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

DOUGLAS MACIEL CARDOSO

Eficiência e Custos das Técnicas de Tolerância a Falhas para Proteger Processadores Superescalares de SEEs

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Microeletrônica

Orientador: Prof. Dr. Antonio Carlos Schneider Beck
Co-orientador: Prof. Dr. José Rodrigo Azambuja

Porto Alegre
2019

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Cardoso, Douglas Maciel

Eficiência e Custos das Técnicas de Tolerância a Falhas para Proteger Processadores Superescalares de SEEs / Douglas Maciel Cardoso. – Porto Alegre: PGMICRO da UFRGS, 2019.

116 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2019. Orientador: Antonio Carlos Schneider Beck; Co-orientador: José Rodrigo Azambuja.

1. Efeito de evento singular. 2. Técnicas de tolerância a falhas. 3. Injeção de falhas. 4. Microprocessadores superescalares OoO. I. Beck, Antonio Carlos Schneider. II. Azambuja, José Rodrigo. III. Título.

AGRADECIMENTOS

Primeiramente, não posso deixar de expressar minha profunda gratidão ao meu orientador Antonio Carlos Schneider Beck e ao meu co-orientador José Rodrigo Azambuja, por todas as vezes que sentaram ao meu lado para me agregar conhecimento, pelas palavras de incentivo nos momentos difíceis, e sugestões de trabalho após cada conquista. Agradeço também aos meus colegas de laboratório de sistemas embarcados que me proporcionaram um ambiente de trabalho agradável, em especial ao Rafael Tonetto palas discussões para refletir sobre os resultados. Agradeço à UFRGS e ao Programa de Pós-Graduação em Microeletrônica pela oportunidade de realizar o sonho de obter o grau de mestre. E, não posso deixar de agradecer a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo auxílio financeiro com a bolsa de estudos para realizar este trabalho.

RESUMO

Os avanços tecnológicos reduziram as dimensões dos componentes eletrônicos com o objetivo de diminuir o tempo de execução e a energia consumida para realizarem suas funções. Porém, isto os tornou mais sensíveis a efeitos causados por partículas energizadas presentes no meio. Portanto, os processadores superescalares utilizados em aplicações críticas e em ambientes onde estes efeitos podem causar maiores problemas precisam de uma proteção para garantir a confiabilidade destes dispositivos. Em vista disso, esta dissertação de mestrado estuda a eficiência de técnicas de tolerância a falhas implementadas em software em termos de tempo de execução e capacidade de detecção de falhas. A análise está dividida em técnicas para detecção de falhas nos dados e no fluxo de controle e também foi expandida para a proteção seletiva de registradores. Um conjunto de programas, composto por 13 aplicações, foi protegido com 9 técnicas e executado em 3 versões de um processador superescalar. Para avaliar as técnicas, 130 milhões de falhas foram injetadas, distribuídas em 12 estruturas micro-arquiteturais do processador. Para complementar as técnicas de tolerância a falhas em software, a fim de alcançarmos a total proteção do processador, este trabalho propõe avaliar as estruturas ainda vulneráveis para incluir proteção em hardware, através da duplicação destas estruturas e comparação de seus resultados. Com o intuito de minimizar os custos em área e, consequentemente em energia, este trabalho propõe, também, otimizar a aplicação da duplicação em hardware com o auxílio do algoritmo problema da mochila. Os resultados mostram que as técnicas de tolerância a falhas implementadas em software são capazes de reduzir a vulnerabilidade do processador superescalar em até 69%. Porém, as técnicas em software não são capazes de proteger todo o processador e, consequentemente, o uso de técnicas em hardware é obrigatório para atingir a completa proteção do processador superescalar. Através da proteção seletiva é possível explorar o espaço de protejo disponível para balancear consumo de energia, confiabilidade e desempenho. Os experimentos mostraram que, em alguns casos é possível reduzir custos de energia, mantendo os altos níveis de resiliência dos processadores.

Palavras-chave: Efeito de evento singular. técnicas de tolerância a falhas. injeção de falhas. microprocessadores superescalares OoO.

Efficiency and Costs of Fault Tolerance Techniques to Protect Superscalar Processors from SEEs

ABSTRACT

Technological advances have reduced the dimensions of electronic components to shorten the runtime and energy consumed to perform their functions. However, this made them more sensitive to the effects caused by energized particles present in the environment. Therefore, superscalar processors used in critical applications and in the environments where these effects can cause significant problems needs protection to ensure the reliability of these devices. Given this, this master thesis studies the efficiency of fault tolerance techniques implemented in software in terms of runtime and fault detection capability. The analysis is divided into techniques for detecting data and control-flow faults and has also been expanded to selective register protection. A set of programs, made up of 13 applications, was protected with 9 techniques and executed on 3 versions of a superscalar processor. To evaluate the techniques, 130 million faults were injected, distributed in 12 processor micro-architectural structures. To complement software fault tolerance techniques to achieve full processor protection, this work proposes to evaluate the still vulnerable structures, including hardware protection by duplicating these structures and comparing their results. To minimize costs in the area and, consequently, in energy, this work also proposes to optimize the application of hardware duplication with the aid of the knapsack problem algorithm. Results show that software-implemented fault tolerance techniques can reduce superscalar processor vulnerability by up to 69%. However, software techniques are not capable of protecting the entire processor and, consequently, the use of hardware techniques is mandatory to achieve full protection of superscalar processors. Through selective protection, it is possible to exploit the available design space to balance energy consumption, reliability, and performance. Experiments have shown that in some cases it is possible to reduce energy costs while maintaining high levels of processor resiliency.

Keywords: Single event effect, fault tolerance techniques, fault injection, OoO superscalar microprocessors.

LISTA DE ABREVIATURAS E SIGLAS

ACCE	Automatic Correction of Control-flow Errors
ACCED	Automatic Correction of Control-flow Errors with Duplication
ACFC	Assertions for Control Flow Checking
ALU	Arithmetic-Logic Units
BB	Basic Block
BFI	Branch Free Interval
BID	Block Identifier
BPD	Backing Predictor
BRA	Inverted branches
BROB	Branch Reorder Buffer
BTB	Branch Target Buffer
CCA	Control-flow Checking using Assertions
CEDA	Control-flow Error Detection using Assertions
CFCSS	Control Flow Checking by Software Signatures
CFE	Control-Flow Error
CFG	Control-Flow Graph
CFID	Control-Flow Identifier
CFT	Configurable Fault Tolerant
Chisel	Constructing Hardware in Scala Embedded Language
CSR	Control/Status Registers
DDG	Dynamic Dependence Graph
DMR	Dual Modular Redundancy
DVFS	Dynamic Voltage Frequency Scaling
ECC	Error Correction Code

ECCA	Enhanced Control-flow Checking using Assertions
EDAC	Error Detection and Correction
EU	Execute Unit
EXE	Execution stage
FID	Function Identifier
FPU	Floating-Point Unit
FU	Fetch Unit
GPU	Graphics Processing Unit
GSR	General Signature Register
HDL	Hardware Description Language
HETA	Hybrid Error-Detection Technique Using Assertions
IC	Integrated Circuit
ILP	Instruction-level parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
iTLB	instruction Translation-Lookaside Buffer
IU	Issue Unit
KSP	KnapSack Problem
LSU	Load-Store Unit
MBU	Multi-Bit Upset
MMR	Multiple Modular Redundancy
NES	Node Exit Signature
NIS	Node Ingress Signature
NT	Node Type
OoO	Out-of-Order
PC	Program Counter

RAS	Return Address Stack
RAW	Read-After-Write
PRF	Physical Register File
ROB	Reorder Buffer
RTL	Register Transfer Level
RTS	Real Time Signature
SETA	Software-only Error-detection Technique using Assertions
SFI	Statistical Fault Injection
SIG	Signatures
SIHFD	Software-Implemented Hardware Fault Detection
SIHFT	Software-Implemented Hardware Fault Tolerance
SWIFT	Software Implemented Fault Tolerance
TID	Total Ionizing Dose
TMR	Triple modular redundancy
VAR	Variables
VLSI	Very Large Scale Integration
YACCA	Yet Another Control-Flow Checking using Assertions

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A crescente demanda por aplicações que necessitam de computação cada vez mais complicada exige avanços tecnológicos para reduzir o tempo de execução das funcionalidades nos dispositivos eletrônicos. Para atingir o desempenho esperado, os transistores foram encolhendo até atingir dimensões nanométricas, onde a espessura do óxido de silício (SiO_2) tem apenas alguns átomos (SHI et al., 2016). Porém, este progresso em VLSI (Very Large Scale Integration) alcançou dimensões que comprometem a confiabilidade dos circuitos integrados (Integrated Circuit - IC), visto que reduzir o tamanho dos transistores para fabricar dispositivos mais densos e aumentar sua frequência de operação torna-os mais susceptíveis a falhas (DODD et al., 2010).

As falhas nos ICs modernos podem ser provocadas devido ao acoplamento capacitivo, vazamentos de corrente, ruídos na fonte de alimentação, radiação ionizante gerada por partículas subatômicas, entre outras causas (WANG et al., 2004). Como a tensão necessária para o chaveamento do transistor diminuiu, as capacitâncias internas e as margens de ruído também foram reduzidas, os circuitos ficaram mais sensíveis aos danos causados pela radiação (DODD et al., 2010). Da mesma forma, técnicas que buscam reduzir a quantidade de calor gerada no chip pela potência dissipada, como o escalonamento dinâmico de tensão e frequência (Dynamic Voltage Frequency Scaling - DVFS) , também torna os microprocessadores mais propensos a erros, reduzindo a confiabilidade destes dispositivos (DIXIT; WOOD, 2011).

Circuitos atuando em foguetes lançados para missões espaciais e em satélites que operam em zonas onde a radiação ionizante é mais comum estão mais vulneráveis. Contudo, a redução nas dimensões dos componentes presentes nos ICs colaborou para o aumento da vulnerabilidade de dispositivos atuando em altitudes cada vez menores, possibilitando que partículas de alta energia afetem a aviação e até mesmo circuitos desenvolvidos para aplicações próximas ao nível do mar (NICOLAIDIS, 2005; FERLET-CAVROIS; MASSENGILL; GOUKER, 2013). A preocupação é ainda maior quando se trata de circuitos atuando em áreas críticas, como carros autoguiados e aplicações médicas, onde manter a execução conforme as especificações é indispensável, e uma simples falha pode resultar em tragédias com perda de vidas.

1.1 Falhas Causadas por Radiação

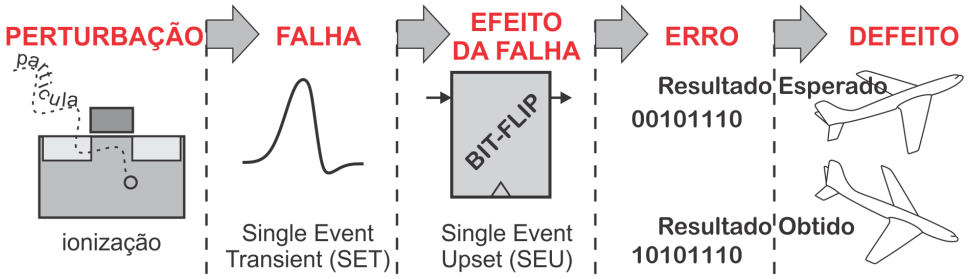
Radiação é uma transferência de energia que pode ocorrer por meio de uma onda eletromagnética ou uma partícula. A radiação é capaz de atravessar a matéria e, quando esta tem energia suficiente para desprender elétrons de átomos ou moléculas, é chamada de radiação ionizante. A ionização de átomos ocorre quando estes adquirem uma carga negativa ou positiva, ganhando ou perdendo elétrons. A ionização pode ser causada diretamente por partículas carregadas, como prótons, alfas (núcleos de hélio) e partículas beta (elétrons ou pósitrons); ou indiretamente, quando a causa são partículas subatômicas sem carga, como nêutrons e ondas eletromagnéticas (raios X e raios gama). Em ambientes espaciais, a radiação é mais comum através de prótons, elétrons e íons pesados; e na atmosfera terrestre é provocada por nêutrons, múons e partículas alfa, que surgem pela interação entre nêutrons e matéria (AZAMBUJA, 2010).

Uma falha pode ser classificada como transiente, permanente ou intermitente. Transiente é uma falha que causa o mau funcionamento temporário do sistema e logo deixa de existir. A falha permanente causa danos que duram até a substituição do componente defeituoso. Intermitente é a falha que dura para sempre, mas só se manifesta ocasionalmente de forma imprevisível, sendo geralmente mais difícil de diagnosticar e prever (CONSTANTINESCU, 2003).

A consequência da falha mais comum em circuitos integrados é o efeito de evento singular (Single Event Effect - SEE), que pode ser classificado como efeito de evento transiente (Single Event Transient - SET) ou perturbação por evento singular (Single Event Upset - SEU). SET ocorre quando algo perturba a tensão de um nó qualquer do circuito, provocando uma oscilação de tensão indesejada, também chamado de *glitch*. A falha do tipo SEU ocorre quando a variação da tensão resulta na inversão do valor lógico armazenado em um elemento de memória, também chamado de *bit-flip*. A Figura 1.1 ilustra tais conceitos quando uma partícula energizada atinge um transistor.

Uma forma de provocar oscilações indesejadas na tensão de um ou mais nós do circuito é através da radiação ionizante. Quando esta radiação passa pelo silício presente nos circuitos, os átomos de silício podem ganhar ou perder carga elétrica, resultando no comportamento anormal do sistema, degradação de integridade ou redução de confiabilidade. Portanto, os erros causados pelas partículas subatômicas devem ser evitados, especialmente em sistemas embarcados para aplicações críticas, onde as falhas podem resultar em vítimas fatais.

Figura 1.1: Exemplo de defeito causado por uma partícula energizada



Fonte: Adaptado de (AZAMBUJA, 2010).

Outra forma das falhas provenientes de efeitos radioativos se manifestarem em ICs é através da dose total ionizante (Total Ionizing Dose - TID) (BARNABY, 2006), resultante do acúmulo de radiação nos circuitos. Neste caso, quando o circuito atinge determinado nível de radiação, medido em *krad(Si)*, este para de funcionar. Porém, esta situação ainda pode ser revertida utilizando o processo de recozimento (*annealing*), onde o circuito é exposto a altas temperaturas para remover a carga acumulada. Já os danos na estrutura cristalina do semicondutor ou danos por deslocamento (Displacement Damage - DD) (SROUR; MARSHALL; MARSHALL, 2003) não podem ser solucionados e, nestas situações, é mais comum a falha ser do tipo permanente, sendo então necessária a substituição do componente com defeito.

Em razão dos custos associados aos procedimentos para resolver os danos provocados por TID e DD, o presente trabalho foca apenas na detecção de erros causados pelos SEEs. Os efeitos SEU e SET podem ser observados em processadores quando os dados gerados não condizem com o esperado ou quando o programa não segue o correto fluxo de execução.

1.2 Tolerância a Falhas em Processadores

A principal forma de garantir o desempenho esperado nos microprocessadores é projetando novas arquiteturas e melhorando suas estruturas organizacionais. Assim surgiram os processadores Superescalares, que buscam a execução paralela de algumas tarefas no nível das instruções (Instruction Level Parallelism - ILP) (SMITH; SOHI, 1995). Entretanto, estes processadores acrescentam uma quantidade considerável de hardware para controle das dependências entre as operações.

Devido a crescente demanda por maior desempenho, estes processadores superescalares se popularizaram e vêm ficando cada vez mais complexos, com a necessidade de agregar ainda mais hardware. Esse *overhead* de hardware, somado à redução das dimensões dos circuitos, agrava a preocupação com a radiação, dado que nessas dimensões uma partícula energizada sozinha é capaz de inverter múltiplos bits (Multi-Bit Upset - MBU) (MANIATAKOS; MICHAEL; MAKRIS, 2012). Isto pode afetar mais de um componente e, consequentemente, a probabilidade de uma falha resultar em um erro ou um defeito também aumenta.

Para aumentar a resiliência destes processadores e evitar ICs ainda mais complexos, com alta densidade de transistores por área, é necessário projetar mecanismos de proteção contra erros para tornar estes componentes tolerantes a falhas sem o acúmulo de hardware. Em vista disto, a utilização de técnicas implementadas puramente em software que não necessitam adicionar hardware, nem mesmo modificar qualquer estrutura de hardware que compõe um processador, é uma alternativa para a redução de custos do sistema final, sendo possível introduzir tais técnicas em processadores comerciais (Commercial Off-The-Shelf - COTS).

A fim de identificar previamente possíveis danos e invalidar operações indesejadas, ao longo dessa evolução tecnológica, foram sendo desenvolvidas técnicas que tentam amenizar o transtorno causado pela radiação. Tais técnicas podem ser desenvolvidas em software, aumentando o tempo de execução das aplicações, ou em hardware, onde é necessário agregar hardware ao circuito e, às vezes, modificar a organização do processador. O uso destas técnicas torna o circuito tolerante a falhas e possibilita reparos do sistema, assim garantindo a correta execução da funcionalidade para a qual o circuito foi projetado.

1.3 Contribuições Deste Trabalho

Entre as técnicas de tolerância a falhas elaboradas por diversos autores, se destacam aquelas que não modificam o hardware. Este é o caso das técnicas implementadas em software que executam instruções extras. Porém, até o presente momento, a eficiência destas técnicas só foi avaliada em arquiteturas de processadores simples e em unidades de processamento gráfico (Graphics Processing Unit - GPU), ou utilizando simulações de alto nível para experimentá-las em processadores mais complexos.

Portanto, a motivação deste trabalho é utilizar uma descrição mais precisa do comportamento de microprocessadores superescalares, detalhados em termos do fluxo de si-

nais no nível de registrador (Register Transfer Level - RTL). Desta forma, é possível simular a injeção de falhas através de *bit-flips* em *flip-flops* do processador e analisar de forma mais aprofundada a eficiência das técnicas de tolerância a falhas implementadas em software, bem como o impacto destas em arquiteturas complexas, como o processador superescalar.

Para tornar nossos resultados mais confiáveis, nós avaliamos diferentes versões de um processador superescalar, quando falhas são aplicadas em diferentes estruturas presentes neste tipo de processador. Assim, pudemos identificar as partes de cada processador mais propensas a erros, bem como o impacto e a eficiência de cada técnica para cada estrutura separadamente. Além disso, fizemos diversas variações das técnicas desenvolvidas em software, bem como a proteção seletiva de registradores lógicos usados pelos programas executados nos processadores.

Os processadores foram sintetizados em 15nm para operar a 2 GHz. A simulação de falhas foi realizada através de bit-flips no RTL dos processadores, onde o flip-flop e o ciclo para injetar a falha foram selecionados aleatoriamente, direcionada a um dos blocos de hardware que compõem a organização do superscalar. Enquanto o processador executa uma aplicação somente um bit-flip é simulado. Este cenário foi repetido para 15000 falhas, com diversas aplicações, protegidas com diferentes técnicas, enquanto são executadas em diferentes configurações do processador.

A fim de proteger os programas com as técnicas implementadas em software, utilizamos uma ferramenta que faz transformações sobre o código dos programas. Para usar esta ferramenta, foi necessário adaptá-la para aplicar as técnicas sobre o conjunto de instruções da arquitetura (Instruction Set Architecture - ISA) RISC-V.

Embora o foco deste trabalho seja as técnicas implementadas em software, neste trabalho também fizemos estimativas dos custos de área e potência, além da cobertura de falhas, utilizado a técnica de redundância modular dupla (Dual modular redundancy - DMR) aplicada a diferentes estruturas do processador. Com o objetivo de minimizar o custo de hardware e manter uma taxa aceitável na detecção de falhas, este trabalho propõe o uso do problema da mochila (KnapSack Problem - KSP) para selecionar de forma eficiente as estruturas que devem ser duplicadas. As técnicas de hardware e software também foram combinadas, buscando o melhor compromisso entre a cobertura de falhas e os custos.

Para avaliar a eficiência das técnicas de tolerância a falhas em processadores superescalares, foram analisadas a taxa de detecção de erros e a vulnerabilidade do processador

antes de depois de cada técnica. Já o impacto das técnicas implementadas em software foi medido através do tempo de execução e quantidade de instruções executadas em cada ciclo (Instructions Per Cycle - IPC), enquanto que o custo das técnicas implementadas em hardware foi observado através do *overhead* de área, além da energia consumida em ambos os casos.

A partir de nossa análise, as técnicas de tolerância a falhas implementadas em software são capazes de melhorar a resiliência dos processadores superescalares em aproximadamente 50%. Porém, alguns módulos de hardware continuam completamente desprotegidos. Além disso, estas técnicas elevam consideravelmente o tempo de execução, e isto se reflete no aumento do tempo em que uma aplicação protegida fica exposta a falhas. Portanto, ao optar pela proteção parcial da aplicação, nós conseguimos reduzir os custos, mantendo níveis de detecção de falhas próximos ao de uma aplicação completamente protegida. Contudo, somente quando combinamos técnicas em software com o DMR seletivo de hardware, foi possível ampliar a capacidade de resiliência dos processadores. Neste caso, o custo de área foi minimizado com a escolha eficiente das estruturas a serem duplicadas.

O restante deste trabalho está dividido nos seguintes capítulos. O capítulo 2 aborda o estado da arte das técnicas de tolerância a falhas propostas por diversos autores. As técnicas utilizadas neste trabalho, a metodologia e as ferramentas utilizadas para realizar as simulações são apresentadas no capítulo 3. O capítulo 4 mostra os resultados. O capítulo 5 finaliza com a conclusão da pesquisa.

2 ESTADO DA ARTE DAS TÉCNICAS DE TOLERÂNCIA A FALHAS

A confiabilidade dos sistemas é uma preocupação desde as fases iniciais de projeto, principalmente em aplicações médicas, veículos autoguiados, além das mais diversas aplicações espaciais onde as partículas subatômicas de alta energia são mais comuns. Para reduzir os riscos das falhas causarem danos aos sistemas, diversas técnicas de tolerância a falhas têm sido propostas. Essas técnicas devem satisfazer três critérios (FRANKLIN, 1995):

- Baixo *overhead* de hardware.
- Mínimo impacto no desempenho.
- Boa cobertura de falhas.

As técnicas de detecção de falhas em processadores de propósito geral podem ser aplicadas no nível de circuito, arquitetural, software transparente ao projetista de software, diretamente no nível da aplicação quando o projetista implementa a técnica, ou envolver uma combinação desses níveis em uma abordagem híbrida (LEE et al., 2011). Estas podem ser implementadas em software ou hardware ou classificadas como híbridas quando hardware e software trabalham juntos. Como o foco deste trabalho são as técnicas desenvolvidas puramente em software, daremos mais destaque aos trabalhos relacionados que propuseram este tipo de técnica.

Cada método de detecção de falhas pode afetar diferentes fatores, como é o caso de performance, área, energia, potência, cobertura de falhas, ou várias destas métricas ao mesmo tempo. Além disso, cada técnica é projetada para detectar diferentes tipos de falhas e, portanto, o projetista deve considerar diversos fatores para selecionar a técnica mais apropriada para proteger seu dispositivo. A fim de garantir o nível de confiabilidade desejado, foram propostos muitos métodos de detecção e outros para correção das falhas.

2.1 Técnicas de Tolerância a Falhas Implementadas em Software

As técnicas desenvolvidas puramente em software são ditas não intrusivas, pois não adicionam hardware nem modificam arquitetura ou organização do processador. Deste modo, elas podem ser aplicadas a processadores já existentes no mercado (commercial off-the-shelf - COTS). Isto é possível com as técnicas implementadas em software que exploram redundância de tempo, porque reexecutam a aplicação ou partes desta sobre o

mesmo hardware.

Porém, somente executar duas vezes a mesma aplicação sobre o mesmo processador pode não solucionar o problema, pois, em caso de discrepância entre os resultados, não é possível identificar qual das duas foi afetada pelo erro, sendo então necessária uma terceira execução. Neste caso, a perda de desempenho seria muito elevada, além de aumentar muito a latência de detecção da falha. Portanto, as técnicas implementadas em software abordadas neste trabalho fazem redundância em um nível mais baixo da aplicação, através de transformações diretamente no código *assembly* mantendo a compatibilidade binária, o que possibilita a detecção da falha e o reparo imediato em tempo de execução com menos custos do que o TMR tradicional. As técnicas de tolerância a falhas implementadas no nível de software também podem ser encontradas na literatura com o acrônimo SIHFT (Software-Implemented Hardware Fault Tolerance) ou SIHFD (Software-Implemented Hardware Fault Detection) (GOLOUBEVA et al., 2005). Estas técnicas proporcionam alta flexibilidade, baixo tempo e custo de desenvolvimento, e portanto são uma boa alternativa para detectar falhas de hardware em processadores embarcados que executam aplicações críticas, onde a detecção da falha deve ser relatada de forma instantânea (OH; MITRA; MCCLUSKEY, 2002).

Estas técnicas podem ser projetadas para detectar erros e proteger a integridade dos dados em um programa (*data-flow*), manter o correto fluxo de execução do programa (*control-flow*), ou técnicas híbridas que protegem tanto o controle quanto os dados. Uma falha no fluxo de dados manifesta-se quando um SEU corrompe alguma variável do programa, acarretando na geração de dados que não correspondem com os esperados. Já a falha no fluxo de controle ocorre quando uma operação de desvio é executada ilegalmente ou de forma indevida.

2.1.1 Detecção de Falhas nos Dados

Para garantir a coerência dos dados utilizados por uma aplicação, as técnicas *data-flow* utilizam o conceito de redundância temporal, isto é, executam o mesmo programa ou partes do programa N vezes sobre o mesmo hardware.

O presente trabalho foca nas técnicas SIHFT implementadas no nível das instruções *assembly*, onde as micro-operações são duplicadas utilizando registradores e posições de memória diferentes das instruções originais, com o propósito de repetir as operações e salvar o mesmo resultado em locais diferentes. Portanto focaremos em trabalhos

que propõem técnicas de tolerância a falhas através de modificações no nível das instruções.

Uma das primeiras técnicas de detecção de falhas projetada puramente em software, que utiliza a redundância de instruções para a proteção dos dados, foi EDDI (Error Detection by Duplicated Instructions) (OH; SHIRVANI; MCCLUSKEY, 2002b). EDDI é uma técnica que duplica todas as instruções do programa, exceto as instruções de salto. Todos registradores e posições de memória são copiados para locais desocupados. Da mesma forma, todas operações *assembly* são duplicadas. Porém, o código duplicado utiliza os registradores e as posições de memória duplicadas. O objetivo é repetir as operações, de modo que operações realizadas sobre algum registrador também sejam realizadas sobre as respectivas cópias, e salvar o resultado de cada operação em um local diferente. As instruções cópias são inseridas imediatamente após as instruções originais. Para detectar uma falha, são inseridas, no mesmo código fonte do programa, instruções que verificam a consistência dos dados, ou seja, instruções que comparam o conteúdo de um registrador com a sua cópia. Estas instruções são inseridas imediatamente antes de qualquer instrução que faça alguma operação de escrita na memória de dados e antes das transferências de controle. É necessário inserir uma instrução de comparação para cada registrador presente na instrução de acesso a memória, desta forma são verificados tanto o conteúdo a ser salvo na memória quanto o endereço de memória, garantindo assim que os valores a serem gravados na memória estejam corretos.

Para avaliar o melhor custo-benefício e aumentar a cobertura de falhas, as técnicas Variáveis 1, 2, e 3 (VAR1, VAR2 e VAR3) (AZAMBUJA, 2010) foram implementadas com base na técnica EDDI, junto com diversas características de outras técnicas presentes na literatura. Cada uma dessas três técnicas segue um conjunto de regras. São chamadas de variáveis porque fazem a cópia de todas as variáveis presentes no programa. A diferença entre estas 3 técnicas está na maneira de fazer a verificação do conteúdo das variáveis originais e suas respectivas variáveis cópias. A técnica VAR1 checa a consistência dos registradores lidos por uma instrução e, portanto, insere instruções para comparar registradores lidos com suas respectivas cópias antes de qualquer instrução que contenha registrador(es) lido(s). Isto é, a integridade de todos registradores lidos por uma instrução é sempre verificada antes desta ser executada. VAR2, por outro lado, compara somente o conteúdo do registrador escritos por uma instrução e, portanto, verifica a consistência dos registradores cujo conteúdo é modificado pela instrução em questão. Neste caso, a comparação do conteúdo do registrador a ser modificado com sua cópia é realizada *após*

a execução das instruções original e cópia. Instruções que não modificam registradores devem seguir a mesma regra da técnica VAR1. Em contrapartida, VAR3 compara apenas o conteúdo dos registradores com suas respectivas cópias *antes* de instruções que realizam acesso à memória de dados. Quando se trata de uma instrução de leitura, somente o registrador lido, isto é, apenas valor utilizado para cálculo de endereço, é verificado. No caso de uma instrução que efetua escrita na memória, todos registradores presentes nesta instrução são verificados. Desta forma, mantém-se a consistência do endereço de memória modificado pela instrução e do conteúdo a ser escrito na memória. A Figura 2.1 mostra um exemplo de aplicação das técnicas VAR1, VAR2 e VAR3, com as instruções inseridas por cada técnica destacadas em negrito.

Figura 2.1: Exemplo de uso das técnicas VAR1, VAR2 e VAR3

Código Original	VAR1	VAR2	VAR3
ld r1, [r4]	***1:*** **bne r4, r4', error** *2:* ld r1, [r4] ***3:*** **ld r1', [r4'+ offset]**	*1:* ld r1, [r4] ***2:*** **ld r1', [r4'+ offset]** ***3:*** **bne r1, r1', error**	***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, r2, r4 ***7:*** **add r1', r2', r4'**	*4:* add r1, r2, r4 ***5:*** **add r1', r2', r4'** ***6:*** **bne r1, r1', error**	*4:* add r1, r2, r4
st [r1], r2	***8:*** **bne r1, r1', error** ***9:*** **bne r2, r2', error** *10:* st [r1], r2 ***11:*** **st [r1'+offset], r2'**	***7:*** **bne r1, r1', error** ***8:*** **bne r2, r2', error** *9:* st [r1], r2 ***10:*** **st [r1'+offset], r2'**	***5:*** **bne r1, r1', error** ***6:*** **bne r2, r2', error** *7:* st [r1], r2 ***8:*** **st [r1'+offset], r2'**

Fonte: Adaptado de (AZAMBUJA, 2010).

Muitas técnicas de redundância implementadas no nível de instruções exigem a duplicação de todos os registradores presentes no programa, porém muitos programas já utilizam a grande maioria dos registradores disponíveis nos processadores, impossibilitando a duplicação de todos os registradores. Nestes casos, a alternativa é proteger apenas os fragmentos mais críticos do código através da redundância seletiva de uma porção de registradores.

Pensando na solução desse problema que o trabalho (CHIELLE et al., 2013) avalia o impacto na cobertura de falhas, na ocupação da memória e no tempo de execução quando somente alguns grupos de registradores são duplicados. Os resultados mostram que a duplicação dos registradores mais utilizados pelo programa, tem maior capacidade de detecção de erros do que a duplicação dos registradores menos utilizados. Porém dupli-

car os registradores que mais aparecem no programa, causam maiores impactos negativos no desempenho. Já a ocupação da memória é sempre a mesma, seja na duplicação dos registradores mais utilizados pelo programa, ou quando os registradores menos utilizados são duplicados. Com isso, os projetistas podem encontrar o melhor custo-benefício quando se trata de redundância seletiva dos registradores.

Devido a grande quantidade de registradores lógicos presentes em cada arquitetura de processador, escolher os melhores registradores para uma proteção seletiva eficiente pode se tornar um processo demorado e exaustivo. Portanto em (ISAZA-GONZÁLEZ et al., 2018) os autores propõem uma métrica baseada em características extraídas de uma análise dinâmica do código fonte, para examinar a eficiência global das técnicas antes de aplicá-las.

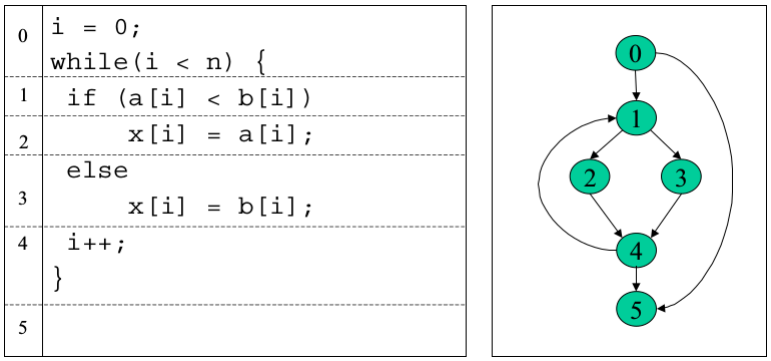
2.1.2 Detecção de Falhas no Controle

Uma falha no fluxo de execução do programa ocorre quando a próxima operação a ser executada pelo hardware desvia da correta ordem de execução das operações. Alguns trabalhos classificam estas falhas como *Control Flow Error* (CFE) (VEMU; ABRAHAM, 2011). As técnicas para detecção de falhas no fluxo de controle são conhecidas como *Control-Flow Checking* (CFC) (SCHUSTER et al., 2017). Muitas dessas técnicas têm sido propostas utilizando um *watchdog*, porém muitos processadores COTS não dispõem deste recurso e, nestes casos, é necessária uma solução não intrusiva implementada completamente em software (VENKATASUBRAMANIAN; HAYES; MURRAY, 2003).

Técnicas implementadas em software para identificar desvios incorretos no fluxo de execução, geralmente utilizam o conceito de blocos básicos (Basic Block - BB) dos programas, e utilizam identificadores únicos para cada BB. Um bloco básico em um programa é um trecho de código que sempre deve ser executado de forma sequencial, sem instruções de desvios e sem o destino de qualquer desvio. Portanto o fluxo de controle correto sempre entra no início do BB e sai no final (GOLOUBEVA et al., 2003), isto é, um trecho de código livre de saltos (Branch Free Interval - BFI) (ALKHALIFA et al., 1999). Desta forma, é possível criar um grafo do fluxo de controle (control-flow graph - CFG), onde os nós são os blocos básicos e os arcos representam o fluxo de controle. A Figura 2.2 esboça um exemplo de um grafo de blocos básicos gerado a partir de um código em linguagem de programação C.

As falhas no fluxo de controle podem ocorrer quando há um salto incorreto para

Figura 2.2: Grafo de Blocos Básicos



Fonte: (GOLOUBEVA et al., 2003).

o início de outro BB, quando ocorre um salto incorreto dentro de um bloco básico para outra parte do mesmo bloco básico ou quando acontece um salto para alguma posição de memória errada. Estes tipos de erros resultam em uma execução mais demorada ou no término precoce do programa. As técnicas para proteção do fluxo de controle implementadas puramente em software deveriam ser capazes de detectar todos estes saltos incorretos, porém a maioria das técnicas propostas foca somente na proteção dos desvios entre diferentes BBs. Consequentemente, a taxa de detecção destas técnicas é mais insatisfatória quando comparadas com técnicas para a detecção de erros de dados e, portanto, geralmente os autores propõem adicionar alguma estrutura de hardware para aumentar a detecção.

As técnicas CCA (Control-flow Checking using Assertions) (MCFEARIN; NAIR, 1998) e ECCA (Enhanced Control-flow Checking using Assertions) (ALKHALIFA et al., 1999) utilizam o conceito de BFIs. Nestes trabalhos, o tamanho de cada BFI é definido com o propósito de obter melhor desempenho e baixo custo. Cada BFI recebe 2 identificadores, que são monitorados durante a execução do programa. O primeiro identificador é o Block Identifier (BID), que deve ser sempre um número primo e único para cada BFI. O segundo é o Control-Flow Identifier (CFID), que é utilizado para representar fluxos de controle, ou seja, transições entre os BFIs. A diferença entre estas duas técnicas é que os identificadores da técnica CCA são criados em tempo de compilação e são sempre os mesmos durante a execução, além disso, a técnica CCA atribui o o mesmo CFID a todos os BFIs imediatamente abaixo de um mesmo BFI. Já os os identificadores da técnica ECCA são do tipo Real Time Signature (RTS), o que significa que estes são modificados durante a execução do programa. Este identificador RTS é calculado por uma instrução OU-exclusivo (XOR) entre a assinatura do BFI com a assinatura do BFI de destino. Com

este método, é possível detectar desvios que deveriam ter tomado um caminho, mas seguiram o caminho incorreto no fluxo de controle. Estas duas técnicas são capazes de detectar desvios incorretos entre BFIs distintos, mas nenhuma consegue detectar desvios errados dentro do mesmo BFI, nem falhas que causam decisão incorreta das instruções de desvio condicional.

Outra técnica, que também utiliza os BBs do programa, é a Control Flow Checking by Software Signatures (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002a). Durante a compilação do programa, é atribuído uma assinatura diferente para cada BB. No início da execução do programa, um registrador global - General Signature Register (GSR) - é inicializado com o valor do primeiro BB. Quando inicia a execução de um novo BB, o GSR é atualizado aplicando-se uma operação XOR entre o valor atual do GSR com uma constante previamente estabelecida, e o resultado dessa operação deve ser o valor de identificação do próximo BB. A checagem da consistência do fluxo de execução é realizada antes de sair de um bloco básico, comparando a assinatura calculada em tempo de execução com a assinatura atribuída ao BB durante a compilação. Esta técnica não é capaz de detectar erros quando vários BBs compartilham o mesmo BB de destino.

A técnica Yet Another Control-Flow Checking using Assertions (YACCA) (GOLOUBEVA et al., 2003) foi proposta com a intenção de reduzir os *overheads* de memória e tempo de execução, mantendo as taxas de detecção de falhas. YACCA também associa um identificador para cada BB durante a compilação do programa. A diferença desta técnica para as anteriores é a forma de gerar os identificadores de controle durante a execução do programa. Em tempo de execução, YACCA utiliza uma variável global que contém a assinatura associada ao nó atual no grafo de fluxo do programa. Duas instruções são inseridas em cada BB, *test* e *set*, onde *test* verifica se o BB a ser executado faz parte do conjunto de BBs filhos do último BB executado, e *set* atualiza a variável global com o valor do identificador do BB que começa a ser executado. Para atualizar a variável global, realiza-se um AND lógico entre a atual variável global e uma máscara que depende das assinaturas dos nós pertencentes ao BB anterior; o resultado então passa por uma XOR com uma máscara relacionada ao identificador do nó atual e dos nós pertencentes ao BB anterior. Os valores das máscaras são definidos de forma que o resultado da próxima variável global seja um identificador válido. Quando comparada com às técnicas ECCA e CFCSS, YACCA ocupa menos memória e é mais rápida do que a técnica ECCA, mas tem *overheads* semelhantes à técnica CFCSS. Quando se trata de taxa de detecção, a técnica YACCA é mais eficiente que as outras duas. Mesmo assim, esta técnica ainda não

consegue detectar todos os erros no fluxo de controle.

Para aumentar a cobertura de falhas da técnica YACCA (GOLOUBEVA et al., 2005), esta foi enriquecida com novas regras, que são aplicadas na descrição de alto nível do programa. Durante a execução de um programa, são utilizadas algumas bibliotecas com funções para quebrar blocos básicos do programa original em mais BBs, e as novas regras servem justamente para proteger estas ramificações. Esta modificação na técnica YACCA permitiu a detecção de quase todas falhas no fluxo de controle, porém a quantidade de memória necessária e, em especial o tempo de execução, aumentaram consideravelmente, já que a nova versão da técnica duplica algumas operações presentes no programa.

A forma de calcular os identificadores em tempo de execução utilizados pela técnica Control-Flow Error Detection Using Assertions (VEMU; ABRAHAM, 2006) (VEMU; ABRAHAM, 2011) (CEDA) melhora o desempenho e a taxa de detecção. O compilador GCC foi modificado para aplicar esta técnica de forma automática sobre o código *assembly*. Nesta técnica, cada BB recebe dois identificadores *d1* e *d2*, que nunca são modificados. Cada BB pode ser do tipo *A* ou *X*: se o BB tiver múltiplos antecessores e pelo menos um de seus antecessores tiver múltiplos sucessores, o BB é do tipo *A*; senão, o BB é do tipo *X*. CEDA também tem um registrador *S* atualizado em tempo de execução no início e no final de cada BB. Quando inicia a execução de um BB; se este é do tipo *A*, então *(S = S AND d1)*; e quando o BB é do tipo *X*, então *(S = S XOR d1)*. Ao final da execução do BB o valor de *S* é novamente atualizado *(S = S XOR d2)*, com isto, o resultado deve ser o identificador do próximo BB. A detecção das falhas é feita por instruções de verificação que comparam *S* com seu valor esperado, inseridas em locais que depende da latência de detecção de erro desejada. Este trabalho também propõe uma métrica para medir a eficiência da técnica, que leva em consideração as falhas não detectadas e a perda de desempenho devido à técnica. CEDA foi comparada com as técnicas YACCA e CFCSS utilizando a métrica proposta, e os resultados indicam que a técnica CEDA tem o melhor custo-benefício.

As técnicas citadas até aqui buscam apenas detectar as falhas. A técnica Automatic Correction of Control-flow Errors (ACCE) (VEMU; GURUMURTHY; ABRAHAM, 2007) é uma extensão da técnica CEDA e foi proposta com o objetivo de detectar e corrigir os CFEs. A intenção é reduzir a penalidade. Então, quando uma falha for detectada, esta técnica repete a execução de apenas algumas instruções, sem a necessidade de reexecutar toda aplicação. Cada função do programa recebe um identificador de função

(Function Identifier - FID) monitorado durante a execução. Com ele é possível identificar qual função estava em execução quando a falha ocorreu. Além das instruções inseridas para a detecção das falhas, foi necessário inserir um trecho de código para cada função, chamado de tratamento de erro da função; e um pedaço de código, chamado de tratamento de erro global. Porém, estes códigos só são executados caso alguma falha seja detectada. O trecho de código que faz o tratamento de erro da função só deve ser executado quando uma falha for detectada pela técnica CEDA, e o trecho de código que faz o tratamento de erro global só pode ser executado quando um dos manipuladores de erro de função o chama. Este trabalho ainda propõe a técnica Automatic Correction of Control-flow Errors with Duplication (ACCED). ACCED usa a técnica ACCE combinada com uma das técnicas de detecção de erros nos dados via duplicação de instruções. A baixa latência para fazer o reparo durante a execução faz desta técnica uma boa opção em sistemas onde a confiabilidade da execução em tempo real é relevante, porém devido ao aumento do código, claramente não deve ser utilizada em sistemas onde o espaço de memória é uma preocupação.

Com a intenção de reduzir os custos e desenvolver uma plataforma portátil, a técnica Assertions for Control Flow Checking (ACFC) (VENKATASUBRAMANIAN; HAYES; MURRAY, 2003) propõe uma abordagem um pouco diferente. Esta técnica não é aplicada sobre o *assembly*, e sim no código C do programa a ser protegido. Um bit extra de paridade é atribuído a cada bloco básico, e a detecção das falhas é feita com base nos erros de paridade.

Para obter melhor cobertura de falhas, (AZAMBUJA et al., 2011b) propõe um conjunto de regras que possibilitam a detecção tanto de saltos para endereços inválidos, quanto alterações em instruções que seriam interpretadas como uma instrução de salto. A técnica BRA (inverted branches) duplica todas as instruções de salto condicional nos dois possíveis destinos desta instrução. Quando o salto não é tomado, a instrução cópia é executada imediatamente da mesma forma que a instrução original, porém o salto da instrução cópia é para uma função que detecta o erro. Quando o salto é tomado, a instrução cópia deve ter um salto condicional com uma lógica invertida da instrução original, de forma que este salto inserido no destino do salto original só possa ser tomado caso algum erro seja detectado. A técnica SIG (signatures) associa um identificador único para cada BB em tempo de compilação. Este identificador é armazenado em uma variável global quando o BB começa a ser executado e, ao final da execução do BB, é realizada uma verificação entre o identificador do BB em questão e a variável global. Os resultados mostram

que, nem mesmo combinadas, as técnicas para garantir fluxo do programa são capazes de atingir níveis aceitáveis de cobertura de falhas.

O trabalho em (AZAMBUJA et al., 2011b) também avalia o impacto na cobertura de falhas e na performance, modificando o tamanho de cada BB. Portanto, nesse caso, a definição de bloco básico passa a ser a quantidade de instruções presentes em cada BB. A conclusão é que, quanto maior o BB, menor o código final, porque há menos instruções para controle dos BBs e, consequentemente, menor é o tempo necessário para execução da aplicação. Reduzir os BBs também diminui as falhas que causam desvio no fluxo dentro do mesmo BB, mas na mesma proporção aumenta falhas que causam saltos incorretos para o início de outro BB (porém, esta variação é bem pequena). A conclusão mais interessante desta análise é que variar tamanho do BB não modifica a taxa de detecção.

2.1.3 Detecção de Ambos Tipos de Falhas (dados+controle)

Para aumentar a taxa de detecção de falhas, tanto nos dados quanto no fluxo de controle, algumas técnicas *data-flow* e *control-flow* implementadas puramente em software são combinadas para formar técnicas híbridas. Devido ao custo para implementar ambas as estratégias para identificar falhas *data-flow* e *control-flow*, a detecção de falhas com estas técnicas são mais caras em termos de performance.

A técnica proposta por (REBAUDENGO et al., 1999) aplica um conjunto de regras de transformação do código C do programa. Parte destas regras são destinadas a detecção das falhas que afetam os dados durante a execução, e outras regras buscam a detecção de desvios no fluxo de execução. As regras propostas que visam a detecção de erros nos dados basicamente duplicam todas variáveis do programa, e qualquer operação sobre uma variável deve também ser realizada sobre sua variável cópia. Antes da execução das duas operações (original e cópia), é realizada uma checagem de consistência das variáveis lidas nestas operações. Estas mesmas regras também são aplicadas sobre qualquer parâmetro utilizado por funções. Para detecção dos CFEs, as primeiras regras de transformação do código seguem a mesma ideia da técnica ECCA. Cada BB é identificado com um valor inteiro *Ki*. Durante a execução de um BB, uma variável global ECF (Execution Check Flag) armazena o *Ki* do BB em execução e, antes de finalizar a execução desse BB, é realizada a verificação para identificar caso tenha ocorrido um salto inválido para outro BB. Outra regra para detecção de erros no controle segue o mesmo conceito da técnica BRA, porém neste caso a duplicação da operação condicional é realizada em linguagem

de alto nível. Desta forma, é possível identificar os erros que se manifestam durante a execução das operações de desvio condicional. A última regra de transformação do código para detectar desvios no fluxo de execução serve para identificar saltos incorretos para o início de sub-rotinas do programa. Esta regra é idêntica as regras aplicadas sobre os BBs, porém, neste caso, não é necessário a divisão do programa em BBs, pois cada sub-rotina é tratada como um BB. Com estas regras, quase todas as falhas foram detectadas, e a latência para detecção da falha é bem baixa, porém o tamanho do código sofre um aumento superior a 7 vezes o tamanho do código original, resultando em uma enorme perda de desempenho.

Em (NICOLESCU; VELAZCO, 2003) é proposto um conjunto de 13 regras para transformação do código em alto nível, que aplicam a técnica para detectar erros tanto nos dados quanto no controle. Primeiro, todas as variáveis presentes no programa são divididas em variáveis temporais, usadas para o cálculo de outras variáveis; e variáveis finais, aquelas que são escritas em memória. Depois, para detecção de erros nos dados, esta técnica aplica de forma semelhante às mesmas regras empregadas por (REBAUDENGO et al., 1999). A diferença é que esta só verifica a consistência entre as variáveis originais e suas respectivas cópias quando ocorre a modificação de uma variável que não é mais utilizada no cálculo de outras variáveis, ou seja, só checa variáveis finais. Para detectar falhas que afetam o fluxo de controle, novamente são utilizados os BBs do programa. Neste caso, além da variável global ECF e do identificador *Ki* para cada BB, também foi atribuído um bit de status para cada BB, que indica quando ele está ativo. No início de cada BB, é realizada a operação *(ECF = (Ki & (status = status + 1)) mod 2)* e, ao final da execução do BB, é feita a verificação da ECF. O último conjunto de regras é idêntico às regras utilizadas por (REBAUDENGO et al., 1999) para duplicação das operações de desvio condicional. Com o emprego destas regras, os custos são menores que os custos da técnica proposta por (REBAUDENGO et al., 1999), porém ainda não é possível obter a completa cobertura das falhas.

A técnica Software Implemented Fault Tolerance (SWIFT) (REIS et al., 2005b) integra as técnicas EDDI e CFCSS. Porém, são realizadas algumas modificações sobre estas técnicas com o objetivo de minimizar os custos. A técnica SWIFT aproveita a proteção dos sistemas de memória modernos como, por exemplo, o código de correção de erro (Error Correction Code - ECC) para reduzir a quantidade de instruções adicionadas ao programa. A modificação na técnica EDDI evita a duplicação de instruções que escrevem na memória e de instruções de salto condicional. Enquanto que a CFCSS faz a

verificação de absolutamente todos os BBs, SWIFT não verifica os BBs que fazem alguma modificação na memória. SWIFT também não faz a verificação de operações de salto, que é realizada pela técnica CFCSS. SWIFT, de fato, é capaz de detectar tanto as falhas nos dados quanto os erros de controle, além de reduzir os custos de memória e performance. Entretanto, sofre uma pequena perda de confiabilidade, além de assumir memórias protegidas com algum outro mecanismo de proteção.

Outra técnica híbrida que integra técnicas de tolerância a falhas Data-Flow e Control-Flow é proposta por (AZAMBUJA et al., 2011a). As técnicas avaliadas destinadas à detecção de falhas nos dados são VAR2 e VAR3, cujo comportamento já fora descrito na seção 2.1.1. Para detectar desvios inválidos no fluxo de execução, foram utilizadas as regras da técnica proposta por (NICOLESCU; VELAZCO, 2003), mas com algumas modificações. Durante a compilação, três identificadores são atribuídos a cada BB, um deles gerado através de uma operação XOR entre todas instruções do BB. Durante execução, um *watchdog* implementado em linguagem de hardware recalcula esta operação XOR enquanto as instruções do BB são executadas e compara com o valor calculado durante a etapa de compilação. A comunicação *software-watchdog* é realizada via instruções que escrevem em endereços de memória específicos. A partir de testes com injeções de SEU e SET aleatórios, concluiu-se que a técnica é capaz de detectar diversos tipos de desvios incorretos, que não eram possíveis de detectar com a técnica proposta por (NICOLESCU; VELAZCO, 2003).

Hybrid Error-Detection Technique Using Assertions (HETA) (AZAMBUJA et al., 2013) detecta erros nos dados através da duplicação dos registradores, posições de memória e das instruções. A checagem da consistência dos dados é realizada da mesma forma que a técnica VAR3. A primeira parte da técnica, destinada a detecção de erros no controle, aplica a técnica BRA. O restante da técnica se baseia na técnica CEDA, onde os BBs são classificados do tipo *A* ou *X*. Assim como na técnica CEDA, um registrador *S* vai sendo transformado através das operações XOR e AND, porém na técnica HETA o BB do tipo *X* só precisa da operação XOR, enquanto que o BB do tipo *A* realiza a operação AND e depois XOR. É necessário um pré-processamento do programa para calcular os identificadores iniciais de cada BB. O valor do identificador *NT* (Node Type) é definido de acordo com os BBs vizinhos, enquanto que *NIS* (Node Ingress Signature) é o valor esperado de *S* no início da execução do BB e *NES* (Node Exit Signature) é o valor esperado para *S* quando a execução do BB é finalizada. *NT* é responsável por detectar saltos incorretos dentro do mesmo BB, enquanto que *NIS* e *NES* servem para detectar

saltos incorretos de um BB para outro. O valor de *S* é armazenado na memória em locais que podem ser definidos pelo usuário, influenciando na latência de detecção da falha. HETA consegue alcançar 100% da cobertura de falhas. Porém, foi necessário adicionar um *watchdog*, além de implementar um modulo de hardware extra que monitora a busca de instruções na memória. O objetivo desse hardware é realizar um XOR dos opcodes das instruções que o processador está acessando e comparar com o valor de *S* quando este é salvo na posição de memória especificada. Com esta técnica, é possível identificar saltos incorretos dentro de um mesmo BB, o que não era possível com as técnicas anteriores.

A técnica Software-only Error-detection Technique using Assertions (SETA) (CHIELLE et al., 2016) foi desenvolvida com base nas técnicas CEDA e HETA. CEDA e HETA têm o objetivo de aumentar a taxa de detecção sem se preocupar com os custos para garantir a alta confiabilidade. Então, SETA foi proposta com o objetivo de manter a taxa de detecção de falhas no fluxo de controle semelhantemente à CEDA, mas reduzindo as penalidades associadas. Para detecção dos erros nos dados, foi realizada uma pesquisa exaustiva para encontrar o melhor custo-benefício da técnica VAR. A duplicação das instruções pode ocorrer de duas formas. A primeira duplica todas instruções, inclusive aquelas que fazem escrita na memória. Entretanto, a cópia das instruções que escrevem na memória devem salvar seu conteúdo em posições de memória diferente das posições utilizadas pela instruções originais. A segunda faz a duplicação de todas instruções, exceto instruções que escrevem na memória, sendo esta mais apropriada quando a memória já está protegida de outra forma. As principais variações da técnica se concentram na forma de verificar a consistência dos dados. A forma de detectar os desvios indesejados no fluxo de execução é baseado na técnica HETA, utilizando os BBs do programa, onde cada BB pode ser do tipo *A* ou *X*, porém SETA não usa nenhum módulo extra de hardware, além de eliminar um dos identificadores. Os outros identificadores continuam sendo calculados em tempo de compilação e analisados durante a execução. Os custos de memória e desempenho causados pela SETA são menores do que em outras técnicas, porém não consegue detectar todas falhas.

2.2 Técnicas de Tolerância a Falhas Implementadas em Hardware

As técnicas de tolerância a falhas desenvolvidas em hardware basicamente replicam ou adicionam módulos de hardware para refazer operações e comparar os resultados. Tais técnicas geralmente utilizam Redundância de Múltiplos Módulos (Multiple Modular

Redundancy - MMR) baseadas na replicação de módulos de hardware em diferentes granularidades, desde transistores até sistemas completos. As mais comuns são redundância modular dupla (Dual modular redundancy - DMR) e redundância modular tripla (Triple modular redundancy - TMR). Neste caso, é necessário adicionar um hardware que verifica a consistência entre os dados nas saídas dos módulos replicados, sendo um comparador no caso do DMR e um votador para o TMR.

Técnicas baseadas em hardware também podem explorar módulos de hardware específicos, como processadores *watchdog* (MAHMOOD; MCCLUSKEY, 1988), usados para monitorar o fluxo de controle dos programas e acessos indevidos à memória principal. Existem dois tipos de processador *watchdog* que podem verificar a consistência no fluxo e controle do programa que está sendo executado no processador principal. Os ativos executam um programa junto com o processador principal e verificam a consistência entre os programas executados pelos processadores. Os passivos não executam nenhum programa, mas observam o barramento do processador principal e realizam verificações de consistência (AZAMBUJA et al., 2013).

Com a intenção de diminuir os custos, as técnicas de tolerância a falhas implementadas em hardware geralmente são aplicadas somente em partes do processador e assumem que os elementos de memória estejam protegidos com ECC ou detecção e correção de erros (Error Detection and Correction - EDAC). De qualquer maneira, estas técnicas não são gratuitas, aumentando a área e a potência e, em alguns casos, também aumentando o tempo de ciclo ou a latência do ciclo e, consequentemente, prejudicando o desempenho. Portanto, técnicas de tolerância a falhas desenvolvidas em hardware podem apresentar boa confiabilidade com baixo custo no desempenho, mas necessitam acréscimo de área e aumentam o consumo de energia (CHIELLE et al., 2016). Outra questão que deixa a desejar nessas técnicas é o fato de não poderem ser aplicadas em processadores COTS, isto é, para usar tais técnicas, é necessário projetar um novo processador, ou adicionar algum hardware específico para detecção das falhas. Além disso, caso haja a necessidade de modificar a técnica existente, é também necessário voltar a etapa de projeto e fabricar um circuito completamente novo, o que acrescenta alto custo financeiro.

Algumas propostas de técnicas para proteção de processadores superescalares são bem conhecidas da literatura, como é o caso da técnica Dynamic Implementation Verification Architecture (DIVA) (AUSTIN, 1999). O processador principal é um superescalar que executa as instruções fora de ordem (Out-of-Order - OoO) e salva os resultados no buffer de reordenamento (Reorder Buffer - ROB). A técnica DIVA insere um pequeno

processador que re-executa as instruções e compara com os seus resultados com os armazenados no buffer para verificar a consistência dos cálculos realizados pelo processador principal. Em caso de detecção de falha, o valor errado é substituído pelo valor correto no buffer de reordenamento e o pipeline do processador principal é esvaziado para reiniciar a execução a partir de um ponto anterior ao da falha. As falhas no fluxo de controle são detectadas com o auxilio de um *watchdog timer*, que monitora se o processador superescalar continua executando corretamente ou se ocorreu alguma situação de travamento. Esta ideia minimiza os custos de um MMR, mas o recálculo, realizado por um pequeno processador, acaba se tornando um gargalo de desempenho.

Em (SMOLENS et al., 2004), foi utilizado um processador com configurações semelhantes ao Alpha EV8 (ou Alpha 21464). Primeiro, os autores fizeram modificações sobre a máquina base para identificar e analisar fatores que afetam o desempenho da execução redundante. Para esta análise, as instruções do programa foram duplicadas de forma dinâmica no estágio de decodificação e a checagem foi feita antes de escrever os dados na memória. Dessa forma, cada programa consumiu praticamente o dobro dos recursos do pipeline do processador, além de duplicar o número de entradas em diversas estruturas. Depois desta avaliação, foi proposta uma microarquitetura tolerante a falhas, buscando o melhor compromisso entre os fatores avaliados e poucas modificações na organização. Os autores nomearam esta microarquitetura de SHared REsource Checker (SHREC), que re-executa as instruções em ordem em um pipeline separado, depois que as instruções já foram executadas no pipeline principal fora de ordem. A fim de minimizar os custos, as instruções executadas no pipeline que faz a checagem usam os mesmos registradores das instruções originais. Semelhante ao DIVA, SHREC também adiciona um verificador que reexecuta instruções em ordem após o processador principal executar as instruções fora de ordem, porém aqui alguns recursos são compartilhados entre os processadores.

Tanto a abordagem utilizada no DIVA quanto no SHREC dependem de um processador que reexecuta as instruções em ordem, o que limita o desempenho do sistema. Ambas as estratégias também assumem que o banco de registradores e as memórias estão protegidas com ECC, além de desconsiderarem falhas nos comparadores. Portanto, muitos tipos de falhas não são avaliados nestes métodos de tolerância a falhas.

Os autores de (WANG et al., 2004) realizaram testes mais precisos com a simulação de programas do SPEC 2000 em um OoO, parecido com Alpha 21264 e com um AMD Athlon, desenvolvido em RTL capaz de executar apenas um conjunto reduzido de

instruções do 21264. Durante a execução, uma falha do tipo SEU é direcionada a um dos dois grupos (somente Latches) ou (Latches e RAM do pipeline), mas não injeta falhas nas caches nem no preditor de saltos. A injeção de falhas nestes grupos foi realizada de forma aleatória no pipeline com um todo, e também falhas foram direcionadas a cada estrutura do processador. Depois de identificar as partes mais sensíveis do processador, os autores inseriram aproximadamente 7% de hardware para detecção de falhas através de: (1) um contador de 100 ciclos para identificar *deadlocks* e limpar o pipeline, (2) 8 bits para ECC no banco de registradores e (3) 3 bits de paridade na instrução quando sua execução é iniciada, com checagem antes de salvar o resultado da instrução. Durante a simulação, o estado de todo o processador que recebeu a falha é comparado com um resultado do processador sem falhas, gerado anteriormente, pelos próximos 10,000 ciclos depois que a falha foi inserida, o que deixa o processo de identificação da falha muito lento.

Para reduzir a perda de desempenho sem acrescentar grande quantidade de hardware, em (NAKKA; PATTABIRAMAN; IYER, 2007) foi utilizada uma proteção seletiva, baseada na redundâncias de tempo e de espaço. A redundância de tempo ocorre porque somente partes críticas do programa são replicadas, em vez de toda aplicação. A redundância de espaço faz a duplicação de apenas algumas partes do pipeline (fetch, rename e commit) de uma arquitetura superescalar. Deste modo, reduzem-se os custos de uma duplicação de todo o processador. A ideia é buscar várias cópias de algumas instruções, renomear seus registradores para executar as cópias de forma independente e compará-las no final do pipeline. A escolha das partes do programa que precisam ser replicadas e o grau de redundância são escolhidos durante a compilação do código, utilizando um grafo que representa as dependências entre as instruções (Dynamic Dependence Graph - DDG).

Em geral, diversas maneiras de proteger microprocessadores tem sido propostas na literatura, porém nenhuma destas técnicas de tolerância a falhas avalia os impactos na dissipação de potência, o consumo de energia nem a escalabilidade das técnicas para diferentes processadores superescalares. Além disso, muitas técnicas implementadas em hardware exigem esforço de projeto para diferentes larguras superescalares. A Tabela 2.1 mostra como algumas das técnicas propostas na literatura foram avaliadas, cada técnica é projetada para detectar diferentes tipos de falhas e cada estratégia tem seu custo, o diferencial deste trabalho se concentra nas colunas Processador e Injeção de Falhas.

Tabela 2.1: Maneira que as técnicas propostas na literatura foram avaliadas

Técnica	Tipo de proteção	Processador	Injeção de Falhas	Custos
EDDI	Dados	Superescalar MIPS R4400 2-issue, Superescalar MIPS R10000 4-issue	SEU na memória de instrução	Desempenho
VAR1,2,3	Dados	miniMIPS	SEU e SET em qualquer sinal	Desempenho
ECCA	Controle	Sun SPARC	SEU na memória de instrução	Desempenho
CFCSS	Controle	MIPS R4400	SEU na memória de instrução	Desempenho
YACCA	Controle	Sparc V8	SEU no processador	Desempenho
CEDA	Controle	In house	Insere, exclui ou modifica Branch	Desempenho
ACCED	Dados e Controle	In house	Insere, exclui ou modifica Branch	Desempenho
SWIFT	Dados e Controle	VLIW Itanium2	SEU em regs arquiteturais	Desempenho
HETA	Dados e Controle	miniMIPS	SEU e SET em qualquer sinal	Desempenho e Área
DIVA	Dados e Controle	SimpleScalar Alpha	Simulação funcional da ISA, apenas avalia CPI	Desempenho e Área
SHREC	Dados	Superescalar Alpha 21464	Não tem, apenas avalia IPC	Desempenho e Área
Este Trabalho SW	Dados e Controle	Superescalar BOOM, 1-, 2-, 4-issue	SEU no RTL do processador	Desempenho
Este Trabalho HW	Dados e Controle	Superescalar BOOM, 1-, 2-, 4-issue	Não tem, apenas avalia custos	Desempenho e Área

Fonte: Elaborada pelo próprio autor.

3 TÉCNICAS IMPLEMENTADAS E METODOLOGIA

Este capítulo estabelece o ambiente de desenvolvimento utilizado durante os experimentos realizados neste trabalho e o comportamento das técnicas de tolerância a falhas implementadas para nossa análise. Para avaliar a capacidade de resiliência dos processadores superescalares, foram analisadas técnicas implementadas em software, no nível de instrução, e técnicas implementadas em hardware, utilizando DMR seletivo nas estruturas dos processadores.

3.1 Técnicas Implementadas em Software

Os métodos de detecção de falhas desenvolvidos puramente em software não necessitaram de qualquer modificação na arquitetura ou na organização do processador. No total, foram avaliadas cinco técnicas para detecção de erros nos dados dos programas, e duas técnicas que têm o objetivo de detectar falhas nas instruções de desvio ou saltos incorretos dentro dos programas.

É importante salientar que as técnicas implementadas em software utilizadas neste trabalho não duplicam instruções que armazenam dados na memória, porque uma proteção com ECC já é suficiente, ou seria necessário duplicar toda a memória de dados, aumentando muito o custo destas técnicas. Portanto, assumimos que a memória esteja protegida pela redundância de informação ECC, presente em memórias modernas. Em contrapartida, este recurso não foi utilizado para proteger estruturas internas dos microprocessadores, como o banco de registradores, pois seria necessário alterar o hardware do processador.

3.1.1 Técnicas para Proteção dos Dados

3.1.1.1 Técnica Variables e suas variantes

A primeira técnica avaliada com o objetivo de detectar dados incorretos foi a técnica VAR (Variables). As regras desta técnica são baseadas em características da técnica VAR1 (AZAMBUJA, 2010). Através de transformações no código *assembly* do programa a ser protegido, esta técnica insere instruções extras para duplicar todo o fluxo de dados, sem alterar a funcionalidade do programa. Para isto, a técnica primeiro copia o conteúdo

de todos os registradores usados pelo programa em registradores livres, isto é, registradores ociosos que não estão sendo utilizados pelo programa em questão. Desta forma, é possível replicar todas as instruções que modificam algum dado do programa com os registradores que não estão sendo utilizados.

A maneira de verificar o conteúdo dos registradores e detectar qualquer inconsistência entre os dados dos registradores originais e os dados armazenados em suas réplicas é inserindo, no mesmo código *assembly*, instruções que comparam estes dados. Estas comparações entre os registradores originais e suas cópias são inseridas antes de todas as instruções onde o registrador é lido, ou seja, quando o registrador é uma fonte de informação. Quando uma falha é detectada, o fluxo do programa é desviado para uma sub-rotina de detecção de erro que sinaliza o erro e interrompe a execução do programa. A Figura 3.1 apresenta um exemplo de uso desta técnica, onde as instruções extras estão destacadas em negrito.

Figura 3.1: Exemplo de uso da técnica VAR

	Código original	**VAR**
1:		**bne r4, r4', error**
2:	ld r2, (r4)	ld r2, (r4)
3:		**ld r2', (r4')**
4:		**bne r2, r2', error**
5:		**bne r7, r7', error**
6:	add r1, r2, r7	add r1, r2, r7
7:		**add r1', r2', r7'**
8:		**bne r1, r1', error**
9:		**bne r3, r3', error**
10:	beq r1, r3, label	beq r1, r3, label
11:		**bne r1, r1', error**
12:		**bne r5, r5', error**
13:	st r1, (r5)	st r1, (r5)

Fonte: Elaborada pelo próprio autor.

O trecho de código *assembly* do programa original é mostrado nas linhas 2, 6, 10 e 13. As instruções adicionadas nas linhas 3 e 7 representam as cópias das instruções 2 e 6, respectivamente. As demais instruções extras foram adicionadas ao programa para comparar os dados armazenados nos registradores originais e suas respectivas cópias. Na instrução 1, o endereço da instrução que busca um dado da memória (*load*) é verificado antes mesmo de executar a instrução, enquanto que, na instrução que armazena dados da memória (*store*), tanto o dado quanto o endereço são verificados através das instruções 11

e 12, respectivamente.

Para instruções aritméticas, todos os registradores fontes são verificados antes de executar a instrução original, como mostrado nas linhas 4 e 5. Porém, esta técnica não verifica o conteúdo armazenado no registrador destino, pois caso não seja detectado qualquer discrepância entre os registradores fontes, dificilmente ocorrerá uma falha no destino desta instrução. Quando se trata de instruções de desvio condicional e incondicional, todos os registradores presentes na instrução são verificados imediatamente antes de executar o salto, como é mostrado nas instruções 8 e 9.

Com o objetivo de reduzir os custos associados à técnica VAR, a próxima técnica avaliada elimina a verificação dos registradores antes de instruções aritméticas, bem como a checagem antes das instruções de controle do fluxo do programa, sejam elas instruções de desvio condicional, salto incondicional ou chamadas de rotinas. Portanto, aqui não é realizada verificação dos registradores em todas as ocasiões em que eles são lidos. Esta estratégia de detecção de falhas mantém a verificação dos dados somente nos registradores presentes em instruções de acesso à memória *load* e *store*. Assim, o nome dessa técnica passa a ser VAR_LS. O exemplo de aplicação desta técnica sobre um trecho de código *assembly* é apresentado na Figura 3.2. Observe que todos os registradores e instruções continuam sendo duplicados, exceto instruções de desvio, para manter a coerência nos dados armazenados em cada registrador.

Figura 3.2: Exemplo de uso da técnica VAR_LS

	Código original	**VAR_LS**
1:		**bne r4, r4', error**
2:	ld r2, (r4)	ld r2, (r4)
3:		**ld r2', (r4')**
4:	add r1, r2, r7	add r1, r2, r7
5:		**add r1', r2', r7'**
6:	beq r1, r3, label	beq r1, r3, label
7:		**bne r1, r1', error**
8:		**bne r5, r5', error**
9:	st r1, (r5)	st r1, (r5)

Fonte: Elaborada pelo próprio autor.

As linhas 2, 4, 6 e 9 exibem as instruções *assembly* do programa original desprotegido, enquanto que as linhas 3 e 5 representam a cópia das instruções 2 e 4, respectivamente. Da mesma forma que a técnica VAR, este segundo método de detecção de falhas

insere uma instrução para comparar o conteúdo armazenado nos registradores utilizados para cálculo de endereço da instrução que carrega dados da memória, com o conteúdo armazenado na réplica deste registrador, mostrado na linha 1. Já as instruções extras, inseridas nas linhas 7 e 8 do programa protegido, servem para verificar o registrador com o conteúdo a ser armazenado na memória e o registrador utilizado para cálculo de endereço da instrução *store*, respectivamente.

A próxima técnica estudada continua replicando todos os registradores, e todas instruções que modificam algum registrador são duplicadas imediatamente após a instrução original. Neste caso, os registradores presentes nas instruções de desvio também são verificados antes de realizar o salto, mas registradores que aparecem em instruções aritméticas não são conferidos antes de executar este tipo de instrução. Portanto, esta estratégia monitora apenas os registradores presentes em instruções *load*, *store* e *branch* e, por este motivo, esta técnica recebe o nome de VAR_LSB. Para exemplificar o uso desta estratégia, a Figura 3.3 mostra o resultado da transformação sobre um trecho código *assembly*.

Figura 3.3: Exemplo de uso da técnica VAR_LSB

	Código original	**VAR_LSB**
1:		**bne r4, r4', error**
2:	ld r2, (r4)	ld r2, (r4)
3:		**ld r2', (r4')**
4:	add r1, r2, r7	add r1, r2, r7
5:		**add r1', r2', r7'**
6:		**bne r1, r1', error**
7:		**bne r3, r3', error**
8:	beq r1, r3, label	beq r1, r3, label
9:		**bne r1, r1', error**
10:		**bne r5, r5', error**
11:	st r1, (r5)	st r1, (r5)

Fonte: Elaborada pelo próprio autor.

Este mecanismo de tolerância a falhas busca menor custo do que a técnica VAR, mas tenta aumentar a taxa de detecção de falhas com relação à técnica VAR_LS. O código fonte original é mostrado nas linhas 2, 4, 8 e 11, e as linhas 3 e 5 utilizam os registradores replicados para duplicar as instruções 2 e 4, respectivamente. As linhas 6 e 7 realizam a verificação de consistência entre os dados dos registradores originais, presentes na instrução de salto da linha 8, e os dados de suas réplicas. As duas instruções de comparação

inseridas nas linhas 9 e 10 verificam o conteúdo dos registradores utilizados pela instrução que armazena dados na memória.

As duas últimas técnicas surgem como uma alternativa às técnicas anteriores. Elas continuam duplicando o conteúdo de todos os registradores, bem como todo o caminho de dados do programa. Porém, o nome dessas técnicas inclui um sufixo "M", porque a réplica das instruções que carregam dados da memória para o processador não é realizada com novas instruções de *load*, e sim com instruções de movimentação de dados entre registradores. Estas instruções que copiam o conteúdo de um registrados para outro são usadas apenas para manter a consistência entre os registradores originais e suas réplicas.

Para verificar a consistência dos conteúdos mantidos nos registradores, as técnicas VARM_LS e VARM_LSB inserem instruções de comparação nos mesmos locais em que são inseridos pelas estratégias VAR_LS e VAR_LSB, respectivamente. Desta forma técnicas com sufixo "LS" fazem a checagem dos registradores presentes nas instruções *load* e *store*, antes que estas sejam executadas, enquanto que as técnicas com sufixo o "LSB" inserem verificações de conteúdo dos registradores antes de operações *load*, *store* ou *branch*. A Figura 3.4 apresenta um trecho de código *assembly* protegido com a técnica VARM_LS.

Figura 3.4: Exemplo de uso da técnica VARM_LS

	Código original	**VARM_LS**
1:		**bne r4, r4', error**
2:	ld r2, (r4)	ld r2, (r4)
3:		**mv r2', r2**
4:	add r1, r2, r7	add r1, r2, r7
5:		**add r1', r2', r7'**
6:	beq r1, r3, label	beq r1, r3, label
7:		**bne r1, r1', error**
8:		**bne r5, r5', error**
9:	st r1, (r5)	st r1, (r5)

Fonte: Elaborada pelo próprio autor.

A Figura 3.5 ilustra a aplicação da técnia VARM_LSB sobre o mesmo trecho de código, com as instruções inseridas pela técnica destacadas em negrito.

Figura 3.5: Exemplo de uso da técnica VARM_LSB

	Código original	VARM_LSB
1:		**bne r4, r4', error**
2:	ld r2, (r4)	ld r2, (r4)
3:		**mv r2', r2**
4:	add r1, r2, r7	add r1, r2, r7
5:		**add r1', r2', r7'**
6:		**bne r1, r1', error**
7:		**bne r3, r3', error**
8:	beq r1, r3, label	beq r1, r3, label
9:		**bne r1, r1', error**
10:		**bne r5, r5', error**
11:	st r1, (r5)	st r1, (r5)

Fonte: Elaborada pelo próprio autor.

3.1.2 Técnicas para Proteção do Controle

As duas técnicas avaliadas neste trabalho designadas a proteção do fluxo de execução e a evitar desvios indesejados no programa são as técnicas BRA e SIG. Estas estratégias foram desenvolvidas por Azambuja (AZAMBUJA, 2010) e testadas com um processador simples que executa instruções em ordem. Nestas técnicas, os registradores não são duplicados.

3.1.2.1 Técnicas BRA e SIG

Assim como as técnicas que protegem a integridade dos dados nos programas, a primeira técnica para detecção de falhas no fluxo de controle também duplica instruções. Neste caso, são duplicadas as instruções de desvios condicionais e, portanto, esta técnica foi nomeada BRA (Branch). Entretanto, duplicar este tipo de instrução requer uma atenção mais cuidadosa, pois estas sempre dividem o fluxo de execução em dois caminhos. Em vista disso, é necessário inserir duas instruções cópias, uma no destino do salto caso o desvio seja tomado, e outra no destino do salto para o caso do desvio não tomado. Para exemplificar o uso desta técnica, observe a Figura 3.6, onde as instruções inseridas pela técnica estão destacadas em negrito.

A instrução de desvio que foi protegida é o *beq* (*branch if equal* - desvia se os dados dos registradores forem iguais) da linha 2. As linhas 3 e 9 são as réplicas da

Figura 3.6: Exemplo de uso da técnica BRA

	Código original	**BRA**
1:	`add r1, r2, r7`	`add r1, r2, r7`
2:	`beq r1, r3, Label1`	`beq r1, r3, Label2`
3:		**`beq r1, r3, error`**
4:	`ld r2, (r4)`	`ld r2, (r4)`
5:	`xor r8, r6, r1`	`xor r8, r6, r1`
6:	`st r1, (r5)`	`st r1, (r5)`
7:		**`j Label1`**
8:		**`Label2:`**
9:		**`bne r1, r3, error`**
10:	`Label1:`	`Label1:`

Fonte: Elaborada pelo próprio autor.

instrução original, mas, ao invés de terem como destino a continuação da execução do programa original (*Label1*), ambas desviam para a sub-rotina de detecção de erro (*error*). Isto porque a linha 3 reexecuta a instrução original da linha 2 e, portanto, em condições normais, jamais realizará o desvio para *error*. A linha 9, por sua vez, deve inverter o desvio lógico original com uma instrução *bne* (*branch if not equal* - desvia se os dados dos registradores não forem iguais), a fim de reexecutar a instrução original e, assim como réplica da linha 3, jamais desviar para *error* em condições corretas de execução.

As demais modificações no código são realizadas para manter a funcionalidade do programa. O destino do desvio original deve ser modificado para saltar para um novo endereço (*Label2*), visto que sua réplica deve ser executada. A instrução da linha 7 foi inserida para garantir que a réplica na linha 9 seja executada só e unicamente após a execução do desvio original na linha 2, impedindo que esta seja executada, por exemplo, após a instrução da linha 6.

Semelhante a maioria das técnicas propostas na literatura destinadas à proteção do fluxo de controle do programa, a técnica SIG (Signatures) utiliza o conceito de Bloco Básico. Esta técnica associa estaticamente uma assinatura única a cada BB durante a fase de compilação do programa e monitora essa assinatura dinamicamente em tempo de execução. A estratégia utilizada segue características apresentadas na técnica CFCSS (OH; SHIRVANI; MCCLUSKEY, 2002a). Para verificar o fluxo de controle da aplicação, quando tem início a execução de um BB, sua assinatura é atribuída a um registrador, e antes de sair do bloco a assinatura é testada, a fim de identificar desvios indesejados entre blocos básicos. A Figura 3.7 mostra um exemplo de aplicação da técnica SIG com as

instruções extras destacadas em negrito.

Figura 3.7: Exemplo de uso da técnica SIG

	Código original	SIG
1:	Label1:	Label1:
2:		**addi gp, zero, signature**
3:	ld r2, (r4)	ld r2, (r4)
4:	add r1, r2, r7	add r1, r2, r7
5:	xor r8, r6, r1	xor r8, r6, r1
6:	st r1, (r5)	st r1, (r5)
7:		**addi tp, zero, signature**
8:		**bne gp, tp, error**
9:	beq r1, r3, Label1	beq r1, r3, Label1

Fonte: Elaborada pelo próprio autor.

A instrução da linha 2 foi adicionada para salvar a assinatura do bloco básico no registrador *gp*. Como a linha 9 é a última instrução do BB, é realizada uma verificação da assinatura do BB corrente. Portanto, a instrução da linha 7 atribui a assinatura do BB que está sendo executado a outro registrador, e a instrução da linha 8 compara os valores dos registradores que devem conter os mesmos valores. Observe que esta técnica não duplica registradores, porém é necessário utilizar dois registradores ociosos para guardar a assinatura dos BBs.

3.1.3 Técnicas para Proteção dos Dados e do Controle

Com o propósito de avaliar a possibilidade de tornar os processadores superescalares completamente tolerantes a falhas, duas técnicas híbridas também foram utilizadas. Estes métodos de detecção de falhas fornecem uma proteção contra falhas que afetam tanto o fluxo de execução do programa quanto os dados do programa. As técnicas híbridas utilizadas neste trabalho são combinações das técnicas *data-flow* e *control-flow* apresentadas anteriormente nas subseções 3.1.1 e 3.1.2, respectivamente.

3.1.3.1 Técnicas VAR_S_BRA e VARM_S_BRA

A primeira estratégia híbrida avaliada foi a técnica VAR_S_BRA. A detecção de falhas que afetam as instruções de desvio é feita pela técnica BRA, pois esta técnica é a

mais barata em termos de energia. A detecção de dados inconsistentes é realizada pela técnica VAR, pois esta técnica é a mais eficiente. Porém, neste caso, optamos por verificar os registradores somente antes de instruções que salvam dados na memória (*store*), a fim de reduzir os custos de desempenho e energia consumida pelas instruções de comparação, já que a técnica BRA também insere instruções deste tipo. A Figura 3.8 exemplifica o uso desta técnica para proteger um trecho de código *assembly*, onde as instruções inseridas pela técnica estão destacadas em negrito.

Figura 3.8: Exemplo de uso da técnica VAR_S_BRA

	Código original	**VAR_S_BRA**
1:	add r1, r2, r7	add r1, r2, r7
2:		**add r1', r2', r7'**
3:	beq r1, r3, Label1	beq r1, r3, **Label2**
4:		**beq r1, r3, error**
5:	ld r2, (r4)	ld r2, (r4)
6:		**ld r2', (r4')**
7:	xor r8, r6, r1	xor r8, r6, r1
8:		**xor r8', r6', r1'**
9:		**bne r1, r1', error**
10:		**bne r5, r5', error**
11:	st r1, (r5)	st r1, (r5)
12:		**j Label1**
13:		**Label2:**
14:		**bne r1, r3, error**
15:	Label1:	Label1:

Fonte: Elaborada pelo próprio autor.

As instruções extras nas linhas 2, 6 e 8 foram inseridas para copiar o fluxo de dados das instruções 1, 5 e 7, respectivamente. As instruções de comparação inseridas nas linhas 9 e 10 verificam o conteúdo dos registradores utilizados na instrução que grava dados na memória, exibida na linha 11. A instrução de desvio condicional do programa original, mostrada na linha 3, é duplicada nas linhas 4 e 14, e representam a cópia do destino do salto para os casos onde ele não é tomado e onde ele é tomado, respectivamente. O restante das transformações realizadas sobre o código serve para manter a coerência do programa.

Outra abordagem híbrida, que visa proteger tanto os dados quanto o fluxo de execução do programa, é a técnica VARM_S_BRA. A diferença deste método híbrido para a técnica anterior é que este reduz a quantidade de acessos à memória. Para isto, ao invés de duplicar instruções que fazem a busca de dados da memória (ld - *load*) com uma nova

instrução *load*, é utilizada uma instrução que move o conteúdo entre registradores (mv - *move*). Desta forma, é possível copiar o conteúdo do registrador que contém o dado carregado da memória para o respectivo registrador cópia. Assim, a única modificação no trecho de código exemplo, apresentado na Figura 3.8, é utilizar uma instrução *mv* ao invés da instrução *ld* na linha 6.

Observe que, por se tratarem de técnicas que incluem duas abordagens de detecção, estas técnicas híbridas incorrem em alto *overhead* de instruções, passando de apenas 6 instruções para 15 em nosso código exemplo, o que causa um impacto negativo no desempenho.

3.2 Técnica Implementada em Hardware

O foco deste trabalho são as técnicas desenvolvidas utilizando apenas software, onde não é necessário adicionar hardware nem realizar modificações na organização do processador. Todavia, com o intuito de se aproximar da máxima cobertura de falhas e comparar a eficiência das técnicas em software com as em hardware, também foram feitas estimativas utilizando a redundância em hardware.

Para esta análise, foi utilizada a técnica de redundância modular dupla (Dual Modular Redundancy - DMR). A metodologia utilizada divide cada processador superescalar em 12 módulos de hardware e, cada uma destas estruturas é duplicada separadamente. Deste modo, as mesmas operações são executadas em cada par de componentes, e os resultados são contrapostos na saída de cada par através de um comparador. Esta técnica também foi aplicada a diversos grupos de hardware, de maneira que diferentes combinações fossem duplicadas em cada cenário.

Devido aos custos de área e potência associados com a redundância em hardware, foi necessário encontrar uma forma mais eficaz de escolher as partes do processador a serem duplicadas. Para tal, foi utilizado o problema da mochila (KSP), cujo o objetivo é preencher uma mochila com os objetos mais valiosos, sem ultrapassar o limite de capacidade que a mochila pode carregar.

3.2.1 Problema da Mochila

O KSP é um problema de otimização combinacional que consiste em encontrar um ou mais objetos para colocar na mochila, a partir de um conjunto finito de itens. Cada objeto tem um custo e um ganho associado, e o objetivo é pôr na mochila os itens que somam o maior ganho, mas sem exceder o custo máximo que a mochila é capaz de sustentar. Neste trabalho foi utilizado o KSP que não permite repetições de objetos.

Nós utilizamos o KSP para decidir, de forma eficiente, o conjunto de módulos de hardware para aplicar a técnica DMR em hardware a fim de atingir determinada cobertura de falhas com mínimo custo de área. Desta forma, nós definimos a confiabilidade desejada, e o algoritmo do KSP se encarrega de indicar as estruturas de hardware a serem duplicadas em cada cenário. Dado que o KSP encontra o conjunto mais apropriado de estruturas que devem ser duplicadas, nenhuma outra combinação de estruturas é capaz de gerar menor *overhead* de área para a mesma meta de confiabilidade.

Em nosso caso, a capacidade da mochila é definida como a máxima vulnerabilidade permitida no processador, o valor associado a cada item é a área da estrutura, e o peso de cada item é a contribuição em vulnerabilidade que o item tem no processador como um todo.

Entende-se por vulnerabilidade base de uma estrutura a probabilidade de um *bit-flip* nessa estrutura resultar em um erro na saída do programa quando nenhuma técnica de tolerância a falhas é aplicada. Esta vulnerabilidade base foi obtida após uma campanha de injeção de falhas.

O algoritmo do KSP escolhe os itens com maior custo (i.e., maior área), ou seja, o KSP seleciona o conjunto de estruturas de forma que a área das estruturas selecionadas é maximizada, enquanto não violando a máxima vulnerabilidade permitida (i.e., a capacidade da mochila). Isto significa que devemos duplicar os itens que *não* são selecionados. Como o KSP maximiza o valor dos itens escolhidos para pôr na mochila, isto implica que os objetos não selecionados são aqueles com menor custo (i.e., o KSP minimiza a área das estruturas não selecionadas).

A fim de esclarecer o uso do KSP em nosso trabalho, o leitor pode observar as seguintes situações:

- Se não for necessário reduzir a vulnerabilidade do processador, então a capacidade da mochila é definida com o próprio valor da vulnerabilidade base. Deste modo o KSP seleciona todas as estruturas. Porém, o DMR é aplicado às estruturas que

ficam de fora, portanto, nenhuma estrutura protegida.

- Se a vulnerabilidade desejada for 0, o KSP não seleciona nenhuma estrutura, porque a capacidade da mochila é 0, então a duplicação deve ser aplicada em todas estruturas rendendo 0% em vulnerabilidade.
- Se o objetivo é reduzir 70% da vulnerabilidade, então a capacidade da mochila permite apenas 30% da vulnerabilidade base. Deste modo o KSP seleciona o conjunto de estruturas com maior área possível que correspondem no *máximo* a 30% da vulnerabilidade base do processador, sendo que o restante das estruturas não selecionadas corresponde a no *mínimo* 70% da vulnerabilidade. Desta forma, duplicando as estruturas não selecionadas tem-se a redução de vulnerabilidade de no mínimo 70% com menor custo em área possível.

3.3 Processador Superescalar

A organização de um processador superescalar é projetada para encontrar instruções independentes entre si, que possam ser executadas ao mesmo tempo em diferentes unidades de processamento. O objetivo é fazer uma especulação dinâmica para explorar o paralelismo no nível de instrução (Instruction-Level Parallelism - ILP) através de BBs para fins de aumento de desempenho. Porém, na prática, a complexidade do processador também aumenta em razão da necessidade de mais circuitos de controle para especular quais instruções buscar na memória, a complexidade para encontrar de instruções independentes para a executar em paralelo (e especulativamente), e complexidade para garantir a correta semântica de execução da aplicação (para processadores com execução fora de ordem), dentre outros fatores.

Devido às dependências verdadeiras entre instruções (Read-After-Write - RAW), em alguns ciclos, unidades de execução podem ficar sem trabalhar, esperando o resultado de outras unidades de execução. Deste modo, é possível utilizar as unidades ociosas com o propósito de executar operações extras para verificar a consistência dos dados ou do fluxo de controle para garantir a correta ordem de execução das operações.

3.3.1 Processador BOOM

O processador utilizado nesta pesquisa foi o BOOM (Berkeley Out-of-Order Machine) (CELIO; PATTERSON; ASANOVIć, 2015). Este processador foi escolhido por se tratar de um superescalar parametrizável, sintetizável, com código-fonte aberto e criado no meio acadêmico com o intuito de servir como base de estudos para futuros processadores. A descrição comportamental do BOOM está implementada em *Chisel* (Constructing Hardware in Scala Embedded Language) (BACHRACH et al., 2012), uma linguagem de descrição de hardware (Hardware Description Language - HDL) derivada da linguagem de programação Scala. Chisel é capaz de traduzir o código fonte em Scala para a linguagem Verilog. O código Verilog, por sua vez, é transformado para um código C++ completamente equivalente ao modelo RTL em Verilog através da ferramenta Verilator (SNYDE, 2005).

Conceitualmente, o pipeline do BOOM está dividido em 10 estágios: *Fetch*, *Decode*, *Register Rename*, *Dispatch*, *Issue*, *Register Read*, *Execute*, *Memory*, *Writeback* e *Commit*. No entanto, muitos desses estágios são combinados para formar seis estágios: *Fetch*, *Decode/Rename/Dispatch*, *Issue/RegisterRead*, *Execute*, *Memory* e *Writeback*, com o *Commit* sendo realizado de forma assíncrona, assim desconsiderado do pipeline.

A organização do superescalar BOOM é capaz de executar a variante RV64G do conjunto de instruções RISC-V. Este superescalar suporta execução de instruções fora de ordem (Out-of-Order - OoO) e pode ser configurado para operar como um processador pipeline com diferentes tamanhos. Há várias características do processador que também podem facilmente ser configuradas, como a quantidade de instruções buscadas da memória de instruções em cada ciclo, o limite de instruções armazenadas na fila antes de serem decodificadas, a quantidade de registradores físicos, o tamanho das tabelas no preditor de saltos, e o número de entradas no buffer de reordenamento.

Para avaliar a eficiência das técnicas SIHFT em diferentes módulos de hardware contidos nos processadores superescalares, este trabalho efetuou injeções de falhas em 12 estruturas micro-arquiteturais de três configurações do BOOM: *single-*, *dual-* e *quad-issue*. É importante destacar que a configuração do *quad-issue* é similar a um ARM Cortex-A15. As configurações dos três processadores utilizados são exibidas na Tabela 3.1.

A Figura 3.9 esboça um diagrama esquemático do BOOM *single-issue*.

Tabela 3.1: Parâmetros utilizados para gerar cada processador

	Single-issue	Dual-issue	Quad-issue
Fetch-/Issue-/Commit-width	1	2	4
Instruction fetch buffer entries	4	4	4
Instruction Window entries	10	20	28
Physical Register File	100	110	128
Num RF Read Ports	3	5	9
Num RF Write Ports	2	3	4
Num Bypass Ports	3	4	5
LSU entries	4	16	32
ROB entries	24	48	64

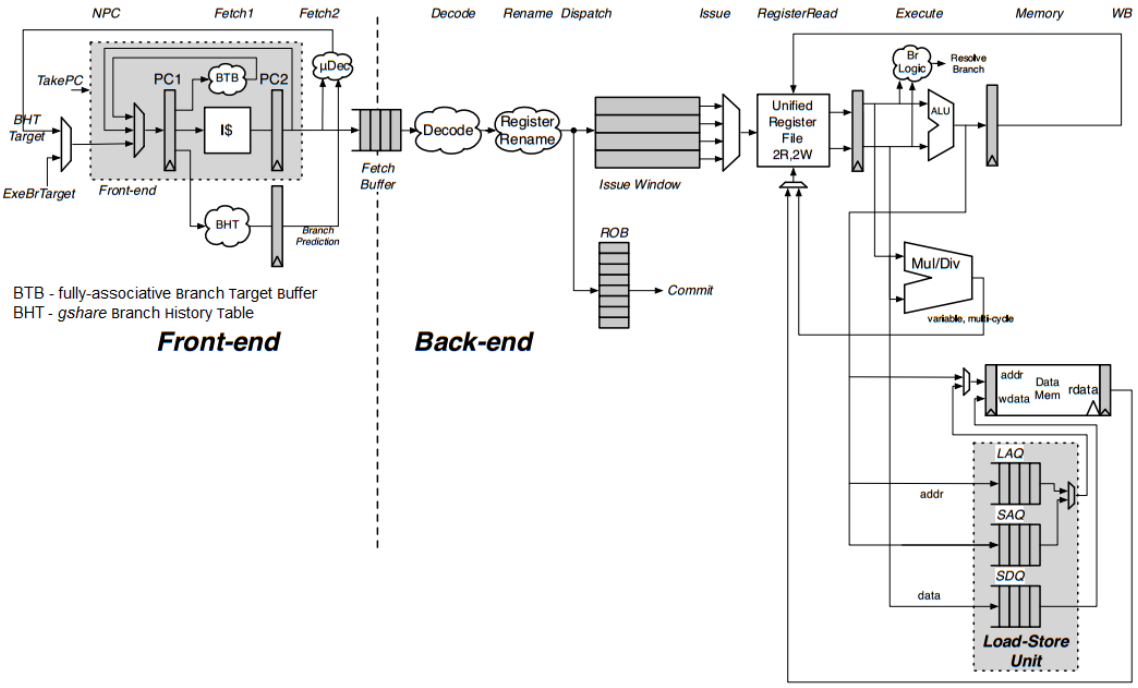
Fonte: Elaborada pelo próprio autor.

Ao utilizar um processador com diversos parâmetros variáveis, foi possível explorar a eficiência das técnicas em diferentes configurações em um amplo espaço de exploração de projeto. Com este cenário, também foi possível identificar se as técnicas são mais ou menos eficientes em um superescalar para cada estrutura interna do processador.

As 12 estruturas injetadas com falhas do tipo SEU foram:

- Banco de registradores físicos (Physical Register File - PRF).
- Unidades de renomeação dos registradores (Register Renaming Units).
- Unidade de despacho (Issue Unit - IU), que inclui a janela de instruções.
- Unidade de busca de instruções (Fetch Unit - FU).
- Buffer de reordenamento de desvios (Branch Reorder Buffer - BROB).
- O segundo estágio do preditor de desvios (Backing Predictor - BPD).
- Unidades de *Load* e *Store* (Load-Store Unit - LSU), que inclui as filas com endereços de memória acessados por estas instruções, e a fila com dados a serem armazenados na memória principal.
- Buffer de reordenamento (Reorder Buffer - ROB).
- O banco de registradores de controle/status (Control/Status Registers - CSR File)
- A cache com traduções mais recentes de endereços virtuais das instruções em endereços físicos (instruction Translation-Lookaside Buffer - iTLB)
- O estágio inicial do preditor de desvios (FeBr), que inclui tabela com endereços alvos dos desvios e dados de resolução de saltos incondicionais BTB.
- Todos os flip-flops no estágio de execução (Execution stage - EXE), incluindo unidades lógicas e aritméticas (Arithmetic-Logic Units - ALUs), unidades de ponto flutuante (Floating-Point Unit - FPU) e a lógica da rede de contorno para adianta-

Figura 3.9: Esquemático do processador BOOM *single-issue*



Fonte: (CELIO; PATTERSON; ASANOVIć, 2015)

mento (bypass).

O estágio de execução do BOOM é composto por Unidades de Execução (UE). A quantidade de UEs em cada versão do BOOM corresponde ao número máximo de micro-operações que podem ser computadas em um ciclo, portanto o BOOM *single-issue* contém apenas uma UE, enquanto que o *quad-issue* conta com quatro UEs. Cada unidade de execução é capaz de resolver diferentes tipos de instruções, mas somente uma por ciclo. Os tipos de micro-operações suportadas por cada Unidade Funcional (UF), em cada versão do BOOM, são apresentadas na Tabela 3.2. Repare que a UE do BOOM *single-issue* deve ser capaz de resolver todos os tipos de instruções, ALUs, FPUs, LSUs e multiplicações e divisões de inteiros (iMul e iDiv), além das multiplicações e divisões com ponto flutuante e operações de raiz quadrada (fDiv/fSqrt).

O preditor de desvios do BOOM está dividido em dois estágios, *front-end* (FeBr) e *back-end* (Backing Predictor - BPD). A estrutura FeBr contém a tabela (Branch Target Buffer - BTB) que inclui os endereços alvos dos desvios condicionais e dados para saltos incondicionais com endereço de retorno (jr), a pilha com endereços de retorno (Return Address Stack - RAS), além de outros recursos de hardware necessários para resolver corretamente a predição de desvios. O estágio inicial do preditor de desvios *front-end*

Tabela 3.2: Unidades de Execução (UE) nas três versões do BOOM

	Unidade de Execução	ALU	FPU	iMul	iDiv	fDiv/fSqrt	Mem
Single-issue	UE#0	✓	✓	✓	✓	✓	✓
Dual-issue	UE#0	✓	✓	✓			
	UE#1	✓			✓	✓	✓
Quad-issue	UE#0	✓	✓	✓			
	UE#1	✓					
	UE#2	✓			✓	✓	
	UE#3						✓

Fonte: Elaborada pelo próprio autor.

utiliza apenas o contador de programa (Program Counter - PC) da instrução de salto para recuperar o endereço de destino com a próxima instrução a ser executada. O BPD é um preditor mais lento e mais complicado, que funciona como um segundo nível para predição de desvios, utilizando um histórico global de desvios anteriores (com correlação entre saltos) para indexar uma tabela com um conjunto de contadores, que indicam se o desvio deve ser tomado ou não. No entanto, esta estrutura não armazena nenhum endereço destino de saltos (o destino usado é o calculado no estágio de execução), mas depende da condição de desvio e do cálculo do endereço alvo, que ocorre somente nos estágios de *back-end* do pipeline.

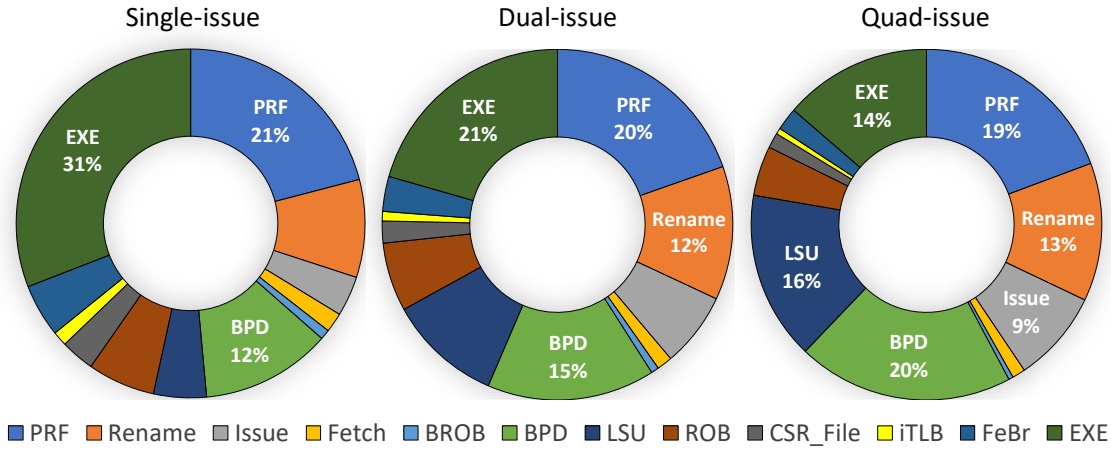
Para obter a área do BOOM, os processadores foram sintetizados com a ferramenta *Cadence RTL Compiler* usando a biblioteca *NanGate* 15nm *standard cell* (MARTINS et al., 2015) para alcançar uma frequência de operação de 2 GHz. A Figura 3.10 esboça a proporção de área que cada estrutura ocupa em cada processador utilizado em nossa análise, enquanto que a Tabela 3.3 mostra os valores de área, a quantidade de células e o número de *flip-flops* em cada estrutura individual dentro dos superescalares, além do total em cada versão do BOOM. Nesta implementação, não foram consideradas as memórias caches de dados e instruções.

3.4 Benchmark

Para avaliar a capacidade de detecção de SEUs pelas técnicas e identificar as técnicas mais apropriadas para o processador superescalar, escolhemos 13 programas para compor nosso *benchmark* de aplicações:

- *crc32* - Verificação Cíclica de Redundância (Cyclic Redundancy Check - CRC) é outra estratégia para detecção de erros, muito utilizada na transmissão de dados,

Figura 3.10: Proporção de área de cada estrutura nos 3 processadores



Fonte: Elaborada pelo próprio autor.

Tabela 3.3: Área (μm^2), número de células e de *flip-flops* em cada estrutura dos processadores

	Single-issue			Dual-issue			Quad-issue		
	Área	#Células	#FF	Área	#Células	#FF	Área	#Células	#FF
PRF	17742	37406	6500	25777	64055	7150	40023	101715	8320
Rename	7680	17610	2976	16178	38362	5412	26390	78824	5600
Issue	3151	7615	8456	9039	24686	17374	17630	52214	24572
Fetch	1537	3098	750	1865	3721	1034	2496	4916	1598
BROB	778	1605	393	938	1919	468	877	1856	416
BPD	10227	22919	257	20350	45590	319	41299	92914	377
LSU	4149	8992	1818	13807	32135	5233	32192	82226	10211
ROB	5303	11340	4427	8319	18068	7778	9495	21000	9692
CSR_File	2662	5963	1686	2660	6047	1686	2913	6502	1686
iTLB	1147	2337	513	1147	2338	513	1150	2352	513
FeBr	4160	9066	1731	4242	9215	1810	4326	9387	1850
EXE	26174	63880	8571	26990	65952	10271	28301	69933	11965
Total	82429	186622	43163	132885	316147	69090	214894	542012	96299

Fonte: Elaborada pelo próprio autor.

via redundância de informação. A verificação é calculada com base na divisão polinomial dos dados, e neste caso, o polinômio gerador tem 33 bits.

- *dijkstra* - Calcula o caminho mais curto entre dois nodos de um grafo. Neste caso, o grafo está representado como uma matriz de adjacências.
- *kmeans* - *K-means* é um algoritmo muito utilizado em *data mining*. Ele agrupa "X" amostras em "K" grupos. Nós utilizamos *X=100*, e *K=5*.
- *matMul* - Multiplica duas matrizes com tamanho 16x16 e armazena o resultado em

outra matriz 16x16.

- *medianFilter* - Aplica um filtro 1D, através da mediana de três elementos, muito usado para remover o ruído de uma imagem ou sinal. Nossa entrada é um conjunto de 400 elementos.
- *multiply* - Aplica um filtro através de multiplicações de elementos adjacentes.
- *pbmsrch* - *Pratt-Boyer-Moore* é um algoritmo para pesquisar palavras em um texto.
- *qsort* - *Quicksort* é um algoritmo eficiente para ordenação de elementos. Nós utilizamos um vetor com 250 elementos.
- *rijndael* - *Rijndael* é também conhecido pelo nome Padrão de Criptografia Avançada (Advanced Encryption Standard - AES), é utilizado para criptografar dados.
- *rsort* - *Radix sort* é um algoritmo de ordenação. Nós utilizamos um vetor com 512 elementos.
- *sha* - O Algoritmo de *Hash* Seguro (Secure Hash Algorithm - SHA) é usado na troca segura de chaves criptográficas e na geração de assinaturas digitais. Neste caso, são utilizados blocos de 512 bits de entrada e produz mensagens resumidas de 160 bits como saída.
- *towers* - Simula torres de Hanói.
- *vvadd* - Soma dois vetores de tamanho 300 e armazena o resultado em um terceiro vetor.

Parte destes programas são disponibilizadas junto com o simulador do BOOM, as demais foram retiradas do pacote de aplicações MiBench (GUTHAUS et al., 2001).

A Tabela 3.4 apresenta a quantidade de instruções executadas por cada programa, onde as instruções do tipo *Jal* (*Jump and link*) referem-se a instruções de salto incondicional com endereço de retorno armazenado em um registrador. O grupo de instruções *Br+Jmp* diz respeito a instruções de desvio condicional (Br) e demais instruções de salto incondicional (Jmp). Para monitorar as instruções executadas dinamicamente, foi necessário acoplar, junto ao simulador do BOOM, um contador para cada grupo de instruções.

A Tabela 3.5 mostra a média de instruções executadas em um ciclo (Instructions per Cycle - IPC), e o tempo necessário para executar cada programa nas três versões do BOOM utilizadas neste trabalho, considerando a frequência de operação de 2 GHz.

Repare que, em alguns casos, o tempo necessário para executar algumas aplicações no *quad-issue* é maior do que no *dual-issue*, com destaque para o CRC32. Este comportamento pode ser explicado, em parte, pelo limite de paralelismo das aplicações,

Tabela 3.4: Mix de instruções dos programas utilizados

	Lógicas	Loads	Stores	Br+Jmp	Jal	Total
crc32	38176	27593	15463	5532	571	87335
dijkstra	18632	5568	2328	1608	281	28417
kmeans	1788	2355	813	378	140	5474
matMul	67902	38453	9255	4645	276	120531
medianFilter	9583	16254	3998	2951	640	33426
multiply	37183	35943	21579	13406	405	108516
pbmsrch	48778	26983	12896	6882	235	95774
qsort	6243	14571	4535	3135	703	29187
rijndael	3776	1359	236	28	22	5421
rsort	40310	37702	12326	2076	26	92440
sha	58355	37775	13172	3082	159	112543
towers	7744	13126	7789	1348	1194	31201
vvadd	4825	6608	1216	605	4	13258

Fonte: Elaborada pelo próprio autor.

obtido já na versão *dual-issue* do BOOM. Outra questão que contribui para esta perda de desempenho é a precisão do preditor de desvios, que é menor no *quad-issue*. Para executar o CRC32, a precisão do preditor de desvios é 11% melhor no *dual-issue* do que no *quad-issue*, e durante a execução do medianFilter, o preditor acerta 13% a mais no *dual-issue* do que no *quad-issue*. Além disso, esta penalidade é mais acentuada nos programas com maior densidade de instruções do tipo desvio condicional.

3.5 Ferramenta CFT

A ferramenta Configurable Fault Tolerant (CFT) (CHIELLE et al., 2012) está descrita em linguagem de programação Java e foi utilizada para aplicar as técnicas de tolerância a falhas sobre o código *assembly* da aplicação, assim sendo é possível fazer modificações nas técnicas e proteger um novo programa de forma rápida e eficiente.

Além do código *assembly* a ser protegido, a CFT também precisa ser configurada com arquivos que descrevem a arquitetura do processador alvo e um arquivo com técnicas a serem aplicadas, bem como as características destas técnicas. A saída da CFT é o código *assembly* com as técnicas, chamado de *Hardened code*.

Embora a proteção dos programas pudesse ser realizada de forma automática, devido às transformações no baixo nível da aplicação, houveram muitas situações em que o programa deixava de funcionar ou perdia sua característica funcional. Em vista disso, foi necessário analisar cada programa em uma execução passo a passo para encontrar

Tabela 3.5: IPC e Tempo de Execução dos programas em três versões do BOOM

Programa	IPC			Tempo de Execução (μs)		
	Single	Dual	Quad	Single	Dual	Quad
crc32	0,712	1,019	0,985	61,36	42,85	44,37
dijkstra	0,721	1,227	1,594	19,71	11,58	8,91
kmeans	0,485	0,780	0,797	5,64	3,51	3,44
matMul	0,755	1,618	1,718	79,84	37,26	35,07
medianFilter	0,629	1,038	0,995	26,59	16,11	16,79
multiply	0,606	1,078	1,064	89,52	50,32	50,99
pbmsrch	0,810	1,409	1,540	59,11	34,00	31,09
qsort	0,522	0,721	0,708	27,98	20,25	20,62
rijndael	0,576	0,905	1,077	4,71	3,00	2,52
rsort	0,694	1,201	1,251	66,59	38,48	36,94
sha	0,649	1,304	1,487	86,75	43,15	37,83
towers	0,548	1,065	1,090	28,48	14,65	14,32
vvadd	0,648	1,337	1,330	10,22	4,96	4,99

Fonte: Elaborada pelo próprio autor.

os motivos que corrompiam as aplicações. Após este diagnóstico, foi necessário efetuar diversas adaptações na ferramenta CFT.

3.5.1 Modificações na Ferramenta CFT

Durante a execução de um programa no processador BOOM, alguns registradores arquiteturais são pré-inicializados no *startup code* (ra, sp, gp, tp, t0, s0, s1, a2, a4) antes da execução da função *main()*. Para finalizar o programa corretamente, o conteúdo destes registradores deve ser o mesmo de quando a *main()* foi iniciada. Portanto, estes registradores nunca puderam ser usados pelas técnicas como cópia de outros registradores.

Quando algum dos registradores pré-inicializados é usado no código fonte do programa, e se ele é um dos registradores a ser protegido, o seu registrador cópia deve receber o conteúdo deste registrador no início da execução. Em vista disso, logo no início da *main()* o conteúdo desse registrador teve de ser copiado para o seu respectivo registrador cópia utilizando a instrução *mv* (move).

Há dois tipos de registradores arquiteturais disponíveis para uso no BOOM, ou eles armazenam valores inteiros ou valores com ponto flutuante. Da mesma forma, existem instruções específicas para operar sobre cada tipo de instrução. Porém, quando o código era replicado pela CFT, a ferramenta usava qualquer registrador global disponível para uso nas instruções duplicadas. Então, às vezes, registradores destinados a armazenar valores

inteiros eram usados em instruções que só devem realizar cálculos com ponto flutuante, e isto não era reconhecido pelo montador.

Além disso, técnicas SIHFT precisam inserir instruções para comparar o conteúdo dos registradores e saltar. Para tal, são utilizadas as instruções do tipo inteiro *bne* e *beq*. Porém, não existe uma instrução específica para comparar registradores de ponto flutuante e saltar. Então, tivemos que adaptar a ferramenta para poder comparar o conteúdo dos registradores usados em operações de ponto flutuante.

A Figura 3.11 expõe uma simples instrução de ponto flutuante que realiza a multiplicação de dois valores desprotegidos. Esta poderia ser protegida com instruções que operam sobre registradores inteiros, se fosse suportado, mas acaba se tornando um código complexo com muitas instruções extras para o uso da técnica SIHFT. A instrução *feq.d* compara o conteúdo dos registradores de ponto flutuante: se os valores são iguais, então armazena o valor '1' no registrador inteiro; caso contrário, é armazenado o valor '0'.

Figura 3.11: Exemplo de uso da técnica VAR para proteger instrução de ponto flutuante

	Instrução original	VAR com erro	VAR corrigida
1:			**feq.d t1, fa4, ft4'**
2:			**beq t1, zero, error**
3:		**bne fa4, ft4', error**	**feq.d t2, fa5, ft5'**
4:		**bne fa5, ft5', error**	**beq t2, zero, error**
5:	fmul.d fa3, fa4, fa5	fmul.d fa3, fa4, fa5	fmul.d fa3, fa4, fa5
6:		**fmul.d ft3', ft4', ft5'**	**fmul.d ft3', ft4', ft5'**

Fonte: Elaborada pelo próprio autor.

Devido a maioria das técnicas implementadas neste trabalho verificarem a integridade dos registradores apenas quando eles são usados como entrada das micro-operações, as instruções que realizam salto incondicional e utilizam registradores onde os erros facilmente se manifestam, ficavam sempre desprotegidas. Portanto, com a intenção de aumentar a cobertura de falhas, nós também adicionamos a verificação de qualquer registrador utilizado pelas instruções deste tipo antes que o salto fosse executado.

Havia também diversas instruções que não constavam na documentação do BOOM, mas que eram inseridas pelo compilador, sendo necessário adaptar diversos arquivos de configuração para que as técnicas fossem aplicadas de forma genérica. Estas instruções referem-se a expansões de realocação do montador, pseudo-instruções, instruções para endereçamento absoluto e endereçamento relativo, entre outras. Outra situação inusitada era causada por causa da nomenclatura de algumas instruções que continham símbolos e

caracteres especiais.

O uso da técnica BRA apresentava uma deficiência quando duas instruções de desvio, distintas, podiam saltar para um mesmo local. Isso ocorria porque esta técnica protege instruções de desvio através da duplicação destas operações em ambos os possíveis destinos. A Figura 3.12 ajuda a compreender este problema.

Figura 3.12: Exemplo de uso da técnica BRA quando duas instruções desviam para o mesmo local

	Código original	**BRA com erro**	**BRA corrigida**
1:	`ble r1, r2, Label`	`beq r1, r2, Label2`	`beq r1, r2, Label2`
2:		**`beq r1, r2, error`**	**`beq r1, r2, error`**
3:	`add r3, r4, r5`	`add r3, r4, r5`	`add r3, r4, r5`
4:	`beq r6, r7, Label`	`ble r6, r7, Label2`	`ble r6, r7, Label3`
5:		**`ble r6, r7, error`**	**`ble r6, r7, error`**
6:	`ld r8, (r9)`	`ld r8, (r9)`	`ld r8, (r9)`
7:		**`j Label`**	**`j Label`**
8:		**`Label2:`**	**`Label3:`**
9:		**`bgt r6, r7, error`**	**`bgt r6, r7, error`**
10:			**`j Label`**
11:			**`Label2:`**
12:		**`bne r1, r2, error`**	**`bne r1, r2, error`**
13:	`Label:`	`Label:`	`Label:`

Fonte: Elaborada pelo próprio autor.

A ferramenta inseria apenas um local extra, que seria utilizado como destino de ambos os desvios caso estes fossem tomados. Porém, caso um dos desvios seja tomado para “Label2”, é muito improvável que os valores dos registradores presentes na outra instrução fossem os valores esperados. Por este motivo, o programa salta para a sub-rotina de detecção de erro, mesmo quando não há injeção de falhas. Em vista disso, foi necessário alterar a ferramenta CFT a fim de inserir um *Label* extra para cada instrução que possa desviar para um mesmo local.

Com as instruções de salto incondicional, é possível saltar até ± 1 megabytes. Já a instrução *bne*, muito utilizada pelas técnicas, só consegue saltar ± 4 kilobytes, então só é permitido saltar no máximo 1.000 instruções. Porém, quando o programa era muito comprido, o montador precisava inserir diversas instruções de salto incondicional, imediatamente após os *bne*, para conseguir alcançar a sub-rotina que indica a detecção de erro. Deste modo, a fim de evitar a execução de diversos saltos desnecessários, a solução empregada foi modificar a ferramenta CFT para inserir diversas sub-rotina de detecção de erro mais próximas de onde ocorrem as comparações entre os conteúdos dos registradores original e cópia.

3.6 Injeção de Falhas

Para realizar a simulação de falhas no processador, o presente trabalho utilizou a plataforma de injeção de falhas desenvolvida em (TONETTO; NAZAR; BECK, 2018). Esta ferramenta de injeção de falhas foi implementada junto ao simulador RTL do BOOM, com precisão de ciclo, e nos permite acessar todos os registradores do processador. Devido ao alto grau de controlabilidade dessa ferramenta, é possível escolher livremente o espaço e o tempo para a injeção de falhas no nível de bit e com precisão de ciclo.

Cada falha simulada corresponde a apenas um SEU injetado em um flip-flop escolhido de forma aleatória, direcionado a uma das 12 estruturas micro-arquiteturais do processador. Cada bit-flip foi injetado enquanto somente um dos programas era executado no processador em questão. Esta campanha de injeção de falhas foi repetida para cada programa, sem técnica de detecção de falhas, e depois com cada uma das técnicas avaliadas.

A fim de obter resultados mais sólidos e precisos, foi necessária uma enorme quantidade de simulação de falhas até os níveis de vulnerabilidade das estruturas estabilizar. Contudo, é impossível injetar falhas em todos os locais de um circuito complexo, e cobrir todos os ciclos de clock em um tempo razoável é impraticável. Portanto, para determinar a quantidade de falhas a serem injetadas em cada estrutura do processador superescalar, foi utilizado o modelo estatístico de injeção de falhas (Statistical Fault Injection - SFI) descrito em (LEVEUGLE et al., 2009), considerando o número de elementos de memória presentes no circuito que podem ser afetados pelas falhas e o número de ciclos de cada aplicação. Com isto, o valor estimado foi de 15.000 falhas em cada estrutura, levando a um nível de confiança dos resultados de 99%, com uma margem de erro de apenas 1%. A simulação de SEUs ocorreu de modo que a quantidade de falhas inseridas fosse igualmente distribuída entre os módulos, até atingir um número suficiente de falhas.

Como nós utilizamos 13 programas em nosso *benchmark*, primeiro sem técnica de tolerância a falhas, e depois protegidos com nove diferentes técnicas. Todas estas aplicações foram submetidas a execução em três diferentes versões do BOOM, e 15k falhas injetadas em cada umas das 12 estruturas, totalizando 70 milhões de SEUs. Quando expandimos nossa análise para a proteção seletiva de registradores, oito programas foram utilizados, e cada um ganhou aproximadamente mais 14 versões protegidas. Com isso, 60 milhões de falhas adicionais foram injetadas, totalizando 130 milhões de SEUs.

Para injetar 15k falhas falhas em todas as estruturas do *dual-issue* executando todo

nosso *benchmark* de aplicações, protegido com uma das técnicas, levou 5 dias e 21 horas, rodando em um servidor com *Intel Core i7-6700 CPU @ 3,40GHz*, 4 núcleos 8 *threads*, com 32 GiB de memória. Para injetar 15k falhas falhas em todas as estruturas do *quad-issue* executando todo nosso *benchmark* de aplicações protegido com uma das técnicas levou 8 dias e 15 horas rodando no mesmo servidor. Em cada um destes dois cenários foram 2.340.000 simulações de falhas, onde cada simulação recebe apenas uma falha simples.

Esta grande quantidade de falhas foi otimizada com o mecanismo de ponto de verificação (*checkpointing*) disponível na própria plataforma de injeção de falhas. Desta forma, o *benchmark* é executado uma primeira vez sem simular qualquer falha e, a cada 500 ciclos, o injetor salva o estado de todos os flip-flops e endereços de memória. Desta forma, obtemos o estado de todo o processador para uma execução livre de falhas. Para realizar a simulação de uma falha, sorteamos um flip-flop para receber a falha e um ciclo de clock para definir o instante em que ocorre a injeção, e restauramos o estado do processador para o *checkpoint* mais próximo, imediatamente antes do ciclo sorteado. Então, a execução continua normalmente até este ciclo, o bit sorteado é invertido, e a execução continua até o final ou até que um dos próximos *checkpoints* reconheça que a falha foi mascarada.

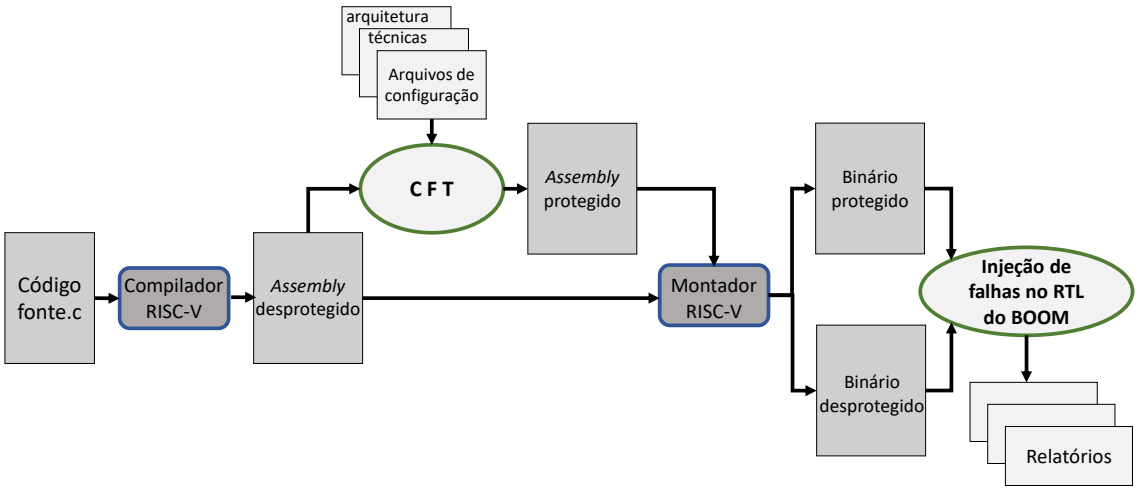
Para discriminar se a falha foi detectada pela técnica ou resultou em mau funcionamento do sistema, cada falha simulada foi classificada de acordo com seu efeito:

- Corrupção silenciosa de dados (Silent Data Corruption - SDC) - quando o programa termina corretamente, mas o conteúdo da memória principal é diferente da memória sem injeção de falhas.
- *Timeout* - quando a execução da aplicação demora mais do que a versão sem falhas. Para evitar que a aplicação execute indefinidamente, a execução para quando ela atinge o dobro de ciclos de uma versão sem falhas.
- *Crash* - quando a execução aborta, por exemplo, devido a uma falha de segmentação, quando a aplicação simulada no processador tenta acessar algum endereço de memória corrompido.
- Detectada - quando o fluxo do programa desvia para uma sub-rotina de detecção de falha utilizada por uma das técnicas de tolerância a falhas implementada em software.
- Mascarada - quando a execução do programa termina sem erros na memória princi-

pal. Isto ocorre porque o *flip-flop* falho não foi usado ou o seu valor foi sobrescrito antes de ser usado.

A Figura 3.13 expõe todo o fluxo utilizado em nossa metodologia, incluindo a ferramenta CFT e o injetor de falhas. Embora o fluxo do binário desprotegido e o fluxo da aplicação com técnica estejam representados ao mesmo tempo, somente uma das opções é executada por vez no simulador do BOOM.

Figura 3.13: Fluxo para aplicar as técnicas SIHFT e injetar falhas



Fonte: Elaborada pelo próprio autor.

4 VULNERABILIDADE DOS PROCESSADORES SUPERESCALARES

O objetivo deste capítulo é demonstrar os níveis de vulnerabilidade de diferentes processadores superescalares e como estes podem se tornar mais resilientes com o emprego de técnicas para mitigação de falhas. Para mensurar esta vulnerabilidade, nós realizamos uma extensa campanha de injeção de SEUs em diferentes processadores e, assim, pudemos compreender como estes se comportam quando são influenciados pelas falhas. Os resultados estão separados em quando os processadores estão desprotegidos, sem técnicas de tolerância a falhas, e quando diferentes técnicas são utilizadas para aumentar a confiabilidade destes processadores. Esta análise também foi realizada para cada estrutura micro-arquitetural dentro do processador, a fim de entender os diferentes motivos que levam ao mau funcionamento de um superescalar devido a uma simples falha.

Embora o foco deste trabalho seja as diferentes técnicas implementadas em software, a fim de contrabalancear os custos de área, energia com a confiabilidade, nós também realizamos uma proteção parcial das aplicação e utilizamos o DMR em hardware para proteger as estruturas mais vulneráveis.

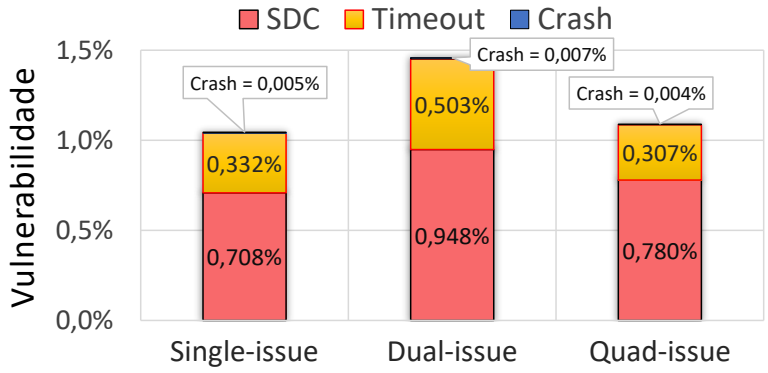
A vulnerabilidade a falhas do BOOM está expressa como a probabilidade de uma falha causar um defeito na saída da aplicação, exposto ao usuário, considerando que um ocorreu um bit-flip em algum flip-flop de alguma estrutura do superescalar. A Equação 4.1 mostra como calculamos a vulnerabilidade de cada estrutura micro-arquitetural do processador, isto é, a porcentagem de SEUs em um módulo de hardware que resultou em um erro. Esta métrica foi estendida para estimar a vulnerabilidade do superescalar como um todo. No entanto, para calcular a vulnerabilidade de todo o processador, não é correto usar apenas uma média aritmética simples entre vulnerabilidade das estruturas em cada processador, pois os módulos de hardware têm diferentes tamanhos em área, e estruturas maiores estão mais expostas à radiação. Portanto, neste caso, foi utilizado uma média ponderada que considera a área de cada estrutura.

$$Vulnerabilidade \approx \frac{\#SDC + \#Timeout + \#Crash}{\#SEUs\_injetados} \tag{4.1}$$

4.1 Sem Técnica de Tolerância a Falhas

A primeira análise foi realizada quando nenhuma técnica de tolerância a falhas estava sendo utilizada. Através desta análise, é possível dizer quais são os componentes mais sensíveis a falhas, além de identificar qual é a vulnerabilidade de diferentes processadores superescalares ao executar uma determinada aplicação. A Figura 4.1 apresenta a vulnerabilidade de três superescalares, separados em diferentes tipos de erros: falhas que causaram SDCs, falhas que resultaram em *timeouts* e falhas que levaram ao *crash* do simulador. A soma destes 3 tipos de erros representa a vulnerabilidade de casa processador.

Figura 4.1: SDCs, Timeouts e Crashes nos três superescalares sem técnica de tolerância a falhas

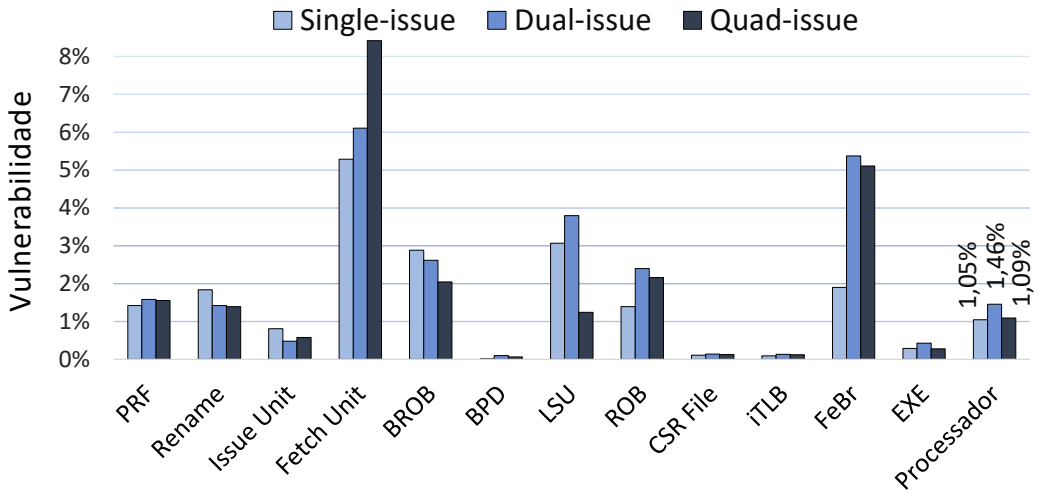


Fonte: Elaborada pelo próprio autor.

Os resultados obtidos mostram que os níveis de vulnerabilidade para os três processadores são diferentes. A versão *dual-issue* se mostra a mais propensa a erros, onde 1,458% dos SEUs injetados neste processador resultaram no mau funcionamento do processador. Dentre todas as falhas que resultaram em algum tipo de erro nos três processadores, os SDCs representam a aproximadamente 67%, enquanto que os *crashes* correspondem a apenas 0,45%. É importante destacar que a grande maioria das falhas foi mascarada pelas estruturas internas, intrínsecas à complexidade dos superescalares: mascaramento lógico, especulação, execução de código morto ou bit-flips injetados em estruturas sem informação relevante para a aplicação em execução.

A Figura 4.2 apresenta a vulnerabilidade de cada componente de hardware, interno aos processadores superescalares avaliados. É possível notar que há uma diferença significativa na vulnerabilidade de cada estrutura. Uma sensibilidade mais acentuada é notada na unidade de busca, principalmente quando o tamanho do processador aumenta.

Figura 4.2: Vulnerabilidade de cada módulo de hardware sem técnica de tolerância a falhas



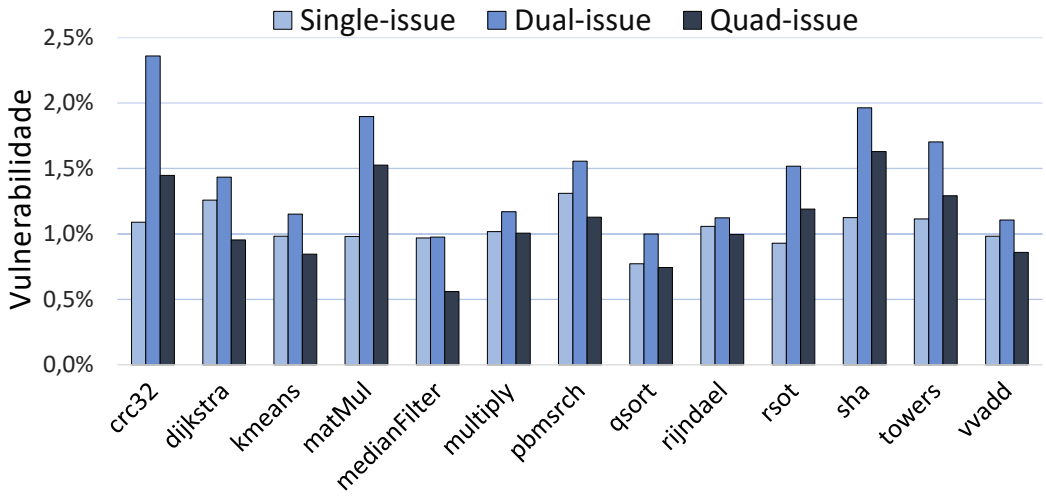
Fonte: Elaborada pelo próprio autor.

Este comportamento pode ser explicado devido ao aumento na ocupação desta estrutura, uma vez que a quantidade de entradas na fila de instruções (*Instruction fetch buffer entries*) buscadas da cache, não varia para todas as versões do BOOM, como mostrado na Tabela 3.1. Neste contexto, a ocupação de um módulo de hardware diz respeito a fração de flip-flops com valores relevantes para executar uma aplicação sem erros, isto é, bits válidos naquela estrutura.

Outra estrutura que apresenta elevado grau de vulnerabilidade é a FeBr. Nota-se que a estrutura FeBr é composta pelo BTB, a pilha com endereços de retorno e diversas estruturas internas com informações necessárias para a resolução correta dos desvios condicionais e, portanto, falhas injetadas nessa estrutura podem levar a erros. Da mesma forma, uma falha que atinge o BPD pode resultar no cálculo incorreto da condição de desvio ou do destino do desvio, e então causar erros na saída do programa.

A Figura 4.3 esboça a sensibilidade a falhas de cada processador superescalar enquanto eles executam cada uma das aplicações em nosso *benchmark*. É importante destacar que a vulnerabilidade do superescalar *dual-issue* se apresenta superior enquanto executa qualquer aplicação sem técnica.

Figura 4.3: Vulnerabilidade dos processadores executando cada aplicação sem técnica de tolerância a falhas



Fonte: Elaborada pelo próprio autor.

4.2 Avaliação das Técnicas Implementadas em SW

Com o intuito de avaliar a capacidade de resiliência dos processadores superescalares utilizando as técnicas de tolerância a falhas implementadas em software, nosso *benchmark* de aplicações foi protegido com diferentes técnicas que visam a proteção dos dados, técnicas para detecção de erros no fluxo de controle e técnicas que combinam ambas as abordagens.

Apesar dos nossos esforços para explicar o comportamento de algumas técnicas com base nos tipos de instruções presentes em cada programa, e no tamanho dos módulos de hardware, o trabalho (WALCOTT; HUMPHREYS; GURUMURTHI, 2007) demonstra que é muito difícil observar uma correlação direta entre estes parâmetros quando estamos lidando com um processador complexo. A vulnerabilidade do processador e, consequentemente, o desempenho de cada técnica depende de muitas variáveis envolvidas. Além disso, a precisão do preditor de desvios do BOOM tem uma variação bastante acentuada quando alteramos a quantidade de instruções na aplicação.

4.2.1 Proteção do fluxo de dados

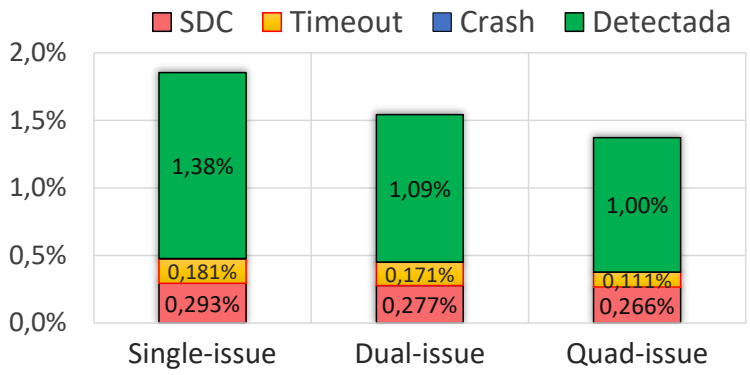
Para detectar falhas nos dados de uma aplicação durante sua execução em um processador, cinco técnicas foram avaliadas. A estratégia para detecção de falhas empregada por estas técnicas foi descrita na subseção 3.1.1, e a eficiência destas técnicas será apresentada nas subseções a seguir.

4.2.1.1 Eficiência da técnicas VAR

VAR foi a primeira técnica de tolerância a falhas implementada puramente em software avaliada. Todas as 13 aplicações em nosso *benchmark* foram protegidas com esta técnica e executadas nos três superescalares. Lembramos que, para cada execução de um programa em um dos processadores, apenas um bit-flip foi injetado em apenas uma das estruturas.

A Figura 4.4 apresenta os diferentes tipos de erros e a porcentagem de falhas detectadas pela técnica VAR enquanto cada processador executava nosso *benchmark* de aplicações protegido por esta técnica.

Figura 4.4: SDCs, Timeouts, Crashes e falhas Detectadas nos três superescalares com a técnica VAR



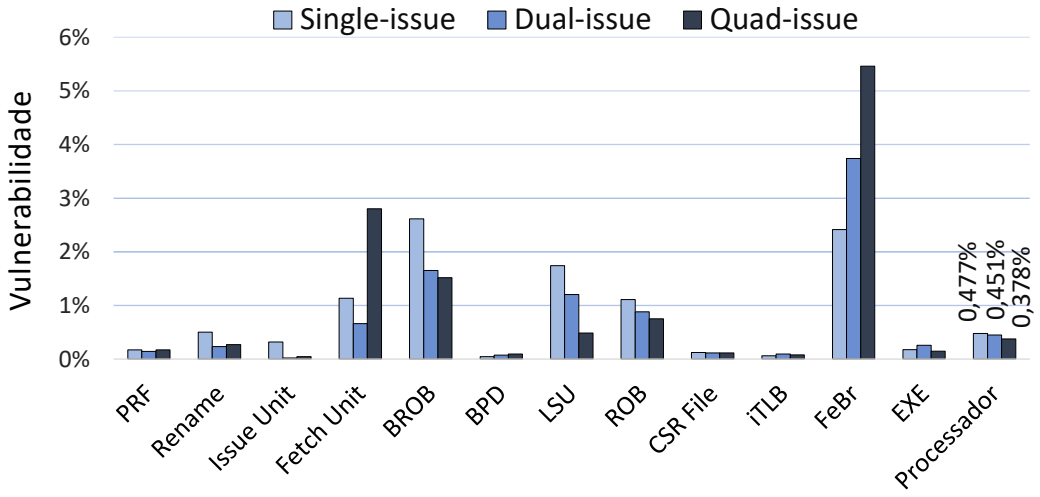
Fonte: Elaborada pelo próprio autor.

O objetivo de qualquer mecanismo de detecção de falhas é reduzir a quantidade de erros, e assim evitar possíveis danos ao usuário do sistema. Observando a Figura 4.4, embora a porcentagem de SDCs e *timeouts* tenha diminuído, quando estes tipos de erros são somados com as falhas detectadas, a técnica VAR aumenta a quantidade total de erros com relação ao sistema original desprotegido. Um dos motivos que explica este

comportamento é a detecção de falhas que não causariam erros na saída. Outra razão disto é que qualquer técnica para detecção de falhas necessariamente introduz algum tipo de redundância, o que aumenta o processamento e o número de flip-flops com valores úteis. Como todos os bits do processador estão vulneráveis, isso também leva ao aumento no número de erros, sejam eles detectados ou não (REIS et al., 2005a).

A vulnerabilidade de cada estrutura micro-arquitetural presente nos processadores superescalares avaliados é mostrada na Figura 4.5.

Figura 4.5: Vulnerabilidade de cada módulo de hardware quando a técnica VAR foi utilizada

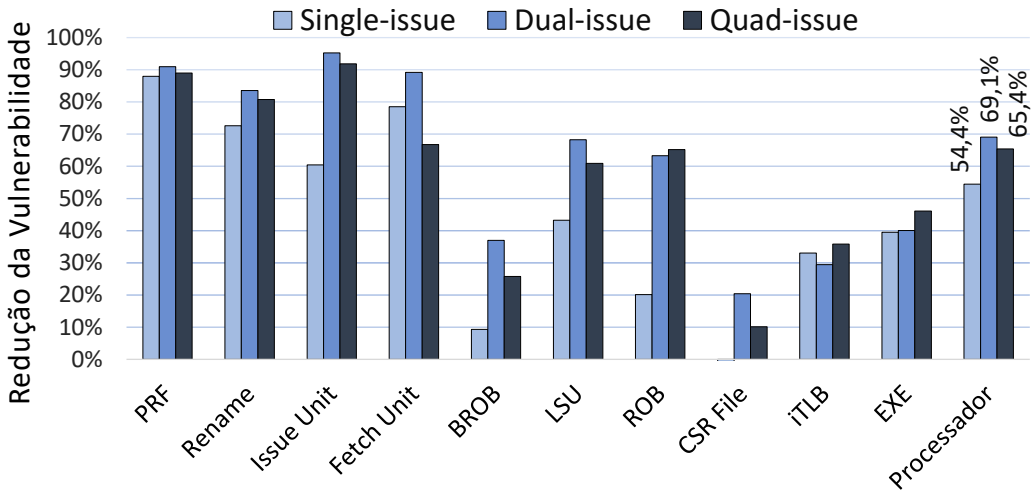


Fonte: Elaborada pelo próprio autor.

O ganho na capacidade de resiliência que uma técnica proporciona, com relação ao processador completamente desprotegido, pode ser representado como redução da vulnerabilidade alcançada por esta técnica. Isto é, a porcentagem de falhas que deixaram de causar erros. Desta forma, quanto maior for a redução da vulnerabilidade, maior é a eficiência dessa técnica. A Figura 4.6 mostra a redução da vulnerabilidade de cada módulo de hardware, além da redução da vulnerabilidade obtida nos três processadores, enquanto executavam nosso *benchmark* protegido com a técnica VAR.

A técnica VAR é capaz de reduzir a vulnerabilidade da maioria dos módulos de hardware. Por outro lado, a vulnerabilidade de outras estruturas continua alta. A vulnerabilidade de algumas estruturas até piorou durante a execução dos programas protegidos, que é o caso dos dois estágios necessários para prever o destino dos desvios condicionais. Em média, para os três processadores, a vulnerabilidade do BPD aumentou 60%, enquanto que a vulnerabilidade do FeBr piorou 26% no superescalar *single-issue* e 6% no

Figura 4.6: Redução da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica VAR



Fonte: Elaborada pelo próprio autor.

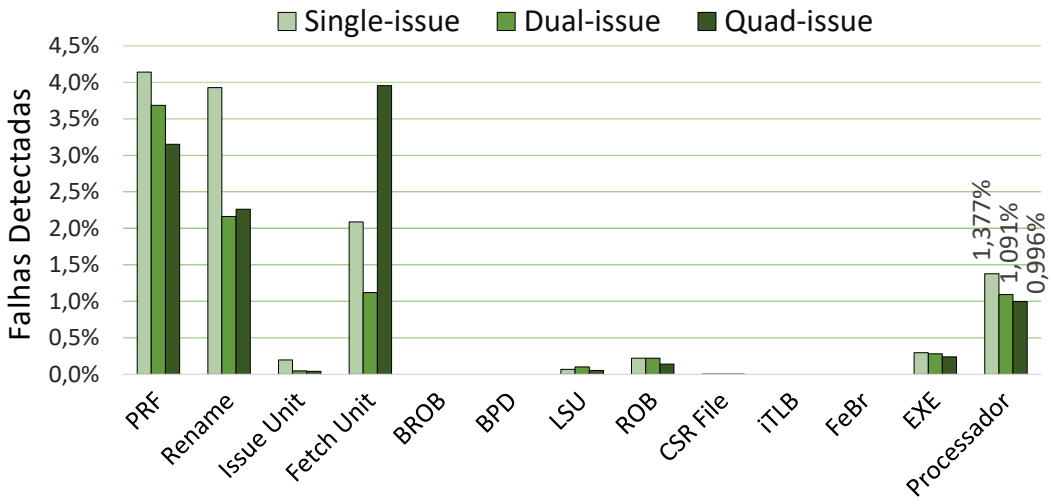
quad-issue. Isto acontece porque esta técnica requer grande quantidade de desvios condicionais para comparar registradores. Deste modo, o número de desvios condicionais aumenta em 19,5%, intensificando o uso destas estruturas. Devido ao aumento de flip-flops com valores indispensáveis para manter a correta execução destas estruturas, estas ficam mais propensas a erros.

A Figura 4.7 mostra a porcentagem de falhas detectadas pela técnica VAR em cada módulo de hardware presente nos processadores superescalares. É importante destacar que a técnica não foi o suficiente para capturar 100% das falhas injetadas, em quaisquer dos módulos avaliados.

Com isto, é possível afirmar que a técnica VAR foi capaz de identificar boa parte das falhas injetadas no banco de registradores físicos (PRF), na unidade de renomeação dos registradores (Rename) e na unidade de busca de instruções (Fetch Unit). No entanto, nenhuma falha pode ser detectada em algumas estruturas, como é o caso de ambos os estágios do preditor de desvios (FeBr) e (BPD), do buffer de reordenamento de desvios (BROB) e do módulo que faz a tradução de endereços lineares das instruções em endereços físicos (iTLB). Portanto, os módulos de hardware mais críticos são os estágios para resolução de desvios condicionais: além de não detectar nenhuma falha injetada nestas estruturas, a vulnerabilidade destas estruturas ainda aumenta quando a técnica VAR é utilizada.

A Figura 4.8 mostra a redução no percentual de falhas que causaram erros na saída

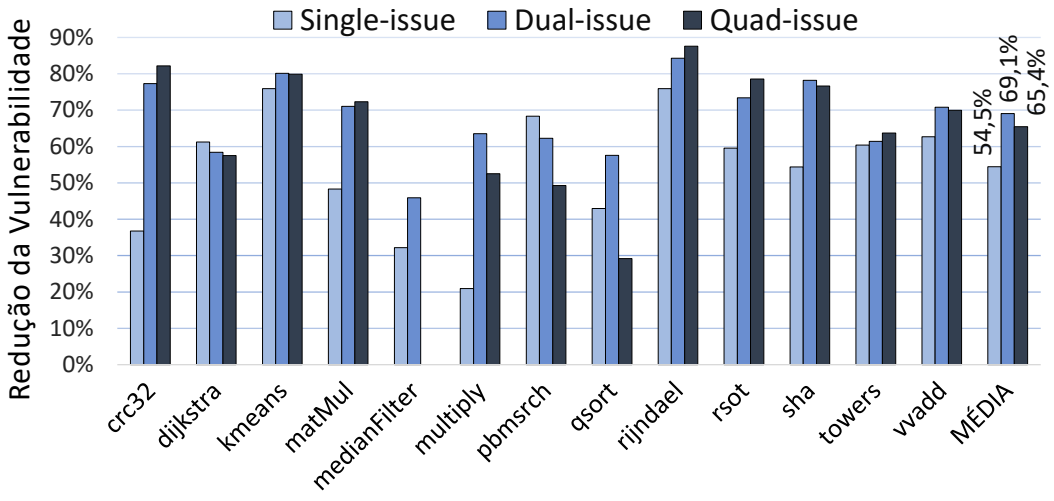
Figura 4.7: Falhas detectadas nas estruturas dos processadores utilizando a técnica VAR



Fonte: Elaborada pelo próprio autor.

dos três processadores enquanto executavam cada aplicação protegida com a técnica de tolerância a falhas VAR.

Figura 4.8: Redução da vulnerabilidade dos processadores executando cada aplicação protegida com a técnica VAR

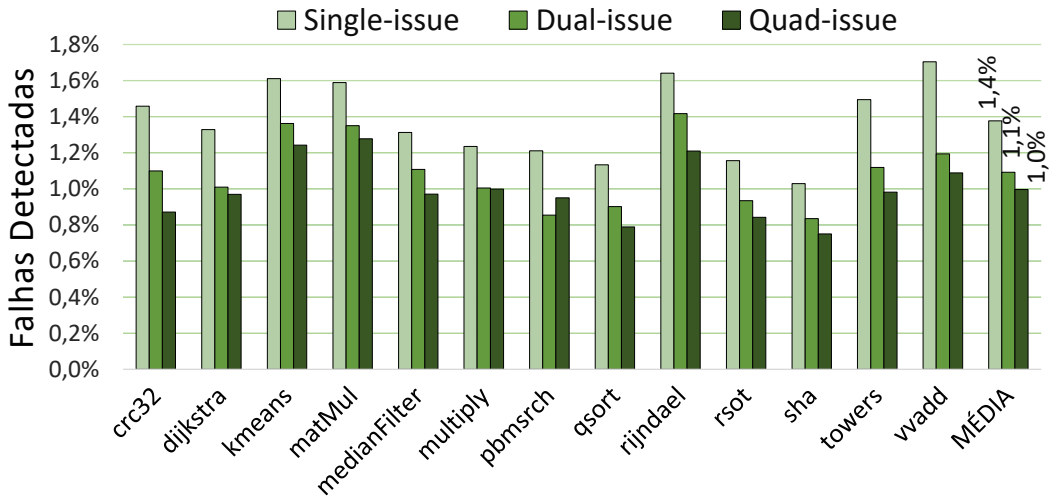


Fonte: Elaborada pelo próprio autor.

A porcentagem de falhas detectadas em cada processador enquanto executavam nosso *benchmark* protegido pela técnica VAR é mostrada na Figura 4.9. Com exceção do algoritmo para pesquisa de strings *pbmsrch*, todos os demais programas detectaram mais falhas no superescalar *single-issue*, e a menor detecção de falhas foi obtida na versão

quad-issue.

Figura 4.9: Falhas detectadas nos processadores executando cada aplicação com a técnica VAR



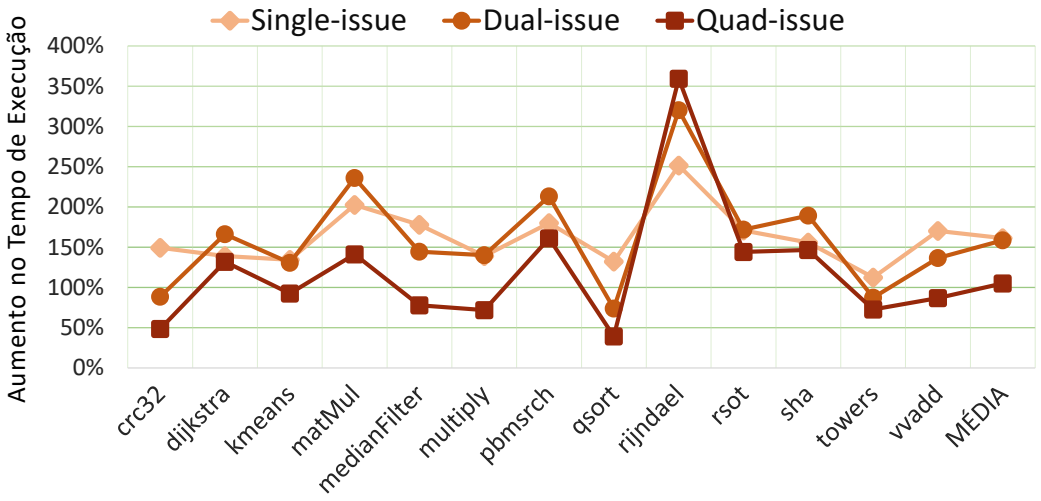
Fonte: Elaborada pelo próprio autor.

Embora a Figura 4.8 sugira que a técnica VAR alcance maior redução de vulnerabilidade no processador *dual-issue*, a Figura 4.9 mostra que esta técnica detecta mais falhas no *single-issue*. Isto ocorre porque o gargalo do *single-issue* é o estágio de execução, então as estruturas que estão antes do EXE estão mais ocupadas e mais propensas a falhas. Por se tratar de falhas localizadas antes da resolução dos desvios, a comparação de registradores favorece a detecção de mais falhas em estruturas que estão antes do EXE, como mostra a Figura 4.7. A exceção é a unidade de busca porque esta estrutura tem o mesmo tamanho para todos os processadores, como o *quad-issue* busca mais instruções por ciclo, neste processadores esta estrutura fica mais ocupada.

O *overhead* no tempo necessário para executar cada aplicação com a técnica VAR é apresentada na Figura 4.10. Esta figura mostra quanto, em porcentagem, a execução de cada programa protegido ficou mais lenta com relação ao tempo de execução das mesmas aplicações apresentadas na Tabela 3.5. Repare que o algoritmo de encriptação Rijndael AES é a aplicação que sofreu o maior *overhead*. Isto ocorre porque o IPC do Rijndael com a técnica VAR é bem menor do que o IPC deste programa desprotegido em todos os processadores. Em contrapartida, o algoritmo de ordenação Quick sort é o cenário onde ocorre o menor aumento do tempo de execução, pois o IPC deste programa é maior com a técnica do que sem a técnica.

A Figura 4.10 também demonstra que o *overhead* no tempo de execução é menor

Figura 4.10: Aumento do tempo para executar cada aplicação protegida com a técnica VAR



Fonte: Elaborada pelo próprio autor.

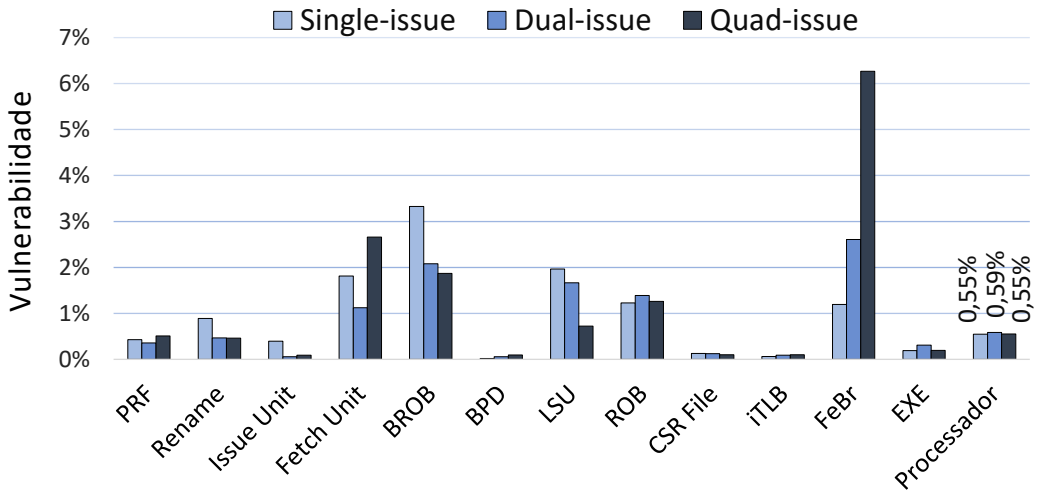
na configuração *quad-issue*. Isto ocorre porque a execução da maioria das aplicações desprotegidas atinge o limite de paralelismo já na versão *dual-issue*, deixando unidades de execução ociosas no *quad-issue*. Portanto, as instruções redundantes, inseridas pela técnica, podem ser resolvidas independentemente das instruções originais em unidades de execução que estavam ociosas no processador *quad-issue*, reduzindo o impacto no tempo de execução.

4.2.1.2 Eficiência da técnicas VAR_LS

A segunda técnica avaliada, VAR_LS, foi a primeira alternativa para minimizar os custos da técnica VAR. A vulnerabilidade de cada estrutura quando esta técnica foi utilizada é mostrada na Figura 4.11.

Com relação ao AVF do processador completamente desprotegido, a técnica VAR_LS é capaz de reduzir a vulnerabilidade da maioria dos módulos de hardware presentes nos superescalares, como mostra a Figura 4.12. Porém, a vulnerabilidade de algumas estruturas continua piorando quando aplicamos alguma técnica SIHFT e, novamente, este problema é mais acentuado nas estruturas relacionadas com o preditor de desvios. No entanto, ao utilizar a técnica VAR_LS, a vulnerabilidade dessas estruturas não aumenta tanto quanto aumenta com a técnica VAR. Isto ocorre porque a técnica VAR_LS insere 32% menos instruções para comparar registradores. Com menos instruções de compa-

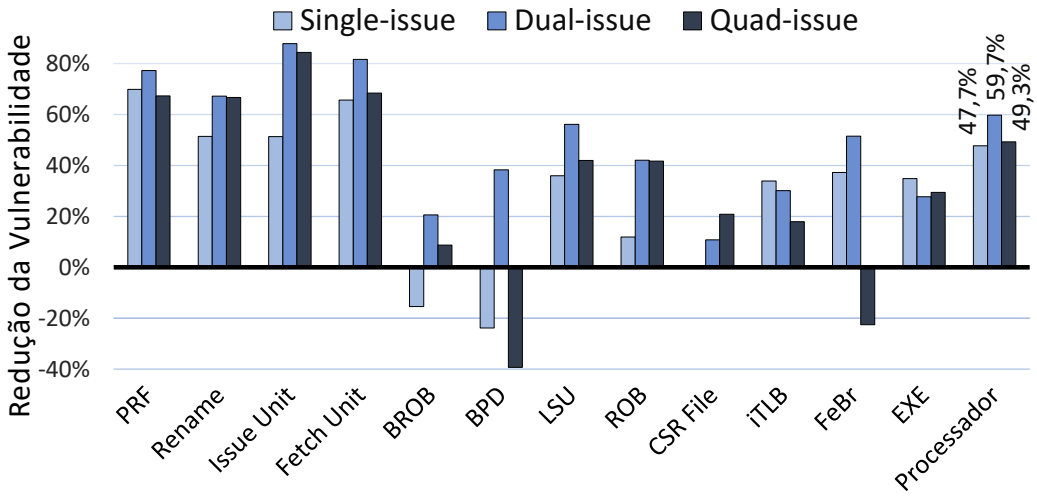
Figura 4.11: Vulnerabilidade de cada módulo de hardware quando a técnica VAR_LS foi utilizada



Fonte: Elaborada pelo próprio autor.

ração, a técnica VAR_LS estressa menos esta área do processador e, consequentemente, tem menos flip-flops com bits úteis nestas estruturas.

Figura 4.12: Redução da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica VAR_LS

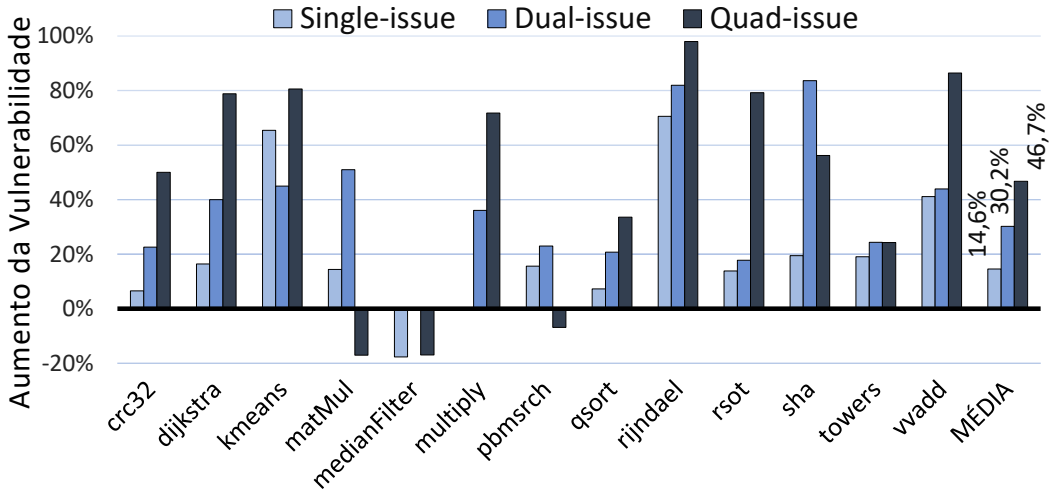


Fonte: Elaborada pelo próprio autor.

A Figura 4.13 diz respeito ao aumento da vulnerabilidade quando a técnica VAR_LS é utilizada, mas está relacionada com a vulnerabilidade da técnica VAR. Portanto, comparando com a vulnerabilidade dos processadores quando estão executando nosso *ben-*

chmark de aplicações protegido com a técnica VAR, esta figura exibe um aumento no percentual de falhas que causaram algum tipo de erro nas aplicações protegidas com a técnica VAR_LS.

Figura 4.13: Aumento da vulnerabilidade dos processadores executando cada aplicação com a técnica VAR_LS, com base na técnica VAR



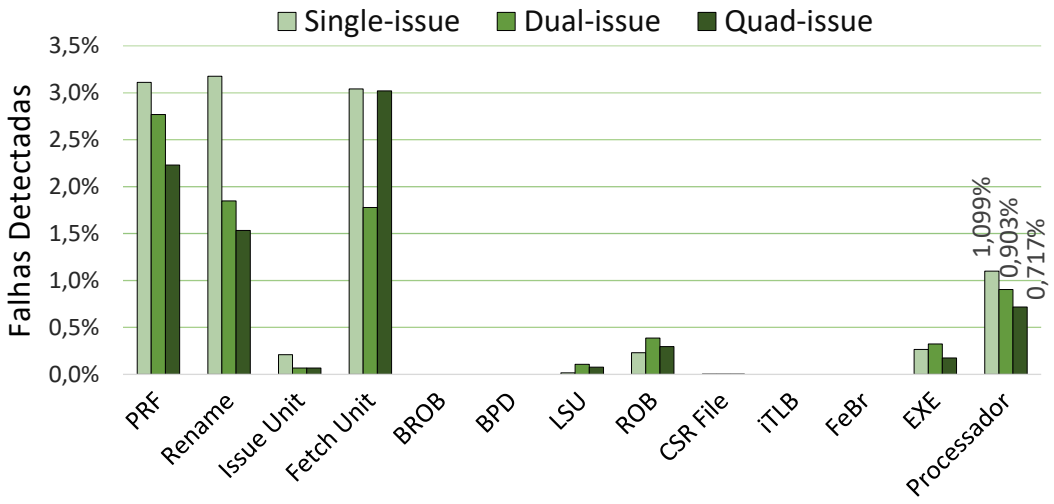
Fonte: Elaborada pelo próprio autor.

A maior semelhança entre as técnicas VAR e VAR_LS pode ser observada a partir da Figura 4.14. Porém, a porcentagem de falhas detectadas pela técnica VAR_LS diminui um pouco e, novamente, falhas injetadas em estruturas como os estágios do preditor de desvios (FeBr) e (BPD), no buffer de reordenamento de desvios (BROB) e (iTLB) nunca são detectadas. Quanto as falhas detectadas em todo o processador, a técnica VAR_LS detecta menos do que a técnica VAR em todas as configurações superescalares e, novamente, a estratégia utilizada para detecção de falhas é mais eficiente no processador superescalar *single-issue*.

Quando comparamos o tempo necessário para executar cada aplicação com as técnicas VAR e VAR_LS, a Figura 4.15 mostra o quanto a técnica VAR_LS é mais rápida. Portanto, a técnica VAR_LS causa menor impacto no desempenho dos processadores. O tempo necessário para executar a aplicação Rijndael chega a reduzir 50% na versão *quad-issue*. No entanto, o tempo de execução do *vvadd* com a técnica VAR_LS piora 15% com relação ao *vvadd* protegido com VAR no mesmo *quad-issue*.

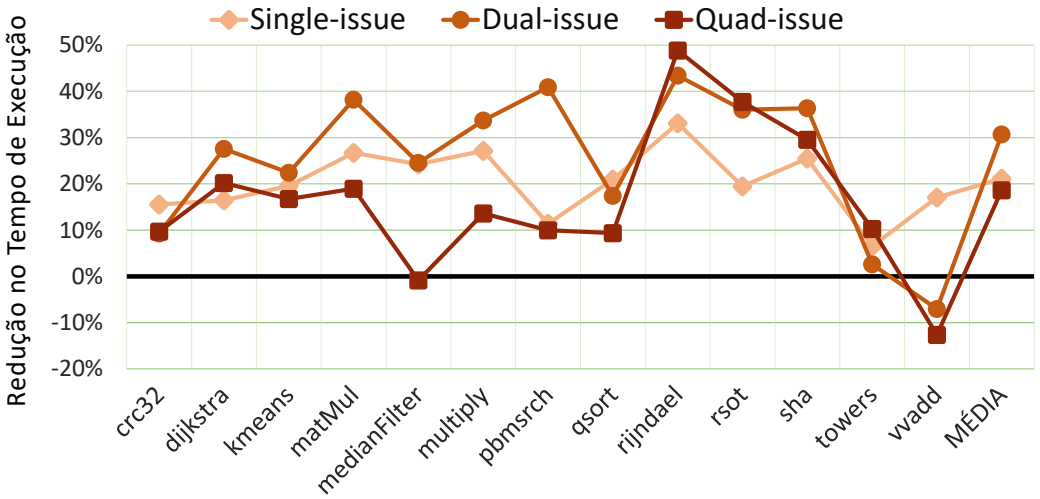
Portanto, quando comparada com a técnica VAR, apesar da técnica VAR_LS reduzir o tempo de execução, a detecção de falhas é menor, refletindo no aumento da vulnerabilidade dos processadores. Assim, após avaliarmos a técnica VAR_LS, constatamos

Figura 4.14: Falhas detectadas nas estruturas dos processadores utilizando a VAR_LS



Fonte: Elaborada pelo próprio autor.

Figura 4.15: Redução do tempo para executar cada aplicação protegida com a técnica VAR_LS, com base na técnica VAR



Fonte: Elaborada pelo próprio autor.

que o tempo de execução reduziu. Porém, a capacidade de detecção ficou comprometida, então novas alternativas foram experimentadas.

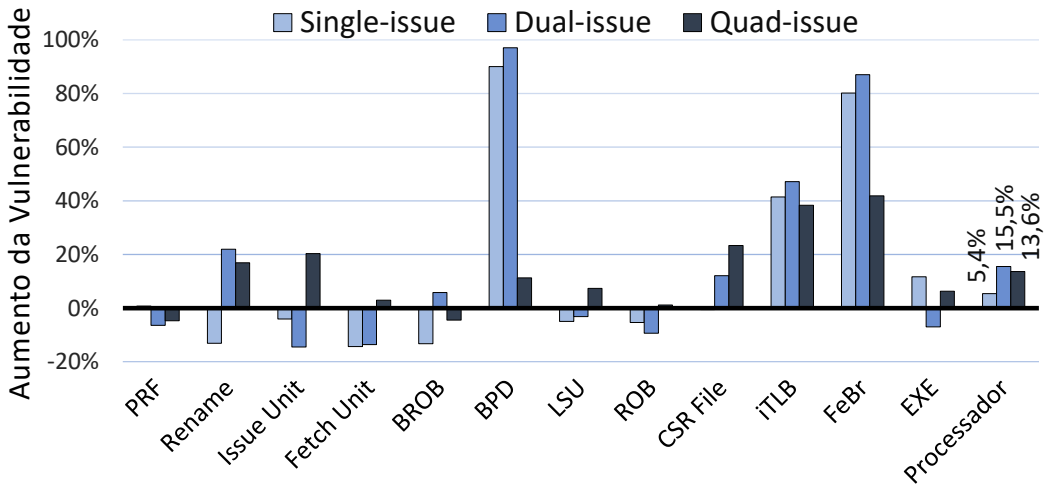
4.2.1.3 Eficiência da técnicas VAR_LSB

Como os estágios do preditor de saltos estavam piorando a vulnerabilidade, a próxima estratégia utilizada foi a técnica VAR_LSB. Como descrito na subseção 3.1.1, a

técnica VAR_LSB verifica, além dos registradores utilizados nas instruções *load* e *store*, o conteúdo dos registradores presentes nas instruções do tipo *Branch* também é monitorado.

Comparando a técnica VAR_LSB com a técnica VAR_LS, a Figura 4.16 demonstra que a técnica VAR_LSB causa o aumento da vulnerabilidade de todos os processadores. Este efeito é causado em decorrência do acréscimo de instruções para comparar o conteúdo dos registradores. Porém, devido a estas instruções extras, as estruturas utilizadas pelas instruções de desvio são mais exploradas, e isto se reflete no aumento da vulnerabilidade. O aumento da ocupação de uma estrutura leva ao aumento da probabilidade de que uma falha nesta estrutura resulte em uma execução incorreta.

Figura 4.16: Aumento da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica VAR_LSB, com base na técnica VAR_LS

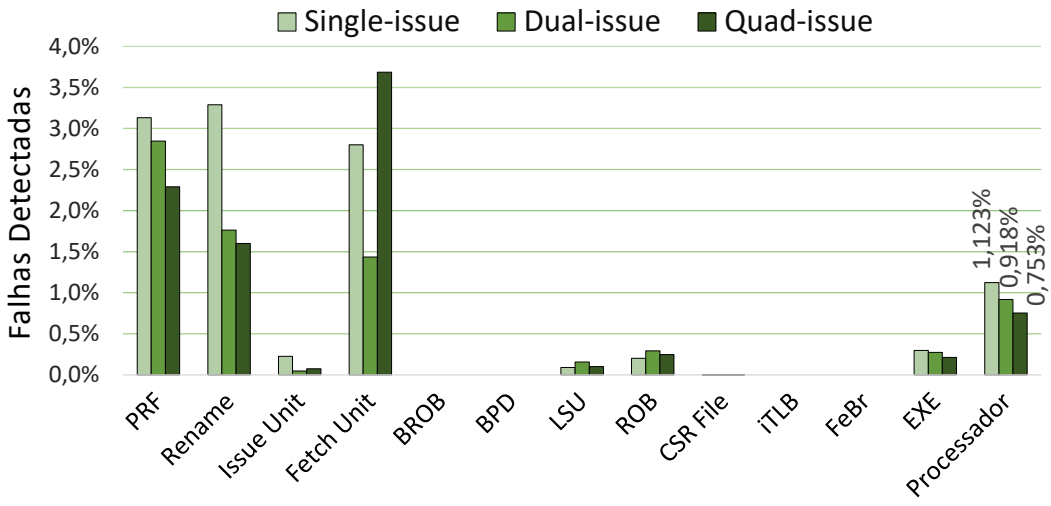


Fonte: Elaborada pelo próprio autor.

Embora haja um aumento da vulnerabilidade, a Figura 4.17 mostra que a técnica VAR_LSB é capaz de detectar mais falhas do que a técnica anterior, que não protege instruções de salto condicional. Isto ocorre porque esta técnica tem mais instruções que comparam o conteúdo dos registradores. Portanto, é importante destacar que não há uma correlação entre a quantidade de falhas detectadas e a redução de vulnerabilidade dos processadores.

A Figura 4.18 diz respeito ao aumento no tempo necessário para executar os programas com a técnica VAR_LSB, mas está relacionado com o tempo de execução da técnica VAR_LS. Portanto, além da técnica VAR_LSB piorar o desempenho, devido a mais instruções de comparação inseridas antes das instruções de desvio condicional, a

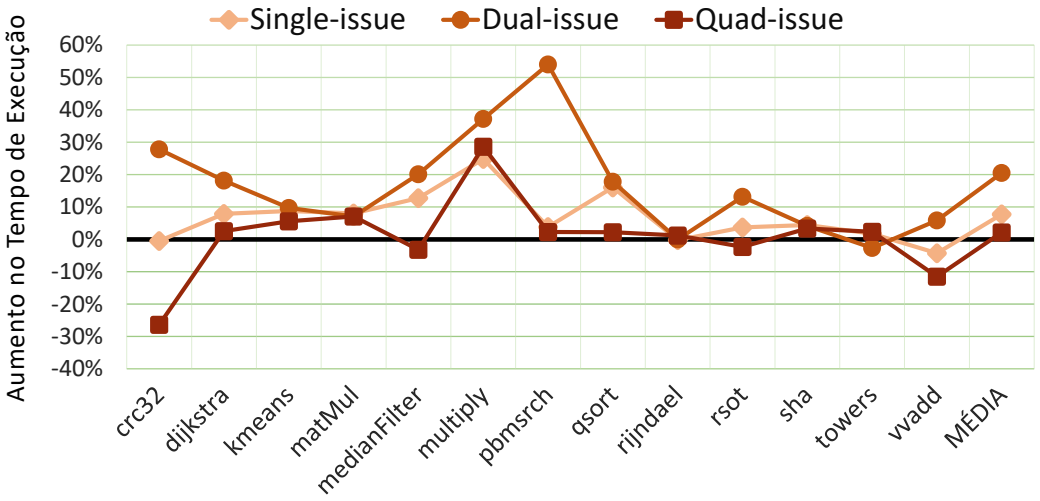
Figura 4.17: Falhas detectadas nas estruturas dos processadores utilizando a técnica VAR_LSB



Fonte: Elaborada pelo próprio autor.

Figura 4.18 mostra que, como esperado, a técnica VAR_LSB demanda mais tempo para executar nosso benchmark de aplicações. O maior aumento no tempo de execução ocorre no processador superescalar *dual-issue*.

Figura 4.18: Aumento do tempo para executar cada aplicação protegida com a técnica VAR_LSB, com base na técnica VAR_LS



Fonte: Elaborada pelo próprio autor.

O aumento no tempo de execução é mais acentuado no algoritmo que faz pesquisa de strings executando na versão *dual-issue*. Como esta técnica apenas aumentou

a quantidade de instruções de desvio condicional, este comportamento está diretamente relacionado com a precisão do preditor de desvios deste processador. Executando esta aplicação com a técnica VAR_LS, o preditor do *dual-issue* acerta 99% dos desvios executados. Quando protegida com a técnica VAR_LSB, o preditor acerta apenas 87% dos desvios. Esta evidência também pode ser observada, de maneira proporcional, com o CRC32.

Após analisarmos estas primeiras técnicas, fica evidente que o principal recurso utilizado pelas técnicas para detectar falhas, que são as instruções de desvio para comparar o conteúdo dos registradores, também causa um aumento da vulnerabilidade destas estruturas. A questão é que as técnicas inserem mais instruções para proteger as instruções originais, mas as instruções que comparam o conteúdo dos registradores ficam desprotegidas.

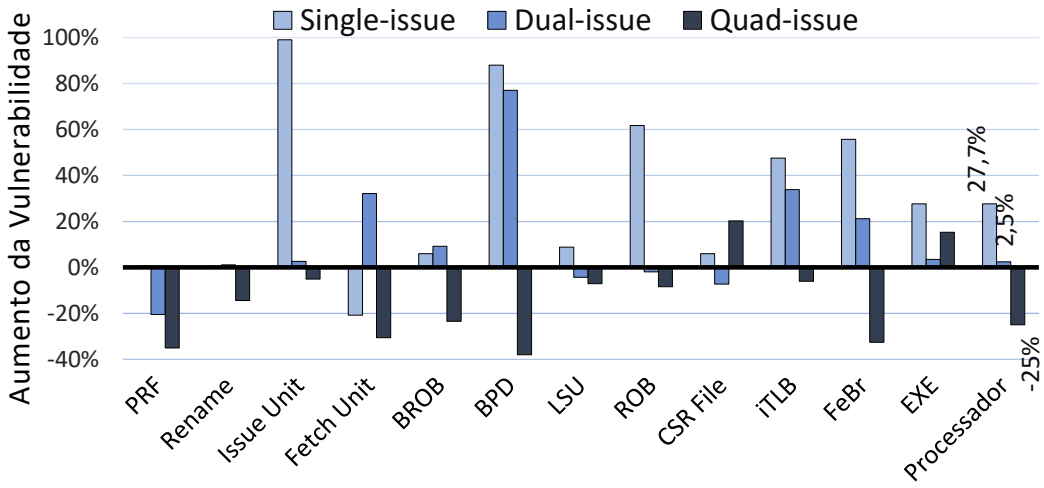
4.2.1.4 Eficiência da técnicas VARM_LS

A quarta técnica avaliada designada a proteção dos dados foi a VARM_LS. Devido ao aumento na vulnerabilidade dos processadores e ao aumento no tempo necessário para executar as aplicações com a técnica anterior (VAR_LSB), o desempenho e a eficiência da próxima técnica (VARM_LS) estão relacionados com a técnica VAR_LS. Lembramos que a diferença entre as técnicas VAR_LS e VARM_LS, é que esta última não duplica instruções de *load*. Ao invés disso, a técnica VARM_LS utiliza uma instrução *Move* para copiar o conteúdo do registrador com o dado buscado da memória para o respectivo registrador cópia. Isto reduz a densidade de *Loads*, aumenta o percentual de instruções aritméticas e mantém a quantidade de desvios condicionais.

A Tabela 3.2 apresentada na seção 3.3.1 mostra que o *quad-issue* só tem uma UE para o cálculo de endereços para a LSU. Quando trocamos a técnica VAR_LS pela técnica VARM_LS, reduzimos a quantidade de instruções do tipo *load*, então esta estrutura deixa de ser um gargalo e mais instruções podem ser executadas em paralelo. Isto se reflete na vulnerabilidade do processador, mostrada na Figura 4.19. Ao executar mais instruções em paralelo, as estruturas no pipeline do superescalar *quad-issue* logo ficam vazias e bit-flips aleatórios muitas vezes acabam por acertar flip-flops sem valores úteis. Portanto, vulnerabilidade do *quad-issue* é 25% menor quando usamos a técnica VARM_LS, ao invés da VAR_LS.

Este comportamento também pode ser observado na vulnerabilidade de quase todos os programas apresentados na Figura 4.20. O programa que realiza a multiplicação

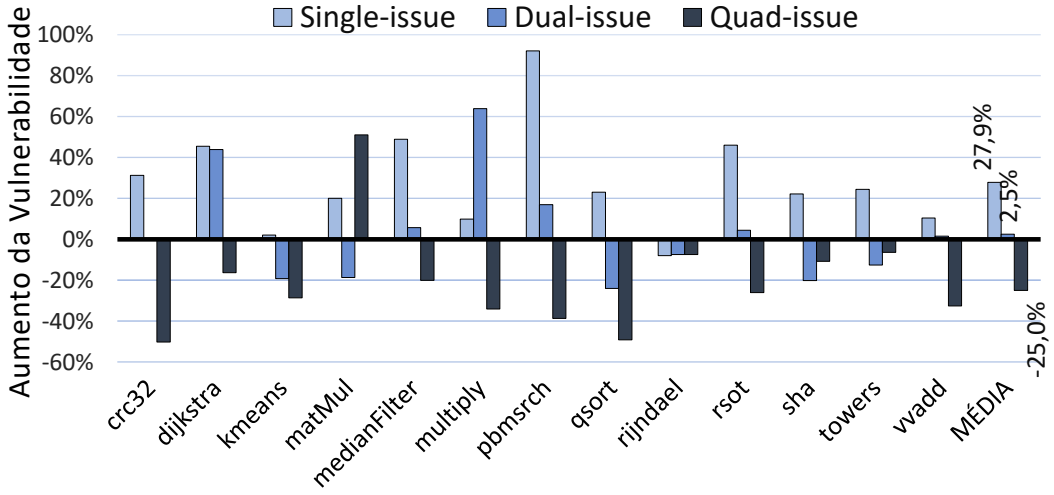
Figura 4.19: Aumento da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica VARM_LS, com base na técnica VAR_LS



Fonte: Elaborada pelo próprio autor.

de duas matrizes não segue este padrão, porque este já tem uma quantidade elevada de instruções aritméticas que exploram todas ALUs, além de ser uma aplicação com baixo percentual de instruções dos tipos *load* e *store*.

Figura 4.20: Aumento da vulnerabilidade dos processadores executando cada aplicação com a técnica VARM_LS, com base na técnica VAR_LS



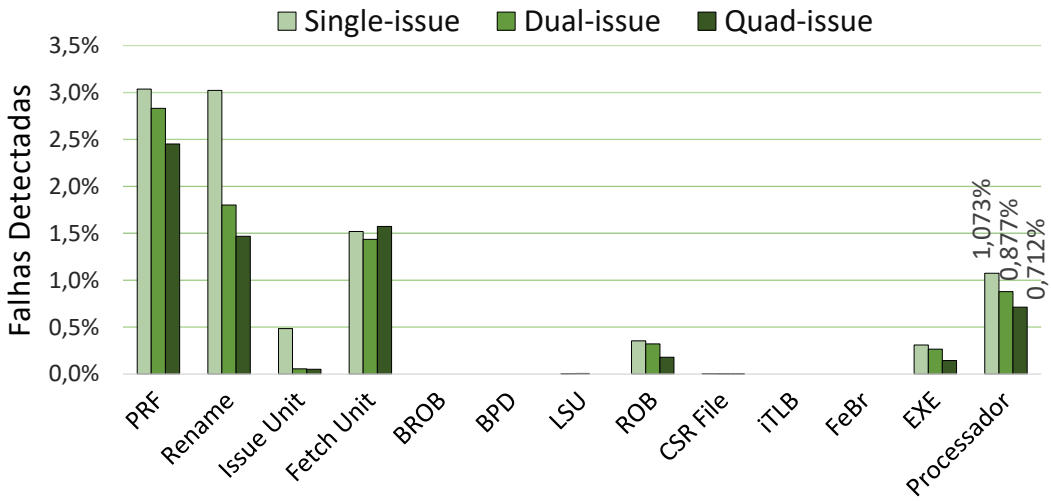
Fonte: Elaborada pelo próprio autor.

Por outro lado, o processador *single-issue* tem mais instruções prontas para serem executadas. Porém, devido ao gargalo na unidade de execução, algumas estruturas ficam

cheias de informações úteis, e bit-flips nestas estruturas resultam na execução incorreta de algumas aplicações. Isto não ocorre na unidade que busca instruções da memória, porque apenas uma instrução é carregada por ciclo e logo é consumida. Além disso, a unidade de busca conta com uma fila de 4 posições, mas a maioria fica vazia durante a execução.

O percentual de falhas injetadas em cada estrutura, que foram detectadas pela técnica VARM_LS, é mostrado na Figura 4.21.

Figura 4.21: Falhas detectadas nas estruturas dos processadores utilizando a técnica VARM_LS



Fonte: Elaborada pelo próprio autor.

Considerando cada processador por completo, a eficiência da técnica VARM_LS é um pouco menor do que todas as técnicas anteriores. Com relação à técnica VAR, a estratégia utilizada por VARM_LS detecta menos falhas em estruturas como PRF, Rename e LSU. Porém, ela detecta mais falhas na unidade de despacho e no buffer de reordenamento.

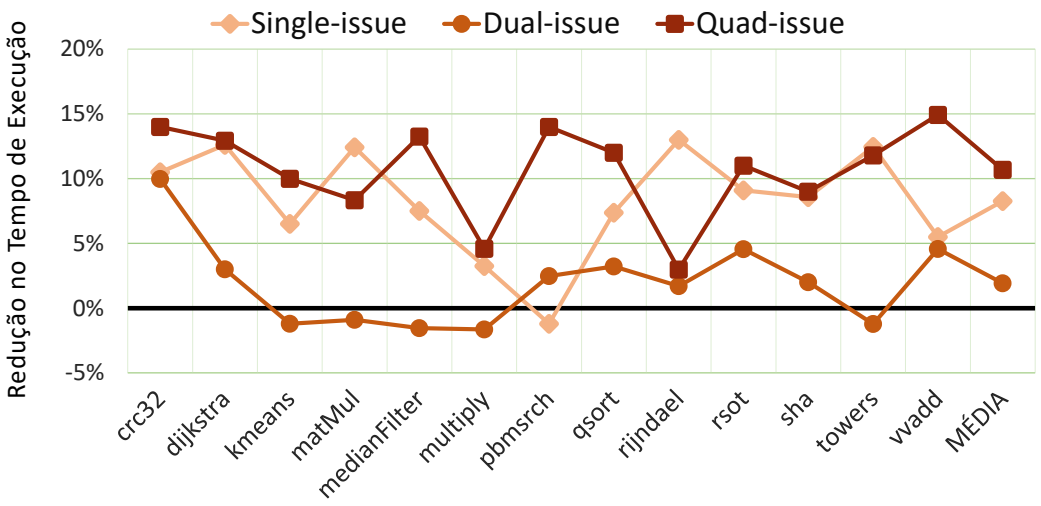
Com relação à técnica VAR_LS, ao utilizar a técnica VARM_LS, houve redução na quantidade de falhas detectadas em todos os processadores. Esta redução é mais acentuada no superescalar *quad-issue*. Porém, as Figuras 4.19 e 4.20 mostram que VARM_LS foi capaz de melhorar a resiliência deste processador.

Ainda relacionado com a técnica VAR_LS: VARM_LS reduz a detecção de falhas na unidade de busca, na Rename e na LSU. Para as demais estruturas, quando VARM_LS consegue detectar mais falhas em alguma estrutura do *single-issue*, ela também piora a detecção de falhas na mesma estrutura nos outros dois processadores.

A Figura 4.22 diz respeito a redução no tempo necessário para executar as aplica-

ções com a técnica VARM_LS. Esta redução está relacionada com o tempo de execução da técnica VAR_LS. No *quad-issue*, por exemplo, o tempo gasto para executar as aplicações protegidas com a técnica VARM_LS é 11% menor do que quando a técnica VAR_LS foi utilizada.

Figura 4.22: Redução do tempo para executar cada aplicação protegida com a técnica VARM_LS, com base na técnica VAR_LS



Fonte: Elaborada pelo próprio autor.

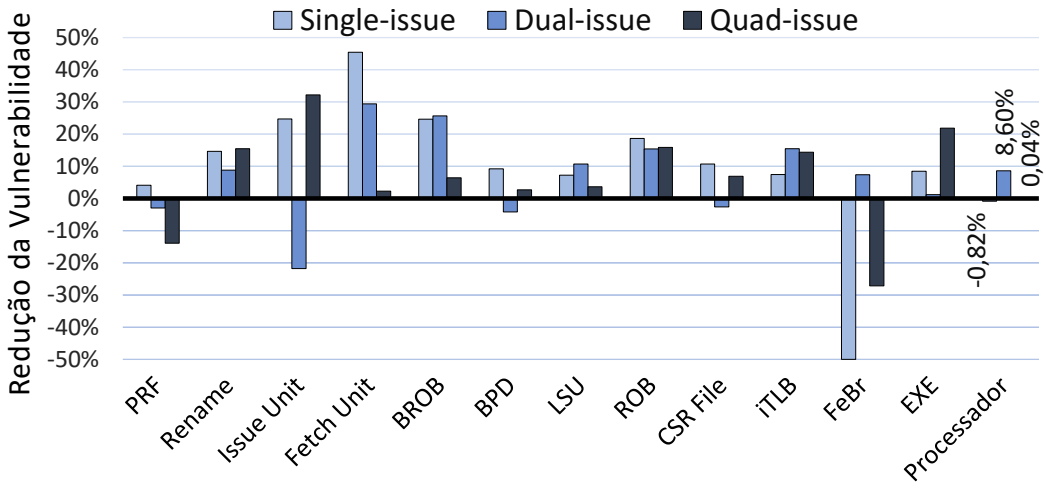
Portanto, além da técnica VARM_LS aumentar a capacidade de resiliência do processador *quad-issue*, ela também é capaz de reduzir o tempo de execução deste processador.

4.2.1.5 Eficiência da técnicas VARM_LSB

A última técnica avaliada, que duplica todas as instruções e o conteúdo dos registradores para identificar inconsistências nos dados, foi a técnica VARM_LSB. Quando comparamos a técnica VARM_LS com a VARM_LSB, a Figura 4.23 mostra uma leve redução na vulnerabilidade em quase todas as estruturas.

Ainda com relação à técnica anterior, VARM_LSB consegue reduzir a quantidade de erros na saída do processador *dual-issue* em 8,6%. O que não acontece quando trocamos da técnica VAR_LS para VAR_LSB. Porém, a vulnerabilidade nos outros dois processadores permanece quase inalterável e, novamente, isso não é possível devido ao aumento de erros causados por falhas injetadas nas estruturas relacionadas com o preditor de saltos. Este efeito também pôde ser observado entre as técnicas VAR_LS e VAR_LSB,

Figura 4.23: Redução da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica VARM_LSB, com base na técnica VARM_LS



Fonte: Elaborada pelo próprio autor.

devido ao aumento na quantidade de instruções para monitorar o conteúdo dos registradores.

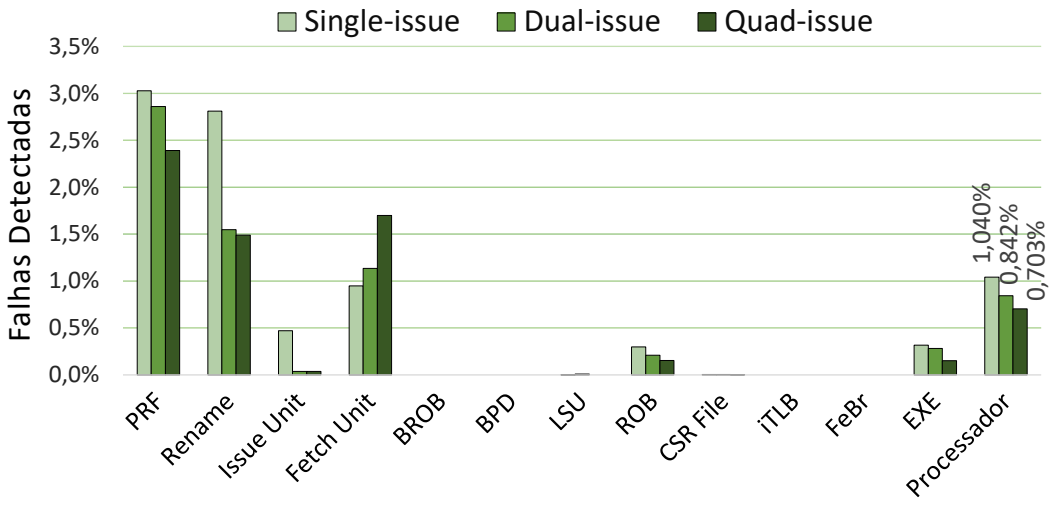
A Figura 4.24 mostra a capacidade de detecção de falhas obtida com a técnica VARM_LSB. Novamente, é possível observar que não há uma relação direta entre as falhas detectadas e a redução da vulnerabilidade dos processadores. A razão para isso é que, apesar da técnica VARM_LSB melhorar a capacidade de resiliência dos processadores *dual-* e *quad-issue*, a quantidade de falhas detectadas por esta é menor do que todas as técnicas anteriores, em todas as três versões superescalares.

A Figura 4.25 mostra que o tempo de necessário para executar cada aplicação protegida com a técnica VARM_LSB é maior do que o tempo gasto para executar as mesmas aplicações com a técnica VARM_LS.

É possível observar que o impacto no tempo de execução é menor em alguns programas executando no *quad-issue*. Este comportamento é observado exatamente nos mesmos programas que levam mais tempo para executar no *quad-issue* do que no *dual-issue* quando nenhuma técnica é utilizada. Isto ocorre porque o preditor de desvios do *quad-issue* melhorou um pouco.

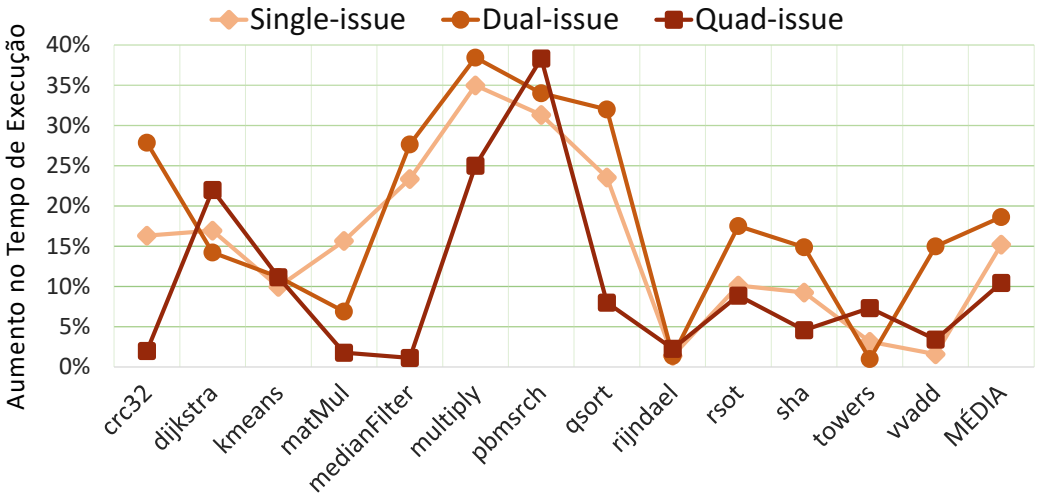
Este comportamento não pôde ser observado com a técnica VAR_LSB na Figura 4.18, devido à grande quantidade de instruções do tipo *load* que sobrecarrega a unidade que calcula endereços para as instruções *load* e *store*. Ainda, na Figura 4.18, o impacto reduzido no *overhead* de tempo só ocorre com o CRC32 executando no *quad-issue*, por-

Figura 4.24: Falhas detectadas nas estruturas dos processadores utilizando a técnica VARM_LSB



Fonte: Elaborada pelo próprio autor.

Figura 4.25: Aumento do tempo para executar cada aplicação protegida com a técnica VARM_LSB, com base na técnica VARM_LS



Fonte: Elaborada pelo próprio autor.

que dentre os programas sem técnica que demoram mais no *quad-issue*, o CRC32 é o que tem menos instruções do tipo *load*.

Portanto, mesmo que a técnica VARM_LSB consiga reduzir um pouco a vulnerabilidade de dois processadores, esta é técnica para proteção dos dados que menos detecta falhas, e que ainda resulta em um aumento considerável no tempo necessário para executar todo o *benchmark* de aplicações.

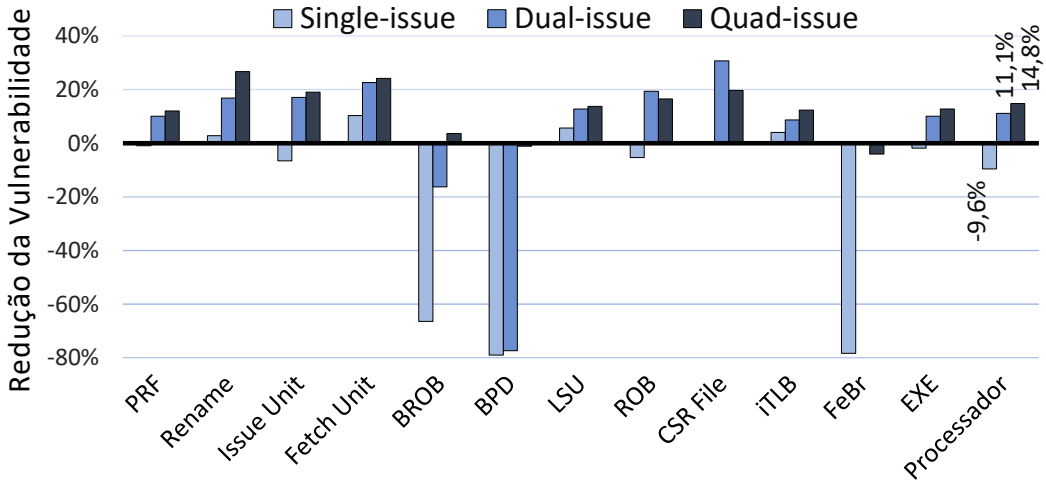
4.2.2 Proteção do fluxo de controle

Para detectar falhas no fluxo de controle de uma aplicação durante sua execução em um superescalar, duas técnicas foram avaliadas. A estratégia para detecção de falhas empregada por estas técnicas está descrito na subseção 3.1.2, e a análise de cada técnica será apresentada nas subseções a seguir.

4.2.2.1 Eficiência da técnica BRA

A técnica BRA é projetada para garantir que as instruções de desvio condicional saltem para o bloco básico esperado em um fluxo de controle sem erros no programa. A Figura 4.26 mostra que usar a técnica BRA para proteger nosso *benchmark* de aplicações reduz em aproximadamente 10% a quantidade de falhas que resultam em erros na maioria das estruturas.

Figura 4.26: Redução da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica BRA



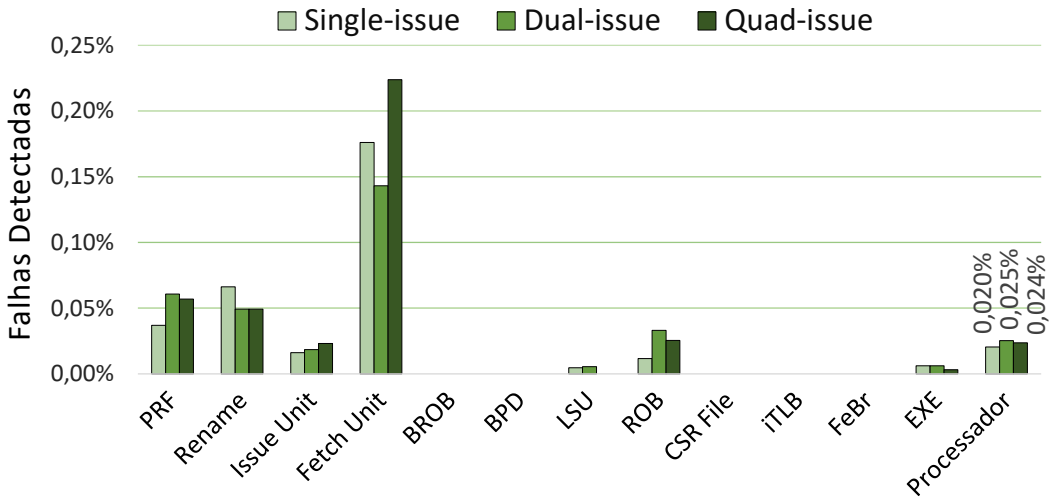
Fonte: Elaborada pelo próprio autor.

Porém, para a técnica BRA preservar a integridade da instrução original é necessário acrescentar outras duas instruções de desvio condicional, além de um salto incondicional. Estas instruções extras agora também fazem parte do fluxo de execução do programa, e também estão vulneráveis, mas ficam desprotegidas. Devido a esta deficiência, usar a técnica BRA aumenta muito a vulnerabilidade das estruturas relacionadas com o preditor de desvios. Isto se reflete nos processadores como um todo, principalmente

no superescalar *single-issue*, onde a porcentagem de falhas que causam erros aumentou 9,6%.

A Figura 4.27 mostra que a técnica BRA não é capaz de detectar a grande maioria das falhas. Esta ineficiência na taxa de falhas detectadas se deve ao fato de que esta técnica só é capaz de detectar falhas que ocorrem nos registradores utilizados pelas instruções de desvio. A detecção também só é possível, se e somente se, a falha ocorrer no momento em que o registrador estiver sendo usado pela instrução de desvio.

Figura 4.27: Falhas detectadas nas estruturas dos processadores utilizando a técnica BRA



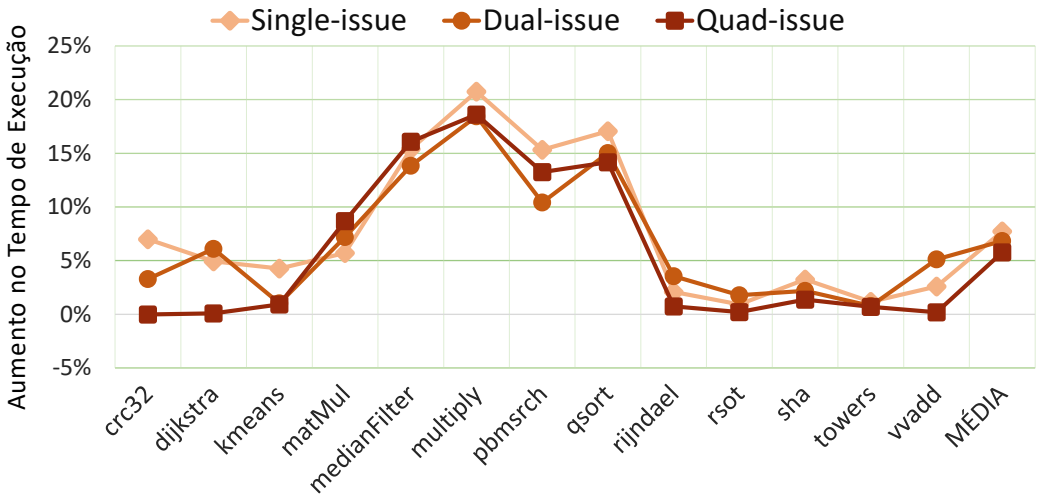
Fonte: Elaborada pelo próprio autor.

Portanto, além da técnica BRA aumentar a vulnerabilidade das estruturas BROB, BPD e FeBr, nenhuma falha é detectada nestas mesmas estruturas. Diferente das técnicas anteriores, que detectavam mais falhas no banco de registradores físicos, esta técnica é mais eficiente na detecção de falhas na unidade que busca instruções da memória.

Com relação ao tempo de execução das aplicações sem técnica, apresentado na Tabela 3.5, a Figura 4.28 mostra que a técnica BRA apresenta um impacto mais tênue no tempo de execução.

Repare que o tempo necessário para executar alguns programas aumenta aproximadamente 20%, enquanto que outros apresentam um impacto bem mais suave. Esta diferença é causada devido a diferença no percentual de instruções de desvio executada em cada programa.

Figura 4.28: Aumento do tempo para executar cada aplicação protegida com a técnica BRA



Fonte: Elaborada pelo próprio autor.

4.2.2.2 Eficiência da técnica SIG

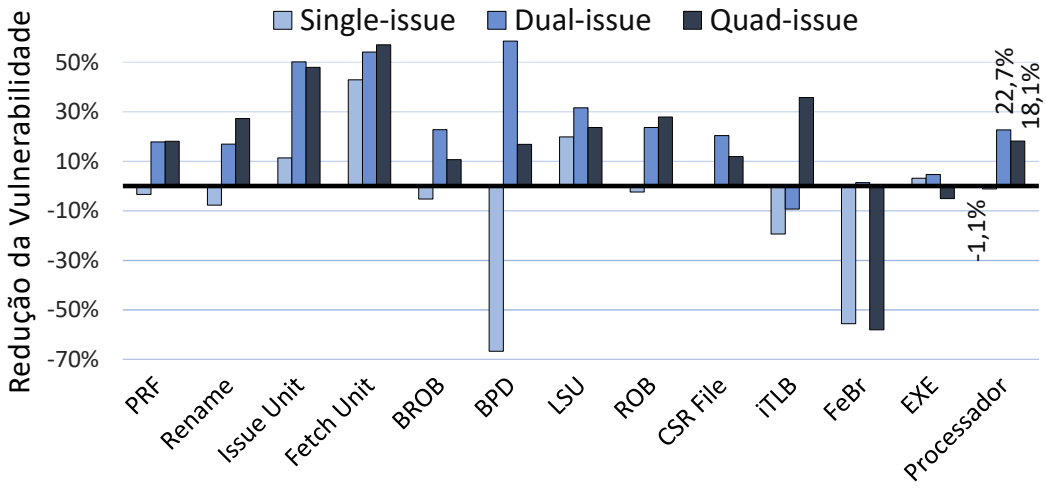
A segunda opção de técnica para proteção do fluxo de controle é a técnica SIG. Esta visa identificar saltos incorretos entre os blocos básicos. Diferentemente da técnica BRA, onde nenhum registrador extra é necessário, a estratégia utilizada pela técnica SIG precisa de dois registradores para proteger um bloco básico.

Com base na vulnerabilidade das estruturas quando nenhuma técnica é utilizada, apresentada na Figura 4.2, a Figura 4.29 mostra que técnica SIG é capaz de reduzir a vulnerabilidade de quase todas as estruturas. Inclusive, é possível aumentar a capacidade de resiliência dos superescalares *dual-issue* e *quad-issue*. Porém, ao proteger os programas com a técnica SIG, a vulnerabilidade da unidade de execução destes processadores é maior do que quando a técnica BRA é utilizada.

A Figura 4.30 mostra o percentual de falhas injetadas que foram detectadas pela técnica SIG. Da mesma forma que a técnica BRA, a técnica SIG detecta mais falhas na unidade que busca instruções da memória. Contudo, a estratégia utilizada pela técnica SIG consegue detectar mais falhas do a técnica BRA em todas as estruturas.

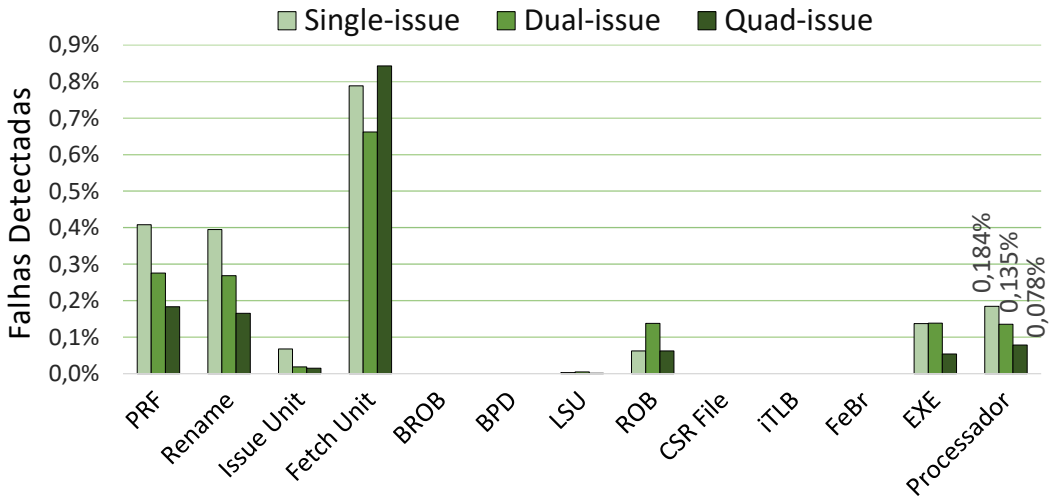
Assim como ocorre em todas as técnicas de proteção dos dados, a técnica para proteção do controle SIG detecta mais falhas no superescalar *single-issue*. Apesar disso, a vulnerabilidade deste processador aumenta quando a técnica SIG é utilizada nos programas. Isto mostra, mais uma vez, que a quantidade de falhas detectadas pela técnica não

Figura 4.29: Redução da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica SIG



Fonte: Elaborada pelo próprio autor.

Figura 4.30: Falhas detectadas nas estruturas dos processadores utilizando a técnica SIG



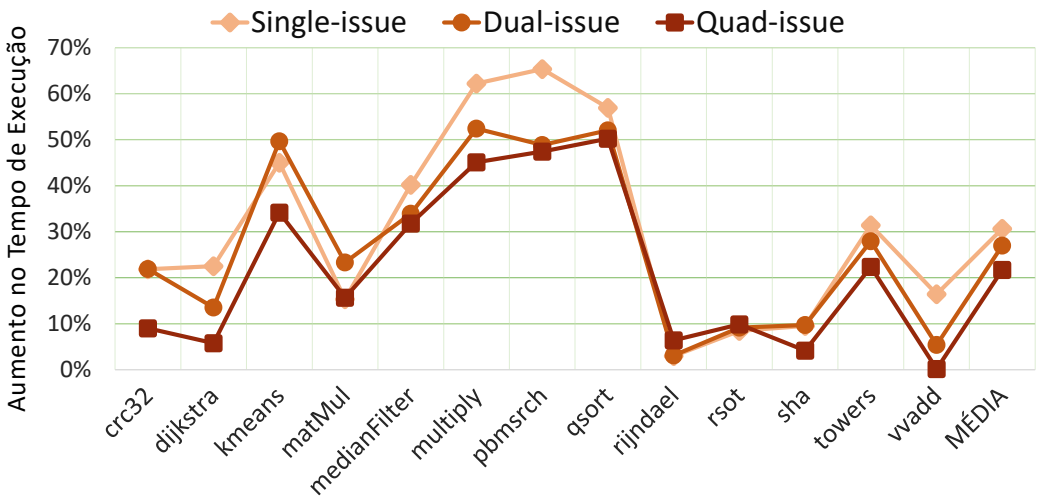
Fonte: Elaborada pelo próprio autor.

está diretamente relacionada com a redução da vulnerabilidade dos processadores.

O impacto no tempo de execução das aplicações protegidas com a técnica SIG, mostrado na Figura 4.31, é semelhante ao observado com a técnica BRA. Isto se deve ao fato do *overhead* de instruções extras, nestas técnicas, está diretamente relacionada com a quantidade de instruções de controle. O tempo necessário para executar a técnica SIG é um pouco mais acentuado devido ao acréscimo de instruções aritméticas e saltos

incondicionais. A diferença é maior nos programas *towers* e *kmeans*, pois estes têm maior percentual de instruções de salto incondicional, o que se reflete em um aumento de blocos básicos executados.

Figura 4.31: Aumento do tempo para executar cada aplicação protegida com a técnica SIG



Fonte: Elaborada pelo próprio autor.

Repare que a técnica SIG consegue ser mais eficiente do que a técnica BRA, porém a SIG causa maior impacto no tempo de execução. O problema de aumentar o tempo de execução é que a aplicação mesmo protegida fica mais tempo exposta a falhas causadas pela radiação.

A eficiência das técnicas para proteção do fluxo de controle é mais apropriada para aumentar a capacidade de resiliência da unidade que busca instruções da memória, pois além de detectarem muito mais falhas nesta estrutura, elas são capazes de reduzir a vulnerabilidade deste módulo de hardware.

Devido ao aumento da vulnerabilidade do superescalar *single-issue*, as técnicas projetadas para proteger o fluxo de controle não são recomendadas para este processador. Isso corrobora com o que foi apresentado em (SCHUSTER et al., 2017) que mostra a ineficiência das técnicas SIHFT para proteção do controle, e com os trabalhos mencionados na seção 2.1.3 que combinam estas técnicas com técnicas para detecção de falhas nos dados para aumentar a detecção de falhas.

4.2.3 Proteção do fluxo de dados e de controle

A combinação de técnicas para detecção de falhas, tanto nos dados quanto no controle, foi realizada com o emprego das técnicas VAR e BRA. A técnica BRA foi escolhida porque seu impacto no tempo de execução é menor, quando comparada com a outra técnica de proteção do controle. A técnica VAR atinge boa cobertura de falhas, mas aumenta muito o tempo de execução. Então, utilizamos esta técnica com a restrição de verificar somente os registradores utilizados pelas instruções que armazenam dados na memória. Desta forma, reduzimos o *overhead* de instruções de desvio condicional, utilizadas para comparar registradores.

4.2.3.1 Eficiência da técnica VAR_S_BRA

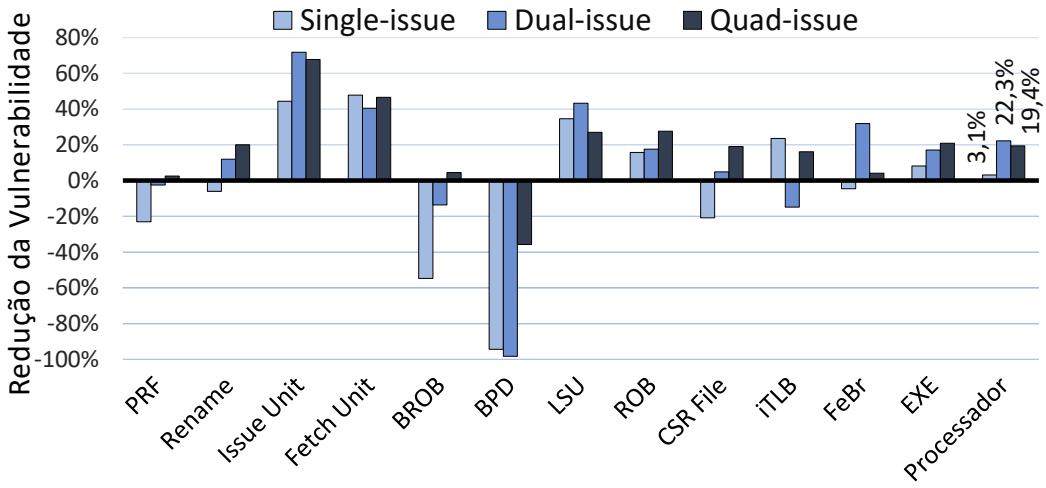
A primeira técnica utilizada para minimizar a quantidade de erros nos dados e no controle foi a técnica VAR_S_BRA. Com esta técnica, além de duplicar todos os registradores, todas as instruções também são duplicadas, inclusive as instruções de desvio condicional. Ao utilizar esta técnica é possível integrar a boa cobertura de falhas da técnica VAR com o baixo custo da técnica BRA.

Tomando como base a vulnerabilidade das três versões do processador superescalar BOOM quando nenhuma técnica de tolerância a falhas é utilizada, a Figura 4.32 apresenta a redução de vulnerabilidade, obtida com a técnica VAR_S_BRA, nas estruturas dos três processadores.

A técnica VAR_S_BRA inclui a estratégia utilizada pela técnica BRA. Contudo, quando a técnica VAR_S_BRA é utilizada para proteger os programas, a vulnerabilidade do banco de registradores físicos e da unidade de renomeação de registradores é maior do que a vulnerabilidade destas estruturas quando somente a técnica BRA é aplicada. Isto ocorre porque a técnica BRA não duplica registradores, enquanto que a técnica VAR_S_BRA utiliza o dobro de registradores, aumentando muito o uso destas estruturas. Além disso, a Figura 4.33 mostra que o percentual de falhas detectadas pela técnica VAR_S_BRA, nestas estruturas, é apenas metade do que é possível com as técnicas projetadas para proteção dos dados e, portanto, o mesmo comportamento não é observado nas demais técnicas que duplicam todos registradores.

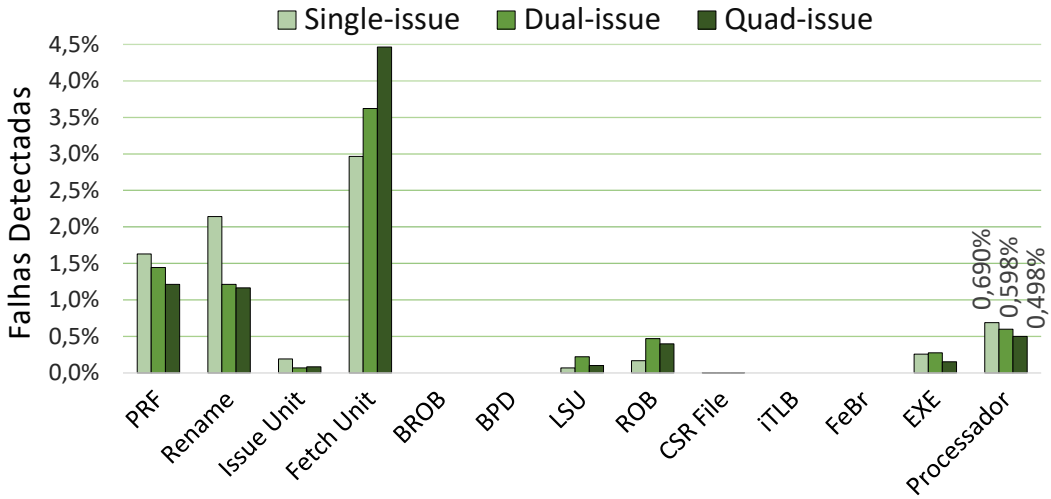
Apesar da vulnerabilidade das estruturas reduzir só um pouco quando a técnica VAR_S_BRA é utilizada, com esta técnica é possível atingir uma ótima cobertura de falhas no módulo de hardware que busca instruções da memória. Isto se deve ao fato

Figura 4.32: Redução da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica VAR_S_BRA



Fonte: Elaborada pelo próprio autor.

Figura 4.33: Falhas detectadas nas estruturas dos processadores utilizando a técnica VAR_S_BRA



Fonte: Elaborada pelo próprio autor.

desta estrutura buscar instruções ciclo a ciclo, mas não pode enviar para execução devido ao gargalo provocado pelo *overhead* de instruções do tipo *load*. Desta forma, a fila de instruções nesta estrutura fica quase o tempo todo cheia. Assim, as falhas nesta estrutura poderiam resultar em erros. Porém, como esta estrutura está localizada no início do pipeline, as falhas podem ser detectadas pela técnica.

Assim como todas as outras técnicas, quando a técnica VAR_S_BRA é aplicada,

a taxa de falhas detectadas na unidade de despacho é bem escassa. Em contrapartida, quando a proteção é feita com a técnica VAR_S_BRA, a maior redução da vulnerabilidade é obtida justamente nesta estrutura. A explicação para este comportamento é que quando não há técnicas, há muita dependência verdadeira e o paralelismo no nível das instruções é baixo. Então, a janela de instruções fica cheia de instruções esperando pelo conteúdo de registradores que ainda serão calculados. Quando a unidade de execução libera o valor de algum registrador, este conteúdo é levado pela lógica de adiantamento diretamente para a instrução que está esperando. Enquanto a instrução espera por um registrador fonte, o conteúdo dos outros registradores fontes estão vulneráveis.

Ao utilizar técnicas implementadas em software que duplicam instruções, o paralelismo no nível das instruções aumenta. Desta forma, as instruções ficam menos tempo na janela de instruções esperando para serem executadas e, portanto, a maioria dos registradores são lidos e logo a instrução é executada. Por consequência, bit-flips aleatórios nesta estrutura muitas vezes acertam flip-flops sem conteúdo relevante para a execução da aplicação. Com poucas instruções vulneráveis na fila de instruções, as técnicas não podem detectar falhas que não afetam as instruções.

A Figura 4.34 mostra o aumento no tempo necessário para executar nosso *benchmark* de aplicações protegido com a técnica VAR_S_BRA. O aumento no tempo de execução é maior no superescalar *single-issue* e é mais acentuado naqueles programas que atingem alto paralelismo no nível das instruções quando nenhuma técnica é utilizada. Quando a técnica é aplicada, estes programas não têm unidades ociosas para alocar as instruções e acabam sofrendo maior impacto no tempo de execução.

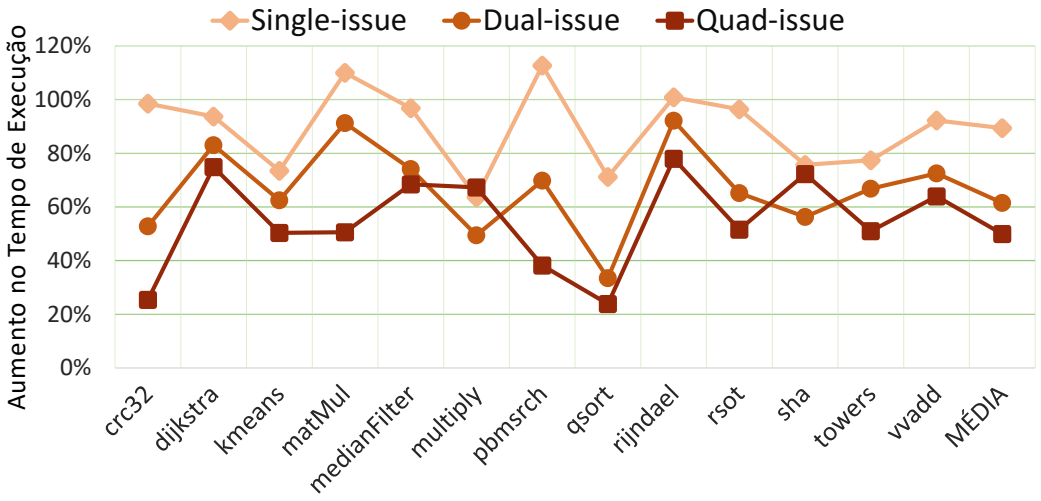
A técnica SIG apresenta uma vulnerabilidade semelhante a obtida com a técnica VAR_S_BRA, mas com custo reduzido. Assim, SIG poderia facilmente substituir a técnica VAR_S_BRA.

Com base na vulnerabilidade dos processadores desprotegidos, a técnica VAR_S_BRA consegue reduzir a vulnerabilidade dos três superescalares avaliados. Porém, este aumento na capacidade de resiliência é bem sutil e, devido ao custo de tempo para executar as aplicações com esta técnica, o seu uso não se justifica.

4.2.3.2 Eficiência da técnica VARM_S_BRA

A segunda técnica avaliada, que combina proteção dos dados e do controle, foi a técnica VARM_S_BRA. Esta técnica é semelhante a anterior, mas substitui a cópia das instruções do tipo *load* por instruções do tipo *move*. Com base na vulnerabilidade

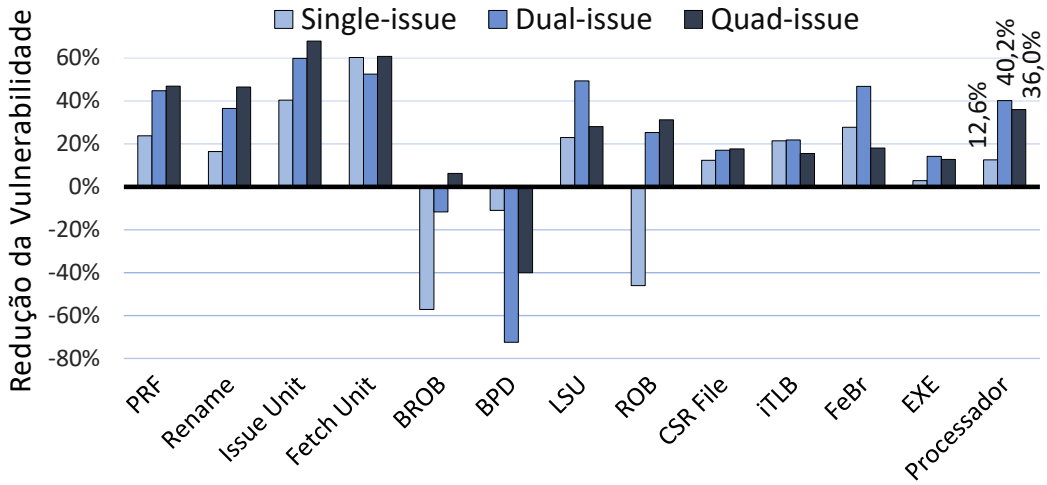
Figura 4.34: Aumento do tempo para executar cada aplicação protegida com a técnica VAR_S_BRA



Fonte: Elaborada pelo próprio autor.

dos processadores executando aplicações sem nenhuma técnica de tolerância a falhas, a Figura 4.35 mostra a redução de vulnerabilidade, obtida com a técnica VARM_S_BRA, nas estruturas dos três processadores.

Figura 4.35: Redução da vulnerabilidade das estruturas dos processadores executando as aplicações protegidas com a técnica VARM_S_BRA



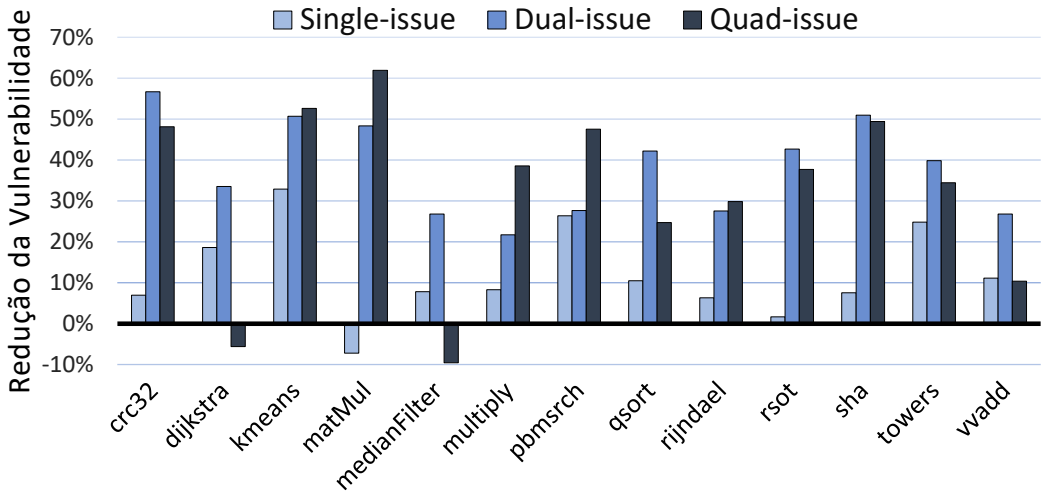
Fonte: Elaborada pelo próprio autor.

Quando comparada com a técnica VAR_S_BRA, é possível notar que a técnica VARM_S_BRA consegue aumentar capacidade de resiliência de todas as três versões

do superescalar BOOM. Isto se deve ao fato de que vulnerabilidade é minimizada em quase todas as estruturas. No entanto, a técnica anterior é mais eficiente para reduzir a quantidade de falhas na unidade de *load* e *store*.

A eficiência obtida com a técnica VARM_S_BRA para reduzir a vulnerabilidade das estruturas se reflete na vulnerabilidade dos processadores ao executar cada programa individualmente. A Figura 4.36 mostra esta redução da vulnerabilidade para cada aplicação.

Figura 4.36: Redução da vulnerabilidade dos processadores executando cada aplicação protegida com a técnica VARM_S_BRA



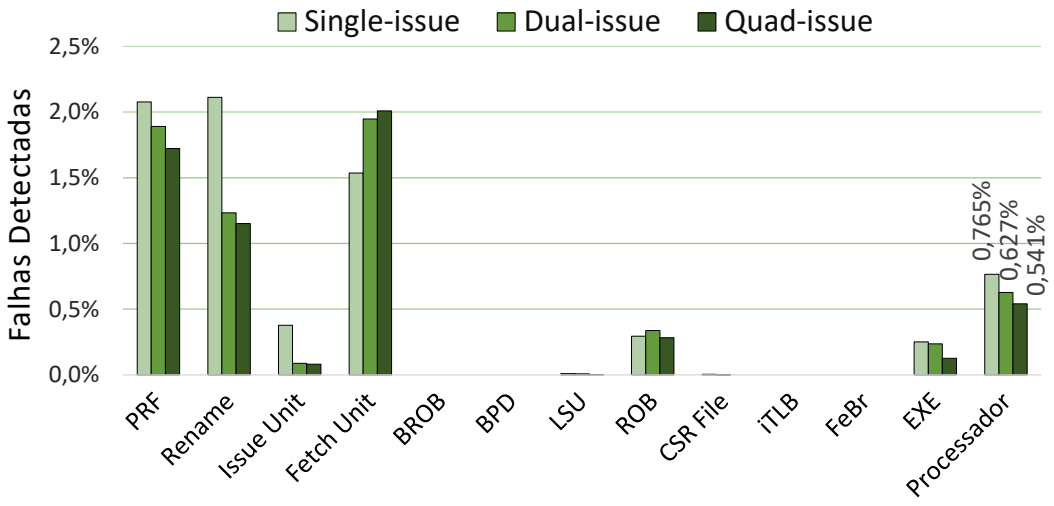
Fonte: Elaborada pelo próprio autor.

Apesar da técnica VARM_S_BRA utilizar grande número de registradores, a vulnerabilidade do banco de registradores e da unidade de renomeação não é tão grande (como ocorre com a técnica anterior), pois esta técnica é capaz de detectar mais falhas nestas estruturas. A Figura 4.37 mostra o percentual de falhas detectadas, pela técnica VARM_S_BRA, em cada estrutura dos três processadores.

Uma observação sobre a LSU, que merece destaque, é que as falhas injetadas nas instruções que estão na fila de instruções do tipo *store*, prontas para serem concluídas, não podem ser detectadas pelas técnicas porque os valores nos registradores já passaram por todo o processamento e logo serão armazenados na memória.

Comparando com a técnica anterior (VAR_S_BRA), a técnica VARM_S_BRA detecta menos falhas na LSU e, isto se reflete em mais erros na nesta estrutura. Isto ocorre porque a técnica VAR_S_BRA tem maior quantidade de instruções do tipo *load*. Em vista disso, a probabilidade de falhas injetadas nesta estrutura atingirem instruções *load*

Figura 4.37: Falhas detectadas nas estruturas dos processadores utilizando a técnica VARM_S_BRA



Fonte: Elaborada pelo próprio autor.

é maior do que com a técnica atual. Sendo assim, a técnica VAR_S_BRA pode detectar mais falhas na LSU, e isto se reflete na redução da vulnerabilidade desta estrutura.

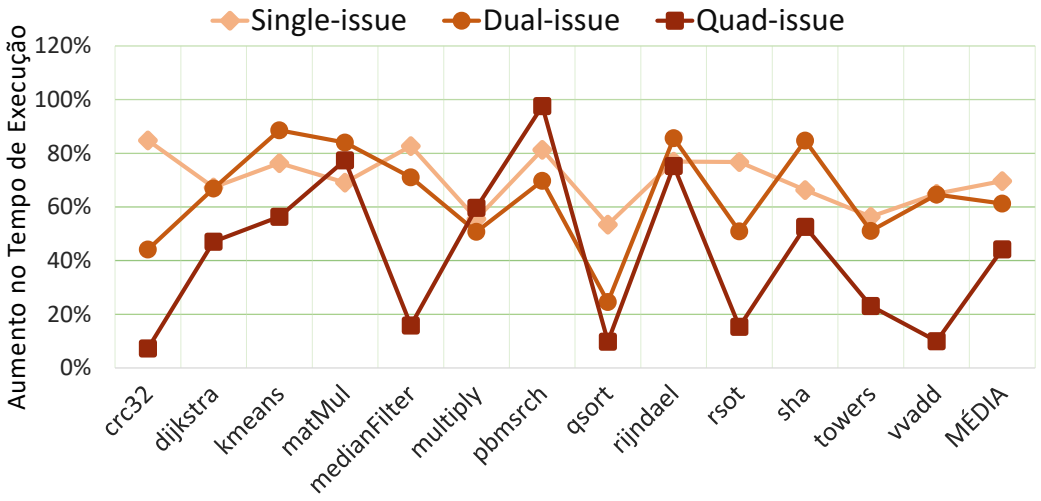
A técnica anterior também detecta mais falhas do que a técnica VARM_S_BRA na unidade que busca instruções da memória. A explicação para isto é apresentada na subseção 4.2.3.1. Apesar da técnica anterior detectar mais falhas nestas duas estruturas, a técnica VARM_S_BRA é capaz de detectar mais falhas do que a técnica VAR_S_BRA em quase todas as estruturas, inclusive nos três processadores.

A execução de instruções do tipo *load* é mais demorada do que instruções aritméticas. Portanto, o gargalo provocado pelo excesso de instruções *load* na técnica VAR_S_BRA não ocorre com a técnica VARM_S_BRA, e isto se reflete em menos impacto no tempo necessário para executar as aplicações protegidas com a técnica VARM_S_BRA, mostrado na Figura 4.38.

No *single-issue*, por exemplo, as instruções *load* inseridas pela técnica VAR_S_BRA demandam mais tempo na única unidade de execução disponível neste processador. Quando a técnica VARM_S_BRA é utilizada para proteger as aplicações, o aumento no tempo de execução do *single-issue* não é tão acentuado. Isto acontece porque esta técnica reduz o excesso de acessos à memória e, portanto, mais instruções podem ser executadas fora de ordem.

Um efeito semelhante pode ser observado no *quad-issue*. Ao reduzirmos a intensidade de uso da unidade de execução que realiza o cálculo de endereços para as instruções

Figura 4.38: Aumento do tempo para executar cada aplicação protegida com a técnica VARM_S_BRA



Fonte: Elaborada pelo próprio autor.

de acesso a memória, este tipo de instrução deixa de ser o gargalo na unidade de execução e mais instruções podem ser executadas em paralelo nas UEs, antes ociosas. Isto pode ser observado no baixo custo para executar alguns programas no *quad-issue*. Estes programas são os mesmos que levam mais tempo para executar no *quad-issue* do que no *dual-issue*, quando nenhuma técnica é utilizada. Isto indica que estes programas, antes estavam no seu limite de paralelismo, agora podem utilizar UEs ociosas para executar a cópia das instruções originais. Este comportamento só não ocorre com o algoritmo para multiplicação de matrizes porque este já alcança um limite no paralelismo das instruções, como já mencionado na subseção 4.2.1.4.

Outra aplicação que consegue minimizar o custo no tempo de execução com a técnica VARM_S_BRA é o algoritmo para ordenar um vetor *rsort*. Isto é possível porque este programa tem baixa taxa de instruções de desvio condicional e incondicional.

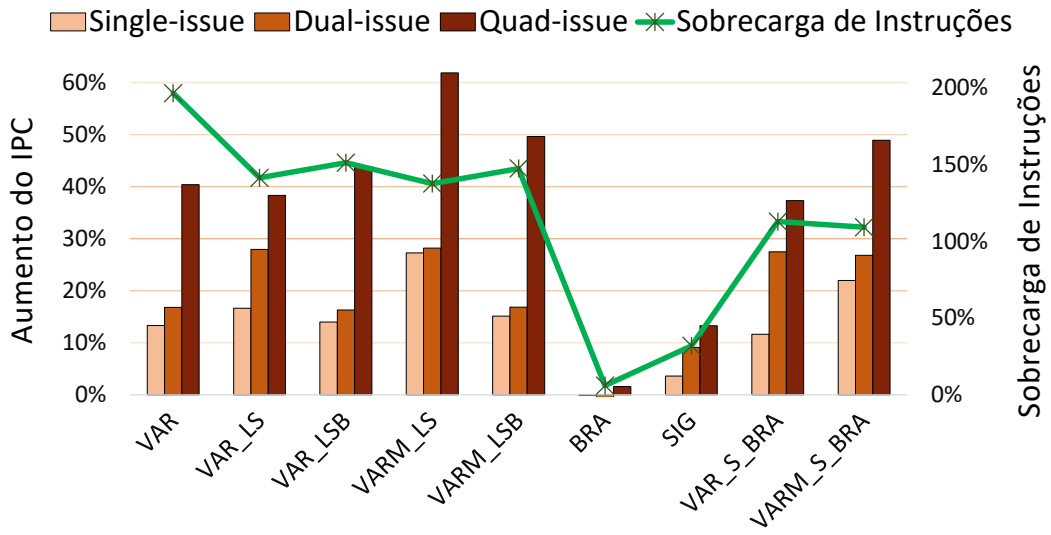
4.2.4 Resumo de todas as técnicas SIHFT avaliadas

O objetivo desta subseção é comparar, de maneira geral, a eficiência e os custos das técnicas avaliadas. Devido às dependências entre instruções, algumas aplicações não conseguem ocupar todo o hardware disponível para atingir um paralelismo no nível das instruções elevado. No processador *single-issue*, o ILP também fica comprometido devido

a gargalos causados por alguns tipos de instruções que sobrecarregam os módulos de hardware.

A Figura 4.39 está normalizada com base na execução dos programas desprotegidos. Esta figura mostra que apesar do enorme *overhead* de instruções adicionadas pelas técnicas, estas instruções são bem absorvidas por superescalares que executam instruções fora de ordem, pois as instruções extras podem ser executadas de forma independente.

Figura 4.39: Aumento do IPC e da sobrecarga de instruções em todas as técnicas



Fonte: Elaborada pelo próprio autor.

Portanto, a tolerância a falhas via redundância de instruções tende a aumentar quantidade de instruções disponíveis para execução paralela e, por isso, o IPC dos programas com as técnicas é maior do que o IPC dos mesmos programas sem técnica. Enquanto que a técnica VAR tem o *overhead* de instruções mais agressivo, a técnica VARM_LS apresenta o maior aumento de IPC em todas as três configurações do superescalar BOOM. Isto ocorre porque a técnica VAR, além de adicionar mais instruções extras, também insere grande número de instruções para comparar registradores, o que acaba limitando o ILP quando essa técnica é aplicada. Por outro lado, a técnica VARM_LS minimiza o *overhead* de instruções para comparar registradores e de instruções do tipo *load*.

É importante destacar que não é possível obter ganhos de IPC com a técnica BRA nos processadores *single-* e *dual-issue*. Isto acontece porque esta técnica não adiciona instruções redundantes que possam ser executadas em UEs ociosas. Todas as instruções inseridas por esta técnica são instruções de controle, o que sobrecarrega o preditor de desvios e limita o paralelismo.

Ao considerar o IPC, os resultados são diferentes para cada configuração do processador superescalar BOOM. O aumento do ILP pode ser melhor explorado por superescalares maiores porque estes processadores têm mais hardware ocioso durante a execução de programas desprotegidos. Por este motivo, ao executar nosso *benchamrk* protegido com qualquer das técnicas avaliadas, o aumento no IPC é mais acentuado no *quad-issue*, enquanto que no *single-issue* o aumento de IPC é mais moderado.

O aumento do IPC intensifica a ocupação de alguns módulos de hardware, isso leva a um aumento da vulnerabilidade. Além disso, o tempo de exposição também aumenta, e este impacto é mais grave no superescalar *single-issue*, devido ao fato deste ser o processador com menor capacidade de melhorar a velocidade de execução após o emprego das técnicas SIHFT.

Os gráficos na Figura 4.40 foram postos na mesma escala para facilitar a comparação entre os processadores. Esta figura mostra que a técnica VAR consegue a maior redução na taxa de SDC, Timeout, Crash e, consequentemente, fornece a melhor redução da vulnerabilidade em todos os processadores. A técnica VAR também é a mais eficiente para detectar falhas em todos os processadores. Isto ocorre devido a grande quantidade de instruções que monitoram o conteúdo de registradores, inseridas por esta técnica. Porém, este *overhead* de instruções contribui para um tempo de execução elevado.

A Figura 4.40 também demonstra que o processador *quad-issue* está menos propenso a erros do que os outros dois superescalares avaliados, para qualquer técnica avaliada.

Com exceção da técnica BRA, todas as outras técnicas SIHFT avaliadas apresentam a mesma característica: elas detectam mais falhas no processador *single-issue*. No entanto, com exceção das técnicas VAR_LSB, todas as outras técnicas conseguem a pior redução da vulnerabilidade justamente no mesmo *single-issue*. Isto ocorre porque a complexidade do superescalar *single-issue* é menor e não consegue atingir um alto grau de paralelismo no nível das instruções, facilitando a detecção de falhas. Por outro lado, as estruturas deste processador saturam mais fácil e aumentam o tempo de exposição dos flip-flops com valores úteis. Com menos hardware ocioso, este processador fica mais propenso a falhas que se manifestam como erros ao final da execução.

A Tabela 4.1 apresenta a vulnerabilidade das três versões do superescalar BOOM enquanto executa cada programa desprotegido, e a vulnerabilidade dos processadores durante a execução dos programas protegidos com cada uma das técnicas SIHFT avaliadas.

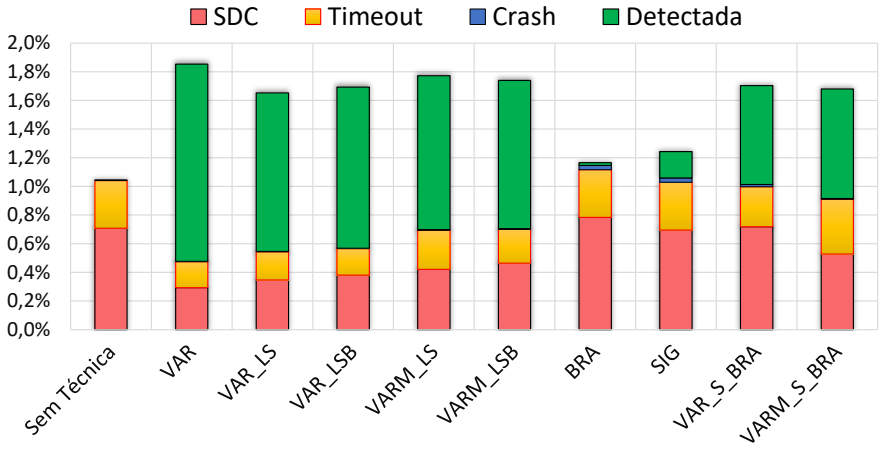
As técnicas conseguem reduzir a vulnerabilidade dos processadores em quase to-

Tabela 4.1: Vulnerabilidade dos três processadores executando cada aplicação sem técnica e com cada uma das técnicas

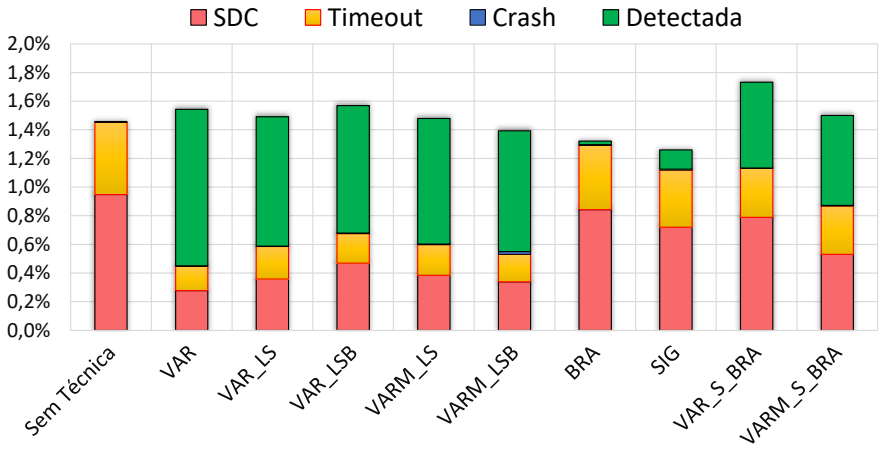
		crc32	dijkstra	kmeans	matMul	median	multiply	pbmsrch	qsort	rijndael	rsort	sha	towers	vvadd	MÉDIA
Single-issue	Sem técnica	1,09%	1,26%	0,98%	0,98%	0,97%	1,02%	1,31%	0,77%	1,06%	0,93%	1,12%	1,11%	0,98%	1,05%
	VAR	0,69%	0,49%	0,24%	0,51%	0,66%	0,81%	0,42%	0,44%	0,26%	0,38%	0,51%	0,44%	0,37%	0,48%
	VAR_LS	0,73%	0,57%	0,39%	0,58%	0,54%	0,81%	0,48%	0,47%	0,44%	0,43%	0,61%	0,53%	0,52%	0,55%
	VAR_LSB	0,70%	0,59%	0,39%	0,52%	0,68%	0,96%	0,46%	0,49%	0,46%	0,49%	0,58%	0,54%	0,53%	0,57%
	VARM_LS	0,96%	0,83%	0,40%	0,70%	0,81%	0,89%	0,92%	0,58%	0,40%	0,63%	0,75%	0,65%	0,57%	0,70%
	VARM_LSB	0,80%	1,15%	0,41%	0,60%	0,78%	1,07%	0,42%	0,57%	0,49%	0,80%	0,77%	0,70%	0,63%	0,71%
	BRA	1,52%	1,35%	0,91%	1,14%	1,10%	1,00%	1,50%	0,89%	1,12%	0,93%	1,40%	1,10%	0,94%	1,15%
	SIG	1,34%	1,21%	0,86%	1,04%	0,86%	1,14%	0,92%	0,74%	1,11%	0,92%	1,55%	1,10%	0,97%	1,06%
	VAR_S_BRA	0,95%	1,00%	0,87%	1,04%	0,98%	1,29%	1,48%	0,77%	1,04%	0,91%	1,09%	0,85%	0,88%	1,01%
	VARM_S_BRA	1,01%	1,02%	0,66%	1,05%	0,89%	0,93%	0,97%	0,69%	0,99%	0,91%	1,04%	0,84%	0,87%	0,91%
Dual-issue	Sem técnica	2,36%	1,43%	1,15%	1,90%	0,98%	1,17%	1,56%	1,00%	1,12%	1,52%	1,96%	1,70%	1,11%	1,46%
	VAR	0,54%	0,60%	0,23%	0,55%	0,53%	0,43%	0,59%	0,42%	0,18%	0,40%	0,43%	0,66%	0,32%	0,45%
	VAR_LS	0,66%	0,61%	0,33%	0,83%	0,53%	0,58%	0,72%	0,51%	0,32%	0,48%	0,79%	0,82%	0,47%	0,59%
	VAR_LSB	0,89%	1,12%	0,27%	1,12%	0,46%	0,50%	0,65%	0,80%	0,32%	0,73%	0,80%	0,74%	0,43%	0,68%
	VARM_LS	0,65%	0,88%	0,27%	0,67%	0,56%	0,95%	0,84%	0,39%	0,30%	0,50%	0,63%	0,71%	0,47%	0,60%
	VARM_LSB	0,75%	0,73%	0,26%	0,94%	0,54%	0,53%	0,37%	0,39%	0,33%	0,73%	0,54%	0,74%	0,31%	0,55%
	BRA	1,62%	1,29%	0,96%	1,20%	1,05%	1,25%	1,32%	1,08%	1,12%	1,20%	1,85%	1,65%	1,26%	1,30%
	SIG	1,41%	1,29%	0,79%	0,68%	0,93%	0,76%	1,51%	0,86%	1,08%	1,06%	1,72%	1,50%	1,04%	1,13%
	VAR_S_BRA	1,43%	0,87%	0,89%	1,39%	0,89%	1,48%	1,29%	0,89%	1,01%	1,09%	1,46%	0,94%	1,10%	1,13%
	VARM_S_BRA	1,02%	0,95%	0,57%	0,98%	0,71%	0,92%	1,13%	0,58%	0,81%	0,87%	0,96%	1,02%	0,81%	0,87%
Quad-issue	Sem técnica	1,45%	0,96%	0,85%	1,53%	0,56%	1,01%	1,13%	0,74%	1,00%	1,19%	1,63%	1,29%	0,86%	1,09%
	VAR	0,26%	0,41%	0,17%	0,42%	0,56%	0,48%	0,57%	0,53%	0,12%	0,26%	0,38%	0,47%	0,26%	0,38%
	VAR_LS	0,87%	0,73%	0,31%	0,35%	0,48%	0,82%	0,53%	0,70%	0,28%	0,46%	0,60%	0,58%	0,48%	0,55%
	VAR_LSB	0,86%	0,44%	0,26%	0,55%	0,41%	0,90%	1,04%	1,13%	0,29%	0,61%	0,69%	0,56%	0,44%	0,63%
	VARM_LS	0,43%	0,61%	0,22%	0,53%	0,39%	0,54%	0,33%	0,36%	0,26%	0,34%	0,53%	0,55%	0,32%	0,42%
	VARM_LSB	0,35%	0,64%	0,22%	0,34%	0,41%	0,63%	0,33%	0,47%	0,23%	0,40%	0,46%	0,64%	0,29%	0,41%
	BRA	0,96%	0,91%	0,72%	0,78%	0,71%	0,75%	1,06%	0,60%	1,09%	1,15%	1,53%	1,09%	0,76%	0,93%
	SIG	0,62%	0,83%	0,69%	0,72%	0,77%	0,57%	1,18%	0,84%	1,10%	0,81%	1,54%	1,09%	0,87%	0,89%
	VAR_S_BRA	1,02%	0,79%	0,79%	0,90%	0,69%	0,79%	1,17%	0,76%	0,82%	0,86%	0,94%	0,98%	0,93%	0,88%
	VARM_S_BRA	0,75%	1,01%	0,40%	0,58%	0,61%	0,62%	0,59%	0,56%	0,70%	0,74%	0,83%	0,85%	0,77%	0,69%

Fonte: Elaborada pelo próprio autor.

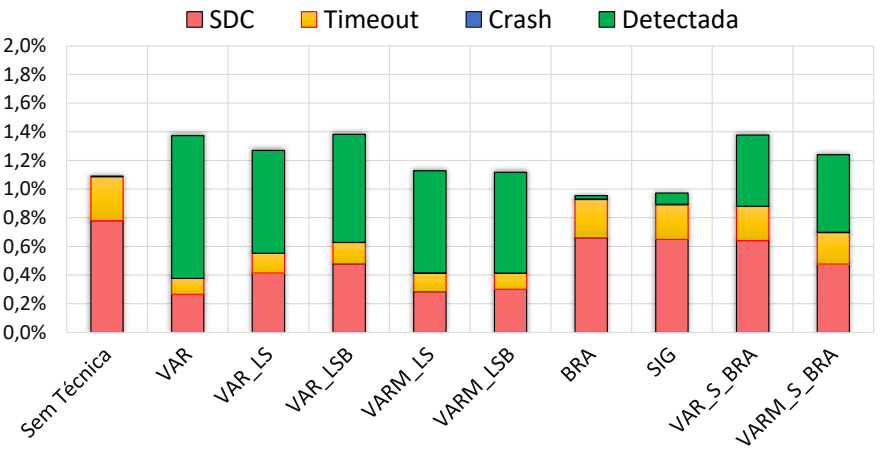
Figura 4.40: SDCs, Timeouts, Crashes e falhas Detectadas nos três superescalares sem técnica, e com cada técnica avaliada



(a) *single-issue*



(b) *dual-issue*



(c) *quad-issue*

Fonte: Elaborada pelo próprio autor.

dos os casos. Porém, é importante enfatizar que há algumas situações em que as falhas não podem ser detectadas pelas técnicas. Além dos obstáculos já citados, como a lógica de adiantamento e das falhas injetadas em instruções do tipo *store* presentes na LSU, há outros pontos fracos das técnicas discutidos a seguir.

- Quando a verificação dos registradores presentes em uma instrução é realizada antes de executar esta instrução, pode ocorrer dessa instrução demorar para ser executada, seja por estar esperando um dos registradores fontes ficar pronto, ou pelo simples fato de não ter uma UE livre para sua execução. Durante o intervalo de tempo entre a validação e o uso do registrador a instrução está vulnerável e uma falha não será detectada. Se a falha acertar o valor de um registrador usado em uma instrução de *store*, isto pode levar a um SDC.
- Devido a execução fora de ordem, os superescalares utilizam o buffer de reordenamento para concluir as instruções em ordem. Para realizar este reordenamento, as instruções que entraram primeiro no pipeline, também devem ser retiradas primeiro. Portanto, durante o tempo que uma instrução, já executada, fica na fila pronta para ser concluída, mas precisa esperar a execução das instruções anteriores, ela está vulnerável. Se a falha modifica o valor de algum dos parâmetros que indicam o status da instrução, esta falha também não pode ser detectada, levando a um *timeout*.
- As técnicas SIHFT avaliadas são projetadas para detectar falhas nos registradores e, por isso, apresentam um ótimo desempenho na detecção de falhas no banco de registradores e de falhas que ocorrem em outras estruturas situadas no início do pipeline de um superescalar. Porém, falhas injetadas nos módulos de hardware após o cálculo na unidade de execução dificilmente são detectadas. Além disso, nenhuma das técnicas avaliadas é capaz de identificar as falhas que ocorrem nas estruturas que fazem parte do preditor de desvios.

A Tabela 4.2 mostra a melhor opção de técnica para reduzir a vulnerabilidade e a melhor técnica para detectar falhas em cada estrutura de cada processador avaliado. A Tabela 4.2 mostra a melhor opção de técnica para reduzir a vulnerabilidade e a melhor técnica para detectar falhas em cada programa executando em cada processador.

As Tabelas 4.2 e 4.2 demonstram que a técnica VAR é a melhor opção na maioria das situação. No entanto, algumas das variações da técnica VAR apresentadas neste trabalho podem ser melhores com menos custos no tempo de execução e no consumo de energia.

Tabela 4.2: Melhor técnica para reduzir vulnerabilidade e melhor técnica para detectar falhas em cada estrutura

	Reduzir vulnerabilidade			Detectar falhas		
	Single-issue	Dual-issue	Quad-issue	Single-issue	Dual-issue	Quad-issue
PRF	VAR	VAR	VAR	VAR	VAR	VAR
Rename	VAR	VAR	VAR	VAR	VAR	VAR
Issue Unit	VAR	VAR	VAR	VARM_LS	VARM_S_BRA	VAR_S_BRA
Fetch Unit	VARM_LSB	VAR	VARM_LSB	VAR_LS	VAR_S_BRA	VAR_S_BRA
BROB	VAR	VAR	VARM_LSB	VAR	VAR	VAR
BPD	VARM_S_BRA	SIG	VARM_LSB	VAR	VAR	VAR
LSU	VAR	VAR	VAR	VAR_LSB	VAR_S_BRA	VAR_LSB
ROB	VAR	VAR	VAR	VARM_LS	VAR_S_BRA	VAR_S_BRA
CSR File	VARM_S_BRA	BRA	VAR_LS	VARM_S_BRA	VARM_LSB	VAR
iTLB	VAR_LS	VAR_LS	VAR	VAR	VAR	VAR
FeBr	VAR_LS	VAR_LS	VARM_S_BRA	VAR	VAR	VAR
EXE	VAR	VAR_LSB	VAR	VARM_LSB	VAR_LS	VAR
Processador	VAR	VAR	VAR	VAR	VAR	VAR

Fonte: Elaborada pelo próprio autor.

4.2.5 Proteção Seletiva de Registradores

A intensão desta análise é avaliar as possibilidades de minimizar a penalidade no tempo necessário para executar as aplicações protegidas com as técnicas SIHFT, mas tentando manter as taxas de cobertura de falhas. Portanto, nós protegemos apenas um registrador, ou um conjunto de registradores arquiteturais (isto é, somente os registradores selecionados são duplicados e monitorados durante a execução).

Alguns programas podem utilizar quase todos os registradores arquiteturais. Quando isto acontece, não sobra uma quantidade suficiente de registradores livres para serem utilizados como cópia dos registradores originais. Nesse caso, cabe ao projetista uma decisão de quais registradores devem ser protegidos. Devido a maior redução da vulnerabilidade obtida com a estratégia utilizada pela técnica VAR, esta técnica foi escolhida para a análise de redundância parcial da aplicação. Esta análise foi realizada com oito aplicações do nosso *benchmark*.

Cada registrador utilizado por um programa é protegido individualmente pelo menos uma vez. Após uma análise prévia com injeção de falhas nestes programas, constatamos que os registradores ponteiro de pilha (stack pointer - sp), endereço de retorno (return address - ra) e o registrador utilizado em chamadas de funções (saved temporary - s0) estão mais propensos a erros. Portanto, em relação aos registradores selecionados para formar conjuntos de registradores a serem protegidos, em todos os programas há conjuntos que combinam a proteção destes registradores, pois estes registradores são

Tabela 4.3: Melhor técnica para reduzir vulnerabilidade e melhor técnica para detectar falhas em cada programa

	Reduzir vulnerabilidade			Detectar falhas		
	Single-issue	Dual-issue	Quad-issue	Single-issue	Dual-issue	Quad-issue
crc32	VAR	VAR	VAR	VAR	VAR	VAR
dijkstra	VAR	VAR	VAR	VAR	VAR_LS	VAR
kmeans	VAR	VAR	VAR	VAR	VAR	VAR
matMul	VAR	VAR	VARM_LSB	VAR	VAR	VAR
medianFilter	VAR_LS	VAR_LSB	VARM_LS	VAR	VAR	VAR
multiply	VAR	VAR	VAR	VAR	VAR	VAR
pbmsrch	VAR	VARM_LSB	VARM_LSB	VAR	VAR_LS	VAR
qsort	VAR	VARM_LSB	VARM_LS	VAR	VAR	VAR
rijndael	VAR	VAR	VAR	VAR	VAR	VAR
rsot	VAR	VAR	VAR	VAR	VAR	VAR
sha	VAR	VAR	VAR	VARM_LS	VAR_LSB	VAR_LSB
towers	VAR	VAR	VAR	VAR	VAR	VAR
vvadd	VAR	VARM_LSB	VAR	VAR	VAR	VAR

Fonte: Elaborada pelo próprio autor.

fundamentais para garantir a execução dos programas e estão mais propensos a erros.

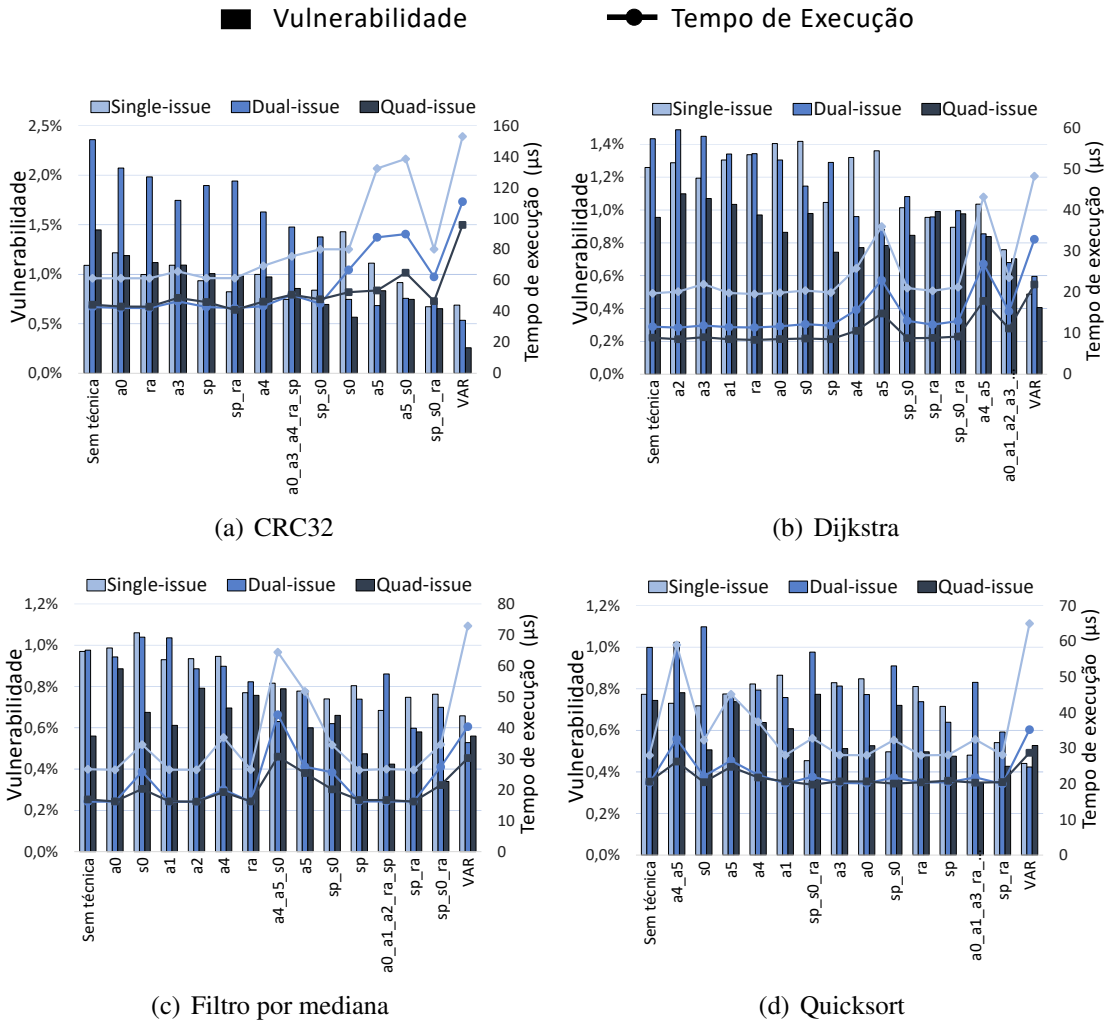
Para selecionar os outros conjuntos de registradores a serem duplicados, duas estratégias foram utilizadas. A primeira estratégia cria outros dois grupos de registradores a serem protegidos. Um grupo que combina a proteção de dois ou três registradores que são usados mais frequentemente durante a execução da aplicação, e o outro grupo combina o restante dos registradores que menos aparecem durante a execução da mesma aplicação. A Figura 4.41 mostra a vulnerabilidade e o tempo necessário para executar as quatro aplicações protegidas de acordo com a primeira estratégia.

A segunda estratégia forma conjuntos de registradores a serem protegidos de maneira diferente da primeira. Neste caso, cada vez que um novo conjunto é criado, incrementa-se um registrador a ser protegido. A vulnerabilidade e o tempo de execução das outras quatro aplicações protegidas utilizando a segunda estratégia são apresentadas na Figura 4.42.

Em todos os gráficos exibidos nas Figuras 4.41 e 4.42, os registradores individuais e os conjuntos de registradores protegidos estão ordenados, da esquerda para a direita, da maior para a menor média de vulnerabilidade dos três processadores. Nestas figuras, as barras representam as vulnerabilidades de cada processador, e as linhas simbolizam o tempo de execução.

Repare na Figura 4.41(a) que em muitos casos o CRC32 continua demorando mais tempo para executar no superescalar *quad-issue* do que no *dual-issue*, assim como ocorre quando nenhuma técnica foi utilizada. Isto se reflete diretamente na vulnerabilidade do processador *dual-issue*, pois um processador com maior porcentagem de bits ocupados

Figura 4.41: Vulnerabilidade e tempo de execução das aplicações protegidas parcialmente de acordo com a primeira estratégia



(a) CRC32

(b) Dijkstra

(c) Filtro por mediana

(d) Quicksort

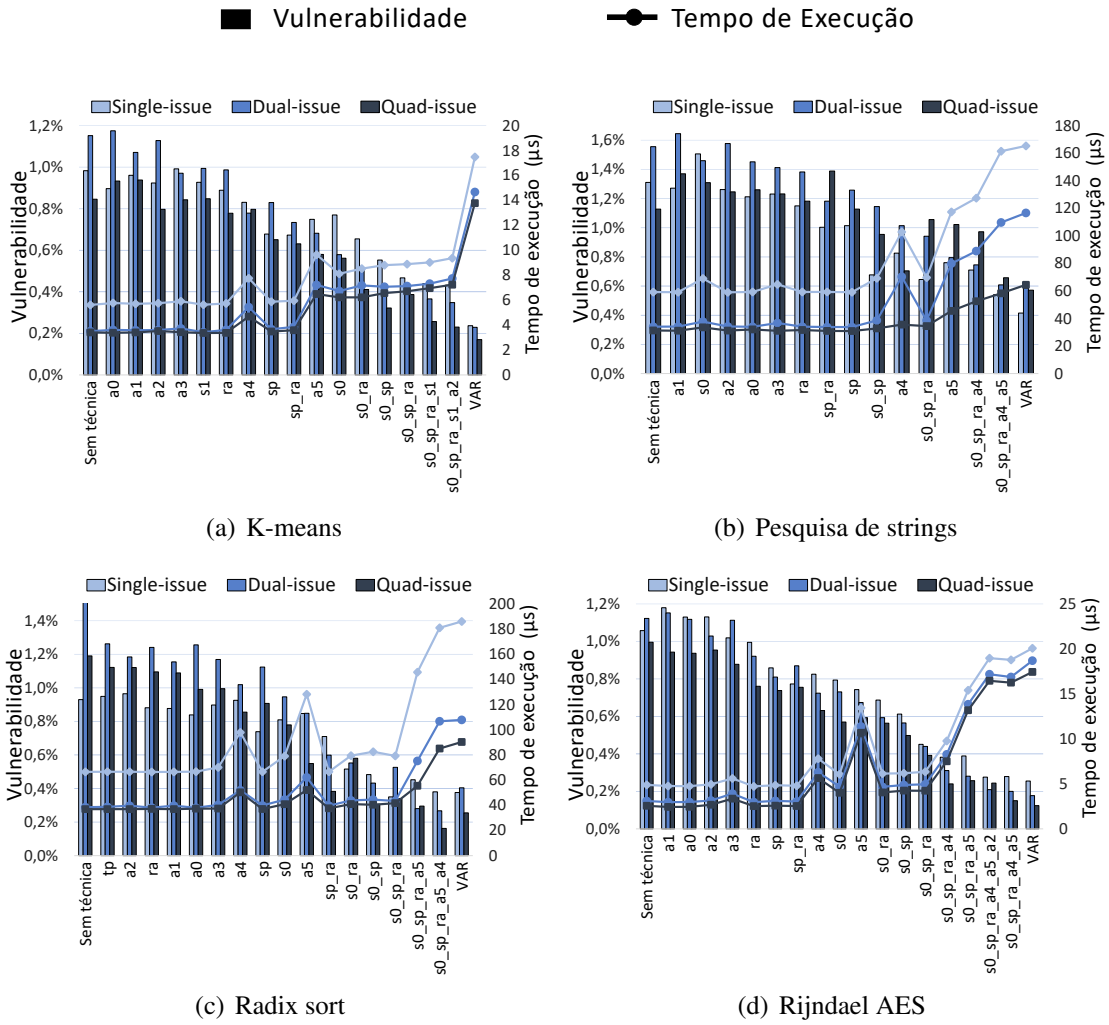
Fonte: Elaborada pelo próprio autor.

está mais propenso a erros. Também nesta aplicação, em alguns casos proteger apenas um registrador deixa a execução mais rápida do que a aplicação completamente desprotegida. A explicação para este comportamento é a precisão do preditor de desvios do BOOM, que pode variar bastante ao adicionar ou remover instruções de uma aplicação.

Como esperado, proteger os registradores mais frequentemente utilizados por um programa provoca o maior impacto no tempo de execução, e o grupo de registradores menos utilizados causa o menor *overhead* de tempo. No entanto, nem sempre o grupo de registradores mais utilizado é a melhor opção para aplicar a técnica VAR. Na Figura 4.41(b), por exemplo, é possível obter uma redução da vulnerabilidade melhor e mais barata ao proteger todos os registradores menos utilizados pelo Dijkstra, do que duplicar ou registradores que mais usados (*a4* e *a5*).

Na Figura 4.41(a), ao proteger apenas os registradores *sp*, *s0* e *ra* do CRC32, o

Figura 4.42: Vulnerabilidade e tempo de execução das aplicações protegidas parcialmente de acordo com a segunda estratégia



(a) K-means

(b) Pesquisa de strings

(c) Radix sort

(d) Rijndael AES

Fonte: Elaborada pelo próprio autor.

impacto no tempo de execução dos três processadores pode reduzir em média 10×. Este ganho de desempenho se reflete no aumento da vulnerabilidade. Porém, esta penalidade é bem baixa, a vulnerabilidade aumenta 55% no *quad-issue* e apenas 2% no *single-issue*. A mesma situação pode ser observada na Figura 4.41(b), o impacto no tempo necessário para executar o Dijkstra parcialmente protegido nos três processadores pode reduzir me média 6,4×, permitindo uma vulnerabilidade média 30% maior do que a obtida com a técnica VAR.

Na Figura 4.42(d), ao proteger apenas 5 registradores utilizados pelo Rijndael, já é possível obter uma capacidade de resiliência dos três processadores próxima ao que se consegue quando toda aplicação está protegida. A proteção destes 5 registradores também é mais eficiente e mais barata do que a duplicação de 6 registradores. Quanto à redundância de registradores individuais, a proteção do registrador *sp* ou do registrador *s0*

são as mais recomendadas, pois a proteção destes registradores aumenta pouco o tempo de execução dos programas e apresenta boa redução da vulnerabilidade. Nas Figuras 4.41(c) e 4.41(d), a duplicação apenas do registrador *sp* é que consegue os melhores níveis de resiliência.

Há também alguns cenários em que proteger apenas um registrador é melhor do que proteger um conjunto de registradores. Na Figura 4.41(d), a resiliência dos três processadores quando executa Quicksort com apenas o registrador *sp* protegido é melhor do proteger o conjunto de registradores *a4_a5*. Neste programa, a proteção do registrador *a4* e/ou do registrador *a5* não é uma boa opção porque, além de ineficientes, a duplicação destes registradores também causa um impacto acentuado no tempo de execução. Este cenário também fica evidente na vulnerabilidade dos processadores *single-* e *quad-issue* quando executam o filtro mediana mostrado na Figura 4.41(c).

Em algumas situações, é possível obter uma resiliência melhor com a proteção parcial do que duplicar todo o código. Na Figura 4.41(d), se a prioridade é reduzir a vulnerabilidade do *quad-issue* quando executa o Quicksort, o projetista pode proteger o conjunto dos seis registradores menos frequentemente utilizados por este programa: além dessa redundância parcial executar 41% mais rápido, a vulnerabilidade deste processador é 51% menor do que a obtida com a duplicação de todos os registradores. Este comportamento também pode ser observado com algoritmo de ordenação Radix sort na Figura 4.42(c), onde a redundância de um conjunto de registradores alcança níveis de resiliência melhores em todos os três superescalares avaliados.

Esta análise mostrou que, através da redundância seletiva de registradores a serem protegidos com técnicas SIHFT, o impacto tempo de execução das aplicações protegidas é minimizado, mantendo a redução de vulnerabilidade dos processadores superescalares próxima ao que se obtém com um programa completamente protegido. Através da proteção parcial, também é possível obter níveis de resiliência melhores com menos custo em alguns cenários.

Dependendo do programa ou do nível de resiliência desejado, nem todos os registradores precisam ser duplicados. Isto é importante para reduzir as penalidades no tempo de execução e na energia, e para programas que utilizam muitos registradores arquiteturais, onde não sobram registradores livres em quantidade suficiente para proteger todos os registradores. Portanto, existe um contrabalanço entre a degradação de desempenho e a capacidade de resiliência dos processadores.

4.3 Avaliação da Técnica Híbrida SW+HW

Em virtude das técnicas implementadas puramente em software não alcançarem a completa proteção dos processadores superescalares, nós procuramos uma alternativa para aumentar a capacidade de resiliência destes processadores. Para alcançar determinada redução de vulnerabilidade, além do que é possível com SIHFT, é necessário incluir mais algum mecanismo de detecção de falhas. Nós escolhemos aplicar o DMR em hardware para proteger um conjunto de módulos de hardware do superescalar.

Os custos de energia e área foram estimados levando em conta o custo do comparador na saída de cada módulo duplicado. Para esta análise consideramos que todas as falhas nas estruturas duplicadas foram detectadas. Assim como na avaliação da proteção seletiva de registradores, nesta análise a técnica VAR foi utilizada para contrabalancear os custos da proteção em hardware com os custos da proteção em software.

4.3.1 DMR com Auxílio do Problema da Mochila

Visando reduzir a vulnerabilidade das aplicações protegidas com VAR, a sugestão é aplicar um DMR sobre os módulos de hardware mais sensíveis. Devido aos fatos de que o DMR de módulos de hardware impõe um *overhead* de área, e que cada estrutura possui uma vulnerabilidade e uma área diferente das demais. Portanto, há um conjunto de estruturas mais apropriado a se duplicar para alcançar determinada redução da vulnerabilidade de um processador com o mínimo impacto de área.

Para encontrar o conjunto ideal de estruturas que devem ser protegidas para atingir a redução de vulnerabilidade requerida, foi utilizado o método do problema da mochila (KSP). Nessa estratégia, o ideal é definido como o melhor conjunto de estruturas que minimiza o *overhead* de área em decorrência do DMR. Uma vez que a heurística do KSP encontra um conjunto de estruturas que devem ser duplicadas, nenhuma outra combinação de estruturas é capaz de causar menor *overhead* de área para atingir o nível de resiliência que atenda as especificações do sistema.

Ao aplicar a técnica VAR sobre todo nosso *benchmark* de aplicações, é possível alcançar uma redução de 54% na vulnerabilidade do superescalar *single-issue*. Portanto, nenhuma estrutura precisa ser duplicada para este nível de confiabilidade. Porém, quando a redução de vulnerabilidade desejada é superior a 54%, a heurística do KSP aponta as estruturas que devem ser duplicadas para reduzir a vulnerabilidade além do limite obtido

apenas com software. A Tabela 4.4 mostra as estruturas que devem ser duplicadas para reduzir a vulnerabilidade do *single-issue* para valores acima de 54%.

Tabela 4.4: Estruturas a serem protegidas, de acordo com o KSP, para alcançar uma certa redução de vulnerabilidade no *single-issue*

	0-54%	55%	60%	65%	70%	75%	80%	85%	90%	95%	100%
PRF											×
Rename									×	×	×
Issue									×	×	×
Fetch								×	×	×	×
BROB		×				×		×	×	×	×
BPD											×
LSU			×		×	×	×	×	×	×	×
ROB							×	×	×	×	×
CSR File								×			×
iTLB											×
FeBr				×	×	×	×	×	×	×	×
EXE										×	×

Fonte: Elaborada pelo próprio autor.

Como o KSP considera a área e a vulnerabilidade de cada estrutura, a tendência é duplicar as estruturas mais sensíveis e menores primeiro. Porém, repare que o conjunto de estruturas a serem duplicadas varia bastante dependendo do nível de resiliência exigido, não ocorrendo de forma incremental. Isto é, uma vez que uma dada estrutura é duplicada para alcançar determinado nível de resiliência, esta estrutura não necessariamente será duplicada para obter resiliências maiores. Por exemplo, para uma redução de 75% na vulnerabilidade do *single-issue*, as estruturas BROB, LSU e FeBr devem ser duplicadas. Quando é necessário uma redução ainda maior, por exemplo, 80%, a duplicação do BROB é substituída pela duplicação do ROB.

A vulnerabilidade do superescalar *single-issue* sem qualquer técnica de tolerância a falhas é de 1,05%. Com a técnica VAR, há uma redução de 54% na vulnerabilidade deste processador. Para alcançar uma redução de 80% na vulnerabilidade, por exemplo, utilizar somente a técnica em software não é o suficiente. Portanto, além da técnica VAR protegendo o software, também é necessário aplicar DMR em hardware nas estruturas LSU, ROB e FeBr. Ao duplicar estes módulos de hardware a vulnerabilidade cai para 0,2%, o que representa uma redução de 80,9% na vulnerabilidade do *single-issue*.

No processador superescalar *dual-issue*, a tolerância a falhas baseada em software fornece 69% de redução da vulnerabilidade, nenhuma estrutura precisa ser duplicada para alcançar essa confiabilidade. A Tabela 4.5 mostra as estruturas que devem ser duplicadas

para alcançar confiabilidades superiores a 69% *dual-issue*.

Tabela 4.5: Estruturas a serem protegidas, de acordo com o KSP, para alcançar uma certa redução de vulnerabilidade no *dual-issue*

	0-69%	70%	75%	80%	85%	90%	95%	100%
PRF								×
Rename								×
Issue Unit								×
Fetch Unit		×					×	×
BROB		×				×	×	×
BPD								×
LSU					×	×	×	×
ROB				×		×	×	×
CSR File								×
iTLB							×	×
FeBr			×	×	×	×	×	×
EXE							×	×

Fonte: Elaborada pelo próprio autor.

Para reduzir a vulnerabilidade além da fornecida pela técnica VAR, o DMR deve ser inserido nas estruturas marcadas nesta tabela. A ordem das estruturas selecionadas é determinada pelo KSP. Por exemplo, para uma obter uma redução de vulnerabilidade de pelo menos 80%, o buffer de reordenamento e o primeiro estágio do preditor devem ser duplicados. Essa duplicação proporciona uma redução de 81,2% na vulnerabilidade do *dual-issue* com apenas 12,54% no *overhead* de área, que é o menor *overhead* possível para essa melhoria de confiabilidade.

Para o processador superescalar *quad-issue*, a técnica VAR consegue aumentar a capacidade de resiliência em 65%. A Tabela 4.6 apresenta as estruturas duplicadas para atingir reduções de vulnerabilidade superiores a 65% neste processador.

Devido a redução da vulnerabilidade das estruturas obtida com a técnicas VAR, nestas tabelas é possível notar que o primeiro estágio do preditor de desvios logo é duplicado, pois ele além de ser uma estrutura sensível a falhas, também é pequena com baixo *overhead* de hardware. Por outro lado, só é necessário proteger o banco de registradores físicos quando a meta é reduzir completamente a vulnerabilidade do processador, já que esta é uma estrutura grande e a técnica VAR consegue ser bastante eficiente.

As penalidades de área e energia para alcançar o nível de confiabilidade desejado em cada versão do superescalar estão representadas na Figura 4.43. Cada gráfico combina dois tipos de resultados, enquanto a linha contínua representa o caso em que somente o DMR em hardware é utilizado, a linha tracejada combina as técnicas VAR em software e

Tabela 4.6: Estruturas a serem protegidas, de acordo com o KSP, para alcançar uma certa redução de vulnerabilidade no *quad-issue*

	0-65%	70%	75%	80%	85%	90%	95%	100%
PRF								×
Rename							×	×
Issue Unit								×
Fetch Unit		×		×	×	×	×	×
BROB		×		×		×	×	×
BPD								×
LSU					×	×	×	×
ROB						×	×	×
CSR File							×	×
iTLB							×	×
FeBr			×	×	×	×	×	×
EXE							×	×

Fonte: Elaborada pelo próprio autor.

DMR em hardware. No caso em que as duas técnicas são aplicadas, primeiro somente a técnica VAR aplicada para alcançar a máxima redução da vulnerabilidade que esta técnica consegue. Para alcançar uma redução maior do que o software proporciona o DMR em hardware é aplicado em conjunto com a técnica VAR.

Através da Figura 4.43 fica evidente que a vantagem da tolerância a falhas baseada em hardware é o impacto reduzido na energia, e a vantagem da proteção baseada em software é que não requer custos de área.

Devido ao modo como nós modelamos o problema da mochila, tanto as curvas de energia quanto de as curvas de área iniciam com suaves custos. Quando nos aproximamos da completa redução da vulnerabilidade ambas os custos aumentam demasiadamente. Isso ocorre porque o KSP deixa o hardware mais caro pro final.

A proteção utilizando a técnica VAR impõe penalidades energéticas significativas. No entanto, este custo de energia para o *quad-issue* é menor que para os outros dois processadores. Isso ocorre porque o processador maior pode explorar o ILP com mais eficiência, o que atenua o tempo de execução imposto pela estratégia baseada no software, reduzindo assim o *overhead* de energia.

No *single-issue*, por exemplo, a técnica VAR sozinha consegue aumentar a capacidade de resiliência em 54%. Para alcançar no mínimo 70% de resiliência é necessário duplicar as estruturas LSU e o primeiro estágio do preditor de desvios. Essa duplicação proporciona uma redução de 73% na vulnerabilidade deste processador. Enquanto a duplicação dessas duas estruturas causa um *overhead* de hardware de apenas 12,8%,

se fosse utilizado somente o DMR em hardware para obter o mesmo nível de resiliência seria necessário duplicar cinco estruturas, causando um *overhead* de 49,3% na área. No entanto, ao utilizar a redundância de software em conjunto com o DMR em hardware a energia consumida é o dobro do que se gasta quando apenas o DMR é utilizado.

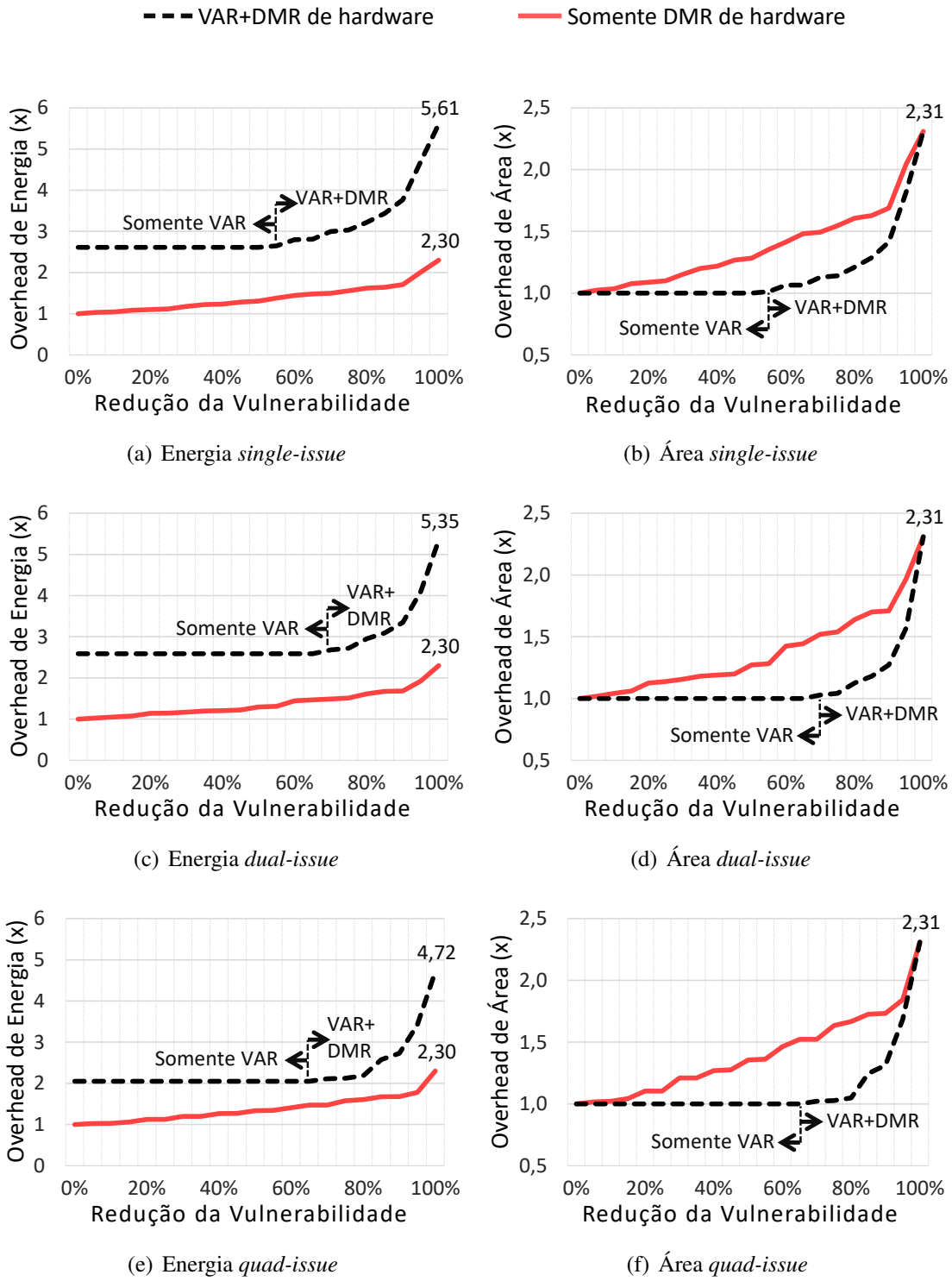
A técnica VAR é capaz de alcançar até 69,1% de redução na vulnerabilidade do *dual-issue*. Com a nossa estratégia é possível obter 90% de redução na vulnerabilidade, duplicando apenas quatro módulos de hardware deste processador. Estas estruturas correspondem apenas a 27% da área total do processador. Devido ao custo de executar um *overhead* de instruções e ainda acrescentar algum hardware, para atingir os 90% utilizando SW+HW o custo em energia aumenta 3,35×.

Por outro lado, quando somente o DMR em hardware é adotado, o mesmo nível de resiliência só é alcançado com a duplicação de 7 estruturas, que correspondem a um acréscimo de 71% de área do processador. Porém, neste caso não há impacto no desempenho, além de aumentar apenas 1,68× o consumo de energia. Portanto, embora a técnica VAR, juntamente com uma duplicação de hardware, seja a melhor escolha quando se trata de otimizar a ocupação da área, ela impõe significativos *overheads* de desempenho e energia quando comparada à duplicação de hardware.

Quando os programas estão protegidos pela técnica implementada software, nossa heurística KSP mostra que um aumento de 90% na confiabilidade processador *quad-issue* pode ser alcançada com 31,2% de *overhead* de área, e gastando 2,73× mais energia. Em outras palavras, quando a técnica VAR é utilizada, apenas 31,2% da área de silício já é responsável pela maior parte da vulnerabilidade deste processador. Quando a técnica VAR não é aplicada, a mesma meta de confiabilidade só pode ser alcançada com o 73,3% de *overhead* de área. Porém, neste caso o consumo extra é de apenas 1,68× a mais.

Esta análise apresentou uma forma mais inteligente de se obter uma confiabilidade superior ao nível que a técnica VAR proporciona. É uma forma de contrabalancear área e energia para atingir determinada redução da vulnerabilidade. Deste modo, projetistas podem selecionar a opção mais apropriada para atender seu objetivo.

Figura 4.43: Custos de energia e área em função da redução da vulnerabilidade



(a) Energia *single-issue*

(b) Área *single-issue*

(c) Energia *dual-issue*

(d) Área *dual-issue*

(e) Energia *quad-issue*

(f) Área *quad-issue*

Fonte: Elaborada pelo próprio autor.

5 CONCLUSÕES E CONSIDERAÇÕES FINAIS

Esta dissertação de mestrado apresentou uma avaliação da taxa de detecção de falhas ao utilizar técnicas de tolerância a falhas implementadas em software, e como estas técnicas afetam o tempo de execução e a vulnerabilidade dos módulos de hardware em diferentes configurações de um processador superescalares. Nove técnicas foram estudadas, sendo cinco para proteção do fluxo de dados, duas para o fluxo de controle e duas que protegem tanto os dados quanto o controle. Treze aplicações foram escolhidas e protegidas utilizando estas técnicas. Uma campanha de injeção de falhas foi efetuada com 130 milhões de falhas injetadas na descrição HDL em nível RTL dos processadores.

Após analisar a vulnerabilidade de três versões do processador superescalar BOOM através de uma extensa injeção de falhas, pudemos constatar que, embora os processadores avaliados e suas estruturas internas sejam vulneráveis, a maioria das falhas injetadas acaba sendo mascarada. Mesmo assim, técnicas de tolerância a falhas devem ser utilizadas para aumentar a capacidade de resiliência destes dispositivos.

As técnicas SIHFT avaliadas se mostraram eficientes para a redução da vulnerabilidade da unidade que busca de instruções da memória, do banco de registradores físicos, da unidade que renomeia os registradores e da unidade de despacho. Porém, o nível de vulnerabilidade foi reduzido muito pouco nas outras estruturas. Ao executar as aplicações protegidas, também há um aumento da vulnerabilidade de estruturas mais estressadas com o emprego das técnicas. Isto é causado devido ao contrabalanço entre aumentar a confiabilidade e o nível de ocupação de cada módulo de hardware do processador.

Além das penalidades causadas pela replicação de instruções, as técnicas apresentam uma proteção contra falhas incompleta. A técnica destinada a proteção dos dados mais eficiente conseguiu melhorar 69% a capacidade de resiliência do processador superescalar configurado como *dual-issue*. Porém, as técnicas para proteção do controle aumentaram a vulnerabilidade da configuração *dual-issue* devido ao aumento da sensibilidade do preditor de desvios, que é mais explorado pelas técnicas. Embora as técnicas SIHFT avaliadas consigam maior redução da vulnerabilidade no *dual-issue*, nossos resultados mostram que estas técnicas são mais apropriadas para detecção de falhas em superescalares menores devido à baixa complexidade destes processadores.

Apesar do impacto no tempo de execução causado pelas técnicas implementadas em software, a tolerância a falhas baseadas na duplicação de instruções podem se beneficiar de ILP extra. Quando as aplicações são executadas sem estas técnicas, as dependên-

cias verdadeiras entre instruções fazem com que algumas unidades de execução fiquem ociosas, esperando o resultado de outras unidades. Por outro lado, quando tais técnicas são adotadas, a maioria das instruções inseridas para proteção do código são independentes das originais, aumentando o ILP da aplicação. Desta forma, é possível utilizar as unidades funcionais que estavam ociosas com o propósito de executar operações extras para verificar a consistência dos dados ou do fluxo de controle. Na técnica de duplicação de instruções, portanto, existe um contrabalanço entre a perda de performance por executar mais instruções e um aumento do IPC da aplicação devido ao aumento de ILP.

Para reduzir os custos de energia e tempo de execução intrínsecos das técnicas SIHFT, também foi realizada uma análise da proteção parcial de oito aplicações através da redundância seletiva de registradores. Embora a proteção parcial minimize o impacto no tempo de execução, os altos níveis de resiliência ficam comprometidos. Contudo, em alguns casos, é possível reduzir a vulnerabilidade a níveis próximos aos obtidos com a duplicação de todos os registradores, com impacto no tempo de execução $10\times$ menor do que a proteção integral. Isto impõe desafios ao projetista, que deve escolher o melhor custo-benefício entre desempenho e confiabilidade.

Com o intuito de aumentar a capacidade de resiliência dos processadores superescalares, além do que as técnicas SIHFT proporcionam, foram feitas estimativas de custo para se implementar um DMR em hardware para duplicar as estruturas mais sensíveis dos processadores. Esta análise foi realizada com o emprego do DMR individualmente, e também combinada com uma das técnicas implementadas em software. Deste modo, foi possível contrabalancear os custos da técnica implementada em software com os custos da técnica implementada em hardware.

Apesar de alcançar alta taxa de detecção de falhas, o uso das técnicas de tolerância a falhas implementadas em software requer tempo extra de processamento devido à redundância de software. Isto consequentemente aumenta o consumo de energia, e o maior tempo de execução também aumenta o tempo em que a aplicação fica exposta à radiação ionizante. Por outro lado, a deficiência das técnicas desenvolvidas em hardware é o aumento da área exposta a radiação ionizante, além do alto custo de projeto para fabricar um hardware novo sempre que houver a necessidade de adaptação da técnica.

Este trabalho resultou em uma publicação na revista *Microelectronics Reliability* entitulada *Exploring the Limitations of Dataflow SIHFT Techniques in Out-of-Order Superscalar Processors*, e uma publicação no *ICECS* com o título de *Improving Software-based Techniques for Soft Error Mitigation in OoO Superscalar Processors*.

REFERÊNCIAS

ALKHALIFA, Z. et al. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Trans. on Parallel and Distributed Systems**, v. 10, n. 6, p. 627–641, June 1999. ISSN 1045-9219.

AUSTIN, T. M. Diva: a reliable substrate for deep submicron microarchitecture design. In: **MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture**. [S.l.: s.n.], 1999. p. 196–207. ISSN 1072-4451.

AZAMBUJA, J. R. **Análise de Técnicas de Tolerância a Falhas Baseadas em Software para a Proteção de Microprocessadores**. Dissertation (Master) — Instituto de Informática, UFRGS, Porto Alegre, Brasil, 2010.

AZAMBUJA, J. R. et al. Heta: Hybrid error-detection technique using assertions. **IEEE Transactions on Nuclear Science**, v. 60, n. 4, p. 2805–2812, Aug 2013. ISSN 0018-9499.

AZAMBUJA, J. R. et al. Evaluating the efficiency of data-flow software-based techniques to detect sees in microprocessors. In: **2011 12th Latin American Test Workshop (LATW)**. [S.l.: s.n.], 2011. p. 1–6. ISSN 2373-0862.

AZAMBUJA, J. R. et al. Exploring the limitations of software-based techniques in see fault coverage. **Journal of Electronic Testing**, v. 27, p. 541–550, 2011. ISSN 0923-8174.

BACHRACH, J. et al. Chisel: Constructing hardware in a scala embedded language. In: **DAC Design Automation Conference 2012**. [S.l.: s.n.], 2012. p. 1212–1221. ISSN 0738-100X.

BARNABY, H. J. Total-ionizing-dose effects in modern cmos technologies. **IEEE Transactions on Nuclear Science**, v. 53, n. 6, p. 3103–3121, Dec 2006. ISSN 0018-9499.

CELIO, C.; PATTERSON, D. A.; ASANOVIć, K. **The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor**. [S.l.], 2015. Available from Internet: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html>.

CHIELLE, E. et al. Evaluating selective redundancy in data-flow software-based techniques. **IEEE Transactions on Nuclear Science**, v. 60, n. 4, p. 2768–2775, Aug 2013. ISSN 0018-9499.

CHIELLE, E. et al. Configurable tool to protect processors against see by software-based detection techniques. In: **LATW'12**. [S.l.: s.n.], 2012. p. 1–6. ISSN 2373-0862.

CHIELLE, E. et al. Reliability on arm processors against soft errors through sihft techniques. **IEEE Transactions on Nuclear Science**, v. 63, n. 4, p. 2208–2216, Aug 2016. ISSN 0018-9499.

CONSTANTINESCU, C. Trends and challenges in vlsi circuit reliability. **IEEE Micro**, IEEE, v. 23, n. 4, p. 14–19, July 2003. ISSN 0272-1732.

DIXIT, A.; WOOD, A. The impact of new technology on soft error rates. In: **2011 International Reliability Physics Symposium**. [S.l.: s.n.], 2011. p. 5B.4.1–5B.4.7. ISSN 1938-1891.

DODD, P. E. et al. Current and future challenges in radiation effects on cmos electronics. **IEEE Transactions on Nuclear Science**, v. 57, n. 4, p. 1747–1763, Aug 2010. ISSN 0018-9499.

FERLET-CAVROIS, V.; MASSENGILL, L. W.; GOUKER, P. Single event transients in digital cmos—a review. **IEEE Transactions on Nuclear Science**, v. 60, n. 3, p. 1767–1790, June 2013. ISSN 0018-9499.

FRANKLIN, M. A study of time redundant fault tolerance techniques for superscalar processors. In: **Proceedings of International Workshop on Defect and Fault Tolerance in VLSI**. [S.l.: s.n.], 1995. p. 207–215. ISSN 1550-5774.

GOLOUBEVA, O. et al. Improved software-based processor control-flow errors detection technique. In: **Annual Reliability and Maintainability Symposium, 2005. Proceedings.** [S.l.: s.n.], 2005. p. 583–589. ISSN 0149-144X.

GOLOUBEVA, O. et al. Soft-error detection using control flow assertions. In: **Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems**. [S.l.: s.n.], 2003. p. 581–588. ISSN 1550-5774.

GUTHAUS, M. R. et al. Mibench: A free, commercially representative embedded benchmark suite. In: **Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)**. [S.l.: s.n.], 2001. p. 3–14.

ISAZA-GONZÁLEZ, J. et al. Sharc: An efficient metric for selective protection of software against soft errors. **Microelectronics Reliability**, Elsevier, p. 903–908, 2018.

LEE, I. et al. Survey of error and fault detection mechanisms. **University of Texas at Austin, Tech. Rep**, v. 11, p. 12, 2011.

LEVEUGLE, R. et al. Statistical fault injection: Quantified error and confidence. In: **2009 Design, Automation Test in Europe Conference Exhibition**. [S.l.: s.n.], 2009. p. 502–506. ISSN 1530-1591.

MAHMOOD, A.; MCCLUSKEY, E. J. Concurrent error detection using watchdog processors-a survey. **IEEE Transactions on Computers**, v. 37, n. 2, p. 160–174, Feb 1988. ISSN 0018-9340.

MANIATAKOS, M.; MICHAEL, M. K.; MAKRIS, Y. Vulnerability-based interleaving for multi-bit upset (mbu) protection in modern microprocessors. In: **2012 IEEE International Test Conference**. [S.l.: s.n.], 2012. p. 1–8. ISSN 2378-2250.

MARTINS et al. Open Cell Library in 15nm FreePDK Technology. In: **ISPD '15**. [S.l.: s.n.], 2015. p. 171–178. ISBN 9781450333993.

MCFEARIN, L. D.; NAIR, V. S. Control flow checking using assertions. 1998.

NAKKA, N.; PATTABIRAMAN, K.; IYER, R. Processor-level selective replication. In: **37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)**. [S.l.: s.n.], 2007. p. 544–553. ISSN 1530-0889.

NICOLAIDIS, M. Design for soft error mitigation. **IEEE Transactions on Device and Materials Reliability**, v. 5, n. 3, p. 405–418, Sep. 2005. ISSN 1530-4388.

NICOLESCU, B.; VELAZCO, R. Detecting soft errors by a purely software approach: method, tools and experimental results. In: **2003 Design, Automation and Test in Europe Conference and Exhibition**. [S.l.: s.n.], 2003. p. 57–62 suppl. ISSN 1530-1591.

OH, N.; MITRA, S.; MCCLUSKEY, J. Ed/sup 4/i: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers**, v. 51, n. 2, p. 180–199, Feb 2002. ISSN 0018-9340.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Control-flow checking by software signatures. **IEEE Trans. on Reliability**, v. 51, n. 1, p. 111–122, March 2002. ISSN 0018-9529.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Error detection by duplicated instructions in super-scalar processors. **IEEE Trans. on Reliability**, v. 51, n. 1, p. 63–75, Mar 2002. ISSN 0018-9529.

REBAUDENGO, M. et al. Soft-error detection through software fault-tolerance techniques. In: **Proceedings 1999 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (EFT'99)**. [S.l.: s.n.], 1999. p. 210–218. ISSN 1550-5774.

REIS, G. A. et al. Software-controlled fault tolerance. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 2, n. 4, p. 366–396, dec. 2005. ISSN 1544-3566. Available from Internet: <http://doi.acm.org/10.1145/1113841.1113843>.

REIS, G. A. et al. Swift: software implemented fault tolerance. In: **International Symposium on Code Generation and Optimization**. [S.l.: s.n.], 2005. p. 243–254.

SCHUSTER, S. et al. Demystifying soft-error mitigation by control-flow checking – a new perspective on its effectiveness. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 16, n. 5s, p. 180:1–180:19, sep. 2017. ISSN 1539-9087. Available from Internet: <http://doi.acm.org/10.1145/3126503>.

SHI, J. et al. On the design of ultra-high density 14nm finfet based transistor-level monolithic 3d ics. In: **2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2016. p. 449–454. ISSN 2159-3477.

SMITH, J. E.; SOHI, G. S. The microarchitecture of superscalar processors. **Proceedings of the IEEE**, v. 83, n. 12, p. 1609–1624, Dec 1995. ISSN 0018-9219.

SMOLENS, J. C. et al. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In: **37th International Symposium on Microarchitecture (MICRO-37'04)**. [S.l.: s.n.], 2004. p. 257–268. ISSN 1072-4451.

SNYDE, W. **Verilator, a free and open-source software tool which converts Verilog to a cycle-accurate behavioral model in C++ or SystemC**. [S.l.], 2005. Available from Internet: <https://www.veripool.org/wiki/verilator>.

SROUR, J. R.; MARSHALL, C. J.; MARSHALL, P. W. Review of displacement damage effects in silicon devices. **IEEE Transactions on Nuclear Science**, v. 50, n. 3, p. 653–670, June 2003. ISSN 0018-9499.

TONETTO, R. B.; NAZAR, G. L.; BECK, A. C. S. Precise evaluation of the fault sensitivity of ooo superscalar processors. In: **2018 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2018. p. 613–616. ISSN 1558-1101.

VEMU, R.; ABRAHAM, J. Ceda: Control-flow error detection using assertions. **IEEE Transactions on Computers**, v. 60, n. 9, p. 1233–1245, Sept 2011. ISSN 0018-9340.

VEMU, R.; ABRAHAM, J. A. Ceda: control-flow error detection through assertions. In: **12th IEEE International On-Line Testing Symposium (IOLTS'06)**. [S.l.: s.n.], 2006. p. 6 pp.–. ISSN 1942-9398.

VEMU, R.; GURUMURTHY, S.; ABRAHAM, J. A. Acce: Automatic correction of control-flow errors. In: **2007 IEEE International Test Conference**. [S.l.: s.n.], 2007. p. 1–10. ISSN 1089-3539.

VENKATASUBRAMANIAN, R.; HAYES, J. P.; MURRAY, B. T. Low-cost on-line fault detection using control flow assertions. In: **9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.** [S.l.: s.n.], 2003. p. 137–143.

WALCOTT, K. R.; HUMPHREYS, G.; GURUMURTHI, S. Dynamic prediction of architectural vulnerability from microarchitectural state. In: **Proceedings of the 34th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 2007. (ISCA '07), p. 516–527. ISBN 978-1-59593-706-3. Available from Internet: <http://doi.acm.org/10.1145/1250662.1250726>.

WANG, N. J. et al. Characterizing the effects of transient faults on a high-performance processor pipeline. In: **Proceedings of the 2004 International Conference on Dependable Systems and Networks**. Washington, DC, USA: IEEE Computer Society, 2004. (DSN '04), p. 61–. ISBN 0-7695-2052-9. Available from Internet: <http://dl.acm.org/citation.cfm?id=1009382.1009722>.