Compiler Assisted Reliability Optimizations

Nazarian, Ghazaleh

DOI
10.4233/uuid:01a602f7-59af-4ee5-a54e-40c536216f58

Publication date
2019

Document Version
Final published version

Citation (APA)
Nazarian, G. (2019). *Compiler Assisted Reliability Optimizations*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:01a602f7-59af-4ee5-a54e-40c536216f58

Ghazaleh Nazarian

Compiler Assisted Reliability Optimizations

Compiler Assisted Reliability Optimizations

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, prof.dr.ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates
to be defended publicly on

Friday 15 February 2019 at 10:00 o'clock

by

Ghazaleh NAZARIAN
Master of Science in Computer Engineering
Delft University of Technology
born in Tehran, Iran

This dissertation has been approved by the promotors:
Prof.dr.ir. G.N. Gaydadjiev
Prof.dr.ir. H.J. Sips

Composition of the doctoral committee:

Rector Magnificus	Delft University of Technology, chairperson
Prof.dr.ir. G.N. Gaydadjiev	Imperial College London, promotor
Prof.dr.ir. H.J. Sips	Delft University of Technology, promotor

Independent members:

Prof.dr. P.J. French	Delft University of Technology
Prof.dr.ir. W.A. Serdijn	Delft University of Technology
Prof.dr. L. Carro	Universidade Federal do Rio Grande do Sul
Prof.dr.ir. J.H.M. Frijns	Leiden University Medical Center
Dr. A. Shahbahrami	University of Guilan

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Ghazaleh Nazarian

Compiler Assisted Reliability Optimizations

Delft:
Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Thesis Delft University of Technology. – With ref. –

Met samenvatting in het Nederlands.

ISBN 978-94-6384-005-7

Subject headings: Reliability, Compiler optimizations, Control flow error detection and recovery.

Printed in EU

To Iradj for the beginning and to Michele for the end

Compiler Assisted Reliability Optimizations

Ghazaleh Nazarian

Abstract

Microprocessors are used in an expanding range of applications from small embedded system devices to supercomputers and mainframes. Moreover, embedded microprocessor based systems became essential in modern societies. Depending on the application domain, embedded systems have to satisfy different constraints. The major challenges today are cost, performance, energy-consumption, reliability, real-time (reactive-operation) and silicon area. In traditional computer systems some of these constraints can be less crucial than others, while performance, area and power-consumption will always remain valid constraints for embedded systems. However, in modern systems reliability has emerged as a new, highly important requirement. Among all above factors performance, power, reactive-operation and reliability can be addressed by software-only techniques that do not require any hardware modifications or additions. Such optimization techniques, however, may impact the performance and power characteristics of the system. The main goal of this work is to find novel software based reliability techniques with affordable power and performance overheads. For this reason the reliability optimization methods are studied in detail and a diligent categorization of existing software techniques is proposed. The strong and the weak points of each category are carefully studied. Using the information obtained from our categorization, two novel optimization techniques for fault detection and one new optimization technique for fault recovery are proposed. Our optimization techniques minimize the required code instrumentation points while guaranteeing equivalent reliability as compared to state of the art approaches. Moreover, a generic methodology is proposed to help with the process of identifying the minimum set of code instrumentation points. For the evaluation we select a challenging baseline that consists of the best known techniques for fault detection and fault recovery found in the public literature. The experimental results on a set of biomedical benchmarks show that using the proposed design methodology and fault detection and recovery methods, the performance and power overheads are significantly reduced while the fault coverage remains in line with previously proposed and widely used methods.

Acknowledgments

Coming from a country like Iran, where relationships between students and professor can be quite formal, the first lesson I had to learn from Georgi was to call him “Georgi”, and not “Professor Gaydadjiev”. Many more lessons had to come until I could finally achieve this goal and write these lines. I am sincerely grateful for the guidance of Georgi during all these years. Even when he was not present at Delft university, his support felt like if he was still in the office next door. Looking back, I feel amazed by how could always rely on him while at the same time he would teach me the importance of carrying on my work independently and self-sufficiently. A special thanks goes to Henk Sips for his valuable contributions to the review of this thesis and his support to finalize the thesis.

I thank all the committee members who honored me by accepting this invitation. The contributions of Asad Shahbahrami to the draft of this thesis added a significant value to this work and I am very grateful for them. A very special thanks goes to Luigi Carro, who literally came from the other side of the world to attend my defense. I am also extremely grateful to him for making my stay as a Guest Researcher at the University of Porto Alegre possible; it was a fruitful period for my research.

I am also thankful to Wouter Serdijn, Paddy French, Johan Frijns and Christos Strydis for making the whole Smart Cochlear Implants project (supported by STW funding) possible. When I started this project, I was lucky not to be completely on my own: the friendly company of Nishant Lawand and Wannaya Ngamkham made it so much easier to get used to the Ph.D. student life.

The experience in Brazil was very important for my work; I need to thank Diego Rodrigues, Ronaldo Ferreira and all the other members of the Computer Department for all the support I received there. Prof. Wong also gave an important contribution to make my visit to Brazil happen, I am sincerely grateful about that.

For this thesis work I received a great deal of support from people at ACE. In particular, I must thank Bryan Olivier, for his outstanding technical support

using CoSy throughout my thesis work. As a compiler engineer, I could not have found a better mentor than he was for me. I am also glad to Hans van Someren, Joeri van Ruth and all the other compiler engineers for the professional and personal support I received during my staying at ACE.

I am also grateful to Jaap de Vos at Brightsight who allowed me to dedicate time to complete my thesis work; without his understanding, this achievement could not have been possible.

The direct and indirect support I received from everybody at CE was extremely important to me. I need to thank Koen Bertels who helped me in the difficult period in between the end of my research period and the beginning of my professional career. The staff was always very supportive, special thanks to Lidwina Tromp, Erik de Vries and Shemara van der Zwet who helped me countless times solving the most diverse problems for me. I am grateful to Carlo Galuzzi for showing interest to be part of the committee, although he eventually could not attend; I wish his family all the best for their new life adventure. Bogdan Spinean, Catalin Ciobanu and Chunyang Gou have been good office mates; life in office was much more pleasant whenever they were around. I also need to thank Rezvan Nane and Rob Seepers; working with them has been a valuable experience.

Moving abroad from my original home town, my friends and colleagues helped me to have Delft as my second home town. I am thankful to Elham Pahlavan, Sanaz Saieed, Sadegh Akbarnejad, Shiva Shayegan, Nikoo Delgoshaiee, Maryam Razavian, Arash Ostadzadeh, Sebastian Isaza, Behnaz Pourebrahimi, Nasim Rezaiee, Valeria Napoli, Elina Iervolino and all my expat friends. A very special thanks to Ostad Wijbrand Luth who helped me with the translation of the Samenvatting.

I thank Michele Squillante, the love of my life for being next to me in the hardest times, bringing me the joyous and beautiful days of life, and also for his sincere assistance and contribution on completing of this work, especially for the design of this dissertation cover. My adorable daughter, Kimia, I would like to thank her for her presence, sharing her time and letting her mom finalize this dissertation. I truly thank my brother, Mohammad Nazarian, and his family, Raheleh Vahidi, Ali and Arta, who have supported me and always been my companion in ups and downs through life. I shall express my gratitude to my uncle, Zain Navabi, and his wife, Irma Alvarado, who motivated me during my bachelor studies and helped me to pursue the master degree in Delft. In loving memory of my mother, Fami Navabi, who would have been more than proud, if she was with us now, finally, I express my greatest thank to my father,

Iradj Nazarian, who has devoted his life to his family and is the main reason I started this journey. I thank him for his presence, for his support and love. He has always inspired me and I am most grateful to have him attending my defense.

Ghazaleh Nazarian Delft, The Netherlands, February 2019

Table of contents

List of Tables

List of Figures

List of Acronyms and Symbols

ACCE Automatic Correction of Control-flow Erors

ACFC Assertions for Control Flow Checking

ACS Abstract Control Signatures

ARM Advanced RISC Machine

BSSC Block Signature Self-Checking

CCA Control-flow Checking using Assertions

CDG Control Dependency Graph

CEDA Control-flow Error Detection through Assertions

CEF Correction Efficiency Factor

CF Control Flow

CFCSS Control Flow Checking by Software Signatures

CFE Control Flow Error

CFG Control Flow Graph

CSC Code Specific Checkpoints

CSUM Checksum

DDG Data Dependency Graph

DE Data Error

DEF Detection Efficiency Factor

ECC Error-Correcting Code

ECCA Enhanced Control-flow Checking using Assertions

ECI Error Capturing Instructions

EDDI	Error Detection by Duplicated Instructions
IR	Intermediate Representation
ISA	Instruction Set Architecture
LFSR	Linear-Feedback Shift Register
LIR	Lower Intermediate Representation
MILP	Mixed Integer Linear Programming
OS	Operating System
PMP	Power Management Point
RISC	Reduced Instruction Set Computer
RS	Runtime Signature
SBL	Susceptible Basic block List
SGF	Signature Generating Functions
SM	Signature Monitoring
SMF	Signature Monitoring Functions
SWIFT	Software implemented fault tolerance
VVP	Variable Voltage Processors
WCET	Worst Case Execution Time
YACCA	Yet Another Control flow Checking Approach

1 Introduction

Microprocessors are used in an expanding range of applications from small embedded systems to supercomputers and mainframes. Embedded systems are essential for modern societies. We can see these systems helping with various aspects of our everyday life from transportation, communication, healthcare to entertainment. Depending on the application domain, embedded systems have quite different requirements. Such systems were always constrained by cost, performance, area, real-time (reactive operation) and energy-consumption. However the behavior of the hardware and software components is continuously changing. Along with technology down-scaling and the reduction of the operating voltages, the probability that phenomena such as radiation or crosstalk impact the state of a transistor and cause a transient fault greatly increases. Transient hardware faults triggered by such events account as one of the major reasons for malfunctioning in today's embedded systems [57]. Therefore, reliability and security are becoming a major concern not only for safety-critical applications but also for mainstream computing systems such as laptops, smart-phones and portable media players, to name a few. Among the aforementioned, performance, power, reliability and reactive-operation are the factors that can be efficiently addressed by software techniques. At the software level the system can be configured to trade-off between different factors based on the specific application requirements. There are many examples of systems requiring high performance, low power and rock-solid reliability in safety critical domains such as health-care, automotive and aviation.

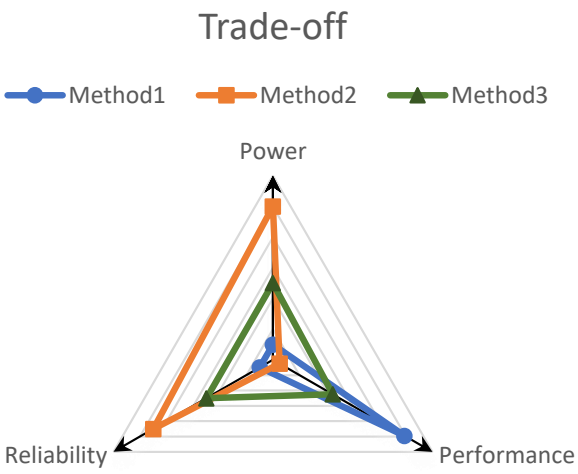


Figure 1.1: Trade offs between power, performance and reliability

End-of-production testing is used to detect permanent faults, while run-time fault-detection and fault-recovery methods are required to cope with transient faults. Conventional software methods aiming at transient fault detection and recovery rely on instrumenting the code without paying much attention to the imposed overheads. Such methods cannot be used efficiently in today's embedded systems which require reliability, low power consumption and high performance at the same time. The chart in Figure 1.1 visualizes three reliability optimizing software methods in terms of power consumption, performance and reliability. The three axes represent the three important metrics: Reliability, Power and Performance. Method 1, depicted in blue line with circle shapes, adds few assertions to the code, therefore can provide low reliability and has low power consumption overhead while the overall system performance remains high. Method 2, illustrated in orange line with rectangle shapes, adds many assertions and additional complex computations to the code, therefore it can deliver high reliability, however, at the expense of significant power consumption. Having many assertions leads to significant overheads and hence low performance of the system. Method 3, shown in green with triangle shapes of the connecting points, adds fewer assertions compared to method 2, therefore it can provide lower reliability and has lower power consumption. Since the number of assertions is lower, overheads are reduced and the system using this method has higher performance as compared to method 2. As can be seen from the above discussion, selecting the best method depends on multiple application specific requirements and is non trivial.

1.1 Problem overview and research questions

As stated earlier, one of the major threats for processor's reliability are transient faults. Existing methods for detection and recovery of transient faults exploit some form of redundancy as hardware extensions, parallel threads or some additional instructions in the executing program able to detect the faults. Conventional remedy for recovering from a detected fault is to use checkpoints and roll back the execution to a pre-stored point which is known to be before the location when the fault has occurred. Several hardware redundancy methods propose to use a watchdog processor, to compare the watchdog results with the main processor results [26] [8], others propose to replicate only parts of the processor such as duplication of the datapath components [14]. These methods have the main drawback of being costly and not applicable to many systems built with off-the-shelf-processors. Methods using redundant threads [38] [40] for reliability optimization also need hardware resources which may not be available in some small processors used in current systems.

Depending on application criticality and requirements, different reliability optimization schemes that are proposed both in hardware and software may be used, e.g, either a scheme in hardware with special circuit checkers or a method in software. For example, safety-critical applications on servers use specialized protecting hardware such as replicated modules [14] or circuit checkers [8] against transient faults. Such computers are, however, typically not limited to tight power consumption and area budgets. Embedded applications using processors with limited power and area budgets can not afford extended hardware techniques for their reliability protection. Hardware solutions such as dual-modular (or triple-modular) redundancy used for example in IBM Z-series, as well as other vendors, are not applicable for many low-cost embedded applications often using off-the-shelf processors. Moreover, traditional hardware techniques for reliability optimization, performance improvements and power reduction may conflict with each other; one factor may have negative impact on the remaining two. A simple example can be observed at the transistor level. With advances in semiconductor technology aiming to increase the performance of the systems, transistors are getting faster and smaller. With increase in frequency and transistor density on the chips, more power is consumed leading to an undesirable increase in power dissipation. In order to compensate for this, the on-chip operating voltage is decreased; yet, voltage reduction makes the device more susceptible to transient faults.

Alternative techniques for run-time fault detection without special hardware are compile-time optimizations. In many systems based on off-the-shelf pro-

cessors software implemented error recovery is the only option to improve the reliability of the system. However, software methods may introduce a large performance overhead on the system.

The result of compiler-aided optimizations is instrumentation of the executable code with extra instructions for fault detection. The advantage of software solutions over their hardware counterparts is their portability to different hardware platforms without requiring any (or significant) hardware modifications. Furthermore, by using software optimizations, the instruction flow at run-time can be adjusted to achieve a desirable trade-off between reliability, power and performance based on application needs.

Many of the problems discussed above for the hardware methods apply also to software optimizations. Also in software not all optimization methods are fully compatible. The compatibility between compiler-optimization techniques for low power and reliability depends on factors such as level of abstraction in which the methods are applied, performance and memory overheads among other issues. Moreover, executing redundant assertions will in general introduce performance and power consumption overheads. Not surprisingly, all contemporary software techniques with high fault-coverage cause significant performance overheads, making such techniques unsuitable for many embedded systems with high-performance requirements. For example, some conventional recovery methods use checkpoints at predefined locations in the code in order to restart the execution in case an error is detected. Due to the high overhead of checkpoints, some methods rely on using fewer locations to add checkpoints. However, decreasing the number of checkpoints increases the recovery time from the moment an error is detected up to the time the execution is fully recovered. In applications with real-time requirements, the recovery time should be kept as low as possible.

Currently available software methods instrument the code with assertions. The added assertions for fault detection and recovery, cause extra performance and energy overheads. For example, one of the main categories of fault detection is aimed at detecting Control Flow Errors (CFE). The performance overhead of CFE detection methods depends on the characteristics of the CFE detection. In fields with multiple design constraints such as biomedical implants aiming at reliability, low power and performance at the same time, deploying such techniques is not straightforward. Thus the main challenge of compile-time optimizations for fault detection is to minimize the overheads while providing adequate fault coverage.

In this thesis, our aim is to improve the existing software reliability optimiza-

tion methods by answering the following research questions:

- Is there a minimum set of code instrumentation points that guarantee equivalent reliability as compared to existing techniques;
- What is the additional information needed to minimize code instrumentation for reliability;
- Can the process of finding the minimum set of code instrumentation points be captured in a generally applicable design methodology.

1.2 Thesis contributions

The main contributions of this thesis can be summarized as follows:

- An improved categorization of modern software-based reliability optimization methods;
- A careful study of the compatibility between software based reliability optimization methods and conventional power reduction techniques;
- A novel, low-overhead reliability optimization method that is compatible with performance optimization methods, such as loop-unrolling, using control flow graph analysis and workload specific assertions at compile time;
- A system aware technique to identify all susceptible locations to CFE to minimize error detection and recovery assertions.

1.3 Thesis organization

This thesis is organized as follows.

Chapter 2, presents the background concepts in software reliability optimization techniques and the related work in this area. Existing methods are divided into methods that address data errors and those focusing on control flow errors. Moreover, software optimization methods that target control flow errors are introduced and classified into three groups based on the way the assertions work and are added into basic blocks.

In Chapter 3, the compatibility between reliability optimizations and currently practiced power reduction optimization techniques is studied. Based on our analysis, promising combinations are identified that can be used in embedded systems requiring reliability with limited power budget. Moreover, the reasoning behind why some reliability optimization methods and power reduction techniques that are not suitable to be used together is given.

In Chapter 4, a new technique for customizable control-flow fault detection is presented. Our technique is a workload-aware hybrid combination of the two main categories of signature monitoring techniques. Based on the topology of the control flow graph the code is instrumented with a combination of different types of assertions. This workload-aware instrumentation of the code, allows our technique to be used with power reduction optimizations such as loop-unrolling The impact of loop unrolling on the new control-flow error detection method is investigated.

In Chapter 5, a framework for identifying susceptible basic blocks is presented. The introduced framework can be used to omit program instrumentation in basic blocks that are not susceptible to CFEs. Moreover, a new signature monitoring method is presented to be used with this framework. This signature monitoring method has suitable assertions for instrumenting only some of the basic-blocks in the control flow graph (the identified susceptible blocks).

Chapter 6 presents a lightweight, low-latency CFE recovery method with checkpoints only in the susceptible source basic blocks. Our proposed recovery scheme is able to detect the CFE and roll back the execution to the beginning of the basic block where the CFE has occurred. In order to assess our recovery method fairly we considered the three metrics of correctability, performance and recovery time. For this reason, we introduced the recovery efficiency factor that is calculated based on these three crucial metrics.

Finally, Chapter 7 summarizes the work presented in this thesis and concludes on the main findings. Moreover, some future research directions are proposed.

2

Background and related research

Errors at software level may be caused by hardware transient faults. Such errors can impact data correctness or the execution flow integrity. There are several software methods to improve reliability by instrumenting programs with additional code to check run-time program execution and data integrity. The extra code (assertions) are instrumented in the original application to detect and recover from data or control-flow errors. Various software optimizations, e.g., during compilation, can provide the means to achieve this goal. In this Chapter, previous related works in reliability optimizations are presented.

2.1 Introduction

We first introduce the impact of hardware transient faults at software level and give an overview of the previous works on software reliability optimization methods. Hardware transient faults may have different impact at the software level. In what follows the impact of hardware transient faults at software level is modeled into two error categories, data errors and control-flow errors. There are different software optimization methods which target one or both of the above error categories.

The rest of the Chapter is organized as follows: Section 2.2 introduces the two error models representing the impact of transient hardware faults at software level. Section 2.3 presents reliability optimization schemes for hardening the program execution flow against transient hardware faults and Section 2.4 presents reliability optimization schemes for making the program resilient to data error manipulation. Section 2.5 presents hybrid optimization schemes which detect both, execution flow and data integrity errors and Section 2.6 concludes the Chapter.

2.2 Impact of hardware faults at the software level

In recent processor-technology nodes with shrinking transistor sizes, transient faults rates are increasing. Transient faults cause runtime errors, for example unintended sequence of instructions. In this thesis, logical errors are defined as the result of the software bugs. These errors are due to incorrect software implementation and also manifest at runtime. Unlike runtime errors that are due to transient faults, logical errors cannot get detected and recovered using compile time instrumentation. Run time errors at software level occur as consequence of hardware transient faults or Single Event Upset (SEU). SEUs are typically caused by electro-magnetic radiation or wire crosstalk and may result in single bit flips. SEUs assume single bit flips in the memory (data or code segment), buses (data or address), functional units or the control logic. An SEU may change the instruction's address, opcode or its operands. As a consequence, two types of errors may occur during the program execution:

1. **Data errors (DE):** When a fault changes the opcode or an operand of an instruction and cause erroneous data storing in the memory or a register;
2. **Control-flow errors (CFE):** Faults affecting the operand of control-flow instructions or faults converting a non-control-flow instruction opcode to a control-flow one or any other fault (such as a fault affecting program counter content) that causes deviation from the expected execution flow. The consequence is a change in the expected instruction sequence of the program [50].

There are cases that although a control-flow instruction is hit by a fault, the result will be a data error. In these cases the fault will not cause a deviation from the expected execution-flow path as determined at compile time. In such cases, a wrong, however entirely valid control-flow path will be taken. One example of such case is when a fault hits the condition value of a control-flow instruction and causes an erroneous branch execution at runtime. Another example of such cases is when a fault converts a control-flow instruction into a non-control-flow instruction. In both given examples, even though the fault has hit the control-flow instruction, the impact is a data error. In these examples, an erroneous branch gets executed at runtime, but still the executed branch at runtime matches one of the expected execution-flows at compile time. Therefore, the impact of such faults cannot be categorized as control-flow errors and instead it can be considered as data errors.

Related works on reliability focus on one of the two or both types of errors.

But since the effect of data-errors and CFEs are not the same, optimization techniques to protect the application against each of the two error types are also different. Experiments on the influence of heavy-ion fault injection on program behavior show that more than 50% of the injected faults cause CFE [16]. Other works indicate that about 75% of injected data errors are masked [21] [12] [52]. Based on these studies, CFE is a major reason for system breakdown and safety-critical systems require a dedicated reliability optimization for detecting and correcting this class of program execution errors.

2.3 Control-flow checking

Software control-flow checking ensures the correct program-execution order. All proposed methods for detection of only control-flow faults are a version of the *signature monitoring (SM)* scheme. After a control-flow error is detected, a common technique for recovery is to use checkpoints [27]. In this Section, (a) we explain the definitions used in SM methods, (b) we explain the control flow error model, (c) we explain the SM technique at length, (d) we show presented related works on this topic and finally introduce a categorization of currently available methods.

2.3.1 Definitions

In SM methods, two notions are used; *basic blocks* and *Control-Flow Graph (CFG)* of a program. *Basic blocks* are branch-free sections of the program [56]. Each basic block is a set of consequent instructions or statements (depending on the level of abstraction), where only the last instruction (statement) can be a branch, and only the first instruction (statement) can be a branch destination. Programs are divided into a number of basic blocks. In order to model the runtime execution flow, the program is represented as *Control-Flow Graph* (CFG). A program's control flow graph represents the order of basic-block execution. In the CFG, each node corresponds to a basic block and an edge between the two nodes denotes a branch from one basic block to another. A CFG consists of two sets: nodes-set represented by $V = \{v_1, v_2, v_3, \cdots\}$ and edges-set by $E = \{E_1, E_2, E_3, \cdots\}$. The list of legal CFG edges defines the expected execution-flow paths at compile-time. Control-flow errors can be modeled as illegal edges between CFG nodes. Signature-monitoring methods check the execution order of basic blocks using the CFG of the program.

Figure 2.1 shows a schematic view of the CFG for an example piece of code.

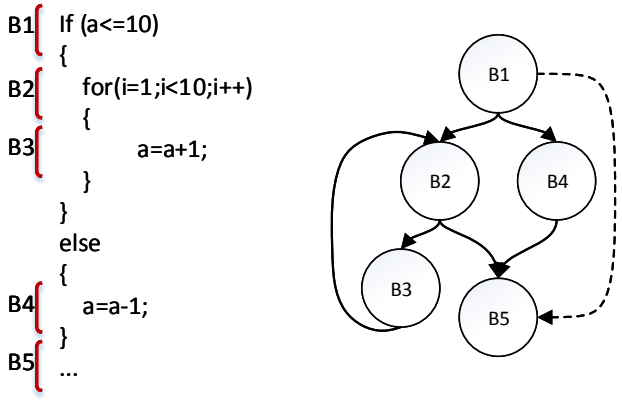


Figure 2.1: Basic blocks and control flow graph of a code sequence

In this Figure, B_1 is the corresponding basic block to the condition statement of the *if* statement. The *if* statement body is mapped to B_2 and B_3 basic blocks, which B_2 is the header clause of the *for-loop* statement and B_3 is the body of the *for-loop* statement. B_4 is the corresponding basic block to the *else* body and B_5 is the consequent branch free statements in the code. In this CFG the node-set is $V = \{B_1, B_2, B_3, B_4, B_5\}$ and edges-set is $E = \{E_{1To2}, E_{1To4}, E_{2To3}, E_{3To2}, E_{2To5}, E_{4To5}\}$. The depicted CFE in the Figure with the dashed line is an edge from B_1 to B_5 which is not present as a legal edge in the CFG edges-set.

2.3.2 Control-flow error model

The impact of transient hardware faults at software execution flow can be categorized in four CFE types based on the reason of the occurrence:

- **NonBranch-To-Branch:** CFEs that occur due to a fault in the opcode of a non-branch instruction and convert it to a branch instruction are referred to as NonBranch-To-Branch. The consequence of this type of CFE is an erroneous branch from the middle of a basic block to an unknown target. This target can be the end of the same basic block or another basic block in the CFG. This type of error is depicted in Figure 2.2(a);

- **Branch-Target-Change:** CFEs which happen due to a fault in the operand bits of a branch instruction cause branch target change are named as Branch-Target-Change. This CFE type causes an erroneous

branch from the end of a basic block to a random location. This type of error is depicted in Figure 2.2(b);

- **Branch-Condition-Change:** Faults affecting the condition argument of conditional branches and cause branch condition change are named as Branch-Condition-Change errors. This type of error cause an erroneous branch execution as depicted in Figure 2.2(c). It is important to note that when this error occurs, the corresponding erroneous edge exists in the list of legal CFG edges, but the wrong edge (branch) is taken at run time due to a faulty conditional argument. The fault affecting the condition argument causes a data error. Data errors should be detected with another group of detection methods as discussed in Section 2.4;

- **Branch-To-NonBranch:** Faults which affect the opcode of branch instructions and change it to a non-branch instruction are called Branch-To-NonBranch. In most systems, the consequent basic blocks are arranged in-order in the memory. For this reason, at least one of the successors of the current basic block[1] is located in the next memory location after the current basic block. When Branch-To-NonBranch error converts the branch instruction at the end of the current basic block, it causes a wrong execution flow to the successor basic block located in the next memory address. Therefore, Branch-To-NonBranch errors may behave as Branch-Condition-Change.

Figure 2.2 shows the execution flow as a result of the above mentioned CFE types on an example CFG. As depicted in this Figure, erroneous branch 1 which is a NonBranch-To-Branch error and erroneous branch 3 which is Branch-Target-Change error can cause skipping part of the current basic block execution without a deviation in the expected execution flow of the program. Such CFE behaviors lead to data errors and depending on the program characteristics can get masked. The corresponding edges to erroneous branch 2 which is a NonBranch-To-Branch error and erroneous branch 4 which is Branch-Target-Change error is not included in the list of legal CFG edges. Such erroneous branches leads to execution flow deviation and are the errors targeted by the signature monitoring methods. Erroneous branch 5 can be the result of a Branch-Condition-Change or Branch-To-NonBranch errors. Since such erroneous branches does not change the expected execution flow, they will not be detected by signature monitoring based methods.

[1] the basic block that contains the instructions being executed

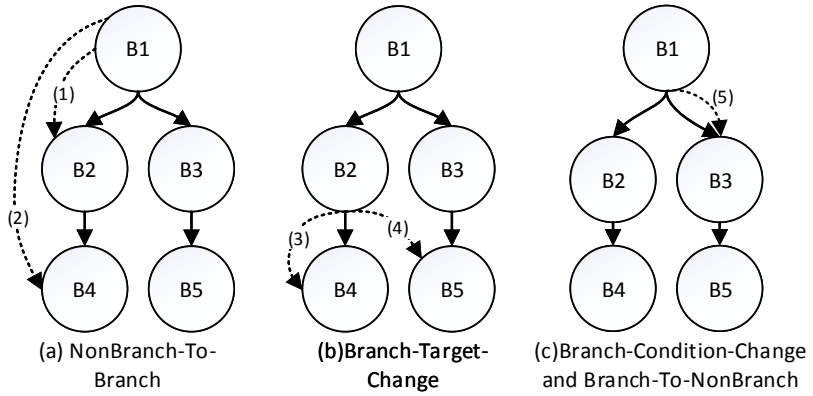


Figure 2.2: Different targeted error types in signature monitoring.

Faults may happen in different hardware components. If a transient fault affects hardware units (such as program counter, address bus and etc) which contain the address of an instruction, the resulting error can be a CFE. However, the way this CFE affects the execution flow in the CFG depends on the affected source address and the address it has been converted to. In other words, the effect of such faults on execution flow can not be known in advance. On the other hand, if the fault occurs in units which contains the instruction, the behavior is known and can be categorized into one of the the above four categories.

2.3.3 Signature monitoring

The basic idea of signature-monitoring techniques is to have a static signature for each basic block of a given program and a global dynamic signature. In all CFE detection methods a unique static signature is associated to each basic block. In addition to the basic block signatures, there is one runtime signature which is calculated and updated at runtime; Runtime Signature (RS). RS value depends on the program's execution flow and the basic blocks that are visited during execution. The content of the static signatures is defined before runtime, while the dynamic signature (RS) is calculated at runtime. At runtime, with each execution transfer to a new basic block, the RS is calculated and updated to the signature of the new block. By comparing the two signatures after control flow transfer, the correct run-time execution order of basic blocks is checked. Figure 2.3 depicts modified basic blocks for deployment of signature monitoring.

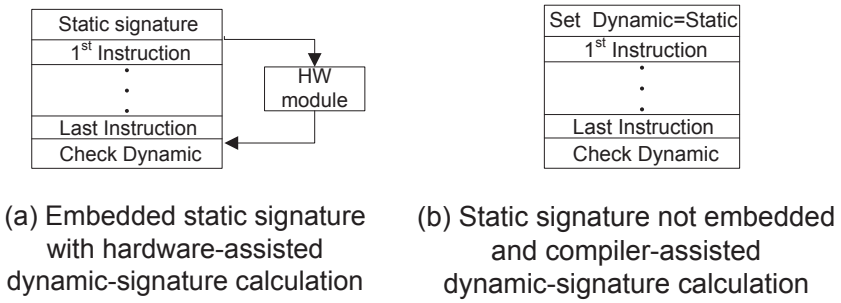


Figure 2.3: Static and dynamic signatures in basic blocks.

The static signature of a basic block can be one of two types; **assigned signatures** or **derived signatures** [30]. Derived signatures are formed by the address of each basic block. They are derived by the linker or assembler and embedded in predefined points of the basic blocks, very often at the very beginning of a basic block. Assigned signatures are unique values given to each basic block that are also embedded in the predefined points in the basic blocks. Figure 2.3(a) shows a sample basic block with derived embedded signature and hardware-assisted run-time signature calculation and Figure 2.3(b) depicts a basic block with assigned signature (not embedded in the basic block) and compiler-assisted run-time signature calculation.

RS calculation can be done by extra dedicated hardware (e.g., a watchdog processor or a signature generation unit) or implemented in software. Calculating the run-time signature in software can be done by a watchdog task or by the assertion code introduced by the compiler. Obviously, using a watchdog task running in parallel with the main program, will require a multi-tasking system.

At compile time, the code is instrumented with dedicated functions for run-time signature calculation and monitoring. The dedicated functions for RS signature calculation are called Signature Generating Functions (SGF) and the functions for checking the RS are called Signature Monitoring Functions (SMF). At runtime, SGF calculates RS signature and SMF checks the consistency between the current basic block signature and this calculated signature at runtime (RS). If the program control flow is fault free, RS content matches the current basic block signature and the control flow is validated. Since SGF assertions set the runtime signature, they can also be called *set* assertions and similarly SMF assertions can be called *test* assertions. These functions are added at predefined points of the basic block. SGF assertions are used in all basic blocks to update RS along the control flow path. However, depending on

the optimization method, SMF assertions can be added only to locations where checking is considered to be critical.

Depending on the implementation, the SGF can be a simple *XOR* instruction or a statement executing a complex function, e.g., a CRC computed of the bit-field of all instructions in the basic block. As result of the specific SGF and SMF complexity, some methods require storage of additional static parameters for each basic block at compile time. These parameters hold information about successors or predecessors of each basic block in the CFG. The amount of static parameters to be stored is determined at compile time and is one of the main contributors to memory and performance overhead of each SM scheme.

2.3.4 State of the art signature monitoring techniques

Signature monitoring related works all use SGF (set) and SMF (test) assertions. Some of these methods require to store additional information at compile time; the so called Control Flow parameters (CF-parameters). Currently available SM schemes are presented in Table 2.1. In the Table, *set* (SFG) and *test* (SMF) assertions and the extra CF-parameters (static parameters) saved at compile-time are presented. In this Table, *RS* is presenting the global run-time signature. The signature of each basic block is denoted by "*i*" subscript (S_i). Predecessor signatures are denoted by "*pre1*" or "*pre2*" subscripts (e.g., S_{pre1}), and successors signatures are denoted by "*nxt1*" or "*nxt2*" subscripts (e.g., S_{nxt1}). Extra parameters stored at compile time are indicated with "*P*". Parameters may belong locally to each basic block. In this case, they are also denoted by "*l*" subscript (P_l). Parameters that are global for all basic blocks are denoted by "*g*" subscript (P_g). As an example in the Table, P_{g2} in the CFCSS method [30] is a global static parameter. Depending on the method, a number of static parameters may be required. As an example, YACCA needs to store three local parameters at compile time as shown in this Table.

These methods differ in a number of aspects: the insertion point of SGF and SMF; the abstraction level; the type of detected errors. The type of errors detected by the methods is affected by the place where SGF and SMF are inserted in the basic blocks. Figure 2.4 shows possible insertion points for each SGF and SMF pair. Each SGF comes with a corresponding SMF checking the updated RS by the SGF. They can be in the same block or in two consecutive blocks. Figures 2.4(a) and 2.4(b) depict methods wherein the SGF_i of the basic block and its corresponding SMF_i are in the same basic block ($block\text{"}i\text{"}$). On the contrary in Figure 2.4(c) the SGF_i of $block\text{"}i\text{"}$ is checked by SMF_{i+1} in the successor basic block. In this Figure, the related works that use each type

SM METHODS	SGF	SMF	ADDITIONAL STATIC PARAMETERS
CFCSS [30]	$RS = RS \oplus P_{l1}$ (1) $RS = (RS \oplus P_{l1}) \oplus P_{g2}$ (2)	$ifRS! = S_i$ *br error*	$P_{l1} = S_i \oplus S_{pre1}$ $P_{g2} = \begin{cases} 0000 & \text{in pred 1} \\ S_{pre1} \oplus S_{pre2} & \text{in pred.2} \end{cases}$
CEDA [46]	$RS = RS \oplus P_{l1}$ (1) $RS = RS \oplus P_{l2}$ (2)	$ifRS! = S_i$ *br error*	$P_{l1} = S_{pre1}$ $P_{l2} = S_{nxt1}$
ECCA [5]	$RS = P_l + (RS - S_i)$	$RS = \frac{S_i}{\overline{RS\%S_i} \cdot (RS\%2)}$	$P_l = \prod S_{nxt}$
YACCA [3]	$RS = (RS\&P_{l1}) \oplus P_{l2}$	*If* $(P_{l3}\%RS)$ *error* ()	$P_{l1} = S_{pre1} \bar{\oplus} S_{pre2}$ $P_{l2} = S_{pre1}\& (S_{pre1} \bar{\oplus} S_{pre2}) \oplus S_i$ $P_{l3} = \prod S_{pre}{}^3$
CCA [19]	$setRS_1 = S_{1i}$ $enqueueS_{2i}$	$dequeueRS_2$ $ifRS_1! = S_{1i}$ $error()$ $ifRS_2! = S_{2i}$ $error()$	not required
[35]	$setRS = S_i$	$ifRS! = S_i$ $error()$	not required
ACFC [50]	$RS = RS \oplus MASK$	$ifRS! = const.$ *error*	not required
ACS [21]			not required
BSSC [28](SM part)	*call entry routine*	*call exit routine*	not required
[11] and [54]	RS calculated in hardware	$ifRS! = S_i$ $error()$	not required

Table 2.1: SGFs, SMFs and additional static parameters of SM methods for RS generation

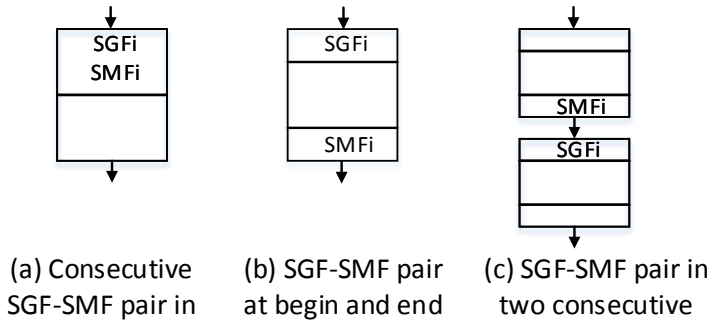


Figure 2.4: Different insertion points for SGF and SMF pair.

of insertion points are also given.

In CFCSS [30], the insertion point of the corresponding SGF_i and SMF_i pair is shown in plot (a) in Figure 2.4. Thus, faults happening after the SGF_i and SMF_i (causing errors of type 2) are not detectable. In ACFC [50] the SGF_i is in the beginning of each basic block and SMFs are added at the end of control flow paths in the program. CCA and BSSC both have 2 signatures for each basic block. In BSSC, among the two signatures per basic block, one is a derived signature and another one is assigned signature. The derived signature is the beginning address of each basic block. CCA also uses two signatures for each basic block, however both of them are assigned signatures. YACCA [3] and ECCA [5] techniques are good examples of having a corresponding SGF_i and $SMFi+1$ pair split over two consecutive blocks, as depicted in Figure 2.4(c). In CEDA [46] the signature update is performed in two locations of the basic block, once at the beginning of the basic block and once at the end. The SGF function at the beginning of the basic block is an XOR between the runtime signature and the static parameter P_{l1}. The SGF at the end of the basic block is also an XOR operation which is performed between the runtime signature and P_{l2} parameter. If the basic block has multiple predecessors, and at least one of the predecessors has multiple successors, the SGF at the beginning of the basic block performs an AND operation between the run time signature and P_{l1} parameter.

The number of assertions in basic blocks and CF-parameters are crucial factors determining the memory overhead. Moreover, the number of assertions used per basic block and their complexity determine the introduced performance and power overheads. CFCSS, ECCA and YACCA save extra CF-parameters to handle multiple predecessors/successors structures. Thus, they introduce

additional memory overhead. CCA does not save CF-parameters in expense of not finding faults in multi-predecessor/successor topologies. ACFC and ACS, use lower number of assertions and therefore have the lowest overheads.

Recovery methods use a detection mechanism to first detect the fault and use checkpoints and roll back to recover the execution. Traditional methods place checkpoints at critical points of the program. Checkpoints impose a high performance overhead and in most systems the performance is traded off for the recovery time. One of the recently proposed methods is ACCE [47]. It does not use checkpoints due to the high performance cost. The authors propose a roll-back mechanism using a global error-handler function and local fault-recoveries per function. In this method the data is not restored after the roll back. However it is suggested to extend the method with data duplication and comparison for data restoration. We believe that the cost of data duplication and comparison will not be lower than when using checkpoints.

2.3.5 Control flow error recovery methods

Recovery methods use a detection mechanism to first detect the fault and then use checkpoints to roll back in order to recover the error free execution. Traditional methods place checkpoints at critical points of the program [27]. However checkpoints impose a high performance overhead. Previous work have tried to find optimal locations for adding checkpoints in order to avoid excessive performance overhead while limiting the recovery time [7]. These studies use a mathematical model to determine the optimal checkpointing intervals. In the proposed solutions the application level performance is traded off for the recovery time.

A recently proposed lightweight checkpointing method, uses static analysis of applications to find idempotent regions[2] [13]. These lightweight checkpoints save only the registers state at the beginning of the idempotent regions. The starting address of the idempotent region is saved to roll-back the execution in case an error is detected. This lightweight checkpointing imposes low runtime performance overhead. However, the recovery time between when an error is detected until the moment that the execution is recovered to the location where the error has occurred is dependent on the idempotent regions size. Larger idempotent regions reduce performance overhead but increase recovery time.

One of the recently proposed methods is ACCE [47]. It does not use checkpoints due to its high performance cost. The authors propose a roll-back mech-

[2] code regions without Write-After-Read dependencies

anism using a global error-handler and recovery routines for each function. ACCE provides the fastest recovery mechanism among the previously proposed recovery techniques. The global error-handler is responsible to determine from which function in the program the error has initiated. After it is determined which function is the source of the error, the recovery routine of that function finds the basic block in which the error has occurred and the execution is rolled-back to the beginning of that basic block. The recovery time in this mechanism is equal to the total number of cycles for executing the global error-hander and the corresponding recovery-routine. One shortcoming of ACCE is that the data is not restored after the roll back. In order to also restore the data after roll-back the authors suggest to use data duplication mechanism. The overall cost of data duplication and comparison in terms of performance is expected to not be lower than using checkpoints.

2.3.6 Error-capturing instructions (ECI)

ECIs are special instructions residing in memory locations which are not reachable during normal program execution. This technique can detect fair amount of control flow errors. The control flow errors detected by ECIs are errors causing execution to diverge permanently from the correct execution and branch to an erroneous location in the code segment memory. ECIs are inserted in unused code and data segments in the main memory. In data segment, they are inserted among ordinary data and in code segment they are inserted in some parts of the program that would not be reached during normal execution. ECIs proposed in [28] are of two types: A software-interrupt instruction or a branch instruction making an infinite-loop along with a watchdog timer.

2.4 Data error detection and recovery

In the following text we discuss the two types of methods that are proposed for detecting data errors; *data-duplication-with-comparison* and *executable assertions*. Each method can be extended to also recover from the errors.

2.4.1 Data error model

Data-value errors affect the system in three ways:

- Erroneous data stored in memory or registers;

- Erroneous non-branch instruction execution due to change of its opcode and conversion to another non-branch instruction;
- Errors in conditional branches due to faulty data value of the condition argument. This data value error causes Branch-Condition-Change type of control flow error (depicted in Figure 2.2(c)). However, since it is data value error, it should be detected by data error detecting methods.

Errors as mentioned above should all be handled using data error detection and recovery methods.

2.4.2 Data-duplication-with-comparison

The basic idea of data redundancy is to copy and save a duplicate version of the original code at compile time, called **shadow** [31], and compare the original and shadow versions at runtime. The comparison is done at critical points of the program. Different software methods of this category vary in three aspects: First, the level of abstraction where data is duplicated; second, parts of the system in which data is duplicated- the so called Sphere of Replication (SoR)[3]; and third, critical points to compare the original and *shadow* data. Critical points to compare the original and *shadow* data are places where the program output is written to the memory or the execution flow of the program is determined. Thus, the three points in the program to insert comparisons are: Before "*store*" instructions; before "*branch*" instructions and before system calls (calls to external libraries).

Data duplication at source code level duplicates variables, assignments and operations. At this level, variable duplication happens regardless of where the values of variables are stored: at a memory location, in the register file or internal cache. At lower levels, e.g., assembly level, assembly instructions are duplicated. Figure 2.6 shows data duplication at high-level source code (a) and at assembly instruction level (b). Well-known methods proposed in this category are EDDI [31] and SWIFT [36] at instruction level; the methods proposed in [34] and [35] at source code level. In EDDI [31], comparison instructions are inserted before *store* and *branch* instructions. In source-code level methods proposed by Rebaudengo [34] and [35], after each *read operation* on a variable, the contents of the original and shadow variables are compared. Figure 2.5 shows an example piece of code that is instrumented at source code level.

[3]This parameter reflects the protected parts in the system

```
int main(int x)
{   int c=3;
      if (x== 5)
            x= c+2;
      else
            x= c-2;
      c= x;}
```

(a)

```
int main(int x0, int x1)
{   int c0=3;       int c1=3;
     if(x0==5)
      if(x0 != x1)
         error();
      x0= c0+2;   x1= c1+2;
      if(c0 != c1)
         error();
   else
      x0= c0 - 2;   x1= c1 - 2;
      if(c0 != c1)
         error();
   c0= x0;          c1= x1;
   if(x0 != x1)
      error();
   return 0;}
```

(b)

Figure 2.5: (a) source code, (b) modified code based on optimization in [34]

EDDI and generally all methods with data duplication and comparison are able to detect control-flow errors of type Branch-Condition-Change. If a transient fault affects an input argument of a conditional branch and an erroneous branch is executed, it can be detected by checking the value of the input arguments of the conditional branch.

2.4.3 Data error recovery methods

For recovering from data errors, data-duplication-with-comparison methods can be extended to recovery methods using triplication or checksum computation. Methods with data duplication with comparison require to keep a redundant version of the data for recovery purposes. Such methods typically cause significant memory and performance overheads.

SWIFT-R is an example which uses triplication with a majority-voting mechanism [9]. The authors in [34] propose to use data duplication and to store a checksum over the data for fault recovery. A mismatch between the duplicated data and the original version identifies a fault. For recovering from this fault the checksum of the two data versions is computed and compared against the stored checksum to identify which data version is the fault-free one.

Another, previously proposed technique is algorithm based fault tolerance [23]. This technique is applicable only for specific workloads which have matrix operation. In this techniques the matrix transformed into checksum matrix which has a row checksum and a column checksum. Each element of

Original code	Optimized code
$a = b + c$	$a_0 = b_0 + c_0;$ $a_1 = b_1 + c_1;$ $if\ ((b_0 != b_1) \cup (c_0 != c_1))$ $error\ ()$

Original code	Optimized code

Add R_3, R_1, R_2	Add R_3, R_1, R_2
	Add R_{23}, R_{21}, R_{22}
	BNE R_{23}, R_{21} goto error

Figure 2.6: Data duplication (a) at source code, (b) at instruction level

the column checksum is the sum of the elements in the corresponding column. In a similar way, each element of row checksum is the sum of the elements in the corresponding row. In order to check whether there is a faulty data in the matrix or not, the sum of all elements in the rows are compared to the row checksum and the sum of the elements in the columns are compared to the column checksum. A mismatch in these comparisons shows that an error has occurred. If the error has occurred both in row and column checksums, it shows that the error is the element at the intersection of the row and column with mismatch. In this case the error can be recovered by recalculating the erroneous element value using row checksum and column checksum. If the error is only in row checksum or only in column checksum, it shows that the original data is not affected and instead the checksum value is affected. In this case, no recovery is needed, because the data is error free.

In [22], the authors determined that the majority of transient faults can either be ignored (because they do not ultimately propagate to user-visible corruptions at the application level) or are easily masked by lightweight symptom-based detection. Therefore, they use compiler analysis to find high-value portions of the application code that are both susceptible to soft errors and statistically

unlikely to be covered by the timely appearance of symptoms and protect only these portions of the code with instruction duplication. Their solution offers an optimized fault recovery for data errors that is trading off performance and fault coverage.

2.4.4 Executable assertions

By adding extra statements to the program, the validity of specific constraints is tested. The added test statements are called executable assertions. An example of this method is discussed in [51]. In this work, *outputs* and *state variables* are checked with executable assertions. Executable assertions can also be used for masking errors, as the assertions proposed in [9]. In this work, assertions protect variables with statically known values against faulty change of their value. This is done by inserting AND/OR operators at compile time.

2.5 Data and control flow checking

There are methods which can be employed for both data and control-flow error detection. These methods combine data-value and control-flow checking as a hybrid technique. Examples of hybrid techniques detecting both data and control-flow errors are [36], [37] and [35]. Hardware support can be used to reduce performance overhead and code size by reducing the amount of data duplication. For example, both [36] and [37] use hardware support to reduce the overhead.

The method in [36] (SWIFT) is based on software optimization, however it is assumed that the memory hierarchy is protected with some form of error correction. It applies a modified version of EDDI [31] for data-error checking and a modified version of CFCSS [30] for control-flow checks. Since it is assumed that the memory is protected by ECC and parity bits, the need for duplicating "*store*" instructions is eliminated. By having CFCSS for control-flow checks, the duplicating branch instructions is also not necessary. Moreover, in order to further limit the imposed overheads, the SMF of control-flow checking is applied only in basic blocks containing "*stores*".

2.6 Conclusions

In this Chapter we explained different previously proposed compiler optimization methodologies for increasing reliability. The techniques for improving the reliability are categorized into: signature-monitoring schemes, error-capturing-instructions, data-redundant methods and executable assertions. While signature monitoring and ECI are targeting control-flow errors, data-redundant and executable-assertion techniques aim to find data-value errors. Some of the proposed methods (SWIFT), consider hardware support for memory protection and therefore is not completely independent from the hardware. The studied works instrument the program without taking into consideration the information available at compile time and therefore impose a large performance overhead to the system. In the following Chapters we propose techniques for instrumenting the code using compile time information to reduce the imposed overheads.

3

Reliability and power optimization techniques investigation

Historically standard built-in compiler optimizations have been used for improving performance in embedded systems. However, for a wide range of today's battery operated embedded devices with restricted power budgets, power consumption becomes a crucial problem that is not often addressed by modern compilers. Biomedical implants are one good example of such systems with highly limited power budgets. Additionally, as discussed in the previous Chapter, these devices need to also satisfy high reliability levels. In this Chapter, we elaborate the needs of a cochlear implant as the case study of these systems. Further, we categorize previous works on compiler optimizations for low power and fault tolerance. Our study considers differences in instruction count and memory overheads, fault coverage, abstraction level of optimizations and requirements for additional hardware support. Finally, the compatibility of different methods from both optimization classes is assessed. Two optimization method pairs from both classes that can be successfully combined with no limitations have been identified.

3.1 Introduction

The goal of this Chapter is to highlight the compatibility of reliability and power-optimization techniques. But first a case study of biomedical implants, a cochlear implant, is elaborated that shows the need of having compatible reliability,power and performance optimization techniques.

Cochlear implants are commonly accepted as therapeutic devices for clinical use restoring the hearing of profoundly deaf people. Cochlear implants devices consist of an external part that comprises a speech processor (DSP) and

a microphone which together receive and convert the sound into a digital data stream using a speech processing strategy. The digital data is then transferred via an RF link to the internal part. The internal part consists of a receiver-stimulator package, which receives power and decodes the instructions for controlling the electrical stimulation via an electrode array placed inside the cochlea. Users can have normal conversation in a relatively clean sound environment, but their hearing performance drops in complex environments, causing poor appreciation of music and inability to converse in crowded rooms (cocktail-party effect). The bottleneck is delivering more sound details with higher performance than is currently possible with the device used today. In order to deliver more sound details from the external to the internal part, software optimization on the algorithms running in the DSP can be used to generate reliable compressed digital data. In addition to this, the device requires the power consumption to be as small as possible to avoid the need for big batteries or capacitors. In order to improve cochlear implant devices a multi-disciplinary approach is required that takes into account reliability, power limitations and performance requirements all at the same time.

The rest of this Chapter the compatibility of reliability and power optimization techniques is analyzed. In a nutshell, the following contributions are made:

- Categorization of reliability-related optimizations based on the targeted errors, level of abstraction and checking method;
- Categorization of optimization methods for power reduction, based on power-consumption sources;
- Analysis of each technique in terms of performance overhead, memory overhead, hardware modifications and compilation time;
- Proposing hybrid optimizations tuples for power reduction and reliability based on the results of our analysis.

The rest of the Chapter is organized as follows: Section 3.2 categorizes signature monitoring schemes for CFE detection and gives a numerical analysis of a method from each category. The methods for optimizing power consumption are categorized in Section 3.3, Section 3.4 discusses the compatible and contradictory methods between reliability and power reduction optimizations and Section 3.5 concludes the Chapter.

3.2 Signature monitoring categorization and analysis

As discussed in the previous Chapter, signature monitoring methods add *set* assertions to all basic blocks to update the runtime signature along the control flow path. However, depending on the optimization method *test* assertions are added at specific program locations considered to be crucial. Based on the locations where the test assertions are added, CFE detection methods can be divided into two main categories. Figure 3.1 represents this categorization.

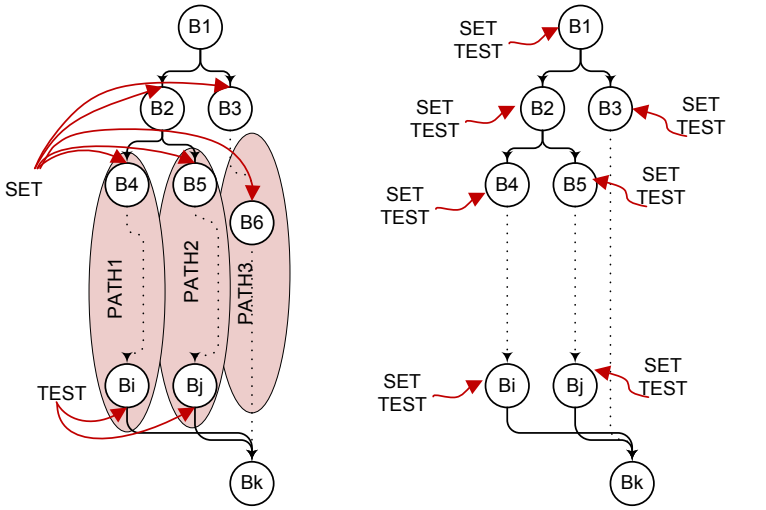


Figure 3.1: Two categories of signature monitoring techniques

The two main categories are *path-asserting* and *predecessor/successor-asserting* methods as described below:

1. **Predecessor/successor-asserting:** methods which assert if the previous (or next) basic block in the execution flow is the correct predecessor (or successor) and is in accordance with the control flow graph edge-set;

2. **Path-asserting:** methods which assert if the control-flow path during the execution is correct or not. A *path* is two or more basic blocks which are executed in a sequence. The path is also defined with a group of edges. Path asserting methods ensure that this group of edges are in accordance with control flow graph edge-set.

Predecessor/successor-asserting methods require more than one assertion per

basic block (at least one *set* and one *test*). Moreover, in some cases there is a need to store information about predecessors/successors (especially for basic blocks with multiple predecessors/successors); the so called Control Flow parameters (CF-parameters). However, path-asserting methods decrease the number of assertions per basic block (since the *test* assertion is added only to the last basic block of the path) and in case of multiple predecessors/successors do not require to save any extra information. On the other hand, path-asserting methods assume programs have symmetric CFG topology, which is not realistic for most of the real programs. Instead, predecessor/successor-asserting methods are independent of the CFG topology (because they save control-flow information for each predecessor/successors pair). The main difference between the two categories is in the number of added *test* assertions in the program. Predecessor/successor-asserting methods add more *tests*, therefore they introduce higher overheads and also potentially improve fault-coverage. Path-asserting methods add lower number of *test* assertions, introduce less overhead but somewhat decreased fault-coverage. CFCSS, ECCA, CEDA, YACCA and CCA represent predecessor/successor-asserting methods with high fault coverage and high overhead. Among this group CCA has the simplest *set*/*test* assertions and it does not have any extra CF-parameters. ACFC and ACS add one *test* assertion for group of basic blocks and are categorized as path-based methods. ACFC is a path-based method that adds *tests* to the last basic block of the loops and the exit basic block of the program. ACFC does not add *tests* at the end of paths which are formed due to conditional branches. As a consequence, it is efficient only in programs with symmetric CFG topology. If a program's CFG is not symmetric due to unbalanced conditional statements (an *if* without an *else* counterpart), ACFC adds dummy *elses* to balance the CFG and then instrument it. This causes extra branches and is the major reason of overhead in ACFC.

ACS is also a path-based method, since it adds one *test* assertion for group of basic blocks in single-entry-multiple-exit regions. ACS offers a coarse grain CFE detection, useful in commodity systems which require high performance while 100% fault-coverage is not demanded as in safety-critical systems. Both methods fall in the path-based category and use simple and low-cost assertions.

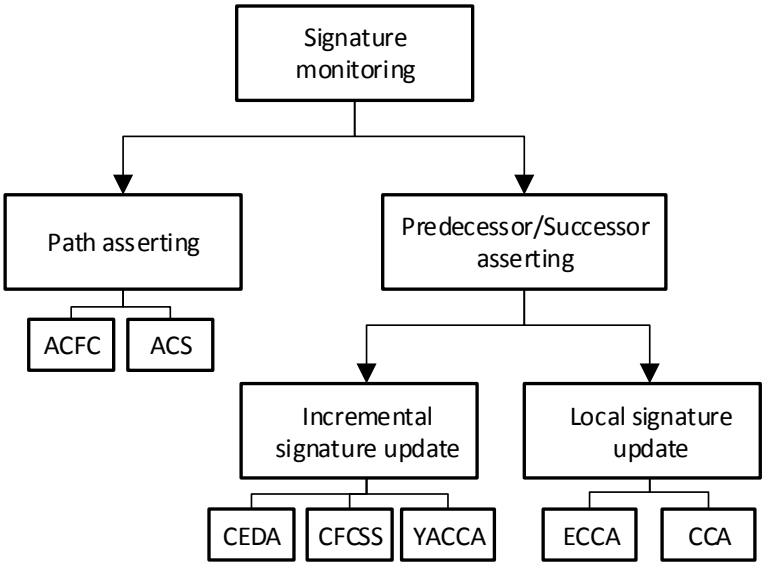


Figure 3.2: Categorization of signature monitoring methods

Figure3.2, shows the above mentioned categorization. As shown in this Figure, furthermore we categorize predecessor/successor assertions into two groups: 1) methods with incremental signatures update; 2) methods with local signature updates. Figure 3.3(a) shows the differences between these two groups and in Figure3.3(b) the instrumentation with path-based assertions is depicted. At incremental signature update, the set functions use the global signature content (S in Figure 3.3(a)) as input. This means that global signature content at each basic-block is dependent on all set assertions in the predecessor basic-blocks along the execution path. Methods with incremental signature update are CEDA, CFCSS and YACCA. On the other hand, local signature updates set the signature at the current basic-block independent of the global signature content. ECCA and CCA are illustrative examples of methods with local signature update. The shortcomings of local signature updates are high overheads and low fault-detection capability for basic-blocks with multiple predecessors. Methods such as CFCSS and CEDA that are incremental signature update methods, have lower overheads than local signature monitoring with the same fault-coverage.

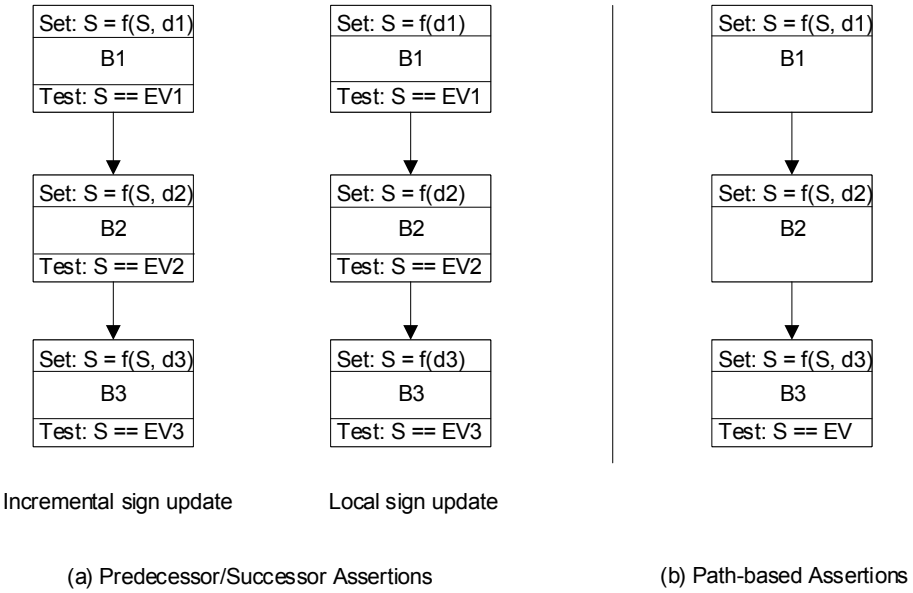


Figure 3.3: Predecessor/successor assertions with incremental/local signature update and path-based assertions

3.2.1 Quantitative analysis

Each of the methods discussed above differs in terms of the type of detected faults, instruction count and memory overheads. Table 3.1 shows a numerical comparison in terms of overhead between the three signature monitoring categories. In this Table, CFCSS and YACCA belong to incremental signatures update category, CCA belongs to local signatures update category and ACFC belongs to path-asserting methods category. To perform this numerical analysis, we have used PowerPC (PPC405) as our target architecture for estimating the overheads. For this estimation, a synthetic workload is used. This workload contains a simple for-loop with an add operation in the body. The CFG of this workload contains four basic block in total. The simple workload is chosen for simplicity in order to facilitate instruction count and comparison between different methods. Figure 3.4 shows the CFG of the used workload. In this Figure, B1 block contains the initialization statement, B2 has the test operation of the loop condition, B3 has the addition operation and the increment of the for-loop and B4 corresponds to the return statement.

In order to compare SM methods, in Table 3.1 the memory overhead of apply-

Methods	Memory overhead (bytes)	Instruction count overhead	Category	Opt. phase
CFCSS [30]	2	61	incremental signatures update	compiler
YACCA [3]	8	88	incremental signatures update	preproc.
CCA [19]	4	356	local signatures update	preproc.
ACFC [50]	1	5	path-asserting	preproc.

Table 3.1: Analysis of reliability optimization methods

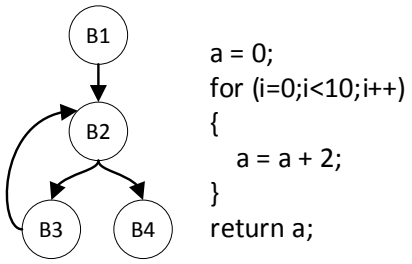


Figure 3.4: CFG of the used workload for overhead estimation

ing different SM methods are estimated. We have used CF static parameters (as given in Table 2.1 in Chapter 2) to estimate the overheads in the studied SM methods. The memory overhead of signature-monitoring schemes is dependent on the number of stored static signatures and CF static parameters for each basic block. The estimation is done based on the number of bits required to store the static signatures and parameters.

Moreover, in order to have a better view for comparing the methods, instruction count overhead is calculated. Instruction count is influenced by the complexity of the SGF and SMF. The workload under study is compiled for the PPC405 architecture and corresponding instructions to SGF and SMF are counted.

The results in Table 3.1 show that the least imposed overheads belong to ACFC which is path-asserting method. This method requires only one bit per basic block to store static information about the control flow. As our small synthetic workload has only four basic blocks, the estimated memory overhead for this method is four bits (half a byte). However, in practice one memory byte will be used as given in the Table. In the second place, CFCSS and YACCA, which

are both incremental signature update methods, have the least imposed overheads. Finally, CCA, which is a local signature update method, has the highest overheads among the two others. This comparison emphasizes the differences between the imposed overheads by the three signature monitoring categories.

It is important to note that in this Table CFCSS is the only method that uses the compiler for instrumenting the code at assembly level. All other methods use preprocessor to instrument the code at source code level. Instrumenting the assembly code results in lower overheads compared to instrumenting the source code. The results of YACCA and CFCSS emphasize this fact. YACCA instruments the source code using the preprocessor while CFCSS instruments the assembly code using the compiler.

3.3 Optimization techniques for power reduction

In addition to static leakage, another major source of power consumption in CMOS circuits is due to dynamic power consumption. Dynamic power consumption is given by [49]:

$$P_{dyn} = \alpha \cdot C \cdot V^2 \cdot f \tag{3}$$

The total energy consumption in a system is the product of the consumed power and the execution time (t) [45]; ($E_{dyn} = P_{dyn} \cdot t = P_{dyn} \cdot N \cdot T$). N is the number of clock cycles that the device is operating and T is the clock period. Thus the dynamic energy consumption is calculated as:

$$E_{dyn} = \alpha \cdot C \cdot V^2 \cdot N \quad , (T = 1/f) \tag{4}$$

In order to reduce the total energy consumption of the system without redesigning the hardware, three factors in the formula can be reduced; reduction of the activity factor (α); reduction of the operating voltage (V) and reduction of the number of cycles device is operating (N). A figure of merit describing the density of bit transitions in a processor executing a consecutive set of instructions is the *Hamming distance*[1].

Based on the current measurements performed in [45] and the defined amount of drawn current in each part of the processor, three main parts are identified as the source of power consumption in the system: (1) Bus lines driving the off-chip storage elements; (2) Processing units; (3) Pipeline latches. Different power optimization methods target one of the above mentioned sources of

[1] Hamming distance is the number of bits that have different values in two words

power consumption and decrease the power by minimizing one of the factors α, V or N.

In the rest of this Section, these methods are explained and are grouped in two main categories; the ones requiring special hardware support (or modification) and the ones that are independent of the underlying hardware.

3.3.1 Hardware assisted power reduction techniques

Optimizing methods for power reduction which require hardware support can be of three types. Each of these methods aims at reducing power by decreasing one of the α, V or N factors.

3.3.1.1 Memory address coding for power reduction

Transmitted data on the buses can be encoded in order to decrease bus lines transitions and hence power and energy consumption. Gray-coding is a well-known technique for keeping the Hamming distance between two consequent codes constant and always equal to one. Instruction addresses in the memory are most of the time contiguous and the execution of programs is sequential. Therefore, Gray-code can be used to address the memory and reduce the Hamming distance between consequent instructions [43]. However in order to use Gray-code for addressing, there is need for hardware support and a memory with Decimal to Gray-code decoder and encoder.

To decrease bus transitions two types of coding are used; Gray-coding and Bus-invert coding. Gray-code is used for Addressing. Hardware and compiler should be modified in order to use Gray-coding; e.g., the program counter should be able to increment in Gray-code and compiler should re-arrange the addresses in branch instructions. An alternative method to Gray-coding is Bus-invert coding. Bus-invert coding is used in case the Hamming distance is greater than half of the bus size. This means that with using the Gray-coding more than half of the bits should be switched in two consecutive accesses. Instead, with this technique the bus-encoding is inverted, thus avoiding that more than half of the bus line switches and saving at least 50% of power consumption. An extra bit line between the memory controller and the memory device is required in order to indicate if the bus is in the original or in inverted state.

3.3.1.2 Deactivating modules for power reduction

In order to decrease the number of active clock cycles of the processor or memory (*N* factor in equation 4), some methods partially shut down idle parts of the module ([41] and [18]). However, this scheme is useful only when the corresponding parts are idle for a long continuous periods. There is a need for look-ahead algorithms (such as the one in [18]) to predict the periods in which the processing units or memory are active. In this method, power-management techniques are used for event-driven applications. The proposed look-ahead algorithm investigates the history of the previous idle periods and the power dissipation of the target system.

3.3.1.3 Dynamic voltage scaling

Variable-Voltage Processors (VVP) are processors with controllable levels of operating voltage. The voltage level can be tuned to high for boosted performance and fast execution and to low for idle or low activity periods. The main challenge is to find a scheduling algorithm exploiting the variable voltage and clock speed of the CPU for power reduction without significant performance overhead. Related works in [29], [25], [55] and [53] give scheduling schemes for solving this issue. In most of these works, in order to schedule the tasks, the Worst-Case-Execution-Time (WCET) of the tasks is considered. But in real execution, the tasks are executed faster and the slack time is divided between other tasks.

A critical challenge in VVP scheduling is to find optimal points in the program for changing the voltage. In [55] and [29] this is done in two different ways. In [55] the optimal insertion points for voltage-scaling instructions are investigated by profiling the program and storing the information about the execution time and amount of energy required by each basic block. By using Mixed-Integer Linear Programming (MILP) technique, the places to change the voltage level without significant performance overhead are defined. The proposed method in [29] uses the compiler and the operating system for scheduling real-time applications on VVPs. The compiler annotates the application sections with timing information (the so-called Power-Management-Points, PMP). At the PMP in the beginning of a task, current time is stored and, at the last PMP, execution time of the task is calculated. At run-time, the actual execution time is compared to WCET (defined at compile time) of the task. Based on the comparison, the operating system adjusts the speed and voltage of the processor and allocates the slack time to other tasks. The critical points to insert

PMPs are loop boundaries and procedure calls.

3.3.2 Software techniques for power reduction

Software techniques for power reduction limit the number of memory accesses or re-schedule instructions in a way that the Hamming distance between them is decreased. In what follows software power reduction techniques, that are independent from hardware, are explained in more details.

3.3.2.1 Power reduction by increasing locality

Proper mapping of data into the memory reduces the accesses to the external memory and bus transitions. Interleaving array elements is one way to increase locality by mapping multiple arrays in the memory into a single array. The method proposed in [10] is an example. Multiple arrays in different addresses in the memory are mapped into a single array in the memory; mapping two arrays of A and B to D:

$$\begin{array}{l} For(i = 1, i \leq N, i++) \\ C[i] = A[i] + B[i]; \end{array} \quad \Longrightarrow \quad \begin{array}{l} For(i = 1, i \leq N, i++) \\ C[i] = D[2i - 1] + D[2i]; \end{array}$$

After mapping, array references and declarations in the program are modified. The result is clustered data storage in the memory. Data storage in clusters is also useful when partial shut-down of in-active parts of the memory is used.

Increasing locality is also possible by making the loops linear [20]. Techniques for making the loops linear are: (1.) loop unrolling; (2.) loop fusion; and (3.) loop fission. Loop-fission breaks down a big loop that does not fit in the cache into smaller loops, in order to increase the locality of the references and decrease the number of memory accesses.

3.3.2.2 Power reduction by re-scheduling instructions

Instruction re-scheduling performed by the compiler changes the order of instruction execution. First, Data Dependency Graph (DDG) and Control Dependency Graph (CDG) of the application source code are obtained. The nodes of these graphs are data and instructions correspondingly. By using the information in the graphs the scheduler reorders the instructions in a way that the

Hamming distance is reduced. The scheduling introduced in [24] is a sample work of scheduling VLIW instructions to reduce the hamming distance between the consecutive instructions.

3.4 Compatibility analysis

Based on the methods characteristics, optimization methods for power reduction and fault tolerance can be combined to form a hybrid optimization providing both. Some methods however can be combined only under certain limitations and are not applicable for all cases. Compatible power reduction and reliability optimization methods are listed below.

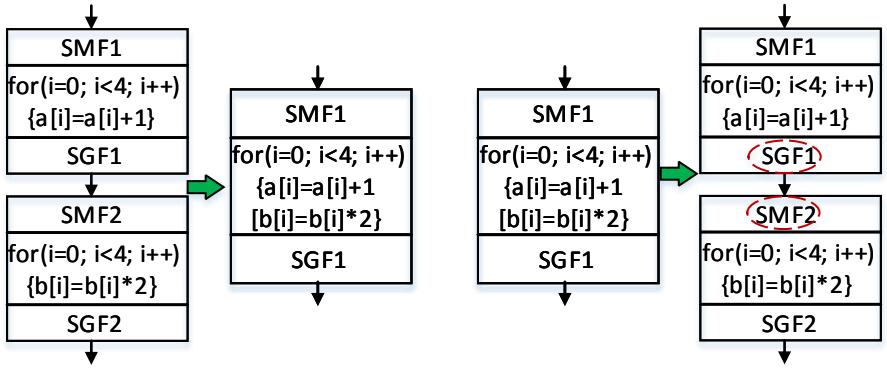
- **Instruction re-scheduling and duplication** Added data-check instructions (*shadow* and *compare* instructions) for data error detection can be re-scheduled to reduce the Hamming distance. However, *compare* instructions should be scheduled before critical points of the program, such as "*store*" and "*branch*" instructions.

- **Loop flattening (unrolling or fusing) and signature monitoring:** Signature-monitoring schemes add extra SGF and SMF, which has an overhead in terms of power and performance. By loop-unrolling and -fusing the number of basic blocks is reduced. Thus, the overhead of added SGF and SMF will be reduced.

  Loop unrolling decreases the number of iterations of a loop, thus the number of times SGFs and SMFs are executed also decreases. Loop fusion combines a number of smaller loops in a larger loop, which decreases the number of required SGFs and SMFs. This is depicted in Figure 3.5(a).

Contradictory reliability optimization and power reduction methods are:

- **Loop-fission and signature monitoring:** In processors with small cache sizes, loop-fission divides bigger loops than the cache size into smaller loops. However, by breaking a loop into several smaller loops, additional SGFs and SMFs are needed for each new loop. If SGFs and SMFs have memory references, including extra SGFs and SMFs causes extra memory accesses. This issue plus the execution of SGF and SMF instructions cause an extra overhead in terms of power consumption.



(a) Loop Fusion and signature monitoring assertions

(b) Loop Fission and signature monitoring assertions

Figure 3.5: (a)Loop fusion reduces the number of SGFs and SMFs, (b)Loop fission adds extra SMF and SGF

However, this is not an issue if the cache is larger and loops are not divided. In the discussed related works of signature monitoring, the SGF in methods such as CFCSS, ECCA and YACCA may have reference to the memory, depending on whether the additional parameters (P_l) are stored in memory locations or in the registers. This is an implementation issue and is decided by the compiler. Figure 3.5(b) depicts an example of a loop fission and signature monitoring scheme (YACCA) presented in [3]. In this case after loop fission, extra SGF_1 and SMF_2 assertions corresponding to the newly generated basic block cause extra overhead. Moreover, the static parameters produced at compile time (introduced in Table 2.1 for YACCA [3]), cause additional overheads.

- **Instruction re-scheduling and signature monitoring:** Scheduling the instructions to reduce dynamic power consumption may change the place of instructions based on the Hamming distance between the consecutive binary form of instructions. In signature-monitoring methods, special instructions for run-time signature generation (in SGF) and run-time signature monitoring (in SMF) are added in specific points of the basic blocks and are typically independent from the rest of the basic block code.

  In discussed signature monitoring schemes, SGF and SMF are embedded either into the source code or into the assembly code. However instruction re-scheduling for reducing the Hamming distance is done after

the assembler and linker when the binary representation of the instructions has been defined. Thus, if one aims to add control-flow checking by the presented signature monitoring schemes and reduce power consumption by scheduling the instructions, signature-monitoring schemes are applied first and scheduling later. However, by instruction re-scheduling, the added SGF and SMF in earlier phases may move to an arbitrary point in the basic blocks. Therefore, instruction re-scheduling can impact the control-flow checking. A possible solution to this limitation is that the signature-monitoring scheme is applied at later phases such as the proposal in [11]. Another solution is to restrict signature-monitoring instructions, and prohibit the scheduler to reschedule them. However, this is still a limitation and rescheduling can not provide the same power reduction that it should.

- **Instruction re-scheduling and masking:** The added mask instructions are placed at specific points in the basic blocks by the compiler, therefore for the same reason mentioned for instruction re-scheduling and signature-monitoring, re-scheduling may corrupt the effect of masking instructions. This is the same case as instruction re-scheduling and signature monitoring. It can be solved with some restrictions to the scheduler at the expense of power consumption.

3.5 Conclusions

In this Chapter we have categorized power optimization methods and analyzed their compatibility with reliability optimization methods. Power optimization methods were divided into scheduling, coding, increasing locality, partial shutdown of system and dynamic voltage scaling. Among the studied power reduction methods, software techniques which include power reduction by increasing locality and power reduction by re-scheduling instructions are most suitable for embedded systems with no special hardware support. Each of the optimization techniques from both classes were analyzed in terms of overheads, abstraction level and implementation issues. Based on our analysis, three pairs have also been identified with certain limitations in place: Loop-fission for power reduction and signature monitoring, Instruction re-scheduling and signature monitoring and Instruction re-scheduling and masking. On the other hand, two promising combinations were identified that can be used in embedded systems requiring reliability with limited power budget. More precisely they are: instruction re-scheduling with instruction duplication and loop

flattening (unrolling or fusing) with signature monitoring. These methods can be implemented both by a specialized compiler and do not require any additional hardware support. However, in order to benefit from signature monitoring methods and loop flattening techniques in a program, signature monitoring methods require to be flexible and adjust the assertion locations based on the program's CFG. Previously proposed signature monitoring methods do not have this flexibility. In the next Chapter our proposed method is addressing this problem.

The content of this Chapter is based on the following paper:

Ghazaleh Nazarian, Christos Strydis, Georgi N. Gaydadjiev. **Compatibility Study of Compile-Time Optimizations for Power and Reliability.** Proceedings of the 14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD), pp. 809-813, Oulu, Finland, August-September, 2011

4

Low overhead control flow fault detection

The experiments of heavy-ion fault injection to programs by Karlsson et al. show that more than half of the injected faults cause control-flow faults [16]. Taking this into consideration together with the distractive impact of erroneous control flow execution, in this Chapter we focus on control flow errors and signature monitoring schemes to detect such errors. Fault tolerant hardware techniques usage is limited, due to the increased design costs. As an alternative, Signature Monitoring (SM) schemes using assertions (implemented as compiler optimization) are an efficient alternative method for control-flow-error detection instead of hardware implemented ones. Among the proposed SM techniques discussed in Chapter 2, no method employs the program-specific knowledge to reduce the imposed overheads caused by the extra added assertions in the Control-Flow-Graph (CFG). In this Chapter we propose a novel SM technique based on CFG analysis at compile time aiming at minimizing the number of assertions by placing those only at optimal locations of the assembly code. Moreover, it is demonstrated how this new technique based on software instrumentation can benefit from loop-unrolling, with significant impact on control-flow reliability. We show the impact of loop-unrolling on fault-coverage and performance of this scheme and compare it against the obtained results from conventional methods. Thanks to the proposed approach, significant fault-coverage concerning CFE can be obtained with reduced assertion costs. As a result of having assertions with less overheads, the instrumented program runs faster than the programs instrumented based on previous methods. We evaluate our proposal using ImpBench [42] benchmark suit and show more than 50% reduction of performance, memory and power-consumption overheads while providing comparable fault coverage as the state-of-the-art SM methods. In comparison with the most-efficient current SM method, our

technique has an average of 13% fault coverage improvement.

4.1 Introduction

The targets of this work are safety-critical systems with high performance requirements and extremely limited energy budgets. A good example of the systems with the aforementioned requirements is biomedical implants, such as Cochlear implants. Cochlear implants are powered by batteries and therefore should be able to operate with limited power budget and at the same time they require to be reliable and meet the necessary performance rate. In such systems, both optimizations for improving performance and reliability are required. Moreover, power consumption should be kept low. The challenge which remains is to understand which error detection/recovery method is efficient to be used with performance-oriented optimizations, in order to provide high performance and reliability with minimal increase of energy consumption. From reliability-optimization point of view, this challenge is satisfied when employing the reliability optimization together with a performance-oriented optimization will not degrade the fault-coverage. There are many compiler-based optimizations for increasing performance in conventional and HPC processors [6]. Among these techniques loop-unrolling seem applicable to embedded systems too. In this Chapter we investigate the impact of loop-unrolling on reliability optimization methods and introduce a novel method that is compatible with loop-unrolling.

The authors of [33], have investigated the impact of compiler optimizations such as loop-unrolling on the fault-recovery ability of ACCE [47]. ACCE is a recovery method which uses CEDA [48] for CFE detection. The result of this investigation shows that several compiler optimizations can increase the fault recovery rate. However, it is concluded that there is no specific optimization that can increase ACCE fault coverage and it is the structure of the workloads which influences how optimizations impact the recovery rate. As described in Chapter 2, existing signature monitoring schemes are divided into two main categories; path-asserting and predecessor/successor-asserting methods. The conventional methods belonging to these categories (such as CEDA, CCA as predecessor/successor-asserting methods and ACFC as path-asserting method) do not instrument the program based on the CFG topology and the program structure. Therefore, the impact of CFG topology change on the fault coverage and performance overhead, after optimizations such as loop-unrolling, is not explored and we would like to look at it in this Chapter.

Loop unrolling may change CFG topology, the number of basic blocks and their respective sizes. As described above, among existing signature-monitoring schemes there is no method which employs compile-time information CFG in order to eliminate unnecessary assertions and reduce overheads while preserving a high fault coverage. In this Chapter a novel SM method for CFE detection is proposed which instruments the assembly code based on CFG analysis performed at compile time. Our method is named as Selective-Control-Flow-Check (SCFC). Depending on the particular CFG topology an optimal assertion is selected for each basic block of the program. SCFC inserts minimal number of assertions in only critical points of the code aiming at low overheads while preserving high fault coverage levels. Since SCFC analyzes the CFG topology to add assertions at the specific program locations, loop-unrolling can be used before SCFC optimization to additionally improve performance while preserving high fault coverage.

Among the CFE detection methods we investigate the impact of loop unrolling on CCA [19] and our proposed method (SCFC). CCA is chosen as the representative of high-coverage CFE detection methods in predecessor/successor-based category due to its simple *set* and *test* assertions and higher coverage compared to other methods in the category. Since methods similar to CCA do not analyze the CFG topology before adding assertions, most compiler optimizations (which are mainly loop-related) go after either fault-coverage or performance. Contrary to these methods, since SCFC instrumentation is done with the knowledge of the CFG topology, it is expected that using SCFC together with compiler optimizations that change this topology (like loop-unrolling) will not degrade the fault coverage. In this Chapter we show that a traditional compiler technique like loop-unrolling can improve SCFC performance while sustaining high fault coverage.

SCFC is evaluated using the ImpBench benchmark suit [42] common to biomedical implants. In addition, three synthetic control-flow driven workloads, each demonstrating a particular CFG topology, are used in our evaluation. Results of the synthetic workloads are presented to emphasize the highest overheads that each method under study may cause. In our experiments we use an embedded 32-bit RISC processor, and the CoSy framework [1] for developing an optimized compiler for this processor. However, since our method is modifying only the intermediate representation of the source code, without loss of generality our optimizations can be re-targeted to any arbitrary processor by simply using its corresponding back-end. Workload binaries generated by our customized compiler are evaluated using Synopsys Processor Designer simulator [2]. SCFC results are compared to two state of the art SM techniques,

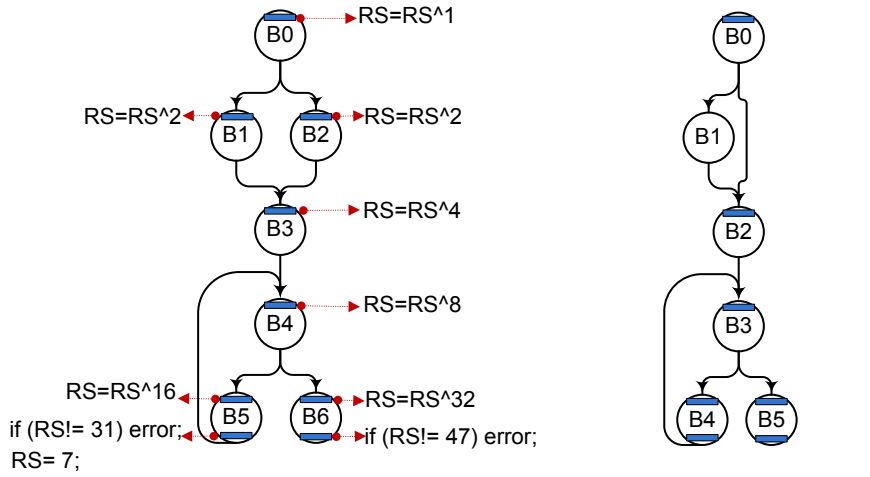
one with lowest overheads (ACFC) and one with highest fault coverage (CCA). While showing a high fault coverage level in pair with CCA, the overheads of our method are close to those of ACFC. The main contributions of the work presented in this Chapter are:

- A novel software only method, not requiring extra hardware provisions and hence widely applicable;
- A workload CFG structure aware method to reduce assertion overhead for each basic block while keeping the fault coverage high;
- On average 13% fault coverage improvement over ACFC with only 2.5% performance degradation and about 2% power consumption overhead respectively;
- Study of the impact of loop unrolling on performance-improvements and fault coverage (additional 9.75% fault coverage on average).

The reminder of this Chapter is organized as follows: Section 4.2 discusses two methods from each SM category with an illustrative example. In Section 4.3 the targeted fault types and their causes are introduced. Section 4.4 describes SCFC and details the improvements over the previous works. In Section 4.5 the impact of loop unrolling on SCFC and CCA is investigated and compared. In Section 4.6 the workloads used for experiments, framework for fault-injection and the results of our evaluation are explained and finally the conclusion of the work is given in Section 4.7.

4.2 Setting up a challenging baseline for comparison

In this Section, we describe and compare two different methods from the categories presented in Section 3.2 more precisely. ACFC (as the best performing path-asserting method example) and CCA (as a representative for predecessor/successor-asserting methods) are presented. These methods are used as building components of the proposed SCFC. We selected the path assertion method with the lowest overhead and the predecessor/successor assertion method with the highest fault coverage to create a worst-case baseline for comparison against our method proposed in this Chapter.



(a) Sym. CFG with ACFC assertions (b) Asym. CFG

Figure 4.1: Asymmetric CFG and Symmetric CFG with ACFC assertions

4.2.1 Path assertion method with the minimal overhead

In ACFC [50] one *set* assertion is used at the beginning of each basic block, and a *test* assertion is used at the end of each possible control-flow path. In general, different control-flow paths are created in the CFG as a consequence of conditional branches and/or loop statements in the program. Each control-flow path has its own RS and each basic block in that control-flow path corresponds to one bit of the RS. The *set* assertion in each basic block sets one bit of the RS which corresponds to the basic block, using bitwise XOR operation between RS and the MASK of the basic block. The MASK of the basic block is unsigned integer value representation of RS when the corresponding bit to the basic block is ”1” and all other bits are ”0”. The *test* assertion compares the runtime signature value with the corresponding constant value to the control-flow path. The constant of the control-flow path is unsigned integer value representation of RS when, all the corresponding bits to basic blocks in the control-flow path are ”1” and other bits of RS are ”0”. In case an error is detected, the routine for error handling is called. Since the *set* assertion in the beginning of the basic block raises the corresponding RS bit using bitwise XOR, if the basic block is executed more than once, the corresponding bit of the basic block in RS is reset. To overcome this problem, ACFC suggests to test and restore RS in the last basic block of each loop. RS is restored with the constant of the control flow path before the next iteration of the loop.

Figure 4.1 shows the ACFC scheme applied on an example simple CFG topology. In the depicted CFG, the possible paths, without considering the loop {B4, B5}, are {B0, B1, B3, B6} and {B0, B2, B3, B6}. ACFC sets the corresponding bits of RS in each basic block. Finally, in the last basic block of the path, which is B6, RS content is compared with constant of the path which is 47 (101111'b). It should be noted that B1 and B2 are multi-path basic blocks that show up in the CFG when there is an if-else statement in the program. ACFC sets the same bit of the RS in multi-path basic blocks (second bit in Figure 4.1), because at runtime only one of the basic blocks is executed. The path in the loop is {B4, B5} which correspond to 4th and 5th bits of the RS. In the last basic block of the loop (B5), RS is compared with 31(11111'b) and it is restored with the value of control-flow path constant before entering the loop, which is 7 (111'b) in this example.

ACFC assumes only for symmetric CFG topologies. The resulting inefficiency of this method for asymmetric topologies (typical for the majority of real-life workloads) is explained with an example. Figure 4.1.b shows an asymmetric CFG in which B1 may or may not be executed at runtime. For this reason, ACFC can not assert B1 execution by *setting* one bit of RS in B1 and *testing* RS content at the end of the path. Therefore, B1 is left without any checks. For programs with *if* statements without an *else* part the resultant topology does not have symmetric multi-path basic block as depicted in Figure 4.1.a. To cope with this problem, the authors of ACFC add a dummy else statement for making the CFG symmetric. It should be noted that, this solution increases the overheads due to the extra code and branches imposed by the dummy else CFG nodes. Moreover, ACFC instruments the source code of the program based on the source-code-level knowledge. However the CFG of the program is not final at this phase and it may change when the corresponding assembly code is generated. For majority of the cases the final CFG topology of the code is determined in the last phases of compilation.

Another shortcoming of ACFC is in loop constructs. At the end of each loop, RS has to be restored with the constant value of the control-flow path before entering the loop again. As a consequence a CFE from a basic block before the loop to the *restore* statement at the end of the loop can not be detected. As an example in Figure 4.1.a the faulty jump from B0 to the last statement of B5 ($RS = 7$) can not be detected using the ACFC *test* assertion.

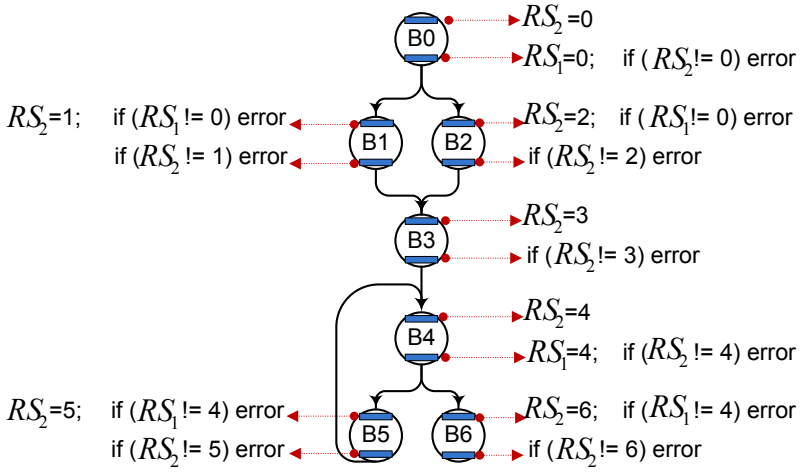


Figure 4.2: Instruction-level CCA assertions

4.2.2 Predecessor/successor method with the highest reliability

CCA [19] is a predecessor/successor-asserting method which has the highest fault coverage as a result of adding the highest number of assertions. It asserts correct control-flow by checking the predecessor of each basic block. In order to do so, this method uses a *set* and *test* assertion pairs. The *set* assertion applies the runtime signature (RS_1) at the end of the predecessor block and the *test* assertion checks RS_1 against the expected value which is the signature of the predecessor block. Moreover, it guards complete execution of each basic block in the absence of faulty jumps to/from middle of the block. For this purpose, a pair of *set* and *test* assertions are used which set and test another runtime signature (RS_2). Figure 4.2 depicts the same CFG instrumented with CCA assertions. A pair of set RS_2 (at the beginning of the basic block) and test RS_2 (in the end of the basic block), asserts complete basic block execution without interruption due to faulty jumps to/from mid of the basic block. A pair of set RS_1 (in the end of the predecessor basic block) and test RS_1 (at the beginning of the current basic block) checks if the previous executed basic block is the valid predecessor of the current block or not. In case inconsistencies are detected the error recovery routine will be called. CCA inserts four assertions per basic block which cause a significant overhead in terms of performance and memory. Another shortcoming of CCA is its limitation in detecting faults in basic block with multiple predecessors. Basic blocks with multiple predecessors can not be guarded with the first pair of assertion ($set/test\ RS_1$). In the depicted CFG, B3 and B4 are not guarded with RS_1 checking.

4.3 Fault model

Targeted errors of the proposed method are NonBranch-To-Branch and Branch-Target-Change as explained in Section 2.3.2. In NonBranch-To-Branch faults, a non-branch instruction is converted to a branch instruction. As a consequence the basic block Section after the faulty jump is not executed and the execution flow is diverged from its correct sequence. Branch-Target-Change causes an erroneous jump from one basic block to a faulty destination, whose corresponding edge to the jump does not exist in the program CFG (illegal branch).

4.4 Selective Control Flow Check (SCFC) method

The Selective Control Flow Check method presented here is a hybrid method which uses both path assertions and predecessor/successor assertions. Compile-time CFG analysis information is used to add low-cost *test* assertions to the last basic block of the identified control-flow paths, including the paths resulted from conditional statements. Basic blocks that are not part of a control-flow path (lonely-blocks) are instrumented by a standard *predecessor-test* assertion.

Considering high number of assertions in predecessor/successor-asserting methods on one hand and the problem of path-asserting methods with asymmetric CFG topologies, there is a need for a SM technique with reasonable number of assertions which is applicable to realistic CFG topologies. To achieve the latter, the optimizations should be applied at the point where (in terms of compiler passes) the final CFG topology is available. For this reason our SCFC optimizes the back-end intermediate code which is the closest version of the code to its executable binary. SCFC is implemented using the CoSy framework; a modular framework for compiler development which enables compiler optimizations at different levels of abstraction. In this Section, first the development framework is explained followed by the SCFC description and a motivational example of CFG which is instrumented with the proposed SCFC assertions.

4.4.1 Experimental framework for compile-time optimizations

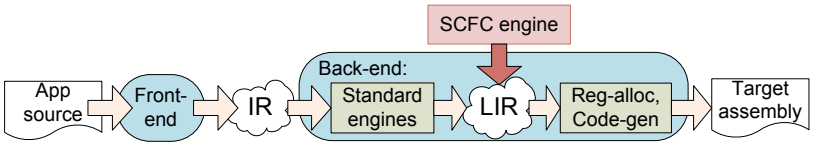


Figure 4.3: The CoSy framework

In order to implement and validate the above mentioned optimizations, the CoSy framework [1] was selected. CoSy is a modular framework specially developed for simplifying compiler design and optimization. Figure 4.3 is a block diagram showing this framework. In CoSy, modules (so called engines) are responsible to carry out different tasks of the compilation process, e.g, register-allocation, scheduling, etc. Each module may carry out only one task or a reduced set of tasks depending on the compiler-designer preferences. Additional optimizations can also be implemented as a new engine and easily included in the compiler. Our SCFC optimization is also implemented in such a module (engine). In the compiler generated by CoSy, similar to other compilers, first the front-end generates an Intermediate Representation (IR) of the program. After IR is processed by group of engines which work at IR-level, the back-end of the compiler transforms the final IR description to Lower-Intermediate-Representation (LIR). LIR is the closer version of IR to the assembly code in which the final CFG topology is final. Also in LIR more detail information that are not present in IR, such as register allocation, is known. SCFC engine manipulates LIR which has the final CFG topology. Moreover, since our SCFC engine is added after the scheduler engine, extra assertions inserted in the begin/end of basic blocks will not be relocated.

4.4.2 Detailed description of the SCFC method

The SCFC analysis of program's CFG identifies the paths (sequence of two or more basic blocks which should be executed sequentially) and the lonely-blocks[1] and stores this information. SCFC processes the CFG of the application program and determines all basic blocks in loops, basic blocks in different control-flow paths and lonely blocks. SCFC then inserts two types of assertions guided by the results of CFG analysis. Basic blocks residing in paths are augmented with path-assertions and lonely blocks with predecessor/successor-assertions. Figure 4.4 depicts the flowchart of the proposed algorithm for CFG

[1] basic blocks that can not be grouped in any path

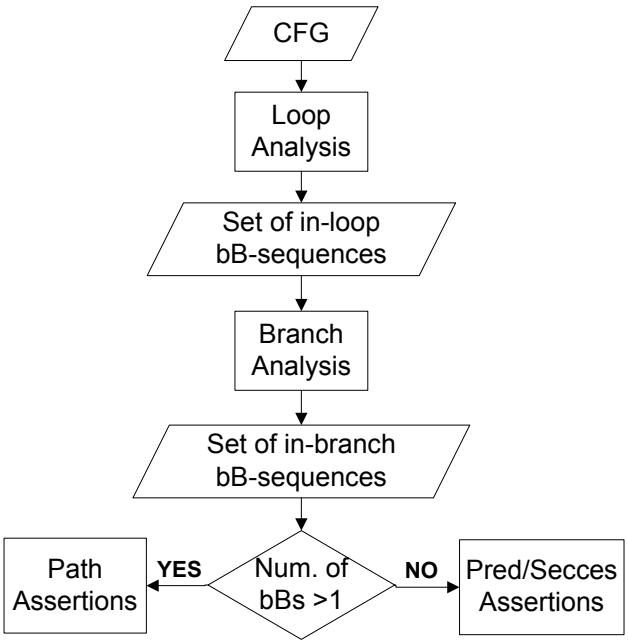


Figure 4.4: CFG processing and SCFC instrumentation

processing and instrumentation of basic blocks. The algorithm consists of three steps; (1) loop analysis, (2) branch analysis and (3) adding assertions. At the first step of CFG processing, basic blocks residing in each loop are extracted and saved in separate sequences and basic blocks which are not taking part on any loop are all saved in another sequence. At the second step of CFG processing, the sequences resulting from the first step are processed to extract basic blocks which are in different paths of the control flow due to conditional branches. At the third step, the resulting basic block sequences of the second step of CFG processing are used to decide which type of assertions should be added to each basic block. *Path-assertion is applied for basic blocks in sequences with two or more basic blocks. Predecessor/successor-assertions are used for all lonely-blocks.*

In order to instrument basic blocks in a path using path assertions, corresponding path-*set*-assertions are added in the beginning of all basic block in the path and only one path-*test*-assertion is added at the end of the last basic block of the control-flow path to check the correct flow of blocks in sequence. Predecessor/successor-*set*-assertions are added to the predecessor basic block of a lonely-block and the corresponding *test* assertions are added at the lonely-blocks. The two categories of *set*/*test* SCFC assertions are explained below.

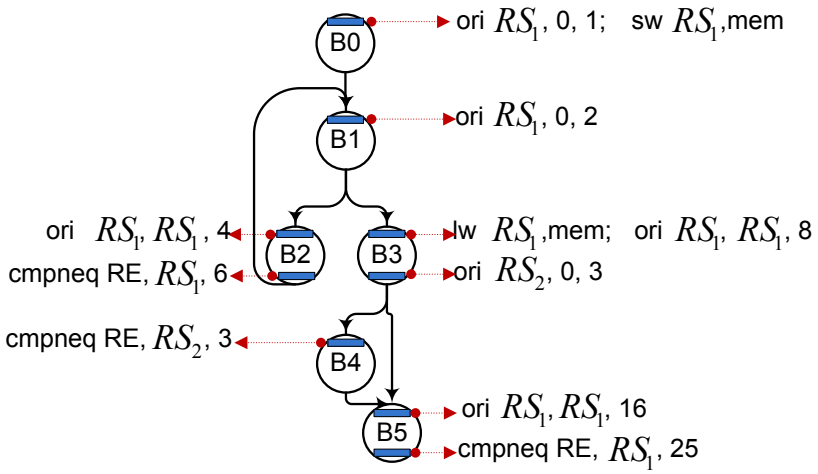


Figure 4.5: CFG with proposed hybrid optimization

1. Path-assertions: basic blocks residing in control-flow paths are instrumented with low-cost path-based assertions, meaning that each basic block in the path has a *set* assertion while the *test* assertion is added only to the very last basic block of the path. This *set* assertion is an OR instruction with an immediate representing the basic block MASK (ori RS_1, RS_1, MASK), which updates path-runtime-signature contents (RS_1). The *test* assertion is *cmpneq* instruction to compare the contents of RS_1 with a constant value that is the signature of the path (cmpneq RE, RS_1, CONST). RE is a restricted register that holds the results of fault detection. If RS_1 is not equal to the CONST, RE is written. The fault recovery routine will be invoked if RE holds a non-zero value;

2. Predecessor/successor-assertions: lonely blocks are guarded with *predecessor-test*, meaning that a *predecessor-set* assertion is added to the predecessor of the lonely-block and a *predecessor-test* assertion is added to the lonely-block. This *set* assertion is also an OR instruction, but it updates the contents of the predecessor-runtime-signature (RS_2) to the signature of the predecessor basic block ($oriRS_2, 0, Sig_{pre}$). The *test* assertion is an instruction comparing RS_2 contents with the predecessor signature ($cmpneqRE, RS_2, Sig_{pre}$). If there is an inconsistency RE is written with a non-zero value.

Path-assertions usage is limited to sequences with at least two basic blocks, because if they are applied to lonely-blocks, two assertions are used in a single basic block which has the same code-size overhead as predecessor/successor-

assertions. While predecessor/successor-assertions use one runtime signature for all basic blocks of the routine, path-assertions occupy one bit of the runtime signature per basic block until the end of the routine, therefore depending on the number of basic blocks we may need more than one runtime signature. Due to this fact, the usage of path assertions is limited to basic blocks that are residing in a sequence with the size of at least two basic blocks.

Figure 4.5 shows a subgraph from the checksum benchmark [42] CFG instrumented with SCFC assertions. The results of the first step of CFG processing in this subgraph are the basic block sequences:
{B1, B2}
{B0, B3, B4, B5}
Considering the same sub-graph (Figure 4.5) and the resulting sequences shown above, the second step of CFG processing generates set of basic block sequences in different control flow paths as follows:
{B1, B2}
{B0, B3, B5}
{B4}
In the above presented set of basic block sequences of CFG in Figure 4.5 B4 is a lonely-block and is instrumented with intra-block predecessor/successor-assertions. The intra basic block predecessor/successor-assertions for B4 are a set assertion (ori RS_2, 0, 3) at the end of the predecessor block (B3) and a test (cmpneq RE, RS_2, 3) in the begin of B4. Basic blocks in sequences {B1, B2} and {B0, B3, B5} are augmented with path-assertions. Path-asserting *set*s are added in each basic block of these two paths and path-asserting *test* is added only in the last basic blocks of the two paths; B2 and B5. {B0, B3, B5} is the main control-flow path and {B1, B2} is a loop path inside the main path. Execution flow at B1 can continue to B3 in the main path or can enter into the loop-path. Therefore before the execution flow reaches to this point the content of path-asserting runtime signature (RS_1) should be stored and retrieved in the next basic block after the loop path. The *sw* RS_1,*mem* instruction in B0 stores RS_1 content to memory and *lw* RS_1,*mem* retrieve RS_1 contents after the loop. RS store and retrieve is needed for while-loop and nested-if-statement structures. This is due to the fact that these statements can cause paths that may or may not be accessed during the execution flow.

Advantages: (1) Due to the fact that SCFC instruments the intermediate code at the point the back-end is ready to generate the program binary, any interference with other compiler optimizations is avoided and hence is more precise than ACFC and CCA which instrument the program high-level source-code. (2) Compared to CCA, SCFC reduces the total number of assertions in each

program based on the program-specific CFG. For instance, for the depicted CFG in Figure 4.5, CCA adds 17 extra statements for assertions while SCFC uses only 9 extra instructions. (3) Compared to the ACFC method, SCFC has the advantage of eliminating the need for *restoring* RS at the end of the loops. This fact causes higher fault coverage in our method and eliminates some of the extra overheads. (4) Our method can instrument a wide range of arbitrary programs with symmetric or asymmetric CFG topologies while ACFC is efficient only in programs with symmetric topologies. For example in the CFG presented in Figure 4.5, ACFC adds 8 extra statements (almost equal to SCFC) for *set*, *test* and storing of RS, while it is still not able to check the correct execution of B4. This is due to the fact that SCFC adds *test* also to the conditional paths (as opposed to ACFC).

Compared to ACS, SCFC has two differences: 1) it defines finer grain paths; and 2) it guards lonely-blocks with *predecessor-test* assertions. SCFC defines the paths as conditional branches and the for-loop bodies while ACS defines regions that are protected with path-assertions. As a result, it provides higher fault coverage. The high level of fault coverage makes SCFC more suitable for safety-critical systems.

In the current work, the result of the comparison in *test* assertion is written into a restricted register which only the *test* assertion can access. The fault-injection/detection framework used in this work checks the value of the restricted register for diagnosing fault occurrence. In real-world implementation of our SCFC technique, in case of a fault, a dedicated fault-recovery routine can be invoked. This fault recovery routine can be a conventional fault recovery method using checkpoints.

4.5 The impact of loop unrolling on SCFC and CCA

Loop unrolling may change the CFG topology, the basic blocks overall number and their corresponding sizes. The effect of unrolling loops on the CFG depends on the type of the loop that is unrolled: *while* loop or a *for* loop; a nested loop or there is a conditional statement in the loop-body that can terminate the loop earlier. Figure 4.6 shows two different CFGs before and after unroll. Unrolling simple for-loops, without conditional-statements or other loop-constructs in the body, does not change the CFG topology as depicted in Figure 4.6(a). In this case, the unrolled CFG has the same basic blocks numbers and only the number of instructions in the unrolled basic block (B2 in the Figure) is increased. On the other hand, the CFG of for-loops with conditional-

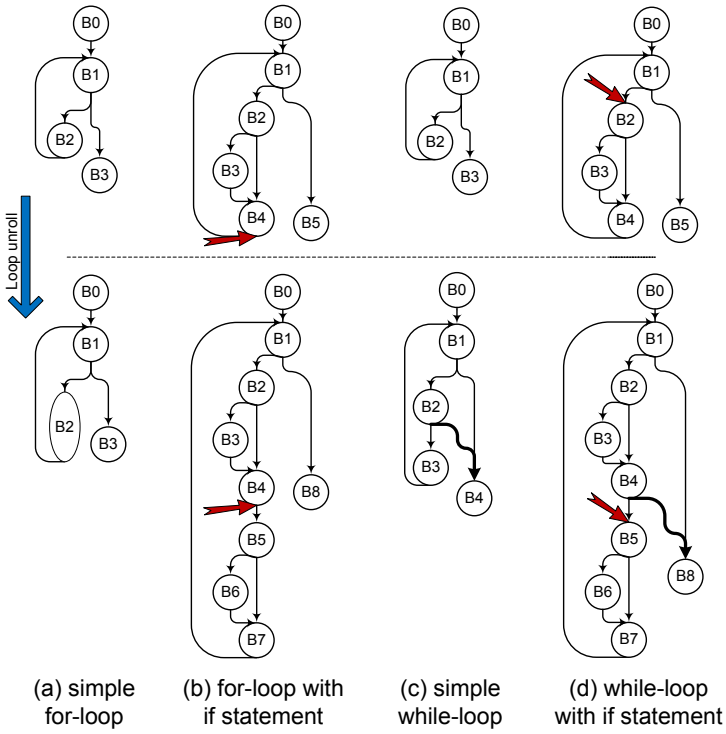


Figure 4.6: Impact of for-loop unrolling on the CFG

statements (or a loop-constructs) in the body, changes after unrolling. Figure 4.6(b) shows how the CFG of such a loop changes after unrolling with increased number of basic blocks. Unrolling while-loops requires checking the loop-condition before loop body repetition and *break* in case the condition does not hold. The *break* statements cause extra branches in the unrolled CFG (as shown with **bold arrows** in Figure 4.6). Figure 4.6(c) shows that, opposed to for-loops, unrolling even a simple while-loop changes the CFG and the total basic blocks number.

Change of CFG after loop-unrolling affects CFE detection fault coverage. As discussed above CCA is weak in fault detection of basic blocks with multiple predecessors. Therefore, if the resultant CFG after unroll has bigger number of such basic blocks, CCA fault coverage may decrease. In Figure 4.6(b) and (d), it is shown that unrolling loops with a conditional statement in the body, adds the number of basic blocks with multiple predecessors in the resultant CFG. This is the case also in nested loops, when the outer-most loop is unrolled. In these cases CCA fault coverage decreases. However, since SCFC analyzes the newly formed CFG after unrolling and groups most of the

multiple-predecessor basic blocks to the new set of control-flow paths, it is expected that SCFC fault coverage will not decrease after loop unrolling in CFGs such as the ones given in 4.6(b) and 4.6(d).

SCFC has a higher fault-coverage when CFG has a bigger control-flow path in the loops than smaller ones. Since SCFC resets the path-runtime-signature in the first block of the control-flow path (RS_1 in Figure 4.5), having a bigger loop-control-flow path with lower number of loop-iterations helps to detect bigger number of erroneous branches to the loop-control-flow path. As an example, the control-flow path of the loop in Figure 4.6(b), before unrolling is {B1, B2, B4}. After unrolling this control-flow path expands to {B1, B2, B4, B5, B7}. An erroneous branch to the end of B4 (depicted by an arrow in Figure 4.6(b)), is not detectable in the loop-control-flow path before unrolling ({B1, B2, B4}). This is due to the fact that the path-runtime-signature (RS_1), is reset at the beginning of B1 and the error is masked. However, SCFC detects this erroneous branch in the loop-control-flow path after unrolling ({B1, B2, B4, B5, B7}).

Contrary to for-loops, unrolling while-loops does not result in bigger loop-control-flow path. SCFC fault coverage in not increased after unrolling while-loops. In Figure 4.6(d), the loop-control-flow path before unrolling has three basic blocks. After unrolling, the resultant loop-control-flow paths are {B1, B2, B4} and {B5, B7}. An erroneous branch similar to the one discussed above, which targets B4, is not detected by SCFC even after unrolling the while-loop. In Figure 4.6(d), an erroneous branch to the begin of B2 (depicted with an arrow) in the second iteration of the loop, is equivalent to an error in B5 at the first iteration of unrolled-loop. The error before unrolling in B2 is detected by SCFC, But, SCFC can not detect the erroneous branch to the begin of B5 after the loop is unrolled.

In our example the loop is unrolled once. Considering fault coverage and performance the number of times loop gets unrolled is not limited and the impact depends on the number of unrolled iterations. On the other hand, loop unrolling increases the code size and in small embedded devices memory capacity can be very limited. The number of times the loop should be unrolled depends on the specific application and the system under consideration, for tiny embedded systems with limited memory capacities the number of unrolled iterations is bounded by the available memory sizes. Therefore, in practice the loop unrolling technique should be a trade off between fault-coverage and performance against the introduced memory overhead.

4.6 Experimental results and analysis

In this Section, the proposed optimizations in SCFC, ACFC and CCA are evaluated in terms of performance, code-size and power-consumption overheads in addition to the fault coverage of the method for a set of injected faults. Each of these metrics is compared to a similar metric obtained for the ACFC and CCA methods, as well as our baseline architecture (i.e. without optimizations).

Moreover, in order to investigate the impact of loop-unrolling on signature monitoring schemes, we compare CCA and SCFC fault-coverages and performances after loop-unrolling is applied. For this reason, we have executed four sets of simulations for four versions of the workloads binaries. The different versions of binaries are generated with four versions of optimized compilers: 1) with SCFC but without loop unrolling; 2) with SCFC and with loop unrolling; 3) with CCA but without loop unrolling and 4) with CCA and loop unrolling. For the ACFC we expect the loop unrolling to have more or less similar impact as for CCA and this is why we did not include it in our study below. The generated workload binaries are evaluated using Synopsys Processor Designer cycle-accurate simulator [2]. Each set of simulation run, consists of 1,000 runs with one error injected in each run. The error generation is discussed in detail in the rest of this Section. The obtained fault-coverage results from running the binaries without loop-unrolling, are used as the baseline to investigate the improvement or degradation of CCA and SCFC fault-coverages after loop-unrolling. ACFC, the path-based-asserting CFE detection method, adds low number of assertions and has low fault-coverage. This method does not analyze CFG control-flow paths, the same way as the SCFC does. Therefore the CFG topology refinement after loop-unrolling does not solve the problem of low fault-coverage. Loop unrolling favors both, SCFC and CCA methods in terms of performance, as the level of instruction-level parallelism increases. We investigate how loop unrolling influences SCFC and CCA fault-coverages.

In what follows, the workloads, our experimental setup and the simulation results of our evaluations are presented.

4.6.1 Workloads used in our study

Without loss of generality, our experiments are performed using ImpBench [42], a benchmark suite targeting low-power application source codes in addition to a set of control-flow driven synthetic test programs. ImpBench is a benchmark suite with applications typical for biomedical implants. The

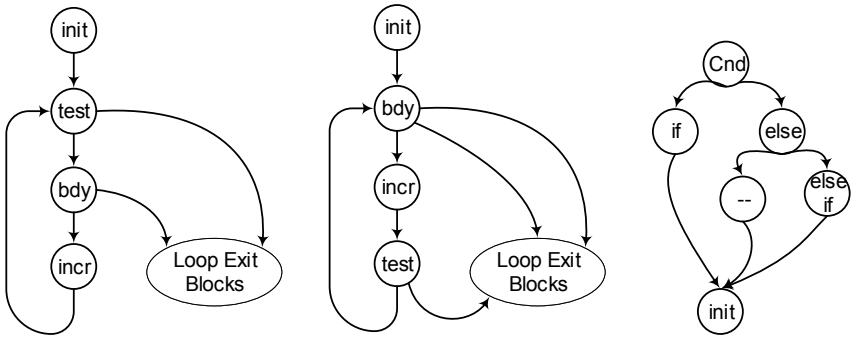


Figure 4.7: Control statements structures

control-flow driven test kernels are constructed in a way to emphasize the highest overheads which the proposed optimization may cause.

Each program in the set of control-flow driven test kernels shows a special case of a CFG topology. The control-flow driven test kernels are designed to be extremely control-flow dominated and to cover a wide set of CFG topologies which are representative of all possible cases which a real CFG may contain. In order to build such test kernels, first we have identified programming statements which cause divergence in the program control flow and cause different topologies in the CFG (so called control statements). We use these statements as building blocks of our synthetic test kernels.

The control statements used are *do-while* and *while-do* loops, *for-loops* and *if-then-else*. To construct CFGs with different topologies, we have to look into different possible combinations of the above mentioned control statements. First, an overview of the CFG structure of each control-flow statement is given. Figure 4.7 depicts CFG structure of these statements as discussed in [4]. As depicted in Figure 4.7, *for-loops* and *while-do loops* have the same CFG-structure. A *for-loop* is a variant of the *while-do loops* which has an initialization statement in the beginning and an increment statement at the end. Due to the similarity of *for-loops* and *while-do* CFG structures, to construct the test kernels, we take into account only for-loops. Therefore, our test kernels contain different mixes of *for-loop*, *do-while* and *if-then-else* statements.

The maximum nesting level in our test kernels is three that is considered as representative. This level represents enough complexity to evaluate our method

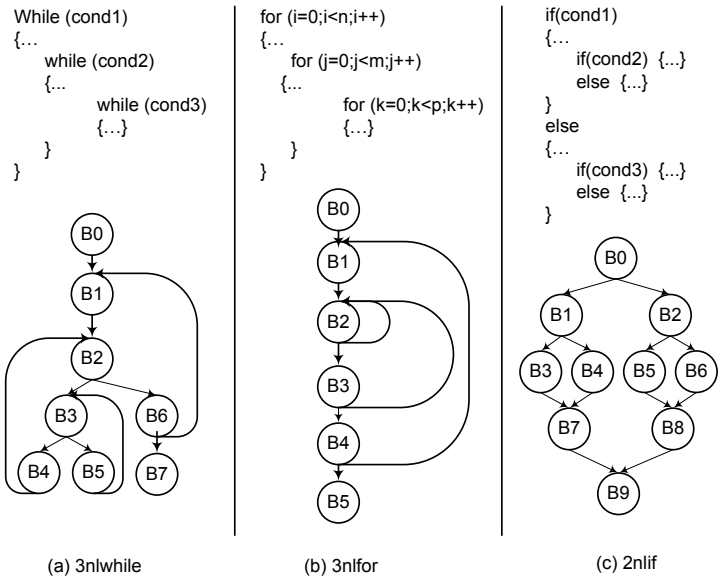


Figure 4.8: Control flow oriented test programs

while it is still simple enough to run 1,000 instances on the simulator (presented next). Taking into consideration the three mentioned control statements of nesting-level of three, a total of 27 combinations (CFG topologies) are possible. Out of these topologies, we build test programs for the three topologies that represents the worst-case scenarios (in terms of number of extra required check assertions), depicted in Figure 4.8. Our test programs are named as *3nlfor*, *3nlwhile* and *2nlif*. *3nlfor* is three nested level of for loops which the body of the loops has a simple addition or subtraction statement. *3nlwhile* is three nested do-while loops with simple body statements. *2nlif* is composed of two nested level if-then-else statements which each if statement has a corresponding else part. As can be seen the number of non-control flow instructions is kept at minimum to emphasize the overhead of the studied techniques.

4.6.2 Experimental setup

In this Section, we briefly describe the target architecture and our implemented fault-injection method which are used for the evaluations.

Target architecture: Our target architecture is a basic, 32-bit, five-stage, in-order RISC processor. This processor is the template processor available in Synopsys Processor Designer simulator. It has no advanced micro-architectural features but has similar load/store-based ISA as any ARM pro-

cessor. The only significant difference is the higher number of registers of ARM processors. Due to higher number of registers in such processors the added instructions for detection and recovery will cause lower register pressure. Therefore, the overhead of our instrumentations in a standard processor (such as ARMv7m) is expected to be lower than what is presented in our results using the Synopsys template processor. Given the fact that our optimization scheme is hardware-agnostic, it is applicable to any target processor.

Fault injection: In our experiments, the fault coverage is measured by, first, injecting a fault using a fault injector instruction and, afterwards, investigating whether the fault has been detected or not. Since our target errors are CFEs, two types of faults which may cause a control-flow error are investigated; NonBranch-To-Branch and Branch-Target-Change faults. In order to evaluate SCFC fault coverage, we have used an fault-injection mechanism that injects these two CFE types. We have emulated these two fault types at runtime using the special fault-injector instruction and a Linear-Feedback-Shift-Register (LFSR) which are implemented in the Synopsys Processor Designer simulator. The LFSR is used to generate a random value which is used as the operand of the targeted instruction.

The fault-injector instruction is designed and included in the processor instruction set architecture to inject an error at a random execution cycle. It should be noted that the faults are injected in different executable codes. Even if the fault was injected at the same execution cycle for the executable binaries under the test, different instructions would be hit in the corresponding executable binaries because of differences in the codes. This means that the fault can not hit the same instruction in the executable binaries under the test. Therefore, we have decided to inject the fault in a random execution cycle. The fault-injector instruction is added in the beginning of the program-under-test along with a random value (generated by RANDOM linux command) as its operand. This random value determines the trigger time of fault injection. It determines the number of execution cycles that should pass before the occurrence of the error. A dedicated flag for error injection is set after the given random number of cycles. When this flag is set, the first branch instruction in the execution path will be corrupted and a CFE is emulated. The random value that is passed as the fault-injector instruction operand is also used as the LFSR seed. The characteristic polynomial of the added LFSR is : $x^{32} + x^{31} + x^{29} + x + 1$ which is used to generate pseudo random numbers [15]. The result of the LFSR is used to randomly flip bits and generate NonBranch-To-Branch and Branch-Target-Change fault types.

For generating NonBranch-To-Branch, the first fetched non-branch instruction after the trigger time is converted to a branch instruction with a random value as its operand. This is done by modifying the corresponding opcode bits of the register between *Fetch* and *Decode* pipeline stages. The random value of the operand is the LFSR register value at that moment. A Branch-Target-Change error is introduced, for the first fetched branch-instruction after the trigger time, by changing the operand bits of the register between *Fetch* and *Decode* to a random value. The random value for branch operand is again provided by the LFSR. Typically a SEU will probably change a single bit of the instruction. We know that we over do bit flipping in this fault injection setup and introduce much more CFEs than reality. In the next Chapter we improve the fault injection mechanism used in our experiments by limiting the number of flipped bits.

In order to measure the fault coverage, one of the registers in the register file is reserved so that only *test* assertions of our optimizations can write to it.e purpose of fault coverage measurement. In real systems, in case of fault detection an appropriate fault recovery is triggered and there is no need of reserving a register. It should be noted that reserving this register is only for th In case a control-flow error is detected by *test* assertions, this register is written. We check the contents of this register in the traces generated by the simulator to determine if a control-flow error has been detected. For each fault type, we run 1,000 simulations for each benchmark in order to get a clear estimation of the fault coverage provided by our optimizations. Note that, for determining the fault coverage, we only run the benchmarks using the smallest data sets, as 1) it is expected that the contribution of the input size with respect to the fault coverage is negligible and 2) evaluating larger inputs significantly increases testing time, which is already quite significant.

4.6.3 Experimental results

The performance, static memory (code-size) and power overheads of the SCFC technique are measured by running the selected benchmarks and the control-flow driven synthetic kernels. The performance overhead is measured by running the benchmark on the simulator and getting the number of cycles required for program completion. In order to report the worst-case overheads of our method and compare them against ACFC and CCA results, we use our control-flow driven test kernels. As explained in Section 4.6.1, each test kernel is a small program in which the body of each control statement contains only one simple operation (addition/subtraction). With this arrangement, the

calculated overheads for the test kernels are approximating the worst-case scenario. Normally, in real applications code there are more than one operation in each control-statement body. Therefore, the overhead of adding the extra *test* and *set* assertions will be lower than the overhead calculated for the control-flow driven test kernels. Performance and static memory overheads of the test kernels and benchmarks are plotted in Figure 4.9 and Figure 4.10, respectively.

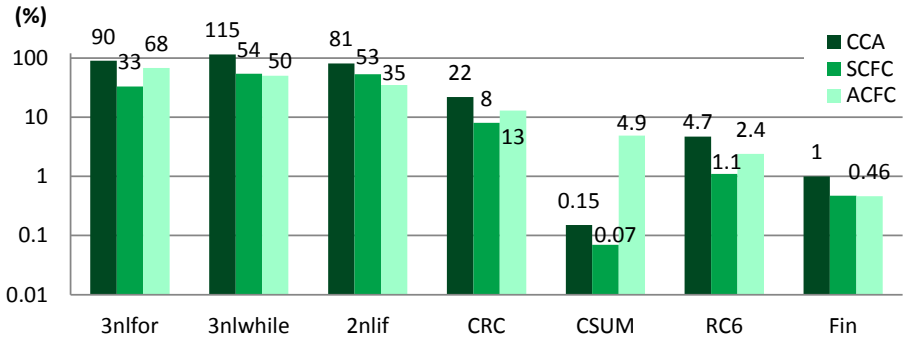


Figure 4.9: Performance overheads

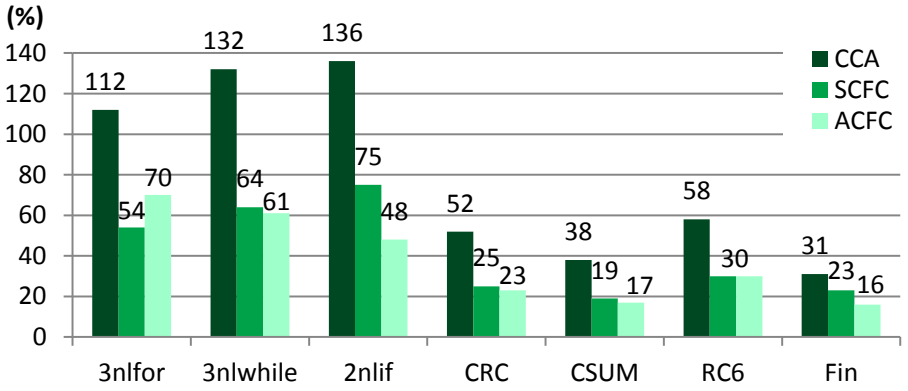


Figure 4.10: Static memory overheads

For power consumption estimation we take the same approach as in [32] which uses the model for calculating power consumption of programs by Tiwari et al. in [44]. In [44] for each instruction category a current-value range is specified. The current value is the measured current drawn from CPU when the corresponding instruction is executed in a loop with a high number of iterations. Instructions with different operands used in the measurement result in having the current range for the instruction. The power-value of each instruction cat-

Instruction category	Current range 0.5-micron (mA)	Power range 0.5-micron (mW)	Power range 90nm (mW)
Arithmetic and logic	172-179	567.6-590.7	272.8-283.9
Load	185-192	610.5-633.6	293.5-304.6
Store	169-175	557.7-577.5	268.1-277.6

Table 4.1: Power model of the ISA

egory is calculated by multiplying supply-voltage with the current estimates ($P = I * V$). The numbers reported in [44] are for 0.5-micron technology with 3.3 V supply-voltage. Table 4.1 shows the estimated numbers for 0.5-micron technology and the scaled down versions for 90-nm technology (with 1.2 V supply-voltage) which is the used technology by our embedded processor. The scale-down factor for power numbers from 0.5-micron to 90nm is calculated based on the dynamic power consumption and transistor capacitance formula in CMOS technology;

$$P_{dyn} = fCV^2$$

$$C = (\epsilon)S/d$$

In the first equation, C is the overall switching capacitance, V is the corresponding supply voltage and f is the operating frequency of the device. In the second formula, ϵ is the dielectric constant of the oxide material, S is the oxide surface (length multiplied by width) and d is the thickness of the oxide. With the assumption that the oxide material in transistors is the same (SiO_2) in the two technologies (ϵ is constant) and similar gate widths the power scale-down factor is simplified as:

$$\alpha = (V^2)(\frac{L_{0.5mic}}{d_{0.5mic}})(\frac{d_{90nm}}{L_{90nm}})$$

With the factors in 0.5micron technology as (V=3.3 V, L=0.5micron, d= 8nm) and in 90nm technology as (V=1.2 V, L=90nm, d=3nm) the scale-down factor is (2.08), which is used to roughly estimate the numbers in the fourth column of Table 4.1 from the numbers of the third column. From the total number of instructions of each category in the program multiplied by the power-value for that instruction category (in 90nm technology) the total power consumed during program execution can be estimated. The power consumption overhead of the three test kernels and benchmarks are plotted in Figure4.11. We are aware that the estimated power numbers are quite rough, but since we evaluate

the differences (the trend between the three methods), we expect that the errors in the estimated numbers cancel each other.

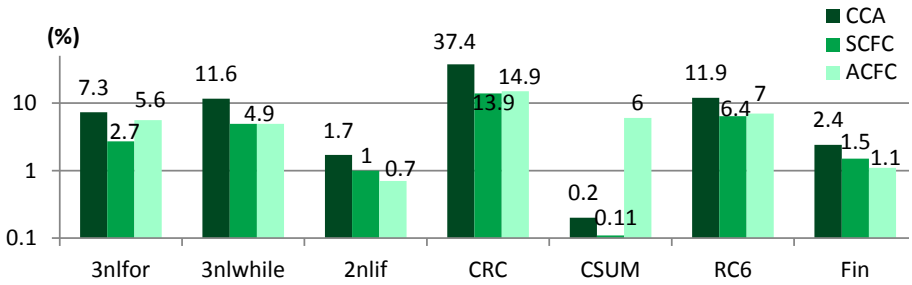


Figure 4.11: Power overheads

With SCFC, overheads are decreased significantly as compared to CCA. When compared to ACFC, our method, has higher performance and memory overheads in *3nlwhile* and *2nlif* synthetic kernels, however in *3nlfor* kernel ACFC shows higher overheads. The reason behind this fact is that SCFC inserts store and load instructions to save and restore RS when the control-flow enters part of CFG which is not in the main path of the execution. This happens only in nested-while-loops and nested-if-statements. As depicted in Figure 4.8.b the control-flow does not diverge from the main path in *3nlfor* kernel, therefore Load/Store instructions are not required for this test kernel. Since ACFC also adds a reset statement to restore the RS value at the end of the loops (in *3nlfor* and *3nlwhile*), the overheads of SCFC is lower than ACFC in *3nlfor*. However ACFC does not add any reset statement for nested-if-statements, which is the reason it has considerable lower overhead than SCFC.

It is important to note that we have designed the *2nlif* kernel to have symmetric CFG topology to facilitate easier ACFC implementation. However, in real workloads this is not always true and ACFC will add dummy *else*s for making the CFG symmetric. The result of adding dummy *else*s is extra code-size, power and performance overheads, with higher effect on performance and power overheads. Since dummy elses cause extra branches, they have a high impact on performance and power consumption overheads. Moreover, knowing the fact that a high percentage of program's time is spent inside loops, in real workloads that have a mix topology the basic control-flow constructs (if-statement, for-loop and do-while loop), SCFC causes overheads comparable to ACFC and in some cases even lower. This fact is proved with the results obtained for the workloads of ImpBench. For all the benchmarks except Fin (a benchmark for compression), SCFC has lower performance and power overheads than ACFC, since no extra *else* statements for symmetrizing CFG as in

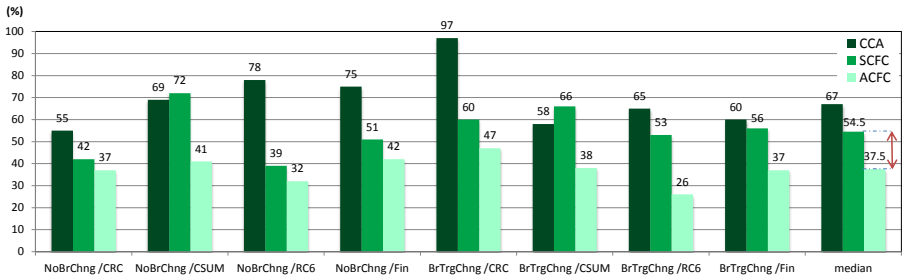


Figure 4.12: Fault coverage comparison between ACFC, CCA and SCFC

ACFC are needed.

Running 1,000 simulations for the two fault types, in the cases of ACFC, CCA and SCFC produces the fault coverage results as plotted in the chart in Figure 4.12. In the plot each group of bars shows the results of injecting one of the fault types for a benchmark. As depicted in this plot, SCFC has a higher fault coverage for fault type Branch-Target-Change compared to NonBranch-To-Branch fault type. This is due to the fact that NonBranch-To-Branch faults, changes a non-branch instruction to a branch instruction with a random destination. Thus, many of the faults from this type may happen in the beginning of a basic block with a faulty branch destination to the end of the same basic block. This leads to skipping some of the instructions and do not cause erroneous control-flow in the rest of the program. Bars showing the fault coverage of the two fault types for the Checksum benchmark show higher coverage for SCFC as compared to CCA. This is due to the fact that Checksum (CSUM) contains two loops and a significant portion of the execution time is spent in these loops. Therefore, a high number of faults are injected in the basic blocks in the loops. The loops contain basic blocks with multiple predecessors, which does not have the predecessor-test from CCA. Thus, the number of faults which cause as erroneous jump to these basic blocks will be undetected by CCA.

The impact of loop unrolling on SCFC and CCA fault-coverage is illustrated in the diagram of Figure 4.13. This diagram shows fault-coverage improvement or loss due to loop unrolling for the two error types under study (Branch-Target-Change and NonBranch-To-Branch). On average, fault-coverage of SCFC is improved by 9.75%, while CCA fault coverage is decreased by 29.87%. The main reason of fault-coverage improvement in SCFC is due to CFG analysis prior to instrumentation. Loop unrolling restructures the CFG topology of the workloads. SCFC takes this restructuring into account, forms new control-flow paths and adds the necessary number of assertions. However CCA, can not adapt to the new topology and instruments the code with the

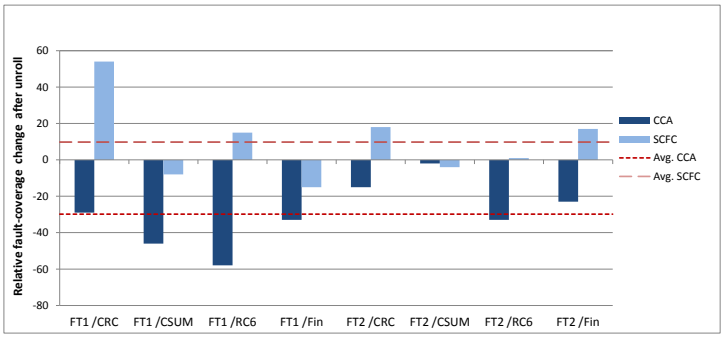


Figure 4.13: Loop-unrolling impact on fault coverage

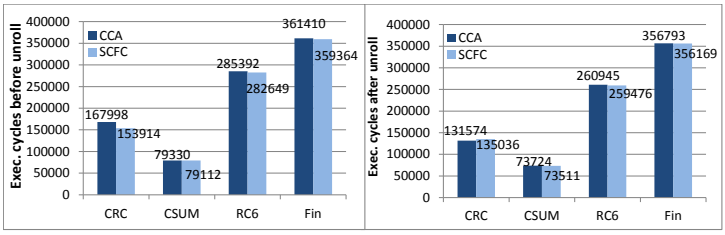


Figure 4.14: Execution cycles in loop-unrolled workloads

same number of assertions before unrolling. The peculiar case where SCFC fault-coverage after unrolling has degraded is for CSUM workload. This workload has two while-loops, that after unrolling have smaller control-flow paths. As discussed in Section 4.5, this condition is not favorable for SCFC and degrades its fault coverage.

The plot in Figure 4.14 shows the total execution-cycles for SCFC and CCA before and after loop-unrolling. As the plot of execution-cycles after loop-unroll shows, in three example workloads SCFC has lower number of execution-cycles than CCA. The only case which SCFC causes higher number of execution-cycles than CCA is for CRC. This is due to the fact that a high percentage of basic blocks of this workload have multiple-predecessors. CCA does not instrument these blocks with inter-block *set/test* assertions, while SCFC does not leave any basic block without being guarded. As a consequence SCFC causes higher execution-cycles, but also higher fault coverage after loop-unrolling, as illustrated in Figure 4.15. High fault-coverage level of SCFC, with loop unrolling (for instruction-level parallelism), makes SCFC a better method for CFE detection than CCA in safety-critical systems with high performance requirements.

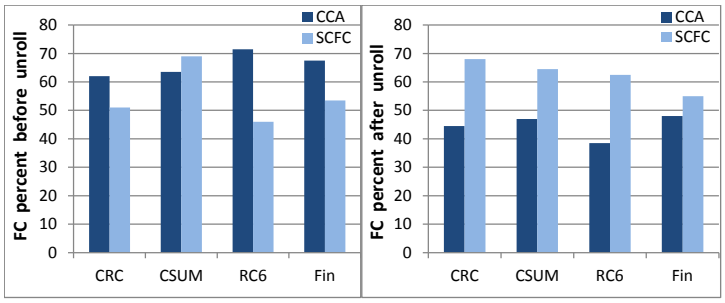


Figure 4.15: Fault coverage in loop-unrolled workloads

4.7 Conclusions

In this Chapter we have presented a novel technique for customizable control-flow fault detection. Our technique (SCFC) is a workload-aware hybrid combination of the two main categories of SM techniques; path-asserting and predecessor/successor-asserting methods. SCFC instruments the code based on the information driven from the CFG. SCFC has been validated on a simple RISC processor for a commonly used biomedical benchmark suite and three control-flow dominated synthetic workloads. It is important to note that the reported results for test programs are worst-case scenarios and, for real applications with larger programs, the overheads will be considerably lower. The results of our evaluation for the programs in ImpBench showed significant overhead reduction compared to CCA. Comparing to ACFC, the method with the lowest overheads, our approach increases fault coverage by 17% with only 2.75% increase in code-size overhead. Meanwhile performance and power-consumption overheads are reduced by 2.7% and 1.6% respectively.

We also investigated, the impact of loop unrolling on the new control-flow error detection method, SCFC. The results are compared to CCA which is a traditional detection scheme with the highest fault-coverage. Comparing results showed that SCFC, thanks to its control-flow graph analysis, can benefit from traditional compiler optimizations such as loop unrolling, both in terms of performance and fault coverage. Previously proposed reliability optimization techniques with similar fault coverage do not have such flexibility. The average fault coverage improvement of SCFC with loop-unrolling compared to a version without loop-unrolling by 9.75%, shows that SCFC is a suitable control-flow error detection method for safety-critical systems with high-performance requirements. Moreover, it shows that this technique can be applied in modern

processors that can exploit instruction-level parallelism, hence providing high fault-coverage and performance at the same time.

The content of this Chapter is based on the following papers:

Ghazaleh Nazarian, Robert M. Seepers, Christos Strydis, Georgi N. Gaydadjiev. **Compiler-aided methodology for low overhead on-line testing.** Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), pp. 219-226, Agios Konstantinos, Greece, July 2013

Ghazaleh Nazarian, Luigi Carro, Georgi N. Gaydadjiev. **Towards Code Safety with High Performance.** Proceedings of the International Conference on Architecture of Computing Systems (ARCS), pp. 209-220, Lubeck, Germany, February 2014

5

Bit-flip aware control-flow error detection

Recent increase of transient fault rates has made processor reliability a major concern in many application domains. Moreover, performance improvements are required for many of today's embedded systems. At the same time software implemented fault detection remains the only option for off-the-shelf processors. Software methods, however, introduce significant performance overheads due to the additional instructions required for the detection. A valid observation is that often code segments not susceptible to faults are unnecessary protected. In this Chapter we propose a technique for systematic analysis of the bit-flip effects on the program control-flow in order to identify only those locations susceptible to control-flow errors and hence minimize the number of fault detection assertions. We instrument the code with minimal overhead, while maintaining high fault coverage level. Our experiments show that using the result of the proposed bit-flip analysis and limiting the code instrumentation to only the susceptible locations releases 28.9% (on average) of the memory that can now be used for other types of assertions while the level of fault coverage remains the same as full instrumentation.

5.1 Introduction

All CFE detection schemes use signature-updates and signature-checks even in basic-blocks that will be never susceptible to faulty transitions. A *susceptible* basic block is a basic block that can be the faulty destination of an erroneous branch instruction. A *non-susceptible* basic block is a basic block that can never be a destination of any potential erroneous branch in a given application binary code. The set of susceptible and non-susceptible basic blocks in

a given piece of code can be determined based on the memory layout and the memory addresses occupied by all basic blocks. Assertions in non-susceptible blocks will unnecessarily increase performance overhead with no impact on fault-coverage. The goal of this work is to avoid assertions in non-susceptible basic-blocks. In order to omit the assertions in non-susceptible basic blocks, the signature monitoring that is used to instrument the code should belong to the local-signature-update category, introduced in Chapter 3. The reason behind this fact is that the other two types of signature monitoring methods, path-asserting and incremental-signature-update as explained in Chapter 3, add *set* assertions in all basic blocks. Therefore, omitting set assertions in non-susceptible basic blocks would corrupt the signature.

In this Chapter, first we show for a set of benchmarks that high percentage of faults can be potentially detected by the OS. We also explain the reasoning of the fact that a high percentage of faults are not detectable using CFE assertions. Afterwards, we explain the proposed systematic approach to analyze the impact of single bit-flips on the control-flow misbehavior of a given program binary and identify all susceptible basic-blocks which constitute the potential destinations of faulty transitions. As explained above, in order to safely omit the assertions in non-susceptible blocks the assertions should be of type local-signature-update. Since, previously proposed CFE detection methods in local-signature-update category have very high performance overheads, we also propose a novel signature monitoring scheme for CFE detection with local-signature update and low overhead. Our Flexible Control Flow Check (FCFC) has lower performance overhead as compared to previously proposed methods in local-signature-update category while the fault coverage is preserved. Based on the result of our bit-flip analysis the assertions in non-susceptible basic blocks are replaced with NOP instruction. Finally, we combine the straight forward implementation of FCFC and our bit-flip analysis in a technique called partial-FCFC and study its benefits.

As authors in [48] investigate, CEDA outperforms CFCSS and YACCA efficiency. Since in this work we are aiming for methods which can provide both high performance and high fault-coverage, we choose CEDA reported in the literature as the most efficient method (considering both performance and fault-coverage) as our reference point for comparison.

The main contributions of this Chapter are:

- Careful study of the effects of single bit-flip faults. The result of our analysis shows that most of the single bit-flip faults cause an error outside the program section boundary. These errors cannot be identified by

the code assertions. However, in systems with an operating system, they can be potentially captured by the operating system;

- A bit-flip analysis framework to allow systematic elimination of all unnecessary assertions for a given program binary;
- A novel, software method (FCFC) for control-flow errors detection that outperforms most efficient state-of-the-art CFE detection methods considering both fault coverage and performance overhead;
- New partial-FCFC approach, combining the above two techniques, able to release 28.9% memory space, while maintaining control-flow error coverage. The released memory can be used for other types of assertions e.g., for data-error detecting;
- A realistic fault-injection scheme which takes into account the execution frequency of instructions.

The reminder of the Chapter is organized as follows: First, the result of our analysis on the impact of single bit-flip fault injection and the targeted fault model are given. Next, the detailed explanation of bit-flip analysis and our FCFC are presented in Section 5.3. Section 5.4 demonstrates the experimental setup, our results and the results analysis. Finally we present the conclusions.

5.2 CFEs detectability observations

A significant percentage of transient faults causing control-flow errors can be detected by the operating systems. Table 5.1 shows the result of control-flow error injection due to single bit-flips into a representative subset of workloads from Mibench [17]. We have selected this subset of Mibench benchmarks based on their CFG topologies in order to represent the entire suite. A large number of injected control-flow errors cause illegal instructions and a significant number of errors transfer the program execution outside the program boundary causing segmentation faults. Since CFE detection methods instrument only the program within its own memory boundaries, such errors are not detectable with any existing software CFE detection technique (such as signature monitoring). These category of faults can be detected by the operating system. To detect the remaining undetected faults leading to incorrect behavior, CFE detection methods are used. Current CFE detection methods add assertions to all basic blocks of the program. However, many of the program

Workloads	OS-detected errors (%)		
	Seg-faults	Illegal instructions	Total
basicmath	78	10	88
qsort	77	13	90
pbmsrch	71	15	86
patricia	67	15	82
FFT	73	11	84
sha	79	8	87
rijndael	85	4	89
CRC	76	11	87

Table 5.1: CFEs detectable by operating system

basic-blocks are not realistic destinations of the erroneous branches caused by single bit-flips in the destination of control instructions. These basic-blocks are non-susceptible blocks and CFE detecting assertions are not needed in these blocks. Contrary to the non-susceptible basic blocks, the blocks that are the potential destination of erroneous branches should be instrumented in order to detect CFEs that result in execution of these basic blocks.

5.2.1 Targeted faults definition

The probability of transforming a non-branch opcode to a branch and vice versa due to a single bit transformation is extremely low and depends on the instruction encoding of a given architecture. It is important to note that branch creation may also happen due to bit-flips in the program counter. However, the probability of CFE occurrence due to single bit-flip in the program counter is relatively low as it is a small circuitry compared to all other processor functional units. For these reasons, in the recent works the main cause of CFEs is considered to be erroneous branch destinations (or as it is classified in this work, Branch-Target-Change) [52]. Therefore, we target CFEs caused by faulty bit-flips in branch instructions destinations causing Branch-Target-Change errors. Since NonBranch-To-Branch type of CFEs has very low probability, adding high-overhead software assertions for detecting such errors in embedded systems with high-performance requirements is not very efficient.

5.3 Instrumenting susceptible basic-blocks

Here we present the framework for deriving susceptible basic blocks. The goal of this framework is to limit the number of set and test assertions significantly by protecting only the susceptible basic-blocks. For this reason, first we need a bit-flip analysis framework to identify all susceptible blocks in a program. Second, considering the problems with all previously proposed CFE detection methods, we also need a new efficient method with the flexibility to remove assertions in non-susceptible blocks without corrupting the runtime signature. In path-based methods with the lowest overheads, we can not remove set assertions in non-susceptible basic-blocks. This is due to the fact that such methods add set assertions to all basic-blocks and the corresponding test assertion only to the last basic-block of the path. Therefore, in order to check the execution flow up to the susceptible basic-block, not only the susceptible block should have the test assertion but all the predecessor basic-blocks in the path should also have set assertions to update the signature correctly. Predecessor/successor assertions with incremental signature update have the same problem of removing set assertions in non-susceptible blocks, since the content of the signature in the susceptible block is calculated based on its previous value and depends on the set assertions in the previous basic-blocks along the path. Local signature updates have very high overhead and do not instrument blocks with multiple predecessors. Therefore, to overcome the aforementioned problems, we propose a novel efficient CFE detection method (FCFC) with local-signature update. We use this method to instrument the code and at the last phase of our framework we replace the assertions in non-susceptible basic blocks with NOP instructions and leave the assertions only in the identified susceptible blocks. In the text to follow we call these memory locations released, but not in the sense that they can be used for normal program code. Instead they can be used for other assertions, e.g., for data error detection. By removing assertions in non-susceptible basic blocks the code is partially instrumented with our proposed assertions, which we call it partial-FCFC instrumentation. The implementation of the framework phases is described next.

5.3.1 Systematic bit-flip analysis

Single bit-flips in branch operands are the main target for CFEs in our framework. However, a large number of single bit-flips in branch operands causes an erroneous branch to a destination outside of the program memory footprint. Such faults cannot be detected by any assertions added to the program. There-

fore we will focus only at faulty branch instructions that will hit inside the program memory space. The faults that cause erroneous branches to a memory location outside the program section can be handled by the operating system in systems that have an operating system. In systems without an operating system a watchdog timer can be used to detect these errors. In order to identify the potential locations (basic-blocks) that an erroneous branch can target within the program memory space we introduce a systematic bit-flip analysis scheme. The proposed scheme takes the assembly and the binary-dump[1] of the program as inputs. Since we consider program executable with static linking, all branch target addresses are resolved and are known prior to execution. The binary-dump file has the program memory footprint and we can extract memory addresses of the program instructions. The final output of the analysis is stored in a text file containing all susceptible basic-blocks named Susceptible Basic-blocks List (SBL). Susceptible basic-blocks are the potential targets of erroneous branches (caused by single bit-flips in branch destinations) and are the only locations where test assertions in a particular program are needed.

Figure 5.1 shows the schematic view of the four steps of the proposed bit-flip analysis scheme. At the first step, all branch instructions operands (the branch target addresses) are extracted and saved in *br-trgt* file. At the second step a set of XORs with MASKs, generate flipped branch target addresses. For instance "$addr\,XOR\,0001$" flips the first bit of the target address and generates an erroneous address. This operation is performed for all the bit positions to generate a list of erroneous addresses. The resulting flipped addresses are saved in *flipped-br-trgt*, which is one of the inputs of the third step. This file can be much larger than the first one and a good optimization could be to just do the bit-flip generation and check against the program address space at the same step. However, in systems with multiple programs running simultaneously the saved potential erroneous addresses can be useful. At the third step we discover which of the generated erroneous addresses are in the range of the program(s) scope. For this reason, the begin and end addresses of the program(s) are used. A simple script compares each of the erroneous addresses in *flipped-br-trgt* file to the extracted instruction addresses within the program scope in the binary-dump. The result of the comparison at step three is a list of potential erroneous target addresses (susceptible to be the target of CFEs) within the program scope and is saved into *err-trgts-in-range* file. Finally at the fourth step, the corresponding basic-blocks to the susceptible target addresses (in *err-trgts-in-range* file) are extracted. To extract the susceptible basic-blocks, we compare the susceptible instructions addresses to the corresponding code sec-

[1] generated from the executable using Linux *objdump* command

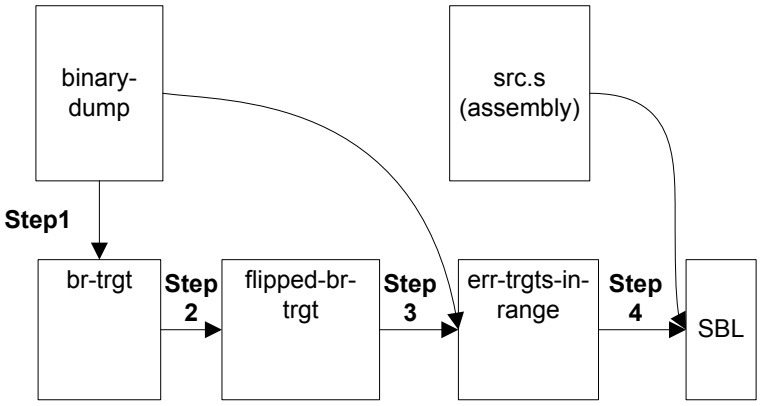


Figure 5.1: Bit-flip analysis scheme

tion in the assembly. With the help of the basic-block labels in the assembly file, the susceptible basic-block labels are extracted and saved to the SBL file.

5.3.2 Flexible Control Flow Check (FCFC)

We introduce a new flexible CFE detection method (shorthand FCFC) that is a signature monitoring method with local-signature-update. FCFC has comparable fault coverage to existing signature monitoring methods with local-signature-update but much lower performance overhead. Furthermore, as we explain next, it can be used in combination with our bit-flip analysis framework to further decrease the performance overhead. In FCFC, similar to other signature monitoring schemes, we use a global signature[2]. In Figure 5.2 the global signature is *S* and the signature of the basic-blocks B1, B2 and B3 are correspondingly (01), (10) and (11). FCFC uses predecessor signature checking for basic-blocks with single predecessor. These blocks have *test* assertions checking the predecessor basic-block correctness and *set* assertions setting the global signature to the current basic-block signature. However this scheme cannot protect blocks with multiple predecessors. Therefore, for those basic-blocks, we employ pairs of set and test assertions similar to CEDA.

To elaborate our method, first we explain the details of CEDA set assertions, as given in Chapter 2, for the single-predecessor block (B2) in Figure 5.2(c). This explanation helps to understand how the set assertion of a multiple-predecessor block in our proposed FCFC is arranged. CEDA has multiple set assertions for updating the global signature to the signature of the current block. There are

[2] a global variable updated by *set* assertions and checked with *test* assertions

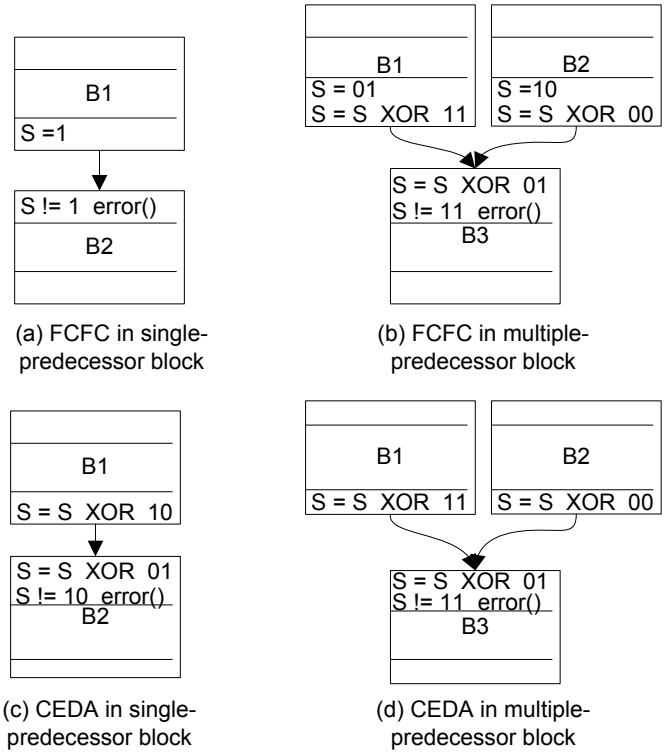


Figure 5.2: FCFC and CEDA assertions

set assertions in each of the predecessor blocks and a set assertion in the current block. In Figure 5.2(c), in case of error-free control-flow, expected signature value in B1 is "1" (01). The set assertion at B1 calculates the bitwise XOR between the global signature (S) and the signature of its successor block which is "2" (10). At the end of B1, the global signature is then updated to the result of the bitwise XOR which is (11), 01 XOR 10 = 11. In the beginning of B2 the second part of set calculates the bitwise XOR between S and the signature of the predecessor which is "1" (01). In case of no error the result of bitwise XOR in the beginning of B2 and the new value of S is (10), 11 XOR 01 = 10 . The test assertion checks for the consistency between the global signature content (S) and the signature of the current block which is "2" (10) at the beginning of B2. Figure 5.2(d) shows CEDA assertions for CFE detection in the multiple-predecessor block (B3). The pair of set assertions in B3 and its first predecessor (B1), are similar to the set assertions of B2 and its single predecessor B1 in Figure 5.2(c). Similar to the set assertions in Figure 5.2(c), the set assertion in B1 does bitwise XOR with the signature of its successor

"3" (11) and the set assertion in B3 does bitwise XOR with the signature of its first predecessor "1" (01). The set assertion in the second predecessor (B2) does bitwise XOR of the global signature content with a constant value. This constant value is the XOR between the signatures of the B2 itself and the signature of the successor block (B3) and the sibling predecessors (B1), 10 XOR 11 XOR 01 = 00. With this arrangement if the execution reaches B3 via B1 or B2, without encountering CFE along the path, the global signature content in B3 is equal to the signature of B3, which is "3". The test assertion checks the global signature consistency at this point. CEDA assertions are explained in more details in [48].

Figures 5.2(a) and 5.2(b) show FCFC assertions in single-predecessor and multiple-predecessor basic-blocks, respectively. In Figure 5.2(a) the test assertion of basic-block (B2) with single predecessor checks the consistency between the global signature (S) and the predecessor block signature which is "1". However, basic-block (B3) in Figure 5.2(b) with multiple-predecessors can not be tested with the simple set and test pair as B2 in Figure 5.2(a). The proposed set assertions for basic-blocks with multiple-predecessors are similar to the ones used in CEDA. The only difference between FCFC set assertions for multiple-predecessors and CEDA is that FCFC adds statements for assigning the block signatures to the global signature in the predecessor blocks while this is not needed in CEDA. These assignments are necessary to make the signature update of the set assertions local and independent of the previous basic-blocks assertions along the path.

5.3.3 Instrumentation using SBL information

The SBL contains the list of all basic blocks which should be guarded with test assertions. The basic blocks that are not in SBL are non-susceptible blocks and hence their assertions can be removed. However, if the assertions are bluntly removed the program with partial-FCFC instrumentation will change memory layout as compared to the full-FCFC instrumented program. This means the branch destination addresses (which are used in the bit-flip analysis step to generate the SBL file) would change making the SBL file invalid. In order to keep the memory layout of the instrumented program consistent, instead of removing the assertions in the non-susceptible basic blocks, they are replaced by "NOP" instructions. FCFC signature monitoring belongs to local-signature-update category and therefore it allows replacing assertions that are not necessary. It is worth to note that these NOP locations can be seen as empty slots suitable for data-error detecting assertions or other instructions that can be used

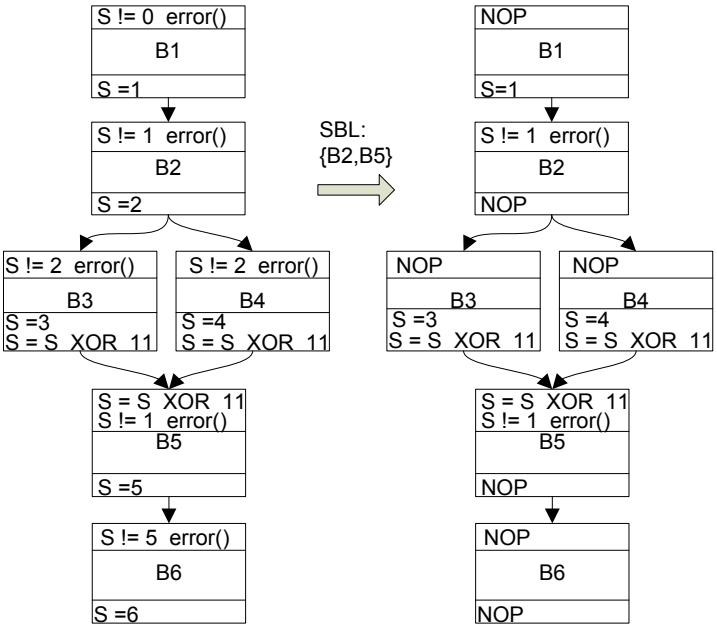


Figure 5.3: Partial-FCFC instrumentation based on SBL

to test specific processor units [39]. Figure 5.3 shows a sample CFG instrumented with FCFC assertions before and after using the information provided by the SBL file. The CFG at the left is instrumented with full-FCFC (with assertions in all blocks) and the one at the right shows partial-FCFC (with assertions only in the susceptible basic-blocks which are B2 and B5).

5.4 Experimental setup and results

To investigate the proposed FCFC scheme, we compare it to one of the state-of-the-art schemes in order to show its effectiveness in removing unnecessary assertions in non-susceptible basic blocks. We have extended the LLVM compiler with CEDA and full-FCFC optimization passes. We have used the result of the SBL file to replace assertions in non-susceptible blocks with NOPs and get a code with partial-FCFC instrumentation. Full-FCFC assertions protect all basic-blocks in the CFG, while partial-FCFC has assertions only to the specified susceptible blocks in SBL file. We use representative workloads from Mibench [17] in our evaluation. With the help of the optimized compilers, four different binaries for each workload are generated; the original binary

without any optimization, binaries with CEDA, full-FCFC and partial-FCFC. To obtain the error coverage of each optimization scheme we inject CFEs into the respective binary and account for the number of detected errors.

5.4.1 Experimental setup

Below we provide an overview of the target architecture and our proposed fault injection mechanism.

Target architecture: For this experiment, we have used a machine with AMD Turion(tm) II P520 dual-core processor, 4GB memory and Ubuntu GNU/Linux 12.04 LTS x86-64 operational system. However, the proposed approach can be easily applied to any other target, general purpose, HPC or embedded.

Fault-injection: In order to introduce CFEs to the binaries at runtime, we have implemented a gdb-based fault-injector (using gdb version 7.4) similar to the fault injection scheme used for the CEDA evaluation in [48]. In the fault injector used in [48], the instruction to be influenced by the error is chosen statically and at runtime an error is injected to the chosen instruction, e.g., by flipping a single bit in the branch instruction operand. The shortcoming of the fault injection scheme in [48], is that it selects the instruction where CFE is injected without taking into account the instruction execution frequency at runtime. As a consequence, an instruction which is executed only once at runtime will be selected with the same probability for CFE injection as an instruction executing hundreds of times. This is not realistic, because instructions with higher frequencies have higher chance to be impacted by faults. In order to address this limitation, we use profiling information to obtain the execution frequencies of each branch instruction. Contrary to the previous method, in our fault injection scheme, we select the instructions for fault injection by considering their runtime frequency as explained below.

```
1-) Sum up all branch frequencies in Total_Freq;
2-) Choose a RANDOM between 1 and Total_Freq;
3-) for ( i ==1 to N)
        Sum_Freq = Sum_Freq+ FREQ(i);
        if (Sum_Freq < RANDOM)
            go to 3;
        else
            chosen_branch_instruction = i ;
```

Figure 5.4: Fault injection mechanism

First, by profiling, all branch instructions and their frequencies are derived into a list and sorted based on the execution order at runtime. Each entry in the list has a branch and its corresponding runtime frequency. Afterwards the branch instruction that is the target of fault injection is chosen by following the steps as depicted in the pseudo code of Figure 5.4. As the pseudo code shows, at the first step all frequencies are accumulated into *Total_Freq*. At the second step, a RANDOM number in the range between "1" and the accumulated *Total_Freq* is randomly chosen. The third step iterates from the first branch instruction up to the last branch instruction (N in Figure 5.4 represents the total number of branch instructions). In each iteration the corresponding frequency to the branch instruction is summed up to *Sum_Freq* and compared with RANDOM. *Sum_Freq* in each iteration has the accumulated frequencies of all branch instructions before the current branch instruction. If the *Sum_Freq* is lower than the RANDOM value, the next branch instruction in the list is investigated. If the resultant *Sum_Freq* is greater than the RANDOM the corresponding branch instruction is chosen for fault injection.

Branch execution order	Execution frequency
Branch1	2
Branch2	100
Branch3	1
Branch4	500
Branch5	73

Table 5.2: Branch execution order with the corresponding execution numbers

Table 5.2 shows a sample list with the sorted branch instructions and their corresponding frequencies. As the result of the first step of random instruction selection, the accumulated sum for all branches is 676. At step two a random number in the range 1 to 676 is chosen, which is 234 in our assumed example. At step three the comparison starts with the execution frequency of Branch1. Since the frequency of Branch1 is lower than the random number (234), the comparison continues as explained in step three. Finally, when repeating the same operation for Branch4, the result of accumulating the frequency of Branch4 to the previous accumulated frequencies (603) becomes greater than the chosen random number. Therefore, Branch4 is the selected instruction for error injection. With this set up, random instruction selection considers the runtime execution frequency and instructions with higher execution frequency have higher probability to be chosen for error injection. The selected branch may be executed multiple times at runtime, therefore a random execu-

tion of the instruction is chosen for the fault injection. This random execution number is between one and the execution frequency of the branch which is derived during profiling. Finally to inject CFEs as discussed in Section 5.2.1, a random bit of the selected branch instruction operand at the randomly selected execution cycle is alternated.

5.4.2 Metric for evaluating error detection methods

Software error detection methods add assertion checks to the program and the number of assertions directly influences fault-coverage and performance (typically reduced number of checks lowers overheads but leads to worse fault-coverage). We propose a new metric to quickly assess such methods while considering both, fault coverage and performance overhead for a given method. The proposed Detection Efficiency Factor (DEF) depends on both performance-overhead[3] and fault-coverage[4] and is a suitable figure of merit for evaluating different reliability optimization methods in high-performance embedded systems:

$$DEF = Fault.coverage / Performance.overhead$$

The main usage of DEF is to quickly find the best method for a given program of a specific size. DEF allows the selection of the best method among multiple fault-detection methods while considering both fault-coverage and performance degradation.

5.4.3 Experimental results

In full-FCFC, all basic blocks are instrumented with assertions, while in partial-FCFC only susceptible blocks have assertions. As it will be explained below, full-FCFC and partial-FCFC have the same DEF efficiency factors. Therefore, first we compare CEDA only with full-FCFC and afterwards we show what is the benefit of using partial-FCFC over full-FCFC. We investigate the error coverage of these CFE detection techniques using our gdb-based fault injector described in Section 5.4.1. Table 5.3 shows the result of fault injection for one thousand executions of Mibench workloads instrumented with full-FCFC and CEDA. The corresponding performance overheads and DEF efficiency factors are also presented in the Table. This set of Mibench workloads is carefully chosen to represent all different topologies and sizes of the

[3] the percentage of additional clock-cycles in the instrumented program

[4] the percentage of detected faults among all injected faults

Workloads	CEDA			full-FCFC		
	overhead [%]	coverage [%]	DEF	overhead [%]	coverage [%]	DEF
basicmath	15.20	9.8	0.6	6.71	8.2	1.2
qsort	5.66	8.0	1.4	6.81	5.6	0.8
pbmsrch	7.43	9.8	1.3	0.88	7.9	8.9
patricia	10.78	8.8	0.8	6.26	8.9	1.42
FFT	8.37	10.3	1.2	1.64	10.7	6.5
sha	66.92	10.4	0.15	19.72	7.5	0.38
rijndael	20.27	11.1	0.54	7.64	7.6	0.99
CRC	16.21	8.3	0.51	13	3.5	0.27

Table 5.3: CEDA and full-FCFC performance overhead (%), fault coverage (%) and DEF efficiency factor

benchmarks in the suite. Varying sizes and topologies emphasize both the advantages and limitations of FCFC. The coverage numbers show slightly higher fault detection for CEDA. However, for the majority of the workloads, the efficiency factor DEF based on performance overhead numbers shows full-FCFC as detection scheme with higher efficiency. The exceptional cases are *qsort* and *CRC*. The reason behind this is that CFGs of these two workloads contain many basic blocks with multiple predecessors. Since FCFC set assertions have an extra statement in multiple predecessor blocks compared to CEDA, in CFGs with high number of such basic blocks, FCFC has higher performance overhead than CEDA.

We have analyzed the same set of workloads with the proposed bit-flip analysis scheme. The result of the analysis is presented in the plot of Figure 5.5. This plot compares the total number of basic-blocks in the workload CFG and the number of the identified susceptible blocks. This plot shows that in some workloads such as *rijndael* and *patricia* more than half of the total basic blocks are non-susceptible blocks. Such workloads have potential for improvement by removing assertions from the non-susceptible blocks. In other smaller workloads such as *CRC* and *qsort* most of the CFG basic blocks are susceptible blocks. Therefore, these workloads have less benefit from limiting the assertions to susceptible blocks compared to *rijndael* and *patricia*. By using the result of this analysis we have replaced assertions in non-susceptible blocks with NOP instruction and converted full-FCFC instrumented benchmarks into partial-FCFC instrumented benchmarks. As a result, depending on the number of non-susceptible blocks in each workload number of memory locations get available. These memory locations can potentially be used

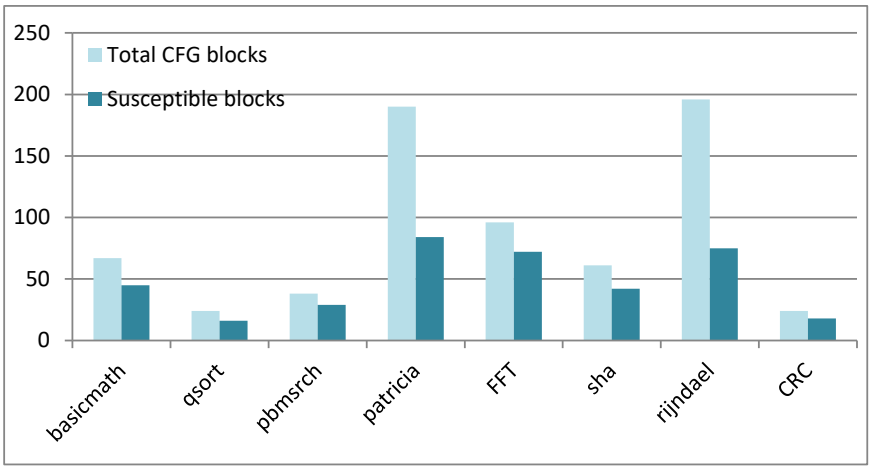


Figure 5.5: The comparison between the number of susceptible blocks and the total number of blocks in the CFG

for other types of assertions , e.g., data error detection. As explained in Section 5.3 in the this Chapter we call these memory locations released, but not in the sense that they can be used for normal program code. Table 5.4 compares the fault coverage of full-FCFC instrumentation and partial-FCFC instrumentation. The fault coverage columns show equal or negligible differences between full-FCFC instrumentation and partial instrumentation based on SBL. The fourth column shows the percentage of memory locations that are released using partial-FCFC instrumentation.The released slots are assertion locations in non-susceptible blocks that are replaced by NOPs. Full-FCFC and partial-FCFC have identical performance overheads and therefore equal efficiency factors. The advantage of partial-FCFC over full-FCFC is to have NOP slots in non-susceptible blocks in place of assertions. These locations can be used for data-error detecting assertions to protect the program against data-errors without additional performance and memory overheads. The last column of Table 5.4 presents the frequency of susceptible basic-block execution. These values give the number of times the susceptible blocks are executed compared to the total number of basic-blocks, showing that on average these blocks are visited 69.8% of the times. This shows that more than half of the program execution time is spent inside the susceptible basic blocks that confirms the importance of instrumenting these blocks with reliability assertions.

Workloads	full FCFC	partial-FCFC		SB execution
	coverage [%]	coverage [%]	released slots [%]	frequency [%]
basicmath	8.2	8.0	20.6	71
qsort	5.6	5.6	18.1	54
pbmsrch	7.9	7.9	17.1	97
patricia	8.9	8.7	57.4	47
FFT	10.7	10.7	20.5	20
sha	7.5	7.5	21.2	98
rijndael	7.6	7.3	63.1	95
CRC	3.5	3.5	13.9	77
Average	7.58	7.41	28.9	69.8

Table 5.4: Fault coverage of full/partial FCFC with released locations ratio and susceptible block execution frequency

5.5 Conclusions

In this Chapter we presented a new CFE detection scheme (FCFC) with local-signature-update assertion types for instrumenting only the identified susceptible blocks for a given workload binary. FCFC has comparable fault coverage to CEDA, however it causes much lower overhead. Comparing the two methods in terms of DEF efficiency factor, FCFC outperforms CEDA in the majority of cases. DEF efficiency factor was presented as a metric to consider both the fault coverage and performance overhead at the same time. Further we proposed a systematic bit-flip analysis framework. This framework takes the program binary-dump and assembly as inputs and generates the list of susceptible basic-blocks to CFEs caused by single bit-flips in branch destination addresses. We combined the result of our bit-flip analysis together with FCFC. As a result we presented a workload aware technique to omit program instrumentation in basic blocks that are not susceptible to CFEs. With this information, we replaced the assertions in non-susceptible blocks with "NOP" instructions. It is important to mention that our proposed bit-flip analysis framework can be used with any other signature monitoring schemes with local-signature-update. In order to evaluate the bit-blip analysis framework we have generated binaries from Mibench benchmark suite with full-FCFC instrumentation and with partial-FCFC instrumentation (assertions only in susceptible blocks). The experiments showed that limiting the instrumentation only to the susceptible blocks releases (on average) 28.9% instruction memory while the fault coverage remains the same as with full instrumentation. The released locations can be used for data-error detecting assertions to additionally improve fault

tolerance of the targeted systems.

The content of this Chapter is based on the following paper:

Ghazaleh Nazarian, Diego G. Rodrigues, Alvaro Moreira, Luigi Carro, Georgi N. Gaydadjiev. **Bit-Flip Aware Control-Flow Error Detection.** Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 215-221, Turku, Finland, March 2015

6

Low-cost Software Control-Flow Error Recovery

In modern safety-critical embedded systems reliability and performance are two important criteria. When error recovery methods are used an important factor is the recovery time, especially in systems with real-time requirements. A key observation that helps improve software recovery methods is that only a finite number of locations in a given program are susceptible to errors. In this Chapter we propose a fast software recovery scheme that instruments the program only at locations vulnerable to control-flow errors. We use the systematic bit-flip analysis, described in Chapter 5, to identify the exact locations susceptible to control-flow errors in a given program binary. This helps us to instrument the code with minimal overheads, while maintaining high-level of correctability and quick recovery times. Our experiments show that with the result of our bit-flip analysis we can limit the code instrumentation to only the susceptible locations and greatly improve the efficiency (taking into account the fault coverage, recovery time and performance overhead) compared to the latest control-flow error recovery methods.

6.1 Introduction

In order to satisfy the three requirements (reliability, performance and short recovery time) in real-time, high performance and safety critical systems, the ideal case for CFE recovery is to know exactly which basic blocks are the potential source of errors and add checkpoints only at those locations. The target fault model in our work is single bit flips in branch instructions destinations, which are due to events such as crosstalk or radiation. As discussed in the previous Chapter, a significant number of basic blocks are not susceptible to

CFEs. Assertions and checkpoints used to protect the non-susceptible blocks do not improve reliability, however, increase performance overhead.

In this Chapter, we propose a novel CFE recovery method which adds checkpoints only at basic blocks that are susceptible to CFEs. We use the bit-flip analysis scheme proposed in Chapter 5. We already have explained how our framework analyzes the impact of single bit-flips on the control-flow misbehavior and identifies all potential destinations of faulty transitions. In this work we extend the previously proposed bit-flip analysis in order to identify the basic blocks where CFEs are initiated from. These blocks are the susceptible sources where an erroneous branch can occur. The result of our extended analysis scheme gives the ordered pairs of susceptible source and destination blocks. Using this information, we instrument the code with only the necessary assertions and checkpoints to protect only the susceptible blocks.

The main contributions of this Chapter are:

- A novel fast control-flow error recovery method;
- Extended framework for identifying the potential source basic blocks where an erroneous branch may stem from;
- Low-cost, highly efficient check-pointing scheme based on placing the checkpoints only at the identified susceptible source blocks;
- Efficient recovery scheme with short recovery time of only 28 cycles and low performance overhead compared to state-of-the-art methods.

The rest of the Chapter is organized as follows: Next, the motivation behind this work is given. The detailed explanation over the extended bit-flip analysis and our check-pointing scheme is given in Section 6.3. Section 6.4 describes the experimental setup and results. Finally, we present our conclusions.

6.2 Motivation

As observed in the previous Chapter, recovery from errors outside the program boundary, which are caused by bit flips in branch instruction operands, is only possible with the help of the operating system that can detect the error as segmentation fault and re-execute the program from the beginning. It should be noted that only the CFEs which lead the execution into an erroneous destination inside the program boundary can be detected and recovered from

software methods used to instrument the program. However, the majority of CFE detection and recovery methods add assertions to all basic blocks of the program, often not necessary. Recovery methods that use checkpoints to save the processor's state at specific program locations, trade-off the recovery time for performance. Placing less checkpoints and dividing the program in larger sections reduces the cost of checkpoints, but increases the recovery time in case an error occurs. Since many of the program basic blocks are not realistic destinations of the erroneous branches caused by single bit-flips in the destination of control instructions, these basic blocks can be excluded from instrumentation. We have implemented an effective CFE detection and recovery scheme with low performance overheads and short recovery time. In this scheme, our main motivation is to limit the CFE detection and recovery instrumentation only to the susceptible basic blocks. The checkpoints, which are the main cause of significant performance overheads in CFE recovery schemes, should only be placed at the sources where a potential faulty jump may occur.

6.3 Fast recovery with workload specific checkpoints

In our proposed recovery method, at compile time, instructions are added to the program in order to detect and recover from the error. Complete recovery from CFEs is accomplished using Code Specific Checkpoints (CSC). The error detecting instructions are executed during the normal flow of the program. However, the instructions added for error recovery are executed only when an error is detected and the execution control is transferred to the special function for error recovery. The goal of our scheme is to have an efficient recovery scheme with short recovery time, low performance overhead and high fault coverage. To achieve the lowest possible recovery time, we need to provide an arrangement that the recovery process is done at the smallest possible granularity, which is at the level of the individual basic blocks. Moreover, to have low performance overhead, we need to add error detecting and recovery instructions only at the necessary locations of the program that are the potential basic blocks where CFEs can occur. As explained in the previous Chapter, it is important to note that not all CFE detection methods offer the flexibility to instrument only a selected number of basic blocks. Therefore, in order to fulfill this requirement, we need to use a flexible error detection method such as FCFC. FCFC has the flexibility to assert the correct execution flow by adding instructions only to the required subset of basic blocks. Moreover, it provides a high fault coverage. In what follows, the important aspects of the proposed

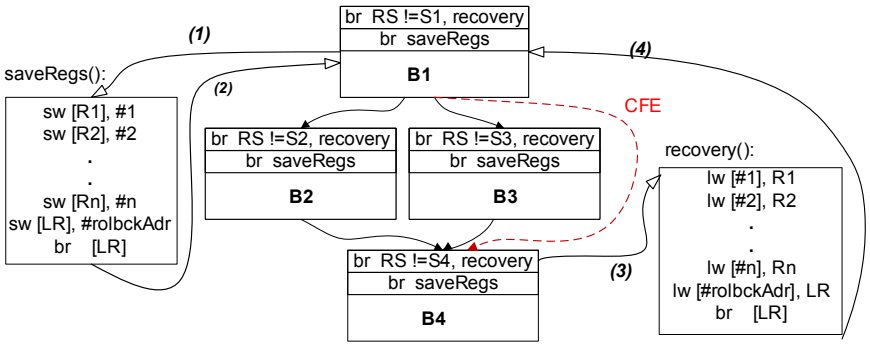


Figure 6.1: Recovery flow

recovery method are explained.

6.3.1 Fast Recovery Scheme

In order to decrease the amount of time between the moment a CFE occurs and the moment the program execution is recovered, we implement the detection and recovery processes at the basic block level. FCFC fault detecting instructions are able to detect CFEs immediately after their occurrence in the faulty target block. Moreover, our proposed recovery method guarantees that the execution control will be transferred right to the basic block that was the source where the CFE occurred. In this way, contrary to the majority of conventional checkpoint based methods, the error recovery time is limited by the basic blocks length.

Figure 6.1 shows how the recovery scheme transfers the control immediately to the block before CFE occurrence. As depicted in the Figure, two statements are added in the beginning of each basic block to implement recovery. The first statement in the basic blocks (*br RS!=Sig, recovery*) is the test assertion of CFE detection scheme. In this Figure for the sake of readability the statements related to set assertions of detection scheme are not shown. In case the test assertion finds a mismatch between the runtime signature content and the expected signature value at the basic block, the recovery function is called. The second statement is a simple function call that invokes the saveRegs function. After saveRegs function is invoked the return address (which is the original start address of the basic blocks before adding the recovery statements) is saved into the Linked-Register (LR). In the Figure a CFE occurrence is depicted by the dashed edge from B1 to B4. Also the steps of code executions

that leads to recovery from the depicted CFE are shown. The first step, before the CFE occurs at the beginning of the source basic block where CFE will stem from, the control transfers to saveRegs function. This function acts as a checkpoint and saves the contents of all registers in the register file. Moreover, it saves the content of the linked register that holds the start address of the basic block. This address is the location where the execution should roll-back to, when an error is detected. In the second step, the execution is transferred back to B1. After CFE occurrence, the test statement in B4 will detect the error and call the recovery function, shown as step 3. The recovery function restores the saved contents of the register file from the previous checkpoint (which is at B1) and loads the roll-back address (which is the start address of B1) into the linked register. Finally, in the fourth step, the execution rolls back to the beginning of B1. This recovery scheme works also when there are multiple predecessors. For example in Figure 6.1, depending on the executed path at runtime, one of the predecessors of B4 block (which is either B2 or B3) is executed. Accordingly, the saveRegs function in invoked from the beginning of the executed block and the corresponding return address (the start address of the executed basic blocks, B2 or B3) is saved. In case an error is initiated from these blocks, after the error is detected the execution rolls back to the saved return address.

Since the extra code that is added to transfer the execution control back to the point before CFE occurrence is minimal and the recovery is designed in a way that there is no need to search for the source basic block where the CFE was initiated, our recovery scheme minimizes recovery time. In our approach, there is no need to search for the source basic block where the CFE was initiated, the recovery time is a fixed number of execution cycles, which is the number of cycles required to execute the recovery function as depicted in Figure 6.1.

```
function-error-handler:

br  F != FID(f), global-error-handler;
err-flag = 0;
num-err = num-err + 1;
br num-err > thresh, exit;
for each block in the function
    find the faulty source block;
jmp function-error-handler;
```

```
global-error-handler:

err-flag = 1;
for each function f in the program
    br F == FID(f), f;
num-err = num-err + 1;
br num-err > thresh, exit;
jmp global-error-handler;
```

Figure 6.2: Error recovery code in ACCE

The recovery steps in a similar recovery method (ACCE) [47] consist of four

function calls that have in total higher number of recovery code compared to our scheme. For error recovery, ACCE makes use of function-error-handlers associated to each function in the code and a global error handler. Figure 6.2 shows the code for function-error-handler and the code for the global error handler. A unique identification number is assigned to each function (FID) which is used in function-error-handler and the global error handler to identify the function that the error was initiated from. In ACCE if an error is detected the steps as depicted in Figure 6.3 are taken to recover from the error. As first step, when an error is detected, the function-error-handler is invoked. In the function-error-handler it is checked whether the error was initiated from the current function or not. If the error was initiated from the current function, the execution is transferred back to the faulty source basic block. While, if the error was not initiated from the current function the global error handler is invoked, depicted as the second step in Figure 6.3. The global error handler finds the function that causes the error and calls this function, depicted as third step in the Figure. Again in the function that the error was initiated from, the corresponding function-error-handler is invoked, depicted as the fourth step in the Figure. This time the function-error-handler finds the basic block that was the source of the error and transfers the execution back to this block. Since, in this scheme after CFE occurs, there is a search to find the source function and the basic block, the recovery time is variable depending on the time spent for searching. Compared to ACCE, which works also at basic block level, the additional amount of recovery code of our method is much lower resulting in shorter recovery time.

Another disadvantage of ACCE is that it does not support full recovery from CFEs. This is due to the fact that ACCE does not save and restore the modified data in the register file due to re-execution of the rolled-back basic block. To provide data integrity and full recovery from CFEs, the authors propose ACCE with Duplication (ACCED). In this technique, the computations on the data and the data container variables are duplicated. ACCED, places the duplicated computations in a separate basic block with instrumentation for CFE detection such as a normal basic block. By comparing the two versions of the same variable an inconsistency in data, after CFE recovery, can be detected. Afterwards, ACCED uses a flag *err_comp* to identify the variable containing the corrupted data. After identifying the corrupted data, the content of the variable with correct data gets copied to the second copy of the variable. This is an expensive solution for data restoration. Having a duplicated computation and variables for data leads to significant performance overhead.

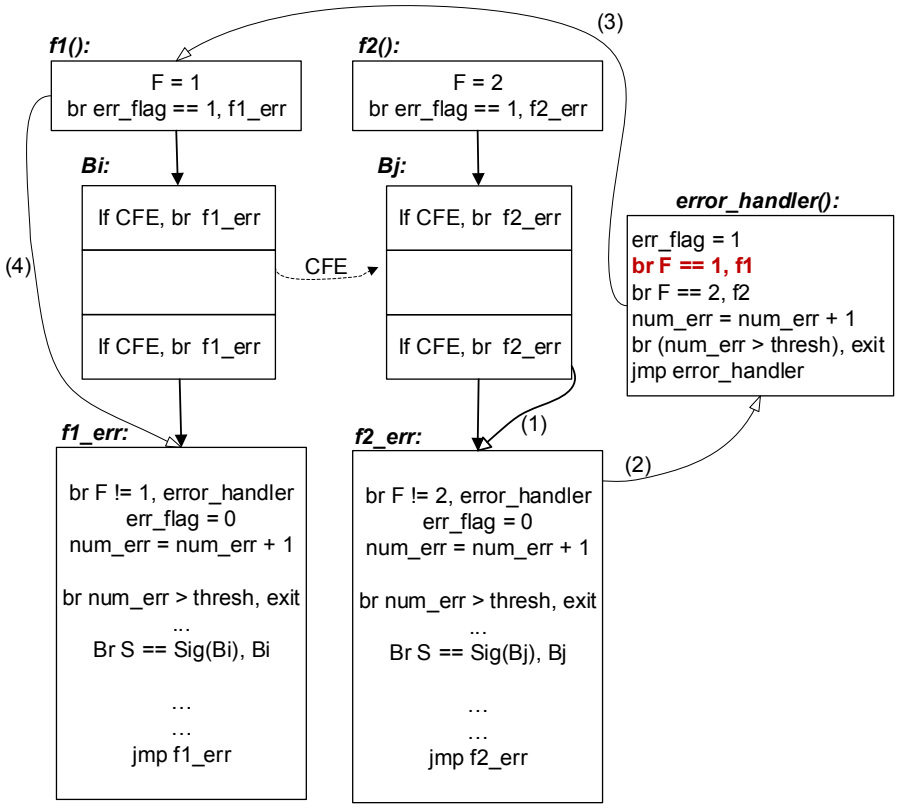


Figure 6.3: Error recovery flow in ACCE

6.3.2 Efficient Checkpoints at Identified Susceptible Blocks

Instrumenting all basic blocks with checkpoints as described above is too costly. Fortunately, as we have shown in the previous Chapter, our study about the impact of single bit flips on the control-flow mis-behavior has shown that not all basic blocks are susceptible to CFEs. In other words, only a number of basic blocks in the CFG are susceptible to CFEs and require protection by assertions and checkpoints.

In order to minimize the performance overhead introduced by checkpoints, we use our bit-flip analysis framework as explained in the previous Chapter. This framework uses the program binary dump and assembly as input and generates a list containing all susceptible blocks that are the potential destinations of CFEs caused by single-bit transitions in branch destinations. In order to use this information and minimize the number of checkpoints, we also need to

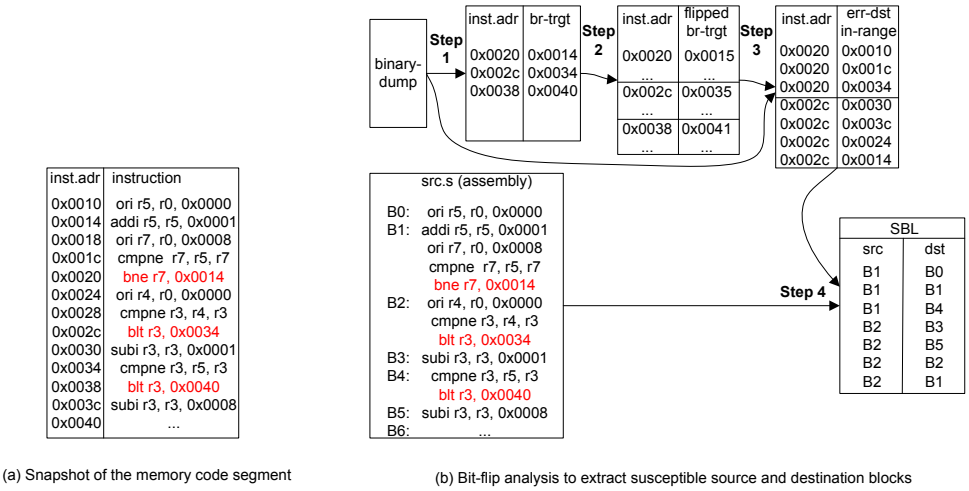


Figure 6.4: Bit-flip analysis scheme illustration

know the susceptible source basic blocks that the CFEs can stem from. We have extended the framework to include this information in the generated list.

Figure 6.4 shows the working of the extended framework to extract susceptible source and destination blocks. An example snapshot of the memory code segment is depicted in Figure 6.4(a) and the steps for extracting the list of susceptible blocks for this part of the code segment are illustrated in Figure 6.4(b). At the first step, all branch targets (*br-trgt*) and the branch instruction addresses (*inst.adr*) are extracted. In the second step a set of XORs with MASKs, generates all possible branch target addresses caused by a single bit flip. For instance "addr XOR 0001" flips the first bit of the target addresses and generates one possible address. The resulting flipped addresses are saved as *flipped-br-trgt*. In the Figure the first bit flipped-target is shown. In the third step, a simple script compares each of the erroneous addresses in *flipped-br-trgt* file to the extracted instruction addresses within the program scope. The result of the comparison at this step produces all potential erroneous target addresses (susceptible to be the target of CFEs) within the program scope that are also saved into err-dstin-range file. Finally in the fourth step, the corresponding source and destination basic blocks of the susceptible target addresses are identified. To extract the susceptible basic blocks, we compare the susceptible instructions offset to the corresponding code section in the assembly. With the help of basic block labels in the assembly file, the susceptible basic block labels are extracted and saved in the SBL file.

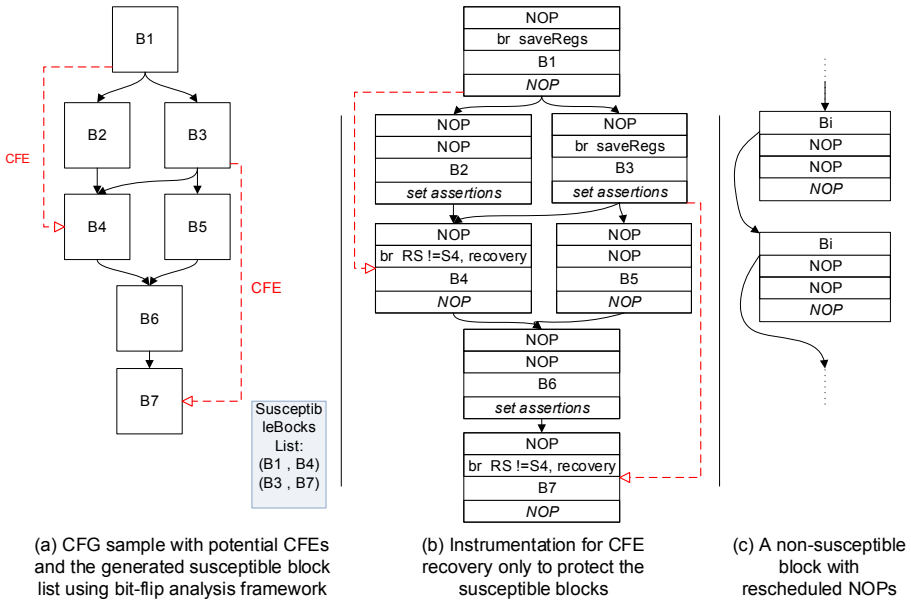


Figure 6.5: Instrumentation and checkpoints in susceptible blocks

Figure 6.5(a) shows a sample CFG, corresponding to an example code, with potential erroneous CFE edges and the output list of the bit-flip analysis framework containing pairs of susceptible source and susceptible destination blocks. In this sample execution flow graph, the potential CFEs are depicted by the dashed edges from B1 to B4 and from B3 to B7. The result of the bit-flip analysis is the list containing two pairs of susceptible blocks as depicted in Figure 6.5(a). The first elements in the block pairs are the susceptible source blocks, B1 and B3, and the second elements represent the susceptible destinations, B4 and B7.

Figure 6.5(b) depicts the same sample CFG, which is instrumented with CFE detecting assertions and checkpoints only at the necessary locations using the susceptible blocks list information. Only the susceptible source blocks need to have a call to the saveRegs function to checkpoint the register file contents. In the sample graph only B1 and B3 have the call to saveRegs function. Accordingly, only susceptible destination blocks need to have test assertions to check if the basic block is reached through a valid control-flow or due to a CFE occurrence. In the example graph only B4 and B7 contain *"br RS!=Sig, recovery"* statement, which is the test assertion. The predecessors of susceptible destination blocks are the only blocks that should have set assertions to update the runtime signature to a valid signature. These blocks in the sample

CFG are B2, B3 and B6.

Assertions and checkpoints in non-susceptible blocks are replaced with NOP instructions. In order to gain performance, we reschedule the replaced NOPs after the branch instruction at the end of the basic block. As a result, at runtime the replaced NOP instructions are not executed. Consequently, the recovery instrumentations causes no performance overhead in non-susceptible blocks.

6.4 Experimental setup and results

To investigate the proposed recovery scheme, we compare it to a recent state-of-the-art work to show its effectiveness in removing unnecessary assertions and checkpoints in non-susceptible blocks. We have implemented and optimized a compiler using the CoSy development framework [1]. The generated compiler targets a basic, 32-bit, five-stage, in-order RISC processor. We have implemented compiler passes for our proposed recovery method and ACCE. We use a representative set of workloads from Mibench [17]. For each workload three different binaries are generated; the original binary without any optimization, binaries compiled with ACCE optimization pass, and binaries compiled with our recovery pass. To obtain the error coverage of each optimization scheme we inject a single CFE per run (by flipping single bits in the branch instruction operands) into the respective binary and inspect the number of detected errors. The error injection mechanism which is used to evaluate the recovery scheme is identical to the improved mechanism described in Chapter 4.

In order to compare the proposed recovery method to other recovery methods, we consider all crucial criteria such as correctability (fault coverage), performance overhead and recovery time. High number of assertions and checkpoints located at short intervals improve the correctability and the recovery time of software recovery methods. However, this has a negative impact on the performance. On the other hand, low number of assertions and checkpoints improves performance but degrades correctability and recovery time. Therefore, in order to asses a software recovery technique these three metrics should be considered. For evaluating the method in terms of fault-coverage and performance overhead[1], we use a metric similar in spirit to the one proposed in Chapter 5. The only difference is that here we consider correct output ratio (correctability[2]) of the method instead of the fault-coverage. This metric is

[1] the percentage of additional clock-cycles in the instrumented program

[2] the percentage of corrected runs against the total number of experiments

Workloads	ACCE		
	correct output	not recoverable errors	
		wrong output	out of program boundary
qsort	83	47	871
pbmsrch	67	84	850
sha	58	58	885
dijkstra	87	28	886
CRC	76	32	893
Average	76.83	62.33	861.66

Table 6.1: Categorization of the outputs in 1,001 ACCE instrumented code runs with random control-flow errors

Workloads	Recovery with CSC		
	correct output	not recoverable errors	
		wrong output	out of program boundary
qsort	376	94	531
pbmsrch	333	159	509
sha	318	218	465
dijkstra	409	124	468
CRC	313	102	586
Average	334.66	166.66	498.83

Table 6.2: Categorization of the outputs in 1,001 CSC instrumented code runs with random control-flow errors

named as Correction Efficiency Factor (CEF) and is calculated as:

$$CEF = \frac{correct.output.ratio}{Performance.overhead}$$

As stated before, another crucial metric is the recovery time, needed to resume error free operation after a CFE is detected. We have measured this by capturing the number of execution cycles between the moment an error is detected until it is recovered. Please note that these numbers are specific for the architecture we used and will differ on another platforms. We however believe that the relative ratios between the two methods will be preserved. In what follows the results of the proposed recovery method are discussed in detail.

6.4.1 Experimental results

Table 6.1 shows the result of fault injection for one thousand and one runs of the Mibench workloads instrumented with ACCE and Table 6.2 shows the same result of Mibench workloads instrumented with our recovery scheme (CSC). In the Tables, the code behavior to the injected CFE is categorized into three columns: correct output, wrong output and execution out of program boundary. The errors that are not recoverable are the ones that cause wrong output and the ones that lead the execution outside the program boundary. In all the execution runs of the workloads with control-flow error, ACCE recovers on average 76.83 of the cases, while the average recovery number of our scheme is 334.66. The main reason for the higher recovery number of CSC compared to ACCE is due to the fact that in the CSC recovery method, the register file content is saved at the checkpoints only in the necessary blocks and restored when needed. As explained in Section 6.3.1, ACCE does not save and restore register file for data integration and instead the authors propose ACCED with duplication of variables and computations. ACCED imposes significant performance overhead due to the duplicated statements.

In ACCE recovery method [47], the number of faults causing wrong output for ACCE is reported, but unfortunately the number of faults causing out-of-boundary execution is not explicitly reported by the authors. The ACCE results, show higher number of injected faults leading to jumps outside of the program boundary as compared to our method. The reason is that the ACCE recovery code introduces many new branch instructions that become themselves the target of the injected faults and cause out of boundary execution. This effect is amplified in the hot regions of the code that will now have twice more branches leading to control flow redirection out of the program scope. Please note that even if those erroneous jumps will be captured by the operating system, the program execution can not be recovered, leaving the complete program restart as the only option.

Please note that in general the fault model used in our experiments is characterized by higher probability of out of program boundary execution as compared to other fault models. Especially this is the case when the instrumented programs are of small size.

The performance overhead imposed by the extra instructions of each recovery method (ACCE and our method) compared to the baseline program binaries (which do not have any optimizations) is illustrated in Figure 6.6. The high performance overhead imposed by both methods for *qsort* and *crc* is due to the fact that both are tiny programs and the extra assertion instructions added by

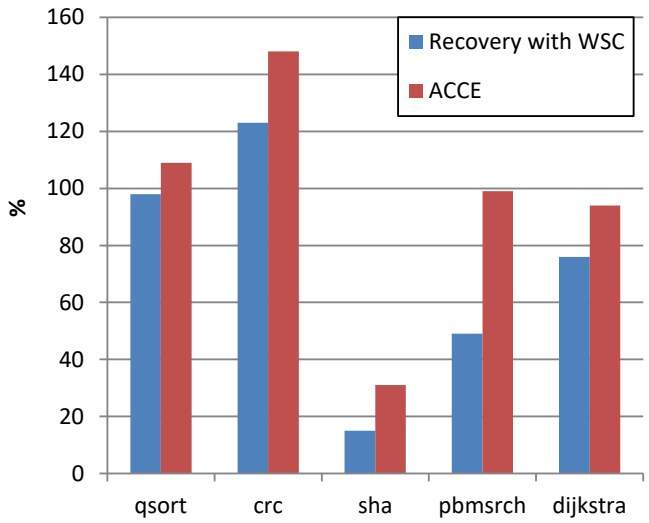


Figure 6.6: CSC and ACCE performance overheads

the recovery schemes result in significant overheads. We deliberately selected these two workloads to enforce a worst-case-scenario.

The chart of Figure 6.7 shows the normalized efficiency factors of ACCE and our method for the same set of workloads. In ACCE instrumentation, the recovery time depends on the number of basic blocks in the functions and whether the erroneous branch destination is inside the same function or it targets a basic block of another function. If it is inside the same function the recovery time is shorter and if the faulty target block belongs to another function the latency will be higher. The minimum recovery time in ACCE is when the CFE initiates in the first basic block of the function and targets a faulty block in the same function. In ACCE implementation for our target processor, the recovery time of the examined workloads is between 32 to 35 cycles. It should be noted that, all the workloads used in our experiments are selected to have a small size and actually contain only one function. Therefore, the ACCE recovery times measured are the lowest possible since in workloads with multiple functions, the recovery time can scale with the number of functions. In addition, the reported ACCE recovery time includes only the execution roll back to the correct point and does not consider restoring the correct data of the modified registers. In contrary, our method has a constant recovery time of 28 cycles that includes the time needed for restoration of the correct data content of the register file. As explained in Section 6.3.1, the recovery time of our scheme is constant and does not depend on the sizes of the workloads. As

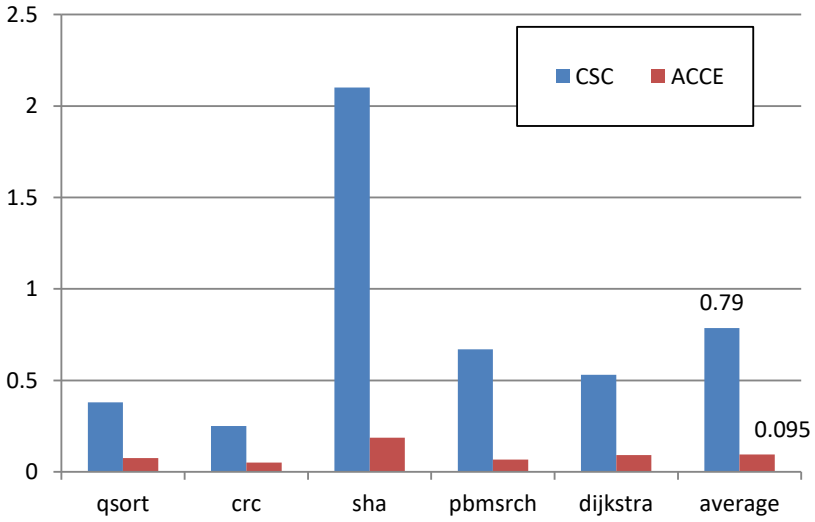


Figure 6.7: CEF factors of CSC and ACCE

depicted in Figure 6.7, CEF factor of CSC recovery method is about 8 times higher than CEF factor of ACCE.

6.5 Conclusions

In this Chapter we introduced a lightweight, low-latency CFE recovery method with checkpoints only at the susceptible source basic blocks. Our proposed recovery scheme is able to detect the CFE and roll back the execution to the beginning of the basic block where the CFE has occurred with a constant latency of only 28 cycles. We extended the previously proposed bit-flip analysis framework to identify the susceptible sources of CFE in the program binary. This information was used to limit the checkpoint locations only to the susceptible source blocks and greatly decrease the imposed overheads. In order to assess our recovery method fairly we considered the three metrics of correctability, performance and recovery time. Comparing our scheme to a well known and widely used recovery scheme (ACCE), shows our method as more efficient considering the above mentioned metrics. The main reason behind the higher efficiency of our proposed method is that the instrumentations for detection and recovery are added exactly at the identified vulnerable spots of the program under consideration.

The content of this Chapter is based on the following paper:

Ghazaleh Nazarian, Razvan Nane, Georgi N. Gaydadjiev. **Low-Cost Software Control-Flow Error Recovery.** Proceedings of the Euromicro Conference on Digital System Design (DSD), pp. 510-517, Madeira, Portugal, August 2015

7 Conclusions

In this thesis, we addressed the problem of significant performance overheads caused by conventional software reliability improvement techniques. Software optimizations are the only solution in embedded devices build with off-the-shelf processors. However, software methods rely on additional assertions into the code for runtime fault detection and recovery and naturally introduce performance overheads. This work has proposed several compile-time methods for reliability optimization. The proposed methods are fully independent of the underlying hardware and are applicable to any arbitrary processor architecture. In the following the summary and the conclusions of this thesis are presented. Finally, we propose promising directions for future work.

7.1 Thesis summary

In Chapter 1, we have elaborated the needs of todays embedded systems and presented the problems related to modern software optimization techniques. We also formulated the research questions that should be answered in order to address these problems.

In Chapter 2, we have presented the background concepts in software reliability optimization techniques and the relevant related works in this area. Further, we have categorized these methods into methods that address data errors and control flow errors. The software optimization methods that target control flow errors were additionally divided into three groups based on the way the assertions are implemented.

In Chapter 3, the compatibility between reliability optimizations and modern techniques for power reduction was studied. Power optimization methods were analyzed and two promising combinations for embedded systems requiring

reliability with limited power budget were identified. More precisely, instruction re-scheduling with instruction duplication and loop flattening (unrolling or fusing) with signature monitoring were found compatible. Additionally, three incompatible pairs have been identified along with the specific limitations.

In Chapter 4, we presented a novel technique for customizable control-flow fault detection. Our technique (SCFC) is a workload-aware hybrid combination of the two categories of signature monitoring techniques; path-asserting and predecessor/successor-asserting methods. In SCFC, we use the control flow graph topology to direct the instrumentation process towards one of the two methods above. Moreover, the impact of loop unrolling on our control-flow error detection method was investigated. The results were compared against CCA which is a widely used detection scheme with the highest fault-coverage. This result showed that SCFC has 50% less performance, memory and power-consumption overheads as compared to CCA while delivering similar fault coverage. Comparing SCFC to ACFC (the best performing path-asserting method) showed that SCFC increases fault coverage by 17% with only 2.75% increase in code-size overhead. Our experimental results also showed that SCFC can benefit from traditional compiler optimizations such as loop unrolling, both in terms of performance and fault coverage. The average fault coverage improvement of SCFC with loop-unrolling compared to a version without loop-unrolling was measured as 9.75%.

In Chapter 5, we introduced a method to omit program instrumentation in basic blocks that are not susceptible to CFEs. To facilitate this a framework for systematic bit-flip analysis was developed. This framework takes the program binary-dump and assembly as inputs and generates the list of susceptible basic-blocks to CFEs caused by single bit-flips in branch destination addresses. With this information, we make sure that only susceptible blocks contain CFE detecting assertions by replacing the assertions in non-susceptible blocks with "NOP" instructions. We have also developed a novel CFE detection scheme (FCFC) with local-signature-update assertions for instrumenting only the relevant basic blocks in a given CFG (the identified susceptible blocks). It is important to mention that our proposed bit-flip analysis framework can be used in combination with any other signature monitoring scheme based on local-signature-update assertions. The experiments show that limiting the instrumentation only to the susceptible blocks releases (on average) 28.9% of the instruction memory while the fault coverage remains the same as the fully instrumented version. The released memory locations can be used for, e.g., data-error detecting assertions to additionally improve fault tolerance of the targeted systems.

In Chapter 6, we introduced a lightweight, low-latency CFE recovery method with checkpoints only in the susceptible source basic blocks. Our proposed recovery scheme is able to detect the CFE and roll back the execution to the beginning of the basic block where the CFE has occurred with a low constant latency. We extended our bit-flip analysis framework to identify the susceptible sources of CFE in the program. This information was used to limit the checkpoint locations only to the susceptible source blocks and decrease the imposed overheads. In order to assess our recovery method fairly we considered the three metrics of correctability, performance and recovery time. When comparing our scheme to a well known recovery scheme (ACCE), we show that our method is more efficient considering (correctability and performance overhead) by a factor of eight. The main reason behind the improved efficiency is that the instrumentations for detection and recovery are limited to exactly the identified vulnerable points of the program. Considering the recovery time, our method has a constant recovery time of 28 cycles, while ACCE recovery time in minimum case is 32 cycles and it will get higher in larger workloads with more than one functions.

7.2 Thesis main contributions

The contributions of this thesis are as follows:

1. **An improved categorization of modern software-based reliability optimization methods.** A careful categorization of existing signature monitoring schemes was performed to emphasize the advantages and disadvantages of these methods. Section 3.2 introduced the three categories and classified existing methods under these categories.

2. **A careful study of the compatibility between software based reliability optimization methods and conventional power reduction techniques.** The result of the this study identified two compatible pairs between reliability optimizations and power reduction techniques. Section 3.4 explained in detail why these optimizations pairs are compatible.

3. **A novel reliability optimization method based on workload specific assertions at compile time that is also fully compatible with loop unrolling.** The proposed method uses the results of the compilation process such as the specific CFG topology to apply workload specific assertions. Therefore, it is compatible with all performance optimization techniques

that change the CFG topology such as loop-unrolling. The fault coverage of the proposed method can be additionally improved on average by 9.75% when combined with loop-unrolling. Section 4.4 explains the workload specific reliability optimization assertions in detail and Section 4.5 analyzes the impact of loop unrolling on this method.

4. **A systematic approach for identification of susceptible locations to CFE and for limiting the assertions required for error detection and recovery to only those locations.** We identify all locations susceptible to CFEs using profiling. Having such information a minimum set of code instrumentation points is defined. Limiting the assertions to only these instrumentation points, can significantly reduce the overheads while guaranteeing equivalent reliability as existing techniques. This is discussed in detail in Chapters 5 and 6.

7.3 Directions for future research

As discussed in the thesis, among all proposed reliability optimization methods the challenge is to find the best suitable method in terms of provided reliability, power consumption and performance overheads. In different systems, based on the specific system requirements, the most suitable method may differ. In one system performance might be more important that reliability and in another system the opposite can be the case. Therefore, there is an open research to tailor the reliability optimization methods for the specific needs of different systems. More specifically, the possible future directions for improving the techniques proposed in this work are discussed below.

In Chapter 4 the impact of loop-unrolling on different signature monitoring methods was investigated. As future work, the impact of other loop-transforming compiler optimizations, e.g., Fission, Skewing, Inversion just to name a few, on the proposed SCFC method and other signature monitoring schemes should be investigated.

As the future work of our framework (proposed in Chapter 5) for profiling and bit-flip analysis of the code we envision two steps. The first step is to improve the fault-coverage and the second step is to additionally reduce the performance overhead. In the current work we have not taken into account indirect branches. This is due to the fact that this type of branches are used only for returning from function calls and not for inter basic block jumps. In order to further improve the fault-coverage and the robustness of our technique the

return addresses of function calls can be calculated prior to runtime by knowing the offsets and the starting address of each function. Moreover, in order to improve the fault coverage and detect also data error faults, the released NOP locations should be used for data error detecting assertions.

The second step of improvement of our bit-flip analysis framework and the proposed instrumentation of susceptible basic blocks is to additionally reduce the performance overhead. For this reason, the terminating branch instruction of the non-susceptible blocks can be re-scheduled before the released NOP locations at the tail of the corresponding blocks. With this rearrangement we can additionally improve performance. As a result our partial-FCFC (proposed in Chapter 5) will have lower performance overhead than full-FCFC. It should, however, be noted that with this arrangement there would be no room left for data-error detecting assertions as discussed above. This decision should be taken depending on the systems requirement whether reliability or performance is more critical. In systems with performance considered more important than reliability, rescheduling the terminating branch instruction will be beneficial. While in reliability sensitive systems the proposed framework in Chapter 5, can be improved by using additional assertions for data error detection.

As the future work, it should also be investigated how to identify the most suitable fault model for a given system. SCFC is proposed to cope with both BranchTrgChange and NonBranchToBranch types of CFEs. However, FCFC targets only BranchTrgChange error type, An example of the future work for investigating which fault model and optimization (SCFC or FCFC) is suitable for which system is given here. NonBranchToBranch error type represents diverging from the correct control flow path and it can happen due to two faults: a fault in program-counter or a fault that makes the decoding stage to invoke a branch incorrectly. However, in many ISAs converting the non-branch instructions to branch by a single bit-flip is not possible. Moreover, in many systems the program-counter is protected by redundant circuitry. In such systems, the probability of having NonBranchToBranch error types is extremely low. Therefore, in such systems instrumenting the code with FCFC assertions is preferable. However, in systems that the program-counter is not protected or non-branch instructions can be converted to branch due to single bit-flips, SCFC assertions are strongly suggested.

Bibliography

[1] Cosy compiler. 43, 49, 96

[2] Synopsys processor designer. 43, 56

[3] O. Goloubeva ad M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error detection using control flow assertions. In *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–588. IEEE, November 2003. 15, 16, 31, 37

[4] A.V. Aho and J.D. Ullman. *Foundations of Computer Science: C Edition*. Principles of computer science series. W. H. Freeman, 1994. ISBN-10 0716782847. 57

[5] Z. Alkhalifa, V S Nair, N. Krishnamurthy, and J.A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. on Parallel and Distributed Systems*, 10(6):627–641, June 1999. 15, 16

[6] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Trans. Computing Surveys*, 26(4):345–420, December 1994. 42

[7] Mohsen Bashiri, Seyed Ghassem Miremadi, and Mahdi Fazeli. A checkpointing technique for rollback error recovery in embedded systems. In *Proceedings of International Conference on Microelectronics*, pages 174–177, December 2006. 17

[8] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO05)*, pages 197–208, November 2005. 3

[9] Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 83–92. DSN, 2006. 20, 22

[10] V. Delaluz, M.Kandemir, N.Vijaykrishnan, and I. Kolcu. Compiler directed array interleaving for reducing energy in multi-bank memories. In *Proc. of ASP-DAC*, pages 288–296, 2002. 35

[11] J. B. Eifert and J. P. Shen. Processor monitoring using asynchronous signatured instruction streams. In *Proceedings of the 14th Annual International Conference on FaultTolerant Computing*, pages 394–399, June 1984. 15, 38

[12] S Feng, S Gupta, A Ansari, and Scott Mahlke. Shoestring: Probabilistic soft-error reliability on the cheap. In *Proceedings of ASPLOS*, pages 385–396, 2010. 9

[13] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A. Mahlke, and David I. August. Encore: Low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 398 – 409, December 2011. 17

[14] T S Ganesh et al. Seu mitigation techniques for microprocessor control logic. In *Proceedings of the Sixth European Dependable Computing Conference (EDCC'06)*, pages 77–86, 2006. 3

[15] Maria George and Peter Alfke. Linear feedback shift registers in virtex devices. www.xilinx.com, 2007. 59

[16] U Gunneflo, J Karlsson, and J Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings of Nineteenth International Symposium on Fault-Tolerant Computing, 1989. FTCS-19*, pages 340–347, June 1989. 9, 41

[17] M Guthaus, J Ringenberg, D Ernst, T Austin, T Mudge, and R Brown. Mibench: A free, commercially representative embedded benchmark suite. In *International Workshop on Workload Characterization*, pages 3–14, 2001. 71, 78, 96

[18] Chi-Hong Hwang and Allen C.-H. Wu. A predictive system shutdown method for energy saving of event-driven computation. 5(2):226–241, April 2000. 34

[19] G. A. Kanawati, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Evaluation of integrated system-level checks for on-line error detection. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pages 292–301. IEEE, September 1996. 15, 31, 43, 47

[20] Mahmut Kandemir, N. Vijaykrishnan, Mary Jane Irwin, and Wu Ye. Influence of compiler optimizations on system power. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(6):801–804, December 2001. 35

[21] Daya Shankar Khudia and Scott Mahlke. Low cost control flow protection using abstract control signatures. In *Proceedings of LCTES*, pages 3–12, June 2013. 9, 15

[22] Daya Shanker Khudia, Griffin Wright, and Scott Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, pages 99–108, May 2012. 21

[23] Huang Kuang-Hua and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984. 20

[24] Chingren Lee, Jenq Kuen Lee, Tingting Hwang, and Shi-Chun Tsai. Compiler optimization on vliw instruction scheduling for low power. 8(2):252–268, April 2003. 36

[25] Yann-Hang Lee and C.M. Krishna. Voltage-clock scaling for low energy consumption in real-time embedded systems. In *Sixth International Conference on Real-Time Computing Systems and Applications*, pages 272–279. RTCSA, 1999. 34

[26] A Mahmood and E. J McCluskey. Concurrent error detection using watchdog processors-a survey. In *IEEE Trans. on Computers*, pages 160–174, 1988. 3

[27] Ganesh Marlowe, A. K. Ganesh, and T. J. Marlowe. A compiler-based approach to fault-tolerance in real-time systems. pages 1–8, 1996. 9, 17

[28] Ghassem Miremadi, Johan Karlsson, Ulf Gunnejlo, and Jan Torin. Two software techniques for on-line error detection. In *22nd International Symposium on Fault-Tolerant Computing*, pages 328–335. FTCS, July 1992. 15, 18

[29] Daniel Mosse, Hakan Aydin, Bruce Childers, and Rami Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *In Workshop on Compilers and Operating Systems for Low Power*, pages 1–9, 2000. 34

[30] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control flow checking by software signatures. *IEEE Trans. on Reliability*, 51(1):111–122, March 2000. 13, 14, 15, 16, 22, 31

[31] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. on Reliability*, 51(1):63–75, March 2002. 19, 22

[32] A Parikh, M Kandemir, N Vijaykrishnan, and M J Irwin. Instruction scheduling based on energy and performance constraints. In *Proceedings of IEEE Computer Society Workshop on VLSI*, pages 37–42, 2000. 61

[33] Rafael B Parizi, Ronaldo R Ferreira, Luigi Carro, and Alvaro F Moreira. Compiler optimizations do impact the reliability of control-flow radiation hardened embedded software. *International Embedded Systems Symposium IESS: Embedded Systems: Design, Analysis and Verification pp 49-60*, pages 49–60, 2013. 42

[34] M. Rebaudengo, M.S. Reorda, and M. Violante. A new software-based technique for low-cost fault-tolerant application. In *Annual Reliability and Maintainability Symposium*, pages 25–28, 2003. xiii, 19, 20

[35] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. Soft-error detection through software fault-tolerance techniques. In *Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems*, pages 210–218. IEEE, 1999. 15, 19, 22

[36] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *Int. Symposium on Code Generation and Optimization*, pages 243–254, March 2005. 19, 22

[37] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *International Symposium on Computer Architecture*, pages 148–159. ISCA, June 2005. 22

[38] N Saxena and E. J McCluskey. Dependable adaptive computing systems the roar project. In *Proceedings of International Conference on Systems, Man, and Cybernetics*, pages 2172–2177, 1998. 3

[39] Robert Seepers, Christos Strydis, and Georgi Gaydadjiev. Architecture-level fault-tolerance for biomedical implants. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 104–112, 2012. 78

[40] A Shye, T Moseley, V.J Reddi, J Blomstedt, and D.A Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *International Conference on Dependable Systems and Networks, DSN*, pages 297 – 306, June 2007. 3

[41] Mani B. Srivastava, Anantha P. Chandrakasan, and R. W. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. 4(1):42–55, March 1996. 34

[42] Christos Strydis, Christoforos Kachris, and Georgi N. Gaydadjiev. Impbench: A novel benchmark suite for biomedical, microelectronic implants. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008.*, pages 82–91, July 2008. 41, 43, 52, 56

[43] Ching-Long Su, Chi-Ying Tsui, and A.M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *Compcon Spring '94, Digest of Papers.*, pages 489–498, February 1994. 33

[44] V Tiwari and M Tien-Chien Lee. Power analysis of a 32-bit embedded microcontroller. In *Proceedings of Design Automation Conference, ASP-DAC '95/CHDL '95/VLSI '95.*, pages 141–148, September 1995. 61, 62

[45] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. *J. VLSI Signal Process. Syst.*, 13(2-3):223–238, August 1996. 32

[46] R. Vemu and J.A. Abraham. Ceda: control-flow error detection using assertions. In *IEEE Transactions on Computers*, pages 1233–1245, September 2011. 15, 16

[47] R Vemu, S Gurumurthy, and J Abraham. Acce: Automatic correction of control-flow errors. In *Int. Test Conference*, pages 1–10, 2007. 17, 42, 91, 98

[48] Ramtilak Vemu and Jacob Abraham. CEDA: Control-flow error detection using assertions. *IEEE Trans. on Computers*, 90(9):1233–1245, September 2011. 42, 70, 77, 79

[49] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. 37(3):195–237, September 2005. 32

[50] Rajesh Venkatasubramanian, John P. Hayes, and Brian T. Murray. Low-cost on-line fault detection using control flow assertions. In *9th IEEE On-Line Testing Symposium*, pages 137–143. IEEE, July 2003. 8, 15, 16, 31, 45

[51] J Vinter, J Aidemark, P Folkesson, and J Karlsson. Reducing critical failures for control algorithms using executable assertions and best effort recovery. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 347–356, 2001. 22

[52] N Wang, J Quek, T Rafacz, and S Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 61–70, July 2004. 9, 72

[53] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 13–23, 1994. 34

[54] Kent Wilken and John Paul Shen. Continuous signature monitoring: efficient concurrent-detection of processor control errors. In *Proceedings of the 1988 international conference on Test: new frontiers in testing*, pages 914–925, September 1988. 15

[55] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 49–62. ACM, 2003. 34

[56] S.S. Yau and Fu-Chung Chen. An approach to concurrent control flow checking. *IEEE Trans. on Software Engineering*, SE-6(2):126–137, March 1980. 9

[57] Dakai Zhu. Energy management for real-time embedded systems with reliability requirements. In *Proceedings of International Conference Computer-Aided Design, ICCAD*, pages 528–534, November 2006. 1

List of Publications

International Conferences

1. Ghazaleh Nazarian, Christos Strydis, Georgi N. Gaydadjiev. **Compatibility Study of Compile-Time Optimizations for Power and Reliability.** Proceedings of the 14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD), pp. 809-813, Oulu, Finland, August-September, 2011

2. Ghazaleh Nazarian, Robert M. Seepers, Christos Strydis, Georgi N. Gaydadjiev. **Compiler-aided methodology for low overhead on-line testing.** Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), pp. 219-226, Agios Konstantinos, Greece, July 2013

3. Ghazaleh Nazarian, Luigi Carro, Georgi N. Gaydadjiev. **Towards Code Safety with High Performance.** Proceedings of the International Conference on Architecture of Computing Systems (ARCS), pp. 209-220, Lubeck, Germany, February 2014

4. Ghazaleh Nazarian, Diego G. Rodrigues, Alvaro Moreira, Luigi Carro, Georgi N. Gaydadjiev. **Bit-Flip Aware Control-Flow Error Detection.** Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 215-221, Turku, Finland, March 2015

5. Ghazaleh Nazarian, Razvan Nane, Georgi N. Gaydadjiev. **Low-Cost Software Control-Flow Error Recovery.** Proceedings of the Euromicro Conference on Digital System Design (DSD), pp. 510-517, Madeira, Portugal, August 2015

Samenvatting

Microprocessors worden gebruikt voor een steeds groter wordende reeks aan toepassingen, van kleine ingebouwde systeemapparaten tot aan grote computers en supercomputers. Ingebouwde microprocessor-systemen zijn in de moderne maatschappij steeds essentiler geworden. Afhankelijk van het toepassingsgebied moeten deze ingebouwde systemen aan steeds meer eisen voldoen. De grootste uitdagingen vandaag de dag zijn kosten, prestaties, het energiegebruik, de betrouwbaarheid, snelle en voorspelbare responstijden en het silicium oppervlak. In alledaagse computersystemen zijn sommige van deze parameters minder belangrijk dan andere, hoewel prestaties, silicium oppervlak en energiegebruik altijd belangrijk zijn voor ingebouwde systemen. Daarnaast is in moderne systemen betrouwbaarheid een nieuwe, zeer essentile eis in opkomst. Van alle bovengenoemde factoren kunnen prestaties, energie, reactie en betrouwbaarheid worden bewerkstelligd middels softwarematige oplossingen, er zijn geen hardware-aanpassingen of toevoegingen nodig. Maar dit soort optimalisatie-technieken kunnen wel de prestaties en de vermogenskarakteristiek benvloeden. Het belangrijkste doel van dit werk is dus om nieuwe software-technieken te vinden die de betrouwbaarheid bevorderen, met klein invloed op prestaties en energieverbruik. Daarom worden de methodes voor de optimalisatie van de betrouwbaarheid in detail bestudeerd en wordt er een zorgvuldige indeling van de bestaande software-technieken gemaakt. De sterke en zwakke kanten van elke categorie worden zorgvuldig onder de loep genomen. Uit de verkregen informatie worden twee nieuwe optimalisatietechnieken voor fout-opsporing en een techniek voor fout-herstel voorgesteld. Deze optimalisatie-technieken minimaliseren de vereiste code-instrumentatiepunten, terwijl de betrouwbaarheid behouden blijft, op niveaus vergelijkbaar met andere state of the art-oplossingen. Daarnaast wordt er een generieke methodiek voorgesteld, om te helpen met het proces van het identificeren van het minimum aan code-instrumentatie-punten. Voor de evaluatie kozen we een uitdagend uitgangspunt, bestaande uit de best bekendstaande technieken voor fout-opsporing en fout-herstel, gevonden in openbare literatuur. De resultaten uit dit onderzoek voor een reeks biomedische benchmarks laten zien dat door het gebruik van de voorgestelde ontwerptechniek, fout-opsporing en -herstelmethodes het prestatie- en energieverbruik drastisch kan worden verbeterd, terwijl de foutendekking op een lijn blijft met voorheen voorgestelde en veelgebruikte methodes.

Curriculum Vitae

Ghazaleh Nazarian was born on the first of April 1982 in Tehran, Iran. She obtained her Bachelor of Science in Computer Engineering at the Azad University Of Central Branch in Tehran. In 2006 she moved to the Netherlands, receiving her Master of Science in Computer Engineering at the Delft University of Technology in 2008. She then joined the CE department at the same university in the pursue of her PhD, focusing her research on compilers optimizations for reliability. In the fall of 2013 she worked as Guest Researcher at the University of Porto Alegre, Brazil. In 2014 she joined Associated Computer Expert (ACE) B.V. in Amsterdam, where she worked as Compiler Engineer. Currently she is with Brightsight B.V. in Delft working as Security Evaluator.