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

FILIPE DE AGUIAR GEISSLER

METODOLOGIA DE INJEÇÃO DE FALHAS BASEADA EM EMULAÇÃO DE PROCESSADORES

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

Prof. Dra. Fernanda Lima Kastensmidt
Orientadora

Porto Alegre, agosto de 2014

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

FILIPE DE AGUIAR GEISSLER

Metodologia de injeção de falhas baseada em emulação de processadores

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

Prof. Dra. Fernanda Lima Kastensmidt
Orientadora

Porto Alegre, agosto de 2014

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Geissler, Filipe de Aguiar

Metodologia de Injeção de Falhas Baseada em Emulação de Processadores / Filipe de Aguiar Geissler. – Porto Alegre: Programa de Pós-Graduação em Microeletrônica, 2014.

85 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica. Porto Alegre, BR – RS, 2014. Orientador: Fernanda Gusmão de Lima Kastensmidt.

1. Efeitos da Radiação em Processadores. 2. Metodologias e ferramentas de injeção de falhas. 3. Emulação de processadores. I. Kastensmidt, Fernanda Gusmão de Lima. II. Metodologia de injeção de falhas baseada em emulação de processadores.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PGMICRO: Prof. Ricardo Reis
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Dedicatória

Dedico este trabalho com muito amor à minha mãe Suzana Pas de Aguiar, à minha tia Sandra Pas de Aguiar, em memória à minha madrinha Honorina Pas de Aguiar, e especialmente em memória à minha avó Maria Pas de Aguiar.

Agradecimentos

Quero agradecer a força do amor da minha vida, minha noiva, Ariane Moraes, por ter estado ao meu lado me incentivando, alegrando e sendo compreensiva em todos os momentos e por me ajudar a revisar a dissertação. A minha mãe Suzana, tia Sandra e avó Maria pelo amor e dedicação. A minha tia Lourdes que mesmo estando longe nunca deixa de estar presente. Ao meu primo Daniel por inúmeras horas de descontração em sua companhia e amizade. A toda a minha família e amigos pela união nas horas de dificuldade.

Um agradecimento a empresa PARKS S. A., pelo incentivo e flexibilidade para a formação de profissionais cada vez mais capacitados. Aos colegas e amigos do setor de pesquisa e desenvolvimento, com os quais convivo e aprendo diariamente. Um agradecimento ao meu coordenador Leandro Mondin, pelo apoio para que este mestrado fosse concluído e ao colega Alexandre Martins pelas sugestões durante o desenvolvimento deste trabalho. E finalmente, gostaria de agradecer em especial à minha orientadora Fernanda Kastensmidt, pela riqueza de ideias, paciência, flexibilidade e compreensão durante a execução deste trabalho.

Fault Injection Methodology Based on Processor Emulation

ABSTRACT

This dissertation aims to present a fault injection methodology based on microprocessor emulation. The effects caused by radiation in microprocessors, operating in space or at high altitudes, have been studied in the literature for the development of fault tolerance mechanisms. With the growing popularity of COTS (Commercial Off-The-Shelf) processors usage, in critical applications, a number of concerns have arisen due to the lack of reliability, presented in these systems. Due to the lack of fault tolerance mechanisms, these COTS devices are more susceptible to radiation effects. In this context, software-based fault tolerance techniques have been studied in the literature in order to increase the reliability of this approach. To validate such fault tolerance mechanisms, the use of fault injection techniques is applicable. These fault injection techniques have several limitations which can preclude their applicability, depending on of its design approach. Factor such as cost, unavailability of hardware description – used by fault injection techniques based on simulation or emulation with FPGA (*Field Programmable Gate Array*), and the long time demanded to execute experiments, are some examples of limitations in the available techniques. Based on this, the alternative fault injection methodology presented in this work aims to reduce these limitations. Based on the dynamic translation of instructions usage to accelerate the execution of application on emulators, the methodology presents a fault model for transient and permanent faults applicable in this scenario. As a classification method of the observed effects in this process, a model in the literature has been used. To validate this methodology, a fault injector based on the QEMU emulator was implemented. Later, a case study with the fault injector was performed for three software structures running at a time on a MIPS 24kc processor, representing three different levels of complexity: Linux operating system, RTEMS (Real-Time Operating System), and a dedicated application. Each system was submitted to a fault injection campaign emulating Single Event Upsets (SEUs). As fault targets it was selected the processor registers and the data memory. Finally, the analysis obtained with the experiments showed the different effects observed for the three levels of complexity. Besides that, the fault injector performance could be evaluated providing in the end a tool to help in the development of software-based fault injection techniques.

Keywords: Radiation effects in microprocessors, Fault Injection Methodology, Processor Emulation, Software-based fault tolerance techniques.

Metodologia de Injeção de falhas baseada em Emulação de Processadores

RESUMO

Esta dissertação tem por finalidade apresentar uma metodologia de injeção de falhas baseada em emulação de processadores. Os efeitos causados pela radiação em processadores, operando no espaço ou em altitudes elevadas, têm sido estudados na literatura para o desenvolvimento de mecanismos de tolerância a falhas. Com a crescente popularidade do uso de processadores comerciais, (COTS – do inglês, *Commercial Off-The-Shelf*), em aplicações críticas, uma série de preocupações tem surgido devido a falta de confiabilidade apresentada por estes sistemas. Sendo desprovidos de mecanismos de tolerância para melhor robustez em ambientes espaciais, estes dispositivos comerciais são mais suscetíveis aos efeitos da radiação. Neste contexto, técnicas de tolerância a falhas baseadas em software vêm sendo estudadas a fim de aumentar a confiabilidade desta abordagem. Para a devida validação de tais mecanismos de tolerância, o uso de técnicas de injeção de falhas é aplicável. Estas técnicas de injeção de falhas possuem uma série de limitações que podem inviabilizar a sua aplicabilidade, dependendo da abordagem utilizada. Fatores como custo, indisponibilidade da descrição de hardware – utilizada em técnicas de injeção de falhas por simulação ou emulação em FPGA (*Field Programmable Gate Array*), e o longo tempo necessário para execução dos experimentos, são alguns exemplos de limitações das técnicas disponíveis. Com base nisso, a metodologia de injeção de falhas alternativa apresentada neste trabalho, visa reduzir as limitações presentes nas mais diversas técnicas. Baseada na utilização de tradução dinâmica de instruções, para acelerar o processo de execução de aplicações em emuladores, a metodologia apresenta um modelo de falhas para efeitos transientes e permanentes, aplicáveis neste cenário. Como método de classificação dos efeitos observados neste processo, um modelo presente na literatura foi utilizado. Para validação desta metodologia, um injetor de falhas baseado no emulador QEMU foi desenvolvido. Posteriormente, um estudo de caso com o injetor de falhas foi realizado para três estruturas de software distintas executando individualmente no processador MIPS 24kc, representando três níveis de complexidade distintos: sistema operacional Linux, sistema de tempo real, (RTEMS – do inglês, *Real-Time Operating System*), e uma aplicação dedicada. Cada sistema foi submetido a uma campanha de injeção de falhas transientes para emulação de efeitos singulares (SEU – do inglês, *Single Event Upset*). Como alvo de falhas, foram selecionados os registradores do processador e a memória de dados. Por fim, as análises obtidas através dos experimentos mostraram os diferentes efeitos observados para os três níveis de complexidade dos softwares executados. Além disso, se pôde avaliar o desempenho do injetor de falhas, disponibilizando ao final do trabalho uma ferramenta para o auxílio no desenvolvimento de técnicas de tolerância a falhas por software.

Palavras-Chave: Efeitos da radiação em processadores, Metodologia de injeção de falhas, Emulação de processadores, Tolerância a falhas em software.

LISTA DE ABREVIATURAS E SIGLAS

AMUSE	Autonomous MultiLevel Emulation System of Soft Error Evaluation
ASIC	Application-specific Integrated Circuit
BDM	Background Debug Mode
CEU	Code Emulating Upset
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CTR	Compile-Time Reconfiguration
DFI	Direct Fault Injection
EDM	Error Detection Mechanism
EMM	Experimental Module Manager
EXFI	Exception-based Fault Injector
FERRARI	Flexible Software-based Fault and Error Injection System
FI	Fault Injector
FIHU	Fault Injection Hardware Unit
FPGA	Field Programmable Gate Array
GAS	GNU AS
GIT	Gerenciamento de Injeção de Falhas
GL	Gate Level
GNU	General Public License
HDL	Hardware Description Level
I2C	Inter-integrated Circuit
IP	Intellectual Property
JTAG3	Joing Tag Access Group 3
LET	Linear Energy Transfer
LLVM	Low Level Virtual Machine
MBU	Single-Event Multiple-Bit Upset
MEFISTO	Multi-level Error/Fault Injection Simulation Tool

MEFISTO-L	Multi-level Error/Fault Injection Simulation Tool
MMU	Memory Unit Management
OCD	Common on-Chip Debugging
OCD-FI	Common on-Chip Debugging Fault Injector
PC	Personal Computer
QEMU	Quick Emulator
RAM	Random Access Memory
RCT	Run Control & Trace
RI	Representação Intermediária
RT	Real-Time
RTEMS	Real-Time Operating System
RTL	Register-Transfer Level
RTR	Real-Time Configuration
SDF	Standard Delay Format
SEE	Single-Event Effects
SET	Single-Event Transient
SEU	Single-Event Upset
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SWIFI	Software-implemented Fault Injection
TB	Bloco de Tradução
VHDL	VHSIC Hardware Description Language

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O uso de processadores comerciais em aplicações críticas tem sido realidade no desenvolvimento de soluções tecnológicas. Denominados de *Commercial Off-The-Shelf* (COTS), esses processadores comerciais representam o estado da arte da indústria de semicondutores. Com alto nível de integração em virtude da baixa dimensão dos transistores, é possível alcançar altas taxas de frequência de operação com baixo consumo de potência. No entanto, tais tecnologias estão mais vulneráveis a falhas transientes, ou *soft errors*, causados pelos efeitos da radiação e interferências eletromagnéticas. Desprovidos de mecanismos de tolerância a falhas muito presentes em processadores para aplicações espaciais e militares, os dispositivos COTS ficam mais suscetíveis aos efeitos conhecidos como singulares, (SEEs – do inglês, *Single Event Effects*). Estes efeitos singulares são caracterizados pela interação de partículas energizadas com dispositivos semicondutores, produzindo comportamentos indesejados no sistema. Devido ao impacto de uma partícula em uma junção PN, localizada em um nó sensível de um circuito, através da transferência de energia da partícula para o nó, pares elétrons-lacuna podem ser gerados e uma corrente produzida. A interpretação desta corrente no circuito pode acarretar alterações no comportamento deste, e consequente efeito no software em execução (DODD, 2003).

Para viabilizar o uso de dispositivos COTS em aplicações críticas, o desenvolvimento de mecanismos de tolerância a falhas em software surge como uma abordagem eficiente e largamente empregada. Com foco em confiabilidade, estas aplicações integram na sua estrutura mecanismos de recuperação diante dos erros detectados, com o mínimo de impacto no desempenho e alteração do comportamento original da aplicação (RHOD, 2008). Contudo, para o processo de validação destes mecanismos de tolerância a falhas, é necessário submeter a aplicação em condições semelhantes a que ela será exposta em uma situação real. Neste cenário, o uso de técnicas de injeção de falhas é altamente empregado para validar a eficiência destas técnicas, permitindo assim uma melhor avaliação da solução proposta.

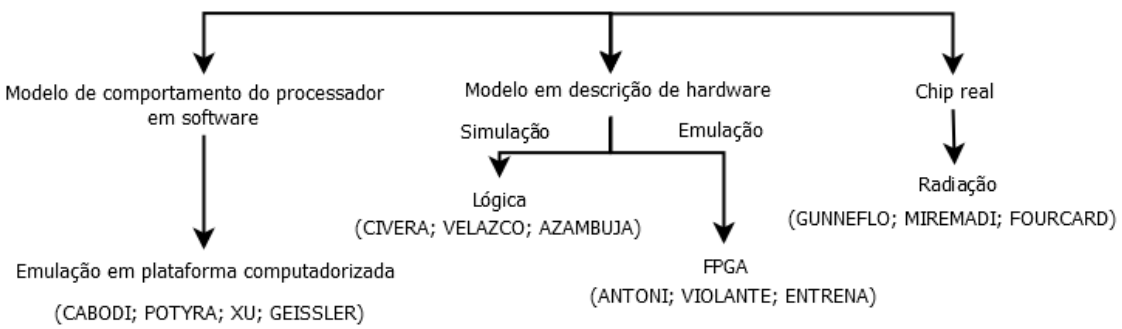
Injeção de falhas é uma técnica efetiva de análise para permitir um melhor entendimento do comportamento de um sistema quando falhas são introduzidas sob o sistema alvo (IYER, 1995). Dependendo da técnica de injeção de falhas utilizada para avaliação do sistema, é necessário construir uma infraestrutura para este fim, contemplando laboratórios com equipamentos dedicados, tais como aceleradores de partículas, lasers, etc. Outra abordagem muito comum é ter o modelo de comportamento de hardware do processador descrito em alto nível, normalmente nas linguagens VHDL (VHSIC *Hardware Description Language*) ou *Verilog*, para realização da injeção de falhas. Nesta situação, as falhas são injetadas por simulação, ou por emulação, sendo esta última com uso de prototipação em FGPA (*Field Programmable Gate Array*).

Para o caso de dispositivos COTS, o modelo de comportamento de hardware de um processador raramente é disponibilizado por parte de fabricantes, inviabilizando o uso de descrição de hardware no processo de injeção de falhas. Em virtude disso, se faz necessário o uso do hardware final do processador ou, para o caso de um hardware mais dedicado a uma aplicação, um ASIC (*Application Specific Integrated Circuit*), em conjunto com a utilização de técnicas de injeção de falhas que se utilizem de radiação. Em tal situação, o aumento de demanda em termos de custo e tempo de desenvolvimento tem grande impacto em projetos.

Uma alternativa a descrição de hardware é o uso do modelo de hardware descrito em software, utilizado em emuladores de arquiteturas, ou máquinas virtuais. Tais modelos são disponibilizados por parte de fabricantes de processadores para viabilizar aplicações com suporte a virtualização e também para permitir o desenvolvimento de software em fases iniciais de projeto, sem a presença do hardware final, ainda em processo de elaboração (CHANG, 2013; MARGINEAN, 2014). Com base nisso, em posse do código fonte destes modelos para emuladores, falhas podem ser injetadas durante a emulação.

Diante da variedade de técnicas de injeção de falhas disponíveis é possível classificar os métodos posicionando alguns autores que representam grupos de trabalho os quais utilizam determinado método. Na figura 1.1 são definidas basicamente as categorias com base no tipo de modelo do processador alvo de falhas. A primeira categoria representa emulação de modelos de hardware descritos em software, utilizados em emuladores. Cabe salientar que o termo emulação utilizado nesta dissertação, quando se referindo ao modelo de processador descrito em software, refere-se em recriar as funções originais de processador em outro, visando desempenho de execução. Neste caso, não há compromisso em reproduzir os estados internos do hardware real. Em contrapartida, o termo simulação envolve modelar fielmente ou tão próximo quanto possível os estados internos de um processador e arquitetura organizacional, permitindo neste caso a execução de um *pipeline*, por exemplo. No entanto, pode se encontrar na literatura o emprego do termo de forma errada, causando em muitos casos confusão. A segunda categoria contempla o uso de modelos de hardware para simulação lógica e emulação em FPGA. E finalmente o último grupo, aplicado quando se tem por finalidade uma análise mais realística, faz uso da injeção de falhas por hardware, com uso de radiação.

Figura 1.1: Categorização dos métodos de injeção de falhas em função do modelo de processador e posicionamento de alguns autores nestas categorias.



Este trabalho tem por finalidade apresentar uma metodologia de injeção de falhas baseada em emulação de processadores. O trabalho é direcionado a redução das limitações apresentadas por outras técnicas, visando o aumento do desempenho de execução de experimentos, bem como a estruturação de um método flexível e aplicável a qualquer arquitetura de processadores, incluindo processadores COTS. Além disso, o trabalho proposto contribui com melhorias em relação a publicações recentes na literatura, que fazem uso de emulação de processadores para injeção de falhas (XU, 2012; YI, 2013). Outra proposta feita no trabalho é sua aplicabilidade no desenvolvimento e análise de técnicas de tolerância para sistemas operacionais. Tal motivação tem origem com base em limitações apresentadas por certas técnicas de injeção de falhas, como é o caso da simulação de descrição de hardware, por exemplo. Devido ao alto tempo necessário para simulação de hardware e software complexos, seu uso se torna restrito. Neste contexto, a emulação de processadores pode ser uma alternativa para acelerar os experimentos de forma considerável.

Este trabalho esta organizado da seguinte maneira: o capítulo 2 apresenta uma breve introdução a falhas transientes oriundas da radiação, abordando também técnicas e metodologias de injeção de falhas em processadores e injetores de falhas aplicáveis neste contexto; o capítulo 3 apresenta plataformas de emulação de processadores e também conceitos utilizados na parte experimental do trabalho; o capítulo 4 apresenta uma proposta de metodologia de injeção de falhas com uso de emulação de processadores, juntamente com um ambiente de injeção de falhas desenvolvido com base na metodologia proposta, fazendo uso do emulador de processadores QEMU (BELLARD, 2005); o capítulo 5 apresenta os experimentos e resultados obtidos com o uso do ambiente de injeção de falhas, para um estudo de caso com a arquitetura MIPS; e para finalizar, o capítulo 6 apresenta as conclusões desta dissertação, publicações decorrentes deste trabalho e as possibilidades de trabalhos futuros.

2 INJEÇÃO DE FALHAS EM PROCESSADORES

Os efeitos observados em processadores devido à interação com radiação têm sido preocupantes tanto para o desenvolvimento de software para aplicações espaciais quanto para aplicações em nível terrestre. Os *soft errors* surgem como um resultado destas interações, ocasionando alterações nos elementos de memória utilizados por estas aplicações executadas nestes processadores. Tendo em vista que processadores COTS não possuem foco em tolerância para aplicabilidade em ambientes críticos, o desenvolvimento de software tolerante é uma abordagem para uso neste contexto. Com base nisso, uma variedade de técnicas e ferramentas para injeção de falhas têm sido desenvolvida na literatura para validação destes mecanismos de tolerância.

2.1 Falhas transientes causadas pela radiação em dispositivos semicondutores

Efeitos causados pela radiação em processadores podem ser classificados como transientes ou permanentes. Os efeitos transientes não causam quaisquer danos ao circuito em teste, produzindo somente alterações no comportamento e nos valores de elementos de memória utilizados por aplicações. Por outro lado, os efeitos permanentes comprometem o processador em parte ou todo ele, impossibilitando em muitos casos a sua utilização.

2.1.1 Injeção de Falhas

As técnicas de injeção de falhas buscam modelar os efeitos singulares para reproduzi-los durante o processo de execução de uma aplicação em um processador. Tais efeitos podem ser classificados em duas categorias, conforme Dodd (DODD, 2003): (a) *single-event upsets* (SEUs) e (b) *single-event transients* (SETs). Os SEUs são inversões de valores lógicos (*bit-flips*) em elementos de memória comumente utilizados por processadores, tais como registradores ou células de memória. Nesta situação, um dos *bits* de um byte pode ser invertido logicamente devido ao efeito causado por partículas em circuitos integrados, ocasionado assim comportamentos inesperados em uma aplicação em execução no processador, dependendo da situação. Para o caso em que múltiplos bits são alterados, a denominação para este tipo de efeito é *single-event multiple-bit upsets* (MBUs), podendo ser observado em elementos de memória do circuito tão bem como no mesmo byte. Outro efeito bastante preocupante em dispositivos semicondutores está relacionado a pulsos transientes propagados na lógica combinacional de um circuito, os SETs, pois estes podem ser interpretados pelo circuito gerando alterações em um ou mais elementos de memória (BALEN, 2010).

2.1.2 Propriedades das técnicas de injeção de falhas

Para definição de um método comparativo entre diferentes abordagens e técnicas de injeção de falhas, Arlat (ARLAT, 2003) apresenta uma série de propriedades as quais definem o nível de efetividade das técnicas e possíveis limitações, são elas:

- Alcançabilidade: define a propriedade de gerar falhas em locais específicos ou muitas vezes inacessíveis para certos tipos técnicas. Um exemplo seria possibilidade de acesso a partes que compõem o circuito integrado;
- Controlabilidade: define a propriedade relacionada a espaço e tempo. O espaço simboliza uma determinada localização para injeção de uma ou mais falhas, e o tempo, simboliza o número de ciclos de relógio que devem ser contados até o momento da aplicação de uma falha;
- Repetibilidade: define a propriedade que trata da repetição exata dos experimentos ou com alto grau de exatidão. Dessa forma, tal propriedade depende da controlabilidade sobre espaço e tempo;
- Reprodutibilidade: define a propriedade de reproduzir os resultados obtidos em experimentos anteriores;
- Não intrusividade: define a propriedade de minimização de qualquer impacto no comportamento do sistema;
- Medição de tempo: define a propriedade relacionada à aquisição de informações de tempo, como contagem de ciclos de relógio, com associação aos eventos observados no experimento. Em alguns casos, um processador de referência é utilizado para geração de resultados;
- Eficácia: define a propriedade de redução do número de experimentos não significantes, que não produzem efeito no sistema alvo, produzindo erros de medida. Normalmente a abordagem para redução desta ineficácia é o aumento da amplitude, tempo ou duração dos estímulos, ou interferência física aplicada ao hardware em teste.

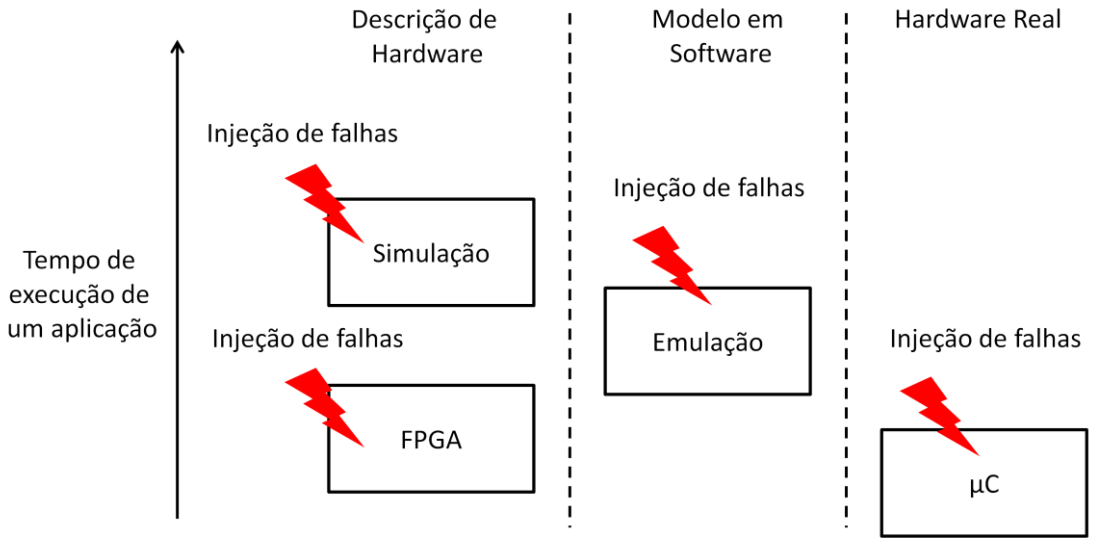
2.2 Abordagens para injeção de falhas em processadores

O uso de descrição de hardware para o processo de injeção de falhas em processadores é uma técnica muito comum na análise de mecanismos de tolerância. Com base na simulação lógica, utilizando-se de ferramentas como ModelSIM® (MENTOR GRAPHICS, 2014), é possível injetar falhas alterando-se elementos que definem o comportamento do hardware, por meio de recursos do simulador ou instrumentação do código VHDL ou *Verilog*. Outra abordagem aplicável ao uso da descrição de hardware é a injeção de falhas durante a emulação em FPGA, por meio de prototipação. Neste cenário, alterações no código que programa a FPGA, ou módulos de hardware para injeção de falhas agindo como sabotadores, são algumas das abordagens normalmente empregadas.

Diante da falta da descrição de hardware, outra abordagem aplicável é a utilização de emulação por software. Neste tipo de técnica é necessário ter o modelo do processador descrito em software. Com a utilização de uma máquina virtual, falhas podem ser injetadas por meio de alterações no código fonte do emulador e modelo, sendo estas aplicadas durante o processo de emulação. Além desta, outra técnica muito comum e mais realística é a injeção de falhas por radiação, submetendo o processador a um feixe de partículas, de forma que estas produzam alterações nos elementos de

memória do hardware real. Neste último, aceleradores de partículas podem ser utilizados para este fim.

Figura 2.1: Aspectos relativos ao desempenho de execução de aplicações em função da técnica de injeção de falhas com diferentes abordagens.



Tendo em vista a variedade de técnicas de injeção de falhas aplicáveis para realização de testes, normalmente o desempenho de execução associado a cada técnica pode ser um fator determinante para escolha da mais apropriada. Na figura 2.1 são apresentadas as abordagens que fazem uso de descrição de hardware, modelo em software e hardware real do processador. É possível notar que quanto mais próximo do hardware, para os casos da injeção de falhas em emulação com FPGA e por radiação, mais rápido é o desempenho de execução de uma aplicação, visando à aproximação da frequência alvo do processador. Sendo assim, quando se quer desempenho, a tendência é optar por técnicas que se aproximam do hardware, tanto quanto possível. Para o caso de técnicas de injeção de falhas com utilização de modelo do processador em software, é possível verificar um aumento em desempenho de execução das aplicações alvo de falhas em comparação a simulação lógica. É importante destacar, que com tal abordagem também é possível injetar falhas em aplicações mais complexas, tais como sistemas operacionais, normalmente não utilizadas com técnicas de simulação. Além disso, fatores relacionados ao desempenho de emulação por software estão diretamente associados ao hardware hospedeiro e qualidade do emulador, podendo este, se aproximar do desempenho do hardware real, dependendo da situação.

2.3 Metodologias de Injeção de Falhas

Os estudos na literatura relacionados ao desenvolvimento de metodologias e ferramentas para injeção de falhas, aplicáveis na realidade de processadores, normalmente fazem uso de uma das seguintes abordagens:

- Injeção de falhas por hardware;
- Injeção de falhas por software;

- Injeção de falhas *built-in*;
- Injeção de falhas por simulação.

A injeção de falhas por hardware é uma técnica que depende de equipamentos adicionais ao dispositivo o qual se está analisando. Sendo considerada uma técnica mais realística utiliza-se de equipamentos que possibilitam emular condições ou gerar erros em circuitos integrados. A principal finalidade desta é viabilizar a observação do comportamento do sistema diante de um modelo de falhas em que se tem interesse. Aceleradores de partículas são utilizados para injetar falhas dentro de circuitos integrados. O uso de feixe de íons é comumente aplicado em tal situação para avaliar *soft errors* no sistema observado (GUNNEFLO, 1989; MIREMADI, 1995). Outro método aplicado nesta abordagem é a utilização de lasers (*laser facilities*), que apresenta uma técnica não invasiva, não destrutiva e controlada, para geração de erros (SAMSON, 1997). A injeção de falhas por software permite através do uso de módulos de software, a modificação do conteúdo de elementos de memória utilizados pelas aplicações em execução. Tais elementos são afetados e erros são emulados sem a necessidade de hardware adicional. Outro aspecto fundamental de tal abordagem é a não necessidade do protótipo ou versão final de hardware em fases iniciais de projeto (KANAWATI, 1995; CARREIRA, 1995). No entanto, tal técnica, conhecida como injeção de falhas baseada em software, ou *software-implemented fault injection* (SWIFI), necessita de alta intrusão, sendo muitas vezes necessária a modificação do código fonte da aplicação, comprometendo o desempenho de execução da mesma. Uma alternativa em relação a abordagem SWIFI tradicional, é apresentada pelo uso de emulação de processadores (POTYRA, 2007; CABODI, 2010). Nesta abordagem, o método de injeção de falhas está associado à utilização de um emulador de processadores para injeção de falhas, ou utilização deste para validação de técnicas baseadas em SWIFI. A injeção de falhas do tipo *built-in* é baseada no princípio do uso de módulos de software, suportados por sistemas operacionais ou uso de interfaces de depuração. A primeira baseia-se no uso de módulos do Kernel do sistema operacional Linux (LINUX KERNEL ORGANIZATION, 2014), ou mecanismos semelhantes de outros sistemas operacionais. Já a segunda faz uso de interfaces de depuração do hardware alvo de falhas. Tais interfaces estão presentes na maioria dos processadores mais recentes, permitindo o acesso a registradores e/ou posições de memória para devida emulação de falhas (FIDALGO, 2006). Por fim, a injeção por falhas está relacionada a injeção de falhas no modelo de descrição de hardware, ou HDL (*Hardware Description Level* (JENN, 1994; BOUÉ, 1998; FOLKESSON, 1998) que normalmente é aplicada para esse fim. Outra variante desta técnica é o uso de prototipação em FPGA (*Field Programmable Gate Array*) (ANTONI, 2001; CIVERA, 2001) do processador alvo, proporcionando um maior número de possíveis simulações em um tempo consideravelmente inferior a outras técnicas.

2.4 Injeção de Falhas em Processadores por Hardware

Técnicas de injeção de falhas baseadas em hardware podem ser classificadas em duas categorias (HSUEH, 1997): (a) injeção de falhas com contato, no qual o injetor requer contato físico com o hardware em teste, explorando modificações nas condições de corrente e voltagem do sistema alvo e (b) injeção de falhas sem contato, em que o uso de equipamentos externos utilizados não exige contato com o circuito ou sistema alvo em teste. Esta última utiliza-se de fenômenos físicos para geração de erros no

sistema alvo como, por exemplo, aceleradores de partículas com utilização de íons pesados. Tais técnicas permitem o acesso a partes do sistema que outros métodos de injeção de falhas não conseguem acessar, impossibilitando a análise completa da solução de tolerância.

A modelagem de falhas deve descrever o mais próximo possível a realidade a qual o equipamento será submetido, neste caso a radiação. Falhas transientes produzidas por íons pesados, gerando SEUs em circuitos integrados, exemplificam uma boa abordagem para análise de mecanismos de tolerância a falhas (GUNNEFLO, 1989; MIREMADI, 1995). Este tipo de abordagem, a qual se utiliza de bombardeamento de circuitos por meio de um feixe de partículas, é definida como teste de radiação em nível terrestre. Bastante difundida, se baseada na utilização de feixes originados por aceleradores de partículas. Nesta técnica, a fidelidade à geração de condições espaciais é função direta da energia do feixe de partículas para um dado valor de transferência linear de energia, (LET - do inglês, *Linear Energy Transfer*). Feixes com alta energia reproduzem com maior aproximação o ambiente espacial. No entanto, há alto custo e alta complexidade para utilização de tal técnica.

Além de injeção de falhas utilizando feixe de íons pesados, é possível utilizar feixe de prótons, permitindo alta transferência de energia, chegando a valores de pico de 230 MeV em comparação a valores típicos gerados por feixes de íons pesados, que atingem em torno de 100 MeV. Dentre as maiores limitações apresentadas por aceleradores de partículas, podem ser destacadas: (i) disponibilidade, (ii) alto custo e (iii) informações sobre espaço e tempo (REED, 2003).

Diante das limitações apresentadas por técnicas comumente utilizadas, uma série de alternativas a injeção de falhas por meio de feixe de partículas tem sido explorada como, por exemplo, laser por pulso e micro feixes. O laser por pulso tem por princípio aumentar a controlabilidade e reprodutibilidade de geração de SEEs. Basicamente, são gerados pulsos de luz de aproximadamente 1 pico segundo com energia suficiente para a geração de pares elétron-lacuna no semicondutor em questão. Para o caso do micro feixe, o princípio baseia-se no uso de campo magnético e campo elétrico. O conjunto de íons, gerado pelo acelerador de partículas, passa por um conjunto de ímãs para produção de diâmetro menor que 1 micro metro e é acelerado pelo campo elétrico. Para valores de diâmetro em torno de 10 micro metros, a energia cai de 50 MeV para 15 MeV (REED, 2003).

Embora a técnica de injeção de falhas baseada em hardware se aproxime de situações reais, a mesma necessita de equipamentos dedicados para execução de experimentos e, em consequência disso, o custo do projeto tende a aumentar. Em adição, aspectos como controlabilidade e reprodutibilidade representam problemas evidentes nos experimentos. O uso de tal técnica é recomendado para fases finais do processo de desenvolvimento, sendo utilizadas outras técnicas nas demais etapas do projeto.

2.5 Injeção de Falhas por Software

Na busca por redução de custo no desenvolvimento de projetos, técnicas de injeção de falhas baseadas em software surgem como uma solução atrativa em virtude da não necessidade de um hardware customizado. Técnicas de SWIFI permitem que em alto

nível se possa acessar hardware e software de forma a reproduzir falhas enfrentadas em situações reais em ambientes com radiação (CARREIRA, 1998).

O desenvolvimento de ferramentas bem como o aprimoramento de técnicas já existentes são objetos de estudo constantes para emulação de ambientes de radiação. Outro fator importante a ser considerado é a flexibilidade quanto ao tipo de aplicação que se deseja executar no sistema alvo. Sistemas operacionais ou aplicações mais complexas podem ser executados em oposição a limitações apresentadas por outras técnicas.

O efeito da radiação em recursos de hardware, como registradores, é normalmente analisados com o uso de emulação de SEUs. Para tal se faz necessária a definição de um modelo de falhas para aplicação da mesma. Entretanto, injeção de falhas por software pode enfrentar uma série de limitações, conforme Hsueh e Carreira (HSUEH, 1997; CARREIRA, 1998):

- Inacessibilidade do software a certos recursos os quais se tem interesse em injetar falhas;
- Limitação de mecanismos para disparar o processo de injeção de falhas;
- Comprometimento da execução da aplicação ou sistema operacional em diversos aspectos devido à instrumentação no software em execução;
- Monitoramento do sistema para coleta de dados para análise do impacto causado por uma falha, ocasionando em muitos casos problemas no desempenho;
- Reprodução de eventos reais emulados em software;
- Baixa resposta em resolução de tempo para verificação de propagação de erros em recursos de hardware utilizados;
- Dificuldade de portabilidade para outros processadores devido a metodologias amarradas a certos recursos da plataforma de hardware em uso.

Tais fatores são diretamente relacionados ao nível de intrusividade da técnica utilizada.

Injeção de falhas pode ser realizada por qualquer tipo de instrução disponível em processadores que tenham acesso a memória e/ou registradores (BENSO, 1999). Tal procedimento pode ser realizado normalmente por dois métodos distintos, conforme Hsueh (HSUEH, 1997), como segue:

- Injeção de falhas em tempo de compilação;
- Injeção de falhas em tempo de execução.

O método de injeção em tempo de compilação realiza a injeção de falhas por meio de modificação do código fonte ou *assembly* em tempo de compilação. Dessa forma, o procedimento pode alterar tanto um recurso, como um registrador, quanto a própria elaboração do software, emulando uma situação de erro gerada pelo hardware. Sendo uma técnica estática, é normalmente utilizada para simulação de falhas permanentes. Além do processo de injeção, se faz necessária a alteração do software para verificação do efeito causado pela falha, não necessitando de outro programa para análise. Nesta técnica é possível notar o alto nível de intrusividade e limitação quanto a possibilidade de dinâmica no processo de injeção. Para solucionar tal requisito, o método de injeção de falhas em tempo de execução tem por finalidade injetar falhas enquanto a aplicação está sendo executada no processador, podendo assim emular uma situação de falha transiente. Sendo um processo dinâmico, deve existir um meio de parametrizar os

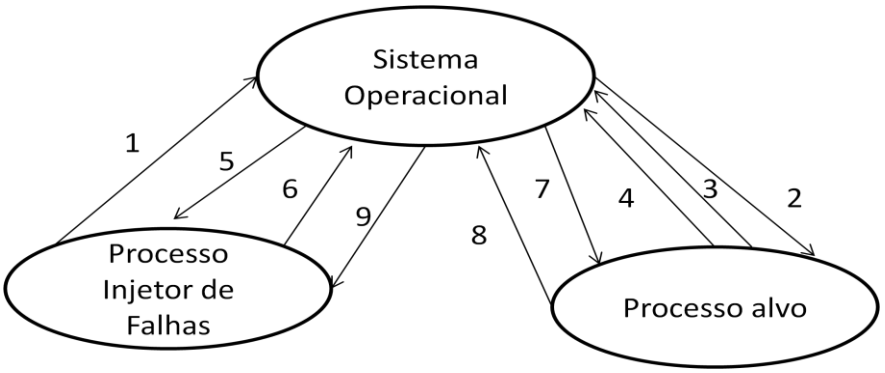
experimentos como, por exemplo, especificar o instante de tempo para injeção de uma falha. Recursos como interrupção, presentes em processadores, podem ser utilizados de forma que sub-rotinas sejam acionadas para injeção das falhas. Outros recursos como traps ou exceptions podem ser acionados com o uso de instruções especiais inseridas no código fonte, como breakpoints. Este tipo de procedimento permite a injeção de uma falha em elementos de memória antes, após ou durante seu uso em um processador.

2.5.1 Metodologias e Ferramentas de Injeção de Falhas

2.5.1.1 Ferramenta de injeção de falhas FERRARI

O desenvolvimento de ferramentas de injeção de falhas tem sido proposto na literatura como um meio de validação de técnicas e metodologias de injeção de falhas (KANAWATI, 1995; CARREIRA, 1998). Devido ao alto grau de acessibilidade a recursos de hardware atualmente disponibilizados por processadores, a injeção de falhas emulada por software pode ser uma abordagem de baixo custo e sem riscos de danificação ao processador em avaliação. Em contraste, as técnicas de injeção de falhas com alteração de código fonte ou *assembly*, bem como modificação em tempo real do programa em execução, Kanawati (KANAWATI, 1995) propôs uma metodologia menos intrusiva, sem necessidade de alteração do programa executado no sistema alvo. Para validação da técnica proposta foi desenvolvida uma ferramenta chamada FERRARI (*Flexible Software-based Fault and Error Injection System*). Esta ferramenta baseia-se na utilização de um processo ou programa adicional ao conjunto de programas já existentes em execução no sistema alvo, que é responsável por corromper elementos de memória em uso pelo processo ou programa alvo em execução.

Figura 2.2: Estrutura de simulação de injeção de falhas da ferramenta FERRARI.



Fonte: Figura adaptada de (KANAWATI, 1995).

A figura 2.2 ilustra a metodologia de injeção de falhas, que é composta pelo processo injetor de falhas e processo alvo.

Toda sistemática de alteração de valores de elementos de memória utilizados pelo processo alvo é realizada com o uso de chamadas de sistemas presentes no ambiente UNIX. Dessa forma, toda e qualquer alteração no processo alvo passa pelo Kernel do sistema operacional em execução. A ferramenta FERRARI, a qual incorpora a metodologia, divide a simulação em 9 passos, conforme segue abaixo:

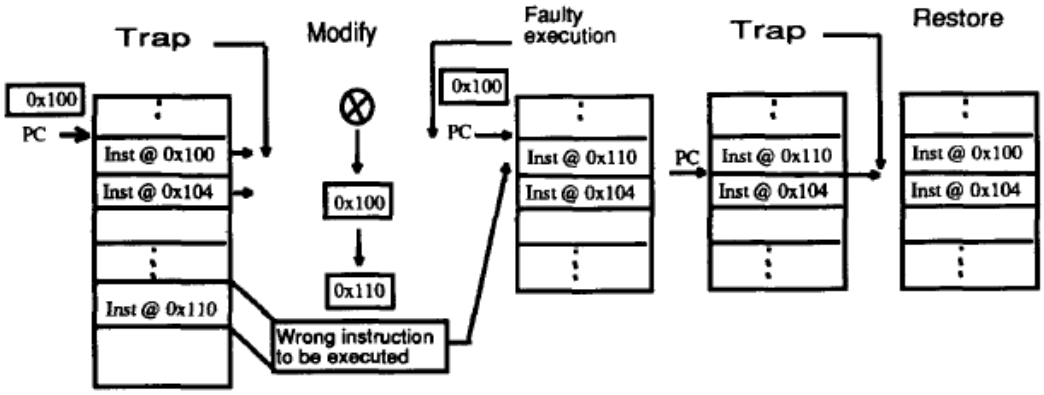
1. Requisição para criação do programa alvo, utilizando a chamada de sistema fork1;
2. Kernel do sistema cria o processo alvo;
3. Configuração do processo alvo de forma que o mesmo possa ser monitorado por outro processo, através do uso da chamada de sistema *ptrace* (LINUX PTRACE MAN PAGE);
4. Requisição para execução do processo alvo por parte do Kernel;
5. Recepção de notificação de estado de início de execução do processo alvo por parte do Kernel;
6. Requisição para alteração de elementos de memória com possível uso por parte do processo alvo, realizada através da chamada *ptrace*;
7. Alteração do programa conforme requisição, através do uso da chamada *ptrace*;
8. Finalização do programa alvo;
9. Notificação de terminação do processo alvo.

A estrutura de implementação do sistema FERRARI é composta por um conjunto de módulos: (i) módulo de inicialização e ativação, (ii) módulo de parametrização, (iii) módulo de injeção de falhas e (iv) módulo de coleta e análise de dados. O módulo de inicialização tem por função mapear recursos utilizados pelo programa alvo e realizar uma execução livre de falhas, na qual ocorre o armazenamento do resultado da execução do programa para futuras comparações. O módulo de parametrização interpreta entradas do usuário para o processo de simulação, tais como tempo de simulação, tipo de falhas e modelos de falhas. A tabela 2.1 apresenta um resumo dos parâmetros disponibilizados pela ferramenta para injeção de falhas na aplicação alvo. É importante destacar o parâmetro de modelo de falhas, o qual mapeia falhas de baixo nível, observadas em hardware, para falhas em alto nível, emuladas por software.

Tabela 2.1: Parâmetros de simulação para injeção de falhas.

Parâmetro	***Descrição***
Tipo de medida	Seleciona o tipo de medida relacionada ao experimento, são elas: **cobertura**, que computa a efetividade de detecção por parte dos mecanismos de recuperação de falhas e **latência**, que mede o tempo de detecção de uma falha de forma que um mecanismo de recuperação possa entrar em ação.
Localização	Especifica uma localização na qual se quer injetar uma falha como posição de memória, registrador e etc.
Tempo	Especifica o tempo e duração no qual uma falha deve ser injetada, medido em ciclos de instrução. Tal parâmetro pode ser aleatório.
Duração	Seleção entre falhas permanentes e transientes.
Tipo de Falha	Alteração de um bit através de uma operação XOR com uma máscara de bits ou alteração de um byte.
Modelo de falhas	Seleciona um modelo de falhas para injeção, são eles: • Erro de decodificação de instrução: alteração do contador de programa em processo de decodificação de instruções para execução de outra instrução; • Erro de operadores: modificação de endereço de memória utilizado em uma instrução que faz operações em posições de memória.

Figura 2.3: Modelo de falhas para execução de instrução errada.



Fonte: KANAWATI (1995).

O processo de injeção de falhas é realizado pelo módulo de injeção. O módulo verifica se a posição de memória definida ou registrador para injeção de uma falha foi

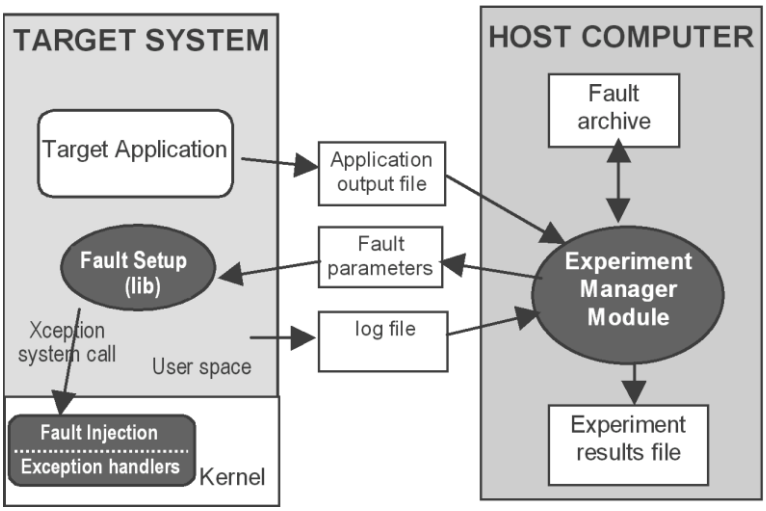
acessado, por meio uma *trap* do processador. Caso seja acessada tal posição, uma ação é executada. A figura 2.3 ilustra o modelo de falhas de erro de decodificação de instrução. Para injeção de uma falha uma máscara de bits é aplicada no contador de programa alterando o seu valor de 0x100 para 0x110. Tal procedimento ocasiona a execução de outra instrução. Após a execução desta outra instrução, é gerado um evento de *trap* e o valor anterior do contador de programa é restabelecido, e então o programa executa a partir da instrução seguinte.

Por fim, após o processo de injeção de uma falha, o módulo de análise e coleta de dados entra em ação. Tal módulo analisa a execução do programa alvo, armazena resultados da execução e guarda valores de latência de detecção de uma falha por parte de mecanismos de recuperação do sistema. O processo de análise de resultados é realizado a partir de arquivos de registros gerados durante as simulações. Tais arquivos permitem ao analisador verificar se o programa terminou normalmente, foi interrompido por um mecanismo de verificação de erro ou uma condição de tempo máximo de execução, pré-definida, foi excedida.

2.5.1.2 Ambiente de injeção de falhas Xception

Como evolução das limitações apresentadas por técnicas que tem alta intrusividade no sistema alvo, Carreira (CARREIRA, 1998) propõe uma metodologia baseada na utilização de recursos de monitoramento e depuração presentes em processadores. O software *Xception* introduz um ambiente de injeção de falhas com mínima interferência na aplicação alvo. Sendo uma técnica de baixa intrusividade, a mesma não prevê alterações na aplicação alvo em execução no sistema operacional. Além de processos em espaço de usuário, tal técnica permite injetar falhas em espaço de Kernel.

Figura 2.4: Arquitetura do ambiente de injeção de falhas Xception.



Fonte: CARREIRA (1998).

A figura 2.4 apresenta a estrutura do ambiente de simulação *Xception*. O ambiente é composto por um sistema alvo e um computador de controle. A estrutura de software é definida por três módulos, são eles:

- Módulo de Kernel, que suporta rotinas de injeção de falhas e tratamento de exceções do processador;
- Módulo de configuração de falhas, que é uma biblioteca responsável por receber parametrizações provenientes do computador de controle para devida configuração do Kernel;

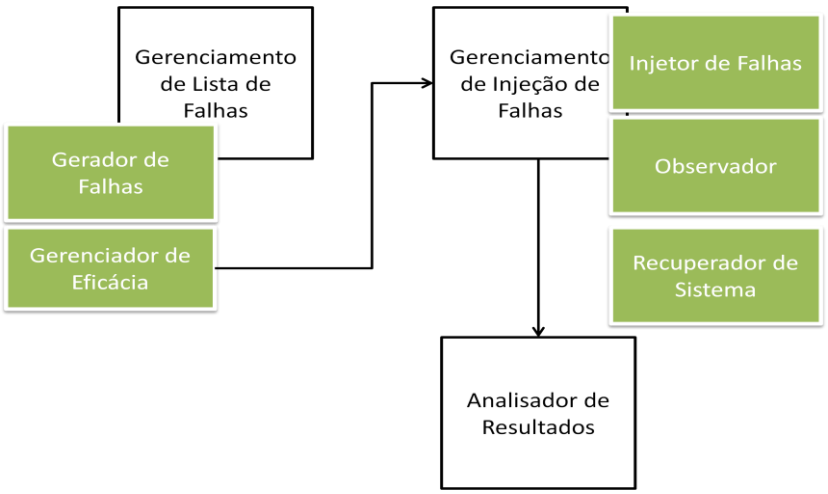
Módulo de gerenciamento de experimentos ou EMM (Experimental Module Manager), que é um software que executa no computador de controle disponibilizando uma interface de configuração do ambiente, injeção de falhas e coleta e análise dos resultados.

A grande vantagem em relação a outros tipos de técnicas mais intrusivas é a utilização de exceções de hardware programadas via recursos de monitoramento e depuração do processador. Tal abordagem extingue o uso de instruções para geração de traps adicionadas no programa alvo. Outro importante aspecto a salientar é a condição para disparar o processo de injeção de uma falha. O conjunto de critérios ou atributos que podem ser observados é mais amplo, proporcionando mais situações para elaboração de modelos de falhas. No entanto, mesmo sendo uma técnica que tem pequeno impacto na execução da aplicação alvo, ela ainda adiciona intrusividade à medida que necessita alterar o Kernel do usuário por um que contenha a implementação das rotinas de injeção de falhas e configuração dos recursos de hardware utilizados pela metodologia.

2.5.1.3 Sistema de Injeção de Falhas EXFI

O trabalho proposto por (BENSO, 1999) apresenta uma metodologia de injeção de falhas transientes que faz uso de um modo de configuração de processadores conhecido como *Trace Exception Mode*. Desta forma, ao posicionar uma rotina de software na memória para tratar eventos reportados pelo processador, neste modo especial de execução, a mesma se encarrega de computar o tempo para injeção de uma falha, entre outras funções.

Figura 2.5: Ambiente de injeção de falhas EXFI.



Fonte: Figura adaptada de (BENSO, 1999).

O procedimento proposto pelo autor resume-se em injetar a falha e verificar se o processo alvo de injeção de falhas não ultrapassa o limite máximo de tempo de

execução, já pré-definido para os experimentos. Ao contrário das ferramentas FERRARI e Xception, o ambiente de software EXFI (Exception-based Fault Injector) não necessita de qualquer modificação no código da aplicação ou sistema operacional alvo.

O sistema EXFI é composto basicamente por três módulos responsáveis pelo processo de experimentação. O ambiente de injeção de falhas é apresentado na figura 2.5. O módulo de gerenciamento de lista de falhas é o primeiro bloco acionado pelo processo. O mesmo tem por finalidade gerar uma lista de falhas conforme parametrização do usuário do sistema. O bloco responsável por esta tarefa é denominado gerador de lista de falhas. A especificação do número de falhas, região de memória de interesse e registradores são exemplos de parâmetros comuns os quais o gerador de falhas leva em conta para elaboração desta lista. Após a geração desta, entra em ação o bloco classificado como gerenciador de eficácia. Neste bloco a lista de falhas é otimizada removendo-se falhas as quais já se sabe previamente o efeito que causará no sistema. Outro papel importante é o armazenamento de informações da execução da aplicação ou sistema em condições livres de falhas, chamada simulação livre de falhas. Tais informações serão utilizadas posteriormente pelo módulo de gerenciamento de injeção de falhas (GIF).

O bloco GIF é o bloco mais complexo do ambiente de experimentação. Além da tarefa de injeção de falhas, efetuada com o controle do bloco injetor de falhas, o módulo GIF possui um bloco denominado observador, responsável pela observação dos efeitos causados pela falha. Por fim há o bloco responsável pela recuperação do sistema, denominado recuperador de sistema. Este bloco é aplicável para falhas que ocasionem eventos de hardware, tais como exceções. Nesta situação o bloco necessita de total controle do sistema de forma que uma nova execução da aplicação seja possível mantendo as mesmas condições sem a presença de uma falha. Cabe destacar outras tarefas fundamentais que desempenhadas pelo sistema GIF, são elas:

- Inicialização do sistema, que tem por finalidade carregar a aplicação em uma região de memória para execução;
- Inicialização dos parâmetros de injeção de falhas conforme a falha selecionada a partir da lista;
- Habilitação do monitoramento de execução de código (*code tracing*) e início de execução;
- Injeção de falhas conforme o número de instruções executadas;
- Verificação do número de instruções máximo pré-definido no experimento;
- Classificação de falhas.

O processo de injeção de falhas é efetuado por uma rotina que se utiliza do modo de monitoramento de execução do processador. Ao ser acionada, a mesma computa o número de instruções. Quando o número de instruções requerido para injeção de uma falha é atingido, a mesma é aplicada. Após a injeção, inicia-se a observação do comportamento da aplicação diante da falha, classificando-se os efeitos observados. São eles:

- Falha silenciosa, representada por uma falha que não produz alteração no comportamento do sistema;
- Mecanismo de detecção de erro (EDM), representado pela detecção presente no próprio software ou sistema;

- Violação por falha silenciosa, representada por uma alteração do comportamento do sistema não detectada por EDM, ocasionando erro nos resultados esperados;
- Limite de tempo de execução, representado pela alteração não detectada do comportamento do sistema que ultrapassa um limite de tempo pré-definido.

Por fim, temos o módulo chamado analisador de resultado, que é responsável por gerar um relatório de todo o processo de experimentação diante da lista de falhas.

2.5.1.4 Rotinas de Software para Injeção de Falhas

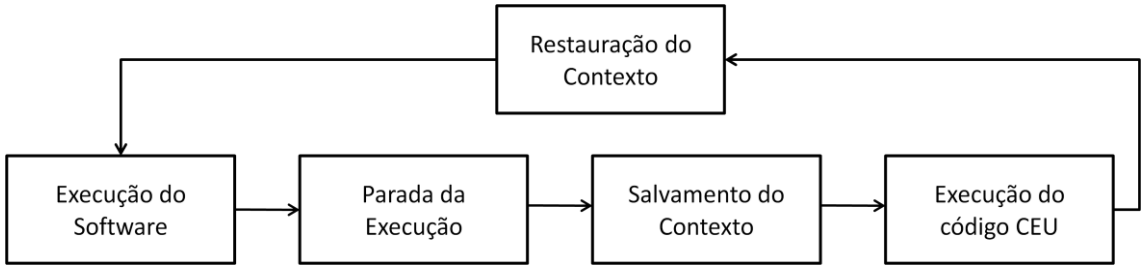
Na busca por alternativas de análise dos efeitos da radiação em equipamentos eletrônicos, (VELAZCO, 2000) apresenta uma nova estratégia para geração de *upsets* via injeção de falhas de forma aleatória em localização e tempo. O método *Code Emulating Upset* (CEU), que emula *upsets* por meio de rotinas de software, introduz uma abordagem baseada no uso de recursos assíncronos, tais como interrupções presentes em processadores. Este tipo de abordagem permite a injeção de falhas em meio a execução de aplicações. Tal metodologia leva em conta a limitação de acesso a elementos internos que compõem um processador.

O processo de injeção de falhas é efetuado por uma rotina de software a qual é executada por meio de uma interrupção do processador. A rotina de falha emula um *bit-flip* ou alteração de valor lógico em um elemento de memória como registrador ou posição de memória. A alteração do bit do elemento de memória é realizada por meio da seguinte sequência:

1. Leitura do conteúdo do elemento de memória para variável temporária;
2. Aplicação de uma máscara de bits com uso de uma operação lógica XOR nesta variável temporária;
3. Escrita do valor da variável temporária no elemento de memória alvo da injeção de falha.

O fluxo de injeção de falhas é ilustrado na figura 2.6. A partir de uma interrupção detectada pelo processador, a execução do software é parada, e então uma rotina programada no vetor de interrupção do mesmo é executada. Tal rotina salva o contexto de execução da aplicação na pilha e salta para posição de memória onde se encontra a rotina de CEU. Após a execução da rotina de CEU, a qual injeta a falha, o contexto de execução da aplicação é retomado e a aplicação segue a execução do ponto onde parou.

Figura 2.6: Metodologia de injeção de falhas por CEU.



Fonte: Figura adaptada de (VELAZCO, 2000).

É importante destacar a flexibilidade de tal técnica, visto que através da manipulação da pilha é possível injetar falhas no contador de programa e outros registradores não acessíveis diretamente por meio do conjunto de instruções. Após o processo de injeção de falhas, começa a etapa de observação dos efeitos no sistema quanto ao resultado esperado após a execução da aplicação. A classificação dos erros é realizada através da comparação do conteúdo de uma região de memória, na qual o programa posiciona estes resultados com os obtidos através de uma execução livre de falhas e a comparação do tempo total de execução com o tempo esperado. São definidos três grupos para classificação de erros. O primeiro grupo, erros tolerados ou mascarados, trata dos erros mascarados durante a execução. Neste grupo são consideradas falhas injetadas em elementos de memória não utilizados pelo programa, assim como falhas injetadas em elementos de memória poucos antes de serem inicializados pela aplicação. O segundo grupo, erros de resultado, trata de erros nos resultados esperados após a execução da aplicação. E por fim, erro de sequência, que engloba uma condição de ultrapassagem de tempo de execução, normalmente ocasionada pela falta de resposta por parte do processador. Tipicamente esse tipo de erro resulta na necessidade de um processo de reinicialização por hardware do processador a fim de iniciar novamente uma nova simulação.

O uso da técnica de injeção de falhas CEUs foi comparado com técnicas de injeção de falhas por radiação demonstrando grande concordância de resultados. Ao contrário de técnicas abordadas por FERRARI, Xception e EXFI, a abordagem sugerida por (VELAZCO, 2000) define uma técnica mais portável para outros processadores, não dependendo de recursos específicos de cada um. Outra abordagem para validação de técnicas de detecção de falhas em nível de software foi utilizada por (NICOLESCU, 2001). A técnica consiste na utilização de um processo adicional, executando simultaneamente a aplicação alvo, denominado injetor de falhas. Tal aplicação aguarda um período de tempo aleatório e quando o mesmo é esgotado seleciona um elemento de memória e bit aleatórios para injeção de uma falha. Após a injeção da falha, o injetor observa o comportamento e transmite os resultados para um computador pessoal que controla os experimentos. Contrastando a técnica de CEU, a técnica citada acima é mais intrusiva, visto que executa uma aplicação com a função de sabotar no mesmo processador em que executa a aplicação alvo de falhas. Para casos em que se necessita de tempo de resposta de uma aplicação, como as de tempo real, o uso de um processo concorrente para injeção de uma falha acarreta no comprometimento da simulação de um ambiente mais realístico.

Uma abordagem semelhante ao método CEU foi utilizada por (ACLE, 2011). Visando analisar técnicas de tolerância por meio de redundância de tarefas em sistemas baseados em FPGA, com uso de memória SRAM executando IP core do processador NIOS-II, o autor propôs uma técnica de validação através do uso de uma rotina de injeção de falhas por software, executada em contexto de interrupção. O método se baseia na injeção de SEUs em partes acessíveis por software. A rotina de injeção de falhas é ativada durante a execução de uma aplicação. O acionamento é realizado em tempo aleatório, medido em ciclos de relógio. A seleção do elemento de memória alvo da injeção da falha e máscara de bits a ser aplicada para geração de SEU, via operação lógica XOR, são especificadas em uma região de memória acessada externamente por meio de uma interface de depuração (debug interface) do processador, por meio de computador de controle dos experimentos. Ao final da injeção de uma falha inicia-se a etapa de análise e classificação dos erros observados no sistema. Tendo em vista a

atratividade do uso FPGAs em aplicações críticas, tal metodologia possibilita avaliar e auxiliar no desenvolvimento de técnicas para melhorar a confiabilidade de sistemas baseados em IP cores.

Em uma análise comparativa de três técnicas de injeção de falhas por hardware e uma técnica SWIFI, (ARLAT, 2003) utiliza-se de uma abordagem alternativa de injeção de falhas por software. A mesma baseia-se na injeção de falhas antes da execução do código de máquina. O método altera o código de máquina antes de carregar a aplicação no processador. Sendo a falha injetada somente na aplicação alvo, o nível de intrusividade é baixo. Além disso, tal técnica permite uma acessibilidade não permitida por alteração de código fonte. No entanto, tal técnica é menos realística para falhas transientes, visto que simula somente falhas permanentes.

2.6 Injeção de Falhas *Built-in*

A técnica de injeção de falhas do tipo *built-in* representa o grupo de técnicas que são desenvolvidas dentro do sistema em teste, podendo ser acionado em tempo real para realização de injeção de falhas. Porém, cabe separar esta abordagem em dois tipos: (i) desenvolvida exclusivamente por software, utilizando para isso módulos de software para injeção de falhas ou (ii) desenvolvida com o uso de uma interface de depuração suportada pelo próprio equipamento em teste, alvo da injeção de falhas. Esta interface é conhecida como recurso de depuração comum dentro do próprio circuito integrado ou OCD (*Common On-Chip Debugging)*.

2.6.1 Módulos de software para injeção de falhas

Com a tendência de utilização de COTS adotada no desenvolvimento de aplicações críticas, surgiu a necessidade de validação de recursos de tolerância implementados em software. O sistema embarcado que viabiliza a função do produto requer alto grau de recuperação diante de eventos externos caracterizando a confiabilidade do sistema. Tendo em vista tal necessidade, Cabodi (CABODI, 2010) propõe um sistema integrado de injeção de falhas para utilização tanto em plataformas reais de hardware quanto em ambientes de simulação. Neste último o objetivo é antecipar, em fases iniciais de projeto, problemas que podem ocorrer na plataforma real de hardware.

A proposta de um framework independente de arquitetura é sugerida pelo autor com base no uso de módulos de Kernel do sistema operacional Linux para emulação de SEUs. Um aspecto destacado é a possibilidade de injeção de falhas em sistemas de tempo real (RT – do inglês, *Real-Time*). A abordagem destaca a não utilização de recursos de depuração normalmente presentes em dispositivos COTS que, quando utilizados, acabam degradando ou comprometendo o desempenho em termos de requisito de tempo de execução, muito importante para aplicações de tempo real. Outro aspecto a salientar é a baixa intrusividade do suporte em software para injeção de falhas, que não necessita parar a execução do Kernel para injetar uma falha.

2.6.1.1 Sistema Linux e suporte a tempo real

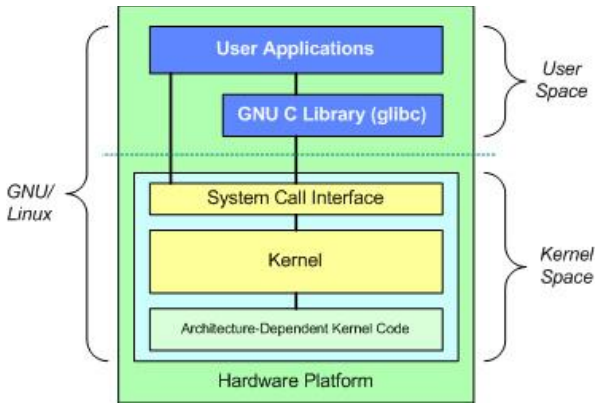
A utilização do sistema operacional Linux, objeto de estudo para validação da metodologia empregada nos experimentos de Cabodi (CABODI, 2010), é justificada

devido a premissa de sistemas atuais necessitarem de maior complexidade no software para gerenciamento do hardware. Desta forma, aplicações para determinado fim acabam por executar em um sistema mais complexo. O Linux não é um sistema de tempo real em sua natureza. Sendo assim, uma série de modificações se faz necessária em sua estrutura para suportar aplicações em tempo real. Alguns projetos como XENOMAI (XENOMAI PROJECT, 2014), entre outros, tentam dar suporte a um escalonamento de tarefas mais apropriado para tempo real.

2.6.1.2 Módulo de Kernel

A estrutura básica de um sistema operacional Linux pode ser visualizada na figura 2.7. Nesta estrutura temos como primeira camada o espaço de usuário, onde as aplicações executam. A segunda camada é denominada de espaço de Kernel, onde módulos são responsáveis por gerenciar recursos de hardware entre outras funções. A comunicação entre espaço de Kernel e espaço de usuário é realizada através de chamadas de sistema, disponibilizadas por uma biblioteca escrita em linguagem C. Sendo considerado um sistema modular, o Linux permite que módulos de Kernel sejam carregados em tempo real ou integrados ao próprio Kernel em tempo de compilação.

Figura 2.7: Arquitetura organizacional do sistema Linux.



O framework proposto por Cabodi (CABODI, 2010) tem, como um dos elementos que compõem o sistema, um módulo não integrado ao Kernel do sistema operacional Linux. Tal módulo é o elemento central do framework. Através dele é possível coletar estatísticas, verificar informações da memória virtual e acessar elementos de memória os quais a aplicação alvo de injeção de falhas irá utilizar na sua execução. Através deste módulo é então possível injetar falhas de uma forma mais rápida, eficiente e transparente para aplicação, visto que tais ações são executadas em espaço de Kernel.

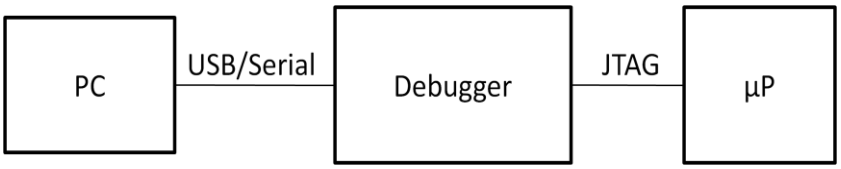
2.6.2 Injeção de falhas com uso de OCD

Na busca por uma solução de baixo custo, mais integrada e efetiva para injeção de falhas, outra técnica, também classificada como *built-in*, é a injeção de falhas por emulação utilizando-se de interfaces de depuração ou *debug* presentes em processadores (FOLKESSON, 1997). Estas interfaces possibilitam o acesso direto a registradores,

posições de memória, entre outros recursos. Disponível na maioria dos processadores, a interface de OCD (*On-chip Common Debugging*) tem como maior desvantagem a falta de padronização na sua elaboração. Devido a esta questão fabricantes de processadores disponibilizam sua própria interface e meios de acesso aos recursos de hardware. Um exemplo é o equipamento BDM (*Background Debug Mode*) disponibilizado para família PowerPC de processadores da empresa Freescale® (FREESCALE SEMICONDUCTOR). Tal equipamento possibilita o controle e monitoramento de um processador através de uma conexão a um PC. O dispositivo BDM acessa os recursos do processador através de sua própria interface JTAG3.

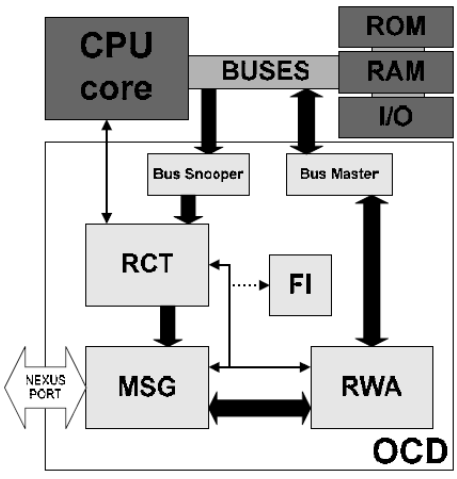
A atratividade da técnica resume-se a acessibilidade dos recursos de hardware em tempo real a execução de uma aplicação. Acesso à memória, registradores, suporte a breakpoints, entre outros, são alguns dos itens disponibilizados por este tipo de tecnologia. Desta forma, é então possível emular uma falha e observar o seu efeito na aplicação em execução. Tal abordagem é transparente para a aplicação não necessitando qualquer modificação no software. No entanto, dependendo do recurso que se deseja emular uma falha, pode ser necessário colocar o processador em um estado especial de depuração, comprometendo o desempenho de execução.

Figura 2.8: Utilização de interface de depuração de processadores.



Um cenário típico para construção de um sistema de injeção de falhas utilizando OCD pode ser observado na figura 2.8. Neste cenário, o PC é conectado ao equipamento de depuração ou *debugger* que, por sua vez, é conectado ao processador através de uma interface apropriada. Todo o processo de injeção de falhas passa pelo PC e ações relacionadas ao hardware são aplicadas pelo *debugger*.

Figura 2.9: Estrutura OCD-FI para injeção de falhas.



Fonte: FIDALGO (2006).

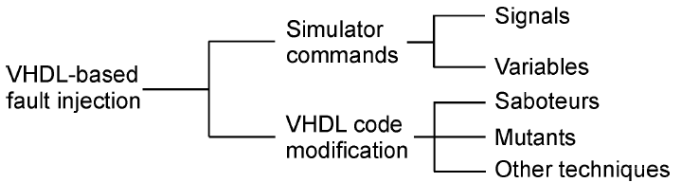
Partindo do princípio que o processador seja colocado em um estado especial durante o processo de injeção de falhas, ou em certos casos a execução da aplicação seja interrompida para o procedimento de emulação de uma falha, o desempenho do sistema é comprometido. Outro fator a ser considerado é o controle dos experimentos, que está intimamente ligado ao desempenho do PC conectado ao equipamento de depuração. Com o intuito de abordar tal limitação, Fidalgo (FIDALGO, 2006) propõe uma modificação na estrutura OCD de processadores para injeção de falhas. Na figura 2.9, a nova estrutura OCD, intitulada OCD-FI, além do suporte para depuração também suporta a injeção de falhas através do bloco FI, desenvolvido para este fim. Nesta estrutura o processo de injeção de falhas não é mais dependente de um dispositivo de depuração. Em tal abordagem a configuração da falha a ser injetada é realizada através de mensagens respeitando o padrão NEXUS1 por meio de uma porta específica de hardware. Para tal, o bloco RCT (*Run Control & Trace*), responsável pelo monitoramento do barramento interno do processador e pela comunicação com o módulo FI, age como um acelerador no processo de injeção de falhas.

Embora seja uma metodologia bastante eficiente, este padrão raramente é adotado pelos fabricantes, tornando inviável a sua utilização para injeção de falhas e avaliação de mecanismos de tolerância.

2.7 Injeção de Falhas por Simulação

Técnicas de injeção de falhas baseadas em simulação ou emulação de hardware tem sido objeto de estudo na literatura (JENN, 1994; BOUÉ 1998; ANTONI, 2001; CIVERA 2001). Por meio de uma conhecida como HDL, é possível construir a estrutura e descrever o comportamento de circuitos lógicos em alto nível. Após esta fase de projeto o código que descreve este hardware produz o próprio. A linguagem VHDL é um exemplo bem difundido e utilizado em projetos de ASICs (*Application Specific Integrated Circuit*), sendo aplicada em análise de mecanismos de tolerância a falhas devido a sua alta controlabilidade e estrutura bem organizada. Esta descrição de hardware é realizada com a definição de componentes que se comunicam entre si por meio de sinais, podendo simbolizar barramentos e pinos, entre outros elementos. Assim, cada componente define suas características e funções no sistema. Outro fator relevante é a hierarquia entre os componentes a qual é definida em tempo de programação.

Figura 2.10: Técnicas de injeção de falhas baseadas em simulação.



Fonte: BARAZA (2008).

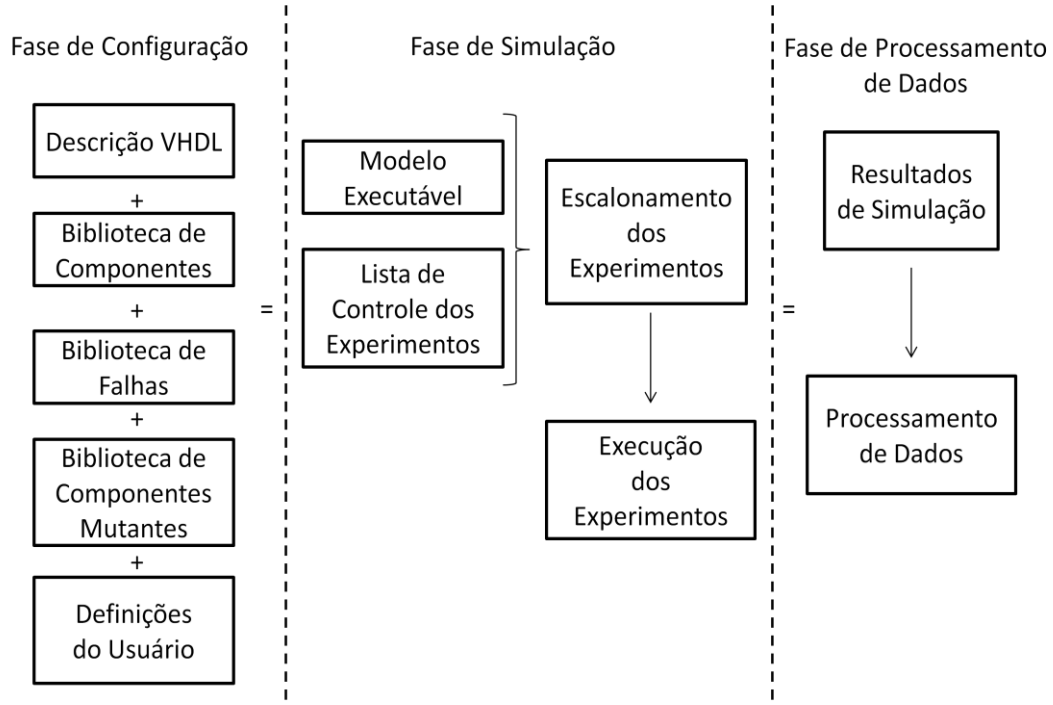
O processo de injeção de falhas utilizando simulação pode ser realizado com alteração do código VHDL e/ou utilizando recursos presentes no próprio simulador, conforme exemplificado na figura 2.10 de Bazara (BAZARA, 2008). Nesta são apresentas as possíveis abordagens no processo de injeção de falhas com utilização de comandos presentes no próprio simulador, para alteração de valores de sinais e/ou variáveis, e introduz alguns elementos que estão presentes no contexto de HDL, tais como os códigos VHDL sabotador e VHDL mutante, entre outras técnicas aplicáveis.

A utilização de emulação de falhas com uso de código VHDL pode ser realizada com simulação, utilizando ferramentas como ModelSIM, ou com emulação de hardware, utilizando FPGA. Esta última torna-se muito atrativa, visto que reduz consideravelmente as limitações presentes na técnica de simulação.

2.7.1 Metodologia de injeção de falhas por simulação

No universo de simulação de hardware, Jenn (JENN, 1994) propõe uma metodologia de injeção de falhas em modelos de hardware descritos em VHDL. Uma ferramenta de injeção de falhas, chamada MEFISTO (*Multi-level Error/Fault Injection Simulation Tool*) é proposta no trabalho para injetar falhas nestes modelos com base nas técnicas de sabotagem, relacionada a sinais e variáveis e mutação de componentes, que descrevem o hardware.

Figura 2.11: Estrutura simplificada da ferramenta MEFISTO.



Fonte: Figura adaptada de (JENN, 1994).

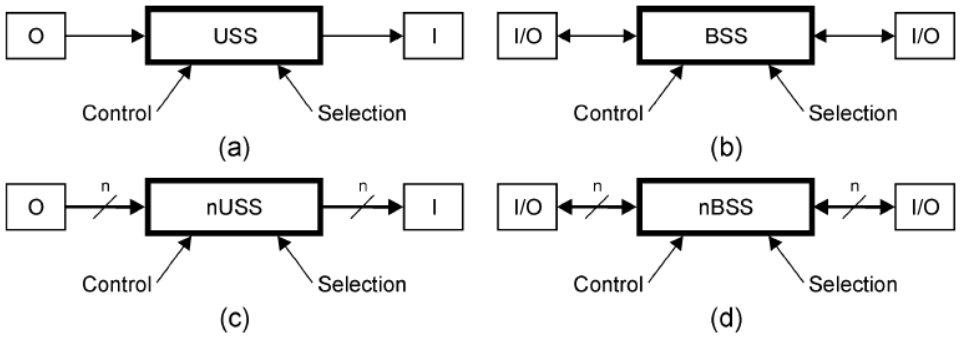
O processo de simulação é realizado em três fases, como ilustrado pela figura 2.11. A primeira fase trata-se da configuração, que objetiva a geração de um modelo de

hardware executável para os experimentos com uma lista de controle dos mesmos, que define comandos executados sob o simulador durante o processo de simulação. Estes comandos definem ações sob os elementos sabotadores e/ou mutantes. Nesta etapa de configuração o usuário participa da seleção dos elementos alvo de falhas (sinais, variável e componentes) dentre uma lista disponibilizada pelo próprio simulador. O tipo de falha a ser emulada é também especificado pelo usuário. Com base em um conjunto de bibliotecas de falhas, componentes e componentes mutantes, o processo de configuração gera, ao final desta etapa, o modelo de falhas e a lista de controle dos experimentos. A segunda fase é denominada simulação, em que o escalonador de experimentos entra em ação alocando um simulador disponível em uma rede. Em cada simulador é realizado o experimento referente à falha selecionada e são aplicados os comandos pertinentes a esta, conforme a lista de controle dos experimentos. Tal lista contém também os elementos alvo a serem observados, os quais devem ser detalhados quanto ao seu comportamento durante o experimento em um arquivo gerado pelo simulador. Com base nisso, o simulador gera arquivos de saída do experimento que são analisados posteriormente. A fase final está relacionada ao processamento dos dados obtidos durante o experimento gerado pelo simulador. Outro aspecto importante é a classificação de erros definida em duas categorias para os experimentos. A primeira define erros diretos, os quais se manifestam nas portas de saída do hardware, e a segunda define erros indiretos, que são representados por alterações do valor de registradores, que permanecem sem efeito por um determinado tempo até que a aplicação faça uso de tais registrados.

O MEFISTO-L introduz melhorias no simulador MEFISTO de forma que os mecanismos de tolerância a falhas sejam avaliados de uma forma mais detalhada (BOUÉ, 1998). Além disso, há um maior esforço em realizar a mutação de código VHDL em troca da utilização de recursos disponíveis no simulador de código. Sendo assim, a ferramenta MEFISTO-L tem seu próprio analisador de código o qual modifica o código VHDL e gera um código que permite ativação de falhas durante a simulação independentemente do simulador utilizado nos experimentos.

A dificuldade em automatizar o processo de inserção de códigos HDL sabotadores e mutantes de forma que uma ferramenta de injeção de falhas genérica possa ser construída tem se mostrado relativamente complexa. Desta forma, Baraza (BARAZA, 2008) apresenta uma nova abordagem de elaboração de sabotadores e mutantes que viabilizam de uma forma mais fácil a sua inserção de forma automatizada em descrições de hardware alvo de análise. A ideia central baseia-se na análise do melhor modelo a ser utilizado em cada caso. Com isso, o autor propõe uma simplificação de modelos de sabotadores e mutantes já existentes eliminando em muitas a ambiguidade no processo de escolha do modelo mais apropriado a ser inserido.

Figura 2.12: Estrutura do modelo de sabotadores: (a) USS, (b) BSS, (c) nUSS e (d) nBSS.

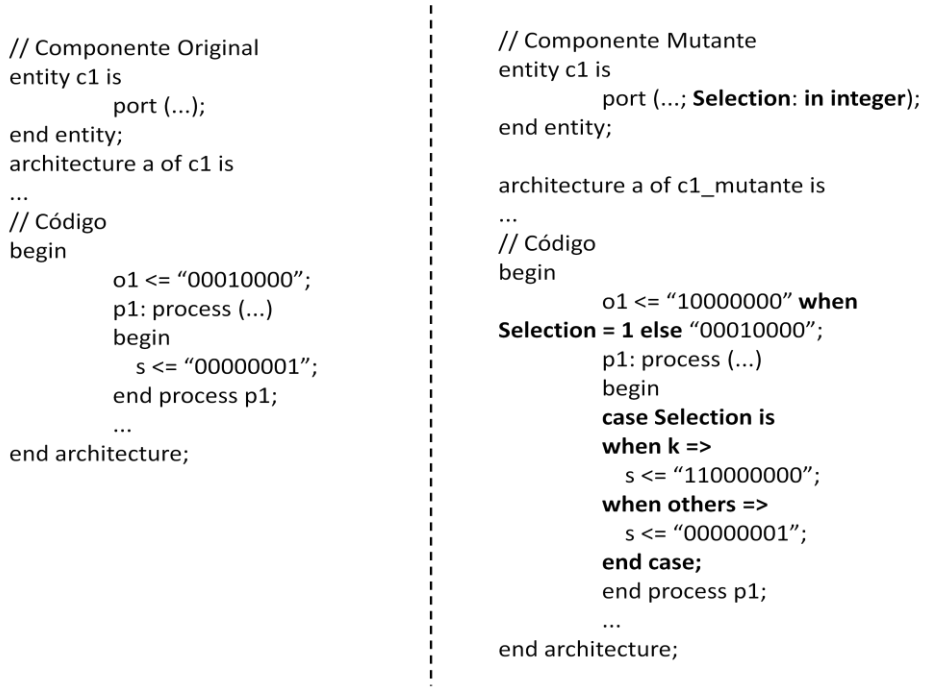


Fonte: BARAZA (2008).

A figura 2.12 ilustra o novo modelo de sabotadores baseada na melhoria de existentes. O sabotador USS, conhecido como sabotador unidirecional serial tem a mesma estrutura do sabotador serial simples (SSS), porém com suporte a mais modelos de falhas. O BSS, sabotador bidirecional serial, além de suportar mais modelos de falhas elimina o sinal de controle de leitura e escrita do modelo anterior, denominado sabotador bidirecional serial simples (SSBS). O nUSS propõe um modelo que elimina todos os outros modelos com suporte injeção de injeção de falhas em múltiplos bits. E finalmente o nBSS substitui todos os modelos bidirecionais com suporte a múltiplos bits. O processo de injeção de falhas resume-se no controle de dois sinais: CONTROL e SELECTION. O primeiro determina a noção de espaço e o segundo, por sua vez, habilita o processo de injeção de uma falha. Tanto falhas únicas como múltiplas podem ser acionadas através destes sinais.

O processo de automatização de inserção de sabotares no código fonte HDL utiliza-se de um analisador de código. Este analisador percorre uma árvore que possui uma completa estrutura dos modelos, e através de processos de cópia de código e devida modificação, ao final do processo é gerado o código modificado com suporte a injeção de falhas por meio de sabotadores. Para o caso de mutantes, o trabalho propõe um método de força bruta, para que cada arquitetura presente no código HDL tenha um elemento mutante correspondente. Porém, nem sempre todos os elementos mutantes seriam utilizados, visto que isso seria especificado durante a fase de configuração do experimento. A ideia básica de construção de um mutante é a inserção de condicionais nos códigos fonte HDL o qual, dependendo de um sinal de controle, tomaria comportamento correto ou errado durante a execução.

Figura 2.13: Alteração da definição de componente e arquitetura em linguagem VHDL para construção de um mutante.



```
// Componente Original
entity c1 is
        port (...);
end entity;
architecture a of c1 is
...
// Código
begin
        o1 <= “00010000”;
        p1: process (...)
        begin
          s <= “00000001”;
        end process p1;
        ...
end architecture;
```

```
// Componente Mutante
entity c1 is
        port (...; Selection: in integer);
end entity;

architecture a of c1_mutante is
...
// Código
begin
        o1 <= “10000000” when
Selection = 1 else “00010000”;
        p1: process (...)
        begin
        case Selection is
        when k =>
          s <= “110000000”;
        when others =>
          s <= “00000001”;
        end case;
        end process p1;
        ...
end architecture;
```

Fonte: Figura adaptada de (BARAZA, 2008).

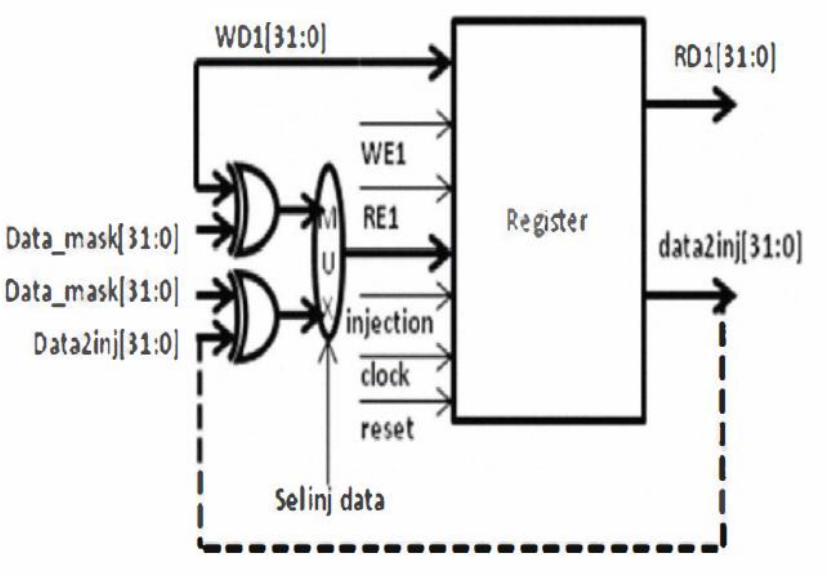
Na figura 2.13 podemos observar as alterações necessárias na arquitetura e definição de um componente em VHDL de forma que mutante possa injetar uma falha.

Com o intuito de analisar a efetividade de técnicas de tolerância a falhas por software, Azambuja (AZAMBUJA, 2012) utiliza-se de uma técnica mais simplificada de injeção, sendo ela manual. A ideia baseia-se na injeção de SEUs e SETs no código VHDL do processador por meio de simulação de hardware utilizando a ferramenta ModelSIM SE 6.6b1. SETs são injetados em sinais utilizados por lógica combinacional e SEUs são injetados nos demais. As falhas permanecem no circuito por um ciclo e meio de relógio. Isso garante que a falha possa atingir o circuito tanto na borda de subida quanto na de descida de relógio. Ao final da injeção de uma falha, o conteúdo de memória o qual deve manter o resultado da execução de uma aplicação é comparado com os resultados de uma execução livre de falhas.

Outra abordagem para emulação de SEUs em processadores descritos em HDL foi proposta por Mansour (MANSOUR, 2012). O método explora a alteração da descrição de hardware em linguagem VHDL do processador LEON3. Denominada de injeção de falha direta ou (DFI – do inglês, *Direct Fault Injection*), tal abordagem emula SEUs em *flip-flops*. Nesta proposta o autor propõe uma alteração no modelo de descrição de hardware dos registradores do processador LEON3 de uma forma genérica. Com uso de um hardware adicional ao modelo do registrador, são adicionados portas e operadores lógicos para emulação de falhas. Para o caso em que há a intenção de se injetar uma falha no mesmo momento em que a CPU está realizando um processo de escrita em um registrador, a prioridade de emulação de uma falha é mais baixa. A figura 2.14 ilustra a nova estrutura do registrador, que possibilita a emulação de SEUs com uso de uma máscara de bits Data_mask e uma porta lógica com operação ou exclusivo, ou XOR. O

processo de injeção de falhas é realizado por meio de uma máquina de estados finitos elaborada para este objetivo.

Figura 2.14: Estrutura do registrador com suporte a injeção de falhas.



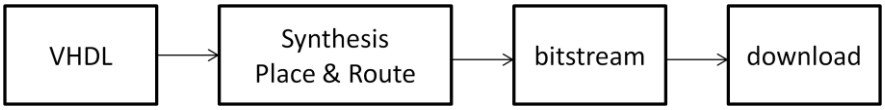
Fonte: MANSOUR (2012).

A alta controlabilidade e facilidade de análise nos experimentos com uso de injeção de falhas em modelos de VHDL é fator motivador para utilização de simuladores no desenvolvimento de técnicas de tolerância. Sem acesso a este modelo, a injeção de falhas com uso de simulação pode ficar restrita a pequenas etapas de projeto nas quais se tem controle sob a descrição de hardware, ficando para o restante a recomendação de outras técnicas de injeção de falhas. Outro fator limitante é o tempo requerido para execução dos experimentos, que é diretamente dependente do número de falhas a serem injetadas, complexidade do processador simulado e aplicação em execução.

2.7.2 Metodologia de injeção de falhas por emulação

O processo de injeção de falhas com uso de prototipação em FPGAs tem como fator primordial verificar o comportamento do hardware em situações reais. Com base nisso, é preciso a partir do código de descrição de hardware, programar a FPGA.

Figura 2.15: Fluxo de emulação de hardware em FPGA.



Em um fluxo normal de emulação de hardware em FPGA, apresentado na figura 2.15, é preciso, a partir da descrição de hardware gerar o arquivo de programação da

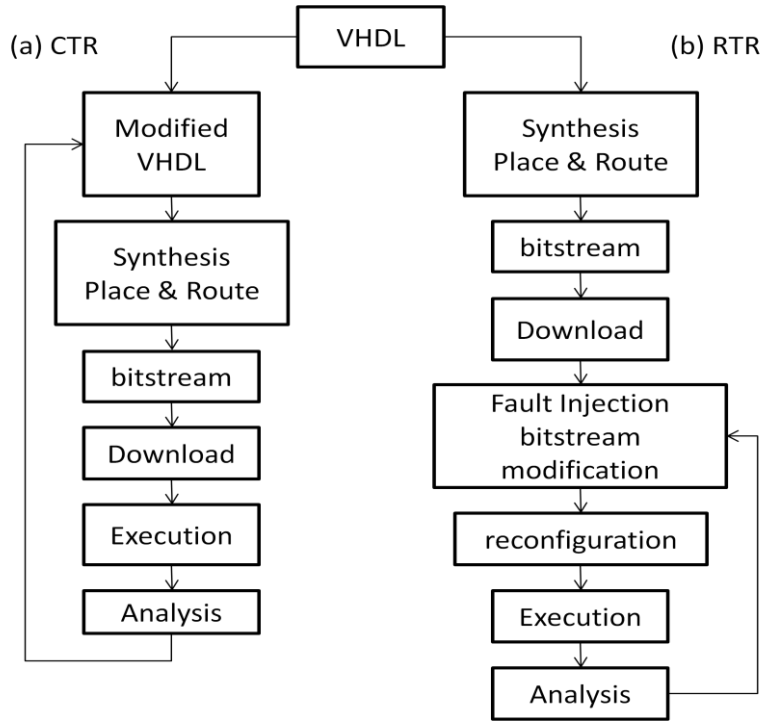
FPGA, conhecido como *bitstream*, para que a mesma assuma o comportamento descrito pelo modelo de hardware.

Suporte a injeção de falhas com FPGAs pode ser basicamente realizado através de duas abordagens, conforme Antoni (ANTONI, 2001):

- Configuração da FPGA em tempo de compilação ou *Compile-Time Reconfiguration* (CTR);
- Configuração parcial ou integral da FPGA em tempo real ou *Real-Time Reconfiguration* (RTR).

A técnica de CTR exemplifica o caso em que a FPGA é configurada uma única vez. Desta forma, um sistema operacional executando sob um processador emulado pela FPGA não será submetido a alterações do comportamento do hardware durante sua execução. No caso RTR, a FPGA pode ser reconfigurada parcialmente ou inteiramente em paralelo a execução de uma aplicação. Na busca por uma metodologia mais eficiente de injeção de falhas Antoni (ANTONI, 2001) propõe uma metodologia de injeção de falhas por meio de configuração de FPGAs sem a necessidade de alteração do código VHDL e consequente processo de síntese, posicionamento, roteamento e geração de *bitstream*. Nesta técnica a injeção de falhas baseia-se na emulação de falhas sob o próprio *bitstream*.

Figura 2.16: Fluxo de injeção de falhas em FPGA CTR (a) e RTR (b).



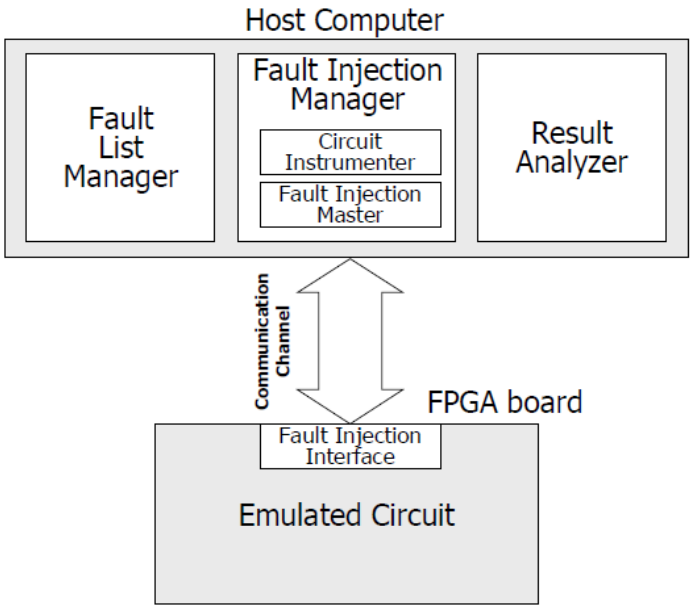
Fonte: Figura adaptada de (ANTONI, 2001).

A figura 2.16 apresenta a diferença entre as abordagens no fluxo de injeção de falhas. Para cada execução, no método CTR, ilustrado na figura 2.16 (a), é necessário regerar o *bitstream* para uma nova falha ser injetada. Após este processo, é realizada a programação da FPGA e então é executado o experimento. Ao final do experimento

ocorre o processo de análise e classificação dos erros. Após este último, existindo uma nova falha a ser injetada no sistema, o fluxo recai no processo de modificação da descrição de hardware. Podemos notar que neste fluxo de injeção de falhas o processo é lento, visto que é preciso passar pelas etapas de síntese, posicionamento e roteamento necessárias a geração do *bitstream*. No fluxo proposto por Antoni (ANTONI, 2001), duas modificações são adicionadas a metodologia. A primeira trata-se da não necessidade de modificar o código VHDL, visto que a falha é injetada no próprio *bitstream*, simbolizando uma técnica de baixo nível. A segunda modificação é o uso dos recursos presentes na maioria das FPGAs, possibilitando a reconfiguração da mesma em tempo de execução. Sendo assim, o fluxo da figura 2.16 (b), ilustra o método RTR acelerando o processo de injeção de falhas.

Tendo em vista a necessidade de reconfiguração da FPGA para cada injeção de falha, outra possível abordagem seria a utilização de modelos de hardware mutantes dentro da FPGA, semelhante ao método empregado em simulação. No entanto, tal metodologia requer mais área ocupada na FPGA. Com base nisso, Civera (CIVERA, 2001) propõe um método alternativo sem a necessidade de reconfiguração da FPGA. Entre outros aspectos também apresenta uma metodologia de melhor observação dos efeitos ocasionados no comportamento do sistema, superando em quatro ordens de grandeza o desempenho atingido pela técnica de simulação de hardware. A figura 2.17 apresenta a metodologia que foca na automatização dos experimentos e escalabilidade relacionada a complexidade do hardware emulado e número de falhas a serem injetadas. Neste trabalho o autor aborda falhas transientes, mais especificamente SEUs. O ambiente de experimentação é constituído por quatro elementos, são eles: (i) gerenciador de lista de falhas ou *Fault List Manager*, (ii) gerenciador de injeção de falhas ou *Fault Injection* Manager, (iii) analisador de resultados ou *Result Analyser* e (iv) interface de injeção de falhas ou *Fault Injection Interface*. Os três primeiros ficam localizados em um computador externo ou host a placa em que o FPGA está posicionado. Este computador controla o processo de experimentação. Através de um canal de comunicação, são enviados comandos para a interface de injeção de falhas presente no FPGA. Esta interface é um módulo adicional ao hardware emulado. Após o processo de injeção da falha a interface de injeção de falhas retorna informações de modo que o analisador de resultados classifique as falhas em relação ao comportamento observado no sistema. Todo o processo de injeção é automatizado, tornando possível seu emprego em um processo de desenvolvimento de hardware.

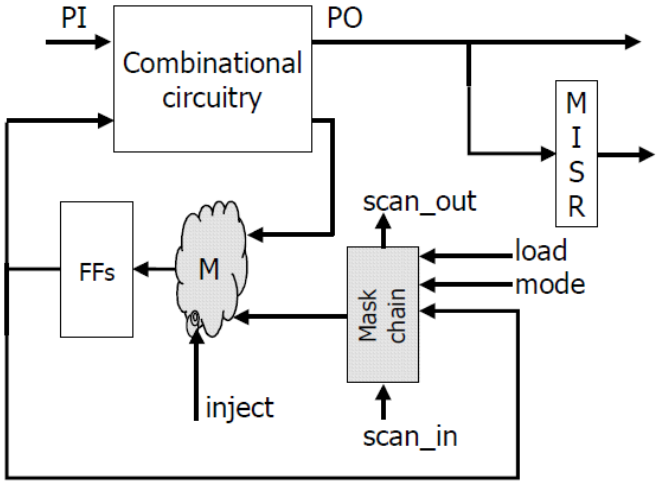
Figura 2.17: Ambiente de injeção de falhas com emulação de hardware por FPGA.



Fonte: CIVERA (2001).

Em contraste ao método de modificação de código VHDL ou emulação de *bitflips* no *bitstream* e consequente processo de reconfiguração da FPGA, a interface de injeção de falhas é baseada na alteração de valores de *flip-flops* de forma a emular falhas transientes. A figura 2.18 ilustra a instrumentação adicionada ao circuito combinacional emulado na FPGA, alvo de injeção de falhas.

Figura 2.18: Circuito de instrumentação para injeção de falhas.



Fonte: CIVERA (2001).

O princípio de funcionamento do circuito de instrumentação para injeção de falhas baseia-se no uso de um registrador chamado Mask Chain e um conjunto de sinais de

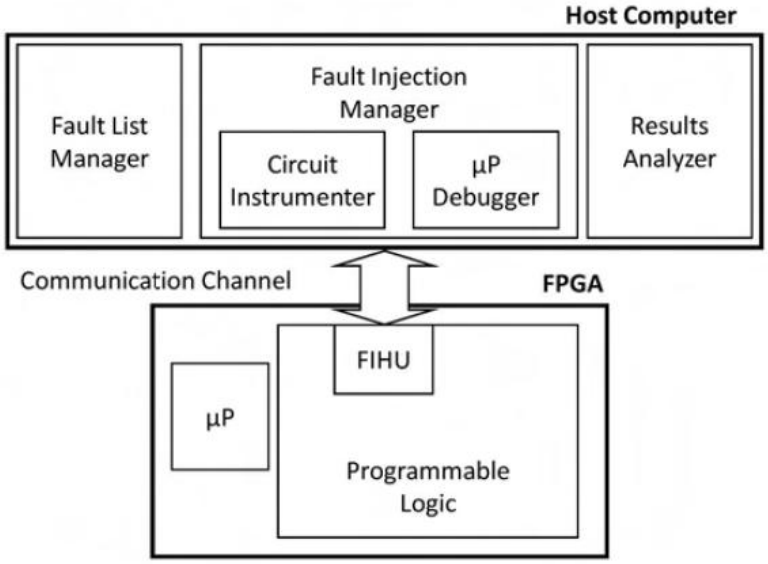
controle proveniente do host denominados INJECT, *load* e *mode*. O registrador Mask Chain é um registrador de deslocamento que armazena de forma binária o mapa de *flip-flops* alvo de injeção de falhas. Quando há um valor 1 no mapa de bits, esse valor corresponde a um flip-flop que deve ser alvo de injeção de falhas. Através do carregamento desta informação no início do experimento, com o uso do sinal SCAN_IN via deslocamento de bits, é possível em determinado instante de tempo, controlado pelo host, injetar uma falha. A aplicação da falha é realizada, por sua vez, pelo circuito combinacional M, que efetua a alteração do valor do flip-flop em questão com base no sinal INJECT controlado pelo host, complementando o valor da saída do circuito combinacional antes do mesmo ser armazenado no flip-flop. Ao final da emulação o conteúdo do flip-flop pode ser armazenado no registrador Mask Chain e enviado pelo sinal SCAN_OUT via deslocamento de bits para o host. Em posse desse valor, posteriormente é realizada a comparação e devida classificação das falhas. Podemos notar a alta controlabilidade e facilidade de coleta de informações para análise do comportamento do sistema diante da injeção de falhas.

Abate (ABATE, 2007) propõe uma metodologia de injeção de falhas que consiste em um ambiente de teste composto por um computador pessoal e uma interface para injeção de falhas em HDL e prototipada em FPGA, juntamente com o core do sistema alvo. A figura 2.19 apresenta o ambiente de emulação. Os módulos que compõem o sistema, conforme Reorda (REORDA, 2006), são:

- *Fault List Manager*, gerenciador de lista de falhas localizado no computador de controle ou *Host Computer*;
- FIHU (*Fault Injection Hardware Unit*), responsável pela injeção de falhas na CPU, reconfiguração da FPGA para emulação de falhas e observação dos efeitos após a injeção da falha;
- *Fault Injection Manager*, módulo responsável por gerenciar os experimentos através de um canal de comunicação com o FIHU e controlar o *debugger* do processador;
- *Result Analyser*, módulo responsável por analisar os resultados e gerar relatórios.

A emulação de falhas efetuada pelo módulo FIHU é semelhante à proposta por Civera (CIVERA, 2001). A técnica centraliza a emulação das falhas neste que, além da geração de falhas em *flip-flops*, trabalha em conjunto com o gerenciador de injeção de falhas, posicionado no computador de controle. A função do módulo FIHU, neste caso, é a realização da contagem do número ciclos de relógio até alcançar o valor especificado para a injeção de uma falha. Neste momento, o módulo FIHU pausa a execução do processador e dá controle ao gerenciador de injeção de falhas para utilizar primitivas do depurador, de forma a injetar uma falha. Após este procedimento, o módulo FIHU retoma a execução do processador (REORDA, 2006).

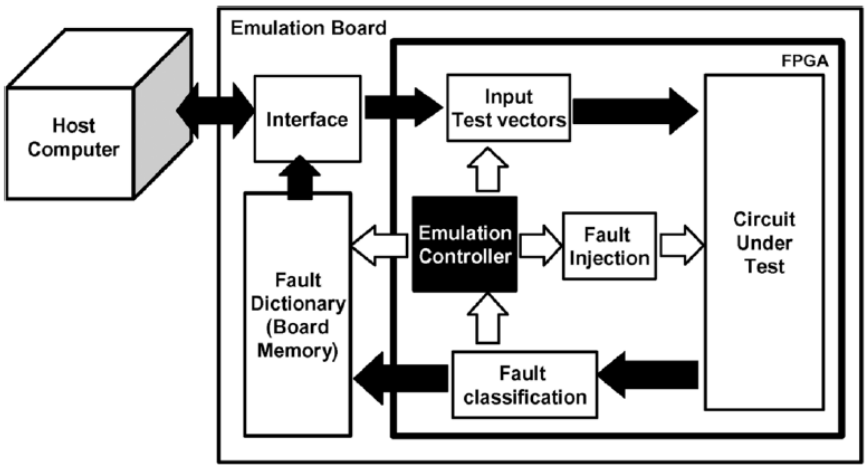
Figura 2.19: Arquitetura do ambiente de injeção de falhas.



Fonte: ABATE (2007).

Em meio ao desenvolvimento de técnicas de injeção de falhas baseada em FPGAs, López-Ongil (LÓPEZ-ONGIL, 2007) propõe um método autônomo de emulação de falhas para acelerar a análise de sistemas com tolerância a falhas. Em contraste a técnica proposta por Abate (ABATE, 2007), o autor propõe um cenário para emulação de SEU mais contido, independente do desempenho do computador pessoal, o qual não faz qualquer controle dos experimentos, como ilustrado na figura 2.20.

Figura 2.20: Sistema de injeção de falhas autônomo.



Fonte: LÓPEZ-ONGIL (2007).

O controle dos experimentos é realizado, por sua vez, pelo bloco denominado *Emulation Controller*, com base na configuração inicial realizada pelo computador de

controle. A função do bloco *Emulation Controller* se resume a seguinte sequência, para emulação de cada falha presente no vetor de testes:

1. Configuração do circuito em teste para o estado anterior a falha;
2. Injeção de uma falha;
3. Emulação da falha por um período de relógio até a classificação da falha ou finalização da execução da aplicação.

Após a injeção de todas as falhas contidas no vetor de teste, o computador de controle pode obter os resultados por meio de uma interface de comunicação entre placa de emulação e ele. O processo de injeção de falhas é realizado através da instrumentação da descrição de hardware alterando a estrutura dos *flip-flops* da arquitetura. Com base nesta técnica foi possível aperfeiçoar o processo de injeção de falhas em relação as técnicas mencionadas anteriormente, com larga vantagem em tempo de simulação, chegando a duas ordens de grandeza para 10^6 falhas por segundo.

Devido a complexidade de emulação de SETs, Entrena (ENTRENA, 2012) propõe com base no mesmo sistema autônomo de injeção de falhas proposto por López-Ongil (LÓPEZ-ONGIL, 2007), um sistema com suporte a emulação de SETs denominado *Autonomous MultiLevel Emulation System of Soft Error Evaluation* (AMUSE). Ao contrário da emulação de SEUs, a emulação de SETs é realizada em duas etapas: (i) injeção da falha e (ii) programação da mesma por alguns ciclos de relógio e devida classificação do efeito observado, após transcorrido este período de tempo. Uma falha transiente pode ser modelada com um pulso de voltagem na saída de uma porta lógica. Este pulso pode ser propagado pelo circuito ocasionando um comportamento inesperado. Para modelar a propagação da falha transiente no circuito combinacional, o circuito em teste é modelado através de duas técnicas: GL (*Gate Level*) e RTL *(Register-Transfer Level*). O primeiro modelo tem por objetivo incluir informações de atraso na propagação da falha no circuito. Já o segundo modelo, RTL, é utilizado para reduzir a ineficiência do primeiro quanto à propagação da falha por muitos ciclos de relógio. No processo de emulação é possível realizar a seleção de um modelo ou outro em tempo real. O elemento responsável por esta tomada de decisão é bloco *Emulation Controller*.

O processo de instrumentação do circuito é realizado através de ferramentas que fazem parte do próprio ambiente AMUSE, proposto pelos autores. Basicamente, as células da lista de conexões ou *netlist* original, geradas pela ferramenta de síntese de um circuito para um determinado modelo de FPGA, são substituídas por células equivalentes com base nas bibliotecas GL e RTL que contém os modelos de células que possibilitam a injeção de falhas e/ou propagação da mesma no circuito. O arquivo SDF (*Standard Delay Format*) também é analisado para extração de informações de atraso, necessárias para a configuração das células GL. Além da substituição das células da *netlist*, os sinais de controle adicionais e dados são conectados apropriadamente as entidades responsáveis, como, por exemplo, o *Emulation Controller*.

O sistema autônomo proposto pelo autor chama a atenção pela possibilidade de injeções de milhões de falhas em tempo aceitável. Cabe também salientar a possibilidade de análise detalhada do comportamento do sistema, reprodutibilidade dos experimentos, emulação de arquiteturas de processadores atualmente utilizados em projetos de hardware, e emulação de SETs em alta velocidade. No entanto, um fator desmotivador é a necessidade da descrição de hardware do processador ou hardware, utilizado para os experimentos.

2.8 Emulação de processadores versus demais abordagens

O uso de emulação de processadores como ponto de partida para uma metodologia de injeção de falhas apresenta uma redução de limitações quando comparado a outras técnicas. Devido ao fato da descrição do modelo de hardware estar disponível em software, é possível obter um alto grau de controle nos experimentos emulando *soft errors* nos mais variados elementos de memória. Em contraste as mais diversas técnicas SWIFI, esta abordagem não necessita de modificação da aplicação ou sistema que será executado. Outro aspecto relevante é a alta redução de tempo de execução dos experimentos comparado a simulação de descrição de hardware. Na tabela 2.2 é apresentada uma comparação completa da técnica de emulação de processadores em relação a outras técnicas. Os critérios de comparação são baseados nos fatores limitantes observados nas mais diversas técnicas. A medida do impacto dos fatores pode assumir os valores nenhum, alto, médio e baixo, com base em Arlat (ARLAT, 2003).

Tabela 2.2: Emulação de arquiteturas comparada a outras técnicas de injeção de falhas.

Fatores Limitantes	***Emulação de processadores por software***	***Simulação de Hardware***	***Emulação de Hardware***	***SWIFI***	***Hardware (íons pesados)***
Código HDL	Não	Sim	Sim	Não	Não
Tempo de simulação	Médio	Alto	Baixo	Alto	Baixo
Alcançabilidade	Alta	Alta	Alta	Média	Alta
Controlabilidade	Alta	Alta	Alta	Média	Baixa
Repetibilidade	Alta	Alta	Alta	Alta	Baixa
Intrusividade	Nenhuma	Nenhuma	Nenhuma	Alta	Nenhuma
Reprodutibilidade	Alta	Alta	Alta	Alta	Baixa
Medição de tempo	Alta	Alta	Alta	Média	Baixa
Eficácia	Alta	Alta	Alta	Alta	Baixa

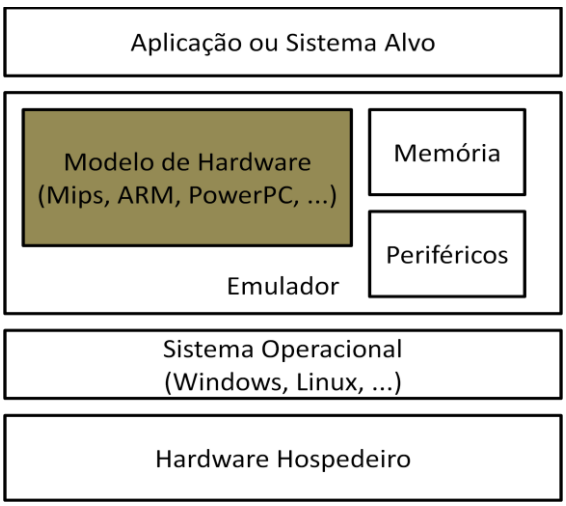
Podemos observar que a técnica de emulação de hardware com uso de prototipação em FPGAs possui melhor avaliação dentre todas as técnicas. O alto grau de controlabilidade, reprodutibilidade, alcançabilidade e muitos outros aspectos, evidencia algumas vantagens em relação a outras técnicas, tornando-a muito atrativa para execução de experimentos de injeção de falhas. No entanto, a necessidade da descrição de hardware do processador é um fator de alta limitação. Em contrapartida, a emulação de processadores por software parece apresentar boas vantagens em relação a outras técnicas. Isso se deve ao fato de apresentar uma maior flexibilidade e facilidade de uso devido ao acesso de todos os recursos por meio do código fonte. Outro aspecto a

destacar é a velocidade de emulação, reduzida em relação a outras técnicas em função do uso da tradução dinâmica presente em ambientes de virtualização.

3 PLATAFORMAS DE EMULAÇÃO DE PROCESSADORES

O uso de emuladores de arquiteturas por parte de programadores é uma realidade no desenvolvimento de aplicações. Um caso típico é o emulador do sistema Android, muito utilizado em telefones inteligentes. Estes emuladores permitem que desenvolvedores possam elaborar aplicações para dispositivos sem mesmo ter contato com o hardware real (GOOGLE INC., 2014). Sua grande flexibilidade varia desde a emulação de processadores como ARMv5, com suporte a unidade de gerenciamento de memória, até a configuração de botões e interfaces utilizadas pelas aplicações. Fabricantes de processadores do estado da arte, como Freescale, exemplificam essa realidade através do suporte a família de processadores QorlQ no emulador QEMU, para sua utilização em virtualização (MARGINEAN, 2013). Com base nisso, a validação de mecanismos de proteção ou tolerância a falhas pode ser realizada com o uso desses modelos, ou seja, com emulação de arquiteturas de processadores. Para tal é necessário ter disponível o código fonte do emulador, e desenvolver um modelo de injeção de falhas de forma a emular falhas transientes e permanentes.

Figura 3.1: Estrutura de um emulador de processador



Diante deste contexto, pesquisadores tem direcionado seus estudos para o uso desta nova abordagem (POTYRA, 2007; CABODI, 2010; XU, 2012; YI, 2013). Com base nisso, o processo de validação de mecanismos de tolerância a falhas é efetuado com o uso da estrutura apresentada na figura 3.1. Nesta uma aplicação ou sistema operacional, com instruções para uma arquitetura alvo, pode executar em outro

processador de diferente arquitetura, através de um emulador, que se trata de uma aplicação executando em um determinado sistema operacional. Para tal, é necessário o suporte a esta arquitetura através em um modelo hardware presente no emulador. Dessa forma, a aplicação ou sistema operacional executando no emulador pode utilizar-se de recursos de hardware, semelhantes aos originais que terá acesso no hardware real.

Cabe salientar que o desempenho de emulação está associado ao hardware hospedeiro e a qualidade do software do emulador. Sendo assim, em muitas situações a emulação de processadores para injeção de falhas pode ter vantagens em relação a outras técnicas quando considerando o tempo requerido para injeção de um determinado número de falhas, entre outros fatores.

3.1 Trabalhos Relacionados

3.1.1 Injetor de Falhas baseado no emulador FauMachine

Um framework para injeção de falhas e devida verificação de mecanismos de tolerância é apresentado por Potyra (POTYRA, 2007), o qual é baseado no projeto FAUmachine. Com o uso de tal ferramenta, que se trata de um emulador da arquitetura Intel, é possível em fases iniciais de desenvolvimento validar mecanismos de tolerância falhas desenvolvidos em software, bem como observação dos efeitos no comportamento do sistema em teste, no caso sistemas operacionais ou aplicações. A não necessidade de alteração do software em teste e possibilidade de avaliação de um sistema mais complexo, tais como sistemas operacionais, apresentou grandes avanços em relação às técnicas de injeção de falhas baseadas em SWIFI. Além disso, um completo modelo de falhas é disponibilizado pelo *framework*, cobrindo injeção de falhas permanentes e transientes em elementos de memória que compõem todo o sistema, tais como registradores, posições de memória, erros de acesso a periféricos, entre outros. Outro aspecto ressaltado é a possibilidade de automatizar testes e reproduzir experimentos, através do uso de *scripts* para simulação.

3.1.2 Sistema integrado de injeção de Falhas

A proposta de um framework para injeção de falhas independente de arquitetura de hardware é sugerida por Cabodi (CABODI, 2010), com base no uso de recursos do sistema operacional Linux para emulação de SEUs. A validação desta metodologia é realizada através do uso do hardware real, bem como do uso de emulação de processadores. Um aspecto destacado nesta abordagem é a possibilidade de injeção de falhas em sistemas de tempo real. Outro aspecto a salientar é a baixa intrusividade quanto ao suporte em software para injeção de falhas, sendo que a técnica não necessita parar a execução do Kernel do Linux para injeção de uma falha.

3.1.3 Injeção de falhas para auto teste

Com o objetivo de desenvolver uma ferramenta para validação de mecanismos de detecção de falhas para auto teste (BIT – do inglês, *Built-in Test*), Xu (XU, 2012) propõe uma técnica de injeção de falhas baseada no emulador QEMU. A técnica baseia-se na injeção de falhas na memória RAM, a qual é emulada pela máquina, com modificações no software desta. Uma evolução deste trabalho, um injetor de falhas

chamado BitVaSim, que cobre injeção de falhas em mais elementos de memória da máquina, foi proposto por Yi (YI, 2013). O objetivo do trabalho foi gerar exceções no processador, através de alterações do código fonte da máquina, para verificar o comportamento do sistema operacional em execução.

3.2 Tradução Dinâmica

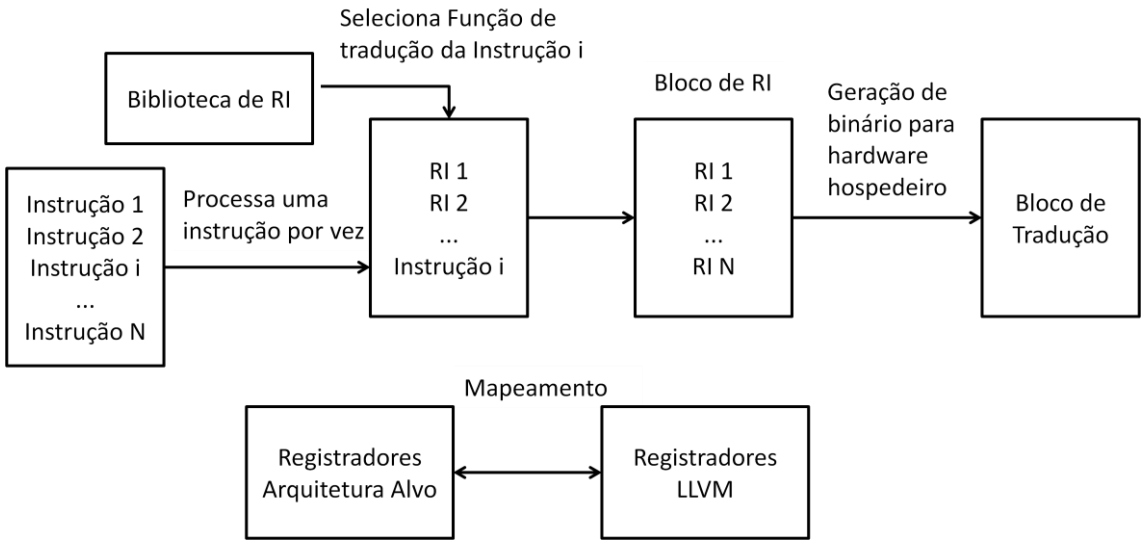
O alto desempenho de emuladores de processadores atualmente disponíveis para o desenvolvimento de aplicações e virtualização se deve ao uso da tradução dinâmica de código binário. Tal processo consiste na analise do código binário para um determinado processador de forma que este seja executado em outro processador, no caso o hardware hospedeiro. Aplicações como QEMU e VMware (NYSE) utilizam-se deste mecanismo para disponibilizar um ambiente de emulação próximo do real, como se o software estivesse executando na plataforma real de hardware.

3.2.1 Processo de tradução de instruções

O processo de tradução dinâmica de código binário é efetuado com o uso de uma representação intermediária (RI), que independe da arquitetura do processador. Ou seja, o conjunto de instruções para uma determinada arquitetura é traduzido para um conjunto de operações. A partir desta representação é gerado código binário para execução no hardware hospedeiro.

Um exemplo de representação intermediária, chamada LLVM (*Low Level Virtual Machine*), juntamente com sua infraestrutura de compilação, é utilizada por Hsu (HSU, 2011) para construção de um ambiente de emulação mais eficiente que o disponibilizado pelo tradutor dinâmico do emulador QEMU. Para tal, foi necessário modificar o emulador para dar suporte ao tradutor dinâmico LLVM.

Figura 3.2: Processo de tradução dinâmica de instruções.



Fonte: Figura adaptada de (HSU, 2011).

O processo de tradução de código binário pode é ilustrado na figura 3.2. Nela podemos observar que cada uma das instruções para o sistema alvo é processada uma a uma e convertida em uma representação LLVM, denominada RI. A instrução 1, por exemplo, é traduzida para a representação RI 1, com base em uma biblioteca de conversão. O procedimento de tradução permanece até que uma instrução determine que a sequência seja interrompida, tais como instruções de desvios condicionais, entre outras, caracterizando o fim de um bloco de tradução. Dessa forma, após uma sequência de tradução, é gerado um bloco de tradução, o qual será compilado em tempo real para execução no hardware hospedeiro. Neste processo de compilação, em muitos casos, podem ocorrer otimizações por parte do compilador para tornar o código traduzido mais eficiente para execução, aumentando o desempenho de emulação. Outro fator envolvido no processo de tradução é o mapeamento dos registradores da arquitetura alvo em registradores LLVM, de forma que durante as execuções os registradores da arquitetura alvo sejam acessados de forma mais eficiente.

3.2.2 Representação Intermediária

A representação intermediária de código é realizada de forma fixa, respeitando regras predefinidas para cada arquitetura. Desta forma, quando o tradutor dinâmico se depara com uma instrução de uma arquitetura, o mesmo consulta uma biblioteca de tradução e cria um conjunto de operações que representam a instrução analisada. Sendo assim, uma instrução para uma determinada arquitetura pode ser dividida em duas ou mais operações.

Figura 3.3: Processo de tradução dinâmica de instruções.

```
define { [16 x i32] }* @block_test({ [16 x i32] }*) {
entry:
    %tmp1 = getelementptr { [16 x i32] }* %0, i32 0, i32 0, i32 0
    %tmp2 = getelementptr { [16 x i32] }* %0, i32 0, i32 0, i32 2
    %tmp3 = load i32* %tmp2
    store i32 %tmp3, i32* %tmp1
    ret { [16 x i32] }* %0
}
```

Fonte: Figura adaptada de (JEFERRY, 2009).

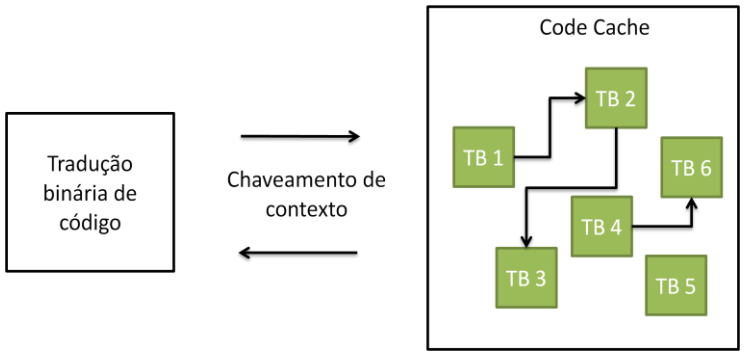
Um exemplo para instrução mov R2, R0 é apresentado na figura 3.3. Nela a a representação intermediária LLVM equivale a instrução mov, conforme Jeferry (JEFERRY, 2009). Nesta representação, as variáveis %tmp1 e %tmp2 mantém ponteiros para os registradores R0 e R2, respectivamente. Após este procedimento, %tmp3 recebe o valor do registrador R2 através de uma operação *load*, com o uso do ponteiro %tmp2. Por fim, R0 recebe o valor de %tmp3 via a operação *store*, com a utilização do ponteiro %tmp1, finalizando a sequência de operações para execução da instrução mov.

Ao final do processo de geração de uma representação intermediária, é gerado o código nativo para o conjunto de instruções do sistema hospedeiro, através do processo de compilação.

3.2.3 Dinâmica de Execução de Código Binário

Em um processo de emulação, os blocos traduzidos são armazenados em uma região de memória denominada *code cache*. Esta região simboliza um contexto em que os blocos são executados e também possibilita a reutilização destes, com o intuito de acelerar a emulação. Quando o emulador opta por executar os blocos traduzidos (TBs - do inglês, *Translation Blocks*), o contexto é trocado para esta região de *cache*. Após a execução de um bloco, ocorre a volta para o contexto do tradutor novamente, e o mesmo continua o processo de tradução. Essa troca entre contextos é ilustrada na figura 3.4.

Figura 3.4: Processo de tradução dinâmica de instruções.



Na figura 3.4 é ilustrado também o processo interconexão entre blocos, caracterizando cadeias de execução de forma a acelerar a emulação, não sendo necessário o chaveamento entre os dois contextos todo o tempo. A volta para o contexto de tradução de instruções pode ocorrer ou por imposição do emulador, ou por um bloco não possuir outro elemento interligado, finalizando assim a cadeia de execução automática.

3.3 Plataforma de Emulação de Processadores QEMU

O emulador QEMU é uma plataforma de emulação de processadores capaz de executar sistemas operacionais e aplicações sem a necessidade de modificação destes. O processo de virtualização permite que o sistema ou aplicação utiliza-se de uma memória física provida pela máquina, entre outros recursos de hardware. Na sua versão 1.6.0 a mesma provê suporte a diversas arquiteturas tais como MIPS, ARM, SPARC, PowerPC, X86, X86_64 e etc.

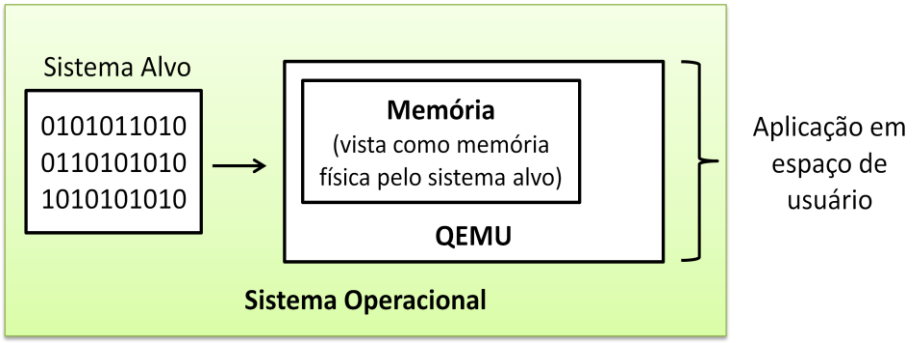
A máquina QEMU tem sido largamente utilizada para predizer o comportamento de sistemas na literatura. Outro aspecto a destacar é seu uso como plataforma de emulação de processadores de tecnologias como Freescale, entre outros. Sendo um projeto de código fonte aberto, e com base na sua estrutura modular, ela permite a adição de novas arquiteturas e suporte a novos processadores de forma relativamente simples. Seu principio básico de operação é baseado na tradução dinâmica de código

binário e armazenamento de código já traduzido para uso futuro, aumentando o desempenho de emulação.

3.3.1 Estrutura de execução da máquina QEMU

O emulador QEMU é um processo em nível de espaço de usuário em um sistema operacional como qualquer outro. Todo e qualquer recurso de hardware disponibilizado por ele, e visto pelo sistema operacional ou aplicação, é emulado por software.

Figura 3.5: Aplicação QEMU como processo em um sistema operacional.



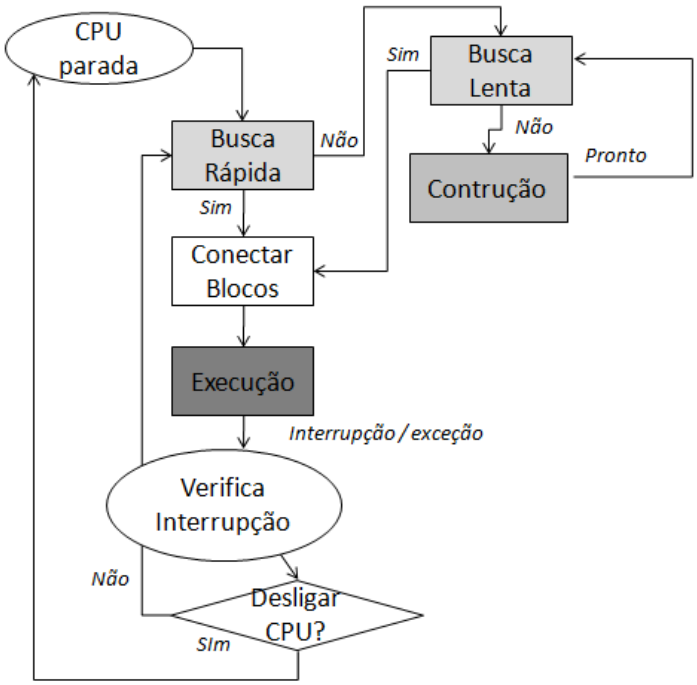
A figura 3.5 exibe esta estrutura como um processo de um sistema operacional. O sistema alvo é representado como uma imagem em código binário para uma arquitetura específica, sendo ela um arquivo de entrada para o emulador.

A escolha pela arquitetura a ser emulada e suporte a gerenciamento de memória são parâmetros especificados em tempo de compilação. Dentre muitas opções de execução, o emulador disponibiliza um conjunto de parâmetros para seleção de um processador específico e variação da arquitetura selecionada. Outra possível configuração é a quantidade de memória física que será vista pelo sistema em execução. Desta forma, é possível executar instruções presentes nos mais diversos modelos de processadores suportados no emulador.

3.3.2 Fluxo de execução do emulador

O fluxo de execução do emulador QEMU pode ser visualizado na figura 3.6. Nele são apresentados os blocos principais para emulação de um processador. O processo de emulação inicia com a CPU em estado parado, denominado **CPU parada**. Após iniciar o processo de leitura das instruções para a arquitetura alvo, o fluxo recai na busca pelo próximo bloco traduzido a ser executado, denominada **Busca Rápida**. Nesta busca, o apontador de programa é utilizado como parâmetro. Caso o bloco não seja encontrado, a busca recai na **Busca Lenta**. Nesta etapa, caso o bloco traduzido não seja encontrado, o processo de tradução dinâmica entra em ação, gerando um novo bloco conforme o apontador de programa. Após o bloco ser encontrado ou construído, é verificado se é possível conectá-lo ao bloco anterior, possibilitando assim uma execução em cadeia dos blocos traduzidos, sem a necessidade do processo de busca novamente.

Figura 3.6: Fluxo de execução do emulador QEMU.



O processo de execução dos blocos é realizado através do bloco **Execução**. Nesta etapa o contexto de execução modifica, sendo o mesmo transferido para o contexto de execução de blocos. A volta deste contexto é realizada quando não há um próximo bloco traduzido a ser executado. Isto se deve a blocos que não estejam conectados a outros ou por uma requisição do próprio emulador. Este último caso ocorrendo quando há uma interrupção ou exceção. Sendo assim, a execução de blocos traduzidos pode ser interrompida.

3.3.3 Tradutor dinâmico TCG

O tradutor dinâmico presente nativamente no emulador QEMU realiza o processo de tradução de instruções através do uso de variáveis temporárias.

Figura 3.7: Representação intermediária da instrução addi.

```
addi r1,r1,-16    # r1 = r1 - 16

movl_T0_r1           # T0 = r1
addl_T0_im -16       # T0 = T0 - 16
movl_r1_T0           # r1 = T0
```

Fonte: Figura adaptada de BELLARD (2005).

A figura 3.7 apresenta este mecanismo, em que a instrução addi é representada por três micro-operações com a variável temporária T0 (BELLARD, 2005).

A micro-operação movl_T0_r1 pode ser codificada conforme a figura 3.8. Podemos observar que a variável temporária T0 recebe o valor do registrador r1, que é mantido em uma estrutura de dados que descreve o estado da CPU no processo de emulação. Dependendo da arquitetura emulada, a estrutura de estado da CPU, descrita na máquina, mantém os valores necessários conforme o modelo de hardware correspondente. Com base nisso, para a arquitetura MIPS, o vetor *regs* contem 32 elementos correspondendo aos registradores da arquitetura.

Figura 3.8: Descrição da operação movl_T0_r1 em linguagem C.

```
void movl_T0_r1(void)
{
    T0 = env->regs[1];
}
```

Fonte: BELLARD (2005).

Ao final do processo de decodificação de um conjunto de instruções de código de máquina para a arquitetura alvo, é gerado um bloco de tradução. Após a composição do bloco é realizado o processo de compilação deste gerando código nativo para ser executado no sistema hospedeiro. Dessa forma, o código nativo para o sistema hospedeiro é semanticamente equivalente ao bloco original para a arquitetura emulada. Assim, dependendo das instruções contidas em um bloco, estas podem realizar operações utilizando-se dos registradores contidos na estrutura de estado da CPU.

Ao final de cada processo de tradução, o código traduzido e compilado é posicionado em uma região de memória chamada *code cache*. Esta memória faz parte do contexto em que os blocos traduzidos são executados. Estes são mantidos nesta região de forma que sejam reutilizados para acelerar o processo de emulação. Outro aspecto a mencionar é a possibilidade de conexão entre blocos traduzidos. Dessa forma, quando um bloco finaliza sua execução, é procurado o próximo bloco conectado dentro do contexto, aumentando o desempenho de execução do sistema emulado.

4 PROPOSTA DE METODOLOGIA DE INJEÇÃO DE FALHAS

O desenvolvimento de ferramentas e técnicas de injeção de falhas para emulação de radiação tem apresentado diversos avanços, mas também limitações. As diferentes abordagens, além de suas vantagens, também, por vezes, trazem dificuldades na sua implantação para análise de mecanismos de tolerância. Fatores como necessidade de alteração do código da aplicação que executa no sistema em teste, dificuldade em termos de controlabilidade e/ou reprodutibilidade dos experimentos, e necessidade da descrição de hardware para simulação ou utilização de FPGAs, são alguns dos obstáculos apresentados. Além disso, aspectos referentes ao custo de projeto também podem impactar negativamente no uso de certas técnicas, se não em todo o projeto, mas pelo menos em grande parte dele. Limitações apresentadas por metodologias de injeção de falhas baseadas em simulação deixam evidente o crescimento do tempo necessário para execução dos experimentos em função da complexidade da aplicação. Sendo assim, barreiras como essas abrem um novo espaço para o desenvolvimento de metodologias alternativas às tradicionais.

Este capítulo tem por finalidade apresentar uma proposta de metodologia de injeção de falhas baseada em emulação de processadores, para análise de sistemas complexos. Seu principal foco é reduzir limitações conhecidas em outras técnicas e facilitar a análise da suscetibilidade de uma arquitetura de hardware e consequente impacto em sistemas operacionais e/ou aplicações que executam nesta. Visando acelerar o processo de injeção de falhas por meio de emulação de processadores por software, esta metodologia tem por finalidade facilitar o desenvolvimento e análise de algoritmos de tolerância a falhas guiando o desenvolvedor no processo de construção do software.

A estruturação da metodologia de injeção de falhas terá como primeiro tópico uma discussão sobre a emulação de arquiteturas e sua potencialidade para injeção de falhas. Como segundo tópico será apresentado um completo modelo de falhas aplicado a emulação de arquiteturas. Em seguida, o terceiro tópico tratará sobre o fluxo de injeção de falhas durante o processo de emulação. O quarto tópico discutirá a potencialidade de utilização da metodologia para análise de sistemas complexos, como sistemas operacionais, juntamente com sua aplicação no desenvolvimento de mecanismos de proteção para estes. E por fim, o quinto tópico apresentará a aplicação da proposta no emulador de processadores QEMU.

4.1 Redução de Limitações

O uso de emulação de processadores nesta metodologia de injeção de falhas vislumbra a redução de limitações quando comparado a outras técnicas. Devido ao fato

do modelo da arquitetura estar disponível em software, é possível obter um alto grau de controle nos experimentos, por meio da alcançabilidade proporcionada pelo código fonte do emulador. Através deste, é possível emular erros de software nos mais variados elementos de memória. Em contraste as mais diversas técnicas SWIFI, esta abordagem não necessita de modificação da aplicação que será executado como alvo de falhas. Outro aspecto relevante é a alta redução de tempo de execução dos experimentos comparado a simulação de descrição de hardware por meio de ferramentas como ModelSIM.

A validação de software para processadores do estado da arte também torna atrativa tal abordagem, sendo que a descrição de hardware de dispositivos COTS não é normalmente disponibilizada pelos fabricantes. No entanto, com a crescente difusão de modelos em software para emuladores de processadores é possível aumentar o uso destes processadores desprovidos dedicados a tolerar radiação.

4.2 Modelo de Falhas

A injeção de falhas e emulação de erros de software pode ser efetuada agindo nos elementos de memória que representam instâncias importantes do hardware. Nesta seção serão apresentados estes modelos e os mecanismos para emulação de erros. Um método de classificação das falhas observadas será apresentado não limitando sua extensão, dependendo do tipo de análise a qual se queira vislumbrar.

4.2.1 Elementos de Memória

No contexto de máquinas virtuais, os modelos arquiteturais em software definem elementos de memória que são utilizados no processo de emulação. Os registradores acessíveis por meio do conjunto de instruções da arquitetura, os registradores utilizados indiretamente, a unidade de gerenciamento de memória, os vetores de interrupção e as interfaces de depuração, são algumas destas representações. De forma que tais estruturas sejam acessadas e utilizadas na emulação, normalmente emuladores mantém uma estrutura de estado da CPU a qual agrupa estes elementos (ALEXANDER, 2012). Além da CPU, outros componentes são definidos como elementos de memória, tais como a memória RAM e dispositivos de entrada e saída. Sendo os elementos de memória componentes fundamentais no processo de emulação, estes podem ser alvo de erros de software entre outros tipos de falhas no processo de injeção.

4.2.2 Alvos e parametrização

A definição de uma falha é realizada com o uso de um conjunto de atributos, como apresentado na tabela 4.1. O primeiro atributo, denominado Context, define a localização do contexto em que uma falha é injetada. Em sistemas complexos basicamente temos dois contextos: (a) *bootloader*, que prepara o processo de inicialização de um sistema, configurando o processador e alguns dispositivos de hardware e o (b) sistema em si, que representa a aplicação alvo que executa no hardware. Na metodologia proposta, a especificação do contexto da falha determina o nível de análise que se quer realizar. São definidos dois níveis, o primeiro está relacionado a falhas injetadas no contexto de *bootloader*, e o segundo relacionado a falhas injetadas somente no contexto do sistema operacional ou aplicação. Em nível de *bootloader* é possível investigar a influência que as falhas provocam no sistema tão bem

como nos recursos de hardware utilizados, quando, devido aos efeitos da radiação, ocorrem erros no processo de inicialização. Em nível de sistema, é somente investigado o comportamento da aplicação alvo. O segundo atributo, denominado Fault_type, determina o tipo de falha com base no elemento de memória alvo. Registradores, posições de memória física e comunicação de dispositivos de entrada e saída estão neste grupo. Dependendo do tipo de falha, somente alguns atributos são necessários para a a representação.

Tabela 4.1: Atributos para definição de uma falha.

Atributos	***Descrição***
Context	*Bootloader* ou sistema.
Fault_type	Registradores, memória e dispositivos de entrada e saída.
Register	Identificador do registrador do processador emulado.
Phys_addr	Posição de memória física.
Time_interval	Ciclo de relógio inicial e final para representação de uma falha transiente ou intermitente.
Time	Ciclo de relógio para injeção de uma falha.
Bit_pos	Bit de um byte do elemento de memória alvo de falhas.
Bit_val	Valor lógico 0 ou 1 assumido pelo atributo **Bit_pos**.
Duration	Duração da falha, podendo assumir os tipos permanente, intermitente, ou transiente.

Nas subseções seguintes são apresentados os modelos de falhas e devida utilização dos atributos da tabela 4.1. Além disso, são apresentados elementos fundamentais a metodologia, de forma que o processo de injeção de falhas reduza as limitações presentes em outras técnicas.

4.2.2.1 Referência de tempo

Diante da necessidade de auxiliar nos estudos de construção e avaliação de mecanismos de tolerância a falhas, baseados em software, surgem como aspectos fundamentais a repetibilidade e reprodutibilidade dos experimentos, para devida verificação de uma solução proposta. Para prover tal requisito, se faz necessário que a simulação tenha uma referência de tempo, no caso ciclos de relógio. Para experimentos comparativos e alguns tipos de benchmarks, esta referência torna-se uma característica muito importante. Com base nisso, a modelagem prevê a definição de tempo para o processo de injeção de falhas. Na tabela 4.1 podemos observar que são definidos atributos de tempo, tais como Time_interval e Time. Estes atributos representam unidades de ciclos de relógio. Considerando tal premissa, para uma falha F ser aplicada no ciclo de relógio Cn, a contagem de ciclos de relógio do processador deve atingir o

valor especificado. A partir do momento em que é iniciado o processo de simulação, o tempo inicial é definido como C0. Dependendo dos atributos que definem uma falha, o atributo Time_interval ou Time é utilizado. O primeiro é normalmente atribuído a falhas intermitentes e o segundo a emulação de uma falha transiente, como um *bitflip*, por exemplo.

4.2.2.2 Memória

Uma falha na memória é parametrizada por uma tripla composta por Duration, Phys_addr e Bit_pos. O atributo Duration define se a falha é do tipo permanente, intermitente ou transiente. Na tabela 4.2 são apresentados os subtipos de falhas para cada uma das falhas e a modelagem em termos de alteração do conteúdo do elemento de memória alvo de uma falha. O elemento de memória é especificado pelo atributo Phys_addr, caracterizando uma posição de endereço da memória RAM. Falhas permanentes estão associadas ao acesso a estes elementos de memória e são consequentemente independentes de tempo. Um acesso é caracterizado por uma operação de leitura ou escrita, realizada pelo processador.

Tabela 4.2: Modelo de falhas para memória.

Tipo	**Subtipo**	**Modelagem**
	Escrita	
Permanente	Acoplamento	Mem(addr) = tmp Mem(addrX) = tmp
	Stuck-at-0	Mem(addr) = tmp AND 0xFFFDFFFF
	Stuck-at-1	Mem(addr) = tmp OR 0x80000000
Transiente/ Intermitente	Bit-flips	Mem(addr) = Mem(addr) XOR 0x00000010 Mem(addr) = tmp
MMU	Decodicação	Mem(addrY) = tmp
	Leitura	
Permanente	Acoplamento	tmp = Mem(addr) tmp = Mem(addrX)
	Stuck-at-0	tmp = Mem(addr) AND 0xFFFDFFFF
	Stuck-at-1	value = Mem(addr) OR 0x80000000
Transiente/ Intermitente	Bit-flips	Mem(addr) = Mem(addr) XOR 0x00000010 tmp = Mem(addr)
MMU	Decodificação	tmp = Mem(addrY)

As falhas de bit-flip se enquadram nos tipos transiente ou intermitente, sendo modeladas por uma operação lógica XOR entre o elemento de memória e uma máscara de bits. Esta máscara de bits é composta pelo atributo Bit_pos, que indica o bit a ser alterado. É importante salientar que um *bit-flip* pode ser emulado antes ou depois da leitura ou escrita sob o elemento de memória. Outros tipos de operações lógicas para modelagem de uma falha podem também ser visualizados na tabela 4.2. Nela, pode ser visualizada a modelagem para falhas do tipo *stuck-at*, que é definida como uma operação lógica AND ou OR, aplicada entre o elemento de memória alvo de falha e uma máscara de bits. Falhas permanentes contemplam o caso de falhas de acoplamento ocasionando acessos a posições de memória adjacentes. Neste caso, para procedimentos de leitura, a leitura de uma posição errada de memória pode ocorrer, e para o caso de escrita, mais de uma posição de memória pode ser acessada e o valor modificado.

Por fim, temos a falha do tipo MMU, relativa à unidade de gerenciamento de memória. Esta está relacionada ao processo de tradução e seleção da posição em que uma operação é realizada. Um exemplo pode ser dado para o caso de um acesso a uma posição de memória de endereço addrX. Para este acesso, como primeira etapa, ocorre a decodificação da instrução do processador e devida busca pela célula de memória alvo de acesso. No caso de uma falha nesta decodificação e tradução de endereço, pode ocorrer a seleção da posição addrY. Tal situação acaba gerando uma leitura ou escrita sob uma posição de memória incorreta.

4.2.2.3 Registradores

A modelagem de falhas em registradores se assemelha ao modelo de injeção de falhas em memórias. A seleção do registrador alvo de falhas é realizada através do atributo Register, que determina o registrador do processador emulado. O tipo de falha neste caso pode ser permanente, transiente ou intermitente, como no caso do modelo de falhas na memória.

Tabela 4.3: Modelo de falhas para memória.

Tipo	***Subtipo***	***Modelagem***
Escrita		
Permanente	Stuck-at-0	R12 = tmp AND 0xFFFDFFFF
	Stuck-at-1	R12 = tmp OR 0x80000000
Transiente/ Intermitente	Bit-flips	R12 = R12 XOR 0x00000010 R12 = tmp
Leitura		
Permanente	Stuck-at-0	tmp = R12 AND 0xFFFDFFFF
	Stuck-at-1	tmp = R12 OR 0x80000000
Transiente/ Intermitente	Bit-flips	R12 = R12 XOR 0x00000010 tmp = R12

A tabela 4.3 resume os tipos e subtipos de falhas associadas a registradores e referente modelo para alteração destes elementos de memória. As operações lógicas para emulação de falhas são exemplificadas pela aplicação no registrador 12, denominado R12, de um processador emulado.

Este modelo de falhas pode ser aplicado a qualquer um dos registradores descritos na arquitetura, acessíveis via conjunto de instruções ou não.

4.2.2.4 Dispositivos de entrada e saída

A emulação de falhas em dispositivos de entrada e saída é especificada pelos atributos Fault_type e Phys_addr. Normalmente o acesso a dispositivos de hardware é realizado através de mapeamento em memória. O sistema operacional ou aplicação acessa estas posições de memória para comunicação com o dispositivo. Outro tipo de acesso é a utilização de interfaces presentes no próprio processador. Alguns tipos típicos presentes são I2C, SPI e etc. Para utilização deste último, registradores específicos do processador são acessados. Para emular falhas em nível de dispositivos, é preciso realizar a injeção de falhas nestes elementos de memória. Os elementos alvo nesta categoria são:

- Posições de memória mapeadas para acesso a dispositivos;
- Registradores que permitem comunicação com dispositivos, sendo o parâmetro Register utilizado nesta situação.

4.2.2.5 Rotinas de Injeção de Falhas

A injeção de falhas age sobre os elementos de memória fora do contexto de execução dos blocos de tradução. Para isso, se faz necessário o uso de rotinas para

devida injeção das falhas no emulador de processadores. Na figura 4.1 são apresentadas as rotinas que permitem aplicar o modelo de falhas durante a emulação. Falhas transientes ou intermitentes são emuladas através do uso de rotinas que produzem *bitflips* em elementos de memória. Estas podem ser aplicadas a registradores e a memória em função do tempo. Para falhas relacionadas a acessos, tais como operações com registradores, memória, dispositivos de entrada e saída e utilização da MMU, as demais rotinas são aplicáveis.

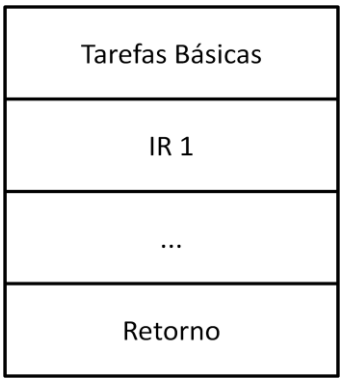
Figura 4.1: Protótipos de rotinas para injeção de falhas no ambiente de emulação de arquiteturas.

```
/* Registers */
FaultInjectionModuleRegBitFlip ( uint32_t  *register, uint32_t bit_pos, bool val );
FaultInjectionModuleRegAccess ( uint32_t  *register, uint32_t bit_pos, bool val );
/* Memory */
FaultInjectionModuleMemBitFlip ( uint64_t *physAddr, uint32_t bit_pos, bool val );
FaultInjectionModuleMemAccess ( uint64_t *physAddr );
/* I/O Devices */
FaultInjectionModuleIODevicesMemAccess ( uint64_t *physAddr );
/* MMU */
FaultInjectionModuleMMUDecode ( uint64_t *virtAddr );
```

4.2.3 Modificações no processo de tradução dinâmica

A estrutura de um bloco de tradução, que armazena um conjunto de representações intermediárias, é ilustrada na figura 4.2. No processo de execução de um bloco de tradução são primeiramente realizadas verificações pelo conjunto de instruções iniciais, denominado **Tarefas Básicas**. Nesta etapa é determinado se há uma ação para abortar a execução do bloco, ocorrendo assim o retorno do contexto de execução deste. A decisão é baseada em informações presentes na estrutura de estado do processador, acessíveis pelo bloco de tradução. Caso não haja a necessidade de abortar, o fluxo segue a execução habitual. Ao final desta, a etapa final verifica se há um próximo bloco a ser executado. Caso não haja, o contexto de emulação é retornado para o contexto de tradução dinâmica.

Figura 4.2: Estrutura básica de um bloco traduzido.



Para proporcionar controlabilidade e repetibilidade no processo de injeção de falhas, a metodologia define modificações na estrutura dos blocos de tradução, são elas:

1. Contagem do número de ciclos de relógio;
2. Desvio condicional para determinação de fim da simulação;
3. Desvio condicional para interromper a execução do bloco de tradução;
4. Contagem do número de instruções do processador emulado executadas em um bloco;
5. Limitação do número de instruções do processador emulado posicionadas em um bloco.

A contagem do número de ciclos de relógio define uma referência de tempo no processo de simulação, sendo esta alimentada nos contextos de execução dos blocos de tradução e no contexto de tradução de instruções. Com base neste, valor é possível interromper a execução de um bloco em um momento de interesse. Tal situação pode ocorrer para determinar o fim da simulação, exemplificando o uso do primeiro desvio condicional, ou para indicar que um determinado tempo foi atingido, exemplificando o uso do segundo desvio condicional, que interrompe a execução do bloco de tradução. Ao retornar do contexto de execução de blocos, é informado pelo bloco o motivo de tal retorno, servindo para o módulo de injeção de falhas como um evento para tomada de decisões. A quarta modificação representa a contagem do número total de instruções do processador durante a emulação, que é a soma do total de instruções necessárias para execução da aplicação ou sistema. Por fim, temos a parametrização do limite do número de instruções da arquitetura alvo inseridas em um bloco de tradução, a fim de definir ciclos de relógio. Considerando o fato de que uma emulação não é fiel aos estados internos de um processador real, uma execução em tempo real não é possível. Desta forma, os ciclos de relógio mencionados neste trabalho não correspondem a ciclos de relógio reais, mas sim um conjunto de instruções executados em cada bloco de tradução, simbolizando, a execução destes, ciclos de relógio.

Para manter precisão no processo de injeção de falhas, os resultados experimentais deste trabalho são obtidos com o número de instruções da arquitetura alvo em um bloco de tradução fixado em um. Isto é, em cada ciclo de relógio é executada uma instrução da arquitetura alvo, representada pelo bloco de tradução.

4.2.4 Classificação de Falhas

Uma etapa muito importante da metodologia é a classificação dos erros observados durante os experimentos. Para tal, é preciso definir um critério comparativo de forma

que ao final da execução de uma aplicação ou sistema, seja possível avaliar os efeitos de uma falha injetada.

A injeção de falhas em uma aplicação ou sistema pode impactar diferentemente dependendo do tipo e do trabalho o qual está sendo desempenhado por esta. O efeito observado pode ser classificado em três categorias, conforme Velazco (VELAZCO, 2000):

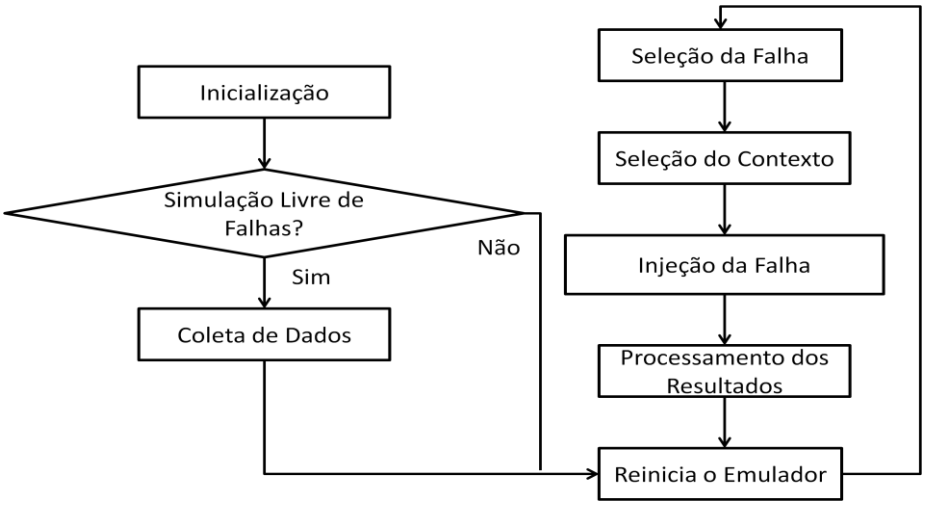
- Erro mascarado ou tolerado;
- Erro de resultado;
- Perda de sequência.

A primeira categoria, erro mascarado ou tolerado, corresponde a erros que não produzem efeito nos resultados esperados. Podem ser considerados mascarados, devido a não utilização do elemento de memória por parte da aplicação no momento da alteração realizada pela injeção da falha, ou tolerados que correspondem a falhas injetadas em elementos de memória não utilizados pela aplicação ou que são sobre-escritos pela própria aplicação ou sistema logo após sua alteração. A segunda categoria, erro de resultado, corresponde a falhas que produzem uma alteração dos resultados em comparação aos valores esperados. E por fim, a terceira categoria, perda de sequência, está relacionada a falhas que produzem o mau funcionamento do processador, não o possibilitando concluir a execução da aplicação ou sistema sendo posicionado em um estado de desligamento.

4.3 Fluxo de Injeção de *Soft Errors*

O fluxo de injeção de falhas proposto nesta metodologia utiliza-se de rotinas para o devido suporte a emulação de radiação. A figura 4.3 apresenta o fluxo de injeção de falhas e rotinas para devido suporte na metodologia.

Figura 4.3: Fluxo de injeção de falhas no emulador de arquiteturas.



O fluxo de injeção de falhas inicia com uma rotina de configuração. Nesta etapa ocorre a inicialização do módulo de injeção de falhas que deve ser posicionado dentro do emulador de processadores. Neste procedimento, o módulo recebe como parâmetro

um arquivo contendo uma lista de falhas conforme modelo de falhas proposto. Nestas falhas os atributos que as descrevem definem seu tipo, localização e ciclo de relógio no qual ela deve ser aplicada. A próxima etapa inicia-se com uma execução livre de falhas, com o intuito de coletar informações da aplicação ou sistema executando em condições normais. Após esta coleta de informações, o fluxo recai na reinicialização do emulador, a fim de iniciar os experimentos. Nesta etapa entra em ação a seleção de uma falha da lista, para ser injetada no hardware alvo de falhas.

Neste processo de emulação de *soft errors*, são definidos dois estágios. O primeiro estágio é representado pela seleção do contexto de injeção de uma falha, em que as opções disponibilizadas são os contextos de *bootloader* ou sistema. A diferenciação do contexto da falha a ser aplicada é necessária para selecionar o tipo de informação a ser mantida durante uma execução. O objetivo é utilizar-se das informações de forma a comparar com as informações obtidas na emulação livre de falhas. O segundo estágio é representado pelo processo de injeção de falhas. Esta etapa leva em conta todos os atributos que descrevem a falha para aplicá-la durante a emulação. Cabe salientar que todo o fluxo de injeção de falhas é executado automaticamente até que todas as falhas da lista sejam consumidas.

Como última etapa do fluxo, ocorre o processamento e exportação dos resultados que são gerados durante os experimentos. São comparadas informações da simulação livre de falhas com as coletadas após a emulação de uma falha durante um tempo de execução. Os resultados esperados podem ser definidos como estados dos recursos de hardware utilizados pela aplicação ou sistema durante a execução. Normalmente são avaliadas regiões de memória, valores de registradores do processador e configurações de dispositivos conectados ao processador. Neste último tipo são de interesse a configuração do dispositivo e devida operabilidade. Os resultados exportados da execução são classificados conforme modelo de classificação de falhas, possibilitando a comparação entre diversas execuções.

É importante salientar que a metodologia apenas considera a emulação de falhas simples. O caso de falhas múltiplas não faz parte do escopo do trabalho.

4.4 Emulação de Sistemas Complexos

Dependendo do tipo de metodologia de injeção de falhas utilizada para análise de mecanismos de tolerância, a complexidade da aplicação ou sistema em teste pode impossibilitar sua implantação. Sistemas operacionais normalmente requerem a execução de milhões de instruções e são aplicados atualmente nas mais diversas soluções em produtos. A utilização de metodologias de injeção de falhas baseadas em simulação de hardware não é muito aplicável para este cenário, visto que o tempo necessário para execução dos experimentos inviabiliza sua adoção. Para o caso de metodologias de injeção de falhas baseadas em emulação de hardware, utilizando FPGAs, o tempo de execução dos experimentos não é comprometido. No entanto, a alta complexidade de implantação, custo, coleta de informações e necessidade da descrição do hardware a ser emulado são fatores limitantes. Com base nisso, a emulação de arquiteturas por software é adequada para auxiliar no desenvolvimento de técnicas de tolerância a falhas por software. Sendo um processo de emulação através do uso de máquinas virtuais, é possível extrair ao máximo o desempenho do hardware no qual a mesma está executando.

4.5 Injetor de Falhas baseado no Emulador QEMU

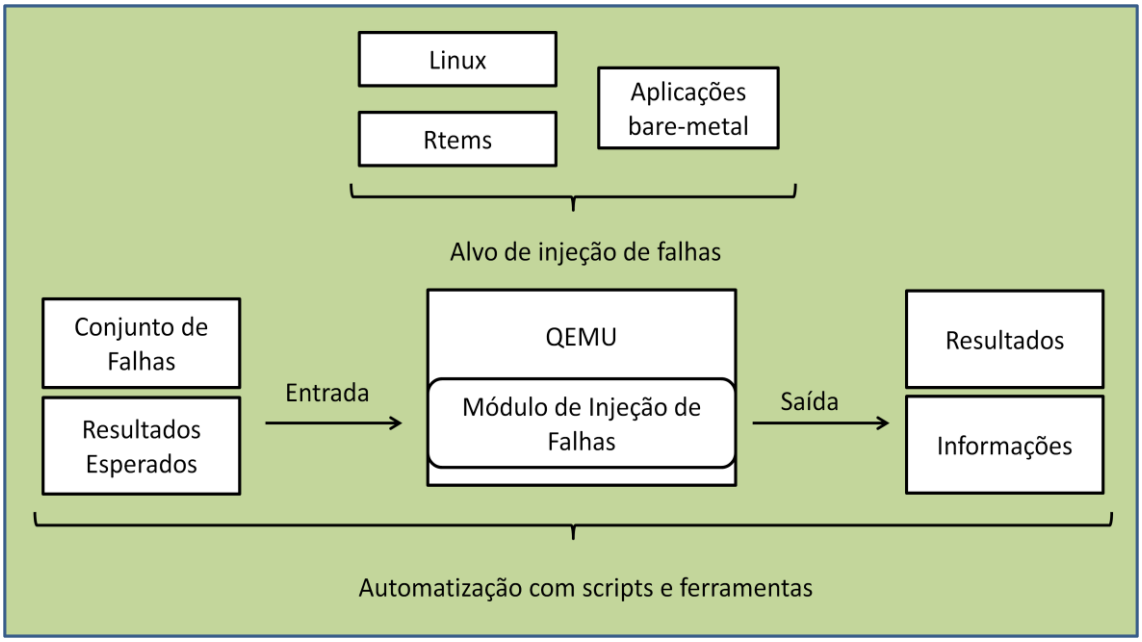
Para validação da metodologia de injeção de falhas proposta neste trabalho, foi desenvolvido um injetor de falhas baseado na máquina QEMU, conforme Geissler (GEISSLER, 2014). O objetivo do injetor de falhas cobre o modelo de falhas transientes para geração de SEUs em elementos de memória da arquitetura MIPS. Como elementos alvo de injeção de falhas, foram selecionados registradores e um conjunto de endereços de memória utilizados pela aplicação em execução na máquina.

Em adição ao injetor propriamente dito, foi desenvolvido um ambiente de experimentação completo disponibilizando benchmarks e sistemas operacionais para utilização do injetor de falhas. Para o processo de parametrização dos experimentos foram desenvolvidas ferramentas para geração de lista de falhas, análise de memória e geração de informações necessárias para o injetor gerar os resultados conforme definido pela metodologia proposta.

4.5.1 Instrumentação e ferramentas

O injetor de falhas baseado no emulador QEMU foi desenvolvido com a adição de um módulo de injeção de falhas (IF), através da modificação do código fonte do emulador.

Figura 4.4: Ambiente de experimentação com injetor de falhas baseado no emulador QEMU.



Este módulo, ilustrado na figura 4.4, tem por finalidade injetar falhas durante a emulação do processador com base na modificação da estrutura dos blocos de tradução e auxílio de rotinas de injeção de falhas. A parametrização deste módulo é realizada

através de dois arquivos principais, que são parâmetros de entrada de configuração do injetor de falhas. O primeiro arquivo trata-se de uma lista de falhas gerada por uma ferramenta disponível no ambiente de simulação, desenvolvida para este fim. O segundo arquivo trata dos resultados esperados no processo de simulação. Este último é utilizado para comparação com os resultados obtidos. Normalmente este arquivo tem um formato binário, contento o padrão que deve ser procurado pelo módulo FI na memória RAM, também emulada.

Para a utilização do ambiente, um conjunto de imagens contendo sistemas como Linux, RTEMS e aplicações simples, é disponibilizado e podem ser utilizados como parâmetros para o emulador. Em cada um dos sistemas é possível executar aplicações como benchmarks alvo de falhas.

Com base nos arquivos de entrada e na imagem selecionada, o processo de experimentação somente finaliza após a aplicação de todas as falhas presentes no arquivo de entrada. De forma a garantir tal requisito, a máquina QEMU é executada por um software denominado de supervisor. Tal software monitora a execução da QEMU comunicando-se com esta através de mensagens, garantindo que todas as falhas sejam injetadas e os resultados sejam disponibilizados ao final.

Na tabela 4.4 são apresentados os principais componentes do ambiente de simulação e experimentação e correspondente descrição.

Tabela 4.4: Componentes do ambiente de experimentação.

Componentes	***Descrição***
Gerador de falhas	Aplicação parametrizável para geração de arquivo com lista de falhas.
Benchmarks	Gerador de resultados esperados para a execução dos benchmarks.
Scripts	Conjunto de comandos para automatização dos experimentos.
qemu-1.6.0	Emulador QEMU.
rtems-4.10.99.0	Sistema operacional RTEMS com suporte a execução de benchmarks.
linux-2.6.32-5	Sistema operacional Linux com suporte aos benchmarks.
Ferramentas de compilação	Conjunto de receitas para compilação de todo o ambiente de simulação.

4.5.2 Ciclo de trabalho do ambiente de experimentação

O ambiente de experimentação é estruturado conforme a figura 4.4. Nele é possível emular os sistemas operacionais LINUX, RTEMS e um sistema simples. Em tais sistemas é possível executar 4 tipos de benchmarks, que podem ser selecionados para os experimentos. Estes são disponibilizados como processos ou tarefas em nível de espaço de usuário nestes sistemas. Na tabela 4.5 são apresentados os benchmarks e devida descrição.

Tabela 4.5: Conjunto de benchmarks disponíveis para o processo de emulação.

Benchmark	***Sigla***	***Descrição***
Matriz	M	Multiplicação de duas matrizes 32 x 32.
Quicksort	Q	Ordenamento de um vetor de 1000 posições.
SHA1	S	Algoritmo hash SHA1 de um vetor de 1000 elementos.
XOR	X	Operações lógicas OU exclusivo executadas sob um vetor de 200 posições por 1000 vezes.

Quando iniciado o processo de experimentação, cada uma das falhas contidas no arquivo de falhas é aplicada individualmente em uma execução do sistema ou aplicação alvo de falhas. Falhas múltiplas não são simuladas. Ao final da aplicação de todas as falhas, são gerados dois arquivos de saída. O primeiro contendo os resultados dos experimentos e o segundo informações e mensagens impressas durante o processo de emulação.

Além da possibilidade de execução de sistemas operacionais, é possível também realizar a injeção de falhas em aplicações simples, denominadas *bare-metal*. Tais aplicações tem por finalidade a execução de tarefas específicas sem a complexidade de um sistema operacional.

Em suma, o ambiente de experimentação, devido ao uso de software com código fonte aberto, pode ser facilmente extensível. Dessa forma, novas aplicações e sistemas podem ser adicionados, assim como novos modelos de falhas podem ser incorporados ao módulo FI.

5 RESULTADOS DA INJEÇÃO DE FALHAS DA METODOLOGIA PROPOSTA

Para validação do injetor de falhas baseado no emulador QEMU, foram selecionados dois sistemas operacionais, LINUX e RTEMS. Além destes, um sistema mais simples, que é basicamente uma aplicação dedicada, também foi selecionado para execução dos benchmarks, presentes no ambiente de experimentação. Estes sistemas têm por finalidade definir o nível de complexidade de execução, que para a aplicação dedicada é considerado o mais baixo. Em cada um destes sistemas foi realizada uma campanha de injeção de falhas tendo como alvos a memória de dados e os registradores. Neste processo comparativo a arquitetura MIPS foi selecionada.

5.1 Configuração do ambiente de experimentação

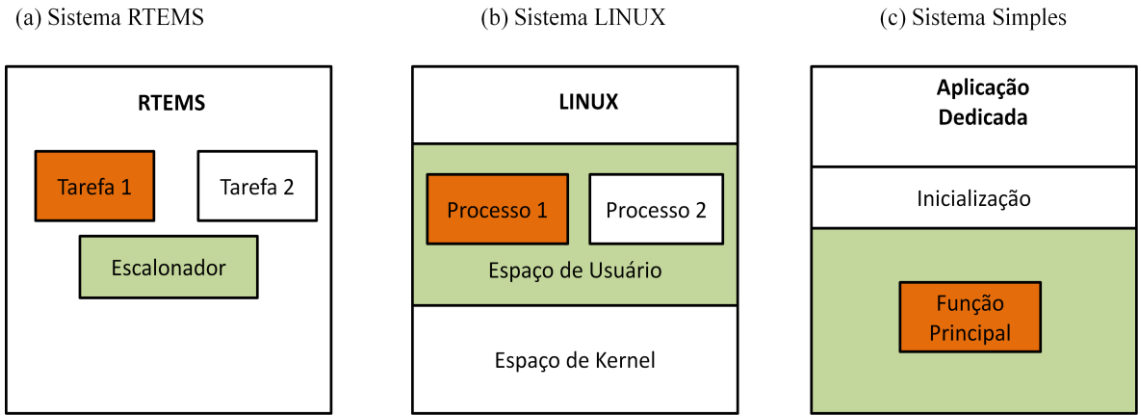
O experimentos foram realizados com o uso do modelo de hardware MIPS 24Kc versão 0.0 com FPU versão 0.0, disponível no emulador QEMU. Além deste, foi reservado um espaço de memória RAM para emulação de 128MB. O computador pessoal, para execução dos experimentos, é composto por um processador Intel Core i7-2670QM CPU de 2.20 GHz, com 8GB de memória RAM e sistema operacional LINUX, distribuição Ubuntu 13.04.

Para o processo de emulação, a tradução dinâmica foi limitada a representar uma instrução da arquitetura alvo por bloco de tradução. Com base neste valor, em cada ciclo de relógio será executada uma instrução do sistema alvo, conforme possibilidade prevista na metodologia proposta. Com base nisso, os três sistemas alvo de falhas serão comparados de forma igualitária.

5.2 Estrutura dos sistemas alvo de falhas

A estrutura de cada um dos sistemas alvo de falhas pode ser visualizada na figura 5.1. Em cada um dos sistemas é executado um benchmark a fim de exercitar o uso de elementos de memória presentes no processador emulado. O sistema RTEMS, figura 5.1 (a), é estruturado principalmente por um escalonador e duas tarefas. A primeira tarefa é responsável por executar o algoritmo definido pelo benchmark selecionado. Já a segunda imprime mensagens de depuração na interface serial do processador. O sistema Linux, figura 5.1 (b), sendo o sistema mais complexo dentre todos, é estruturado basicamente por dois níveis principais: espaço de usuário e espaço de Kernel. No espaço de usuário foi definido um processo para execução do benchmark, o processo 1. Já o processo 2, concorrente ao primeiro em termos de escalonamento, é utilizado também com o intuito de gerar mensagens de depuração a fim de indicar em que condição se encontra a execução do sistema no emulador. No espaço de Kernel foi selecionado um conjunto de módulos a fim de ilustrar uma situação real de aplicação. A utilização de dois processos ou tarefas, para o caso do sistema RTEMS, foi abordada para adicionar certa complexidade na estrutura do software de forma que ao final se possa compará-los. Por fim, o terceiro alvo de falhas é a aplicação dedicada. Esta é definida basicamente em duas camadas. A camada de inicialização do processador e a função principal, a qual é responsável pela execução de benchmarks.

Figura 5.1: Estrutura dos sistemas alvo de injeção de falhas (a) RTEMS, (b) LINUX e (c) sistema simples.



Na tabela 5.1 são apresentadas informações referentes ao processo de emulação. O número de ciclos de relógio e correspondente tempo de execução em milésimos de segundo, visto pelo sistema hospedeiro, são exibidos para cada benchmark.

Tabela 5.1: Número de ciclos de relógio emulados e correspondente tempo visto pelo sistema hospedeiro no processo de emulação do processador MIPS 24Kc.

Complexidade	***Benchmark***	***Ciclos de relógio ($x10^6$)***	***Tempo de execução (ms)***
RTEMS	M	180	2720
	Q	150	2020
	X	140	1950
	S	140	1950
LINUX	M	220	5150
	Q	200	5060
	X	195	5050
	S	195	5050
Sistema Simples	M	20	540
	Q	15	480
	X	10	230
	S	10	230

5.3 Injeção de Falhas em Registradores

A campanha de injeção de falhas teve como alvo 32 registradores disponíveis do conjunto de instruções do processador MIPS. Além destes, foi também alvo de falhas o contador de programa, presente na descrição do modelo de hardware da arquitetura no emulador QEMU. A tabela 5.2 apresenta estes registradores com correspondente função.

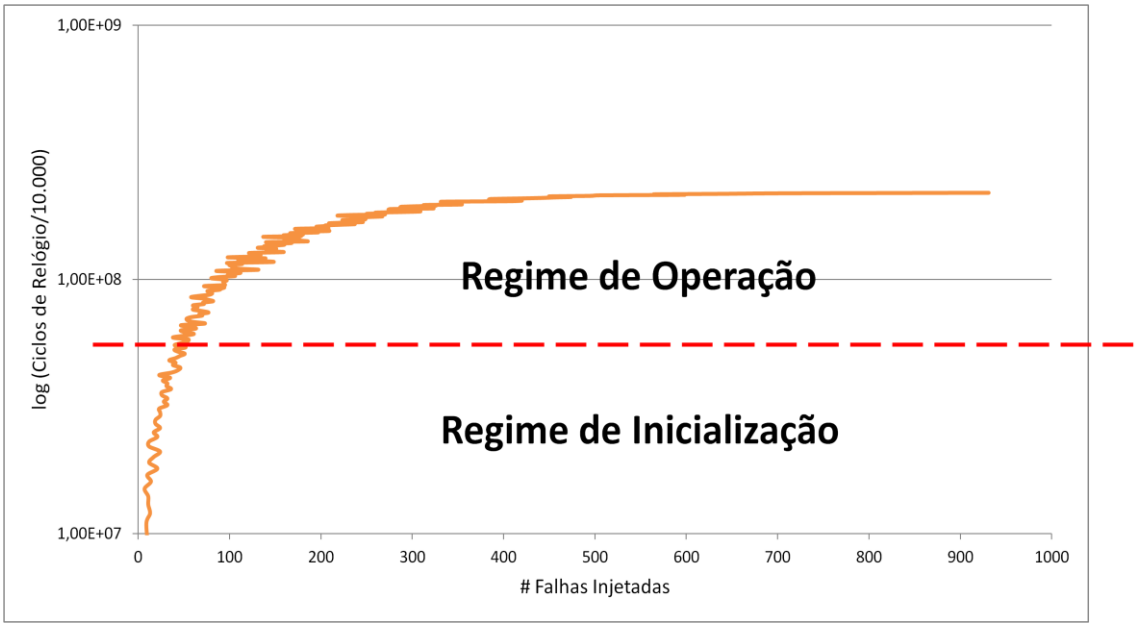
Tabela 5.2: Registradores da arquitetura MIPS, alvos de falhas durante os experimentos.

Registrador	***Símbolo***	***Função***
R0	$zero	Sempre zero.
R1	$at	Reservado para o montador.
R2-R3	$v0-$v1	Armazenamento de resultados.
R4-R7	$a0-$a3	Armazenamento de argumentos.
R8-R15	$t0-$t7	Temporários não salvos.
R16-R13	$s0-$s7	Salvos para uso futuro.
R24-R25	$t8-$t9	Temporários não salvos.
R26-R27	$k0-$k1	Reservados pelo sistema operacional
R28	$gp	Ponteiro global.
R29	$sp	Apontador de pilha.
R30	$fp	Apontador de frame.
R31	$ra	Endereço de retorno.

5.3.1 Distribuição de Falhas

O processo de injeção de falhas nos 32 registradores da arquitetura MIPS foi efetuado de forma igualmente distribuída. Isto é, em cada registrador foram injetadas 1000 falhas, definindo o espaço, totalizando 32 mil falhas para todos os registradores. Dessa forma, um total de 384 mil falhas foram injetadas durante os experimentos, visto que são 4 benchmarks executados em 3 sistemas distintos. Para assegurar uma condição conhecida para a etapa de comparação dos resultados, ao final da execução de um benchmark, e correspondente processo de injeção de uma falha, o ambiente de experimentação tem seu estado retornado para o um valor conhecido. Após tal procedimento, uma nova execução é iniciada, a fim de emular uma nova falha.

Figura 5.2: Distribuição de falhas no tempo.



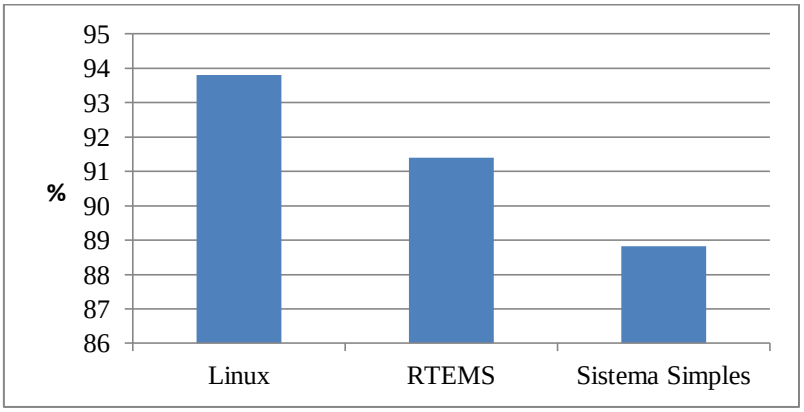
As figuras 5.2 exibe a distribuição das falhas no tempo, aplicada em cada um dos benchmarks em execução, em cada um dos níveis de complexidade abordados no trabalho. Nesta distribuição é ilustrada a tendência da aplicação das falhas no tempo, a qual cresce à medida que os ciclos de relógio se aproximam do fim da execução de cada benchmark. Tal tendência foi adotada a fim de injetar mais falhas durante a execução dos benchmarks e não no processo de inicialização dos sistemas. São definidos dois regimes: inicialização e operação, conforme a figura. O segundo regime é o de interesse na análise, visto que caracteriza a injeção de falhas no momento em que um benchmark é executado.

5.3.2 Comparação em nível de complexidade de sistemas

A legenda que segue define os benchmarks utilizados nos experimentos:

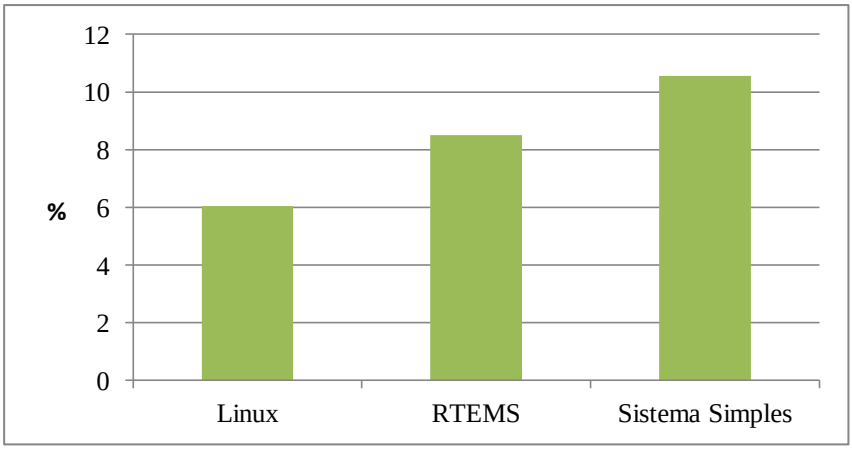
- M: algoritmo de multiplicação de matrizes;
- Q: algoritmo de ordenamento quicksort;
- X: algoritmo de sequência de operações ou exclusivo;
- S: algoritmo de *hash* SHA1.

Figura 5.3: Falhas mascaradas e/ou toleradas campanha de injeção de falhas nos registradores.



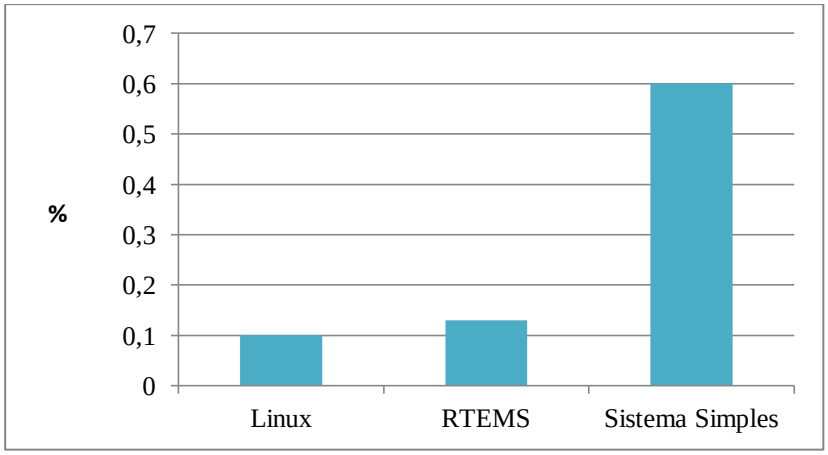
Na figura 5.3, podemos observar os efeitos causados pelas falhas em cada um dos sistemas operacionais. É possível notar que o sistema operacional RTEMS tem um percentual de falhas mascaradas ou toleradas inferior ao sistema LINUX. Por sua vez, o sistema simples, de modo geral apresenta o menor percentual de tolerância. Tal comportamento pode ser devido a complexidade envolvida em cada um dos sistemas. Sendo o sistema operacional Linux estruturado um escalonador de processos baseado em preempção, o chaveamento entre processos é frequente. Dessa forma, uma falha injetada em um registrador, com o intuito de prejudicar um benchmark, pode ser aplicada em um momento em que este não está em execução. Outra possibilidade pode ser a restauração de contexto após o chaveamento entre processos, que pode sobre-escrever o registrador que contem a falha. Com base nisso, o alto percentual de mascaramento do sistema operacional LINUX em comparação aos outros dois sistemas é relativamente superior.

Figura 5.4: Perda de sequência na campanha de injeção de falhas nos registradores.



Outro aspecto a destacar é o elevado número de perda de sequência presente nos experimentos, gerado pelo processo de injeção de falhas, que pode ser visualizado na figura 5.4. Nele, o sistema simples se mostra mais vulnerável a este tipo de erro em virtude da simplicidade envolvida na elaboração do mesmo, não possuindo chaveamento entre processos ou tarefas.

Figura 5.5: Erros de resultados na campanha de injeção de falhas nos registradores.

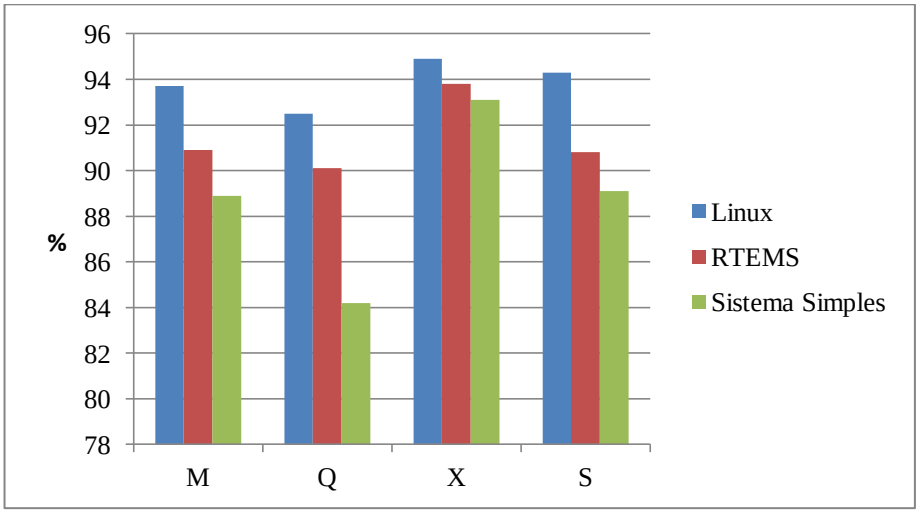


Na figura 5.5 é possível verificar que os erros de resultados são superiores no sistema RTEMS, em relação ao sistema LINUX. Nesta semântica, o nível de complexidade do sistema operacional contribuiu para o incremento das falhas mascaradas ou toleradas. Realizando o mesmo tipo de comparação com o sistema simples, é possível observar a contribuição da complexidade de um sistema para com o mascaramento ou tolerância às falhas injetadas. Para o sistema simples, os valores de erro e perda de sequência mostraram-se superiores aos dos sistemas Linux e RTEMS. Já o número de falhas mascaradas ou toleradas resultou em um valor menor, evidenciando a maior vulnerabilidade.

5.3.3 Comparação de benchmarks

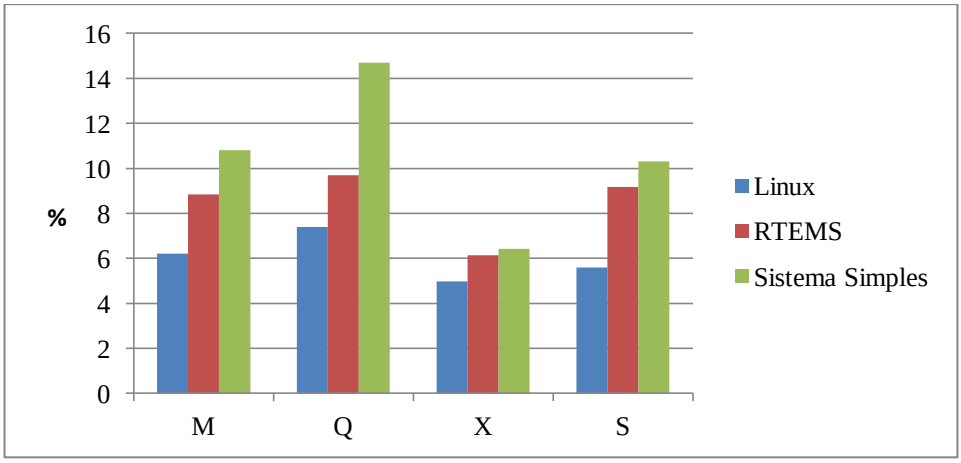
Ao analisar os resultados em função de cada benchmark, é possível observar a suscetibilidade de cada um. A figura 5.6 a classificação de falhas mascaradas e/ou toleradas para cada um dos benchmarks executados em cada um dos sistemas.

Figura 5.6: Falhas mascaradas e/ou toleradas na campanha de injeção de falhas nos registradores.



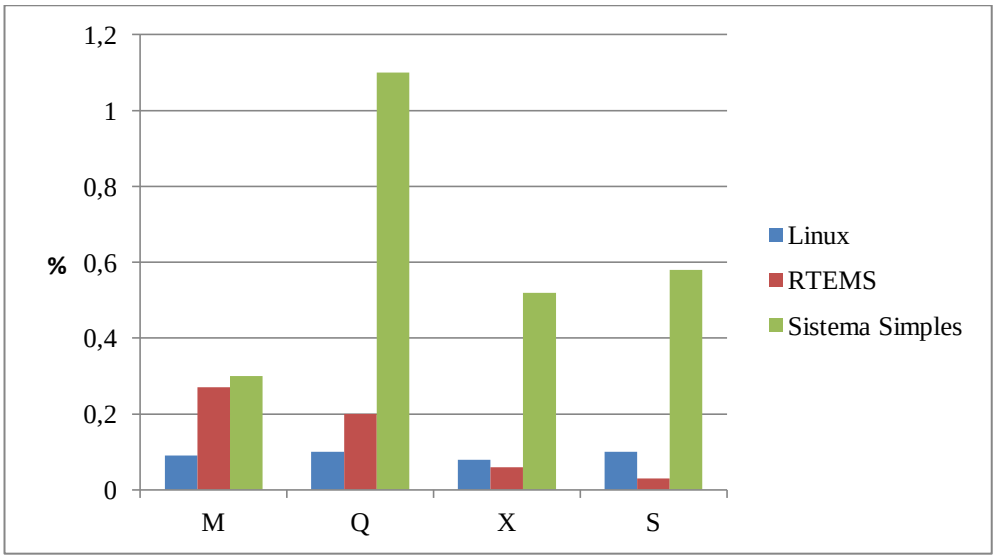
Considerando a estrutura de cada um dos algoritmos, o maior percentual é representado pelo algoritmo de *hash* SHA 1.

Figura 5.7: Perda de sequência na campanha de injeção de falhas nos registradores.



Na figura 5.7, é apresentada a classificação de perda de sequência. Nela podemos observar uma maior vulnerabilidade do algoritmo no sistema simples.

Figura 5.8: Erros de resultado na campanha de injeção de falhas nos registradores.



Por fim, a figura 5.8 apresenta os erros de resultado. Neste caso, o algoritmo *quicksort* sofreu o maior impacto diante do processo de injeção de falhas. Ainda assim, a representatividade é baixa, visto que a perda de sequência teve maior impacto no processo de injeção de falhas.

É importante destacar que a análise de vulnerabilidade de um sistema pode ajudar o desenvolvedor de mecanismos de tolerância a falhas na tomada de decisões por uma melhor abordagem no desenvolvimento destas. Sendo assim, a metodologia proposta disponibiliza um conjunto de informações úteis neste processo. Ao proporcionar tais características, possibilita ter conhecimento de quais registradores afetam mais um ou outro benchmark, ou qual sistema é mais sensível a falhas. Em suma, no final de uma campanha de injeção de falhas, o desenvolvedor pode ter um panorama detalhado da aplicação ou sistema em teste.

5.4 Injeção de Falhas na Memória

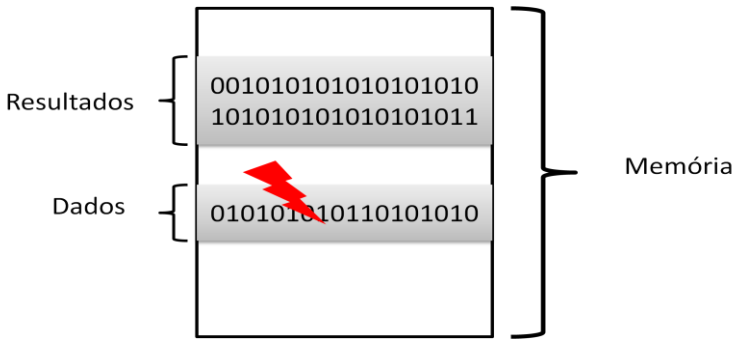
A injeção de falhas na memória foi realizada em uma área definida como área de dados. Nesta área as aplicações ou tarefas que executam os benchmarks fazem uso do conteúdo armazenado para efetuar as operações de forma a gerar os resultados. Os resultados, por sua vez, são posicionados em outra área de memória. Com o uso de ferramentas para análise da memória utilizada pelo emulador de processadores, e coleta de informações através da execução livre de falhas, foi possível realizar a injeção de falhas mais direcionada, evitando injetar falhas em posições que não são utilizadas pelas aplicações.

5.4.1 Abordagem para injeção de falhas na memória

As falhas na memória foram injetadas de forma simular ao processo de injeção nos registradores. Na campanha foram somente emuladas falhas transientes para produção de SEUs. Neste processo foram selecionadas posições de memória com base na especificação do intervalo alvo de falhas. Tal intervalo foi fixado em um valor igual para os quatro benchmarks executados nos sistemas, cobrindo 1000 posições de memória de 32 bits cada.

A figura 5.9 apresenta a disposição da memória utilizada por um benchmark. A mesma é divida basicamente em duas áreas: memória de dados e memória de resultados. A área de dados é utilizada por um benchmark para geração dos resultados, servindo de entrada para execução dos algoritmos. Já a área de resultados, por sua vez, é destinada a manter o produto final da execução destes algoritmos. A memória em que o programa executa não foi alvo de injeção de falhas nos experimentos.

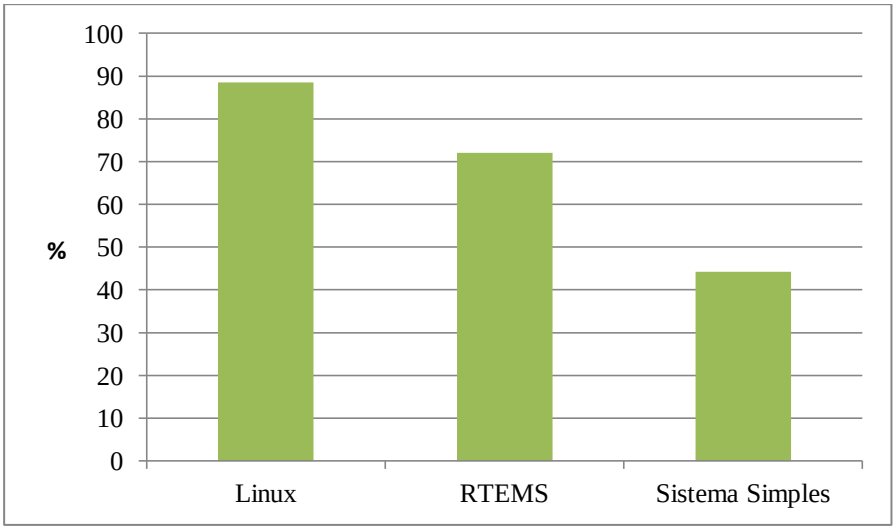
Figura 5.9: Definição da área de dados alvo de falhas nos experimentos e área de resultados, gerada após a execução de um benchmark.



5.4.2 Classificação das falhas para o caso da memória

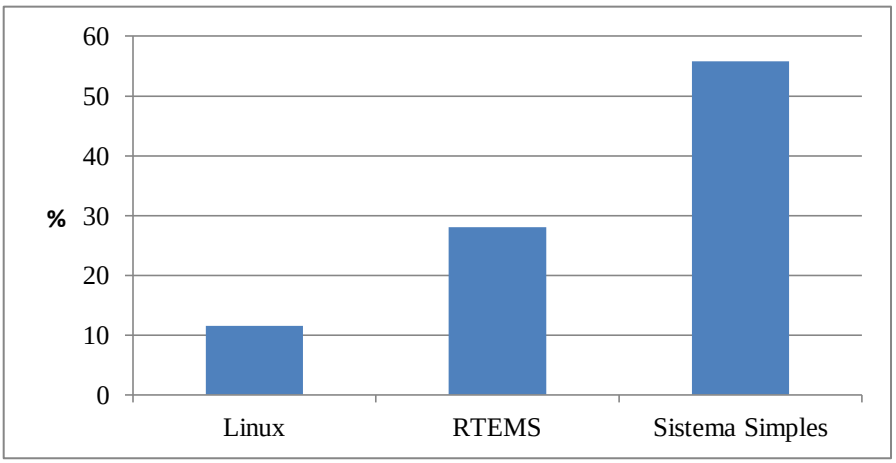
Nas 1000 posições de memória alvo de falhas, foram injetadas 5000 falhas distribuídas no tempo. Dessa forma, o número total de falhas atingiu o valor de 60 mil, visto que são 4 benchmarks executados em 3 sistemas distintos.

Figura 5.10: Perda de sequência na campanha de injeção de falhas nos registradores.



Na figura 5.10, podemos observar que a maior parte das falhas injetadas é considerada do tipo mascarada ou tolerada. Cabe destacar que ao aumentar a complexidade do sistema, temos uma redução de erros de resultado, apresentado na figura 5.11. No entanto, as falhas existem, mas não estão produzindo erros nos resultados esperados. O caso de perda de sequência não foi detectado durante a injeção de falhas na memória dados. Contudo, para um conjunto de posições de memória diferente, tal tipo de efeito pode ocorrer.

Figura 5.11: Erros de resultado na campanha de injeção de falhas na memória.



5.5 Considerações sobre testes exaustivos

No processo de análise de confiabilidade de um sistema, pode ser requisito de teste um conjunto de falhas que possa cobrir e exercitar todas as ineficiências de uma aplicação ou sistema em teste. O modelo FARM de Arlat (ARLAT, 1993) define alguns

atributos para avaliação de mecanismos de tolerância a falhas para análise de confiabilidade de sistemas.

O modelo FARM consiste em um conjunto de falhas F que representa o vetor de testes utilizado para injetar falhas em um sistema. Tal conjunto é composto por um modelo M para uma falha, localização L em que a falhas serão aplicadas, e tempo T, o qual representa um intervalo possível para aplicação destas. Neste cenário, outro atributo é definido, a ativação A. Tal atributo especifica como os experimentos são conduzidos. A ativação pode representar um conjunto de benchmarks a serem analisados nos testes, podendo influenciar no cálculo e alterar o tamanho da lista de falhas. Por fim, o último atributo é denominado R, o qual representa a leitura da saída dos experimentos, para devida avaliação dos efeitos causados pelas falhas no sistema em teste. Com base nisso, uma sequência de experimentos é definida como sendo um dos pontos pertencentes ao espaço definido pela equação (1), sendo o conjunto de falhas F definido pela equação (2).

$$(1)\ Espaço = \{F\ x\ A\ x\ R\}$$
$$(2)\ F = \{M\ x\ L\ x\ T\}$$

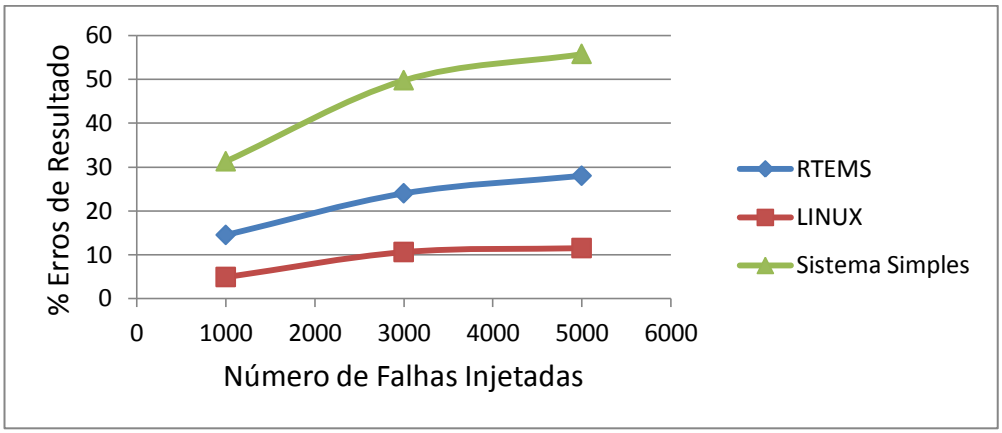
Considerando o caso da injeção de falhas na memória e o modelo FARM, a localização seria o número de posições alvo de falhas durante a execução de um benchmark. Já para caso dos registradores, seria o número de registradores alvo de falhas. Considerando que temos M = 1 e R = 1, o total de falhas, simbolizando o espaço E pode ser obtido pela equação (3), sendo T o número de ciclos de relógio. Tal equação gera como resultado o total de falhas, ou vetor de falhas, beirando o exaustivo.

$$(3)\ total\ de\ falhas = ciclos\ de\ relógio\ x\ T$$

Para exemplificar tal situação, podemos considerar a injeção de falhas na memória no caso do benchmark de multiplicação de matrizes, executado no sistema RTEMS. Nesta situação, o total de número de falhas para beirar o exaustivo atingiria o valor de $1{,}8x10^{11}$.

Tal valor demandaria um tempo muito alto para o processo de injeção de falhas, sendo impraticável. Como alternativa a um número exaustivo de falhas, é realizada, para o caso das memórias, uma análise empírica, de forma a convergir para um valor razoável no processo de injeção de falhas.

Figura 5.12: Erros de resultado observados durante a injeção de falhas nos benchmarks para os três sistemas com vetores de 1000, 3000 e 5000 falhas.



A figura 5.12 apresenta tal análise realizando injeção de falhas para campanhas de 1000, 3000 e 5000 falhas. Podemos notar que de 1000 para 3000 falhas obtivemos um crescimento razoável do número de erros de resultado aumentado em 9,5%, 6,7% e 18,5% para o RTEMS, Linux e sistema simples, respectivamente. Porém ao subir o valor do vetor de falhas para 5000 falhas, o percentual de erros observados cresceu menos, ficando em 4%, 0,9% e 6% para o RTEMS, Linux e sistema simples, respectivamente. Conforme o gráfico, ao continuar a aumentar o vetor de falhas percebe-se a estabilização dos erros observados, caracterizando uma tendência. Desta forma, nos experimentos com a memória optou-se por adotar o vetor com valor de 5000 para campanha de injeção de falhas.

5.6 Tempo de Execução dos Experimentos

O tempo execução de experimentos é um fator preocupante nas metodologias de injeção de falhas. Dependendo do tipo de abordagem, nível de complexidade do processador e técnica adotada, o tempo pode sofrer grande impacto no processo de execução dos experimentos.

Tabela 5.3: Tempo de execução dos experimentos para cada benchmark, em cada sistema.

Complexidade	***Benchmark***	***Tempo de Simulação por benchmark (horas)***	***Tempo de Simulação (horas)***
RTEMS	M	25,3	78,3
	Q	17,6	
	X	18,3	
	S	17,1	
LINUX	M	49,7	195,3
	Q	48,8	
	X	46,9	
	S	49,9	
Sistema Simples	M	5,3	16,6
	Q	4,6	
	X	3,6	
	S	3,1	

Outro fator limitante é a complexidade da aplicação ou sistema a ser executado, gerando em algumas situações dificuldades de coleta e análise dos efeitos, tão bem como influência no tempo total de simulação, impossibilitando muitas vezes seu uso. Tendo em vista tais limitações, a metodologia apresentada neste trabalho utiliza-se de tradução dinâmica para acelerar o processo de experimentação e injeção de falhas. Logo, como resultado, podemos verificar na tabela 5.3 o tempo total de injeção de falhas para cada um dos benchmarks executados nos sistemas. Os tempos englobam falhas injetadas nos registradores e memórias. O tempo total para execução dos experimentos em cada sistema é também apresentado.

Devemos levar em conta que no tempo total dos experimentos foi o impacto em *speed down* em relação ao QEMU original ficou em torno de 34,1%, devido ao limite de uma instrução da arquitetura alvo por bloco de tradução, no processo de tradução dinâmica. No entanto, mesmo sendo adicionado esse acréscimo de tempo, a técnica se torna atrativa para utilização no desenvolvimento de mecanismos de tolerância, visto que é possível reproduzir os experimentos, injetar grande quantidade de falhas quando comparada a outras técnicas, e facilidade de classificação das falhas em virtude da acessibilidade fornecida por todos os pontos do software do emulador.

5.7 Análise comparativa de desempenho

A fim de comparar os resultados obtidos com o injetor de falhas, foram selecionados 3 trabalhos os quais fazem uso de técnicas que utilizam-se de FPGA. Para comparação foram selecionados o sistema AMUSE (ENTRENA, 2012), sistema THESIC (VALDERAS, 2007) e a plataforma ASTERICS *(Advanced System for the Test under Radiation of Integrated Circuits and Systems)* (MANSOUR, 2012).

Figura 5.13: Total de falhas injetadas com os sistemas AMUSE, ASTERICS e injetor desenvolvido no trabalho.

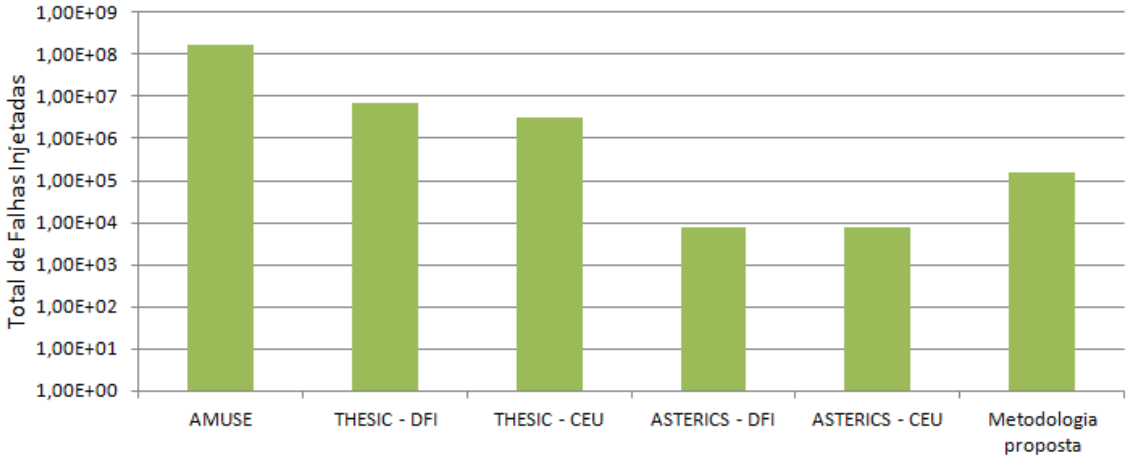
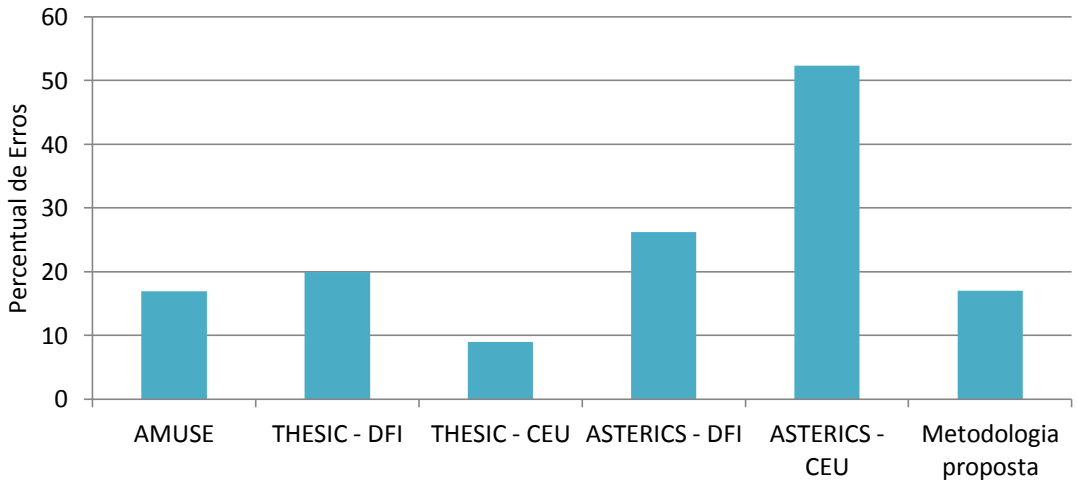


Figura 5.14: Percentual de erros de resultado dos 3 trabalhos selecionados em comparação a metodologia proposta.



O sistema AMUSE permite emular falhas transientes, tendo como um caso especial destas a emulação de SEUs. Os sistemas THESIC e ASTERICS, por sua vez, são utilizados pelos autores dos trabalhos para comparar duas abordagens distintas. A primeira, denominada DFI (*Direct Fault Injection*), é baseada em uma modificação da estrutura dos *flip-flops* que compões o processador, de forma que SEUs possam ser

emulados em tempo real. A segunda é baseada na técnica C.E.U. (VELAZCO, 2000), a qual injeta falhas através de rotinas de software, acionadas por meio de uma interrupção do processador.

Na figura 5.13 é apresentado o total de falhas injetadas em cada um dos trabalhos selecionados. Considerando o estudo de caso para o sistema simples, realizado com o injetor de falhas do trabalho, podemos notar que a quantidade de falhas injetadas atingiu um valor intermediário em relação às outras técnicas. Este total de falhas aplicado nos experimentos pode ser considerado razoável quando comparado ao valor que vem sendo aplicado em trabalhos na literatura com total superioridade do sistema AMUSE.

Na figura 5.14 é apresentado o percentual de erros de resultado obtidos com os experimentos em comparação com os 3 trabalhos selecionados. O objetivo de tal comparação é dar uma noção geral das potencialidades do injetor de falhas diante de técnicas do estado da arte, visto que os trabalhos injetaram falhas em arquiteturas distintas com diferentes benchmarks. A metodologia proposta atingiu um total de 17,03% de erros de resultado, ficando próxima ao sistema AMUSE e THESIC (caso CEU). Podemos notar a desvantagem da metodologia baseada em emulação de processadores em relação aos métodos que se utilizam de hardware no processo de injeção falhas com relação ao fator tempo demandado para os experimentos. A quantidade de falhas que podem ser injetadas com tal abordagem a supera em duas ou mais ordens de grandeza. Considerando o sistema AMUSE, por exemplo, 164.720.820 falhas são injetadas em 14,58 horas. Já para o injetor de falhas do trabalho, são injetadas 148.000 falhas em 16,6 horas. Em uma análise comparativa da capacidade de injeção de falhas, o sistema AMUSE é capaz de injetar 3.136 falhas por segundo. Já o injetor proposto no trabalho, 2,48 falhas por segundo.

É notória a vantagem do sistema AMUSE em termos de desempenho de injeção de falhas comparado a metodologia proposta. No entanto, é importante salientar que o injetor proposto é facilmente portável para outra arquitetura e sistema, evidenciando sua flexibilidade. Outro fator é seu desempenho quando comparamos com simulação de hardware, não praticável quando o alvo de falhas é um processador complexo ou sistema operacional.

CONCLUSÃO

Este trabalho consiste na apresentação de uma proposta de metodologia de injeção de falhas baseada em emulação de processadores, e estudo prático através de um ambiente de injeção de falhas desenvolvido com base na metodologia proposta. Inicialmente foi apresentada uma breve revisão teórica no que se refere ao efeito da radiação em dispositivos semicondutores. Após esta, foram apresentadas técnicas e metodologias de injeção de falhas aplicáveis no contexto de processadores, buscando destacar as vantagens, desvantagens e limitações presentes em cada uma das técnicas. Dessa forma, o capítulo 2 desta dissertação contribui como um texto de revisão sobre o tema, auxiliando pesquisadores que buscam por informações com foco em injeção de falhas em processadores.

Posteriormente, no capítulo 3 foram apresentadas plataformas de emulação de processadores e devida aplicabilidade em trabalhos na literatura, relacionados ao contexto de injeção de falhas. Além disso, foram apresentados conceitos referentes à emulação de processadores com exemplificação através da estrutura da máquina QEMU, destacando o processo de tradução dinâmica.

Baseado nos conceitos apresentados no capítulo 3, o capítulo 4 apresentou uma metodologia de injeção de falhas baseada em emulação de processadores. Com base na utilização da tradução dinâmica de instruções, para acelerar o processo de execução de aplicações por meio de emulação, e auxiliar no processo de injeção de falhas, um modelo de falhas com foco na emulação de *soft errors* foi apresentado cobrindo falhas transientes e permanentes, para todos os elementos de memória, acessíveis por meio de um modelo de hardware descrito em software, presente em emuladores como o QEMU.

Além da apresentação da metodologia de injeção de falhas, foram realizados experimentos com o ambiente de injeção de falhas desenvolvido neste trabalho. Para investigar os efeitos e o comportamento de um software submetido a estes modelos, foram selecionados três tipos de aplicações alvo de falhas, definindo três graus de complexidade. O primeiro trata-se do sistema Linux, representando o maior grau. O segundo, o sistema RTEMS, representando grau intermediário. E por fim, o terceiro trata-se de um sistema simples, ilustrando uma aplicação mais dedicada, representando o menor grau de complexidade. Em cada um destes sistemas, foram executados quatro benchmarks para serem analisados. A injeção de falhas foi realizada em um processador da arquitetura MIPS. Como outra contribuição, neste trabalho foi possível comparar estes três níveis de complexidade para dois tipos de elementos de memória alvos de falhas: registradores e memória dados. A comparação entre os três sistemas permitiu avaliar a contribuição de cada um destes níveis no mascaramento ou tolerância das falhas injetadas, não produzindo erros nos resultados esperados. Além disso, foi possível registrar o tempo de realização dos experimentos, totalizando 444 mil falhas injetadas nos elementos de memória, com duração de 290,1 horas.

Neste trabalho também foram propostas melhorias para as limitações apresentadas por outras técnicas de injeção de falhas. Aspectos como disponibilidade de descrição de hardware, controlabilidade e reprodutibilidade dos experimentos, acessibilidade a recursos normalmente inacessíveis, e tempo aceitável para execução dos experimentos, foram temas de comparação e proposta de solução presente na metodologia. Como

melhoria em relação a técnicas recentes, apresentadas por Xu e Yi (XU, 2012; YI, 2013), o trabalho propõe uma forma metódica de análise das vulnerabilidades de aplicações e/ou sistemas em relação a estes, através da execução dos experimentos de forma controlada e reproduzível, explorando a flexibilidade como uma característica fundamental da abordagem.

Em uma visão mais abrangente, o trabalho propõe um método independente de arquitetura ou processador, aplicável no suporte ao desenvolvimento de software tolerante a falhas, bem como sua aplicação na análise de vulnerabilidade de aplicações diante de condições adversas, relacionado a um ambiente com radiação. Cabe destacar também a flexibilidade do ambiente de injeção de falhas desenvolvido, possibilitando a emulação de outras arquiteturas com diferentes processadores, em contraste a outras técnicas de injeção de falhas muito dependentes da arquitetura alvo das falhas.

Um trabalho relacionado a esta dissertação foi apresentado no *15th IEEE Latin American Test Workshop* (LATW) realizado em Fortaleza (GEISSLER; KASTENSMIDT; SOUZA, 2014), propondo um injetor de falhas baseado no emulador QEMU.

Para trabalhos futuros, os estudos podem contemplar comparações dos resultados desta dissertação com mais técnicas de diferentes abordagens de injeção falhas. Além disso, podem ser realizados experimentos com outras arquiteturas de processadores para análise dos efeitos em sistemas operacionais. Outro aspecto a avaliar é a injeção de falhas em processadores com múltiplos núcleos, muito aplicáveis a soluções tecnológicas atualmente.

REFERÊNCIAS

BALEN, Tiago Roberto. R. **Efeitos da Radiação em Dispositivos Analógicos Programáveis (FPAAs) e Técnicas de Proteção**. 2010. 208f. Tese (Doutorado em Engenharia Elétrica) - Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre.

ABATE, F.; STERPONE, L.; VIOLANTE, M. A New Mitigation Approach for Soft Errors in Embedded Processors. **IEEE Transactions on Nuclear Science**, vol. 55, p. 2063-2069, Sep. 2007.

ACLE, J. P.; REORDA, M. S.; VIOLANTE, M. Implementing a safe embedded computing system in SRAM-based FPGAs using IP cores: a case study based on the Altera NIOS-II soft processor. In: Circuits and Systems (LASCAS), 2011. **IEEE Second Latin American Symposium**, 2011. p. 1-5.

ALEXANDER, B.; DONNELAN, S.;; JEFFRIES, A.; OLDS, T.; SIZER, N. Boosting Instruction Set Simulator Performance with Parallel Block Optimisation and Replacement. In: Conferences in Research end Practice and Information Technology (CRPIT), 2012. **Proceedings of the Thirty-Fifth Australasian Computer Science Conference (ACSC)**, 2012. vol. 122 p. 11-20.

ANTONI, L.; LEVEUGLE, R.; FEHER, B. Using run-time reconfiguration for fault injection applications. In: Instrumentation and Measurement Technology Conference (IMTC), 2001**. Proceedings of the 18th IEEE**, 2001. vol. 3 p. 1773-1777.

ARLAT, J.; COSTES, A.; COUZET, Y.; LAPRIE, J.; POWELL, D. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. **IEEE Transactions on Computer Science**, vol. 42, Ago. 1993.

ARLAT, J.; CROUZET, Y.; KARLSSON, J.; FOLKESSON, P.; FUSHS, E.; LEBER, G. H. Comparison of Physical and Software-Implemented Fault Injection techniques. **IEEE Transactions on Computers**, vol. 52, No. 9, Sep. 2003.

AZAMBUJA, J. R.; PAGLIARINI, S.; ALTIERI, M.; KASTENSMIDT, F. L.; HUBNER, M.; BECKER, J.; FOUCARD, G.; VELAZCO, R. A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware. **IEEE Transactions on Nuclear Science**, vol. 59, No. 4, Aug. 2012.

BARAZA, J. C.; GRACIA, J.; BLANC, S.; GIL, D.; GIL, P. J. Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code. **IEEE Transactions on very large scale integration (VLSI) Systems**, vol. 16, No. 6, Jun. 2008.

BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In: FREENIX Track: USENIX Annual Technical Conference, 2005.

BENSO, A.; PRINETTO , P.; REBAUDENGO, M.; REORDA, M. S. EXFI: A Low-Cost Fault Injection System for Embedded Microprocessor-Based, **ACM Transactions on Design Automation of Electronic Systems**, vol. 3, No. 4, p. 626-634, Oct. 1998.

BOUÉ, J.; PETILLON, P.; COUZET, Y. MEFISTO-L: a VHDL-based fault injection tool for the experimental assessment of fault tolerance. In: Twenty-Eighth Annual International Symposium of Fault-Tolerant Computing, 1998**. Proceedings of International Symposium of Fault-Tolerant Computing, Digest of Papers**, 1998. p. 168-173.

CABODI, G.; MURCIANO, M.; VIOLANTE, M. Boosting software fault injection for dependability analysis of real-time embedded applications. **ACM Transactions on Embedded Computing Systems**, vol. 10, No. 24, Dec. 2010.

CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. **IEEE Translactions of Software Engineering**, vol. 24, No. 2, Feb. 1998.

CIVERA, P. ; MACCHIARULO, L. ; REBAUDENGO, M. ; REORDA, S. M.; VIOLANTE, M. Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits. In: Defect and Fault Tolerance in VLSI Systems, 2001**. Proceedings 2001 IEEE International Symposium**, 2001. p. 250-258.

DODD, P. E.; MASSENGILL, MASSENGILL, L. W. M. Basic mechanism and modeling of single-event upset in digital microelectronics, **IEEE Transactions Nuclear Science**, vol. 50, no. 3, pt. 3, pp. 583–602, Jun. 2003.

ENTRENA L.; GARCÍA-VALDERAS, M.; FERNÁNDEZ-CARDENAL, R.; LINDOSO, A.; PORTELA-GARCÍA, M.; LÓPEZ-ONGIL, C. Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection. **IEEE Transactions on Computer**, vol. 61, No. 3, Mar. 2012.

FAROKH IROM. Guideline for Ground Radiation Testing of Microprocessors in the Space Radiation Environment, **Jet Propulsion Laboratory Publication**, Aug. 2008.

FIDALGO, A. V.; ALVES, G. R.; FERREIRA, J. M. A Modified Debugging Infrastructure to Assist Real Time Fault Injection Campaigns. **Design and Diagnostics of Electronic Circuits and Systems**, p. 172-177, April 2006.

FOLKESSON, P.; SVENSSON, S.; KARLSSON, J. Evaluation of the Thor Microprocessor Using Scan-chain-Based and Simulation Based Fault-Injection. In: 8th European Workshop on Dependable Computing, 1997.

FOURCARD, G.; PERONNARD, P.; VELAZCO R. Reliability Limits of TMR Implemented in a SRAM-based FPGA: Heavy Ion Measures vs. Fault Injection Predictions. In: 11th Latin American Test Workshop (LATW), 2010. p. 1-5.

GEISSLER, F.; KASTENSMIDT, F.; SOUZA, J. Soft Error Injection Methodology based on QEMU Software Platform. In: 15th Latin American Test Workshop (LATW), 2014. p. 1-5.

GUIBBAUD, N.; MILLER, F.; MOLIÈRE, F. New Combined Approach for the Evaluation of the Soft-errors of Complex IC, **IEEE Transactions of Nuclear Science**, Vol, 60, 2013.

GUNNEFLO, U. J. B.; JHONSSON, J.; KARLSSON, S. LOEB; TORIN J. A Fault Injection System for Study of Transient Effects on Computers. In: Technical Report No. 47, Dept. of Computer Engineering, 1987.

GUNNEFLO, U.; KARLSSON, J.; TORIN, J. Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. In: 19th International Symposium on Fault Tolerant Computing, 1989. **Proceedings of the IEEE 19th International Symposium on Fault Tolerant Computing**, 1989. p. 340–347.

HSU, CHUN-CHEN; LIU, PANGFENG; WANG, CHIEN-MIN; WU, JAN-JAN; HONG, DING-YONG; YEW, PEN-CHUNG; HSU, WEI-CHUNG. LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends. In: Parallel Processing (ICPP) International Conference**,** 2011. **Proceedings of Parallel Processing (ICPP) International Conference**, 2011. p. 226-234.

HSUEH, M.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. **IEEE Computer**, vol. 30, p. 75–82, 1997.

IYER, R. K. Experimental Evaluation. Special Issue FTCS-25 Silver Jubilee, **25th IEEE Symposium on Fault Tolerant Computing**, FTCS-25, pp. 115-132, June 1995.

JENN, E.; ARLAT, J. ; RIMEN, M. ; OHLSSON, J. Fault injection into VHDL models: the MEFISTO tool. In: Twenty-Fourth International Symposium, 1994. **Proceedings of Fault-Tolerant Computing FTCS-24**, 1994. p. 66-75.

KANAWATI, G. A.; NASSER A. K.; ABRAHAM, J. A. FERRARI: A Flexible Software-Based Fault end Error Injection System. **IEEE Transactions on Computers**, vol. 44, No. 2, Feb. 1995.

LÓPEZ-ONGIL, C.; GARCÍA-VALDERAS, M.; PORTELA-GARCÍA, M.; ENTRENA, L. Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation. **IEEE Transactions on Nuclear Science**, vol. 54, No. 1, Feb. 2007.

MANSOUR, W.; VELAZCO, R. SEU fault-injection in VHDL-based processors: A case study. In: 13th Latin American Test Workshop (LATW), 2012. p. 1-5.

MIREMADI G.; TORIN J. Evaluating Processor-Behavior end Three Error-Detection Mechanisms Using Physical Fault-Injection. **IEEE Transactions on Reliability**, vol. 44, 1995.

NICOLESCU, B.; VELAZCO, R.; REORDA, M. S. Effectiveness and limitations of various software techniques for "soft error" detection: a comparative study. In: On-Line Testing Workshop, 2001**. Proceedings of Seventh International On-Line Testing Workshop**, 2001. p. 172-177.

POTYRA, S.; SIEH, V.; CIN, M. D. Evaluating Fault-Tolerant System Designs using FAUmachine. In: Engineering fault tolerant systems EFTS '07, 2007**. Proceedings of the Engineering Fault Tolerant Systems Workshop**, 2007.

REED, R. A.; KINNISON, J.; PICKEL, J. C.; BUCHNER, S.; MARSHALL, P. W. KNIFFIN, S.; LABEL, KENNETH; LABEL A. Sinlge-Event Effects Ground Testing and On-Orbit, Rate Prediction Methods: The past, present and future. **Nuclear Science, IEEE Transaction** on Vol. 50, p. 622-634, 2003.

REORDA, S. M.; STERPONE, L.; VIOLANTE, M.; PORTELA-GARCIA, M.; LOPEZ-ONGIL, C.; ENTRENA, L. Fault Injection-based Reliability Evaluation of SoPCs. In: Test Symposium ETS '06, 2006. **Proceedings of Test Symposium ETS '06**, 2006. p. 75-82.

RHOD, E.; LISBOA, C.; CARRO, L.; REORDA, M. S.; VIOLANTE, M. Hardware and software transparency in the protection of programs against SEUs and SETs. **J. Electron Test**, no. 24, pp. 45–56, 2008.

SAMSON, J. R. Jr; MORENO, W.; FALQUEZ, F. Validating fault tolerant designs using laser fault injection (LFI). In: Defect and Fault Tolerance in VLSI Systems. **Proceedings of IEEE International Symposium**, 1997. p. 173-183.

VALDERAS, M. G.; PERONNARD, P.; ONGIL, C. L.; ECOFFET, R.; BEZERRA, F.; VELAZCO, R. Two Complementary Approaches for Studying the Effects of SEUs on Digital Processors. **IEEE Transactions on Nuclear Science**, vol. 54, Ago. 2007.

VELAZCO, R.; REZGUI, S.; ECOFFET, R. Predicting Error Rate for Microprocessor-Based Digital Architectures through C.E.U. (Code Emulating Upsets) Injection. **IEEE Transactions on Nuclear Science**, vol. 47, No. 6, Dec. 2000.

XU, J.; PING., X. The Research Of Memory Fault Simulation And Fault Injection Method For BIT Software Test. In: Second International Conference on Instrumentation & Measurement, Computer, Communication and Control, 2012. **Proceedings of the 2nd International Symposium on Computer, Communication and Control**, 2012. p. 718-722.

YANG, R. Freescale: Development Tools for i.MX Applications Processors, May 2013.

YI, L; PING, X.; HAN, W. A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing. In: 2nd International Symposium on Computer, Communication, Control and Automation, 2013. **Proceedings of the 2nd International Symposium on Computer, Communication, Control and Automation**, 2013**.**

YUI, C. C.; SWIFT G. M. SEU Mitigation Testing of Xilinx Virtex II FPGAs. In: Radiation Effects Data Workshop, 2003. **Proceedings of Radiation Effects Data Workshop**, 2003.

CHANG, J. Freescale Software Defined Network (SDN) Solutions Overview. Disponível em: http://cache.freescale.com/files/training/doc/dwf/DWF13_Freescale_Software_Defined_Network.pdf. Acesso em: janeiro, 2014.

FREESCALE SEMICONDUCTOR. Freescale, 2014. Disponível em: <http://www.freescale.com>. Acesso em: 2014.

GOOGLE INC. Using the Emulator. Disponível em: <http://developer.android.com/tools/devices/emulator.html>. Acesso em: maio, 2014.

LINUX KERNEL ORGANIZATION. Kernel, 2014. Disponível em: <https://www.kernel.org>. Acesso em: 2014.

LINUX PTRACE MAN PAGE. Ptrace, 2014. Disponível em: < https://www.linux.die.net/man/2/ptrace>. Acesso em: 2014.

MARGINEAN, A. Benchmark Virtualization Solutions for QorlQ Processors, Apr. 2014. Disponível em: <http://cache.freescale.com/files/training/doc/ftf/2014/FTF-SDS-F0028.pdf>. Acesso em: maio, 2014.

MENTOR GRAPHICS. ModelSim, 2014. Disponível em: <http://www.model.com/producst/fpga/model>. Acesso em: 2014.

NYSE: VMWARE. VMware, 2014. Disponível em: <http://www.vmware.com>. Acesso em: 2014.

XENOMAI PROJECT. Xenomai, 2014. Disponível em: <http://xenomai.org>. Acesso em: 2014.

APÊNDICE A – ARTIGO LATW (2014)

Um artigo relacionado ao tema da dissertação é apresentado neste apêndice. O artigo, denominado *Soft Error Injection Methodology based on QEMU Software Platform*, foi aceito para publicação no *Latin American Test Workshop* (LATW) no ano de 2014 para apresentação oral.