

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

JOSÉ RODRIGO FURLANETTO DE AZAMBUJA

**Análise de Técnicas de Tolerância a Falhas
Baseadas em *Software* para a Proteção de
Microprocessadores**

Dissertação apresentada como requisito parcial
para a obtenção do grau de Mestre em Ciência
da Computação

Prof^a. Dr^a. Fernanda G. L. Kastensmidt
Orientadora

Porto Alegre, Novembro de 2010.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Azambuja, José Rodrigo Furlanetto de

Análise de Técnicas de Tolerância a Falhas Baseadas em *Software* para a Proteção de Microprocessadores / José Rodrigo Furlanetto de Azambuja – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

106 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2010. Orientadora: Fernanda Gusmão de Lima Kastensmidt.

1. Técnicas de tolerância a falhas baseadas em *software*. 2. Microprocessadores. 3. Injeção de falhas. I. Kastensmidt, Fernanda G. L.. II. Análise de Técnicas de Tolerância a Falhas Baseadas em *Software* para a Proteção de Microprocessadores.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion

Diretor do Instituto de Informática: Prof. Flávio Rech Wagner

Coordenador do PPGC: Prof. Álvaro Freitas Moreira

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Agradecimentos

Gostaria de agradecer aos meus Pais e ao Cado, por sempre estarem disponíveis quando precisei de qualquer tipo de ajuda e por me oferecerem uma base sólida em casa, para assim poder me dedicar unicamente aos estudos. Aos meus avós, tios e primos por todo o suporte e mais variadas comemorações. Também à Kare por seus conselhos motivacionais e por aturar meus dias mais mal-humorados.

Gostaria também de agradecer ao Sorriso - que de alegre não tem nada - pela co-fundação do cafezão da tarde na Faufrgs, em repúdio ao Ildo, e pelas longas discussões que resultaram em Xboxes, televisões, xampus, uma Scooter e muito chá. A todos aqueles que mais tarde se juntaram ao movimento e tornaram as tardes na UFRGS muito mais divertidas: Rodrigo, Raphinha, Gabibo, Samuka e Paola.

A todos os amigos que me tiraram do laboratório nos fins de semana - e também aos que nele me mantiveram durante a semana - e minhas sinceras desculpas por não poder citá-los todos. Ao pessoal da Brigada dominical pela morte ao bom futebol, ao pessoal da FDC pelas praias de inverno e carnavais furados e, por fim, ao pessoal da banda por não se deixarem levar por solos de baixo em momentos totalmente inoportunos - e errados.

Ao pessoal do laboratório: os mais velhos Jefferson e Luciéli; a contemporânea Carol; e os mais novos Anelise, Lucas, Marisa e Eduardo. Aos bolsistas Conrado, Fernando, Lucas, Condessa e João pelas inúmeras horas injetando falhas, artigos de última hora e, principalmente, por se juntarem e tentarem - sem sucesso - oferecer um desafio à minha altura no Quake, durante as horas vagas.

Especialmente, gostaria de agradecer à professora Fernanda por me aturar durante um TCC, um Mestrado e agora um Doutorado! Agradeço também aos professores Luigi, Erika, Lisboa e Luba por toda a ajuda, principalmente, durante a decisão do tema a ser abordado no Mestrado.

À toda a coordenação do PPGC, ao Instituto de Informática, à UFRGS em geral e ao apoio financeiro oferecido pelo CNPq.

Sem deixar o lado espiritual de lado, gostaria de agradecer também às forças sobrenaturais que fazem com que a gente continue estudando e indo até o final.

A todos, os meus mais sinceros Agradecimentos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS.....	7
LISTA DE FIGURAS.....	9
LISTA DE TABELAS.....	11
RESUMO.....	12
ABSTRACT.....	13
1 INTRODUÇÃO.....	14
2 TÉCNICAS DE TOLERÂNCIA A FALHAS EM SOFTWARE – ESTADO DA ARTE.....	18
2.1 Técnicas de Proteção aos Dados.....	18
2.1.1 Error Detection by Duplicated Instructions (EDDI).....	18
2.2 Técnicas de Proteção ao Controle.....	20
2.2.1 Control Flow Checking using Assertions (CCA).....	20
2.2.2 Enhanced Control Flow Checking using Assertions (ECCA).....	23
2.2.3 Control-Flow Checking by <i>Software</i> Signatures (CFCSS).....	24
2.3 Técnicas de Proteção aos Dados e ao Controle.....	26
2.3.1 Conjunto de Regras de Transformação Proposto por Rebaudengo.....	26
2.3.2 Conjunto de Regras de Transformação Proposto por Nicolescu.....	29
2.3.3 <i>Software</i> Implemented Fault Tolerance (SWIFT).....	31
2.4 Técnicas de Proteção à Memória de Dados.....	32
3 METODOLOGIA PROPOSTA.....	35
3.1 Microprocessador miniMIPS.....	35
3.1.1 <i>Datapath</i>	37
3.1.2 <i>Controlpath</i>	38
3.2 Conjunto de Técnicas Baseadas em <i>Software</i> Implementadas em Linguagem Assembly	38
3.2.1 Técnica de Tolerância em Assembly: SIG - Assinaturas.....	39
3.2.2 Técnica de Tolerância em Assembly: BBD - Divisão de blocos básicos.....	40
3.2.3 Técnica de Tolerância em Assembly: VAR1 - Variáveis 1.....	40

3.2.4	Técnica de Tolerância em Assembly: VAR2 - Variáveis 2	42
3.2.5	Técnica de Tolerância em Assembly: VAR3 - Variáveis 3	43
3.2.6	Técnica de Tolerância em Assembly: BRA - Desvios condicionais.....	44
3.3	HPCT: Ferramenta de Transformação de Código em Linguagem Assembly usada Pós-Compilação.....	45
3.3.1	Visão geral.....	47
3.3.1.1	HPCSuite	49
3.3.1.2	Classe: PosCompiler	49
3.3.1.3	Classe: BranchCorrection	49
3.3.1.4	Classe: Relations	49
3.3.1.5	Classe: BBlock	49
3.3.1.6	Classe: Line.....	49
3.3.1.7	Classe: Registers	50
3.3.1.8	Classes: SIG, VAR1, VAR2, VAR3, BRA	50
3.3.1.9	Classe: Util.....	50
3.3.1.10	Classe: FileUtil.....	50
3.3.2	Funções Implementadas	50
3.3.2.1	Processamento dos Blocos Básicos	50
3.3.2.2	Replicação da Chamada de Sub-rotina	51
3.3.2.3	Escolha de Registradores para Duplicação.....	51
3.3.2.4	Método para Correção de Desvio: BranchCorrection	52
3.3.2.5	Lista de Desvios Condicionais Invertidos	52
3.4	Ferramenta de Simulação de Falhas.....	53
3.4.1	Geração de Conjunto de Falhas.....	54
3.4.2	Simulação de Conjunto de Falhas	55
3.4.3	Coleta de Resultados	56
4	RESULTADOS DE SUSCEPTIBILIDADE A SINGLE EVENT UPSETS (SEU AND SET).....	58
4.1	Aplicações para Estudos de Caso.....	58
4.1.1	Multiplicação de Matrizes	59
4.1.2	Ordenação <i>Bubble Sort</i>	60
4.2	Construção do Conjunto de Falhas.....	61
4.2.1	Injeção Preliminar de Falhas.....	62
4.2.2	Classificação de Falhas.....	63
4.2.3	Grupos de Falhas.....	63
4.2.3.1	Multiplicação de Matrizes.....	64
4.2.3.2	Ordenação <i>Bubble sort</i>	65
4.3	Técnicas Implementadas	65
4.4	Mapeamento de Falhas.....	68
4.5	Injeção de Falhas.....	70
5	ANÁLISE E CLASSIFICAÇÃO DOS RESULTADOS.....	72
5.1	Análise por Efeito de Falhas.....	79
5.2	Análise por Local de Injeção de Falhas	80
5.3	Análise por Tipo de Falha.....	82

5.4	Análise de Latência para a Detecção de Falhas	83
6	CONCLUSÃO.....	85
	REFERÊNCIAS.....	87
	ANEXO A - ARQUIVO DE CONFIGURAÇÃO DO INJETOR DE FALHAS	90
	ANEXO B - ARTIGOS PUBLICADOS	92

LISTA DE ABREVIATURAS E SIGLAS

ASIC	Application Specific Integrated Circuits
BB	Bloco Básico
BEQ	Branch if EQual
BFI	Branch Free Intervals
BID	Block Identifier
BNE	Branch if Not Equal
CCA	Control Flow Checking using Assertions
CFCSS	Control-Flow Checking by <i>Software</i> Signatures
CFID	Control Flow Identifier
CMOS	Complementary Metal-Oxide Semiconductor
COTS	Commercial-of-the-Shelf Processors
CRC	Cyclic Redundancy Checks
DATE	Design, Test and Automation in Europe
DRAM	Dynamic Random Access Memory
DWC	Duplication With Comparison
ECCA	Enhanced Control Flow Checking using Assertions
EDAC	Error Detection and Correction
EDDI	Error Detection by Duplicated Instructions
ELF	Executable and Linking Format
ENSERG	Ecole Nationale Supérieure d'Electronique et de Radioélectrique de Grenoble
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Electrically Programmable Read Only Memory
ESA	European Space Agency
FPGA	Field Programmable Gate Array
GPR	General Signature Register
HDL	<i>Hardware</i> Description Language

HPCT	Hardening Post Compiling Tool
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
ITAR	International Traffic in Arms Regulation
JAL	Jump and Link
JETTA	Journal in Electronic Testing: Theory and Applications
JR	Jump to Register
JRE	Java Runtime Environment
LUT	Look-Up Tables
MAC	Multiply and Accumulate
NRE	Non-Recurring Engineering
PC	Program Counter
PCB	Printed Circuit Board
RAM	Random Access Memory
RISC	Reduced Instruction Set Architecture
ROM	Read Only Memory
RTL	Register Transfer Level
RTS	Real Time Signature
SEE	Single Event Effect
SET	Single Event Transient
SEU	Single Event Upset
SP	Stack Pointer
SPARC	Scalable Processor ARChitecture
SRAM	Static Random Access Memory
SWIFT	<i>Software</i> Implemented Fault Tolerance
TCL	Tool Command Language
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
TNS	Transactions on Nuclear Science
UFRGS	Universidade Federal do Rio Grande do Sul
ULA	Unidade Lógica e Aritmética
UML	Unified Modeling Language
VTS	VLSI Test Symposium
XOR	Exclusive OR

LISTA DE FIGURAS

<i>Figura 2.1: Aplicação da técnica EDDI (REIS, 2005).</i>	20
<i>Figura 2.2: Grafo de fluxo de controle (ALKHALIFA, 1999).</i>	21
<i>Figura 2.3: Implementação da técnica CCA (ALKHALIFA, 1999).</i>	22
<i>Figura 2.4: Vulnerabilidade no fluxo de controle (ALKHALIFA, 1999).</i>	23
<i>Figura 2.5: Aplicação da técnica ECCA (OH, 2005).</i>	24
<i>Figura 2.6: Aplicação da técnica CFCSS (OH, 2005).</i>	25
<i>Figura 2.7: Aplicação das regras sobre dados (REBAUDENGO ET AL., 1999).</i>	26
<i>Figura 2.8: Aplicação das regras sobre sub-rotinas (REBAUDENGO ET AL., 1999).</i>	27
<i>Figura 2.9: Aplicação das regras #4 e #5 (REBAUDENGO ET AL., 1999).</i>	28
<i>Figura 2.10: Aplicação da regra #6 (REBAUDENGO ET AL., 1999).</i>	28
<i>Figura 2.11: Aplicação para chamada e retorno de sub-rotinas (REBAUDENGO, 1999).</i>	29
<i>Figura 2.12: Aplicação do primeiro grupo de regras (NICOLESCU, 2003).</i>	30
<i>Figura 2.13: Aplicação do segundo grupo de regras (NICOLESCU, 2003).</i>	31
<i>Figura 2.14: Arquitetura de EDAC sobre memória (CHEN et al., 2010).</i>	33
<i>Figura 3.1: Arquitetura miniMIPS pipeline.</i>	35
<i>Figura 3.2: Classes de instruções do miniMIPS.</i>	36
<i>Figura 3.3: Fluxo de compilação para o miniMIPS.</i>	37
<i>Figura 3.4: Conjunto de registradores.</i>	38
<i>Figura 3.5: Transformação Assinaturas 1.</i>	40
<i>Figura 3.6: Transformação de registradores Variáveis 1.</i>	41
<i>Figura 3.7: Transformação Variáveis 2.</i>	43
<i>Figura 3.8: Transformação Variáveis 3.</i>	44
<i>Figura 3.9: Transformação Desvios condicionais.</i>	45
<i>Figura 3.10: Papel do HPCT.</i>	46
<i>Figura 3.11: Fluxo de compilação HPCT.</i>	47
<i>Figura 3.12: Diagrama de classes HPCT.</i>	48
<i>Figura 3.13: Transformação BranchCorrection.</i>	52
<i>Figura 3.14: Papel do Injetor de Falhas.</i>	54
<i>Figura 3.15: Script com o caminho completo da lista de falhas.</i>	55
<i>Figura 3.16: Script com o tempo de injeção da lista de falhas.</i>	55
<i>Figura 3.17: Exemplo de Injeção de falha.</i>	55
<i>Figura 3.18: Script de simulação de falhas.</i>	56
<i>Figura 3.19: Script de coleta de resultados.</i>	57
<i>Figura 4.1: Código compilado no formato coe.</i>	59
<i>Figura 4.3: Código C da aplicação ordenação bubble sort.</i>	61
<i>Figura 4.4: Relação de desempenho da multiplicação de matrizes.</i>	66
<i>Figura 4.5: Relação de desempenho da ordenação bubble sort.</i>	68
<i>Figura 4.6: Lista de transições do PC.</i>	69
<i>Figura 5.1: Relação de desempenho da multiplicação de matrizes.</i>	73
<i>Figura 5.2: Relação de desempenho da multiplicação de matrizes.</i>	73
<i>Figura 5.3: Visão geral das técnicas (I) SIG, (II) BBD, (III) VARI, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de multiplicação de matrizes sobre falhas do tipo SEU.</i>	77
<i>Figura 5.4: Visão geral das técnicas (I) SIG, (II) BBD, (III) VARI, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de multiplicação de matrizes sobre falhas do tipo SET.</i>	77
<i>Figura 5.5: Visão geral das técnicas (I) SIG, (II) BBD, (III) VARI, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de ordenação bubble sort sobre falhas do tipo SEU.</i>	78

<i>Figura 5.6: Visão geral das técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de ordenação bubble sort sobre falhas do tipo SET.....</i>	<i>78</i>
<i>Figura 5.7: Taxa de detecção de falhas de acordo com seu efeito para a aplicação de multiplicação de matrizes.....</i>	<i>79</i>
<i>Figura 5.8: Taxa de detecção de falhas de acordo com seu efeito para a aplicação de ordenação bubble sort.....</i>	<i>79</i>
<i>Figura 5.9: Taxa de detecção de falhas de acordo com o local de injeção para a aplicação de multiplicação de matrizes.</i>	<i>81</i>
<i>Figura 5.10: Taxa de detecção de falhas de acordo com o local de injeção para a aplicação de ordenação bubble sort.</i>	<i>81</i>
<i>Figura 5.11: Taxa de detecção de falhas de acordo com o seu tipo para a aplicação de multiplicação de matrizes.....</i>	<i>82</i>
<i>Figura 5.12: Taxa de detecção de falhas de acordo com seu tipo para a aplicação de ordenação bubble sort.....</i>	<i>83</i>

LISTA DE TABELAS

<i>Tabela 4.1: Grupo de falhas para aplicação de multiplicação de matrizes.....</i>	<i>64</i>
<i>Tabela 4.2: Grupo de falhas para aplicação de ordenação bubble sort.....</i>	<i>65</i>
<i>Tabela 4.3: Características físicas da multiplicação de matrizes</i>	<i>66</i>
<i>Tabela 4.4: Características físicas da ordenação bubble sort.....</i>	<i>67</i>
<i>Tabela 4.5: Porcentagem de falhas perdidas durante o mapeamento para a multiplicação de matrizes. .</i>	<i>70</i>
<i>Tabela 4.6: Porcentagem de falhas perdidas durante o mapeamento para a ordenação bubble sort.....</i>	<i>70</i>
<i>Tabela 5.1: Taxa de detecção de falhas para multiplicação de matrizes.....</i>	<i>72</i>
<i>Tabela 5.2: Taxa de detecção de falhas para ordenação bubble sort.</i>	<i>72</i>
<i>Tabela 5.3: Resultado da injeção de falhas sobre a aplicação multiplicação de matrizes paras as técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA.....</i>	<i>75</i>
<i>Tabela 5.4: Resultado da injeção de falhas sobre a aplicação de ordenação bubble sort paras as técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA.....</i>	<i>76</i>
<i>Tabela 5.5: Tempo médio de detecção de falha para multiplicação de matrizes.</i>	<i>84</i>
<i>Tabela 5.6: Tempo médio de detecção de falha para ordenação bubble sort.....</i>	<i>84</i>

RESUMO

Da mesma maneira que novas tecnologias trouxeram avanços para a indústria de semicondutores, diminuíram a confiabilidade dos transistores e conseqüentemente dos sistemas digitais. Efeitos causados por partículas energizadas antes só vistos em ambientes espaciais hoje se manifestam a nível do mar, introduzindo novos desafios para a fabricação e projeto de sistemas que requerem confiabilidade.

Sistemas de alta confiabilidade que utilizam circuitos integrados exigem a utilização de técnicas de tolerância a falhas capazes de detectar ou mesmo corrigir os erros causados por partículas energizadas. Esta proteção pode ser implementada em diferentes níveis: *hardware* ou *software*. Enquanto o primeiro exige a modificação interna de circuitos integrados desprotegidos e oferece alto desempenho, o segundo altera somente o código de programa, porém com perdas de desempenho que variam conforme o grau de proteção do sistema.

O objetivo deste trabalho é analisar a eficiência na detecção de falhas em microprocessadores através de técnicas de tolerância a falhas baseadas somente em *software*. Para isto, são propostas diferentes técnicas de tolerância a falhas baseadas somente em *software* inspiradas em técnicas apresentadas no estado da arte. Estas são implementadas separadamente e combinadas, de maneira a encontrar suas vulnerabilidades e descobrir como estas podem ser combinadas, a fim de apresentar uma solução ideal para diferentes sistemas em termos de desempenho e confiabilidade.

A análise se dá através de uma campanha de injeção de falhas direcionada para cada parte de um microprocessador e observando-se os efeitos causados por cada falha no resultado do sistema.

Palavras-Chave: Técnicas de tolerância a falhas baseadas em *software*, microprocessadores, injeção de falhas.

An Analysis on Software-based Fault Tolerant Techniques to Protect Microprocessors

ABSTRACT

As new technologies brought advances to the semiconductor industry, they also lowered transistors' reliability and therefore decreased digital systems' reliability. Effects caused by energized particles which were only seen in spatial environments nowadays manifest at sea level, introducing new challenges in the design and fabrication of systems that require high reliability.

High reliable systems based on integrated circuits require fault tolerant techniques in order to detect or even correct errors caused by energized particles. This protection can be implemented in different levels: hardware or software. While the first requires internal modifications in the integrated circuit and offers high performance, the second modifies only the program code, but causes system's performance degradation, which can vary according the system's protection level.

This work's objective is to analyze software-based fault tolerant techniques efficiency to detect faults in microprocessors. In order to achieve it, different fault tolerance techniques based in software are proposed inspired in techniques presented in state-of-the-art techniques. They are implemented separately and then combined, to analyze their vulnerabilities and realize how to combine them, in order to present an ideal solution for each system, taking into account performance and reliability.

The analysis is based in a fault injection campaign directed to each part of the microprocessor, considering the effects caused by each fault in the system's response.

Keywords: Software-based fault tolerant techniques, microprocessors, fault injection.

1 INTRODUÇÃO

Os recentes avanços tecnológicos na indústria de semicondutores têm permitido a fabricação de circuitos integrados de maior densidade com um número cada vez mais funcionalidades. Entretanto, a evolução da tecnologia de fabricação tem trazido desafios na área de confiabilidade. A tecnologia CMOS tem evoluído de acordo com a lei de Moore (MOORE, 1965). Desta maneira, o transistor fabricado em tecnologia nanométrica como 45nm, 32nm e menor se aproxima dos limites físicos impostos pela disponibilidade de poucos átomos para formar o canal do transistor (KIM et al., 2003), (HOMPSON et al., 2005), tornando assim um desafio sua fabricação e uso em ambientes sujeitos a ruído. A redução de confiabilidade dos dispositivos CMOS nas novas tecnologias é uma consequência de diversos problemas provenientes das características físicas dos dispositivos, dentre eles:

- Menores tensões de operação;
- Menor dimensão dos transistores;
- Dispositivos mais velozes;
- Proximidade dos transistores no silício.

O ruído pode ser proveniente de diversas partículas energizadas ou não presentes no meio de operação. Na terra, a presença de nêutrons pode perturbar os circuitos gerando partículas secundárias como partículas alfa que perturbam os transistores gerando pulsos de efeito transitórios (ZIEGLER, 1996). No espaço, há partículas como prótons, íons pesadores e elétrons que podem ionizar diretamente ou indiretamente os transistores e também gerar pulsos transitórios. Esses pulsos transitórios, podem ser considerados como ruídos, e dependendo de sua amplitude em corrente e tensão e duração, podem ser interpretados como sinal interno do circuito, gerando erros.

Os efeitos transientes mais comum observados em circuitos integrados operando no espaço são os *Single Event Effects* (SEE). Quando o pulso transiente ocorre um elemento de memória, então este efeito é denominado *Single Event Upset* (SEU). Esse efeito é visto como uma inversão do valor armazenado no flip-flop, ou seja, um *bit-flip*. Quando esse efeito ocorre em uma porta lógica de um bloco combinacional, ele é chamado de *Single Event Transient* (SET). Além disso, existe a falha de acúmulo de radiação conhecida como *Total Ionizing Dose* (TID) medida em krad (Si). Normalmente, circuitos integrados operando no espaço devem tolerar pelo menos até 100 krad (Si), dependendo do local e duração da missão.

A largura do canal dos transistores vem diminuindo a cada nova tecnologia, diminuindo assim o tamanho dos transistores. Esta redução permite o projeto de circuitos integrados mais densos, com uma maior quantidade de transistores por área e, conseqüentemente, maior quantidade de lógica implementada por milímetro quadrado.

Apesar de reduzir o atraso dos transistores e permitir uma maior ocupação do *wafers* de silício, a maior densidade dos transistores faz com que uma partícula energizada tenha maior probabilidade de afetar um componente, ao perturbar um maior número de transistores. Além disso, cada nodo do circuito de alta densidade armazena um valor menor de carga, logo é mais fácil para uma partícula energizada carregar ou descarregar um nodo do circuito, provocando um SEE. Por consequência, de maneira geral, as novas tecnologias estão mais sensíveis a *bit-flips* (SEU) e a SET em sua lógica.

As novas tecnologias, da mesma maneira que possibilitaram a redução na largura do *gate*, têm diminuído o atraso dos transistores e oferecem componentes lógicos mais rápidos, possibilitando assim a utilização de maiores frequências de relógio. Por outro lado, a duração dos pulsos transientes introduzidos por partículas energizadas se manteve constante, independentemente da frequência de relógio (DOOD, 2004), (FERLET-CAVROIS, 2006). Com isto, a probabilidade de um pulso transiente (SET) ser capturado pela borda do relógio ou até mesmo de durar mais do que um período de relógio tem aumentado a cada nova tecnologia, aumentando assim a probabilidade de uma falha se tornar um erro no sistema.

A fim de utilizar circuitos integrados em sistemas de alta confiabilidade, faz-se necessário o uso de técnicas de tolerância capazes de detectar ou até mesmo de corrigir esses tipos de erros (SEEs). Conforme o tipo de circuito integrado e a aplicação alvo, diferentes técnicas de tolerância podem ser utilizados em diversos níveis do projeto. Técnicas podem ser aplicadas desde o nível de leiaute, passando pelo nível de transistor, nível lógico, algorítmico em descrição de *hardware* e até mesmo em *software* no caso de processadores. Neste trabalho, focaremos focar no uso de processadores em aplicações de alta confiabilidade. Atualmente, os microprocessadores estão presentes em grande parte das aplicações, desde sistemas embarcados simples e de uso geral, como telefones celulares e reprodutores de áudio, até aplicações complexas, de alta confiabilidade, como satélites e sistemas embarcados na indústria aviônica. Alguns processadores podem ser encontrados na versão *radiation hardening* (RadHard), que são circuitos integrados projetados para tolerar SEE e TID em aplicações sob radiação. Como exemplo de um processador RadHard, citamos o processador Leon 2 FT, baseado na arquitetura *Scalable Processor ARChitecture* (SPARC) V8, IEEE-1754 (IEEE, 1994) que é utilizado pela *European Space Agency* (ESA).

O problema dos processadores RadHard é que normalmente são muito caros devido ao *Non-Recurring Engineering* (NRE) versus o número de peças vendidas. Além disso, esses componentes podem estar sob embargos de importação ou exportação como o *International Traffic in Arms Regulation* (ITAR) e às vezes são antigos e de baixa capacidade. Logo, há duas alternativas para o uso de processadores em aplicações de alta confiabilidade:

- Técnicas baseadas em *hardware*, onde o sistema físico é alterado para a inserção de módulos adicionais ativos (triplicação, por exemplo) ou passivos (verificadores) e o processador é desenvolvido e fabricado com essas técnicas tendo assim um novo processador RadHard;
- Técnicas baseadas em *software*, onde a parte física do sistema permanece intacta e somente o código de programa é modificado.

As técnicas baseadas em *hardware* normalmente se resumem à replicação e à comparação de elementos de memória e outros blocos sensíveis do processador. Durante a etapa inicial de projeto de um microprocessador, quando a arquitetura interna

pode ser alterada, técnicas de tolerância a falhas como redundância de informação (PFLANZ, 2001) e redundância de componentes (PRADHAN, 2006) podem ser exploradas. Apesar de serem muito efetivas na detecção e, até mesmo, correção de falhas, estas técnicas podem apresentar grandes aumentos em termos de área, potência e, conseqüentemente, custos. Por outro lado, a frequência de operação do sistema tende a permanecer a mesma, visto que a operação dos componentes duplicados pode ser realizada em paralelo com a operação normal do sistema, introduzindo assim apenas o atraso do comparador. Por outro lado, a alteração da arquitetura de um microprocessador é altamente custosa em termos de projeto e, principalmente, custos de fabricação (fabricação das máscaras). Por esta razão, estas técnicas são utilizadas em grande parte em *soft-IPs* e em aplicações onde os custos não apresentam um problema (aeroespaciais, por exemplo).

Um segundo método baseado em *hardware* pode ser utilizado quando a arquitetura é fixa e consiste em introduzir um módulo chamado co-processador *watchdog*, capaz de monitorar as atividades do processador principal, de forma não-intrusiva, procurando anormalidades. Conectado aos barramentos de memória, o *watchdog* informa ao processador principal ou mesmo a um sistema externo a existência de uma falha. A implementação deste processador pode ser encontrada em (MAHMOOD, 1988). A utilização de um co-processador acarreta na implementação do mesmo, uma vez que suas funções variam de acordo com o processador, tendo este de operar na mesma frequência do processador original, mantendo assim o desempenho do mesmo. Além disso, a introdução de um co-processador exige o retrabalho da placa do sistema, ou *Printed Circuit Board* (PCB) do circuito, testes e verificações do mesmo.

As técnicas baseadas em *software* cresceram significativamente nos últimos anos pelo fato de não necessitarem mudanças na parte física do sistema. Com isto, há economia de uma grande quantidade de recursos ao se utilizar processadores comerciais, também conhecidos por COTS (*commercial-of-the-shelf processors*), uma vez que são mais baratos do que um processador específico, considerando custos em pesquisa e desenvolvimento. Estas técnicas, como (CHEYNET, 2000), (OH, 2002) e (OH e MITRA, 2002), entretanto, aumentam consideravelmente o tempo de computação da aplicação e ocupações da memória de programa e de dados.

Este trabalho, trata de técnicas de tolerância a falhas em nível de *software* para que processadores comerciais possam ser usados em aplicações espaciais. Atualmente, as técnicas existentes na literatura não são capazes de obter total detecção de falhas (REBAUDENGO, 2006), (BOLCHINI, 2005). Isto acontece, principalmente, devido a falhas com efeito sobre o controle dos microprocessadores, que fazem parte de um grupo pequeno, mas que devem ser levados em consideração para aplicações que requerem alta confiabilidade. (REBAUDENGO, 2006) foi capaz de obter 100% de detecção das falhas, mas para isso modificou a arquitetura do microprocessador, utilizando uma técnica híbrida entre *hardware* e *software* de tolerância a falhas.

Este trabalho tem como principal objetivo implementar diferentes técnicas de tolerância a falhas baseadas em *software*, com o intuito de compará-las e analisá-las detalhadamente. Através desta análise espera-se encontrar as principais vulnerabilidades de cada uma delas, formando assim uma base sólida para o desenvolvimento de novas técnicas capazes de chegar a uma taxa de detecção de 100% e até mesmo para a combinação de diferentes técnicas, buscando a melhor adequação da aplicação aos requisitos do sistema. A análise levará em consideração os seguintes aspectos:

- Tamanho do código de programa;
- Tamanho da memória de dados;
- Tempo de execução das aplicações;
- Taxa de detecção de falhas.

A análise das técnicas de tolerância a falhas exige a criação de uma metodologia para a transformação do código de programa num código protegido, uma vez que este é um trabalho complexo e sua realização manual é impraticável, e uma metodologia para a criação de um conjunto de falhas, injeção deste numa determinada aplicação, coleta de resultados e análise de resultados. Infelizmente, não existem ferramentas abertas que exerçam estas funções, sendo assim necessária a implementação delas.

Para estudo de caso, será utilizado o microprocessador miniMIPS devido a sua ampla utilização na literatura e pela disponibilidade de seu código-fonte sintetizável para as placas de prototipação e desenvolvimento disponíveis nos laboratórios de pesquisa da UFRGS. Como aplicações, serão utilizados algoritmos de multiplicação de matrizes, devido a alta quantidade de operações aritméticas, e de ordenação de dados (*bubble sort*), devido à alta quantidade de instruções de controle envolvidas.

Num primeiro momento, serão implementadas ferramentas de geração e injeção de falhas e de coleta e análise de resultados, em conjunto com o *software* de simulação ModelSim. Após esta etapa, será implementada uma ferramenta de transformação automática de códigos de programa em códigos protegidos de acordo com as técnicas de proteção escolhidas.

Posteriormente, serão geradas versões de códigos protegidos para as aplicações de multiplicação de matrizes e ordenação *bubble sort* através de diferentes técnicas de tolerância a falhas e construído um conjunto de falhas para a simulação das técnicas. A partir dos resultados obtidos, após a campanha de injeção de falhas, serão realizadas análises e comparações entre as técnicas utilizadas, para o estabelecimento de conclusões e indicações de trabalhos futuros.

Este trabalho é organizado da seguinte maneira:

- Capítulo 2 – Apresenta o estado da arte das técnicas de tolerância a falhas baseadas somente em *software*.
- Capítulo 3 – Propõe regras de transformação de código divididas em grupos e apresenta as ferramentas de transformação automática de código e de injeção de falhas.
- Capítulo 4 – Descreve a campanha de injeção de falhas, desde a criação do conjunto de falhas até a coleta dos resultados gerados pela ferramenta de injeção de falhas.
- Capítulo 5 – Analisa os resultados obtidos durante a campanha de injeção de falhas.
- Capítulo 6 – Conclui este trabalho e apresenta os trabalhos futuros.

2 TÉCNICAS DE TOLERÂNCIA A FALHAS EM SOFTWARE – ESTADO DA ARTE

Neste capítulo são apresentadas técnicas de tolerância a falhas implementadas somente em *software*, no estado da arte, para a proteção de microprocessadores. As técnicas de proteção se dividem em quatro grandes grupos: (1) técnicas de proteção aos dados da aplicação, (2) técnicas de proteção ao fluxo de controle do programa, (3) técnicas híbridas, ou seja, de proteção aos dados e ao fluxo de controle e (4) técnicas de proteção à memória de física de dados. O estado da arte de cada grupo será detalhado a seguir.

2.1 Técnicas de Proteção aos Dados

As técnicas de proteção aos dados visam somente preservar a integridade dos valores armazenados nos registradores e na memória de dados do microprocessador. Cabe a elas realizar checagens regulares dos valores armazenados, sempre indicando um erro na presença de valores incorretos. A proteção de dados, entretanto, não compreende a proteção da correta execução do código e, portanto, uma falha desta natureza, que modifique os dados, poderá não ser detectada.

2.1.1 Error Detection by Duplicated Instructions (EDDI)

A técnica chamada EDDI (OH2 ET AL., 2002) é um mecanismo de detecção de falhas implementado puramente em *software*, que explora a redundância de instruções e de variáveis para obter tolerância a falhas. As instruções do código-fonte original são duplicadas pelo compilador e intercaladas com as instruções originais. A cópia do programa, entretanto, utiliza registradores e posições de memória diferentes dos utilizados pelo código de programa original, de maneira que a computação dos dados replicados não interfira na computação dos dados originais.

Em certos pontos do código, com o intuito de sincronizar a cópia com o código original, instruções de verificação são inseridas pelo compilador, certificando assim que os valores utilizados pelo código original são iguais aos valores utilizados pela cópia. Os pontos de sincronização, ou seja, de verificação de consistência dos dados e sua réplica influenciam diretamente no desempenho do sistema e, por isto, devem ser inseridos no programa somente quando necessário.

Assumindo que as entradas e saídas do programa são armazenadas em memória, e que a correta execução de um programa é definida pelas saídas do mesmo, conclui-se que um programa foi executado corretamente se ele executou corretamente todas as instruções de *store* na memória. Conseqüentemente, é natural utilizar instruções de *store* como pontos de sincronização, mas sabendo que estes, somente em instruções de *store*

são insuficientes, visto que desvios condicionais errôneos podem alterar o fluxo de execução do sistema, evitando a execução de instruções de *store*, executar incorretamente instruções de *store* ou ainda fazer com que um *store* grave valores errados na memória. Portanto, instruções de desvios também devem ser utilizadas como pontos de sincronização, onde valores redundantes devem ser comparados.

A figura 2.1 mostra um exemplo de código desprotegido comparado com o mesmo código transformado segundo as regras da técnica de EDDI. Neste exemplo, a instrução de *load* de uma variável global, presente na linha 1, é duplicada através da linha 2. É importante salientar que a instrução replicada carrega um valor presente numa posição de memória diferente, deslocada pelo valor de *offset*, evitando assim o conflito entre os valores do código original e da cópia. Da mesma maneira, a instrução de *add*, presente na linha 3, é duplicada através da instrução 4 para criar uma cadeia computacional redundante. Além da instrução 4, são inseridas as instruções 5 e 6 devido a instrução de *add*, com a função de verificar a consistência dos valores utilizados na soma.

A instrução de *store*, encontrada na linha 8, é um ponto de sincronização e, portanto, é adicionado uma instrução de verificação através da instrução *br*, localizada na linha 7. A instrução *br* tem a função de verificar o valor de *pl*. Na presença de uma discrepância de informações, a instrução *br* executará um desvio para a sub-rotina chamada *faultDetected*, identificando assim um erro. Por fim, a instrução de *store* é replicada, porém seu valor é salvo numa posição de memória diferente da original, mantendo assim a independência entre o código original e sua cópia.

Como mostrado no exemplo, as instruções replicadas são inseridas logo após as originais. Entretanto, um compilador otimizado é livre para utilizar a técnica de Instruction Level Parallelism (ILP) disponíveis para paralelizar a execução e assim minimizar a perda de desempenho causada pela transformação do código original. Dependendo da maneira como as instruções duplicadas são executadas duas formas de redundâncias serão exploradas: temporal ou espacial. A redundância temporal acontece quando instruções são executadas sequencialmente pelo mesmo *hardware*, enquanto que a espacial acontece quando as instruções são executadas em paralelo por diferentes módulos de *hardware*.

Apesar de oferecer uma boa proteção contra erros, em torno de 95% (OH2 ET AL., 2002) de detecção para aplicações como *Fast Furrier Transform* (FFT) e multiplicação de matrizes, o método EDDI aumenta consideravelmente a utilização de memória. Como mostrado na figura 2.1, cada posição de memória exige uma posição sombra de memória para armazenar o valor duplicado. A duplicação de memória, entretanto, causa custos significativos de *hardware*, como maior consumo de potência e área devido a necessidade de utilização de mais posições de memória e perda de desempenho devido a maior quantidade de tráfego de informações com a memória *cache*. Da mesma maneira, é necessário duplicar todos os registradores. Não sendo possível alterar o *hardware* ou realizar a troca do microprocessador por um modelo com mais registradores, existe a possibilidade de utilizar a técnica de *register renaming*, onde o compilador, sob a pena de perder desempenho, renomeia registradores possibilitando seu uso pelo EDDI.

<pre>ld r12=[GLOBAL] add r11=r12,r13 st m[r11]=r12</pre>	<pre>ld r12=[GLOBAL] 1: ld r22=[GLOBAL+offset] add r11=r12,r13 2: add r21=r22,r23 3: cmp.neq.unc p1,p0=r11,r21 4: cmp.neq.or p1,p0=r12,r22 5: (p1) br faultDetected st m[r11]=r12 6: st m[r21+offset]=r22</pre>
(a) Original Code	(b) EDDI Code

Figura 2.1: Aplicação da técnica EDDI (REIS, 2005).

Além das penalidades causadas pela replicação das variáveis, a técnica EDDI apresenta uma proteção contra falhas incompleta, uma vez que não apresenta um modo de detectar erros de controle, ou seja, que causam um desvio de fluxo de execução do programa. Apesar de verificar os operandos dos desvios condicionais, existe a possibilidade de que o fluxo de controle seja modificado incorretamente, devido a erros que podem acontecer durante a execução de uma instrução de desvio condicional, como a corrupção de um registrador após a sua verificação com sua cópia, ou até mesmo em instruções de chamada de sub-rotina.

A técnica EDDI apresenta, portanto, somente a replicação e verificação de instruções que operam sobre dados, não apresentando qualquer mecanismo para a verificação do fluxo de controle do programa.

2.2 Técnicas de Proteção ao Controle

As técnicas de proteção ao controle visam somente a integridade do fluxo de execução da aplicação. Para isto, introduzem variáveis globais e identificadores de blocos básicos, uma vez que é impossível replicar o fluxo de execução *software*. A proteção de controle deve levar em consideração três aspectos: instruções de desvio (condicional ou não), desvios incorretos entre diferentes blocos básicos e desvios incorretos dentro de um mesmo bloco básico. A grande maioria, entretanto, leva em consideração apenas os dois primeiros aspectos, não oferecendo total cobertura de falhas.

A proteção de dados não está inclusa nesse grupo de técnicas e desta forma, falhas em registradores ligados a instruções de desvio condicional que podem gerar erros de controle poderão não ser detectadas. A seguir, algumas das principais técnicas são detalhadas.

2.2.1 Control Flow Checking using Assertions (CCA)

Na técnica chamada CCA, apresentada por (MCFEARIN ET AL., 1995), o programa é dividido num conjunto de blocos básicos (BBs), ou seja, intervalos sem instruções de desvio, ou *Branch Free Intervals* (BFIs). Para cada BB, dois identificadores são assinalados, sendo um deles um identificador de bloco básico, ou *Block Identifier* (BID) e um identificador de fluxo de controle, ou *Control Flow Identifier* (CFID). O primeiro é um valor único para identificar cada BB, enquanto que o segundo é utilizado para representar fluxos de controle, ou seja, transições entre blocos básicos.

A técnica de CCA utiliza os identificadores BID e CFID para monitorar o comportamento do fluxo de controle, assinalando e verificando seus valores durante a execução do programa. Na entrada de um BB, o seu BID único é assinalado a uma variável global. Na saída de um BB, esta variável global é comparada com o BID esperado. Desta maneira, se um erro de controle provocar um desvio no fluxo de programa para o meio de um BB, sem o assinalamento da variável global com seu valor correto, um erro será detectado quando o programa sair do BFI e a verificação apontar uma discrepância entre o valor esperado e o contido na variável global.

O identificador CFID é utilizado para garantir o fluxo correto entre os BBs, ou seja, que a transição entre os blocos básicos seja executada na ordem correta. O CFID é armazenado numa fila de duas posições, inicializada com o CFID do primeiro BB. Ao entrar num BB, o CFID do próximo BB é adicionado à fila, ainda que este seja o próprio BB de origem. Ao sair de um bloco básico, o CFID mais antigo é retirado da fila e verificado com o CFID atual. As operações de adicionar e retirar elementos da fila precisam verificar se ela está num estado de operação correto, ou seja, não está cheia ao adicionar um novo elemento nem vazia ao retirar um elemento, para então realizar a operação. A verificação adiantada da fila reduz a latência de detecção de falhas.

A figura 2.2 mostra o grafo de fluxo de controle de uma operação SE_ENTÃO_SENÃO, enquanto a figura 2.3 mostra o procedimento de assinalamento dos BIDs e CFIDs para a mesma operação. A figura 2.3 mostra, também, o conteúdo da fila de dois elementos e o BID assinalado à variável global em determinados pontos de execução. É importante mencionar que ambos os BBs B e C possuem o mesmo CFID, visto que possuem o mesmo BB de origem, ou seja, o BB A.

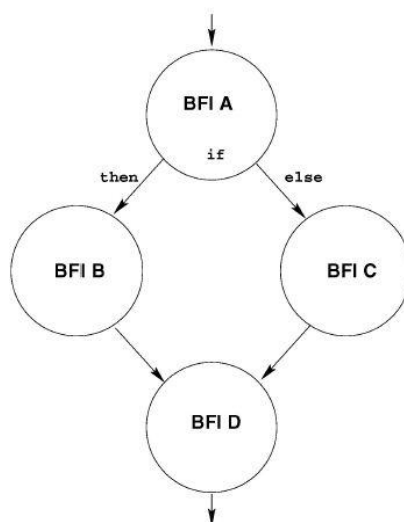


Figura 2.2: Grafo de fluxo de controle (ALKHALIFA, 1999).

A CCA é capaz de detectar erros nas seguintes situações:

1. Ao sair de um bloco básico, a variável global possui um BID diferente do esperado, de acordo com o BB.
2. Ao remover um CFID da fila, ele não confere com o valor de CFID esperado, de acordo com o BB.

3. Ao realizar operações inválidas na fila, ou seja, adicionar elementos na fila cheia ou remover elementos da fila vazia.

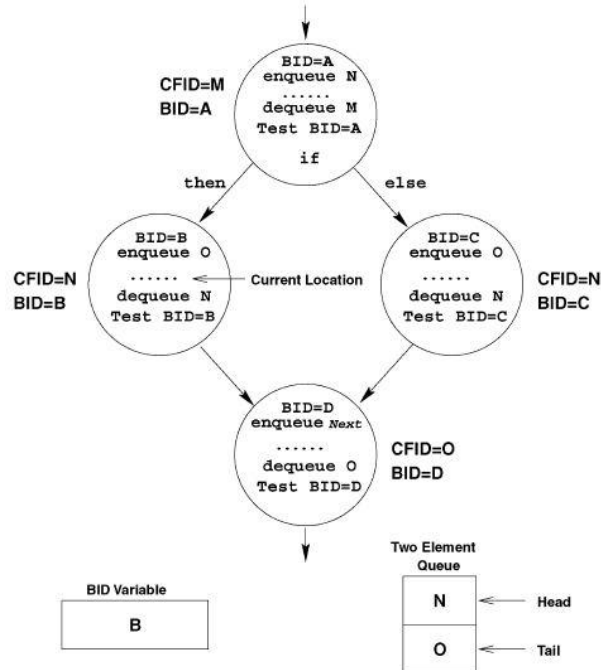


Figura 2.3: Implementação da técnica CCA (ALKHALIFA, 1999).

A implementação da CCA exige três registradores, um para a variável global e dois para a implementação da fila de duas posições. É um valor relativamente pequeno para o alto grau de detecção oferecido. Entretanto, as instruções incluídas para adicionar, retirar e deslocar os elementos da fila, além das instruções para verificar o estado desta, podem aumentar consideravelmente o tempo de execução de um programa, diminuindo assim o desempenho do sistema. Outras instruções para assinalar BIDs e verificá-los também podem prejudicar o sistema, embora em menor escala.

Além destas penalidades, o CCA possui algumas vulnerabilidades. A principal delas são os desvios de fluxo dentro de um mesmo BB, visto que esta técnica não apresenta controle de fluxo interno aos blocos básicos. A segunda vulnerabilidade é relativa ao assinalamento dos CFIDs. Uma vez que todos os filhos de um BB possuem o mesmo CFID, uma transição incorreta entre um BB para o início de um BB filho não será detectada, visto que a verificação dos CFIDs retornará um valor igual ao retirado da fila. A figura 2.4 mostra um exemplo de um grafo de fluxo onde todos os BBs filhos possuem o mesmo CFID, visto que possuem os mesmos BBs de origem. Neste caso, um desvio incorreto do BB B para o início do BB C não será detectado, pois ambos os filhos possuem o mesmo valor de CFID.

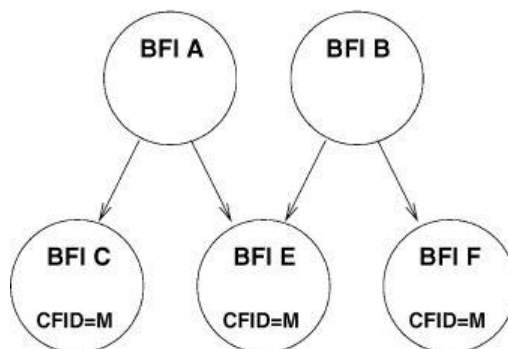


Figura 2.4: Vulnerabilidade no fluxo de controle (ALKHALIFA, 1999).

2.2.2 Enhanced Control Flow Checking using Assertions (ECCA)

A técnica ECCA (ALKHALIFA ET AL., 1999), apresentada em 1999, propôs uma nova metodologia de proteção ao fluxo, ao modificar algumas características da técnica CCA. Semelhante a CCA, a técnica proposta realiza transformações sobre o código utilizando assinaturas de controle. A principal diferença entre estas técnicas é que as assinaturas utilizadas pelas ECCA são do tipo *Real Time Signature* (RTS), ou assinaturas de tempo real, ou seja, são ajustadas durante a execução do programa, assim evitando umas das vulnerabilidades da técnica de CCA, causada pelo fato de blocos básicos com mesmo BB de origem possuírem a mesma assinatura de fluxo de controle (CFID).

Na prática, cada bloco básico, em tempo de execução, modifica seu CFID com uma assinatura de ajuste (RTS) e assinala seus BBs de destino com o resultado (GSR). O bloco básico de destino então verifica a transição, conferindo a assinatura GSR. Conceitualmente, a assinatura RTS combinada com a GSR funciona como uma réplica redundante do PC.

A figura 2.5 exemplifica a transformação do código realizada pela aplicação da técnica ECCA. A instrução 3 calcula a assinatura RTS para o destino da instrução de desvio condicional *br Ll*. Esta assinatura é calculada realizando-se uma instrução de *exclusive OR* (XOR) entre a assinatura do bloco básico com a assinatura do bloco básico de destino. A instrução 4 é a equivalente da instrução 3, utilizada para manter a consistência do fluxo de controle para o restante do programa.

A instrução 5 é utilizada no destino da instrução de desvio para calcular a assinatura do novo bloco, realizando a operação de XOR sobre RTS e o GSR. Esta assinatura é então comparada com a assinatura estaticamente assinalada pela instrução 6 e, na presença de uma diferença, uma sub-rotina de detecção do erro é invocada através da instrução 7.

<pre> (p11) br L1 ... L1: st m[r11]=r12 </pre>	<pre> 3: (p21) xor RTS=sig0,sig1 (p11) br L1 ... 4: xor RTS=sig0,sig1 L1: 5: xor GSR=GSR,RTS 6: cmp.neq.unc p2,p0=GSR,sig1 7: (p2) br faultDetected 8: cmp.neq.unc p3,p0=r11,r21 9: cmp.neq.or p3,p0=r12,r22 10: (p3) br faultDetected st m[r11]=r12 </pre>
(a) Original Code	(b) EDDI+ECC+CFE Code

Figura 2.5: Aplicação da técnica ECCA (OH, 2005).

Outro exemplo pode ser visto na figura 2.6. Se uma falha ocorrer sobre a instrução p11 após a leitura do predicado, a execução permaneceria errada, sem que o erro fosse detectado. Esta falha, entretanto, seria detectada pela técnica ECCA que não só garante que o fluxo de execução seja transferido para um bloco básico válido, mas também que o desvio condicional seja corretamente tomado. Este erro é detectado atualizando dinamicamente a assinatura e checando no início do bloco básico de destino.

A transformação realizada pela ECCA é capaz de detectar uma falha mesmo que a instrução de desvio seja incorretamente tomada (característica que a CCA não possui), uma vez que o registrador contendo a RTS teria um valor incorreto. Assim, esta transformação aumenta a confiabilidade do sistema, quando comparada com as demais técnicas, para falhas sobre o controle de fluxo de programa.

Apesar de apresentar uma taxa de detecção de falhas de controle mais elevada do que a técnica de CCA, chegando até 85% (ALKHALIFA ET AL., 1999), a ECCA traz uma perda de desempenho maior, devido ao cálculo e à verificação da assinatura RTS. Além disso, permanece a vulnerabilidade de desvios incorretos dentro de um mesmo bloco básico, ou seja, a técnica não oferece um controle interno aos blocos básicos, apresentando somente um controle de transição entre eles.

2.2.3 Control-Flow Checking by *Software Signatures* (CFCSS)

A fim de tornar a técnica EDDI robusta contra falhas de controle, (OH ET AL., 2002) propôs uma técnica chamada CFCSS, adicionando instruções de verificação para garantir a correta execução do fluxo de programa. A principal idéia apresentada pelo autor é a combinação desta técnica com a EDDI, visando oferecer uma solução completa para a proteção de sistemas microprocessados. As técnicas são, entretanto, apresentadas separadamente e com objetivos distintos: a primeira visa a proteção dos dados, enquanto a CFCSS visa somente a proteção ao fluxo de controle.

Para controlar o fluxo de controle, a técnica CFCSS, assim como a CCA divide o programa em blocos básicos. Para cada bloco básico é gerada uma assinatura. Uma variável global, chamada *General Signature Register* (GSR) é utilizada para armazenar o valor da assinatura do bloco básico em execução e é utilizada para detectar falhas de controle na execução do programa. Ao entrar num bloco básico novo, é realizada uma operação de XOR sobre o valor de GSR com uma constante determinada estaticamente, assim transformando o valor de GSR para a assinatura do próximo bloco básico. Após a

transformação, o valor de GSR pode ser comparado com o valor esperado, geralmente realizado na saída do bloco básico, a fim de verificar se ocorreu algum desvio de fluxo inesperado.

O uso de uma constante estática para transformar o GSR na próxima assinatura faz com que blocos básicos com um bloco básico de origem comum tenham a mesma assinatura, como na técnica CCA com relação ao CFID. Esta característica traz consigo a vulnerabilidade de um desvio incorreto entre blocos de mesma assinatura. O que pode ser evitado através de uma assinatura calculada em tempo real assinalada a um segundo registrador. Desta maneira é possível realizar a operação de XOR entre a variável GSR, o valor armazenado no segundo registrador e a assinatura estática. Visto que a assinatura calculada em tempo real pode ser diferente dependendo do fluxo de controle, ela pode ser utilizada para diferenciar o GSR entre blocos básicos pai diferentes.

A transformação necessária para a aplicação da CFCSS pode ser vista na figura 2.6. A instrução 8 é responsável por transformar o registrador contendo a variável GSR na assinatura do próximo bloco básico. As instruções 9 e 10 realizam a verificação da assinatura, sendo a 9 responsável por verificar se o valor de GSR está correto e a 10, por realizar um desvio para uma sub-rotina de erro na presença de uma falha.

	3: mov r1=0
	4: (p11) xor r1=r1,1
	5: (p21) xor r1=r1,1
	6: cmp.neq.unc p1,p0=r1,0
	7: (p1) br faultDetected
(p11) br L1	(p11) br L1
...	...
L1:	L1:
	8: xor GSR=GSR,L0_to_L1
	9: cmp.neq.unc p2,p0=GSR,sig_1
	10: (p2) br faultDetected
	11: cmp.neq.unc p3,p0=r11,r21
	12: cmp.neq.or p3,p0=r12,r22
	13: (p3) br faultDetected
st m[r11]=r12	st m[r11]=r12

(a) Original Code

(b) EDDI+ECC+CF code

Figura 2.6: Aplicação da técnica CFCSS (OH, 2005).

Esta técnica pode alcançar em torno de 95% de detecção de erros de controle causando um desvio entre diferentes BBs (VEMU et al., 2007), um valor relativamente baixo, visto que este dado não considera desvios dentro de um mesmo BB. Entretanto, a transformação exige apenas um registrador para armazenar o valor de GSR, apresentando assim um custo mais baixo do que o apresentado pela técnica CCA. O motivo desta baixa taxa de detecção se deve ao fato de que a CFCSS detecta apenas erros de controle entre diferentes blocos básicos e não erros de desvio dentro de um mesmo bloco básico. Além disso, as assinaturas permeáveis demonstram um ponto de vulnerabilidade desta técnica, principalmente em programas do tipo *controlflow*, onde as instruções de desvio são muito frequentes.

2.3 Técnicas de Proteção aos Dados e ao Controle

As técnicas de proteção híbridas procuram uma proteção completa de sistemas microprocessados, visando a proteção tanto dos dados quanto do controle de execução do programa. Para isto, as técnicas compreendem regras de transformação de código, formadas pela composição de diferentes técnicas para a proteção de dados ou de controle.

Apesar de buscarem uma proteção completa do sistema, estas técnicas não visam a proteção da memória física de dados, utilizando para este fim técnicas do tipo *Error Detection and Correction* (EDAC) (capítulo 2.4). A seguir serão apresentadas algumas técnicas de proteção híbridas.

2.3.1 Conjunto de Regras de Transformação Proposto por Rebaudengo

Em 1999, (REBAUDENGO ET AL., 1999) introduziu um conceito de técnicas de tolerância a falhas baseado num conjunto de regras de transformação de código, e não apenas numa técnica específica. As regras de transformação são apresentadas em alto nível de abstração, permitindo assim a fácil alteração dos códigos-fonte e fornecendo assim uma solução completa somente em *software* para a tolerância a falhas em microprocessadores. As regras são divididas em dois grupos: proteção contra erros nos dados e no controle.

Para o grupo de erros afetando os dados do programa, são introduzidas as seguintes regras:

- Regra #1: Toda a variável x deve ser duplicada: seja x_1 e x_2 os nomes das duas cópias;
- Regra #2: Toda a operação de escrita sobre a variável x deve ser realizada sobre x_1 e x_2 ;
- Regra #3: Após cada instrução de leitura sobre x , deve ser realizada a verificação das duas cópias x_1 e x_2 . Em caso de discrepância, um sinal de erro deve ser assinalado.

Através destas regras, as variáveis do programa são duplicadas e mantidas em sincronia, pois cada operação de escrita realizada sobre uma cópia, é obrigatoriamente realizada sobre a segunda cópia. A regra #3 realiza a verificação das cópias sempre que a original é lida por alguma instrução, seja esta instrução aritmética e lógica ou de desvio condicional. A figura 2.7 mostra dois exemplos simples das regras acima aplicadas sobre instruções aritméticas realizadas sobre as variáveis a , b e c .

<i>Original code</i>	<i>Modified Code</i>
<code>a = b;</code>	<code>a₀ = b₀; a₁ = b₁; if (b₀ != b₁) error();</code>
<code>a = b + c;</code>	<code>a₀ = b₀ + c₀; a₁ = b₁ + c₁; if ((b₀!=b₁) (c₀!=c₁)) error();</code>

Figura 2.7: Aplicação das regras sobre dados (REBAUDENGO ET AL., 1999).

As mesmas regras se aplicam aos parâmetros passados e retornados por sub-rotinas, que devem ser vistos como variáveis. As regras #1, #2 e #3 podem ser estendidas desta maneira:

- Regra #1 estendida: Todo parâmetro de uma sub-rotina deve ser duplicado;
- Regra #2 estendida: O valor de retorno de uma sub-rotina deve ser duplicado;
- Regra #3 estendida: A cada leitura de um parâmetro por uma sub-rotina, uma verificação do parâmetro com a sua cópia deve ser realizado. Na presença de uma discrepância, um sinal de erro deve ser assinalado.

A figura 2.8 mostra um exemplo de uma sub-rotina transformada por estas três regras estendidas.

<i>Original code</i>	<i>Modified code</i>
<pre>res = search (a); ... int search (int p) { int q; ... q = p + 1; ... return(1); }</pre>	<pre>search(a0, a1, &res0, &res1); ... void search (int p0, int p1, int *r0, int *r1) { int q0, q1; ... q0 = p0 + 1; q1 = p1 + 1; if (p0 != p1) error(); ... *r0 = 1; *r1 = 1; return; }</pre>

Figura 2.8: Aplicação das regras sobre sub-rotinas (REBAUDENGO ET AL., 1999).

Os erros de controle são classificados pelo autor como sendo aqueles que afetam o assinalamento de variáveis, a computação de instruções aritméticas, comparações de desvios condicionais, *loops*, chamadas de sub-rotinas, retornos de sub-rotinas, dentre outros. A fim de proteger o sistema contra estes tipos de erros, são apresentadas duas regras de transformação. São elas:

- Regra #4: Um inteiro K_i é associado a cada bloco básico i do código;
- Regra #5: É definida uma *Execution Check Flag* (ECF) global, ou seja, uma variável global de verificação. Na entrada de cada bloco básico, a ECF é assinalada com o valor K_i . Ao sair de um bloco básico i , é realizada uma verificação entre ECF e K_i e na presença de uma discrepância, um sinal de erro deve ser assinalado.

Para a aplicação destas regras, é necessário inicialmente identificar todos os blocos básicos do programa, para então transformar o código. Por meio delas, é possível detectar erros cujo efeito é modificar o fluxo de execução do programa e assim levar a execução de instruções errôneas, ou mesmo a não executar instruções que deveriam ser executadas. Um exemplo desta regra de transformação pode ser visto na figura 2.9.

<i>Modified Code</i>	<i>Original Code</i>
<pre>/* basic block beginning */ ... /* basic block end */</pre>	<pre>/* basic block beginning #371 */ ecf = 371; ... if (ecf != 371) error(); /* basic block end */</pre>

Figura 2.9: Aplicação das regras #4 e #5 (REBAUDENGO ET AL., 1999).

A regra #6 realiza a duplicação das instruções de desvio condicional. Na presença de um erro afetando o sistema durante a execução de uma instrução de desvio, a sua réplica é capaz de detectar o erro. A figura 2.10 mostra um exemplo de transformação utilizando a regra #6.

- Regra #6: Para cada comparação de desvios condicionais, o teste é repetido no destino do desvio, tanto para o resultado positivo quanto no negativo. No caso do teste positivo, a instrução de desvio condicional deve ser invertida. Se o resultado dos desvios condicionais inseridos for positivo, um sinal de erro deve ser assinalado.

<i>Original code</i>	<i>Modified Code</i>
<pre>if (condition) { /* Block A */ ... } else { /* Block B */ ... }</pre>	<pre>if (condition) { /* Block A */ if (!condition) error(); ... } else { /* Block B */ if (condition) error(); ... }</pre>

Figura 2.10: Aplicação da regra #6 (REBAUDENGO ET AL., 1999).

Com as regras #7 e #8, é possível detectar erros causando desvios incorretos para o código de uma sub-rotina e erros em chamadas e retornos de sub-rotinas. Um exemplo de aplicação destas regras pode ser visto na figura 2.11.

- Regra #7: Um inteiro K_j é assinalado para cada sub-rotina j do sistema.
- Regra #8: Imediatamente antes do retorno de uma sub-rotina, o valor K_j é assinalado à variável ECF. Uma verificação de K_j e ECF é introduzida após cada chamada de sub-rotina.

<i>Original code</i>	<i>Modified Code</i>
<pre> ... ret = my_proc(a); /* procedure call */ ... /* procedure definition */ int my_proc(int a) { /* procedure body */ ... return(0); } </pre>	<pre> ... /*call of procedure #790 */ ret = my_proc(a); if(ecf != 790) error(); ... /* procedure definition */ int my_proc(int a) { /* procedure body */ ... ecf = 790; return (0); } </pre>

Figura 2.11: Aplicação para chamada e retorno de sub-rotinas (REBAUDENGO, 1999).

As regras de transformação apresentadas pelo autor, quando combinadas, apresentaram uma taxa de detecção de 100% contra falhas injetadas na parte de dados do microprocessador. Entretanto, o mesmo resultado não foi obtido quando falhas foram injetadas na parte de controle e isto se deve ao fato de que as regras combinadas não são capazes de detectar algumas falhas de controle, como desvios incorretos para a primeira instrução de cada bloco básico (assinalamento da ECF) e para o mesmo bloco básico, visto que nenhuma regra oferece proteção contra este último caso.

Em termos de área de ocupação de memória e de desempenho, as regras combinadas mostraram, em média, um aumento de 7,5 vezes no código-fonte; 1,9 vezes no tamanho do código executável e 4,44 vezes o tempo original de computação de um programa. Estes valores são relativamente grandes, visto que com um aumento de duas vezes no tempo de computação, é possível computar o mesmo programa duas vezes e comparar os resultados das duas versões. Com um aumento de três vezes no tempo de computação, seria possível executar o código três vezes, compará-los e corrigir um erro único.

2.3.2 Conjunto de Regras de Transformação Proposto por Nicolescu

Da mesma maneira que REBAUDENGO, (NICOLESCU ET AL., 2003) propôs um conjunto de regras aplicadas em alto nível de abstração para a detecção de erros de controle e de dados, composto por um conjunto com 13 regras de transformação, divididas em três grupos: erros que afetam os dados, erros que afetam instruções básicas e erros que afetam instruções de controle.

O primeiro grupo de regras, que visa detectar falhas que afetam os dados, inicia por definir as relações de dependência entre as variáveis do programa e classificá-las de acordo com o seu papel no programa. Cada variável é classificada como variável intermediária (que são usadas para o cálculo de outras variáveis) ou variável final (não são utilizadas no cálculo de outras variáveis). Para este grupo, são apresentadas as seguintes regras de transformação:

- Identificação da relação entre as variáveis;
- Classificação das variáveis de acordo com o seu papel na execução do programa: variáveis temporais ou variáveis finais;

- Toda variável x deve ser duplicada: seja $x1$ e $x2$ os nomes das duas cópias;
- Toda a operação sobre a variável x deve ser realizada sobre $x1$ e $x2$;
- Após cada instrução de escrita sobre uma variável final, a consistência das duas cópias $x1$ e $x2$ deve ser verificada. Em caso de discrepância, uma sub-rotina de detecção de erro deve ser ativada.

As principais diferenças destas regras para as propostas por Rebaudengo são a identificação das variáveis finais, otimizando assim as verificações das variáveis (visto que as variáveis intermediárias não são verificadas), e o momento no qual as verificações são realizadas. Nas regras de transformação propostas por Nicolescu, a verificação acontece sobre o registrador escrito e apenas após as instruções de escrita terem sido realizadas, enquanto Rebaudengo verifica os registradores lidos e antes da operação ser realizada. A figura 2.12 ilustra as regras acima aplicadas a um trecho de código.

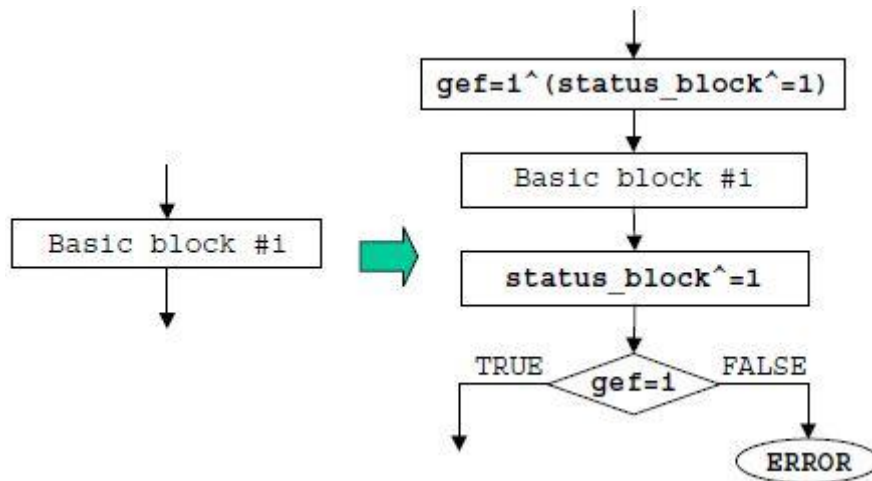


Figura 2.12: Aplicação do primeiro grupo de regras (NICOLESCU, 2003).

O segundo grupo de regras foca a detecção de erros, ou seja, para os erros sobre instruções básicas, são apresentadas as seguintes regras:

- Uma variável booleana *status_flag* é associada para cada bloco básico i . O valor verdadeiro é utilizado para o estado inativo e 0 para o estado ativo;
- Um inteiro K_i é associado para cada bloco básico i ;
- Uma variável global ECF é definida;
- Na entrada de cada bloco básico, a ECF é assinalada com o valor $(K_i \& (status_block = status_block + 1) \bmod 2)$. Ao sair de um bloco básico i , o valor de ECF é verificado.

Estas regras de transformação visam detectar falhas que provocam desvios no fluxo de execução do programa, como um erro no *opcode* de uma instrução aritmética que a transforma numa instrução de desvio, ou ainda um erro no *Program Counter* (PC) do microprocessador. O mecanismo de detecção é muito parecido com o proposto por Rebaudengo. A principal diferença está na variável *status_flag*. A figura 2.13 mostra estas regras sendo aplicadas a um bloco básico genérico $\#i$.

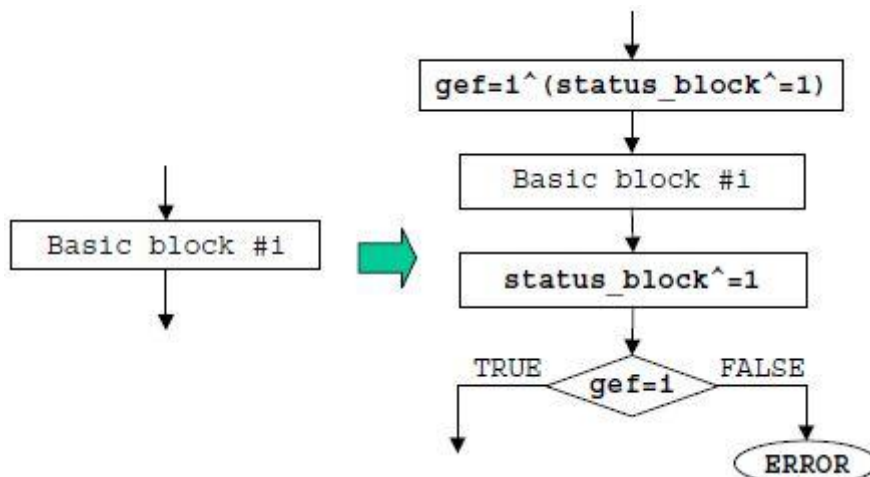


Figura 2.13: Aplicação do segundo grupo de regras (NICOLESCU, 2003).

O último grupo de regras proposto pelo autor busca a proteção de erros que afetam instruções de desvio condicional e as instruções de chamada de sub-rotina. Para este grupo, são introduzidas as seguintes regras:

- Para cada comparação de desvios condicionais, o teste é repetido no destino do desvio, tanto para o resultado positivo quanto no negativo. No caso do teste positivo, a instrução de desvio condicional deve ser invertida. Se o resultado dos desvios condicionais inseridos for positivo, um sinal de erro deve ser assinalado;
- Uma variável *ctrl_branch* é definida;
- Um inteiro K_j é associado para cada sub-rotina j ;
- No início de cada sub-rotina, a variável K_j é assinalada à variável *ctrl_branch*. A verificação da variável *ctrl_branch* é inserida antes e após cada chamada de sub-rotina.

Estas regras são implementadas da mesma maneira que as regras #1, #2 e #4 estendidas e combinadas com a regra #6, propostas por REBAUDENGO.

O conjunto de regras apresentado pelo autor apresentou um aumento no tempo de execução de 144%, aumento de código de programa de 262% e aumento da área de dados em 100%. Em termos de detecção de falhas, estas regras variaram de 56% de detecção para falhas injetadas no controle até 85% para falhas injetadas na parte de dados do microprocessador.

Entretanto, as vulnerabilidades ainda persistem: o conjunto não apresenta nenhum tipo de proteção contra desvios incorretos de fluxo, com fonte e destino no mesmo bloco básico, além de uma detecção precária para falhas inseridas no controle do microprocessador.

2.3.3 Software Implemented Fault Tolerance (SWIFT)

A técnica chamada SWIFT foi apresentada por (REIS ET AL., 2005), tendo como proposta inicial é agrupar as técnicas de EDDI e CFCSS. Ao verificar que a combinação destas técnicas era viável e que otimizações poderiam ser realizadas, assim reduzindo a ocupação de memória e aumentando o desempenho, propôs algumas modificações sobre a combinação das técnicas EDDI e CFCSS.

A primeira otimização é relativa à aplicação da técnica CFCSS sobre todos os blocos básicos do sistema. Considerando que toda a escrita em memória é realizada pelas instruções de *store*, é preciso garantir apenas que estas instruções sejam executadas nos instantes corretos e que estas escrevam os dados corretos na posição de endereço correto. Assim, é possível restringir a aplicação da CFCSS somente aos blocos básicos que possuem instruções do tipo *store*. As atualizações da variável global com a assinatura de cada bloco básico deve ser realizada em todos os blocos básicos, a fim de manter a consistência, mas a verificação pode ser realizada somente naqueles com instruções de *store*.

A remoção da verificação de assinaturas nos blocos básicos sem instruções de *store* é capaz de melhorar o desempenho, principalmente em programas com poucos acessos à memória, pois grande parte das instruções de verificação são removidas. Além disso, a exclusão dessas instruções do código de programa pode reduzir consideravelmente a ocupação da memória de programa, mas possui uma desvantagem: apesar de não modificar a confiabilidade do sistema, o tempo de detecção de uma falha de controle é aumentado, visto que a frequência de verificação das assinaturas é reduzida.

A segunda otimização proposta é o fato de que a verificação das instruções de desvio condicional e a técnica de CFCSS são redundantes. Enquanto a primeira garante que as instruções de desvio são tomadas na direção correta, a CFCSS garante que todas as mudanças no fluxo de programa são realizadas para o endereço de memória correto. Considerando que a garantia da tomada de desvios condicionais para os endereços corretos abrange a garantia de que um desvio tenha sido tomado na direção correta, sabe-se que apenas a CFCSS é suficiente. Assim, é possível remover a verificação dos desvios condicionais e reduzir assim a perda de desempenho e diminuir a ocupação da memória de programa. Esta otimização, assim como a primeira, não reduz a confiabilidade do sistema.

Apesar de melhorar consideravelmente o desempenho e a ocupação da memória de programa, possui duas grandes vulnerabilidades. A primeira delas consiste no atraso entre a validação e o uso de registradores validados e qualquer falha ocorrida durante este atraso pode corromper o estado do programa. Enquanto a maioria das instruções possui alguma forma de redundância, *bit-flips* em endereços de escrita ou em registradores de dados não são detectados, levando o sistema a executar o programa de forma incorreta. O segundo ponto de falha é um erro sobre o *opcode* das instruções, transformando assim uma instrução qualquer em uma instrução de *store*, que não são protegidas, porque não são previstas pelo compilador e, portanto, podem executar livremente e corromper a memória do sistema.

Além destas vulnerabilidades, a técnica de SWIFT continua apresentando algumas características das técnicas de EDDI e CFCSS, como a perda de desempenho e maior ocupação de memória (embora reduzidas em torno de 13% com relação a aplicação das mesmas técnicas sem as otimizações) e algumas vulnerabilidades da técnica de CFCSS se mantém, como a possibilidade de blocos básicos com o mesmo identificador e a não verificação de desvios de fluxo errôneos dentro de um mesmo bloco básico.

2.4 Técnicas de Proteção à Memória de Dados

A arquitetura de um sistema microprocessado envolve pelo menos um microprocessador, uma memória de dados e uma memória de programa (que pode ser a

mesma de dados). As técnicas de tolerância a falhas visam a proteção interna ao microprocessador, mas não são capazes de proteger a memória de dados nem de programa contra falhas. A figura 2.14 mostra as arquiteturas do tipo Harvard (memórias de programa e de dados separadas) de um sistema microprocessado.

Normalmente, a memória de programa é do tipo estática, pois esta não pode perder os dados cada vez que o sistema é desligado ou reiniciado, mesmo que isto aumente o caminho crítico do sistema. Assim, o tipo de memória utilizado geralmente é *Read Only Memory* (ROM), sendo estas *Electrically Programmable Read Only Memory* (EPROM), *Electrically Erasable Programmable Read Only Memory* (EEPROM) ou *flash*. Estas memórias, devido a sua organização interna de escrita com altas voltagens e utilização de transistores com *gate* flutuante, são tolerantes a radiação.

A memória de dados, por outro lado, tem seus valores alterados frequentemente e deve operar em frequências maiores. Em geral, são utilizadas memórias do tipo *Random Access Memory* (RAM) *Static Random Access Memory* (SRAM) ou *Dynamic Random Access Memory* (DRAM), que são extremamente sensíveis a radiação, devido a sua organização interna de armazenamento de dados em capacitores. A memória de dados, portanto, deve ser protegida de alguma forma para a obtenção de um sistema tolerante a falhas. Alguns sistemas utilizam uma memória do tipo RAM entre o microprocessador e a memória de programa (ROM) para aumentar o desempenho do sistema. Nestes casos, também a memória de programa deve ser protegida.

As técnicas de proteção à memória de dados podem ser implementadas em *software*, mas não serão mencionadas neste trabalho, visto que são constantemente combinadas com as técnicas implementadas em *hardware*. Estas técnicas, também chamadas de EDAC, são baseadas na redundância de informação, ou seja, o tamanho das palavras é aumentado e é adicionado um espaço para o armazenamento de informação redundante, como bits de paridade, por exemplo. Porém devido ao alto custo de implementação em *software*, é mais comum implementar os códigos de correção de erros em *hardware* entre o processador e a memória, como na figura 2.14.

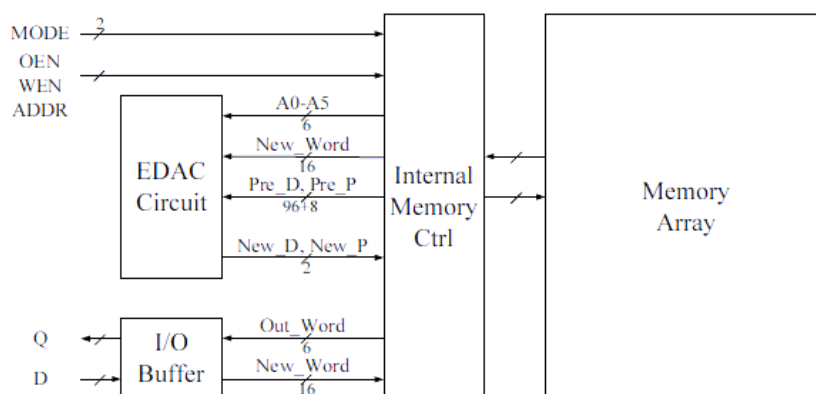


Figura 2.14: Arquitetura de EDAC sobre memória (CHEN et al., 2010)

A técnica mais simples é a paridade de bits. A paridade é capaz de detectar erros simples, inclusive no próprio bit de paridade. Entretanto, esta técnica não é capaz de corrigir erros. Sua implementação, por outro lado, é simples, visto que uma simples porta lógica XOR implementa a função de paridade ímpar, enquanto uma porta lógica XNOR implementa a função de paridade par. Existem dois tipos de paridade: a par e a ímpar, sendo que em ambas é adicionado um bit às palavras de memória, informando se a quantidade de 1's da palavra é par (paridade par), ou ímpar (paridade ímpar). Por

exemplo, a palavra 1111 receberia o bit extra '0' no caso da paridade ímpar, visto que a palavra possui um número par de 1's (4), ou o bit extra '1' no caso da paridade par, devido a quantidade ímpar de 1's.

A técnica de código de *hamming*, introduzida em 1950, tem o mesmo princípio da paridade, devido a utilização de bits redundantes. A principal diferença entre elas é que o código de *hamming* utiliza um número maior de bits redundantes, de acordo com o tamanho da palavra e os arranja de forma intercalada com a palavra. Para sua implementação, é necessário um circuito mais complexo do que a paridade, embora de baixo custo de implementação, e uma maior quantidade de bits extras, mas oferece a detecção e mesmo correção de falhas múltiplas dentro de uma palavra. Devido a seu baixo custo, a maioria das memória do tipo RAM utilizam esta técnica intrinsecamente.

Devido a alta densidade das memórias nanométricas, múltiplos bits podem ser atacados e códigos de correção de erros capazes de corrigi-los devem ser empregados. Um exemplo é o código Reed-Solomon que é capaz de corrigir grupos de bits. A escolha do código EDAC deve ser feito com base de suscetibilidade das memórias utilizadas no sistema e o impacto do código de correção deve ser considerado no projeto final do sistema processador mais memória.

Neste trabalho, falhas transientes (SEU) na memória de dados e programa não serão consideradas já que é assumida a existência de técnicas de EDAC nelas, capazes de corrigir estes erros. Com isso, os impactos aqui medidos e apresentados levam em consideração somente os efeitos das falhas transientes (SEU e SET) no processador e as técnicas utilizadas para proteger tais falhas.

3 METODOLOGIA PROPOSTA

Este capítulo descreve a metodologia proposta e utilizada neste trabalho, que parte da descrição do microprocessador, passa pelas regras de transformação de código utilizadas para a proteção de códigos de programa e finaliza com a apresentação da metodologia de injeção de falhas utilizada para a verificação das técnicas de tolerância a falhas baseadas em *software*.

3.1 Microprocessador miniMIPS

O microprocessador miniMIPS é um processador baseado na arquitetura MIPS (*Microprocessor without Interlocked Pipeline Stages*), porém com um subconjunto reduzido de 52 instruções do processador original. É baseado na arquitetura de Von Neumann, onde a memória de dados e programa é unificada. A versão usada miniMIPS possui organização RISC (*Reduced Instruction Set Architecture*) com um *pipeline* de cinco estágios: *Instruction Address Calculation* (PC), *Instruction Fetch* (FETCH), *Instruction Decode* (DECODE), *Execution* (EXEC) e *Memory Access* (MEM), como ilustrado na figura 3.1. Com base nisso, a cada ciclo de relógio, uma nova instrução é buscada da memória (PATTERSON, 2007).

A escolha deste microprocessador como base para testes se deve principalmente a sua ampla utilização tanto na literatura como nos grupos de pesquisa da UFRGS. Além disso, o microprocessador em questão pode ser facilmente sintetizado para as placas disponíveis no Instituto de Informática e assim ser submetido a uma campanha física de irradiação de partículas energizadas.

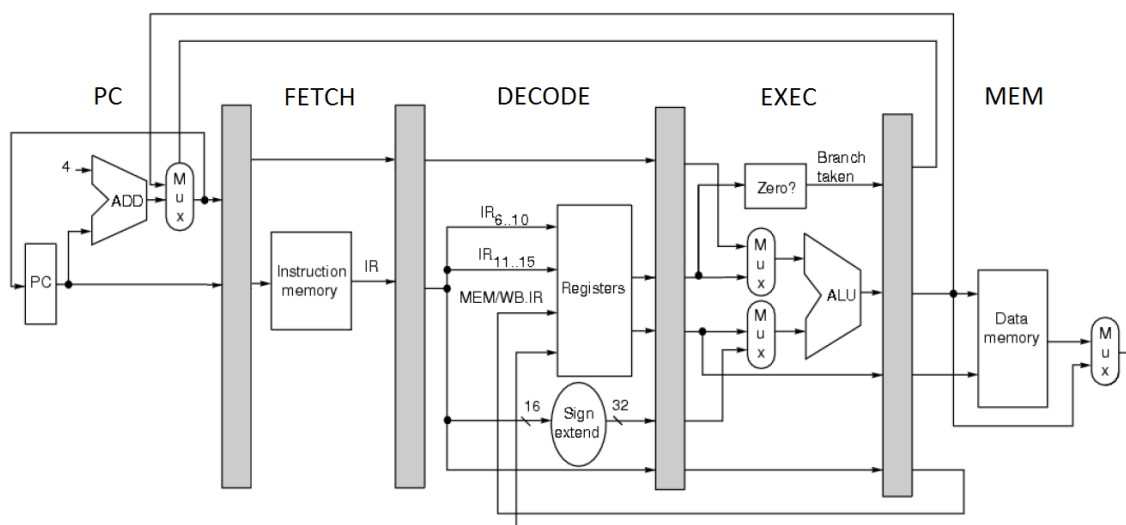


Figura 3.1: Arquitetura miniMIPS *pipeline*.

As instruções utilizadas pelo miniMIPS possuem tamanho fixo de 32 bits, sendo que apenas duas instruções acessam a memória: uma instrução de leitura (*load*) e uma instrução de escrita (*write*). As instruções possuem cabeçalho fixo de 6 bits e são divididas em 3 classes: instruções do tipo R, que especificam 3 registradores (*rs*, *rt* e *rd*), um campo para deslocamento (*shamt*) e um campo para função (*funct*); instruções do tipo I que possuem dois campos para registrador (*rs* e *rt*) e um valor de 16 bits imediato (*immediate*); e instruções do tipo J que, além do cabeçalho (*opcode*), possuem um campo de endereço de 26 bits. A figura 3.2 mostra cada classe de instrução.

Type	-31-	format (bits)					-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)	
I	opcode (6)	rs (5)	rt (5)	immediate (16)			
J	opcode (6)	address (26)					

Figura 3.2: Classes de instruções do miniMIPS.

O microprocessador miniMIPS possui um registrador de endereço de instrução, chamado *Program Counter* (PC). A lógica de próxima instrução deste processador é simples, uma vez que as instruções possuem tamanho fixo e a predição de desvio é estática. Além do PC, existem outros registradores especiais, como o *Stack Pointer* (SP), ou registrador de pilha, e o registrador com a constante zero (R0), que se encontram no banco de registradores.

O processo de compilação para o miniMIPS, que pode ser visto na figura 3.3, envolve as seguintes etapas:

- Programação na linguagem C;
- Compilação do código C para arquivo do tipo ELF (*Executable and Linking Format*), ou Formato Executável e de Ligação;
- Tradução do arquivo ELF para arquivo do tipo *assembly*;
- Tradução do arquivo *assembly* para arquivo em linguagem VHDL para o miniMIPS, descrito através de uma cadeia de 0's e 1's.

A versão do miniMIPS utilizada neste trabalho foi a desenvolvida inicialmente pela universidade ENSERG (*Ecole Nationale Supérieure d'Electronique et de Radioélectricité de Grenoble*) e disponibilizada em (OPENCORES, 2009). Entretanto, teve seu módulo de predição de desvios modificada e outros detalhes menores corrigidos. Apesar das alterações com relação ao código original disponível, o núcleo do sistema se manteve o mesmo.

A organização do microprocessador pode ser dividida internamente em dois grupos: *controlpath* e *datapath*, sendo o segundo subdividido entre unidade lógica e aritmética (ULA), banco de registradores e outros. A seguir, cada uma delas será detalhada.

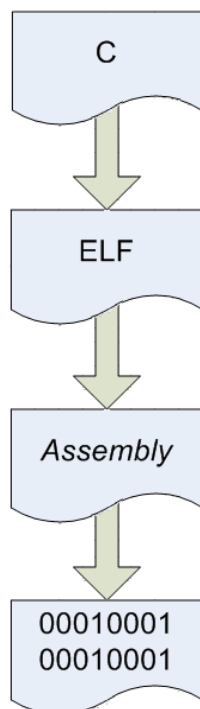


Figura 3.3: Fluxo de compilação para o miniMIPS.

3.1.1 Datapath

Tendo uma arquitetura baseada em *pipeline*, a divisão entre o *datapath* e o *controlpath* é bastante tênua, uma vez que muitos sinais internos do microprocessador afetam tanto o fluxo de dados quanto o de controle de execução do programa. Assim, foi adotado que o *datapath* corresponde a todos os componentes que fazem o caminho entre duas células de armazenamento (banco de registradores ou memória), passando obrigatoriamente pela ULA.

A ULA, apesar de pertencer ao *datapath*, foi adicionada a um subgrupo, a fim de se obter uma melhor classificação das falhas que ocorrem na mesma e impactam na aplicação. Da mesma maneira, o banco de registradores também foi adicionado a um subgrupo particular, apesar de armazenar dados em seus registradores e, portanto, fazer parte do *datapath*. Sendo assim, restou ao grupo outros somente as estruturas que ligam estes módulos.

A ULA é responsável por todas as operações lógicas e aritméticas do microprocessador e, conseqüentemente, por todo o processamento de dados do miniMIPS. Por esta razão, diversas técnicas de tolerância a falhas, tanto em *software* quanto em *hardware*, endereçam somente a ULA, desconsiderando o restante dos módulos de dados. É interessante, portanto, verificar o efeito de falhas afetando somente a ULA. Suas operações são divididas entre (1) aritméticas: soma, subtração e multiplicação e (2) lógicas: and, nor, or, xor, e deslocamento. A maioria destas operações operam sobre registradores ou valores constantes, bem como sobre números com ou sem sinal (no caso das operações aritméticas).

O banco de registradores do miniMIPS é composto por 32 registradores de 32 bits, totalizando 1024 bits de armazenamento, ou 1Kb. Além dos registradores, este grupo é composto pelos sinais de seleção do banco de registradores, como os sinais de leitura e escrita e seleção. Dentre os 32 registradores, alguns são tratados como registradores

especiais. É o caso do registrador R0, responsável por armazenar o valor lógico 0, que não possui acesso de escrita. Da mesma maneira, encontra-se no banco de registradores o registrador SP, ou registrador de pilha. A relação completa dos registradores pode ser vista na figura 3.4.

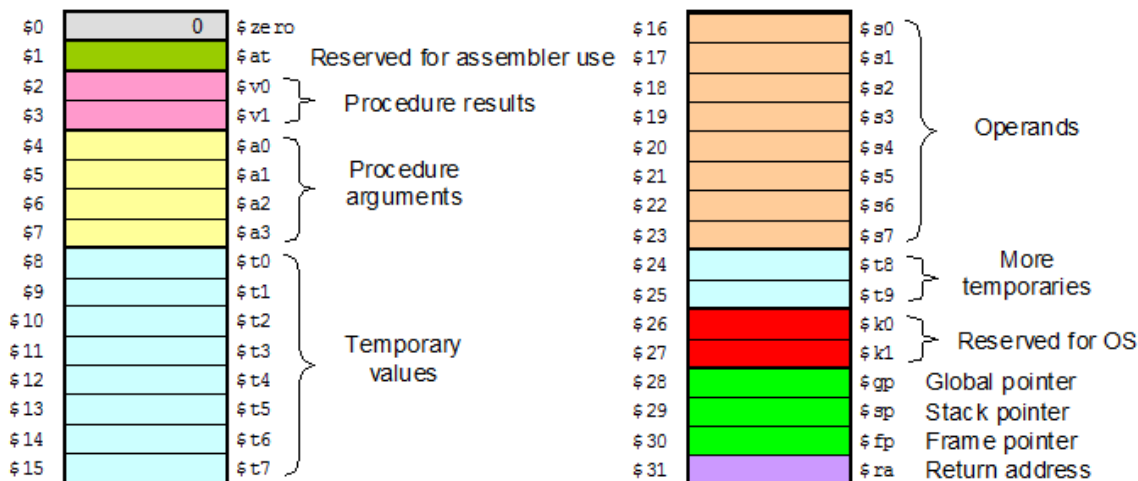


Figura 3.4: Conjunto de registradores.

3.1.2 Controlpath

O *controlpath* é composto por todas as estruturas que influenciam no controle do fluxo de execução do programa, ou seja, todas as estruturas que não fazem parte do *datapath* nem das estruturas de armazenamento, compostas pela memória (pois esta é externa ao microprocessador) e pelo banco de registradores. O *controlpath*, portanto compreende a lógica de próxima instrução, previsões de desvio, decodificação de instruções, além dos registradores das barreiras de *pipeline* responsáveis pelo armazenamento dessas informações.

3.2 Conjunto de Técnicas Baseadas em Software Implementadas em Linguagem Assembly

A aplicação de técnicas de tolerância a falhas pode ser executada em diferentes níveis de abstração. Podem ser aplicadas em alto nível, através de um programa escrito na linguagem C, passando por um nível intermediário, sobre um programa escrito na linguagem *assembly*, chegando até o mais baixo nível possível, quando aplicadas diretamente sobre a imagem de memória do microprocessador (*memory dump*). Quanto maior o nível de abstração, mais fácil é a aplicação da técnica sobre o código, uma vez que a interpretação do código é mais simples e, em alguns casos, independente de plataforma.

Apesar da maior facilidade na aplicação das técnicas num alto nível de abstração, maior deve ser o controle sobre as etapas de compilação. Isto acontece devido às otimizações realizadas pelos compiladores sobre o código, podendo retirar as redundâncias inseridas pelas técnicas de tolerância a falhas. Alguns compiladores podem também inverter a ordem de execução das instruções, assim diminuindo ou mesmo anulando o efeito das técnicas aplicadas.

Da mesma maneira, quanto menor o nível de abstração, menor deve ser o controle sobre o compilador, chegando ao ponto em que o compilador não é mais utilizado, como, por exemplo, a nível de código de máquina. Por outro lado, a interpretação do código se torna mais difícil devido a perda de algumas características retiradas do código em alto nível, como o início e fim de blocos básicos e chamadas e retornos de sub-rotinas.

A independência de compilador é uma característica interessante, pois o código-fonte não fica amarrado a um fluxo de compilação. Da mesma maneira, o usuário fica livre para realizar as otimizações de código que achar necessário, sem ter de implementar um controle maior sobre o processo de compilação. Em termos de implementação das regras de transformação, algumas dificuldades são introduzidas, como uma análise mais complexa do código, podendo chegar ao ponto de emular parte deste a fim de encontrar endereços de desvios condicionais e retornos de sub-rotinas. Entretanto, os ganhos com a independência sobre o compilador justificam a escolha pela aplicação das regras de transformação a nível de código de máquina.

A fim de analisar separadamente diferentes técnicas de tolerância a falhas, foram criados grupos contendo um conjunto de regras de transformação do código, baseado nos trabalhos relacionados apresentados no capítulo 2. Com a separação em grupos e uma análise detalhada dos resultados de injeção de falhas, é possível identificar pontos de vulnerabilidade da cada grupo e, em alguns casos, de regras de transformação em específico. Da mesma maneira, a separação em grupos permite a combinação dos mesmos, visando uma maior proteção do sistema.

A seguir, é descrito cada grupo de regras utilizado neste trabalho e as regras de transformação que os compõem.

3.2.1 Técnica de Tolerância em Assembly: SIG - Assinaturas

O grupo de regras de transformação chamado de assinaturas tem a função de proteger o fluxo de controle do programa, adicionando uma variável global ao sistema, chamada ECF, e assinalando um identificador único para cada bloco básico do programa. Através deste grupo de regras, espera-se detectar desvios incorretos no fluxo de execução do programa com origem e destino em diferentes blocos básicos.

Este grupo é composto pelas seguintes regras de transformação:

- Regra #1: Um inteiro K_i é associado a cada bloco básico i do código;
- Regra #2: É definida uma ECF global;
- Regra #3: Na entrada de cada bloco básico, a ECF é assinalada com o valor K_i ;
- Regra #4: Ao sair de um bloco básico i , é realizada uma verificação entre ECF e K_i . Na presença de uma discrepância, um sinal de erro deve ser assinalado.

Um exemplo da transformação realizada por este conjunto de regras pode ser visto na figura 3.5, onde a variável ECF é armazenada no registrador rX. Neste exemplo, existem 2 blocos básicos, compostos pelas instruções 3 e pelas instruções 6 e 7. No início de cada um deles foi adicionada uma instrução para assinalar os valores de "signature 1" e "signature 2" no registrador rX, através das instruções 2 e 5. Na saída de

cada um dos blocos, são adicionadas as instruções 4 e 8, para verificar o valor armazenado em rX.

<code>beq r1, r2, 6</code>	1: <code>beq r1, r2, 6</code>
<code>add r2, r3, 1</code>	2: <code>mv rX, signature 1</code> 3: <code>add r2, r3, 1</code> 4: <code>bne rX, signature 1, error</code>
<code>add r2, r3, 9</code> <code>st [r1], r2</code>	5: <code>mv rX, signature 2</code> 6: <code>add r2, r3, 9</code> 7: <code>st [r1], r2</code> 8: <code>bne rX, signature 2, error</code>
<code>jmp end</code>	9: <code>jmp end</code>

Figura 3.5: Transformação Assinaturas 1.

3.2.2 Técnica de Tolerância em Assembly: BBD - Divisão de blocos básicos

Dentre todas as técnicas apresentadas no capítulo 2, nenhuma foi capaz de detectar falhas de controle causando um desvio de fluxo com origem e destino dentro de um mesmo bloco básico, como, por exemplo, a não execução de uma instrução. Este grupo propõe uma maneira de reduzir este tipo de falhas, diminuindo o tamanho dos blocos básicos e utilizando a técnica de assinaturas, descrita em 3.2.1.

O grupo de divisão de blocos básicos não é composto de regras de transformação, mas sim de uma limitação de tamanho durante a definição dos blocos básicos do programa e pode ser resumido pela seguinte regra:

- Um bloco básico deve possuir no máximo i instruções, devendo ser dividido sempre que a quantidade de instruções ultrapassa este valor.

3.2.3 Técnica de Tolerância em Assembly: VAR1 - Variáveis 1

O grupo de regras chamado variáveis 1 é o principal responsável pela detecção de erros sobre os dados do sistema. A proteção oferecida por este grupo é também responsável pelas maiores perdas de desempenho e ocupação de memória (tanto de dados quanto de programa). Baseado nas regras introduzidas por REBAUDENGO, o VAR1 tem como metodologia a duplicação de todas as variáveis do sistema, tanto as armazenadas em memória quanto em registradores, e a verificação constante das mesmas. Cada registrador utilizado pelo programa é replicado sobre os registradores não utilizados. Da mesma maneira, as posições de memória ocupadas pelo programa são replicadas sobre as posições não ocupadas.

Neste grupo, as verificações de consistência das variáveis e suas cópias são realizadas sempre que um registrador é lido por uma instrução, fazendo com que até n registradores sejam verificados, onde n é o número máximo de registradores utilizado por uma instrução.

Implementada somente em *software*, este grupo de regras não é capaz de aumentar o número de registradores nem de memória disponível e, eventualmente, não haverá posições de armazenamento livres (principalmente no caso de registradores) para a duplicação das variáveis. Neste caso, é possível a utilização de técnicas de otimização

de memória, como a técnica de renomeação de registradores, onde o compilador renomeia registradores e mesmo posições de memória a fim de uma menor utilização dos mesmos. Ainda assim, existem casos em que não será possível reduzir a quantidade de posições de memória para a metade da quantidade total (permitindo assim a sua duplicação). Nestes casos, é preciso realizar uma análise sobre o código a ser transformado e escolher um conjunto das unidades de armazenamento (registradores ou posições de memória) para a duplicação, baseado na sua utilização.

Este grupo é composto pelas seguintes regras de transformação:

- Regra #1: Toda a variável x deve ser duplicada: seja $x1$ e $x2$ os nomes das duas cópias;
- Regra #2: Toda a operação de escrita sobre a variável x deve ser realizada sobre $x1$ e $x2$;
- Regra #3: Antes de cada instrução de leitura sobre x , deve ser realizada uma verificação das duas cópias $x1$ e $x2$. Em caso de discrepância, um sinal de erro deve ser assinalado.

A figura 3.6 mostra o resultado desta transformação sobre instruções, registradores e posições de memória. As instruções 1 e 3 são utilizadas para proteger a instrução de *load* (instrução 2), sendo a primeira para verificar o registrador base de endereço ($r4$) com sua réplica ($r4'$) e a segunda para replicar a instrução de *load*, carregando o valor de memória sobre a cópia de $r1$, ou seja, $r1'$. As instruções 8, 9 e 11, são utilizadas para proteger a instrução de *store*. Enquanto as duas primeiras são utilizadas para verificar o valor dos registradores de base e de destino ($r1$ e $r2$, respectivamente) com suas cópias ($r1'$ e $r2'$), a instrução 11 é utilizada para replicar a instrução sobre uma nova posição de memória, distanciada pelo valor de *offset* da posição original.

Além da replicação de variáveis armazenadas em memória, a figura 3.6 mostra a replicação de variáveis em registradores, realizada pelas instruções 4, 5 e 6. As duas primeiras são utilizadas para verificar os valores dos registradores lidos pela instrução de soma ($r3$ e $r4$) com a suas réplicas ($r3'$ e $r4'$, respectivamente), enquanto a instrução 7 é utilizada para replicar a instrução 6, do código original.

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

Figura 3.6: Transformação de registradores Variáveis 1.

3.2.4 Técnica de Tolerância em Assembly: VAR2 - Variáveis 2

O grupo variáveis 2 é proposto como uma alternativa ao grupo VAR1, tendo sido baseado nas técnicas para a proteção de dados propostas por VELAZCO, modificando os pontos de verificação das variáveis. Com este grupo, espera-se menor ocupação de memória de programa e mais desempenho, quando comparado ao VAR1.

Diferentemente do VAR1, que realiza verificações de consistência das variáveis lidas por instruções, o grupo VAR2 verifica os registradores escritos pelas instruções, assim reduzindo para, no máximo, uma verificação por instrução. A redução na quantidade de registradores verificados reduz a quantidade de instruções necessárias e, conseqüentemente, da memória de programa utilizada e aumenta o desempenho do sistema. A taxa de detecção, teoricamente, deve permanecer inalterada, uma vez que o valor final armazenado no registrador escrito terá sua consistência checada com a sua cópia.

Algumas instruções não escrevem em registradores, mas precisam ser verificadas, como é o caso da instrução de *store*. Para estas instruções, a mesma técnica utilizada no VAR1 é utilizada.

As regras que descrevem a transformação são as seguintes:

- Regra #1: Toda variável x deve ser duplicada: seja $x1$ e $x2$ os nomes das duas cópias;
- Regra #2: Toda a operação sobre a variável x deve ser realizada sobre $x1$ e $x2$;
- Regra #3: Após cada instrução de escrita sobre um registrador, a consistência das duas cópias $x1$ e $x2$ deve ser verificada. Em caso de discrepância, um erro deve ser sinalizado.

A transformação resultante da aplicação destas regras pode ser vista na figura 3.7. Como se pode ver, o resultado da transformação é semelhante à VAR1 (figura 3.6). A principal diferença está na proteção da instrução 5, que utilizará apenas uma instrução de verificação (instrução 6), e não duas, como é o caso da técnica de VAR1. A instrução de *load* (instrução 1) possui apenas um registrador lido e outro escrito e portanto não apresenta diferença de desempenho com relação a VAR1, embora a instrução de verificação (instrução 3) seja adicionada após a operação e sua réplica (instrução 2). A instrução de *store* (instrução 9) não escreve sobre nenhum registrador e, por esta razão, a sua proteção é realizada pelas técnicas do VAR1, através das instruções 7 e 8 para a verificação dos registradores lidos e da instrução 10 para a duplicação da instrução original.

<code>ld r1, [r4]</code>	1: <code>ld r1, [r4]</code> 2: <code>ld r1', [r4' + offset]</code> 3: <code>bne r1, r1', error</code>
<code>add r1, r2, 1</code>	4: <code>add r1, r3, r4</code> 5: <code>add r1', r3', r4'</code> 6: <code>bne r1, r1', error</code>
<code>st [r1], r2</code>	7: <code>bne r1, r1', error</code> 8: <code>bne r2, r2', error</code> 9: <code>st [r1], r2</code> 10: <code>st [r1' + offset], r2'</code>

Figura 3.7: Transformação Variáveis 2.

3.2.5 Técnica de Tolerância em Assembly: VAR3 - Variáveis 3

O grupo variáveis 3 surge como alternativa aos grupos VAR1 e VAR2, oferecendo proteção aos dados do sistema. Baseado em REIS e na técnica de SWIFT, sabe-se que para garantir o correto funcionamento do sistema, basta manter a consistência dos dados que são escritos na memória. Este grupo, portanto, limita a verificação de consistência das variáveis somente para os registradores utilizados pelas instruções de acesso a memória, ou seja, para os registradores utilizados como base para o cálculo de endereço (tanto na instrução de *store* quando de *load*) e para os registradores que armazenam o dado a ser escrito na memória (somente para as instruções de *store*).

A menor quantidade de registradores verificados diminui a quantidade de verificações e, conseqüentemente aumenta o tempo de detecção do sistema, o que pode ser prejudicial em sistemas onde a quantidade de falhas é alta. Isto acontece pelo fato de que uma falha será detectada apenas quando seu resultado for gravado em memória, o que pode demorar diversos ciclos de relógio, enquanto que os grupos VAR1 e VAR2 detectam uma falha assim que o registrador é utilizado numa operação qualquer.

Em compensação, espera-se que este grupo apresente melhor desempenho e menor ocupação da memória de programa, quando comparado com os grupos VAR1 e VAR2. A ocupação da memória de dados, por outro lado, se mantém, uma vez que as variáveis permanecem sendo replicadas.

As regras que descrevem este grupo são as seguintes:

- Regra #1: Toda variável x deve ser duplicada: seja $x1$ e $x2$ os nomes das duas cópias;
- Regra #2: Toda a operação sobre a variável x deve ser realizada sobre $x1$ e $x2$;
- Regra #3: Antes de cada instrução de escrita em memória, os registradores utilizados pela instrução devem ser verificados com suas cópias. Em caso de discrepância, um erro deve ser sinalizado.

A figura 3.8 exemplifica a transformação segundo as regras acima. Diferentemente dos grupos VAR1 e VAR2, a instrução de soma (instrução 4) não tem seus registradores verificados. Entretanto, as instruções de *load* e de *store* realizam verificações de

consistência entre seus registradores e suas cópias. A instrução de *load* possui apenas um registrador lido e portanto a técnica utilizada para sua proteção (VAR1 ou VAR2) é indiferente. A instrução de *store*, por outro lado, não pode ser protegida pelo VAR2, e por esta razão é protegida através do grupo VAR1.

<code>ld r1, [r4]</code>	1: bne r4, r4', error 2: <code>ld r1, [r4]</code> 3: ld r1', [r4' + offset]
<code>add r1, r2, 1</code>	4: <code>add r1, r3, r4</code>
<code>st [r1], r2</code>	5: bne r1, r1', error 6: bne r2, r2', error 7: <code>st [r1], r2</code> 8: st [r1' + offset], r2'

Figura 3.8: Transformação Variáveis 3.

Os desvios condicionais também tem seus registradores verificados, a fim de evitar erros no fluxo de execução do programa que indiretamente podem prejudicar as instruções de escrita em memória.

3.2.6 Técnica de Tolerância em Assembly: BRA - Desvios condicionais

O grupo de regras de transformação chamado de desvios condicionais fornece uma proteção contra falhas que afetam tanto o fluxo de execução do programa quanto os dados. Sua principal função é replicar os desvios condicionais, da mesma maneira que os grupos de VAR1, VAR2 e VAR3 fazem com as demais instruções.

A replicação de desvios condicionais vai além da simples duplicação da instrução, visto que todas estas instruções geram uma divisão no fluxo de execução do programa: quando o desvio é tomado ou não. A replicação deve, portanto, ser realizada nos dois possíveis fluxos de uma instrução de desvio condicional. Quando o desvio não é tomado, a replicação do desvio condicional é simples: basta replicar a instrução na posição de memória de programa seguinte, modificando o endereço de desvio para uma sub-rotina que informe um erro. Desta maneira, se o desvio não for tomado na instrução original, também não será tomado na instrução replicada, a menos que um erro afete o programa.

No caso do desvio ser tomado, ou seja, da ocorrência de um desvio no fluxo de execução, a instrução replicada deve ser invertida, de maneira que o desvio replicado seja tomado apenas na presença de uma falha, assim executando uma sub-rotina que informe o erro ao sistema. A instrução replicada, após invertida, deve ser inserida no endereço de destino do desvio.

Outro fator a ser levado em consideração é que a instrução replicada, invertida e inserida no fluxo de execução onde o desvio é tomado, não pode interferir no fluxo original de execução, ou seja, ela só pode ser executada quando precedida do desvio original. Para que isto aconteça, é necessário adicionar uma instrução de desvio incondicional para contornar a instrução replicada, fazendo com que esta seja acessada somente pelo desvio condicional. A figura 3.9, apresentada a seguir, exemplifica este procedimento.

As regras que descrevem este grupo são as seguintes:

- Regra #6: Cada instrução de desvio condicional é replicada em ambos os possíveis fluxos de execução. No caso do desvio não ser tomado, a instrução é duplicada com seu endereço de desvio apontando para uma sub-rotina de erro. No caso do desvio ser tomado, a instrução de desvio deve ser invertida e adicionada no endereço de destino da instrução original, tendo seu endereço de desvio modificado para uma sub-rotina de erro;
- Regra #7: Imediatamente antes da instrução replicada e invertida deve ser inserida uma instrução de desvio incondicional.

A figura 3.9 exemplifica a transformação segundo as regras acima. A instrução de desvio condicional *Branch if Equal* (BEQ), ou desvie se igual, localizada na posição 1, realizará um desvio de fluxo se os registradores r1 e r2 estiverem armazenando os mesmos valores. Inicialmente, o desvio será replicado e inserido após a instrução 1, através da instrução 2. A instrução BEQ é então invertida com a utilização da instrução de desvio condicional *Branch if Not Equal* (BNE), ou desvio se diferente, e inserida no destino da instrução original, através da instrução 5.

A instrução 5, entretanto, não pode ser executada após a instrução 3, uma vez que esta poderia modificar o valor armazenado no registrador r2 e assim causar uma falsa detecção de erro, devido à instrução 5. Para isto, é adicionada a instrução de desvio incondicional 4, tornando a instrução 5 acessível somente através da instrução de desvio condicional original (instrução 1).

<code>beq r1, r2, 6</code>	1: <code>beq r1, r2, 5</code> 2: <code>beq r1, r2, error</code>
<code>add r2, r3, 1</code>	3: <code>add r2, r3, 1</code>
	4: <code>jmp 6</code> 5: <code>bne r1, r2, error</code>
<code>add r2, r3, 9</code> <code>jmp end</code>	6: <code>add r2, r3, 9</code> 7: <code>jmp end</code>

Figura 3.9: Transformação Desvios condicionais.

3.3 HPCT: Ferramenta de Transformação de Código em Linguagem Assembly usada Pós-Compilação

A transformação de um código de programa é uma tarefa complexa, que requer diversas etapas de análise, processamento e replicação de instruções. Dependendo da técnica implementada, a dificuldade de transformação também se altera. A própria construção dos blocos básicos da aplicação (etapa requerida por parte das técnicas de tolerância a falhas e dos grupos de regras de transformação propostos) pode ser um trabalho exaustivo, quando realizado manualmente. Com o intuito de automatizar a aplicação das regras de transformação, foi desenvolvida uma ferramenta chamada Ferramenta de Transformação de Código Pós-Compilação, ou *Hardening Post Compiling Translator* (HPCT).

A HPCT foi desenvolvida na linguagem Java, devido a sua portabilidade, onde qualquer sistema operacional compatível com o *Java Runtime Environment* (JRE) pode executá-la e devido a sua facilidade na manipulação de *strings* e funções de *parsing* de texto. Além dessas características, a implementação da ferramenta na linguagem Java possibilita a combinação do HPCT com outras ferramentas de injeção e análise e falhas, como o próprio Injetor de falhas, descrito no capítulo 3.4. A estrutura da linguagem Java, orientada a objetos, permite também a modificação e inserção de novas técnicas de tolerância a falhas, com a inserção de novas classes que implementam as técnicas desejadas.

Como pode ser visto na figura 3.10, a HPCT tem como entradas um arquivo de definição do microprocessador, contendo basicamente sua *Instruction Set Architecture* (ISA) dividida em grupos segundo o formato das instruções, o código a ser transformado e as técnicas a serem aplicadas. A saída da ferramenta é então o código transformado.

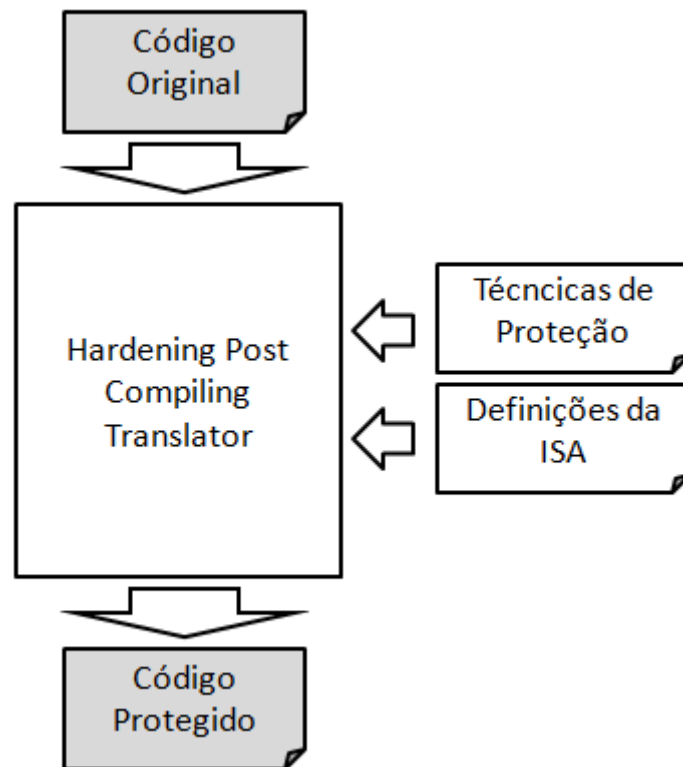


Figura 3.10: Papel do HPCT.

O código da aplicação a ser transformado pelo HPCT deve estar no formato de código de máquina, pois é neste nível que as técnicas propostas na sessão 3.2 são aplicadas. Por esta razão, a aplicação deve ser compilada até esta fase. Desta maneira, a HPCT entra no fluxo de compilação como mostra a figura 3.11, partindo de uma descrição na linguagem C.

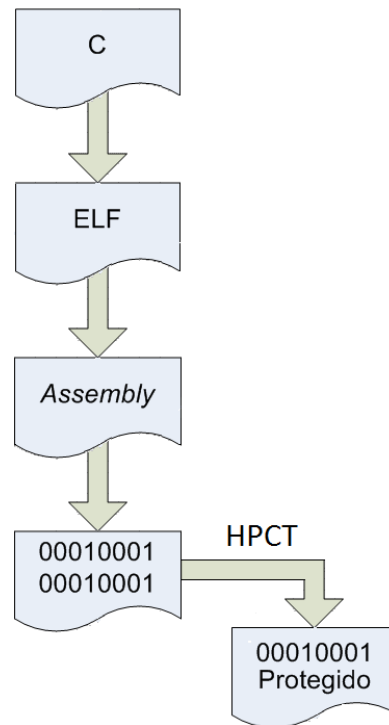


Figura 3.11: Fluxo de compilação HPCT.

3.3.1 Visão geral

Na figura 3.12 é ilustrado o diagrama *Unified Modeling Language* (UML) de classes, contendo a estrutura básica da ferramenta. As principais classes são descritas a seguir.

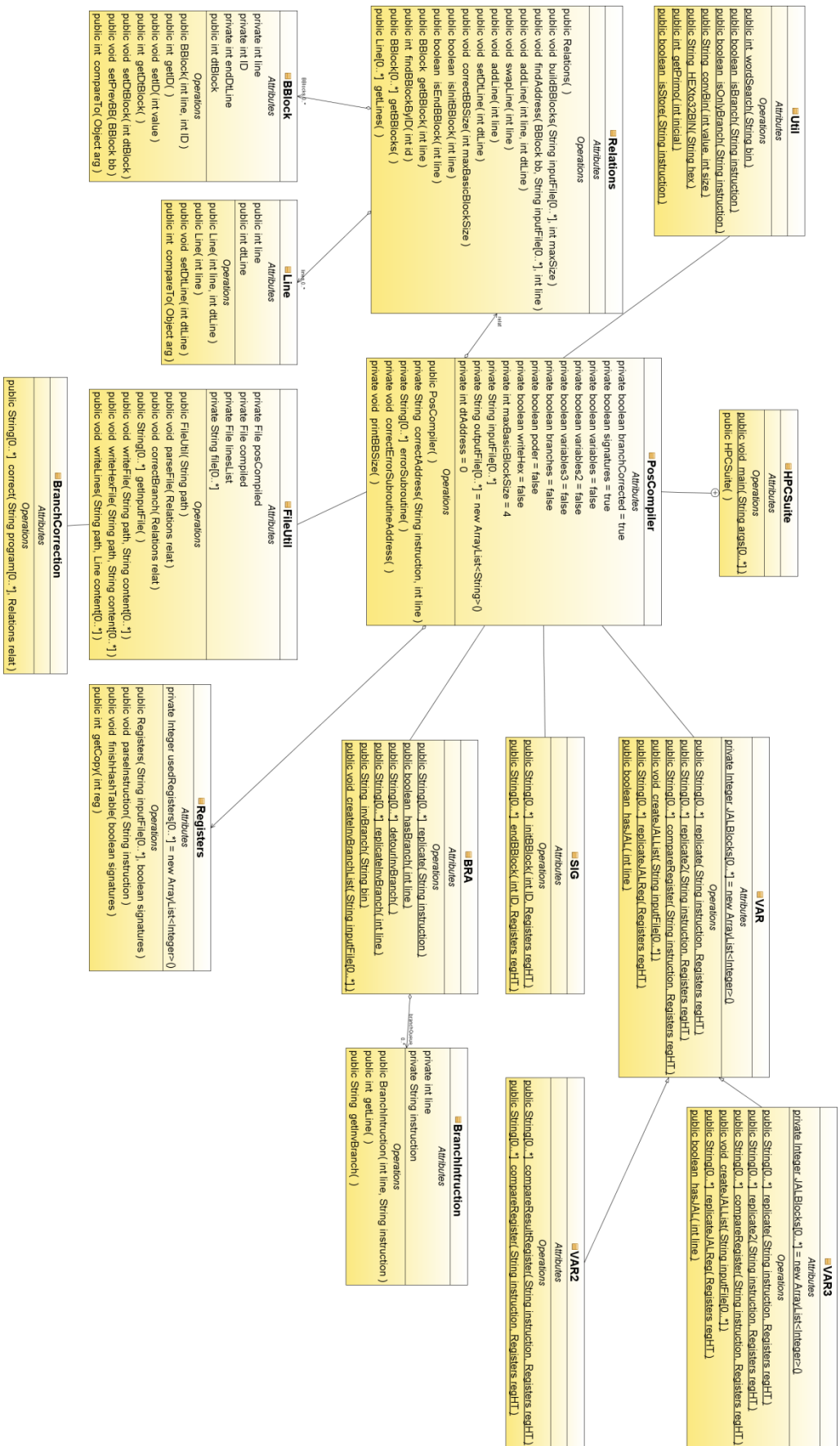


Figura 3.12: Diagrama de classes HPCT.

3.3.1.1 *HPCSuite*

A HPCSuite é a classe principal da ferramenta e sua principal função é agrupar todas as ferramentas num aplicativo único. Entretanto, na versão atual, a única ferramenta presente é o HPCT, uma vez que o Injetor de Falhas, o Mapeador (ambos descritos a seguir) e as demais ferramentas de análise estão implementadas em diferentes aplicativos. Como trabalho futuro espera-se unificar todas as ferramentas através desta classe.

3.3.1.2 *Classe: PosCompiler*

A classe PosCompiler representa o topo da hierarquia da HPCT. Sua principal função, portanto, é interpretar as entradas, ou seja, as definições da ISA e as regras de transformação a serem utilizadas, instanciar os objetos necessários para a aplicação das técnicas desejadas e realizar o controle sobre a transformação do código.

Esta classe implementa métodos para percorrer o código de entrada, analisando sua estrutura de blocos básicos (instruções de início, meio e fim de bloco básico) e verificando onde cada técnica de transformação de código deve ser aplicada. A técnica de Assinaturas, por exemplo, é aplicada somente sobre as instruções de início e fim de bloco básico, enquanto a técnica de Desvios Invertidos modifica somente as instruções de desvio condicional.

Como saída, esta classe gera um arquivo no mesmo formato do arquivo de entrada, porém transformado segundo as regras exigidas pelo usuário.

3.3.1.3 *Classe: BranchCorrection*

Esta classe implementa um método responsável pela transformação descrita no capítulo 3.3.2.4.

3.3.1.4 *Classe: Relations*

Relations é a classe responsável pelo armazenamento de toda a estrutura do código, desde os vetores contendo os blocos básicos do sistema até a relação das linhas das instruções no seu formato original e transformado. Esta classe é também a responsável por criar todas estas estruturas e fornecê-las para os objetos de transformação de código.

Além disso, alguns métodos de busca nos vetores, bem como de escrita no mesmo estão implementados e são acessíveis por outros objetos, a fim de permitir a manipulação destes dados por outros objetos.

3.3.1.5 *Classe: BBlock*

As instâncias dessa classe são armazenadas em um vetor contido na classe *Relations*, onde cada uma contém os dados principais de cada bloco básico, ou seja, linhas inicial e final, identificação, ponteiros para os blocos básicos filhos e a nova linha inicial atualizada, após a transformação do código.

3.3.1.6 *Classe: Line*

Esta classe relaciona a posição original de cada instrução e sua posição após a transformação do código. Através dela é possível fazer um mapeamento entre o código original e o modificado, ao buscar a nova posição de cada instrução no código modificado.

3.3.1.7 Classe: Registers

A classe Registers faz uma análise estática do código, determinando quais são os registradores utilizados e disponíveis para uso pelas técnicas que replicam dados. Nesta classe também é encontrada uma relação entre os registradores e suas réplicas (quando invocado por técnicas que utilizam a duplicação de registradores), também acessíveis pelos demais objetos.

3.3.1.8 Classes: SIG, VAR1, VAR2, VAR3, BRA

Estas classes implementam as funções de transformação descritas na sessão 3.1, sendo que cada uma possui um número variado de métodos, dependendo da complexidade da transformação.

3.3.1.9 Classe: Util

A classe Util agrupa funções úteis utilizadas pelas demais classes do aplicativo. Dentre as funções implementadas estão a conversão de números no formato hexadecimal para binário de 32 bits, conversão de números no formato decimal para o formato binário de tamanho variável e funções que realizam um parsing sobre o arquivo de definições do microprocessador e retornam o tipo de instrução (baseado em seu formato, ou seja, quantos registradores são utilizados, posição deles dentro da instrução, dentre outras características).

3.3.1.10 Classe: FileUtil

Esta classe contém funções utilizadas para facilitar a leitura e escrita dos arquivos de entrada e saída, respectivamente. Funções para abertura de arquivos, passagem dos dados do arquivo para um vetor e escrita de arquivos tornam mais simples o tratamento de arquivos. É também através desta que a classe *BranchCorrection* é invocada.

3.3.2 Funções Implementadas

O HPCT é uma ferramenta que agrupa diversas funções de análise e processamento de código. Esta sessão descreve as principais funções implementadas pelas HPTC e alguns dos algoritmos utilizados em sua implementação.

3.3.2.1 Processamento dos Blocos Básicos

A construção dos blocos básicos é uma importante etapa na transformação do código, uma vez que todas as demais etapas são influenciadas pela estrutura gerada. Durante a construção dos blocos básicos, são armazenados os seguintes dados: identificador do bloco básico, linha de início do bloco básico e linha de fim de bloco básico. É importante lembrar que as instruções de desvio não pertencem a nenhum bloco básico e, por esta razão, é preciso armazenar a linha de fim de bloco básico, uma vez que esta não está diretamente relacionada à linha de início do próximo bloco básico.

Inicialmente, é criado o primeiro bloco básico, com linha de início igual à posição de memória da primeira instrução do código. A partir de dele, o código é percorrido buscando-se instruções de desvio. Sempre que uma instrução de desvio é encontrada, as instruções localizadas na posição de memória de destino e na posição de memória seguinte à da instrução de desvio são analisadas. Para cada instrução analisada, se esta não for outra instrução de desvio, um novo bloco básico é adicionado e sua posição de memória é adicionada como linha de início.

Algumas instruções de desvio têm como destino o valor armazenado num registrador, como é o caso da instrução *Jump to Register* (JR). Neste caso, é preciso localizar alguma instrução que carregue uma variável imediata no registrador para o qual a instrução realiza o desvio. Este procedimento pode ser inviável, dependendo do código gerado pelo compilador (quanto maior for o fluxo de dados entre a leitura de uma constante e a utilização do registrador para o desvio, maior é a complexidade para se encontrar o endereço de desvio). Quando o desvio não é encontrado, as técnicas não são aplicadas na transição entre blocos básicos, diminuindo a confiabilidade do sistema. A utilização de tais instruções, entretanto, é pouco utilizada e pode ser evitada com a devida configuração do compilador.

Após uma primeira varredura do código, todos os blocos básicos estarão criados com suas linhas de início de bloco básico assinaladas. É então feito um novo processamento onde, para cada bloco básico, a linha de início do próximo bloco básico decrementada de 1 é adicionada como linha de fim de bloco. A instrução localizada nesta posição é então analisada e caso esta seja uma instrução de desvio, o processo é realizado novamente (a posição é decrementada de 1 linha e a instrução localizada nesta posição é analisada).

Ao fim deste processamento, todos os blocos básicos estarão criados e com suas variáveis de início e fim de bloco básico corretamente assinaladas.

3.3.2.2 *Replicação da Chamada de Sub-rotina*

A instrução de chamada de sub-rotina, chamada de *Jump and Link* (JAL) no processador miniMIPS manipula indiretamente o registrador de retorno de sub-rotina (RA). Por este motivo, as técnicas que utilizam a replicação de variáveis devem replicar também o RA. Para isto, o HPCT se utiliza da característica de BRANCH CORRECTION, onde a instrução após uma instrução de desvio é sempre executada e adiciona um instrução para manter a consistência entre RA e sua cópia.

3.3.2.3 *Escolha de Registradores para Duplicação*

Esta função é responsável por mapear cada registrador utilizado no programa para um registrador desocupado.

Inicialmente alguns registradores específicos são marcados como utilizados, como é o caso do registrador zero, que não é endereçável. Num segundo momento é realizada uma varredura no código do programa, em busca de registradores endereçados dentro das instruções, sejam eles de leitura ou de escrita. Por fim, é realizada uma varredura em instruções que utilizam registradores implícitos, como é o caso da instrução JAL, que utiliza o registra RA implicitamente, ou seja, sem endereçá-los na instrução.

Uma vez encontrados os registradores utilizados no código, é criada uma função de *hash* ligando cada registrador encontrado aos demais registradores não utilizados.

Ainda durante este processo, são escolhidos alguns registradores não utilizados pelo programa para servirem como variáveis globais das técnicas, como é o caso da variável ECF presente nos grupos do tipo Assinaturas.

Em casos onde o compilador utiliza mais da metade dos registradores disponível no microprocessador, é necessário escolher um conjunto de registradores para ser protegido. As variáveis com mais alta prioridade de alocação são as utilizadas pelas técnicas que dependem de tais variáveis, como no caso do grupo de Assinaturas. Em

seguida, é realizada uma análise do código em busca dos registradores mais utilizados, recebendo estes mais alta prioridade de alocação.

Nestes casos, alguns registradores não serão duplicados e desta maneira a confiabilidade do sistema poderá ser reduzida. Outra opção é mudar as configurações do compilador, exigindo que este gere um código para um conjunto menor de registradores. Isto pode acarretar em perda de desempenho e maior ocupação de memória, mas garante a duplicação de todos os registradores do sistema.

3.3.2.4 Método para Correção de Desvio: *BranchCorrection*

Alguns microprocessadores, como é o caso do miniMIPS, possuem uma arquitetura onde toda a instrução após uma instrução de desvio condicional é executada, sendo o desvio tomado ou não. Esta característica se chama *delayed branch*, ou desvio retardado, e é muitas vezes explorada pelos compiladores, a fim melhorar o desempenho da aplicação.

Entretanto, esta característica deve ser levada em consideração quando as técnicas de tolerância a falhas são aplicadas. No caso da técnica de variáveis, é impossível replicar uma instrução após um desvio condicional, visto que apenas uma instrução será executada caso o desvio não seja tomado, quando na verdade pelo menos 2 instruções deveriam ser executadas (a original e sua réplica). Nestes casos, é necessário corrigir o código, colocando a instrução após o desvio antes do mesmo e adicionando uma instrução do tipo NOP após a instrução de desvio.

A figura 3.13 ilustra esta transformação. O código original, mostrado à esquerda, possui uma instrução de desvio condicional (BNEZ) seguida de uma instrução de transferência entre registradores (MOV). Neste caso, essas instruções devem ser invertidas, como mostram as instruções 3 e 4 à direita. Como a instrução BNEZ obrigatoriamente executará a instrução seguinte, a instrução 5 (NOP) deve ser adicionada. Com esta transformação, a instrução de MOV pode ser protegida pelas técnicas de tolerância a falhas e a instrução NOP mantém a consistência do fluxo de execução, impedindo que a instrução 6 (SRL) seja executada caso o desvio condicional não seja tomado.


<code>or</code>	<code>r0, r1, r0</code>	1: <code>or r0, r1, r0</code>
<code>andi</code>	<code>r0, r0, 0x3</code>	2: <code>andi r0, r0, 0x3</code>
<code>bnez</code>	<code>r0, 0</code> 	3: <code>move r2, r0</code>
<code>move</code>	<code>r2, r0</code>	4: <code>bnez r0, 0</code>
		5: <code>nop</code>
<code>srl</code>	<code>r0, r2, 0x4</code>	6: <code>srl r0, r2, 0x4</code>

Figura 3.13: Transformação *BranchCorrection*.

3.3.2.5 Lista de Desvios Condicionais Invertidos

Antes de iniciar a transformação de Desvios Condicionais, esta função é chamada para realizar uma varredura de todos os saltos existentes no código e, a cada salto, é salva a sua linha respectiva juntamente com o salto com a condição invertida para quando o processamento chegar ao ponto designado esta nova instrução será inserida.

3.4 Ferramenta de Simulação de Falhas

O desenvolvimento de técnicas de tolerância a falhas exige uma etapa de testes que é composta pela programação das aplicações devidamente modificadas num microprocessador e uma campanha de injeção de falhas. Idealmente, irradia-se o sistema com partículas energizadas, tais como X e Y. Entretanto, este é um processo que requer instalações apropriadas, a utilização de uma placa de teste personalizada e equipamentos para controle e extração de dados do projeto sob teste, o que resulta num alto custo. Para contornar este problema, foi utilizado um *software* de simulação chamado ModelSim (MODELSIM, 1999) e desenvolvido um programa para a geração e injeção de falhas.

O *software* ModelSim, da fabricante *Mentor Graphics*, é um aplicativo de simulação e *debug* de códigos escritos em *Hardware Description Language* (HDL), mais especificamente, são aceitas as linguagens VHDL, Verilog e SystemC. Ele disponibiliza tanto um ambiente gráfico para simulação (suporta a entrada de dados através de comandos e a saída de resultados através da interface) quanto um console para a escrita de *scripts* na linguagem *Tool Command Language* (TCL) e saída em formato de texto. Além destas facilidades, a característica que mais interessa no escopo deste trabalho é a possibilidade de acessar, ler e escrever qualquer sinal lógico existente no sistema simulado em qualquer tempo desejado. Através destes comandos, é possível injetar falhas com precisão superior a nanosegundos e ter controle total sobre o tempo de duração e a extração dos resultados. Apesar de todas estas facilidades, o ModelSim não fornece um ambiente de geração e injeção de falhas nem coleta de resultados.

Para isto, foi desenvolvido um *software* chamado Injetor de Falhas. Esta ferramenta foi desenvolvida na linguagem Java, visando a união com a HPCT para a formação de uma ferramenta única para a proteção e teste de sistemas microprocessados.

O Injetor de Falhas tem como objetivo principal a geração de *scripts* no formato TCL para serem executados no ModelSim, assim contornando a sua ausência de um ambiente de testes. Desta maneira, a simulação é realizada a nível de *Register Transfer Level* (RTL), ou seja, num nível onde o circuito é descrito em termos do fluxo dos dados, não estando os módulos do sistema fixos a componentes em específico. Os módulos presentes no nível RTL ainda serão transformados e otimizados durante a síntese lógica e assinalados a componentes reais, no caso, uma biblioteca de células no fluxo *standard cell* para *Application Specific Integrated Circuits* (ASICs) ou *Look-Up Tables* (LUTs) para o fluxo de projeto para *Field Programmable Gate Array* (FPGA). Entretanto, a injeção de falhas a nível RTL oferece uma boa estimativa de cobertura de falhas.

O *software* desenvolvido tem como entradas três arquivos: (1) um arquivo de definições das falhas a serem geradas, contendo a duração de cada falha e a quantidade de falhas desejadas; (2) um arquivo com definições do microprocessador, contendo a frequência de relógio do sistema, o caminho completo de todos os sinais a serem perturbados por falhas, informações sobre a memória de dados e sobre mecanismos de detecção e (3) um arquivo de definições da aplicação, contendo o tempo total de execução do programa, os valores corretos de saída e a posição de memória onde os valores serão armazenados. Como saída, o injetor de falhas cria um script TCL contendo todas as falhas geradas, pronto para ser executado dentro do ambiente de simulação do ModelSim. A figura 3.14 mostra o papel do Injetor, enquanto o Anexo A

apresenta um arquivo de configuração aceito pelo Injetor que contém todas as informações necessárias em um mesmo arquivo, além de algumas informações próprias do Injetor, como opções para simulação do projeto de forma transparente e envio de resultados por correio eletrônico.

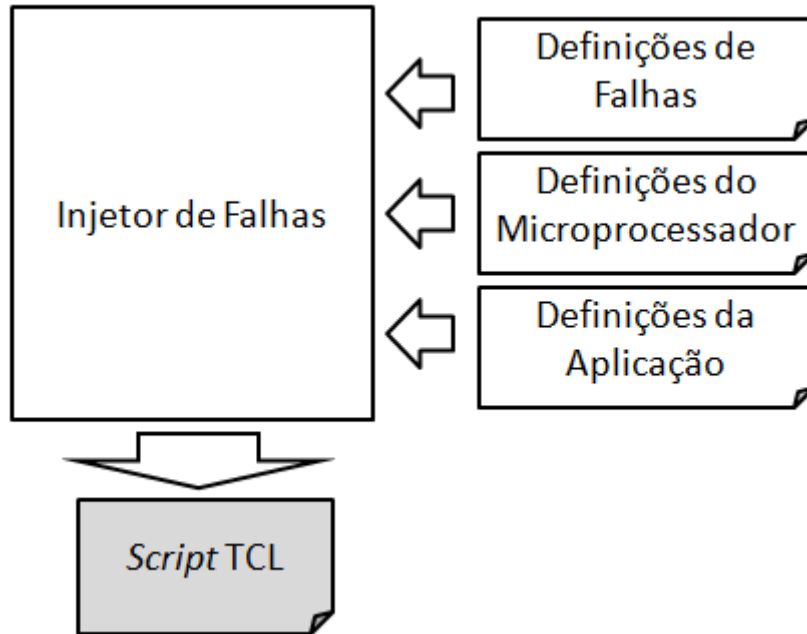


Figura 3.14: Papel do Injetor de Falhas.

O Injetor de Falhas é dividido em três etapas distintas. São elas: (1) geração de um conjunto de falhas, (2) simulação do conjunto de falhas gerado sobre um sistema simulado no ModelSim e (3) coleta dos resultados gerados pelo ModelSim. Essas etapas, bem como o formato dos scripts TCL gerados serão descritos a seguir.

3.4.1 Geração de Conjunto de Falhas

A geração do conjunto de falhas depende bastante das definições contidas nos três arquivos de entrada do Injetor de Falhas. Do arquivo de definições das falhas são retirados os dados de quantas falhas serão geradas. Do arquivo de definições do microprocessador são utilizados os dados de frequência de relógio e o caminho completo dos sinais (contendo a estrutura hierárquica dentro do sistema) e, por fim, é extraído o tempo total da aplicação do arquivo de definições da aplicação. Com estes dados de entrada, o Injetor de Falhas é capaz de gerar uma falha sobre o caminho completo de um sinal num dado período de tempo, sendo este dentro do tempo de execução da aplicação.

Baseado nestes dados de entrada, o Injetor de Falhas gera dois vetores. O primeiro, mostrado na figura 3.15 possui o caminho completo de cada sinal e sua geração acontece aleatoriamente, escolhendo o caminho completo de um sinal pertencente ao arquivo de definição do microprocessador. O segundo vetor, que pode ser visto na figura 3.16, contém o tempo de injeção da falha e também sendo gerado aleatoriamente, escolhe um valor múltiplo do período de relógio variando entre o valor zero e o tempo total da aplicação.

Quando combinados pelo índice, os dois vetores apresentam um sinal e um tempo aleatoriamente escolhidos. São utilizados posteriormente durante a etapa de injeção de falhas, descrita a seguir.

```

1 # Sinais para Injeção
2 array set signal {
3   1 "/sim_minimips/u_minimips/u1_pf/pc_interne(11) "
4   2 "/sim_minimips/u_minimips/u2_ei/ei_adr(17) "
5   3 "/sim_minimips/u_minimips/u2_ei/ei_instr(11) "
6   4 "/sim_minimips/u_minimips/u3_di/di_adr "
7   5 "/sim_minimips/u_minimips/u3_di/di_adr_reg_dest(0) "
8   6 "/sim_minimips/u_minimips/u3_di/di_bra "
9   7 "/sim_minimips/u_minimips/u3_di/di_offset(13) "
10  8 "/sim_minimips/u_minimips/u3_di/di_op_mem "
11  9 "/sim_minimips/u_minimips/u4_ex/ex_adr_reg_dest(0) "
12}

```

Figura 3.15: *Script* com o caminho completo da lista de falhas.

```

1 # Sinais para Injeção
2 array set time {
3   1 971828
4   2 347372
5   3 1145540
6   4 1367930
7   5 1430510
8   6 583328
9   7 704960
10  8 1100180
11  9 926468
12}

```

Figura 3.16: *Script* com o tempo de injeção da lista de falhas.

A descrição dos vetores apresentados nas figuras 3.15 e 3.16 estão no formato TCL, aceito pelo simulador ModelSim. Em resumo, há uma lista de sinais onde as falhas serão inseridas (falhas do tipo SEU e SET) e o tempo em termos de ciclo de relógio de injeção das falhas. A figura 3.17 mostra a injeção de uma falha no sinal `adr_reg1`, no bit 3 e no tempo `Tempo de Injeção`, com duração de `Duração da Falha`, durante a execução de uma aplicação no microprocessador miniMIPS.

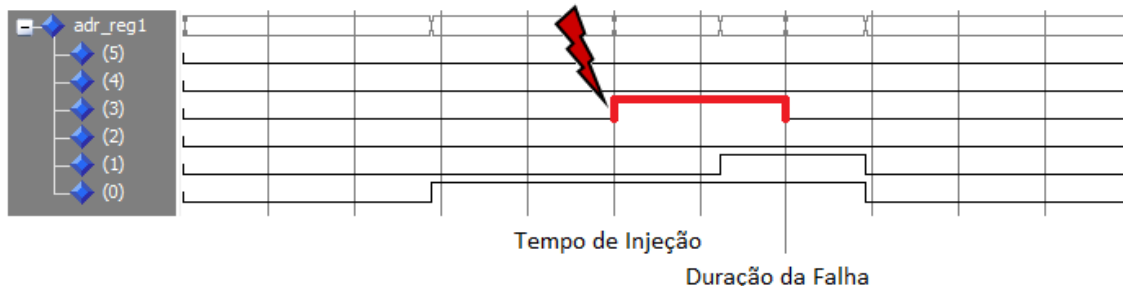


Figura 3.17: Exemplo de Injeção de falha.

3.4.2 Simulação de Conjunto de Falhas

A etapa de injeção de falhas é responsável por combinar os vetores gerados durante a etapa de geração de falhas e utilizá-los para perturbar os sinais durante a simulação realizada pelo ModelSim. O Injetor de Falhas também precisa escolher o período pelo qual o sinal será perturbado, ou seja, a duração da falha (dado extraído do arquivo de definições das falhas) e extrair o tempo total da aplicação (extraído do arquivo de

definições do microprocessador), a fim saber quanto tempo é necessário para a executar a simulação.

Com base nestes dados, o injetor de falhas gera um trecho de código, mostrado na figura 3.18, que inicializa a simulação, executa a simulação até o tempo de falha, inverte o valor lógico do sinal escolhido durante o tempo desejado e roda o sistema até o tempo total de execução da aplicação. Para isto são necessário quatro comandos na linguagem TCL. O *script* descrito a seguir utiliza os dois vetores gerados durante a etapa de geração de falhas, a seguir chamados de $\$signal(\$i)$, para o vetor com o caminho dos sinais e $\$time(\$i)$, para o vetor com os tempos de injeção de falha.

```
1 # Injeção de Falha
2 restart -f
3 run "$time($i) ns"
4 if { [ examine $signal($i) ] == 1 }
5   { force -freeze $signal($i) 0 -cancel $faultTime ns; set fValue 0 }
6 else
7   { force -freeze $signal($i) 1 -cancel $faultTime ns; set fValue 1 }
8 set timeLeft [ expr $runtime - $time($i) ]
9 run "$timeLeft ns"
```

Figura 3.18: *Script* de simulação de falhas.

O comando *restart* com o parâmetro *-f* inicializa o sistema, zerando os registradores e a memória do microprocessador. O parâmetro *-f* serve para que não apareça uma confirmação da inicialização, tornando-a automática.

Seguido deste comando, aparece o comando *run*, que serve para que o simulador simule o sistema até o tempo $\$time(\$i)$, que é um valor extraído do vetor de tempo gerado durante a etapa de geração de falhas, ou seja, simula o sistema até o momento onde a falha será inserida.

Após o *run*, é utilizado o comando *examine \$signal(\$i)*, capaz de retornar o valor do sinal $\$signal(\$i)$, encontrado no vetor de sinais. Com ele, é possível analisar o valor lógico do sinal e inverter o mesmo. Para isto é utilizado o comando *force*.

O comando *force*, seguido dos parâmetros *freeze* e *cancel* é capaz de congelar o sinal $\$signal(\$i)$ analisado no valor lógico '0' ou '1' (dependendo do resultado do comando *examine*) durante o tempo $\$faultTime$. O tempo *faultTime* define a duração de cada falha.

Finalmente, o comando *run* utilizado novamente executa a simulação até o tempo restante, ou até o tempo total da execução, obtido através da subtração entre o tempo total de execução da aplicação ($\$runtime$) e o tempo de injeção da falha ($\$time(\$i)$).

3.4.3 Coleta de Resultados

A última etapa, chamada de coleta de resultados, é responsável por armazenar valores úteis na análise dos resultados em arquivos de registro. Estes resultados podem variar de acordo com o tipo de análise desejada. Entretanto, qualquer valor utilizado na simulação ou gerado pela mesma pode ser registrado. Dentre eles, os que merecem maior destaque são a quantidade de falhas injetadas, tempo total de execução, o resultado individual de cada falha, o resultado gravado em memória pelo microprocessador (saídas do sistema) e até mesmo a lista de transição do PC.

A criação do *script* de coleta de resultados não requer nenhum dado de entrada, embora seja interessante utilizar os valores de saída esperados (obtidos pela execução do sistema sem falhas esperados na saída) e encontrados no arquivo de definição da aplicação. Com estes dados, o próprio *script* é capaz de informar se a falha afetou o sistema, modificando os resultados gravados em memória, comparando os resultados salvos pela simulação e os desejados.

A figura 3.19 mostra um trecho gerado pelo injetor de falhas para a coleta de alguns dados. A primeira construção *if/else* compara os valores das variáveis *\$goldRes* e *\$actRes*. O primeiro valor representa os valores esperados, obtidos após a execução do sistema sem falhas, enquanto o segundo representa os valores gerados pela simulação e gravados em memória. Se os valores forem iguais, é inserida uma linha no arquivo de registro informando o índice da falha (*\$i*) e os dizeres "*Correct answer*", ou seja, resposta correta. Caso contrário, é inserida uma linha informando que os valores diferem, com os dizeres "*Wrong answer*".

```

1 # Coleta de Resultados
2 if { $goldRes == $actRes }
3   { set data "$i - Correct answer -"; set correctResult 1; }
4 else
5   { set data "$i - Wrong answer -"; set varWrong 1; }
6
7 # Escrita de Lista
8 add list -hex {sim:/sim_minimips/u_minimips/u1_pf/pf_pc }
9 write list -window "$window lista_PC{$i}.lst"
10 noview List

```

Figura 3.19: *Script* de coleta de resultados.

O comando *add list* gera um arquivo interno ao ModelSim no formato hexadecimal (devido ao parâmetro *-hex*) com todas as transições do sinal *sim_minimips/u_minimips/u1_pf/pf_pc*, ou seja, do registrador PC. Esta lista é então gravada no arquivo *lista_PC{\$i}.lst* através do comando *write list*, onde *\$i* é o índice da falha no vetor. O trecho de código é finalizado com o comando *noview*, que finaliza a lista, permitindo que outra seja gravada após a próxima inicialização da simulação.

Da mesma maneira que os trechos de código apresentados nas figuras anteriores, este trecho de código está descrito na linguagem TCL.

4 RESULTADOS DE SUSCEPTIBILIDADE A SINGLE EVENT UPSETS (SEU AND SET)

A grande maioria dos trabalhos encontrados na literatura apresentam resultados sobre o total de falhas injetadas, incluindo a quantidade de falhas injetadas no circuito que não afetaram os resultados do mesmo, ou seja, falhas mascaradas que não se tornaram erros. Isto acontece pelo fato de que estes trabalhos buscam provar que as técnicas de tolerância a falhas apresentadas por eles oferecem um dado grau de confiabilidade, não levando em consideração quais partes ficam mais suscetíveis a erros dentro do circuito.

Este trabalho visa comparar diferentes técnicas e o ideal, portanto, é formar um grupo de falhas e injetá-las no mesmo ponto, tanto físico como temporal, em todos os programas protegidos por diferentes técnicas. Por esta razão, falhas que não resultam em erros no sistema são desprezíveis. Da mesma maneira, este grupo de falhas deve ser normalizado de modo que possibilite a comparação e análise de cada técnica de proteção de falhas.

Este capítulo descreve todo o processo de injeção de falhas, desde a escolha das aplicações, passando pela criação e classificação do conjunto de falhas, pelas técnicas de tolerância a falhas escolhidas, pelo mapeamento das falhas originais para as diferentes implementações e, finalmente, com os resultados obtidos após a injeção das falhas.

4.1 Aplicações para Estudos de Caso

A escolha das aplicações é uma etapa importante, pois tem influência direta sobre os resultados. Idealmente, seria interessante utilizar um *benchmark* completo de aplicações. Entretanto, a campanha de falhas descrita neste capítulo envolve diversas etapas de processamento de falhas, algumas delas ainda realizadas manualmente, o que torna a simulação de diversas aplicações muito demorada. Por esta razão, foram escolhidas duas aplicações: um algoritmo de multiplicação de matrizes e um algoritmo de ordenação chamado *bubble sort*. Foram escolhidas estas duas aplicações pois a multiplicação de matrizes envolve um grande processamento da parte de dados do microprocessador, enquanto o algoritmo de classificação utiliza muito a parte de controle, ou seja, é um código com muitos desvios e poucas operações aritméticas e lógicas.

Ambas as aplicações foram compiladas através do compilador GCC versão 3.3.1. O processo de compilação para o miniMIPS envolve as seguintes etapas:

- Compilação do código C para arquivo do tipo *Executable and Linking Format* (ELF), ou Formato Executável e de Ligação;
- Tradução do arquivo ELF para arquivo do tipo *assembly*;
- Compilação e otimização do arquivo *assembly* para arquivo do tipo *coe*.

O resultado da compilação foram dois arquivos com a extensão *coe* (um para cada aplicação), composto por 0's e 1's, sendo cada instrução *assembly* representada por uma linha de texto de 32 números. A seguir as aplicações e os códigos envolvidos serão descritos. A figura 4.1 mostra um pequeno trecho de código no formato *coe*, mostrando cinco instruções.

```

1 001111000001111010000000001010000
2 00100011101111011111110000000000
3 00111100000111000000000000000001
4 000011000000000000000000001000000
5 00100111100111001000010001110000

```

Figura 4.1: Código compilado no formato *coe*.

4.1.1 Multiplicação de Matrizes

A operação de multiplicação de matrizes é muito comum em aplicações que envolvem cálculo vetorial, como aplicações de processamento de imagem, filtros, dentre outras. Ainda que muito utilizada, é uma operação complexa que envolve um grande número de acessos a memória, controles de laço e operações de multiplicação e soma. Por isto, é difícil de ser tratada por microprocessadores, devido ao grande tempo computacional envolvido (uma simples multiplicação e acumulação pode chegar a envolver mais de 6 instruções, sem contar o controle de laço). Alguns processadores tem inclusive módulo específicos para executar a instrução *Multiply and Accumulate* (MAC), capaz de multiplicar dois valores e somar o resultado a um terceiro, atualizando este na memória.

A alta quantidade de operações envolvida nesta aplicação e o baixo controle sobre ela (o controle de laço é fixo e, portanto, existem poucas instruções de desvio condicional, além do fato de que esta aplicação possui um tempo fixo de execução, independente dos valores de entrada) a tornam uma aplicação do tipo *datapath*, ou seja, que exige bastante dos módulos de cálculo do microprocessador (ULA) e pouco dos módulos de controle (módulo preditor de desvios, lógica de próxima instrução, dentre outros). Outra característica interessante é que esta aplicação utiliza um número relativamente grande de registradores, visto que utiliza, além de três variáveis de controle (i , j e v), registradores para realizar a operação MAC separada em instruções de multiplicação, soma e de acesso à memória.

A figura 4.2 ilustra o código escrito na linguagem C. As linhas de 2 a 23 realizam a inicialização das variáveis de entrada (a e b), de saída (c) e de controle (i , j e v). As linhas de 25 a 28 percorrem as variáveis de entrada, multiplicando e somando o resultado na variável de saída c . Ao final da execução, o resultado estará armazenado nas posições de memória alocadas para a variável de saída c .

```

1 int main()
2 {
3     int a[6][6] = {{1,1,1,1,1,1},
4                   {2,2,2,2,2,2},
5                   {3,3,3,3,3,3},
6                   {4,4,4,4,4,4},
7                   {5,5,5,5,5,5},
8                   {6,6,6,6,6,6}};
9
10    int b[6][6] = {{1,1,1,1,1,1},
11                 {2,2,2,2,2,2},
12                 {3,3,3,3,3,3},
13                 {4,4,4,4,4,4},
14                 {5,5,5,5,5,5},
15                 {6,6,6,6,6,6}};
16
17    int c[6][6] = {{0,0,0,0,0,0},
18                 {0,0,0,0,0,0},
19                 {0,0,0,0,0,0},
20                 {0,0,0,0,0,0},
21                 {0,0,0,0,0,0},
22                 {0,0,0,0,0,0}};
23    int i, j, v;
24
25    for (i=0; i<6; i++)
26        for (j=0; j<6; j++)
27            for (v=0; v<6; v++)
28                c[i][j] = c[i][j] + a[i][v]*b[v][j];
29 }

```

Figura 4.2: Código C da aplicação multiplicação de matrizes.

4.1.2 Ordenação *Bubble Sort*

Operações de ordenação de dados são muito utilizadas na indústria e na literatura, principalmente em aplicações de armazenamento de dados, como bancos de dados e *data warehouses*. Qualquer *software* que lide com estruturas de dados (listas, filas, pilhas, etc..) implementa funções de ordenação de dados. Dentre elas, encontra-se alguns algoritmos de ordenação, como é o caso do *bubble sort*.

O algoritmo *bubble sort* percorre a lista a ser classificada repetidamente, comparando valores adjacentes e trocando-os de lugar quando necessário (este parâmetro varia de acordo com a forma de ordenação, ou seja, ascendente ou decrescente). O desempenho médio deste algoritmo é de $O(n^2)$, onde n é o tamanho do vetor, ou seja, é um algoritmo quadrático com duração de execução variável, de acordo com a quantidade e a ordem das entradas.

Esta aplicação de ordenação possui poucas instruções aritméticas, sendo todas concentradas no controle de laço de percorrimento do vetor de entrada, ou seja, nas instruções "*i++*" e "*j++*", encontradas nas linhas 7 e 8, respectivamente, segundo a figura 4.3. Todas as demais operações são instruções de desvio condicional e

transferência entre registradores. Esta característica torna a ordenação *bubble sort* uma aplicação do tipo *controlpath*, pois exige um grande processamento da parte de controle do microprocessador, como o controle de desvios condicionais (incluindo a previsão de desvio), lógica de leitura da próxima instrução, chamadas de sub-rotinas, dentre outros, enquanto a parte aritmética (ULA) é utilizada somente para incrementar os controles de laços.

```

1 int main()
2 {
3     int v[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
4     int i, j, aux;
5     int k = 9;
6
7     for (i=0; i<10; i++) {
8         for (j=0; j<k; j++) {
9             if (v[j] > v[j+1]) {
10                aux = v[j];
11                v[j] = v[j+1];
12                v[j+1] = aux;
13            }
14        }
15    }
16 }

```

Figura 4.3: Código C da aplicação ordenação *bubble sort*.

As linhas de 3 a 5 realizam a inicialização da variável de entrada e saída (*v*), visto que o algoritmo modifica a entrada do sistema e as variáveis temporárias (*i*, *j*, *aux* e *k*). As linhas 7 e 8 percorrem o vetor, enquanto a linha 9 verifica se existem valores que devem ser trocados de posição. Finalmente, as linhas 10 a 12 trocam de posição dois valores, com a ajuda da variável auxiliar *aux*. O resultado final da operação ficará armazenado sobre a variável *v*.

4.2 Construção do Conjunto de Falhas

A construção do conjunto de falhas, para cada aplicação, deve atender alguns requisitos para que seja possível realizar uma análise completa dos resultados coletados. São eles:

- Conter falhas para todos os sinais utilizados pelo microprocessador durante a execução da aplicação;
- Gerar tanto falhas do tipo SEU quanto do tipo SET;
- Normalizar os subgrupos de falhas;
- Ser composta de no mínimo 3 e máximo de 5 falhas por sinal.

A primeira característica se refere ao fato de que nem todos os módulos do microprocessador são utilizados durante a execução de uma aplicação. Como exemplo, temos que a aplicação de ordenação *bubble sort* não realiza nenhuma multiplicação. Por esta razão, o registrador localizado dentro da ULA e utilizado para armazenar o

resultado de uma multiplicação nunca é endereçado e, portanto, uma falha afetando-o jamais comprometerá o resultado do sistema. Entretanto, todos os demais sinais utilizados durante a aplicação devem ser atingidos com falhas, uma vez que todos tem o potencial de causar um erro na execução do programa.

Diferentemente da maioria dos trabalho encontrados na literatura, este trabalho visa analisar falhas do tipo SEU e SET. É preciso, portanto, que o conjunto de falhas seja composto pelos dois tipos de falhas.

O processador miniMIPS possui um total de X sinais sensíveis a falhas, dos quais 1024 pertencem ao banco de registradores, ou seja, quase 50% dos sinais do microprocessador estão relacionados com os registradores do banco de registradores. Uma simples injeção aleatória de falhas seria tendenciosa, visto que probabilisticamente, 50% das falhas seriam injetadas nos registradores e, desta maneira, técnicas visando somente a proteção dos registradores teriam resultados muito superiores às técnicas de proteção ao restante do circuito. Assim, o conjunto de falhas deve seguir a característica de normalização de falhas para cada subgrupo.

Idealmente, seria necessário realizar uma campanha de falhas exaustiva, de modo que cada sinal possua uma falha para cada ciclo de relógio. Esta análise, entretanto, requer um tempo de processamento muito grande, tornando-a inviável. Para isto, foi escolhido um número mínimo de 3 e máximo de 5 falhas por sinal. Desta maneira, todos os sinais receberão uma quantidade parecida de falhas e suficiente para a análise.

A seguir serão descritas as etapas realizadas para a criação dos grupos de falhas. Todas as etapas foram realizadas para ambas as aplicações descritas na sessão 4.1.

4.2.1 Injeção Preliminar de Falhas

Num primeiro momento, a aplicação foi executada sem nenhuma interferência e seus resultados armazenados. A seguir, foram extraídas as características do miniMIPS (frequência de relógio e caminho completo de todos os sinais) e da aplicação (tempo total de execução do programa e os valores corretos de saída), ainda durante a primeira execução, a fim de criar os arquivos necessários para a execução do Injetor de Falhas, descrito no capítulo 3.4. Por fim, foi criado o arquivo de definições das falhas para o Injetor de Falhas, de modo a gerar 50 mil falhas sobre todos os sinais do miniMIPS.

Como saída, o Injetor de Falhas criou um *script* na linguagem TCL contendo as 50 mil falhas desejadas. Este *script* foi então executado pelo *software* ModelSim para simular a execução dos programas sendo afetados pelas falhas. Como resultado, a simulação gerou um arquivo identificando o resultado do sistema, quando comparado com os resultados esperados. Após a análise dos resultados, foram excluídas as falhas que apresentaram resultado final correto e descartadas as falhas que excediam a quantidade de 5 falhas por sinal.

Após a primeira injeção de falhas alguns sinais ficaram com número inferior a 3 falhas, tanto para a aplicação de multiplicação de matrizes quanto para a ordenação *bubble sort*. Por esta razão, a lista de sinais contida nas definições do microprocessador foi modificada, excluindo todos os sinais com 5 falhas e novas falhas foram geradas pelo Injetor de Falhas. Estas novas falhas foram então injetadas no sistema. Este processo foi repetido até que todas os sinais tivessem pelo menos 3 falhas.

Durante esta etapa viu-se que alguns sinais não afetavam o resultado do sistema. Nestes casos, cada sinal foi analisado e se de fato não eram utilizados pelo

microprocessador, foram retirados da injeção de falhas. Como exemplo, tem-se o registrador de saída do multiplicador encontrado na ULA. Uma vez que a ordenação *bubble sort* não utiliza este módulo, é impossível afetar o sistema perturbando este sinal.

4.2.2 Classificação de Falhas

A segunda etapa na construção do conjunto de falhas é a classificação das falhas. A fim de realizar uma análise detalhada sobre os resultados obtidos por cada técnica de tolerância a falhas, é necessário classificá-las. Desta maneira, torna-se possível identificar locais vulneráveis no microprocessador e mesmo quais os efeitos de cada falha dentro dele, com o objetivo de combinar diferentes técnicas e assim obter um alto grau de confiabilidade.

Inicialmente, as falhas foram divididas com relação ao local de injeção de falha. Para isto, foram utilizados 4 grupos: *controlpath* e *datapath*, sendo o segundo subdividido em ULA, banco de registradores e outros sinais. Estes grupos estão descritos nos capítulos 3.1.2, 3.1.1, 3.1.1.1 e 3.1.1.2, respectivamente. A divisão com relação ao local de injeção de falhas é interessante, principalmente, por dividir o microprocessador em suas principais áreas físicas e, desta maneira, escolher técnicas adequadas para proteger os locais mais suscetíveis a efeitos dos tipos SEU e SET. Ainda nesta divisão, é possível dividir o microprocessador em área de dados, composta pelo grupo *datapath* e seus subgrupos ULA e banco de registradores e numa área de controle, composta pelo grupo *controlpath*. Esta divisão é interessante, visto que a maioria das técnicas de tolerância a falhas são voltadas a proteção somente da parte de dados ou somente da parte de controle.

A análise de técnicas voltadas ou para dados ou para controle, entretanto, não é completa apenas analisando-se o local de injeção da falha, uma vez que uma falha injetada no banco de registradores pode afetar a tomada de um desvio condicional, assim interferindo no controle do programa. Por este motivo, foi escolhida uma segunda classificação das falhas, relacionando-a com seu efeito.

O efeito de uma falha foi classificado como efeito de dados, quando esta interferiu somente nos resultados da aplicação, ou seja, o programa executou corretamente, mas o resultado final estava errado, ou como efeito de controle, quando a falha interferiu na ordem de execução das instruções, devido a desvios incorretamente tomados, instruções não executadas ou instruções executadas fora de ordem. Para realizar esta classificação, foi extraída da aplicação original, sem falhas, uma lista das instruções executadas, fazendo com que cada falha tenha sido injetada sobre o microprocessador rodando a aplicação original.

Cada lista gerada pela injeção de uma falha foi então comparada com a lista de instruções do programa original. Se a ordem de execução das instruções era idêntica, a falha foi classificada como "falha com efeito sobre os dados" e, em caso contrário, foi classificada como "falha com efeito de controle".

Por fim, foi criada outra classificação com relação ao tipo de falha injetada. Falha sobre registradores, com duração e mais de um ciclo foram consideradas falhas do tipo SEU, enquanto falhas sobre a lógica combinacional foram consideradas do tipo SET.

4.2.3 Grupos de Falhas

Após a injeção preliminar e a classificação das falhas com relação ao local de injeção e efeito no sistema, foram obtidos os grupos de falhas descritos a seguir.

4.2.3.1 Multiplicação de Matrizes

A tabela 4.1 mostra a quantidade de sinais atingidos pelas falhas, somando um total de 81 sinais e a quantidade de falhas por sinal, totalizando 377 falhas, classificadas com relação ao local de injeção da falha e de seu efeitos no sistema.

Tabela 4.1: Grupo de falhas para aplicação de multiplicação de matrizes

Tipo de Falha	Local de Injeção		Quantidade de Sinais	Efeito de Dados	Efeito de Controle
SET	<i>Controlpath</i>		26	83	33
	<i>Datapath</i>	ULA	8	27	10
		Banco	2	9	1
		Outros	9	42	2
	Total		45	161	46
SEU	<i>Controlpath</i>		22	67	36
	<i>Datapath</i>	ULA	1	4	-
		Banco	8	25	13
		Outros	5	18	7
	Total		36	114	56

4.2.3.2 Ordenação Bubble sort

A tabela 4.2 mostra a quantidade de sinais atingidos pelas falhas, somando um total de 76 sinais e a quantidade de falhas por sinal, totalizando 344 falhas, classificadas com relação a seus efeitos no sistema. Como pode ser visto, o algoritmo de *bubble sort* possui menos sinais do que o de multiplicação de matrizes. Isto acontece pelo fato de que a multiplicação de matrizes utiliza mais estruturas do miniMIPS do que a ordenação *bubble sort*.

Tabela 4.2: Grupo de falhas para aplicação de ordenação *bubble sort*

Tipo de Falha	Local de Injeção		Quantidade de Sinais	Efeito de Dados	Efeito de Controle
SET	<i>Controlpath</i>		24	22	89
	<i>Datapath</i>	ULA	5	7	14
		Banco	2	3	4
		Outros	9	14	28
	Total		40	46	135
SEU	<i>Controlpath</i>		22	24	81
	<i>Datapath</i>	ULA	0	-	-
		Banco	8	2	33
		Outros	5	4	19
	Total		36	30	133

4.3 Técnicas Implementadas

Através da aplicação HPCT, descrita no capítulo 3.3, foram gerados programas protegidos sobre as aplicações de multiplicação de matrizes e ordenação *bubble sort*. As técnicas implementadas foram: Assinaturas (SIG, capítulo 3.2.1), Divisão de Blocos Básicos (BBD, capítulo 3.2.2), Variáveis 1 (VAR1, capítulo 3.2.3), Variáveis 2 (VAR2, capítulo 3.2.4), Variáveis 3 (VAR3, capítulo 3.2.5), Desvios condicionais (BRA, capítulo 3.2.6) e a combinação das técnicas de Assinaturas, Variáveis 1 e Desvios Condicionais, chamada de SIG_VAR_BRA.

A tabela 4.3 ilustra o tempo total de execução (em milissegundos), ocupação total da memória de dados (em bytes), o que corresponde à ocupação de registradores mais o espaço reservado na memória de dados, e a ocupação da memória de programa (em bytes) de cada técnica implementada sobre a aplicação da multiplicação de matrizes. Como era esperado, as técnicas de replicação de variáveis são as que mais prejudicam o sistema, com os maiores tempos de execução e ocupação de memória de dados e de controle.

Tabela 4.3: Características físicas da multiplicação de matrizes

	Original	SIG	BBD	VAR1	VAR2	VAR3	BRA	SIG_VAR_BRA
Tempo de Execução (ms)	1.24	1.40	4.12	2.55	2.74	1.64	1.30	2.71
Memória de Programa (bytes)	1548	3500	6228	4340	3672	2964	2580	6012
Memória de Dados (bytes)	524	532	532	1048	1048	1048	524	1056

A figura 4.4 ilustra a relação de tempo de execução e ocupação de memória de dados e de programa com relação ao programa original da aplicação de multiplicação de matrizes.

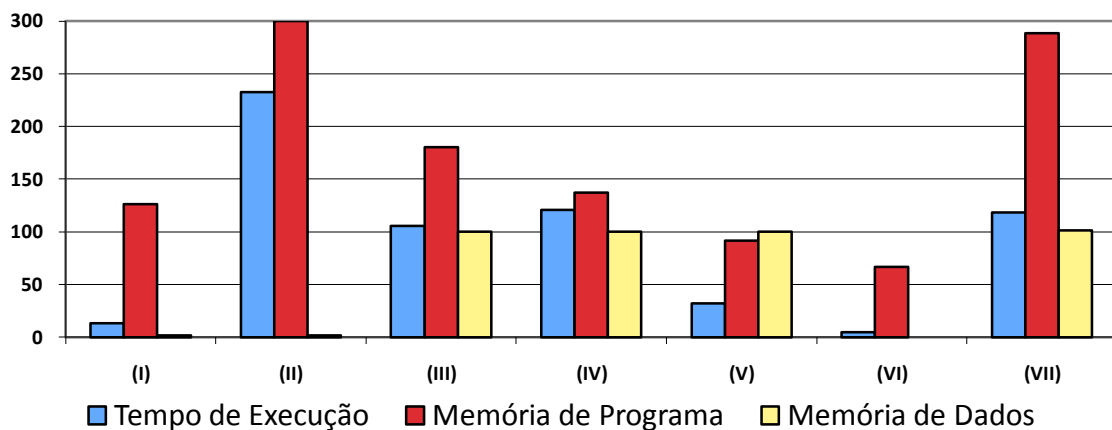


Figura 4.4: Relação de desempenho da multiplicação de matrizes.

Inicialmente, percebe-se que as técnicas de proteção aos dados VAR1, VAR2 e VAR3 apresentam um aumento de 100% na ocupação da memória de dados. Isso se deve à duplicação de todos os registradores e da memória de dados. As técnicas de SIG e BBD apresentam um pequeno aumento, devido a utilização de dois registradores a mais para o controle das assinaturas dos blocos básicos. A técnica de BRA, por outro lado, não apresenta mudanças na ocupação da memória de dados, visto que não necessita de nenhum registrador de controle. A técnica SIG_VAR_BRA demonstra um aumento levemente superior a 100%, devido às técnicas VAR1 e SIG.

A memória de programa apresenta alguns resultados interessantes, uma vez que não acompanha o gráfico de tempo de execução, como, por exemplo, a técnica de SIG, que apresentou um pequeno aumento no tempo de execução, de 13%, enquanto que a ocupação da memória de programa aumentou em 126%. Isto acontece devido à proteção de trechos de código instanciados pelo compilador que não são utilizadas pela aplicação. O mesmo dado pode ser visto pela técnica de BRA, que apresentou um aumento de 5% no tempo de computação, mas de 91% na ocupação da memória de programa. A técnica BBD apresentou os piores resultados em termos de memória de programa, com um aumento de 302%, e tempo de execução, com um aumento de 232%.

As técnicas de VAR1 e VAR2 apresentaram resultados importantes quando levada em consideração a ocupação da memória de programa e o tempo de execução. A VAR2,

como era esperado, apresentou uma menor ocupação de memória de programa em relação à técnica VAR1, pois esta utiliza menos instruções de verificação. Entretanto, esta redução também era esperada quanto ao tempo de computação, visto que a técnica executa menos instruções. Uma possível explicação encontrada para o maior tempo de computação, quando comparada a VAR1, está relacionado à organização do *pipeline*, fazendo com que a verificação dos registradores após a escrita insira um atraso devido a dependência de dados introduzida na técnica de VAR2. A técnica de VAR3, como esperado, apresentou menor ocupação da memória de programa do que as técnicas de VAR1 e VAR2, bem como menor tempo de execução.

Finalmente, pode-se ver que a técnica SIG_VAR_BRA apresentou resultados em tempo de execução e ocupação de memória de programa levemente melhor do que a soma das técnicas separadas. A ocupação da memória de dados, por outro lado, apresentou exatamente a soma das técnicas separadas. No total, as técnicas combinadas apresentaram um aumento de 281% na memória de programa, enquanto apresentou um aumento de 118% no tempo de execução.

A tabela 4.4 ilustra o tempo total de execução (em milisegundos), ocupação total da memória de dados (em bytes), o que corresponde a ocupação de registradores mais o espaço reservado na memória de dados, e a ocupação da memória de programa (em bytes) de cada técnica implementada sobre a aplicação de ordenação *bubble sort*. Da mesma maneira que a multiplicação e matrizes, o algoritmo de *bubble sort* mostrou que as técnicas de replicação de variáveis são as que mais prejudicam o sistema, com os maiores tempos de execução e ocupação de memória de dados e de controle.

Tabela 4.4: Características físicas da ordenação *bubble sort*

	Original	SIG	BBD	VAR1	VAR2	VAR3	BRA	SIG_VAR_BRA
Tempo de Execução (ms)	0.23	0.28	0.75	0.47	0.53	0.35	0.24	0.53
Memória de Programa (bytes)	1212	2404	4756	2664	2588	2108	1580	3924
Memória de Dados (bytes)	120	128	128	240	240	240	120	248

A figura 4.5 ilustra a relação de tempo de execução e ocupação de memória de dados e de programa com relação ao programa original e desprotegido da ordenação *bubble sort*. Os resultados apresentados são condizentes com as comparações entre as técnicas realizada para aplicação de multiplicação de matrizes.

Entretanto, alguns resultados interessantes surgem quando comparadas as duas aplicações. Inicialmente, percebe-se que a técnica VAR1 para a multiplicação de matrizes chega em torno de 175% de ocupação da memória de programa, enquanto a ordenação *bubble sort* chega somente a 130%. Isto acontece pelo fato de que a aplicação de multiplicação de matrizes possui uma porcentagem maior de instruções de manipulação de dados do que a ordenação *bubble sort*. Da mesma maneira, percebe-se na técnica VAR2 um aumento no tempo de execução e uma redução no código de programa atribuídos ao fato de que a ordenação *bubble sort* acessa a memória com maior frequência, embora a multiplicação de matrizes tenha mais instruções de acesso a memória em seu código de programa.

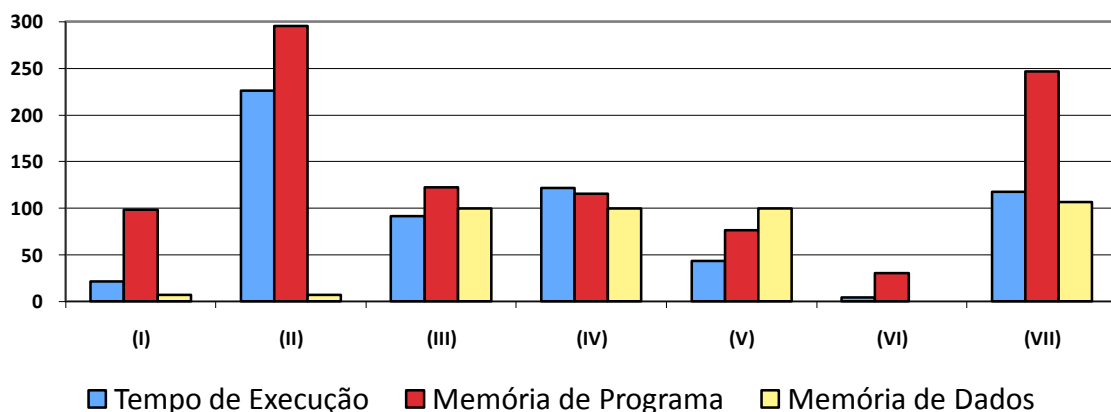


Figura 4.5: Relação de desempenho da ordenação *bubble sort*.

4.4 Mapeamento de Falhas

O principal objetivo deste trabalho é comparar diferentes técnicas de tolerância a falhas e a analisar suas vulnerabilidades. Para que este objetivo seja alcançado, é necessário injetar uma mesma falha nas diferentes versões de programa, protegidos por diferentes técnicas de tolerância a falhas. Como apresentado na sessão 3.4.1, uma falha é descrita através de um sinal e de um tempo. Sabendo que o sinal permanecerá o mesmo, é preciso descobrir o novo tempo no qual o sinal deverá ser perturbado de modo que cause o mesmo efeito sobre o microprocessador. Para isto, é necessário realizar um mapeamento temporal sobre o conjunto original de falhas.

Com a modificação do programa original por diferentes técnicas de tolerância a falhas, novas instruções são adicionadas e, em alguns casos, até mesmo a ordem de execução das instruções também é alterada. A análise e o mapeamento de falhas são tarefas extremamente custosas, devido à grande quantidade de falhas a serem analisadas e mapeadas e à complexidade envolvida. Por estas razões, este processo foi automatizado.

Para realizar automaticamente o mapeamento, foi implementado um *software* chamado Mapeador, capaz de analisar o fluxo de instruções de uma aplicação e mapear uma grande quantidade de falhas. Esta ferramenta, assim como o HPCT, foi implementada em Java, pela facilidade oferecida para a manipulação de *strings* e pela compatibilidade com o HPCT, visando a união das ferramentas no futuro.

O Mapeador utiliza três arquivos de configuração, além do conjunto original de falhas. São eles:

- Arquivo de definições da aplicação;
- Listas contendo as transições do registrador PC;
- Arquivo de mapeamento gerado pelo HPCT.

O arquivo de listas contém as transições do registrador PC em todos os estágios do *pipeline*, a cada ciclo de relógio, tanto do código original quanto do código protegido. Cada lista contém, para todos os estágios do *pipeline*, o tempo de execução e o endereço da instrução sendo executada, ou seja, o valor do PC. Um exemplo desta lista pode ser

visto na figura 4.6. Em destaque, a posição de memória de uma instrução passando pelos diferentes estágios de *pipeline*.

	u1_pf	u2_ei	u3_di	u4_ex	u5_mem
1	00000000	00000000	00000000	00000000	00000000
2	00000000	00000000	00000000	00000000	00000000
3	00000004	00000000	00000000	00000000	00000000
4	00000008	00000004	00000000	00000000	00000000
5	0000000c	00000008	00000004	00000000	00000000
6	00000010	0000000c	00000008	00000004	00000000
7	00000014	00000010	0000000c	00000008	00000004
8	00000014	00000010	0000000c	00000008	00000004
9	00000018	00000014	00000010	0000000c	00000008
10	0000001c	00000018	00000014	00000010	0000000c

Figura 4.6: Lista de transições do PC.

O arquivo de mapeamento, que pode ser gerado pelo HPCT, contém uma relação entre o endereço de memória das instruções originais e o endereço de memória das instruções após a transformação do código. Este será utilizado pelo Mapeador, como descrito a seguir.

Inicialmente, o Mapeador analisa a lista contendo as transições do PC da versão original e, baseado no tempo de injeção da falha, encontra todas as instruções sendo executadas pelo microprocessador naquele instante de tempo, ou seja, todas as instruções sendo processadas em algum estágio do *pipeline*. No caso do microprocessador miniMIPS, até 5 instruções podem estar sendo executadas simultaneamente, visto que este possui um *pipeline* de 5 estágios.

Num segundo momento, o caminho completo do sinal perturbado pela falha (encontrado no conjunto de falhas) é analisado com o objetivo de se descobrir o estágio do *pipeline* onde se encontra o sinal afetado e assim é encontrada a instrução afetada pela falha.

Sabendo a instrução afetada pela falha, é necessário encontrá-la no programa transformado. Através do arquivo de mapeamento gerado pelo HPCT, a posição original de memória da instrução é descoberta, e, com esta informação, sua nova posição de memória é encontrada, também através do arquivo de mapeamento.

A próxima etapa realizada pelo Mapeador é descobrir a vez na qual a instrução é executada, visto que uma mesma instrução pode ser executada inúmeras vezes durante uma execução (através de laços e chamadas de sub-rotinas). Sobre a lista de transições do PC original, é realizada uma contagem de quantas vezes a instrução afetada é executada e, baseado no tempo de injeção da falha, é encontrada a vez na qual a instrução foi executada que ela foi afetada pela falha. A mesma contagem é realizada sobre a lista de transições de PC do programa protegido, e é descoberta em qual das execuções da instrução que a falha foi injetada, caso a quantidade de execuções da instrução seja igual a do original. Caso contrário, a falha é descartada e substituída por uma nova, também gerada pelo Gerador de Falhas.

É importante lembrar que nenhuma das técnicas de transformação retira instruções do código original e, portanto, caso a quantidade de execuções de uma mesma instrução

no código protegido seja igual à quantidade de execuções no código original, necessariamente o mapeamento terá sido realizado corretamente.

Após esta análise, o tempo de injeção da falha é descoberto, mas ainda é necessário fazer com que esta falha afete o microprocessador. Em alguns casos, a falha deve ser armazenada num registrador para que gere um erro no microprocessador, sendo que, nestes casos, a falha deve ser injetada no estágio anterior ao encontrado pelo Mapeador, de maneira que esta seja registrada. São estas as falhas do tipo SEU.

A última análise realizada pelo Mapeador, portanto, é com relação ao tipo de falha injetada. No caso de uma falha do tipo SEU, ela é adiantada em um ciclo de relógio, dando assim tempo para que seja registrada. Em compensação, as falhas do tipo SET não precisam nenhuma alteração, visto que afetam diretamente a parte combinacional.

Após todo este processamento, o novo tempo de injeção da falha é descoberto e um novo arquivo de falhas é gerado, no mesmo formato descrito no capítulo 3.4.1, que descreve a geração do conjunto de falhas.

Devido à organização das instruções no *pipeline*, algumas falhas, mesmo mapeadas, não afetaram o resultado final da computação para algumas técnicas. Nestes casos, elas foram somente retiradas das estatísticas. A tabela 5.X mostra a quantidade de falhas que não afetaram o resultado final para cada técnica de proteção de falhas.

Tabela 4.5: Porcentagem de falhas perdidas durante o mapeamento para a multiplicação de matrizes.

	Técnica implementada						
	SIG	BBD	VAR1	VAR2	VAR3	BRA	SIG_VAR_BRA
Taxa de falhas perdidas (%)	1,6	10,6	12,7	13,5	34,7	2,6	13,3

Tabela 4.6: Porcentagem de falhas perdidas durante o mapeamento para a ordenação *bubble sort*.

	Técnica implementada						
	SIG	BBD	VAR1	VAR2	VAR3	BRA	SIG_VAR_BRA
Taxa de falhas perdidas (%)	1,7	9,9	10,7	11,3	10,5	1,2	11,3

A quantidade de falhas não mapeadas introduz uma certa variabilidade na comparação dos resultados, uma vez que uma falha não detectada por uma técnica pode não ter sido mapeada por outra técnica que não a detectaria. Além disso, existe uma variação entre o total de falhas injetadas para cada técnica.

4.5 Injeção de Falhas

Os grupos de falhas descritos no capítulo 4.2.3 foram inicialmente injetadas, para ambas as aplicações, no microprocessador miniMIPS e simuladas através do *software*

ModelSim. Da simulação inicial, foram coletadas as listas de transições de PC e um arquivo de registro, contendo o resultado da simulação de cada falha injetada, ou seja, se a falha causou um erro de execução e se este foi detectado pela aplicação. No total foram injetadas X falhas para a aplicação de multiplicação de matrizes e X falhas para a aplicação *bubble sort*.

As falhas foram então mapeadas para cada uma das técnicas implementadas através do Mapeador e novamente simuladas através do ModelSim. Como resultado, foi criado um arquivo contendo a resposta do sistema (correta ou errada), a resposta da técnica de tolerância a falhas (erro detectado ou não) e o tempo decorrido entre a injeção da falha e a detecção através da sub-rotina de erro.

Para a simulação, foi utilizado o ModelSim XE III versão 6.3 sobre um servidor Intel Core 2 Quad Q8400 com 8Gb de memória, rodando o sistema operacional Windows 7. O tempo total de simulação para todas as implementações de multiplicação de matrizes foi de 22 horas, enquanto que o tempo total de simulação para todas as implementações da ordenação *bubble sort* foi de 5 horas.

5 ANÁLISE E CLASSIFICAÇÃO DOS RESULTADOS

Este capítulo apresenta os resultados obtidos durante a campanha de injeção de falhas descrita no capítulo X, divididos entre a aplicação de multiplicação de matrizes e da ordenação *bubble sort*. As tabelas 5.1 e 5.2 apresentam o total de falhas detectadas sobre o total de falhas que causaram um erro na saída do sistema, ou seja, foram descartadas as falhas apresentadas nas tabelas 4.5 e 4.6.

Tabela 5.1: Taxa de detecção de falhas para multiplicação de matrizes.

	Técnica implementada						
	SIG	BBD	VAR1	VAR2	VAR3	BRA	SIG_VAR_BRA
Taxa de detecção (%)	2.7	5.6	87.5	88.6	80.8	0.5	88.6

Tabela 5.2: Taxa de detecção de falhas para ordenação *bubble sort*.

	Técnica implementada						
	SIG	BBD	VAR1	VAR2	VAR3	BRA	SIG_VAR_BRA
Taxa de detecção (%)	3.8	8.3	85.3	85.9	86.3	0.6	88.8

Os resultados apresentados nas tabelas 6.1 e 6.2 mostram que as técnicas de duplicação de variáveis (VAR1, VAR2, VAR3) apresentaram os melhores resultados de detecção de falhas, variando entre 89% e 86% para a ordenação *bubble sort* e multiplicação de matrizes, respectivamente. Quando comparadas entre si, nota-se que sua taxa de detecção de falhas é muito próxima com uma pequena variação menor do que 1%. O único valor que foge dessa estatística é a técnica de VAR3 aplicada à multiplicação de matrizes. Este valor difere das demais devido à alta taxa de falhas não mapeadas (enquanto as técnicas de VAR1 e VAR2 apresentaram uma taxa na ordem de 10%, a VAR3 apresentou uma taxa de perda de falhas na ordem de 30%). Em compensação, quando aplicada à ordenação *bubble sort*, onde a taxa de perda de falhas foi na mesma ordem das demais técnicas, a sua detecção foi parelha com as demais.

As técnicas de SIG, BBD e de BRA, diferentemente das demais, apresentaram resultados de detecção muito baixos. A primeira apresentou até 3,8% de falhas detectadas, enquanto a segunda detectou até 8,3% e a última não ultrapassou 1% de falhas detectadas. Este resultado era esperado, uma vez que as duas primeiras detectam

somente falhas que causam desvios entre diferentes blocos básicos enquanto que a última é capaz de detectar somente falhas sobre instruções de desvio condicionais.

Quando combinadas, as técnicas de SIG, VAR1 e BRA, através da técnica SIG_VAR_BRA, apresentaram taxas de detecção de falhas mais altas para ambas as aplicações. Este resultado mostra que é possível a combinação de diferentes técnicas na mesma aplicação. Com resultados melhores para as técnicas de SIG e BRA, provavelmente a técnica de SIG_VAR_BRA apresentaria resultados melhores.

Além disso, cabe ressaltar que as técnicas voltadas para a proteção de dados (VAR1, VAR2 e VAR3) apresentaram resultados melhores para a aplicação de multiplicação de matrizes, enquanto que as técnicas voltadas para a proteção do controle (SIG e BRA) apresentaram melhores taxas de detecção para a aplicação de ordenação *bubble sort*.

Os gráficos apresentados nas figuras 5.1 e 5.2 combinam os gráficos de desempenho apresentados nas figuras 4.4 e 4.5 com as taxas de detecção obtidas após a campanha de injeção de falhas.

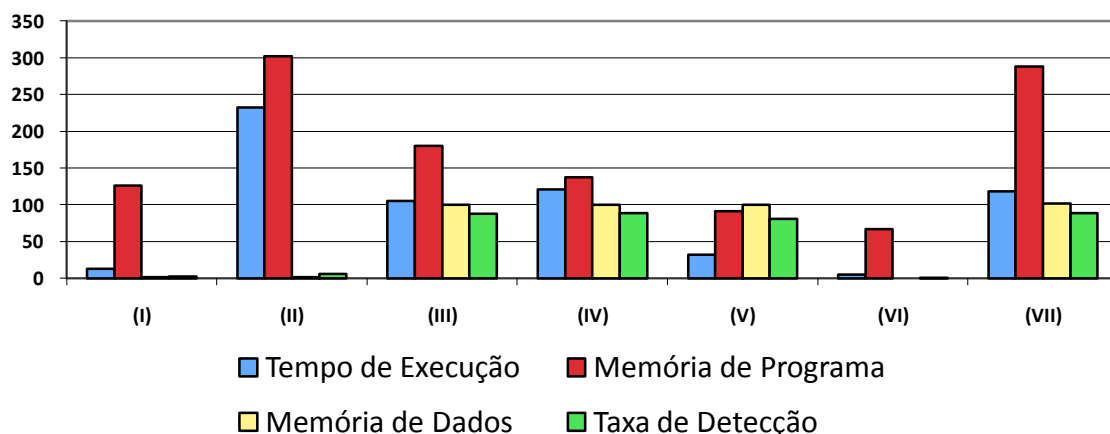


Figura 5.1: Relação de desempenho da multiplicação de matrizes.

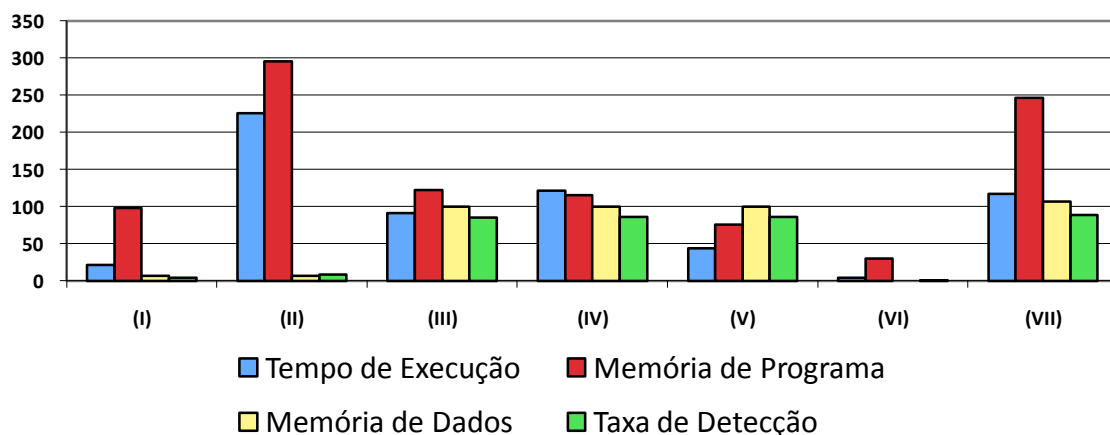


Figura 5.2: Relação de desempenho da multiplicação de matrizes.

De todas as técnicas, a que apresentou a melhor taxa de detecção foi a SIG_VAR_BRA ao combinar as técnicas SIG, VAR1 e BRA. Entretanto, a técnica VAR3, quando comparada com a técnica de VAR1, apresentou melhores resultados de desempenho com relação ao tempo de execução e ocupação de memória de programa, mantendo os resultados de ocupação de memória de dados e taxa de detecção de falhas.

Portanto, acredita-se que a combinação da técnica de SIG, VAR3 e BRA seja capaz de obter a mesma taxa de detecção da técnica SIG_VAR_BRA, mas com melhor desempenho relacionado ao tempo de execução e a ocupação da memória de programa. Por outro lado, espera-se também um aumento na latência de detecção de falhas com a utilização da técnica de VAR3 (a latência das técnicas de proteção aos dados será analisada no capítulo 6.4).

Para uma melhor análise das vulnerabilidades de cada grupo de regras e uma melhor comparação das técnicas, as taxas de detecção de falhas foram divididas entre o tipo de falha injetada (SEU e SET), o local da injeção da falha (*controlpath* e *datapath*, sendo este subdividido entre ULA, banco de registradores e outros) e o efeito da falha injetada (efeito de dados e efeito de controle), como descrito nas tabelas 4.1 e 4.2.

A tabela 5.3 apresenta a porcentagem de falhas detectadas sobre o total de falhas injetadas sobre os grupos SIG, BBD, VAR1, VAR2, VAR3, BRA e SIG_VAR_BRA que causaram um erro no resultado final do sistema, ou seja, excluídas as falhas apresentadas na tabela 4.5, executando a aplicação de multiplicação de matrizes. A tabela 5.4 apresenta os mesmos resultados para o algoritmo de ordenação *bubble sort*, excluídas as falhas apresentadas na tabela 4.6.

Os gráficos apresentados nas figuras 5.3 e 5.4 mostram uma comparação entre os dados da tabela 5.3 (multiplicação de matrizes), divididos entre o efeito da falha, ou seja, SEU ou SET, respectivamente. Da mesma maneira, os gráficos apresentados nas figuras 5.5 e 5.6 contém uma comparação com relação aos dados constantes na tabela 5.4 (ordenação *bubble sort*).

Devido à grande quantidade de dados apresentados, a análise dos resultados será dividida entre o efeito, o local de injeção e o tipo de cada falha injetada no microprocessador. As sessões a seguir discutirão os resultados detalhadamente.

Tabela 5.3: Resultado da injeção de falhas sobre a aplicação multiplicação de matrizes para as técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA.

Tipo de Falha	Local de Injeção		Técnica de proteção													
			Efeito de Dados							Efeito de Controle						
			(I)	(II)	(III)	(IV)	(V)	(VI)	(VII)	(I)	(II)	(III)	(IV)	(V)	(VI)	(VII)
SET	<i>Controlpath</i>		0	10.2	100	100	100	0	100	6.3	20	43.8	40.6	34.4	6.5	45.5
	<i>Datapath</i>	ULA	0	0	100	100	100	0	100	0	0	100	100	100	0	100
		Banco	0	0	100	100	100	0	100	0	0	100	100	100	0	100
		Outros	0	2.4	100	100	100	0	100	0	50	100	100	100	0	100
	Total		0	5.1	100	100	100	0	100	4.4	16.3	60	57.8	53.3	4.5	60.9
SEU	<i>Controlpath</i>		0	0	100	100	100	0	100	20	11.4	37.1	34.3	21.2	0	42.4
	<i>Datapath</i>	ULA	0	0	100	100	100	0	100	-	-	-	-	-	-	-
		Banco	0	0	100	100	100	0	100	0	7.7	100	100	100	0	100
		Outros	0	0	100	100	100	0	100	16.7	0	100	100	100	0	100
	Total		0	0	100	100	100	0	100	14.8	9.8	59.3	55.8	50.9	0	63.5

Tabela 5.4: Resultado da injeção de falhas sobre a aplicação de ordenação *bubble sort* para as técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA.

Tipo de Falha	Local de Injeção		Técnica de proteção													
			Efeito de Dados							Efeito de Controle						
			(I)	(II)	(III)	(IV)	(V)	(VI)	(VII)	(I)	(II)	(III)	(IV)	(V)	(VI)	(VII)
SET	<i>Controlpath</i>		4.5	15.8	100	100	100	0	100	8	16.4	68.9	70.7	71.6	2.3	80.8
	<i>Datapath</i>	ULA	0	0	100	100	100	0	100	0	0	100	100	100	0	100
		Banco	0	0	100	100	100	0	100	0	0	100	100	100	0	100
		Outros	0	0	100	100	100	0	100	0	7.1	100	100	100	0	100
	Total		2.2	7	100	100	100	0	100	5.3	11.8	80.5	81.5	82.5	1.5	87.9
SEU	<i>Controlpath</i>		0	0	100	100	100	0	100	1.3	6.9	68.1	69.1	69.1	0	71.4
	<i>Datapath</i>	ULA	-	-	-	-	-	-	-	-	-	-	-	-	-	-
		Banco	0	0	100	100	100	0	100	12.1	12.1	100	100	100	0	100
		Outros	0	0	100	100	100	0	100	0	0	100	100	100	0	100
	Total		0	0	100	100	100	0	100	3.8	7.4	81.8	82.2	82.5	0	83.5

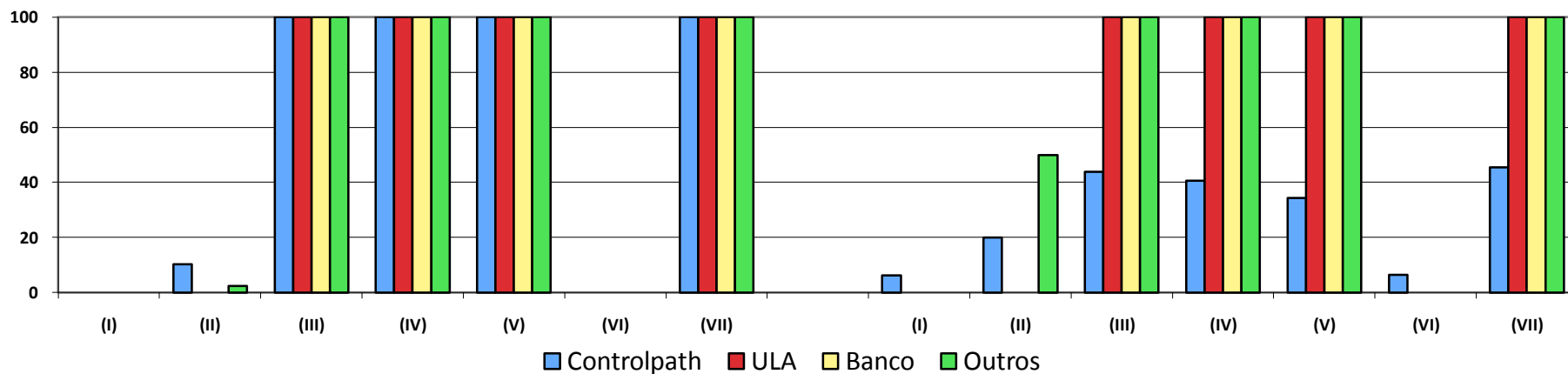


Figura 5.3: Visão geral das técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de multiplicação de matrizes sobre falhas do tipo SEU.

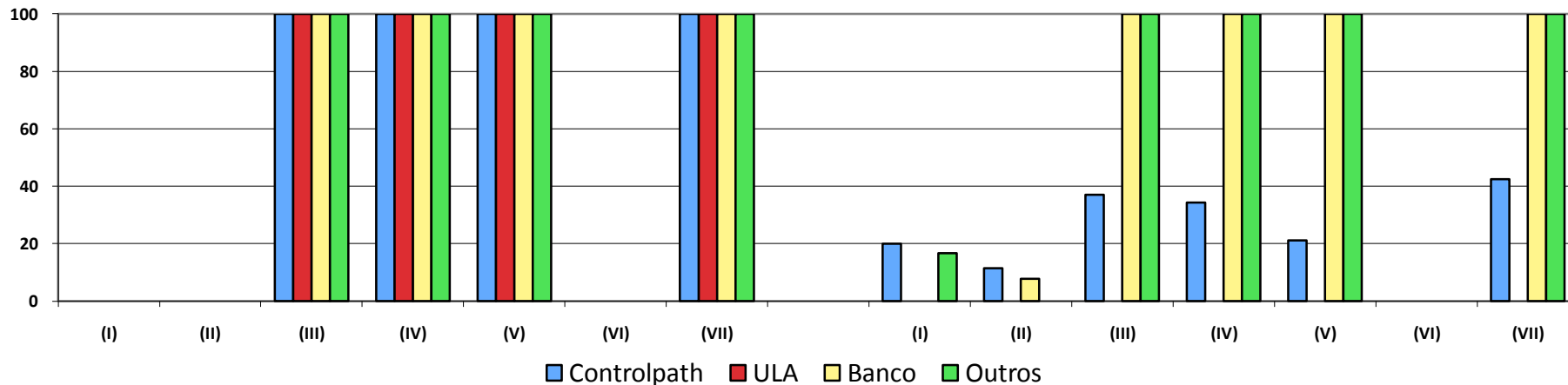


Figura 5.4: Visão geral das técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de multiplicação de matrizes sobre falhas do tipo SET.

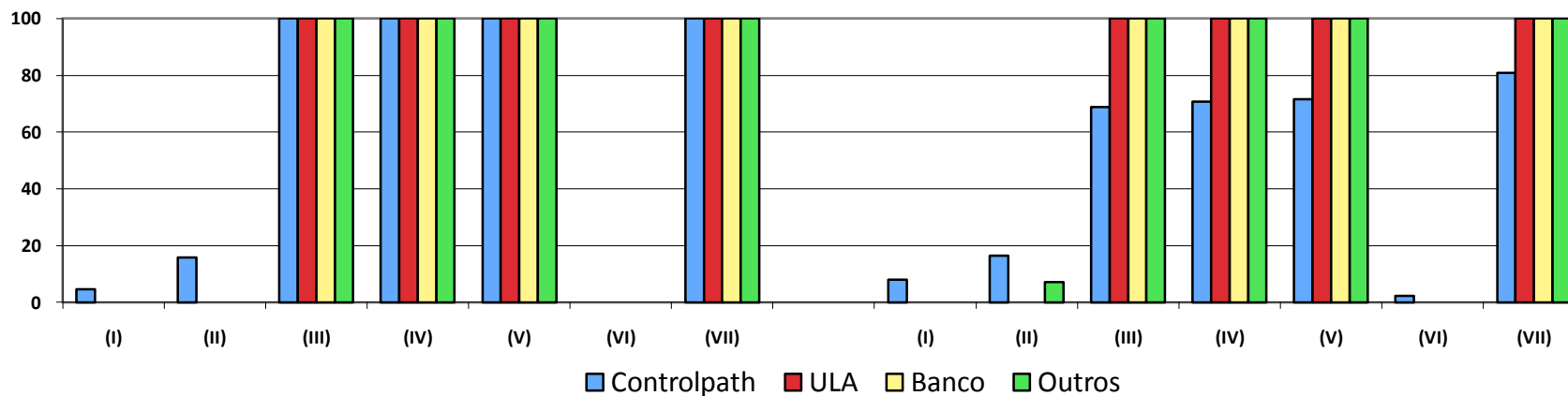


Figura 5.5: Visão geral das técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de ordenação *bubble sort* sobre falhas do tipo SEU.

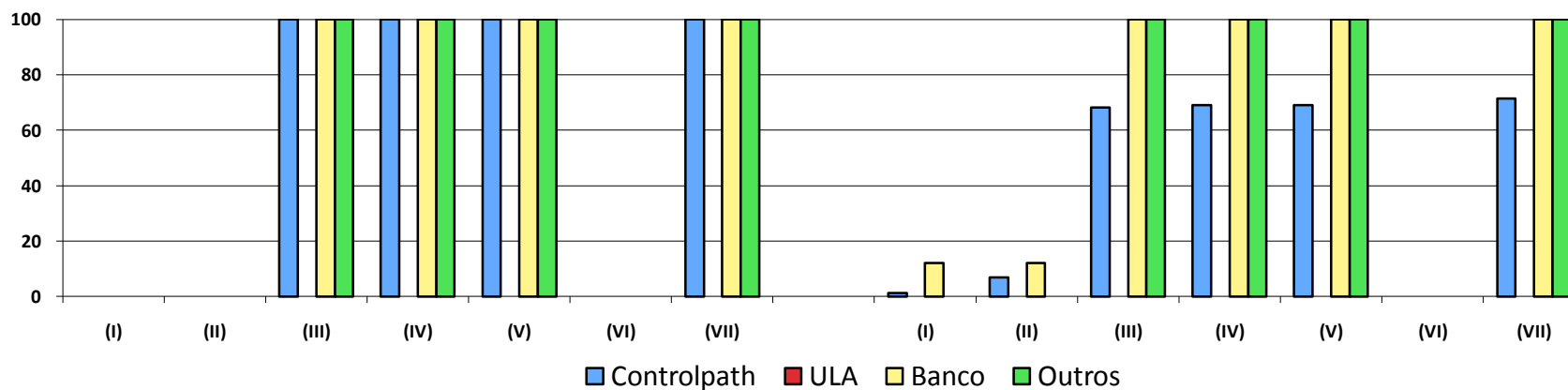


Figura 5.6: Visão geral das técnicas (I) SIG, (II) BBD, (III) VAR1, (IV) VAR2, (V) VAR3, (VI) BRA e (VII) SIG_VAR_BRA para o algoritmo de ordenação *bubble sort* sobre falhas do tipo SET.

5.1 Análise por Efeito de Falhas

A primeira análise realizada diz respeito às taxas de detecção de cada técnica com relação ao efeito produzido pela falha sobre o microprocessador. Como discutido no capítulo 4.2.2, os efeitos das falhas foram caracterizados como de dados (quando somente o resultado final foi afetado) e de controle (quando o fluxo de execução da aplicação foi alterado).

As figuras 5.7 e 5.8 apresentam uma comparação das taxas de detecção de cada técnica testada, divididas entre efeito de dados e efeito de controle para as aplicações de multiplicação de matrizes e ordenação *bubble sort*, respectivamente.

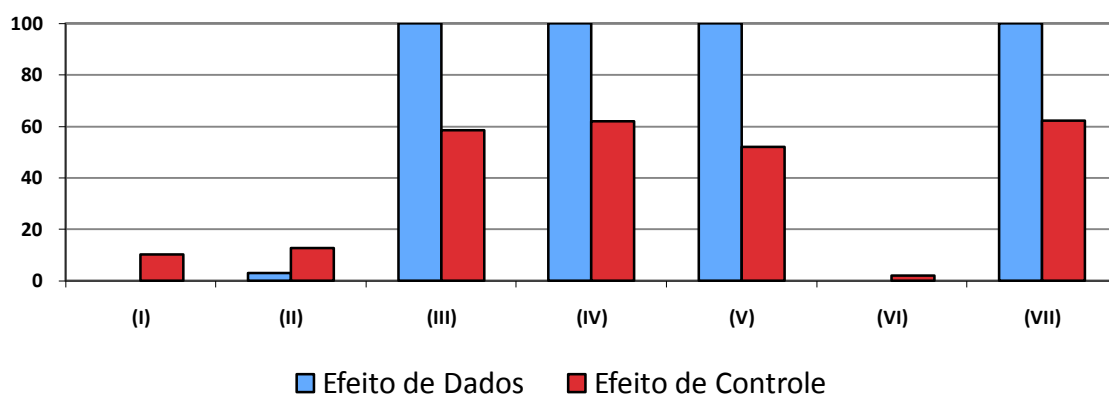


Figura 5.7: Taxa de detecção de falhas de acordo com seu efeito para a aplicação de multiplicação de matrizes.

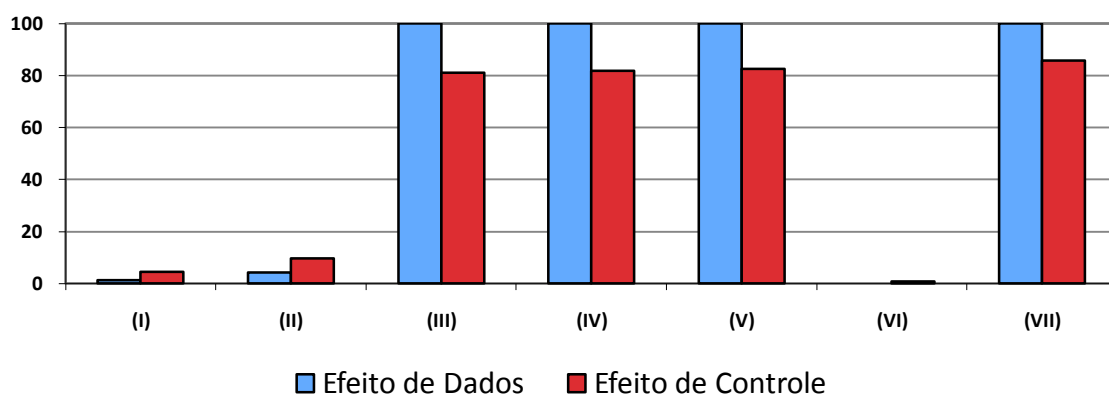


Figura 5.8: Taxa de detecção de falhas de acordo com seu efeito para a aplicação de ordenação *bubble sort*.

Analisando ambos os gráficos, observa-se que as técnicas de proteção de dados VAR1, VAR2 e VAR3 foram capazes de detectar todas as falhas com efeito de dados, tanto na aplicação de multiplicação de matrizes quanto de ordenação *bubble sort*. Este resultado era esperado, uma vez que o objetivo principal destas técnicas é a proteção de falhas com este efeito. Por outro lado, é interessante a taxa de detecção destas mesmas

técnicas com relação às falhas com efeito de controle, visto que estas três técnicas não apresentam qualquer implementação que vise a sua detecção.

Existem duas explicações para este resultado. A primeira é com relação ao desvio de fluxo ter acontecido entre a execução da instrução original e sua cópia, gerando assim uma inconsistência entre as variáveis, que pode ser detectada quando esta for lida novamente. A segunda explicação é com relação à classificação dos efeitos de cada falha, que mesmo classificada como um efeito de controle, ela pode ter afetado a parte de dados do microprocessador, causando assim a alteração do fluxo de execução (falha sobre um registrador de desvio condicional, por exemplo). Nestes casos, é possível que o fluxo de controle tenha sido alterado antes que a falha pudesse ser detectada, caracterizando assim uma falha com efeito sobre o controle.

Os resultados apresentados pelas técnicas visando a proteção do controle, ou seja, SIG, BDD e BRA mostram valores muito abaixo do esperado. Analisando-se os resultados obtidos pela multiplicação de matrizes, percebe-se que nenhuma das técnicas foi capaz de detectar as falhas com efeito de dados, resultado aceitável, visto que estas não apresentam qualquer implementação no sentido de proteger o sistema contra este tipo de falhas. Entretanto, ao analisar a taxa de detecção das falhas de controle, percebe-se que são muito baixas, na ordem de 10% para a técnica de SIG e 2% para a técnica de BRA.

A principal explicação para estas baixas taxas de detecção está relacionada à baixa ocorrência de falhas que causam um desvio incorreto entre diferentes blocos básicos e a falhas sobre desvios condicionais, situações nas quais as técnicas de SIG e BRA detectariam o erro.

Ainda como esperado, a combinação das técnicas SIG, VAR1 e BRA mostra-se útil somente para a detecção de falhas de controle, uma vez que não alterou em nada os resultados sobre as falhas com efeito de dados.

5.2 Análise por Local de Injeção de Falhas

A análise das taxas de detecção com relação ao local de injeção de falhas é muito interessante, visto que não necessariamente uma falha com efeito de dados foi injetada na parte de dados do microprocessador, da mesma maneira que uma falha com efeito de controle não necessariamente foi injetada na parte de controle do microprocessador. Como descrito no capítulo 4.2.2, o microprocessador foi dividido em 2 grupos: *datapath* e *controlpath*.

As figuras 5.9 e 5.10 apresentam uma comparação das taxas de detecção de cada técnica testada, divididas entre o local de injeção, sendo este o *datapath* ou o *controlpath*, para as aplicações de multiplicação de matrizes e ordenação *bubble sort*, respectivamente.

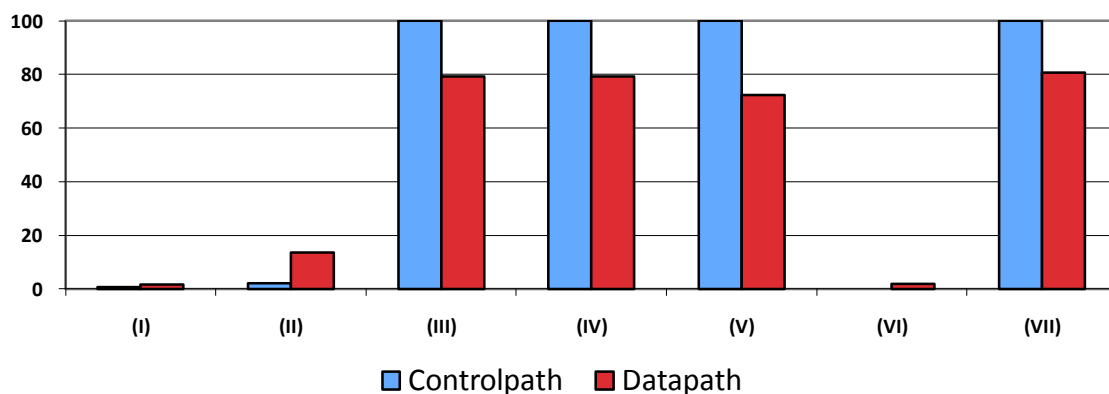


Figura 5.9: Taxa de detecção de falhas de acordo com o local de injeção para a aplicação de multiplicação de matrizes.

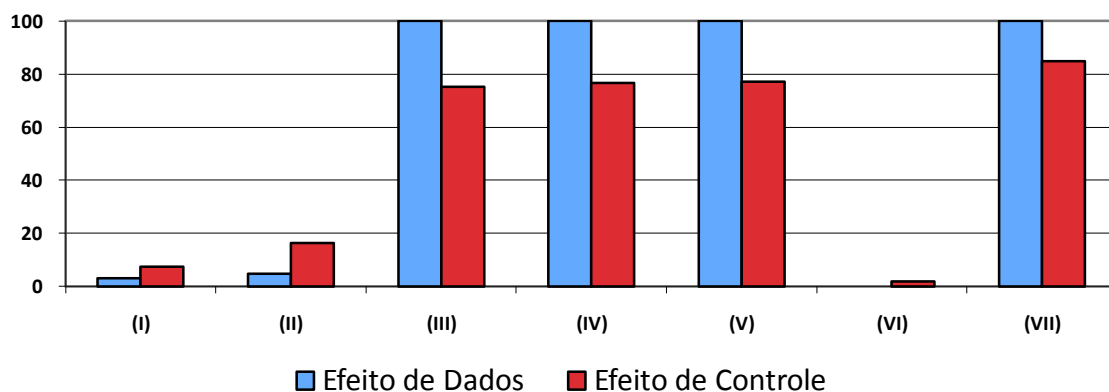


Figura 5.10: Taxa de detecção de falhas de acordo com o local de injeção para a aplicação de ordenação *bubble sort*.

Os resultados para as técnicas de proteção de dados (VAR1, VAR2 e VAR3) foram semelhantes aos obtidos no capítulo 5.7 e 5.8, quando as taxas de detecção foram comparadas com relação ao efeito de cada falha. Tanto para a multiplicação de matrizes quanto para a ordenação *bubble sort*, as três técnicas apresentaram 100% de detecção de falhas para falhas no *datapath*. É importante lembrar que estas falhas geraram tanto efeitos de dados quanto de controle sobre o microprocessador, ou seja, que parte destas falhas gerou erros no fluxo de execução do programa e, mesmo assim, foram detectadas pelas três técnicas testadas.

Quando unidos, os gráficos apresentados neste capítulo e os gráficos 5.7 e 5.8, mostram que as técnicas de proteção aos dados são capazes de detectar todas as falhas injetadas no *datapath* e todas as falhas com efeito sobre os dados da aplicação, ou seja, estas apresentam um alto grau de proteção com relação à parte de dados do microprocessador, objetivo para o qual foram propostas.

Da mesma maneira, as técnicas de proteção ao controle SIG, BBD e BRA foram capazes de detectar falhas injetadas na parte de controle do microprocessador, ainda que com resultados abaixo do esperado. Ao contrário das técnicas de proteção aos dados, percebe-se, ao analisar os gráficos apresentados neste capítulo com os gráficos 5.7 e 5.8,

que as técnicas de proteção ao controle detectam mais falhas injetadas na parte de controle e com efeito sobre o fluxo de execução da aplicação. Diferentemente do esperado, as técnicas de SIG e BBD foram capazes de detectar falhas com efeito de dados, e isto se explica pelo fato de que com o mapeamento e a troca na ordem de execução de algumas instruções, as falhas com efeito de dados apresentaram também um efeito de controle e, assim, foram detectadas por estas técnicas.

Quando combinadas, as técnicas SIG, VAR1 e BRA apresentaram a mesma taxa de detecção de falhas injetadas na parte de dados do microprocessador e uma taxa de detecção mais elevada para as falhas injetadas na parte de controle com relação as técnicas analisadas separadamente. Este resultado comprova que é possível combinar as técnicas visando melhores taxas de detecção de falhas, da mesma maneira que comprova que cada técnica é capaz de proteger um conjunto único de falhas.

5.3 Análise por Tipo de Falha

A última análise se refere às taxas de detecção das técnicas implementadas quanto ao tipo de falha injetada no microprocessador, ou seja, falhas do tipo SEU e do tipo SET, definidas no capítulo 4.2.2. Esta análise é importante, uma vez que a maioria dos trabalhos encontrados na literatura visa a proteção somente das falhas do tipo SEU, embora a ocorrência das falhas do tipo SET vem aumentando consideravelmente devido aos avanços tecnológicos.

As figuras 5.11 e 5.12 apresentam uma comparação das taxas de detecção de cada técnica testada, divididas entre os tipos de falha SEU e SET para as aplicações de multiplicação de matrizes e ordenação *bubble sort*, respectivamente.

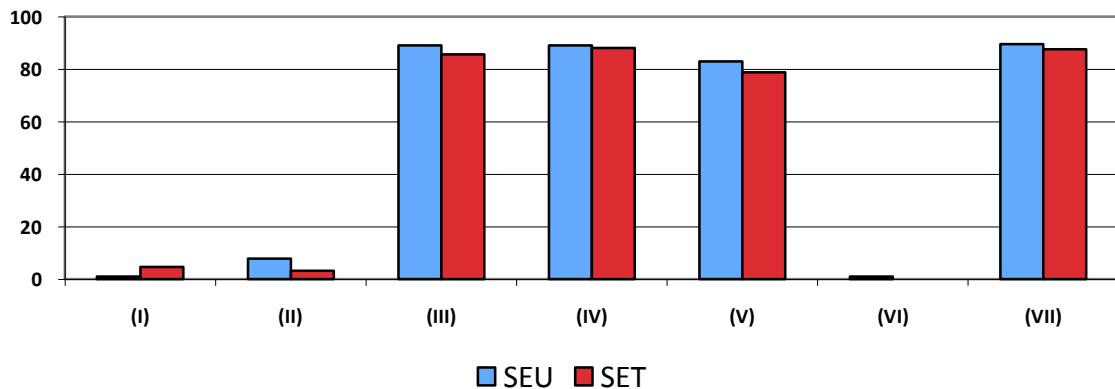


Figura 5.11: Taxa de detecção de falhas de acordo com o seu tipo para a aplicação de multiplicação de matrizes.

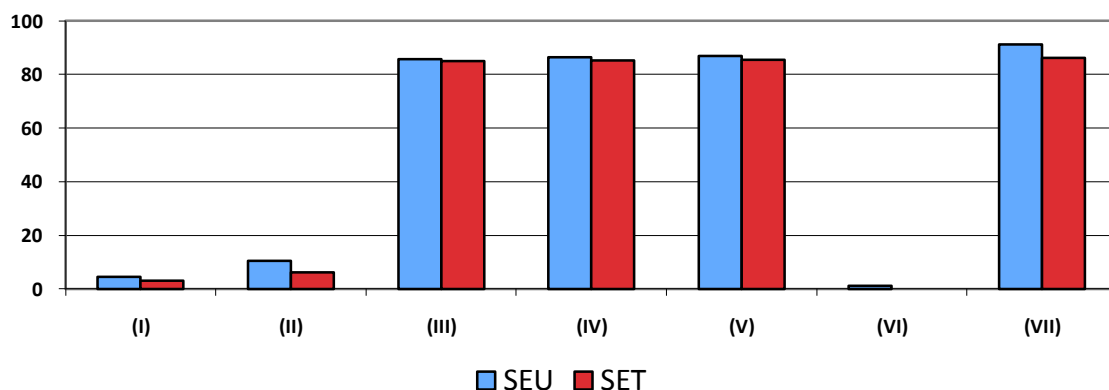


Figura 5.12: Taxa de detecção de falhas de acordo com seu tipo para a aplicação de ordenação *bubble sort*.

Ambos os gráficos mostram taxas de detecção de falhas com variações menores do que 5% entre falhas do tipo SEU e SET. Outro dado interessante é a maior taxa de detecção para falhas do tipo SEU do que SET para todas as técnicas de tolerância a falhas, com exceção da técnica SIG aplicada à multiplicação de matrizes.

5.4 Análise de Latência para a Detecção de Falhas

A única diferença entre as técnicas de VAR1, VAR2 e VAR3 é a verificação de consistência entre as variáveis originais e suas réplicas. As técnicas de VAR1 e VAR2 são as que mais verificam a consistência das variáveis, enquanto que a VAR3 é a técnica que menos realiza verificações. A principal diferença entre as técnicas de VAR1 e VAR2 é que a primeira verifica a consistência antes de utilizar os valores, enquanto que a segunda verifica a consistência após a utilização dos dados. A técnica de VAR3, por outro lado, verifica apenas os registradores utilizados para a leitura e escrita em memória.

Por outro lado, as verificações de consistência são a maneira pela qual estas técnicas detectam falhas e informam o sistema das mesmas, fazendo com que, quanto menos verificações forem realizadas, maior será o tempo médio que a técnica precisará para informar o sistema de uma falha, ou seja, maior será a latência de detecção da falha.

A fim de verificar esta característica, as técnicas de VAR1, VAR2 e VAR3 foram simuladas novamente, porém agora verificando-se o tempo decorrido entre a injeção da falha no sistema e o aviso dado pela técnica da presença da falha. Este procedimento foi realizado através da modificação do *script* de coleta de dados, fazendo com que este gravasse um arquivo de registro contendo o momento em que a falha foi detectada. As falhas não detectadas pelas técnicas não foram contabilizadas.

As tabelas 5.5 e 5.6 mostram a latência média de detecção de falhas das técnicas de VAR1, VAR2 e VAR3 para as aplicações de multiplicação de matrizes e ordenação *bubble sort*, respectivamente.

Tabela 5.5: Tempo médio de detecção de falha para multiplicação de matrizes.

	Técnica implementada		
	VAR1	VAR2	VAR3
Tempo médio de detecção (ciclos de relógio)	19.49	21.21	69.76

Tabela 5.6: Tempo médio de detecção de falha para ordenação *bubble sort*.

	Técnica implementada		
	VAR1	VAR2	VAR3
Tempo médio de detecção (ciclos de relógio)	11.26	10.98	19.46

As tabelas mostram que as latências de detecção de falha das técnicas VAR1 e VAR2 são muito parecidas, com uma diferença entre 2.5% e 8.1%. No caso da aplicação de multiplicação de matrizes a VAR1 apresentou uma menor latência média, com uma redução de 8.1%, enquanto que no caso da ordenação *bubble sort*, a técnica VAR2 apresentou uma latência 2.5% menor do que a técnica de VAR1.

A técnica VAR3, como esperado, apresentou os maiores tempos de latência das técnicas testadas, com um acréscimo variando de 72.82% a 257.93%, quando comparada a técnica VAR1. Isto se deve à menor frequência de realização de verificações de consistência. Esta variação, entre as aplicações de multiplicação de matrizes e ordenação *bubble sort*, se dá pelo fato de que a aplicação de multiplicação de matrizes acessa a memória menos frequentemente do que a aplicação de ordenação *bubble sort*, aumentando consideravelmente a latência de detecção. Este último dado é confirmado pelos gráficos apresentados nas figuras 5.1 e 5.2, mostrando um menor tempo de execução da técnica de VAR3 aplicada à multiplicação de matrizes do que à ordenação *bubble sort*.

A maior latência de detecção da aplicação de multiplicação de matrizes, que chega a ser o dobro do tempo de latência da ordenação *bubble sort*, acontece pelo fato de que a multiplicação de matrizes possui mais variáveis e menor localidade temporal, ou seja, a multiplicação de matrizes utiliza suas variáveis com menor frequência do que a ordenação *bubble sort*.

6 CONCLUSÃO

Os avanços tecnológicos na indústria de semicondutores permitiu a fabricação de circuitos integrados com maior densidade e com um número cada vez maior de funcionalidades. Da mesma maneira, reduziu a confiabilidade dos dispositivos CMOS como consequência de diversos problemas provenientes das características físicas dos dispositivos, dentre eles: menores tensões de operação, menor dimensão dos transistores, dispositivos mais velozes e menor distância entre os transistores no silício. Com estes avanços e a redução de confiabilidade, efeitos transientes passaram a afetar sistemas microprocessados com maior frequência e, de maneira geral, as novas tecnologias tornaram-se mais sensíveis a *bit-flips* (SEU) e a efeitos do tipo SET em sua lógica.

Este trabalho teve como principal objetivo a implementação de diferentes técnicas de tolerância a falhas baseadas em *software*, com o intuito de compará-las e analisá-las detalhadamente com relação ao tamanho do código de programa, tamanho da memória de dados, tempo de execução das aplicações e taxa de detecção de falhas. Para isto, foram implementadas ferramentas de geração e injeção de falhas e de coleta e análise de resultados e uma ferramenta de transformação automática de códigos de programa em códigos protegidos de acordo com as técnicas de proteção escolhidas. Diferentes versões de códigos protegidos para as aplicações de multiplicação de matrizes e ordenação *bubble sort* foram gerados através de diferentes técnicas de tolerância a falhas. Por fim, um conjunto de falhas para a simulação das técnicas foi gerado e uma campanha de injeção de falhas foi realizada.

A análise dos resultados obtidos através da campanha de injeção de falhas mostra que as técnicas de duplicação de variáveis (VAR1, VAR2, VAR3) apresentaram os melhores resultados de detecção de falhas na ordem de 87%. As técnicas de SIG, BBD e de BRA, diferentemente das demais, apresentaram resultados de detecção muito baixos. Enquanto a SIG detectou cerca de 4%, a BBD detectou até 8% e a técnicas de BRA não ultrapassou 1% de falhas detectadas. Quando combinadas, as técnicas de SIG, VAR1 e BRA, através da técnica SIG_VAR_BRA, apresentaram taxas de detecção de falhas mais altas, na ordem de 89%. Este resultado mostra que é possível a combinação de diferentes técnicas na mesma aplicação. Com resultados melhores para as técnicas de SIG e BRA, provavelmente a técnica de SIG_VAR_BRA apresentaria resultados melhores.

Com a separação das falhas entre seu tipo (SEU e SET), local da injeção (*controlpath* e *datapath*) e efeito (efeito de dados e efeito de controle), percebeu-se que a técnicas de proteção aos dados (VAR1, VAR2 e VAR3) detectaram todas as falhas injetadas no *datapath* e as falhas com efeito de dados, enquanto que as demais técnicas não foram capazes de detectar todas as falhas injetadas no *controlpath* nem as falhas com efeito de controle. Com estes dados, é possível identificar o principal desafio na proteção de microprocessadores com técnicas de tolerância a falhas baseadas em *software*: as falhas que afetam o fluxo de execução do código de programa. Quando analisada a latência de detecção de falhas, foi atestado que a técnicas de VAR1 possui a menor latência, enquanto que a técnica de VAR3 possui uma latência até duas vezes maior do que a VAR1. Por outro lado, a técnica de VAR apresentou uma perda de desempenho muito inferior à técnica de VAR1.

As ferramentas implementadas durante este trabalho formam uma base concreta para a implementação de novas técnicas baseadas em *software*, uma vez que toda a estrutura de processamento de código (leitura e interpretação do código na linguagem Assembly, processamento do código, montagem da estrutura de blocos básicos, correção dos endereços de desvio, dentre outros) foi codificada e testada para o microprocessador miniMIPS. Estes métodos permitem também a combinação de técnicas baseadas em *software* com técnicas em *hardware*, fazendo que com o HPCT introduza instruções no código original capazes de controlar um módulo de hardware externo, sem alterar a arquitetura interna do microprocessador. Como trabalho futuro, é interessante estender o HPCT para outros microprocessadores e a implementação de novas técnicas de tolerância a falhas.

Da mesma maneira, tais ferramentas permitem, através do Injetor, a simulação de outras técnicas, tanto em *software* quanto em *hardware*, propostas e implementadas por outros alunos, dos laboratórios da UFRGS e de outras universidades. A simulação de outras arquiteturas de microprocessadores pode ser facilmente adaptada, da mesma maneira que módulos descritos nas linguagens VHDL e Verilog. Como trabalho futuro, o Injetor será otimizado e implementado nos servidores do laboratório, possibilitando o acesso remoto para usuários em geral, através de uma página web. Com isto, serão oferecidas as opções de gerar, injetar, coletar e enviar os resultados remotamente para o usuário de forma automática e transparente.

Este trabalho resultou em algumas publicações para congressos e eventos da área, tais como (AZAMBUJA et al., 2010a), (AZAMBUJA et al., 2010b) e (AZAMBUJA et al., 2010c), que são também apresentados no Anexo B. Existe ainda um trabalho em processo de revisão para a revista *Transactions on Nuclear Science* (TNS) e outros quatro artigos que utilizam as metodologias e técnicas apresentada neste trabalho aguardando resposta para publicação na revista *Journal of Electronic Testing: Theory and Applications* (JETTA) e nos congressos *Design, Test and Automation in Europe* (DATE), *VLSI Test Symposium* (VTS) e *Defect and Fault Tolerance in VLSI Systems* (DFT).

REFERÊNCIAS

- AGARWAL, A. et al. A process-tolerant cache architecture for improved yield in nanoscale technologies. **IEEE Transactions on Very Large Scale Integration Systems**, Princeton, USA: IEEE Circuits and Systems Society, 2005, v. 13, n. 1, p. 27-38.
- ALKHALIFA, Z. et al. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Transactions on Parallel and Distributed Systems**, New York, USA: IEEE Computer Society, 1999, v. 10, n. 6, p. 627-641.
- AZAMBUJA, J. R.; SOUSA, F.; ROSA, L.; KASTENSMIDT, F. Non-Intrusive Hybrid Signature-Based Technique to Detect SEU and SET Faults in Microprocessors. In European Conference on Radiation and Its Effects on Components and Systems: **Proceedings...** Washington, USA: IEEE Computer Society, 2010.
- AZAMBUJA, J. R.; SOUSA, F.; ROSA, L.; KASTENSMIDT, F. The limitations of software signature and basic block sizing in soft error fault coverage. In Latin-American Test Workshop: **Proceedings...** Washington, USA: IEEE Computer Society, 2010, p. 1-6.
- AZAMBUJA, J. R.; SOUSA, F.; ROSA, L.; KASTENSMIDT, F. Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors. In Latin American Symposium on Circuits and Systems: **Proceedings...** Washington, USA: IEEE Computer Society, 2010, p. 300-303.
- BOLCHINI, C. et al. Reliable system specification for self-checking datapaths. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, DATE 2005, Munich, GER. **Proceedings...** Washington, USA: IEEE Computer Society, 2005, p. 334-342.
- CHEN, C.; WU, C. An adaptative code rate EDAC scheme for random access memory. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, DATE 2005, Munich, GER. **Proceedings...** Washington, USA: IEEE Computer Society, 2010, p. 735-740.
- CHEYNET, P. et al. Experimentally evaluating an automatic approach for generating safety-critical *software* with respect to transient errors. **IEEE Transactions On Nuclear Science**, [S.l.]: IEEE Nuclear and Plasma Sciences Society, 2000, v. 47, n. 6 (part 3), p. 2231-2236.
- DODD, P. et al. Production and propagation of single-event transients in high-speed digital logic ics. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA: IEEE Computer Society, 2004, v. 51, n. 6 (part 2), p.3278–3284.

FERLET-CAVROIS, V. et al. Direct measurement of transient pulses induced by laser irradiation in deca-nanometer SOI devices. **IEEE Transactions On Nuclear Science**, Los Alamitos, USA : IEEE Computer Society, 2005, v. 52.

HANGOUT, L.; JAN, S: The minimips project, 2009. Available at: <<http://www.opencores.org/projects.cgi/web/minimips/overview>>. Accessed: July 13th, 2010.

HEIJMEN, T. **Radiation induced soft errors in digital circuits: a literature survey**, Eindhoven, NDL: Philips Electronics National Laboratory, 2002.

HOMPSON, S. et al. In search of forever: continued transistor scaling one new material at a time. **IEEE Transactions on Semiconductor Manufacturing**, New York, USA: IEEE Computer Society, 2005, v. 18, n.1, p. 26-36.

IEEE,

INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS, 2008 UPDATE, ITRS 2008, 2008, [S.l.]. Semiconductor industry association. Available at: <http://www.itrs.net/Links/2008ITRS/Update/2008_Update.pdf>. Accessed: July 13th, 2010.

KARNIK, T.; HAZUCHA, P.; PATEL, J. Characterization of soft errors caused by single event upsets in CMOS processes. **IEEE Transactions on Dependable and Secure Computing**. New York, USA: IEEE Computer Society, 2004, v. 1, n. 2, p. 128-143.

KIM, N. S. et al. Leakage current: moore's law meets static power. **Computer**, Los Alamitos, USA : IEEE Computer Society, 2003, v. 36, p. 68-75.

MAHMOOD, A.; McCLUCKEY, E. Concurrent error detection using watchdog processors-a survey. **IEEE Transactions on Computers**. [S.l.]: [IEEE Computer Society?], 1988, v. 37, n. 2, p. 160-174.

MCFEARIN, L; NAIR, V. Control-flow checking using assertions. In CONFERENCE ON DEPENDABLE COMPUTING FOR CRITICAL APPLICATIONS, Urbana-Champaign, USA: **Proceedings...** Washington, USA: IEEE Computer Society, 1995, p. 103-112.

MODELSIM SUPPORT, 2010. Mentor Graphics. Available at: <<http://www.model.com/content/modelsim-support>>. Accessed: July 13th, 2010.

MOORE, G., Cramming More Components onto Integrated Circuits, **Electronics Magazine**, 38, 114-117, 1965.

NICOLESCU, B.; VELAZCO, R. Detecting soft errors by a purely *software* approach: method, tools and experimental results. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, DATE 2003, Munich, GER. **Proceedings...** Washington, USA: IEEE Computer Society, 2003, p. 20057.

OH, N.; MITRA, S.; McCLUSKEY. ED4I: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers**, 2002, v. 51, n. 2, p. 180-199.

OH, N.; SHIRVANI, E.; McCLUSKEY, E. Control-flow checking by *software* signatures. **IEEE Transactions on Reliability**. [S.l.]: IEEE Computer Society?, 2002, v. 51, n. 2, p. 111-122.

- PATTERSON, D. A.; HENNESSY, J. L. Computer organization and design: the *hardware/software* interface. Morgan Kaufmann, 2009.
- PFLANZ, M.; VIERHAUS, H. Online check and recovery techniques for dependable embedded processors. **IEEE Micro**, 2001, v. 21, n. 5, pp. 24-40.
- PRADHAN, D. Fault-tolerant computer system design. Upper Saddle River, USA : Prentice-Hall, 1995.
- REBAUDENGO, M. et al. Combined *software* and *hardware* techniques for the design of reliable IP processors. **Proceedings...** Washington, USA: IEEE Computer Society, 2006, p.265-273.
- REBAUNDENGO, M. et al. Soft-error detection through *software* fault-tolerance techniques. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 14., DFT1999, 1999, Albuquerque, USA. **Proceedings...** Washington, USA: IEEE Computer Society, 1999, p. 210-218.
- REIS, G. et al. SWIFT: *software* implemented fault tolerance. In SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, San Francisco, USA: **Proceedings...** Washington, USA: IEEE Computer Society, 2005, p. 243-254.
- ROSSI, D. et al. Multiple transient faults in logic: an issue for next generation ICs? In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 20., DFT 2005, 2005, Monterey, USA. **Proceedings...** Los Alamitos, USA: IEEE Computer Society, 2005, p. 352-360.
- VELAZCO, R.; FOUILLAT, P.; REIS, R. **Radiation effects on embedded systems**. [S.l.: Springer], June 2007.
- VEMU, R.; ABRAHAM, J. A. CEDA: Control-flow Error Detection through Assertions. In: IEEE International On-Line Testing Symposium. **Proceedings...** Los Alamitos, USA : IEEE Computer Society, 2006, p. 6-11.
- ZIEGLER, J. F., **Terrestrial cosmic rays**. IBM Journal of Research and Development, v.40 n.1, p.19-39, Jan. 199.

ANEXO A - ARQUIVO DE CONFIGURAÇÃO DO INJETOR DE FALHAS

- Definition file - MRFI V3.7 -

Definições de FALHAS

#Number of faults to be injected

10

#Fault Duration (ns)

10.0

####

Definições do MICROPROCESSADOR

#Signals List Filename

ALL.txt

#Correctness signal | mem (path)

/TB_minimips/u_ram/ram_xilinx/u0/sp_primitive/mem

#Detection Signal Path (if not need detection, fill with NODETECT)

/TB_minimips/error

#Value of Detection on Detection Signal (binary)

1

####

Definições da APLICAÇÃO

#Runtime (ns)

1400.0

#Start address of results on memory (leave it zero if no mem)

0

#Final Address of Results on Memory (leave it zero if no mem)

0

####

Definições do INJETOR

#Injection Time Interval Start(ns)

0.0

#Injection Time Interval End (ns)

1400

#ModelSim Path

```
C:/Modeltech_xe_starter/win32xoem/  
#Project Name  
MRFIV3.7 - DF Debugging  
#Project Path  
C:/Users/Rodrigo/Desktop/PODER_CLEAN_FINAL/src/  
#Signal Hierarchy  
src.txt  
#Testbench  
TB_minimips  
#Report Type (STD | FULL)  
FULL  
#E-mail results to (leave it NONE to disable)  
NONE  
#Give results after X faults  
5  
####
```

ANEXO B - ARTIGOS PUBLICADOS

Como anexo estão, respectivamente, os artigos publicados no Latin American Symposium on Circuits and Systems (LASCAS), Latin-American Test Workshop (LATW) e European Conference on Radiation and Its Effects on Components and Systems (RADECS), no ano de 2010, apresentando maiores detalhes sobre a metodologia e técnicas apresentadas neste trabalho e suas aplicações.

Evaluating the Efficiency of Software-only Techniques to Detect SEU and SET in Microprocessors

José Rodrigo Azambuja, Fernando Sousa, Lucas Rosa, Fernanda Lima Kastensmidt, *Member, IEEE*
Universidade Federal do Rio Grande do Sul (UFRGS) - Instituto de Informática
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{jrfazambuja, faacsousa, lucas.rosa, fglima} @ inf.ufrgs.br

Abstract— This paper presents a detailed evaluation of the efficiency of software-only techniques to mitigate SEU and SET in microprocessors. A set of well-known rules is presented and implemented automatically to transform an unprotected program into a hardened one. SEU and SET are injected in all sensitive areas of a MIPS-based microprocessor architecture. The efficiency of each rule and a combination of them are tested. Experimental results show the inefficiency of the control-flow techniques in detecting the majority of SEU and SET faults. Three effects of the non-detected faults are explained. The conclusions can lead designers in developing more efficient techniques to detect these types of faults.

Index Terms — Control flow signatures, Fault tolerance, SEU, SET, Soft errors, Software techniques.

I. INTRODUCTION

THE last-decade advances in the semiconductor industry have increased microprocessor performance exponentially. Most of this performance gain is due to smaller dimensions and low voltage transistors. However, the same technology that made possible all this progress also lowered the transistor reliability by reducing threshold voltage and tightening the noise margins [1, 2] and thus making them more susceptible to faults caused by energized particles [3]. As a consequence, high reliable applications demand fault-tolerant techniques capable of recovering the system from a fault with minimum implementation and performance overhead.

One of the major concerns is known as *soft error*, which is defined as a transient effect fault provoked by the interaction of energized particles with the PN junction in the silicon. This upset temporally charges or discharges nodes of the circuit, generating transient voltage pulses that can be interpreted as internal signals, thus provoking an erroneous result [4]. The most typical errors concerning soft errors are *single event upsets* (SEU), which are bit-flips in the sequential logic and *single event transients* (SET), which are transient voltage pulses in the combinatorial logic that can be registered by the sequential logic.

In areas where computer-based dependable systems are being introduced, the cost and development time are often major concerns. In such areas, highly efficient systems called systems-on-chip (SoC) are being used. SoC's are often designed using intellectual property (IP) cores and commercial off-the-shelf (COTS) microprocessors, which are only guaranteed to function correctly in normal environmental

characteristics, while their behavior in the presence of soft errors is not guaranteed. Therefore, it is up to designers to harden their systems against soft errors. Fault tolerance by means of software techniques has been receiving a lot of attention on those systems, because they do not need any customization of the hardware.

Software implemented hardware fault tolerance (SIHFT) techniques exploit information, instruction and time redundancy to detect and even correct errors during the program flow. All these techniques use additional instructions in the code area to either recompute instructions or store and check suitable information in hardware structures. In the past years, tools have been implemented to automatically inject such instructions into C or assembly code, reducing significantly the costs.

Nevertheless, the drawbacks of software only techniques are the impossibility to achieve complete fault coverage [5], usual high overhead in memory and degradation in performance. Memory increases due to the additional instructions and often memory duplication, while the performance degradation comes from the execution of redundant instruction [6, 7, 8].

In this paper, the authors implemented a set of software-only techniques to harden a matrix multiplication algorithm in order to point out the main vulnerable areas that are not mitigated by these techniques, more specifically the ones affecting the control-flow. Results can guide designers to improve efficiency and detection rates of soft errors mitigation techniques based on software.

The paper is organized as follows: Section 2 presents related works on the area of software only techniques. Section 3 presents the software rules implemented and a tool that implements the automatic injection of such rules. Section 4 presents the fault injection campaign and results. Section 5 concludes the paper and presents future work.

II. THE PROPOSED CASE-STUDY HARDENED PROGRAM METHODOLOGY

A set of transformation rules has been proposed in the literature. In [9], eight rules are proposed, divided in two groups: (1) aiming data-flow errors, such as data instruction replication [9, 10] and (2) aiming control-flow errors, such as Structural Integrity Checking [11], Control-Flow Checking by Software Signatures (CFCSS) [12], Control Flow Checking using Assertions (CCA) [13] and Enhanced Control Flow

Checking using Assertions (ECCA) [14]. The proposed techniques could achieve a full data-flow tolerance, concerning SEU's, being able to detect every fault affecting the data memory, which would lead the system to a wrong result. On the other hand, the control-flow techniques have not yet achieved full fault tolerance.

Most control-flow techniques divide the program into basic blocks by starting them in jump destination addresses and memory positions after branch instructions. The end of a basic block is on every jump instruction address and on the last instruction of the code.

ECCA extends CCA and is capable of detecting all the inter-BB control flow errors, but is neither able to detect intra-BB errors, nor faults that cause incorrect decision on a conditional branch. CFCSS is not able to detect errors if multiple BBs share the same BB destination address. In [15], several code transformation rules are presented, from variable and operation duplication to consistency checks.

Transformation rules have been proposed in the literature aiming to detect both data and control-flow errors. In [9], eight rules are proposed, while [16] used thirteen rules to harden a program. In this paper, we address six rules, divided into faults affecting the datapath and the controlpath.

A. Errors in the Datapath

This group of rules aims at detecting the faults affecting the data, which comprises the whole path between memory elements, for example, the path between a variable stored in the memory, through the ALU, to the register bank. Every fault affecting these paths, as faults affecting the register bank or the memory should be protected with the following rules:

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

Figure 1 illustrates the application of these rules to a program with 3 instructions. Instructions 1, 3, 7 and 8 are inserted due to rule #3, while instruction 3, 6 and 10 are inserted due to rules #1 and #2.

ld r1, [r4]	1: bne r4, r4', error 2: ld r1, [r4] 3: ld r1, [r4 + offset]
add r1, r2, 1	4: bne r2, r2', error 5: add r1, r3, 1 6: add r1', r3, 1
st [r1], r2	7: bne r1, r1', error 8: bne r2, r2', error 9: st [r1], r2 10: st [r1 + offset], r2

Figure 1: datapath rules

Combined, these techniques duplicate the used data size, such as number of registers and memory addresses and, therefore, the microprocessor must have spare registers and the memory must have spare memory positions. This issue can

also be solved by setting the compiler options to restrict the program to a given number of registers and restrict the data section.

B. Errors in the Controlpath

This second group of rules aims at protecting the program's flow. Faults affecting the controlpath usually cause erroneous jumps, such as an incorrect jump address or a bitflip in a non-jump instruction's opcode which becomes a jump instruction. To detect these errors, three rules are used in this paper:

- Rule #4: every branch instruction is replicated on both destination addresses.
- Rule #5: an unique identifier is associated to each basic block in the code;
- Rule #6: At the beginning of each basic block, a global variable is assigned with its unique identifier. On the end of the basic block, the unique identifier is checked with the global variable.

Branch instructions are more difficult to duplicate than non-branch instructions, since they have two possible paths, when the branch condition is true or false. When the condition is false, the branch can be simply replicated and added right after the original branch, because the injected instruction will be executed after the branch. When the condition is taken, the duplicated branch instruction must be inverted and inserted on the branch taken address.

Figure 2 illustrates rule #4 applied to a simple program. For the branch if equal (BEQ) instruction, instructions 2, 4 and 5 must be inserted to replicate it, where instruction 5 is the inverted branch (branch if not equal). Instruction 4 is necessary to avoid false error alerts.

beq r1, r2, 6	1: beq r1, r2, 5 2: beq r1,r2, error
add r2, r3, 1	3: add r2, r3, 1
	4: jmp 6 5: bne r1,r2, error
add r2, r3, 9 jmp end	6: add r2, r3, 9 7: jmp end

Figure 2: rule #4

beq r1, r2, 6	1: beq r1, r2, 6
add r2, r3, 1	2: mv rX, signature 1 3: add r2, r3, 1 4: bne rX, signature 1, error
add r2, r3, 9 st [r1], r2	5: mv rX, signature 2 6: add r2, r3, 9 7: st [r1], r2 8: bne rX, signature 2, error
jmp end	9: jmp end

Figure 3: rules #5 and #6

The role of rules #5 and #6 is to detect every erroneous jump in the code. They achieve this by inserting a unique identifier to the beginning of each basic block and checking its value on its end. Figure 3 illustrates a program divided in two basic blocks (instructions 2-4 and 5-8). Instructions 2 and 5 are

inserted to set the signature, while instructions 4 and 8 are inserted to compare the signatures with the global value.

III. FAULT INJECTION EXPERIMENTAL RESULTS

The chosen case-study microprocessor is a five-stage pipeline microprocessor based on the MIPS architecture, but with a reduced instruction set. The miniMIPS microprocessor is described in [17]. In order to evaluate both the effectiveness and the feasibility of the presented approaches, an application based on 6x6 matrix multiplication algorithm is used.

A tool called PosCompiler was developed to automate the software transformation. The tool receives as input the program's binary code and therefore is compiler and language independent and is capable of implementing the presented rules, divided in 3 groups. The first group, called variables, implements rules #1, #2 and #3; group 2, called inverted branches, implements rule #4 and, finally, group 3, also known as signatures, implements rules #5 and #6. The user is allowed to combine the techniques in a graphical interface.

We generated through PosCompiler four hardened programs, implementing: (1) signatures, (2) variables, (3) inverted branches and (4) signatures, variables and inverted branches combined. Table 1 shows the original and modified program's execution time, cod size and data size.

Table 1: original and hardened program's characteristics

Source	Original	(1)	(2)	(3)	(4)
Exec. Time (ms)	1.24	1.40	2.55	1.30	2.71
Code Size (byte)	2060	3500	4340	2580	6012
Data Size (byte)	524	532	1048	524	1056

First, thousands of faults were injected in the non-protected microprocessor, one by program execution. At the end of each execution, the results stored in memory were compared with the expected correct values. If the results matched, the fault was discarded. The amount of faults masked by the program is application related and it should not interfere with the analysis.

When 100% signal coverage was achieved and at least 3 faults per signal were detected we normalized the faults, varying from 3 to 5 faults per signal, and those faults build the test case list.

In order to achieve a detailed fault analysis, we sorted the

faults by their source and effect on the system. We defined four groups of fault sources to inject SEU and SET types of faults: datapath, controlpath, register bank and ALU. We assumed the program and data memories are protected by Error Detection and Correction (EDAC) and therefore faults in the memories were not injected.

The fault effects were classified into 2 different groups: program data and program flow, according to the fault effect. To sort the faults among these groups, we continuously compared the Program Counter (PC) of a golden microprocessor with the PC of the faulty microprocessor. In case of a mismatch, the injected fault was classified as flow effect. If the PC matched with the golden's, the fault was classified as a data effect.

When transforming the program, new instructions were added and therefore the time in which the faults were injected changed. Since the injection time is not proportional to the total execution time, we mapped each fault locating the instruction where the fault was injected (by locating its new PC) and pipeline stage where the fault was manifested. Around 1% of the total number of faults could not be mapped and were changed by new faults.

Results show that the technique called variables (2) presented the highest detection rate among the three. It was capable of detecting all the faults injected in the register bank and the faults that caused errors on the data flow. The ALU was not completely protected because it also has control flow signals and some of these signals affected the program's flow. With 110% code size overhead, this technique could detect 77% overall. Technique (3) was able to complement technique (1) by detecting faults on branch instructions, mainly in the ALU, where most of the branch errors were found. By using technique (2) and (3), with 135% in code size overhead, these techniques combined could detect 79% overall.

The signatures (1), on the other hand, were responsible for detecting the faults affecting the program's flow, but it could not reach a high detection rate.

When combining all of them, the fault detection coverage reaches 80% with a code size increase of 192% and execution time increase of 118%. However, 20% of faults remain undetected.

Table 3: (1) signatures, (2) variables, (3) inverted branches and (4) signatures, variables and inverted branches combined.

	Source	# of Signals	Data	Hardened program versions (%)				Flow	Hardened program versions (%)			
				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)
SET	Reg. Bank	2	9	-	100	-	100	1	-	100	-	100
	ALU	10	22	-	100	-	100	21	-	52.3	28.5	80.8
	Controlpath	29	90	-	100	-	100	46	2.17	23.9	2.17	23.9
	Datapath	8	37	-	100	-	100	3	-	100	-	100
	Total	49	158	-	100	-	100	71	1.4	36.7	9.8	45.1
SEU	Reg. Bank	36	18	-	100	-	100	18	-	100	5.5	100
	ALU	2	2	-	100	-	100	0	-	-	-	-
	Controlpath	126	63	-	100	-	100	56	1.8	17.8	1.7	19.6
	Datapath	20	19	-	100	-	100	1	-	-	100	100
	Total	184	102	-	100	-	100	75	1.3	37.3	4	40

IV. ANALYZING THE UNDETECTED FAULTS

Technique (2) presented a high detection rate for data effects and faults injected in the datapath and register bank, while (3) protected the branch instructions and complemented (1). The techniques combined (4) also presented an interesting result, which is that these techniques can be scaled in order to achieve a higher detection rate. On the other hand, the technique (1) presented a detection rate below expected and this result will be analyzed in this session.

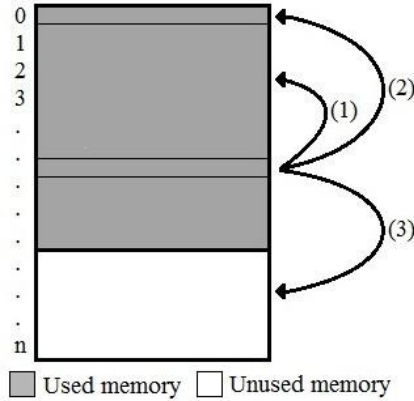


Figure 4: signatures drawbacks

The signatures have three major drawbacks, which caused the low detection rate. The first one is incorrect jumps to the same basic block (intra-block), which cannot be detected since the unique identifier is an invariant and therefore does not depend on the instructions. The application used in this paper spends 83% of its time on the same basic block, which occupies 20% of the program data, and therefore increases the occurrence of this drawback, that can be seen on figure 4, (1).

The second drawback is incorrect jumps to the beginning of a basic block that also cannot be detected, because the global variable containing the unique identifier is updated in the beginning of the basic blocks. The occurrence of such error is proportional to the number of basic blocks per instructions, which is higher in control-flow applications. Also, some microprocessors, such as the used in this paper, have a fault detection mechanism that resets themselves in some cases, such as when an inexistent instruction is fetched. This drawback can be seen on figure 4, (2).

Finally, the last drawback is incorrect jumps to unused memory positions, which are filled with NOP instructions. To correct this drawback, a watchdog with a timer could be used. This drawback can be seen on figure 4, (3).

Table 3: signatures drawbacks

Drawback	Same Basic Block (1)	Beginning of Basic Block (2)	Unused Memory (3)
(%)	37.4	42.2	20.4

Table 3 shows the distribution of undetected faults among the three drawbacks.

V. CONCLUSIONS

In this paper we presented a set of rules based on software-only techniques to detect soft errors in microprocessors. A set of faults was built and a fault injection campaign was realized on the implemented techniques. Results showed that the variables and inverted branches presented a high detection rate, up to 77%, while the signatures showed results below expected. The signatures were then analyzed and three drawbacks were found explaining the undetected faults.

We are currently working on improving the detection rates and decreasing the impact of the drawback on the signatures technique.

REFERENCES

- [1] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.
- [2] T. J. O’Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41–49, January 1996.
- [3] International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, 2005, pp. 6-7.
- [4] P. E. Dodd, L. W. Massengill, “Basic Mechanism and Modeling of Single-Event Upset in Digital Microelectronics”, *IEEE Transactions on Nuclear Science*, vol. 50, 2003, pp. 583-602.
- [5] C. Bolchini, A. Miele, F. Salice, and D. Sciuto. A model of soft error effects in generic ip processors. In *Proc. 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 334–342, 2005.
- [6] Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2003) Soft-error detection using control flow assertions. In: *Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems—DFT 2003*, November 2003, pp 581–588
- [7] Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. *IEEE Trans Comput* 33:518–528 (Dec)
- [8] Oh N, Shirvani PP, McCluskey EJ (2002) Control flow Checking by Software Signatures. *IEEE Trans Reliab* 51(2):111–112 (Mar)
- [9] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 210–218, 1999
- [10] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto. Reliable system specification for self-checking datapaths. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 1278–1283, Washington, DC, USA, 2005. IEEE Computer Society.
- [11] D.J. Lu, “Watchdog processors and structural integrity checking,” *IEEE Trans. on Computers*, Vol. C-31, Issue 7, July 1982, pp. 681-685.
- [12] N. Oh, P.P. Shirvani, and E.J. McCluskey, “Control-flow checking by software signatures,” *IEEE Trans. on Reliability*, Vol. 51, Issue 1, March 2002, pp. 111-122.
- [13] L.D. McFearin and V.S.S. Nair, “Control-flow checking using assertions,” *Proc. of IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05)*, Urbana-Champaign, IL, USA, September 1995.
- [14] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy and J.A. Abraham, “Design and evaluation of system-level checks for on-line control flow error detection,” *IEEE Trans. on Parallel and Distributed Systems*, Vol. 10, Issue 6, June 1999, pp. 627 – 641.
- [15] Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Trans Nucl Sci* 47(6 part 3): 2231–2236 (Dec)
- [16] B. Nicolescu and R. Velazco, “Detecting soft errors by a purely software approach: method, tools and experimental results”, *Proceedings of the Design, Automation and Test Europe Conference and Exhibition*, 2003.
- [17] L. M. O. S. S. Hangout, S. Jan. The minimips project. available online at <http://www.opencores.org/projects.cgi/web/minimips/overview> 2009.

The Limitations of Software Signature and Basic Block Sizing in Soft Error Fault Coverage

José Rodrigo Azambuja, Fernando Sousa, Lucas Rosa, Fernanda Lima Kastensmidt, *Member, IEEE*

Instituto de Informática

Universidade Federal do Rio Grande do Sul (UFRGS)

Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil

{jrfazambuja, faacsousa, lucas.rosa, fglima} @ inf.ufrgs.br

Abstract— This paper presents a detailed analysis of the efficiency of software-only techniques to mitigate SEU and SET in microprocessors. A set of well-known rules is presented and implemented automatically to transform an unprotected program into a hardened one. SEU and SET are injected in all sensitive areas of MIPS-based microprocessor architecture. The efficiency of each rule and a combination of them are tested. Experimental results show the limitations of the control-flow techniques in detecting the majority of SEU and SET faults, even when different basic block sizes are evaluated. A further analysis on the undetected faults with control flow effect is done and five causes are explained. The conclusions can lead designers in developing more efficient techniques to detect these types of faults.

Index Terms — Control flow signatures, Fault tolerance, SEU, SET, Soft errors, Software techniques.

I. INTRODUCTION

THE last-decade advances in the semiconductor industry have increased microprocessor performance exponentially. Most of this performance gain has been due to small dimensions and low voltage transistors, which have led to complex architectures with large parallelism combined with high frequency. However, the same technology that made possible all this progress also has reduced the transistor reliability by reducing threshold voltage and tightening the noise margins [1, 2] and thus making them more susceptible to faults caused by energized particles [3]. As a consequence, high reliable applications demand fault-tolerant techniques capable of recovering the system from a fault with minimum implementation and performance overhead.

One of the major concerns is known as soft error, which is defined as a transient effect fault provoked by the interaction of energized particles with the PN junction in the silicon. This upset temporarily charges or discharges nodes of the circuit, generating transient voltage pulses that can be interpreted as internal signals, thus provoking an erroneous result [4]. The most typical errors concerning soft errors are *single event upsets* (SEU), which are bit-flips in the sequential logic and *single event transients* (SET), which are transient voltage pulses in the combinatorial logic that can be registered by the sequential logic.

Fault tolerance techniques based on software can provide a high flexibility, low development time and cost for computer-based dependable systems. High efficient systems called systems-on-chip (SoC) composed of a large number of microprocessors and others cores connected through a network

on chip (NoC) are getting more popular in many applications that require high reliability, such as data servers, transportation vehicles, satellites and others. When using those systems, it is up to designers to harden their applications against soft errors. Fault tolerance by means of software techniques has been receiving a lot of attention on those systems, because they do not need any customization of the hardware.

Software-only fault tolerance techniques exploit information, instruction and time redundancy to detect and even correct errors during the program flow. All these techniques use additional instructions in the code area to either recompute instructions, or to store and to check suitable information in hardware structures. In the past years, tools have been implemented to automatically inject such instructions into C or assembly code, reducing significantly the costs.

Related works have pointed out the drawbacks of software-only techniques, such as the impossibility of achieving complete fault coverage of SEU [5], usual high overhead in memory and degradation in performance. Memory increases due to the additional instructions and often memory duplication. Performance degradation comes from the execution of redundant instruction [6, 7, 8]. However, there is no study in the literature that analyzes both SEU and SET faults and correlated the fault location and effects to detect or undetected status. This information is important to guide designers to improve efficiency and detection rates of soft errors mitigation techniques based on software.

In this paper, the authors implement a set of software-only techniques to harden a matrix multiplication algorithm to SEU and SET faults that can occur in a miniMIPS microprocessor. Faults were classified by location and effect. The implemented techniques target data and control flow fault tolerance. Results have shown that many faults that lead to control flow error could not be detected by common and well-know control flow fault tolerance based techniques. The effect of basic block sizing combined with software signature was investigated to improve fault coverage. However, software-only based techniques present limitations on fault detection that cannot be solved without considering some hardware characteristics. Results highlight the main vulnerable areas and describe the unresolved issues.

The paper is organized as follows. Section 2 presents the software-based techniques and the proposed methodology.

Section 3 presents the fault injection campaign and results. Section 4 shows the evaluation of block sizing with signature control and the limitations on fault coverage. Section 5 presents main conclusions and future works.

II. THE PROPOSED CASE-STUDY HARDENED PROGRAM METHODOLOGY

A set of transformation rules has been proposed in the literature. In [9], eight rules are proposed, divided in two groups: (1) aiming data-flow errors, such as data instruction replication [9, 10] and (2) aiming control-flow errors, such as Structural Integrity Checking [11], Control-Flow Checking by Software Signatures (CFCSS) [12], Control Flow Checking using Assertions (CCA) [13] and Enhanced Control Flow Checking using Assertions (ECCA) [14]. The proposed techniques can achieve a full data-flow fault tolerance, concerning SEU's, being able to detect every fault affecting the data memory, which lead the system to a wrong result. On the other hand, the control-flow techniques have not yet achieved full fault tolerance.

Most control-flow techniques divide the program into basic blocks by starting them in jump destination addresses and memory positions after branch instructions. The end of a BB is on every jump instruction address and on the last instruction of the code.

ECCA extends CCA and is capable of detecting all the inter-BB control flow errors, but is neither able to detect intra-BB errors, nor faults that cause incorrect decision on a conditional branch. CFCSS is not able to detect errors if multiple BBs share the same BB destination address. In [15], several code transformation rules are presented, from variable and operation duplication to consistency checks.

Transformation rules have been proposed in the literature aiming to detect both data and control-flow errors. In [9], eight rules are proposed, while [16] used thirteen rules to harden a program. In this paper, we address six rules, divided into faults affecting the datapath and the controlpath.

A. Errors in the Datapath

This group of rules aims at detecting the faults affecting the data, which comprises the whole path between memory elements, for example, the path between a variable stored in the memory, through the ALU, to the register bank. Every fault affecting these paths, as faults affecting the register bank or the memory should be protected with the following rules:

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

Figure 1 illustrates the application of these rules to a program with 3 instructions. Instructions 1, 3, 7 and 8 are inserted due to rule #3, while instruction 3, 6 and 10 are inserted due to rules #1 and #2.

ld r1, [r4]	1: bne r4, r4', error 2: ld r1, [r4] 3: ld r1, [r4 + offset]
add r1, r2, 1	4: bne r2, r2', error 5: add r1, r3, 1 6: add r1', r3, 1
st [r1], r2	7: bne r1, r1', error 8: bne r2, r2', error 9: st [r1], r2 10: st [r1 + offset], r2

Figure 1: Datapath rules

These techniques duplicate the used data size, such as number of registers and memory addresses. Consequently, the applications must use a limit portion of the available registers and memory address. The compile can perform this restriction when possible.

B. Errors in the Controlpath

This second group of rules aims at protecting the program's flow. Faults affecting the controlpath usually cause erroneous jumps, such as an incorrect jump address or a bitflip in a non-jump instruction's opcode, which becomes a jump instruction. To detect these errors, three rules are used in this paper:

- Rule #4: every branch instruction is replicated on both destination addresses.
- Rule #5: a unique identifier is associated to each basic block in the code;
- Rule #6: At the beginning of each basic block, a global variable is assigned with its unique identifier. On the end of the basic block, the unique identifier is checked with the global variable.

Branch instructions are more difficult to duplicate than non-branch instructions, since they have two possible paths, when the branch condition is true or false. When the condition is false, the branch can be simply replicated and added right after the original branch, because the injected instruction will be executed after the branch. When the condition is taken, the duplicated branch instruction must be inverted and inserted on the branch taken address.

Figure 2 illustrates rule #4 applied to a simple program. For the branch if equal (BEQ) instruction, instructions 2, 4 and 5 must be inserted to replicate it, where instruction 5 is the inverted branch (branch if not equal). Instruction 4 is necessary to avoid false error alerts.

beq r1, r2, 6	1: beq r1, r2, 5 2: beq r1,r2, error
add r2, r3, 1	3: add r2, r3, 1
	4: jmp 6 5: bne r1,r2, error
add r2, r3, 9 jmp end	6: add r2, r3, 9 7: jmp end

Figure 2: rule #4

The role of rules #5 and #6 is to detect every erroneous jump in the code. They achieve this by inserting a unique identifier to the beginning of each basic block and checking its

value on its end. Figure 3 illustrates a program divided in two basic blocks (instructions 2-4 and 5-8). Instructions 2 and 5 are inserted to set the signature, while instructions 4 and 8 are inserted to compare the signatures with the global value.

beq r1, r2, 6	1: beq r1, r2, 6
add r2, r3, 1	2: mv rX, signature 1 3: add r2, r3, 1 4: bne rX, signature 1, error
add r2, r3, 9 st [r1], r2	5: mv rX, signature 2 6: add r2, r3, 9 7: st [r1], r2 8: bne rX, signature 2, error
jmp end	9: jmp end

Figure 3: rules #5 and #6

C. Hardening Post Compiling Translator Tool

In order to automate the software transformation, we built a tool called *Hardening Post Compiling Translator* (HPC-Translator). Implemented in Java, the HPC-Translator tool is able to automatically transform an unprotected program into a hardened one, by inserting additional instructions and error subroutines to the software.

The HPC-Translator receives as input the program's binary code and therefore is compiler and language independent. The tool is than capable of implementing the presented rules, divided into groups. The first group, called variables, implements rules #1, #2 and #3; the second group, called inverted branches, implements rule #4 and, finally, the third group, also known as signatures, implements rules #5 and #6. The user is allowed to combine the techniques in a graphical interface. The implemented tool outputs a binary code, microprocessor dependent, which can be directly interpreted by the target microprocessor.

III. FAULT INJECTION EXPERIMENTAL RESULTS

The chosen case-study microprocessor is a five-stage pipeline microprocessor based on the MIPS architecture, but with a reduced instruction set. The miniMIPS microprocessor is described in [17]. In order to evaluate both the effectiveness and the feasibility of the presented approaches, an application based on 6x6 matrix multiplication algorithm is used.

Four hardened programs were generated using the *Hardening Post Compiling Translator*, each one implementing the following software-only techniques: (I) signatures, (II) variables, (III) inverted branches and (IV) signatures, variables and inverted branches combined. Table 1 shows the original and modified program's execution time, code size and data size.

First, thousands of faults were injected in all signals of the non-protected microprocessor, one by program execution. The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim XE/III 6.3c [18]. Both types of fault were injected with a duration of one and a half clock cycle. The fault injection campaign is done automatically by a script generation. At the end of each

execution, the results stored in memory were compared with the expected correct values. If the results matched, the fault was discarded. The amount of faults masked by the program is application related and it should not interfere with the analysis. So, only faults not masked by the application were considered in the analysis. When 100% signal coverage was achieved and at least 3 faults per signal were detected we normalized the faults, varying from 3 to 5 faults per signal. Those faults build the test case list.

Table 1: Original and hardened program's characteristics

Source	Original	(I)	(II)	(III)	(IV)
Exec. Time (ms)	1.24	1.40	2.55	1.30	2.71
Code Size (byte)	2060	3500	4340	2580	6012
Data Size (byte)	524	532	1048	524	1056

The faults were classified by their source and effect on the system. We defined four groups of fault sources to inject SEU and SET types of faults: datapath, controlpath, register bank and ALU. Program and data memories are assumed to be protected by Error Detection and Correction (EDAC) and therefore faults in the memories were not injected.

The fault effects were classified into two different groups: program data and program flow, according to the fault effect. To sort the faults among these groups, we continuously compared the Program Counter (PC) of a golden microprocessor with the PC of the faulty microprocessor. In case of a mismatch, the injected fault was classified as flow effect. If the PC matched with the golden's, the fault was classified as a data effect.

Note that used miniMIPS has a fault detection mechanism that resets itself, when a wrong instruction is fetched. So, faults that may change the instruction opcode have a program flow effect.

When transforming the program, new instructions were added and as a result the time in which the faults were injected changed. Since the injection time is not proportional to the total execution time, we mapped each fault locating the instruction where the fault was injected (by locating its new PC) and pipeline stage where the fault was manifested. Around 1% of the total number of faults could not be mapped and were changed by new faults.

Results are presented in table 2. They show that the technique called variables (II) presented the highest detection rate among the three. It was capable of detecting all faults that caused errors on the data flow and in addition some of program flow effects. More specifically, the faults injected in the register bank that affected the program flow could be detected by variables because those data were protected. With 110% code size overhead, this technique could detect 77% of the faults (SEU and SET). Technique (III) could complement very well technique (I) by detecting faults in the ALU, which main effects are on branch instructions. By using technique (II) and (III) combined, 79% of the faults were detected with 135% in code size overhead.

Table 2: Results for SET and SEU fault injection: source and effect classifications, and detection coverage of techniques (I) signatures, (II) variables, (III) inverted branches and (IV) signatures, variables and inverted branches combined

	Source Classification	# of Signals	Data Effect	Hardened program versions (%)				Flow Effect	Hardened program versions (%)			
				(I)	(II)	(III)	(IV)		(I)	(II)	(III)	(IV)
SET	Reg. Bank	2	9	-	100	-	100	1	-	100	-	100
	ALU	10	22	-	100	-	100	21	-	52.3	28.5	80.8
	Controlpath	29	90	-	100	-	100	46	2.17	23.9	2.17	23.9
	Datapath	8	37	-	100	-	100	3	-	100	-	100
	Total	49	158	-	100	-	100	71	1.4	36.7	9.8	45.1
SEU	Reg. Bank	36	18	-	100	-	100	18	-	100	5.5	100
	ALU	2	2	-	100	-	100	0	-	-	-	-
	Controlpath	126	63	-	100	-	100	56	1.8	17.8	1.7	19.6
	Datapath	20	19	-	100	-	100	1	-	-	100	100
	Total	184	102	-	100	-	100	75	1.3	37.3	4	40

The signatures technique (I), on the other hand, was not able to detect a high number of faults affecting the program's flow, around only 1% of the faults were detected. When combining all of them (IV), the overall fault detection coverage reaches 80% with a code size increase of 192% and execution time increase of 118%. However, 20% of faults remain undetected.

IV. ANALYZING BASIC BLOCK SIZING TO IMPROVE FAULT COVERAGE

Although the combined rules called here as signatures, variables and inverted branches are able to detect most errors, they could not detect 20% of the injected faults due to mainly these following reasons:

- Every incorrect jump from a given basic block to the same basic block (also known as intra-block jump) will not be detected, since the unique identifier is an invariant and therefore does not depend on the instructions. The application used in this paper spends 83% of its time on the same basic block, which occupies 20% of the program data and therefore increases the occurrence of this drawback, which can be seen on figure 4 (1);
- Incorrect jumps to the beginning of a different basic block will not be detected, because the global variable containing the unique identifier is updated in the beginning of each basic block. The occurrence of such error is proportional to the number of basic blocks per instructions, which is higher in control-flow applications. Figure 4 (2 and 3) shows this drawback;
- The used microprocessor has a fault detection mechanism that resets itself when an inexistent instruction is fetched or when an interruption is caused. This drawback can be seen on figure 4 (3);
- Incorrect jumps to unused memory positions, which are filled with NOP instructions, result in time out. This drawback can be seen on figure 4 (4).
- Incorrect jumps to branch instructions inside the code will also not be detected, since such instructions are not inside a basic block and therefore not protected by the technique. This drawback can be seen on figure 4 (5).

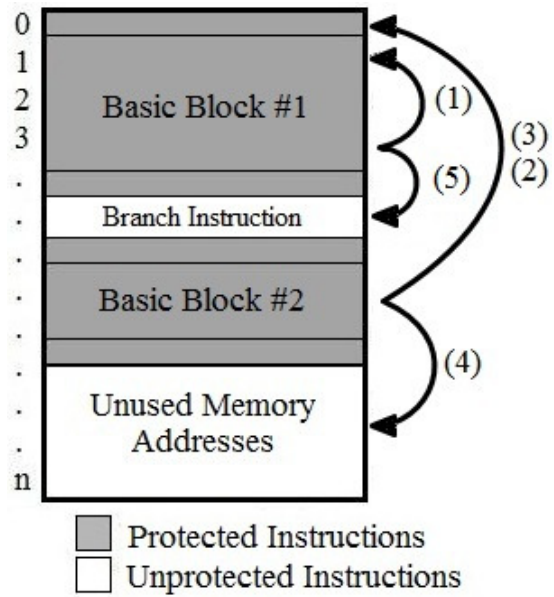


Figure 4: Control flow Effects of undetected faults

Table 3 shows the amount of undetected faults, which effect followed the classification: (1), (2) and (3) and (4) represented in figure 4. On data flow applications, such as the matrix multiplications used as first case-study in this paper, there are only few large basic blocks, which are executed during most of the execution time. In such cases, the effect (1) is more common. On control flow applications, such as a Bubble sort algorithm, the effect (2) are expected to happen more often, since there are more and smaller basic blocks. The use of watch-dog can solve the effect (4), while more complex signatures based on the program's execution flow may cope with the effect (2).

However, the effect (1) is still a big issue. Note that more than 50% of the undetected faults are classified as intra-block jumps and jumps to the beginning of basic blocks. Efficient solutions in terms of execution time to those discussed effects (1), (2) and (3) must be investigated.

Table 3: Effects of the undetected faults responsible to cause control flow errors that were not detected by the signatures software-based technique

Type of Effects: jumps to	Signatures BB – max. size allowed
Same Basic Block (1)	27.2%
Beginning of Basic Block (2)	30%
System Restart (3)	18.4%
Unused Memory (4)	20.4%
Unprotected Instructions (5)	1%
Total	97.2%

As shown in Table 3, 18.4% of the undetected faults caused a system restart due to an exception and 20.4% jumps to an unused part of the program memory. These faults are not a problem because the exception circuit is already able to detect them. From the remaining undetected faults, 27.2% were an incorrect jump to the same basic block, while 30% were an incorrect jump to the beginning of a basic block. From the total faults injected, only 1% caused an incorrect jump to an unprotected branch instruction (5). Finally, the only faults detected by this technique are the faults that caused an incorrect inter-block (from one basic block to another) jump, which represent less than 3% of the total faults injected. This result matches the one presented in table 2.

According to these results, solutions must focus on two effects: Same basic block (1) and Beginning of basic block (2).

A. Undetected faults: jumps to the same basic block

In order to protect this type of faults, one can think on different basic block sizes to reduce the number of address of a same basic block. The basic block maximum size is defined by the branch instructions placed by the software compile. Each basic block must end before a branch instruction and start in both possible destination branch addresses.

An analysis on the original code shows a total of 79 basic blocks and 454 instructions, where the largest basic blocks has 65 instructions, followed by 36 and 12. Note that from the 26% of the faults that jumps to the same basic block, 89.3% were faults in those large basic blocks.

Considering the average basic block size of 5.74, the maximum number of instructions per basic block was set to 4. This number is increased to 7 when the signatures technique is applied (the chosen microprocessor requires 2 assembly instructions to compare a register with a constant). The program characteristics of BB sizing can be seen on table 4.

The execution time of the program protected by signatures in BB sized with 4 instructions is 47% larger than the one protected by signatures with BB sized by the maximum allowed, and 67% larger than the original code. When reducing the basic block size for 4 instructions, one can see that the code size has also increased in 20% compared to the signatures technique without minimal BB sizing.

Table 4: Original, signatures program's overhead characteristics

Source	Original	Signatures BB – max. size allowed	Signatures BB – 4 instruction size
Exec. Time (ms)	1.24	1.40	2.07
Code Size (byte)	2060	3500	4200
Data Size (byte)	524	532	532

In order to inject faults following the same strategy presented in section 3, a new analysis on the effect of each fault was done and a new set of faults was built. The faults were randomly injected on the microprocessor without protection and results were gathered. Faults causing a wrong system result were selected and normalized to each signal (around 5 faults per signal). The faults affecting the data were then excluded, remaining only the faults which effect caused an error on the control flow.

Faults causing a system to restart or to jump to unused memory addresses are easily detected by exception circuits. Those faults are around 39% of the total faults with control flow effect. Such faults should be considered if the system's detection hardware could not detect them. Since these faults were detected by the chosen microprocessor, they will not be considered in the results.

Table 5: Effects of the undetected faults responsible to cause control flow errors that were not detected by the signatures software-based technique when BB is sized

Type of Effects: jumps to	Signatures BB – max. size allowed		Signatures BB – 4- instruction size	
	Total	%	Total	%
Same Basic Block (1)	31	46.3	29	43.9
Beginning of Basic Block (2)	35	52.2	36	54.6
Unprotected Instructions (5)	1	1.5	1	1.5
Total Faults Injected	67	100	66	100

Table 5 shows that when decreasing the maximum size of the basic blocks to 4 instructions, the number of faults causing an erroneous jump to the beginning of basic blocks increased in the same proportion that the faults causing an erroneous jump to the same basic block decreased. That means that a tradeoff between the two types of effects can be achieved, but the detection rate has not been improved by sizing the basic blocks.

B. Undetected faults: jumps to the beginning of basic block

Faults causing an incorrect jump to the beginning of a basic block cannot be detected by the signatures technique, since the extra instructions check only if the last basic block to start is the first to finish. That means that the signatures do not check the control flow itself, but the basic blocks' consistency.

Therefore, a control flow error that maintains the basic block consistency, such as an incorrect jump to the beginning of a basic block or an incorrect path taken by a branch instruction, cannot be detected by the signatures technique.

That issue is partially solved by [14] using the ECCA technique. The ECCA technique introduces a new invariant to each basic block and a two-element queue that keeps track of the current executing basic block and the possible next basic blocks (which have the same identifier in order to fit the two element queue). A few instructions are added to each basic block in order to manage and check the consistency of the queue.

Even increasing the detection rate, ECCA cannot achieve 100% fault coverage in control flow errors, since it cannot guarantee that a possible path was not incorrectly taken, such as an incorrect path taken by a branch instruction. On the other hand, such errors could be detected by the combination of the variables and inverted branch techniques.

In order to detect all incorrect jumps to the beginning of a basic block, a combination of techniques should be implemented, resulting in a higher overhead in program code and execution time.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a set of rules based on software-only techniques to detect soft errors in microprocessors. A set of rules was presented in order to harden usual programs. Then, a tool was implemented to automatically harden binary programs based on presented set of rules. A set of faults was then built and a fault injection campaign was realized on the implemented techniques to evaluate both their effectiveness and feasibility. Results showed that the variables and inverted branches presented a high detection rate, up to 77%, while the signatures showed results below expected, up to 0.5%.

The signatures were then analyzed and some drawbacks were found explaining the undetected faults. In order to further analyze the signatures' undetected faults, a new software implementation was built dividing the basic blocks with more than four instructions and a new set of faults were injected. The results showed that the execution time was 47% bigger, while the detection rate remained the same.

We are currently working on improving the detection rates aiming at the signatures technique and based on the results and decreasing the impact of the drawback on the signatures technique.

REFERENCES

- [1] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.
- [2] T. J. O’Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41–49, January 1996.
- [3] International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, 2005, pp. 6-7.
- [4] P. E. Dodd, L. W. Massengill, “Basic Mechanism and Modeling of Single-Event Upset in Digital Microelectronics”, *IEEE Transactions on Nuclear Science*, vol. 50, 2003, pp. 583-602.
- [5] C. Bolchini, A. Miele, F. Salice, and D. Sciuto. A model of soft error effects in generic ip processors. In *Proc. 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 334–342, 2005.
- [6] Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2003) Soft-error detection using control flow assertions. In: *Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems—DFT 2003*, November 2003, pp 581–588.
- [7] Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. *IEEE Trans Comput* 33:518–528 (Dec).
- [8] Oh N, Shirvani PP, McCluskey EJ (2002) Control flow Checking by Software Signatures. *IEEE Trans Reliab* 51(2):111–112 (Mar).
- [9] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 210–218, 1999.
- [10] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto. Reliable system specification for self-checking datapaths. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 1278–1283, Washington, DC, USA, 2005. IEEE Computer Society.
- [11] D.J. Lu, “Watchdog processors and structural integrity checking,” *IEEE Trans. on Computers*, Vol. C-31, Issue 7, July 1982, pp. 681-685.
- [12] N. Oh, P.P. Shirvani, and E.J. McCluskey, “Control-flow checking by software signatures,” *IEEE Trans. on Reliability*, Vol. 51, Issue 1, March 2002, pp. 111-122.
- [13] L.D. Mcfearin and V.S.S. Nair, “Control-flow checking using assertions,” *Proc. of IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05)*, Urbana-Champaign, IL, USA, September 1995.
- [14] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy and J.A. Abraham, “Design and evaluation of system-level checks for on-line control flow error detection,” *IEEE Trans. on Parallel and Distributed Systems*, Vol. 10, Issue 6, June 1999, pp. 627 – 641.
- [15] Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating na automatic approach for generating safety-critical software with respect to transient errors. *IEEE Trans Nucl Sci* 47(6 part 3): 2231–2236 (Dec).
- [16] B. Nicolescu and R. Velazco, “Detecting soft errors by a purely software approach: method, tools and experimental results”, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.
- [17] L. M. O. S. S. Hangout, S. Jan. The minimips project. available online at <http://www.opencores.org/projects.cgi/web/minimips/overview>, 2009.
- [18] Mentor Graphics, <http://www.model.com/content/modelsim-support>, 2009.

Non-Intrusive Hybrid Signature-Based Technique to Detect SEU and SET Faults in Microprocessors

José Rodrigo Azambuja, Fernando Sousa, Lucas Rosa, Fernanda Lima Kastensmidt, *Member, IEEE*

Instituto de Informática – PPGC – PGMICRO
Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{jrfazambuja, faacsousa, llrosa, fglima} @ inf.ufrgs.br

Abstract— This paper presents a hybrid technique based on software signatures and watchdog processors to detect SEU and SET faults in microprocessors. A light watchdog module is implemented. Results show high detection rates and minor performance degradation and area overhead.

Index Terms — Control flow signatures, Watchdogs, Fault tolerance, SEU, SET, Soft errors.

I. INTRODUCTION

MICROPROCESSORS working in harsh environments can be upset by soft errors. The main effects are Single Event Upset (SEU), which is a bit-flip in the memory elements, and Single Event Transient (SET), which is a transient voltage pulse in the combinational logic. The use of fault tolerance techniques is mandatory to detect or correct these types of faults. Fault tolerance techniques for microprocessors can be based on software or hardware redundancy.

The first one is based on adding instruction redundancy and comparison to detect or correct faults. They can be able to detect faults that affect the data and control flow. Software-based techniques are non-intrusive because no modifications in the hardware of the microprocessor are required. Consequently, they provide high flexibility, low development time and cost. In addition, new generations of microprocessors that do not have RadHard versions in hardware can be used. As a result, aerospace applications can use commercial of the shelf (COTS) microprocessors with RadHard software. However, software-based techniques cannot achieve full system protection against soft errors [1], due to control flow errors [2, 3]. This limitation is due to the inability of the software in protecting all the possible control flow effects that can occur in the microprocessor.

Hardware-based techniques usually change the original architecture by adding logic redundancy, error correcting codes and majority voters. But they can be based on hardware monitoring and in this case are non-intrusive. They exploit special purpose hardware modules, called *watchdog processors* [4], to monitor memory accesses. Watchdogs usually can have access to the data and code memory connections. Since they only have access to the memory, watchdogs do not detect faults that are latent inside the microprocessor, as well faults in the register bank.

Combining software-based techniques and watchdog seems interesting in order to increase the fault detection coverage. In this paper, the authors present a method to detect soft errors in microprocessors using a non-intrusive light watchdog module that works in tandem with the software-based technique. The goal is to increase the coverage of faults with control effect. A new method using hybrid signature is proposed. A fault injection campaign is performed in a MIPS microprocessor running two test-case applications. Experimental results prove both the effectiveness and the feasibility of the proposed technique.

II. THE PROPOSED HYBRID SIGNATURE-BASED TECHNIQUE

A set of software transformation rules has been proposed in the literature in order to protect the system against control-flow errors, such as Structural Integrity Checking [5], Control-Flow Checking by Software Signatures (CFCSS) [6], Control Flow Checking using Assertions (CCA) [7] and Enhanced Control Flow Checking using Assertions (ECCA) [8]. None of the proposed techniques could achieve full fault tolerance against control flow errors due to technique vulnerabilities, faults affecting instruction opcodes and faults inducing the system into a control flow loop.

Most control-flow techniques divide the program into basic blocks by starting them in jump destination addresses and memory positions after branch instructions. The end of a BB is on every jump instruction address and on the last instruction of the code.

In [2], a set of faults were injected in a program divided into basic blocks and with a set of transformation rules applied. Results shown that the technique called variables was capable of detecting 100% of the faults affecting only the data, while the maximum detection rate for faults affecting the control flow was 45%. According to [2], there are 2 main problems not covered by the software-based techniques: incorrect jumps to the beginning of basic blocks and incorrect jumps inside the same basic block.

The proposed approach involves two techniques to protect the system against faults with control flow effect causing incorrect jumps between different basic blocks and inside basic blocks.

A. Jumps to the Beginning of Basic Blocks

In order to protect the system against jumps to the beginning of basic blocks (when the basic block's initialization is performed), a control flow graph is required, once the only option to detect an error is to analyze the source and destiny of the transition between the basic blocks.

Using this technique, every basic block is assigned with two identifiers: the Block Identifier (BID) represents each basic block with a unique prime number and the Control Flow Identifier (CFID) represents the control flow, by storing the multiplication product of the next basic blocks' BIDs. Since the BIDs are composed by prime numbers, the operation rest of division between a CFID and a BID will always return zero, unless a wrong control flow transition has been performed.

CFIDs are stored in a two element queue, initialized with the first basic block's CFID. Upon entry to a basic block, its CFID is enqueued. Upon exit of a basic block, the first CFID is dequeued and divided by the BID. Errors are detected when one of these situations occur: (1) the rest of the division is not zero, (2) overflow in the queue or (3) underflow in the queue.

The queue management is a heavy task to be performed purely in software and would result in a huge memory and performance loss. Therefore, the watchdog should be modified to perform the queue management and the operation rest of division, while the software informs the watchdog of the CFIDs to enqueue the basic blocks' BID.

Figure 1 shows the code transformation required to send the necessary values to the watchdog processor represented by instructions 2 and 7 (send BID), 3 and 8 (enqueue CFID) and 6 and 12 (dequeue CFID). These instructions are implemented in assembly through store instructions with special addresses.

B. Intra-Basic Block Jumps

Incorrect jumps to the same basic block require intra-block control and therefore each instruction within a basic block must be considered. In order to detect such faults, we propose a hybrid technique that computes each basic block signature by XOR'ing all instructions inside a basic block.

beq r1, r2, 9	1: beq r1, r2, 7
	2: reset XOR/send BID
	3: enqueue CFID
add r2, r3, 1	4: add r2, r3, 1
	5: check XOR
	6: dequeue CFID
	7: reset XOR/send BID
	8: enqueue CFID
add r2, r3, 4	9: add r2, r3, 4
st [r1], r2	10: st [r1], r2
	11: check XOR
	12: dequeue CFID
jmp end	13: jmp end

Figure 1: Proposed technique's code transformation

Basic block signatures are pre-computed by the compiler, which also sends these values to the watchdog upon exiting from a basic block. The watchdog is also modified to calculate

the basic block signature by performing XOR operations in real time. When the software sends an instruction with a signature value, the watchdog verifies it with its calculated value. When a mismatch is found, an error is notified. In order to inform the watchdog of an entry in a basic block, a reset instruction should be used.

Figure 1 shows the transformation required on an assembly code to adapt the software to the watchdog with instructions 2 and 7 (reset XOR) and 5 and 11 (check XOR). These instructions are also implemented in assembly through store instructions with special addresses.

III. IMPLEMENTATION

The chosen case-study microprocessor is a five-stage pipeline microprocessor based on the MIPS architecture. The miniMIPS microprocessor is described in [8].

A. Watchdog Hardware

The light watchdog was implemented in VHDL language based on a timer that signals an error if not reset after a given number of clocks. Its enhancement was performed by adding a 16-bit register to store the real-time calculated XOR value, a 64-bit register to store the 2-element queue, a rest of division module and a control module to spoof the instructions described in Section II, such as check CFID and reset XOR value. Table 1 shows the size and performance of the implemented microprocessor and the watchdog. The watchdog implementation has a total of 128 registers and is not protected against faults, but could be protected with a DMR approach.

Table 1: Original and modified architecture characteristics

Source	miniMIPS	Watchdog + miniMIPS
Area (μm)	24261.32	3640.21 (+15%)
Frequency (MHz)	66.7	66.7

In order to find instructions and calculate the XOR value, the watchdog spoofs the address and data bus and the r_w signal. Figure 2 shows the implemented watchdog connected a microprocessor.

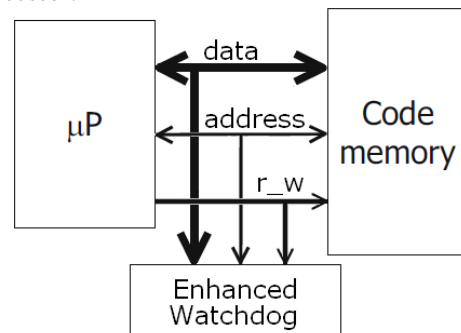


Figure 2: Watchdog connected to microprocessor

B. Hardening Post Compiling Translator Tool

A tool called *Hardening Post Compiling Translator* (HPC-Translator) was implemented to automate the software transformation, since this is a complex process. It receives as input the program's binary code and therefore is compiler and language independent. The tool is then capable of

implementing the necessary instructions and modifying the code in order to control the watchdog. The implemented tool outputs a binary code, microprocessor dependent, which can be directly interpreted by the target microprocessor.

In order to evaluate both the effectiveness and the feasibility of the presented approaches, two applications based on 6x6 matrix multiplication and a bubblesort classification algorithms were chosen to be transformed and adapted to the watchdog processor.

A technique called variables based on [9] was introduced in [3], aiming to protect the variables used by an application. Through instruction replication, this technique duplicates all variables used by the application, both in registers and data memory, duplicates the execution flow and performs consistency checks between variables and their copy when they are read by any instruction. When the values do not match, an error subroutine is performed.

Figure 3 shows the transformation performed when this technique is applied to an unprotected code. Instructions 3, 7, and 11 duplicate the execution flow over the variables' replicas, while instructions 1, 4, 5, 8 and 9 perform the consistency check between the variables (r1, r2, r3 and r4) and their replicas (r1', r2', r3' and r4').

ld r1, [r4]	1: bne r4, r4', error 2: ld r1, [r4] 3: ld r1', [r4' + offset]
add r1, r2, r4	4: bne r2, r2', error 5: bne r4, r4', error 6: add r1, r3, r4 7: add r1', r3', r4'
st [r1], r2	8: bne r1, r1', error 9: bne r2, r2', error 10: st [r1], r2 11: st [r1' + offset], r2'

Figure 3: Variables' code transformation

Three hardened programs were generated using the *Hardening Post Compiling Translator*, each one implementing the following techniques: (I) techniques proposed in session III combined, (II) technique called variables [3] described in figure 3 and (III) techniques I and II combined. Table 2 and 3 show the original and modified program's execution time, code size and data size for the matrix multiplication and bubblesort algorithms, respectively.

Table 2: Matrix multiplication's transformation characteristics

Source	Original	(I)	(II)	(III)
Exec. Time (ms)	1.260	1.665	2.537	2.944
Code Size (byte)	1548	4572	3832	6708
Data Size (byte)	524	528	1048	1052

Table 3: Bubblesort's transformation characteristics

Source	Original	(I)	(II)	(III)
Exec. Time (us)	231	373	443	585
Code Size (byte)	1212	3316	2692	4772
Data Size (byte)	120	124	240	244

IV. FAULT INJECTION EXPERIMENTAL RESULTS

In order to start the fault injection campaign, 50 thousand faults were injected in all signals of the non-protected microprocessor (including registered signals), one by program execution. The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim XE/III 6.3c [10]. SEUs were injected in registered signals, while SETs were injected in combinational signals, both during one and a half clock cycle. The fault injection campaign is performed automatically. At the end of each execution, the results stored in memory were compared with the expected correct values. If the results matched, the fault was discarded. The amount of faults masked by the program is application related and it should not interfere with the analysis. In the end, only faults not masked by the application were considered in the analysis. When 100% signal coverage was achieved and at least 4 faults per signal were detected we normalized the faults, varying from 4 to 5 faults per signal. Those faults build the test case list.

The faults were classified by their source and effect on the system. We defined four groups of fault sources to inject SEU and SET types of faults: datapath, controlpath, register bank and ALU. Program and data memories are assumed to be protected by Error Detection and Correction (EDAC) and therefore faults in the memories were not injected. Faults were also not injected in the watchdog.

The fault effects were classified into two different groups: program data and program flow. To sort the faults among these groups, we continuously compared the Program Counter (PC) of a golden microprocessor with the PC of the faulty microprocessor. In case of a mismatch, the injected fault was classified as flow effect. If the PC matched with the golden's, the fault was classified as a data effect.

When transforming the program, new instructions were added and as a result the time in which the faults were injected changed. Since the injection time is not proportional to the total execution time, each fault was mapped locating the instruction where the fault was injected (by locating its new address) and pipeline stage where the fault was manifested. In some cases, the fault did not manifest itself, due to the stored instructions in the pipeline.

The results presented in table 4 and 5 for the matrix multiplication and bubblesort algorithms, respectively, show that the proposed technique (I) could not achieve full detection over the Flow Effect faults. This is due to faults that affect initially the data, and then lead to a control flow error, such as a fault affecting a conditional branch register. On the other hand the variables technique (II), is capable of detecting such faults.

When combined, the proposed technique and variables (III) achieved full tolerance against control flow errors, in both matrix multiplication and bubblesort algorithms. Another interesting result is that techniques I and II complement themselves, by summing their detection rates, as one can see in Controlpath SET and SEU.

Table 4: SET and SEU fault injection in the Matrix: source and effect classifications and detection coverage of (I) techniques proposed in session III combined, (II) technique called variables proposed in [2] and (III) techniques I and II combined.

	Source Classification	Data Effect Faults	Hardened program versions (%)			Flow Effect Faults	Hardened program versions (%)		
			(I)	(II)	(III)		(I)	(II)	(III)
SET	Reg. Bank	9	-	100	100	1	-	100	100
	ALU	22	9	100	100	10	-	100	100
	Controlpath	59	7	100	100	27	56	44	100
	Datapath	37	3	100	100	2	100	100	100
	Total	127	6	100	100	40	42	92	100
SEU	Reg. Bank	25	-	100	100	13	15	100	100
	ALU	4	-	100	100	0	-	-	-
	Controlpath	54	13	100	100	34	68	44	100
	Datapath	18	-	100	100	5	-	100	100
	Total	101	7	100	100	52	48	71	100

Table 5: SET and SEU fault injection in bubblesort: source and effect classifications and detection coverage of (I) techniques proposed in session III combined, (II) technique called variables proposed in [2] and (III) techniques I and II combined.

	Source Classification	Data Effect Faults	Hardened program versions (%)			Flow Effect Faults	Hardened program versions (%)		
			(I)	(II)	(III)		(I)	(II)	(III)
SET	Reg. Bank	3	67	100	100	4	-	100	100
	ALU	7	100	100	100	14	14	100	100
	Controlpath	18	83	100	100	65	42	69	100
	Datapath	13	69	100	100	24	-	100	100
	Total	41	80	100	100	107	27	81	100
SEU	Reg. Bank	2	-	100	100	33	18	100	100
	ALU	0	-	-	-	0	-	-	-
	Controlpath	19	5	100	100	66	35	73	100
	Datapath	4	-	100	100	18	6	100	100
	Total	25	4	100	100	117	26	85	100

V. CONCLUSIONS AND FUTURE WORK

In this paper, the authors presented two hybrid techniques based on signatures and implemented partially over a watchdog processor. Then, a tool was implemented to automatically harden binary programs and a set of faults was built. A fault injection campaign based on simulation was realized on the implemented techniques to evaluate both their effectiveness and feasibility. Results showed that the variables and proposed techniques combined presented full detection.

We are currently working on improving the detection rates and reducing the overhead of the proposed techniques. Final version of the paper will compare the presented results with fault injection performed on a prototype board when thousands random techniques are injected for a set of benchmarks.

REFERENCES

- [1] C. Bolchini, A. Miele, F. Salice, and D. Sciuto. "A model of soft error effects in generic ip processors". In Proc. of the 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 334–342, 2005.
- [2] J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt, "The limitations of software signature and basic block sizing in soft error fault coverage", IEEE Latin-American Test Workshop, 2010.
- [3] J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt, "Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors", IEEE Latin American Symposium on Circuits and Systems, 2010.
- [4] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors – a survey", IEEE Transactions on Computers, 1988.
- [5] D. J. Lu, "Watchdog processors and structural integrity checking", IEEE Trans. on Computers, Vol. C-31, Issue 7, July 1982, pp. 681-685.
- [6] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-flow checking by software signatures", IEEE Trans. on Reliability, Vol. 51, Issue 1, March 2002, pp. 111-122.
- [7] L.D. Mcfearin and V.S.S. Nair, "Control-flow checking using assertions", Proc. of IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05), Urbana-Champaign, IL, USA, September 1995.
- [8] L. M. O. S. S. Hangout, S. Jan. "The minimips project", 2009, available online at <http://www.opencores.org/projects.cgi/web/minimips/overview>.
- [9] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. "Soft-error detection through software fault-tolerance techniques". In Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 210–218, 1999.
- [10] Mentor Graphics, <http://www.model.com/content/model-support>, 2009.