processor

Firstly, we compared SETA with CEDA. Fig. 5.10 shows the execution time, code size, MWTF, and fault coverage of both techniques for all benchmarks. The average (AVG) is also included. The execution time, code size, and MWTF are presented normalized by the equivalent unhardened application (left axis). The fault coverage is expressed in percentage (right axis). The results show that both techniques present fault coverage around of 94% on average.

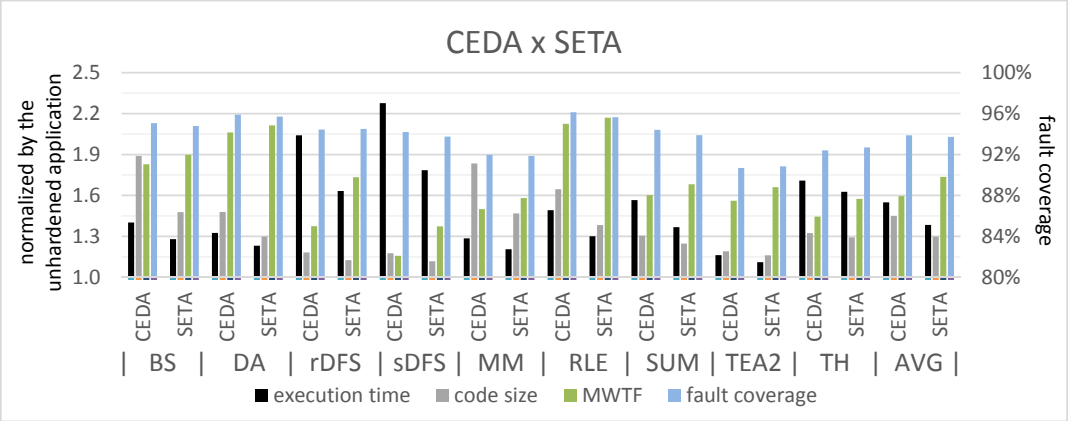


Fig. 5.10: Comparison between CEDA and proposed SETA techniques. The execution time, code size, and MWTF are presented normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

In all case-study applications, one can notice an improvement of the MWTF from CEDA to SETA, as shown in Fig. 5.11. It is due to the reduction of the execution time. The sequential depth-first search (sDFS) and the recursive depth-first search have very high overheads when protected by control-flow techniques. These applications have small basic blocks that are executed many times, which makes the addition of signature updates and checkers more noticeable in the execution time. The inverted branches check makes CEDA way more costly in such cases.

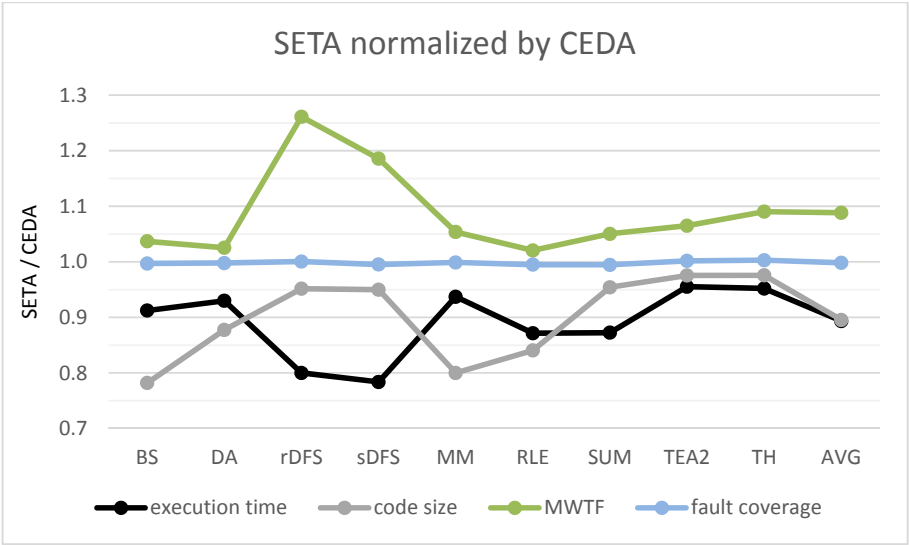


Fig. 5.11: Comparison between SETA and CEDA. The results obtained with SETA are normalized by the ones obtained with CEDA.

Fig. 5.12 shows the average results for each technique. The execution time, code size, and MWTF are presented normalized by the unhardened applications (left axis). Fault coverage and error detection rates are showed in percentage (right axis). The horizontal axis presents the techniques. When CEDA and SETA have one checker per basic block, the fault coverage and error detection of both techniques are similar. One can notice that even the techniques have been designed to detect control-flow errors, they are capable of detecting more than half of the data-flow errors. The advantages of SETA are due to its

reduced overheads. SETA is 11.0% faster and occupies 10.3% fewer memory positions than CEDA. Once SETA runs faster than CEDA, the application protected by SETA has a lower chance of being hit by an energized particle that causes a bit flip. And since both have similar fault coverage, SETA is more reliable than CEDA. That is reason why while SETA's MWTF is 1.74x, while CEDA's is 1.60x.

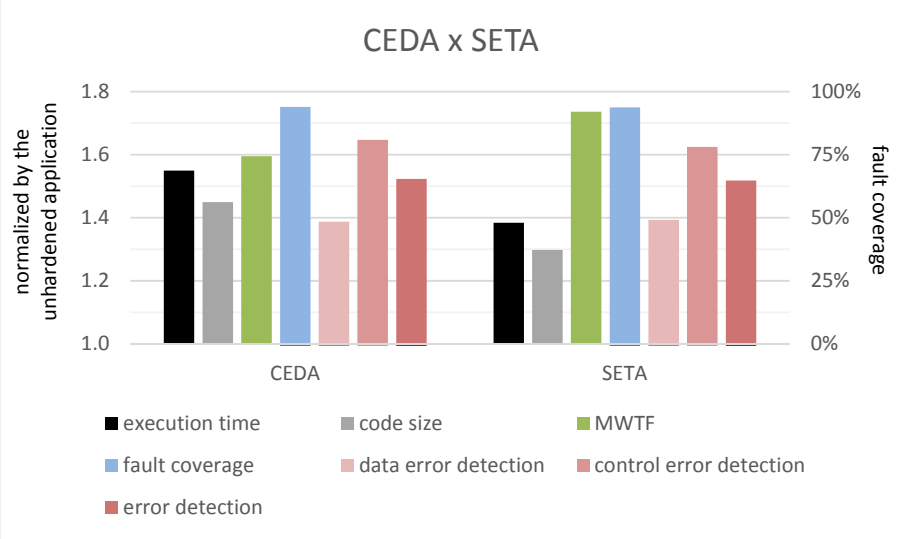


Fig. 5.12: Comparison between CEDA and SETA. The average results are presented. The execution time, code size, and MWTF are normalized by the unhardened applications (left axis). The fault coverage and error detection rates are presented in percentage (right axis).

5.3 Combined data-flow and control-flow techniques

Once the aim is to protect both the data-flow and the control-flow of a running application, SETA was combined with some of the proposed data-flow VAR techniques (3, 3+, 3++, 4, 4+, 4++, 5, 5+, and 5++). VAR0 and VAR0+ are not evaluated in this section because they do not detect errors. They are just used to evaluate the overhead that the duplication rules cause. The remaining VAR techniques (1, 1+, 1++, 2, 2+, and 2++) are also not included in this section because they have similar data error detection rate to VAR3 and VAR3+, but higher overheads. All the case-study applications are evaluated considering all the combinations of a VAR technique and SETA in terms of execution time, code size, MWTF, and fault coverage. As well as the previous sections, the fault coverage is provided by fault injection simulation. Nevertheless, some selected cases are also submitted to radiation experimements in order to validate the simulated fault injection.

5.3.1 Methodology and implementation

The code is hardened by one VAR technique and SETA. Firstly, a VAR technique is applied to the unhardened code, and then SETA is applied to this code hardened by VAR. The VAR techniques are implemented as discussed in 5.1.1, and SETA is implemented as explained in 5.2.1. The only consideration is with regards to the checkers inserted by VAR when applying SETA. SETA counts the checkers as instructions, but they are not considered ends of basic blocks. Thus, the unhardened code and the code hardened by VAR have the same basic blocks division, and SETA is applied evenly to both codes, as

one can see in Table 5.8. The table presents four versions of a code, an unhardened and three others hardened by SETA, VAR3, and VAR3 and SETA. The original code is formatted as normal text, the code inserted by VAR3 is shown in italics, and the code inserted by SETA is bold. We can notice that SETA ignores the checkers inserted by VAR3 when dividing the code into basic blocks. Thus, the signature updates and checkers are inserted as in the unhardened code.

Table 5.8: Example of VAR3 and SETA techniques for the miniMIPS processor

#	Unhardened code	Code hardened by SETA	Code hardened by VAR3	Code hardened by VAR3 and SETA
1	jal dfs	jal dfs	jal dfs	jal dfs
2		**xori $7,$7,41407**		**xori $7,$7,41407**
3	la $2,$result	la $2,$result	la $2,$result	la $2,$result
4			*la $12,$result*	*la $12,$result*
5			*bne $6,$16,error*	*bne $6,$16,error*
6	lw $4,0($6)	lw $4,0($6)	lw $4,0($6)	lw $4,0($6)
7			*lw $14,1000($16)*	*lw $14,1000($16)*
8			*bne $6,$16,error*	*bne $6,$16,error*
9			*bne $2,$12,error*	*bne $2,$12,error*
10	sw $6,4($2)	sw $6,4($2)	sw $6,4($2)	sw $6,4($2)
11			*sw $16,1004($12)*	*sw $16,1004($12)*
12			*bne $4,$14,error*	*bne $4,$14,error*
13			*bne $2,$12,error*	*bne $2,$12,error*
14	sw $4,0($2)	sw $4,0($2)	sw $4,0($2)	sw $4,0($2)
15			*sw $14,1000($12)*	*sw $14,1000($12)*
16		**li $8,41407**		**li $8,41407**
17		**bne $7,$8,error**		**bne $7,$8,error**
18		**xori $7,$7,29184**		**xori $7,$7,29184**
19	j loop	j loop	j loop	j loop

If SETA considered the VAR checkers as ends of basic blocks, the code hardened by VAR would have a considerably higher number of basic blocks than the unhardened code, which would result in a significant increase in the overheads in execution time and code size when SETA is applied.

5.3.2 Fault injection results in the miniMIPS processor

Fig. 5.13 presents the average results (AVG) for combining the VAR techniques with SETA. The execution time, code size, and MWTF are expressed normalized by the corresponding unhardened application (left axis). The fault coverage is presented in percentage (right axis). The horizontal axis identifies the data-flow technique. For example, *3++* means that the data-flow technique VAR3++ and the control-flow

technique SETA have been applied. We can see that the overheads in performance and memory introduced by the data-flow techniques for the target applications and processor are higher than the ones presented by the control-flow techniques. It is justified by the insertion of redundancy and checkers in the entire code, instruction by instruction, and not by dividing into basic blocks. The execution time ranges from 1.74x to 2.20x, and the code size ranges from 1.68x a 1.95x. However, one can notice an increase of up to 5.19x in the MWTF when VAR3+ and SETA are applied. All the data-flow techniques, when combined with SETA, present a significant increase in the MWTF. It is clear from the chart that techniques VAR3++, VAR4++, and VAR5++ have inferior MWTF. These three techniques share a common feature, they do not implement checking rule C6. This rule states that registers must be checked before they are used by branches or jumps. All the other VAR techniques implement C6, and they have higher MWTF. Therefore, checking registers before they are used by branches or jumps is important to provide reliability to the application. An error affecting a register used by a branch would not be detected by either SETA or CEDA, because it would be a valid basic block transition. The *inverted branches check* implemented by CEDA is also incapable of detecting such errors because the redundant branch would have the same decision of the original one. It was observed in Chielle (2016) that almost the totality errors causing *wrong, but legal branches* are due to an incorrect value in a register used during the branch comparison.

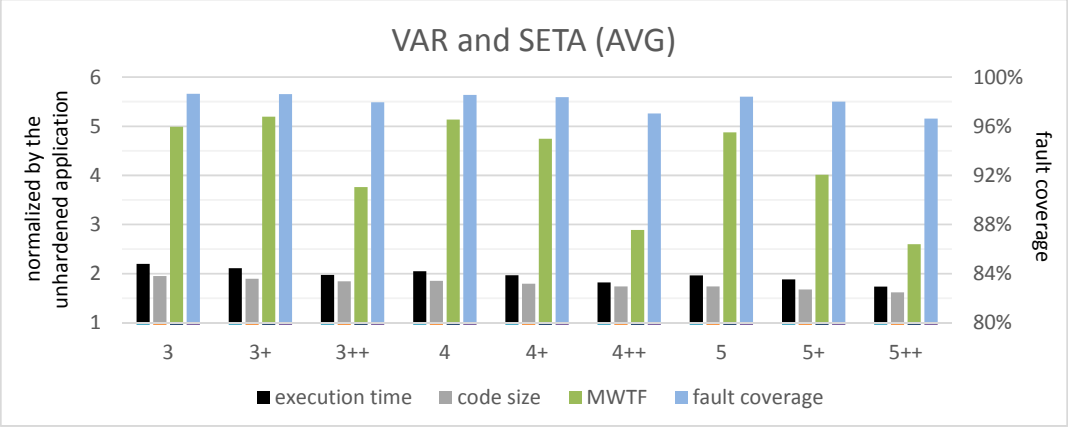


Fig. 5.13: Average results for combining VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

In Fig. 5.14, the results for the bubble sort (BS) are presented. They are very similar to the average results. Therefore, the same conclusions from the average results can be applied to the BS. The execution time ranges from 1.65x to 2.20x, the code size goes from 1.89x to 2.31x, and the MWTF reaches up to 5.45x when VAR4+ is selected.

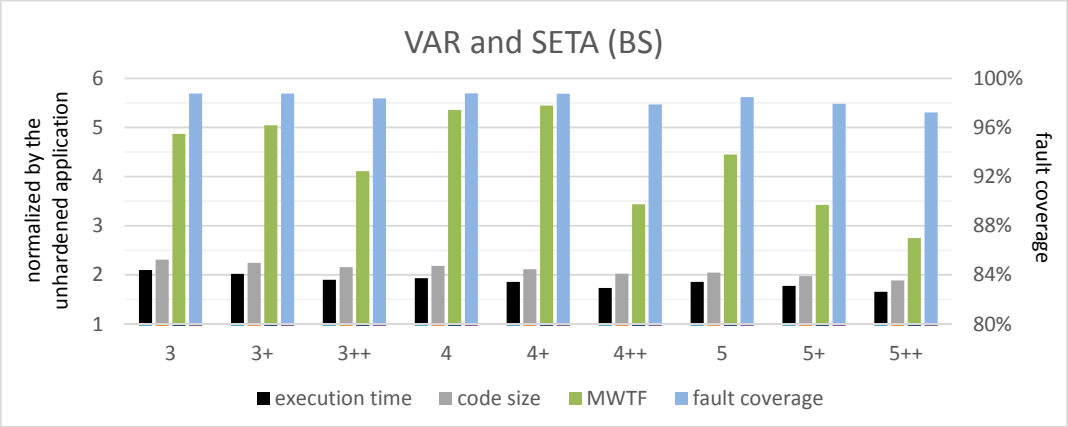


Fig. 5.14: Results of combining VAR and SETA for the bubble sort (BS). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 5.15 shows the execution time, code size, MWTF, and fault coverage for the Dijkstra's algorithm (DA) when protected by VAR and SETA. The execution time ranges from 1.52x to 1.93x, the code size goes from 1.76x to 2.25x, and the MWTF reaches up to 6.39x when VAR3++ is selected.

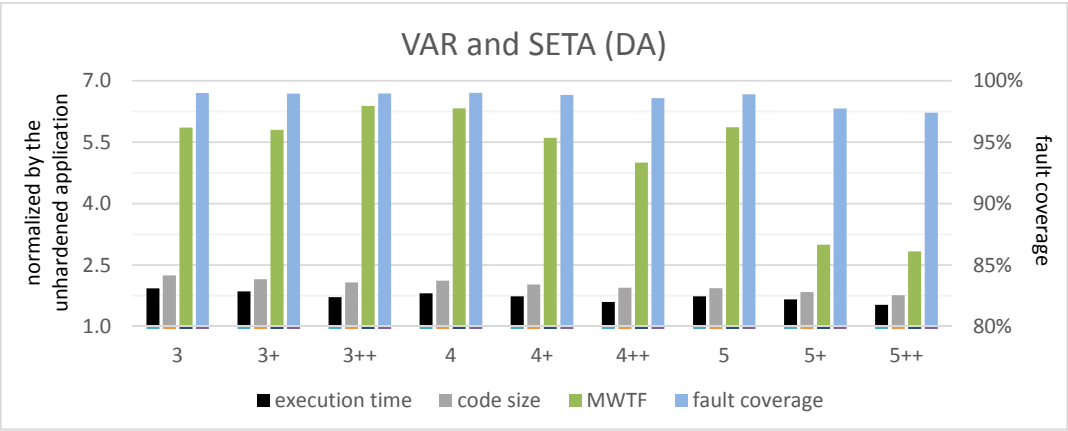


Fig. 5.15: Results of combining VAR and SETA for the Dijkstra's algorithm (DA). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The recursive depth-first search presents high fault coverage when protected by SETA and a VAR technique that implements checking rule C6, as shown in Fig. 5.16. On the other hand, when this rule is not applied, the application suffers a considerable drop in the fault coverage, which can be seen by the lower MWTF presented by VAR3++, VAR4++, and VAR5++ (3.28x, 1.74x, and 1.78x, respectively). In recursive applications, the number of errors causing *wrong, but legal branches* is very high due to many condition tests and returns from subroutines. An error affecting a register used by such instructions is not detected if C6 is not implemented. The highest MWTF is 5.85x, achieved when VAR4+ and SETA are applied. The execution time and code size range from 1.88x to 2.37x, and from 1.21x to 1.29x, respectively.

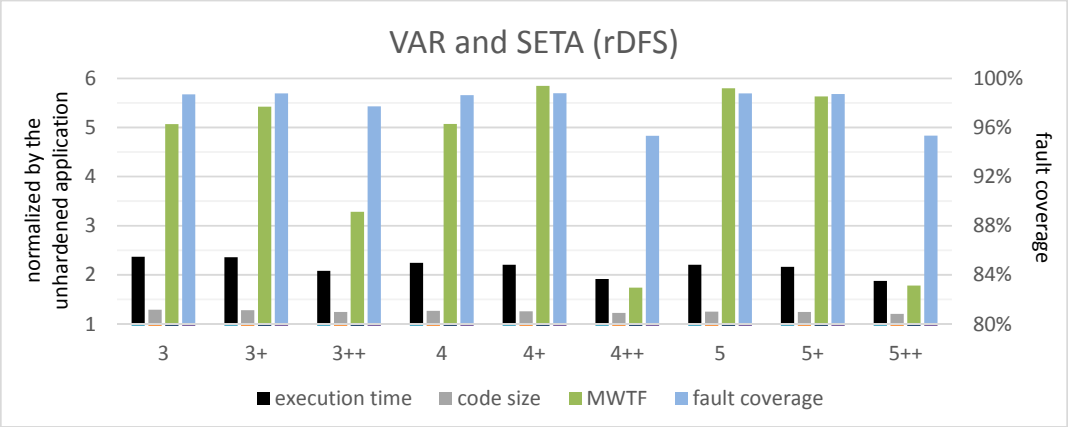


Fig. 5.16: Results of combining VAR and SETA for the recursive depth-first search (rDFS). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 5.17 presents the execution time, code size, MWTF, and fault coverage for the sequential depth-first search (sDFS). The MWTF is inferior to rDFS, mainly due to the higher execution time overhead. And although sDFS has a similar percentage of arithmetic, load/store, and branch instructions to rDFS, the percentage of branches is a little higher[13]. It explains why techniques that do not implement checking rule C6 (VAR3++, VAR4++, and VAR5++) present lower MWTF. The highest MWTF for this application is achieved by VAR5 and SETA (4.11x), and the execution time and code size range from 2.03x to 2.46x, and from 1.17x to 1.22x, respectively.

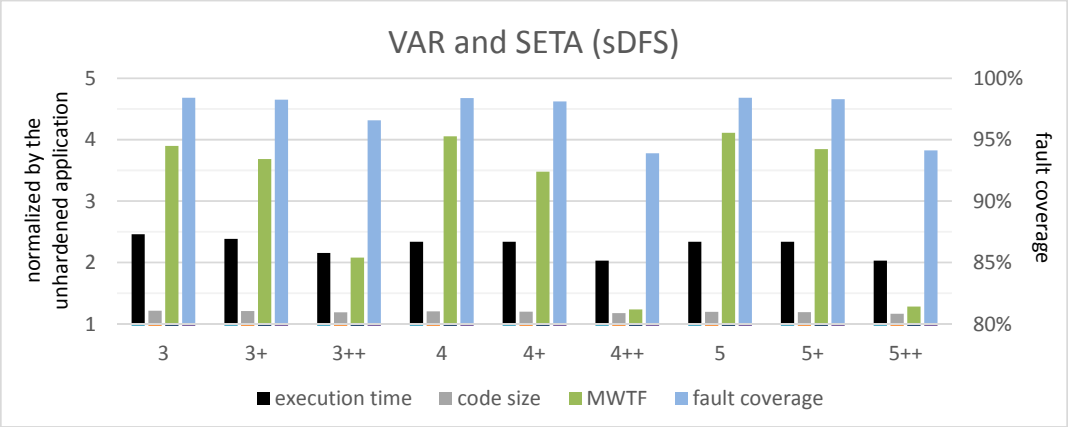


Fig. 5.17: Results of combining VAR and SETA for the sequential depth-first search (sDFS). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The matrix multiplication (MM) has data processing and load/store instructions. The only VAR techniques that check both load and store instructions are VAR3, VAR3+, and VAR3++. As a result, they provide a higher fault coverage for this application, as one can see in Fig. 5.18. It results in an MWTF of up to 6.46x when MM is protected by VAR3+

[13] More information about the benchmarks is available in the appendix C.

and SETA. The execution time ranges from 1.50x to 1.81x, and the code size varies from 1.88x to 2.31x.

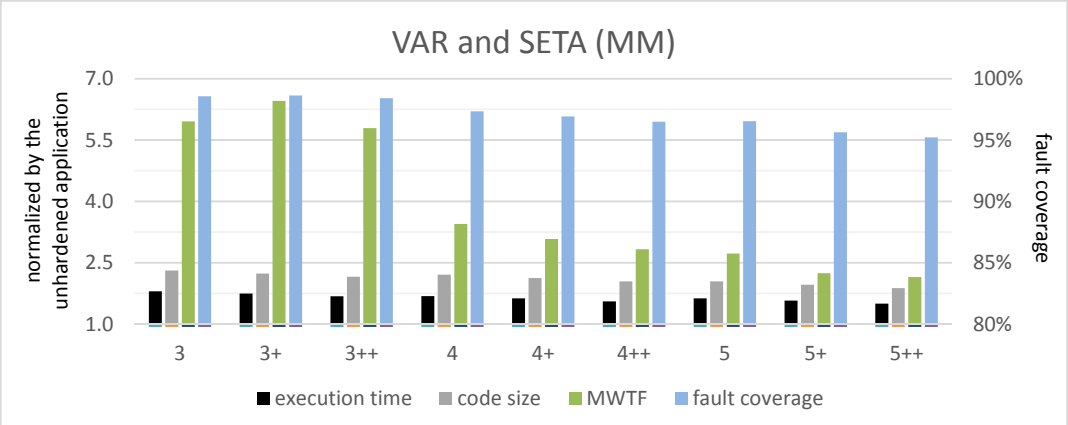


Fig. 5.18: Results of combining VAR and SETA for the matrix multiplication (MM). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 5.19 shows the results for the run-length encoding (RLE). RLE is the largest code, it explains why its code size overhead is higher when compared to the execution time overhead. The execution time varies from 1.60x to 2.09x, and the code size ranges from 1.90x to 2.61x. The MWTF reaches up to 6.30x.

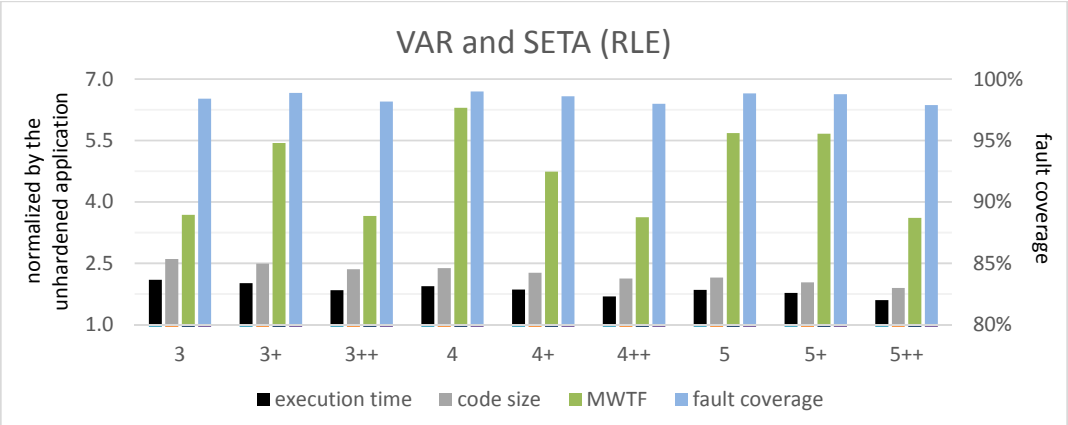


Fig. 5.19: Results of combining VAR and SETA for the run-length encoding (RLE). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Summation (SUM) is the smallest code, and its code is well-balanced regarding the types of instructions (arithmetic, load/store, and branch). However, its execution is heavily loop-based, and the single output depends on the entire program execution. Thus, an error affecting the program at any time during the execution will very likely propagate to the program output. The program output is a store instruction that is checked by all techniques. That is why all the techniques present a high fault coverage, as one can see in Fig. 5.20. The MWTF of techniques that do not protect the branches (VAR3++, VAR4++, and VAR5++) is compensated by their lower execution time. The execution time and code size range, respectively, from 1.77x to 2.37x, and from 1.39x to 1.58x.

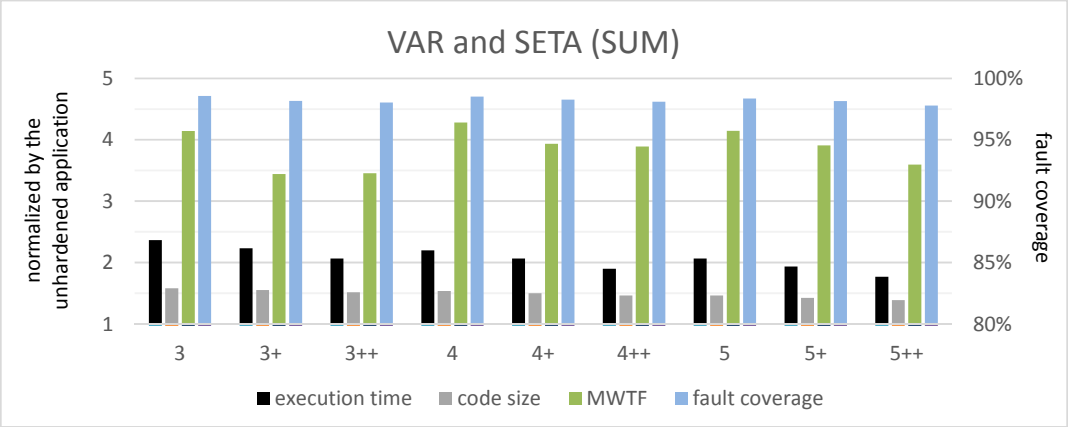


Fig. 5.20: Results of combining VAR and SETA for the summation (SUM). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The TETRA encryption algorithm is, by far, the case-study application with the highest average number of instructions per basic block. Furthermore, it is a code heavily based on load/store instructions. That is why all techniques perform well in detecting errors and providing fault coverage, once the store instructions are hardened. In addition, there are very few branches or jumps that rely on registers to get the target address. Fig. 5.21 presents the average results for each combined technique. The hardened application achieves up to 7.87x when VAR5+ and SETA are applied. The execution time ranges from 1.52x to 1.92x, and the code size varies from 1.57x to 2.06x.

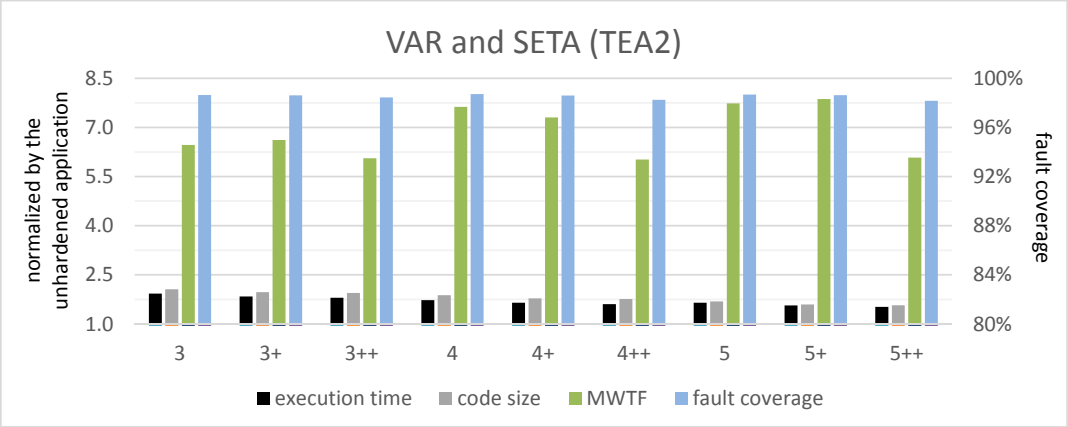


Fig. 5.21: Results of combining VAR and SETA for the TETRA encryption algorithm (TEA2). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The Tower of Hanoi is a high arithmetic, load/store application, with just a few conditional branches. On the other hand, it has many subroutine calls, and consequently many returns. A return instruction consists of a jump and a register containing the return address. If this register contains a wrong value, the return will go to the wrong position, and that may cause an incorrect execution. Once again, checking rule C6 is essential to improve the fault coverage and MWTF, which reaches up to 6.20x with VAR5 and SETA. The execution time and code size range from 2.15x to 2.72x, and from 1.64x to 1.95x, respectively, as shown in Fig. 5.22.

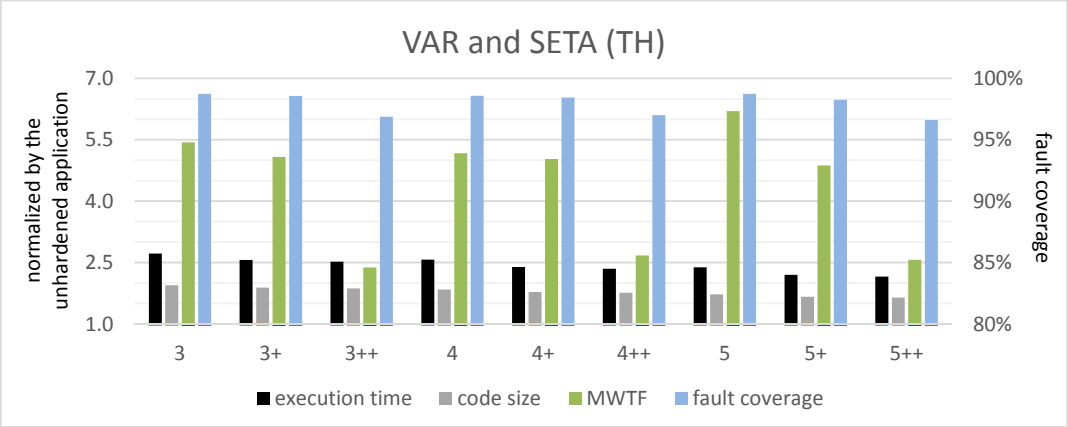


Fig. 5.22: Results of combining VAR and SETA for the Tower of Hanoi (TH). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 5.23 presents the average execution time, code size, MWTF, and fault coverage for some combinations of VAR techniques with SETA, and for an improved version of the state-of-the-art SWIFT technique. We used the original data-flow part of SWIFT and replaced the control-flow part, which is a modified version of CFCSS technique, by CEDA because CEDA achieves and extends the capabilities of detecting errors of the control-flow part of SWIFT with very similar overheads. The proposed techniques follow the same representation of previous charts, and the state-of-the-art technique is represented by the name *SoA*. Its execution time, code size, MWTF, and fault coverage are, respectively, 1.99x, 1.87x, 2.85x, and 97.30%. VAR3+, SETA presents the highest MWTF (5.19x), and fault coverage (98.62%), with a higher execution time (2.11x) and little higher code size (1.90x). If the aim is reliability, then VAR3+, SETA should be chosen. If the execution time cannot be higher than SoA, then VAR5 can replace it (once it has an execution time of 1.97x) and improve significantly the reliability (the MWTF is 4.88x). If the code size is the constraint, VAR3+, SETA could replace SoA with little increase in the code size, from 1.87x to 1.90x. Another option is to use VAR4, SETA, which presents a code size of 1.86x, with an execution time of 2.05x and MWTF of 5.14x. Finally, if the reliability achieved by SoA is enough, it is possible to reduce the overheads by replacing SoA by VAR4++, SETA or VAR5+, SETA. The first presents 1.82x of execution time, 1.74x of code size, and 2.89x of MWTF. And the second presents an execution time of 1.88x, a code size of 1.68x, and an MWTF of 4.01x. If CEDA is used with another state-of-the-art data-flow technique, it can provide a fault coverage similar fault to the proposed techniques. However, the overheads will be higher than any combination of VAR with SETA. Therefore, its MWTF will be lower.

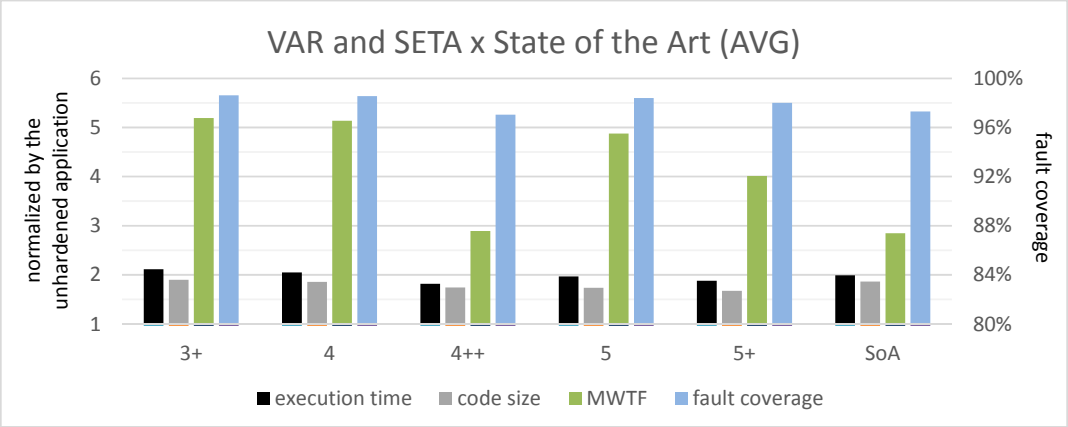


Fig. 5.23: Average results for combining VAR and SETA vs. state-of-the-art (SoA) techniques. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

5.3.3 Radiation test results in the ARM Cortex-A9 processor

Radiation test is considered one of the approaches more close to the real application environment to measure the reliability of new fault tolerant techniques. However, contrarily to the simulations carried out in previous section an exhaustive evaluation of every combination is not feasible with radiation due to limited beam time. Therefore, we used the fault injection campaigns as a guideline to select the most suitable combination of SIHFT techniques based on MWTF.

5.3.3.1 Test with neutrons

Experiments were performed at Los Alamos National Laboratory's (LANL) Los Alamos Neutron Science Center (LANSCE) Irradiation of Chips and Electronics House II, Los Alamos, US, in order to validate the fault injection campaign by simulation. As mentioned in (VIOLANTE, 2007), LANSCE provides a white neutron source that emulates the energy spectrum of the atmospheric neutron flux. The relationship between neutron energy and modern devices cross section is still an open question. Nevertheless, LANSCE beam has been empirically demonstrated to be suitable to mimic terrestrial radiation environment (VIOLANTE, 2007).

The setup is the one presented in section 4.2.2. It consists of a board, computer, USB net switch, cables for communication, and cables for power supply. The neutron flux was approximately 1.5×10^6 n/(cm².s) for energies above 10 MeV. The beam was focused on a spot with a diameter of 2 inches plus 1 inch of penumbra, which provided uniform irradiation of the device without directly affecting nearby board power control circuitry. Irradiation was performed at room temperature with normal incidence and nominal voltages.

Two versions of case-study Tower of Hanoi (30 elements in the stack) have been tested, one unhardened and the other hardened by VAR4++ and SETA techniques. Table 5.9 summarizes the data from the neutron experiment. The unhardened version was executed for 100 minutes under the beam, receiving a total fluence of 9.0×10^9 n/cm² in average. The hardened version was executed for 730 minutes under the beam, receiving a total fluence of 6.57×10^{10} n/cm² in average. We observed 6 incorrect executions out of 1557, which results in an SER of 3.854×10^{-3} and a cross section of 6.67×10^{-10} cm² for the unhardened application. In the hardened version, we observed 5 undetected errors that

lead to incorrect output on a total of 4872 executions, which results in an SER of 1.026×10^{-3} and a cross section of 7.61×10^{-11} cm². The detection techniques were capable of detecting 90.9% of the errors affecting the processor. That is the reason why we can see a reduction of the SER by 3.76 and of the cross-section by one order of magnitude when hardening using VAR4++ and SETA. However, the execution time of the hardened case-study application used in ARM is 2.33x, and the code size is 2.13x compared to the unhardened application. That results in an MWTF of 1.61x for the hardened application.

Table 5.9: Summary of radiation test with neutrons on the ARM Cortex-A9 processor (VAR4++ and SETA)

BB Type	Unhardened	Hardened by VAR4++ and SETA
Flux	1.5×10^{6} n/(cm².s)	1.5×10^{6} n/(cm².s)
Time of exposure	100 min	720 min
Fluence	9.0×10^{9} n/cm²	6.57×10^{10} n/cm²
SER	3.854×10^{-3}	1.026×10^{-3}
Cross-section	6.67×10^{-10} cm²	7.61×10^{-11} cm²
Executions	1557	4872
Execution time	3.85 s	9.00 s
Code size	472 B	1004 B
MWTF	1.00x	1.61x

The MWTF obtained by simulation on the miniMIPS processor for the Tower of Hanoi hardened by VAR4++ and SETA was 2.68x. The same benchmark, but running on the ARM Cortex-A9 processor and tested under neutrons reached an MWTF of 1.61x. A factor that influenced in this difference is the different processor used in both tests. Thus, the final code and the processor architecture are not the same. Anyhow, it is noticeable an increase of the MWTF from the unhardened to the hardened version.

5.3.3.2 Test with heavy ions

Heavy ions experiments were conducted at Laboratório Aberto de Física Nuclear of the Universidade de São Paulo (LAFN-USP), Brazil (AGUIAR, 2014). The ion beams were produced and accelerated by the São Paulo 8UD Pelletron Accelerator. Aiming to achieve a very low particle flux in the range from 10^{2} to 10^{5} particles.cm^{-2}.s^{-1}, as recommended by the European Space Agency (ESA) for SEU tests (ESA, 2014). A standard Rutherford scattering setup using a gold foil was used. The experiment was performed in air. A silicon barrier detector was mounted inside the vacuum chamber at an angle of 45º to monitor the beam intensity. In front of the detector, it was mounted a collimator with a diameter of 4 mm, defining a solid angle of about 0.085 msr. The SEU events were observed irradiating ^{16}O beams, scattered by an 184 µg/cm² gold target, with an energy of 51 MeV (effective energy of 41 MeV), which provided a Linear Energy Transfer (LET) of 5 MeV/mg/cm² and penetration in Si of 29 µm. To achieve the desired particle flow, the DUT was positioned at a scattering angle of 15º, resulting in an average flux of 584.44 particles.cm^{-2}.s^{-1}. Finally, the DUT was also positioned in a way that the center of the beams was focused in the PS part.

The package of the device was thinned to allow that irradiated particles penetrate the active region of the silicon. Fig. 5.24(a) shows the chip surface without its package. It is possible to distinguish between the PS and the PL part. Fig. 5.24(b) shows a microscopic section of the chip performed to evaluate the energy loss of the heavy ions after passing the passive layers. The passive layers consist of eleven copper metallization layers separated by dielectric layers. The total thickness of the passive layers is 12.87 µm. To estimate the energy loss of the heavy ions, it was assumed a total thickness of the copper metallization layers of 7.87 µm, and a total thickness of the dielectric layers of 5.0 µm.

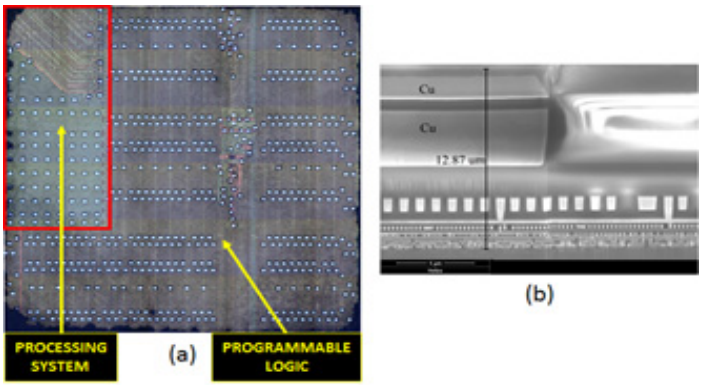


Fig. 5.24: (a) View of the surface of the XC7Z020-CLG484 device, and (b) Microscopic section of the XC7Z020-CLG484 device.

The setup is the one presented in section 4.2.2. It consists of a board, computer, USB net switch, cables for communication, and cables for power supply. Only one ARM core was utilized during the test, data and instruction L1 caches were enabled, and L2 was disabled. The processor was running a target application that sends the output by UART to the computer and, then, restarts its execution. The computer was running a monitoring application that listens to the COM port connected to the board UART and classifies the output. In case of error in the ARM processor, the processor is reset.

Two versions of a Tower of Hanoi (20 elements in the stack) have been tested, one unhardened, and the other hardened by VAR3+ and SETA techniques. Table 5.10 summarizes the parameters utilized in the radiation test with heavy ions. The unhardened version was 92 minutes under radiation, receiving a total fluence of 3.23×10^6 part/cm² in average. The hardened version was 91 minutes under radiation, receiving a total fluence of 3.19×10^6 part/cm² in average. We observed an SER of 5.43×10^{-3} and a cross section of 9.30×10^{-6} cm² for the unhardened application. For the hardened version, we observed an SER of 1.47×10^{-3} and a cross section of 2.51×10^{-6} cm². One can see a reduction of the SER and cross section by a factor of 3.71 when hardening using VAR3+ and SETA. However, the execution time of the hardened case-study application is 2.62 times, and the code size is 3.95 times the unhardened application for the ARM processor. That results in a normalized MWTF of 1.66x for the hardened application.

Table 5.10: Summary of radiation test with heavy ions on the ARM Cortex-A9 processor (VAR3+ and SETA)

BB Type	Unhardened	Hardened by VAR3+ and SETA
Flux	$5.84x10^2$ part/(cm².s)	$5.84x10^2$ part/(cm².s)
Time of exposure	92 min	91 min
Fluence	$3.23x10^6$ part/cm²	$3.19x10^6$ part/cm²
SER	$5.43x10^{-3}$	$1.47x10^{-3}$
Cross section	$9.30x10^{-6}$ cm²	$2.51x10^{-6}$ cm²
Execution time	$6.08\ x10^{-2}$ s	$1.59\ x10^{-1}$ s
Code size	252 B	996 B
MWTF	1.00x	1.66x

Although the results obtained from simulated fault injection cannot be directly compared with the ones obtained from the radiation experiment, it is possible to notice that the MWTF of the hardened version in the radiation experiment was not very high, only 1.66x. There are many factors that influenced the lower MWTF of the radiation test. One of the major causes is the presence of cache memories. The experiments show that ARM caches are very sensitive to radiation and prone to faults that become errors. Another concern is that simulation model does not include microarchitectural registers.

Data-flow technique VAR3+ implements duplication rule D2, which does not create redundancy in the main memory and, consequently, in the cache memories. Therefore, a fault affecting the L1 cache (enabled in the heavy ion experiment) is not detected by VAR3+. Anyhow, it is important to mention that the fault injection method must not be used to replace radiation because it cannot reproduce the complexity of the radiation flux and the complete hardware architecture implementation. The fault injection simulator was designed only for comparing the increase of reliability offered by different SIHFT techniques, but not to get estimations of absolute reliability values.

Another test with heavy ions comparing VAR3 and SETA with the unhardened application was performed. The same configuration utilized, including the same benchmark (a Tower of Hanoi with 20 elements in the stack). However, it is not possible to directly compare the results of this experiment with the previous one because the beam does not cover the entire chip. Therefore, we cannot ensure that particles are hitting the same area. Table 5.11 summarizes the parameters utilized in the radiation test with heavy ion. The unhardened version was 88.5 minutes under radiation, receiving a total fluence of $3.10x10^6$ part/cm² in average. The hardened version was 179.8 minutes under radiation, receiving a total fluence of $6.30x10^6$ part/cm² in average. We observed an SER of $5.35x10^{-4}$ and a cross section of $1.22x10^{-5}$ cm² for the unhardened application. For the hardened version, we observed an SER of $1.38x10^{-4}$ and a cross section of $1.11x10^{-6}$ cm². One can see a reduction of the SER by a factor of 3.88 and a reduction of the cross section by a factor of 11.03 when hardening using VAR3 and SETA. However, the execution time of the hardened case-study application is 2.69x for the ARM processor. As a consequence, the hardened application presents an MWTF of 1.64x.

Table 5.11: Summary of radiation test with heavy ions on the ARM Cortex-A9 processor (VAR3 and SETA)

BB Type	Unhardened	Hardened by VAR3 and SETA
Flux	$5.84x10^{2}$ part/(cm².s)	$5.84x10^{2}$ part/(cm².s)
Time of exposure	88.5 min	179.8 min
Fluence	$3.10x10^{6}$ part/cm²	$6.30x10^{6}$ part/cm²
SER	$5.35x10^{-4}$	$1.38x10^{-4}$
Cross section	$1.22x10^{-5}$ cm²	$1.11x10^{-6}$ cm²
Execution time	$6.08\ x10^{-2}$ s	$1.64\ x10^{-1}$ s
Code size	252 B	1020 B
MWTF	1.00x	1.66x

5.4 Summary

In this chapter we proposed new data-flow and control-flow techniques. The goal of providing similar reliability of state-of-the-art techniques with lower overheads was achieved. In addition, it was possible to improve the reliability by keeping a similar fault coverage and reducing overheads when compared to state-of-the-art techniques. We discussed the importance of checking registers used by store, load, and branch instructions. Furthermore, the variations of VAR techniques open a set possibilities for hardening an application depending on different constraints due to their differents levels of reliability and overheads.

6 PROPOSED SELECTIVE HARDENING

Selective implementations using one of the proposed data-flow techniques plus the proposed control-flow technique are introduced in this chapter. Firstly, the selective hardening is performed with a data-flow technique through the selection of which register will be hardened. Then, the selective data-flow technique is complemented by a control-flow technique (with no selective hardening). After that, the selectiveness is implemented in the control-flow technique in two ways: (1) removing checkers from selected basic blocks and (2) removing the entire protection of selected basic blocks. These two approaches are tested individually and with a data-flow technique. Finally, the selective hardening is applied in both data-flow and control-flow techniques. By applying the techniques selectively, it is possible to lower even more the overheads. The aim is to find the point with minimum overheads where the fault coverage is still similar to applying the techniques completely. Furthermore, finding the best trade-off between reliability and performance is a point of interest.

6.1 Selective data-flow technique

Data-flow techniques duplicate all the registers used by an application, which may cause significant overheads. Thus, the use of data-flow techniques may be infeasible if the application has performance or memory constraints. Furthermore, sometimes the application uses many registers, not leaving enough for duplicating all used registers. In such cases, a subset of the used registers can be hardened. It will present lower overheads than hardening all registers, which can meet the application constraints and provide some reliability. Nevertheless, the registers cannot be randomly selected. A random selection of registers may provide a lower reliability than a smarter approach based on the application behavior. Therefore, the method to select registers is of great importance and will affect the application reliability.

6.1.1 Methodology and implementation

In order to improve the trade-off between reliability and overheads, every register must be analyzed. However, testing all the possible subset of used registers is infeasible due to its exponential property. Eq. 6.1 demonstrates that the number of possible subsets C of an application depends on the number of used registers n. It is given by the summation from 0 to n of the Newton's binomial with n and i as coefficients, where i is the summation variable. This equation is equivalent to 2^n.

$$C_{i:0\rightarrow n} = \sum_{i=0}^{n} \frac{n!}{i!\,(n-i)!} = \sum_{0\leq i\leq n} \binom{n}{i} = 2^n \qquad \text{(Eq. 6.1)}$$

A metric to define the criticality of registers was proposed by Restrepo-Calle (2015). It analyzes the application dynamically, evaluating the registers' effective lifetime, functional dependencies, and their use in branch instructions. As result, the metric provides a list of registers ranked by their criticality. Thus, there are no need for exhaustive tests. This work protects registers following this metric. As follows, we present a brief explanation of the main topics of this metrics:

- **Dynamic code analysis**: it consists of using dynamic measurements for the computation algorithms during runtime. This kind of assessment does not represent any inconvenience in the usual design flow of embedded systems, and alternatively, improves significantly the accuracy of the estimations
- **Lifetime**: the register *lifetime* represents the time when useful data in present in the register. Any fault occurring to the register during that time destroys the data integrity. Therefore, the higher the lifetime, the longer the register is prone to faults. The lifetime is expressed as the sum of clock cycles of all the register *living intervals* during the program execution
- **Living interval**: a living interval starts with a write operation and ends with the last read operation before the next write operation in the same register
- **Effective lifetime**: it is an improved evaluation of the register lifetime. Registers with the same lifetime, but with a different number of living intervals have a different effective lifetime. The more living intervals a register has, the lower is its effective lifetime. That is a very important consideration that takes into account characteristics of the pipeline
- **Weight in conditional branches**: errors affecting registers used by branch instructions may lead the application control to take an incorrect path. That is the reason to give more attention to branch instructions
- **Functional dependencies**: this criterion is the count of functional dependencies among registers. Registers having a lot of descendants are more sensitive to the whole application because an error affecting it has a higher chance of propagating to other registers and the application output. In Restrepo-Calle (2015), only the direct descendants were considered
- **Criticality**: it is the score that each register receives based on its effective lifetime, weight in conditional branches, and functional dependencies. The registers are then ranked by their criticality. The higher the value, the more critical the register.

The criticality, as well as its components for each case-study application of this thesis work, are listed and discussed in the appendix C, which presents the benchmarks.

VAR3+ was selected as the data-flow technique for the selective hardening due to its high fault coverage and MWTF when applied together with SETA. VAR4 is also a good candidate because it presented a slightly lower MWTF with lower overheads. However, it means that it has a lower fault coverage too. And since the overheads of VAR3+ are higher, it means that the selective hardening of this technique will reduce more the overheads. That in addition to the fact that VAR3+ has a higher fault coverage makes it is the best candidate to improve the reliability with the selective hardening. The selective VAR3+ technique will be referred as S-VAR from now on. The methodology is similar to the one presented in section 3.4.1. However, the registers are hardened based on the ranking provided by the metric for criticality cited above. Table 6.1 shows examples of S-VAR when different numbers of registers are hardened. The original code is presented as normal text, and the code inserted by S-VAR is bold. Note that the S-VAR with all registers hardened is equivalent to VAR3+. In order to apply the selective hardening in the applications, we used features of the CFT-tool that allow us to indicate which registers must be hardened, and which are their priorities to be hardened. There are built-in static metrics for criticality in the tool, but once Restrepo-Calle (2015)'s metric is dynamic, it was necessary to calculate the registers criticality for each case-study

application before hand. Then, we inform CFT-tool of the registers that must have a higher priority to be hardened through a customized option to select registers.

Table 6.1: Example of a selective data-flow technique (S-VAR)

#	Unhardened code	Hardened by S-VAR ($2)	Hardened by S-VAR ($2,$6)	Hardened by S-VAR ($2,$4,$6)
1	la $2,$result	la $2,$result	la $2,$result	la $2,$result
2		**la $12,$result**	**la $12,$result**	**la $12,$result**
3			**bne $6,$16,error**	**bne $6,$16,error**
4	lw $4,0($6)	lw $4,0($6)	lw $4,0($6)	lw $4,0($6)
5				lw $14,0($16)
6	sll $4,$4,2	sll $4,$4,2	sll $4,$4,2	sll $4,$4,2
7				sll $14,$14,2
8			**bne $6,$16,error**	**bne $6,$16,error**
9		**bne $2,$12,error**	**bne $2,$12,error**	**bne $2,$12,error**
10	sw $6,4($2)	sw $6,4($2)	sw $6,4($2)	sw $6,4($2)
11				**bne $4,$14,error**
12		**bne $2,$12,error**	**bne $2,$12,error**	**bne $2,$12,error**
13	sw $4,0($2)	sw $4,0($2)	sw $4,0($2)	sw $4,0($2)

6.1.2 Fault injection results in the miniMIPS processor

Figs. 6.1 to 6.9 present the execution time, code size, MWTF, and fault coverage for all benchmarks hardened with S-VAR. The horizontal axis represents the percentage of registers hardened by S-VAR, where 0% is equivalent to the unhardened application, and 100% is equivalent to VAR3+. The execution time, code size, and MWTF are normalized by the unhardened application. The fault coverage is presented in percentage. Although the case-study applications have a different behavior and use a different number of registers, they all present similar results. The fault coverage saturates after a certain percentage of registers is hardened. This percentage depends on the application, but in general, the more registers an application have, the lower is the percentage to saturate the fault coverage. The number of more critical registers does not vary much, which means that if there are more used registers, the percentage of critical ones is lower than the average. Furthermore, the overheads roughly follow the behavior of the fault coverage, which explains why the MWTF does not change much with the increase of protection. Any difference can also be explained by the fact the metric for criticality, although good, is not perfect. As stated in the previous chapter, data-flow techniques are not enough for protecting an application. The use combined with a control-flow technique is necessary.

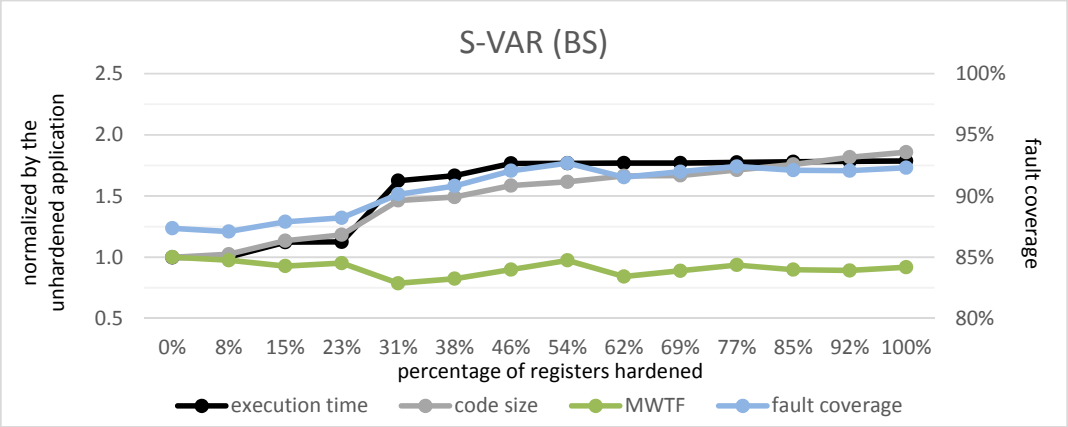


Fig. 6.1: Results for the bubble sort (BS) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

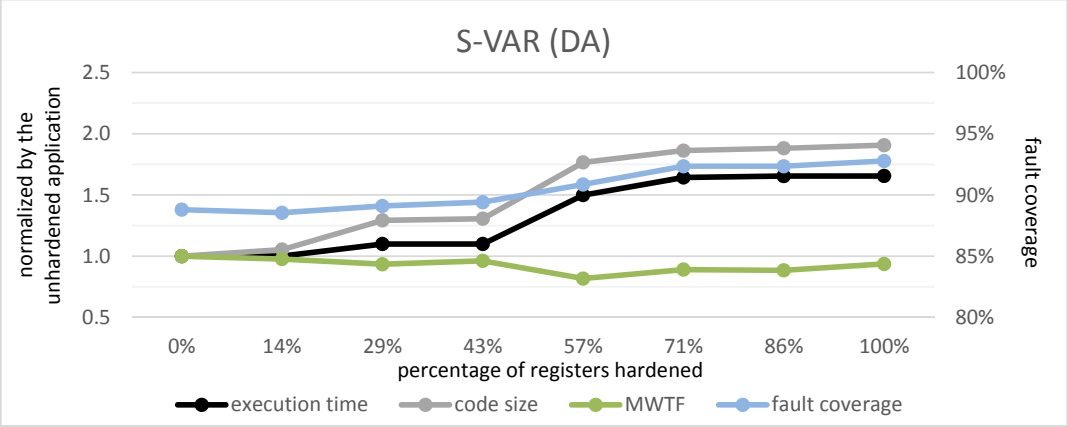


Fig. 6.2: Results for the Dijkstra's algorithm (DA) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

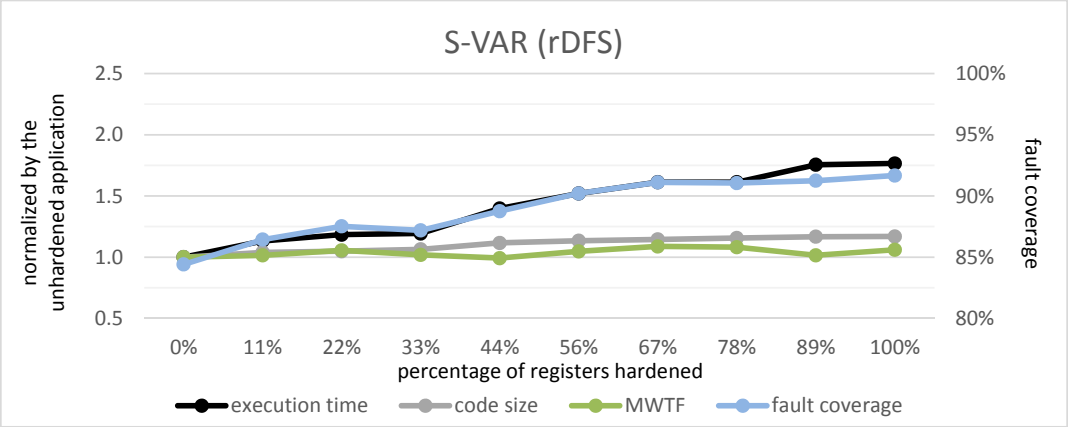


Fig. 6.3: Results for the recursive depth-first search (rDFS) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

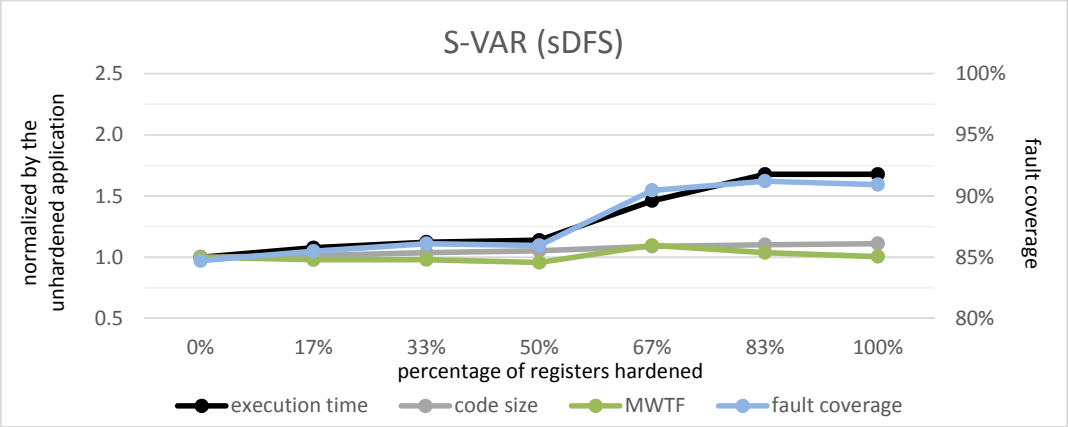


Fig. 6.4: Results for the sequential depth-first search (sDFS) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

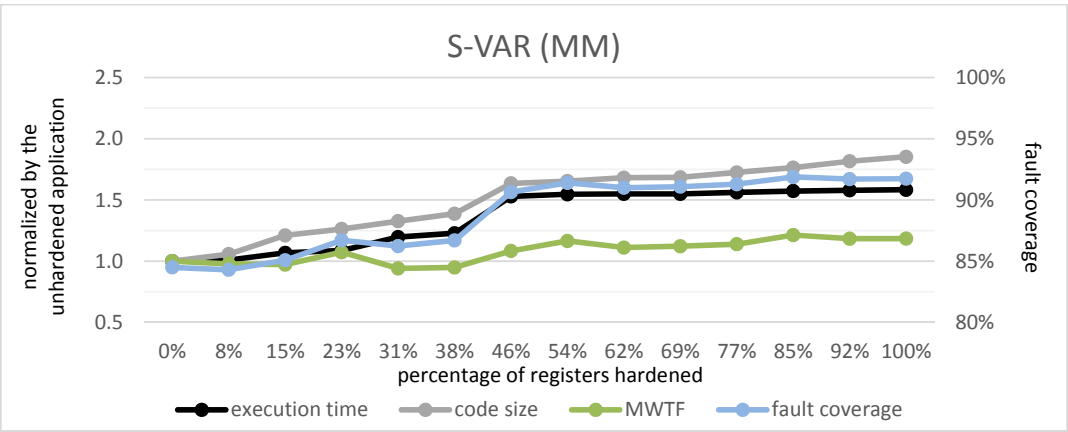


Fig. 6.5: Results for the matrix multiplication (MM) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

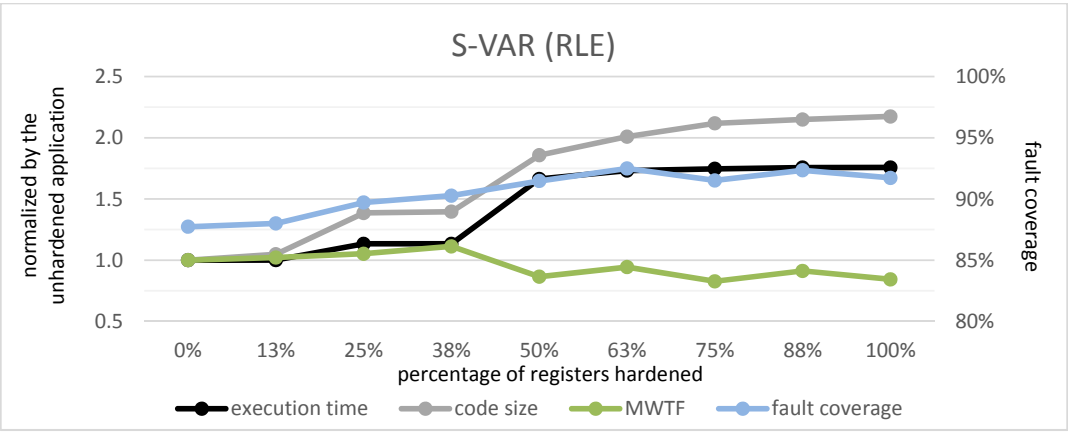


Fig. 6.6: Results for the run length encoding (RLE) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

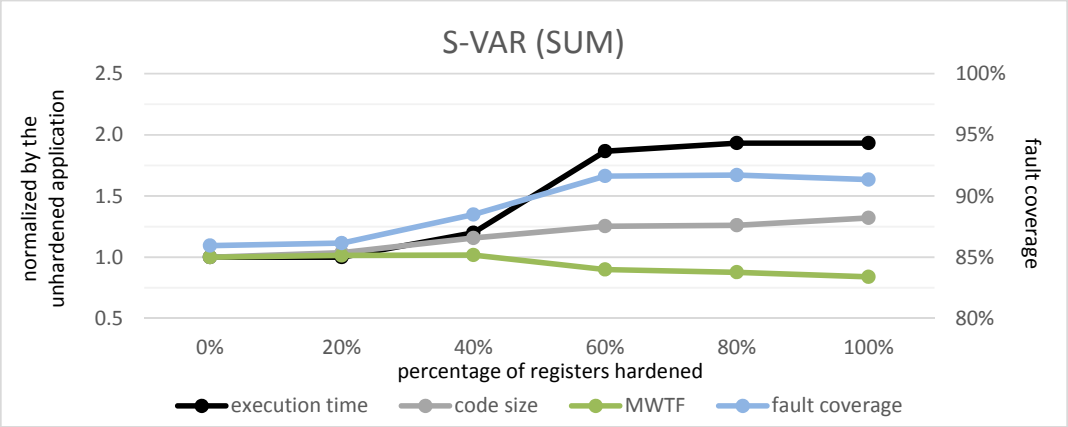


Fig. 6.7: Results for the summation (SUM) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

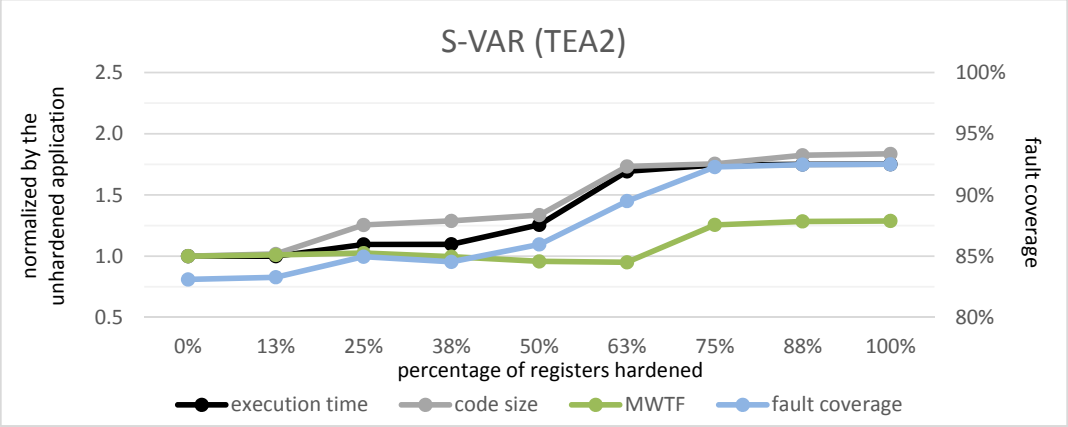


Fig. 6.8: Results for the TETRA encryption algorithm (TEA2) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

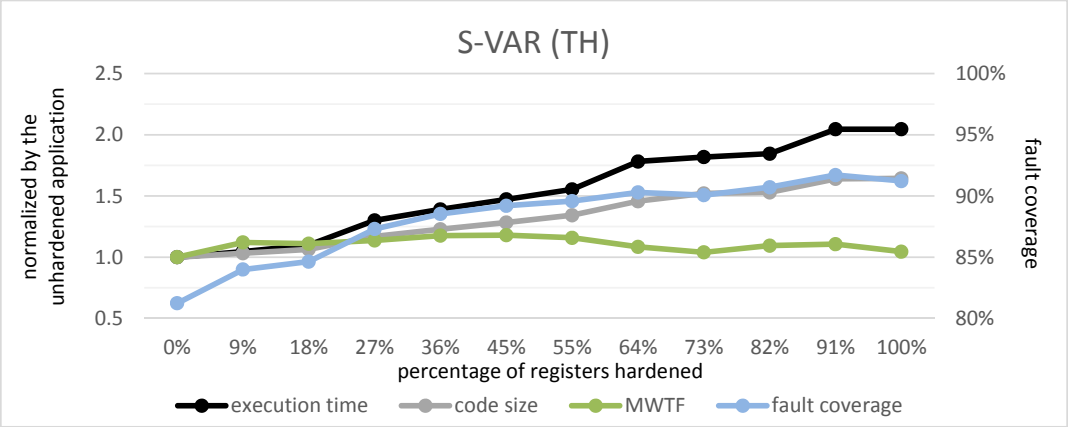


Fig. 6.9: Results for the Tower of Hanoi (TH) hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

In Fig. 6.10 we can see the highest MWTF achieved for each application and the average results. In summary, the MWTF stays a little higher than 1.0, and the higher the fault coverage, the higher the overheads. The execution time, code size, and MWTF are normalized by the equivalent unhardened application, and the fault coverage is presented in percentage.

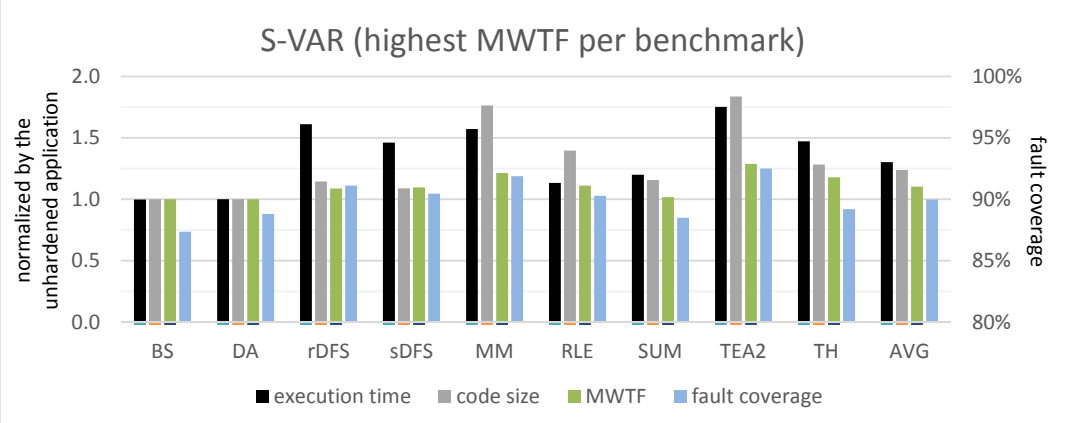


Fig. 6.10: Highest MWTF for the benchmarks hardened by the S-VAR technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 6.11 to 6.19 present the execution time, code size, MWTF, and fault coverage for all the benchmarks (one per chart) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the corresponding unhardened application. It is possible to notice that the same behavior observed with S-VAR only can be seen with the addition of SETA. However, the fault coverage saturates much later (with a higher percentage of registers hardened). It means that the small increase in the fault coverage that a not very critical register provides is still enough to compensate the additional overheads. Furthermore, the MWTF is considerably higher than S-VAR because both data-flow and control-flow are hardened, as also observed in the previous chapter.

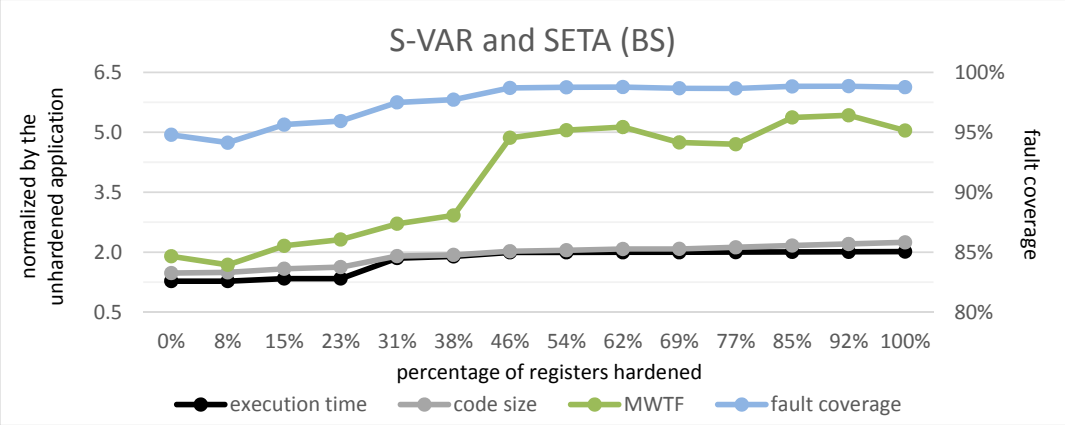


Fig. 6.11: Results for the bubble sort (BS) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

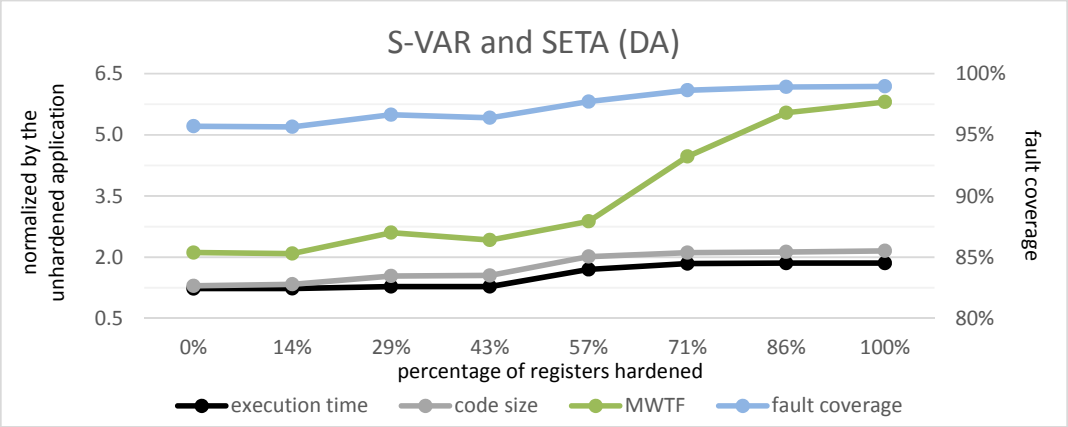


Fig. 6.12: Results for the Dijkstra's algorithm (DA) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

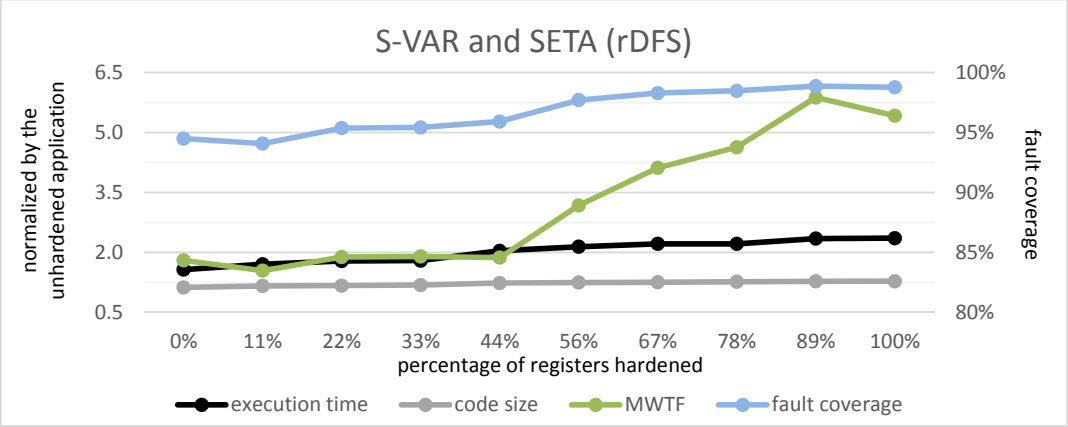


Fig. 6.13: Results for the recursive depth-first search (rDFS) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

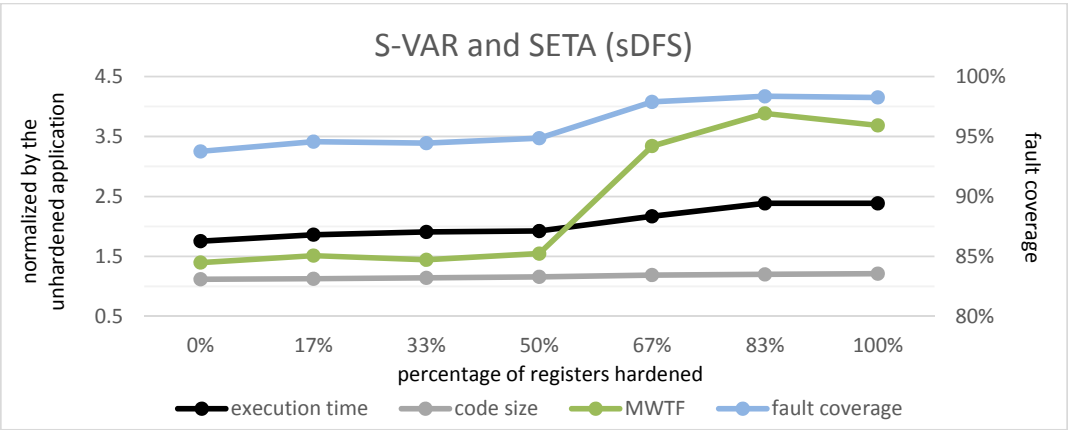


Fig. 6.14: Results for the sequential depth-first search (sDFS) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

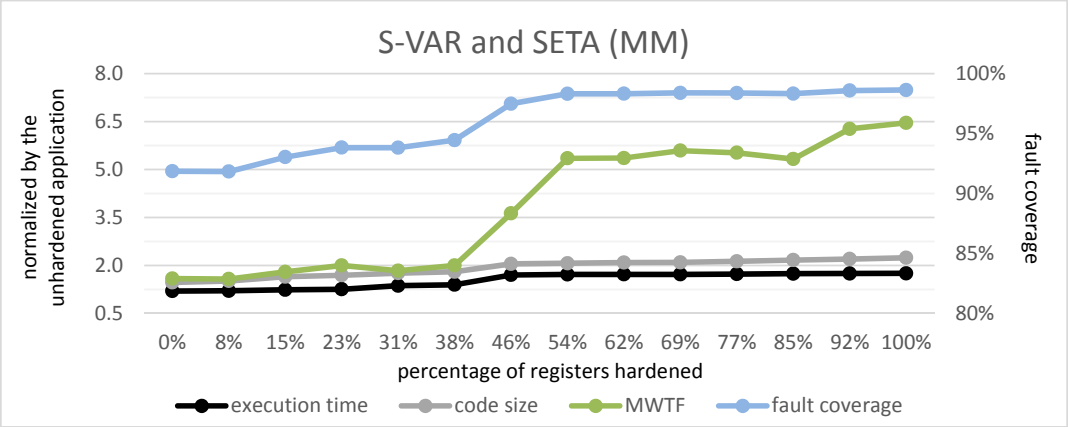


Fig. 6.15: Results for the matrix multiplication (MM) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

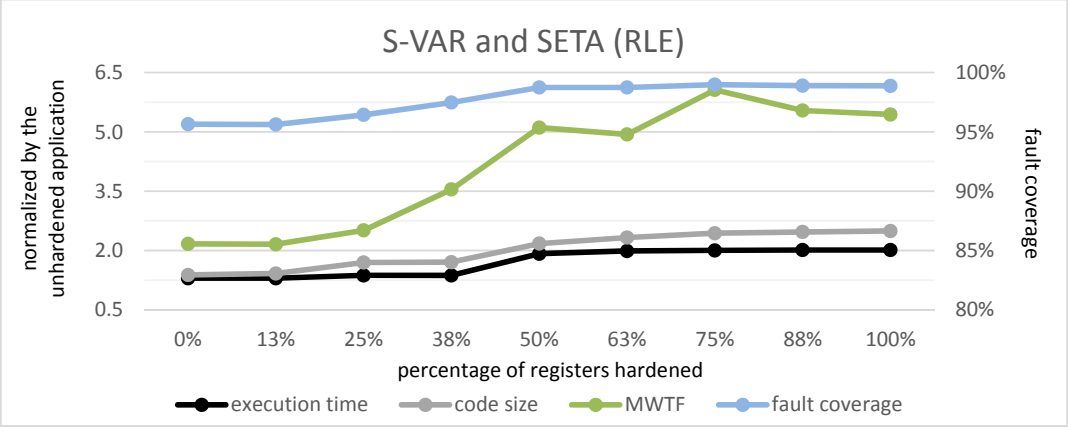


Fig. 6.16: Results for the run length encoding (RLE) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

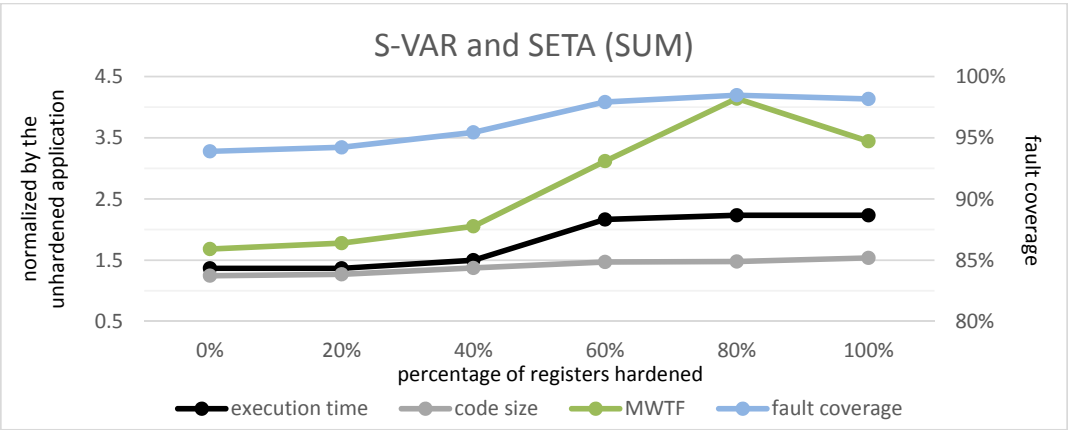


Fig. 6.17: Results for the summation (SUM) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

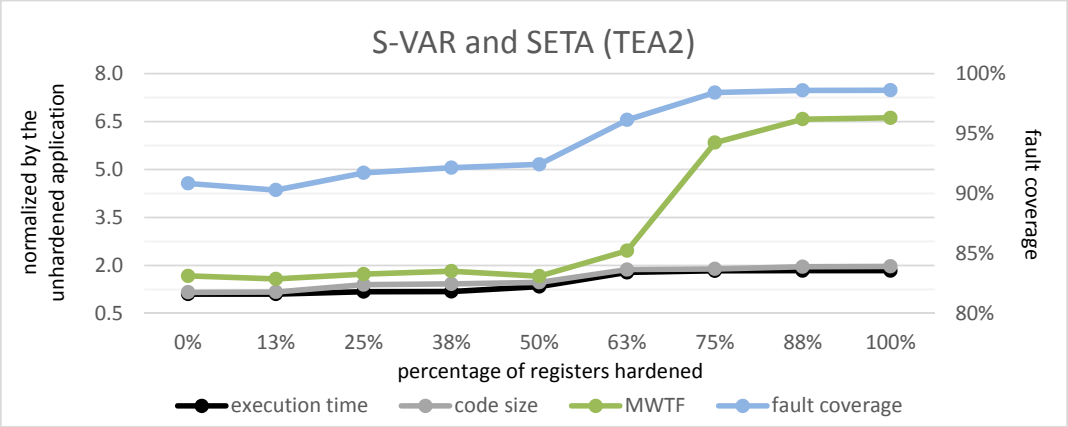


Fig. 6.18: Results for the TETRA encryption algorithm (TEA2) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

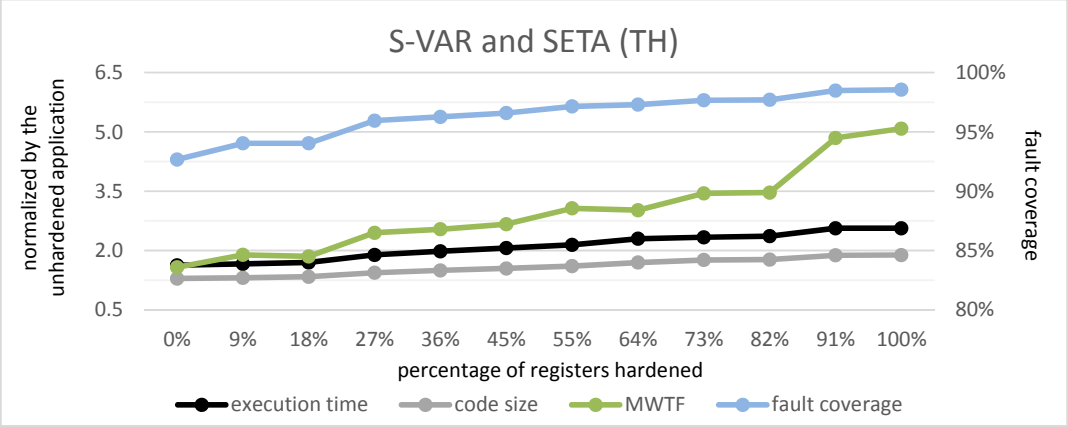


Fig. 6.19: Results for the Tower of Hanoi (TH) hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

In Fig. 6.20 one can see the highest MWTF achieved for each application and the average results. The execution time, code size, and MWTF are normalized by the equivalent unhardened application, and the fault coverage is presented in percentage. The average MWTF achieved by S-VAR and SETA is a little higher than when no selective hardening is implemented (VAR3+ and SETA). The reason is that VAR3+ is one of the cases of S-VAR, so the highest MWTF of S-VAR will never be lower than VAR3+. Furthermore, some cases of S-VAR near but lower than 100% achieve higher MWTF than when 100% of the registers are hardened. Thus, S-VAR increases the MWTF of VAR3+.

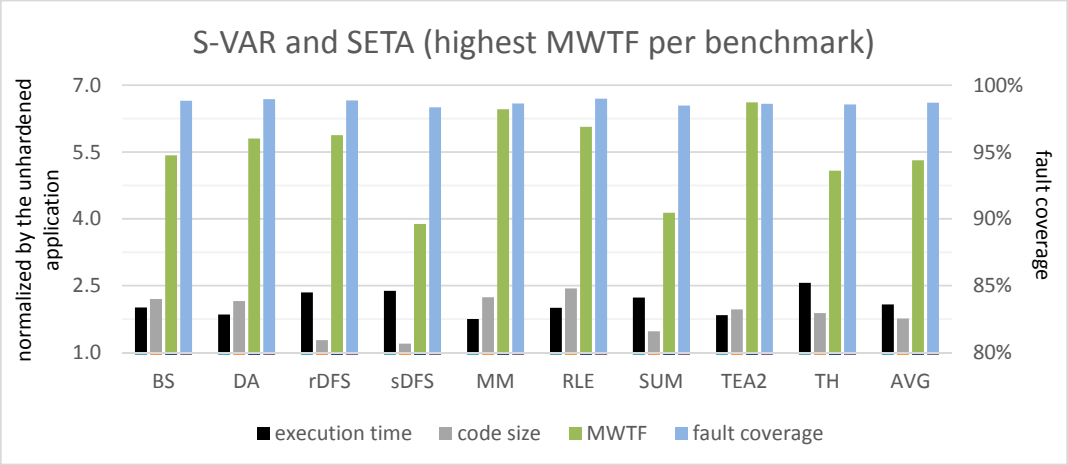


Fig. 6.20: Highest MWTF for the benchmarks hardened by S-VAR and SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

In conclusion about selective data-flow techniques, it is noticeable that a higher MWTF is reached near to 100% of registers hardened. Therefore, it is recommendable to protect the most registers possible, respecting possible constraints, to increase the fault coverage and MWTF. In addition, it is clear that the use of a control-flow technique together with a data-flow technique is key to the improvement of the reliability.

6.2 Selective control-flow technique

The selective hardening on control-flow techniques is applied to the basic blocks. Selected basic blocks are hardened with the control-flow technique while the others will lack protection in some manner, which depends on the selective hardening method implemented.

6.2.1 Methodology and implementation

As SETA showed to be a superior control-flow technique, the selective hardening on control-flow techniques is implemented using this technique. Regarding the selective hardening methods, there are two approaches, one cited by Vemu (2011), but with no implementation or evaluation, and another proposed in this work. Table 6.2 summarizes these approaches. In addition, they are explained as follow.

- **SETA-C (SETA minus Checkers)**: consists of removing checkers from the basic blocks, as stated in (VEMU, 2011). All the basic blocks are protected by SETA with signatures. However, not all of them receive a checker. Basic blocks with more connections (predecessors and successors) have a higher priority to receive a checker. If an error occurs in a basic block with no checker, it can be detected in a subsequent basic block since the error will propagate. It presents lower overheads than the standard SETA
- **S-SETA (Selective SETA)**: is a new selective method. It consists of completely ignoring some basic blocks. The ignored basic blocks receive no signatures or checkers. Thus, it is possible to provide overheads even lower than only removing checkers. This selective hardening method of SETA is better explained below.

Table 6.2: Example of a selective data-flow technique (S-VAR)

S-SETA	SETA-C
• Protect only selected basic blocks with signatures and checkers • Other basic blocks are ignored	• Protect all basic blocks with signatures • Only insert checkers in selected basic blocks

6.2.1.1 S-SETA

S-SETA ignores some basic blocks in order to reduce costs. This method was named as *tunnel effect*. It creates the effect of a tunnel between the predecessors and successors of ignored basic blocks. Thus, S-SETA does not see ignored BBs and does not protect them. The criterion used to select the basic blocks to be hardened is the basic block size. Larger basic blocks have higher priority to be selected and, thus, hardened. The size was selected as the criterion based on the following assumption:

- Small basic blocks are quickly executed and uses fewer memory positions. Therefore, the chance of being affected by a fault is lower. If they are executed just a few times, they would not be very sensitive, so its protection is not very important. On the other hand, if they are frequently executed, their susceptibility to faults increase due to their increased time of exposure, but the insertion of protection in such small basic blocks would cause significant performance degradation.

SETA-C could have implemented the same criterion to select the basic blocks to receive checkers, but it would have the two following consequences:

1. The execution time of the applications hardened by SETA-C would decrease. However, the reduction would be around 50% smaller than for S-SETA because SETA-C removes only checkers, while S-SETA removes all the basic block protection
2. The error detection rate would decrease if the size was the criterion when compared with the number of connections. Once SETA-C protects all the basic blocks, the errors are propagated to the following basic blocks. Thus, if the basic blocks with more connections receive checkers, the chance of detecting an error increases, once the chance of executing a basic block with a checker also increases.

Fig. 6.21 shows how the tunnel effect is applied to a program. Fig. 6.21(a) presents the default program flow where all the basic blocks are hardened. If the protection is reduced to 70%, as shown in Fig. 6.21(c), basic blocks 1, 4, 8, and 9 are removed. The successors of BB 1 are attributed to its predecessor, BB 0. The successors of BB 2 now are BBs 3, 5, and 6, once BB 4 was removed. BBs 5 and 6 now point to BBs 2 and 7 instead of BB 1. Furthermore, BB 8 was removed. Therefore, BB 9 has no longer a successor. Following the same idea, Fig. 6.21(b), Fig 6.21(d) and Fig. 6.21(e) show how S-SETA sees the program flow for hardening 80%, 30%, and 20% of the basic blocks, respectively.

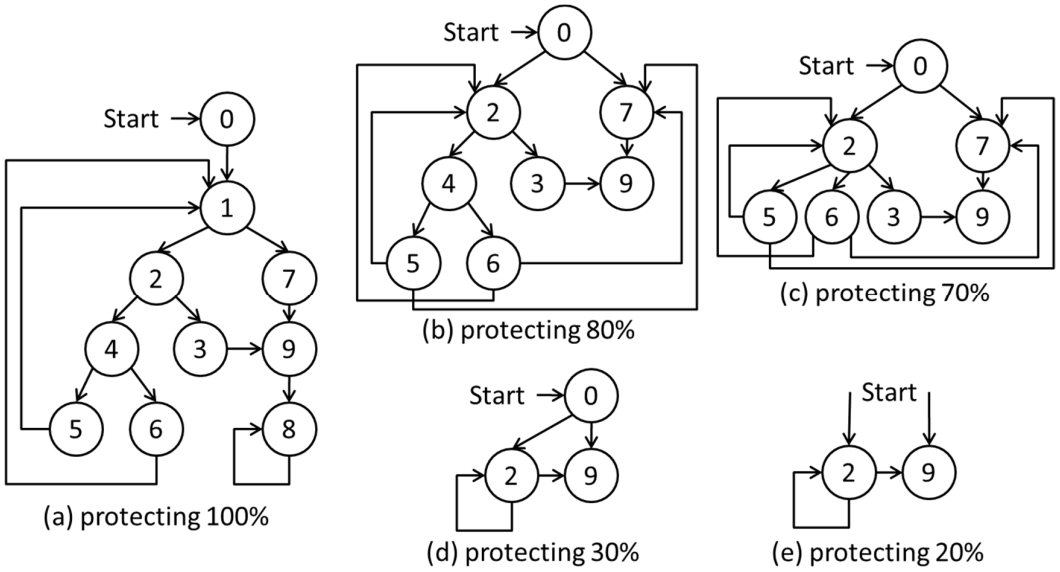


Fig. 6.21: Example of tunnel effect (S-SETA) (a) protecting 100% of BBs, equivalent to SETA, (b) protecting 80%, (c) protecting 70%, (d) protecting 30%, and (e) protecting 20% of BBs.

6.2.2 Fault injection results in the miniMIPS processor

Fig. 6.22 shows the execution time, code size, MWTF, and fault coverage for the bubble sort (BS) hardened by S-SETA. The horizontal axis represents the percentage of basic blocks hardened. When 0% of the basic blocks are hardened, S-SETA is equivalent to the unhardened application, and when 100% are hardened, S-SETA is equivalent to SETA. One can notice that for a low percentage of basic blocks hardened, S-SETA does not increase much the fault coverage and MWTF. It happens because there is not enough protection for the application to detect control-flow errors. However, from half of basic blocks hardened on, it is possible to notice an increase of the MWTF when compared to SETA. That is explained by a similar fault coverage and a lower execution time.

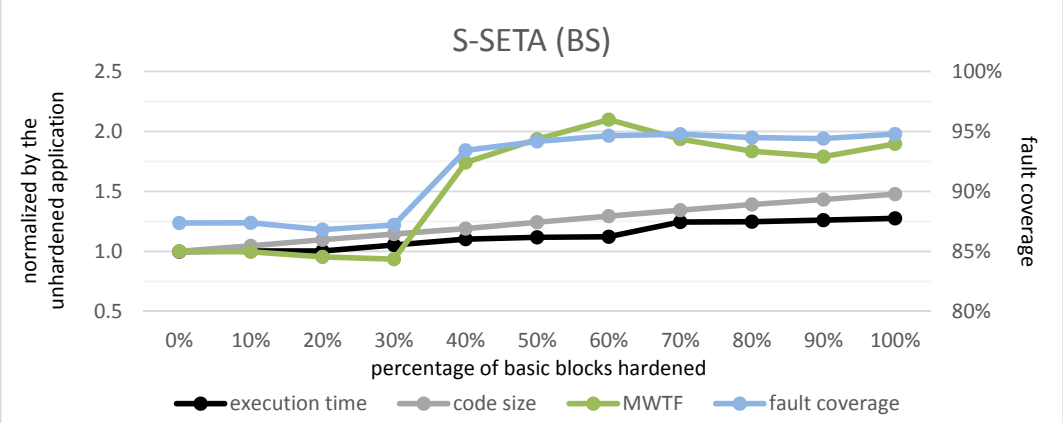


Fig. 6.22: Results for the bubble sort (BS) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

With regards to SETA-C hardening BS, shown in Fig. 6.23, it is possible to notice a constant MWTF, similar to SETA. SETA-C protects all the basic blocks, which means

that the selective hardening is only related with where checkers are inserted. Therefore, there is a chance that an error in basic block will be detected by a later checker. On the other hand, the overheads caused by SETA-C are higher because there are signature updates in all basic blocks.

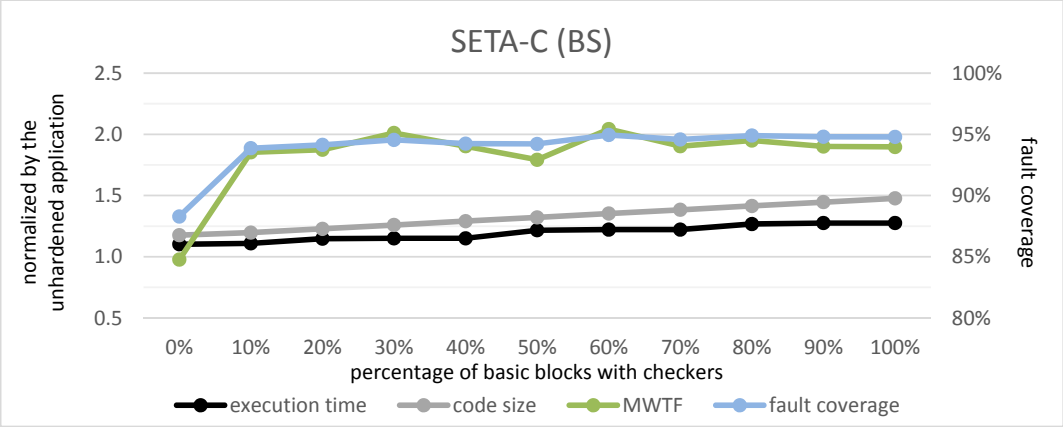


Fig. 6.23: Results for the bubble sort (BS) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 6.24 compares the S-SETA with SETA-C for the bubble sort (BS). SETA-C reaches a high MWTF with fewer basic blocks protected. However, it does not mean that SETA-C is a better option for implementations with low overhead because it always presents higher overheads than S-SETA. For example, SETA-C 10% has similar overheads to S-SETA 50%. Therefore, S-SETA is a better option for meeting overhead constraints. Furthermore, S-SETA presents similar fault coverage with lower overheads, and, in consequence, higher MWTF. With the percentage of basic blocks getting near to 100%, S-SETA and SETA-C start to converge to SETA.

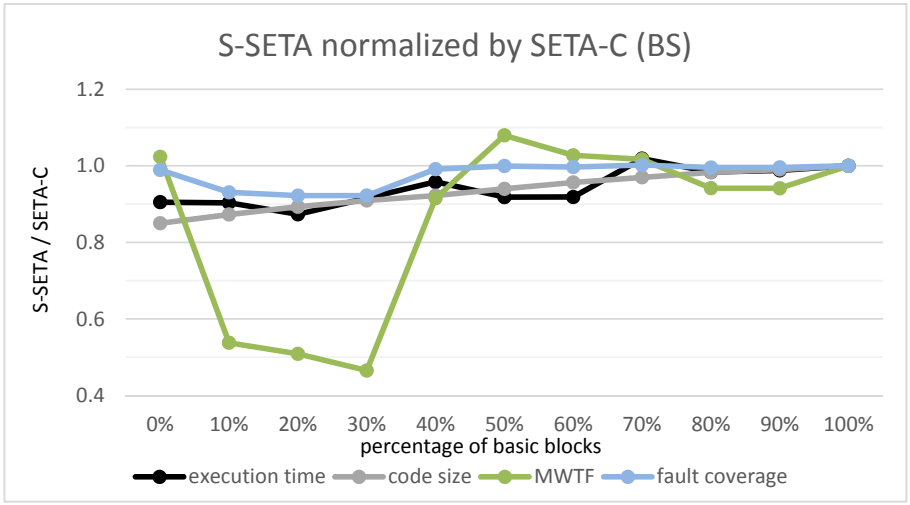


Fig. 6.24: Comparison between S-SETA and SETA-C for the bubble sort (BS). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Similar results to BS can be found for the Dijkstra's algorithm (DA). Fig. 6.25 and 6.26 present, respectively, the execution time, code size, MWTF, and fault coverage for DA. The fault coverage is presented in percentage, and the other parameters are showed

normalized by the unhardened application. The MWTF when hardening with S-SETA increases after a certain percentage of basic blocks is hardened. When hardening with SETA-C, the MWTF is constant. Similarly to BS, S-SETA reaches a higher MWTF when it achieves similar fault coverage, as one can see by the comparison presented in Fig. 6.27. Note: SETA-C 0% presents overheads higher than 1.0 because all basic blocks have signature updates, even so that no checker is inserted. This case would never be implemented in real cases because it would be better to use the unhardened application instead.

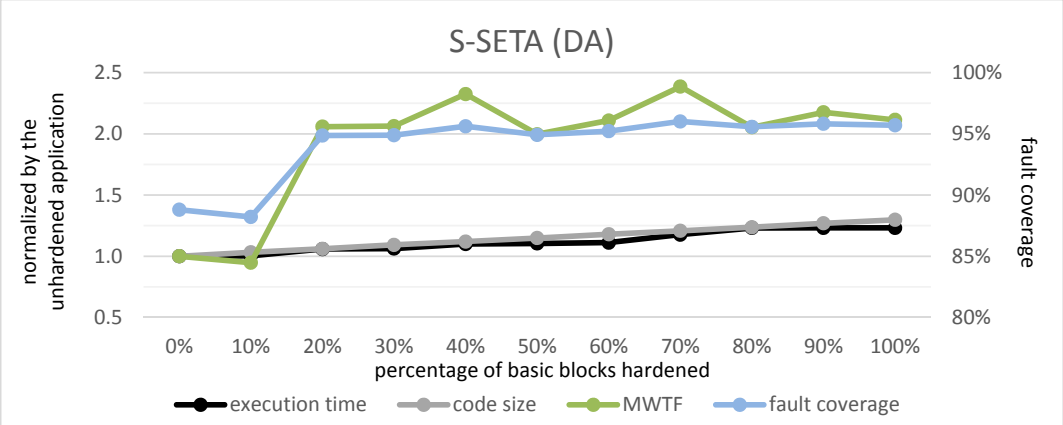


Fig. 6.25: Results for the Dijkstra's algorithm (DA) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

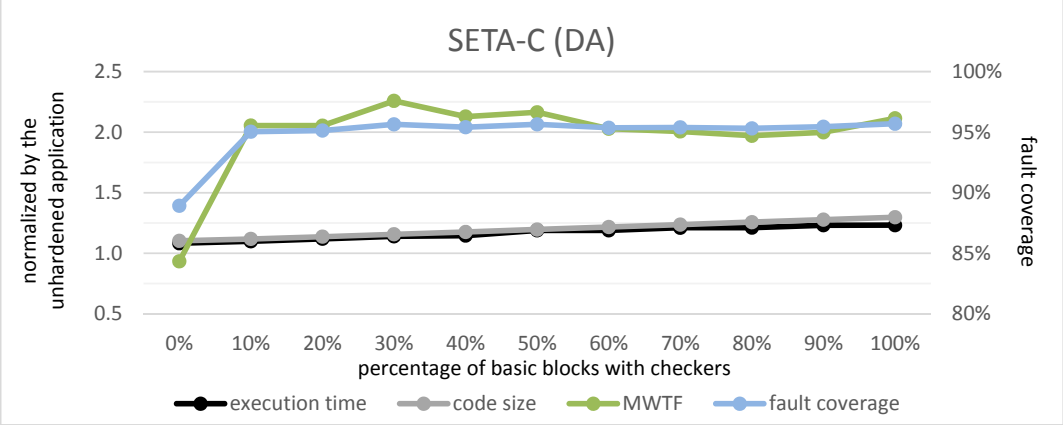


Fig. 6.26: Results for the Dijkstra's algorithm (DA) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

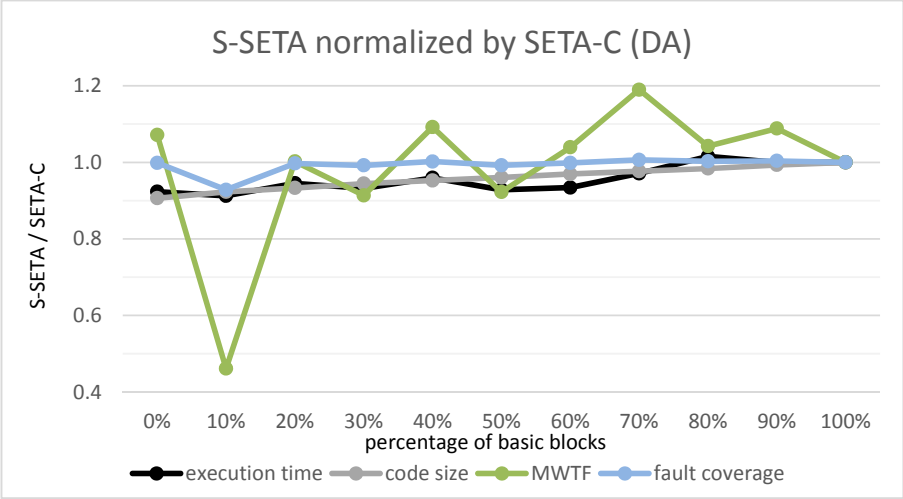


Fig. 6.27: Comparison between S-SETA and SETA-C for the Dijkstra's algorithm (DA). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Fig. 6.28 presents the execution time, code size, MWTF, and fault coverage for the recursive depth-first search (rDFS) hardened by S-SETA. As one can see, S-SETA reaches high fault coverage with low overheads when 20% of the basic blocks are hardened. The MWTF reduces with the increase of the protection because the additional gain in fault coverage is very low if compared to the increase of the execution time. When compared to SETA-C, presented in Fig. 6.29, it is possible to notice that exceptionally for this application, the MWTF of SETA-C increases for a greater percentage of basic blocks hardened if compared to S-SETA. It is due the way each technique select the most critical basic blocks. S-SETA selects based on the BB size, and SETA-C selects based on the number of connections the BB have. It means, for this application, that the number of connections is not a major factor to define which basic block should receive a checker. Furthermore, the MWTF is not constant because the execution time increases more than most other case-study applications with no additional fault coverage. That explain the earlier convergence of SETA-C to SETA.

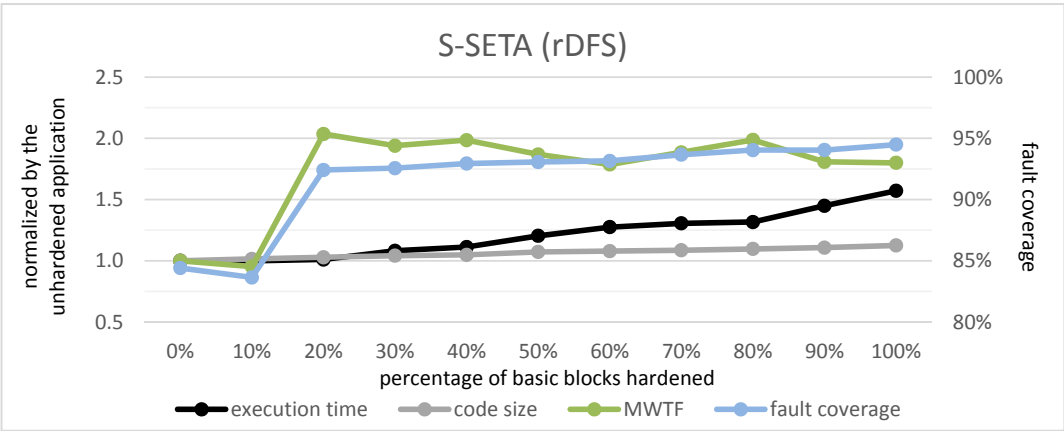


Fig. 6.28: Results for the recursive depth-first search (rDFS) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

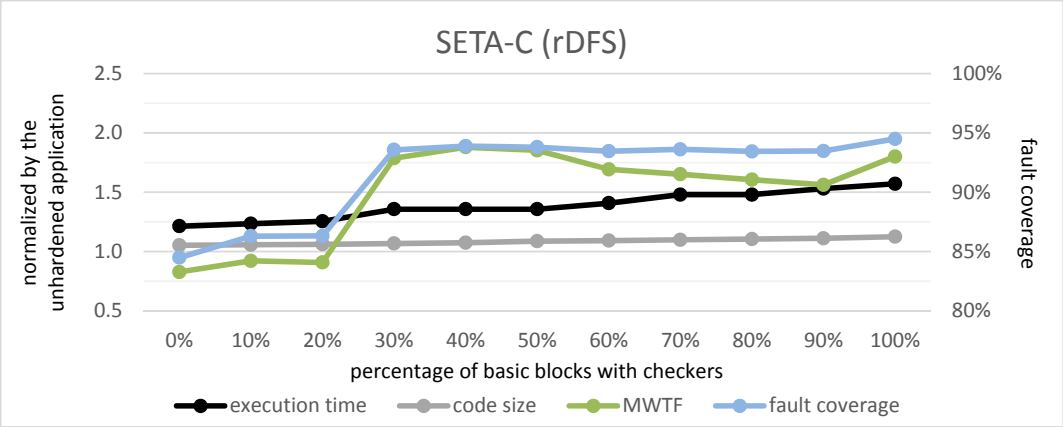


Fig. 6.29: Results for the recursive depth-first search (rDFS) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Fig. 6.30 shows the comparison between S-SETA and SETA-C for rDFS. The execution time, code size, MWTF, and fault coverage of S-SETA are normalized by the respective parameters of SETA-C. For 20% of the basic blocks hardened, there is a peak in the difference of the MWTF because SETA-C still does not provide an increase in the fault coverage. From that point on, we can see a slight advantage of S-SETA, due to its lower execution time overhead and similar fault coverage.

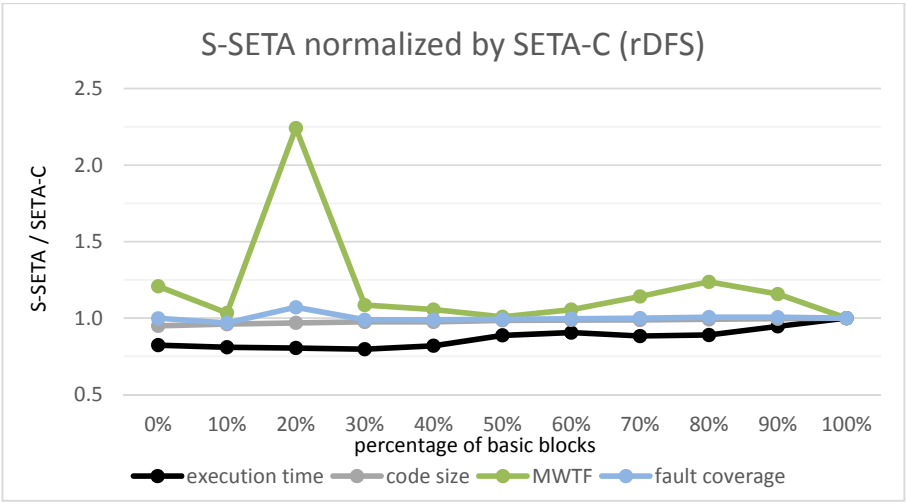


Fig. 6.30: Comparison between S-SETA and SETA-C for the recursive depth-first search (rDFS). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Fig. 6.31 presents the results for a depth-first search (sDFS) hardened by S-SETA, and Fig. 6.32 shows the results of the sDFS hardened by SETA-C. As for the other applications, the fault coverage and the MWTF provided by SETA-C are more or less constant. However, there are a little increase in the MWTF for a lower percentage of basic blocks with checkers due to the lower execution time overhead. With regards to S-SETA, we can see an earlier increase of the MWTF. The two largest basic blocks are in the beginning and end of the application, and both are very interactive, mainly the last one. Thus, many errors are detected when these two large basic blocks are hardened (S-SETA

20%) by the interaction of them through the *tunnel effect* with very low overheads (execution time of 1.02x and code size of 1.03x).

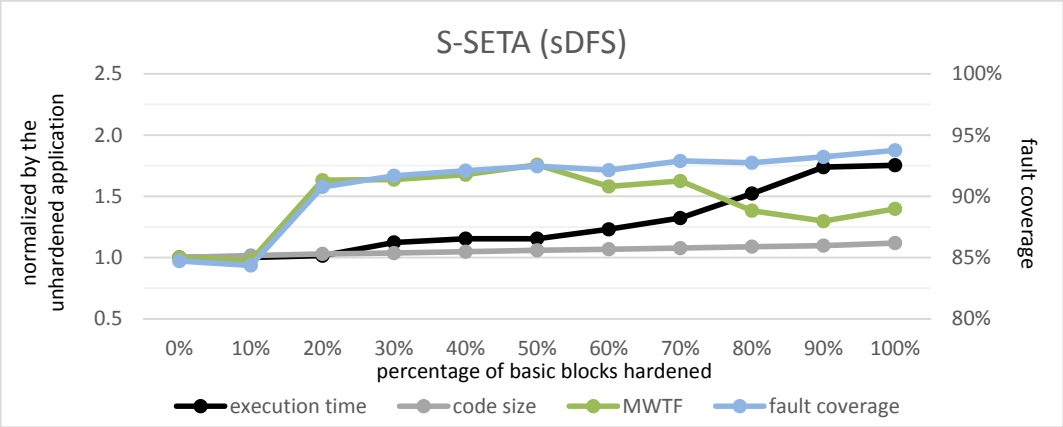


Fig. 6.31: Results for the sequential depth-first search (sDFS) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

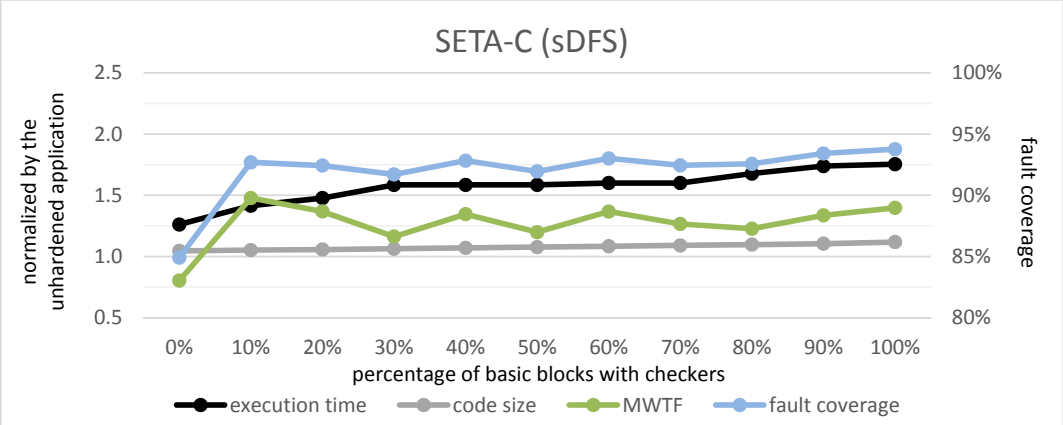


Fig. 6.32: Results for the sequential depth-first search (sDFS) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

The comparison between S-SETA and SETA-C for the sDFS presented in Fig. 6.33 shows a higher MWTF for S-SETA from 20% on. The techniques converge to SETA near 100%, which is why they have similar MWTF near that point. It is important to notice that the significantly lower overhead of S-SETA is the factor responsible for improving the MWTF.

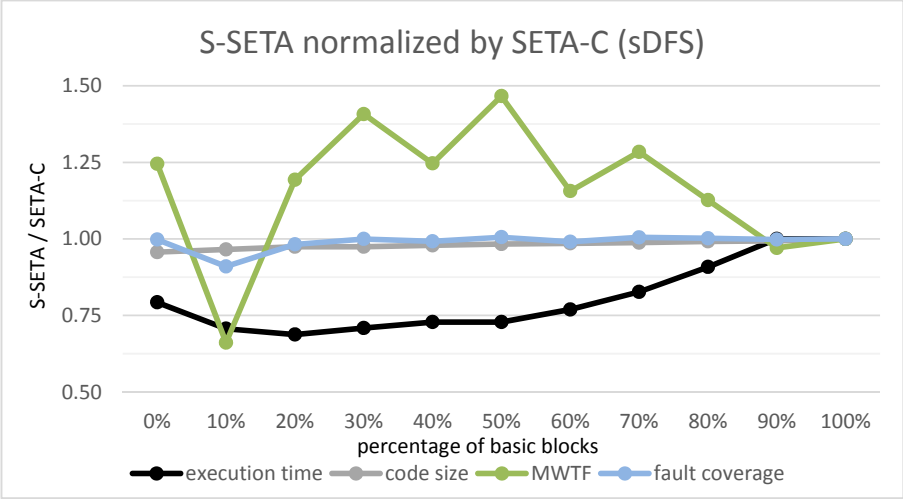


Fig. 6.33: Comparison between S-SETA and SETA-C for the sequential depth-first search (sDFS). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

The matrix multiplication (MM) has a basic block configuration (number of basic blocks, average size of the basic blocks, and percentage of basic blocks of one type) similar to the bubble sort. Therefore, in this case, we can also see a later increase in the MWTF for MM hardened by S-SETA, as shown in Fig. 6.34. When SETA-C is implemented (Fig. 6.35), one can notice the same configuration of all applications, an almost constant MWTF.

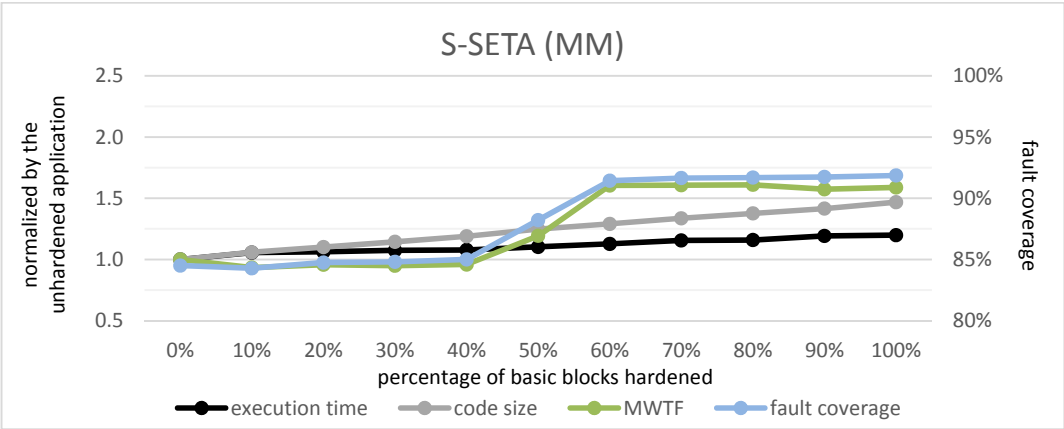


Fig. 6.34: Results for the matrix multiplication (MM) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

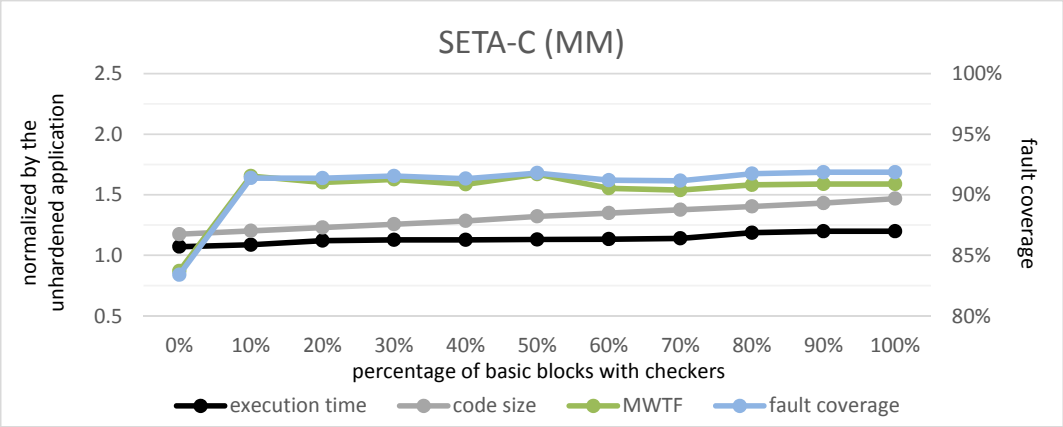


Fig. 6.35: Results for the matrix multiplication (MM) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

We can see by the comparison between S-SETA and SETA-C for MM in Fig. 6.36 that while S-SETA does not increase the fault coverage, SETA-C has a higher MWTF, although the higher overheads. However, from 60% of the basic blocks hardened on, S-SETA provides a higher MWTF until both techniques converge to SETA.

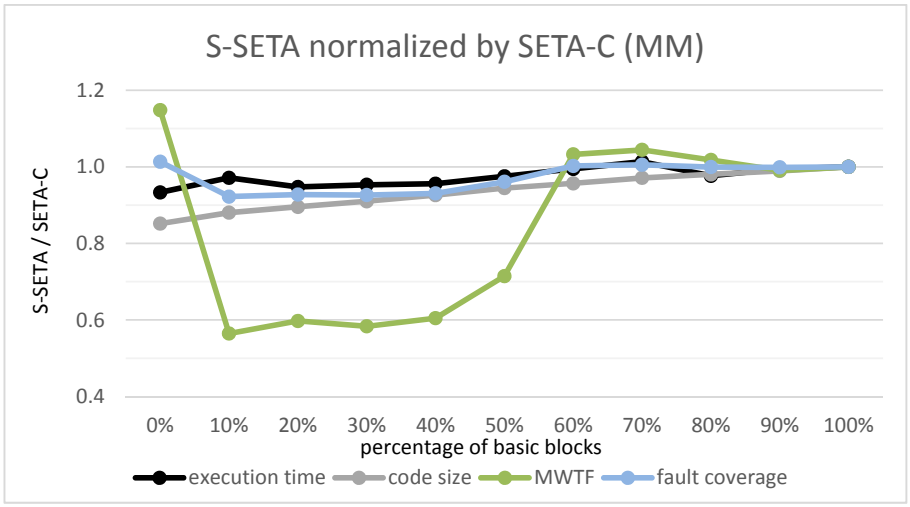


Fig. 6.36: Comparison between S-SETA and SETA-C for the matrix multiplication (MM). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

The run length encoding (RLE) is the application with the greatest number of basic blocks. It means that for the same percentage of basic blocks hardened, a greater absolute number of basic blocks was hardened. That explains why since 10% of the basic blocks hardened, both S-SETA (Fig. 6.37) and SETA-C (Fig. 6.38) reach a high fault coverage and MWTF. Once both approaches reach similar high MWTF, the comparison between them (presented in Fig. 6.39) does not vary much from 1.0.

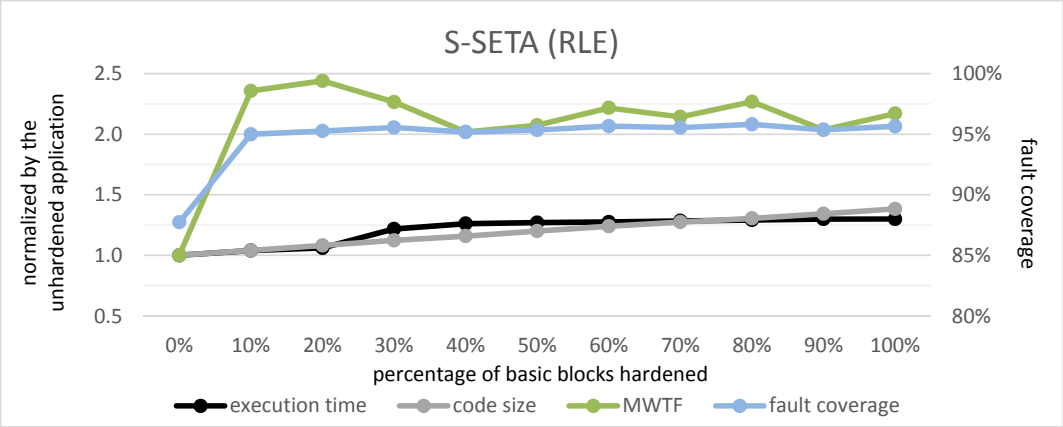


Fig. 6.37: Results for the run length encoding (RLE) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

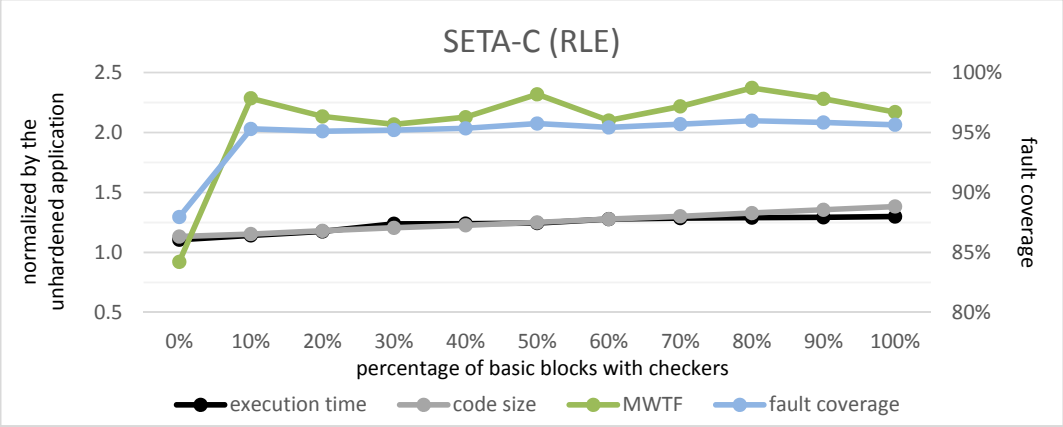


Fig. 6.38: Results for the run length encoding (RLE) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

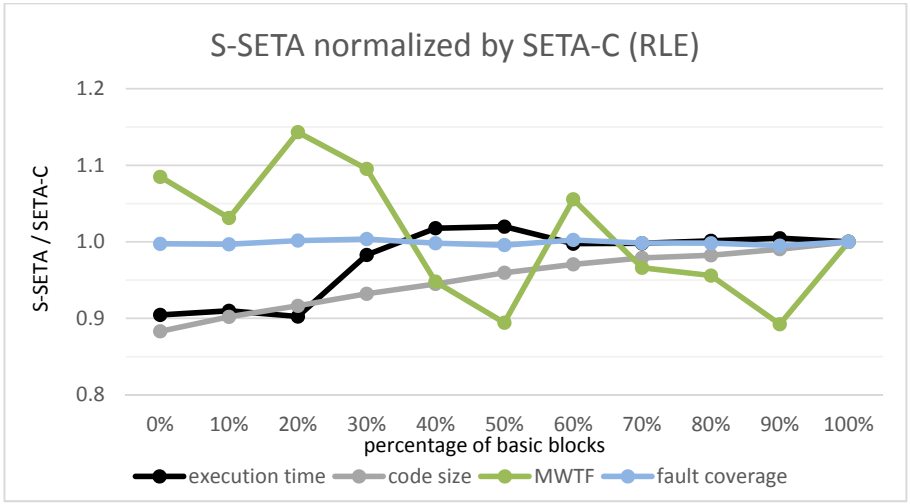


Fig. 6.39: Comparison between S-SETA and SETA-C for the run length encoding (RLE). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Although the summation (SUM) has a very different basic block configuration than BS and MM, it has a similar execution property. It consists of a loop controlled by a value stored in a register. Furthermore, it has very few basic blocks, so a higher percentage need to be hardened (in comparison to other benchmarks) in order to protect a greater number of basic blocks. Fig. 6.40 presents the execution time, code size, MWTF, and fault coverage for SUM hardened by S-SETA. And Fig. 6.41 presents the same parameters for SUM hardened by SETA-C. In the comparison between both selective hardening approaches (Fig. 6.42), one can notice a punctual higher execution time for S-SETA. It would not happen if the method to select the basic blocks were the same. However, there is a difference in the implementation that justify this result. S-SETA selects the basic blocks by their size. SETA-C select the basic block with more connections (predecessors and successors). This difference explains punctual higher execution time of S-SETA.

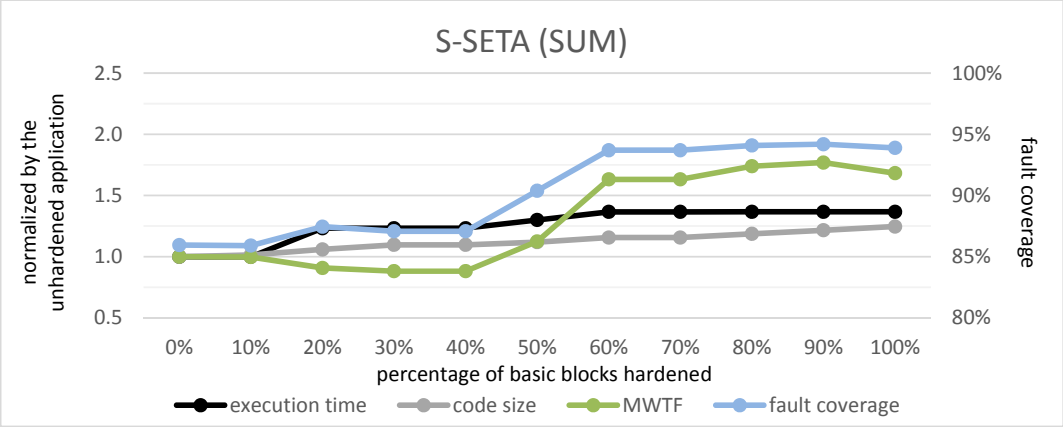


Fig. 6.40: Results for the summation (SUM) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

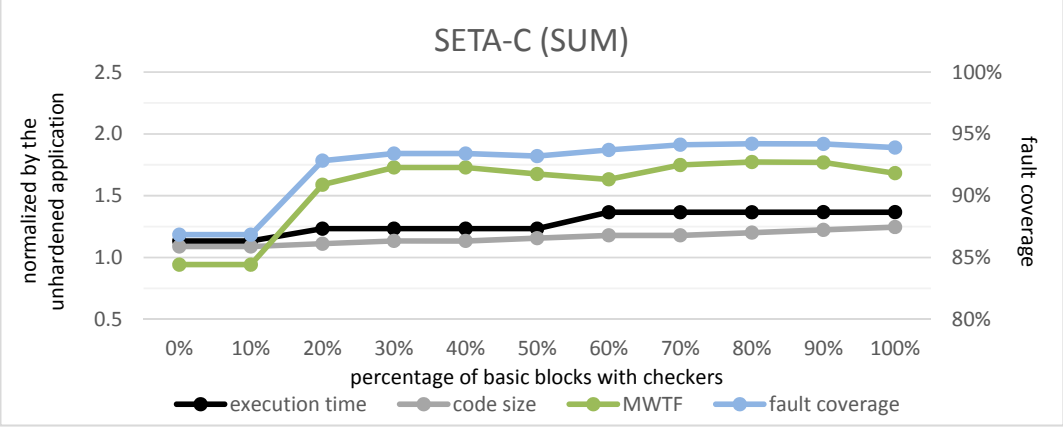


Fig. 6.41: Results for the summation (SUM) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

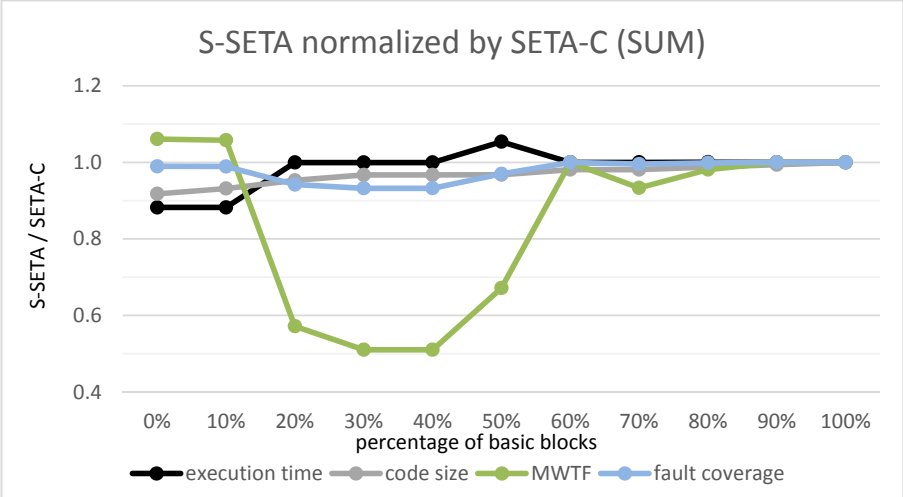


Fig. 6.42: Comparison between S-SETA and SETA-C for the summation (SUM). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Fig. 6.43 and Fig. 6.44 show the execution time, code size, MWTF, and fault coverage for the TETRA encryption algorithm hardened, respectively, by S-SETA and SETA-C. We can see that both approaches present the expected behavior. S-SETA increases the MWTF after a certain percentage of basic blocks is hardened, while SETA-C keeps a constant MWTF. We can see in the comparison presented in Fig. 6.45 that after S-SETA increases the MWTF, both approaches converge to SETA.

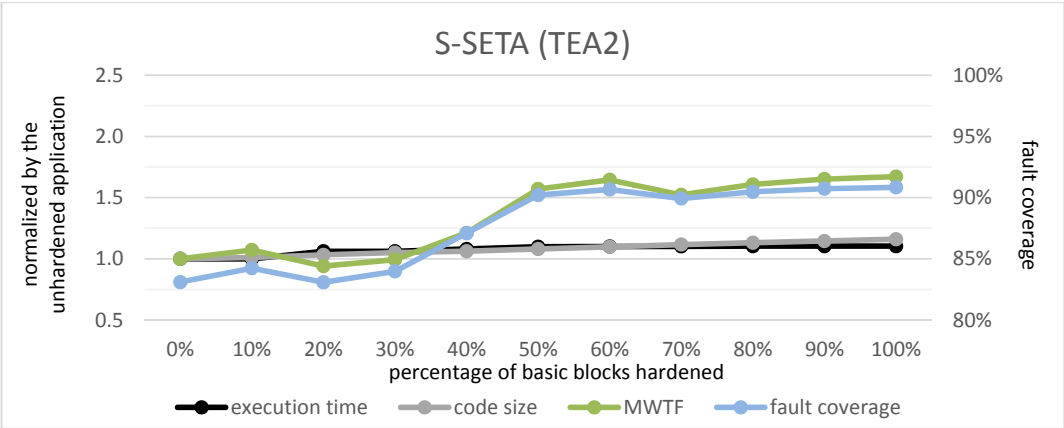


Fig. 6.43: Results for the TETRA encryption algorithm (TEA2) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

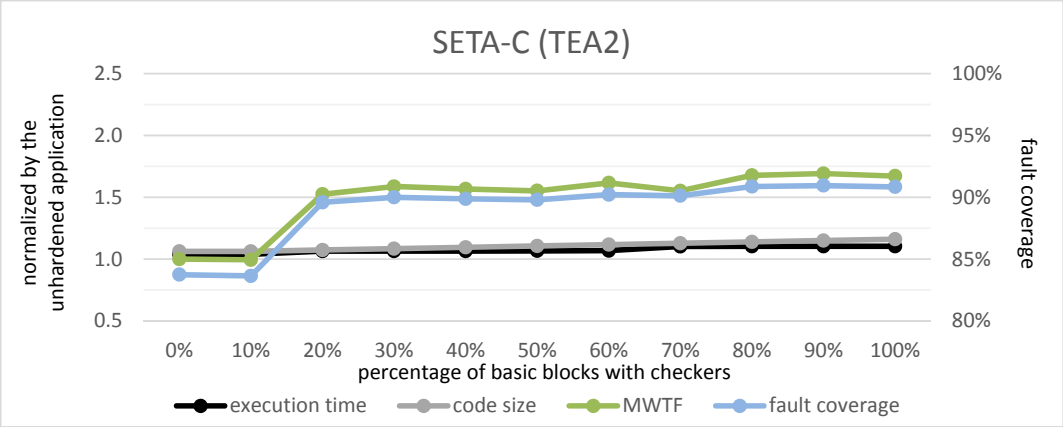


Fig. 6.44: Results for the TETRA encryption algorithm (TEA2) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

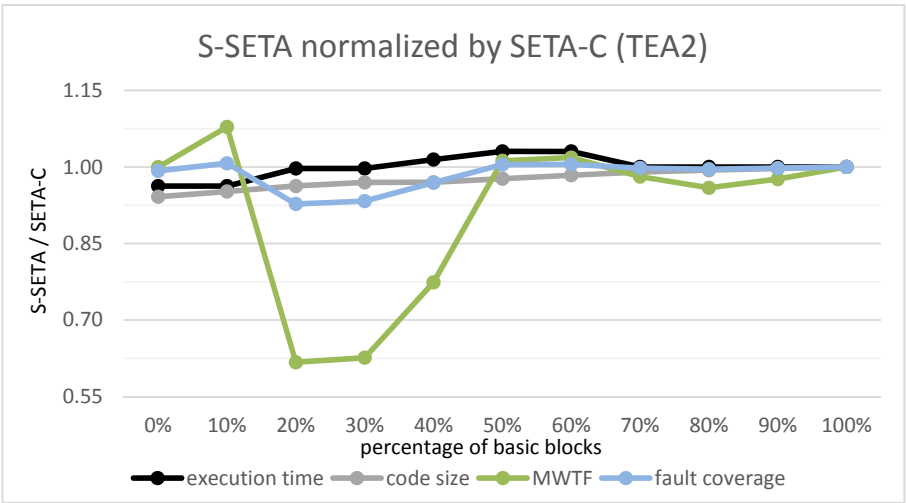


Fig. 6.45: Comparison between S-SETA and SETA-C for the TETRA encryption algorithm (TEA2). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Fig. 6.46 shows that S-SETA when hardening a Tower of Hanoi (TH) achieves a high fault coverage with a small percentage of basic blocks hardened (S-SETA 10%). Due to its negligible execution time overhead, the MWTF reaches the maximum value. With the increase in the percentage of basic blocks hardened, the fault coverage increases, but it is not enough to compensate the increase in the overheads, and that reduces the MWTF, converging to SETA. In Fig. 6.47, one can see the standard behavior of SETA-C, which presents an almost constant MWTF. In the comparison between S-SETA and SETA-C for TH, S-SETA always presents higher MWTF than SETA-C, being higher for a lower percentage due to the lower overhead and converging to SETA with the increase of the percentage of basic blocks hardened, as one can see in Fig. 6.48.

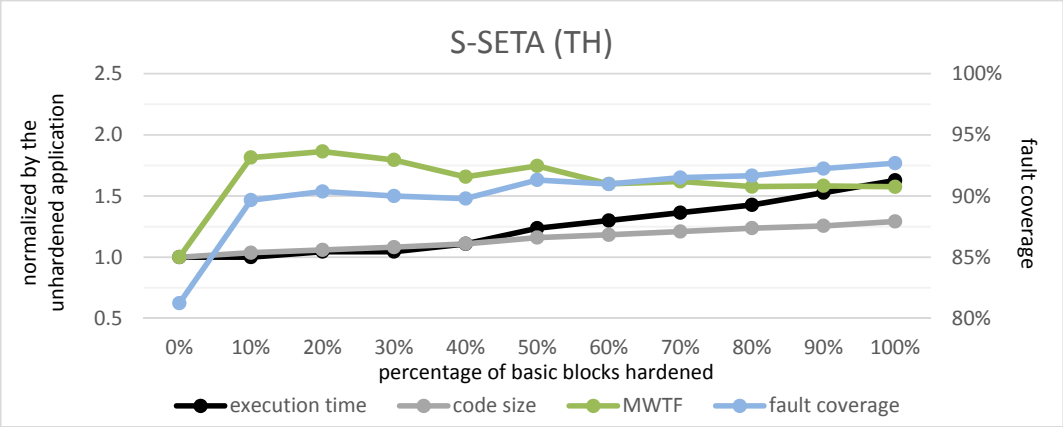


Fig. 6.46: Results for the Tower of Hanoi (TH) hardened by the S-SETA technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

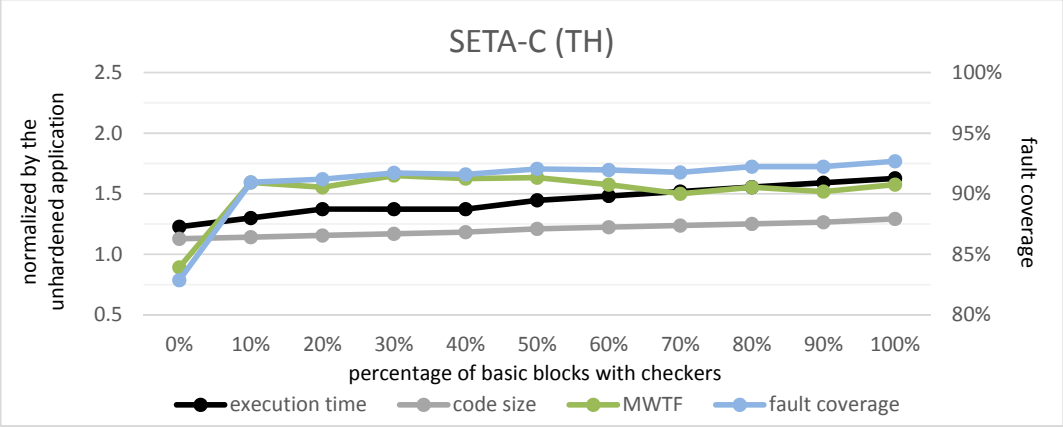


Fig. 6.47: Results for the Tower of Hanoi (TH) hardened by the SETA-C technique. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

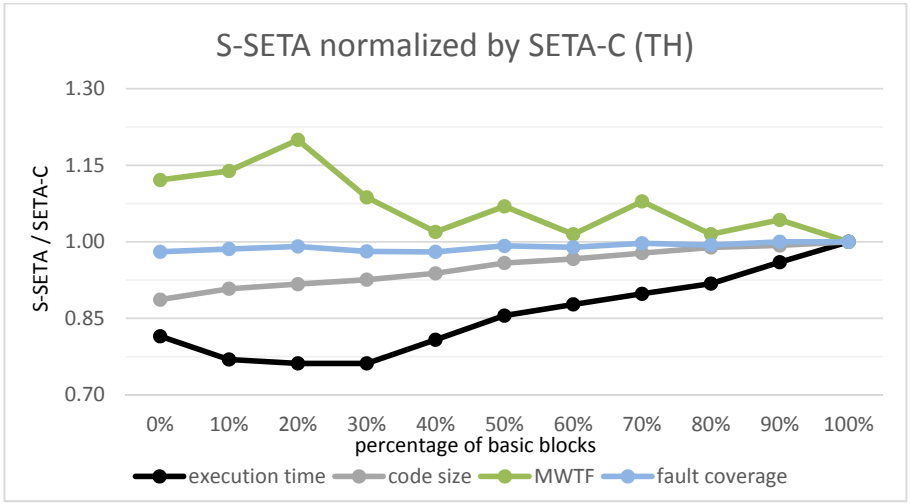


Fig. 6.48: Comparison between S-SETA and SETA-C for the Tower of Hanoi (TH). The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Fig. 6.49 shows the highest MWTF with its respective execution time, code size, and fault coverage for each application hardened either by S-SETA or SETA-C. The execution time, code size, and MWTF is presented normalized by the unhardened application, and the fault coverage is showed in percentage. As one can notice, for most of the applications, S-SETA reaches a higher MWTF. And for the other ones, the difference between the MWTF of S-SETA and SETA-C is small. Anyhow, it is necessary to test the selective hardening for control-flow techniques with a data-flow technique, which is discussed below.

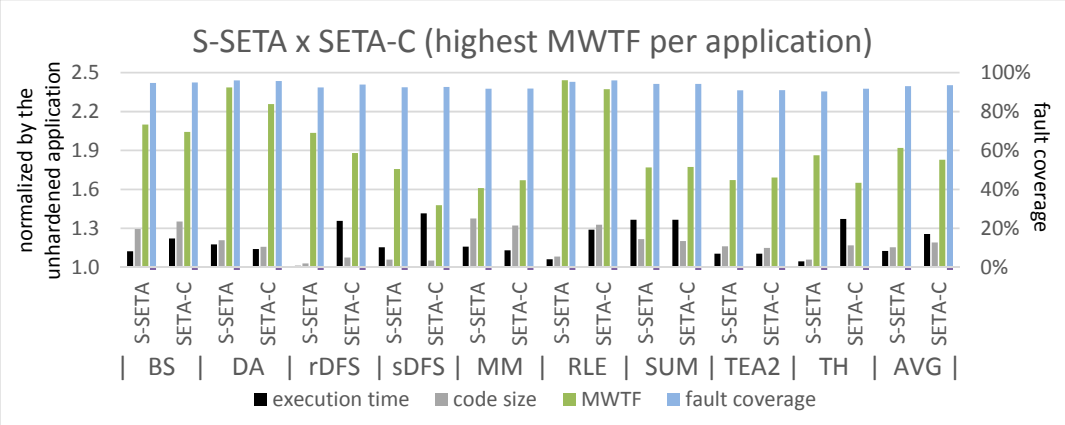


Fig. 6.49: Highest MWTF for the benchmarks hardened by the S-SETA or SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

Although the overheads, fault coverage, and MWTF of using VAR3+, and S-SETA or SETA-C are higher than only using a selective control-flow technique, the behavior for each application and selective hardening approach is similar. Thus, the same conclusions from using only S-SETA or SETA-C can be extended to VAR3+, S-SETA, and VAR3+, SETA-C. Figs. 6.50 to 6.76 present the results for all benchmarks hardened by VAR3+, S-SETA or VAR3+, SETA-C, and also the comparison between both selective hardening approaches applied together with VAR3+.

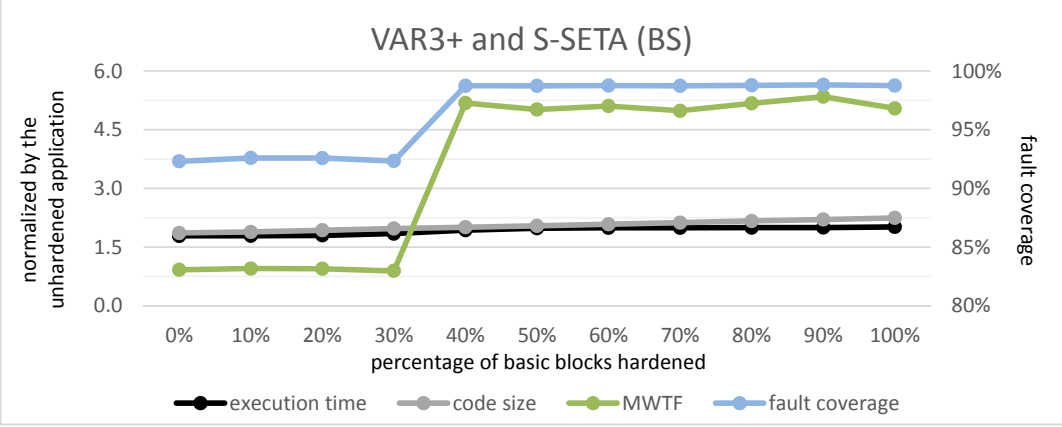


Fig. 6.50: Results for the bubble sort (BS) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

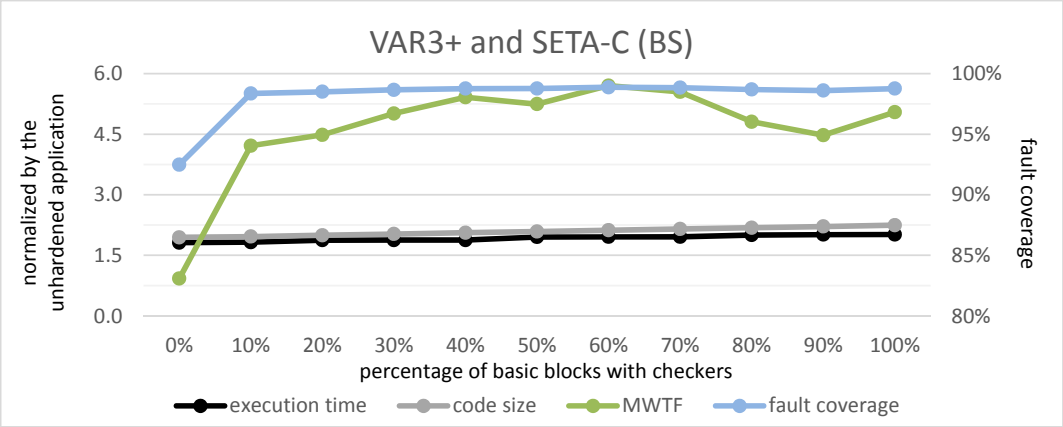


Fig. 6.51: Results for the bubble sort (BS) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

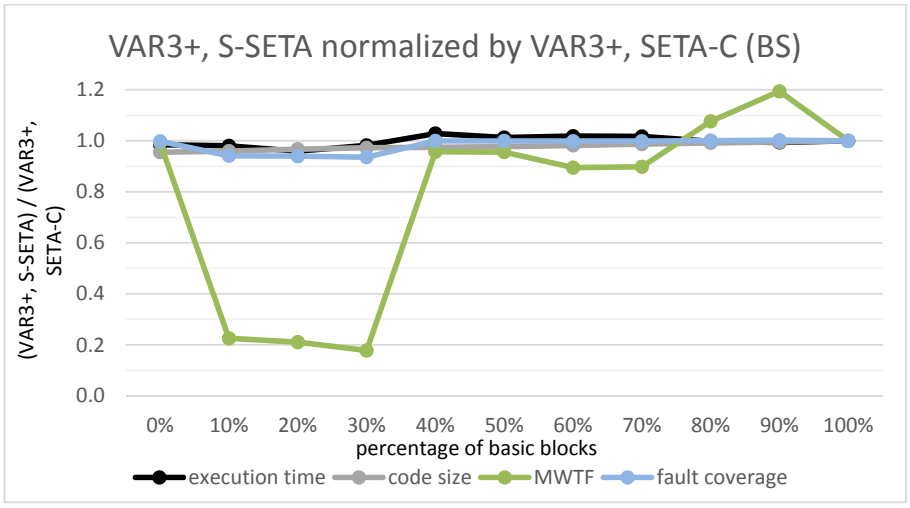


Fig. 6.52: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the BS. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

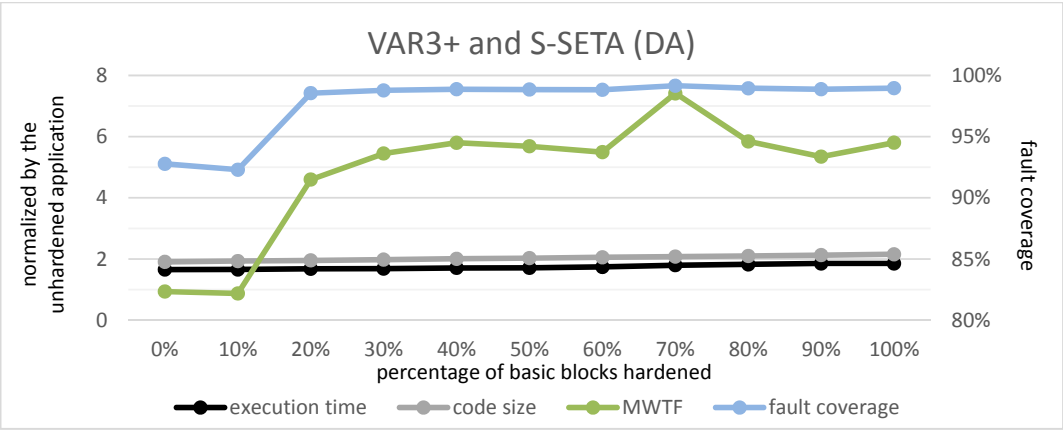


Fig. 6.53: Results for the Dijkstra's algorithm (DA) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

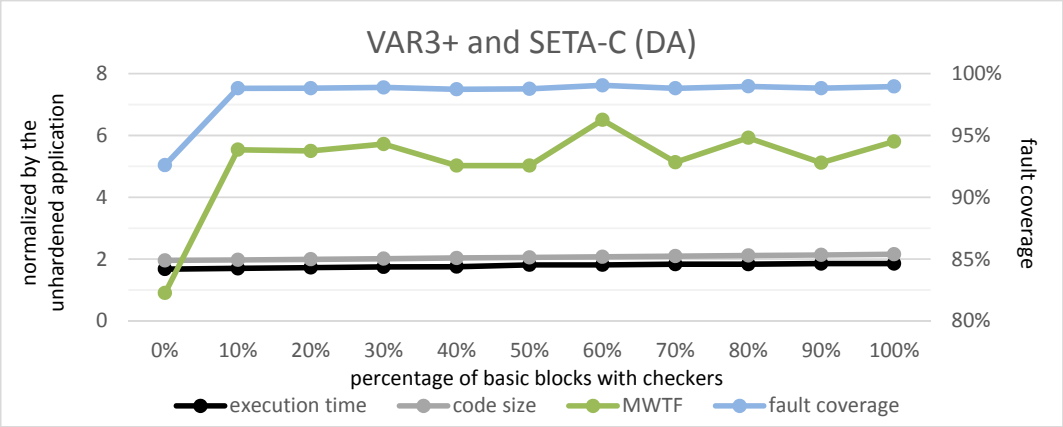


Fig. 6.54: Results for the Dijkstra's algorithm (DA) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

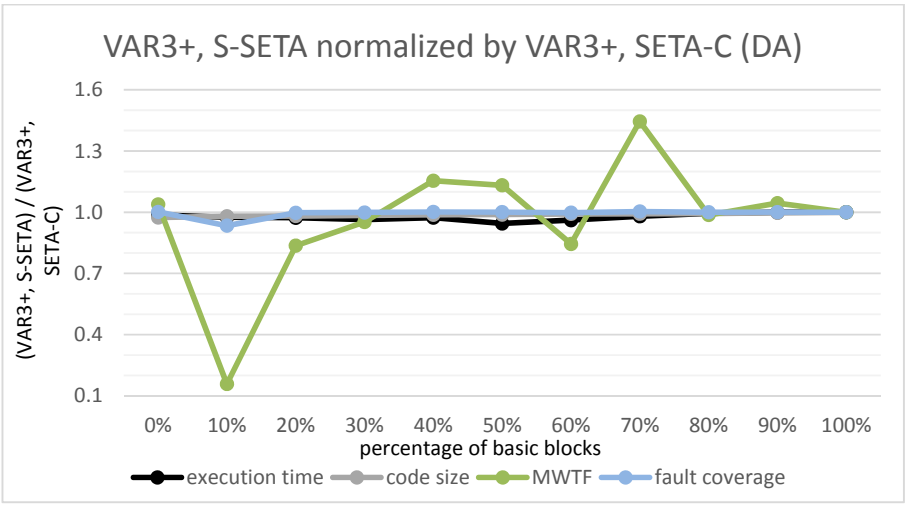


Fig. 6.55: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the DA. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

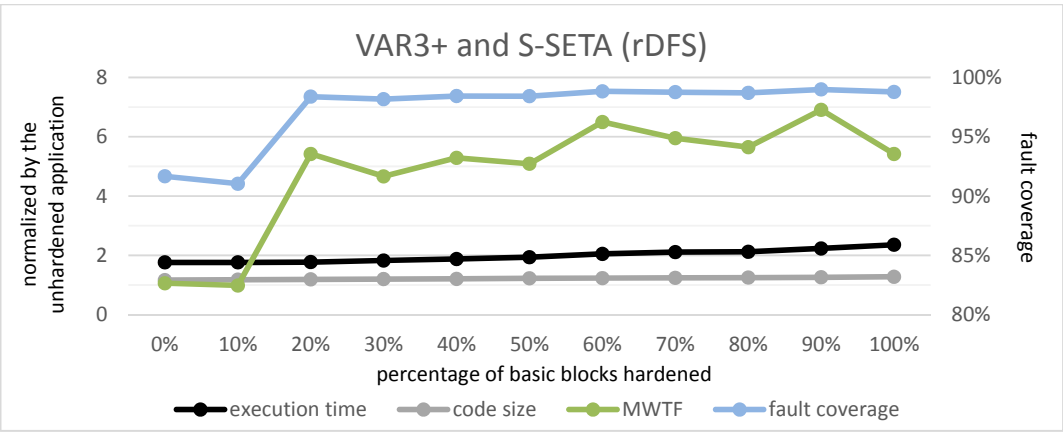


Fig. 6.56: Results for the recursive depth-first search (rDFS) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

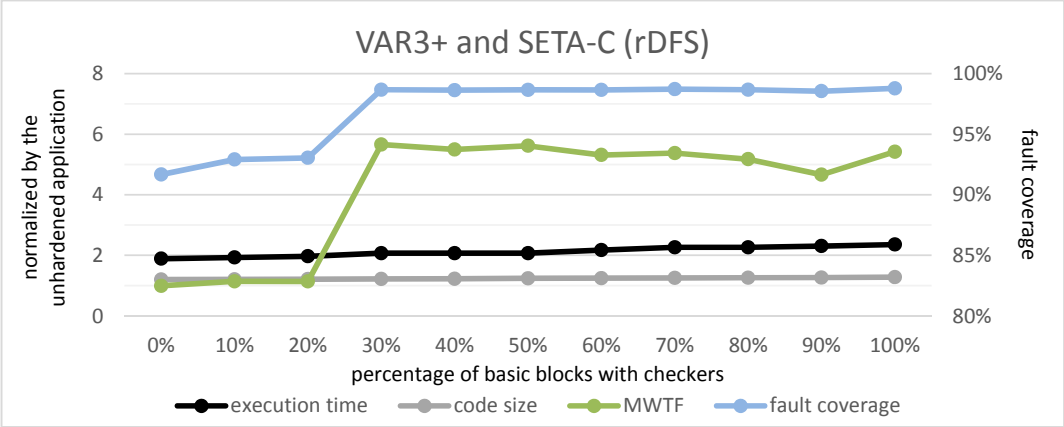


Fig. 6.57: Results for the recursive depth-first search (rDFS) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

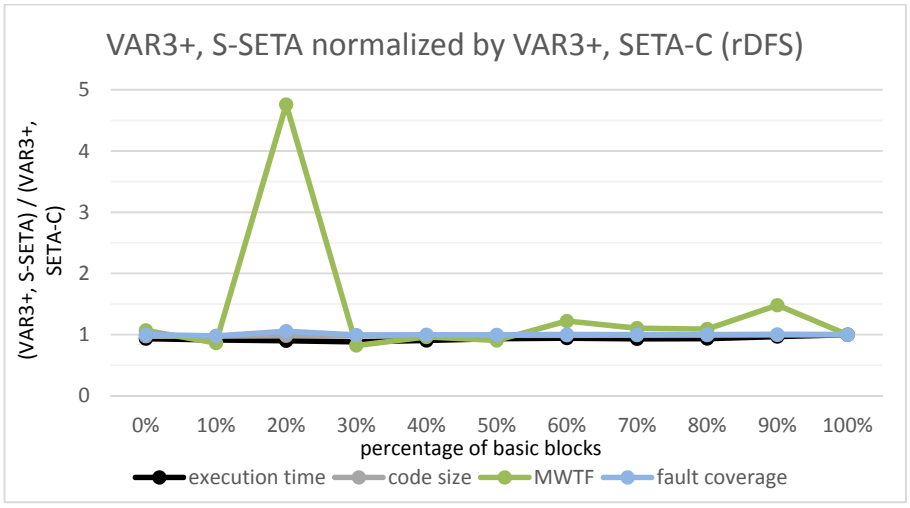


Fig. 6.58: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the rDFS. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

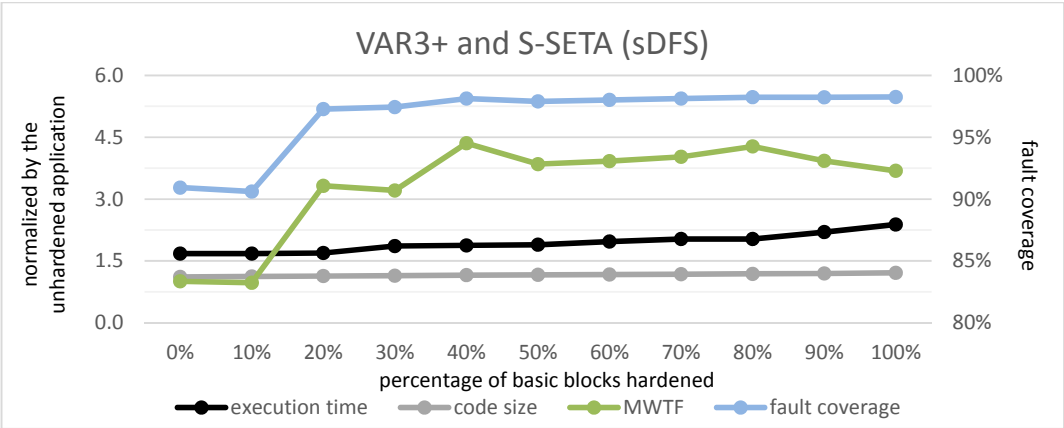


Fig. 6.59: Results for the sequential depth-first search (sDFS) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

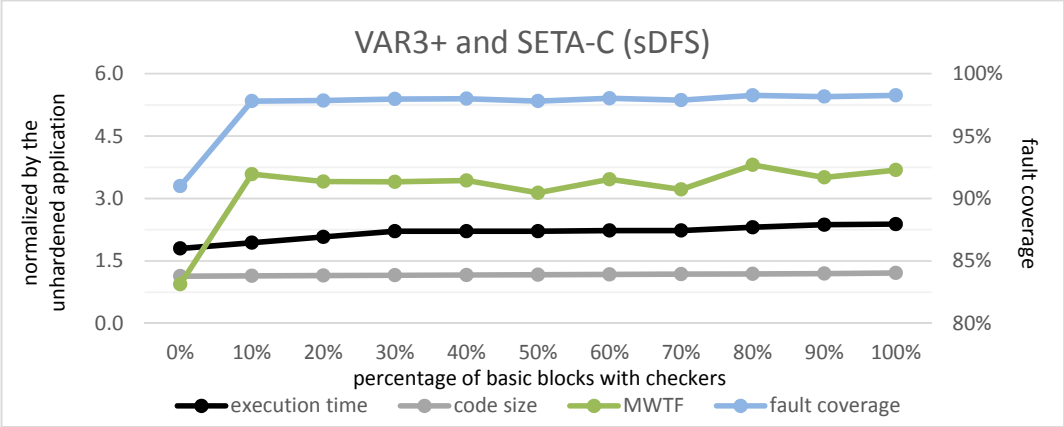


Fig. 6.60: Results for the sequential depth-first search (sDFS) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

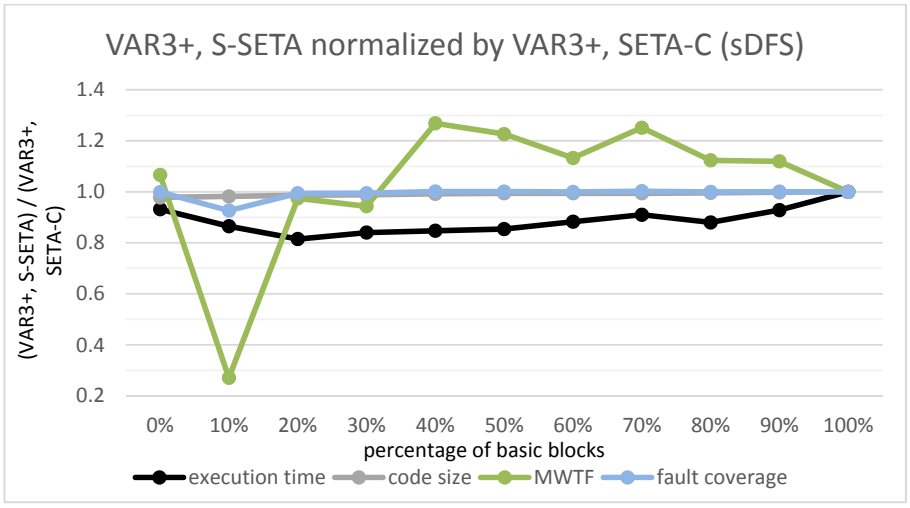


Fig. 6.61: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the sDFS. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

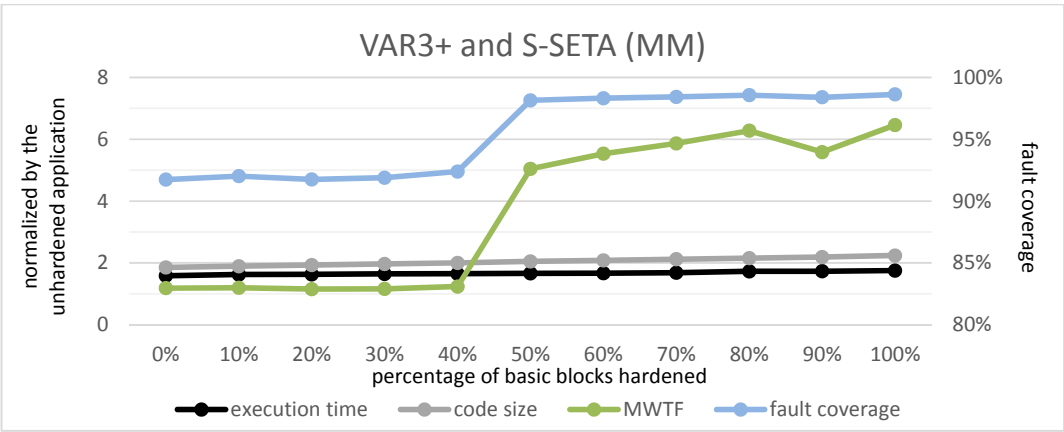


Fig. 6.62: Results for the matrix multiplication (MM) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

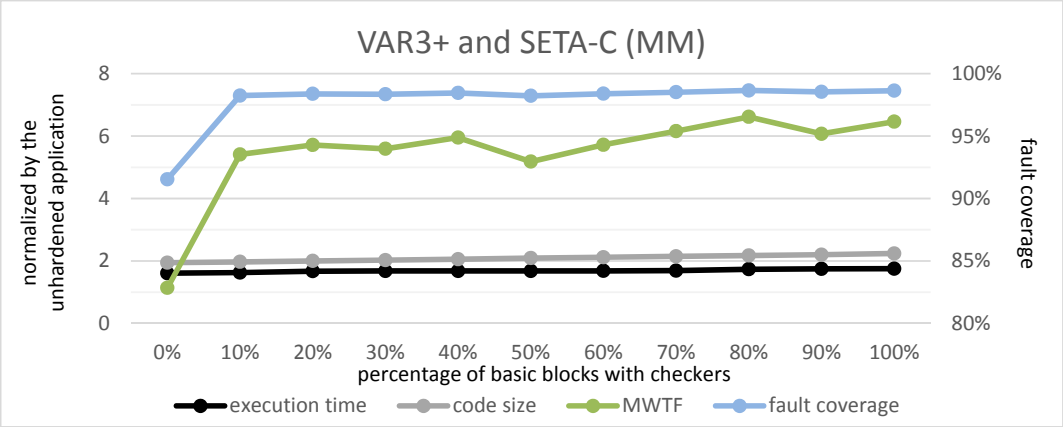


Fig. 6.63: Results for the matrix multiplication (MM) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

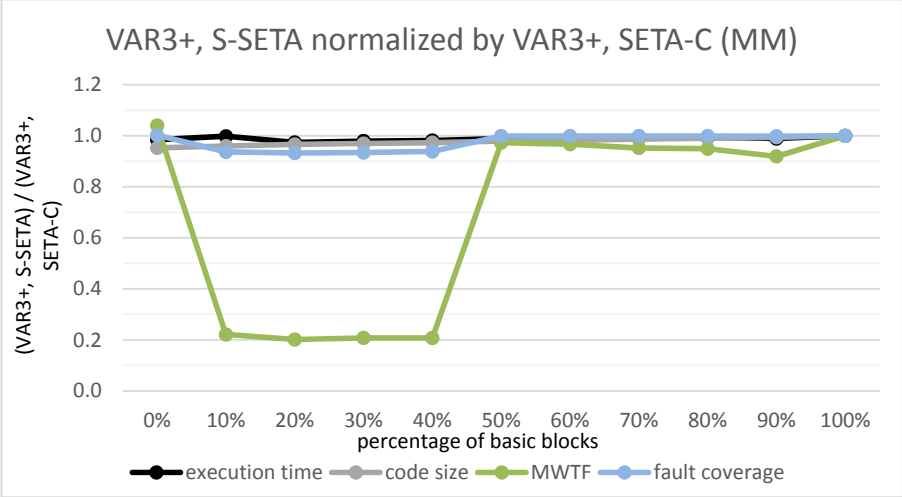


Fig. 6.64: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the MM. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

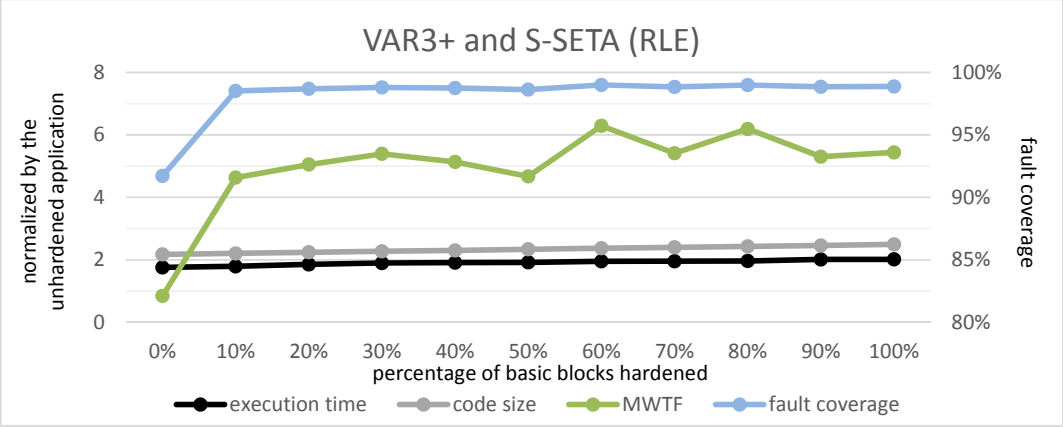


Fig. 6.65: Results for the run length encoding (RLE) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

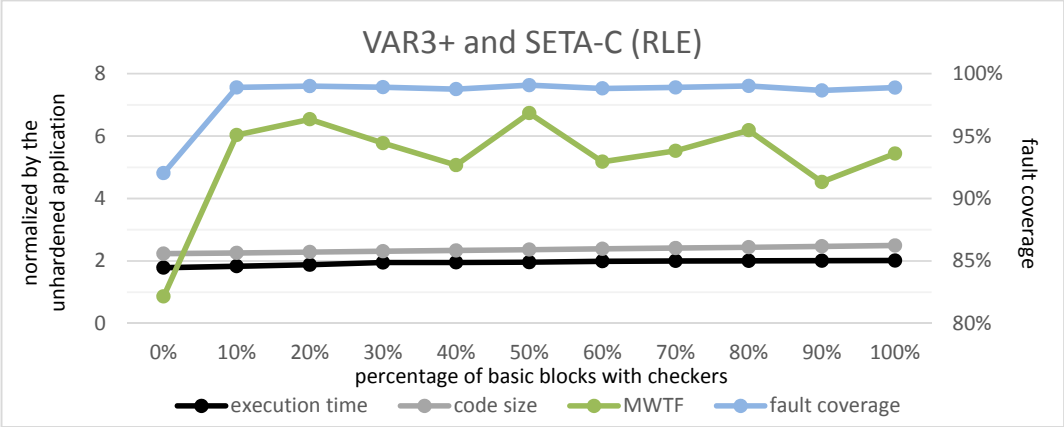


Fig. 6.66: Results for the run length encoding (RLE) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

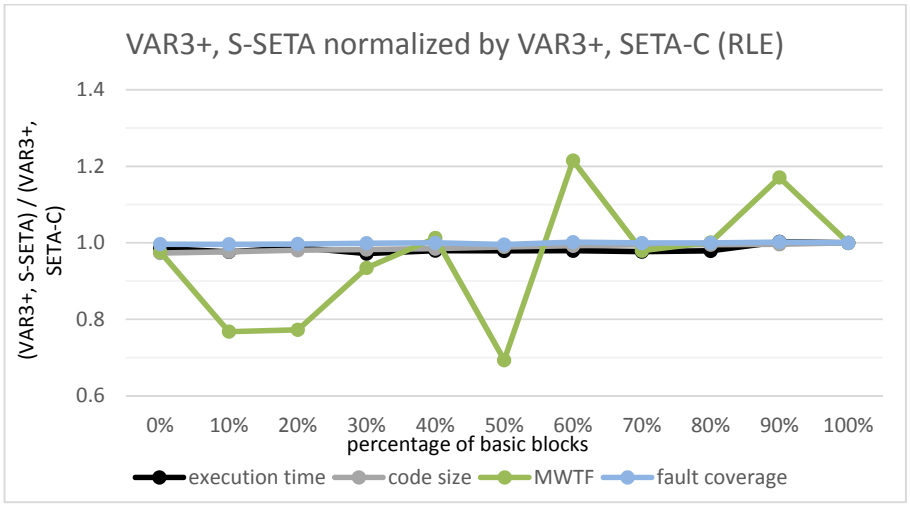


Fig. 6.67: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the RLE. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

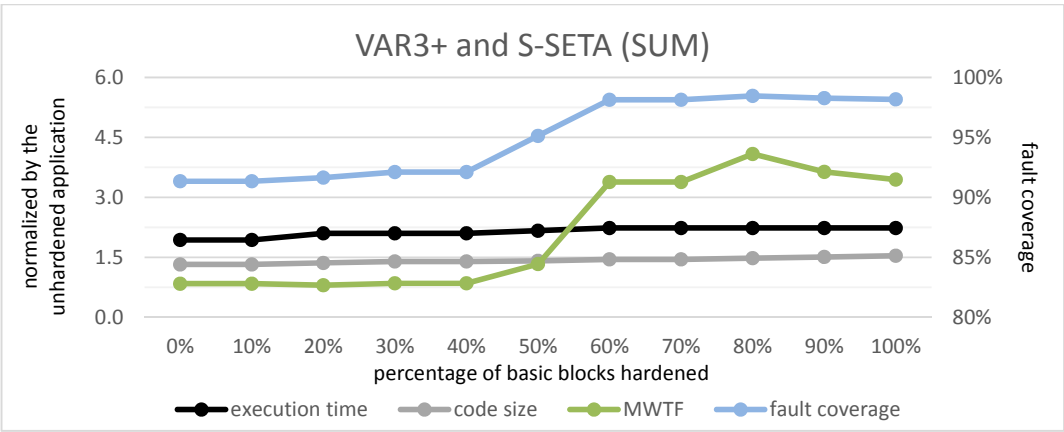


Fig. 6.68: Results for the summation (SUM) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

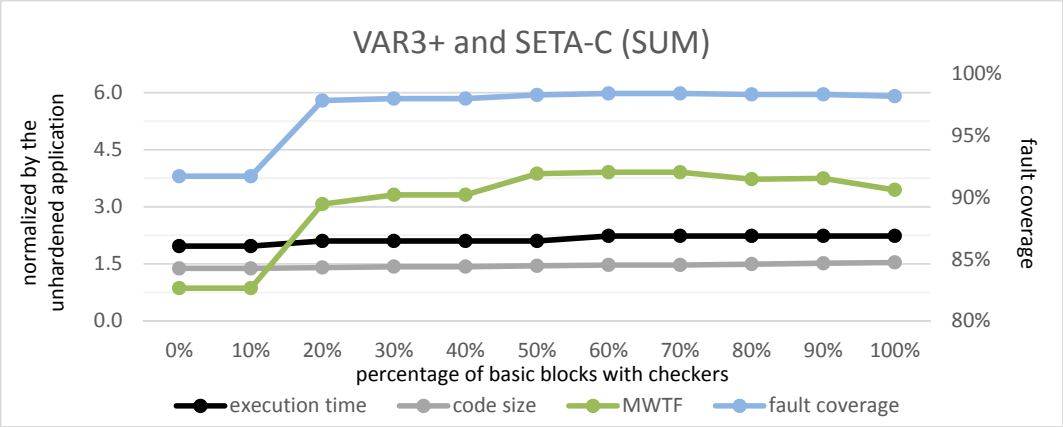


Fig. 6.69: Results for the summation (SUM) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

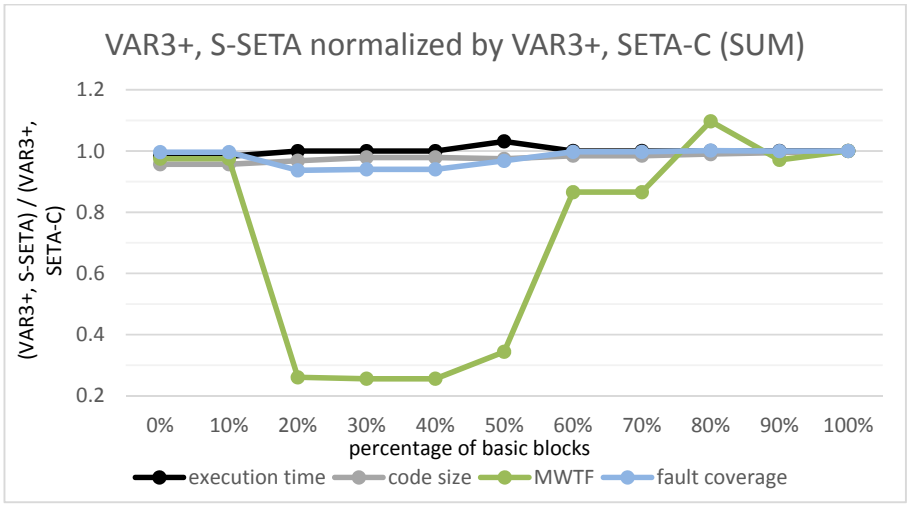


Fig. 6.70: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the SUM. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

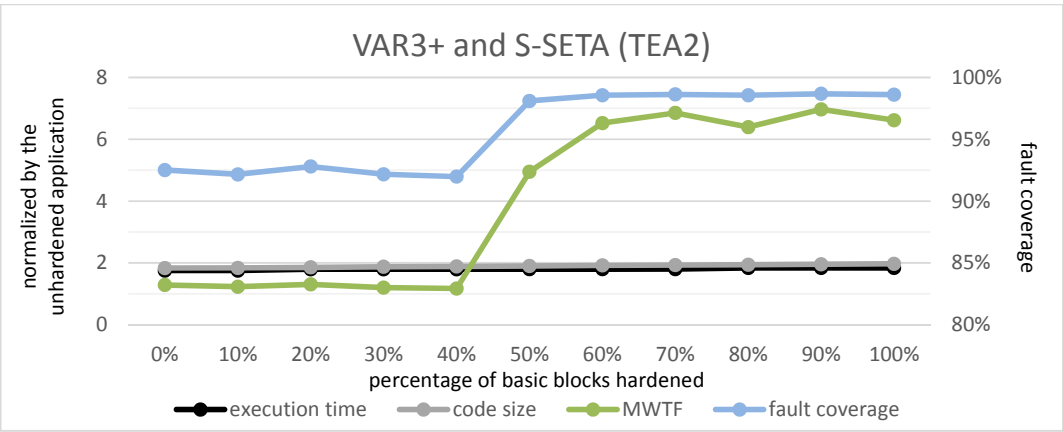


Fig. 6.71: Results for the TETRA encryption algorithm (TEA2) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

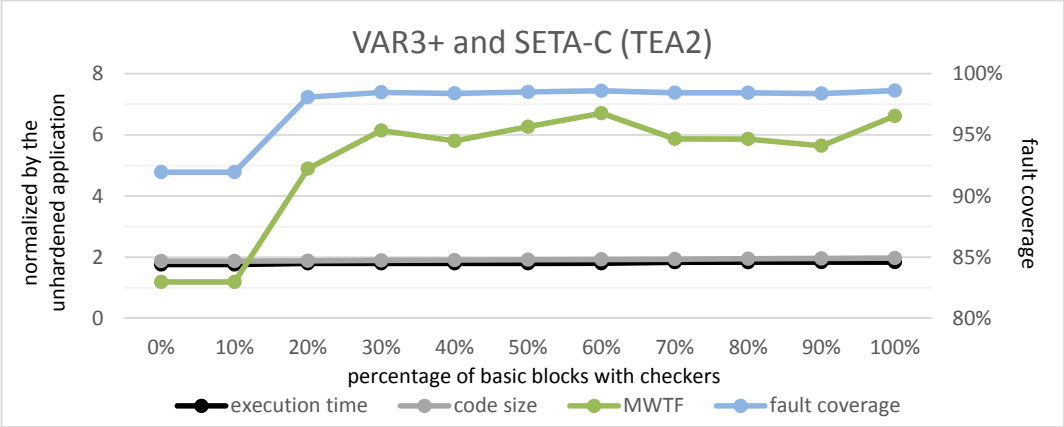


Fig. 6.72: Results for the TETRA encryption algorithm (TEA2) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

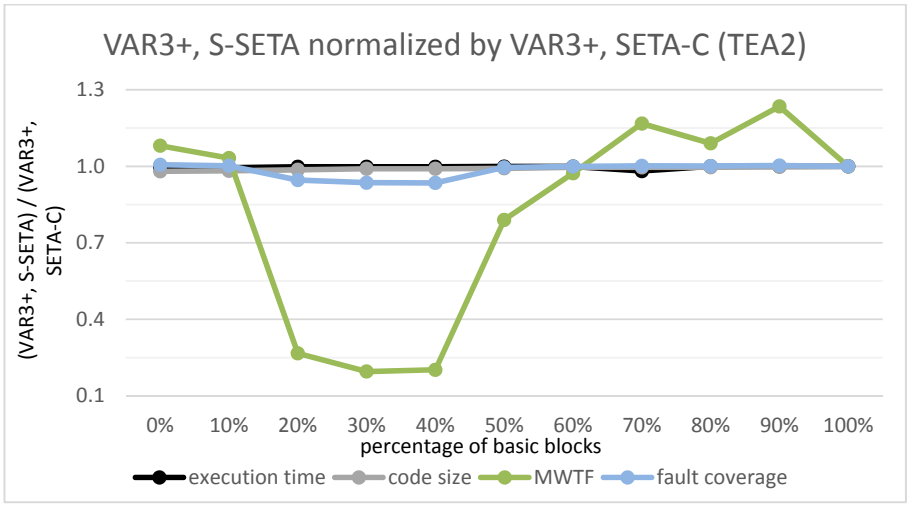


Fig. 6.73: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for TEA2. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

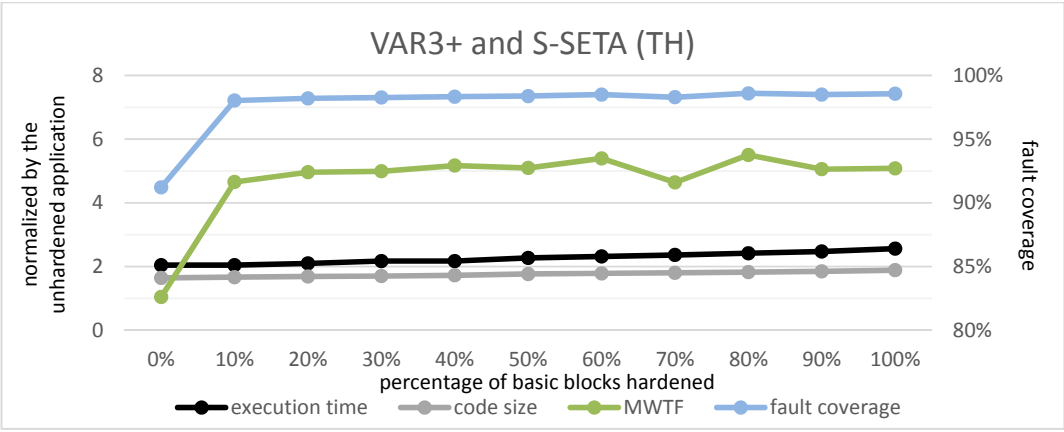


Fig. 6.74: Results for the Tower of Hanoi (TH) hardened by VAR3+ and S-SETA. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

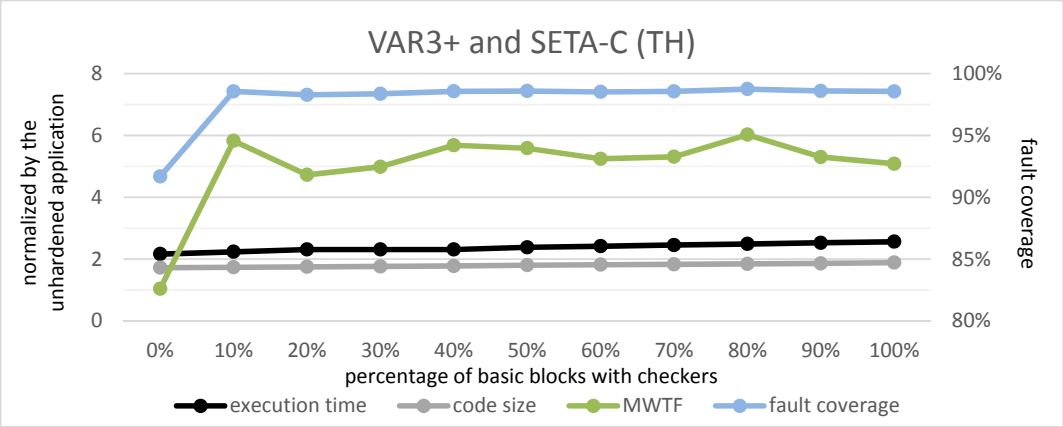


Fig. 6.75: Results for the Tower of Hanoi (TH) hardened by VAR3+ and SETA-C. The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

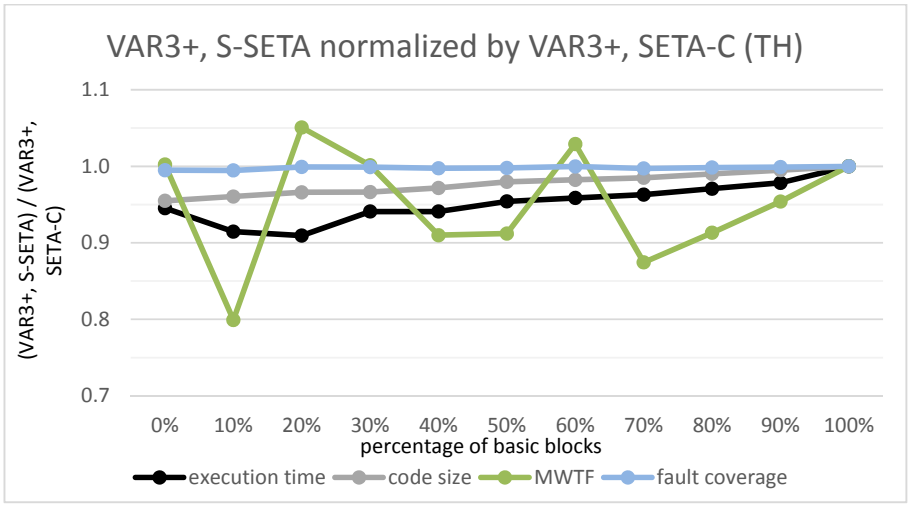


Fig. 6.76: Comparison between (VAR3+, S-SETA) and (VAR3+, SETA-C) for the TH. The results obtained with S-SETA are normalized by the ones obtained with SETA-C.

Fig. 6.77 shows the highest MWTF with its respective execution time, code size, and fault coverage for each application hardened by either VAR3+, S-SETA or VAR3+, SETA-C. The execution time, code size, and MWTF are presented normalized by the unhardened application, and the fault coverage is showed in percentage. As one can notice, VAR3+, S-SETA usually reaches a higher MWTF. For the other applications, the difference between the MWTF of VAR3+, S-SETA, and VAR3+, SETA-C is small. We can see a significant increase in the fault coverage and MWTF when a data-flow technique is applied together with S-SETA or SETA-C.

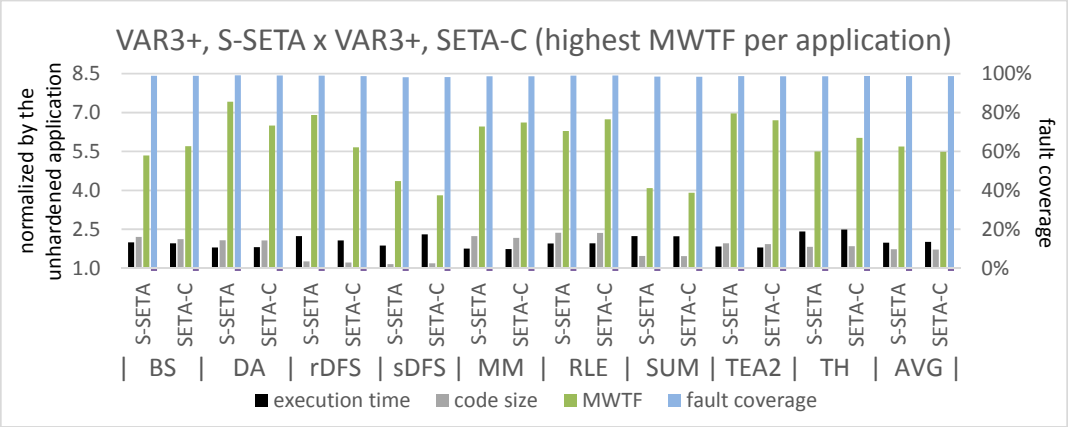


Fig. 6.77: Highest MWTF for the benchmarks hardened by (VAR3+, S-SETA) or (VAR3+, SETA-C). The execution time, code size, and MWTF are normalized by the unhardened application (left axis). The fault coverage is presented in percentage (right axis).

S-SETA presents better gains in the MWTF. On the other hand, SETA-C keeps the fault coverage high for a lower percentage of basic blocks with checkers. Nevertheless, it does not mean that SETA-C can provide reliability with lower overheads. Actually, it is the opposite. S-SETA presents significantly lower overheads than SETA-C. For example, sDFS hardened by SETA-C 10% presents 1.42x of execution time, while S-SETA 70% presents 1.32x. Thus, S-SETA is a better solution to meet time or energy constraints. Furthermore, it achieves a general higher MWTF due to its similar fault coverage and lower overheads. It is possible to combine S-SETA with SETA-C, S-SETA would select the basic blocks that will be hardened, and then, SETA-C would select which of the hardened basic blocks should receive a checker. This combination is an interesting approach to be evaluated in future work.

Table 6.3 presents selected data from all the selective hardening approaches presented above. The data selected respect some specific constraints. However, it is necessary cautiousness when using these results. It is important to take into account that these results are for the miniMIPS processor using a specific fault injection methodology. Adjusts may be necessary depending the target processor and the real rate of upsets over time affecting the system. Furthermore, it does not give the best options for every scenario, we just point out some selected cases. For example, when using VAR3+ with S-SETA, it may be necessary to protect around 40% of the basic blocks to achieve a fault coverage of at least 98%. The *X* means that the constraint is unachievable. For example, for many applications, it is not possible to reach more than 92% of fault coverage with only data-flow techniques.

Table 6.3: Summary of selective hardening. Fault coverage (FC) showed in percentage, execution time (ET) presented normalized by the unhardened application.

S-VAR (% of hardened registers)									
LIMIT	**BS**	**DA**	**rDFS**	**sDFS**	**MM**	**RLE**	**SUM**	**TEA2**	**TH**
FC 88%	23%	0%	44%	67%	46%	13%	40%	63%	36%
FC 90%	31%	57%	56%	67%	46%	38%	60%	75%	64%
FC 92%	46%	71%	X	X	X	63%	X	75%	X
ET 1.25x	23%	43%	33%	50%	38%	38%	40%	38%	18%
ET 1.50x	23%	57%	44%	67%	38%	38%	40%	50%	45%
ET 1.75x	38%	100%	78%	100%	58%	75%	40%	100%	55%
S-VAR, SETA (% of hardened registers)									
limit	**BS**	**DA**	**RDFS**	**SDFS**	**MM**	**RLE**	**SUM**	**TEA2**	**TH**
FC 94%	0%	0%	11%	17%	38%	0%	20%	63%	9%
FC 96%	31%	29%	56%	67%	46%	25%	60%	63%	27%
FC 98%	46%	71%	67%	83%	54%	50%	80%	75%	91%
ET 1.4x	23%	43%	X	X	38%	38%	20%	50%	X
ET 1.7x	23%	57%	11%	X	46%	38%	40%	50%	18%
ET 2.0x	77%	100%	33%	50%	100%	75%	40%	100%	36%
S-SETA (% of BBs hardened)									
limit	**BS**	**DA**	**RDFS**	**SDFS**	**MM**	**RLE**	**SUM**	**TEA2**	**TH**
FC 90%	40%	20%	20%	20%	60%	10%	50%	50%	20%
FC 92%	50%	30%	20%	40%	X	10%	60%	X	90%
FC 94%	50%	60%	80%	X	X	10%	80%	X	X
ET 1.1x	40%	50%	30%	20%	50%	10%	10%	100%	30%
ET 1.2x	60%	70%	50%	50%	100%	20%	10%	100%	40%
ET 1.3x	100%	100%	60%	60%	100%	100%	50%	100%	60%
SETA-C (% of BBs with checkers)									
limit	**BS**	**DA**	**RDFS**	**SDFS**	**MM**	**RLE**	**SUM**	**TEA2**	**TH**
FC 90%	10%	10%	30%	10%	10%	10%	20%	30%	10%
FC 92%	10%	10%	30%	10%	X	10%	20%	X	50%
FC 94%	20%	10%	X	X	X	10%	70%	X	X
ET 1.1x	0%	10%	X	X	10%	X	X	100%	X
ET 1.2x	40%	60%	X	X	100%	20%	10%	100%	X
ET 1.3x	100%	100%	20%	0%	100%	100%	50%	100%	30%
VAR3+, S-SETA (% of BBs hardened)									
	BS	**DA**	**RDFS**	**SDFS**	**MM**	**RLE**	**SUM**	**TEA2**	**TH**
FC 92%	0%	0%	20%	20%	10%	10%	30%	0%	10%
FC 95%	40%	20%	20%	20%	50%	10%	50%	50%	10%
FC 98%	40%	40%	20%	40%	50%	10%	60%	50%	10%
ET 1.7x	X	30%	X	20%	70%	X	X	X	X
ET 1.9x	30%	100%	40%	50%	100%	30%	X	100%	X
ET 2.1x	100%	100%	60%	80%	100%	100%	40%	100%	20%
VAR3+, SETA-C (% of BBs with checkers)									
	BS	**DA**	**RDFS**	**SDFS**	**MM**	**RLE**	**SUM**	**TEA2**	**TH**
FC 92%	0%	0%	10%	10%	10%	0%	20%	0%	10%
FC 95%	10%	10%	30%	10%	10%	10%	20%	20%	10%
FC 98%	10%	10%	30%	30%	10%	10%	30%	20%	10%
ET 1.7x	0%	10%	X	X	70%	X	X	X	X
ET 1.9x	40%	100%	0%	0%	100%	20%	X	100%	X
ET 2.1x	100%	100%	50%	20%	100%	100%	50%	100%	X

Based on the results, we present some conclusions:

- We observed that the applications can reach a maximum fault coverage of around 98-99%. However, it is very complicated to predict the fault coverage for each level of selective hardening by online analyzing the code. There are too many variables that influence on that. Any change in one of the following items may cause a significant chance in the fault coverage and overheads
    - It depends on what task the application performs
    - It depends on how the application was implemented
    - It depends on how it was compiled
        - An application compiled with no optimizations will have few registers that perform most of the calculations. These registers are much more critical than the others because they are much more utilized. On the other hand, it means that the overheads when protecting them are higher
        - The same application compiled with optimizations would have completely different results, with a more distributed use of registers
        - If the application was implemented in assembly, the fault coverage and overheads will depend on how the application was implemented
- One characteristic that was possible to observe regards the checking rule C6 and was discussed in the previous chapter. When the average number of times that the basic blocks are executed is high, there is a higher chance of errors affecting branches due to upsets in the registers used by these branches. Therefore, the checking rule C6 increases significantly the fault coverage in such cases.

The difficulty in finding patterns in the code due to the high number of parameters and the high variability they cause in the overheads and fault coverage of a selectively hardened application makes interesting the search for other approaches to find the best trade-offs between fault coverage and overheads when using selective hardening. In the following section, we introduce a method to extrapolate the results using a small set of results as input, and provide an accurate overall picture of the application reliability and overheads.

6.3 Selective data-flow technique and selective control-flow technique

The use of selective hardening is possible to be done at the same time in both data and control-flow techniques. Thus, it would be possible to get even better trade-offs between reliability and performance than only applying selective hardening to one of the techniques. However, it is difficult to find patters to use in the selective hardening because the application is highly variable due to the many parameters that influence its fault coverage and overheads. Furthermore, exploring all the possibilities for every application is infeasible, mainly due to the need to perform fault injection campaigns in all possible selective hardened versions. The test of each version takes from several hours to weeks in the RTL level, depending on the application. This means that testing all possibilities for selective hardening would take from weeks to years. A less time-consuming manner to find the points of interest, or at least to indicate the most promising areas, can speed up the time needed to protect and evaluate an application.

6.3.1 Methodology and implementation

An extrapolation of the results was performed using a linear interpolation of four hyperbolic tangent regressions. Each hyperbolic tangent (like the one in Eq. 6.2) represents one extreme of the S-VAR, S-SETA combination. Firstly, let us define these four extremes:

- **S-VAR(x), S-SETA(0)**: all the results with no basic block hardened and variable hardening of the registers
- **S-VAR(x), S-SETA(1)**: all the results with all basic blocks hardened and variable hardening of the registers
- **S-VAR(0), S-SETA(x)**: all the results with no register hardened and variable hardening of the basic blocks
- **S-VAR(1), S-SETA(x)**: all the results with all registers hardened and variable hardening of the basic blocks.

For example, when S-SETA is 0 (has no basic block hardened), x is the percentage of registers hardened, and y the fault coverage. Then, we fit Eq. 6.2 with the data from fault injection with respect to S-VAR(x), S-SETA(0). The same process is done for the other three extremes. Finally, the four equations are linearly interpolated in order to get a surface of fault coverages for all possible variation of the number of registers hardened or basic blocks hardened.

$$y = A \cdot tanh(B \cdot x + C) + D \qquad \text{(Eq. 6.2)}$$

The method for extrapolating the results was implemented in the Matlab. The function *fit* was used with the Levenberg-Marquardt algorithm to fit the fault coverage obtained from simulation in the Eq. 6.2. Fig. 6.78 shows an example using the data from the extreme S-VAR(x), S-SETA(1) of the bubble sort (BS).

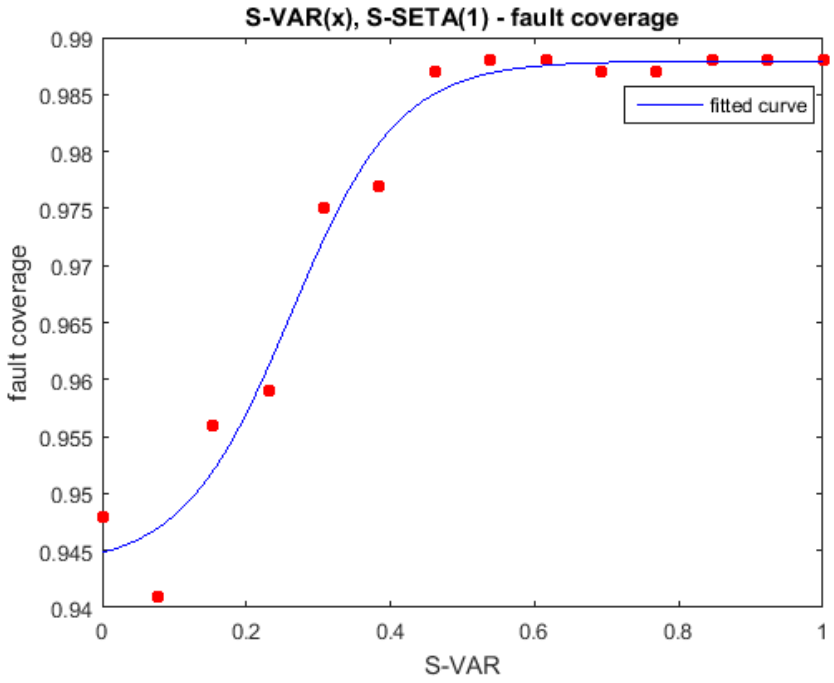


Fig. 6.78: Estimated fault coverages for BS hardened by S-VAR, S-SETA.

All the four extremes defined in the beginning of this subsection are fitted with the same procedure described above. Then, the four curves generated are linearly interpolated to create a surface of fault coverage. The Matlab function *fit* is used with the parameter *linearinterp* to create the surface.

It is possible to notice that the Eq. 6.2 is very suitable for fitting the fault coverage of selective hardening methodologies. And as for the fault coverage, Eq. 6.2 showed very suitable to fit the execution time too. Although the execution time can be found in a short time, we decided to use the same equation to fit and extrapolate it. The advantage of extrapolating also the execution time is observed in the reduction of the time to generate all the possible selective hardened versions, added to the time to compile them all, and, finally, the time to execute the many versions and get all the execution times.

Finally, with the surfaces of fault coverage and execution time, it is possible to calculate the MWTF, which is our metric of reliability. All the results are presented in the next subsection and validated in the following subsection.

6.3.2 Fault injection results in the miniMIPS processor

Fig. 6.79 presents the estimated fault coverage for the bubble sort (BS). The horizontal axes indicate the level of protection of S-VAR and S-SETA, where 0 represents no protection, and 1 represents 100% of protection. The red dots are the results obtained from fault injections. They are used as inputs for the method. The estimation shows some differences for the simulated values, mainly for low level of protection. However, for most level of protection, the estimation is quite approximate.

The same method was used to extrapolate the execution time. Fig. 6.80 presents the surface with the estimated execution times for the bubble sort (BS). The red dots indicate the real values used as input. The gathering of the execution time is not as time consuming as the fault coverage. The application just needs to be hardened by the possibilities of selective hardening and, then, the execution time of all created versions have to be extracted. Anyhow, the estimation is very approximate to real values since the increasing of the execution time with the increasing of the level of protection is more predictable. Thus, it can also be used to speed up this process, and there will be no need to create all the possibilities of selective hardening and get all the execution times.

Fig. 6.81 presents the MWTFs for the bubble sort (BS), which were calculated using the estimated values. Although there are some differences in the values of the highest MWTFs, the estimation shows correctly the area where such MWTFs are. Using the estimation, it is possible to find quickly points with high MWTF. The same results for each benchmark are presented from Fig. 6.82 to Fig. 6.105. We can see in all cases that although the method does not provide a precise magnitude in the estimations, it points the regions that present the highest MWTF. In general, the highest estimated MWTF is when 100% of registers are hardened and around 80% of basic blocks are hardened. These results match the ones from the previous sections of this chapter.

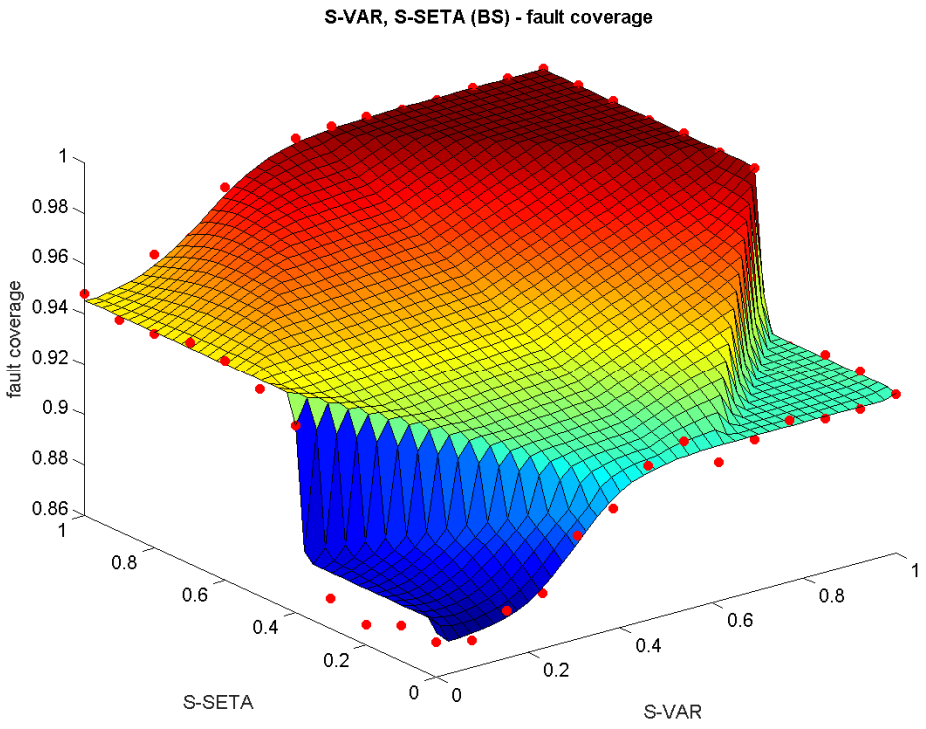


Fig. 6.79: Estimated fault coverages for BS hardened by S-VAR, S-SETA.

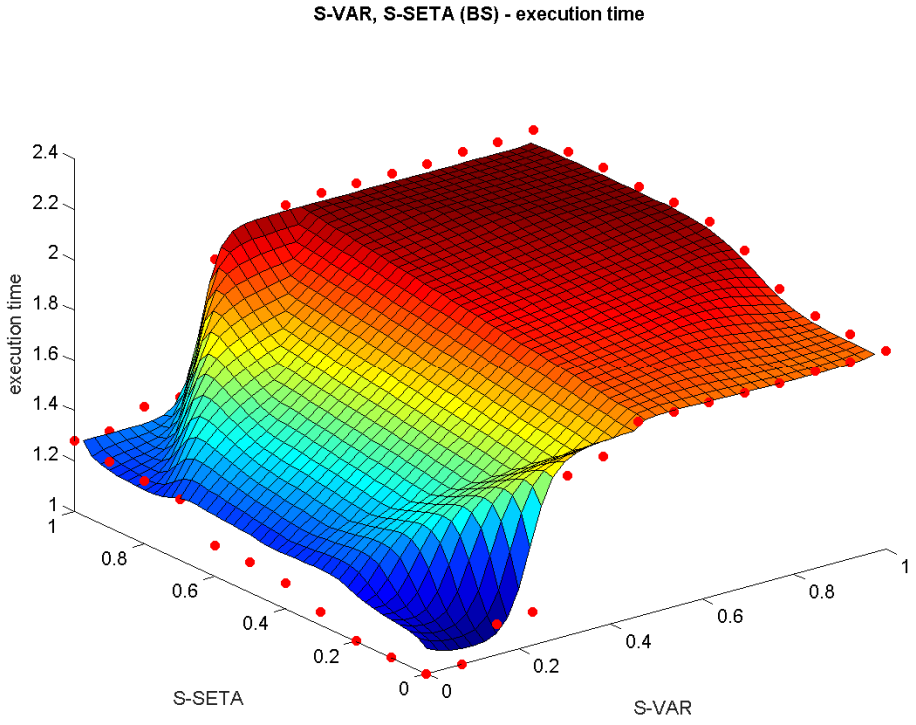


Fig. 6.80: Estimated execution times for BS hardened by S-VAR, S-SETA.

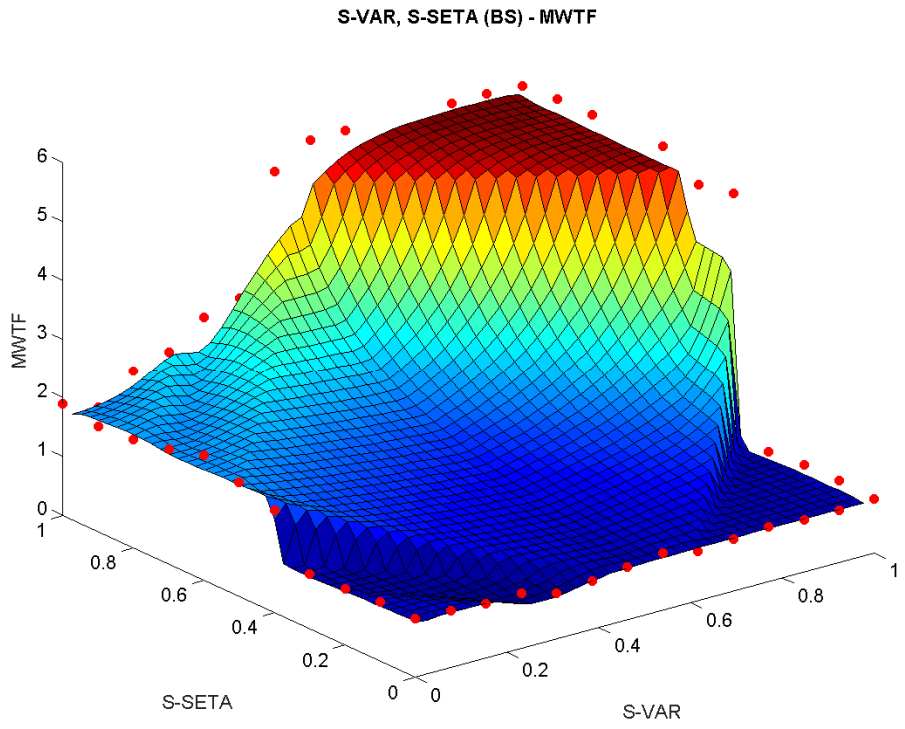


Fig. 6.81: MWTF for BS hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

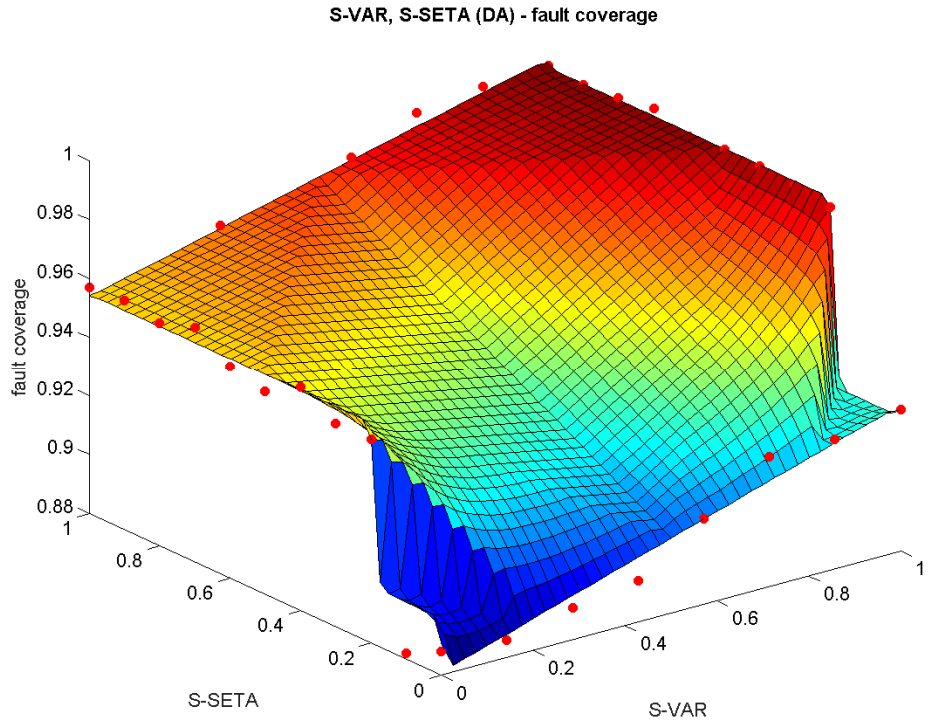


Fig. 6.82: Estimated fault coverages for DA hardened by S-VAR, S-SETA.

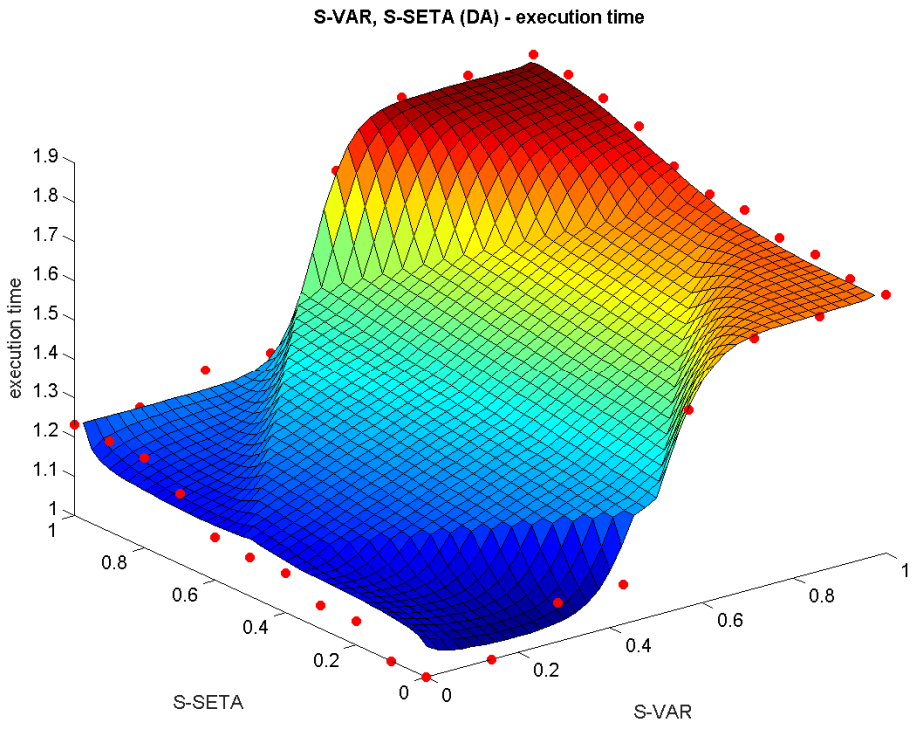


Fig. 6.83: Estimated execution times for DA hardened by S-VAR, S-SETA.

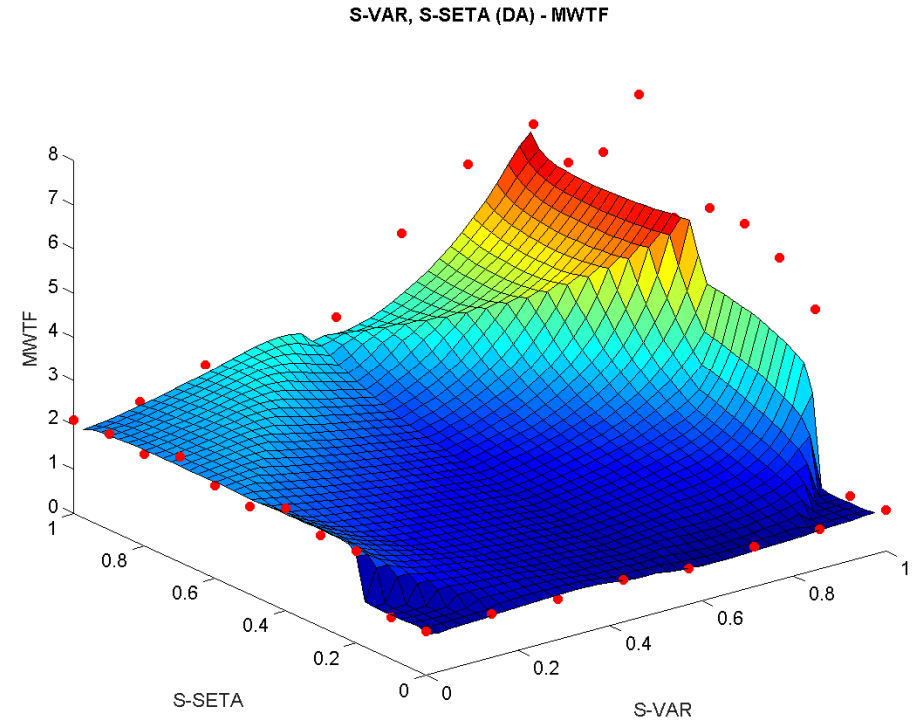


Fig. 6.84: MWTF for DA hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

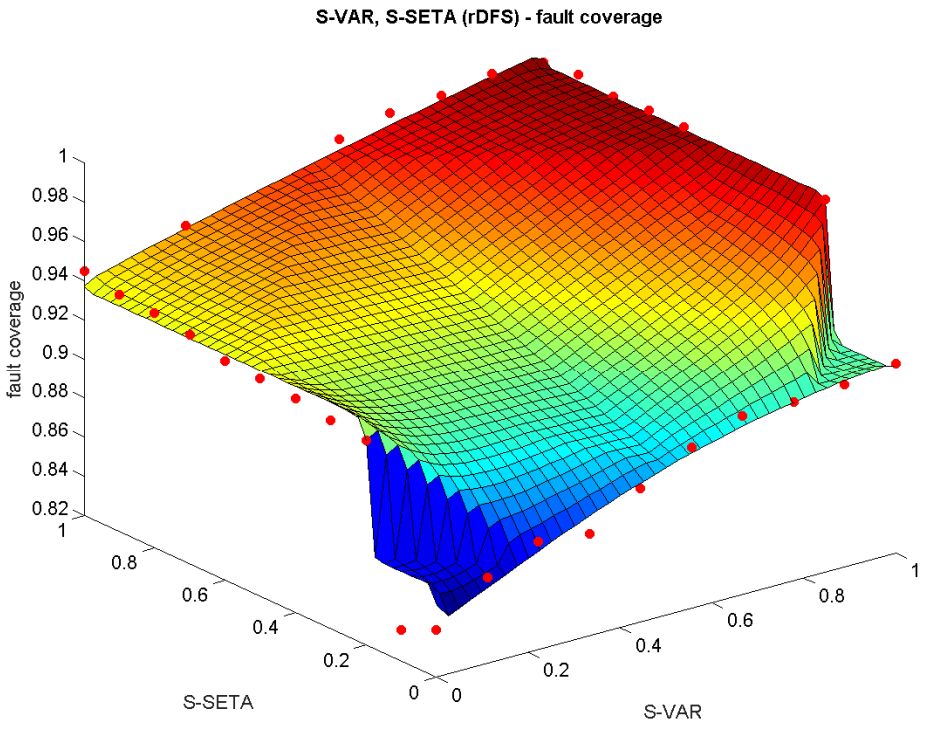


Fig. 6.85: Estimated fault coverages for rDFS hardened by S-VAR, S-SETA.

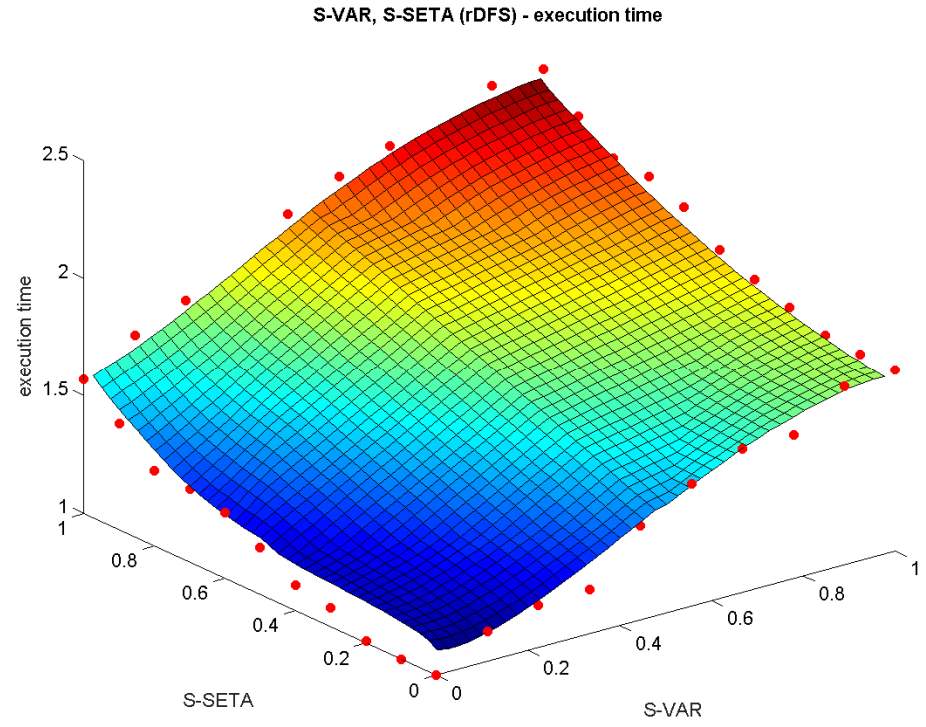


Fig. 6.86: Estimated execution times for rDFS hardened by S-VAR, S-SETA.

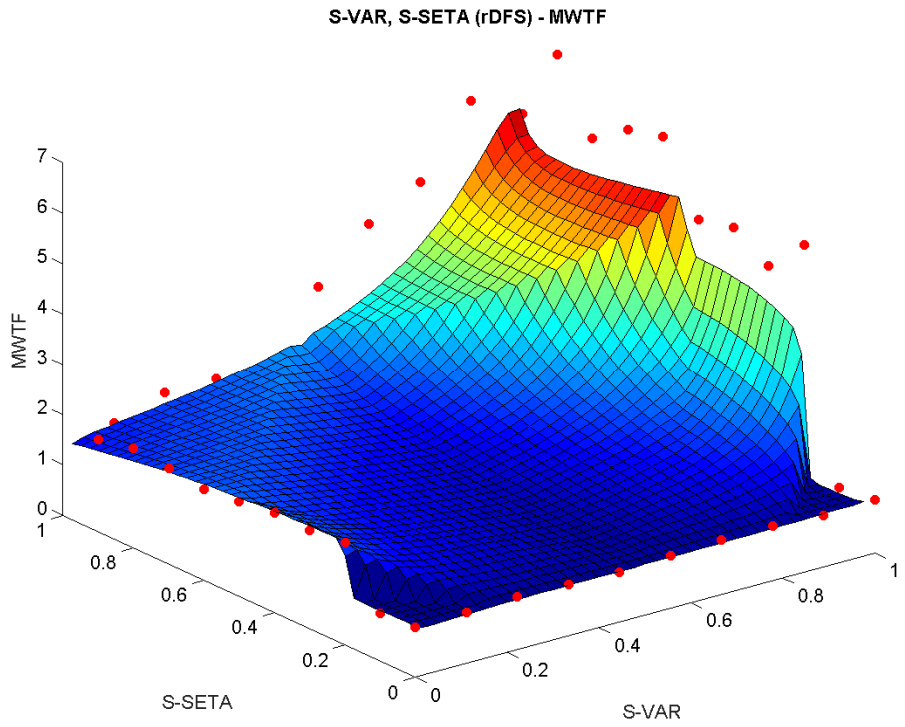


Fig. 6.87: MWTF for rDFS hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

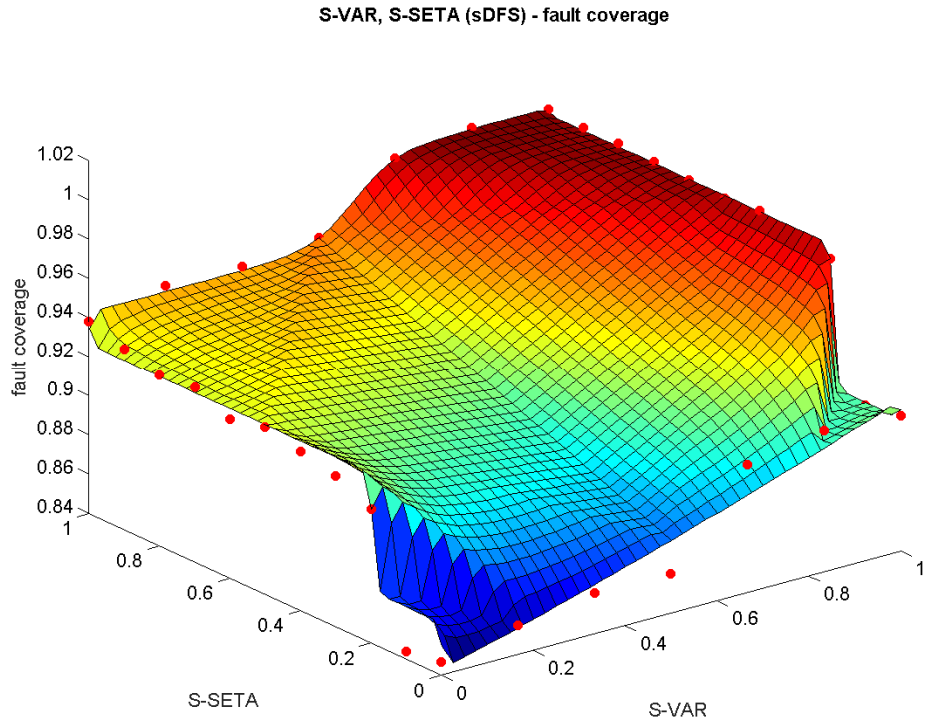


Fig. 6.88: Estimated fault coverages for sDFS hardened by S-VAR, S-SETA.

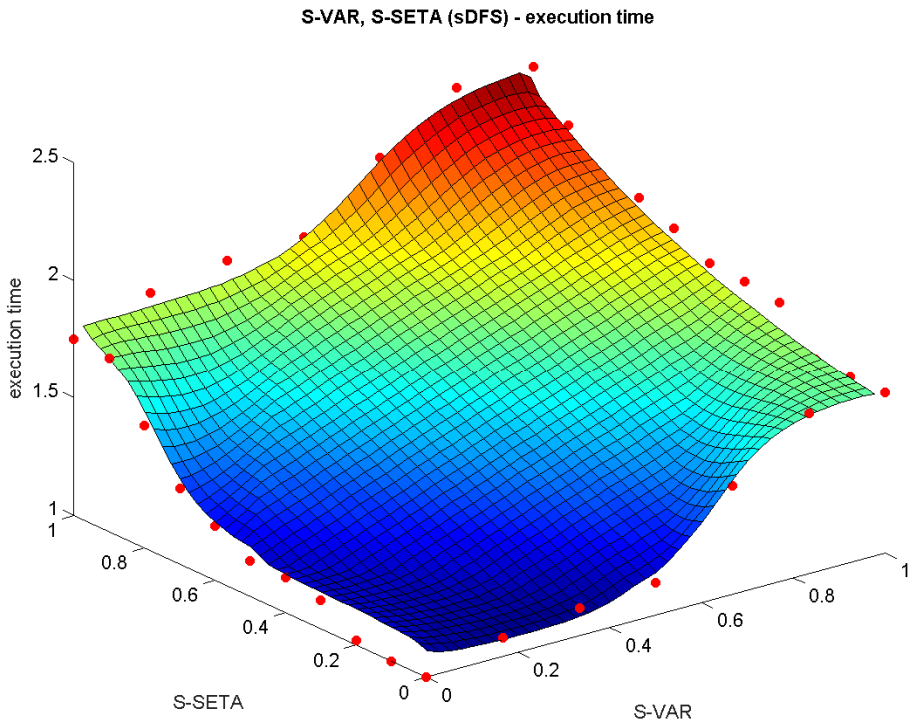


Fig. 6.89: Estimated execution times for sDFS hardened by S-VAR, S-SETA.

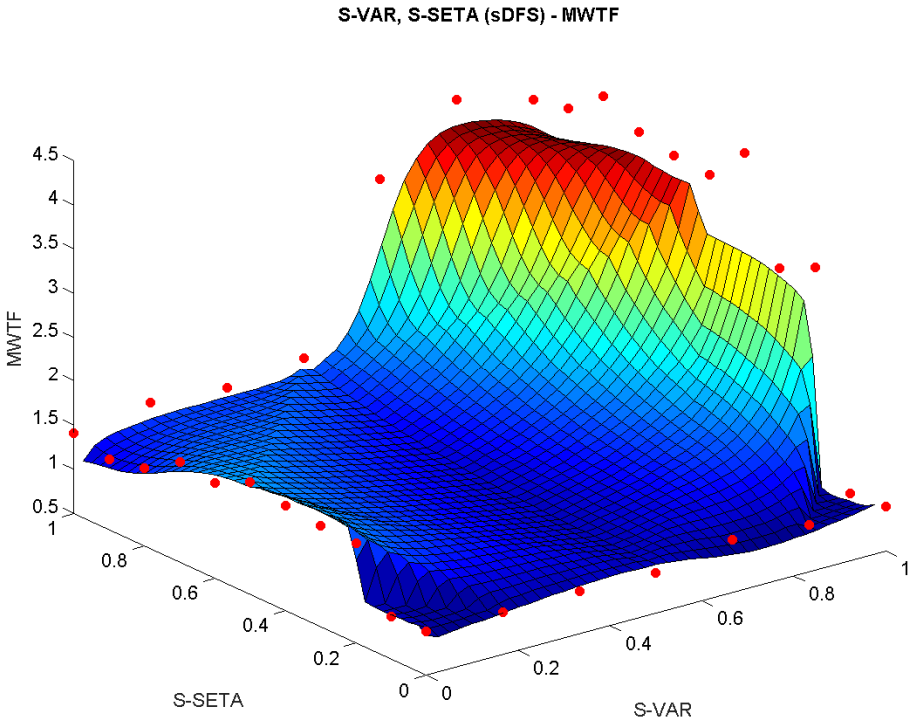


Fig. 6.90: MWTF for sDFS hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

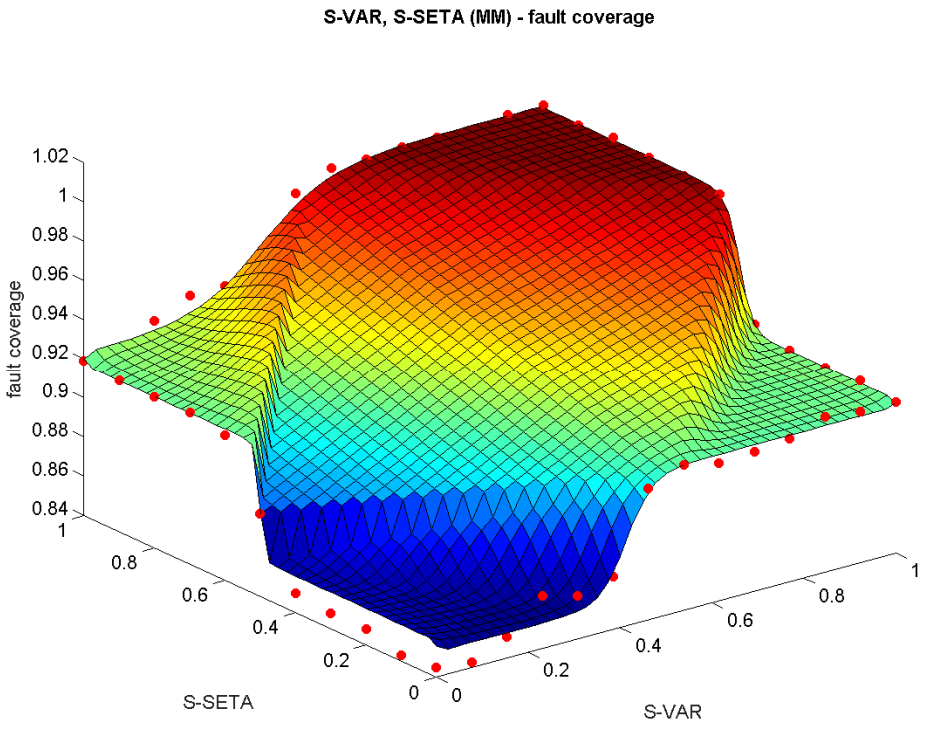


Fig. 6.91: Estimated fault coverages for MM hardened by S-VAR, S-SETA.

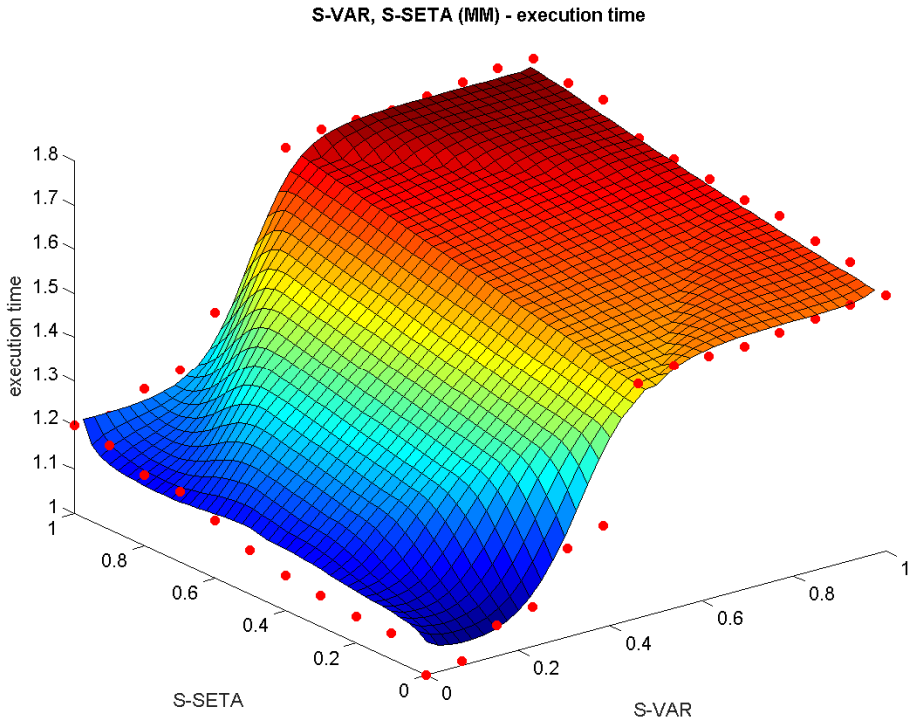


Fig. 6.92: Estimated execution times for MM hardened by S-VAR, S-SETA.

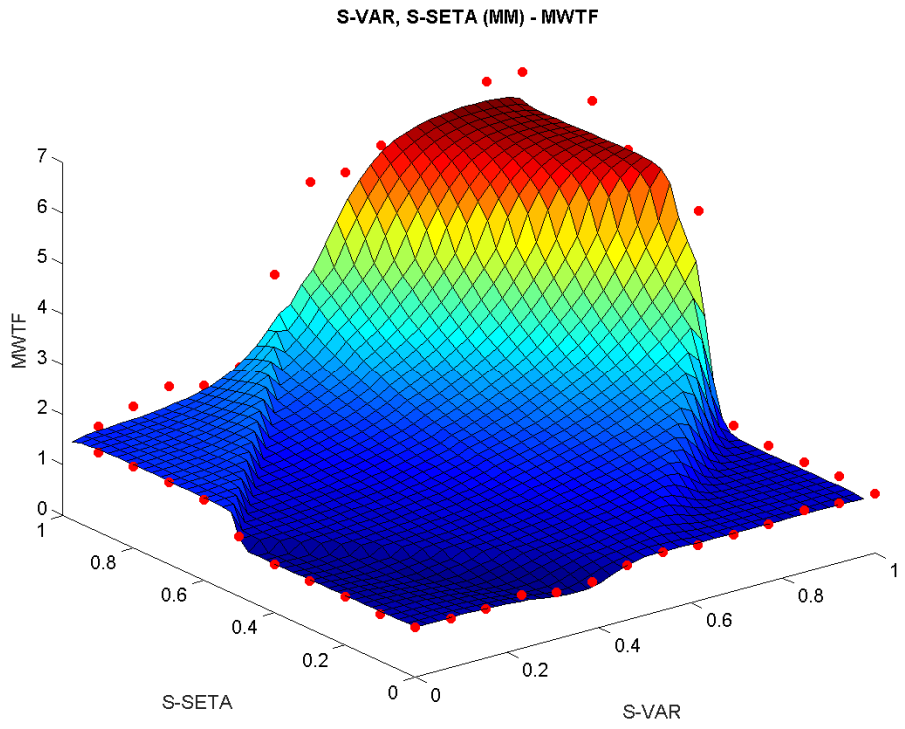


Fig. 6.93: MWTF for MM hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

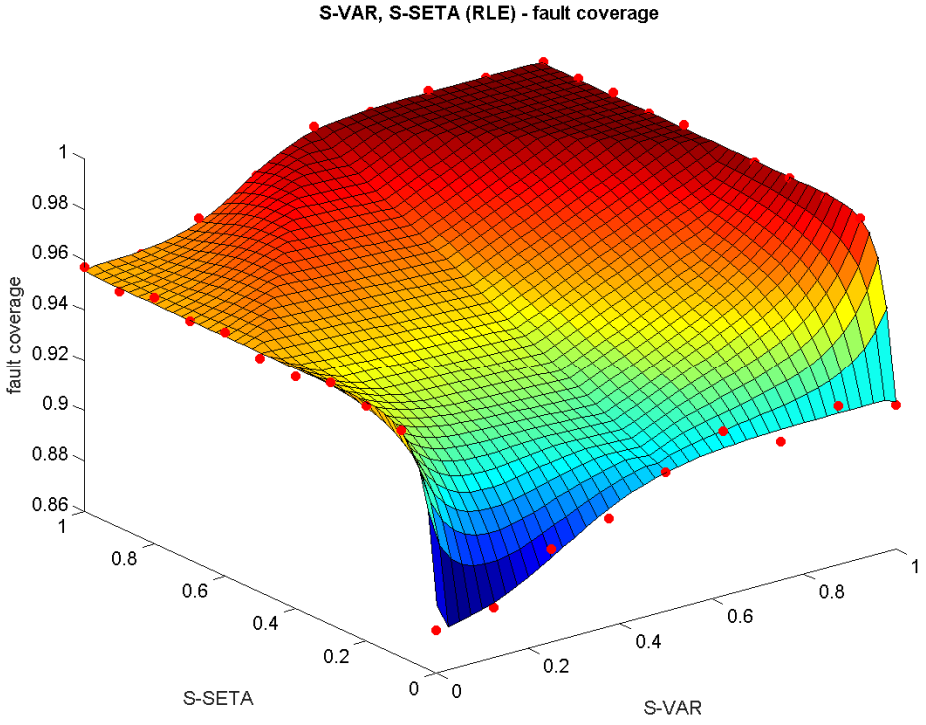


Fig. 6.94: Estimated fault coverages for RLE hardened by S-VAR, S-SETA.

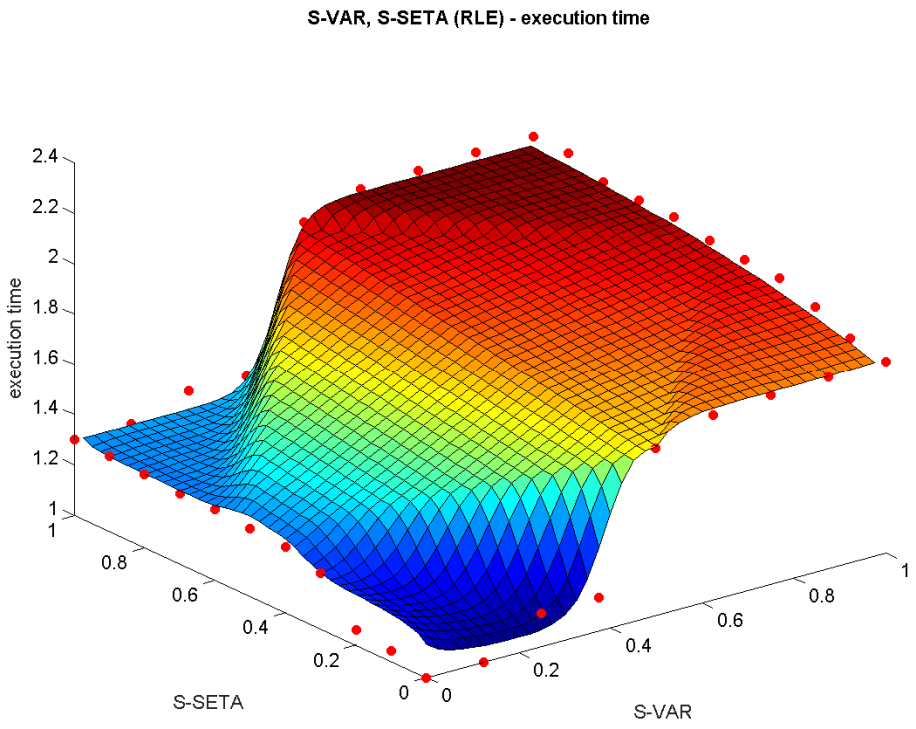


Fig. 6.95: Estimated execution times for RLE hardened by S-VAR, S-SETA.

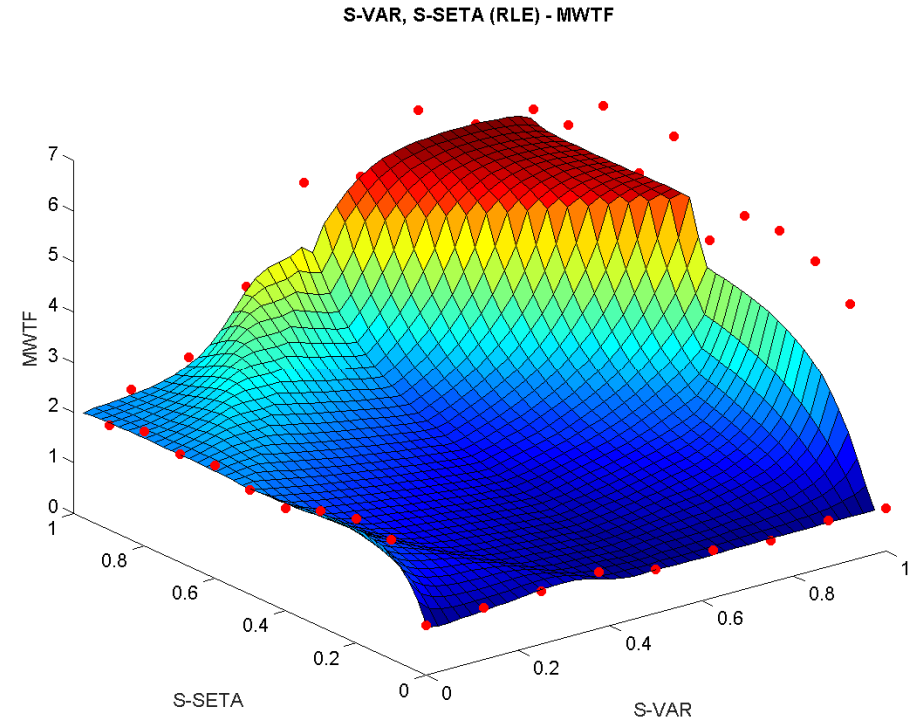


Fig. 6.96: MWTF for RLE hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

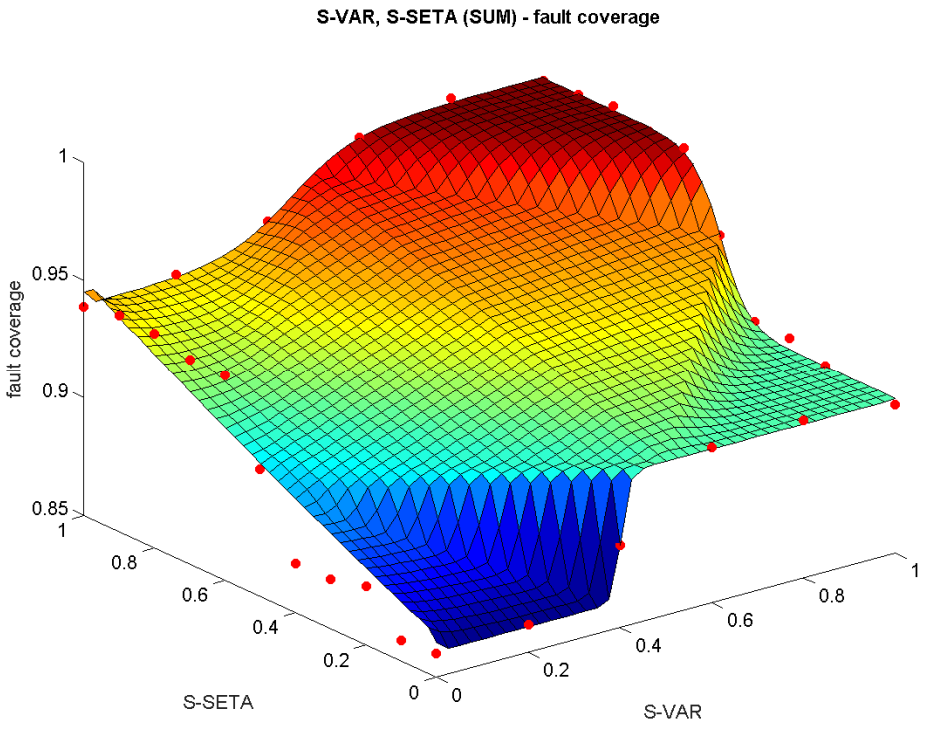


Fig. 6.97: Estimated fault coverages for SUM hardened by S-VAR, S-SETA.

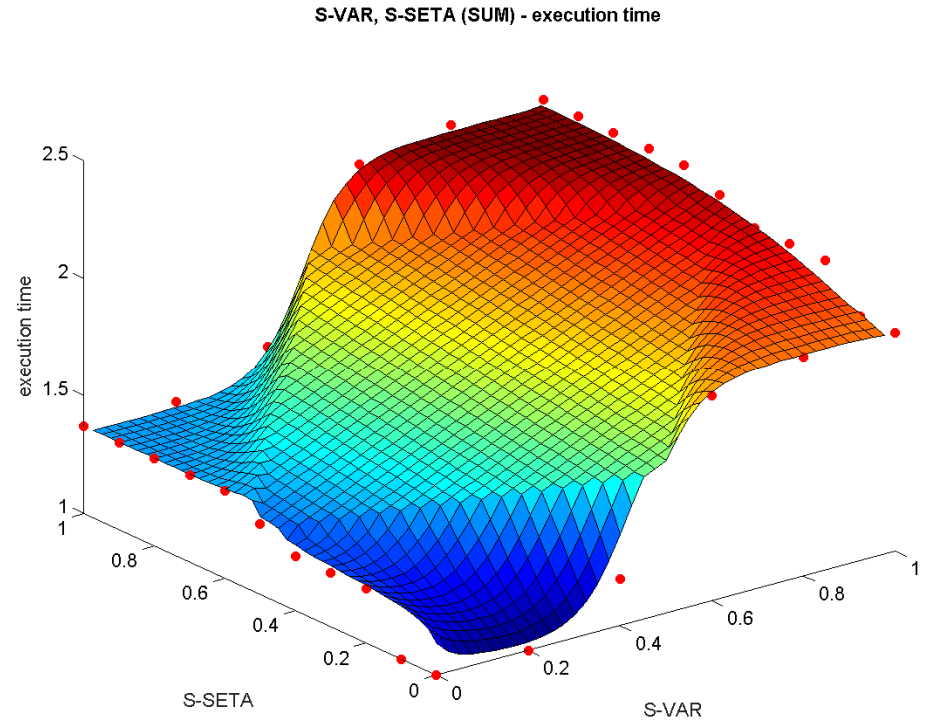


Fig. 6.98: Estimated execution times for SUM hardened by S-VAR, S-SETA.

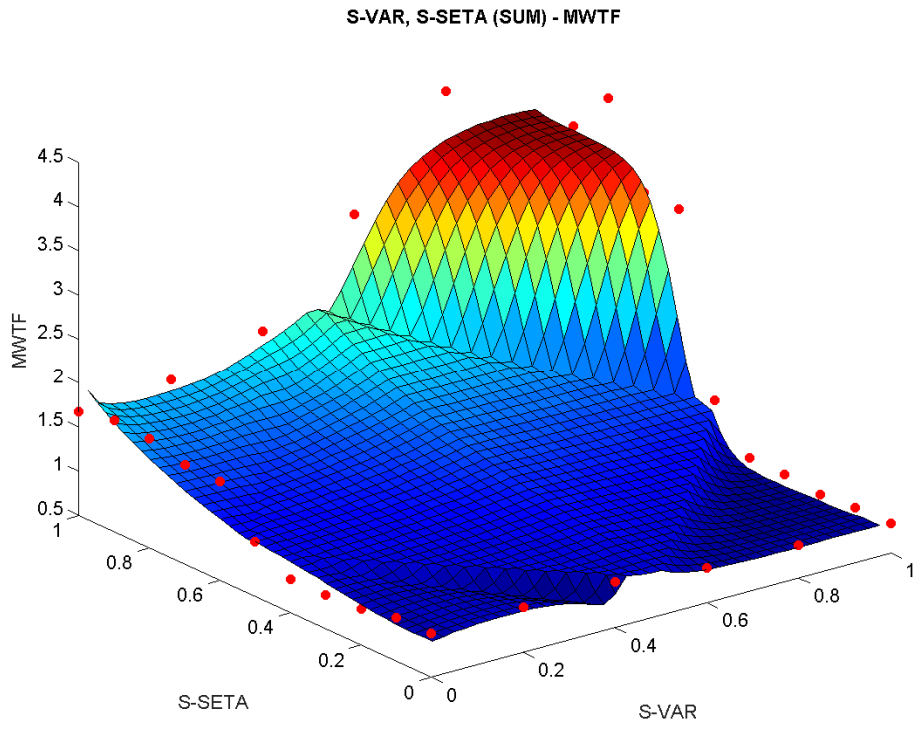


Fig. 6.99: MWTF for SUM hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

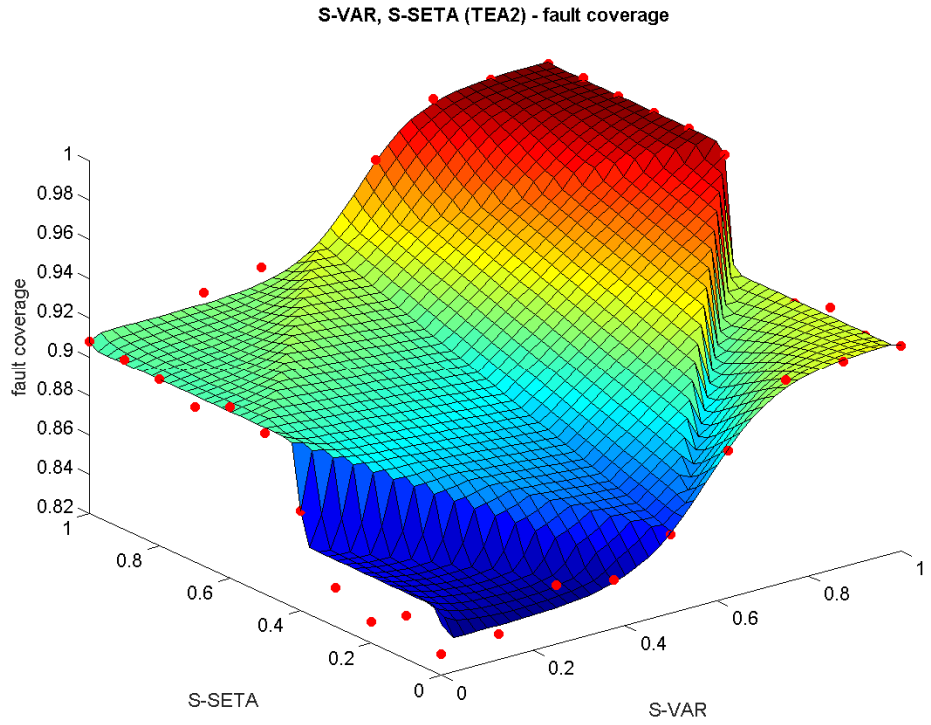


Fig. 6.100: Estimated fault coverages for TEA2 hardened by S-VAR, S-SETA.

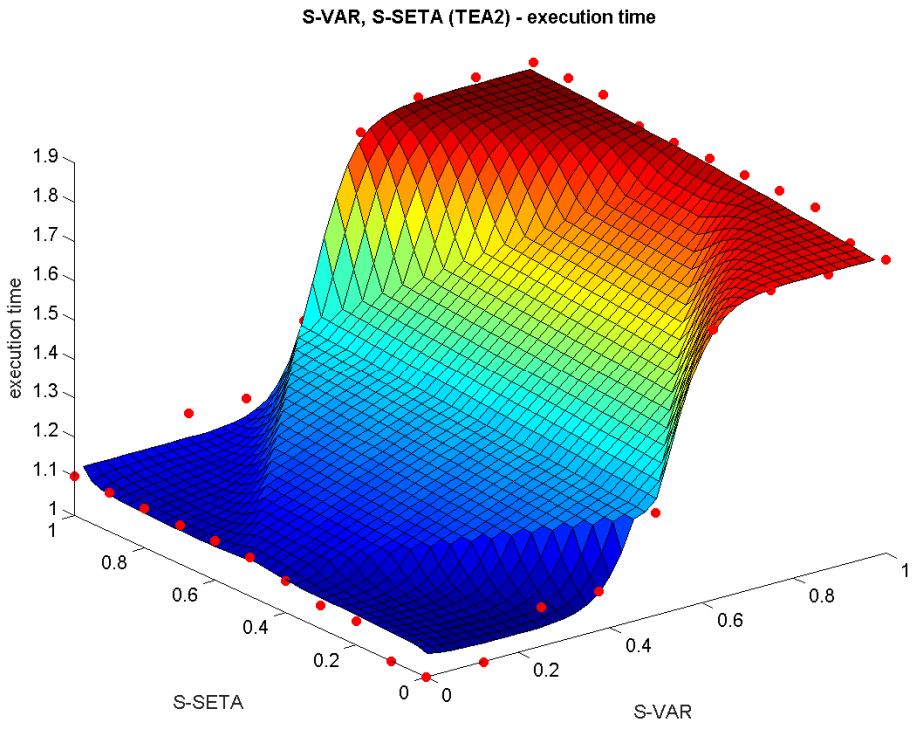


Fig. 6.101: Estimated execution times for TEA2 hardened by S-VAR, S-SETA.

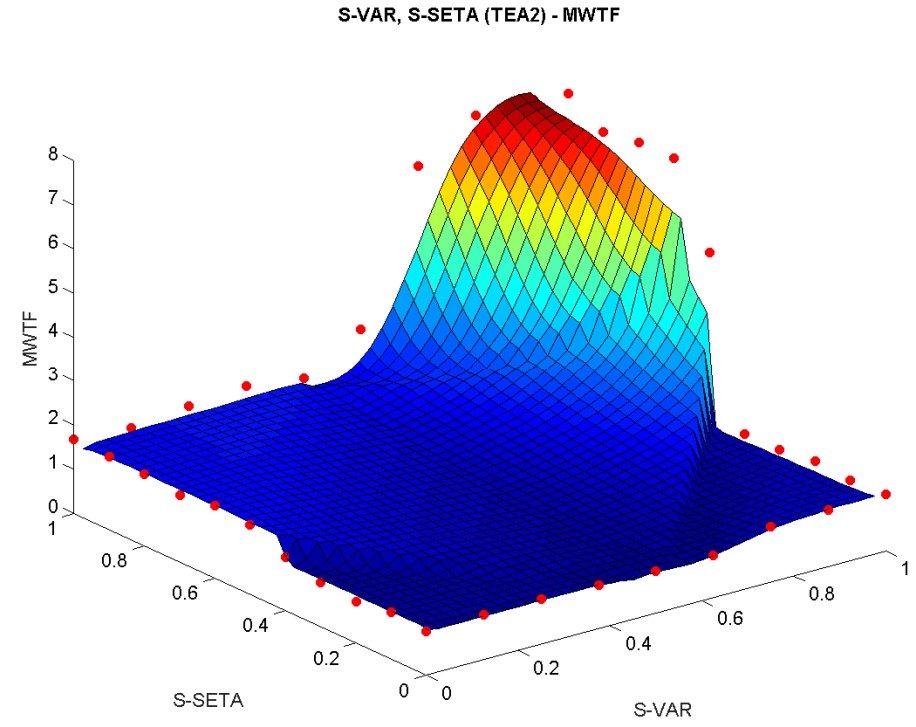


Fig. 6.102: MWTF for TEA2 hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

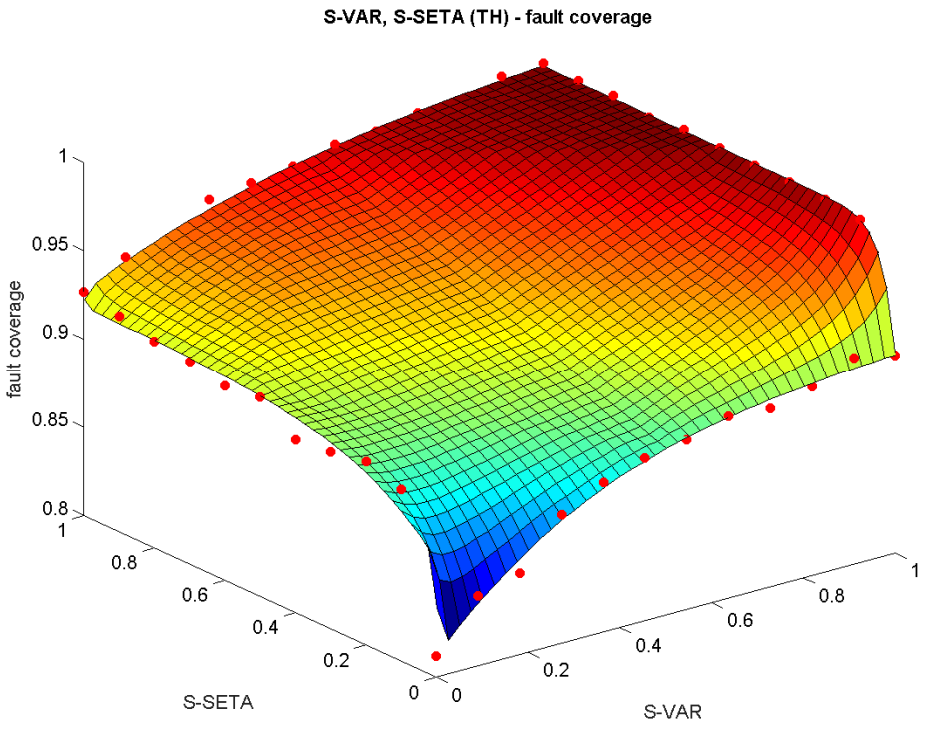


Fig. 6.103: Estimated fault coverages for TH hardened by S-VAR, S-SETA.

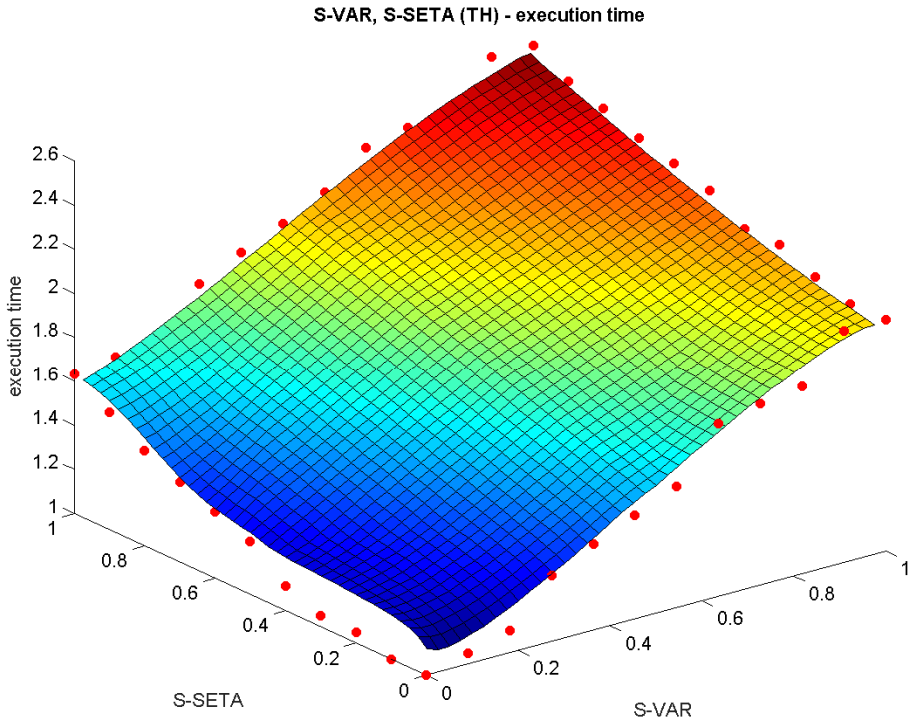


Fig. 6.104: Estimated execution times for TH hardened by S-VAR, S-SETA.

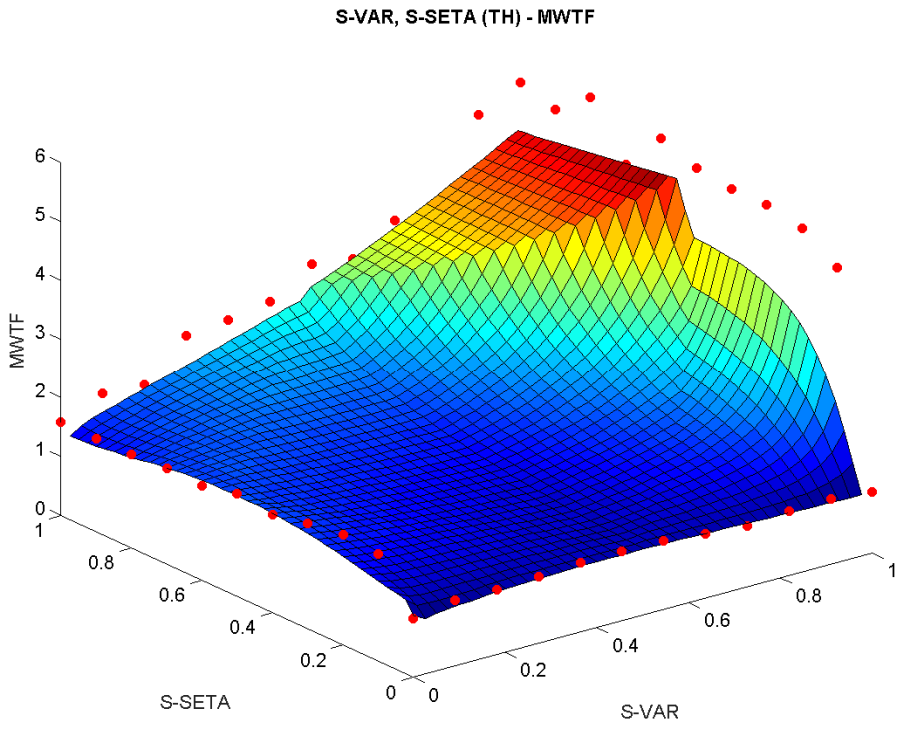


Fig. 6.105: MWTF for TH hardened by S-VAR, S-SETA based on estimated fault coverage and execution time.

In conclusion, we can say that the highest MWTF are reached near 100% of registers hardened by S-VAR and 80% of the basic blocks hardened by S-SETA. Nevertheless, due to the imprecision in the magnitude of the estimated values, it is not possible to find the highest fault coverage for a given maximum execution time, or the lowest execution time for a given minimum fault coverage. A more precise method is necessary in this regard.

6.3.3 Validation

Three case-study applications were used to validate the model to extrapolate the results due to the different surfaces they produced (rDFS, MM, TH). For the rDFS, three points tested. They were not included in the model inputs. One can see that the results presented from Figs. 6.106 to 6.109 match very well with the results predicted by the model. The test dots are the green dots, and the red dots are the model inputs. There is one extra charting presenting the execution time from another perspective to show the test points that are under the surface. The mean and maximum deviations in the fault coverage from the estimated results to the simulated points are of 0.4% and 0.6%, respectively. And the mean and maximum deviations in the execution time are of 3.0% and 4.7%, respectively. For the MM, four points were tested, as showed from Figs. 6.10 to 6.112. Once again, the results match very well the model. The mean deviation in the fault coverage is of 1.3%, and in the execution time is of 2.2%. The maximum deviation is of 3.6% in the fault coverage and 5.5% in the execution time at S-VAR(0.46), S-SETA(0.5). For the TH, presented from Figs. 6.113 to 6.115, two points were tested, S-VAR(0.91), S-SETA(0.7) and S-VAR(0.82), S-SETA(0.6). Their deviations were, respectively, of 0.0% and 0.6% in the fault coverage, and of 1.5% and 0.2% in the

execution time. Therefore, we conclude that the model can extrapolate the results with a good precision, providing accurate predictions of fault coverage, execution time, and MWTF, and pointing the areas with higher MWTF.

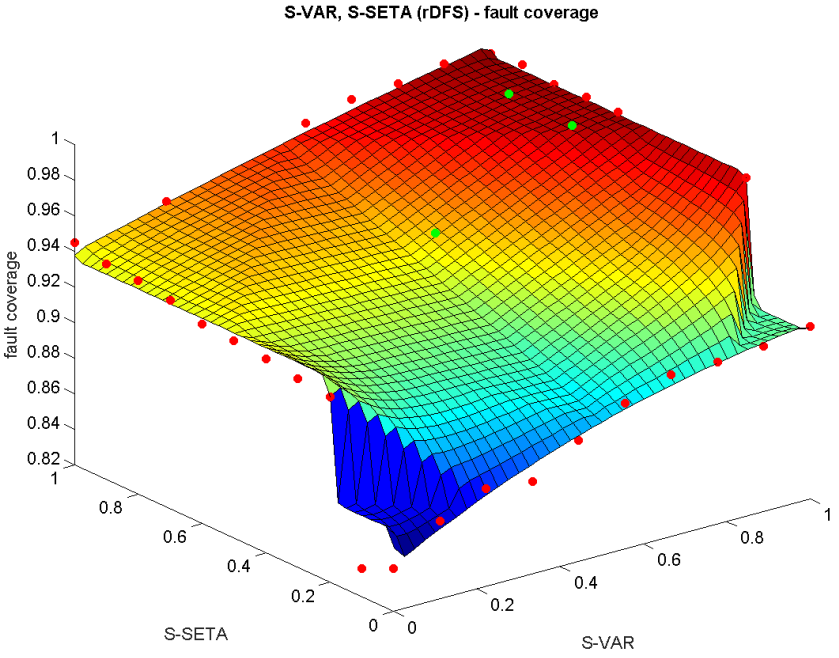


Fig. 6.106: Validation of model for estimating the fault coverages of rDFS hardened by S-VAR, S-SETA. Green dots are the validation points.

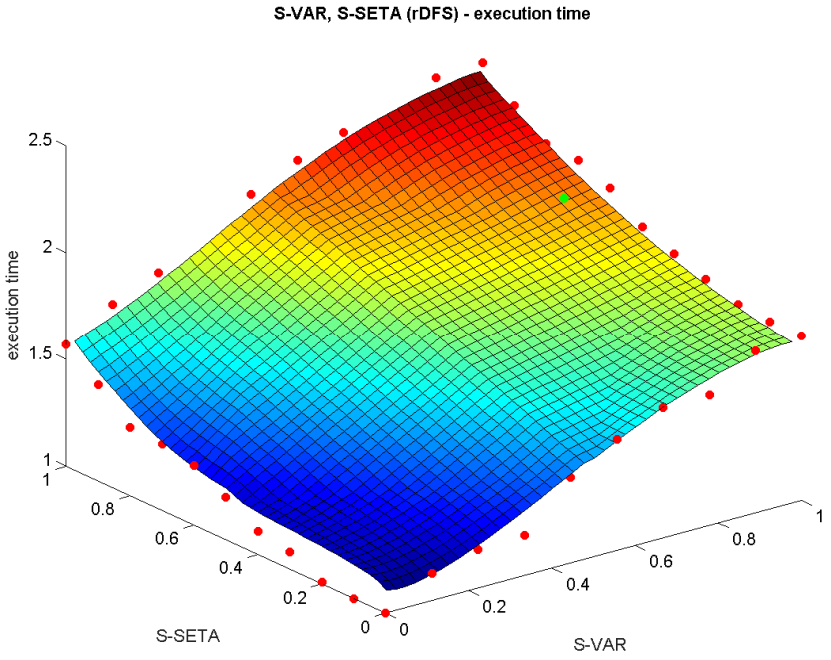


Fig. 6.107: Validation of model for estimating the execution time of rDFS hardened by S-VAR, S-SETA. Green dots are the validation points.

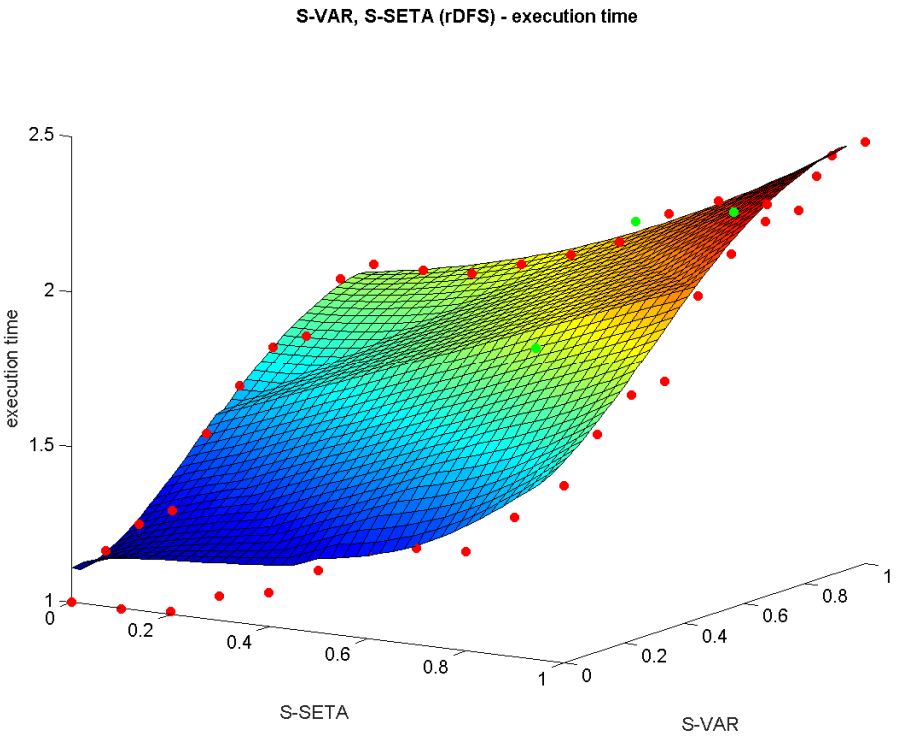


Fig. 6.108: Validation of model for estimating the execution time of rDFS hardened by S-VAR, S-SETA. Green dots are the validation points. View from another perspective.

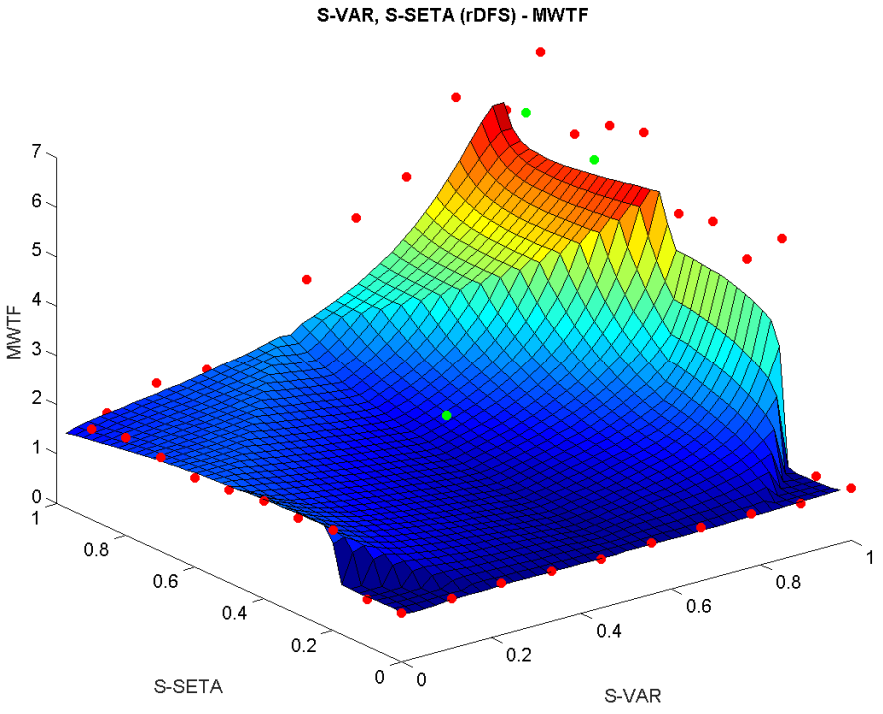


Fig. 6.109: MWTF for rDFS hardened by S-VAR, S-SETA based on estimated fault coverage and execution time. Green dots are the validation points.

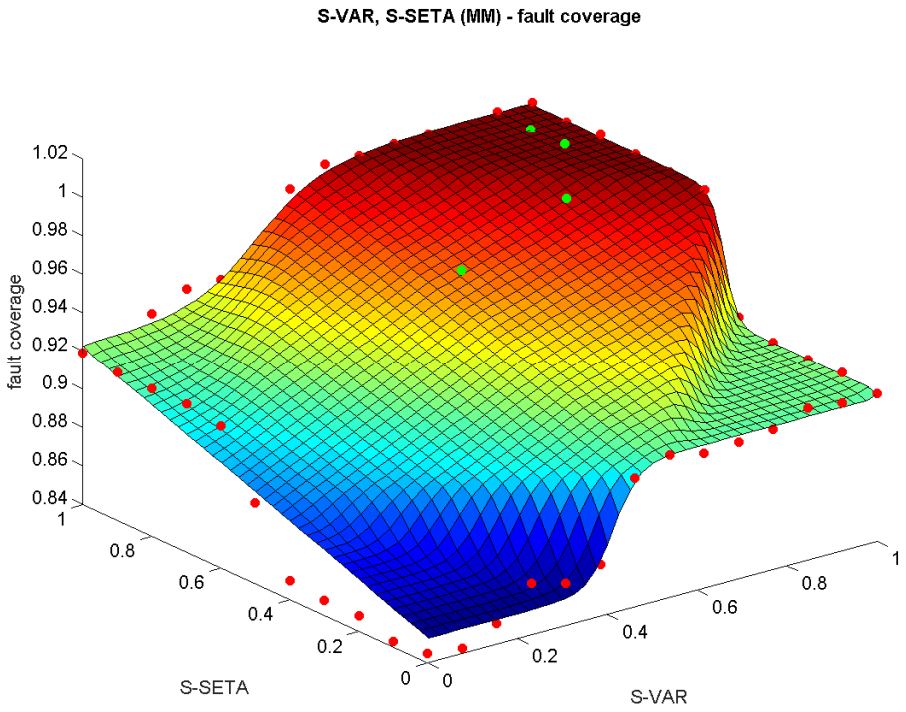


Fig. 6.110: Validation of model for estimating the fault coverages of MM hardened by S-VAR, S-SETA. Green dots are the validation points.

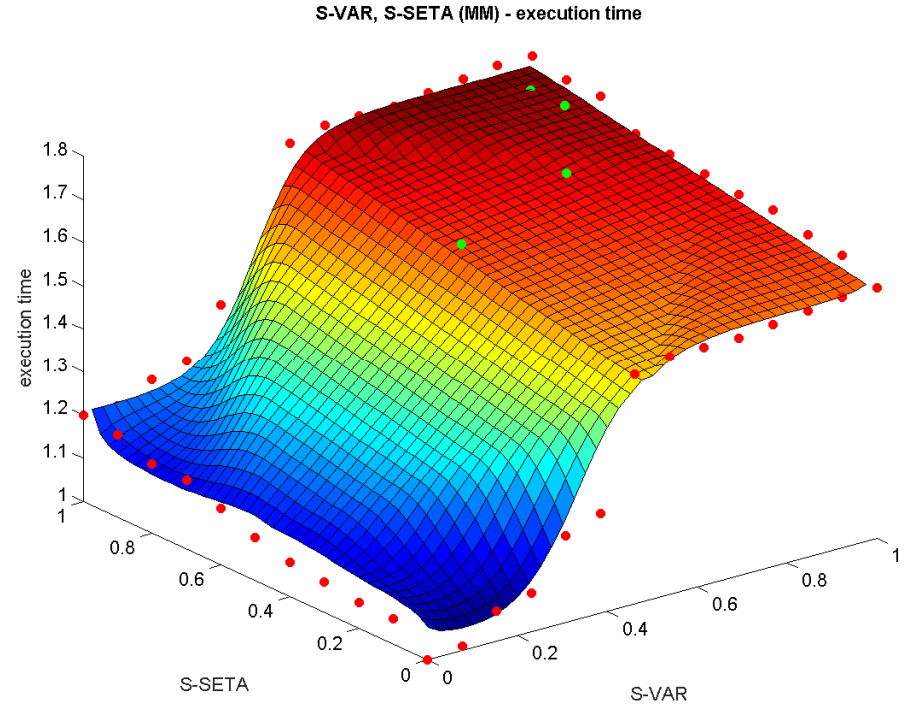


Fig. 6.111: Validation of model for estimating the execution time of MM hardened by S-VAR, S-SETA. Green dots are the validation points.

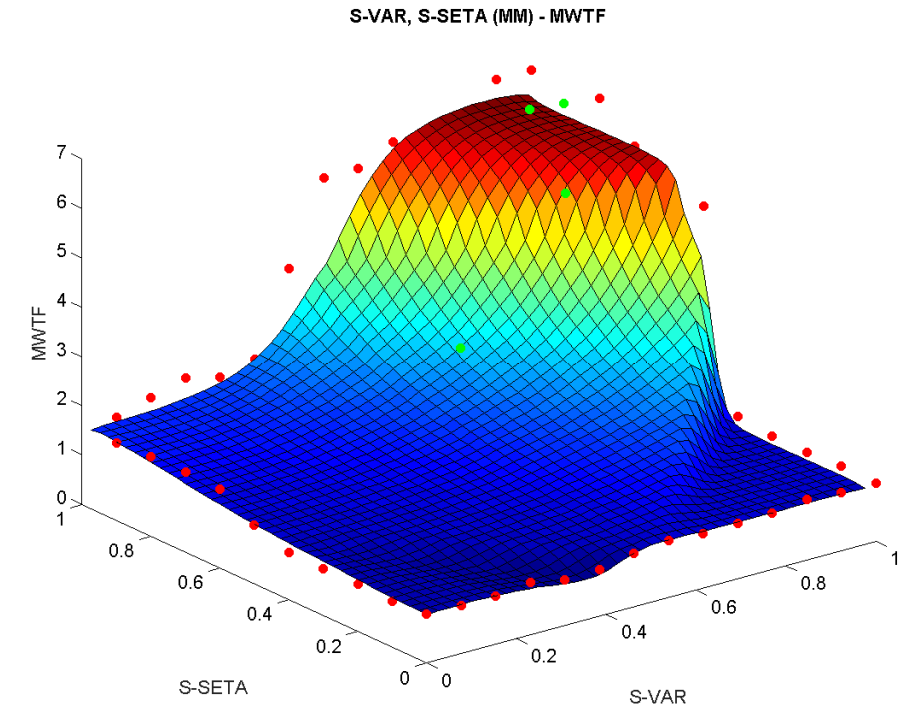


Fig. 6.112: MWTF for MM hardened by S-VAR, S-SETA based on estimated fault coverage and execution time. Green dots are the validation points.

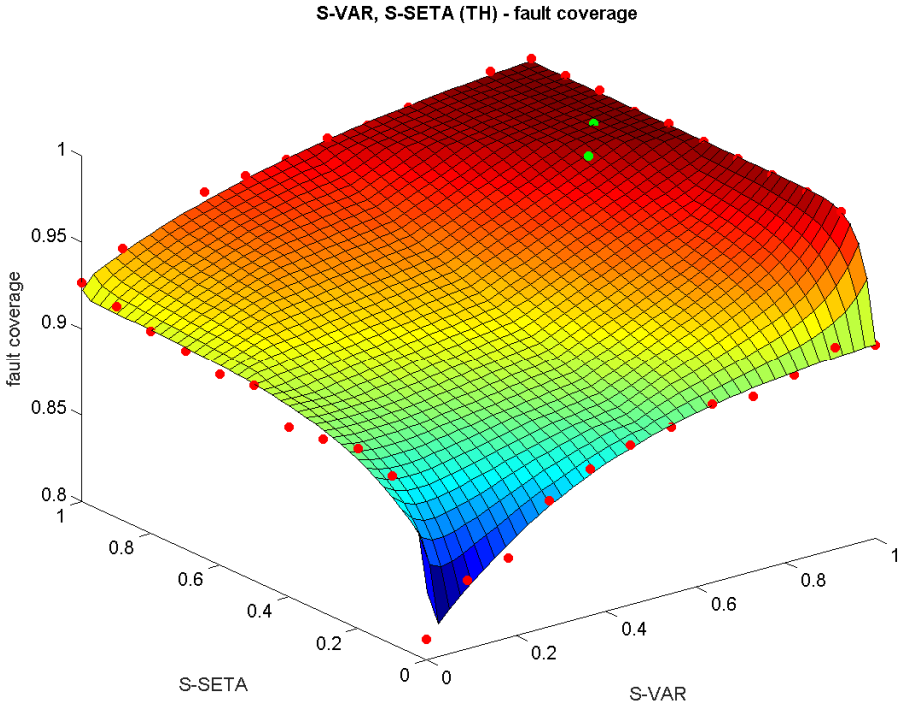


Fig. 6.113: Validation of model for estimating the fault coverages of TH hardened by S-VAR, S-SETA. Green dots are the validation points.

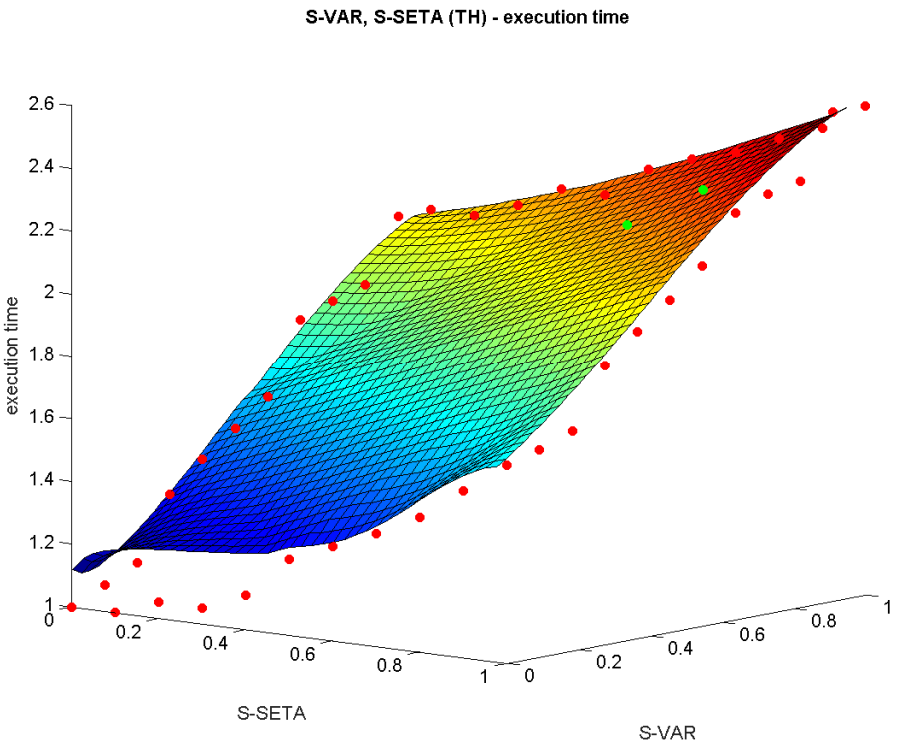


Fig. 6.114: Validation of model for estimating the execution time of TH hardened by S-VAR, S-SETA. Green dots are the validation points. View from another perspective.

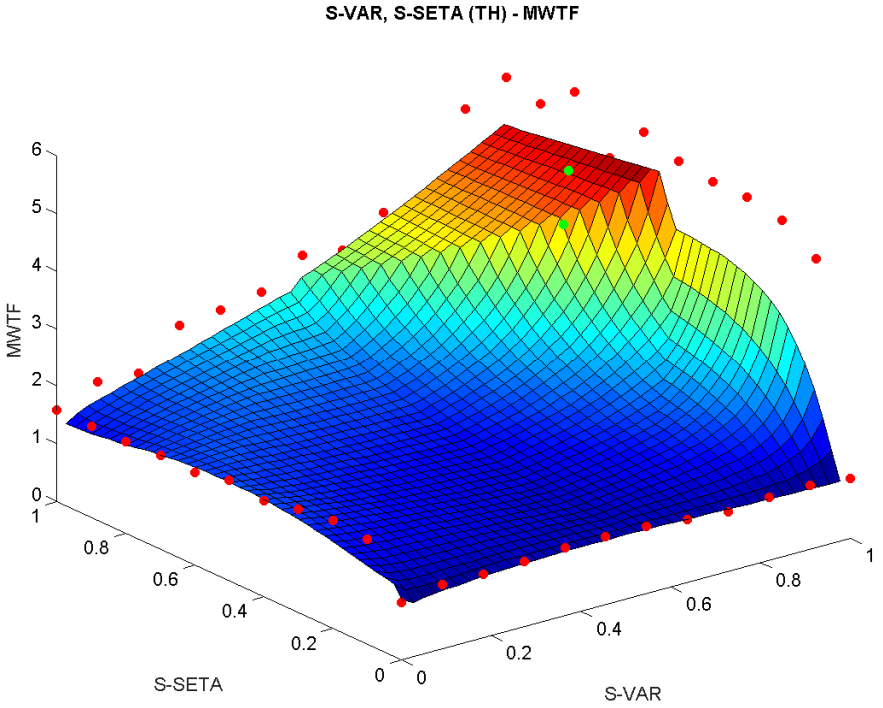


Fig. 6.115: MWTF for TH hardened by S-VAR, S-SETA based on estimated fault coverage and execution time. Green dots are the validation points.

6.3.4 Reducing number of points for fitting model

Depending on the fault injection platform, the simulation of around 40 cases per application may be time-consuming. Thus, we decided to test the fitting model using fewer input points to discover if it still produces an accurate surface. For this, three applications that present different set of surfaces were selected. They are the recursive depth-first search (rDFS), the matrix multiplication (MM), and the Tower of Hanoi (TH). For each of these applications, the three following cases with a different number of points were evaluated:

- **Original**: all the simulated points are used as input. The rDFS, MM, and TH have 38, 46, and 42 points, respectively
- **Half**: roughly half of the simulated points are used as input. The rDFS, MM, and TH use 18, 22, and 20 points, respectively
- **Minimum**: each of the four curves needs a minimum of 4 points as input. Thus, the minimum number of points for the fitting model is 12 (4 points are shared by two curves).

Figs. 6.116 to 6.124 show the fault coverage, execution time, and MWTF, for the rDFS with a different number of input points for the fitting method. It is possible to notice that the curves, and consequently the surface, get smoother with fewer input points. Nevertheless, the surfaces of fault coverage, execution time, and MWTF, for the approaches with original, half, and minimum number of points are similar. Figs. 6.125 to 6.133 present the results for MM, and Figs. 6.134 to 6.142 show the results for TH. The fault coverage, execution time, and MWTF for the Tower of Hanoi using a different number of input points are very similar. For the matrix multiplication, one aspect must be pointed out. The fault coverage surface starts to drop nearer to S-VAR(1), S-SETA(1), mainly in the S-SETA axis. The same can be said of the execution time, but in this case, it is more noticeable in the S-VAR axis. This difference result is better seen in the charts of MWTF. With regards to the original approach, the half approach reduces the area of high MWTF in the S-SETA axis, and the minimum approach reduces this area in both S-VAR and S-SETA axes. Anyhow, the results are still accurate for most coordinates.

The test points used to validate the method were also added in this section in order to make able the calculation of the deviations from the values predicted by the method to the values obtained by simulation. Table 6.3 presents the mean and maximum deviations in the fault coverage (FC) and execution time (ET) for the rDFS, MM, and TH, with the three cases with a different number of input points. It is possible to notice that the deviations do not chance much when using fewer input points. It means that for the current precision of the model, the use of the minimum number of points as input is enough to provide accurate estimations of the fault coverage and execution time. In addition, the *half* approach could be used to reinforce the estimations predicted by the *minimum* approach. Another thing that is worth commenting is the higher deviations presented by the matrix multiplication. It happens due to inconsistencies added by the interpolation. That creates invalid behaviors in the surface of fault coverages in the case of the matrix multiplication, which reduces the accuracy of the model. The replacement of the linear interpolation to connect the four fitted curves by a method that avoid these invalid behaviors would solve this issue. This subject is discussed in the future works.

Table 6.3: Mean and maximum deviations in the fault coverage (FC) and execution time (ET) for the rDFS, MM, and TH with different numbers of input points to the method to estimate the results

benchmark	case	# of input points	mean dev. (FC)	max dev. (FC)	mean dev. (ET)	max dev. (ET)
rDFS	original	38	0.4%	0.6%	3.0%	4.7%
	half	18	0.4%	0.4%	3.6%	6.1%
	minimum	12	0.2%	0.5%	2.7%	3.7%
MM	original	46	1.3%	3.6%	2.2%	5.5%
	half	22	1.5%	3.3%	1.6%	4.1%
	minimum	12	1.8%	4.4%	3.5%	8.9%
TH	original	42	0.3%	0.6%	0.9%	1.5%
	half	20	0.3%	0.4%	1.3%	1.7%
	minimum	12	0.3%	0.5%	2.0%	2.4%

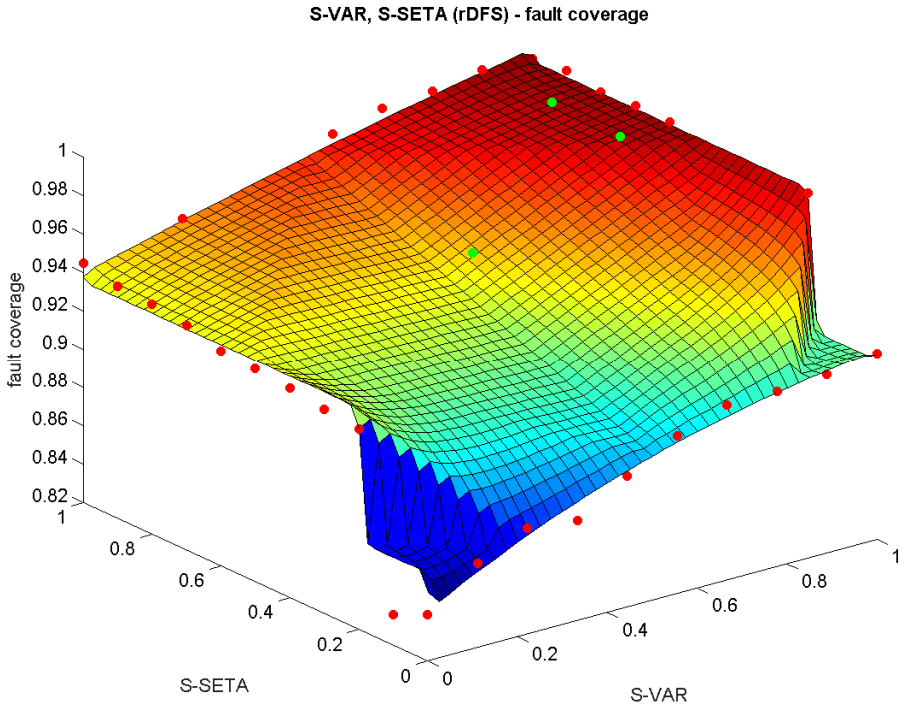


Fig. 6.116: Fault coverage for rDFS (38 points). The red dots are the input points and the green dots are the validation points.

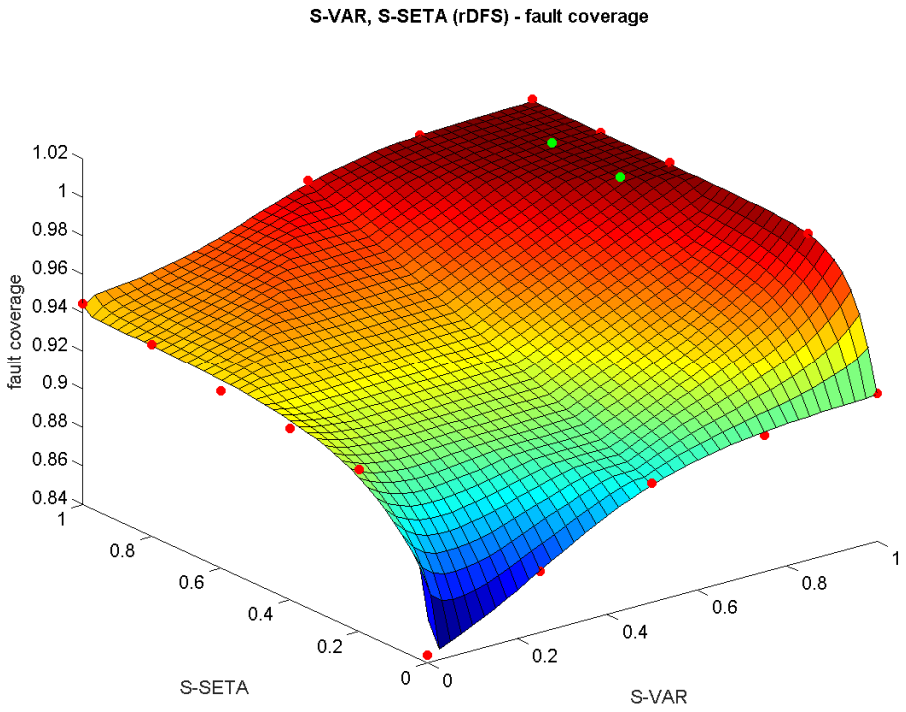


Fig. 6.117: Fault coverage for rDFS (18 points). The red dots are the input points and the green dots are the validation points.

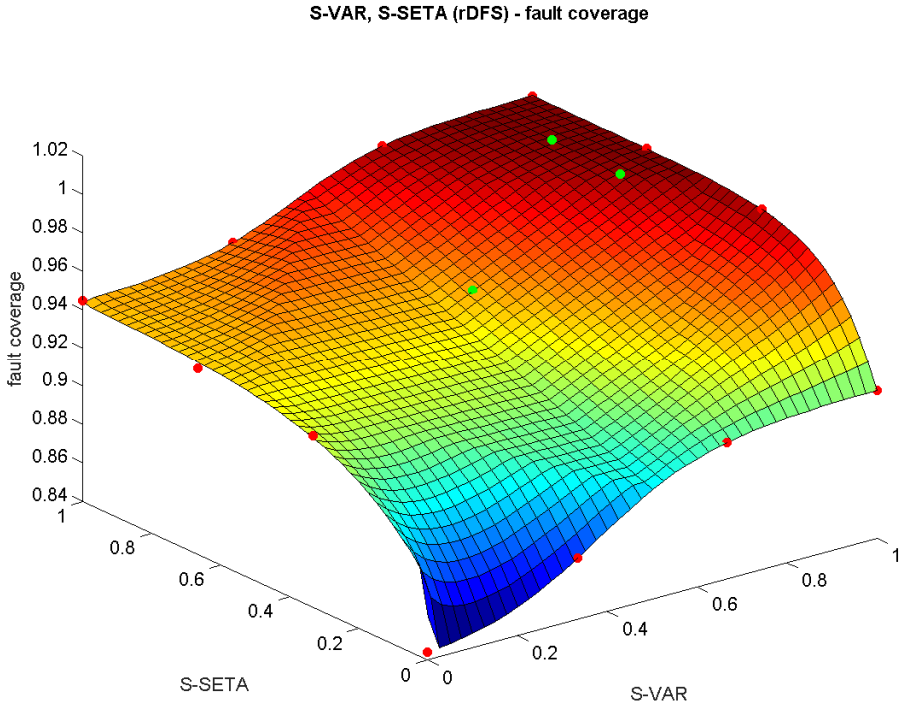


Fig. 6.118: Fault coverage for rDFS (12 points). The red dots are the input points and the green dots are the validation points.

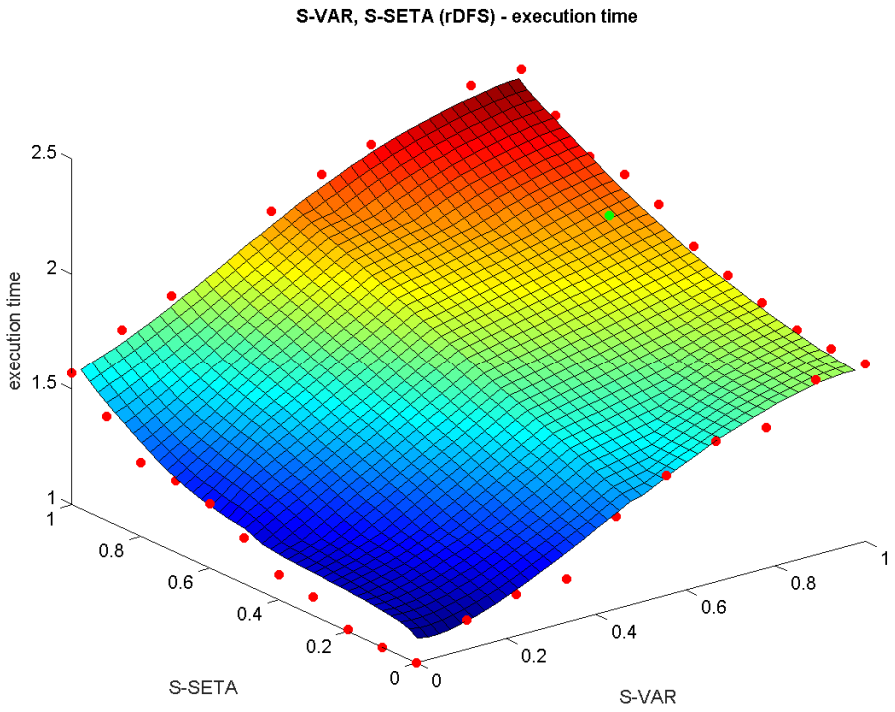


Fig. 6.119: Execution time for rDFS (38 points). The red dots are the input points and the green dots are the validation points.

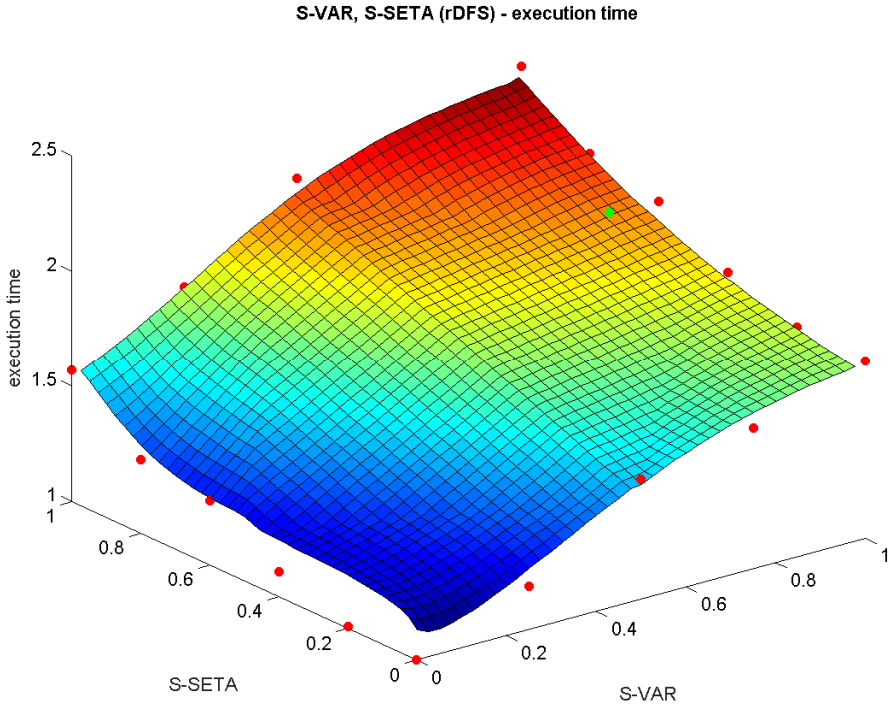


Fig. 6.120: Execution time for rDFS (18 points). The red dots are the input points and the green dots are the validation points.

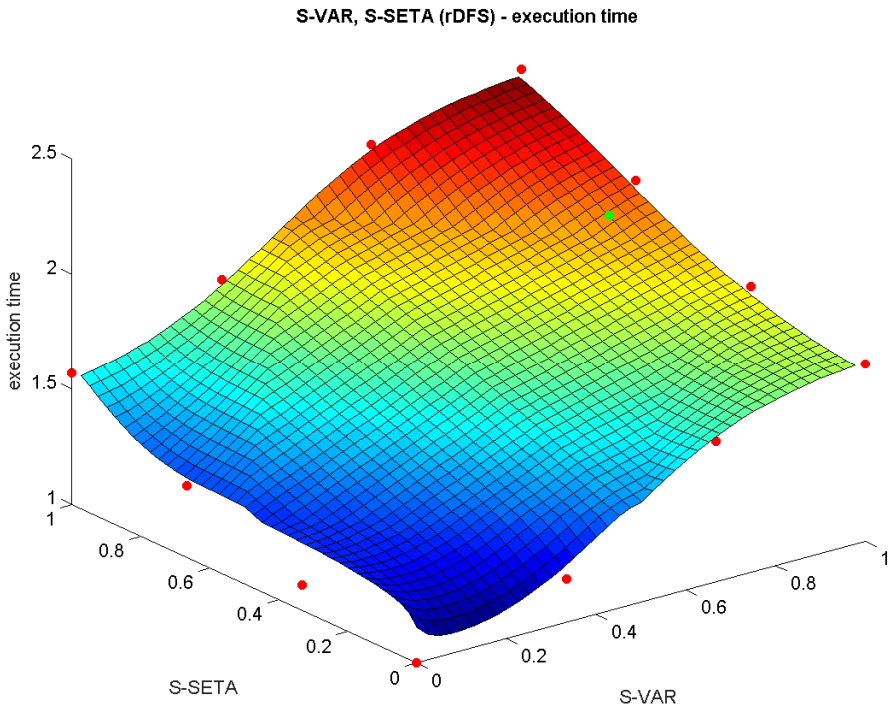


Fig. 6.121: Execution time for rDFS (12 points). The red dots are the input points and the green dots are the validation points.

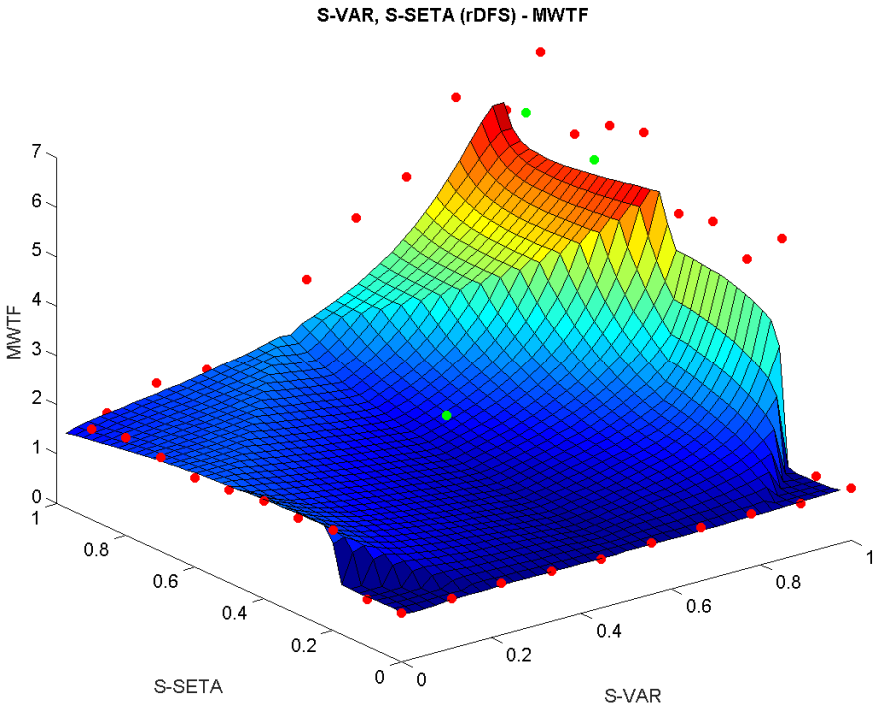


Fig. 6.122: MWTF for rDFS (38 points). The red dots are the input points and the green dots are the validation points.

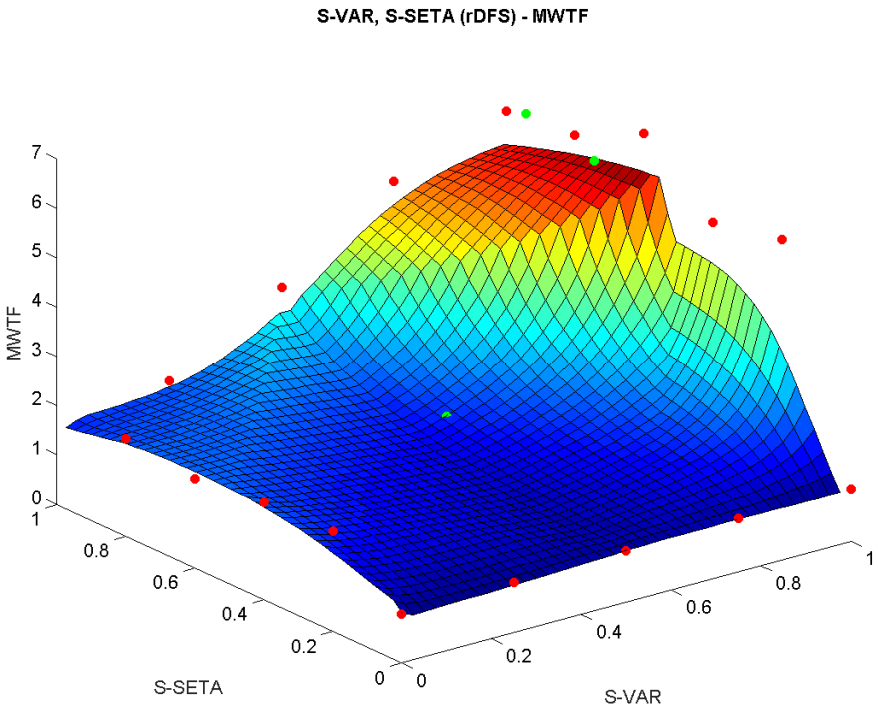


Fig. 6.123: MWTF for rDFS (18 points). The red dots are the input points and the green dots are the validation points.

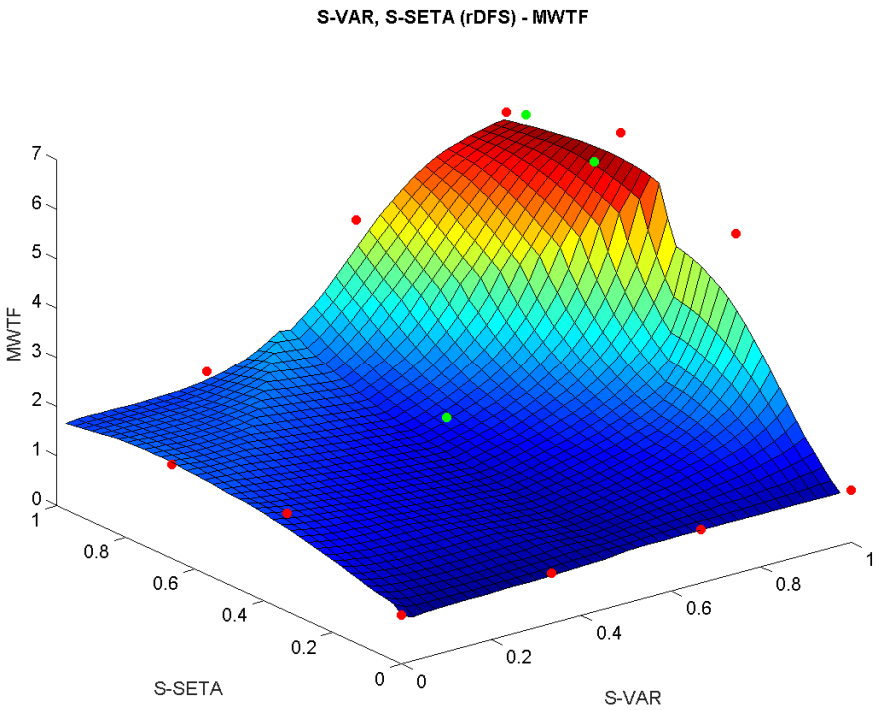


Fig. 6.124: MWTF for rDFS (12 points). The red dots are the input points and the green dots are the validation points.

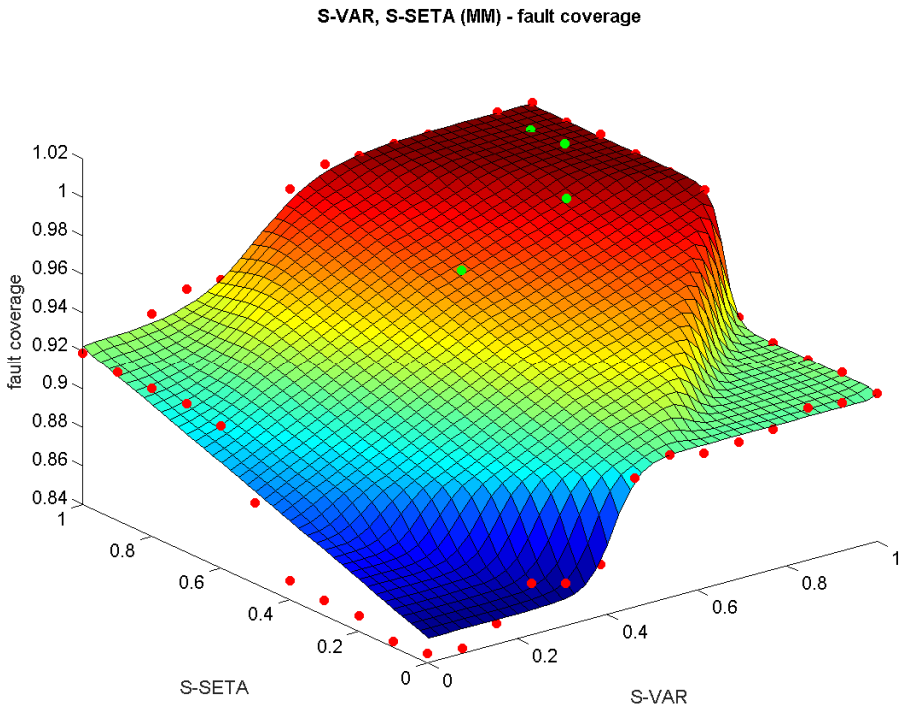


Fig. 6.125: Fault coverage for MM (46 points). The red dots are the input points and the green dots are the validation points.

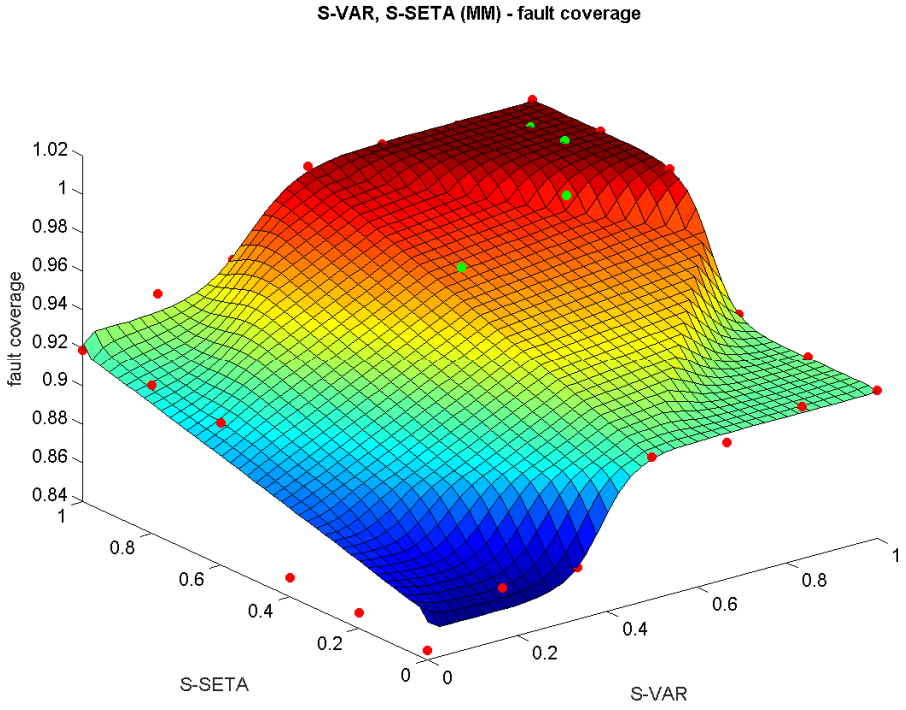


Fig. 6.126: Fault coverage for MM (22 points). The red dots are the input points and the green dots are the validation points.

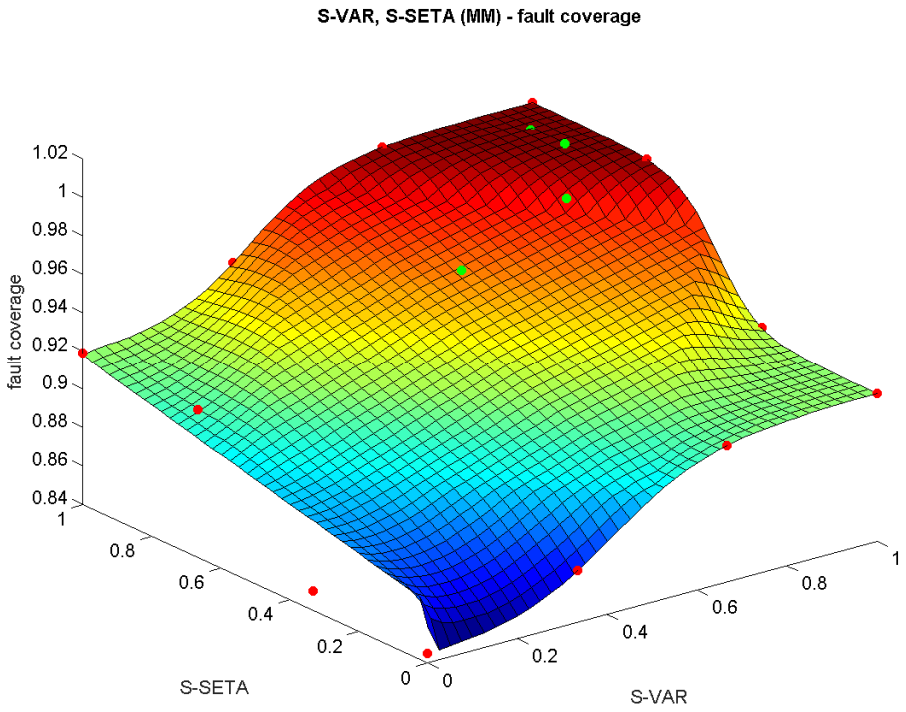


Fig. 6.127: Fault coverage for MM (12 points). The red dots are the input points and the green dots are the validation points.

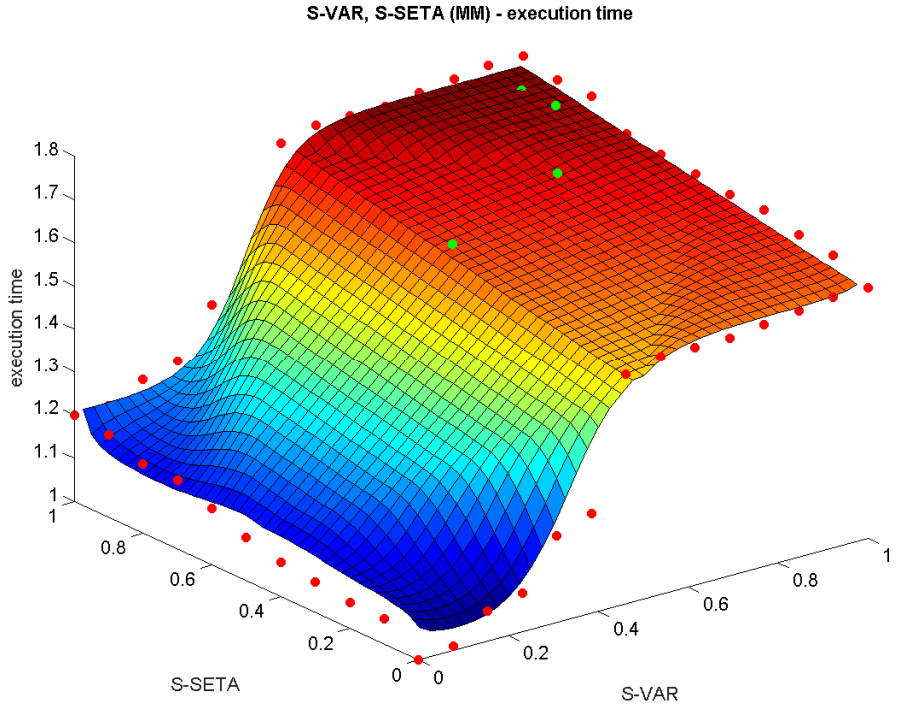


Fig. 6.128: Execution time for MM (46 points). The red dots are the input points and the green dots are the validation points.

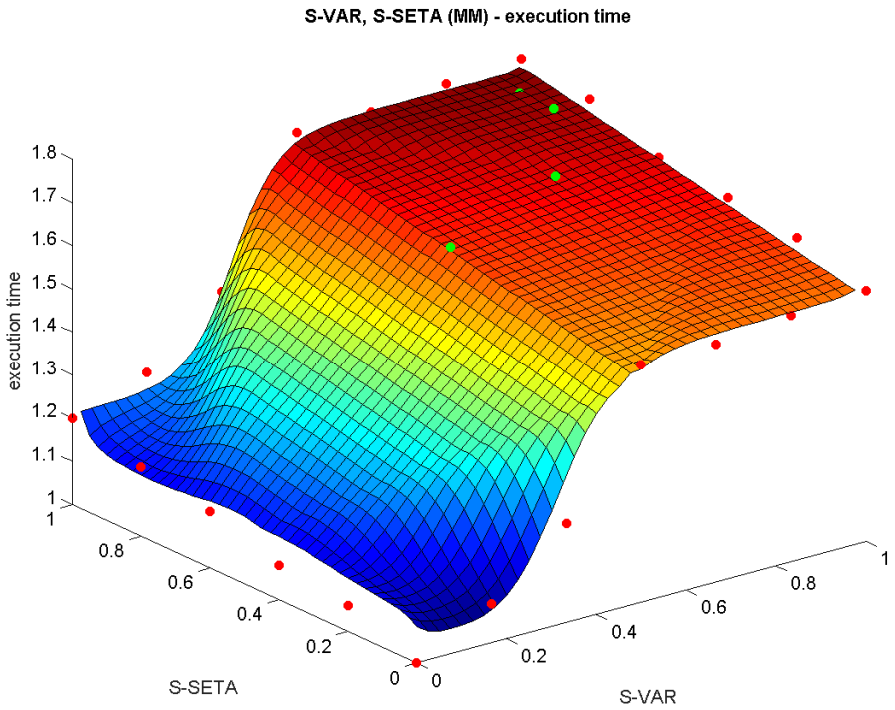


Fig. 6.129: Execution time for MM (22 points). The red dots are the input points and the green dots are the validation points.

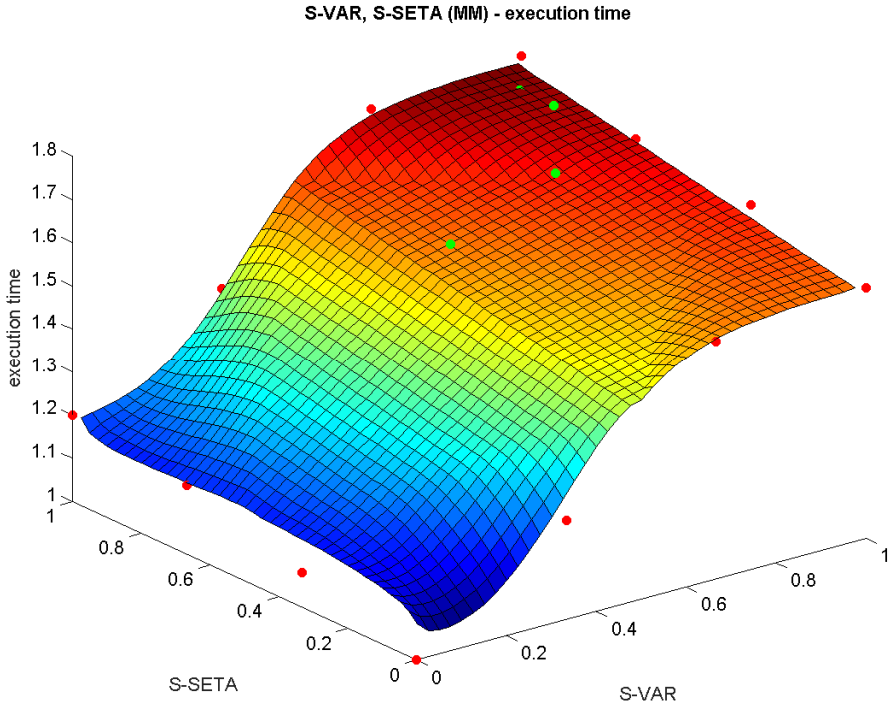


Fig. 6.130: Execution time for MM (12 points). The red dots are the input points and the green dots are the validation points.

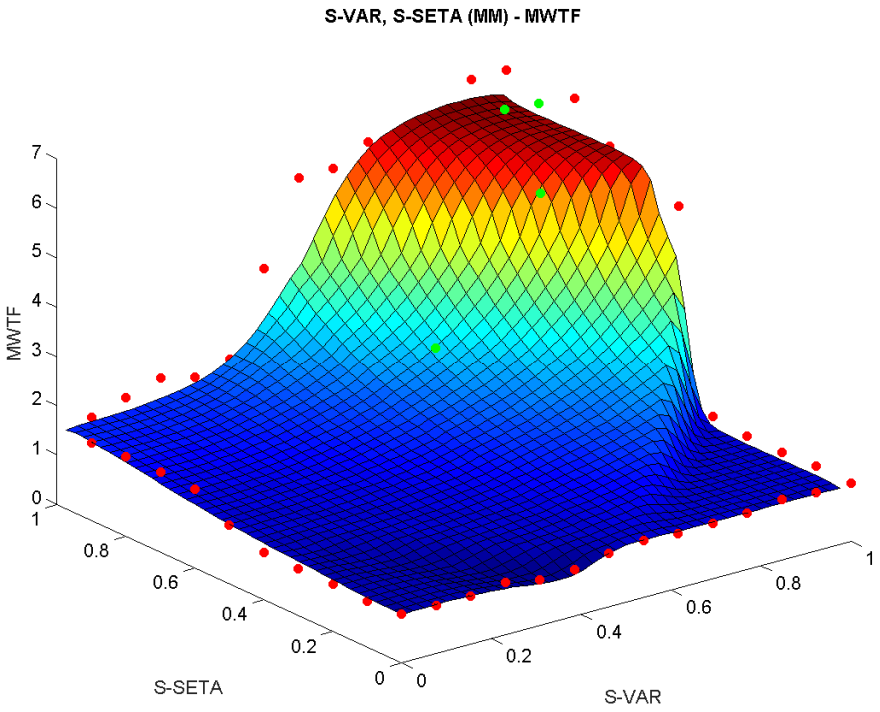


Fig. 6.131: MWTF for MM (46 points). The red dots are the input points and the green dots are the validation points.

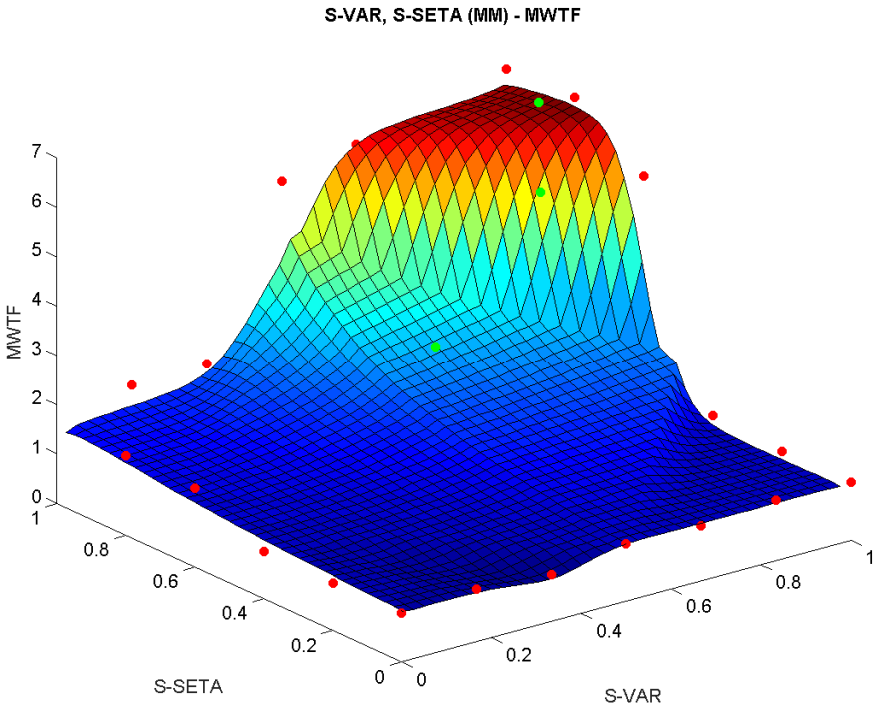


Fig. 6.132: MWTF for MM (22 points). The red dots are the input points and the green dots are the validation points.

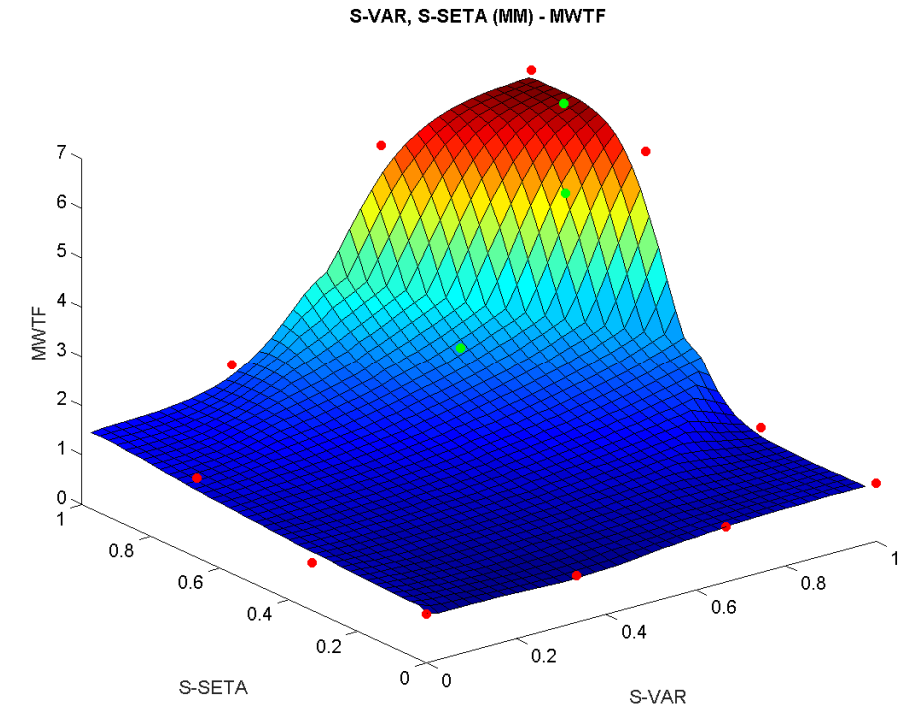


Fig. 6.133: MWTF for MM (12 points). The red dots are the input points and the green dots are the validation points.

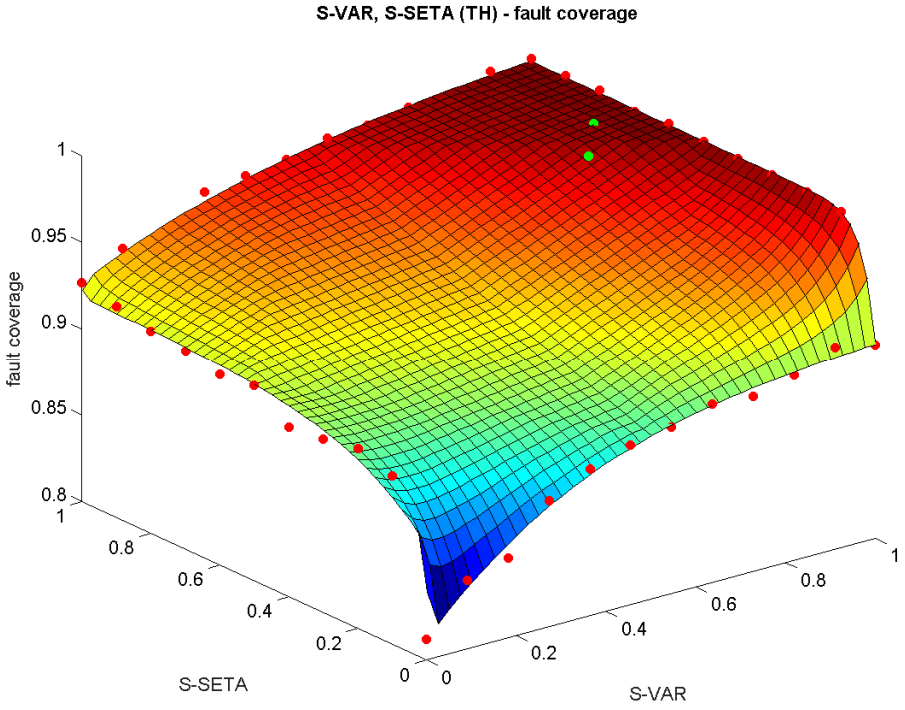


Fig. 6.134: Fault coverage for TH (42 points). The red dots are the input points and the green dots are the validation points.

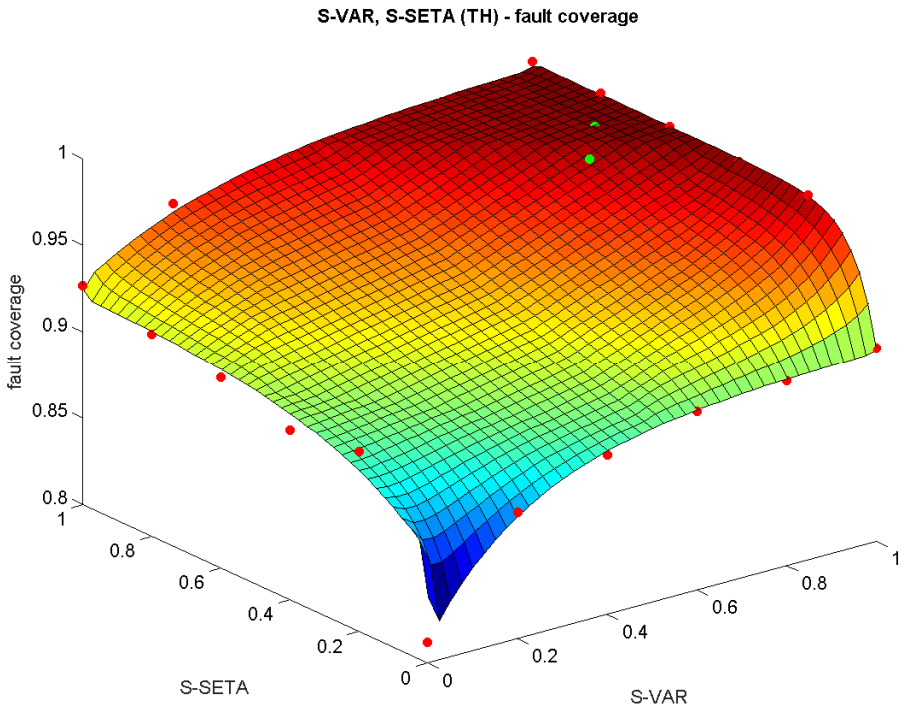


Fig. 6.135: Fault coverage for TH (20 points). The red dots are the input points and the green dots are the validation points.

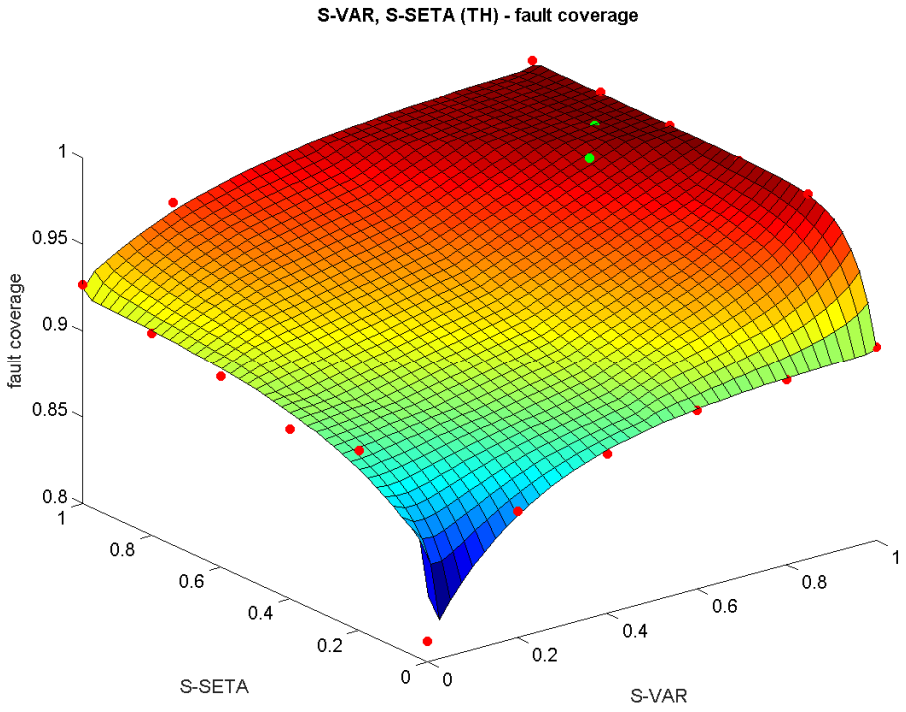


Fig. 6.136: Fault coverage for TH (12 points). The red dots are the input points and the green dots are the validation points.

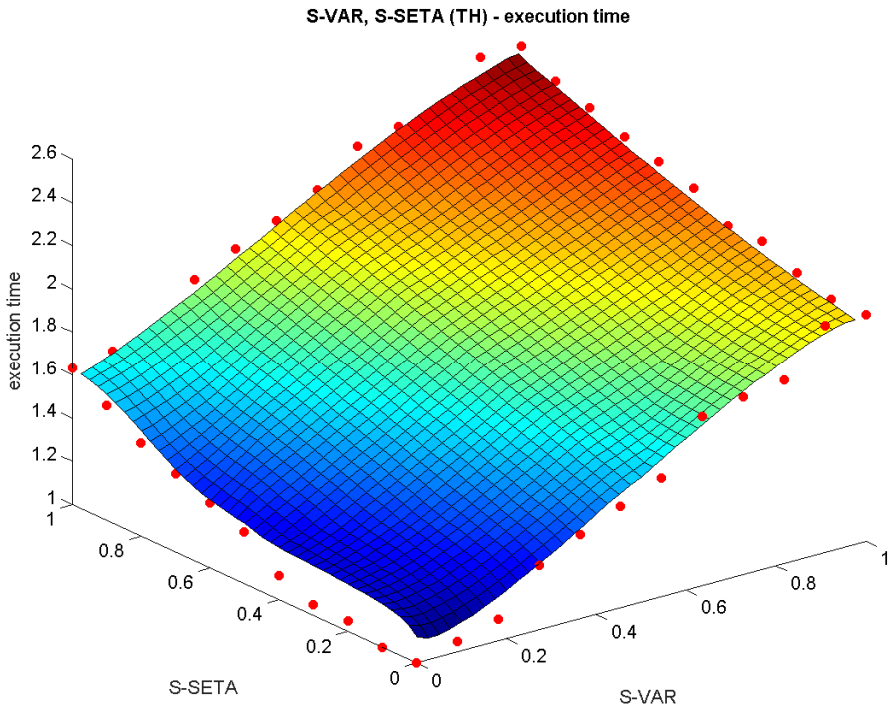


Fig. 6.137: Execution time for TH (42 points). The red dots are the input points and the green dots are the validation points.

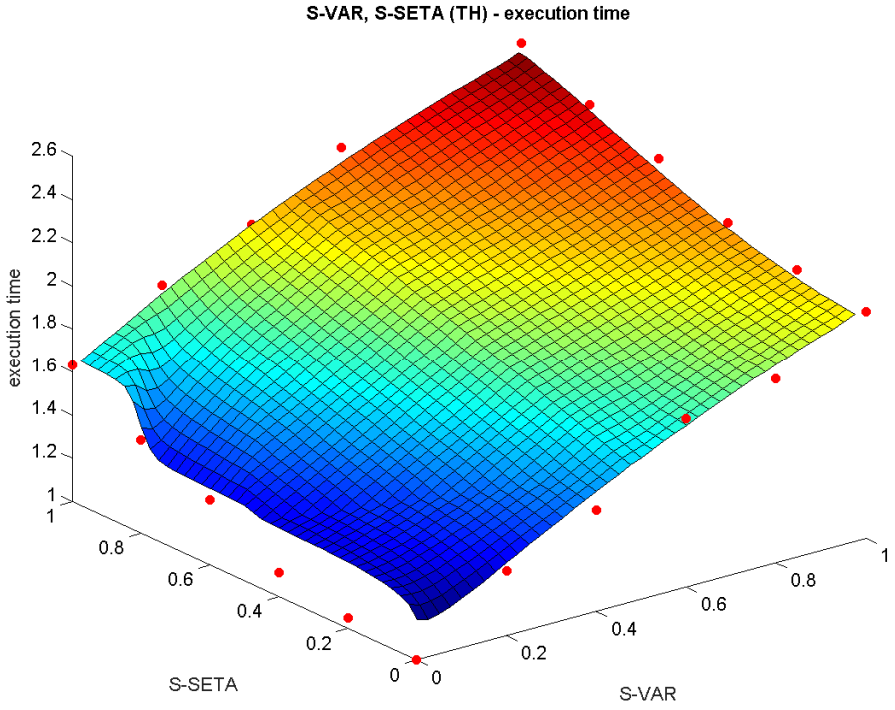


Fig. 6.138: Execution time for TH (20 points). The red dots are the input points and the green dots are the validation points.

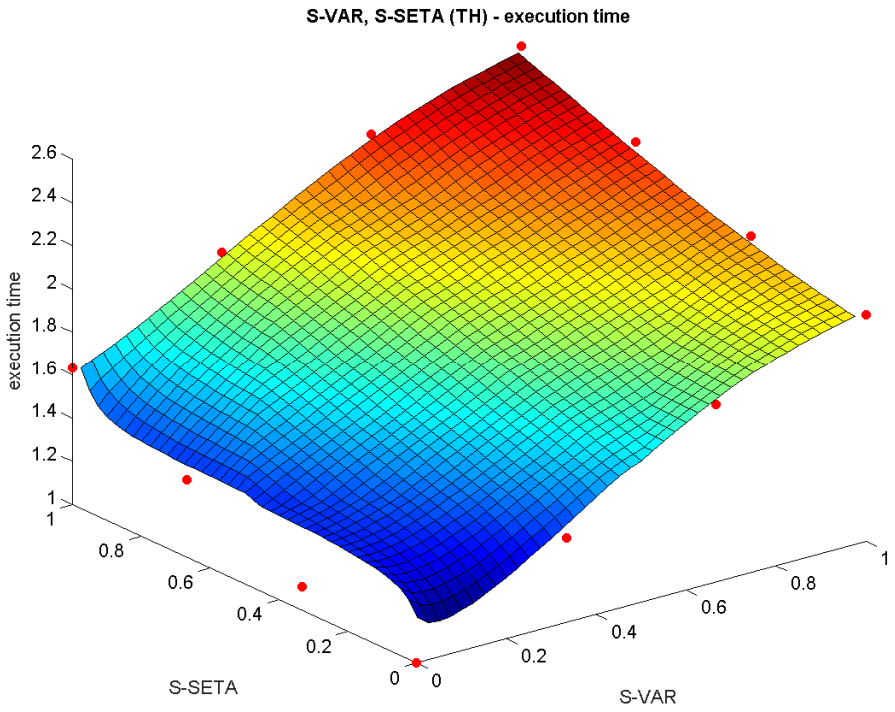


Fig. 6.139: Execution time for TH (12 points). The red dots are the input points and the green dots are the validation points.

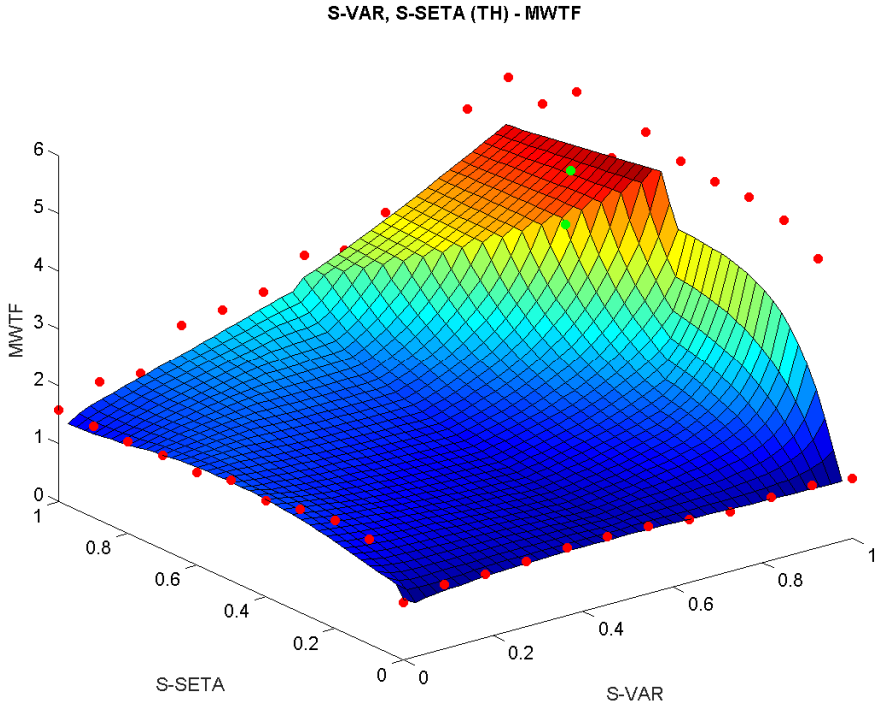


Fig. 6.140: MWTF for TH (42 points). The red dots are the input points and the green dots are the validation points.

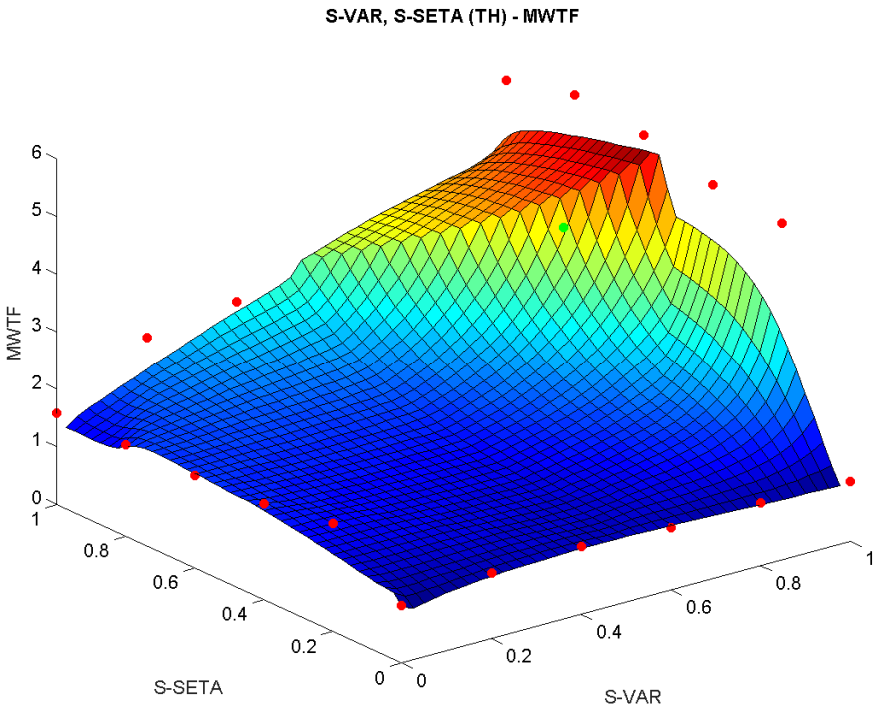


Fig. 6.141: MWTF for TH (20 points). The red dots are the input points and the green dots are the validation points.

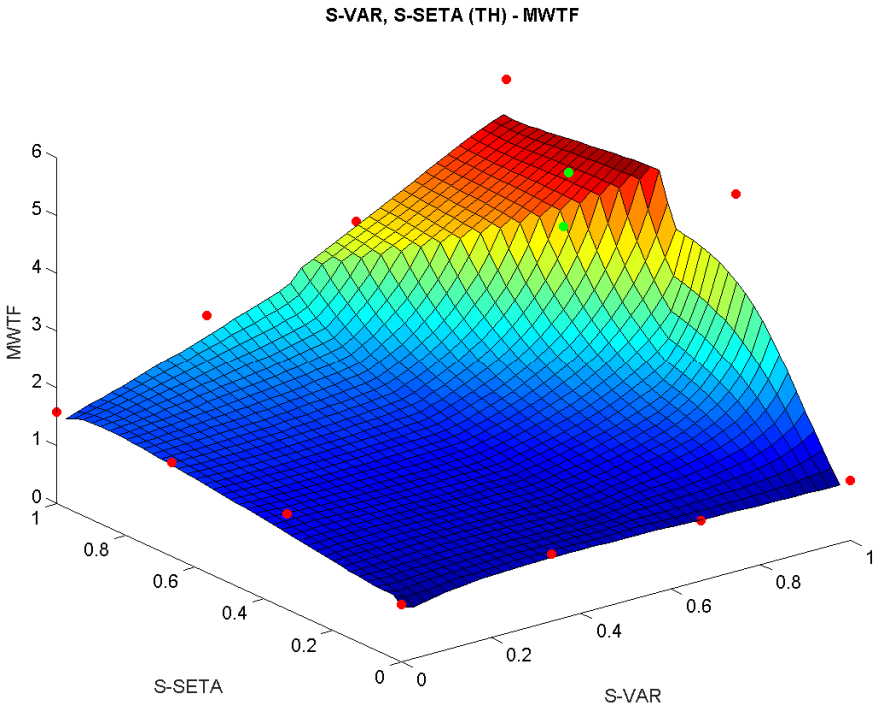


Fig. 6.142: MWTF for TH (12 points). The red dots are the input points and the green dots are the validation points.

6.4 Summary

In this chapter, we proposed and applied selective hardening methods to data-flow and control-flow techniques. The selective hardening of data-flow techniques consists of selecting the registers to be hardened. Considering S-VAR, SETA, we observed that the more registers are hardened, the higher the reliability, even when normalizing the results by the execution time (which is higher for when more registers are hardened). Nevertheless, it is important to notice that the reliability increases more for hardening the most critical registers than the extra reliability provided by the subsequent registers (based on the rank of criticality). There are two justifications for that: (1) the most critical registers are more critical; therefore, it increases more the fault coverage when hardened; and (2) the more registers are hardened, the more the protections of the registers overlap. In summary, it means that it is still possible to achieve high fault coverages when hardening fewer registers.

With regards to selective hardening on control-flow techniques, we compared two approaches: SETA-C and S-SETA. SETA-C provides high reliability even when few basic blocks are selected. That is justified because SETA-C protects all basic blocks and select which basic blocks will receive checkers. Thus, a fault affecting a basic block may propagate and may be detected in one of the following basic blocks. The downside is that SETA-C does not reduce very much the overheads. In some cases, hardening 70% of the basic blocks with S-SETA presents lower overheads than hardening 10% of the basic blocks with SETA-C. S-SETA implements a new method for selecting the basic blocks when basic blocks are completely ignored. It does not improve the reliability when very few basic blocks are hardened. However, it achieves very high MWTF when more than half of the basic blocks are hardened. And that is mainly due to its extreme lower overheads.

Finally, to evaluate all the possibilities of using selective hardening with both data-flow techniques and control-flow techniques, and helping designers to get a global picture of the reliability for an application, a model to extrapolate the simulated results was proposed. The model showed accuracy estimating the fault coverage, execution time, and MWTF, and pointing the areas of higher MWTF, even when fewer points are used as input to the model.

7 CONCLUDING REMARKS

This chapter concludes the thesis. Firstly, the conclusions of the work are drawn. Then, a list of published works developed during the Ph.D. is presented.

7.1 Conclusions

This work presented a study to reduce the execution time and memory overheads without losing reliability. In this way, a set of data-flow techniques based on general building rules called VAR techniques was proposed. They include and extend previous data-flow techniques due to the general building rules. With the general building rules, a complete analysis of the data-flow techniques based on redundancy was possible. By following the MWTF results, it was shown that may be not worth to protect an application with only data-flow techniques because the drawbacks caused by the overheads may overcome the reliability provided by the technique. Nevertheless, data-flow techniques area meant to be used together with control-flow techniques. In this situation, the benefits provided by SIHFT techniques are clear, once that the MWTF achieved by combining data-flow and control-flow techniques is much higher than the unhardened application. In addition, the highest MWTF is achieved by a new data-flow technique (VAR3+) when combined with SETA.

SETA is a new control-flow technique that is 11.0% faster and occupies 10.3% fewer memory positions with similar fault coverage of a state-of-the-art technique. When using one of the most promising VAR data-flow techniques with SETA, a similar fault coverage to techniques present in the literature is achieved, with reduction of the overheads. In such scenario, the hardened applications reached an average MWTF of 5.17x.

A step further on reducing overheads or increasing the MWTF is the use of selective hardening. It can significantly reduce the overheads with none or small loss in fault coverage. Regarding the data-flow techniques, the selective hardening was implemented in order to find which registers should and which should not be hardened based on their criticality. One can notice that it is possible to achieve similar fault coverage hardening fewer registers than hardening all used ones. This information is very useful in many applications that do not have enough available registers to be assigned as replicas because high reliability can be achieved only hardening the most critical registers. Anyhow, the highest MWTF were achieved when most registers are hardened. Therefore, if there are enough unused registers, and the overheads still meet the application constraints, it is recommendable the protection of all registers.

Concerning selective hardening on control-flow techniques, two methods were evaluated using SETA. One hardening all basic blocks, but removing checkers from the least critical ones (called SETA-C), and another hardening only selected basic blocks and ignoring the remaining (called S-SETA). The last method is a novelty of this work. Both selective hardening methods can improve the MWTF of SETA by reducing the overheads. This improvement is more notable in S-SETA due to its significantly lower overheads. The highest MWTFs are usually achieved starting from 60% of basic blocks hardened on when using S-SETA. The results converge to SETA near to 100% of basic blocks hardened.

In summary, the two main contributions of the proposed SIHFT techniques and selective hardening methods are: (1) it was possible to reduce significantly the overheads with a small reduction in the fault coverage, which can bring reliability to applications with restrict time or energy constraints; and (2) it was possible to keep similar fault coverage of state-of-the-art SIHFT techniques and reduce the overheads. In other words, it means an increase in the reliability, because the application with the same fault coverage will be exposed for a shorter time.

Finally, it was proposed a method to extrapolate the fault coverage and execution time, and estimate the results achievable by the combination of selective data-flow and selective control-flow techniques. The method gives estimated results with high accuracy, which can significantly speed up a project since only a few cases need to be tested, instead of testing all possibilities, in order to find the best combination of selective data-flow technique and selective control-flow technique that suit the application requirements.

7.2 Future work

This work evaluated and improved SIHFT techniques. Nevertheless, there are some points that were not investigated in this work that could contribute to the improvement of SIHFT techniques. They are listed and discussed as follows:

- **Selection of basic blocks**: the criteria utilized in this work to select basic blocks were based on assumptions. Therefore, the evaluation of different criteria to select basic blocks could improve the selective control-flow techniques, or at least, provide data that support the assumptions
- **S-SETA-C**: two different approaches for selective hardening on control-flow techniques were tested. The results show the advantages of the proposed approach. However, it is possible to use both approaches together. Selecting the basic blocks that will be hardened using S-SETA, and then, removing checkers from some of the selected basic blocks, as in SETA-C, resulting in the S-SETA-C control-flow technique. This combined approach could improve the selective control-flow techniques
- **Extrapolation method**: the method to extrapolate the results using curve fitting on hyperbolic tangents matches very well the results. However, the linear interpolation of the four hyperbolic tangents may not be the best method to find a surface of fault coverage for S-VAR and S-SETA. Some method that can keep better the hyperbolic tangent characteristic of the curves could improve the method's precision. Furthermore, a generalization of the four 2D hyperbolic tangent equations in one single 3D general equation with a regression method that can fit the results in this equation is desirable
- **Reset and rollback**: this work proposed and improved software detection techniques. Any system with detection techniques needs a method to return to a safe state. It could be achieved by resetting the processing and restarting the application, or returning a previously saved state (rollback). This was not in the scope of this work, but it is crucial for using the proposed SIHFT techniques (or any detection technique) in real world applications
- **Other SIHFT techniques**: the data-flow techniques are based on replicating data and instructions and comparing the original values with their replicas. Other ways of hardening the data-flow that not the ones based on replication and comparison

must be investigated. They may provide reliability with very low overheads, or could be used to complement the current SIHFT techniques, as by improving the fault coverage, or as allowing a more aggressive use of selective hardening methods.

7.3 Publications

7.3.1 Book chapters

Overhead Reduction in Data-flow Software-Based Fault Tolerance Techniques. E. Chielle, F. L. Kastensmidt, and S. Cuenca-Asensi. FPGAs and Parallel Architectures for Aerospace Applications, part V, pp. 279-291. Springer International Publishing AG, Cham, part V, ch. 18, pp. 279-291, Jan. 2016. DOI 10.1007/978-3-319-14352-1_18.

7.3.2 Journals

Reliability on ARM Processors against Soft Errors through SIHFT Techniques. E. Chielle et al. IEEE Transactions on Nuclear Science, 2016. (accepted)

Analyzing the Impact of Radiation-induced Failures in Programmable SoCs. L. A. Tambara, P. Rech, E. Chielle, J. Tonfat, and F. L. Kastensmidt. IEEE Transactions on Nuclear Science, 2016. (accepted)

S-SETA: Selective Software-Only Error-Detection Technique Using Assertions. E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn. IEEE Transactions on Nuclear Science, vol. 62, no. 6, pp. 3088-3095, Dec. 2015. DOI 10.1109/TNS.2015.2484842.

Application-Based Analysis of Register File Criticality for Reliability Assessment in Embedded Microprocessors. F. Restrepo-Calle, S. Cuenca-Asensi, A. Martinez-Alvarez, E. Chielle, and F. L. Kastensmidt. Journal of Electronic Testing, vol. 31, no. 2, pp. 139-150, Apr. 2015. DOI 10.1007/s10836-015-5513-9.

Evaluating Selective Redundancy in Data-Flow Software-Based Techniques. E. Chielle, J. R. Azambuja, R. S. Barth, F. Almeida, and F. L. Kastensmidt. IEEE Transactions on Nuclear Science, vol. 6, no. 4, pp. 2768-2775, Aug. 2013. DOI 10.1109/TNS.2013.2266917.

7.3.3 Conferences

Hybrid Soft Error Mitigation Techniques for COTS Processor-based Systems. E. Chielle et al. Latin-American Test Symposium (LATS 2015), Foz do Iguaçu, Brazil.

Reliability on ARM Processors against Soft Errors by a Purely Software Approach. E. Chielle, F. Rosa, G. S. Rodrigues, F. L. Kastensmidt, R. Reis, and S. Cuenca-Asensi. Conference on Radiation Effects on Components and Systems (RADECS 2015), Moscow, Russia. DOI 10.1109/RADECS.2015.7365660.

Analyzing the Failure Impact of Using Hard- and Soft-cores in All Programmable SoC under Neutron-induced Upsets. L. A. Tambara, P. Rech, E. Chielle, and F. L. Kastensmidt. Conference on Radiation Effects on Components and Systems (RADECS 2015), Moscow, Russia. DOI 10.1109/RADECS.2015.7365586.

Selective Software Techniques to Detect Neutron-induced Soft Errors in Processors with Minimum Overhead. E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn. IEEE Nuclear and Space Radiation Effects Conference (NSREC 2015), Boston, USA.

Reducing Performance Degradation in Software Detection Techniques on Embedded Processors. E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, and S. Cuenca-Asensi. Military and Aerospace Programmable Logic Devices (MAPLD 2015), San Diego, USA.

Software Error-Detection Techniques with Reduced Overheads on Embedded Processors. E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, and S. Cuenca-Asensi. Simpósio Sul de Microeletrônica (SIM 2015), Santa Maria, Brazil.

Tuning Software-based Fault-tolerance Techniques for Power Optimization. E. Chielle, F. L. Kastensmidt, and S. Cuenca-Asensi. 24th IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2014), Palma de Mallorca, Spain. DOI 10.1109/PATMOS.2014.6951871.

Tuning Software-based Fault-tolerance Techniques for Soft-core Processors. E. Chielle, F. L. Kastensmidt, and S. Cuenca-Asensi. FPGAs for Aerospace Applications (FASA 2014). Munich, Germany.

Efficient metric for register file criticality in processor-based systems. F. Restrepo-Calle, S. Cuenca-Asensi, A. Martinez-Alvarez, E. Chielle, and F. L. Kastensmidt. 15th IEEE Latin American Test Workshop (LATW 2014), pp. 1-6. Fortaleza, Brazil. DOI 10.1109/LATW.2014.6841922.

Evaluating Software-Based Fault Detection Techniques Applied at Different Programming Abstraction Levels. E. Chielle, J. R. Azambuja, and F. L. Kastensmidt. 10th Workshop on Silicon Errors in Logic – System Effects (SELSE 2014). Stanford, USA.

Comparing Software-Based Fault Detection Techniques Applied at Different Abstraction Levels. E. Chielle, D. H. Grehs, J. R. Azambuja, and F. L. Kastensmidt. Radiation Effects on Components and Systems (RADECS 2013). Oxford, UK. DOI 10.1109/RADECS.2013.6937383.

Evaluating the Effectiveness of a Diversity TMR Scheme under Neutrons. L. A. Tambara, J. R. Azambuja, E. Chielle, F. Almeida, G. L. Nazar, P. Rech, M. S. Lubaszewski, F. L. Kastensmidt, and C. Frost. Radiation Effects on Components and Systems (RADECS 2013). Oxford, UK. DOI 10.1109/RADECS.2013.6937382.

Improving error detection with selective redundancy in software-based techniques. E. Chielle, J. R. Azambuja, R. S. Barth, and F. L. Kastensmidt. 14th IEEE Latin American Test Workshop (LATW 2013). Cordoba, Argentina. DOI 10.1109/LATW.2013.6562659.

Evaluating Selective Redundancy in Data-flow Software-based Techniques. E. Chielle, J. R. Azambuja, R. S. Barth, F. Almeida, and F. L. Kastensmidt. Radiation and its Effects on Components and Systems (RADECS 2012). Biarritz, France.

Single-Event-Induced Charge Sharing Effects in TMR with Different Levels of Granularity. F. Almeida, F. L. Kastensmidt, S. Pagliarini, L. Entrena, A. Lindoso, E. S. Millan, E. Chielle, L. Naviner, and J. F. Naviner. Radiation and its Effects on Components and Systems (RADECS 2012). Biarritz, France.

Soft Error Rate Reduction by Using Configurable SET Temporal Filtering. J. E. Souza, F. L. Kastensmidt, F. Almeida, and E. Chielle. Radiation and its Effects on Components and Systems (RADECS 2012). Biarritz, France.

Soft-Error Probability Due to SET in Clock Tree Networks. R. Chipana, E. Chielle, F. L. Kastensmidt, J. Tonfat, and R. Reis. IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2012), Amherst, USA. DOI 10.1109/ISVLSI.2012.39.

REFERENCES

AGUIAR, V.A.P. et al. Experimental setup for Single Event Effects at the São Paulo 8UD Pelletron Accelerator. **Nuclear Instruments & Methods in Physics Research**, vol. 332, pp. 397-400, Aug. 2014.

ALKHALIFA, Z.; NAIR, V.S.S.; KRISHNAMURTHY, N.; ABRAHAM, J.A. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Transactions on Parallel and Distributed Systems**, New York, vol. 10, no. 6, p. 627-641, Jun. 1999.

AMAZON. Amazon Best Sellers: Best Computer CPU Processors, 2015. Available at: <http://www.amazon.com/gp/bestsellers/electronics/229189/>. Accessed on: Aug. 2015.

ANTHONY, S. Boeing 787 Dreamliner: Powered by Android, and 69TB of solid-state storage. **ExtremeTech**, Jul. 2012.

ARM. ARM – The Architecture For The Digital World, 2015. Available at: <http://www.arm.com/>. Accessed on: Aug. 2015.

ARM. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. Cambridge: ARM Limited, 2014.

ARM. The ARM Cortex-A9 Processors, 2009. Available at: <http://www.arm.com/files/pdf/armcortexa-9processors.pdf>. Accessed on: May 2015.

ASENSI, S.C.; ALVAREZ, A.M.; CALLE, F.R.; PALOMO, F.R.; MIRANDA, H.G.; AGUIRRE, M.A. A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems. **IEEE Transactions on Nuclear Science**, vol. 58, no. 3, pp. 1059-1065, Jun. 2011.

ASSAYAD, I.; GIRAULT, A.; KALLA, H. Tradeoff Exploration between reliability, power consumption and execution Time for embedded systems. **International Journal on Software Tools for Technology Transfer**, vol. 15, no. 3, pp. 229-245, Jun. 2013.

AVIZIENIS, A.; LAPRIE, J.C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, vol. 1, no. 1, pp. 11-33, Mar. 2004.

AVNET. ZedBoard, featuring the Zynq-7000 All Programmable SoC, 2015. Available at: <http://www.em.avnet.com/en-us/design/drc/Pages/Zedboard.aspx>. Accessed on: May 2015.

AVNET. ZedBoard (Zynq Evaluation and Development) Hardware User's Guide, 2014. Available at: <http://zedboard.org/support/documentation/1521>. Accessed on: May 2015.

AZAMBUJA, J.R.; LAPOLLI, A.; ALTIERI, M.; KASTENSMIDT, F.L. Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors. **IEEE Latin American Symposium on Circuits and Systems**, Mar. 2011a.

AZAMBUJA, J.R.; LAPOLLI, A.; ROSA, L.; KASTENSMIDT, F.L. Detecting SEEs in microprocessors through a non-Intrusive hybrid technique. **IEEE Transactions on Nuclear Science**, vol. 58, no. 3, p. 993-1000, Jun. 2011b.

AZAMBUJA, J.R., PAGLIARINI, S., ROSA, L., KASTENSMIDT, F.L. Exploring the limitations of software-only techniques in SEE detection coverage. **Journal of Electronic Testing**, vol. 27, no. 4, pp. 541-550, Aug. 2011c.

AZAMBUJA, J.R.; ALTIERI, M.; BECKER, J.; KASTENSMIDT, F.L. HETA: Hybrid Error-Detection Technique Using Assertions. **IEEE Transactions on Nuclear Science**, vol. 60, no. 4, pp. 2805-2812, Aug. 2013.

BAUMANN, R. Soft errors in advanced semiconductor devices-part I: the three radiation sources. **IEEE Transactions on Device and Materials Reliability**, Los Alamitos, USA, vol. 1, no. 1, p. 17-22, Mar. 2001.

BELCASTRO, C.M.; EURE, K.; HESS, R. Testing a Flight Control System for Neutron-Induced Disturbances. **Los Alamos Science**, no. 30, pp. 104-111, 2006.

BOUDENOT, J. Radiation Space Environment. In: VELAZO, R.; FOUILLAT, P; REIS, R. Radiation Effects on Embedded Systems. **Springer Netherlands**, Dordrecht, pp. 1-9, 2007.

CASTRO NETO, J.C; STEFANI, M.; BARBALHO, S. A indústria e os obstáculos ao desenvolvimento de pesquisas, produtos e aplicações na área espacial no Brasil. In: ROLLEMBERG, R. A política espacial Brasileira, parte II - Análises Técnicas, **Cadernos de Altos Estudos, Câmara dos Deputados, Edições Câmara**, Brasília, vol. 7, pp. 17-35, 2010.

CHEN, K.Y.; HSU, P.H.; CHAO, K.M. Hardness of comparing two run-length encoded strings. **Journal of Complexity**, vol. 26, no. 4, pp. 364-374, Aug. 2010.

CHEYNET, P; NICOLESCU, B.; VELAZCO, R.; REBAUDENGO, M.; REORDA, M. S.; VIOLANTE, M. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. **IEEE Transactions on Nuclear Science**, vol. 47, no. 6, pp. 2231-2236, Dec. 2000.

CHIELLE, E.; AZAMBUJA, J.R.; BARTH, R.S.; ALMEIDA, F.; KASTENSMIDT, F.L. Evaluating Selective Redundancy in Data-Flow Software-Based Techniques. **IEEE Transactions on Nuclear Science**, vol. 6, no. 4, pp. 2768-2775, Aug. 2013.

CHIELLE, E.; AZAMBUJA, J.R.; KASTENSMIDT, F.L. Evaluating Software-Based Fault Detection Techniques Applied at Different Programming Abstraction Levels. **Workshop on Silicon Errors in Logic – System Effects (SELSE)**. Stanford, Apr. 2014.

CHIELLE, E.; BARTH, R.S.; LAPOLLI, A.C.; KASTENSMIDT, F.L. Configurable Tool to Protect Processors against SEE by Software-based Detection Techniques. **Latin American Test Workshop**, Quito, Apr. 2012.

CHIELLE E. et al. Hybrid Soft Error Mitigation Techniques for COTS Processor-based Systems. **Latin American Test Symposium**, Foz do Iguaçu, Apr. 2016.

COBHAM GAISLER AB. Processors, 2015. Available at: <http://www.gaisler.com/index.php/products/processors>. Accessed on: Aug. 2015.

CONG, J.; GURURAJ, K. Assuring application-level correctness against soft errors. **IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**, San Jose, pp. 150-157, Nov. 2011.

DIJKSTRA, E.W. A note on two problems in connexion with graphs. **Numerische Mathematik**, vol. 1, no. 1, pp. 269-271, Dec. 1959.

DU, B.; SONZA REORDA, M.; STERPONE, L.; PARRA, L.; PORTELA-GARCIA, M.; LINDOSO, A.; ENTRENA, L. On-line Test of Control Flow Errors: A new Debug Interface-based approach. **IEEE Transactions on Computers**, vol. PP, no. 99, Jul. 2015.

E2V. Space grade semiconductor solutions, 2015. Available at: <http://www.e2v.com/resources/account/download-literature/113>. Accessed on: Aug. 2015.

EBAY. Computer Processors | eBay, 2015. Available at: <http://www.ebay.com/sch/CPUs-Processors-/164/i.html>. Accessed on: Aug. 2015.

ESA. ESA/SCC Basic Specification No. 25100: Single Event Effects Test Methods and Guidelines. **European Space Components Coordination**, Noordwijk, no. 2, pp. 1-24, Oct. 2014.

FERLET-CAVROIS. V. et al. Direct measurement of transient pulses induced by laser irradiation in deca-nanometer SOI devices. **IEEE Transactions on Nuclear Science**, vol. 52, pp. 2104-2113, 2005.

FERNANDEZ-LEON, A.; GARDENYES, R. Trends and patterns of ASIC and FPGA use in European space missions. Master thesis, **TUDelft and European Space Agency (ESA)**, 2013.

GINOSAR, R. Survey of Processors for Space. **Data System in Aerospace**, 2012.

GOLOUBEVA, O.; REBAUDENGO, M.; REORDA, M.S.; VIOLANTE, M. Soft-error detection using control flow assertions. **IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems**, p. 581–588, Nov. 2003.

GOLOUBEVA, O.; REBAUDENGO, M.; REORDA, M.S.; VIOLANTE, M. Software-Implemented Hardware Fault Tolerance. **Springer US**, 2006.

HANGOUT, L.M.O.S.S.; JAN, S. The minimips project, 2009. Available at: <http://www.opencores.org/projects.cgi/web/minimips/overview>. Accessed on: Mar. 2012.

ITRS. Design. In: International Technology Roadmap for Semiconductors, pp. 6-7, 2005.

KIM, N. et al. Leakage current: Moore's law meets static power. **IEEE Computer**, vol. 36, no. 12, pp. 68-75, Dec. 2003.

KOUBA, D. The Algebra of Summation Notation, 1999. Available at: <https://www.math.ucdavis.edu/~kouba/CalcTwoDIRECTORY/summationdirectory/Summation.html>. Accessed on: May 2015.

LI, S.; LAI, E.M.K.; ABSAR, M.J. Minimizing Embedded Software Power Consumption Through Reduction of Data Memory Access. **International Conference on Information, Communications & Signal Processing**, vol. 1, pp. 309-313, Dec. 2003.

MAHMOOD, A.; McCLUSKEY, E. Concurrent error detection using watchdog processors – a survey. **IEEE Transaction on Computers**, vol. 37, no. 2, pp. 160-174, Feb. 1988.

MARTINEZ-ALVAREZ, A.; CUENCA-ASENSI, S.; RESTREPO-CALLE, F.; PINTO, F.R.P.; GUZMAN-MIRANDA, H.; AGUIRRE, M.A. Compiler-Directed Soft Error Mitigation for Embedded Systems. **IEEE Transactions on Dependable and Secure Computing**, vol. 8, no. 2, pp. 159-172, Mar. 2012.

MAXWELL TECHNOLOGIES. SCS750 Super Computer for Space, 2015. Available at: <https://www.maxwell.com/images/documents/scs750_rev7.pdf>. Accessed on: Mar. 2016.

MCFEARIN, L.D.; NAIR, V.S.S. Control-flow checking using assertions. **IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-05)**, Urbana-Champaign, pp. 103-112, Sept. 1995.

MCHALE, J. Budget cuts pressuring rad-hard designers to maintain quality while cutting costs. **Military Embedded Systems**, Jun. 2014.

MENTOR GRAPHICS. ModelSim, 2010. Available at: <http://www.model.com/content/modelsim-support>. Accessed on: Mar. 2012.

MORISON, I. Introduction to Astronomy and Cosmology. **John Wiley & Sons**, Dec. 2008.

MUKHERJEE, S.S.; WEAVER, C.; EMER, J.; REINHARDT, S.K.; AUSTIN, T. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. **Annual IEEE/ACM International Symposium on Microarchitecture**, pp. 29-40, 2003.

NICOLESCU, B.; VELAZCO, R. Detection soft errors by a purely software approach: method, tools and experimental results. **Design, Automation and Test in Europe Conference and Exhibition**, pp. 57-62, 2003.

O'BRYAN, M. Radiation Effects & Analysis, Nov. 2015. Available at: <http://radhome.gsfc.nasa.gov/radhome/see.htm>. Accessed on: Dec. 2015.

OH, N.; SHIRVANI, P.P.; McCLUSKEY, E.J.; Error detection by duplicated instructions in super-scalar processors. **IEEE Transactions on Reliability**, vol. 51, no. 1, pp. 63–75, Mar. 2002a.

OH, N.; SHIRVANI, E.; McCLUSKEY, E. Control-flow checking by software signatures. **IEEE Transactions on Reliability**, vol. 51, no. 2, pp. 111-122, Mar. 2002b.

PARKINSON, D.W. TETRA Security. **BT Technology Journal**, vol. 19, np. 3, pp. 81-88, 2001.

PILOTTO, C.; AZAMBUJA, J.R.; KASTENSMIDT, F.L. Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications. **Symposium on Integrated Circuits and Systems Design**, Gramado, pp. 199-204, 2008.

PRADHAN, D. Fault-Tolerant Computer System Design. **Prentice Hall**, Upper Saddle River, 1996.

REBAUDENGO, M; REORDA, M.S.; TORCHIANO, M.; VIOLANTE, M. Soft-error detection through software fault-tolerance techniques. **IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems**, Albuquerque, pp. 210-218, 1999.

REIS, G.A.; CHANG, J.; AUGUST, D.I. Automatic Instruction-Level Software-Only Recovery, **IEEE Micro**, vol. 27, no. 1, pp. 36-47, 2007.

REIS, G.A.; CHANG, J; VACHHARAJANI, N.; MUKHERJEE, S.S.; RANGAN, R.; AUGUST, D.I. Design and Evaluation of Hybrid Fault-Detection Systems. **International Symposium on Computer Architecture**, pp. 148-159, Jun. 2005a.

REIS, G.A.; CHANG, J.; VACHHARAJANI, N.; RANGAN, R.; AUGUST, D.I. SWIFT: software implemented fault tolerance. **Symposium on Code Generation and Optimization**, San Francisco, pp. 243-254, 2005b.

RESTREPO-CALLE, F. Co-diseño de sistemas hardware/software tolerantes a fallos inducidos por radiación. Ph.D. thesis, **Computer Technology Department, University of Alicante**, pp. 78, 2011.

RESTREPO-CALLE, F.; CUENCA-ASENCI, S.; MARTINEZ-ALVAREZ, A. Reducing implicit overheads of soft error mitigation techniques using selective hardening. In: KASTENSMIDT, F.; RECH, P. FPGAs and Parallel Architectures for Aerospace Applications. **Springer International Publishing**, 2016.

RESTREPO-CALLE, F.; CUENCA-ASENSI, S., MARTINEZ-ALVAREZ, A.; CHIELLE, E.; KASTENSMIDT, F.L. Application-Based Analysis of Register File Criticality for Reliability Assessment in Embedded Microprocessors. **Journal of Electronic Testing**, vol. 31, no. 2, pp. 139-150, Apr. 2015.

RESTREPO-CALLE, F.; MARTINEZ-ALVAREZ, A.; CUENCA-ASENSI, S.; JIMENO-MORENILLA, A. Selective SWIFT-R. **Journal of Electronic Test**, vol. 29, no. 6, pp. 825-838, Dec. 2013.

STURESSON, F. Single Event Effects (SEE) Mechanism and Effects. **Space Radiation and its Effects on EEE Components**, Jun. 2009.

SUDARAM, A.; AKAEL, A.; LOCKHART, D.; THAKER, D.; FRANKLIN, D. Efficient fault tolerance in multi-media applications through selective instruction replication. **Workshop on Radiation effects and fault tolerance in nanometer technologies**, pp. 339-346, 2008.

TEXAS INSTRUMENTS. TMS320C6748 | C674x DSP | C6000 DSP | Description & parametrics. Available at: <http://www.ti.com/product/tms320c6748>. Accessed on: Aug. 2015.

THOMPSON, S.E.; CHAU, R.S.; GHANI, T.; MISTRY, K.; TYAGI, S.; BOHR, M.T. In search of "Forever," continued transistor scaling one new material at a time. **IEEE Transactions on Semiconductor Manufacturing**, New York, vol. 18, no. 1, pp. 26-36, Feb. 2005.

UNSAL, O.S.; KOREN, I.; KRISHNA, C.M. Towards Energy-Aware Software-Based Fault Tolerance in Real-Time Systems. **International Symposium on Low Power Electronics and Design**, 2002.

VEMU, R.; ABRAHAM, J. CEDA: Control-Flow Error Detection Using Assertions. **IEEE Transactions on Computers**, vol. 60, no. 9, pp. 1233-1245, Sept. 2011.

VIOLANTE, M. et al. A New Hardware/Software Platform and a New 1/E Neutron Source for Soft Error Studies: Testing FPGAs at the ISIS Facility. **IEEE Transactions on Computers**, vol. 54, no. 4, pp. 1184-1189, Aug. 2007.

VOGELSANG, T. Understanding the Energy Consumption of Dynamic Random Access Memories. **Annual IEEE/ACM International Symposium on Microarchitecture**, 2010.

WAKERLY, J. Error detection codes, self-checking circuits and applications, **North-Holland**, New York, 1978.

WANG, P.S.P. Handbook of Optical Character Recognition and Document Image Analysis. **World Scientific Publishing Company**, 1997.

WANG, W.; DEY, T. A Survey on ARM Cortex A Processors, 2011. Available at: <http://www.cs.virginia.edu/~skadron/cs8535_s11/ARM_Cortex.pdf>. Accessed on: Aug. 2015.

WURSTER, S.; EGYEDI, T.M.; HOMMELS, A. The development of the public safety standard TETRA: lessons and recommendations for research managers and strategists in the security industry. **International Conference on Standardization and Innovation in Information Technology (SIIT)**, 2013.

YAO, T.; ZHOU, H.; FANG, M.; HU, H. Low Power Consumption Scheduling Based on Software Fault-tolerance. **International Conference on Natural Computation**, 2013.

YEH, T.Y.; REINMAN, G.; PATEL, S.J.; FALOUTSOS, P. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. **ACM Transactions on Graphics**, vol. 29, no. 1, pp. 5.1-5.11, Dec. 2009.

APPENDIX A <CFT-TOOL>

CFT-tool (Configurable Fault-Tolerant tool) is a configurable tool that applies SIHFT techniques to the assembly code of target applications (CHIELLE, 2012). Its configurability comes from two aspects:

- **Processor**: the target processor is informed by configuration files that describe the processor architecture and organization. Thus, different processors can be targeted by modifying the configuration files
- **Techniques**: a set of data-flow and control-flow SIHFT techniques is available in the CFT-tool. It is possible to select the techniques, the order that they are applied, and the registers and basic blocks that shall be hardened.

Fig. A.1 shows the steps that a program pass until a hardened executable is created. The code in high-level language is compiled, generating the equivalent assembly code. After, the code is assembled, creating the executable, which is, then, disassembled. CFT-tool reads the assembly code, the disassembly, and the configuration files about the target processor and techniques. The assembly is the base code to create the hardened application. The code from subroutines presents in libraries, which is not present in the assembly code, is extracted from the disassembly. Information about the processor, such as the instruction set, register file, is read from the configuration files about the processor. And the configuration files about the techniques informs the selected techniques, and how they shall be applied. Then, CFT-tool creates a new assembly code hardened by the selected techniques. The hardened assembly is assembled, creating the hardened executable, which is ready to use.

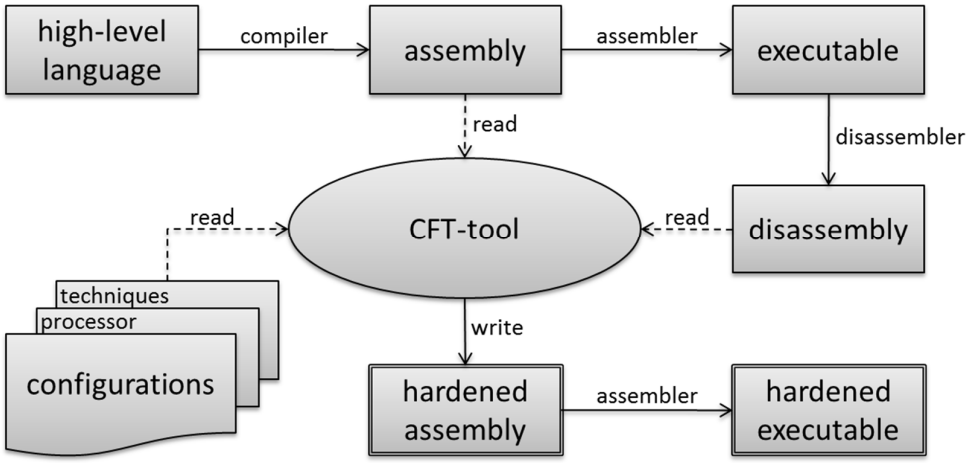


Fig. A.1: Steps to protect a code using the CFT-tool.

A.1 Configuration

CFT-tool is independent of the processor architecture and organization. Considering that the SIHFT techniques are applied by CFT-tool to the assembly code of the target application and that the assembly code is architecture dependent, it is necessary to provide information about the processor to the tool. All instructions, registers, as well as many

other vital characteristics of the processor, must be informed in order to CFT-tool work properly.

In this regard, there is a set of configuration files that describe the processor architecture and organization. One of these files contains general configurations. For example, the label format as showed in Fig. A.2; the mnemonic of instruction to check if two values are different (Fig. A.3); if there is the implementation of the *branch delay slot*[14] informing how many instructions are reordered (Fig. A.4); or the mnemonics of the logically inverse conditional branches, as showed in Fig. A.5.

```
[LABEL_FORMAT]
label:
```

Fig. A.2: Example of label format configuration.

```
[NON_EQUALITY_COMPARISON_INSTRUCTION]
bne
```

Fig. A.3: The *branch not equal* must be informed because it is necessary for the implementation of checkers.

```
[BRANCH_DELAY_SLOT]
1
```

Fig. A.4: Number of instructions reordered by the *branch delay slot*.

```
[INVERTER_INSTRUCTIONS]
bne:beq, bltz:bgez, bgtz:blez, beqz:bnez, bgezal:bltzal
```

Fig. A.5: Configuration informing the logically inverse conditional branches.

Another configuration file contains information about the instructions, such as mnemonic, format, and type. Fig. A.6 presents an example of an instruction format, *ins* indicates where the mnemonic is placed, *rd* concerns to a destination register, *rs* refers to the source register, and *offset* is the memory offset with regards to *rs*.

```
ins rd,offset(rs)
```

Fig. A.6: Example of an instruction format.

The instructions are organized in groups. Fig. 3.8 presents the configuration of a group of instructions. The tag *[GROUP]* marks the start of a group, while *{INSTRUCTIONS}* indicates the mnemonics, *{FORMAT}* identifies the format of these instructions in the assembly code, and *{TYPE}* points the type of the instructions (arithmetic, in the example).

[14] *Branch delay slot* is a feature of the pipeline present in some processors, in which the instructions immediately before branches are shifted down (to after the branch) in order to increase performance because the processor will always execute the instructions subsequent to branches.

```
[GROUP]
{INSTRUCTIONS}
add, addu, and, nor, or, sllv, slt, sltu, srav, srlv, sub, subu, xor
{FORMAT}
    ins rd,rs,rs
{TYPE}
arithmetic
```

Fig. A.7: Example of a group of instructions.

CFT-tool recognizes instruction overload, i.e., instructions with the same mneumonic but slightly different features. For example, an *add* instruction that sum two registers and another *add* that sum a register with an immediate. Fig. A.8 shows another group of instructions containing instructions with the same mnemonic of Fig. A.7. By comparing both figures, it is possible to notice that the format in both cases is different. One has two registers, and the other has one register and one immediate.

```
[GROUP]
{INSTRUCTIONS}
addi, addu, addiu, andi, ori, sll, slt, slti, sltiu, sltu, sra, srl, subu, xori
{FORMAT}
    ins rd,rs,imm
{TYPE}
arithmetic
```

Fig. A.8: Example of a group with overloaded instructions with regards to Fig. A.7.

As well as the instructions, the registers are configured in groups. In Fig. A.9, one can see an example of a group of registers. This group contains the registers $2 to $15. They are readable and writable, and can be accessed from any part of the code due to their type (global).

```
[GROUP]
{REGISTERS}
$2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14, $15
{READABLE}
YES
{WRITABLE}
YES
{TYPE}
global
```

Fig. A.9: Example of a group of registers.

There is also a configuration file to make CFT-tool able to understand the disassembled code. This file identifies the equivalent configurations of the assembly code in the disassembly. For example, Fig. A.10 shows the configuration for the label format; Fig. A.11, the instruction format; and Fig. A.12, the equivalent name of each register in the disassembly and the assembly.

```
[LABEL_FORMAT]
a <t>:
```

Fig. A.10: Example of a label format in the disassembly.

```
[INSTRUCTION_FORMAT]
a:  XX XX XX XX     i  o
```

Fig. A.11: Example of the instruction format in the disassembly.

```
[REGISTERS]
zero,$0
at,$1
v0,$2
v1,$3
a0,$4
a1,$5
a2,$6
a3,$7
t0,$8
t1,$9
t2,$10
```

Fig. A.12: Equivalent names of the registers in the disassembly (left) and in the assembly (right).

There are many other configurations, but they are not in the scope of this work. We just wanted to illustrate how CFT-tool works.

A.2 Parameters

It is possible to select and configure the SIHFT techniques that shall be applied to the code with configuration files. In addition, there is also another option, the use of command line parameters when running the CFT-tool. Table A.1 presents some of the parameters that can be passed to the tool.

Table C.1: Some parameters of CFT-tool

parameter	description
assemblyFilename	the filename of the assembly code of the target application
techniques	the SIHFT techniques that shall be applied
targetProcessor	the target processor. It works for the processors already configured
selectedRegisters	the registers that shall be hardened
priorityMode	indicates the criterium to rank the registers
offset	offset in memory between original data and replica
setaHigherPriority	reserves registers to guarantee the implementation of SETA

CFT-tool is a very complex hardening tool that is in constant improvement, with the add of new techniques and features. Its use was vital for the development of this work.

APPENDIX B <DEVICES>

This appendix presents information about the target processors (miniMIPS and ARM Cortex-A9) and talks about the ZedBoard, a low-cost implementation of the Xilinx Zynq®-7000 All Programmable SoC, which has a dual-core ARM Cortex-A9 embedded.

B.1 miniMIPS

miniMIPS is a 32-bit processor core based on MIPS I architecture. It implements a total of 52 instructions, all with 32 bits of length, and it has a pipeline of 5 stages. All miniMIPS instructions take five cycles to be executed, and the peak throughput is one instruction per cycle (HANGOUT, 2009).

Fig. B.1 shows the register set. It is composed of 32 registers of 32 bits. They are general purpose registers and are accessible at any part of the program. Register $0 cannot be written (its value is the constant zero). And register $31 receives by default the return address during subroutine calls.

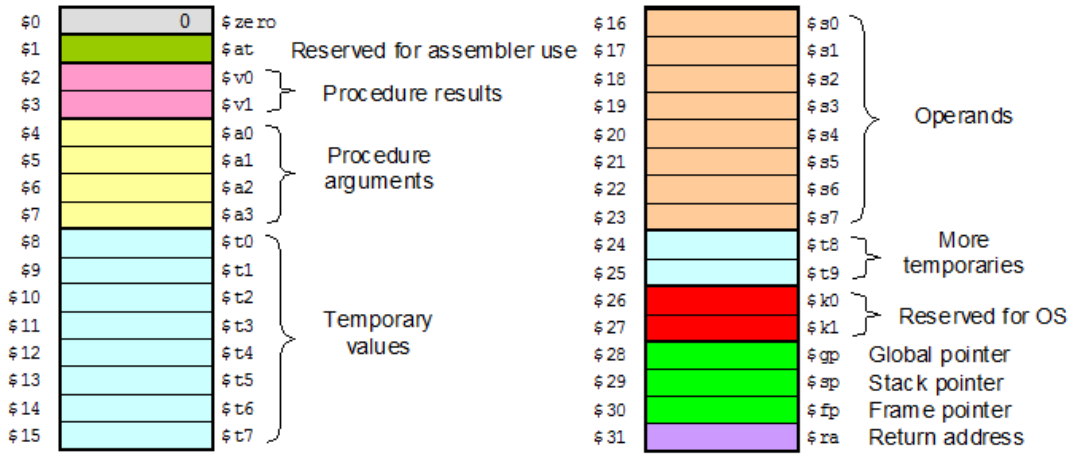


Fig. B.1: miniMIPS register set.

Branch instructions are an important issue for the SIHFT techniques. The way the comparisons and branches are performed may affect the overheads and how checkers are inserted. The miniMIPS processor do the comparison and the branch in the same instruction. Fig. B.2 presents an example of instruction *bne* (branch not equal) that compares two registers (*$2* and *$3*) and takes, or not, the branch to *target* address depending on the result of the comparison.

```
bne $2, $3, target
```

Fig. B.2: Comparison and branch in the miniMIPS processor.

In this work, the miniMIPS was simulated at RTL level using a hardware description (VHDL) of the processor. The target application has to be in the COE format to be executed by the processor. The COE contains the binary code that is loaded into the memory. It is obtained by translating from the disassembly.

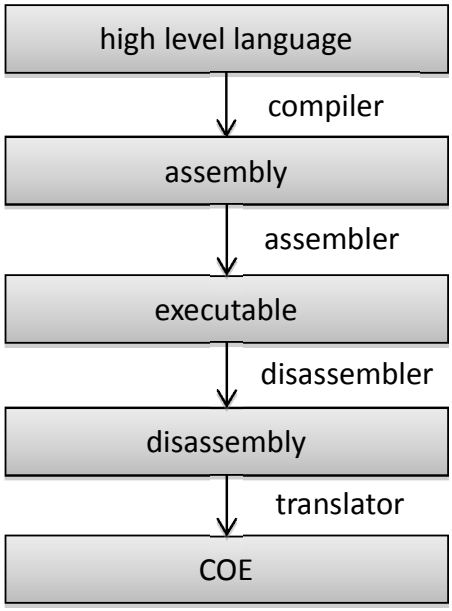


Fig. B.3: Transformations to run an application on the miniMIPS processor

B.2 ARM Cortex-A9

The ARM Cortex-A9 is a 32-bit processor core that implements the ARMv7-A architecture. It is an out-of-order superscalar processor, with 32 kB L1 instruction and data caches, 512 KB L2 cache, and speculating 8-stages pipeline (ARM, 2009). There is also a 256 KB L3 cache, also known OCM (on-chip memory), which is DRAM memory shared among all processors and other devices in the board in which the processor we utilized is embedded.

In the application level, the architecture ARMv7-A has thirteen general-purpose registers, R0 to R12, and three 32-bit registers with special uses, SP (stack pointer), LR (link register), and PC (program counter), that can also be called R13, R14, and R15, respectively. The processor uses SP to point the active stack and LR to hold the return link information of subroutine calls (ARM, 2014). Furthermore, there are 32 64-bit registers for SIMD/floating-point (D0 to D31), which can also be referred as 16 128-bit registers in dual view. Other sets of registers, mainly for control, are also available.

In general, the ARM Cortex-A9 processor uses two instructions to compare and take, or not, a branch. Table B.1 presents the instructions used to perform comparisons. They can compare two registers or one register and one immediate.

Table B.1: Instruction to compare in the ARM Cortex-A9 processor

Instruction	Mnemonic	Notes
Compare Negative	CMN	Set flags. Like *ADD* but with no destination register.
Compare	CMP	Set flags. Like *SUB* but with no destination register.

Fig. B.4. shows an example using the instruction *cmp*. It compares two registers and set internal flags accordingly.

```
cmp r2, r3
```

Fig. B.4: Example of comparison in the ARM Cortex-A9 processor.

The branches are taken or not depending on the flags. Fig. B.5 shows an example of a branch instruction called *bne* (branch not equal). It will take, or not, the branch to *target* depending on the values of the flags, set by a previous comparison.

```
bne target
```

Fig. B.5: Example of branch in the ARM Cortex-A9 processor.

In this work, a hardcore version of the ARM Cortex-A9 processor embedded in a ZedBoard is utilized. The standard binary executable is used. The code in high-level language is compiled and assembled in order to be executed. Information about the ZedBoard is presented as follows.

B.3 ZedBoard

ZedBoard is a low-cost development board for the Xilinx Zynq®-7000 All Programmable SoC. It has a dual-core ARM Cortex-A9 processor and 85,000 Serie-7 Programmable Logic cells (AVNET, 2014). Each core has an individual L1 cache. Caches L2 and L3 are shared between the cores, and L3 cache is also shared with other devices in the board. Fig. B.6 presents the ZedBoard block diagram with its features. They consist of:

- Xilinx® XC7Z020-1CLG484C Zynq-7000 AP SoC
  - Primary configuration = QSPI Flash
  - Auxiliary configuration options
    - Cascaded JTAG
    - SD Card
- Memory
  - 512 MB DDR3 (128M x 32)
  - 256 Mb QSPI Flash
- Interfaces
  - USB-JTAG Programming using Digilent SMT1-equivalent circuit
    - Accesses PL JTAG
    - PS JTAG pins connected through PS Pmod
  - 10/100/1G Ethernet
  - USB OTG 2.0
  - SD Card
  - USB 2.0 FS USB-UART bridge
  - Five Digilent Pmod™ compatible headers (2x6) (1 PS, 4 PL)
  - One LPC FMC
  - One AMS Header
  - Two Reset Buttons (1 PS, 1 PL)
  - Seven Push Buttons (2 PS, 5 PL)
  - Eight dip/slide switches (PL)
  - Nine User LEDs (1 PS, 8 PL)
  - DONE LED (PL)
- On-board Oscillators
  - 33.333 MHz (PS)
  - 100 MHz (PL)
- Display/Audio

  - HDMI Output
  - VGA (12-bit Color)
  - 128x32 OLED Display
  - Audio Line-in, Line-out, headphone, microphone
- Power
  - On/Off Switch
  - 12V @ 5A AC/DC regulator

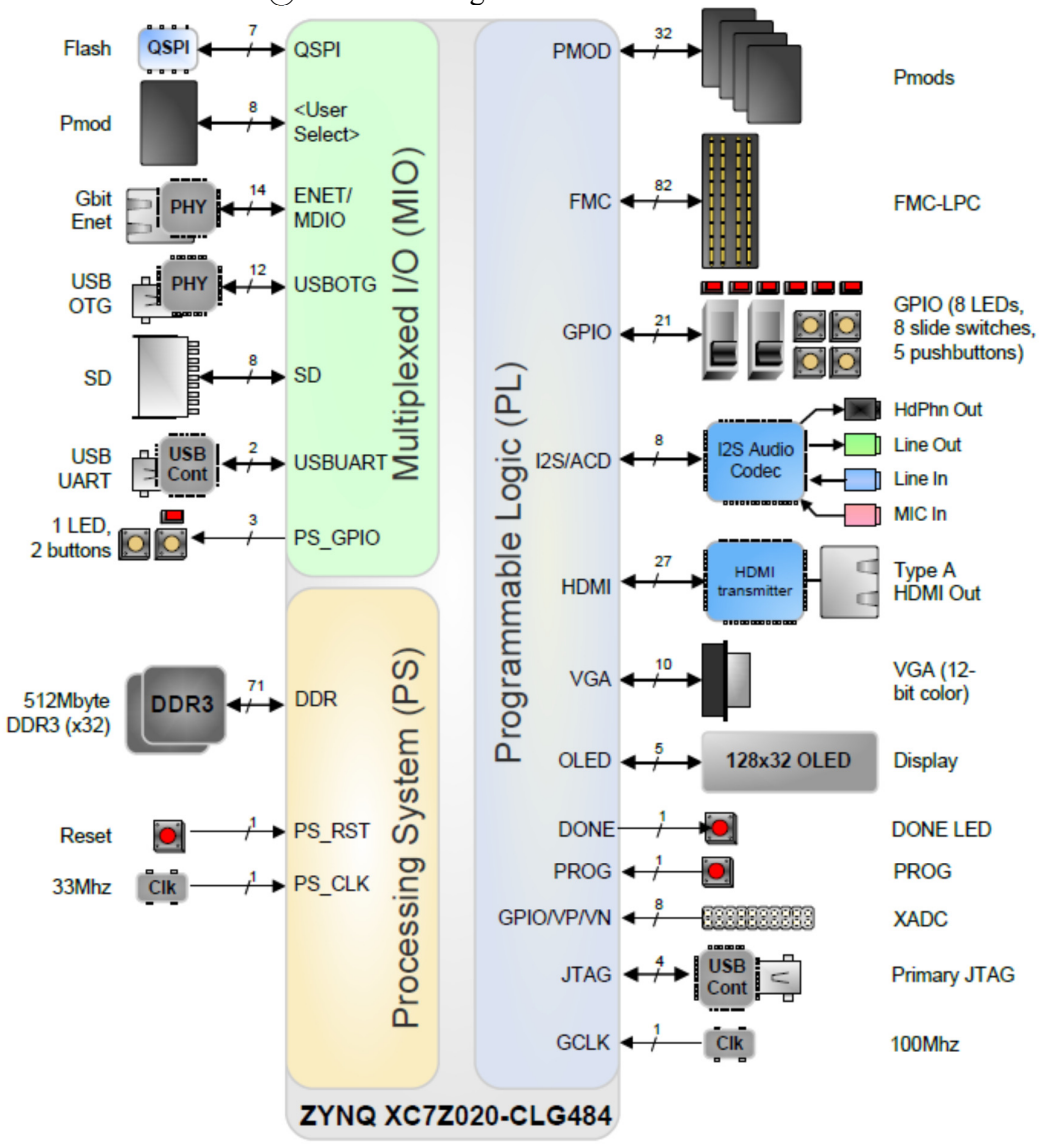


Fig. B.6: ZedBoard block diagram (AVNET, 2015).

The use of ZedBoard in this work regards with the radiation tests because it is possible to program remotely the board to run applications on the embedded ARM Cortex-A9 processor. Thus, the proposed software-based techniques can be tested.

APPENDIX C <BENCHMARKS>

Nine applications were selected as benchmarks in this work. They consist of a bubble sort (BS), the Dijkstra's algorithm (DA), a recursive depth-first search (rDFS), a sequential depth-first search (sDFS), a matrix multiplication (MM), the run-length encoding (RLE), a summation (SUM), the TETRA encryption algorithm (TEA2), and a Tower of Hanoi (TH). Table C.1 shows a summary, for each application, regarding the instructions present in the code. *Branches* are all the branches, jumps, and subroutine calls; *load/stores* are the load and stores; and arithmetics are the remaining instructions, such as *add, sub,* etc.

Table C.1: Summary of instructions for each benchmark

benchmark	instructions	arithmetics	load/stores	branches
BS	144	70 (48.6%)	48 (33.3%)	26 (18.1%)
DA	310	158 (51.0%)	122 (39.4%)	30 (9.7%)
rDFS	38	16 (42.1%)	10 (26.3%)	12 (31.6%)
sDFS	26	10 (38.5%)	6 (23.1%)	10 (38.5%)
MM	171	87 (50.9%)	55 (32.2%)	29 (17.0%)
RLE	459	176 (38.3%)	232 (50.5%)	51 (11.1%)
SUM	24	8 (33.3%)	10 (41.7%)	6 (25.0%)
TEA2	115	38 (33.0%)	69 (60.0%)	8 (7.0%)
TH	75	38 (50.7%)	27 (36.0%)	10 (13.3%)

In Table C.2, one can see a more detailed division, for each benchmark, of the instructions present in the code. In comparison to Table C.1, the *no operations (nops)* have been separated from *arithmetics* and included in a new column. *Loads* and *stores* have been divided. And subroutine calls, unconditional branches *(jumps)*, and conditional branches have been partitioned in the respective following categories: *calls, jumps,* and *branches*.

Table C.3 presents the basic block division of each application. The table includes the number of basic blocks, the average number of instructions per BB, the number of basic blocks of types X and A, and the average number of successors and predecessors per BB. Table C.4 complements Table C.3 with additional information, which includes the number of networks that an application has, the average number of basic blocks in the predecessor network (*predNet*)[15], and the average number of times a BB is executed.

[15] *predNet* is the predecessor network. It contains the predecessor of the BBs of a given network.

Table C.2: Detailed division of instructions for each benchmark

benchmark	nops	arithmetics	loads	stores	branches	jumps	calls
BS	0 (0.0%)	70 (48.6%)	28 (19.4%)	20 (13.9%)	15 (10.4%)	10 (6.9%)	1 (0.7%)
DA	1 (0.3%)	157 (50.6%)	67 (21.6%)	55 (17.7%)	14 (4.5%)	15 (4.8%)	1 (0.3%)
rDFS	3 (7.9%)	13 (34.2%)	6 (15.8%)	4 (10.5%)	3 (7.9%)	6 (15.8%)	3 (7.9%)
sDFS	2 (7.7%)	8 (30.8%)	4 (15.4%)	2 (7.7%)	3 (11.5%)	6 (23.1%)	1 (3.8%)
MM	0 (0.0%)	87 (50.9%)	28 (16.4%)	27 (15.8%)	15 (8.8%)	12 (7.0%)	2 (1.2%)
RLE	0 (0.0%)	176 (38.3%)	143 (31.2%)	89 (19.4%)	26 (5.7%)	24 (5.2%)	1 (0.2%)
SUM	0 (0.0%)	8 (33.3%)	5 (20.8%)	5 (20.8%)	1 (4.2%)	5 (20.8%)	0 (0.0%)
TEA2	0 (0.0%)	38 (33.0%)	43 (37.4%)	26 (22.6%)	1 (0.9%)	6 (5.2%)	1 (0.9%)
TH	5 (6.7%)	33 (44.0%)	14 (18.7%)	13 (17.3%)	1 (1.3%)	4 (5.3%)	5 (6.7%)

Table C.3: Overall information about the basic blocks for each benchmark

benchmark	# BBs	avg. # instructions	type X	type A	avg. # successors	avg. # predecessors
BS	29	4.97	18 (62.1%)	11 (37.9%)	1.48	1.48
DA	39	7.95	33 (84.6%)	6 (15.4%)	1.36	1.36
rDFS	12	3.17	9 (75.0%)	3 (25.0%)	1.92	1.92
sDFS	11	2.36	11 (100.0%)	0 (0.0%)	1.27	1.27
MM	32	5.34	22 (68.8%)	10 (31.3%)	1.47	1.47
RLE	67	6.85	58 (86.6%)	9 (13.4%)	1.39	1.39
SUM	7	3.43	7 (100.0%)	0 (0.0%)	1.00	1.00
TEA2	9	12.78	9 (100.0%)	0 (0.0%)	1.11	1.11
TH	12	6.25	11 (91.7%)	1 (8.3%)	1.25	1.25

Table C.4: Additional information about the basic blocks for each benchmark

benchmark	# networks	avg. # BBs in predNet	avg. # times a BB is executed
BS	18	0.66	8.00
DA	25	0.87	23.69
rDFS	7	0.75	5.92
sDFS	8	0.91	5.36
MM	20	0.72	4.28
RLE	41	0.88	27.01
SUM	6	0.71	286.43
TEA2	8	0.89	7.89
TH	9	0.92	767.67

Not all faults affecting the unhardened application will result in failures. Actually, the masking rate is for the miniMIPS processor considering the fault injection methodology utilized. Table C.5 presents the fault coverage, as well as the execution time and code size, of the unhardened applications. It is clear that the error detection rate of unhardened applications is zero. Thus, the fault coverage is equivalent to the masking rate.

Table C.5: Execution time, code size, and fault coverage of the unhardened applications

benchmark	execution time	code size	fault coverage
BS	195.5	1156	87.36%
DA	872.5	2364	88.80%
rDFS	24.50	1820	84.41%
sDFS	16.25	1764	84.71%
MM	161.0	1308	84.50%
RLE	1316.0	3080	87.73%
SUM	1262.0	536	85.95%
TEA2	154.0	1096	83.10%
TH	4729.0	876	81.24%

C.1 Bubble sort

Bubble sort (BS) is a sorting algorithm that compares the value of adjacent positions and may swap them if necessary. In this algorithm, there are many loops, branches, and registers used to determine the execution flow.

Table C.6 shows a dynamic evaluation of the registers usage for the bubble sort. It illustrates the number of times each register was used either as destination or source register during the program execution.

Table C.6: Dynamic evaluation of the registers usage for the bubble sort

register	# times used as destination	# times used as source	total # times used
0	0	122	122
2	1111	1110	2221
3	354	353	707
4	80	82	162
5	4	4	8
6	3	5	8
7	5	15	20
8	6	10	16
9	5	15	20
10	1	1	2
sp	3	7	10
fp	2	766	768
31	2	2	4
total	**1576**	**2492**	**4068**

Table C.7 shows the effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), the criticality and the rank of criticality of the registers used by the bubble sort. The register $0 was not evaluated using Restrepo-Calle (2015) metric for criticality because it has the constant zero and cannot be modified. Thus, fault injection could not be performed directly in this register. We decided to set the register $0 as the one with the highest criticality by default (when it is utilized) because this register zero showed to increase the fault coverage with very low overheads (CHIELLE, 2013). The bubble sort was compiled with no optimizations. For that reason, most of the calculations are performed using few registers. These registers the most sensitive ones, which is indicate by their high criticality. On the other hand, once they are the most used registers, they cause higher overheads when hardened.

Table C.7: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the bubble sort

register	EL	WCB	FD	criticality	rank
2	4.25E-01	2.40E-02	5.36E-03	1.50E-01	3
3	2.28E-01	0.00E+00	9.03E-03	7.84E-02	5
4	2.37E-01	0.00E+00	9.06E-04	7.87E-02	4
5	9.84E-01	0.00E+00	1.92E-04	3.25E-01	2
6	1.82E-02	6.44E-04	2.74E-06	6.24E-03	7
7	1.33E-02	0.00E+00	3.24E-05	4.40E-03	10
8	1.14E-02	1.50E-03	0.00E+00	4.25E-03	11
9	1.46E-02	0.00E+00	4.02E-04	4.95E-03	9
10	1.70E-02	0.00E+00	3.18E-05	5.61E-03	8
sp	1.50E-03	0.00E+00	1.56E-04	5.47E-04	12
fp	9.94E-01	0.00E+00	2.22E-02	3.35E-01	1
31	1.98E-02	0.00E+00	0.00E+00	6.52E-03	6

Tables C.8 and C.9 presents detailed information about the basic block division of the bubble sort. Table C.8 includes the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the

basic block was executed. Table C.9 shows the BBs in the predecessor network of each network.

Table C.8: Detailed information about the basic blocks of the bubble sort

BB	# instructions	type	successors	predecessors	network	# times executed
0	10	X	14		17	1
1	3	X	2	21	16	1
2	3	X	3, 4	1, 10	15	11
3	1	X	11	2	14	1
4	1	X	5	2	14	10
5	4	X	6, 7	4, 9	13	55
6	1	X	10	5	12	10
7	13	X	8, 9	5	12	45
8	24	X	9	7	11	32
9	4	A	5	7, 8	11	45
10	7	X	2	6	10	10
11	5	X	13	3	9	1
12	1	X			8	0
13	1	X	13	11, 13	7	1
14	4	X	15, 22	0	6	1
15	5	X	16, 17	14	5	1
16	12	A	17, 16	15, 16	4	2
17	3	A	18, 19	15, 16	4	1
18	6	A	19, 18	17, 18	3	2
19	3	A	20, 21	17, 18	3	1
20	6	A	21, 20	19, 20	2	0
21	2	A	1	19, 20, 25, 27	2	1
22	3	X	23, 28	14	5	0
23	4	A	24, 25	22, 28	1	0
24	8	A	25, 24	23, 24	0	0
25	1	A	26, 21	23, 24	0	0
26	6	A	27, 26	25, 26	2	0
27	1	X	21	26	2	0
28	2	X	23	22	1	0

Table C.9: BBs in the predecessor network for each network of the bubble sort

network	BBs in the predNet
0	23
1	22
2	25, 19
3	17
4	15
5	14
6	0
7	11
8	
9	3
10	6
11	7
12	5
13	4, 9
14	2
15	1, 10
16	21
17	

C.2 Dijkstra's algorithm

Dijkstra's algorithm (DA) is an algorithm proposed by Edsger W. Dijkstra (1959) to find the shortest path between nodes in a graph. It selects the unvisited node with the shortest distance to the origin. It is done iteratively, always selecting the current shortest path, until finding the destination, or failing.

Table C.10 illustrates a dynamic evaluation of the registers usage for the Dijkstra's algorithm. It shows the number of times each register was used either as destination or source register during the program execution.

Table C.11 shows the effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), the criticality and the rank of criticality of the registers used by the Dijkstra's algorithm. As the bubble sort, it is compiled with no optimizations, which concentrates the calculations in few registers. One thing that is important to notice is the high criticality of the register $31. It is used just a few times during the program execution. However, the register lifetime is huge because it is written in the beginning of the application and read in the end.

Table C.12 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.13 shows the BBs in the predecessor network of each network.

Table C.10: Dynamic evaluation of the registers usage for the Dijkstra's algorithm

register	# times used as destination	# times used as source	total # times used
0	0	338	338
2	5379	5378	10757
3	1466	1936	3402
4	110	110	220
sp	6	12	18
fp	4	2786	2790
31	2	2	4
total	**6967**	**10562**	**17529**

Table C.11: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the Dijkstra's algorithm

register	EL	WCB	FD	criticality	rank
2	3.24E-01	2.57E-02	3.85E-03	1.17E-01	3
3	3.07E-01	1.01E-02	6.94E-03	1.07E-01	4
4	4.04E-02	0.00E+00	5.06E-04	1.35E-02	5
sp	1.06E-03	0.00E+00	7.94E-05	2.65E-04	6
fp	9.99E-01	0.00E+00	1.46E-02	3.35E-01	1
31	9.99E-01	0.00E+00	0.00E+00	3.30E-01	2

Table C.12: Detailed information about the basic blocks of the Dijkstra's algorithm

BB	# instructions	type	successors	predecessors	network	# times executed
0	18	X	1	35	24	1
1	3	X	2, 3	0, 3	23	101
2	1	X	4	1	22	1
3	31	X	1	1	22	100
4	1	X	5	2	21	1
5	3	X	6, 7	4, 7	20	11
6	1	X	8	5	19	1
7	16	X	5	5	19	10
8	14	X	9, 10	6	18	1
9	3	X	12	8	17	1
10	6	X	11, 12	8	17	0
11	1	X	12	10	16	0
12	29	A	13	9, 10, 11	16	1
13	2	X	14, 15	12, 23	15	11
14	1	X	34	13	14	1
15	10	X	16, 21	13	14	10
16	30	X	17, 18	15	13	10
17	5	X	20	16	12	2
18	6	X	19, 20	16	12	8
19	1	X	20	18	11	0
20	3	A	21	17, 18, 19	11	10
21	1	A	22	15, 20	13	10
22	3	X	23, 24	21, 33	10	110
23	1	X	13	22	9	10
24	15	X	25, 33	22	9	100
25	7	X	26, 28	24	8	100
26	10	X	27, 28	25	7	91
27	1	X	33	26	7	91
28	17	A	29, 30	25, 26	7	9
29	3	X	32	28	6	1
30	6	X	31, 32	28	6	8
31	1	X	32	30	5	0
32	32	A	33	29, 30, 31	5	9
33	4	A	22	24, 27, 32	8	100
34	9	X	36	14	4	1
35	6	X	0		3	1
36	6	X	38	34, 37	2	1
37	2	X	36		1	0
38	1	X	38	36, 38	0	1

Table C.13: BBs in the predecessor network for each network of the Dijkstra's algorithm

network	BBs in the predNet
0	36
1	
2	34, 37
3	
4	14
5	29, 30
6	28
7	25

8	24, 27, 32
9	22
10	21, 33
11	17, 18
12	16
13	15, 20
14	13
15	12, 23
16	9, 10
17	8
18	6
19	5
20	4, 7
21	2
22	1
23	0, 3
24	35

C.3 Recursive depth-first search

Two different versions of a depth-first search were implemented. They both perform the same task, but in a different way, one sequential (sDFS) and another recursive (rDFS). While rDFS uses many recursive subroutine calls, sDFS uses many loops. Thus, it is possible to compare if the use of recursion may affect the reliability.

The rDFS was implemented in assembly with the aim of maximizing performance by distributing the calculation among the registers and avoiding transferring unnecessary data to the memory. Table C.14 illustrates a dynamic evaluation of the registers usage for the recursive depth-first search. It shows the number of times each register was used either as destination or source register during the program execution. The criticalities of the registers used by the rDFS, showed in Table C.15, were more distributed than the applications compiled with no optimizations exactly because the calculations were more distributed. We can see that the intuitive thought that the most used registers are more sensitive is true. On the other hand, it also means that the protection of the most sensitive registers causes higher overheads. Finally, Table C.16 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.17 shows the BBs in the predecessor network of each network.

Table C.14: Dynamic evaluation of the registers usage for the recursive depth-first search

register	# times used as destination	# times used as source	total # times used
2	2	16	18
3	1	27	28
4	28	53	81
5	14	28	42
6	1	2	3
7	28	42	70
8	14	27	41
sp	0	1	1
31	27	27	54
total	**115**	**223**	**338**

Table C.15: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the rDFS

register	EL	WCB	FD	criticality	rank
2	6.99E-01	0.00E+00	4.95E-03	2.32E-01	3
3	7.24E-01	0.00E+00	2.00E-04	2.39E-01	2
4	9.09E-01	0.00E+00	9.62E-03	3.03E-01	1
5	2.90E-01	2.46E-02	4.95E-03	1.05E-01	6
6	2.64E-01	0.00E+00	5.27E-04	8.72E-02	7
7	6.10E-01	0.00E+00	8.68E-03	2.04E-01	5
8	2.21E-01	0.00E+00	2.00E-04	7.31E-02	8
sp	8.79E-03	0.00E+00	5.27E-04	3.07E-03	9
31	6.38E-01	0.00E+00	0.00E+00	2.11E-01	4

Table C.16: Detailed information about the basic blocks of the recursive depth-first search

BB	# instructions	type	successors	predecessors	network	# times executed
0	1	X	1, 2	5, 7, 9	6	14
1	2	X	6, 8, 10	0	5	0
2	4	X	3, 4	0	5	14
3	2	X	6, 8, 10	2	4	1
4	1	X	5, 7	2	4	13
5	5	X	0	4	3	7
6	3	A	6, 8, 10	1, 3, 6, 8	2	7
7	5	X	0	4	3	6
8	3	A	6, 8, 10	1, 3, 6, 8	2	6
9	6	X	0		1	1
10	5	A	11	1, 3, 6, 8	2	1
11	1	X	11	10, 11	0	1

Table C.17: BBs in the predecessor network for each network of the recursive depth-first search

network	BBs in the predNet
0	10
1	
2	1, 3
3	4
4	2
5	0
6	5, 7, 9

C.4 Sequential depth-first search

The sequential depth-first search (sDFS) is the nonrecursive version of the depth-first search. Table C.18 illustrates a dynamic evaluation of the registers usage for the sequential depth-first search. It shows the number of times each register was used either as destination or source register during the program execution. Table C.19 shows the criticality of the registers used by the sDFS. This application was implemented in assembly with the aim of maximizing performance by distributing calculations among the registers (as the rDFS). Therefore, we can see a more distributed criticality among the

registers. It is also possible to notice that the registers $31, although little used, has high criticality due to its high effective lifetime. Table C.20 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.21 shows the BBs in the predecessor network of each network.

Table C.18: Dynamic evaluation of the registers usage for the sequential depth-first search

register	# times used as destination	# times used as source	total # times used
2	2	16	18
3	1	27	28
4	43	71	114
5	14	27	41
6	1	2	3
31	1	1	2
total	**62**	**144**	**206**

Table C.19: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the sDFS

register	EL	WCB	FD	criticality	rank
2	2.35E-01	0.00E+00	1.05E-02	3.10E-01	3
3	3.53E-01	0.00E+00	4.65E-04	3.18E-01	1
4	4.71E-01	2.94E-02	4.42E-03	2.80E-01	4
5	1.47E-01	0.00E+00	4.65E-04	1.12E-01	5
6	2.35E-01	0.00E+00	7.47E-04	6.44E-03	6
31	6.18E-01	0.00E+00	0.00E+00	3.14E-01	2

Table C.20: Detailed information about the basic blocks of the sequential depth-first search

BB	# instructions	type	successors	predecessors	network	# times executed
0	1	X	1	8	7	1
1	1	X	2, 7	0, 5, 6	6	14
2	5	X	3, 4	1	5	14
3	2	X	9	2	4	1
4	1	X	5, 6	2	4	13
5	2	X	1	4	3	7
6	2	X	1	4	3	6
7	2	X	9	1	5	0
8	4	X	0		2	1
9	5	X	10	3, 7	1	1
10	1	X	10	9, 10	0	1

Table C.21: BBs in the predecessor network for each network of the sequential depth-first search

network	BBs in the predNet
0	9
1	3, 7
2	
3	4
4	2
5	1
6	0, 5, 6
7	8

C.5 Matrix multiplication

The matrix multiplication (MM) has a large amount of data processing within few loops. It is ideal to verify the coverage of data-flow techniques since there are few branches in the code. The version we used was compiled with no optimizations.

Table C.22 illustrates a dynamic evaluation of the registers usage for the matrix multiplication. It shows the number of times each register was used either as destination or source register during the program execution. Table C.23 shows the criticality of the registers used by MM. As stated in section C.1, register $0 set as the most critical by default. Table C.24 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.25 shows the BBs in the predecessor network of each network.

Table C.22: Dynamic evaluation of the registers usage for the matrix multiplication

register	# times used as destination	# times used as source	total # times used
0	0	74	74
2	925	923	1848
3	386	492	878
4	33	37	70
5	35	35	70
6	33	37	70
7	8	26	34
8	10	16	26
9	8	26	34
10	2	2	4
sp	3	7	10
fp	2	480	482
31	3	3	6
total	**1448**	**2158**	**3606**

Table C.23: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the MM

register	EL	WCB	FD	criticality	Rank
2	3.24E-01	1.41E-02	6.37E-03	1.14E-01	5
3	3.90E-01	0.00E+00	1.32E-02	1.33E-01	3
4	2.07E-01	0.00E+00	1.10E-03	6.85E-02	6
5	5.59E-01	0.00E+00	1.80E-04	1.85E-01	2
6	3.92E-01	1.05E-03	8.01E-04	1.30E-01	4
7	2.72E-02	0.00E+00	6.73E-05	8.99E-03	10
8	2.25E-02	3.14E-03	0.00E+00	8.45E-03	11
9	3.03E-02	0.00E+00	6.70E-04	1.02E-02	9
10	3.56E-02	0.00E+00	5.84E-05	1.18E-02	8
sp	1.83E-03	0.00E+00	1.86E-04	6.65E-04	12
fp	9.95E-01	0.00E+00	1.46E-02	3.33E-01	1
31	4.08E-02	0.00E+00	0.00E+00	1.35E-02	7

Table C.24: Detailed information about the basic blocks of the matrix multiplication

BB	# instructions	type	successors	predecessors	network	# times executed
0	10	X	17		19	1
1	6	X	17	24	18	1
2	10	X	3	24	18	1
3	3	X	4, 5	2, 13	17	4
4	1	X	14	3	16	1
5	1	X	6	3	16	3
6	3	X	7, 8	5, 12	15	12
7	1	X	13	6	14	3
8	1	X	9	6	14	9
9	3	X	10, 11	8, 11	13	36
10	1	X	12	9	12	9
11	50	X	9	9	12	27
12	4	X	6	10	11	9
13	4	X	3	7	10	3
14	5	X	16	4	9	1
15	1	X			8	0
16	1	X	16	14, 16	7	1
17	4	X	18, 25	0, 1	6	2
18	5	X	19, 20	17	5	2
19	12	A	20, 19	18, 19	4	4
20	3	A	21, 22	18, 19	4	2
21	6	A	22, 21	20, 21	3	2
22	3	A	23, 24	20, 21	3	2
23	6	A	24, 23	22, 23	2	0
24	2	A	1, 2	22, 23, 28, 30	2	2
25	3	X	26, 31	17	5	0
26	4	A	27, 28	25, 31	1	0
27	8	A	28, 27	26, 27	0	0
28	1	A	29, 24	26, 27	0	0
29	6	A	30, 29	28, 29	2	0
30	1	X	24	29	2	0
31	2	X	26	25	1	0

Table C.25: BBs in the predecessor network for each network of the MM

network	BBs in the predNet
0	26
1	25
2	28, 22
3	20
4	18
5	17
6	0, 1
7	14
8	
9	4
10	7
11	10
12	9
13	8, 11
14	6
15	5, 12
16	3
17	2, 13
18	24
19	

C.6 Run length encoding

Run-length encoding (RLE) is an algorithm for lossless data compression in which a homogeneous sequence of data are stored as a single datum and a count (CHEN, 2010). RLE is useful for simple graphics images, i.e., images of large scale with identically valued pixels (WANG, 1997).

Table C.26 illustrates a dynamic evaluation of the registers usage for the run length encoding. It shows the number of times each register was used either as destination or source register during the program execution. Table C.27 shows the criticality of the registers used by the RLE. This application was compiled with no optimizations. For that reason, we can see few registers with high criticality (similar as showed to the Dijkstra's algorithm). Table C.28 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.29 shows the BBs in the predecessor network of each network.

Table C.26: Dynamic evaluation of the registers usage for the run length encoding

register	# times used as destination	# times used as source	total # times used
0	0	1251	1251
2	7495	7495	14990
3	1142	1241	2383
4	206	209	415
5	60	119	179
sp	6	12	18
fp	4	4748	4752
31	2	2	4
total	**8915**	**15077**	**23992**

Table C.27: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the RLE

register	EL	WCB	FD	criticality	rank
2	3.71E-01	3.15E-02	9.20E-04	1.33E-01	3
3	1.91E-01	0.00E+00	2.88E-03	6.40E-02	4
4	7.29E-02	0.00E+00	2.63E-04	2.41E-02	5
5	3.80E-02	0.00E+00	1.23E-04	1.26E-02	6
sp	4.47E-04	0.00E+00	5.27E-05	1.65E-04	7
fp	9.99E-01	0.00E+00	2.01E-02	3.36E-01	1
31	9.99E-01	0.00E+00	0.00E+00	3.30E-01	2

Table C.28: Detailed information about the basic blocks of the run length encoding

BB	# instructions	type	successors	predecessors	network	# times executed
0	9	X	1, 61	63	40	1
1	1	X	2	0	39	1
2	3	X	3, 4	1, 4	38	257
3	1	X	5	2	37	1
4	9	X	2	2	37	256
5	1	X	6	3	36	1
6	4	X	7, 8	5, 8	35	61
7	1	X	9	6	34	1
8	23	X	6	6	34	60
9	3	X	10	7	33	1
10	3	X	11, 12	9, 14	32	256
11	1	X	15	10	31	1
12	12	X	13, 14	10	31	255
13	2	X	14	12	30	2
14	4	A	10	12, 13	30	255
15	15	X	16, 56	11	29	1
16	12	X	17	15	28	1
17	3	A	18, 48	16, 54, 55	28	31
18	4	A	19, 23	17, 22	27	36
19	3	X	20, 23	18	26	35
20	4	X	21, 22	19	26	28
21	1	X	23	20	25	0
22	14	X	18	20	25	28
23	3	A	24, 37	18, 19, 21	26	8
24	5	X	25, 31	23	24	1
25	3	X	26, 27	24	23	0
26	20	X	34	25	22	0
27	1	X	28	25	22	0
28	4	X	29, 30	27, 30	21	0
29	1	X	34	28	20	0
30	13	X	28	28	20	0
31	15	X	32, 33	24	23	1
32	11	X	33	31	19	0
33	19	A	34	31, 32	19	1
34	4	X	35, 36	26, 29, 33	18	1
35	13	X	52	34	17	0
36	2	X	52	34	17	1
37	6	X	38, 44	23	24	7
38	3	X	39, 40	37	16	5
39	20	X	47	38	15	0
40	1	X	41	38	15	5

41	4	X	42, 43	40, 43	14	16
42	1	X	47	41	13	5
43	13	X	41	41	13	11
44	15	X	45, 46	37	16	2
45	11	X	46	44	12	0
46	19	A	47	44, 45	12	2
47	5	X	52	39, 42, 46	11	7
48	3	X	49, 50	17	27	23
49	18	X	51	48	10	0
50	9	X	51	48	10	23
51	4	X	52	49, 50	9	23
52	4	X	53, 54	35, 36, 47, 51	8	31
53	12	X	54	52	7	30
54	4	A	55, 17	52, 53	7	31
55	3	X	56, 17	54	28	1
56	3	A	57, 60	15, 55	28	1
57	3	X	58, 59	56	6	0
58	18	X	60	57	5	0
59	9	X	60	57	5	0
60	3	A	62	56, 58, 59	6	1
61	1	X	62	0	39	0
62	5	X	64	60, 61	4	1
63	7	X	0		3	1
64	6	X	66	62, 65	2	1
65	1	X	64		1	0
66	1	X	66	64, 66	0	1

Table C.29: BBs in the predecessor network for each network of the run length encoding

network	BBs in the predNet
0	64
1	
2	62, 65
3	
4	60, 61
5	57
6	56, 58, 59
7	52
8	35, 36, 47, 51
9	49, 50
10	48
11	39, 42, 46
12	44
13	41
14	40, 43
15	38
16	37
17	34
18	26, 29, 33
19	31
20	28
21	27, 30
22	25
23	24
24	23
25	20

26	18, 21
27	17, 22
28	54, 15
29	11
30	12
31	10
32	9, 14
33	7
34	6
35	5, 8
36	3
37	2
38	1, 4
39	0
40	63

C.7 Summation

Summation (SUM) is the definite integral of a continuous function (KOUBA, 1999). In this work, it was implemented the summation given by the equation C.1. It is the sum of a sequence of numeric values. The version we implemented was compiled with no optimizations.

$$S = \sum_{i=1}^{n} i \qquad \text{(Eq. C.1)}$$

Table C.30 illustrates a dynamic evaluation of the registers usage for the summation. It shows the number of times each register was used either as destination or source register during the program execution. Table C.31 shows the criticality of the registers used by the summation. Table C.32 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.33 shows the BBs in the predecessor network of each network.

Table C.30: Dynamic evaluation of the registers usage for the summation

register	# times used as destination	# times used as source	total # times used
0	0	1004	1004
2	6003	6002	12005
3	1000	1000	2000
sp	3	5	8
fp	2	6005	6007
total	**7008**	**14016**	**21024**

Table C.31: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the summation

register	EL	WCB	FD	criticality	rank
2	3.33E-01	3.33E-02	0.00E+00	1.21E-01	2
3	6.66E-02	0.00E+00	4.03E-03	2.33E-02	3
sp	1.33E-04	0.00E+00	2.20E-05	5.12E-05	4
fp	1.00E+00	0.00E+00	1.28E-02	3.34E-01	1

Table C.32: Detailed information about the basic blocks of the summation

BB	# instructions	type	successors	predecessors	network	# times executed
0	5	X	1		5	1
1	3	X	2, 3	0, 3	4	1001
2	1	X	4	1	3	1
3	8	X	1	1	3	1000
4	5	X	6	2	2	1
5	1	X			1	0
6	1	X	6	4, 6	0	1

Table C.33: BBs in the predecessor network for each network of the summation

network	BBs in the predNet
0	4
1	
2	2
3	1
4	0, 3
5	

C.8 TETRA encryption algorithm

The Terrestrial Trunked Radio (TETRA) is a European standard designed for emergency, transport and military services (WURSTER, 2013). The TETRA Encryption Algorithm is an algorithm used to provide confidentiality to the TETRA air interface. This encryption algorithm protects against eavesdropping as well as protection of signaling (PARKINSON, 2001). In this work, the second version of the TETRA Encryption Algorithm (TEA2) was implemented. It was compiled with no optimizations.

Table C.34 illustrates a dynamic evaluation of the registers usage for the TETRA encryption algorithm. It shows the number of times each register was used either as destination or source register during the program execution. Table C.35 presents the criticality of the registers used by the TEA2. Table C.36 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.37 shows the BBs in the predecessor network of each network.

Table C.34: Dynamic evaluation of the registers usage for the TETRA encryption algorithm

register	# times used as destination	# times used as source	total # times used
0	0	36	36
2	799	798	1597
3	290	290	580
4	129	129	258
5	1	1	2
sp	6	12	18
fp	4	738	742
31	2	2	4
total	**1231**	**2006**	**3237**

Table C.35: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the TEA2

register	EL	WCB	FD	criticality	rank
2	3.55E-01	9.02E-03	6.26E-03	1.22E-01	4
3	3.69E-01	0.00E+00	9.29E-03	1.25E-01	3
4	3.35E-01	0.00E+00	0.00E+00	1.11E-01	5
5	1.91E-03	0.00E+00	0.00E+00	6.31E-04	7
sp	4.10E-03	0.00E+00	4.47E-04	1.50E-03	6
fp	9.93E-01	0.00E+00	2.15E-02	3.35E-01	1
31	9.82E-01	0.00E+00	0.00E+00	3.24E-01	2

Table C.36: Detailed information about the basic blocks of the TETRA encryption algorithm

BB	# instructions	type	successors	predecessors	network	# times executed
0	32	X	1	5	7	1
1	3	X	2, 3	0, 3	6	33
2	1	X	4	1	5	1
3	40	X	1	1	5	32
4	11	X	6	2	4	1
5	20	X	0		3	1
6	6	X	8	4, 7	2	1
7	1	X	6		1	0
8	1	X	8	6, 8	0	1

Table C.37: BBs in the predecessor network for each network of the TETRA encryption algorithm

network	BBs in the predNet
0	6
1	
2	4, 7
3	
4	2
5	1
6	0, 3
7	5

C.9 Tower of Hanoi

Tower of Hanoi (TH) is a mathematical puzzle. It consists of disks piled up in ascending order of size from top to down. The aim is to move from one stack to another using an auxiliary stack. It has been done respecting the following statements:

- Only one disk can be moved each time. It has to be on top of a stack and goes to the top of another stack;
- A larger disk cannot be on top of a smaller one;

The TH was implemented in assembly to maximize the performance and avoid that intermediary calculations are stored in the memory. Table C.38 illustrates a dynamic evaluation of the registers usage for the Tower of Hanoi. It shows the number of times each register was used either as destination or source register during the program execution. Table C.39 shows the criticality of the registers used by the Tower of Hanoi. One can notice that since the work was better distributed among the registers, the criticality also was. Register $7 was the most used one. However, the metric says it is one of the least critical. It happens because register $7 has many living intervals with very short lifetimes, which makes its data to be often overwritten. Thus, an error in this register will quickly disappear.

Table C.40 presents the number of instructions per BB, type, successors, predecessors, the network to which the BB belongs, and the number of times that the basic block was executed. And Table C.41 shows the BBs in the predecessor network of each network.

Table C.38: Dynamic evaluation of the registers usage for the Tower of Hanoi

register	# times used as destination	# times used as source	total # times used
2	4093	16369	20462
3	3070	4095	7165
4	3070	4094	7164
5	4093	4094	8187
6	2047	4094	6141
7	9212	11259	20471
8	1023	1023	2046
9	2050	7165	9215
10	2047	4092	6139
sp	0	1	1
31	5116	5116	10232
total	**35821**	**61402**	**97223**

Table C.39: Effective lifetime (EL), weight in conditional branches (WCB), functional dependencies (FD), and criticality of the registers used by the Tower of Hanoi

register	EL	WCB	FD	criticality	rank
2	9.45E-01	0.00E+00	1.25E-02	3.16E-01	3
3	5.45E-01	0.00E+00	1.29E-03	1.80E-01	6
4	6.27E-01	0.00E+00	2.05E-03	2.08E-01	5
5	6.36E-01	0.00E+00	3.52E-03	2.11E-01	4
6	9.73E-01	2.73E-02	0.00E+00	3.30E-01	1
7	1.73E-01	0.00E+00	4.56E-03	5.85E-02	10
8	2.45E-01	0.00E+00	0.00E+00	8.10E-02	9
9	2.55E-01	0.00E+00	3.11E-03	8.50E-02	8
10	9.72E-01	0.00E+00	9.54E-04	3.21E-01	2
sp	1.60E-04	0.00E+00	2.77E-06	5.37E-05	11
31	3.18E-01	0.00E+00	0.00E+00	1.05E-01	7

Table C.40: Detailed information about the basic blocks of the Tower of Hanoi

BB	# instructions	type	successors	predecessors	network	# times executed
0	15	X	5		8	1
1	6	X	2	11	7	1
2	1	X	2	1, 2	6	1
3	6	X	9	8	5	1023
4	7	X	8	7	4	1023
5	1	X	6, 11	0, 6, 9	3	2047
6	12	X	5	5	2	1023
7	7	X	4	11	7	1023
8	3	X	3	4	1	1023
9	9	X	5	3	0	1023
10	7	X	11	11	7	1023
11	1	A	1, 7, 10	5, 10	2	1

Table C.41: BBs in the predecessor network for each network of the Tower of Hanoi

network	BBs in the predNet
0	3
1	4
2	5, 10
3	0, 6, 9
4	7
5	8
6	1
7	11
8	