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

GUILHERME ESPINDOLA MEDEIROS

Avaliação do Impacto de *Flags* de Compilação na Ocorrência e Propagação de *Soft Errors* em Sistemas Multiprocessados Baseados em NoC

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

Orientador: Prof. Dr. Ricardo Reis
Co-orientador: Prof. Dr. Luciano Ost

Porto Alegre
2020

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Espindola Medeiros, Guilherme

Avaliação do Impacto de *Flags* de Compilação na Ocorrência e Propagação de *Soft Errors* em Sistemas Multiprocessados Baseados em NoC / Guilherme Espindola Medeiros. – Porto Alegre: PGMICRO da UFRGS, 2020.

65 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2020. Orientador: Ricardo Reis; Co-orientador: Luciano Ost.

1. Soft errors. 2. Sistemas embarcados. 3. Redes intra-chip. 4. Confiabilidade de sistemas. 5. Compiladores. 6. Multiprocessadores. I. Reis, Ricardo. II. Ost, Luciano. III. Título.

AGRADECIMENTOS

.

RESUMO

A confiabilidade de sistemas é uma métrica essencial para os projetos de sistemas embarcados multiprocessados em larga escala. Os projetistas devem identificar a suscetibilidade a *soft error* de várias aplicações no início do projeto para garantir um sistema com maior confiabilidade. Existem diversos fatores que afetam a qualidade dos projetos, como a arquitetura e a compilação das aplicações. Deste modo, a tecnologia dos compiladores desempenha um papel importante em aplicações embarcadas: desempenho e eficiência de energia. Os compiladores fornecem aos engenheiros de *software* uma ampla variedade de configurações de otimização (ou *flags*), que podem ser usadas para configurar mensagens de depuração e aviso, ou para obter otimização de código. Embora o uso de *flags* de otimização possa melhorar substancialmente o desempenho da aplicação embarcada, seu impacto na resiliência a erros de software ainda não está claro. Esta dissertação, estendeu o suporte do injetor de falhas desenvolvido em cima de uma plataforma multiprocessada chamada HeMPS para realizar injeções nos modelos em SystemC, trazendo as vantagens de aumento de desempenho de simulação com uma pequena desvantagem de precisão. Outro tema pesquisado consiste na investigação do impacto das *flags* de otimização do compilador (-O1, -O2, -O3 e -Os) na confiabilidade de *soft error* de um processador MIPS executando 24 *benchmarks* com aplicações com até $2,2$ milhões de instruções e em sistemas multiprocessados. Os resultados obtidos mostram que o nível -Os aumentou a resiliência a *soft error* para 75% do conjunto de aplicações quando comparado ao nível -O0. Além disso, o nível -Os forneceu melhorias de até $3,1x$ na resiliência das aplicações. Por fim, é apresentado uma avaliação da propagação de falhas em sistemas multiprocessados.

Palavras-chave: Soft errors. sistemas embarcados. redes intra-chip. confiabilidade de sistemas. compiladores. multiprocessadores.

ABSTRACT

Software reliability is an essential design metric in emerging large-scale multiprocessor embedded systems. Designers must identify the susceptibility to soft error of multiple applications in the early stages of the project to ensure a more reliable system. There are several factors that affect a processor design, such as architecture and application compilation. In this way, compiler technology plays an important role in embedded applications, performance and power efficiency. Compilers provide software engineers with a wide variety of optimization settings (i.e. flags), which can be used to either configure debugging and warning messages or to achieve code optimization. While the use of optimization flags can substantially improve the performance of embedded application, their impact on soft error resiliency remains unclear. This dissertation extended the support of the fault injector developed on the HeMPS plataform to perform injections on models described in SystemC. Bringing the advantages of increased simulation performance with a small precision drawbrack. Another researched topic is the evaluation of the impact of compiler optimization flags (i.e. -O1, -O2, -O3, and -Os) on soft error reliability of a MIPS processor running 24 benchmarks with up to 2.2 million instructions. The results show that the -Os level increased the soft error resilience to 75% of the application set when compared to the -O0 level. Moreover, -Os level provided enhancements up to $3.1x$. Finally, an evaluation of the propagation of failures in multiprocessed systems is presented.

Keywords: soft errors, embbeded systems, network on chip, systems reliability, compilers, multiprocessors.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADPCM	*Adaptive Differential Pulse-Code Modulation*
BTI	*Bias Temperature Instability*
CAD	*Computer-Aided Design*
DMA	*Direct Memory Access*
DMNI	*Direct Memory Network Interface*
DTW	*Dynamic Time Warping*
EP	Elemento de Processamento
FPGA	*Field Programmable Gate Array*
GCC	*GNU Compiler Collection*
HCI	*Hot Carrier Injection*
HDL	*Hardware Description Language*
HeMPS	*Hermes Multiprocessor System-on-Chip*
ICC	Intel C++ *Compiler*
LAFN	Laboratório Aberto de Física Nuclear
MIF	Módulo de Injeção de Falhas
MIPS	*Microprocessor without Interlocked Pipeline Stages*
MNT	Modelagem à Nível de Transação
MOS	Metal Óxido Semicondutor
MPEG	*Moving Picture Experts Group*
MPSoC	*Multiprocessor system-on-chip*
NoC	*Network on Chip*
OMM	*Output Mismatch*
ONA	*Output not Affected*
RTL	*Register Transfer Level*

SBU	*single-bit upset*
SDC	*Silent Data Corruption*
SEE	*Single Event Efects*
SEFI	*Single Event Functional Interruption*
TBBD	*Time Dependent Dielectric Breakdown*
TLM	*Transaction level modeling*
ULA	Unidade Lógica Aritmética
UT	*Unexpected Termination*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very-High-Speed Integrated Circuit*

LISTA DE SÍMBOLOS

μ	Média Aritmética

SUMÁRIO

1 INTRODUÇÃO

Os sistemas multiprocessadores são utilizados em diversos segmentos da indústria, incluindo os setores embarcados, médico e automotivo, devido à sua eficiência e desempenho energéticos. Esses sistemas aumentam o desempenho empregando vários processadores, que variam em relação à estrutura e à eficiência energética (BENINI; DE MICHELI, 2002). Enquanto vários processadores aprimoram o desempenho do sistema, o número crescente de núcleos de processamento e de memória interna, juntamente com as crescentes densidades de energia do *chip* e a contínua redução das dimensões da tecnologia, tornam os sistemas subjacentes mais suscetíveis à ocorrência de *soft errors* causadas por partículas, tais como, alpha, prótons ou nêutrons (BAUMANN, 2005) (SNIR et al., 2014).

O encolhimento dos transistor, tornaram os dispositivos eletrônicos mais suscetíveis a estas partículas carregadas e, portanto, mesmo circuitos no nível do solo também podem ser afetados (GRANLUND; GRANBOM; OLSSON, 2003). *Soft errors* também conhecidos como *Single-Event Effects*(SEEs) são *glitches* nos dispositivos semicondutores causados por partículas com alta carga ao atingir dispositivos. Essas partículas carregadas podem gerar a efeitos danosos nos componentes de um sistema eletrônico (e.g., gpu (GONÇALVES de OLIVEIRA et al., 2016)). Por exemplo, ao atingir um processador, uma partícula carregada pode alterar os valores armazenados em seus elementos de memória (e.g., registradores e memória SRAM). Como exemplo de possíveis comportamentos incorretos citam-se o *loop* infinito, o qual pode computar incorretamente exceções de dados e *hardware*.

Com o objetivo de avaliar a vulnerabilidade à *soft errors* de processador único (MANSOUR; VELAZCO, 2013) e multiprocessado (ROSA et al., 2017), abordagens baseadas em simulações tem sido utilizadas, levando em consideração diferentes níveis de abstração e seus *trade-offs*. Enquanto abordagens de alto nível fornecem melhores recursos de modelagem e desempenho de simulação (ROSA et al., 2017), (KALIORAKIS et al., 2015), tais abordagens apresentam limitações como menor precisão (FLENKER et al., 2017) quando comparados à abordagens nos níveis mais baixos como emulação em *hardware* e simulações de portas lógicas (BARAZA et al., 2000), (MUKHERJEE et al., 2003) e (SOLINAS et al., 2017). Apesar da penalidade em desempenho, as abordagens no nível de portas lógicas ou RTL possuem uma fidelidade de modelagem e precisão de resultados mais altas do que a primeira.

Em geral, os pesquisadores se concentraram em avaliar a vulnerabilidade de *soft errors* dos sistemas de processador único devido a vários motivos, incluindo o alto custo de simulação e a falta de plataformas de multiprocessador livres e estáveis. Para permitir uma analise mais rápida e eficaz da ocorrência de *soft errors* em sistemas multiprocessados, ferramentas e técnicas emergentes devem fornecer meios eficazes aos engenheiros para identificar as fontes mais comuns de erros e seu impacto no comportamento de tais sistemas, os quais apresentam um considerável aumento de eventos simultâneos e condições de corrida quando comparados a sistemas com um único processador.

O desenvolvimento de *software* de sistemas *single* e multiprocessados desempenha um papel importante não apenas no desempenho e na eficiência de energia do sistema, mas também em sua confiabilidade (PO-KUAN HUANG; GHIASI, 2006) (PANDA; DUTT; NICOLAU, 1997). Nesse contexto, os líderes industriais estão explorando os recursos de otimização do compilador para aprimorar a programação e aproveitar o desempenho dos processadores embarcados disponíveis, enquanto atendem às restrições de energia (EICHENBERGER et al., 2006). Os conjuntos de otimização dos compiladores, *flags*, afetam significativamente a estrutura do código e o fluxo de dados, ou seja, o uso de memória e banco de registradores. Portanto, é imperativo considerar o impacto das configurações de otimização do compilador na vulnerabilidade de erro programável (LINS et al., 2017).

Nesse aspecto, a maioria dos trabalhos disponíveis sobre *flags* de compilação se concentra na otimização do desempenho, tamanho do código e detecção de corrida de dados (JIA; CHAN, 2013; SONG et al., 2014; MACHADO et al., 2017). Enquanto, os trabalhos de (DEMERTZI et al., 2011), (SANGCHOOLIE et al., 2014), (LINS et al., 2017), (SERRANO-CASES et al., 2019) e (GAVA et al., 2019), realizaram avaliação no impacto de *flags* e compiladores na resiliência a *soft error*. Este trabalho também avalia o impacto que *flags* de compilação tem na ocorrência e propagação de *soft error* em sistemas multiprocessados baseados em redes intra-chip (do inglês, NoC). Para isso, adotou-se a plataforma HeMPS (CARARA et al., 2009) que provê a capacidade de gerar sistemas multiprocessados descritos em distintos níveis de abstração - RTL e TLM.

1.1 Objetivos

Os objetivos deste trabalho compreendem grupos distintos: estratégicos e específicos, e são definidos a seguir.

Objetivos estratégicos:

- Domínio da tecnologia de projetos de sistemas multiprocessados em chip (MPSoC);
- Investigar os efeitos de diferentes falhas em sistemas multiprocessados que adotam NoC como meio de interconexão;
- Compreensão das diferentes técnicas para injetar falhar em sistemas multiprocessados;
- Entendimento dos diferentes conjuntos de otimização do compilador GCC;

Objetivos específicos:

- Comparação da precisão entre distintos modelos de abstração RTL e TLM;
- Definir qual a *flag* que possuí maior resiliência a falhas no conjunto de aplicação;
- Suporte ao injetor de falhas proposto por (BORTOLON et al., 2018) para a linguagem SystemC;
- Criação de uma métrica para classificar a propagação de falhas em cada Elemento de Processamento (PE);

1.2 Contribuições

O presente trabalho tem como contribuição principal (i) a avaliação do impacto de *flags* de compilação na ocorrência e propagação de *soft errors* em sistemas multiprocessados, (ii) avaliação de propagação de falhas em sistemas multiprocessados dentro da plataforma HeMPS.

Outra importante contribuição é (iii) a comparação de desempenho e precisão dos distintos níveis de abstração. Esta avaliação foi realizada para sistemas *baremetal* e multiprocessados com sistema operacional.

1.3 Estrutura do Documento

Além da Introdução esta Dissertação inclui cinco capítulos.

Capítulo 2 - Conceitos Básicos na análise de *Soft Error* e nas Opções de Otimização do Compilador GCC: A primeira seção deste capítulo (Seção 2.1) introduz um breve histórico sobre os distintos tipos de falhas, a Seção subsequente apresenta as distintas abordagens para avaliar e classificar *soft error*. Por fim a Seção 2.3 relata os distintos conjuntos de otimização presente no compilador GCC.

Capítulo 3 - Estado da Arte: a Seção 3.1 apresenta o estado da arte e posiciona o presente trabalho relacionado ao tema da avaliação ou proposta de técnicas em compiladores com a confiabilidade de sistemas. Seção 3.2 posiciona o trabalho proposto ao distintos injetores de falhas, injetores que modelam falhas em distintos níveis de abstração RTL, TLM e Plataforma Virtuais, relacionando os principais pontos de cada abordagem.

Capítulo 4 - *Framework*: A primeira Seção, descreve o injetor que foi utilizado como referência ao trabalho. A Seção 4.2 detalha a implementação realizada do módulo de injeção de falhas para suportar o nível de abstração TLM com modelos descritos em SystemC.

Capítulo 5 - Análise de falhas em alto nível: Este capítulo apresenta uma análise dos experimentos realizados. A Seção 5.1 apresenta a avaliação realizada referente a precisão dos distintos modelos de abstração, RTL e TLM. A primeira avaliação realizada comparou a precisão com um conjunto de aplicação comparado em ambientes com apenas um processador, esta avaliação é estendida para cenários multiprocessados. Por fim, após ser identificada qual a *flag* que tem um maior aumento na confiabilidade do sistema, foram realizados experimentos para analisar o impacto da propagação de falhas e identificar os elementos de processamento que tiveram um maior número de falhas.

Capítulo 6 - Conclusão e Trabalhos Futuros: Este capítulo sumariza as contribuições deste trabalho. Também, descreve trabalhos futuros relacionados a esta Dissertação.

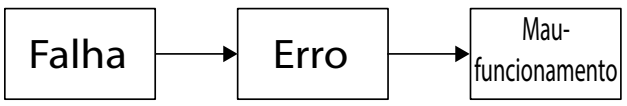
2 CONCEITOS BÁSICOS SOBRE AVALIAÇÃO DE *SOFT ERROR* E OPÇÕES DE OTIMIZAÇÃO DO COMPILADOR GCC

Este capítulo detalha alguns conceitos utilizados nesta dissertação. Primeiro, a Seção 2.1 apresenta conceitos relacionados à falhas ocorrentes em circuitos, suas diferenças e como são modeladas. Em seguida, a Seção 2.2 apresenta algumas das técnicas utilizadas para avaliar *soft error* em sistemas embarcados, além das distintas classificações nos diferentes níveis em abstrações. Por fim, a Seção 2.3 descreve as *flags* do compilador GCC utilizadas neste trabalho.

2.1 Falha, Erro e Mau-Funcionamento

Um sistema é um conjunto de elementos composto por um ou mais subsistemas, por exemplo, *hardware* e *software*. Todos os subsistemas interagem entre si, enquanto o sistema interage com outros sistemas em seu ambiente (AVIZIENIS et al., 2004). Diante disso, a indústria de semicondutores vem enfrentando desafios para garantir a confiabilidade e o correto funcionamento dos mesmos (KARNIK; HAZUCHA, 2004). De acordo com (MUSHTAQ; AL-ARS; BERTELS, 2011), uma falha ativa afeta o estado total de uma ou mais entidades do sistema. A divergência do estado correto de um elemento é conhecido como *erro*. Quando um erro propaga e afeta o estado externo, por exemplo, o sistema para de funcionar, neste caso, é classificado como um *mau-funcionamento*. O fluxo é ilustrado na Figura 2.1 (IBE, 2015). Dentro da categoria de *falhas*, três grupos abrangem os principais desafios para a indústria:

Figura 2.1: Fluxo de Falha, Erro e Mau-Funcionamento.



Falhas transientes ou *soft errors*: abrangem qualquer *mau-funcionamento*, sem dano permanente ao circuito. *Single-Event Effects* (SEEs) é a forma genérica para descrever o grupo destes eventos, como por exemplo, partículas energizadas (nêutrons, prótons, íons, partículas alpha) que interagem com dispositivos semicondutores, gerando carga elétrica na forma de *glitch* (pulso elétrico de curta duração). Este pulso em um processador

pode ser propagado para elementos de memória (por exemplo, registradores, *latches*), alterando o valor de um bit em um componente de memória dentro do sistema (VELAZCO; MCMORROW; ESTELA, 2019). Devido ao avanço tecnológico, alguns parâmetros são alterados visando o aumento de desempenho ou afim de garantir um baixo consumo energético. Entre estes, destacam-se o aumento na frequência de operação, circuitos com baixa tensão de alimentação e o encolhimento do tamanho dos transistores. Tais alterações contribuem para o aumento da suscetibilidade dos sistemas eletrônicos a ocorrência de *soft error* (SEIFERT; OUTROS, 2010; SLAYMAN, 2010).

Falhas permanentes são imperfeições físicas ocorridas no circuito, como violações de tempo, chamados de *stuck-at* zero, que podem ocorrer através do processo de envelhecimento do circuito. Desta forma, diminuir o tamanho do transistor pode acelerar o envelhecimento e, eventualmente, o desgaste devido a fenômenos distintos, como a aceleração de portadores da fonte para o dreno pela tensão aplicada, fazendo com que alguns desses portadores ganhem energia para romper a barreira potencial de interface do dispositivo. A presença desses portadores pode alterar drasticamente as características do dispositivos. Este evento é chamado de *Hot Injection Carrier* (HCI) (KUEING-LONG CHEN et al., 1985; ALAM et al., 2007). Outros exemplos desta categoria que podem ocorrer são Instabilidade de Temperatura de Viés (do inglês, BTI), Eletromigração e *Time Dependent Dielectric Breakdown* (TDDB).

A última categoria de falhas consiste na variabilidade do processo de fabricação. Esta variabilidade está relacionada com o tamanho do transistor, conforme ocorre uma diminuição, a variabilidade no processo de fabricação tende a aumentar (PANDINI, 2009), afetando assim múltiplos fatores, dentre eles, pode-se citar, a espessura do óxido, a largura do canal, concentração de dopagem, quebra de óxido, transistores parasitas, entre outros, criando dispositivos que possuem o mesmo design lógico, porém com características físicas distintas. Para mitigar essas alterações e garantir um maior valor de aproveitamento de chips em um *wafer* de silício, diversos parâmetros do projeto são reduzidos, como, por exemplo, a potência e o desempenho (BORKAR et al., 2003).

Se por um lado, falhas permanentes criam um desgaste no sistema e consequentemente reduz sua vida útil, a algumas técnicas de projeto podem mitigar parcialmente a ocorrência desta categoria, enquanto a variabilidade no processo aumenta o tempo de desenvolvimento e o custo de produção para mitigar sua ocorrência. Por outro lado, falhas transientes introduzem um comportamento inesperado no sistema. *Soft errors* ocorrem em tempos aleatórios, seja quando o sistema está operando com uma alta taxa de carga ou

está em momento ocioso. Dentre as categorias de falhas citadas anteriormente, garantir uma maior confiabilidade de sistemas à *soft error* é um dos mais proeminentes tópicos de pesquisa da indústria de semicondutores (GRANLUND; GRANBOM; OLSSON, 2003).

2.2 Avaliação de *Soft Errors*

Uma maneira de avaliar a ocorrência e impacto de *soft error* em sistemas computacionais eletrônicos é através da injeção de falhas. Nesta abordagem, é comparado o comportamento da aplicação com e sem a presença de falhas. Existem diversas abordagens para extrair o comportamento de um dado sistema sob a presença de falhas. Uma abordagem consiste no uso de dispositivos lógicos programáveis (FPGAs) ou na utilização de circuitos integrados. Em ambos os casos é necessário o desenvolvimento de técnicas para a modelagem, uma delas pode ocorrer através de interrupções realizadas no processador, onde a falha é realizada.

A segunda alternativa consiste na exposição da placa à radiação de nêutrons. Estes testes possuem um alto custo de tempo, onde cada teste pode levar dias para criar uma campanha de falha confiável, porém produzem resultados mais precisos. Por último podemos avaliar as injeções de falhas através de simulações. Esta técnica é amplamente utilizada e aceita como uma maneira eficiente de realizar a avaliação de *soft error*, permitindo a possibilidade de detectar previamente partes mais suscetível a falhas (KOOLI; DI NATALE, 2014; CHO et al., 2013). Esta dissertação adotou a técnica de avaliação de falhas através de simulações. A Seção 3.1 apresenta os principais trabalhos referente à injetores de falha.

2.2.1 Classificação de Falhas

Após a exposição de um sistema a ocorrência de falhas, seja por radiação direta ou simulação das mesmas, torna-se necessário classificar os tipos mais comuns de *soft erros* ocorridos afim de permitir uma análise mais eficiente e sólida. Existem diferentes classificações na literatura. Enquanto alguns trabalhos utilizam a classificação de falhas em duas categorias: (i) *Silent Data Corruption* (SDC), quando o resultado produzido difere do esperado e (ii) *Single Event Functional Interruption* (SEFI) quando o sistema para de

funcionar, necessitando de uma reinicialização para restaurar o estado inicial. (SANTINI et al., 2015) Esta dissertação adotou a classificação de falhas que foi proposta por (CHO et al., 2013), que consiste em dividir as falhas em cinco grupos distintos, criando uma maneira mais detalhada de avaliar e identificar qual elemento do sistema está sendo afetado. As cinco classes são:

- *Vanished*: quando não existe indicador de falha;
- ONA (*Output not Affected*): quando a aplicação termina sem a presença de erros. Contudo, a memória é afetada, ou seja, um or mais bits estão incorretos;
- OMM (*Output Mismatch*): quando o resultado da memória é modificado;
- UT (*Unexpected Termination*): quando a aplicação termina de forma anormal contendo uma indicação de erro;
- Hang: quando a aplicação não é concluída, seu término ocorre através da preempção de um *timeout*;

2.3 Opções de Otimização do Compilador GCC

O GNU Compiler Collection (GCC) (GNU, 2006) é um dos compiladores no mercado que fornece uma ampla variedade de *flags* para configurar a depuração, mensagens de aviso, otimização de código e muitas outras opções de compilação. *Flags* são um conjunto de otimizações que podem ser habilitadas durante a compilação de algum código fonte, onde cada *level* de otimização abrange diversas opções que podem ser habilitadas ou desabilitadas manualmente.

Este trabalho utiliza as opções de otimização disponíveis sem habilitar ou desabilitar alguma opção específica e, portanto, é necessário introduzir brevemente as *flags* relacionados ao GCC. Sem qualquer conjunto de otimização, o objetivo do compilador é reduzir o custo da compilação e fazer com que a depuração produza os resultados esperados, considerando o tempo de execução. Por outro lado, ativar uma *flag* de otimização faz com que o compilador tente melhorar o desempenho e/ou o tamanho do código às custas do tempo de compilação e possivelmente da capacidade de depurar o programa. O GCC fornece uma variedade de níveis gerais de otimização, indicados pelo sinalizador `-Olevel`, em que *level* é um número de 0 a 3, além de opções individuais para tipos específicos de otimização (GOUGH; STALMMAN, 2004). Além dessas *flags* numeradas, o GCC também fornece a *flag* `-Os`. Em suma os efeitos de cada flag de otimização

explorados neste trabalho são descritos abaixo:

- `-O0`: não realiza nenhuma otimização e cada comando do código-fonte é convertido diretamente na instrução correspondente no arquivo executável.
- `-O1`: o compilador visa reduzir tamanho do código e o tempo de execução. Permitindo reordenação de instruções, otimizações nos laços de repetição.
- `-O2`: aprimora ainda mais que `-O1`, incluindo otimizações para agendamento de instruções, otimização de chamadas recursivas.
- `-O3`: otimização que visa melhorar o tempo de execução da aplicação. Move condições invariantes do laço de repetição, função embutida. Código maior - Mais custoso.
- `-Os`: usa todas as otimizações de `-O2` que não aumentam o tamanho do código e seleciona otimização para reduzir o tamanho do executável gerado.

Por fim, vale ressaltar que compilar com níveis mais altos de otimização implica em maior complexidade de depuração e aumento de recursos, como memória e tempo, durante a compilação.

2.4 Resumo do Capítulo

Este capítulo apresentou uma visão geral dos tipos de falhas que podem ocorrer em sistemas. A discussão compreendeu as três categorias distintas de falhas transientes de processo de fabricação. Da mesma forma, foram apresentadas as alternativas existentes para avaliar *soft error* (falhas transientes), como realizar injeções através de interrupções no processador, expor a experimentos de íons e simulações digitais, Em seguida, foi apresentado o modelo de classificação utilizada no trabalho. Por fim, foram apresentados os distintos conjuntos de otimizações presentes no compilador GCC que foram utilizados neste trabalho.

3 ESTADO DA ARTE

Este capítulo posiciona a contribuição dessa dissertação em relação ao estado da arte em avaliação do impacto de *flags* de compilação na ocorrência e propagação de *soft errors* em sistemas, dividindo em 2 temas distintos. A Seção 3.1 apresenta o estado da arte dos trabalhos que abordam a otimização de compiladores relacionados com confiabilidade de sistemas. Por fim a Seção 3.2 apresenta uma breve descrição dos trabalhos alusivos a injetores de falha, abordando as distintas maneiras de realizar uma campanha de injeção, apontando pontos fortes e fracos de cada abordagem.

3.1 Trabalhos Relacionados ao Impacto de Compiladores na Confiabilidade de Sistemas

A maioria dos trabalhos disponíveis sobre *flags* de otimização analisam a relação entre desempenho e tamanho do código. (JIA; CHAN, 2013) avaliam o impacto dos níveis de otimização disponíveis na detecção de corrida de dados. Analisando quatro *benchmarks* diferentes, eles concluíram que as otimizações padrões (`-O1`, `-O2`, `-O3`, `-Os` e `-Ofast`) fizeram com que a detecção de corrida ocorresse mais rápido do que usando a opção de linha de base (`-O0`). Song (SONG et al., 2014) propõem três técnicas de otimização de compilação para aumentar o desempenho geral do sistema. Enquanto uma técnica visa reduzir o fluxo de dados entre a CPU e o co-processador, a segunda lida com o acesso irregular à memória e a terceira é uma mescla das duas técnicas anteriores. As otimizações desenvolvidas melhoram o desempenho de 9 das 12 aplicações testadas em até $52,21$x. Machado (MACHADO et al., 2017) comparam o desempenho de dois compiladores: GCC e o Intel C ++ *Compiler* (ICC). Eles mostraram que a flag `-O2` fornece uma melhoria no desempenho, enquanto o `-O3` fornece a melhor aceleração na arquitetura Intel.

Até onde sabemos, os únicos trabalhos que avaliam o impacto do conjunto de otimização do compilador na resiliência à *soft errors* são os descritos em (DEMERTZI et al., 2011), (SANGCHOOLIE et al., 2014), (LINS et al., 2017), (GAVA et al., 2019) e (SERRANO-CASES et al., 2019) . Os autores (DEMERTZI et al., 2011) analisaram a relação entre as otimizações do compilador e os efeitos produzidos na confiabilidade do sistema com o uso de três estruturas de um processador - reordenamento de *buffer*, fila de busca de instruções e fila de leitura e escrita. As avaliações foram realizadas atra-

vés do código *assembly* criado pós compilação. Os autores concluíram que os códigos otimizados em sua maioria possuem tamanho e tempo de execução menor comparado à linha base e que essas características permitem um aumento na resiliência e num melhor uso do *hardware*. O trabalho de (SANGCHOOLIE et al., 2014) analisa *soft error* de 12 *benchmarks* em execução de modo *bare metal* em um processador baseado em PowerPC. O trabalho de (LINS et al., 2017) considera três *flags* de compilação (isto é, `-O0`, `-O2` e `-O3`) e seus efeitos em seis *benchmarks* executados no processador ARM Cortex-A9. O mecanismo de injeção de falha empregado depende de solicitações de interrupções enviadas de um computador para um FPGA executando a aplicação de destino. Os autores também avaliam a correlação através de experimentos com íons pesados no Laboratório Aberto de Física Nuclear (LAFN) (AGUIAR et al., 2014). Os resultados foram obtidos a partir de apenas seis parâmetros de referência e o sistema de classificação de falhas adotado é restrito a dois grupos (ou seja, desapareceu a corrupção de dados silenciosos e a interrupção da função de evento único). Como demonstrado em (ROSA et al., 2017) e (CHO et al., 2013), um sistema de classificação mais rico permite uma análise e compreensão mais profunda do comportamento do sistema sob a presença de *soft errors*, o que é vital para identificar técnicas adequadas de mitigação dessas falhas transientes. Por vez, o trabalho desenvolvido por (GAVA et al., 2019) utiliza a classificação de (CHO et al., 2013) para avaliar o impacto dos conjuntos de otimizações de 3 compiladores, incluindo 2 versões do GCC e Clang. Além dos 3 compiladores, 5 conjuntos de otimizações e o modelo de programação paralela OpenMP, foram considerados nos experimentos, que foram conduzidos através do injetor de falhas OVPSim-Fim (ROSA et al., 2017). Neste trabalho, os autores avaliaram 16 *benchmarks* executando sobre o modelo do processador Arm Cortex-A72, variando o número de elementos de processamento, i.e., *single*, *dual* e *quad-core*. Resultados mostraram que o compilador Clang apresenta uma resiliência 15% maior que as versões do compilador GCC. (SERRANO-CASES et al., 2019) demonstra que a confiabilidade pode ser aprimorada ajustando o processo de compilação. Eles propõem uma estratégia automática para orientar a busca das versões com as melhores compensações entre diversos objetivos, que influenciam a confiabilidade das aplicações. Além disso, os resultados de suas simulações correspondem ao comportamento das aplicações sob irradiação de prótons, entretanto a avaliação é realizada em 2 *benchmarks*. Diferentemente, este trabalho considera um modelo de microprocessador em SystemC TLM com precisão de ciclo. A Tabela 3.1 posiciona o trabalho da presente Dissertação em relação ao estado da arte.

Tabela 3.1: Trabalhos que utilizam compiladores para detecção e/ou redução da susceti-bilidade de sistemas a soft errors.

Ref	Avaliação	Nro Flags	Nro Benchmark	Compilador	Processador
(JIA; CHAN, 2013)	Impacto dos níveis na detecção de corrida de dados	6	4	GCC	Desktop
(SONG et al., 2014)	Três técnicas para otimizar a compilação	3	12	GCC	Intel Xeon Phi
(MACHADO et al., 2017)	Comparação de desempe--nho entre compiladores	3	5	GCC e ICC	Desktop
(DEMERTZI et al., 2011)	Impacto das *flags* na resiliência de *soft error* em 3 estruturas do processador	3	5	GCC	Desktop
(SANGCHOOLIE et al., 2014)	Avaliação de processador em modo bare metal	4	12	GCC	MPC565
(LINS et al., 2017)	Impacto das *flags* na resiliência de *soft error*	3	6	GCC	ARM Cortex A9
(GAVA et al., 2019)	Avaliação dos efeitos de *soft error* em distintos compiladores com API OpenMP	5	16	2 versões de GCC e Clang	ARM Cortex A72
(SERRANO-CASES et al., 2019)	Técnica de compilação para encontrar app. com maior resiliência	5	2	GCC	ARM Cortex A9
Trabalho Proposto	Avaliação dos efeitos das *flags* de otimização	4	24	GCC	MIPS

3.2 Trabalhos Relacionados a Injetores de Falha

Como mencionado na Seção 2.2, existem distintas maneiras de avaliar a ocorrência e impacto de *soft errors* em sistemas *single* e multiprocessados. Esta Seção tem como objetivo posicionar este trabalho em relação à literatura, referente injetores de falhas que utilizam simulações para avaliar seus sistemas.

O primeiro grupo de trabalho está relacionado a injetores de falha no nível de portas lógicas, ou linguagens de descrição de hardware HDL (BARAZA et al., 2000) desenvolveram uma ferramenta chamada VFIT que integra uma série de elementos em VHDL que opera anexado ao simulador Modelsim. (SOLINAS et al., 2017) desenvolveram um método para injeção de falhas em FPGAs chamado NETFI-2, o qual permite emular a suscetibilidade de circuitos integrados aos efeitos da radiação. O método proposto modifica o *netlist* antes da geração do *bitstream*, que é carregado para o FPGA. Os autores comparam a sua emulação de falhas com falhas injetadas no mesmo projeto e simuladas no Modelsim. Um trabalho semelhante apresentado por (RUANO et al., 2007), estende os recursos do ModelSim para injetar falhas nas descrições de RTL usando scripts Perl e Tcl. (BORTOLON et al., 2018) apresentam um injetor de falhas não intrusivo baseados em simulações utilizando o simulador Modelsim com auxílio de scripts na linguagem Tcl que permite não só realizar análises de *soft error*, mas também identificar e entender a propagação de falhas em sistemas multiprocessados. O trabalho é dependente do simulador, no qual utiliza comandos internos para a realização da injeção de falha. Os resultados foram avaliados em três *benchmarks* leves em sistemas multiprocessados com comunicação em redes intra-chip descritos em VHDL. Seus experimentos demonstraram que 19% dos *soft error* são propagados para elementos de processamento distintos dos quais a falha foi injetada. O trabalho demanda um alto tempo para a avaliação de sistemas com uma maior complexidade. Um trabalho que visa FPGAs foi o injetor de falhas desenvolvido por (LINS et al., 2017), no qual os autores realizam a injeção de falhas através da geração de interrupção do sistema e compararam seus resultados através da exposição dos dispositivos a íons pesados. Por serem realizados em nível de RTL/FPGA, são mais próximos de uma descrição de *netlist*, apresentando assim uma maior precisão nos resultados. Todavia, a simulação em RTL é muito custosa e a prototipação é um processo bem trabalhoso que tem limite físico, o que acaba limitando a complexidade dos *testcases*. Por isso é vantajoso explorar níveis de abstração de *hardware* mais elevados, visando acelerar as campanhas de injeção de falhas, assim como prover maior flexibilidade na exploração

do desenvolvimento do projeto (e.g., uso distintos *kernels*, processadores/ISA do estado da arte, etc). Outra desvantagem da abordagem apresentada por (LINS et al., 2017) é que as modificações no projeto é intrusiva na maioria das vezes.

Por outro lado, acelerar a injeção de falha e prover possíveis alterações de projeto com uma maior antecedência são os trabalhos do segundo grupo. O trabalho proposto em (EBNENASIR; HAJISHEYKHI; KULKARNI, 2013) sugere uma maneira de modelar falhas em SystemC TLM. Um *framework* baseado em *Python* chamado ReSP foi proposto por (BELTRAME; FOSSATI, 2008). O ResSP fornece *wrappers* para ser utilizado em componentes em SystemC com suporte a distintos processadores descritos no ArchC (RIGO et al., 2004). O trabalho descrito em (SHAFIK; AL-HASHIMI, 2008) cria uma técnica minimamente intrusiva pois requer apenas a substituição dos tipos de dados ou sinais originais pelos tipos de ativadores de injeção de falhas. Os autores validaram a técnica através da comparação da descrição comportamental de um MPEG-2 em SystemC com abordagens mais evasivas de *hardware*, como mutantes e sabotadores, que requerem uma direta intrusão no *hardware* do sistema. Por último (BELTRAME; BOLCHINI; MIELE, 2009) fornecem um caso de estudo de como as falhas podem ser modeladas e injetadas em diferentes níveis de abstração de *hardware* e como estão relacionadas com o respectivo TLM. Embora SystemC reduza o custo de simulação comparado a descrições em HDL, a falta de modelos de processadores limitam o uso em sistemas complexos.

O último grupo de injetores de falhas está relacionado às plataformas virtuais. Uma plataforma virtual é um simulador completo de um sistema, que emula componentes de *hardware*, diferenciando dos demais trabalhos apresentados, onde existe uma descrição do *hardware* ou um componente físico, como no caso de injetores em FPGA. Estes simuladores fornecem um conjunto maior de distintas arquiteturas de *hardware*, entretanto possuem penalidades na precisão de seus modelos. (ROSA et al., 2017) propõem um injetor de falhas que utiliza o simulador de plataforma virtuais com precisão de conjunto de instruções (o OVPsim (OVP, 2020)). O autor realiza campanhas de injeções de falhas com mais de 1 milhão de injeções realizadas em um processador *multicore* ARM. Os resultados são comparados com outro trabalho baseado em plataforma virtual, que utiliza a ferramenta gem5 (BLINKERT et al., 2011) como a plataforma virtual para emular os processadores apresentados em (ROSA et al., 2015), onde, em alguns casos, obteve uma disparidade superior a 20% dos resultados. Um *framework* baseado no emulador e virtualizador de máquinas de código aberto QEMU (QEMU, 2020) foi proposto por (de

Tabela 3.2: Estado-da-Arte em Injetores de Falha comparado com o trabalho proposto

Ref	Ambiente de Injeção de Falhas
(BARAZA et al., 2000)	Simulador - RTL
(SOLINAS et al., 2017)	Simulador - RTL e FPGA
(RUANO et al., 2007)	Simulador - RTL
(BORTOLON et al., 2018)	Simulador- RTL
(LINS et al., 2017)	Simulador - RTL e FPGA
(BELTRAME; FOSSATI, 2008)	Simulador - TLM e Python
(SHAFIK; AL-HASHIMI, 2008)	Simulador - TLM
(ROSA et al., 2015)	Plataforma Virtual - gem5
(de AGUIAR GEISSLER et al., 2014)	Plataforma Virtual - QEMU
(ROSA et al., 2017)	Plataforma Virtual - OVP
Trabalho Proposto	Simulador - SystemC

AGUIAR GEISSLER et al., 2014), onde falhas são injetadas na arquitetura $X86$ executando aplicações de sistemas de tempo real. A utilização de plataformas virtuais pode ser uma alternativa para se ter um resultado mais rápido, a fim de avaliarmos os *soft errors* em fases iniciais de projeto.

A Tabela 3.2 posiciona o presente trabalho aos demais trabalhos citados. Esta Dissertação desenvolveu um injetor de falhas utilizando a linguagem SystemC e como método de avaliação foi utilizado o simulador Modelsim.

3.3 Resumo do Capítulo

Este Capítulo apresentou uma visão geral do estado da arte em diferentes temas, posicionando o trabalho desenvolvido nas respectivas áreas. Primeiro considerando os trabalhos relacionados a compiladores e confiabilidade de sistemas, onde 5 trabalhos compreendem este tema. Os demais trabalhos visam otimizar o desempenho ou detectar condições de corrida. O segundo grupo de trabalhos apresentados é referente a injetores de falhas, onde foram apresentados trabalhos em três diferentes níveis baseados em HDL, TLM e Plataformas Virtuais. Cada nível com suas vantagens e seus pontos fracos.

4 FRAMEWORK

Este capítulo descreve a implementação realizada baseada no trabalho apresentado por (BORTOLON et al., 2018), a fim de estender o suporte a linguagem SystemC TLM, visando utilizar os ganhos desta modelagem, como um aumento no desempenho e uma alta capacidade de depuração. A Seção 4.1 descreve o injetor de falhas de referência, que utiliza uma descrição RTL VHDL capaz de identificar a ocorrência de *soft errors* em sistemas multiprocessados, a plataforma utilizada capaz de gerar sistemas multiprocessados baseados em NoC e o ambiente de simulação utilizado. A Seção 4.2 descreve a extensão do injetor de referência para SystemC TLM, com o desenvolvimento do módulo de injeção de falhas. A Seção 4.3 apresenta o fluxo de injeção de falhas e, por último, a Seção 4.4 explica o modo em que é realizado a avaliação da propagação de falhas.

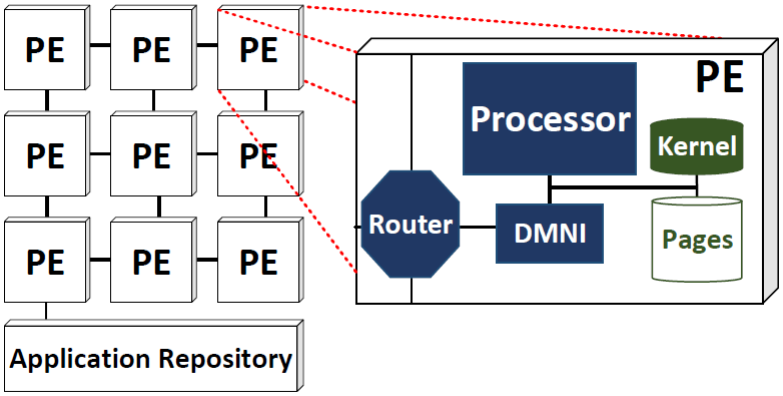
4.1 Injetor de Falhas de Referência (BORTOLON et al., 2018)

Visando suprir a lacuna ao suporte e análise da confiabilidade de *soft errors* para plataformas multiprocessadas baseadas em NoC, foi desenvolvido um *framework* não invasivo e automatizado que permite injetar falhas e avaliar seu impacto em sistemas *single* e multiprocessados. Este *framework* classifica automaticamente os efeitos de uma injeção de falhas com base no comportamento do sistema (BORTOLON et al., 2018). O método de injeção de falhas é realizado através de simulações na ferramenta Modelsim e compõe-se de quatro distintas fases: (i) simulação sem presença de falhas, com o objetivo de buscar o comportamento da aplicação no sistema (*gold model*); (ii) a configuração da falha onde é gerado aleatoriamente a localização (registrador), posição (*bit*) e o tempo de ocorrência; (iii) a simulação com a injeção de falhas e (iv) a análise e classificação de acordo com o resultado obtido.

O trabalho criou um conjunto de scripts da linguagem Tcl, optou-se por essa linguagem devido a linguagem ser muito utilizada em ferramentas de *Computer-Aided Design* (CAD). Optou-se pela utilização da plataforma de código aberto HeMPS (CARARA et al., 2009) para avaliar campanha de falhas. A Figura 4.1 ilustra a plataforma HeMPS, a qual integra módulos de *hardware* e *software*, incluindo memrória, uma NoC e elementos de processamento (PE). Cada PE contém um processador MIPS que executa um μ*kernel*, além de uma interface de rede DMNI - com acesso direto a memória (DMA).

As injeções são realizadas de maneira não intrusiva, através de comandos da fer-

Figura 4.1: Exemplo da Plataforma retirada.



ramenta Modelsim, criando uma dependabilidade de ferramenta. Outra característica é o suporte somente à simulação no nível RTL, fazendo com que as campanhas de injeção de falhas levem um alto tempo a serem executadas. O baixo desempenho de simulação influencia principalmente as etapas inicias do projeto de sistemas integrados, trazendo um custo elevado para obtenção de informações, assim como na tomada de decisão e/ou eliminação de soluções inadequadas. Estas foram as principais motivações que levaram ao desenvolvimento do suporte a SystemC. Tal abordagem apresenta duas grande vantagens: (i) facilidade de modelagem e execução de sistemas mais complexos em um menor tempo, e (ii) independência de uma ferramental comercial (e.g., Modelsim, Incisive).

4.2 Suporte a SystemC

Para avaliar a confiabilidade de *soft errors* de sistemas computacionais eletrônicos complexos, são necessárias estruturas de injeção de falhas rápidas e flexíveis. Essas ferramentas devem ser capazes de lidar com arquiteturas complexas e grandes campanhas de injeção de falhas com o objetivo de fornecer alta cobertura de falhas. Nesta direção, foi desenvolvido o *Fault Injection Module* (do inglês, FIM) automatizado e não intrusivo, baseado na linguagem Tcl padrão. O mecanismo reproduz o comportamento de *soft error* usando *single-bit-upsets* (SBUs) devido a maior probabilidade de ocorrência em sistemas eletrônicos em operação ao nível do mar (JOHANSSON et al., 1999). O modelo de SBU utilizado consiste em um *bit-flip* gerado aleatoriamente na memória e nos componentes gerais da microarquitetura de processadores descritos em SystemC. Para a campanha de falhas proposta, este trabalho considerou apenas a injeção de um *bit-flip* único nos registradores de proposito-geral do modelo MIPS a cada campanha de teste, que é a abordagem

Figura 4.2: Sinais adicionados ao *testbench* do *framework*.

```
/* @FIM SIGNALs */
/* Utilizado para indicar se a simulacao com/sem falhas */
sc_signal< int > fim_sim_type;
/* Sinal para realizar a preempcao da simulacao - timeout */
sc_signal< int > fim_hang;
/* Tempo final da simulacao sem falhas */
sc_signal< int > fim_golden_time;
/* PE para injetar a falha */
sc_signal< int > fim_pe ;
/* Tempo da injecao de falha */
sc_signal< int > fim_time;
/* Registrador alvo a ser injetado falha */
sc_signal< int > fim_reg;
/* Bit do registrador */
sc_signal< int > fim_bit;
```

comum para avaliar individualmente o efeito das falhas. Os tamanhos das campanhas de injeção foram calculados baseando-se nas equações de (LEVEUGLE et al., 2009) e visando ter uma confiança de $99,8\%$, com uma margem de erro de 5%.

O modulo desenvolvido não dependente de uma ferramenta específica de simulação em SystemC, pelo desenvolvimento de um monitor de *hardware* na plataforma para a realização da injeção de falhas. A alteração do conjunto de scripts foi necessário para garantir essa condição, diferente da versão em RTL onde é executado um comando específico da ferramenta para monitorar e realizar a injeção de falhas. Para o suporte em SystemC, sinais foram adicionados ao *testbench*, visando ter o mesmo objetivo. A Figura 4.2 apresenta os sinais que foram criados, indicando qual a classe de classificação que a simulação obteve: o tempo para realizar a suspensão da simulação em caso excedente de tempo, tempo da execução sem falhas, o PE em que será injetado a falha, tempo da injeção, registrador e o *bit* em que será injetado a falha.

Após a definição dos sinais adicionados ao *testbench* da plataforma, o próximo passo foi a criação do mecanismo para finalizar a simulação após exceder um determinado tempo específico, classificando assim como *Hang*. A dissertação adotou um tempo excedente de 20% conforme mostra a Equação 4.1, mantendo a mesma porcentagem do trabalho utilizado como referência. A plataforma HeMPS possui toda a sua descrição em SystemC com precisão de sinal de *clock* e já possui uma variável para indicar o tempo corrente. A implementação consistiu-se na adição de uma condição de avaliação do tempo

Figura 4.3: Código desenvolvido para suspender simulações que excedam 20% do tempo *gold* da simulação de referência

H

```
/* @FIM
Hang simulation if time exceed 20% of golden time
*/
void test_bench::hang() {
unsigned int int_fim_golden_time;

/* Convert FIM golden time */
int_fim_golden_time = (unsigned int) fim_golden_time.read();
/* Verificacao do sinal de relogio -ATIVO e o tempo corrente de simulac
if( clock.read() == 1 ) {
        if( ( int_fim_golden_time != 0 ) &&
        ( current_time  >= (int_fim_golden_time * 1.2))){
                fim_hang.write(1);
                sc_stop();
        }
}
}
```

com o tempo de *Hang*, este trecho de código está ilustrado na Figura 4.3.

$$t_{hang} = 1,2 * t_{gold} \tag{4.1}$$

As simulações que têm um término inesperado são classificados como UT. Esta dissertação adotou as seguintes definições para classificar uma simulação como UT:

- Leitura/Escrita inválida;
- Leitura/Escrita desalinhada;
- Divisões por 0;
- Busca por instrução inexistente.

A avaliação de Leitura/Escrita inválida é realizada na memória de cada PE. A memória interna do PE é composta por uma RAM de $64\,\mathrm{kB}$, podendo endereçar de 0x00000000 à 0x0000FFFF. O meio de avaliação foi através da verificação do endereço de memória requisitado, em conjunto com o sinal para habilitar o funcionamento da memória RAM. Caso a memória esteja habilitada e uma requisição à um endereço superior a 0x000FFFF, a avaliação de Leitura/Escrita é realizada, comparando o valor do sinal que habilita a escrita na memória, A Figura 4.4 apresenta o trecho de código desenvolvido para esta avaliação.

Figura 4.4: Avaliação de Leitura/Escrita inválida.

```
/* @FIM: Verify if the CPU does not try to access
an invalid memory position Internal RAM (64KB):
0x00000000 - 0x0000ffff*/
void pe::private_mem_fim()
{
    if( reset.read()  )
{
        fim_invalid_write        = 0;
        fim_invalid_read        = 0;
}
/* Verificacao de leitura em endereco invalido */
if( ( cpu_enable_ram.read()  ) &&
    ( addr_a.read() > 0x0000FFFF ) )
{
  /* Verificacao da escrita invalida */
        if( cpu_mem_write_byte_enable.read() != 00 )
        {
                fim_invalid_write = 1;
                sc_stop();
        }
  /* Leitura invalida */
        else
        {
                fim_invalid_read = 1;
                sc_stop();
        }
}
else
{
        fim_invalid_write = 0;
        fim_invalid_read = 0;
}
}
```

Figura 4.5: Código desenvolvido para verificação de divisão por zero

```
    /* Verificao do registrador responsavel por conter o dividendo das
if( r[ rt ] == 0 ) {
        fim_abort_arith = true;
    sc_stop();
}
```

O desenvolvimento de término inesperado na ocorrência de divisão por zero é realizado através da verificação do valor contido no registrador alvo das instruções DIV e DIVU. Ambas as instruções possuem a mesma sintaxe em *assembly*, *<DIV/DIVU $s $t>*, onde $s é o registrador que contém o dividendo e $t o registrador que representa o divisor, o resultado de ambas instruções é armazenado em 2 distintos registradores, $LO recebe o quociente da operação e o registrador $HI o resto da divisão. O trecho de código adicionado em ambas as instruções para término de simulação na ocorrência de divisão por zero é apresentado na Figura 4.5.

A última etapa do desenvolvimento para se classificar uma falha em UT transcorreu na avaliação por busca de instrução inexistente. Na descrição da HeMPS em SystemC, o conjunto de instrução do processador é realizado através do comando - *switch case*, onde cada instrução do conjunto possui um valor específico (*opcode*). Neste sentido, a implementação foi realizada através da inserção do comando para classificar e finalizar a simulação após o comando *default*, que é utilizado quando nenhuma das alternativas anteriores é verdadeira.

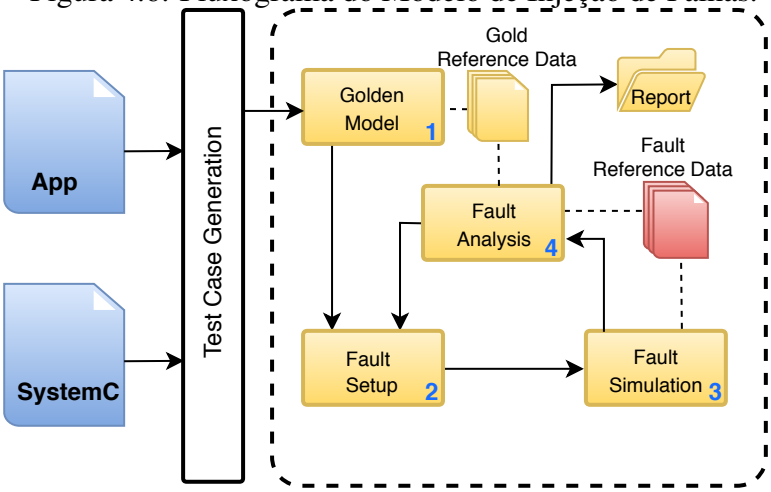
A classificação das falhas em OMM ou ONA é realizada pós-simulação, através dos *scripts* em Tcl. Após o término de cada simulação é realizado um *dump* da memória e do registrador, e estes são comparados ao equivalente do seu modelo sem falhas e são classificados como ONA. As simulações que não possuem a saída afetada, ou seja, os valores na memória de cada PE são iguais ao modelo *gold*, mas algum valor contido nos registradores difere, por exemplo, quando existe uma divergência entre a memória de um elemento, a falha é classificada como OMM. Para uma simulação ser considerada sem falhas, Vanished, os valores dos registradores, memória e o tempo de simulação deverão ser os mesmos.

Esta Seção apresentou as diferentes formas em que são avaliadas cada categoria de falha. Podemos observar que ocorre uma inserção miníma de *hardware* na arquitetura, em grande maioria somente a adição de sinais são necessários para realizar a classificação.

4.3 Fluxo de Injeção de Falhas

Esta Seção tem como objetivo apresentar o fluxo de injeção de falhas e sua iteração com os *scripts* desenvolvidos. A Figura 4.6 ilustra o fluxo de injeção de falha que compreende quatro fases.

Figura 4.6: Fluxograma do Modelo de Injeção de Falhas.



Na primeira fase, *gold model*, a arquitetura de destino é simulada na ausência de falhas para extrair o comportamento do sistema de referência. As informações obtidas compreendem o tempo total da simulação, a contagem de instruções e os estados da memória e do banco de registradores no momento em que a aplicação termina a sua execução. Esta informação faz parte da entrada da última fase do fluxo.

Posteriormente, a fase *Fault Setup* define a configuração da falha ou, em outras palavras, o local da falha (registrador), a posição (bit do registrador) e o tempo. Uma função uniforme aleatória gera esses parâmetros, que são aceitos como técnica de injeção de falhas que abrange vários tipos de falhas em um sistema a um baixo custo computacional (FENG et al., 2010). No instante de tempo $t = 0$ da simulação, o injetor insere as informações nos respectivos sinais apresentados na Figura 4.2, fazendo com que o monitor inicie seu funcionamento. Esta etapa é realizada em conjunto com um arquivo de configuração, este arquivo foi criado contendo os principais sinais úteis para obter informação da simulação. A criação deste arquivo consiste em apontar os caminhos para acessar os sinais criados, porém são adicionados outras informações relevantes do sistema como, por exemplo o número de registradores, o caminho para a criação de *logs*, e o período de *clock*. A Figura 4.3 apresenta um modelo de arquivo de configuração para um sistema com apenas um PE. Por ser um arquivo que exige poucas informações, este modelo é facilmente alterado podendo garantir suporte a diferentes arquiteturas, tornando

Figura 4.7: Exemplo do Arquivo de Configuração do Injetor de Falhas.

```
LOG_PATH          log/FIM
RESET             /test_bench/reset
CLOCK_PERIOD_NS   10
FAULT_DELAY_NS    5
PC_REGISTER       32
PE0_PATH          {[configure_path 0]}
0_PE_RAM          {PE0_PATH}/ram/ram_data
0_PE_RG           {PE0_PATH}/cpu/state_instance.r
PE_PATH           {[configure_path target_pe]}
PLASMA_PATH       {PE_PATH}
CPU_PATH          {PLASMA_PATH}/cpu
RAM_MEM_SIZE      32
PC_REG            {CPU_PATH}/state_instance.pc
END_SIM_REG       {PE0_PATH}/end_sim_reg
MEM_I_WRITE       {PLASMA_PATH}/fim_invalid_write
MEM_I_READ        {PLASMA_PATH}/fim_invalid_read
MEM_UA_WRITE      {PLASMA_PATH}/fim_unaligned_write
MEM_UA_READ       {PLASMA_PATH}/fim_unaligned_read
ABORT_FETCH       {CPU_PATH}/fim_abort_fetch
ABORT_ARITH       {CPU_PATH}/fim_abort_arith
HANG              {/test_bench/fim_hang}
SC_OK             0
SC_TE             1
SC_AF             2
SC_AA             3
SC_IW             4
SC_IR             5
SC_UW             6
SC_UR             7
```

o injetor uma ferramenta flexível.

A terceira fase, denominada de simulação de falha (*Fault Simulation*), a arquitetura de destino é simulada na presença de uma falha com as características definidas anteriormente. A injeção de falha ocorre quando o tempo de injeção é equivalente ao tempo corrente do sistema. A injeção é realizada através de um XOR lógico com o valor 1 no *bit* específico do registrador. Nesta fase, coletam-se as mesmas informações do projeto que a primeira fase para fins de comparação. A última etapa é a análise dos dados extraídos (*Fault Analysis*). A análise e a simulação de falhas são processos interativos, ou seja, para cada falha definida na fase *Fault Setup*, uma simulação é executada e os resultados são analisados.

4.4 Propagação de Falhas

Em arquiteturas multiprocessadas, é desejável avaliar não apenas o impacto de uma falha em um PE individual, mas se a falha injetada também é propagada para outros PEs. Esta dissertação define um erro como propagado quando atende às seguintes definições.

- **Definição 1**. Um erro é considerado propagado quando uma falha injetada em um PE afeta a memória e/ou banco de registrador de outro PE.
- **Definição 2**. No final da execução de uma aplicação, um banco de registradores ou memória é considerado afetado quando armazena um valor diferente do sistema sem falhas. Isso é válido apenas quando a aplicação é encerrada espontaneamente.

A segunda definição exclui os casos em que a simulação termina inesperadamente (UT) e simulações que excedem o tempo limite de 20%, casos que são classificados como *Hang*. Com essas definições, consideramos, para fins de cálculo como 100% a soma das ocorrências de OMM e ONA. O desenvolvimento da avaliação de propagação é realizado na etapa de análise do fluxo do *framework*. Desconsidera-se a comparação com a memória e banco de registradores do PE injetado e realizando a comparação dos demais PEs classificando a ocorrência de propagação na memória, banco de registradores ou ambos os casos. Vale ressaltar que a falha injetada pode propagar para mais de um PE.

4.5 Resumo do Capítulo

Este Capítulo apresentou o trabalho utilizado como referência para as implementações desenvolvidas, apontou alguns pontos negativos deste trabalho, como ser dependente do simulador e ter uma alta demanda de tempo para execução de uma campanha de falha. Apresentou, ainda, o desenvolvimento realizado para garantir suporte do framework a SystemC, explicando todos os elementos necessários para injeção de falhas e a métrica utilizada para classificação de falhas e avaliação de propagação de falhas.

5 ANÁLISE DE FALHAS EM ALTO NÍVEL

O Capítulo5 está organizando da seguinte forma: Seção 5.1 contém a avaliação de precisão entre distintos modelos de abstração de *hardware* (RTL e SystemC TLM). Logo após, o impacto das *flags* de otimizações do compilador na ocorrência de *soft error* é investigado na Seção 5.1.2. A Seção 5.2 tem como objetivo expandir as avaliações realizadas sobre *Soft Errors* em sistemas multiprocessados levando em consideração a propagação de falhas na rede.

5.1 Avaliação entre os níveis RTL x TLM

Todas as campanhas de injeção de falhas utilizaram a mesma configuração demonstrada na Tabela 5.1. As aplicações foram compiladas sem a realização de alguma otimização, ou seja com a *flag* `-O0`, onde o código é diretamente convertido nas respectivas instruções, evitando que possíveis otimizações realizadas pelo compilador mascarem os resultados.

5.1.1 Desempenho Entre Níveis

Avaliar o desempenho de simulação entre as abordagens de injeção é importante para que os projetistas possam compreender o possível ganho de tempo e então definir a maneira mais efetiva entre as possíveis campanhas de injeção a serem conduzidas. De posse dessa informação, torna-se possível avaliar quais parâmetros entre os diversos contidos no projeto de sistemas multiprocessados possam ser considerados. Os tempos foram extraídos através das informações fornecidas pelo simulador e armazenadas em *logs* após o término da campanha de injeção executada.

Tabela 5.1: Configuração dos Experimentos de Avaliação de Precisão de Modelos (RTL x TLM)

Processador	*Plasma*
Níveis de Abstração	RTL e TLM
Sistema Operacional	-
Otimização do Compilador	O0
*Número de Aplicaçõe*s	*24*
Número de Cenários	*48*
Injeções de Falha por Cenário	*1000*

Tabela 5.2: Tempo de Execução da Campanha de Injeção em RTL e TLM

Aplicação	Tempo RTL(horas)	Tempo TLM(horas)
Adpcm	24,35	3,30
BSearch	13,47	1,97
bitManipulation	12,13	1,50
Bfsh	17,28	9,93
Bubble	27,91	5,61
Compress	34,80	1,26
Counts	33,55	6,17
Crc	21,79	4,54
Expint	21,21	4,54
Factorial	27,86	5,14
Fdtc	20,23	6,73
Fibonacci	24,81	4,59
Hanoi	28,25	11,86
Harm	12,80	2,15
iSort	20,08	3,07
Jfdct	16,67	2,73
mMult	21,07	5,35
mdc	22,71	8,01
pSpeed	10,39	3,06
Petri	16,58	4,03
Prime	23,81	4,62
sCases	9,03	1,95
Ud	17,41	3,65
Usqrt	18,09	4,86
μ (horas)	20,68	4,62

A primeira avaliação foi aferir o desempenho de simulação entre as duas abordagens de injeção, para isto, a média aritmética (μ) dos tempos de cada nível de abstração, foi considerado como comparação. Os resultados são amostrados na Tabela 5.2, onde observa-se um ganho médio de $4,5$x no tempo de execução das aplicações em SystemC TLM. Um dos motivos dessa melhoria de desempenho é que no nível VHDL RTL, por exemplo, é que toda a estrutura de uma máquina de estágios necessita ser descrita, isto resulta em um projeto maior e menos reutilizável (ERICKSON, 2016). Entretanto, a diferença de tempo altera conforme a aplicação, com o melhor caso chegando a um aumento de $27,47$x e no pior caso uma melhora de $2,3$x. Com este ganho é possível simular projetos de circuitos integrados mais complexos em menor tempo, tornando-a escolha deste modelo de abstração de *hardware* seja uma opção para projetistas.

5.1.2 Precisão das Arquiteturas Baremetal

O segundo conjunto de avaliações realizadas, concerniu em aferir a disparidade ocorrida na classificação de falhas. Considerando um determinado conjunto de aplicações, utilizou-se primeiramente somente um processador sem sistema operacional. Este trabalho adotou um conjunto de 24 aplicações distintas e o processador MIPS. Visando garantir uma maior precisão dos resultados, o conjunto de aplicações foi compilado sem alguma otimização, ou seja, com a *flag* `-O0` do compilador GCC, onde cada comando do código fonte é convertido diretamente à instrução correspondente no arquivo executável.

A comparação realizada entre as distintas classes de falhas foi a métrica utilizada para a avaliação de precisão entre os modelos. Foi comparado cada classe de falha com o seu respectivo entre os modelos. Essa comparação gera uma disparidade maior nas porcentagens de valores do que uma comparação mais simples, como agrupar falhas e sem falhas, visto que o aumento em uma categoria de falha implica na diminuição de outra.

As Figuras 5.1 e 5.2 apresentam o resultado de duas campanhas de falhas. A primeira, corresponde à classificação da aplicação *hanoi*, que obteve a maior divergência entre as categorias de somente 1% nas simulações e um ganho no desempenho de 2.3x, conforme o tempo extraído da Tabela 5.2. O outro exemplo apresentado é a classificação da aplicação *prime*, que calcula os 40 primeiros números primos. A aplicação apresentou uma melhoria no desempenho de $4,1$x e uma discrepância de apenas 1% em falhas classificadas como OMM e UT.

Figura 5.1: Classificação de Falhas da Aplicação hanoi em Distintos Níveis de Abstração.

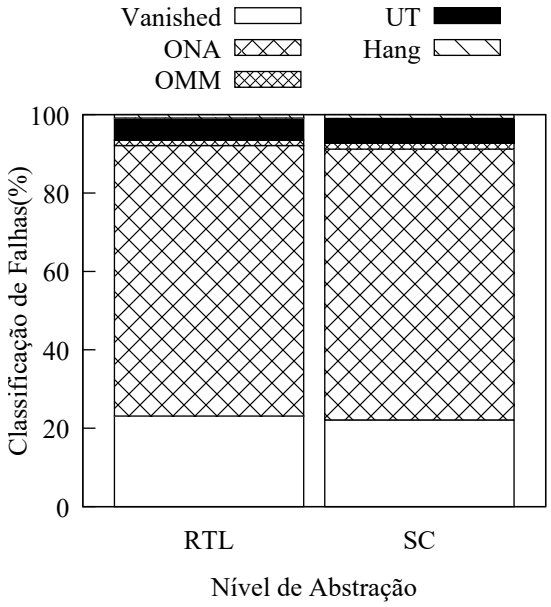
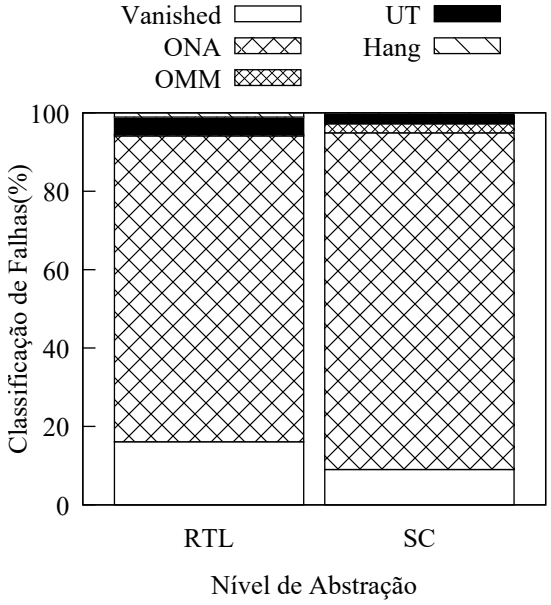


Tabela 5.3: Divergência Média Entre Classificações

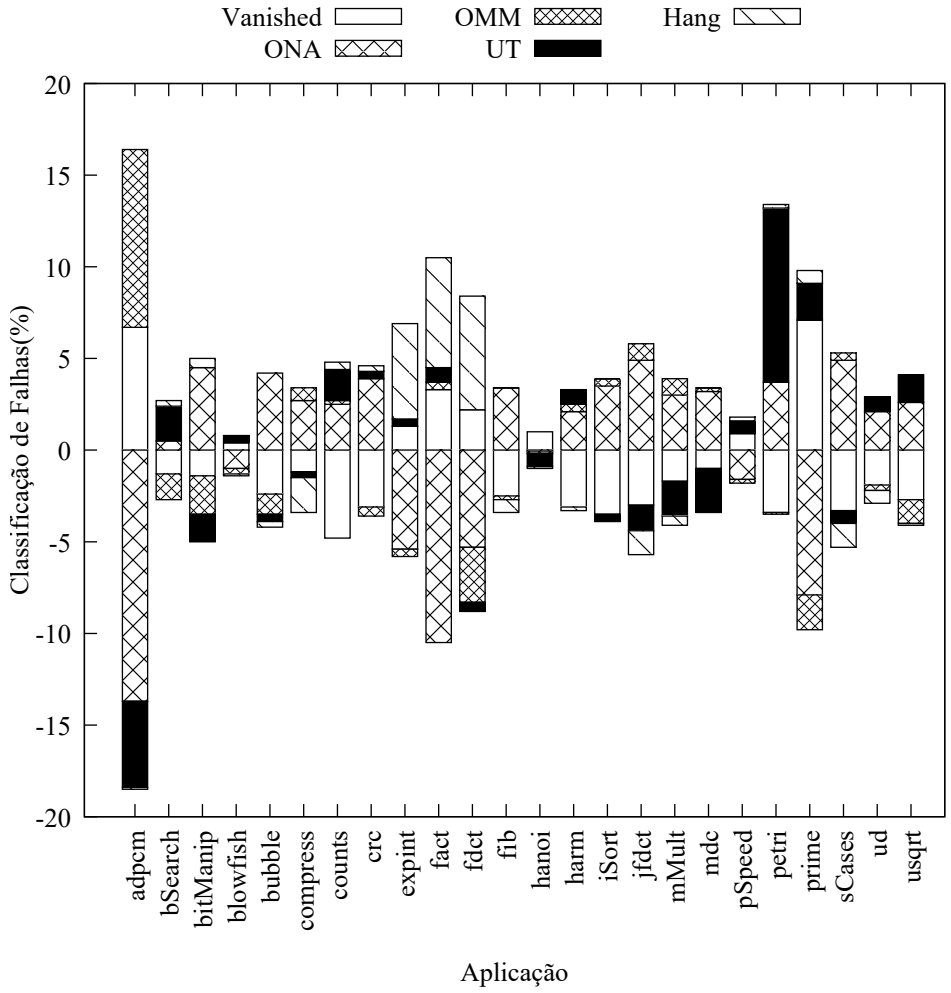
	Divergência Média (%)
Vanished	2,38
ONA	4,05
OMM	1,12
UT	1,49
Hang	1,1

Figura 5.2: Classificação de Falhas da Aplicação prime em Distintos Níveis de Abstração.



Expandindo a análise para todo o conjunto de *benchmarks*, foi realizado o cálculo da incompatibilidades entre os resultados das campanhas de falhas por aplicação, ou seja, VHDL (%) - SystemC (%), por aplicação, os resultados são apresentados na Figura 5.3. Onde obteve-se o pior caso, uma disparidade entre as classificações de falhas dos modelos foi na classe de ONA da aplicação ADPCM (do inglês, Adaptive Differential Pulse-Code Modulation) (CUMMISKEY; JAYANT; FLANAGAN, 1973) 13.70%, que visa converter som ou informação analógica em informação binária, muito utilizado na área de processamento digital de sinais. O código recebe uma amostra de $16\,\mathrm{kHz}$ e realiza diversas séries de decodificação, quantização e codificação de dados até o fim da aplicação, constituindo-se de um alto número de operações matriciais, deslocamentos lógicos e leitura/escrita de memória, estas características impactaram na maior divergência de resultado. A média de divergência de cada categoria é apresentado na Tabela 5.3. As aplicações obtiveram uma divergência média inferior a 5%, fazendo assim, que a escolha por SystemC, seja uma boa opção levando em consideração o aumento do desempenho de simulação obtido, em até $27,47$x.

Figura 5.3: Divergência dos Resultados Entre os Modelos RTL vs TLM.



5.1.3 Precisão dos Modelos com Arquiteturas Multiprocessadas

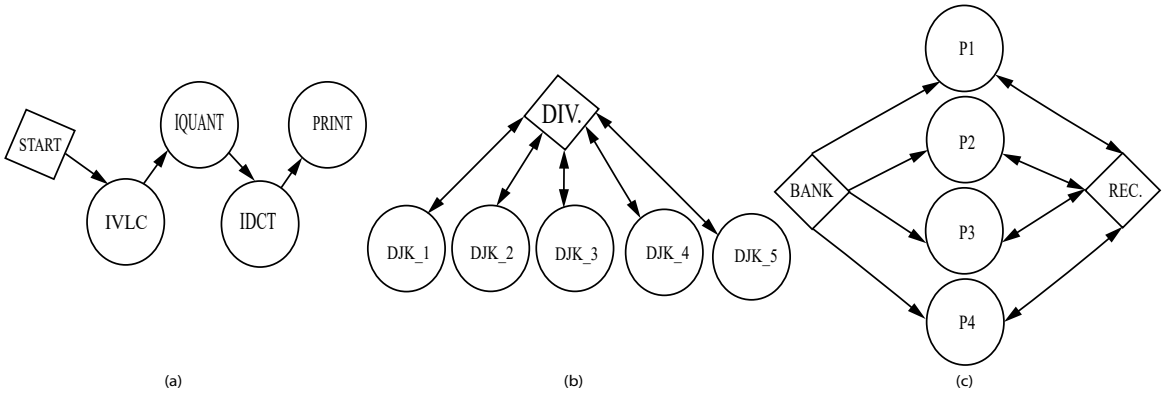
Avançando na avaliação de precisão entre os modelos, um segundo conjunto de experimentos foi realizado, visando aferir a precisão entre sistemas mais complexos. Utilizando a plataforma HeMPS, foram criados *testcases* considerando 3 distintas aplicações, aonde cada *testcase* foi executado em uma rede-intrachip com 4 PEs, nesse caso com dimensão $2x2$. A HeMPS utiliza a arquitetura mestre-escravo, onde um PE é responsável pela distribuição das tarefas e controle do sistema. Nos experimentos realizados o PE0 é o mestre e os demais PEs são responsáveis pela execução de cada conjunto de tarefas. As aplicações utilizadas foram: MPEG, DTW e Dijkstra, cada uma das distintas aplicações possuem diversos comportamentos, garantindo um vasto intervalo de resultados, o comportamento de cada aplicação representado por um grafo, onde as tarefas iniciais e finais são representadas por um quadrado, setas indicam o sentido de comunicação e dependência e os círculos as demais tarefas. A Figura 5.4 apresenta o grafo de cada

aplicação. *Moving Picture Experts Group* (MPEG) possuí uma baixa comunicação, entretanto um acesso elevado a memória. Por outro lado, a *Dynamic Time Warping* DTW realiza uma série de comunicação entre as tarefas da aplicação, em uma sequência similar a produtor-consumidor. Por sua vez, similar a DTW, a plicação Dijkstra, possui uma alta comunicação entre as tarefas da aplicação, entretanto de uma maneira distinta à DTW, concentrando o resultado em uma tarefa distinta. As configurações são apresentadas na Tabela 5.4.

Tabela 5.4: Configuração dos Experimentos de Avaliação de Precisão de Modelos em Sistemas Multiprocessados (RTL x TLM)

Processador	*Plasma*
Modelos de Abstração	RTL e TLM
Dimensões da NoC	2x2
Sistema Operacional	μkernel
Otimização do Compilador	O0
*Número de Aplicaçõe*s	*3*
Número de Cenários	*12*
Injeções de Falha por Cenário	*1000*

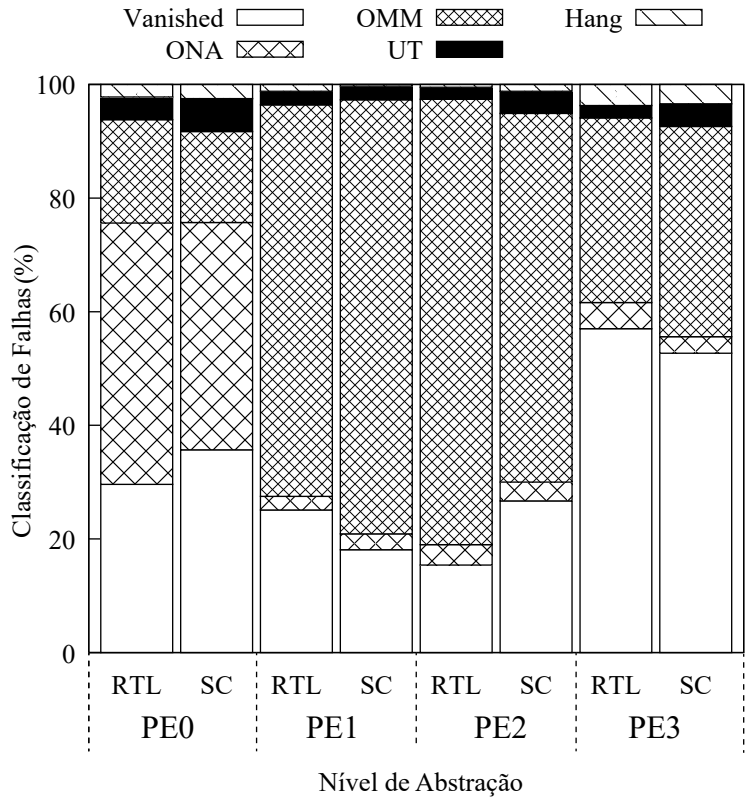
Figura 5.4: Grafos de Comportamento das aplicações: (a) Mpeg, (b) Dijkstra e (c) DTW.



A primeira avaliação foi a comparação da aplicação MPEG em ambos os níveis. Esta aplicação tem 5 tarefas distintas, os resultados obtidos são expostos na Figura 5.5. Pode-se observar uma maior discrepância na campanha de injeção realizada no PE2, onde ocorre um divergência na classe OMM dos valores entre os modelos de $13,5\%$. Entretanto a divergência média de todas as classes e todos de PEs para essa aplicação é de somente $3,61\%$.

Seguindo a análise da comparação entre os níveis de abstração, a aplicação Dijkstra apresentou um pequeno aumento na disparidade entre os modelos, chegando ao pior caso de $16,1\%$ em falhas classificadas como OMM, injetadas ao PE2. O conjunto de ta-

Figura 5.5: Classificação de Falhas da Aplicação MPEG em Distintos Níveis de Abstração em uma NoC $2x2$.



refas da aplicação Dijkstra, possui uma dependabilidade de comunicação onde uma tarefa produz os valores iniciais e a mesma tarefa aguarda o recebimento dos dados computados, fazendo com que seja mais suscetível a falha, de OMM. Entretanto a divergência média é de apenas $4,96\%$.

Por fim, a aplicação que teve a menor disparidade foi a DTW, chegando à somente $9,6\%$ na divergência entre falhas OMM, quando injetadas no PE0. A aplicação DTW é a que contém a maior taxa de comunicação entre as tarefas do conjunto. Contudo, teve uma divergência média de apenas $2,97\%$, a Figura 5.7 apresenta a classificação da campanha de injeção realizada.

As campanhas realizadas obtiveram uma divergência média por categoria apresentada na Tabela 5.5. A maior discrepância está situada nas classificações OMM, ou seja, quando a saída da aplicação é divergente ao modelo *gold*, entretanto este é um valor pequeno. O *hardware* descrito em SystemC TLM provou ter uma precisão aceitável comparado à VHDL, mesmo quando foram realizados experimentos mais complexos, no caso, envolvendo 4 PEs, cada um executando um μkernel e uma comunicação de rede. Fazendo com que seja uma boa escolha para projetistas que desejam ter uma pequena desvantagem na precisão, porém com um alto ganho de desempenho em tempo de simulação.

Figura 5.6: Classificação de Falhas da Aplicação Dijkstra em Distintos Níveis de Abstração em uma NoC $2x2$.

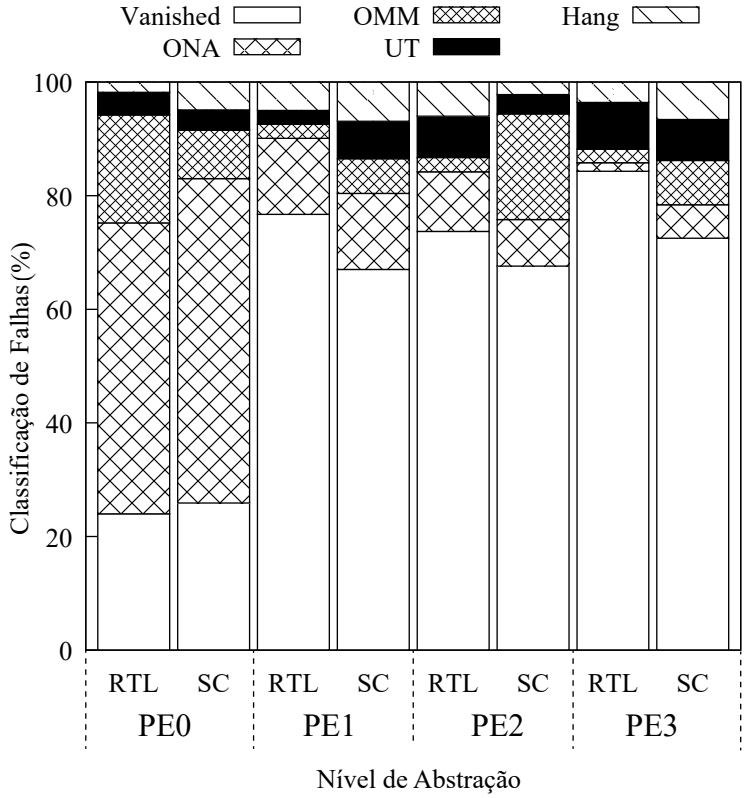


Figura 5.7: Classificação de Falhas da Aplicação DTW em Distintos Níveis de Abstração em uma NoC $2x2$.

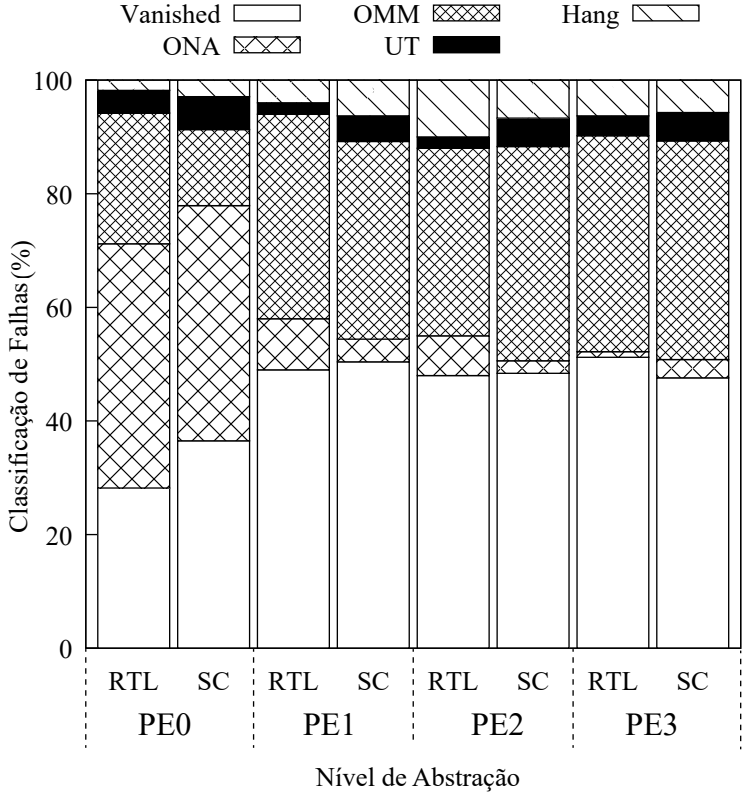


Tabela 5.5: Divergência Média Entre Classificações de Falhas nos Sistemas Multiprocessados

	Divergência Média (%)
Vanished	6,51
ONA	2,71
OMM	6,78
UT	1,88
Hang	1,62

5.2 Avaliação dos efeitos das Flags de otimização do compilador

Uma vez avaliado a precisão entre os distintos modelos, um segundo estudo foi realizado para investigar o impacto das *flags* de compilação na ocorrência de *soft error*. Para validar os efeitos das otimizações, adotamos um processador MIPS descrito em SystemC com um banco de registradores, contendo 32 registradores distintos. Cada um com $32bits$. A Tabela 5.6 descreve a extensão dos cenários, cada cenário consiste em uma aplicação combinada com uma *flag* `-Olevel`, permitindo assim uma comparação mais precisa entre níveis.

Tabela 5.6: Configuração dos Experimentos

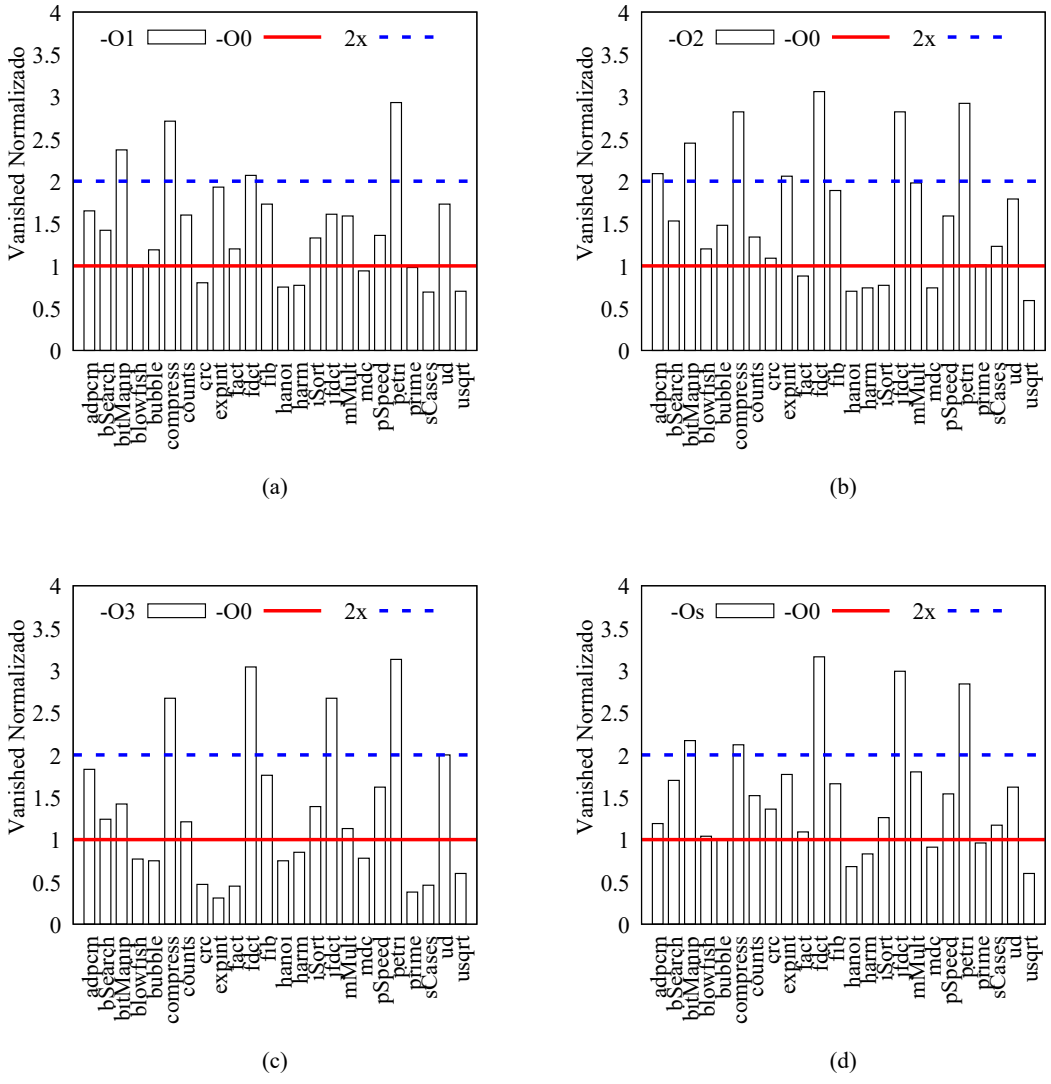
Plasma	*Plasma (MIPS)*
Pilha de Software	*Bare Metal*
Número de Aplicações	*24*
Número de Cenários	*120*
Injeções de Falhas por Cenário	*1.000*

5.2.1 Resiliência de *Soft Error*

Esta subseção apresenta os resultados de todas as campanhas de injeção de falhas conduzidas. Os gráficos da Figura 5.8 são normalizados para a opção onde nenhuma otimização é realizada, ou seja `-O0`. Em tais gráficos, aplicações (isto é, barras) abaixo da linha vermelha correspondem às aplicações mais suscetíveis a falhas, ou seja, uma redução do número de *Vanished* em comparação com `-O0`. Em adição à linha normalização (ou seja, a linha vermelha), uma segunda linha $y = 2$ foi desenhada para categorizar as aplicações que obtiveram um maior aumento na resiliência pelo menos de 2 ordens de grandeza. Nesta categoria estão aplicações com uma melhoria regular na resiliência, ou seja, valores entre 1 e 2, enquanto aquelas acima de $2x$ são considerados como um ganho mais significativo. A escolha de realizar esta avaliação em vez de meramente tomar a

média como métrica é devido ao fato que poucas aplicações com ganhos consideráveis mascarem aqueles com maior suscetibilidade a falhas, ou seja, abaixo da linha de normalização. Essa abordagem prova ser uma métrica útil para computação de propósito geral, que permite identificar o comportamento médio do sistema em vez de apenas a tolerância média.

Figura 5.8: Número de falhas *Vanished* para (a) `-O1`, (b) `-O2`, (c) `-O3` e (d) `-Os` todas normalizadas em `-O0` para todas as aplicações. A linha vermelha sólida representa os resultados da linha de base (`-O0`), enquanto a linha azul tracejada representa $2x$ o ponto de melhoria.



A Figura 5.8(a) apresenta os resultados para `-O1`, normalizado para a linha de base definida. Os resultados mostram que o uso da *flag* `-O1` reduz o número de falhas, o que leva a uma melhoria de $1.66x$. No entanto, as aplicações abaixo da linha de normalização, ou seja, $y = 1$, indicam que a suscetibilidade do circuito à falha aumentou.

Em média, com um melhor caso de $2.93x$, e o pior com caso uma redução de 31%. Além disso, é possível verificar que $66,67\%$ dos *benchmarks* considerados permaneceram acima da linha de normalização (Tabela 5.7). Os resultados subjacentes demonstram que o comportamento geral do sistema, para uma ampla gama de *benchmarks* distintos, tende a melhorar com o uso dessa *flag*.

A Figura 5.8(b) ilustra a comparação entre as *flags* `-O2` e `-O0`. Os resultados descritos mostram que o melhor caso tem um aprimoramento significativo de $3,06x$, enquanto o pior caso apresenta uma penalidade de resiliência de 41%, o que está longe das melhorias obtidas. Além disso, a Figura 5.8(b) indica que 75% dos *benchmarks* estão acima da linha de normalização. `-O1` e `-O2` apresentam melhorias devido à reordenação de instruções, otimização de laços de repetição e ramificações. Portanto, isso demonstra que a otimização altera a dependência de dados, aumentando a probabilidades de falhas serem substituídas. No entanto, a partir da comparação entre as estatísticas `-O1` e `-O2`, é possível observar que `-O2` possui um melhor comportamento do sistema, isto é, 75% das aplicações estão acima da linha de normalização com algumas aplicações com ganhos consideráveis, maiores que 2 vezes. Observa-se que utilizar apenas a média como métrica não levaria à mesma conclusão que os valores de `-O1` e `-O2` são muito próximos, $1,66$ e $1,62$, respectivamente.

A Figura 5.8(c) descreve o nível de otimização `-O3`, cujo foco principal é melhorar o desempenho do código. Comparado com `-O0`, o principal recurso que aumenta a velocidade é ativar as funções *inline*. No entanto, a desvantagem dessa abordagem é uma dependência aumentada do uso dos registradores, o que reduz a resiliência a erros de *software*. Analisar os resultados pela média levaria, novamente, a conclusões errôneas, pois `-O3` possui uma melhoria média de $1,36x$, o que significa que um maior número de falhas desapareceria comparado ao `-O0`. No entanto, a média, neste caso, é altamente dependente de quatro aplicações que apresentam uma melhoria superior a $2x$. Complementando, a Figura 5.8(c), a Tabela 5.7 mostra que em 11 *benchmarks*, 45% apresentaram pior resiliência. A avaliação utilizada nos gráficos das três categorias, representa um comportamento mais preciso do sistema de maneira geral. Concluindo, a escolha da *flag* que visa otimizar a melhoria do desempenho deve ser bem avaliada, quase metade das aplicações apresentou um aumento de falhas para `-O3`, indicando que este não é uma *flag* útil para os propósitos apresentados.

A Figura 5.8(d) mostra a comparação entre a linha de base e o `-Os`, que contém quase todos as *flags* `-O2`, mas evita aqueles que afetam o tamanho do código. Em

contraste com `-O2`, algumas das otimizações são desativadas como o alinhamento de funções, laços, ramificações e saltos. Os resultados mostram que há um aumento de resiliência de $1,79x$ em todos os *benchmarks*, proporcionando um melhor caso de $3,16x$ para uma aplicação e um pior caso com diminuição de 40% da resiliência. Comparando a média de `-O2` e `-Os`, $1,62x$ contra $1,54x$, um projetista escolheria a primeira *flag*. No entanto, analisando a Tabela 5.7, é possível notar que `-Os` possui mais aplicações acima a linha de normalização, ~5% mais, sugerindo assim que esta *flag* fornece um melhor comportamento geral do sistema.

Portanto, embora `-Os` tenha um pouco menos de média que `-O2`, parece preferível ter mais aplicações acima da normalização, ou seja, melhorias de `-O0`. Em resumo, a otimização de aplicações é essencial para aumentar a resiliência do sistema. Otimizações como agendamento de instruções, uma melhor avaliação de saltos e desvios, geram um aumento na resiliência de até 79% do conjunto de aplicações. As otimizações de declaração de laços e reordenamento de instruções está diretamente relacionada à resiliência. Além disso, o uso da métrica de análise proposta (Tabela 5.7) permitiu uma melhor avaliação do comportamento geral das aplicações. Consequentemente, o aumento mais significativo de instruções de controle nas aplicações e falhas ausentes foi `-Os`. Por outro lado, se a média fosse usada como base, o nível que teria o maior número de falhas desaparecidas seria `-O2`. Embora a média seja uma métrica útil para avaliar o número de falhas ausentes, ela não indica quantas aplicações foram aprimorados ou piorados.

Tabela 5.7: Porcentagem de aplicações classificadas por intervalo de melhoria

	Intervalo Normalizado em *Vanished* [%]		
`-Olevel`	$y < 1$	$1 \leq y > 2$	$y \geq 2$
`-O1`	33.33	50	16.67
`-O2`	25	45.83	29.17
`-O3`	45.83	33.33	20.83
`-Os`	20.83	58.33	20.83

5.2.2 Relação de *Soft-Error* com `-Olevel`

Esta subseção explora ainda mais os resultados usando a classificação proposta por (CHO et al., 2013) para avaliar detalhadamente os efeitos dos *soft-errors* nos experimentos. Como a classificação de falhas está fortemente relacionada ao comportamento da aplicação, esta subseção, primeiro, analisa os resultados de duas aplicações separadas. Após essa discussão, é fornecida uma análise geral. Para relacionar o impacto que cada *flag* cria nas aplicações e a classificação das falhas. Além disso, este trabalho analisou as

instruções que compõem cada aplicação para entender melhor a relação entre sua estrutura e o resultado da falha As instruções são classificadas em três grupos: (i) operações de memória; que consistem em instruções de escrita/leitura; (ii) operação de controle com saltos e ramificações; e (iii) operações relacionadas à Unidade Lógica Aritmética (ULA).

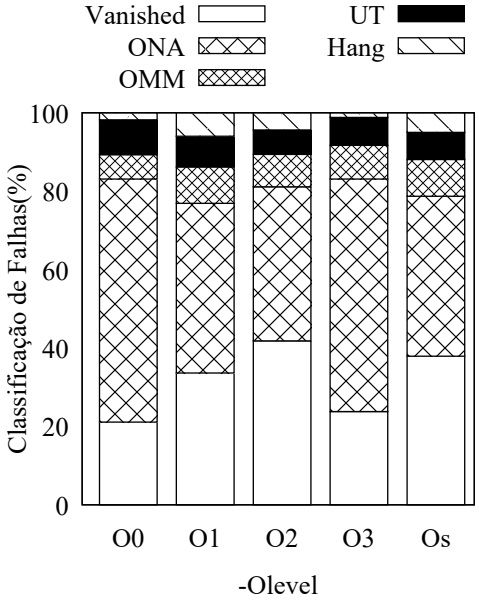
A Figura 5.9 apresenta os resultados das simulações obtidas do *benchmark* mMult, que é uma aplicação essencial no campo da engenharia. Esta aplicação é composta principalmente de acesso à memória e operações matemáticas, conforme demonstrado na Tabela 5.8. Comparando `-O1`, `-O2` e `-Os`, é possível observar que eles têm uma distribuição de classificação de falha semelhante. Todas elas apresentam uma alta diminuição nas operações de memória, uma redução no número total de instruções para 35%, 21% e 20% e um aumento nas operações de controle e ULA quando comparadas à linha de base. Essa diminuição nas instruções de operação da memória gera uma aplicação com uma menor dependências de memória e, consequentemente, uma melhoria na velocidade, pois o acesso à memória requer um tempo maior comparado a operação da ULA. Mesmo que essa otimização reduza o número de instruções de memória, há um aumento na ocorrência de OMM. Isso ocorre devido a otimização adicionar mais instruções de controle, por exemplo, saltos e ramificações. Para esta aplicação, dentro de um laço de repetição, existem diversas operações de escrita na memória e, portanto, qualquer aumento ou diminuição no número de iterações levará a um estado final diferente da memória. Em `-O0`, 36% do número total de instruções são leituras e $4,4\%$ escritas, enquanto em `-O1` a porcentagem dessas instruções é de 14% e $6,5\%$. Portanto, a Tabela 5.8 mostra que a redução das operações de memória não implica diretamente em menor ocorrência de OMMs. Esse aumento nas instruções de controle também afeta a classificação *Hang*, que também aumentou nas *flags* `-O1`, `-O2` e `-Os`. Com maior número de instruções de controle, há uma maior probabilidade de modificar uma estrutura crítica da aplicação e, portanto, nunca terminar sua execução. Por outro lado, houve uma diminuição nas classificações de falhas ONA e UT. Para esta aplicação específica, `-O2` apresentou os resultados mais relevantes em falhas desaparecidas em comparação com os demais `-Olevel`. Isso ocorre devido ao agendamento das instruções, onde o compilador tenta reordenar as instruções para eliminar as paralisações, criando mais operações de controle. Os resultados são semelhantes à técnica de reprogramação de instruções sugerida por (YAN; ZHANG, 2005), que visa reduzir o atraso nas operações, atingiu uma média de $10,9\%$ em falhas desaparecidas. Esta aplicação foi a mesma utilizada em experimentos de íons pesados por (LINS et al., 2017). A conclusão encontrada pelos autores é que `-O2` aumentou a confiabilidade do sistema.

Tabela 5.8: Classificação de Instruções da Aplicação mMult

Aplicação	-Olevel									
mMult	O0	%	O1	%	O2	%	O3	%	Os	%
Total de Instruções	493230	**100**	173581	**100**	104882	**100**	76492	**100**	103051	**100**
Leitura	177456	**36**	24873	**14,3**	24874	**23,7**	16058	**21**	24871	**24,1**
Escrita	21892	**4,4**	11364	**6,5**	11365	**10,8**	2951	**3,8**	11362	**11**
Saltos	1651	**0,4**	1651	**0,9**	1649	**1,5**	41	**0,05**	1647	**1,5**
Ramificações	19341	**3,9**	9598	**5,5**	9618	**9,1**	9598	**12,5**	9618	**9,3**
Operações de ULA	272890	**55,3**	126095	**72,6**	57376	**54,7**	47844	**62,5**	55553	**53,9**

Isso é importante para demonstrar que, mesmo com um modelo de abstração mais alto, os resultados obtidos são consistentes com experimentos reais.

Figura 5.9: Classificação de Falhas da Aplicação mMult.

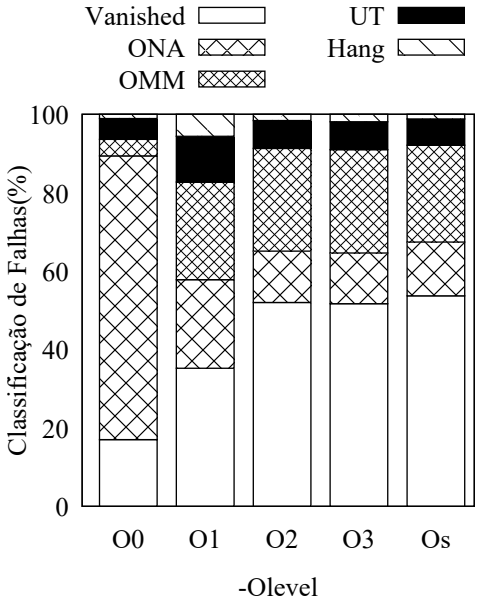


Com `-O3`, houve um aumento notável na ocorrência de falhas de OMM para esta aplicação específica. A razão para este comportamento é que, nesse nível, o compilador move as condições invariantes dos laços para fora do laço, eliminando uma porcentagem maior de saltos e inserindo mais operações de ramificações. Isso ocorre para aumentar o desempenho na execução do laço. A Tabela 5.8 mostra que as instruções de controle representam 12% do total de operações nesse nível, enquanto o `-O0` possui apenas 4% de controle. De todas as instruções de controle para o nível `-O3`, $99,06\%$ são Ramificações e $0,04\%$ são Saltos. Por outro lado, para `-O0`, 93% das instruções são Ramificações e 7% são Saltos.

Expandindo as análises para outra aplicação, a *Forward Discrete Cosine Transform* (FDCT), que utiliza blocos de imagens para calcular a quantização da imagem. Esta aplicação é composta principalmente de cálculos matemáticos envolvendo blocos de ima-

gens com constantes em vários ciclos de repetição. A Figura 5.10 mostra os resultados da classificação de falhas no *benchmark* FDCT. Como pode ser observado a partir desta figura, `-O1` tem um aumento no OMM, UT e Hang. A Tabela 5.9 demonstra a classificação das instruções para esse *benchmark*. Como discutido anteriormente, o aumento no OMM está relacionado ao maior número de instruções de ramificação e operações de escrita de memória. O aumento da UT também está relacionado ao aumento da operação da memória, pois aumenta a probabilidade de acessar um endereço de memória inválido/privado. Por fim, o aumento de *Hang* está relacionado ao aumento nas instruções de controle, fazendo com que o sistema entre em um estado desconhecido.

Figura 5.10: Classificação de falhas da aplicação FDCT



A otimização de `-O3` não modifica significativamente o código resultante quando comparado a `-O2` para esta aplicação específica, porque as diferentes *flags* avaliadas em `-O3` podem ser utilizadas ou não. Portanto, seus resultados são semelhantes, pois possuem código executável semelhante, com uma diferença de apenas 1%. O motivo desse comportamento é que o compilador não pode executar otimizações diferentes daquelas que foram executadas. Esse nível apresentou uma queda maior na classificação ONA de 69%, devido a um maior número de instruções de controle, 1,4 vezes mais instruções que em `-O0`. Finalmente, o `-Os` apresentaram as mesmas características de outros níveis. Uma redução nas falhas de ONA para $13,7\%$ e falhas *Vanished*, enquanto ocorre um aumento de OMM, Hang e UT. A Tabela 5.9 mostra que a aplicação tem uma dependência maior de laços de repetições pois não foram otimizadas nenhuma instrução Salto em nenhum nível.

Tabela 5.9: Classificação de Instruções da Aplicação FDCT

Aplicação	-Olevel									
fdct	O0	%	O1	%	O2	%	O3	%	Os	%
Total de Instruções	449904	**100**	147201	**100**	164303	**100**	164303	**100**	157299	**100**
Leitura	189562	**42,1**	15462	**10,5**	27463	**16,71**	27463	**16,71**	22562	**14,**
Escrita	80951	**17,9**	14051	**9,54**	18552	**11,29**	18552	**11,29**	17651	**11,**
Saltos	241	**0,05**	241	**0,16**	241	**0,14**	241	**0,14**	241	**0,1**
Ramificações	3659	**0,81**	958	**0,65**	1758	**1,06**	1758	**1,06**	1758	**1,1**
Operações de ULA	175491	**39**	116489	**79**	116289	**70,7**	116289	**70,7**	115087	**73,**

No geral, outros níveis de otimização além de `-O0` apresentaram uma redução no número total de instruções e no número de operações de memória. No entanto, houve um aumento percentual no número de operações de Controle e ULA para todos os casos. A redução da utilização da memória combinada com mais instruções de controle cria uma aplicação mais complexa, entretanto mais resiliente. O nível `-O3` representa os piores resultados de toda otimização e seus piores casos ocorrem quando o compilador executa uma otimização mais alta nas instruções de Saltos. Esse comportamento foi ilustrado com a aplicação mMult, mas um comportamento semelhante ocorre para 63% de todas as aplicações, que são, coincidentemente, abaixo da linha de base na Figura 5.8(c), ou seja, resultados piores que `-O0`. O `-O3` é o nível que gerou a redução mais considerável no número de instruções, mostrando que o número de instruções executadas não está relacionado à resiliência do aplicativo. De todos os níveis de otimização, o `Os` é a *flag* que mais impactou o desaparecimento de falhas, considerando um conjunto genérico de aplicações, conforme apresentado neste estudo. `-Os` melhorou a resiliência a *soft error* para 75% do conjunto de aplicações. A razão para o melhor desempenho é devido a um aumento maior nas instruções de controle em comparação com as outras instruções. Além disso, em geral, as aplicações mostraram um número mais significativo de instruções de controle, e isso tem dois efeitos, aumentando a probabilidade de falha na propagação e, consequentemente, aumentando o número de OMM. Por outro lado, com um número mais significativo de instruções de controle, ocorre uma diminuição na ONA. Essa avaliação sugere que as instruções de controle têm uma influência maior nas falhas de memória (OMM) do que o número de operações de memória em si. O que aumenta a probabilidade da aplicações permanecer em laços de repetição e o UT está relacionado ao uso da memória e às instruções de controle. Além disso, a combinação destes pode ocorrer um salto para uma posição reservada da memória e após a leitura ou escrita, causando uma falha.

Tabela 5.10: Configuração dos Experimentos de Propagação de Falhas

Número de Tarefas por EP	2
Dimensão da NoC	2x2
Otimização do Compilador	`-Os`
Número de Aplicações	3
Número de Cenários	12
Injeções de Falha por Cenário	1000

5.3 Propagação de Falhas

Uma vez definido qual a *flag* de otimização obteve uma melhoria em um maior conjunto de aplicações, neste caso `-Os`, partindo desta afirmação, esta *flag* foi adotada como a padrão para o desenvolvimento dos próximos experimentos.

Como visto anteriormente as otimizações do compilador influenciam diretamente na resiliência dos sistemas. Partindo desta afirmação, é necessário avaliar a propagação de falhas em sistemas multiprocessados, uma vez que esses sistemas possuem um processo de depuração difícil, muitas vezes sendo impossível identificar se uma falha injetada em um elemento propagou ou não. Estes fatores, mais a pouca literatura referente a propagação de falhas em sistemas multiprocessados foram as principais motivações para a realização deste conjunto de experimentos.

O primeiro conjunto de experimentos realizados, constituiu-se de uma rede intra-chip de tamanho $2x2$, compilada com a *flag* `-Os` e 3 distintas aplicações, a Tabela 5.10 apresenta a configuração destes experimentos.

A Figura 5.11 ilustra os resultados obtidos da campanha de injeção da aplicação MPEG. As Figuras 5.12 e 5.13 apresentam a porcentagem de falhas propagadas em cada PE para à memória e o banco de registradores respectivamente. Analisando os resultados podemos observar que a porcentagem de falhas propagadas chega a quase 50% no PE2, entretanto esse valor corresponde a diversas falhas injetada neste PE que acabaram se propagando para os demais. A Tabela 5.11 apresenta o número de falhas, onde cada coluna significa o PE no qual a falha foi injetada, e pode ser observado que uma mesma falha acaba propagando para mais de um PE ao mesmo tempo. Contudo, esse comportamento não persiste na classificação de falhas propagadas no banco de registradores. A Tabela 5.12 descreve as falhas propagadas para o banco de registradores. Uma característica é que a mesma falha pode ser propagada para ambos elementos - memória e registradores, a análise foi realizada avaliando individualmente cada simulação e categorizando quais elementos foram propagados.

Figura 5.11: Classificação de Falhas da Aplicação MPEG Compilada com a *Flag* `-Os`

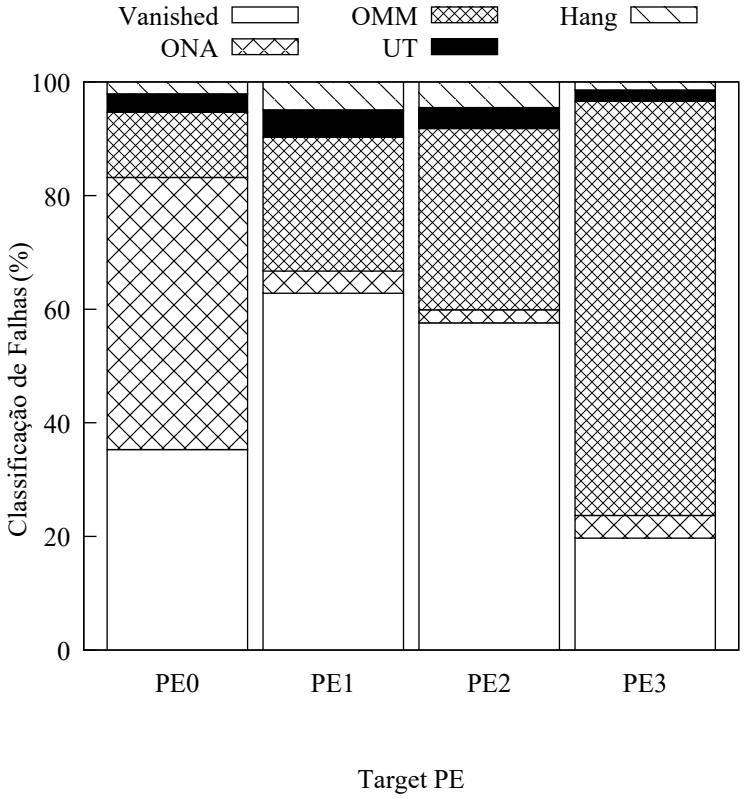


Figura 5.12: Falhas Propagadas para a Memória no Cenário com a Aplicação MPEG.

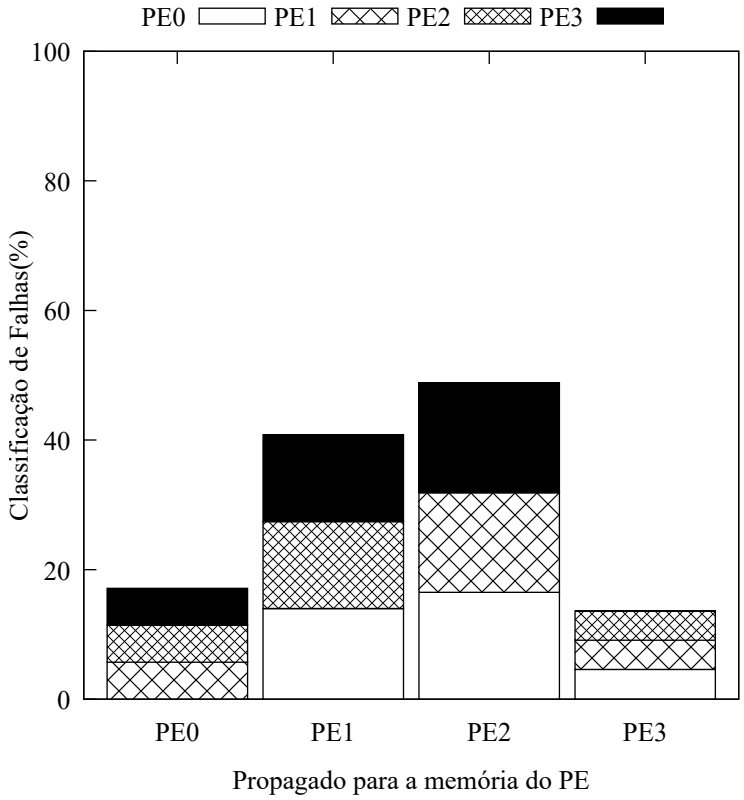


Tabela 5.11: Número de Falhas Propagadas para a Memória

	PE0	PE1	PE2	PE3
PE0	-	144	165	46
PE1	57	-	154	45
PE2	57	134	-	45
PE3	57	134	169	-
Número de Falhas Propagadas	57	144	169	46

Figura 5.13: Falhas Propagadas para o Banco de Registradores no Cenário com a Aplicação MPEG.

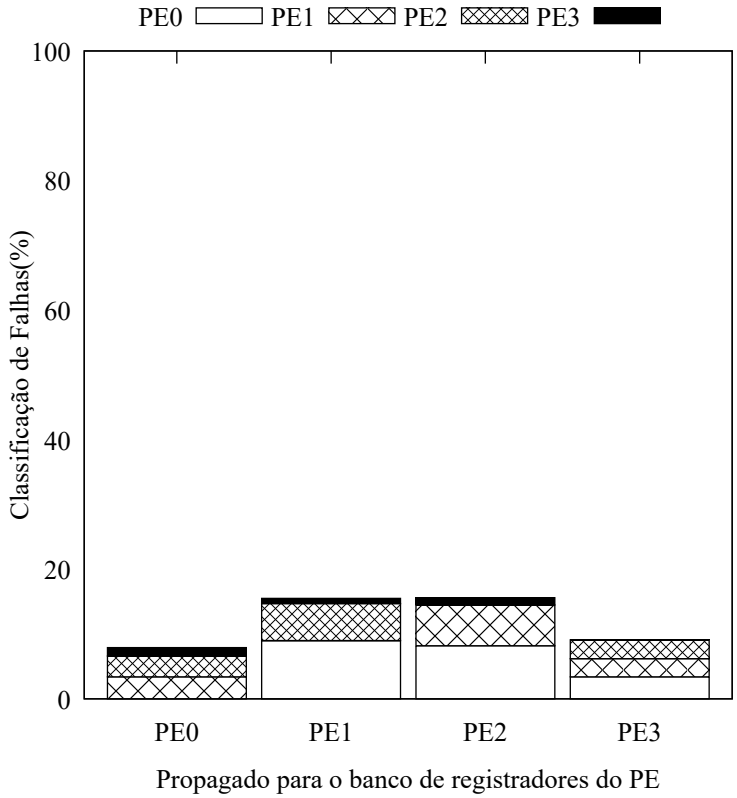


Tabela 5.12: Número de Falhas Propagadas para o Banco de Registradores

	PE0	PE1	PE2	PE3
PE0	-	97	82	34
PE1	34	-	63	28
PE2	32	57	-	29
PE3	13	8	11	-
Número de Falhas Propagadas	57	144	169	46

Figura 5.14: Classificação de Falhas da Aplicação DTW Compilada com a *Flag* `-Os`

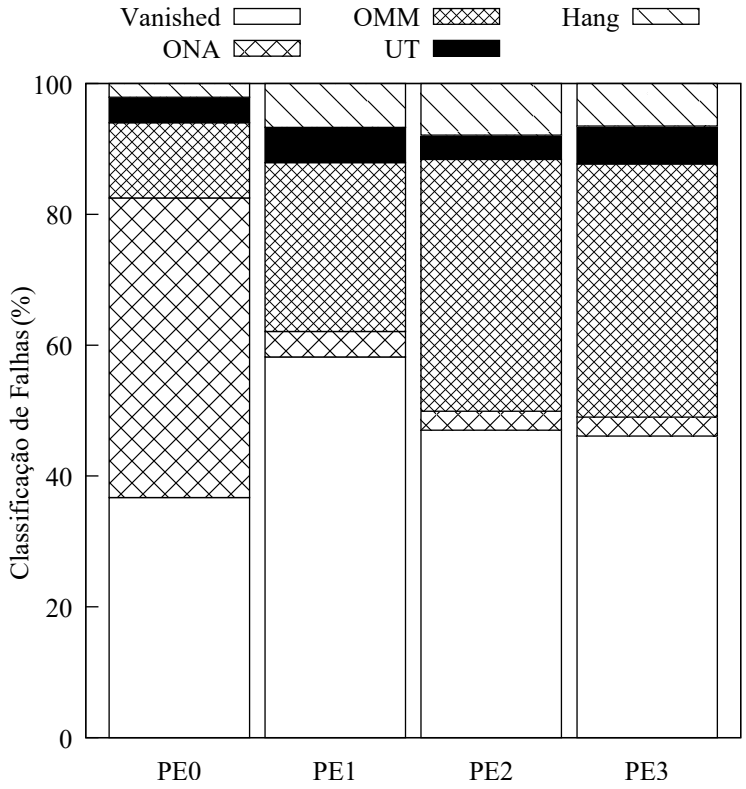


Tabela 5.13: Número de Falhas Propagadas para a Memória

	PE0	PE1	PE2	PE3
PE0	-	144	208	203
PE1	66	-	208	170
PE2	66	137	-	203
PE3	66	138	170	-
Número de Falhas Propagadas	66	144	208	203

O último *testcase* avaliado, foi o da aplicação DTW, que apresenta um maior volume de comunicação, comparado com as demais. A Figura 5.14 apresenta os resultados da classificação de falhas. Expandindo a análise da propagação de falhas, a porcentagem de falhas propagadas para a memória chega a quase 60%. Por possuir uma comunicação entre diversas tarefas, a aplicação está mais suscetível a falhas, a Tabela 5.13 ilustra os valores das falhas, onde observa-se o comportamento de uma mesma falha propagando para os demais PEs.

Em linhas gerais o PE mestre, PE0, tem a menor taxa de propagação de falhas, isso ocorre, devido a esse PE ser o mestre da arquitetura e executa somente tarefas de gerenciamento, enviando pacotes de controle aos demais PEs. Outro atributo observado, é referente à comunicação de tarefas de uma aplicação. Conforme, a aplicação possuir

Figura 5.15: Falhas Propagadas para a Memória no Cenário com a Aplicação DTW.

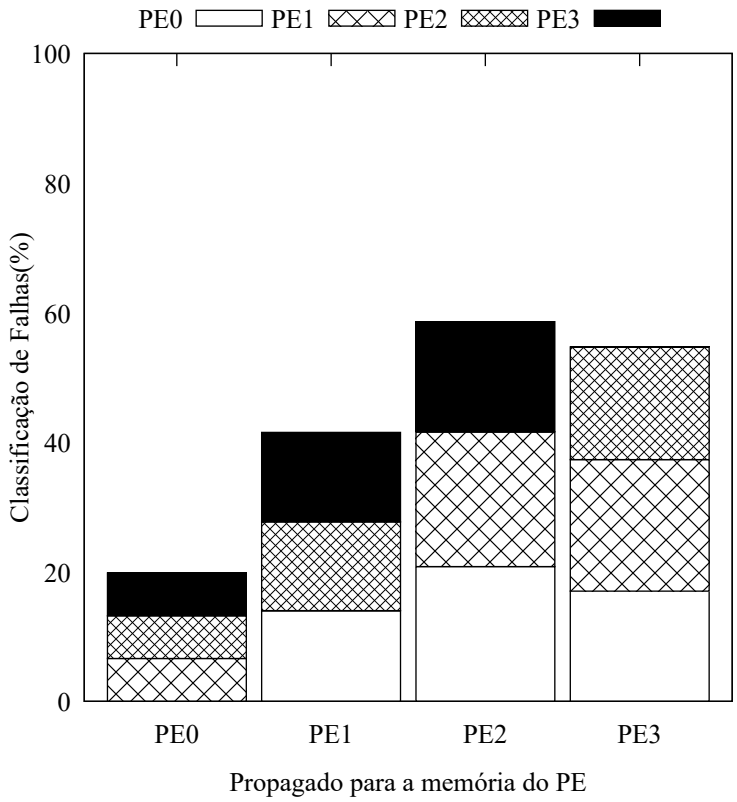


Figura 5.16: Falhas Propagadas para o Banco de Registradores no Cenário com a Aplicação DTW.

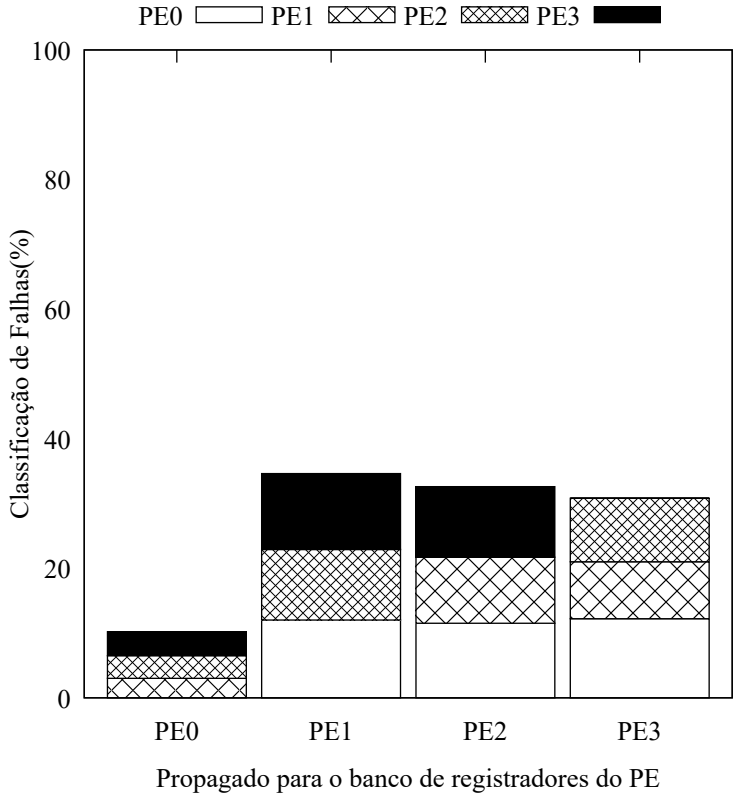


Tabela 5.14: Número de Falhas Propagadas para o Banco de Registradores

	PE0	PE1	PE2	PE3
PE0	-	120	115	122
PE1	30	-	102	88
PE2	35	109	-	98
PE3	37	117	109	-
Número de Falhas Propagadas	66	144	208	203

um maior número de comunicações, maior é a possibilidade da falha propagar pela rede. Contudo, técnicas podem ser desenvolvidas para mitigar essa propagação, como avaliar algoritmos de mapeamento de tarefas em sistemas multiprocessados, para que tarefas que dependam uma da outra, não necessitem acessar PEs distintos.

5.4 Resumo do Capítulo

Este Capítulo analisou os experimentos realizados. A primeira seção conduziu uma verificação sobre a precisão dos distintos modelos em RTL com TLM, realizados em sistemas *single* e multiprocessados. Identificando que o modelo TLM descrito em SystemC possui uma precisão aceitável com os ganhos adquiridos em tempo de simulação. Posteriormente foi realizado um conjunto de experimentos visando identificar o impacto das *flags* de compilação em *soft error*, para por fim apontar uma *flag* que apresenta uma maior resiliência. O último conjunto de experimento foi a avaliação da propagação de falha em sistemas multiprocessados.

6 CONCLUSÕES

Uma das contribuições deste trabalho foi desenvolver o suporte à injeção de falhas em sistemas multiprocessados que utilizam NoC como infraestrutura de comunicação modelados em SystemC. Realização de uma avaliação de desempenho e precisão entre os modelos RTL e TLM. Esta avaliação resultou que TLM aumenta o desempenho de tempo de simulação em $4.5x$ e como desvantagem teve uma divergência média de precisão no pior caso de apenas 6.78%. Na maioria dos casos a modelagem em SystemC difere em falhas relacionadas à memória dos modelos. A discrepância é considerada aceitável considerando as vantagens que o modelo possui, entre elas a facilidade de depurar problemas, visto que podem ser utilizadas tais ferramentas desenvolvidas para C++. Outro fator importante é a capacidade da elaboração de sistemas híbridos - *hardware* e *software* com a mesma linguagem.

Outro tópico pesquisado nesta dissertação foram os efeitos de várias *flags* de otimização do compilador na confiabilidade de *soft error*. Os resultados mostram que em 75% dos casos o uso da *flag* `-Os` aumenta a resiliência a *soft error* das aplicações em $3x$ em comparação ao nível `-O0`. Os resultados também demonstram que a ocorrência de falhas está diretamente associada à dependência das instruções com o uso de registradores-*flags*, bem como à porcentagem de instruções de controle e uma melhor utilização dos registros. Por outro lado, nenhuma relação pôde ser estabelecida entre o tempo de execução de uma aplicação e a classificação de falha.

Por fim foi realizada uma avaliação da propagação de falhas em sistemas multiprocessados utilizando `-Os` como base de compilação das aplicações, com o objetivo de avaliar o sistema. Uma vez as aplicações já possuindo maior resiliência, o quão suscetível a falhas ele permanece. Os resultados mostram que praticamente 60% das falhas são propagadas para outros PEs. Em diversos casos essa propagação ocorre para múltiplos PEs. Outro fator identificado, é que o aumento de comunicação entre tarefas afeta a quantidade de falhas propagadas. A comunicação afeta no aumento de falhas propagadas na memória como também no banco de registradores. A importância de avaliar propagação de falhas para identificar os riscos nos estágios iniciais do projeto e realizar possíveis alterações, como alterar mapeamento de tarefa, algoritmo de comunicação da rede ou desenvolver técnicas para tornar o elemento menos suscetível a falhas são algumas das opções que visam mitigar o impacto das falhas no sistema.

Embora a simulação em RTL ofereça uma alta precisão, o tempo necessário para

concluir uma campanha de injeção de falhas é impraticável para plataformas de larga escala. Nessa direção, trabalhos futuros incluem o uso do melhor desempenho de simulação e recursos de depuração do SystemC para investigar a vulnerabilidade de *soft error* de plataformas de multiprocessadores em larga escala.

6.1 Trabalhos Futuros

Este trabalho abre diversos tópicos para futuras pesquisas. Os tópicos mais diretos são o desenvolvimento de métricas para identificar se a ocorrência de falhas transcorre na memória ou no banco de registrados. Avaliação da relação da propagação de falhas com aumento de carga de trabalho no sistema Outra possibilidade é verificar se o mapeamento de tarefas ou o algoritmo de roteamento da rede intra-chip afeta a resiliência destes sistemas. Como foi apresentado um estudo de caso da ocorrência de múltiplas falhas no sistema, este é outro tópico que pode ser explorado.

REFERÊNCIAS

AGUIAR, V. et al. Experimental setup for single event effects at the são paulo 8ud pelletron accelerator. **Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms**, Elsevier, v. 332, p. 397–400, Ago 2014.

ALAM, M. A. et al. A comprehensive model for pmos nbti degradation: Recent progress. **Microelectronics Reliability**, Elsevier, v. 47, n. 6, p. 853–862, Jun 2007.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, IEEE, v. 1, n. 1, p. 11–33, Jan 2004. ISSN 2160-9209.

BARAZA, J. C. et al. A prototype of a vhdl-based fault injection tool. : **Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems**. Yamanashi, Japão: , 2000. p. 396–404. ISSN 1550-5774.

BAUMANN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. **IEEE Transactions on Device and Materials Reliability**, IEEE, v. 5, n. 3, p. 305–316, Set. 2005. ISSN 1558-2574.

BELTRAME, G.; BOLCHINI, C.; MIELE, A. Multi-level fault modeling for transaction-level specifications. : **Proceedings of the 19th ACM Great Lakes symposium on VLSI**. Boston, EUA: , 2009. p. 87–92.

BELTRAME, G.; FOSSATI, L. Resp: a design and validation tool for data systems. : **DASIA 2008-Data Systems In Aerospace**. Noordijk, Holanda: , 2008. v. 665.

BENINI, L.; DE MICHELI, G. Networks on chips: a new soc paradigm. **IEEE Computer**, IEEE, v. 35, n. 1, p. 70–78, Jan 2002.

BLINKERT, N. et al. The gem5 simulator. **ACM SIGARCH computer architecture news**, ACM New York, NY, USA, v. 39, n. 2, p. 1–7, Ago 2011.

BORKAR, S. et al. Parameter variations and impact on circuits and microarchitecture. : **Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)**. Anaheim, EUA: , 2003. p. 338–342.

BORTOLON, F. T. et al. Exploring the impact of soft errors on noc-based multiprocessor systems. : **2018 IEEE International Symposium on Circuits and Systems (ISCAS)**. Florência, Italia: , 2018. p. 1–5. ISSN 2379-447X.

CARARA, E. A. et al. Hemps - a framework for noc-based mpsoc generation. : **2009 IEEE International Symposium on Circuits and Systems**. Taipei, Taiwan: , 2009. p. 1345–1348. ISSN 2158-1525.

CHO, H. et al. Quantitative evaluation of soft error injection techniques for robust system design. : **2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)**. Austin, EUA: , 2013. p. 1–10. ISSN 0738-100X.

CUMMISKEY, P.; JAYANT, N. S.; FLANAGAN, J. L. Adaptive quantization in differential pcm coding of speech. **The Bell System Technical Journal**, Nokia Bell Labs, v. 52, n. 7, p. 1105–1118, Set. 1973. ISSN 0005-8580.

de AGUIAR GEISSLER et al., F. Soft error injection methodology based on qemu software platform. : **2014 15th Latin American Test Workshop - LATW**. Fortaleza, Brasil: , 2014. p. 1–5. ISSN 2373-0862.

DEMERTZI et al., M. Analyzing the effects of compiler optimizations on application reliability. : **2011 IEEE International Symposium on Workload Characterization (IISWC)**. Austin, EUA: , 2011. p. 184–193.

EBNENASIR, A.; HAJISHEYKHI, R.; KULKARNI, S. S. Facilitating the design of fault tolerance in transaction level systemc programs. **Theoretical Computer Science**, Elsevier, v. 496, p. 50–68, Jul 2013.

EICHENBERGER, A. E. et al. Using advanced compiler technology to exploit the performance of the cell broadband engine™ architecture. **IBM Systems Journal**, IBM, v. 45, n. 1, p. 59–84, Jan 2006.

ERICKSON, J. Tlm-driven design and verification–time for a methodology shift. **Cadence Design Systems, Inc**, Cadence, Jan 2016.

FENG, S. et al. Shoestring: probabilistic soft error reliability on the cheap. **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 38, n. 1, p. 385–396, Mar 2010.

FLENKER, T. et al. Towards making fault injection on abstract models a more accurate tool for predicting rt-level effects. : **2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. Bochum, Alemanha: , 2017. p. 533–538. ISSN 2159-3477.

GAVA, J. et al. Evaluation of compilers effects on openmp soft error resiliency. : **2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. Miami, EUA: , 2019. p. 259–264. ISSN 2159-3469.

GNU. **GNU Compiler Collection Available**. 2006. Disponível na Internet: <https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Optimize-Options.html>.

GONÇALVES de OLIVEIRA, D. A. G. et al. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. **IEEE Transactions on Computers**, v. 65, n. 3, p. IEEE, Jun 2016.

GOUGH, B.; STALMMAN, M. An introduction to gcc for the gnu compilers gcc and g++. Citeseer, Março 2004.

GRANLUND, T.; GRANBOM, B.; OLSSON, N. Soft error rate increase for new generations of srams. **IEEE Transactions on Nuclear Science**, IEEE, v. 50, n. 6, p. 2065–2068, Dez. 2003. ISSN 1558-1578.

IBE, E. H. **Terrestrial radiation effects in ULSI devices and electronic systems**. : John Wiley & Sons, 2015.

JIA, C.; CHAN, W. K. A study on the efficiency aspect of data race detection: A compiler optimization level perspective. : **2013 13th International Conference on Quality Software**. Najing, China: , 2013. p. 35–44. ISSN 2332-662X.

JOHANSSON, K. et al. Neutron induced single-word multiple-bit upset in sram. **IEEE Transactions on Nuclear Science**, IEEE, v. 46, n. 6, p. 1427–1433, Dez 1999. ISSN 1558-1578.

KALIORAKIS, M. et al. Differential fault injection on microarchitectural simulators. : **2015 IEEE International Symposium on Workload Characterization**. Atlanta, EUA: , 2015. p. 172–182.

KARNIK, T.; HAZUCHA, P. Characterization of soft errors caused by single event upsets in cmos processes. **IEEE Transactions on Dependable and Secure Computing**, IEEE, v. 1, n. 2, p. 128–143, Abril 2004. ISSN 2160-9209.

KOOLI, M.; DI NATALE, G. A survey on simulation-based fault injection tools for complex systems. : **2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)**. Santorini, Grécia: , 2014. p. 1–6.

KUEING-LONG CHEN et al. Reliability effects on mos transistors due to hot-carrier injection. **IEEE Transactions on Electron Devices**, IEEE, v. 32, n. 2, p. 386–393, Feb 1985. ISSN 1557-9646.

LEVEUGLE, R. et al. Statistical fault injection: Quantified error and confidence. : **2009 Design, Automation Test in Europe Conference Exhibition**. Nice, França: , 2009. p. 502–506. ISSN 1558-1101.

LINS, F. M. et al. Register file criticality and compiler optimization effects on embedded microprocessor reliability. **IEEE Transactions on Nuclear Science**, IEEE, v. 64, n. 8, p. 2179–2187, 2017.

MACHADO, R. S. et al. Comparing performance of c compilers optimizations on different multicore architectures. : **2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)**. Campinas, Brasil: , 2017. p. 25–30.

MANSOUR, W.; VELAZCO, R. An automated seu fault-injection method and tool for hdl-based designs. **IEEE Transactions on Nuclear Science**, IEEE, v. 60, n. 4, p. 2728–2733, Aug 2013. ISSN 1558-1578.

MUKHERJEE, S. S. et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. : **Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.** San Diego, EUA: , 2003. p. 29–40.

MUSHTAQ, H.; AL-ARS, Z.; BERTELS, K. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. : **2011 IEEE 6th International Design and Test Workshop (IDT)**. Beirute, Líbano: , 2011. p. 12–17. ISSN 2162-0601.

OVP. **Open Virtual Plataforms(OVP)**. 2020. Disponível na Internet: <http://www.ovpworld.org>.

PANDA, P. R.; DUTT, N. D.; NICOLAU, A. Memory data organization for improved cache performance in embedded processor applications. **ACM Trans. Des. Autom. Electron. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. 4, p. 384–409, Out 1997. ISSN 1084-4309.

PANDINI, D. Variability in advanced nanometer technologies: challenges and solutions. : SPRINGER. **International Workshop on Power and Timing Modeling, Optimization and Simulation**. Delft, Holanda, 2009. p. 2–2.

PO-KUAN HUANG; GHIASI, S. Power-aware compilation for embedded processors with dynamic voltage scaling and adaptive body biasing capabilities. : **Proceedings of the Design Automation Test in Europe Conference**. Munique, Alemanha: , 2006. v. 1, p. 1–2. ISSN 1558-1101.

QEMU. **QEMU**. 2020. Disponível na Internet: <https://www.qemu.org/>.

RIGO, S. et al. Archc: A systemc-based architecture description language. : IEEE. **16th Symposium on Computer Architecture and High Performance Computing**. Foz do Iguaçu, Brasil, 2004. p. 66–73.

ROSA, F. et al. A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability. : **2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)**. Amherst, EUA: , 2015. p. 211–214. ISSN 2377-7966.

ROSA, F. et al. Evaluation of multicore systems soft error reliability using virtual platforms. : **2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)**. Estrasburgo, França: , 2017. p. 85–88.

RUANO, O. et al. A simulation platform for the study of soft errors on signal processing circuits through software fault injection. : **2007 IEEE International Symposium on Industrial Electronics**. Vigo, Espanha: , 2007. p. 3316–3321. ISSN 2163-5145.

SANGCHOOLIE, B. et al. A study of the impact of bit-flip errors on programs compiled with different optimization levels. : **2014 Tenth European Dependable Computing Conference**. Newcastle, Inglaterra: , 2014. p. 146–157.

SANTINI, T. et al. Reliability analysis of operating systems for embedded soc. : **2015 15th European Conference on Radiation and Its Effects on Components and Systems (RADECS)**. Moscou, Russia: , 2015. p. 1–5.

SEIFERT, N.; OUTROS. Radiation-induced soft errors: A chip-level modeling perspective. **Foundations and Trends® in Electronic Design Automation**, Now Publishers, Inc., v. 4, n. 2–3, p. 99–221, Nov 2010.

SERRANO-CASES, A. et al. Nonintrusive automatic compiler-guided reliability improvement of embedded applications under proton irradiation. **IEEE Transactions on Nuclear Science**, IEEE, v. 66, n. 7, p. 1500–1509, Abril 2019.

SHAFIK, R. A.; AL-HASHIMI, P. R. e B. M. Systemc-based minimum intrusive fault injection technique with improved fault representation. : **2008 14th IEEE International On-Line Testing Symposium**. Rhodes, Grécia: , 2008. p. 99–104. ISSN 1942-9401.

SLAYMAN, C. Soft errors — past history and recent discoveries. : **2010 IEEE International Integrated Reliability Workshop Final Report**. Fallen Leaf, EUA: , 2010. p. 25–30. ISSN 1930-8841.

SNIR, M. et al. Addressing failures in exascale computing. **The International Journal of High Performance Computing Applications**, SAGE, v. 28, n. 2, p. 129–173, Mar 2014.

SOLINAS, M. et al. Preliminary results of netfi-2: An automatic method for fault injection on hdl-based designs. : **2017 18th IEEE Latin American Test Symposium (LATS)**. Bogota, Colombia: , 2017. p. 1–4. ISSN null.

SONG, L. et al. Comp: Compiler optimizations for manycore processors. : **2014 47th Annual IEEE/ACM International Symposium on Microarchitecture**. Cambridge, Inglaterra: , 2014. p. 659–671. ISSN 2379-3155.

VELAZCO, R.; MCMORROW, D.; ESTELA, J. **Radiation Effects on Integrated Circuits and Systems for Space Applications**. : Springer, 2019.

YAN, J.; ZHANG, W. Compiler-guided register reliability improvement against soft errors. : ACM. **Proceedings of the 5th ACM international conference on Embedded software**. Nova Iorque, EUA, 2005. p. 203–209.