Center for Reliable Computing

TECHNICAL REPORT

Software Implemented Hardware Fault Tolerance

Nahmsuk Oh

CRC-TR 00-9

December, 2000

Center for Reliable Computing
Gates Building 2A, Room 236
Computer Systems Laboratory
Dept. of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-9020

ABSTRACT

This technical report contains the text of Nahmsuk Oh's thesis "Software Implemented Hardware Fault Tolerance."

FUNDING

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047

SOFTWARE IMPLEMENTED HARDWARE FAULT TOLERANCE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Nahmsuk Oh

December 2000

ABSTRACT

Transient errors in computer systems can cause abnormal behavior and degrade system reliability, data integrity and availability. This is especially true in a space environment where transient errors are a major cause of concern. Fault avoidance techniques such as radiation hardening and shielding have been the major approaches to obtaining the required reliability. Recently, unhardened *Commercial Off-The-Shelf* (COTS) components have been investigated for space applications because of their higher density, faster clock rate, lower power consumption and lower price.

Since COTS components are not radiation hardened, and it is desirable to avoid shielding, *Software-Implemented Hardware Fault Tolerance* (SIHFT) has been proposed to increase the data integrity and availability of COTS systems. This dissertation presents three new SIHFT techniques for error detection: *Control Flow Checking by Software Signatures* (CFCSS), *Error Detection by Duplicated Instructions* (EDDI), and *Error Detection by Diverse Data and Duplicated Instructions* (ED^4I).

Previously studied software techniques are either inadequate or require assistance from special hardware, but CFCSS, EDDI and ED^4I are pure software methods. In CFCSS, signatures are embedded into the program during compilation and compared with run-time signatures during execution. In EDDI, instructions are duplicated at compile-time, and scheduled by exploiting *Instruction-Level Parallelism* (ILP) to reduce performance overhead. CFCSS and EDDI detect transient errors but not permanent faults. However, in ED^4I, a program is compiled to a new program with diverse data so that it can detect a permanent fault.

Our fault injection experiment simulating bit flips in memory shows that, for the designs simulated, EDDI provides over 98% fault coverage without any extra hardware. Because of instruction duplication, code size overhead is approximately 100%, but by exploiting ILP, we reduce the performance overhead down to 61% on average. For control flow checking experiment simulating branching faults, CFCSS provides 97% fault coverage. In addition, when we duplicate programs or instructions, we can use ED^4I to enhance data integrity in the system.

Furthermore, for space experiments, we have implemented EDDI and CFCSS in sort and FFT programs running in the ARGOS satellite. During a 136 day period, our techniques have detected a total of 198 out of 203 errors, and show 98% error detection coverage.

While traditional error detection and fault tolerance techniques require special dedicated hardware, our SIHFT techniques use time redundancy for error detection and significantly improve data integrity without requiring special hardware.

ACKNOWLEDGEMENTS

I express my deep gratefulness to my advisor, Professor Edward J. McCluskey, for his guidance and support during my study at Stanford. He modeled the high quality teaching and research that I aspire to follow in my career. He taught me how to solve research problems and clearly presents results. He encouraged me when I faced difficulties in research as well as life. Many things I learned from him will be of great help to me.

I would like to thank Professor Giovanni De Michelli, my associate advisor and Professor David Bloom for being my third reader.

I have greatly appreciated my colleagues at the Center for Reliable Computing: Dr. Nirmal Saxena, Chao-Wen Tseng, James Li, Ray Chen, Ahmad Al-yamani, Subhashishi Mitra, Robert Huang, Catherine Yu, Samy Makar, and Sungroh Yoon. I want to especially thank Philip Shirvani for helping me get my start and answering my many questions. His comments were always helpful to me. I want to especially thank Siegrid Munda for her administrative support. I very greatly appreciated her kindness and helpfulness.

I am grateful to Professor Yoon and Professor Hong at Yonsei University for their guidance during my undergraduate years.

I want to thank to my aunt and specially to my cousin, Vicky Kim for her proof reading of this dissertation.

I would like to thank my many friends at Stanford. Wonil Roh was a great help to me when I first came to US. Jung-Suk Goo and his wife always treated to me when I have no time to cook for myself. Carlos Aldana has been my roommate for four years and helped me to study in new environment. And many thanks to Jaejune Jang, Hyuk-jun Lee, Won-Joon Choi, Suk Hwan Lim, Dong-Hyun Kim, Jaewon Shin and Kiyoung Nam. I want to especially thank to Eui-Young Chung. He was always a great help to me when we discuss about new ideas. Special thanks to Hee-Joo Kang for her encouragement and support. She did proof reading of this dissertation and gave me inspiration whenever I faced difficulties.

Finally, I would like to thank my parents and my brother for their tremendous love, endless support, and many prayers. They have always believed in me and always been there for me. I dedicate this dissertation to them.

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047.

Table of Contents

Chapter 1: Introduction 1

1.1 Motivation 1

1.1.1 COTS Components in Space 1

1.1.2 The CRC ARGOS Project 3

1.1.3 Software Implemented Hardware Fault Tolerance 4

1.2 Contributions 4

1.3 Outline of thesis 6

Chapter 2: Background and Previous Work 7

Chapter 3: Control Flow Checking 11

3.1 Signature Monitoring 11

3.1.1 Previous Work 11

3.1.2 Control Flow Checking by Software Signatures 11

3.2 CFCSS Algorithm Description 12

3.2.1 Preliminaries 12

3.2.2 The run-time signature *G* 13

3.2.3 The run-time adjusting signature *D* 15

3.2.4 Algorithm Description 17

3.2.5 Aliasing 18

3.3 Results 19

3.3.1 Fault injection simulation experiment 19

3.3.2 Overhead 20

3.4 Summary 20

Chapter 4: Error Detection by Duplicated Instructions 21

4.1 Error Detection by Duplication 21

4.1.1 Previous Work 21
4.1.2 Error Detection by Duplicated instructions 22

4.2 EDDI Algorithm Description 23
4.2.1 Preliminaries 23
4.2.2 Algorithm for EDDI 24
4.2.3 Scheduling Instructions Exploiting Instruction-Level Parallelism 26

4.3 Results 27
4.3.1 Fault Injection Simulation Experiment 27
4.3.2 Overhead reduction by instruction-level parallelism 28

4.4 Summary 29

Chapter 5: Error Detection by Diverse Data and Duplicated Instructions 30

5.1 Data Diversity for Error Detection 30
5.1.1 Diversity overview 30
5.1.2 ED^{4}I: New approach to data diversity 30

5.2 Transformation Algorithm Description 31
5.2.1 Expression and branching condition transformation 31
5.2.2 Determination of *k* 33

5.3 Overflow Handling 36

5.4 Results 37

5.5 Floating Point Numbers 38

5.6 Summary 39

Chapter 6: Concluding Remarks 40

Chapter 7: References 41

APPENDIX I. ……..…..……..….46

“Control Flow Checking by Software Signatures,”

CRC Technical Report 00-4

(To appear in *IEEE Transactions on Reliability*, September 2001.)

APPENDIX II ……..……..72

“Error Detection by Duplicated Instructions In Super-scalar Processors,”

CRC Technical Report 00-5

(To appear in *IEEE Transactions on Reliability*, September 2001.)

APPENDIX III ……..……107

“ED^{4}I: Error Detection by Data Diversity and Duplicated Instructions,”

CRC Technical Report 00-8

(Presented in *High-Performance Embedded Computing Workshop*,

Lincoln Laboratory, MIT, Sep. 20-21, 2000.)

List of Tables

Table 5.1. Data Integrity calculated with various values of *k* in benchmark programs. 35
Table 5.2. Fault detection probability with various values of *k* in benchmark programs. 36
Table 5.3. Optimum value of *k* determined for each benchmark program. 37

List of Figures

Figure 1.1 Our software tool flow. 5
Figure 2.1. A watchdog processor monitoring the system. 7
Figure 2.2. A signature derived from the instructions in the basic block. 8
Figure 3.1 A sequence of instructions and its graph 12
Figure 3.2 A basic block with checking instructions 14
Figure 3.3 The checking instructions in a correct control flow 14
Figure 3.4 The detection of an illegal branch 15
Figure 3.5 Node v_1 and v_3 have different signatures 16
Figure 3.6 Percentage of faults that produced undetected incorrect outputs 19
Figure 4.1 Extended-precision based control-flow checking 21
Figure 4.2 Duplicated instruction (shadow instruction) and comparison instruction 23
Figure 4.3 Branching at the end of the loop 23
Figure 4.4 Storeless basic block example 24
Figure 4.5 (a) Dependency graph (b) Constructing H_{Dk} (c) Adding H_{Dk} to G_D 26
Figure 4.6 Percentage of faults that produced incorrect outputs without being detected 28
Figure 4.7 Execution time overhead in R4400 (2-way issue) and R10000 (4-way issue) 29
Figure 5.1 (a) The original program (b) The transformed program with k = -2. 33
Figure 5.2 (a) The original program (b) The transformed program with k = -2. 33

Chapter 1: Introduction

1.1 Motivation

Transient faults can cause abnormal behavior of computer systems. Radiation, electromagnetic interference and power glitches are some of the causes of transient faults. For example, in radiation environment, alpha-particles, cosmic rays and solar wind flux can cause a *single event upset* (SEU), which is one of the major sources of bit-flips in digital electronics. A *bit-flip* is *an undesired change in the state of a memory cell*; an SEU can cause the state of a memory cell to change from 0 to 1 or 1 to 0. In this radiation environment, high reliability and dependability can be obtained if the system avoids failures by using radiation-hardened components or shielding the devices, or if the system tolerates failures and continues to operate in the presence of the faults.

Radiation hardened components have been widely used to solve radiation effect problems, but these devices are much more expensive than non-radiation hardened components and have limited availability. In addition, the hardened devices are typically two or three generations behind non-hardened devices in terms of performance, and as the number of radiation hardened device manufacturers decreases, very few of the radiation hardened components will consequently be available in the future.

To overcome the drawbacks of using radiation hardened components, space system designers have recently considered the use of unhardened *Commercial-Off-The-Shelf* (COTS) components in the system [Titus 98] because they can use the state-of-art technology available in the market at a cost very low compared to radiation hardened devices. However, there is only a scarce amount of data available about the reliability of COTS components in space.

The primary objective of the Stanford ARGOS project [Shirvani 99] is the comparative evaluation of two reliable computing techniques in space discussed above: *fault avoidance* using radiation hardened components and *fault tolerance* using *Software Implemented Hardware Fault Tolerance* (SIHFT) techniques in COTS parts. Before introducing the ARGOS project at the Center for Reliable Computing (CRC) at Stanford University, the next section briefly introduces related projects that study COTS components in space.

1.1.1 COTS Components in Space

COTS components generally have limited fault tolerance or avoidance capabilities compared to radiation hardened components that are specially designed for radiation environment; thus, the

reliability of the COTS components should be verified before they are actually used in the radiation environment. We can test COTS components in radiation facilities on the ground, but the testing environment is not exactly the same as real space. An experiment testing COTS components in space is needed.

Microelectronics and Photonics Test Bed (MPTB) is a space experiment in a highly elliptical orbit to test the performance and reliability of COTS microelectronics and photonic components in space [Titus 98]. The satellite carrying the experiment was launched in November 1997. One of the main objectives of the MPTB experiment is to examine whether the COTS devices are suited for space applications. The experiment also includes a ground radiation test on COTS components and the results are compared with actual space measurement data. A number of experiment results are reported in [Dale 95][Holland 96].

The *Mars Pathfinder Microrover Flight Experiment* (MFEX) [Matijevic 96][Golombek 99] is an experiment in which COTS components are tested in Mars. In the MFEX, one of the primary missions is to test the autonomous mobile vehicle performance on the Martian surface. The mission of the rover – implemented using combinations of commercial, mil-spec, and space qualified parts – includes navigating, traversing, gathering data, and carrying an alpha proton x-ray spectrometer [Matijevic 96]. Using COTS components such as a RNET modem from Motorola contributed to reducing the total cost of development an order of magnitude compared to the cost of previous interplanetary spacecraft [Stone 96]. The commercial components of the rover were rigorously tested to qualify their survival in the Martian environment during the development phase, and some of the devices, such as motors, were modified to achieve the required performance and reliability.

While MPTB and MFEX examine whether the COTS components are suitable for space applications, the *NASA Remote Exploration and Experimentation* (REE) project aims at taking commercial supercomputing technology to space for immense onboard processing capability in science missions [REE 99]. Its objective is to develop a fault-tolerant, high-performance, low-power super computer that enables the collected data to be processed dynamically in space rather than downloading the data to the ground and analyzing it. Furthermore, one of the REE's goals is to design a fault tolerant system that operates reliably for 10 or more years using COTS components. Since the system consists of COTS components that have limited fault tolerance, its software must provide fault tolerance mechanisms. For this purpose, REE adopts software implemented fault tolerance for a *middleware layer* that resides between the operating system and the application. The middleware layer provides fault detection, isolation and tolerance

mechanisms. Also *Algorithm Based Fault Tolerance* (ABFT) [Huang 84][Nair 96], one of the SIFHT techniques, is applied to application programs such as Fast Fourier Transform.

As we have seen in MPTB, MFEX and REE, COTS components have recently been considered for space systems; this motivates the Stanford CRC ARGOS project, in which we test radiation hardened and COTS components simultaneously, to evaluate the suitability of COTS components in space and assess the SIHFT techniques for space applications.

1.1.2 The CRC ARGOS Project

The CRC ARGOS project is an experiment carried out as part of the NRL-801: the *Unconventional Stellar Aspect* (USA) experiment on the *Advanced Research and Global Observations Satellite* (ARGOS). The USA experiment on the ARGOS satellite [Wood 94] is primarily a low-cost X-ray astronomy experiment, and the need for high computing power to analyze the X-ray data on board gives us an opportunity to perform an experiment in fault-tolerant computing in real space. The test-bed consists of two processor modules: the *Hard board*, built around the Harris RH3000 radiation hardened chip set, and the *COTS board*, constructed with IDT3081 COTS microprocessor. Both boards can access the full down link science telemetry stream, and data can be uploaded and downloaded on the two boards during the ARGOS mission.

The *McCluskey test*, in which both the radiation hardened and commercial processors of the same class are operated in the same orbital environment, is performed to analyze the radiation effect on the COTS components and evaluate the two techniques: fault avoidance and SIHFT. We assess the fault avoidance by observing SEUs in radiation-hardened components. For example, we run memory test programs to check whether SEUs occur in radiation-hardened components. We can evaluate the radiation effect on the COTS components by executing two identical programs – such as memory tests – simultaneously in the two boards and comparing the error rates of the boards. We assess the SIHFT techniques by implementing them in the application programs (FFT and sorting) and executing the programs in the two boards; then, we observe if SIHFT can detect and tolerate errors in the COTS board.

The McCluskey test was performed on the sun synchronous orbit, and we have observed that the COTS memory gets SEU errors approximately 10bits/Mbyte·day while the radiation hardened memory gets few or no errors. Based on this observation, SIHFT is essential in the COTS components if they have to be used in space.

1.1.3 Software Implemented Hardware Fault Tolerance

Software Implemented Hardware Fault Tolerance (SIHFT) detects or tolerates faults in the hardware by software method without any special hardware for error detection or fault tolerance. The benefit of employing SIHFT is that we can improve the availability of the system at low cost using the existing design of the hardware available in the market. In other words, if we want to increase the reliability of the existing system, SIHFT can add error detection capability or fault tolerance to the system without any modification of the hardware in the system.

Let us first model an error that occurs during program execution. A program can be considered as a sequence of instructions, and the execution of the program can be viewed as executing instructions in a desired sequence. For a more precise description, let us define a *basic block* as *a sequence of instructions without any branching inside or outside except for the last instruction*; then, a program can be represented by a *program graph*, which consists of basic blocks and directed edges connecting the basic blocks. If the correct execution sequence in the program graph is broken, it is a *control flow error*. If the computation result is incorrect, it is a *computational error*. If the information stored in memory is corrupted, it is a *memory error*.

For example, one of the control flow errors is a branch creation; the correct execution sequence among basic blocks is broken. An example for memory error is the case in which an instruction is changed to another type of instruction because a bit-flip occurred in opcode field of the instruction.

1.2 Contributions

In this dissertation, we present three new SIHFT techniques checking for control flow errors, computational errors and memory errors. We have developed *Control Flow Checking by Software Signatures* (CFCSS) for checking control flow errors, and *Error Detection by Duplicated Instructions* (EDDI) for checking transient errors in computation units or memory. Furthermore, we present a new *Error Detection by Data Diversity and Duplicated Instructions* (ED^4I) technique to detect permanent faults using data diversity. In order to generate programs automatically with error detection capability using these techniques, we have implemented a software tool, illustrated in Fig.1.1. Before compilation, our preprocessor can add data diversity to C source code. We compile the source code by `gcc` [Stallman 98] and generate the assembly code. We modified the compiler to reallocate registers and variables for error detection. Our postprocessor adds the additional instructions for EDDI and CFCSS (each can be enabled

independently) to the assembly code. We compile the resultant code by an assembler and obtain an executable object code with error detection capability.

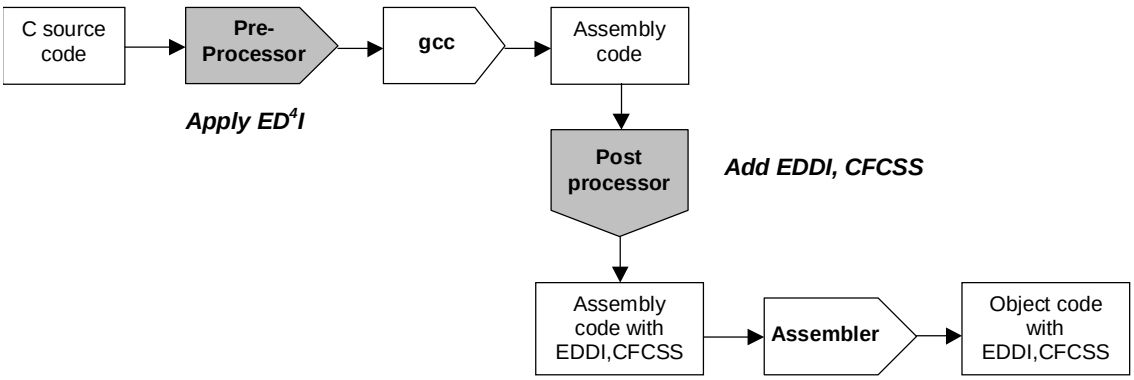


Figure 1.1 Our software tool flow.

In particular, the following is the contributions of this dissertation:

- A new SIHFT technique, *Control Flow Checking by Software Signatures*, is developed to check the control flow among basic blocks.
- A new SIHFT technique, *Error Detection by Duplicated Instructions*, is developed to check the computational and memory errors inside basic blocks.
- Formulas for estimating error detection coverage are derived so that the error detection capability can be predicted during compilation.
- It is shown that we can significantly reduce the execution time overhead incurred by additional instructions for error detection by exploiting instruction-level parallelism available in super-scalar processors.
- A preprocessor, a postprocessor and a software tool flow were developed to automatically place additional instructions for checking control flow errors, computational errors and memory errors in the source code.
- A new SIHFT technique, *ED^{4}I: Error Detection by Data Diversity and Duplicated Instructions*, is developed for hardware fault detection using data diversity. An algorithm that transforms a program to a new program with diverse data is presented and the correctness of the algorithm is formally proved.
- A metric originally developed for quantifying diversity among hardware designs [Mitra 99] is extended to measure the diversity between the original and the transformed program.
- Based on probabilistic analysis, it is shown how we can maximize the data integrity and the fault detection probability in the transformed program.

1.3 Outline of dissertation

Chapter 2 discusses the background of SIHFT and control flow checking, and briefly mentions previous work in related fields.

Chapter 3 introduces CFCSS, the control flow checking technique. Section 3.1 covers previous work in signature monitoring, Section 3.2 describes the CFCSS algorithm, Section 3.3 presents the experimental simulation results, and Section 3.4 summarizes my work on CFCSS.

Chapter 4 describes EDDI, the computation and memory error checking technique. Section 4.1 discusses previous work in related fields, Section 4.2 explains the EDDI algorithm and instruction scheduling for overhead reduction, Section 4.3 discusses the simulation results, and Section 4.4 summarizes the chapter.

Chapter 5 presents ED^{4}I that aims at detecting hardware faults in the system. Section 5.1 discusses previous diversity techniques, Section 5.2 describes the program transformation algorithm and shows how to optimize the program with diverse data, Section 5.3 discusses the simulation results, Section 5.4 explains overflow handling, Section 5.5 discusses floating point number computations, and Section 5.6 summarizes my work on data diversity.

Chapter 6 gives concluding remarks.

Chapter 2: Background and Previous Work

In this chapter, previously studied techniques are briefly introduced. Chapters presenting CFCSS (Chapter 3), EDDI (Chapter 4) and ED^{4}I (Chapter 5) have their own previous work sections and discuss about the related techniques in more detail.

Correct control flow is a fundamental part of correct execution of computer programs, and we need to detect a control flow error during program execution and tolerate the fault occurred in the system. One approach to checking correct execution of the program is to have auxiliary processors such as a watchdog processor. A *watchdog processor* is *a small and simple processor that observes the bus transactions generated by the main processor and detects errors by monitoring the behavior of the system*, as shown in Fig.2.1.

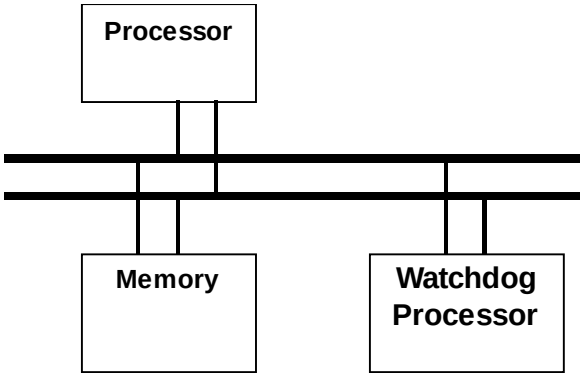


Figure 2.1. A watchdog processor monitoring the system.

One of the main issues in designing the watchdog processor is to determine which operations to check for errors, how to encode and transmit information from the main processor to the watchdog processor, and how to program the watchdog processor [Lu 82]. In *Structural Integrity Checking* (SIC) [Lu 82], a high-level control flow structure is analyzed. The SIC attaches labels to the structure and checks these labels at run time using the watchdog processor.

Another technique using the watchdog processor is *Signature Analysis*, in which a signature associated with a block of instructions is calculated and saved during compile time; then, the same signature is generated during run time and compared with the saved one. There are two types of signatures: assigned signature and derived signature. The assigned signature method assigns a unique signature to each node of the control flow graph. The derived signature method derives signatures from the binary code of the instructions.

The basic concept of the derived signature technique is shown in Fig.2.2. A signature is derived for every basic block by adding (or performing XOR operation) the binary codes of all the instructions in the basic block. This signature is computed at compile-time and embedded at the beginning of the basic block. During the program execution, the watchdog processor monitors the bus and computes the signature by capturing the binary codes of instructions when they are fetched from the memory. After it generates a signature based on the observed instruction stream of the basic block, it compares its signature with the embedded signature in the next basic block. If it sees a mismatch between the two signatures, it reports an error and initiates a recovery process.

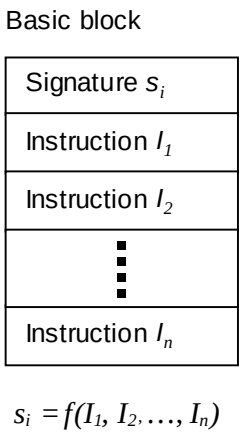


Figure 2.2. A signature derived from the instructions in the basic block.

The derived signature technique checks inter-block control flow as well as intra-block control flow of the program, and a number of methods have been studied for optimizations in terms of performance and area overhead [Shen 83][Wilken 89]. However, it inherently needs an additional hardware (such as the watchdog processor) to compute the signatures dynamically during run time. A dedicated hardware for signature analysis is specially designed for a particular system and it is not suitable for the system consisted of the COTS components; thus, the derived signature method using the watchdog processor is not appropriate for SIHFT.

On the other hand, the assigned signature method is a good candidate for SIHFT since it does not need any information for the signature from the instructions but assigns a unique signature to each basic block. During program execution, the signatures can be checked by software method, for example, additional instructions as we do in CFCSS or a *watchdog task* – a special process running concurrently with the main program [Ersoz 85]. The drawback of the

assigned signature technique is that it is not able to check the intra-block control flow because the signature is assigned to a node (basic block) in the control flow graph, not the instructions in the basic block.

Our approach to computational error and memory error as well as intra-block control flow checking is to duplicate instructions inside the basic block. Hardware duplication or Triple Modular Redundancy (TMR) is a traditional approach to fault tolerance [Pradhan 96], but our technique is software duplication using time redundancy. Time redundancy has received attention in the effort to decrease hardware cost. Time redundancy reduces hardware overhead at the expense of additional time. If the system can finish its operation before its specified time limit, the redundant time can be used for error detection or fault tolerance. Basically, the concept of time redundancy is to repeat computations to detect errors by comparing computation results. If we detect transient errors, we can tolerate them by re-computation. Many other time redundancy techniques such as assertions [Ersoz 85], block entry exit checking [Miremadi 95], and error capturing instructions [Miremadi 92] have been proposed, but time redundancy techniques have to pay inherent execution time overhead and performance loss.

In order to reduce execution time overhead occurred by time redundancy, we could utilize unused computation resources available in modern *super-scalar* architecture, which accelerates performance by exploiting *Instruction-level Parallelism* (ILP). An increasing use of ILP in processor architecture to improve performance has been popular; in addition to pipelining individual instructions, it became very attractive to fetch multiple instructions at the same time and issue them in parallel to utilize functional units whenever possible. However, the limitation of ILP in a program prevents full utilization of resources and, consequently, some functional units are idle during program execution. If we utilize these idle resources for error detection, we can reduce the performance overhead caused by additional instructions. Our technique EDDI duplicates instructions for error detection but schedules the instructions to utilize idle resources in super-scalar architecture minimizing performance overhead. (Chapter 4 describes the technique in detail).

We also have developed ED^{4}I, a new software design diversity technique based on data diversity to improve error detection capability. *Design diversity* is defined as *independent generation of two or more different software or hardware elements to satisfy a given requirement* [Avizienis 84]. Design diversity can be useful when we duplicate programs or instructions for error detection; instead of executing two identical copies of the program or instructions, executing two different versions of the program (producing the same result) will increase the error detection probability.

Hardware diversity has been applied to many commercial systems [Briere 93] including the Primary Flight Computer (PFC) system in Boeing 777 [Riter 95]. The Boeing 777 used three different processors from AMD, Intel and Motorola to avoid a common failure among three processor modules. Design diversity also has been applied to software systems [Lyu 91]. *N version programming* (NVP) [Avizienis 77][Chen 78][Avizienis 85] and *Consensus recovery block* [Scot 83][Scot 87] are examples of the design diversity in software. *N self-checking programming*, a variant of NVP, is another data diversity technique and is used in the Airbus A310 system [Laprie 90][Dugan 93].

ED^{4}I will be described in Chapter 5. In our method, a program is transformed to a new program with diverse data. If these two programs are executed simultaneously (or serially) to allow comparison of the results, errors will be detected by mismatch of the results if one of them avoids the fault while the other is corrupted by the fault.

Chapter 3: Control Flow Checking

3.1 Signature Monitoring

3.1.1 Previous Work

Signature monitoring and pure software methods have been proposed to check the control flow of a computer system. *Signature monitoring* is a method in which a signature associated with a block of instructions (one or more nodes in the control flow graph) is calculated and saved somewhere during compile time; then, the same signature is generated during run time and compared with the saved one. Signatures are assigned arbitrarily (assigned signatures) or derived from the binary code or the address of the instructions (derived signatures). Structural Integrity Checking (SIC) [Lu 82] employs an assigned signature method while Path Signature Analysis (PSA) [Namjoo 82], Signatured Instruction Streams (SIS) [Shen 83], Asynchronous SIS (ASIS) [Eifert 84], Continuous Signature Monitoring (CSM) [Wilken 89][Wilken 90], the extended-precision checksum method [Saxena 90], and On-line Signature Learning and Checking (OSLC) [Madeira 92] all use derived signatures.

Many signature monitoring techniques use dedicated hardware to compute the run-time signatures and to compare them with the saved signatures. A watchdog processor is proposed for this purpose [Lu 80][Mahmood 85]. However, if the hardware design is fixed and cannot be changed, pure software method for error detection is necessary. Examples of software methods include assertions [Andrews 79][Ersoz 85], watchdog task [Ersoz 85], Block Signature Self-Checking (BSSC) [Madeira 92], Error Capturing Instructions (ECI) [Madeira 92], timers to check the behavior of the program [Madeira 93], Available Resource-driven Control-flow monitoring (ARC) [Shuette 94], and temporal redundancy methods [Ignatushchenko 94].

3.1.2 Control Flow Checking by Software Signatures

Our new technique, *Control Flow Checking by Software Signatures* (CFCSS), differs from other signature monitoring techniques in using no watchdog processor or extra hardware to achieve inter-block control flow checking. It monitors assigned signatures for inter-block control flow checking using only instructions.

The program is divided into basic blocks. All nodes in the program graph are assigned different arbitrary numbers (signatures), which are embedded into the program during

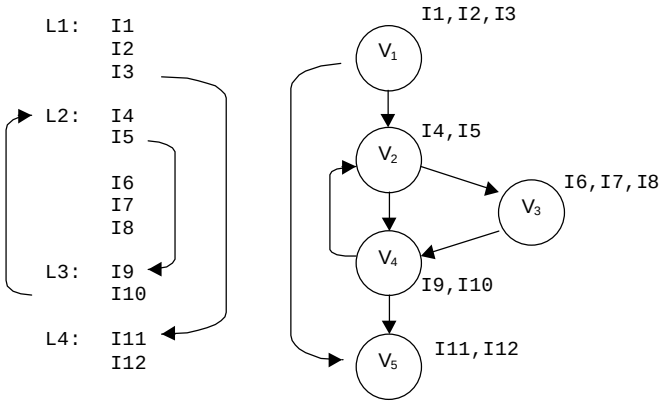
preprocessing or compile time. During program execution, a run-time signature *G* is stored in one of the general purpose registers called the *global signature register* (GSR) and compared with the stored signature of the node whenever control is transferred to a new node. The GSR is not a special or additional register in the CPU. It is one of the general purpose registers of the CPU selected by the compiler or assembler to serve as the GSR. For multiple branching cases, a run-time adjusting signature *D* is combined with *G*.

The distinctive feature of the CFCSS over previous signature monitoring techniques is that CFCSS needs no dedicated hardware such as a watchdog processor for control flow checking because it is a pure software method. A watchdog task in multi-tasking environment also needs no extra hardware, but the advantage of the CFCSS over it is that CFCSS can be used even when the operating system does not support multi-tasking.

3.2 CFCSS Algorithm Description

3.2.1 Preliminaries

Let us define $V = \{v_1, v_2, \ldots, v_n\}$ as the set of vertices denoting basic blocks, and $E = \{br_{ij} | br_{ij}$ is a branch from v_i to $v_j\}$ as the set of edges denoting possible flow of control between the basic blocks; then, a program can be represented by a *program graph*, $P = \{V, E\}$. These br_{ij} represent branch instructions, fall through execution paths, jumps, subroutine calls and returns. They are shown in Fig.3.1 as an example.



$E = \{br_{12}, br_{23}, br_{24}, br_{34}, br_{42}, br_{45}, br_{15}\}$

$V = \{v_1, v_2, v_3, v_4, v_5\}$

Figure 3.1 A sequence of instructions and its graph

Vertex v_j is in the set *suc*(v_i) if and only if br_{ij} is included in E. Similarly, vertex v_i is in the set *pred*(v_j) if and only if br_{ij} is included in E. If a program is represented by its program graph P = {V, E}, br_{ij} is *illegal* if br_{ij} is not included in E [Yau 80]. This illegal branch indicates a control flow error, which can be caused by transient or permanent faults in hardware such as the program counter, address circuits, or memory system [Lu 82].

If a node v_i receives more than two transfers of control flow, it is said to be a *branch-fan-in-node*, meaning that the number of nodes in *pred*(v_i) is greater than one. A *branch insertion* occurs when one of the instructions in the node is changed to a branch instruction as the result of an error. A *branch deletion* occurs when an error causes the branch instruction of a node to change to a non-branch instruction. As a result, the node without the branch instruction merges with the node that is adjacent to it in the memory address space.

The *xor-difference* of *a* and *b* is the result of performing the bitwise XOR operation of *a* and *b*, i.e., xor-difference = $a \oplus b$, where *a* and *b* are binary numbers.

3.2.2 The run-time signature *G*

In CFCSS, a dedicated register called the global signature register (GSR) is used for control-flow checking. The GSR contains the run-time signature *G* associated with the current node (the node that contains the instruction currently executed) in the program flow graph. Every basic block is identified and assigned a unique signature s_i when the program is compiled. Let G_i be the run-time value of *G* when the program flow is in node v_i. Under normal execution of the program, G_i should be equal to s_i. If *G* contains a number different from the signature associated with the current node, it means an error has occurred in the program.

When control is transferred from one basic block to another, a new run-time signature *G* is generated by a signature function *f* at the destination node of the branch. A *signature function f* is a function that updates *G* for the current node by using two values: the signature of the previous node (source node of the branch) and the signature of the current node (destination node of the branch). We use these two values since the source and destination nodes of the branch uniquely determine each branch in E.

Suppose that the signature function *f* is defined as $f(G, d_i) = G \oplus d_i$, and that s_s and s_d are the signatures of the source node v_s and the destination node v_d of branch br_{sd}. The signature difference d_d ($d_d = s_s \oplus s_d$) is calculated in advance at compile-time and stored in the destination node v_d. Before the branch br_{sd} is taken, *G* contains G_s (the signature s_s of the source node v_s). After the branch is taken, *G* is updated with a new value, $G_d = f(G_s, d_d)$, based on the previous value G_s and the signature difference d_d. If G_d is equal to the signature s_d of the destination node

v_d, it means there is no control flow error. On the other hand, if G_d is different from s_d, it tells us that a control flow error has occurred.

We chose the XOR operation as the signature function because the XOR operation is better than the other ALU operations for the purpose of checking or generating signatures. As XOR operations use fewer gates in the ALU than addition and multiplication, they have less chance of having an error in the ALU than addition and multiplication.

For checking the control flow, checking instructions are located at the top of each basic block; in other words, checking instructions are executed prior to the execution of the original instructions in the basic block. In Fig.3.2, the basic block B_k consists of instructions I_1, I_2, …, I_n, and additional checking instructions are located at its beginning. The checking instructions consist of two parts: the signature function that generates the run-time signature ($G = G \oplus d_k$), and the branch instruction, `br (G≠sk) error`, that compares the run-time signature with the signature of basic block B_k. In this way, a node v_k represents a basic block B_k with the checking instructions associated with B_k.

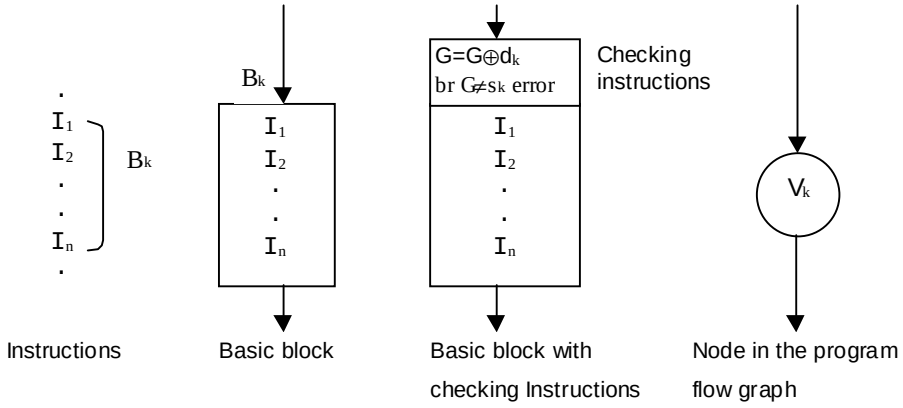


Figure 3.2 A basic block with checking instructions

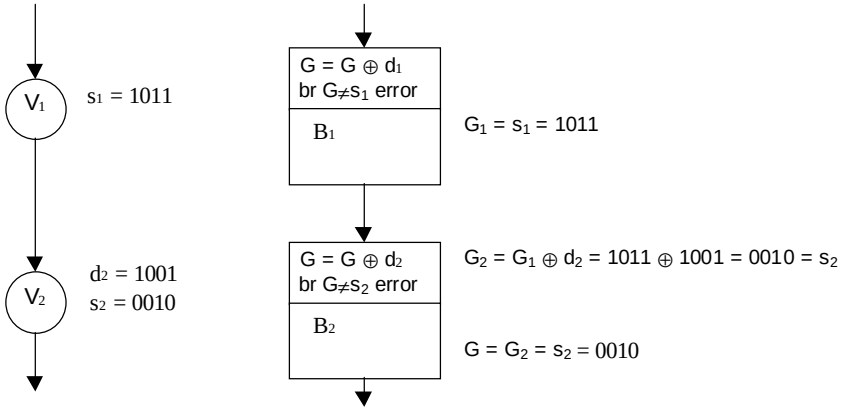


Figure 3.3 The checking instructions in a correct control flow

Figure 3.3 shows how the checking instructions work to detect errors. The control is going to be transferred from node v_1 to node v_2. G is equal to $G_1 = s_1 = 1011$, the signature of the current node v_1. After the branch br_{12} is taken, the signature function f generates the new run-time signature $G = G_2 = G_1 \oplus d_2 = 1011 \oplus 1001 = 0010$, and G is compared with the signature s_2 by the '`br (G≠s) error`' instruction. The conditional branch instruction '`br (G≠s) error`' branches to the error handler if G and s_2 are different. In contrast, Fig.3.4 shows the case where an illegal branch is taken and how it is detected by the checking instructions. Before an illegal branch br_{14} is taken, G has the value s_1. However, at node v_4, the new run-time signature $G = G_4$ is different from s_4 since G is 0101 and s_4 is 0110 ($G_4 = G_1 \oplus d_4 = 1011 \oplus 1110 = 0101 \neq s_4 = 0110$. This mismatch causes the following instruction '`br (G≠s) error`' to transfer the control to the error handling routine.

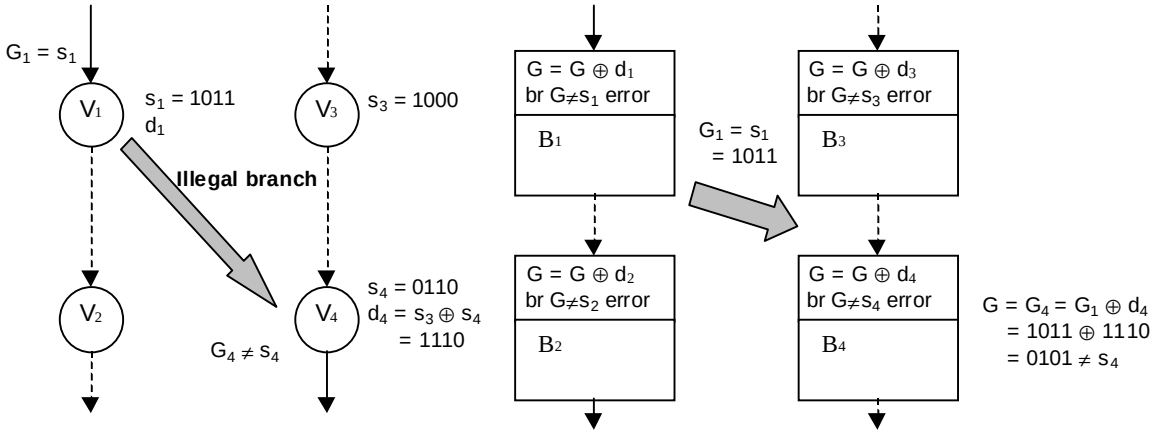


Figure 3.4 The detection of an illegal branch

3.2.3 The run-time adjusting signature *D*

It was shown that illegal branches violating the control flow could be detected by assigning unique signatures to each of the nodes in the program graph and adding signature checking instructions to them. However, there are cases where the same signature has to be assigned to multiple nodes, for example, a branch-fan-in node. In Fig.3.5, the two nodes, v_1 and v_3 have branches to the same node, a branch-fan-in node v_5. If d_5 is the signature difference between nodes v_1 and v_5 ($d_5 = s_1 \oplus s_5$), there is no problem when the branch br_{15} is taken because $G_5 = G_1 \oplus d_5 = s_1 \oplus s_1 \oplus s_5 = s_5$, which is the signature of node v_5. If the branch br_{35} is taken, however, the run-time signature G at node v_5 is not equal to s_5 as $G_5 = G_3 \oplus d_5 = s_3 \oplus s_1 \oplus s_5 \neq s_5$, if $s_3 \neq s_1$.

However, if we use $s_1 = s_3$ as the signatures, then an illegal branch from v_1 to v_4, or from v_3 to v_2, will not be detected. In order to solve the problem of assigning the same signature to multiple predecessors of a branch-fan-in node, a *run-time adjusting signature D* is introduced. After the run-time signature *G* is generated by the signature generation function, *G* is XORed with *D* to get the signature of the branch-fan-in node; thus, at the source node, *D* has to be set to the value that makes *G* equal to the signature of the destination node. Figure 3.5 illustrates an example where *D* is used in the branch-fan-in node. At node v_5, one more checking instruction $G = G \oplus D$ is added. After the signature generation function $G = G \oplus d_5$, *G* is XORed with *D* that should be determined at the source nodes v_1 and v_3. Since d_5 is initially set to the XOR-difference between s_1 and s_5 ($d_5 = s_1 \oplus s_5$), when the branch br_{15} is taken, the updated run-time signature *G* is already the same as s_5; we do not need to change *G*, thus, *D* is set to zero at v_1 ($G_5 = G_5 \oplus D = s_5 \oplus 0000 = s_5$). When the branch br_{35} is taken, the updated *G* at the first line of v_5 is $G_5 = G_3 \oplus d_5 = s_3 \oplus (s_1 \oplus s_5)$. To make *G* equal to s_5, *G* should be XORed with $s_1 \oplus s_3$ at the second line, i.e., $G = G_5 \oplus D = s_3 \oplus (s_1 \oplus s_5) \oplus (s_1 \oplus s_3) = s_5$. Therefore, at the source node v_3, *D* should be set to $D = s_1 \oplus s_3$.

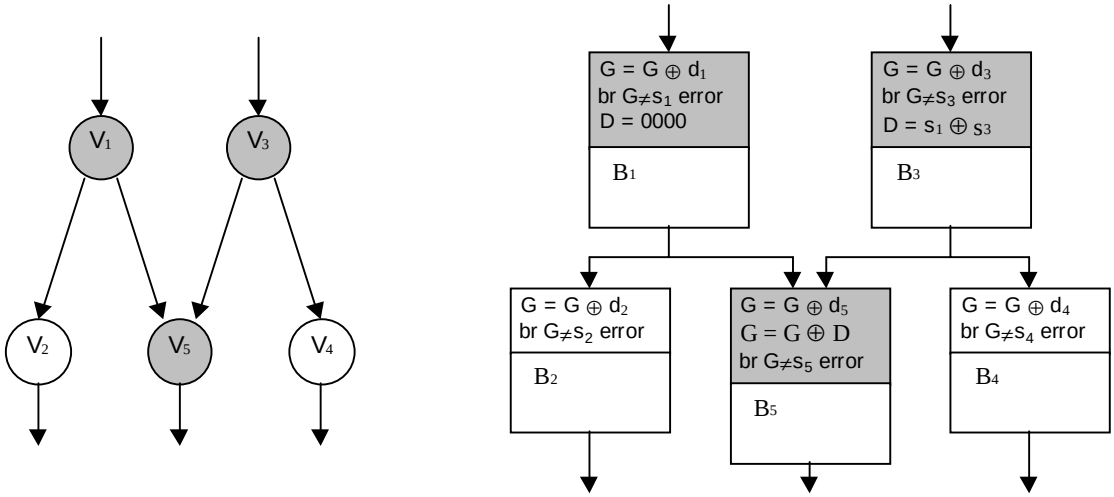


Figure 3.5 Node v_1 and v_3 have different signatures

For the branch br_{12}, *D* is not necessary as node v_2 is not a branch-fan-in node; only one branch is coming into v_2 and d_2 is equal to $s_1 \oplus s_2$. Thus, the updated *G* at node v_2 is equal to s_2. In summary, if one source node has a branch to the branch-fan-in node, the node has to have one extra instruction for *D* in the checking instructions to set *D* to the appropriate value before branching. If the branch to the branch-fan-in node is taken, *D* is XORed with *G* at the destination

node. If not, *D* is just ignored. In this way, we can assign various arbitrary numbers to the entire nodes in the program graph.

3.2.4 Algorithm Description

The following is the complete description of the algorithm that assigns a signature to each node in a program flow graph when a program is compiled.

1. Identify all basic blocks, build program flow graph and number all nodes in the program flow graph.
2. Assign a signature s_i to node v_i in which $s_i \neq s_j$ if $i \neq j$, i,j = 1,2,…,N, N is the total number of nodes in the program.
3. For each node v_j, j = 1,2,…,N
   - 3.2 For node v_j whose *pred*(v_j) is only one node v_i, the signature difference d_j is calculated as $d_j = s_i \oplus s_j$
   - 3.2 For node v_j whose *pred*(v_j) is a set of nodes v_{i1}, v_{i2}, …, v_{iM} – therefore, v_j is a branch-fan-in node - the signature difference is determined by one of the nodes (picked arbitrarily) as $d_j = s_{i1} \oplus s_j$. For node v_{im}, m = 1,2,…,M, insert an instruction $D_{im} = s_{i1} \oplus s_{im}$ into node v_{im}. This instruction should be located after '`br (G≠sj) error`' instruction in v_{im}.
   - 3.3 Insert an instruction $G = G \oplus d_j$ at the beginning of node v_j
   - 3.4 If v_j is a branch-fan-in node, insert an instruction $G = G \oplus D$ after $G = G \oplus d_j$ in node v_j
   - 3.5 Insert an instruction '`br (G≠sj) error`' after the instructions placed in step 3.3 or 3.4.

When a branch br_{ij} is taken, if the destination node v_j is not a branch-fan-in node, the run-time signature G_j is generated by the signature function $f(G_i, d_j) = G_i \oplus d_j$ and compared with the signature s_j of node v_j. If they match, it means no control flow error has occurred in taking branch br_{ij}.

In addition, when a branch br_{ij} is taken, if the destination node v_j is a branch-fan-in node, the run-time signature G_j is generated by the signature function and *D*, i.e.,

$$G_j = f(f(G_i, d_j), D_j).$$

If they match, it means no control flow error has occurred in taking branch br_{ij}.

CFCSS will detect the following types of control flow errors. They are presented as *Corollary 1 – 5* and proved in Appendix I Section 4.3.

Corollary 1. An illegal branch taken to the signature function instruction – the first line of the node – will be detected.

Corollary 2. An illegal branch taken to the instruction `br (G≠s`$_j$`) error` – the second line of the node – will be detected.

Corollary 3. An illegal branch to the body of the node where the instructions of the original basic block are located will be detected.

Corollary 4. A branch insertion inside a node will be detected if it is an illegal branch.

Corollary 5. The deletion of an unconditional branch instruction from the node will be detected.

3.2.5 Aliasing

If multiple nodes share multiple branch-fan-in nodes as their destination nodes, aliasing may occur between legal and illegal branches, and cause an undetectable control flow error. The condition for aliasing error is:

An aliasing error occurs when $d_i = s_s \oplus s_i$, $d_j = s_s \oplus s_j$, but $pred(v_i) \neq pred(v_j)$. If $pred(v_i) - pred(v_j) \neq \varnothing$, an illegal branch from a node in $pred(v_i) - pred(v_j)$ (assuming $pred(v_i) \subset pred(v_j)$) to node v_j is undetectable when that branch is taken to the location of the instruction for the signature function.

If the illegal branch is taken to any location except for the first line of the node – the instruction for the signature function – the control flow error is detected because the new run-time signature associated with the destination node is not generated. In other words, the illegal branch is detected unless it lands at the first line of the destination node that satisfies the condition described above.

However, if we assume a single bit error, and the Hamming distance between the addresses of the first instructions in nodes v_i and v_j is greater than one, this illegal branch is avoided; one bit error in the destination field of the branch instruction at node v_i cannot cause an illegal branch to the location of the first line of node v_j.

3.3 Results

3.3.1 Fault injection simulation experiment

For the simulation experiment, first the source files were compiled and assembly codes were generated. One of the branch deletions, branch creations or branch operand changes was randomly applied to the assembly code, then a machine code was generated by compiling the faulty assembly program. This machine code was executed, and we observed the behavior of the original program when it contains a branch fault.

For the second part of the experiment, CFCSS is included in the assembly source code, and the branch fault (branch deletion, branch creation, and operand change) is inserted into the code. The resulting assembly code is compiled and executed, and we observed the behavior of the program when it is augmented by CFCSS.

The graph in Fig.3.6 illustrates the percentage of faults that are not detected for both the original program and the program with CFCSS. CFCSS shows high error detection capability: In the programs without CFCSS, an average of 33.7% of the injected branching faults produced undetected incorrect outputs; however, in the programs with CFCSS, only 3.1% of branching faults produced undetected incorrect outputs. The simulation results show that CFCSS increased the error detection capability by an order of magnitude.

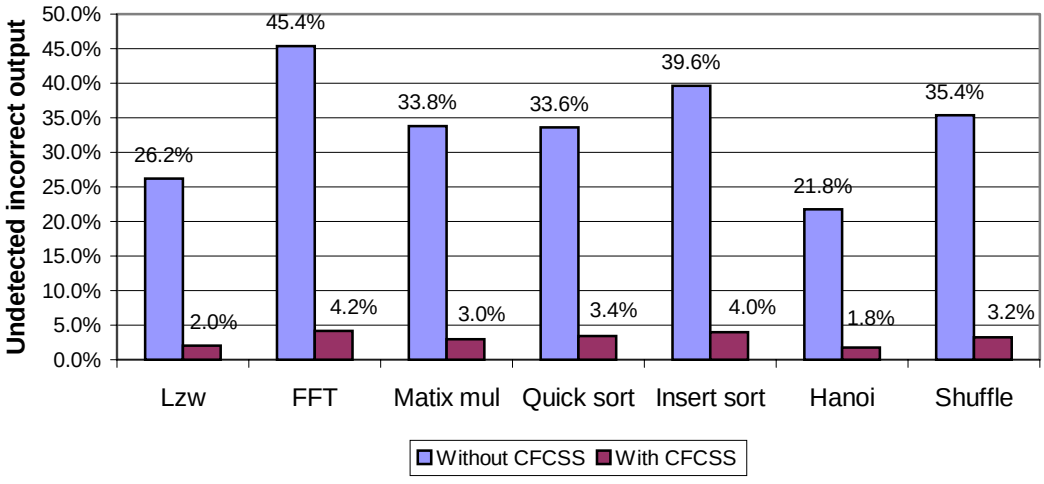


Figure 3.6 Percentage of faults that produced undetected incorrect outputs

3.3.2 Overhead

The calculation intensive programs (such as FFT) have larger size basic blocks than data analysis programs. Thus the overhead is smaller for these programs (around 30% for FFT). On the other hand, programs such as sorting and searching have small size basic blocks because they have frequent branch instructions. Therefore, the overhead of checking instructions in these programs is relatively high compared to calculation intensive programs. The details are shown in Appendix I Section 5.

3.4 Summary

CFCSS is a pure software method that checks the control flow of a program using assigned signatures. While special hardware for error checking is required in other signature monitoring techniques, CFCSS does not need the help of extra hardware for error detection; this is the advantage of CFCSS.

An algorithm was developed to assign a unique signature to each node in the program graph and to add instructions in the basic block for error detection. Signatures are embedded into the program during compilation time and compared with run-time signatures when the program is executed.

The distinctive feature of the CFCSS over previous signature monitoring techniques is that CFCSS needs no dedicated hardware such as a watchdog processor for control flow checking because it is a pure software method. CFCSS can be used even when the operating system does not support multi-tasking because signatures are embedded during the compilation time and checked by instructions.

Chapter 4: Error Detection by Duplicated Instructions

4.1 Error Detection by Duplication

4.1.1 Previous Work

One of the techniques for checking computational or memory errors is to use signatures or checksums, which contain information about the control flow in a basic block. For example, Fig.4.1 shows extended-precision checksum [Saxena 90] for control flow checking. Extended-precision checksum is the sum-total of instruction[1] through instruction[S] and is sent to the watchdog processor. After the watchdog receives the checksum value, it starts to subtract instructions, as illustrated in Fig.4.1. At the end of the block, the watchdog checks for zero. If the result is not zero, the error is reported.

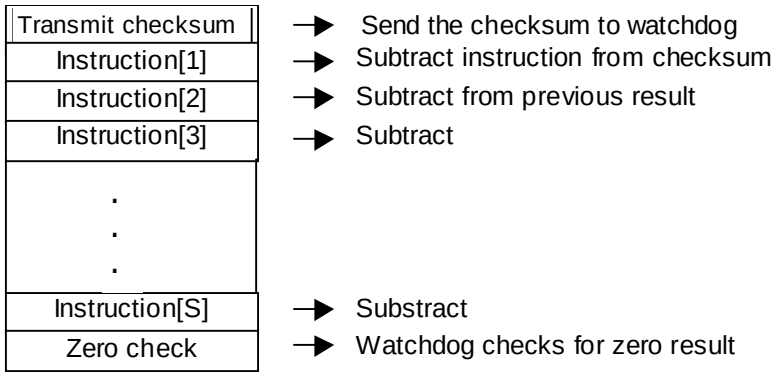


Figure 4.1 Extended-precision based control-flow checking

In a similar way, a number of techniques called signature monitoring techniques have been developed. In the signature monitoring, a signature associated with a basic block is calculated and saved in the code during compilation time; then, the same signature is generated during run time and compared with the saved one. This technique includes Structural Integrity Checking [Lu 82], Path Signature Analysis [Namjoo 82], Signatured Instruction Streams [Shen 83], Asynchronous Signatured instruction Streams [Eifert 84], Continuous Signature Monitoring [Wilken 89][Wilken 90], and On-line Signature Learning and Checking [Madeira 92]. However, most signature monitoring as well as the checksum approach still needs dedicated hardware such as a watchdog processor to compute the run-time signatures and to compare them with the saved signature; therefore, this approach is not suitable for a pure software method.

On the other hand, time redundancy has received attention because of its potential of decreasing hardware cost. Time redundancy eliminates the redundant hardware or reduces the amount of extra hardware at the expense of additional time. If the system is able to complete its computations before its specified time limit, the extra time can be used for error detection. Basically, time redundancy involves repeating computations to detect errors by comparison. It includes alternating logic [Reynolds 78], alternate-data retry [Shedletsky 78], data complementation [Takeda 80], REcomputing with Shifted Operands (RESO) [Patel 82] and time redundancy in neural network [Hsu 95].

In our new technique, *Error Detection by Duplicated Instructions* (EDDI), we use time redundancy checking for control flow error, computational error and memory error; we duplicate instructions in basic blocks with different registers and variables. Unfortunately, duplication of instructions will cause execution overhead and performance loss. In order to minimize the execution time overhead, we explore the use of idle resources for control flow checking in a super-scalar processor. Researchers have made attempts to exploit the unused resources of systems for concurrent error checking. In a multi-tasking and multi-computer environment, idle computers are used to execute replicated tasks for error checking [Fabre 88]. Application of the RESO [Patel 82] technique to the Cray-1 has been studied using unutilized resource employing machine parallelism [Sohi 89]. Utilizing the spare capacity in super-scalar and *Very Long instruction Word* (VLIW) processors to tolerate functional unit failures has been proposed in [Blough 92].

4.1.2 Error Detection by Duplicated instructions

Duplicated instructions in EDDI have no effect on the result of the program but detect errors in the system during run-time. The basic idea of error detecting instructions is duplication of the original instructions in the program but with different registers and variables. A *master* instruction is the original instruction in the source code and a *shadow instruction* is the duplicated instruction added to the source code. General purpose registers and memory are partitioned into two groups for master and shadow instructions. The registers and memory for master instructions should always have the same value as the registers and memory for shadow instructions; thus, if there has been a mismatch between a pair of registers for master and shadow instruction, an error can be detected by comparing these two register values. A comparison instruction compares the values of the two registers and invokes an error handler if they do not match. Figure 4.2 illustrates an example.

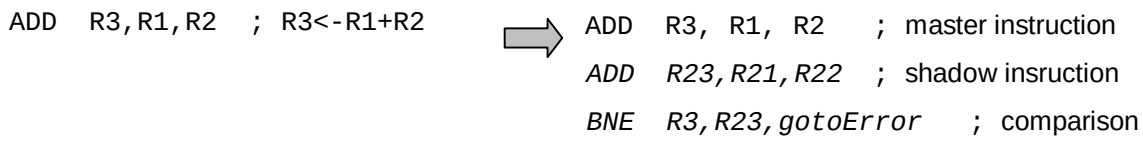


Figure 4.2 Duplicated instruction (shadow instruction) and comparison instruction

4.2 EDDI Algorithm Description

4.2.1 Preliminaries

When we duplicate instructions, we also have to determine where we insert this comparison instruction. We propose that a comparison should be performed immediately before storing register values in memory or deciding the direction of the branching instruction. Registers are used to hold temporary values for computation and the results of computation are stored in memory to free up the registers for later use. We only compare final calculation results that will be stored in memory for later use. We do not need to compare intermediate computation results that propagate and corrupt final results. If an error changes an intermediate result, the effect of the change will probably propagate down to the final computation and it will also corrupt the final result (if not, i.e., the error is masked out in intermediate computation, then we can take the final result as a correct answer). Thus, we will have different final computation results in master and shadow instructions and the error can be detected by comparison of the two registers. On the other hand, registers often hold values to resolve branching directions. An example is shown in Fig.4.3.

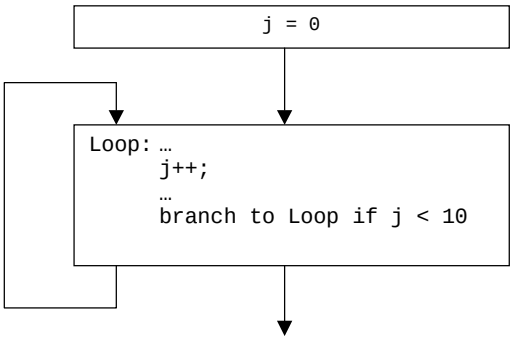


Figure 4.3 Branching at the end of the loop

After executing instructions in the block, the branching direction is determined by the value of the variable *j*. If a register (master register) holds a value for this variable *j* and is corrupted while executing instructions, it will produce an unexpected result. Thus, in order to detect erroneous branching, the value in the master register should be compared with the value in a shadow register that holds the same value for *j*.

A *storeless block* (Fig.4.4) is a sequence of instructions in which there is no store instruction except for the last instruction and the last instruction may be a store instruction or a branch instruction. Within a storeless block, shadow instructions are scheduled to maximize resource utilization by attempting to use idle resources, which are not used by master instructions. If the last instruction of a storeless basic block is a store instruction, a comparison instruction is placed before the store to compare the master and shadow register values that will be stored in memory.

```
ADD   R3, R2, R1   ⎫
SUB   R3, R3, R4   ⎬  storeless basic block
ST    0(SP), R3    ⎭
AND   R1, R1, R2   ⎫
SUB   R2, R2, R1   ⎬  storeless basic block
ST    0(SP), R2    ⎭
. . .
```

Figure 4.4 Storeless basic block example

4.2.2 Algorithm for EDDI

Let I_i be the ith instruction of the program and i = 1, 2, …, N, where N is the number of instructions in the program. An instruction I_j is *dependent* on an instruction I_i, if I_j uses the result of I_i; thus I_j will be executed after the completion of I_i. The *dependency* between I_i and I_j is denoted by a directed edge, an ordered pair (I_i, I_j). A *dependency graph* G_D is an ordered tuple (V, E) consisting of a nonempty set $V = \{I_j ; j = 1, 2, \ldots, N\}$ and a set $E = \{(I_i, I_j) ; i,j = 1, 2, \ldots, N\}$. $G_{Dk}(V_k, E_k)$ is a subgraph of G_D denoting dependencies within the kth storeless basic block, consisting of $V_k = \{I_{kj} ; j = 1, 2, \ldots, n_k\}$ and $E_k = \{(I_{ki}, I_{kj}) ; i,j = 1, 2, \ldots, n_k\}$ where n_k is the number of instructions in the kth storeless basic block. The G_{Dk}s form a partition on G_D

The algorithm for EDDI is:

1 Build a dependency graph G_D of the program

2 For each G_{Dk} {

3 Build a graph $H_{Dk}(V_k', E_k')$ where

4 $V_k' = \{I_{kj}'$; a shadow instruction of $I_{kj} \in V_k \}$

5 $E_k' = \{(I_{ki}', I_{kj}')$; $(I_{ki}, I_{kj}) \in E_k \}$

6 If I_{kn} is a store or conditional branch instruction

7 Create a comparison instruction I_{kc}

8 Create an ordered edge (I_{kn}, I_{kc}) and (I_{kn}', I_{kc}) to connect H_{Dk} to G_D

9 Schedule I_{ki}, I_{ki}', $i = 1, 2, \ldots, n_k$ and I_{kc} according to the algorithm described in Sec 4.2.3.

10 }

Figure 4.5 illustrates how to construct a subgraph H_{Dk} within a storeless basic block and attach it to G_D according to the algorithm. The portion of G_D for the storeless basic block in Fig.4.5 is shown in (a). Lines 3, 4 and 5 in the algorithm build a graph H_{Dk} in (b) and the vertices in H_{Dk} are shadow instructions $(I_1', I_2', I_3', I_4', I_5')$ corresponding to master instructions $(I_1, I_2, I_3, I_4, I_5)$ in G_D. The dependencies in H_{Dk} are inherited from the dependencies in G_{Dk}; thus the edges in H_{Dk} show the same dependencies among shadow instructions as those of master instructions. After constructing H_{Dk}, a comparison instruction denoted as I_c is added to G_{Dk} because I_5 and I_5' are store instructions and the registers should be compared before storing the results in memory. I_c compares the results of I_5 and I_5', i.e., its result depends on I_5 and I_5', thus two directed edges (I_5, I_c) and (I_5', I_c) are added connecting H_{Dk} to G_D. Now, G_D represents a new relation between master, shadow and comparison instructions. These instructions are scheduled in line 9. The scheduled instructions are shown in the following:

```
I1:   ADD   R1, R2, R3
I2:   SUB   R4, R1, R2
I1':  ADD   R21, R22, R23
I3:   AND   R5, R1, R2
I3':  AND   R25, R21, R22
I4:   MUL   R6, R4, R5
I2':  SUB   R24, R21, R22
I4':  MUL   R26, R24, R25
Ic:   BNE   R6, R26, go_to_error_handler
I5:   ST    R6
I5':  ST    R26
```

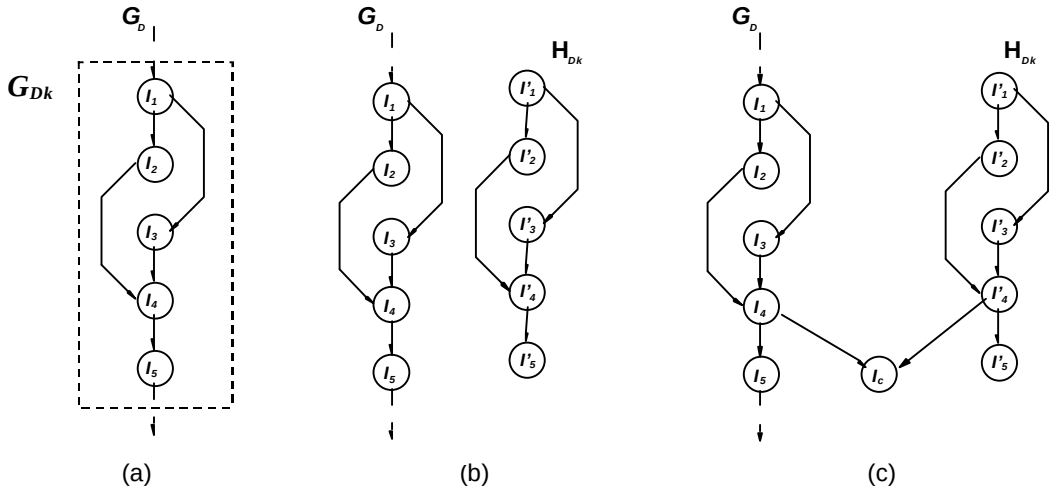


Figure 4.5 (a) Dependency graph (b) Constructing H_{Dk} (c) Adding H_{Dk} to G_D

4.2.3 Scheduling Instructions Exploiting Instruction-Level Parallelism

The shadow instructions should be scheduled with master instructions to maximize error detection coverage and to minimize time overhead using idle resources. Resource-constrained scheduling is a well-known intractable problem. We will first formalize the scheduling of master and shadow instructions and consider an exact solution method (linear integer programming). Heuristic algorithms for static and dynamic scheduling are presented in Appendix II.

G_{Dk} (V_k, E_k) is a subgraph of G_D denoting dependencies within the kth storeless basic block, where V_k = { I_{kj} ; j = 1, 2, …, n_k } is a set of n_k instructions to be scheduled in kth storeless basic block and the edge set E_k = { (I_{ki} , I_{kj}) ; i, j = 1, 2, …, n_k } represents dependencies. Let D_k = { d_j ; j = 1, 2, …, n_k } be the set of instruction execution delays and T_k = { t_j ; j = 1, 2, …, n_k } denote the *start time* for the instructions, i.e., the cycles in which the instructions start. There are n_{res} types of resources and we denote the mapping of the instructions to the unique resource types that they use by function $F_k : V_k \rightarrow \{1, 2, \ldots, n_{res}\}$. A resource constrained scheduling problem is one where the number of resources of any given type m is bounded by a set of integers { a_m ; m = 1, 2, …, n_{res} }.

A formal model of the scheduling problem under resource constraints can be achieved by using binary variables with two indices: X_k = { x_{jl} ; j = 1, 2, …, n_k ; l = 1, …, λ }, where λ represents an upper bound on the latency of kth storeless basic block. A binary variable x_{jl} is 1 only when the instruction I_{kj} starts in step l of the schedule. Then, the scheduling can be formulated as a set of equations (from (1) to (5)) that have to be satisfied. The first four equations

are the resource-constrained minimum latency scheduling equations and described in detail in [33]. Briefly, equation (1) tells us that the start time of each instruction is unique and equation (2) shows us that the dependency relations in G_{Dk} must be satisfied. The resource constraint must be satisfied at every cycle. An instruction I_{kj} is executing at cycle l when $\sum_{m=l-d_j+1} x_{jm} = 1$. The number of all instructions at cycle l of type k must be less than or equal to a_k. This is shown in (3). We add one more condition in (5) and restrict the number of master instructions to being always greater than the number of shadow instructions until step λ-1 in the kth storeless basic block. The number of master and shadow instructions should be equal at step λ, when scheduling is completed.

minimize $\sum_j t_j$ such that

(1) $$\sum_l x_{jl} = 1,\ j = 1,2,...,n$$

(2) $$\sum_l l \cdot x_{il} - \sum_l l \cdot x_{jl} - d_j \geq 0, i, j = 1,2,...,n_k : (I_{ki}, I_{kj}) \in E_k$$

(3) $$\sum_{j:F(I_j)=m} \sum_{q=l-d_j+1}^{l} x_{jq} \leq a_m, m = 1,2,...,n_{res}, l = 1,...,\lambda$$

(4) $$x_{jl} \in \{0,1\}, j = 1,2,...,n_k, l = 1,...,\lambda$$

(5) $$\sum_{i:\text{master instruction}} \sum_{q=1}^{l} x_{iq} - \sum_{j:\text{shadow instruction}} \sum_{q=1}^{l} x_{jq} > 0, l = 1,...,\lambda - 1$$

4.3 Results

4.3.1 Fault Injection Simulation Experiment

The benchmark programs are compiled by gcc [Stallman 98] with level 2 optimization option (O2). Most compiler optimization techniques that do not involve a space-speed tradeoff (such as loop unrolling and function inlining) are performed in level 2 optimization. The compiler was modified to allocate registers for master and shadow instructions. First, the compiler produced the assembly code of the program. Then, shadow instruction and comparison instructions are added in assembly code by our post-processor and the resultant assembly code with EDDI is

compiled into object code by an assembler. Our target machines are SGI Indigo with a MIPS R4400 processor and SGI Octane that employs the 4-way super-scalar R10000 Mips processor.

In the fault injection simulation, the source files are compiled and the target machine codes are generated without error detection instructions. A fault injector forced a bit flip in the code segment of the machine code in which the location of the bit flip is determined by a random number generator. We execute the machine code and observe the behavior of the program after fault injection. In the second part of the experiment, EDDI is included in the benchmark programs by the compiler postprocessor. We injected a bit flip into the generated machine code, executed the corrupted machine code, and observed the behavior of the program.

EDDI shows high error detection capability: approximately 98.5% of injected faults in most programs produced incorrect outputs detected. The result is illustrated in Fig.4.6.

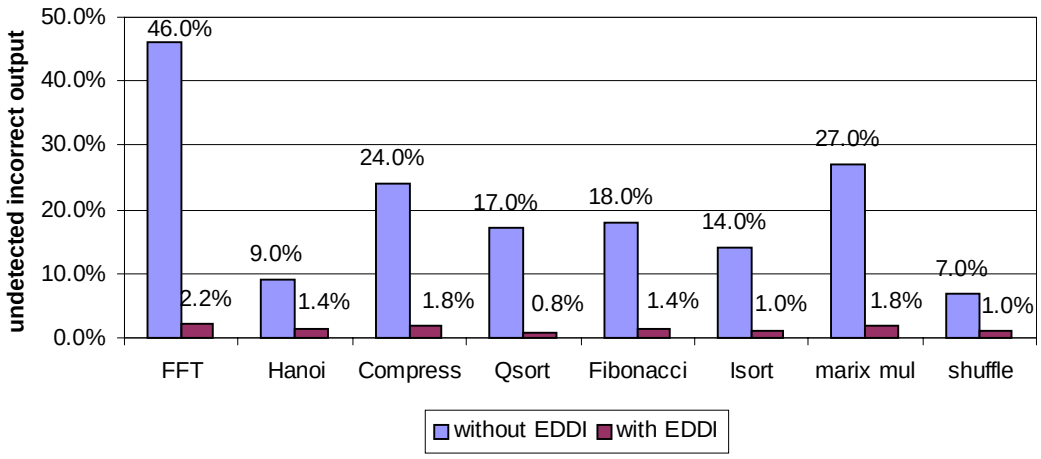


Figure 4.6 Percentage of faults that produced incorrect outputs without being detected

4.3.2 Overhead reduction by instruction-level parallelism

Since extra instructions are added to the original assembly code, the program with EDDI suffers from an increase in code size and loss of performance compared to the original one. The execution time overhead is shown in Fig.4.7.

Notice that for most of the programs, the overhead in execution time is less than 100%. As we have duplicated instructions, the overhead might be close to or greater than 100%. There are two reasons for this low performance overhead in time. First, shadow instructions fill the empty slot of the pipeline; thus, we achieve higher utilization of the processor resources. Second, shadow instructions add more parallelism to the program. A 4-way super-scalar processor can exploit this parallelism better than a 2-way super-scalar processor. This effect can be seen in

Fig.4.7 where the execution times for running the programs in two different processors are shown. We can observe that the execution time overhead in the R10000 is less than the one in the R4400; a 4-way issue machine has less time overhead than a 2-way issue machine. EDDI has less execution time overhead in processors that employ more aggressive super-scalar architectures.

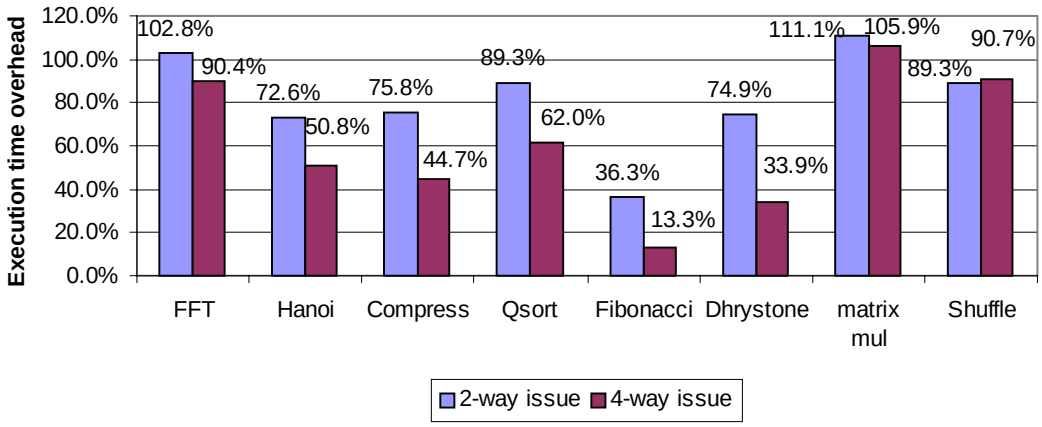


Figure 4.7 Execution time overhead in R4400 (2-way issue) and R10000 (4-way issue)

4.4 Summary

The EDDI technique is a pure software method that achieves high fault coverage in the computer system in which adding any extra hardware or modifying the existing hardware is not possible. In EDDI, we duplicate instructions during compilation and use different registers and variables for the new instructions.

In the fault injection simulation experiment, the result shows that approximately 1.5% of injected faults produced incorrect results in eight benchmark programs with EDDI, while on average, 20% of injected faults produced undetected incorrect results in the programs without EDDI. Based on the experiment result, we showed that EDDI could provide over 98% error detection coverage without any extra hardware. This pure software technique is especially useful when designers cannot change the hardware system but they need dependability in the computer system.

In order to reduce the performance overhead caused by the instructions for detecting errors, our technique schedules the instructions by exploiting instruction-level parallelism available in super-scalar processors. The more instructions the processor can execute at the same time, the less performance overhead we can achieve.

Chapter 5: Error Detection by Diverse Data and Duplicated Instructions

5.1 Data Diversity for Error Detection

5.1.1 Diversity overview

Design diversity has been proposed in the literature to increase the reliability of the system. *Design diversity* is defined as *the independent generation of two or more different software or hardware elements to satisfy a given requirement* [Avizienis 84]. *Common-mode failures* are defined as *the failures that affect more than one module at the same time*, and they have often been the targets of design diversity in hardware [Lala 94]. Design diversity also has been used in software systems for fault tolerance. *N version programming* (NVP) [Avizienis 77][Chen 78][Avizienis 85] is one of the common fault tolerant software schemes. It proposes parallel execution of *N* independently developed and functionally equivalent versions with adjudication of their outputs by a voter.

Consensus recovery block [Scot 83][Scot 87], a hybrid system combining NVP and recovery block, is another example in which diversity is used in software. A variant of NVP, *N self-checking programming*, also employs a design diversity concept and is used in the Airbus A310 system [Laprie 90][Dugan 93].

5.1.2 ED^{4}I: New approach to data diversity

Our approach to data diversity is different from the previous one. Our target is not software faults but hardware faults – both permanent and transient faults in the system. The motivation of our research comes from SIHFT, which is necessary when we cannot modify hardware design but need fault tolerance in the system. Our technique is a pure software technique for hardware fault detection; it is not necessary to change the hardware to add error detection capability; thus, it can be easily implemented in different hardware systems at low cost.

We have developed an algorithm to transform a program to a new program with diverse data without changing the complexity of the original program. In the transformed program, the values of all variables and constants are *k* times greater than the values of the corresponding variables in the original program. Depending on the diversity factor *k*, the original and the transformed program may use different parts of the hardware and propagate fault effects in

different ways; therefore, if the two different programs produce different outputs due to the fault, we can detect the fault by comparing their outputs.

The diversity factor *k* should satisfy two goals. The primary goal is to guarantee *data integrity*; that is, *the probability that the two programs do not produce identical erroneous outputs*. The secondary goal is to maximize *fault detection probability*. However, the factor *k* should not cause an overflow in the system. In order to determine the optimum value of *k*, we have developed an analysis technique based on design diversity metric [Mitra 99], which was used to quantify diversity among several designs. We use this metric to measure the diversity between the original and the transformed programs.

5.2 Transformation Algorithm Description

5.2.1 Expression and branching condition transformation

By defining $V = \{v_1, v_2,\ldots,v_n\}$ as *the set of vertices denoting basic blocks*, and $E = \{(i,j) \mid (i, j)$ is a branch from v_i to $v_j\}$ as *the set of edges denoting possible flow of control between the basic blocks*, a program can be represented by a *program graph*, $P_G = \{V, E\}$.

If *x* is *k* times greater than *y*, *x* is *k-multiple* of *y*. *Program transformation* transforms a program *P* to a diverse program *P′* in which all variables and constants have *k*-multiple of the original values when the program is executed. It consists of two transformations: *expression transformation* and *branching condition transformation.* The expression transformation changes the expressions in *P* to new expressions in *P′* so that the value of every variable or constant in the expression of *P′* is always *k*-multiple of the corresponding value in *P*. Since the values in *P′* are different from the original values, when we compare two values in the conditional statement, the inequality relationship might need to be changed. For example, the conditional statement `if (i < 5)` in *P* needs to be changed to `if (i > -10)` in *P′* when *k* is -2. Otherwise, the control flow determined by the conditional statements in *P′* would be different from the control flow in *P*, and the computation result from the diverse program would not be the *k*-multiple of the result from the original program. Therefore, the branching condition transformation adjusts the inequality relationship in the conditional statement in *P′* so that the control flows in *P* and *P′* are identical.

The program transformation generates a new program with a program graph $P_G' = \{V', E'\}$ that is isomorphic with P_G, but all the variables and constants in *P′* are *k*-multiples of the ones in *P*; therefore, *P′* can possibly use a part of the hardware that is not used by *P*.

Let us denote S and S' as sets of variables in P and P' respectively, and define n as *the number of vertices (basic blocks) executed*; then, $S(n)$ and $S'(n)$ are:

$S(n)$: *a set of values of the variables in S after n vertices are executed,*

$S'(n)$: *a set of values of the variables in S′ after n vertices are executed.*

Then, the program transformation should satisfy:

(1) P_G and P_G' are isomorphic.

(2) $k \cdot S(n) = S'(n)$, for $\forall n > 0$

(where $k \cdot S(n)$ is obtained by multiplying all elements in $S(n)$ by k).

The condition in (1) and (2) tells us that the control flow in the two programs should be identical. The condition in (2) requires that all the variables in the transformed program are always k-multiples of those in the original program.

In the expression transformation, we build a parse tree for every expression in P and produce a new expression by recursively transforming the parse tree. In the branching condition transformation, we examine the inequality relationship in the conditional statements and modify it according to the value of k. Then, the transformed program always satisfies the conditions stated in (1) and (2). (The formal description for the program transformation algorithm as well as the correctness and proofs of the algorithm is presented in Appendix III)

Figure 5.1 and Fig.5.2 show us an example in which a sample program P is transformed to a new program P' with the diversity factor $k = -2$.

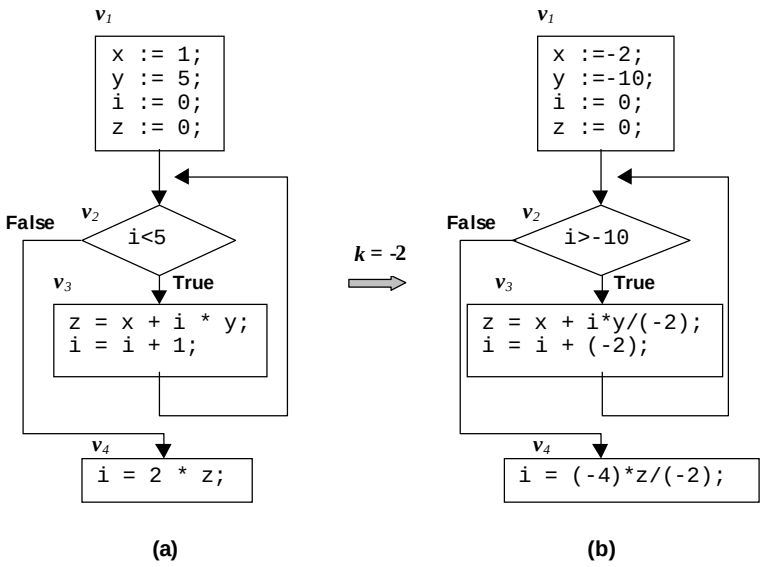


Figure 5.1 (a) The original program (b) The transformed program with k = -2.

```
x := 1;                          x := -2;
y := 5;                          y := -10;
i := 0;          k = -2          i := 0;
while (i < 5) {                  while (i > -10) {
    z = x + i * y;                   z = x + i * y /(-2);
    i = i + 1;                       i = i + (-2);
}                                }
i := 2 * z;                      i := (-4) * z / (-2);
```

(a) (b)

Figure 5.2 (a) The original program (b) The transformed program with k = -2.

5.2.2 Determination of *k*

A program can be transformed to a diverse program with the factor k by the expression and branching condition transformation. Modifying the factor k will affect how diverse the transformed program is and may change the error detection capability; therefore, we need to quantify the diversity of the program to determine the optimum value of k.

Researchers have studied techniques to quantify diversity in multiple designs and which technique should be used to measure the diversity [Eckhardt 85][Litt 89][Lyu 94]. They use random variables Π and X to represent arbitrary programs and arbitrary inputs, and the probability that Π fails on X is calculated. Our diversity metric is somewhat different because we need to compute the probability that a program fails due to a hardware fault in the system.

In our approach, fault detection probability and data integrity quantify diversity between two programs running on the same hardware. Let us define:

X *the set of corresponding inputs to a functional unit* h_j *in the system when an integer program P executes*

X' *the set of corresponding inputs to a functional unit* h_j *in the system when an integer program P′ executes*

x *a particular input to* h_j *produced by P and* $x \in X$

x' *a particular input to* h_j *produced by P′ and* $x' \in X'$

$|X|$ *the number of elements in the set X*

Then, the output y of h_j with input x and the output y' of h_j with input x' should satisfy the relationship $y' = k \cdot y$ unless a fault occurs in h_j without any overflow. In the presence of a fault f_i in h_j, let us define:

E_i *the subset of X that contains inputs producing incorrect outputs in* h_j

E_i' *the subset of* E_i *that contains inputs producing incorrect outputs that erroneously satisfies the relationship* $y' = k \cdot y$ *in the presence of a fault* f_i

Then, $|E_i - E_i'|$ is the number of incorrect outputs that have the relationship $y' \neq k \cdot y$, so that we can detect the error by mismatch.

Therefore, the fault detection probability in h_j by ED^{4}I is,

$$C_j(k) = \sum_i \Pr\{f_i\} \Pr_i\{y' \neq ky\} = \sum_i \Pr\{f_i\} \left(\frac{|E_i - E_i'|}{|X|} \right). \tag{3}$$

Moreover, we define the data integrity in h_j as:

$$D_j(k) = \sum_i \Pr\{f_i\} \left(1 - \frac{|E_i'|}{|X|} \right) \tag{4}$$

If we assume a uniform distribution for all faults,

$$\begin{aligned} C_j(k) &= \sum_i \Pr\{f_i\} \left(\frac{|E_i - E_i'|}{|X|} \right) = \sum_i^{N_f} \frac{1}{N_f} \left(\frac{|E_i - E_i'|}{|X|} \right) \\ D_j(k) &= \sum_i^{N_f} \Pr\{f_i\} \left(1 - \frac{|E_i'|}{|X|} \right) = \sum_i^{N_f} \frac{1}{N_f} \left(1 - \frac{|E_i'|}{|X|} \right) \end{aligned} \tag{5}$$

where N_f denotes the total number of faults in h_j.

For various values of k, $C_j(k)$ and $D_j(k)$ of h_j can be obtained by either the probabilistic analysis or simulation method. They are weighted by execution frequency u_j of h_j in the program execution profile and added to obtain $C(k)$ and $D(k)$ for a particular program.

$$C(k) = \sum_j u_j C_j(k)$$
$$D(k) = \sum_j u_j D_j(k) \quad (6)$$

An optimal value of k is the value that satisfies two goals: the primary goal is to maximize the data integrity $D(k)$ and the secondary goal is to maximize the fault detection probability $C(k)$. Data integrity is more important because it guarantees no undetected errors. For any given program, we first analyze the data integrity and fault detection probability of each functional unit of the system for various values of k. Next, we combine these values to create an optimal value of k for the transformed program by looking at execution profiles of the programs.

In Secs. 5.2 to 5.5 of Appendix III, we analyze data integrity and fault detection probability of functional units: bus, a ripple carry adder, a carry look-ahead adder, a multiplier, and a shifter. In our analysis, we consider integers from –5 to 5 for the value of k. Integers whose absolute values are greater than 5 are not considered for k in this paper because they have higher probability of overflow than the values considered. The analyzed data integrity and fault detection probability for functional units will be used later to determine the optimum value of k for benchmark programs using execution profiles of the programs. Table 5.1 shows the data integrity $D(k)$ calculated with various values of k in benchmark programs. In the table, shaded areas indicate the highest data integrity in a row. Then, in Table 5.2, we show the highest fault detection probability under the condition that data integrity is the highest.

Table 5.1. Data Integrity $D(k)$ calculated with various values of k in benchmark programs. Note that Shaded areas indicate the highest data integrity in a row.

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
I-sort	1.0000	1.0000	1.0000	1.0000	0.9641	1.0000	0.9998	1.0000	0.9998
Q-sort	1.0000	1.0000	1.0000	1.0000	0.9656	1.0000	0.9998	1.0000	0.9998
Lzw	0.9940	0.9940	0.9940	0.9940	0.9625	0.9940	0.9939	0.9940	0.9939
Fib	1.0000	1.0000	1.0000	1.0000	0.9650	1.0000	1.0000	1.0000	1.0000
M-mul	1.0000	1.0000	1.0000	1.0000	0.8714	1.0000	0.9999	1.0000	0.9999
Shuffle	0.9900	0.9900	0.9900	0.9900	0.9580	0.9900	0.9888	0.9900	0.9887
Hanoi	1.0000	1.0000	1.0000	1.0000	0.9782	1.0000	0.9999	1.0000	0.9998

Table 5.2. Fault detection probability *C(k)* calculated with various values of *k* in bench mark programs. Shaded areas indicate the highest fault detection probability under the condition that data integrity is the highest (shaded areas in Table 5.1).

	k = -5	*k* = -4	*k* = -3	*k* = -2	*k* = -1	*k* = 2	*k* = 3	*k* = 4	*k* = 5
I-sort	0.6491	0.6621	0.6511	0.6661	0.7068	0.6134	0.5692	0.6262	0.5837
Q-sort	0.6479	0.6607	0.6505	0.6646	0.7089	0.6134	0.5680	0.6256	0.5824
Lzw	0.6642	0.6763	0.6678	0.6796	0.7373	0.6346	0.5837	0.6484	0.5986
Fib	0.6710	0.6836	0.6734	0.6872	0.7370	0.6386	0.5900	0.6536	0.6053
M-mul	0.6617	0.6813	0.6647	0.6766	0.6691	0.6359	0.5884	0.6480	0.6023
Shuffle	0.5586	0.5705	0.5654	0.5751	0.6187	0.5258	0.4838	0.5328	0.4946
Hanoi	0.6699	0.6934	0.6769	0.6823	0.7679	0.6519	0.5903	0.6638	0.6051

5.3 Overflow Handling

The primary cause of the overflow problem is the fact that in the transformed program (after multiplication by *k*), the size of the resulting data may be too big to fit into the data word size of the processor. Previous hardware techniques like RESO [Patel 82] eliminated this overflow problem by adding extra bit-slices in the datapath. This solution does not work for a SIHFT technique like ED^{4}I because we cannot control the hardware design.

One possible solution is scaling; before the execution of the original program, we can scale up the precision of the data or scale down the range of the data in order to avoid overflow. In the first solution, we scale up the precision of the data to multiple precision; data types such as 16-bit single-precision integers can be scaled up to 32-bit double-precision integer data type. Thus, modification of the program is required. However, in the second solution, we scale down the original input data to avoid overflow and do not modify the program.

As an example of scaling up precision, suppose a 32-bit integer data type in a particular machine. Then, a 64-bit integer x can be represented by combination of two 32-bit double-precision integers: x_l representing the lower 32 bits of x and x_h representing the higher 32 bits of x. Suppose y is also represented by y_l and y_h. We need two additions when we add two numbers x and y; addition of lower bit words and addition of higher bit words. A pseudo code for this addition is:

```
z = x + y  → add zl, xl, yl     ; the first addition for lower bit word
             addc zh, xh, yh    ; the second addition for higher bit word
```

In the code above, `addc` is an addition with a carry bit generated by the previous addition. The first addition in (a) reads two lower bit words (x_l and x_h) from registers, performs an addition and stores a result back in the lower bit word of the result z. The second addition in (b) reads two higher bit words (y_l and y_h) from the registers, performs an addition with the carry generated from (a), then stores the result in the higher bit word of the result z. Finally, z can be represented by z_l and z_h.

Similarly, scaling up the data precision requires modification of other arithmetic computations such as subtraction, multiplication and division.

Other techniques for handling overflow such as data range check at compilation will be discussed in more detail in Section 4 of Appendix III.

5.4 Results

Section 5 in Appendix III shows that we have different values of k that maximize the fault detection probability and data integrity in different functional units. For example, the bus has the highest fault detection probability when k = -1, but the array multiplier has the highest fault detection probability when k = 4. Therefore, programs, such as matrix multiplication that use a multiplier extensively, will need 4 or -4 for the value of k to optimize the highest fault detection probability. However, programs such as sorting that use memory buses heavily to communicate with memory for loading and storing data need –1 for the value of k to maximize its fault detection probability. Hence, an execution profile of a program showing execution frequencies of each functional unit is necessary to determine k for a particular program.

We examined the execution profiles of seven benchmark programs; then, based on $C_j(k)$ and $D_j(k)$ obtained for each functional unit, we calculated the optimal values for k using equation (6) with the benchmark programs. Table 5.3 shows the optimum value of k estimated for each benchmark program. In six out of seven benchmark programs, -2 is the best choice for k because the fault detection probability is maximized with that value under the condition that data integrity is the highest.

Table 5.3. Optimum value of k determined for each benchmark program.

	I-sort	Q-sort	Lzw	Fib	Mat-mul	Shuffle	Hanoi
Optimum k	-2	-2	-2	-2	-4	-2	-2

After we determined the optimal values for the programs, we performed simulation experiments to verify the estimated results. We built a MIPS simulator for fault injection experiment. It reads

an assembly program, executes instructions, and emulates the functional units in gate level. We injected a stuck fault into a randomly chosen node of the functional units, executed the benchmark program in the simulator, and generated output patterns that might be corrupted if the inputs had provoked the fault. We generated two output patterns: one from the original program and the other from the transformed program. We compared their outputs, calculated the fault detection probability and data integrity, and averaged them over all single stuck faults simulated.

The detailed simulation results are shown in Tables 11 and 12 of Appendix III. From these tables, readers can verify that the simulation results are identical to Table 5.3; this demonstrates that our analysis agrees with the simulation results.

The execution profiles of those six programs show that adders are the most frequently used functional units in the programs; thus, the transformation with k = -2 might be the most desirable choice for the programs that use adders extensively. However, program profiling and fault injection simulations in functional units are necessary to get the exact optimum value for k because data integrity and fault detection probability depends on the implementation of functional units.

5.5 Floating Point Numbers

Among several representations for non-integers, only *floating point* representation has gained widespread use [Hennessy 96]. In this floating point system, a computer word is divided into three parts: a sign, an exponent and a fraction. As an example, in the IEEE standard 754 [IEEE 85], single-precision numbers are stored in 32 bits: 1 bit for the *sign s*, 8 bits for the *exponent e*, and 23 bits for the *fraction f*. The exponent is a signed number represented using the bias method with a bias of 127. The fraction f is the fraction part of the *mantissa*, i.e., *mantissa* = 1.f. Thus, the number being represented is $s \times 1.f \times 2^{e-127}$ [IEEE 85].

Having three parts in one word in floating point representation creates some difficulty in applying ED^{4}I to floating point numbers because multiplying by k may not shift nor change many bits in one word. Our approach to solve this problem is to find a value of k for the fraction and the exponent separately and combine those values to get the best value for k.

For the fraction, we choose $k = \frac{3}{2}$ because it satisfies the following criteria: (1) guaranteed data integrity in the fraction, (2) no underflow in the transformed program, (3) low probability of overflow, and (4) ability to easily locate an error. The details are explained in Appendix III Section 7.1.

For the exponent, we need two values for *k*; thus, we use two transformations with these two values to guarantee data integrity in the exponent. As an example for the 8-bit exponent in the IEEE standard 754, we choose $k = 2^{10101010_2}$ for the first transformation and $k = 2^{01010101_2}$ for the second transformation. In this case, we have assumed that these *k*'s will not cause an overflow. (However, if there is an overflow, we can use the scaling techniques.) The proof that data integrity is guaranteed using these two values for the transformations is shown in Appendix III Section 7.2.

We have selected $k = \frac{3}{2}$ for the fraction and chosen $k = 2^{10101010_2}$ and $k = 2^{01010101_2}$ for the exponent. As a result, we use $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ for the first transformation and $k_2 = -\frac{3}{2} \times 2^{01010101_2}$ for the second transformation. Then, as shown in Appendix III Section 7.3, comparing the results from the original program with results from the first transformed program, and subsequently with results from the second transformed program will guarantee data integrity.

5.6 Summary

In this chapter, we have developed a new approach to data diversity, ED^{4}I, to detect hardware faults in the system without any hardware modification. Unlike previous techniques, our technique is a pure software method that can be easily implemented in different systems. ED^{4}I transforms a program to a new program with the same functionality but with diverse data. We can detect the fault by comparing the results of the two programs.

The factor *k* determines how diverse the transformed program is. To determine the optimum value of *k*, we have used data integrity and fault detection probability as metrics to quantify the diversity of the transformed program. Based on these metrics, we demonstrated how to choose the optimum value of *k* for creating data diversity. For example, we have observed that –2 is the optimum value for *k* in six out of seven benchmark programs we have simulated.

ED^{4}I is a SIHFT technique that detects both permanent and temporary errors by executing two "different" programs (with the same functionality) and comparing their outputs. If we cannot modify hardware design (such as in COTS) and can use only software technique for error detection, we can use ED^{4}I to improve the data integrity and availability of the system.

Chapter 6: Concluding Remarks

This dissertation presented my contributions to development of three new software implemented hardware fault tolerance techniques (SIHFT). The major advantage of our SIHFT technique is that we do not need any hardware modification for error detection; we can easily implement error detection capability in the program during compilation. We first divided a program into basic blocks and constructed a program flow graph; then, using only software techniques, we checked the control flow among the basic blocks and subsequently, we checked computational and memory errors inside the basic blocks. In this way, we can detect the abnormal behavior of the program caused by transient or permanent fault without any extra hardware for error detection.

In order to demonstrate the effectiveness of our techniques, we applied our SIHFT techniques to real space experiment: The Stanford ARGOS project [Shirvani 00] is an experiment that is carried out on the computing test-bed of *Unconventional Stellar Aspect* (USA) experiment on the *Advanced Research and Global Observations Satellite* (ARGOS). The satellite was launched in February 1999 and has a Sun-synchronous orbit with mission life of three years.

The EDDI and CFCSS were implemented in application programs (FFT, insert sort, and quick sort) and were executed in the ARGOS satellite. Preliminary results of the ARGOS satellite experiment show that we can achieve very high error detection coverage if the programs have both EDDI and CFCSS. For a 136 day period, we have observed 203 SEUs occurring in the application programs on the COTS board, and 198 out of 203 errors were detected by CFCSS and EDDI.

Without these error detection schemes, the SEUs in orbit would have corrupted the correct functionality of the programs running in the satellite. This result proves that our SIHFT techniques are very effective for error detection when COTS components are used in space applications without any assistance from fault tolerance hardware. The COTS components are more vulnerable to SEU than radiation hardened components, but employing SIHFT techniques can increase COTS components availability and keep them to operate continuously even if they have errors in low radiation environment.

Chapter 7: References

[Andrews 79] Andrews, D., "Using executable assertions for testing and fault tolerance," *9th Fault-Tolerance Computing Symp.*, Madison, WI, June 20-22, 1979.

[Caldwell 97] Caldwell, D. W. and D. A. Rennels, "A Minimalist Hardware Architecture for Using Commercial Microcontrollers In Space," *16th Digital Avionics Systems Conference*, Oct. 28-30, 1997.

[Chang 91] Chang, P.P., et al., "IMPACT: An Architectural Framewrk for Mulitple-Instruction Issue Processors," Proc. 18th Int'l Symposium on computer Architecture (ISCA) 19(3), pp.266-275, 1991.

[Chen 78] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," IEEE 8th FTCS, pp. 3-9, 1978.

[Cusick 86] J. Cusick, "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors," IEEE Transactions on Nuclear Science, Vol. NS-32, No. 6, pp.4206-4211, Dec. 1986.

[Dale 95] Dale, C.J. et al. "Fiber optic data bus space experiment on board the Microelectronics and Photonics Test Bed (MPTB)" *Proceedings of the SPIE – The International Society for Optical Engineering* (1995) vol. 2482 pp. 285-93, April 1995

[Eifert 84] Eifert, J. B. and J. P. Shen, "Processor Monitoring Using Asynchronous Signatured Instruction Streams," *Digest of Papers, 14th Annual Int'l Conf. on Fault-Tolerant Computing*, pp. 394-399, June 1984.

[Ersoz 85] Ersoz, A., D. M. Andrews, and E. J. McCluskey, "The Watchdog Task: Concurrent Error Detection Using Assertions," Stanford University, Center for Reliable Computing, TR 85-8.

[Fabre 88] J. C. Fabre, Y. Deswarte, J-C. Laprie, and D. Powell, "Saturation: Reduced idleness for improved fault-tolerance," IEEE 18th FTCS pp. 200-205, June 1988.

[Furtado 96] Furtado, P., H. Madeira, "Fault Injection Evaluation of Assigned Signatures in RISC Processors," *Proc., Second European Dependable Computing Conference*, Taormina, Italy, pp. 55-72, October 1996.

[Golombek 99] Golombek, M.P. et al. "Overview of the Mars Pathfinder mission: launch through landing, surface operations, data sets, and science results," *JOURNAL OF GEOPHYSICAL RESEARCH* American Geophys. Union, 25 April 1999. vol.104, no.E4, pp. 8523-53

[Hennessy 96] Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second edition, 1996.

[Holland 96] Holland, A.D. et al. “The bulk damage monitor on MPTB,” *Proceedings of the SPIE – The International Society for Optical Engineering*, vol. 2811, pp. 136-41, Aug. 1996

[Hopkins 78] A. L. Hopkins, Jr., et al., “FTMP – A highly reliable fault-tolerant multiprocessor for aircraft,” Proc. IEEE, vol. 66, pp. 1221-1239, Oct. 1978

[Hsu 95] Yuang-Ming Hsu et al. "Time redundancy for error detecting neural networks," *Proceedings IEEE International Conference on Wafer Scale Integration*, pp. 111-121, Jan. 1995.

[Huang 84] Huang, K.H. and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE TRANSACTIONS ON COMPUTERS* vol.C-33, no.6, pp. 518-28, June 1984

[Ignatushchenko 94] Ignatushchenko, V. V., et al., “Effectiveness of temporal redundancy of parallel computational processes,” *Automation and Remote Control*, Vol. 55, No. 6, pt. 2, pp. 900-911, June 1994.

[Johnson 91] M. Johnson, “Superscalar Microprocessor Design,” Englewood Cliffs, NJ, Prentice Hall.

[Kanawati 96] G. A. Kanawati, V.S.S. Nair, N. Krishnamurthy, J.A.Abraham, “Evaluation of Integrated System-Level Checks for On-Line Error Detection,” Proceedings IEEE International Computer Performance and Dependability Symposium, pp. 292-301, 1996

[Lala 91] Lala, P.K., et al., “On seof-checking software design,” IEEE Proc. Of SOUTHEASTCON ’91, Williamsburg, VA, pp. 331-335, Apr. 7-10, 1991

[Lu 80] Lu, D. J., “Watchdog Processors and VLSI,” *Proceedings of the National Electronics conference*, Vol. 34, pp. 240-245, Chicago, Illinois, October 27-28, June 1980.

[Lu 82] Lu, D. J., “Watchdog Processor and Structural Integrity Checking,” *IEEE Transactions on Computers*, vol. C-31, No. 7, pp. 681-685, July 1982.

[Madeira 92] Madeira, H. and J. G. Silvia, “On-line Signature Learning and Checking,” *Dependable Computing for Critical Applications 2*, Springer-Verlag, J. F. and R. D. Schlichting (eds), pp. 395-420, 1992.

[Madeira 93] Madeira, H., M. Rela, and J. G. Silvia, “Time Behavior Monitoring as an Error Detection Mechanism,” *Dependable Computing for Critical Applications 2*, Springer-Verlag, J. F. and R. D. Schlichting (eds), pp. 395-420, Feb. 1993.

[Mahmood 85] Mahmood, Aamer and E. J. McCluskey, “Watchdog Processor: Error Coverage and Overhead,” *Digest, The Fifteenth Annual Int’l Symposium on Fault-Tolerant Computing (FTCS-15)*, pp. 214-219, Ann Arbor, Michigan, June 19-21, 1985.

[Matijevic 96] Matijevic, J. et al. "The mission and operation of the Mars Pathfinder microrover," *CONTROL ENGINEERING PRACTICE* Elsevier, vol.5, no.6, p. 827-35, June 1997

[Miremadi 92] Miremadi, G., J. Karlsson, J. U. Gunneflo, and J. Torin, “Two Software Techniques for On-line Error Detection,” *Digest of Papers, 22nd Annual Int’l Symp. on Fault Tolerant Computing*, pp. 328-35, July 1992

[Miremadi 92] Miremadi, G., J. Karlsson, J. U. Gunneflo, and J. Torin, “Two Software Techniques for On-line Error Detection,” *Digest of Papers, 22nd Annual Int’l Symo. on Fault_Tolerant Computing*, pp. 328-335, July 1992.

[Miremadi 95] Miremadi, G., J. TJ. Ohlsson, M. Rimen, and J. Karlsson, “Use of Time, Location and Instruction Signatures for Control Flow checking,” Proc. Of the DCCA-5 Int’l Conf., Springer-Verlag Series for Dependable Computing Systems, Sep. 1995.

[Miremadi 95] Miremadi, G., J. Tj. Ohlsson, M. Rimen, and J. Karlsson, “Use of Time, Location and Instruction Signatures for Control Flow Checking,” *Proc. of the DCCA-5 Int’l Conf.*, Springer-Verlag Series for Dependable Computing Systems, Sep. 1995

[Mitra 99] Mitra, S., N. Saxena and E. J. McCluskey, “A design diversity metric and reliability analysis for redundant systems,” *Intl. Test Conference*, pp. XX 1999

[Nair 96] Nair, V.S.S., J.A. Abraham and P. Banerjee, "Efficient techniques for the analysis of algorithm-based fault tolerance (ABFT) schemes" *IEEE TRANSACTIONS ON COMPUTERS*, vol.45, no.4, pp. 499-503, Apr. 1996

[Namjoo 82] Namjoo, M., “Techniques for Concurrent Testing of VLSI Processor Operation,” *Digest of Papers, IEEE Test Conf.*, pp. 461-468, Nov. 1982.

[Oh 98] Oh, N., Shirvani, P., and McCluskey, E. J.,”Signature Analysis by Instruction,” Center for Reliable Computing Technical Report, 1998.

[Ohlsson 95] J. Ohlsson and M Rimen, “Implicit Signature Checking,” *Digest of Papers, Twenty-Fifth International symposium on Fault-Tolerant Computing,* pp. 218-227, June 1995.

[Patel 82] Patel, J.H. et al. "Concurrent error detection in ALUs by recomputing with shifted operands," *IEEE Transactions on Computers*, vol.C-31, no.7, pp. 589-595, July 1982.

[Pradhan 96] Pradhan, D. K., “Fault-Tolerant Computer System Design,” *Prentice Hall*, 1996

[Rabejac 96] Christophe Rabejac, J-P Blanquart, and J-P Queille, "Executable Assertions and Timed Traces for On-Line Software Error Detection," Proceedings of the 26th International Symposium on Fault-Tolerant Computing, pp. 138-47, Jun, 1996.

[REE 99] REE Annual reports for FY99, Jet Propulsion Laboratory, NASA

[Reynolds 78] D. Reynolds and G. Metze, "Fault detection capabilities of alternating logic," IEEE Transactions on Computers, vol. C-27, pp.1093-1098, Dec. 1978

[Saxena 90] Saxena, N. R., and E. J. McCluskey, "Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums," *IEEE Trans. on Computers*, Vol. 39, No. 4, pp. 554-559, April 1990.

[Schuette 94] Schuette, M. A., J. P. Shen, "Exploiting Instruction-Level Parallelism for Integrated Control-Flow Monitoring," *IEEE Trans. on Computers*, Vol. 43, No. 2, pp.129-140, Feb. 1994

[Shedletsky 78] Shedletsky, J.J. et al., "Error correction by alternate-data retry," IEEE Transactions on Computers, vol. C-27, no.2, pp. 106-112, Feb.1978.

[Shen 83] Shen, J. P. and M. A. Schuette, "On-line Self-Monitoring Using Signatured Instruction Streams," *Int'l Test Conf. proceedings*, pp. 275-282, Oct. 1983

[Shirvani 99] Shirvani, P. P. and at al, "Fault-Tolerance Projects at Stanford CRC," *Military and Aerospace Applications of Programmable Devices and Technologies Conference* (MAPLD), Sep. 28-30, 1999, Laurel, MD.

[Shirvani 00] Shirvani, P. P. and at al, "Software-Implemented Hardware Fault Tolerance Experiments: COTS in Space," *Proc. International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, Fast Abstracts, pp. B56-7, New York, NY, June 25-28, 2000.

[Sohi 89] G. Sohi, et al., "A study of time-redundant fault tolerance techniques for high-performance pipelined computers," IEEE 19the FTCS, 436-443, Jun, 1989.

[Stallman 98] Stallman, R. "Using and Porting GNU CC," Free Software Foundation, 1998.

[Stone 96] Stone, H.W., "Mars Pathfinder Microrover: A Low-Cost, Low-Power Spacecraft," *Proceedings of the 1996 AIAA Forum on Advanced Developments in Space Robotics*, Madison, WI, Aug. 1996

[Takeda 80] Takeda, K. et al. "Logic design of fault-tolerant arithmetic units based on the data complementation strategy," *10th International Symposium on Fault-Tolerant Computing*, pp. 348-350, Oct. 1980.

[Titus 98] Titus, J. L., and et el, "First Observations of Enhanced Low Dose Rate Sensitivity in Space: One part of the MPTB Experiment," *IEEE Trans. on Nuclear Science,* vol. 45, no. 6, Dec. 1998

[Tung 90] C. H. Tung and C. W. McCarron, "Concurrent Control Flow Checking in Sequential and Parallel Program," *Twenty-Fourth Asilomar Conference on Signals, Systems and Computers*, Maple Press, Vol. 2, pp. 851-855, Nov. 1990.

[Wilken 89] Wilken, K., and J.P. Shen, "Concurrent Error Detection Using Signature Monitoring and Encryption: Low-Cost Concurrent-Detection of Processor Control Errors," *Dependable Computing for Critical Applications, Springer-Verlag, A. Avizienis, J.C. Laprie (eds)*, Vol. 4, pp. 365-384, 1989.

[Wilken 90] Wilken, K. and J. P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent-Detection of Processor Control Errors," *IEEE Trans. on Computer Aided Design*, Vol. 9, No. 6, pp. 629-641, June 1990.

[Wood 94] Wood, K.S., and et al., "The USA Experiment on the ARGOS Satellite: A Low Cost Instrument for Timing X-Ray Binaries," *EUV, X-Ray, and Gamma-Ray Instrumentation for Astronomy V,* ed. O.H. Siegmund & J.V. Vellerga, SPIE Proc., vol. 2280, pp.19-30, 1994

[Yau 80] Yau, S. S. and Fu-Chung Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 2, March 1980