Paulo Ricardo Cechelero Villa

Reliability enhanced microprocessor architecture for the on-board computer of future satellites

Florianópolis
Abril de 2018

Paulo Ricardo Cechelero Villa

RELIABILITY ENHANCED MICROPROCESSOR ARCHITECTURE FOR THE ON-BOARD COMPUTER OF FUTURE SATELLITES

Tese submetida ao Programa de Pós-Graduação em Engenharia Elétrica, na área de concentração de Sistemas Embarcados, da Universidade Federal de Santa Catarina para a obtenção do Grau de Doutor em Engenharia Elétrica.
Orientador: Prof. Dr. Eduardo Augusto Bezerra
Coorientador: Prof. Dr. Fabian Luis Vargas

Florianópolis
Abril de 2018

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Villa, Paulo Ricardo Cechelero
Reliability enhanced microprocessor architecture for the on-board computer of future satellites / Paulo Ricardo Cechelero Villa ; orientador, Eduardo Augusto Bezerra, coorientador, Fabian Luis Vargas, 2018.
118 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós Graduação em Engenharia Elétrica, Florianópolis, 2018.

Inclui referências.

1. Engenharia Elétrica. 2. Tolerância a falhas. 3. Checkpoint Recovery. 4. Processador Soft-core. 5. FPGAs. I. Bezerra, Eduardo Augusto. II. Vargas, Fabian Luis. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

Paulo Ricardo Cechelero Villa

Reliability enhanced microprocessor architecture for the on-board computer of future satellites

Esta Tese foi julgada adequada para obtenção do Título de "Doutor", e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica.

Florianópolis, 27 de abril de 2018

Prof. Dr. Marcelo Lobo Heldwein
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Eduardo Augusto Bezerra
Orientador
Universidade Federal de Santa Catarina

Prof. Luigi Dilillo, Ph.D.
Université Montpellier (videoconferência)

Prof. Dr. Marcelo Daniel Berejuck
Universidade Federal de Santa Catarina

Prof. Dr. César Augusto Missio Marcon
Pontifícia Universidade Católica do
Rio Grande do Sul (videoconferência)

Dedico este trabalho ao amor da minha vida — Francine

ACKNOWLEDGEMENTS

Acknowledgments are in Portuguese since it is my native language.

Não teria como iniciar os agradecimentos se não pelo meu Orientador, Prof. Dr. Eduardo Augusto Bezerra. O Prof. Bezerra me deu a oportunidade de começar a trabalhar em um projeto de pesquisa em 2006 e, desde então, sempre me deu apoiou para seguir a carreira acadêmica. Foram muitas histórias, caronas, conversas aleatórias, correções de texto, churrascos e outros para resumir a lista, pois volta e meia, ao comentar algo que, até então era novidade para mim, recebia um retorno do tipo: *"Uhmm, pois é, já trabalhei com isso."*. Sou muito grato por ter a oportunidade de ter sido orientado desde a graduação, mestrado e agora doutorado pelo Professor, ressaltando que sem o seu suporte não teria chego até aqui. Para não me alongar muito, deixo aqui um muito obrigado e fico na expectativa de continuarmos contribuindo de forma conjunta.

Igualmente, agradeço ao meu Coorientador, Prof. Dr. Fabian Luis Vargas, que sempre me ajudou a ver a minha pesquisa de uma forma crítica. Seus conselhos e incentivos foram essenciais na produção de vários artigos e tese.

Diversas pessoas fizeram parte desse caminho. Muitos colegas de trabalho tornaram essa tese possível, em especial um agradecimento aos amigos Roger Goerl, Marcos Corino e Rodrigo Travessini. Minha família e amigos pessoais, que sempre me incentivaram com a pergunta clássica: "*Eai?! Falta muito para acabar o Doutorado?*". Minha esposa (*esposa?*), Francine, sempre presente para me dar apoio. Até mesmo colocando "outrossim" em um texto de engenharia. **Muito obrigado**.

A UFSC, CNPq, IFRS, funcionários e integrantes do LCS/GSE por fornecerem os recursos e auxílio financeiro necessário para o desenvolvimento do trabalho.

Aviso legal: estes agradecimentos foram escritos antes de passar pela banca, então...

"The fact that we live at the bottom of a deep gravity well, on the surface of a gas covered planet going around a nuclear fireball 90 million miles away and think this to be normal is obviously some indication of how skewed our perspective tends to be."

— Douglas Adams, The Salmon of Doubt

RESUMO

Processadores *soft-core* embarcados são a solução usual para lidar com interconexão de comunicação e dados dentro de FPGAs. Tarefas altamente paralelas implementadas em blocos de IP podem ser facilmente integradas com processadores durante o fluxo de desenvolvimento de FPGAs. No entanto, ao desenvolver aplicações espaciais, o projetista deve considerar os efeitos da radiação ionizante, principalmente sob a forma de SEUs. Os SEUs podem afetar os elementos de memória da aplicação, no qual o processador *soft-core* depende para funcionar corretamente. A maioria das técnicas de mitigação de SEUs em FPGAs são baseadas em redundância espacial de hardware. Notavelmente, a TMR é a mais comum. Quando implementado corretamente, o TMR pode mascarar erros únicos e detectar erros duplos. Em contrapartida, uma abordagem de tolerância a falhas muitas vezes negligenciada é usar redundância temporal. No caso de SEUs, ao reescrever um valor incorreto dentro de um registrador do processador pode restaurar o correto funcionamento do sistema. Este processo é feito ao custo do tempo de processamento em vez de replicação de hardware.

Esta tese apresenta uma técnica de tolerância a falhas, baseada no conceito de redundância temporal, com pontos de inspeção e recuperação para processadores soft-core. A arquitetura modificada proposta é voltada para sistemas embarcados para aplicações espaciais, com base em FPGAs. Nossos resultados experimentais mostram que a técnica CR é uma alternativa válida para TMR e até DMR, especialmente quando se considera a área de lógica limitada e o requisito de energia presente em um satélite. Os resultados têm níveis de confiabilidade comparáveis às técnicas mais convencionais de tolerância a falhas. Além disso, nossa abordagem não requer modificações no código-fonte ou compilador do software.

Palavras-chave: Tolerância a falhas, Checkpoint Recovery, Processador Soft-core, FPGAs, Efeitos Únicos, Single-Event Upsets.

RESUMO EXPANDIDO

Introdução

Não há como negar que os FPGAs (do inglês, *Field Programmable Gate Arrays*) estão aqui para ficar. Eles não são mais usados exclusivamente para prototipagem de ASICs (do inglês, *Application Specific Integrated Circuits*). Na verdade, eles são tão versáteis que, para até em aplicações mais conservadoras, como satélites, eles assumem cada vez mais o processamento de dados e controle de aviônica. O trabalho de (Gardenyes, 2012) indica que há uma tendência crescente no emprego de FPGAs em aplicações espaciais.

Os FPGAs atuais oferecem alta capacidade lógica (para implementar um circuito), razoáveis frequências de operação e uma grande quantidade de blocos embutidos (como conversores analógico-digital e processadores de sinais digitais). Diversos fatores contribuíram para o alcance deste estágio, porém a densidade de transistores dos Circuitos Integrados (Cis) constitui um fator principal, por força da miniaturização nos processos de fabricação.

Ao considerar aplicações espaciais, espera-se que futuras missões adquiram e processem grandes quantidades de dados. Além disso, a eletrônica embarcada deve ter a capacidade de ser reprogramada após o lançamento da missão e enquanto ainda estiver em operação. Microprocessadores tradicionais e ASICs não podem atender a esse requisito por completo, razão pela qual surgem os FPGAs como uma boa opção. Além dos blocos de propriedade intelectual (do inglês, *Intellectual Property Blocks* – IPs) personalizados dentro do FPGA, é comum o uso de processadores embarcados para lidar com dados e comunicações. Toda essa integração pode comprometer a confiabilidade geral do sistema. Dadas estas circunstâncias, encontrar um compromisso entre a capacidade de processamento e o nível de confiabilidade contra falhas do processador é importante do ponto de vista de pesquisa.

Levando em conta o ambiente hostil que os satélites estão expostos, eventos externos podem causar o mau funcionamento do sistema. Interferência eletromagnética e radiação são alguns dos responsáveis pelos efeitos que os circuitos estão suscetíveis. Um dos problemas mais comuns é conhecido como efeitos de evento único (do inglês, *Single Event Effect* – SEE), os quais podem causar falhas temporárias ou permanentes em um sistema, como por exemplo, invalidando-o e ocasionando o término prematuro de uma missão espacial.

Para atingir confiabilidade em nível de missão, a tolerância a falhas deve ser considerada em todo o desenvolvimento do sistema, ou seja, do *layout* do CI até a implementação do software. No nível mais baixo de abstração, os FPGAs endurecidos contra radiação (conhecidos como *rad-hard*) podem lidar com os efeitos da radiação no circuito, garantindo condições mínimas para o sistema funcionar.

Entretanto, para alguns programas espaciais, como no caso do Brasil, o processo de aquisição de componentes endurecidos é controlado por agências governamentais, além de custar significativamente mais do que os componentes de prateleira (do inglês, *Commercial Off-The-Shelf – COTS*) tradicionais. Se assumirmos um FPGA COTS não endurecido, o próximo nível de abstração do sistema deve mitigar possíveis erros (ou seja, SEEs) do hardware subjacente. Assim, uma forte motivação é a possibilidade de implementar sistemas tolerantes a falhas com o uso de FPGAs COTS.

Nesse sentido, o processador *soft-core* LEON3 vem sendo usado em algumas missões espaciais. Ainda, dado o interesse do Instituto Nacional de Pesquisas Espaciais (INPE) do Brasil em migrar do ERC32 sem ter que reprojetar todo o código, um processador *soft-core* LEON3, com tolerância a falhas, constituiria uma substituição válida para o ERC32.

Por todo o exposto, esta tese apresenta uma arquitetura microprocessada com tolerância a falhas, destinada a processadores *soft-core* que podem ser utilizados em aplicações espaciais embarcadas.

Objetivos

As interconexões de comunicação e dados dentro dos FPGAs são frequentemente tratadas com o uso de processadores *soft-core*. Tarefas altamente paralelas implementadas em blocos IP podem ser facilmente integradas com processadores durante o fluxo de desenvolvimento de FPGA. No entanto, ao desenvolver aplicações baseadas no espaço, o projetista de sistemas embarcados também deve considerar os efeitos da radiação ionizante, principalmente na forma de SEUs (do inglês, *Single Event Upset* – SEU). Por sua vez, os SEUs podem afetar os elementos de memória (registradores e *flip-flops*) do usuário e a memória principal, ambos necessários para o adequado funcionamento do processador.

A maioria das técnicas para mitigação de SEUs em FPGAs é baseada em redundância espacial de hardware. Notavelmente, a Redundância Modular Tripla (do inglês, *Tiple Modular Redundancy* – TMR) é a mais comum e, quando implementada corretamente, pode mascarar erros únicos e detectar erros duplos. Contudo, dependendo do

nível de implementação no processador, pode ser difícil recuperar a unidade defeituosa do TMR.

Uma abordagem de tolerância a falhas frequentemente negligenciada no escopo de processadores é a redundância temporal. No caso de SEUs, o processo de reescrever um valor incorreto dentro de um registrador do processador pode restaurar o funcionamento do sistema. Este processo é feito ao custo de tempo de processamento, em vez de replicar hardware.

De uma forma geral, a principal contribuição desta tese é uma técnica de tolerância a falhas, baseada no conceito de redundância temporal, com pontos de verificação e recuperação (do inglês, *Checkpoint and Recovery* – CR) voltados para processadores *soft-core*. A arquitetura modificada proposta não exige a refatoração do código fonte usado no processador original, destinando-se a sistemas embarcados para aplicações espaciais, baseadas em FPGAs.

Ademais, a pesquisa tem como foco demonstrar que a técnica de CR é uma alternativa válida para TMR e até mesmo para a redundância modular dupla (do inglês, *Dual Modular Redundancy* – DMR). Essas contribuições são especialmente importantes quando lidamos com restrições determinantes para aplicações espaciais: área lógica limitada e baixo consumo de energia. Todas essas restrições, no entanto, devem estar aliadas à níveis mínimos de confiabilidade.

Por fim, o desenvolvimento de técnicas de tolerância a falhas e recursos humanos especializados são indispensáveis para garantir soberania e independência ao programa espacial brasileiro.

Metodologia

A técnica de CR funciona salvando os pontos de verificação considerados seguros durante a execução de um processador. Sempre que um erro é detectado, é executada uma reversão para o último estado seguro conhecido, ou seja, uma recuperação. Assim, se o SEU ocorre em um elemento de memória do circuito e, se o elemento é sobrescrito com o valor correto após a identificação do SEU, o erro pode ser corrigido.

O modelo de falha assumido e que está sendo mitigado é o SEE, mais precisamente seu subtipo SEU. A literatura mostra que o SEU é a falha predominante quando considerados processadores. Para a técnica de CR, a granularidade dos pontos de verificação (*checkpoints*) precisa ser levada em conta, uma vez que introduz sobrecusto na execução do processador. Utilizamos a métrica que, após cada operação de escrita na

memória principal, o estado pode ser salvo, de forma semelhante ao trabalho de (Violante et al., 2011).

Uma vez que estamos lidando com SEUs, a memória principal do sistema é vital para mantê-lo funcionando. Como o programa é executado na memória principal, se esta apresentar erros, o processador pode interpretar equivocadamente as instruções. Para essa hipótese, a memória principal é considerada externa e protegida por uma técnica de detecção e correção de erros. Além disso, as memórias cache são desabilitadas por dois motivos: em razão de representarem áreas adicionais suscetíveis a SEUs e por interferirem na sincronização do processador, uma vez que usamos escritas na memória principal como pontos de referência para criação dos *checkpoints*.

Como qualquer outra técnica de tolerância a falhas, existem dois passos necessários para a sua implementação (detecção e correção do erro). A pesquisa propõe o uso da técnica de CR para detectar erros, através da execução repetida de cada fatia de instruções (compreendida entre dois pontos de verificação), além de uma terceira execução, se necessária, para corrigir um erro detectado. Não obstante, outros esquemas de detecção de erros são implementados para fins de comparação.

O LEON3 é o processador escolhido como sistema alvo deste trabalho, haja vista a sua significativa aceitação no escopo de aplicações espaciais. Finalmente, é importante mencionar que não há modificações fora do código VHDL do LEON3, o que significa dizer que o mesmo código fonte de software, compilado para o LEON3 original, pode ser executado perfeitamente em nossa arquitetura. A única diferença reside na forma como as instruções serão executadas e o sistema recuperado (no caso de um erro).

Resultados e Discussão

Para realizar nossos testes, o processador LEON3 foi simulado na ferramenta *Modelsim*. Uma campanha de injeção de falhas, baseada no modelo de falha única, foi executada usando um conjunto de quatro programas: *basic, bubble sort, NMEA e hamming*. É importante notar que não foi utilizado um programa de teste mais clássico (como *dhry, stanford* ou *whetstone*), pois o tempo de simulação se torna proibitivo.

Quatro arquiteturas foram consideradas para comparação, quais sejam, original, TMR, *Flow-control* (DMR) e *Time-redundant* (TR). Para as arquiteturas TMR e DMR, a taxa de execução sem falhas foi equivalente, na ordem de 79%, o que significa que a falha está latente ou não se manifestou. A taxa de falhas do original é ligeiramente inferior

aos valores de detecção nas abordagens TMR e DMR. Isso ocorre porque um erro detectado nem sempre se torna uma falha. A abordagem TR mostra a maior porcentagem de resultados corretos, na ordem de 95%. Isso se deve à reexecução do segmento de código, pois a falha injetada pode ser sobrescrita antes mesmo de se manifestar durante a execução do programa.

As taxas de correção para o TMR, devido ao uso do modelo de falha única, são sempre de 100% (para falhas detectadas). Na abordagem TR, a média de erros corrigidos está perto da marca de 98%, enquanto que a média do DMR fica ligeiramente acima de 63%. O principal problema com o DMR, em nossos testes, é a recuperação dos dois processadores corretamente. As médias de correção, considerando detecção e correção, são 92% e 95% para as abordagens DMR e TR, respectivamente. Para o TR, os 5% de falhas pode ser atenuado com técnicas adicionais em sinais internos específicos do LEON3.

Como comparação final, o custo total das técnicas foi calculado com base nos dados de taxa de detecção/correção e sobrecusto de tempo, área e potência. Nossos números mostram que a abordagem TR é comparável com a técnica de DMR, apresentando melhor fator *detecção recuperação*. Em relação à técnica de TMR, sua maior desvantagem é o fator *área potência*, mesmo apresentando 100% de taxas de detecção e recuperação, sem sobrecusto de tempo.

Considerações Finais

Nesta tese, apresentamos uma arquitetura modificada de processador tolerante a falhas, usando a técnica de recuperação temporal CR, com foco em aplicações espaciais baseadas em FPGAs.

Com nossos resultados experimentais, foi mostrado que a técnica CR é uma alternativa válida para TMR e DMR. Esta conclusão é válida também para a limitada área lógica e o consumo de potência, assuntos de interesse em satélites. As restrições são aliadas a níveis comparáveis de confiabilidade. Em nossa abordagem, não há necessidade de realizar modificações no código-fonte ou no compilador do software.

Palavras-chave: Tolerância a falhas, Checkpoint Recovery, Processador Soft-core, FPGAs, Efeitos Únicos, Single-Event Upsets.

ABSTRACT

Embedded soft-core processors are the usual solution to deal with network and data communications inside *Field Programmable Gate Array*s (FPGAs). High-parallel tasks implemented in *Intellectual Property* (IP)-blocks can be easily integrated with processors during the FPGA development flow. However, when developing space-based applications, the designer must consider the effects of ionizing radiation, mainly in the form of *Single-Event Upset*s (SEUs). SEUs can affect user flip-flops and memory where the soft-core processor relies on to function properly. The majority of techniques for mitigation of SEUs on FPGAs are based on hardware spatial-redundancy. Notably, *Triple Modular Redundancy* (TMR) is the most common. When implemented correctly, TMR can mask single-errors and detected-double errors. In contrast, an often neglected fault-tolerance approach is to use time-redundancy. In the case of SEUs, when rewriting an erroneous value inside a processor register can restore the system correctness. This process is done at the cost of processing time instead of hardware replication.

This thesis presents a fault-tolerance technique, based on the concept of temporal redundancy, with checkpoints and recovery for soft-core processors. The proposed modified architecture is aimed at embedded systems for spatial applications, based on FPGAs. Our experimental results show that the *Checkpoint and Recovery* (CR) technique is a valid alternative to TMR and even *Dual Modular Redundancy* (DMR), especially when considering limited logic area and power budget present on a satellite. The results have comparable levels of reliability to the more conventional fault-tolerance techniques. Additionally, our approach does not require modifications to the software source code or compiler.

Keywords: Fault-tolerance, Checkpoint Recovery, Soft-core Processors, FPGAs, Single-Event Effects, Single-Event Upsets.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADC	*Analog-to-Digital Converter*
ALU	*Arithmetic Logic Unit*
AMBA	*Advanced Microcontroller Bus Architecture*
ASIC	*Application Specific Integrated Circuit*
ASIP	*Application Specific Instruction Processor*
BRAM	*Block-RAM*
COTS	*Commercial Off-The-Shelf*
CR	*Checkpoint and Recovery*
CRAM	*Configuration-RAM*
CRC	*Cyclic Redundancy Check*
DCLS	*Dual-Core Lock Step*
DMR	*Dual Modular Redundancy*
DSP	*Digital Signal Processor*
DSU	*Debug Support Unit*
DUT	*Device Under Test*
ECC	*Error Correction Code*
EDAC	*Error Detection And Correction*
EMI	*Electromagnetic Interference*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
FT	*Fault Tolerance*

GEO *Geosynchronous Earth Orbit*

GPIO *General Purpose Input Output*

HDL *Hardware Description Language*

IC *Integrated Circuit*

INPE *National Institute For Space Research*

IP *Intellectual Property*

IU *Integer Unit*

LEO *Low Earth Orbit*

MCU *Multiple Cell Upset*

MBU *Multiple Bit Upset*

MOS *Metal-Oxide-Semiconductor*

MPSoC *Multi-Processor* System-on-Chip *(SoC)*

OS *Operating System*

OBC *On-Board Computer*

PR *Partial Reconfiguration*

RP *Reconfigurable Partition*

RTL *Register-Transfer Level*

SEC-DED *Single-Error Correction / Double-Error Detection*

SEE *Single-Event Effect*

SEFI *Single-Event Function Interrupt*

SEGR *Single-Event Gate Rupture*

SEL *Single-Event Latch-up*

SET *Single-Event Transient*

SEU *Single-Event Upset*

SoC *System-on-Chip*

SRAM *Static Random Access Memory*

TMR *Triple Modular Redundancy*

TR *Time Redundancy*

TID *Total Ionizing Dose*

CONTENTS

1 INTRODUCTION

There is no denying *Field Programmable Gate Array*s (FPGAs) are here to stay (RODRIGUEZ-ANDINA; VALDES-PENA; MOURE, 2015). They are not used any longer exclusively for prototyping of *Application Specific Integrated Circuit*s (ASICs) (ALKHAFAJI et al., 2018). In fact, they are so versatile that, for one of the most conservative applications — satellites — they have been increasingly taking over the data processing and avionics control (BOUHALI et al., 2017).

The work of (GARDENYES, 2012) was used to generate the chart in Fig. 1 that shows a comparison of the use of ASICs, FPGAs, Microprocessors and Standard ASICs in 11 satellite missions. Except for two missions (Immarsat 4 and Galileo IOV)[1], the amount of FPGAs out-stands the number of ASICs used. There is a growing trend in the employment of FPGA on space applications (FRIEND; ARROYO; HANSEN, 2016).

Today's FPGAs offer high logic capacity (to implement a circuit), reasonable operating frequencies and a plethora of embedded hard-blocks (such as *Analog-to-Digital Converter*s (ADCs) and *Digital Signal Processor*s (DSPs)) (EEJournal, 2017). Several factors have contributed to reach this stage, mainly the *Integrated Circuit* (IC) transistor density due to manufacturing process scaling down (TORRENS, 2017). Nonetheless, ASICs have their place on the market, especially on high-volume and high-performance applications such as SAMPA Chip (BARBOZA et al., 2016).

When considering space applications, future satellite missions are expected to acquire and process large amounts of data (NORTON et al., 2009). Additionally, onboard electronics are required to be re-programmable after the mission launch and even further, while still operating. Traditional microprocessors and ASICs cannot fulfill this requirement entirely, leaving FPGAs as the primary option.

[1] Both Immarsat 4 and Galileo IOV are communication satellites, hence the use of dedicated custom ASICs.

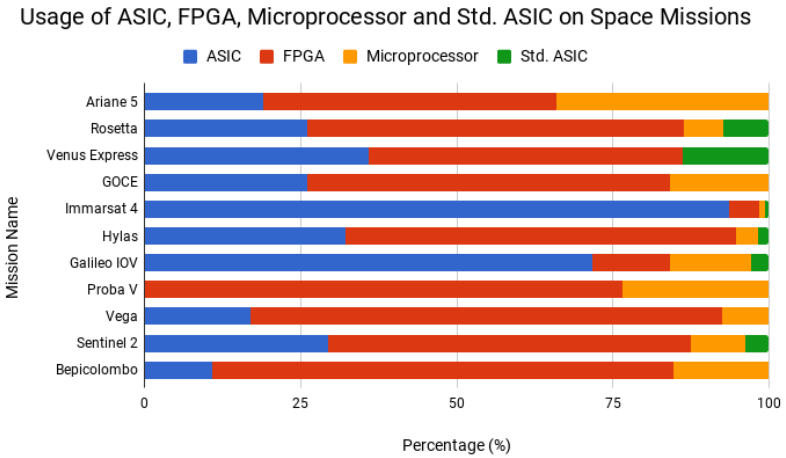

Figure 1 – Usage of ASIC, FPGA, Microprocessor and Standard ASIC on Space Missions
Source: (GARDENYES, 2012)

Apart from custom IP[2]-blocks inside the FPGA, it is common to have embedded processors (KLETZING et al., 2013; GLEIN, 2014; WILSON, 2011; GUZMÁN et al., 2011; Aeroflex Gaisler, 2015c) to handle data and communications. All this integration can compromise overall system reliability (BERNARDESCHI; CASSANO; DOMENICI, 2015; SABENA et al., 2014). Given these circumstances, finding a compromise between the processing capacity and the level of reliability against processor failures is important from the research point of view.

Given the harsh environment satellites are exposed to, external events can cause the system to malfunction. *Electromagnetic Interference* (EMI) and radiation account for effects that the circuits are susceptible. One of the most common problems is known as *Single-Event Effect* (SEE)(TANG; OLSSON, 2003; BAUMANN, 2003), which can cause temporary or permanent failures in a system, even with the potential to cause invalidation of the entire system, in the form of the premature

[2] *Intellectual Property* (IP)

termination of a space satellite mission, for example.

To attain mission-level reliability, fault-tolerance must be considered throughout the entire design of the system, i.e., from IC layout to software implementation. On the lower level of abstraction, radiation hardened (rad-hard) FPGAs can deal with the effects of radiation on the circuit, assuring minimal conditions to the system to function.

But, for some space programs, such as the case in Brazil, the acquisition process of radiation hardened (rad-hard) components is controlled by government agencies, besides costing significantly more than traditional *Commercial Off-The-Shelf* (COTS) components. For instance, *International Traffic in Arms Regulations* (ITAR) (U.S. State Department, 2018) rules[3]. If we assume an unhardened COTS FPGA, the next level of abstraction of the system must mitigate possible errors (i.e., SEEs) from the underlying hardware. Hence, a strong motivation is the possibility to implement fault-tolerant systems with the use of COTS FPGAs.

On that matter, the LEON3 (Aeroflex Gaisler, 2015b) processor has been used for a few missions on space. And given Brazil's *National Institute For Space Research* (INPE) interest in migrating from the ERC32[4] without having to redesign the entire code, a soft-LEON3 processor with fault-tolerance is a good substitution for the rad-hard ERC32.

Case in point, INPE's Multi-mission satellite platform (INPE, 2018) *On-Board Computer* (OBC) employs the ERC32 microprocessor, and the adoption of a compatible SPARC architecture (i.e., LEON3) can reduce redesign of the source code from scratch.

For these reasons, this thesis presents a microprocessor architecture with fault-tolerance, aimed at soft-core processors that can be used on embedded space applications.

[3] Given the two major FPGA companies are based on the USA.

[4] ERC32 is a discontinued radiation-tolerant SPARC V7 processor developed for space applications.

1.1 OBJECTIVES AND CONTRIBUTION

Network and data communications inside FPGAs are often handled with the use of soft-core processors(KLETZING et al., 2013; GLEIN, 2014; WILSON, 2011; GUZMÁN et al., 2011). High-parallel tasks implemented in IP-blocks can be easily integrated with processors during the FPGA development flow. However, when developing space-based applications, the designer of embedded systems must also consider the effects of ionizing radiation, mainly in the form of *Single-Event Upset*s (SEUs) (BERNARDESCHI; CASSANO; DOMENICI, 2015; SABENA et al., 2014). SEUs can affect user flip-flops and memory where the soft-core processor relies upon to function properly.

The majority of techniques for mitigation of SEUs in FPGAs are based on hardware spatial-redundancy. Notably, *Triple Modular Redundancy* (TMR) is the most common. When implemented correctly, TMR can mask single-errors and detected double-errors. But, depending on the level of implementation for a processor, it can be hard to recover the faulty unit.

Therefore, an often neglected fault-tolerance approach in the scope of processors is to use time-redundancy. In the case of SEUs, when rewriting an erroneous value inside a processor register can restore the system correctness (KOREN; KRISHNA, 2010). This process is done at the cost of processing time instead of hardware replication.

In general, this thesis' main contribution is **a fault-tolerance technique**, based on the concept of temporal redundancy, **with checkpoints and recovery aimed at soft-core processors**. The proposed **modified architecture does not require the redesign of code**[5] and is **aimed at embedded systems for space applications, based on FPGAs**.

The research is intended to demonstrate that the *Checkpoint and Recovery* (CR) technique is a valid alternative to TMR and even *Dual Modular Redundancy* (DMR). This contribution is especially important

[5] That is, the same compiled code for the standard processor can be used here in the modified architecture.

when dealing with determinant constraints for space applications: limited logic area and power budget. All of these constraints are allied to comparable levels of reliability.

Lastly, the development of fault-tolerance techniques and specialized human resources are necessary to the sovereignty and independence of the Brazilian's space program.

During the doctorate studies, I have worked with space-related applications. The outcome publications on conferences and journal paper are presented in Appendix A.

1.2 TEXT ORGANIZATION

The remaining of this document is organized as follows: **Chapter 2** gives a background of main concepts related to space-based applications, and related works are discussed. **Chapter 3** shows in detail the test vehicle and the implementation process for the proposed technique. **Chapter 4** presents how we performed the experiment on the proposed technique and analyze the results for detection and recovery capability; time, area, and power overhead. **Chapter 5** demonstrate current improvements that can contribute to the the proposed to attain space-grade qualification. Finally, the **Chapter 6** concludes this work and gives an outlook on future works.

2 RELIABILITY IMPROVEMENT STRATEGIES FOR MICROPROCESSORS

This chapter presents the main problems and definitions associated to the space environment when considering embedded electronic circuits. Also, some of the techniques used in this context are described with focus on the time redundant approach. And lastly some of related works are discussed and compared in this scope.

2.1 RADIATION EFFECTS ON ELECTRONICS

The effects of radiation come from particles emitted from a variety of sources (such as solar rays, etc.), causing degradation of electronic components, faulty logic, or even damaging the component. Significant advances were obtained with the alternative of the hardening of electronic components, at all project levels, which, however, did not exempt the radiation effect.

Space radiation is generated by particles emitted from various sources that go beyond the solar system. There are three main sources of charged particles responsible for faults in electronic components, namely: Cosmic Rays, Solar Winds and Van Allen's Belt (VELAZCO; FOUILLAT; REIS, 2007; BATTEZZATI; STERPONE; VIOLANTE, 2010). Cosmic Rays are formed of highly energetic ion nuclei, these heavy ions represent only 1% of the component of cosmic radiation, being the remaining 83% protons, 13% helium nuclei and 3% electrons.

The fusion process inside the sun produces a large number of protons and electrons. These particles travel in space through the Solar Wind. The Solar Wind is radiated by the Sun in all directions. Variations in the solar crown due to the rotation of the Sun and its magnetic activities make the Solar Wind variable and unstable.

In addition to the particles provided by the Sun, Solar Winds carry particles of other stars and highly charged ions generated by phenomena such as Supernova. Supernova is the name given to the celestial bodies arisen after the explosions of stars that produce extremely bright objects.

All these charged particles are influenced by the magnetic fields of the planets forming radioactive belts around them. In the case of Earth, the belt is known as the Van Allen Belt. The Van Allen Belt is made up of electrons and protons trapped in Earth's gravitational field. Fig. 2 illustrates the influence of charged particle sources on Earth.

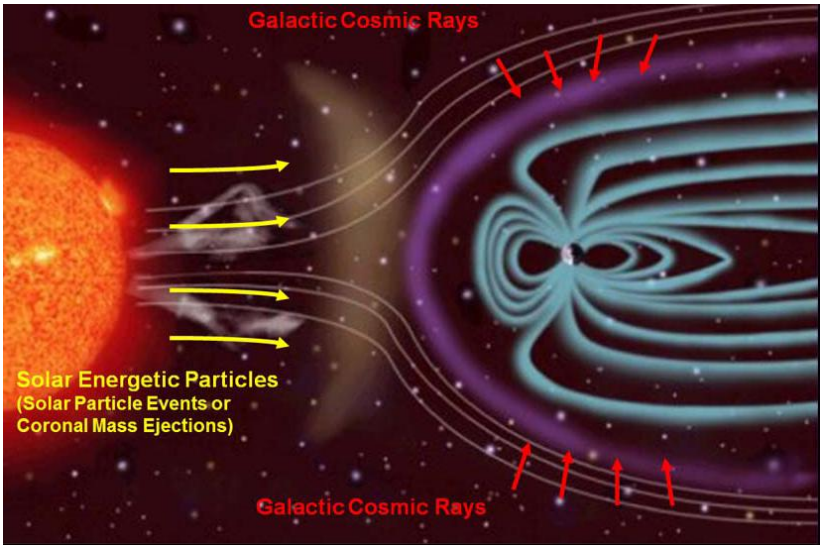

Figure 2 – Sources of Ionizing Radiation in Interplanetary Space
Source: (NASA/JPL-Caltech/SwRI, 2018)

All these sources of radiation interact with electronics causing different effects on integrated circuits. (SIEGLE; VLADIMIROVA, 2015) presents the common radiation effects that must be mitigated on FPGA in the form of a tree in Fig. 3.

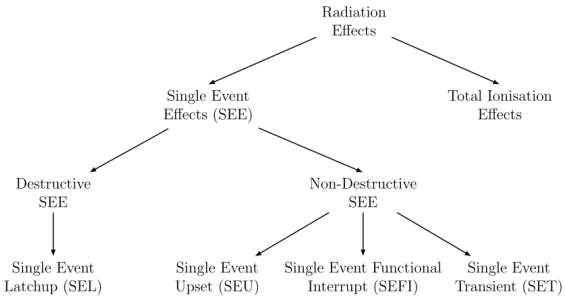

Figure 3 – Radiation effects on integrated circuits
Source: (SIEGLE; VLADIMIROVA, 2015)

The SEE is detailed in Section 2.3. The effect called *Total Ionizing Dose* (TID) is depicted in detail on Fig. 4. The authors (BARNABY, 2006) describe the effects of TID on *Metal-Oxide-Semiconductor* (MOS) transistor as:

> *"... Fig. 4(a) shows the normal operation of a MOSFET. The application of an appropriate gate voltage causes a conducting channel to form between the source and drain so that current flows when the device is turned on. In Fig. 4(b), the effect of ionizing radiation is illustrated. Radiation-induced trapped charge has built up in the gate oxide, which causes a shift in the threshold voltage (that is, a change in the voltage which must be applied to turn the device on). If this shift is large enough, the device cannot be turned off, even at zero volts applied, and the device is said to have failed by going depletion mode."(BARNABY, 2006)*

Both radiation effects (SEE and TID) needs to be taken into consideration when designing systems for space applications. However, each of them has different approaches to be mitigated, and they are also connected with the underlying technology/topology of the system (e.g., Flash memories are more susceptible to TID while *Static Random Access Memory* (SRAM) are more vulnerable to SEEs).

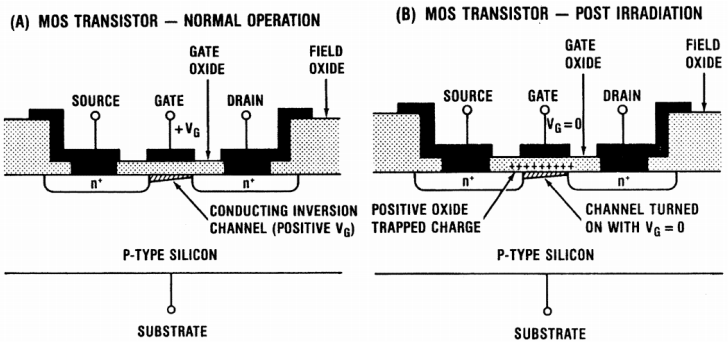

Figure 4 – Schematic of n-channel MOSFET illustrating radiation-induced charging of the gate oxide: (a) normal operation and (b) post-irradiation.

Source: (BARNABY, 2006)

2.2 FAULT, ERROR AND FAILURE

For this section, the concepts are in agreement with (AVIZIENIS et al., 2004), according to which, a system can be seen as an entity that interacts with other entities, that is, other systems, including hardware, software, humans and the physical world with their environment. These entities, then, are the environment of a given system.

The service delivered by a system is its behavior in the way it is perceived by its user(s) so that the correct service is provided when the system implements its function. Thus, a system failure, or failure, is in the event of the delivered service diverting from the correct service, i.e., not performing the function of the system.

The period in which the incorrect service delivery occurs is known as service interruption, just as a transition from the wrong service to the correct service is called service restoration. An error is a system state that can lead to a service failure. However, not all errors can lead to the system state that causes a failure. It is the error that classifies the fault as active or inactive. The definition of reliability is the ability to avoid service failures that are more frequent or severe than is acceptable.

The relationship between failure, error, and failure, in the form

of a chain of threats, can be seen according to Fig. 5.

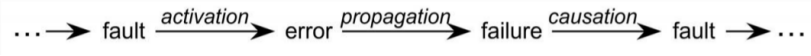

Figure 5 – Error Propagation - Relationship cause/effect between fault, error and failure.
Source: (AVIZIENIS et al., 2004)

The system is operating until a fault is activated (so far, the fault may be inactive, or an external fault generates the activation), generating an error, i.e., leading this system to the error state. The failure occurs when the error situation is propagated, causing this service to fail to deliver the expected result (deviate from the correct service).

Using as an analogy a processor where the *Arithmetic Logic Unit* (ALU) has one of the fixed output bits at a logical level. The fault is the bit that does not change. The error is the result of the failure (a sum with the wrong value for example). And the failure is when another processor unit uses the result with error, propagating the problem to the rest of the system.

Several techniques have been developed in the last decades to obtain reliability in computational systems, which can be divided into four main groups:

- Fault prevention: to prevent the occurrence or introduction of faults.
- Fault tolerance: to avoid service failures in the presence of faults.
- Fault removal: to reduce the number and severity of faults.
- Fault forecasting: to estimate the present number, the future incidence, and the likely consequences of faults.

2.3 SINGLE EVENT EFFECTS

The error caused by radiation known as SEE can be further classified as soft-error and hard-error, and subdivided into the following terms (BATTEZZATI; STERPONE; VIOLANTE, 2010):

- Soft-Errors
  - *Single-Event Transient* (SET)
  - SEU
  - *Multiple Cell Upset* (MCU)
  - *Single-Event Function Interrupt* (SEFI)
- Hard-Errors
  - *Single-Event Latch-up* (SEL)
  - *Single-Event Gate Rupture* (SEGR)

The effect called SET occurs when a high energy particle reaches a certain point in the circuit, with the ability to change the output of a transistor. This changes the signal level for a period (in the order of nano/picoseconds), causing a glitch. As the name implies, it is transient, that is, there is a double transition (0 - 1 - 0 or 1 - 0 - 1) within this space of time. The effect of the SET is shown in Fig. 6, where a fault is indicated in the upper left AND gate, the transition of the output can be perceived in the third logical port, where the undesired effect occurs.

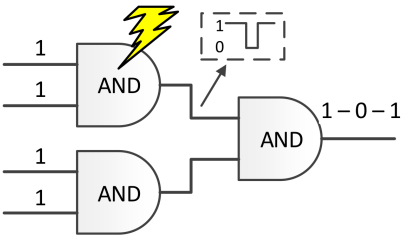

Figure 6 – SET Example
Source: (GOLOUBEVA et al., 2006)

SEUs occur on the assumption that the particle reaches an element of memory by changing the stored data. The SEU is not considered permanent because, in the next write operation of the memory element affected, the wrong value will be replaced. However, if the memory element is read-only from the system, the error can be propagated to the rest of the circuit and thus be considered a permanent error. Another

situation of occurrence of SEUs is in the case where the SET propagates until it reaches a memory element, storing the undesired value.

When more than one SEU happen on a circuit, the effect is called the MCU. And if it occurs on memory elements that make up a larger register, the *Multiple Bit Upset* (MBU) is determined.

The SEFIs are the result of the particle that reaches a region that contains functional control elements of the circuit. This effect can affect reset signals or read/write enable signals in a memory, for example, and may leave the element not functional.

If there is a destructive SEE, it can configure the occurrence of an SEL - when the output of an element is at a fixed logical level (high or low) independent of the input - or a SEGR - situation in which a particle causes the rupture of the gate of the transistor, leading to an increase of the leakage voltage. In both cases, an undesired increase in current consumption, heat dissipation, and even complete component disruption can occur, i.e., to burn the circuit.

2.4 FAULT-TOLERANCE TECHNIQUES

The techniques for fault tolerance integrate a line of research in the area of reliability of systems, being the subject well-established computational systems (GOLOUBEVA et al., 2006; KOREN; KRISHNA, 2010). It is safe to say that there are no 100% fault-tolerant systems (BATTEZZATI; STERPONE; VIOLANTE, 2010; GOLOUBEVA et al., 2006); several factors are involved, and therefore there will always be a variable that can not be predicted or controlled.

In the specific case of embedded systems for space applications, fault tolerance is not only a necessity but an indispensable design requirement to increase the chances of success of the mission.

The techniques commonly used for fault tolerance use the concept of redundancy, which can be defined as the existence (logical or physical) of more than one resource needed to perform the action that must be fault tolerant. Although the word redundant, when used in the context of computational systems, can represent the idea of physical repli-

cation of components, there are four basic types of redundancy(KOREN; KRISHNA, 2010):

- Hardware: The most natural concept to replicate hardware and use it whenever a fault is identified.
- Software: To be used in software failures, there may be two or more code snippets running to prevent failure.
- Information: When the basic information is inflated with redundant data, like the checksum in a register, for fault tolerance.
- Time: Use redundancy in time to tolerate failure, i.e., perform the same activity two or more times, one after the other, to ensure a correct result.

Examples of hardware redundancy can be simple implementations - such as the addition of duplicated circuits, one of which is used as the primary circuit and the remaining redundant. When the fault is detected, the logic is switched to use one of the redundant circuits. This type of technique is known as static hardware redundancy.

We can also find hybrid solutions, where all redundant circuits are active (performing the same function). The correct output (fault-free) is selected through a majority voter. Fig. 7 presents the idea of hardware redundancy with a voter, also known as TMR. The TMR technique can be applied at several levels of abstraction, such as architecture (the ALU within a processor) or at lower levels in the system.

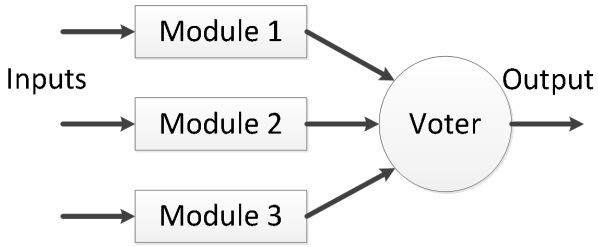

Figure 7 – TMR overview
Source: (GOLOUBEVA et al., 2006)

In the case of software redundancy, it is possible to have variations at all levels, such as data, program flow control, and hybrid combinations. For example, we perform the same task for two different versions of software with the same objective. If there is a divergence of results, an action is taken.

For information redundancy, the most explicit example would be to add data to information of interest, to identify, mask, and tolerate errors. The data coding technique, known as a checksum, calculates the data (as a *xor* operation) and adds the result to its end before transmitting or using it. Once coded, one must make the same calculation and compare with the attached result, in which case, if there is a divergence, the failure can be identified.

Finally, temporal redundancy is the repetition of the computation of the same task over time, with the results of each of the repetitions being compared, to be able to identify the fault. Fig. 8 shows the temporal execution of a computation, which in the end is compared to identify the error.

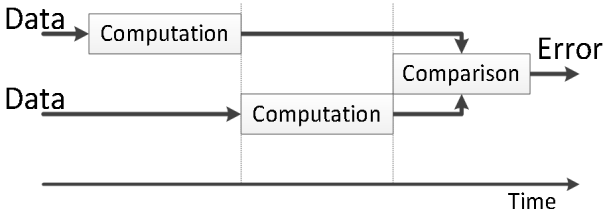

Figure 8 – TR overview
Source: (GOLOUBEVA et al., 2006)

The rollback recovery technique is done by performing checkpoints during the execution of a program, at specific intervals. Assuring that these points do not contain errors, in the event of a failure, the system can return to the last checkpoint and redo the execution.

The technique of interest in this work is based on the concept of inspection points (Checkpoint), which has a slightly simple idea, exemplified by an analogy: Suppose that it is necessary to sum the total

of ten items from a shopping list with the use of a calculator. To obtain the total value, one must add the ten values. However, if in the ninth step a button is pressed incorrectly, it will be necessary to repeat all of ten operations. The CR technique, in this analogy, would be, after a given amount of sums - for instance, six - the partial result copied to another place, i.e., perform a checkpoint. Thus, if an error occurs after the sixth sum operation, merely repeat the sums starting from the last partial value (recovery) and continue the operations.

Checkpoint and recovery can be done in computer systems, such as processors, simply by saving the state of interest and, if an error is detected, return to that state to redo the execution. Therefore, checkpoint and recovery integrate the techniques of the type of time redundancy.

Considering that there is no single taxonomy for fault tolerance techniques, this section was intended to demonstrate one of the possible approaches to the subject.

2.4.1 Checkpoint Recovery

The CR technique is a classic fault-tolerance technique, which enables computing systems to execute correctly even when affected by transient faults (SIEWIOREK; SWARZ, 2017; HENKEL et al., 2013). The works based on the technique of CR are traditionally classified according to the level of abstraction implemented by the system. This classification is divided into techniques that make changes to software-only or hardware-only (CETIN et al., 2016). While software solutions are cheaper from the perspective of implementation, purely hardware based have a very low overhead potential in the execution time of the same software. It is also possible to have a combination of both, denominated hybrid (hardware and software).

Although the concept of the technique is simple, several problems arise with the implementation, especially when taking into account the essential details of the development, as exemplified:

- What level of abstraction (user, *Operating System* (OS), instruc-

tion, hardware) should it have?
- What are the pros and cons of each level?
- How transparent should the checkpoint be?
- How many checkpoints should be made?
- At what points in the program should we checkpoint?

This is not an extensive list of problems, but it is possible to have an idea of those intrinsic to the technique. Once you know which target system to apply the technique and how this system should behave, some of the issues can be addressed immediately.

In the sections that follow, the points that must be taken into account when applying a CR technique to a processor will be presented.

2.4.2 Checkpoint Overhead

Like all redundancy-based techniques, there is an associated overhead, whether temporal or physical. In the case of the CR technique applied to a processor, the overhead is associated with the additional execution time of the program, while there are no errors. In other words, the amount of time when the system is blocked from execution to perform a checkpoint. Fig. 9 demonstrates the additional execution required on a system with CR. The execution of the program with the CR has points where it is necessary to perform the checkpoint, represented in grey tone in the figure. When execution is interrupted to the checkpoint, the same program suffers an addition at runtime.

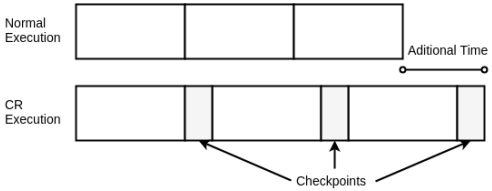

Figure 9 – CR overhead
Source: (GOLOUBEVA et al., 2006)

2.5 RELATED WORKS

The works developed by (KELLER; WIRTHLIN, 2017) and (LINDOSO et al., 2017) present combined fault-tolerance techniques applies to the LEON3 soft-core processor for FPGAs.

(KELLER; WIRTHLIN, 2017) uses five different SEU mitigation variations: no SEU mitigation, TMR alone, TMR with *Block-RAM* (BRAM) scrubbing, TMR with *Configuration-RAM* (CRAM) scrubbing, and TMR with both BRAM scrubbing and CRAM scrubbing. Both fault injection and neutron radiation testing were conducted. Improvement is measured in terms of sensitivity reduction for fault injection and cross section reduction for neutron radiation testing when compared to the unmitigated design. The results from both fault injection and radiation testing demonstrate that each variation of SEU mitigation techniques improve the SEU sensitivity of the LEON3, and that improvement increases as more mitigation techniques are combined. When compared to the unmitigated design, SEU sensitivity is improved from 16 to up 50 times. The full mitigated version comes at a cost of 4.7 times increase in area of the FPGA.

(LINDOSO et al., 2017) implements a hybrid fault-tolerant LEON3 soft-core processor in a Xilinx Artix-7 FPGA and evaluated its error detection capabilities through neutron irradiation and fault injection. The error mitigation approach combines the use of *Single-Error Correction / Double-Error Detection* (SEC-DED) codes for memories, a hardware monitor to detect control-flow errors, software-based techniques to detect data errors and configuration memory scrubbing with repair to avoid error accumulation. Radiation test results shows an improvement of 4.13 times for the hardware-only mitigation techniques. Fault-injection tests includes the software hardened approach in combination to hardware and have an average 20 times better improvement. Both results are compared against the unmitigated variant of the processor.

The authors in (LI et al., 2017) propose a transient-fault countermeasure called RELI, which is a fine-grained CR approach for

Application Specific Instruction Processor (ASIP)-based embedded processors. RELI is supposed to be the first to realize CR at the basic-block level by leveraging custom instruction design. To implement RELI, an ASIP design flow based on one of the existing commercial tool (ASIP-meister), generate the *Register-Transfer Level* (RTL) description of the resultant processors with RELI functionality. The costs concerning execution time, area, and power are reduced significantly compared to existing techniques.

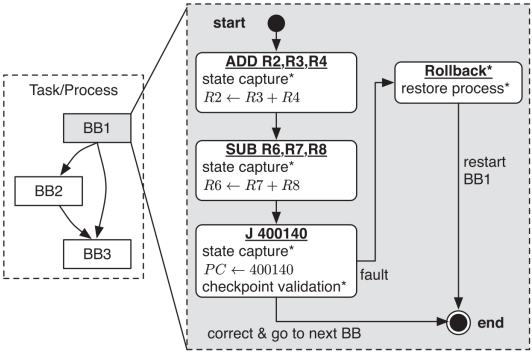

Figure 10 – Overview of RELI functionality
Source: (LI et al., 2017)

The augmented processor (i.e., RELI processor) allows CR to be executed at a finer granularity than other works, such that the checkpoint data size is reduced. Assembly code from MiBench benchmark suite (GUTHAUS et al., 2001), compiled using SimpleScalar toolset is used to generate the comparisons. The experimental results show that the fault-free execution time overhead is 0.76 percent on average. In the fault injection test, for the worst case, the recovery time is 62 cycles. RELI costs 44.4 percent area and 45.6 percent leakage power overhead on average (for the TMSC65nm technology), and 79.3 and 77.8 percent in the worst case found in SPEC-INT2006 and MiBench suites. This work is a complete flow continuation from the work in (RAGEL; PARAMESWARAN, 2012) discussed further in this chapter.

(T. Li et al., 2016) presents another work, aimed at the em-

bedded processor internal registers. The register data dependency is used to minimize the register file traffic required by the register file CR. The proposed logging CR scheme, named RECORD, considers various register data dependencies, which can potentially identify and eliminate the redundant executions of register file checkpointing at runtime. This approach is supposed to be the first to realize a hardware-based logging checkpointing mechanism, which strategically utilizes the first processor executions to diminish the additional checkpointing operations at runtime, for embedded processors. RECORD is implemented in an ASIPs to evaluate the proposed scheme for embedded processors. The technique presents a lower register file traffic and better dynamic power saving with little hardware and performance overhead when compared to other works.

The authors in (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017) propose a *Dual-Core Lock Step* (DCLS) approach to increase the dependability of hard-core processors embedded in programmable *System-on-Chip*s (SoCs), which combines the programmable logic with the high-performance hard-core processor. The DCLS is a dual-core ARM Cortex-A9 processor embedded into the Zynq-7000 APSoC. It is a novel implementation of lockstep in the dual-core Cortex-A9. ARM provides some processors versions with built-in lockstep, such as Cortex-R5 processor, which could be configured to application reliability. An overview of the proposed system is depicted in Fig. 11, note that PS and PL stand for Processing System and Programmable Logic, respectively.

Two versions of the technique are compared with the unhardened Cortex-A9 processor. The first uses only the BRAMs to store the checkpoint data, and the second uses the external DDR memory as secondary storage for the checkpoint data. Area results show an increase of 100% for the processor and memories. As for the execution time, three matrix multiply programs are evaluated. Being the longer the execution time, lower is the time overhead. The BRAM version has an increase of 26%, and the DDR version has a 47% increase on the total clock cycles for the 20x20 matrix. The further work by the authors in (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017) shows that up to

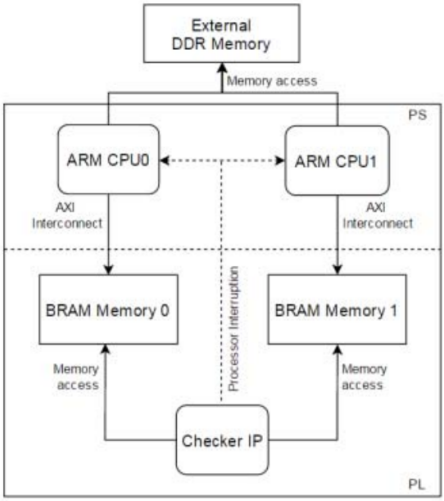

Figure 11 – Proposed DCLS Architecture for dual-core ARM Cortex-A9
Source: (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017)

91% of the bit flips injected in the ARM registers are mitigated by the technique.

The work presented by (VIOLANTE et al., 2011) present a design flow that can be used by designers to mitigate radiation-induced errors affecting processor IP cores embedded in FPGA-based SoCs for systems that have to be deployed in harsh environments. The design flow used the concepts of lockstep, checkpoint with rollback recovery, and on-demand configuration memory scrubbing (in case of SRAM-based FPGAs) to provide a balance between resources overhead and fault tolerance. The flow can be automated, reducing the total development costs, while increasing the quality of the resulting product. The authors provide a prototypical implementation of a design environment, supporting the proposed flow, and applied it to the design of a system using a Leon processor IP core. An overview of the proposed system is presented in Fig. 12.

The overhead on time for this implementation ranges from 17% to 54%, depending on the software being executed. In the fault injection

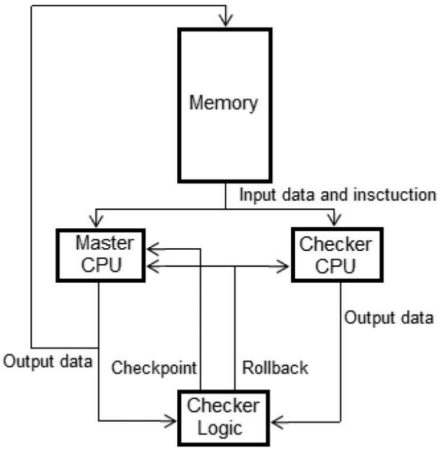

Figure 12 – Overview of the system in (VIOLANTE et al., 2011)
Source: (VIOLANTE et al., 2011)

campaign, 10.000 random SEEs were injected, being 84% of them being latent or detected and corrected; 15% triggered errors in the system (the authors modified the instruction trap of the processor to perform a rollback), and the configuration memory scrubber handled the last 1%.

The following subsections present older, but also necessary, works related to CR implementations in computational systems.

2.5.1 Hardware Approaches

The work developed by (KANG et al., 2014) aims for the best positioning to perform checkpoints on a CR technique implemented together with DMR to tolerate transient errors. The target application is specified by a task graph, with the scheduling and placement of the checkpoints determined at design time. The architecture used is shown in Fig. 13, which follows.

The system has two pools of processors, called Master and Checkers. The first one works independently, that is, it makes the accesses and writes to the system output as if it were the only processor(s) in the system. The Checker pool executes the same program as the

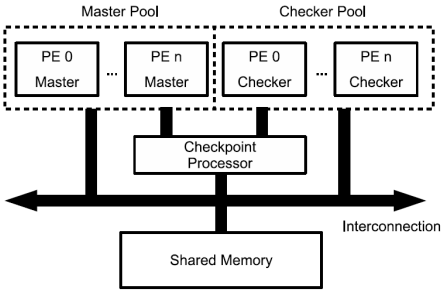

Figure 13 – Proposed Architecture by (KANG et al., 2014)
Source: (KANG et al., 2014)

master, but it is only used to compare results for error detection. Thus, once the divergence is detected, by the checkpoints processor, the correction is performed returning to the last valid state.

The experimental results were analyzed for the situations of a single processor and multiple processors, where algorithms are proposed for each case. To obtain the results were used as benchmark graphs generated with the tool SDF3 (STUIJK; GEILEN; BASTEN, 2006).

The main contribution of this work is the checkpoint cut lattice (CCL) algorithm, which was efficient in reducing the latency for multiprocessor fault-tolerant applications when compared to the greedy and equidistant reference algorithms. Also, when compared to the other reference algorithms, the CCL algorithm drastically reduced the overhead caused by the checkpoints.

Another work that uses changes in the hardware level is presented by (KOCH; HAUBELT; TEICH, 2007), with a mechanism to perform the CR. The authors consider that a hardware module can be modeled as a *Finite State Machine* (FSM), and with this premise to obtain the state of the FSM to perform the checkpoint. Fig. 14 shows a pure state machine (a) and the modification to collect checkpoints (b), using the authors' technique.

Also, in this work, a development flow for the hardware is proposed, as shown in Fig. 15, which performs an analysis of all flip-

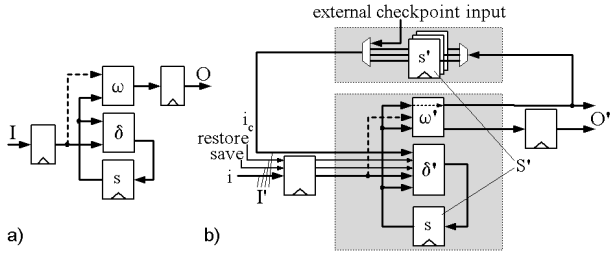

Figure 14 – Circuit comparison of an implementation with (a) a traditional FSM and a FSM with checkpoint
Source: (KOCH; HAUBELT; TEICH, 2007)

flops in the system and replaces them with the CR technique. Finally, ways to obtain the status of the FSM and its costs are analyzed.

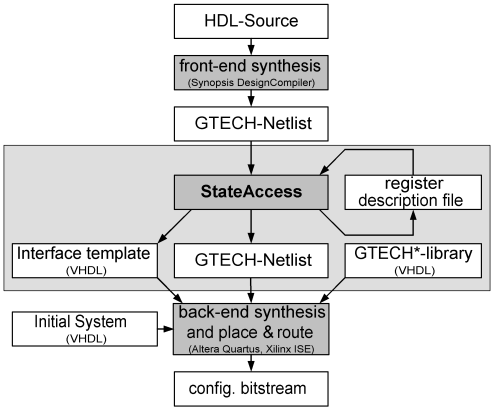

Figure 15 – Development flow of the work by (KOCH; HAUBELT; TEICH, 2007)
Source: (KOCH; HAUBELT; TEICH, 2007)

In works (Enshan Yang et al., 2013) and (FOUAD et al., 2014), implementations of CR are analyzed focusing on SRAM FPGAs. The former, authors propose Hierarchical Hardware Checkpointing (HHC), which is intended to improve system recovery performance after a detected error. The reference system is based on checkpoints made from reading the bitstream of the FPGA. HHC introduces a hardware

solution implemented within the FPGA to increase bandwidth when performing context recovery. The authors compare with the traditional way, where the state (checkpoint) of the system is stored in an external chip. The proposed architecture is presented in Fig. 16.

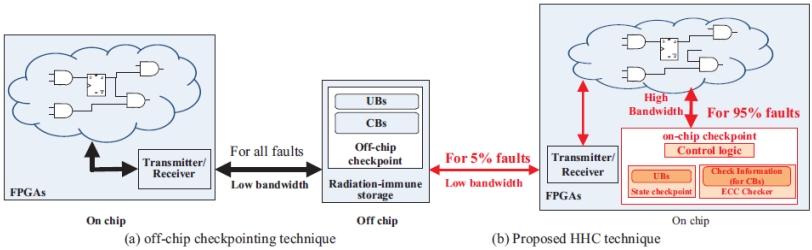

Figure 16 – Comparison between the checkpoint method proposed in HHC: (a) off-chip and (b) on-chip.
Source: (Enshan Yang et al., 2013)

The work in (FOUAD et al., 2014) also aims to reduce system recovery time after an error. For this, contexts of different parts of the hardware are saved and, in the case of the error, this context is resumed only for a defined piece of the hardware, using dynamic partial reconfiguration. The authors' contribution is presented as an algorithm, called Context-Aware Resources Placement (CARP), executed in the place and route phase, which analyzes the hardware from the point of view of tasks, positioning the components to decrease the overload times for dynamic partial reconfiguration.

In older work, such as (AHMED; FRAZIER; MARINOS, 1990) and (WU; FUCHS; PATEL, 1990), modifications are proposed in the cache memory update algorithm, aiming at the integrity of the system checkpoints and ensuring that the data exchanged between the processors are coherent. In these implementations, the CR technique becomes transparent to the user level and is fully implemented in hardware. However, the prerequisite of the method for the cache algorithm to work is the existence of a shared memory bus, in which everyone can detect the information that travels.

2.5.2 Software Approaches

The work of (OH; SHIRVANI; MCCLUSKEY, 2002b; OH; SHIRVANI; MCCLUSKEY, 2002a) and (REIS et al., 2005) present solutions for temporal redundancy made purely in software, bringing instructions and compiler level modifications to avoid transient errors. The Error Detection by Duplicate Instructions in super-scalar processors (EDDI) (OH; SHIRVANI; MCCLUSKEY, 2002b) and Control-Flow Checking by Software Signatures (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002a) are complementary.

In the first one, it is used the duplication of instructions, in registers and different variables for a superscalar processor (JOHNSON, 1991), seeking the detection of errors. It also takes into account the moment of insertion of the instructions to take full advantage of the parallelism of the processor, reducing the overhead of the technique. Fig. (REIS et al., 2005) shows the original code snippets and the EDDI technique.

```
ld r12=[GLOBAL]
add r11=r12,r13
st m[r11]=r12
```

(a) Original Code

```
   ld r12=[GLOBAL]
1: ld r22=[GLOBAL+offset]
   add r11=r12,r13
2: add r21=r22,r23
3: cmp.neq.unc p1,p0=r11,r21
4: cmp.neq.or p1,p0=r12,r22
5: (p1) br faultDetected
   st m[r11]=r12
6: st m[r21+offset]=r22
```

(b) EDDI Code

Figure 17 – Code snippet of the (a) original code and (b) after the technique of the EDDI.
Source: (REIS et al., 2005)

The CFCSS adds signatures in the instructions to identify problems in the program execution control flow. The Software Implemented Fault Tolerance (SWIFT) (REIS et al., 2005) work, makes use of EDDI and CFCSS techniques, adds *Error Correction Code* (ECC) and makes optimizations in each of the techniques. The results presented show superior performance in SWIFT, when compared to the other works, in

all the experimental tests.

Note that none of the works in this section makes use of the CR technique, but they present fault detection schemes. In fact, approaches for recovery and detection of failures are complementary and consequently related, because, after an error-detection, it is necessary to act to correct it or to take action, preventing the system from entering an invalid state. Therefore, these techniques can be easily extended to support fault tolerance.

2.5.3 Hybrid Approaches

The authors in (RAGEL; PARAMESWARAN, 2012) implement a hardware and software-based scheme for embedded processors using the CR technique. The processor architecture is based on the SimpleScalar tool (LLC, 2015) and the PISA (portable instruction set architecture) instruction set (very similar to MIPS (HENNESSY; PATTERSON, 2002)).

The proposed solution, called Reli, adds three instructions to perform the recovery and makes changes to the other relevant instructions to carry out checkpoints. The implementation flow, presented in Fig. 18, is done in two steps: hardware modification at the RTL level to add the CR in the architecture and at the software level to use the application (assembly code) as input of the integration tool, which will generate a memory containing the set of target instructions, different from those produced by the standard compiler.

In this work, three test flows are performed to obtain the results: runtime without failures, fault recovery time, and analysis of the area, energy loss, and timing. The software executed to get the reference times is based on the MiBench benchmarks (GUTHAUS et al., 2001). The fault detection technique implemented in Reli is similar to the one proposed in the IMPRES (RAGEL; PARAMESWARAN, 2006) work where the detection system monitors the bit-flips in the instructions and communicates to the Reli at the end of each block.

In the case of fault-free execution, an average execution over-

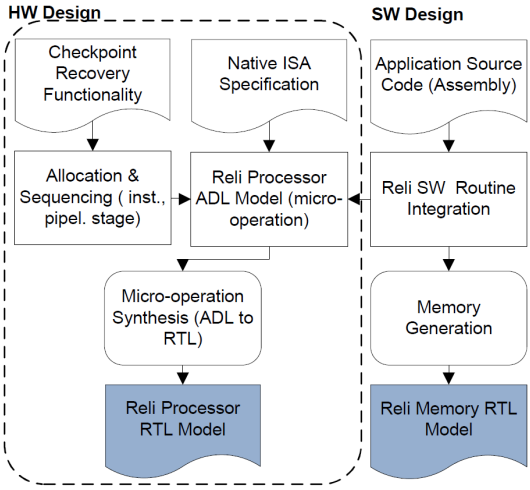

Figure 18 – Proposed flow for Reli
Source: (RAGEL; PARAMESWARAN, 2012)

head of 1.45% was measured. The fault injection execution, the single-flip bit model was used, with 1000 failures injected for each run, obtaining a recovery latency with a mean of 17.9 and 13.8 clock cycles, for the worst and best cases, respectively.

The last test performed was focused on the implementation of the system for an ASIC, using the 65nm TSMC library. Area overhead was calculated by comparison with the base processor, without the CR technique, which led to an average increase of 45%. The current leakage showed an average rise of 46%, while the operating frequency timings of the circuit remained at the same levels of the base architecture, i.e., for this application set, there was no change in the clock period.

A second work focused on the implementation of fault tolerance in embedded processors, is presented by (LI et al., 2013a). The authors apply parameters of performance, area and power consumption to generate a processor with two fault tolerance primitives. Instruction Vote and Replay (IVR) performs instruction level recovery by executing the same instruction three times and voting at the end - similarly to TMR systems. Block Checkpoint and Recovery (BCR) aims to save the

state with each update of the system information within an instruction block and, if the update contains an error, the state is reverted to the last safe point, as the CR technique predicts.

The decision problem for the choice between the available techniques (IVR and BCR) is NP-complete. Therefore, a heuristic, in the form of an algorithm, is presented as the contribution of the work to apply in the available code. The base processor is modified according to the design parameters, adding information in the instructions to determine if they are of type IVR or BCR. The execution flow is shown in Fig. 19:

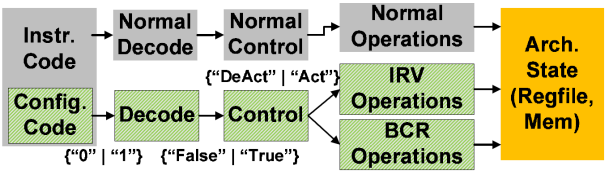

Figure 19 – Proposed flow for CSER
Source: (LI et al., 2013a)

The obtained results are comparisons between the implementation of SWIFT (REIS et al., 2005) work with fault tolerance in software, a Reli implementation (RAGEL; PARAMESWARAN, 2012), and three variations of the application of the proposed technique (with the pure IVR, pure BCR, and Hybrid configurations). The results show that the reliability of the hybrid configuration is up to nine times better than the other techniques when the parameters are pushed to the limits.

Finally, Dynamic heterogeneous adaptation for soft-error resiliency in ASIP-based multi-core systems (DHASER) by (LI et al., 2013b) presents an efficient way of recovering errors for ASIP-based multiprocessing systems. The proposed work is divided into three parts, as shown in Fig. 20: (1) task level correctness (TLC), (2) processor/core personalization based on TLC and (3) mechanism of adaptation at runtime.

To generate the system, it starts by analyzing the tasks that

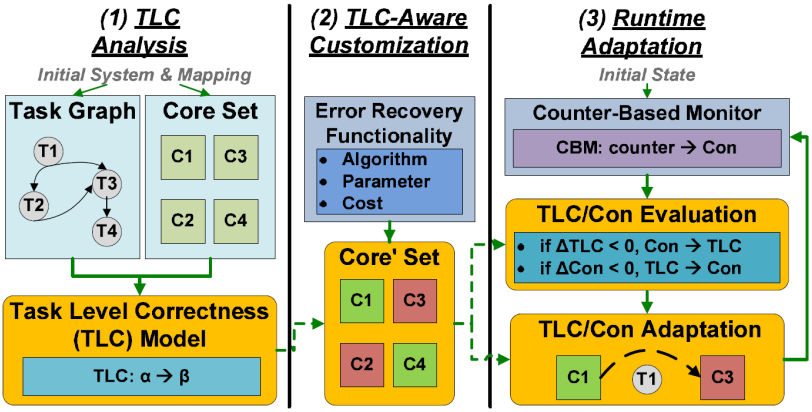

Figure 20 – Proposed approach for DHASER
Source: (LI et al., 2013a)

will be executed, to obtain a TLC index, which indicates the level of task requirement reliability - the higher the value, the lower the requirement for reliability of the task set.

The TLC index set is used in the second step, which generates a multi-processed heterogeneous system, in which all processors have the same instruction set (Single-ISA), but with different reliability techniques (such as spatial redundancy, temporal redundancy or no redundancy). Therefore, task sets are grouped into different processors at design time.

Finally, there is a run-time adaptation, which seeks to maintain reliability levels for running tasks, in order to correct any errors in the task/processor association during the design time.

The results are taken from implementations based on the LEON2 processor, doing an analysis for ASIC synthesis and *Hardware Description Language* (HDL) performance simulations. The simulations in HDL use six tasks, with a number of cores between four and six. Thus, the fault injection test presented reliability gains of up to 20% when compared to the base models.

2.6 WORK POSITIONING

In view of the related work described above, this section presents a comparison, analyzing the negative and positive points,for the following selected works:

W1. Fine-Grained Checkpoint Recovery for Application-Specific Instruction-Set Processors (LI et al., 2017)
W2. Applying lockstep in dual-core ARM Cortex-A9 to mitigate radiation-induced soft errors (OLIVEIRA; TAMBARA; KASTENSMIDT, 2017)
W3. A Low-Cost Solution for Deploying Processor Cores in Harsh Environments (VIOLANTE et al., 2011)
W4. Optimal Checkpoint Selection with Dual-Modular Redundancy Hardening (KANG et al., 2014)
W5. Efficient hardware checkpointing (KOCH; HAUBELT; TEICH, 2007)
W6. HHC: Hierarchical hardware checkpointing to accelerate fault recovery for SRAM-based FPGAs (Enshan Yang et al., 2013)
W7. SWIFT: Software Implemented Fault Tolerance (REIS et al., 2005)

Table 1 presents the above related works in contrast with two implementations on this thesis (the time redundant (**PTR**) and flow control DMR (**PDMR** approaches)).

Firstly, related works are classified according to the target hardware. The abstraction layer is according to Section 2.4.1, that is Hardware, Software or Hybrid. The modification stage is related to three locations where the fault-tolerance technique acts. That is modifications to hardware, compiler, and source code. Lastly, the fault-tolerance techniques that were used.

All hybrid works have modifications in the hardware. Since they differ on the target, W1 focus on an ASIP while W2 and W3 aim at Processors in SRAM FPGAs. When compared with PTR and PDMR, which are also focused on processors, the proposed approaches have no modifications to the source code. Regarding the *Fault Tolerance* (FT)

Table 1 – Related work positioning

	Target Hardware	Abstraction Layer	Modification Stage			FT Technique
			Hardware	Compiler	Source Code	
W1	ASIP/ASIC	Hybrid	Y	Y	N	CR+ISA
W2	Processors/SRAM FPGA	Hybrid	Y	N	Y	DCLS+CR
W3	Processors/SRAM FPGA	Hybrid	Y	N	Y	DMR+CR
W4	MPSoC/Generic	Hardware	Y	N	N	DMR+CR
W5	IPs/FPGA	Hardware	Y	N	N	TMR
W6	IPs/SRAM FPGA	Hardware	Y	N	N	CR
W7	-	Software	N	Y	N	CR
PDMR	Processors/Flash FPGA	Hardware	Y	N	N	DMR+CR
PTR	Processors/Flash FPGA	Hardware	Y	N	N	CR

technique, the aforementioned related works use the CR in combination with another technique, mainly to detect faults — the only exception is PTR which uses CR to both detect and correct errors.

Solutions in W4, W5, and W6 while based on hardware, have different targets and FT techniques. W4 is based on DMR+CR for *Multi-Processor SoC*s (MPSoCs), which means if there is a dual-core system, it will end up with two dual-core systems in a DMR configuration (four cores on total), recovering the system using CR. W5 focus on FPGAs, but the target application is an IP-block. Supposedly, it would be possible to have soft-core processor inside the IP. Likewise, W6 is for IP/FPGA, but the focus is on the configuration memory recovery time using CR.

Lastly, PDMR and PTR are pure hardware based designed to processors on FPGAs. The main difference is the use of flash FPGA for both. The CR implementation used on PTR and PDMR is the same, being the fault detection mechanism different. The fault recovery is done via the rollback feature of the CR.

All strategies aimed at processors presented modifications to either software and/or compiler with hardware. We present a pure hardware-based solution to deal with SEUs. In the present approach, there are no need to rewrite, or even recompiled the original code, the entire fault-tolerance is done by the modified architecture.

3 PROPOSED CHECKPOINT RECOVERY TECHNIQUE

The CR technique works by saving checkpoints considered safe during the execution of a processor (KOREN; KRISHNA, 2010). Whenever an error is detected, a rollback to the last known safe state is performed, namely recovery. To better understand the CR technique, Fig. 21 depicts a hypothetical scenario: after a checkpoint (Ck) is performed at $t = 2$, instructions I_{n+1}, I_{n+2} and I_{n+3} are executed. At time $t = 6$ the error is detected, causing the recovery to occur. After recovery, the three instructions are executed in the same fashion and the fault is overwritten with the right result.

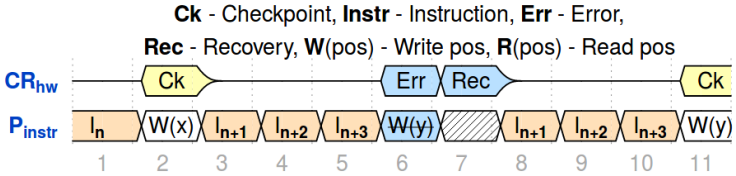

Figure 21 – Checkpoint Recovery Technique Scenario

If the SEU occurs in an element of the circuit, and, if the element is overwritten with the correct value after the SEU is identified, the error can be corrected. Therefore, the CR technique, which repeats the operation of a point considered safe, is a reasonable solution.

The following sections present in detail the implementation of the CR technique.

3.1 CONSTRAINTS AND ASSUMPTIONS

Before we advance into more details about the proposed technique implementation, some of the design decisions made need to be explained. We consider the environment to be the space, more precisely, an embedded FPGA on satellites. Up to *Low Earth Orbit* (LEO), the expected radiation dose is around 0.1 krad/year, meaning a five-year mission can have ∼0.5 krad dose (PETKOV, 2003). The *Geosynchronous*

Earth Orbit (GEO) can also be considered once it has a dose rate of ∼10 krad/year, but the FPGA of choice has to withstand this dose.

For that reason, the FPGA hardware is flash-based, in our case the Microsemi ProASIC3e FPGA (Microsemi Inc., 2017). In this type of FPGA, the configuration memory is not affected by SEUs (VILLA et al., 2017). Furthermore, this hardware can be migrated to an anti-fuse FPGA, where the configuration memory, after written, cannot be changed, resembling the FPGA to an ASIC (MCCOLLUM, 2009).

Although the configuration memory on the ProASIC3e is susceptible to TID, a dose of up to 30 krad seems not to affect the FPGA implemented circuit (KASTENSMIDT et al., 2011). The use of SRAM-based FPGAs, at the present stage, have not been considered, mainly because the configuration memory suffers significantly with SEE. Once the configuration memory is affected, the underlying implemented hardware (in our case, the soft-core processor) can behave erroneously. The error mitigation of configuration memory is a vast field of study with specific techniques, that could be integrated into this work.

The assumed fault model that is being mitigated is the SEE, more precisely its subtype SEU. Literature shows that SEU is the predominant failure when considering processors (REORDA et al., 2009; LESAGE; MEJIAS; LOBELLE, 2011). Also, we assumed in our fault model that only single-faults can occur.

For the CR technique, the granularity of the checkpoints (*i.e.* how often checkpoints are performed) need to be taken into consideration, once it introduces overhead in the processor execution. We used the metric that after every write operation to the main memory, the state can be saved, similarly to (VIOLANTE et al., 2011). This approach assumes that up to that point, if the error-detection mechanism did not identify the error, the state of the system has not been compromised. Another possible approach that could be used, presented by (RAGEL; PARAMESWARAN, 2012), is to save a checkpoint before the occurrence of a jump instruction in the execution of the program.

Once we are dealing with SEUs, the main storage system is vital to keep the system running. Since the program runs on the main memory,

if it presents errors the processor can misinterpret the instructions. For that matter, the main memory is assumed to be external and protected by an *Error Detection And Correction* (EDAC) technique. Furthermore, the cache memories are disabled for two reasons: they are additional area susceptible to SEUs, and since we are using writes to the main memory as reference points, the caches can interfere on the processor synchronization.

Also, like any other technique of fault-tolerance, there are two stages to implement fault-tolerant systems: Error-detection and Error-correction. These are two separated phases, which most methods integrate them into one single scheme. *e.g.* the TMR approach works by voting the majority of results and masking the disagreeing information. The voting process can be seen as the error-detection stage, and thus the masking is the error-correction. With this in mind, we propose the use of the CR technique to detect errors, by executing twice every slice of instructions (comprised between two checkpoints) and performing a third execution of the slice to correct a possible detected error. Nonetheless, other error detection schemes are implemented to be compared.

3.2 TEST VEHICLE

The LEON3 (Aeroflex Gaisler, 2015b) is the processor chosen as the target system of this work, due to the significant acceptance in the scope of space applications. It is a synthesized model, described in VHDL, of a 32-bit processor compatible with the SPARC V8 architecture, made available by the company Aeroflex Gaisler, under the GNU GPL license. The source code is free to use for research and educational purposes and is distributed as part of the GRLIB IP library (Aeroflex Gaisler, 2015a).

LEON3 is very configurable, being easily integrated into SoCs, accepting the multiprocessing configuration (up to four CPUs) and a wide variety of peripherals. A LEON3 configuration, with a single CPU, is shown in Fig. 22, where the main peripherals available in the GRLIB are shown. The LEON3 is a type of SoC based on the *Advanced*

Microcontroller Bus Architecture (AMBA) bus.

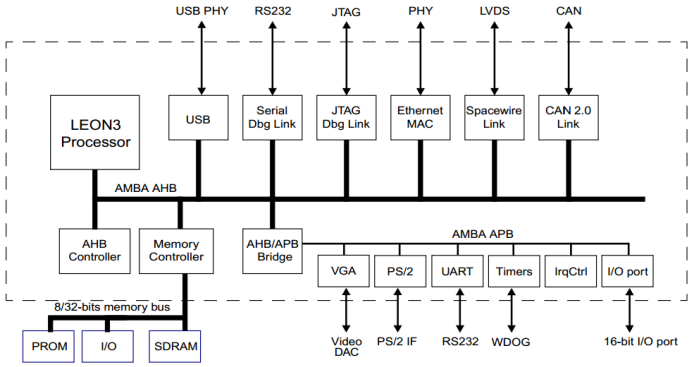

Figure 22 – LEON3 SoC Architecture Overview
Source: (Aeroflex Gaisler, 2015a)

More specifically, the LEON3 CPU core is based on a seven-stage pipeline, and may include other processing modules, such as a floating-point unit. In addition, a unit called *Debug Support Unit* (DSU) is integrated with the processor, which is also connected to the AMBA bus, to aid in debugging the CPU. These details are presented in Fig. 23, which also depicts the interface between the CPU and the AMBA bus.

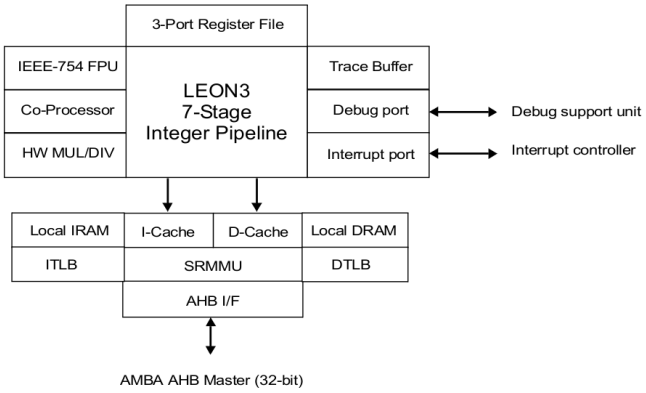

Figure 23 – LEON3 Internal Components
Source: (Aeroflex Gaisler, 2015a)

The LEON3 pipeline is similar to the five-stage classic RISC (HENNESSY; PATTERSON, 2002), with additional two stages, one for access to the register bank and one for the resolution of traps and interrupts. The stages of the LEON3 pipeline are depicted in Fig. 24 with the main registers shown and each stage is described as follows:

- FE (Instruction Fetch): If the instruction cache is enabled, the instruction is read from the cache. Otherwise, the search is routed to the bus. The instruction is valid at the end of this stage and stored in the Integer Unit.
- DE (Decode): The instruction is decoded, the jump addresses and CALL are generated.
- RA (Register access): Operands are read from the register bank or diverted by internal data.
- EX (Execute): ALU operations, logic and offset are performed. For memory (LD) and JMPL / RETT operations, the address is generated.
- MA (Memory): Access to main memory.
- XC (Exception): Traps and interrupts are resolved. For cache readings, the data is properly aligned.
- WB (Write Back): The result of an ALU operation, logic, offset, or cache is written back to the registrar database.

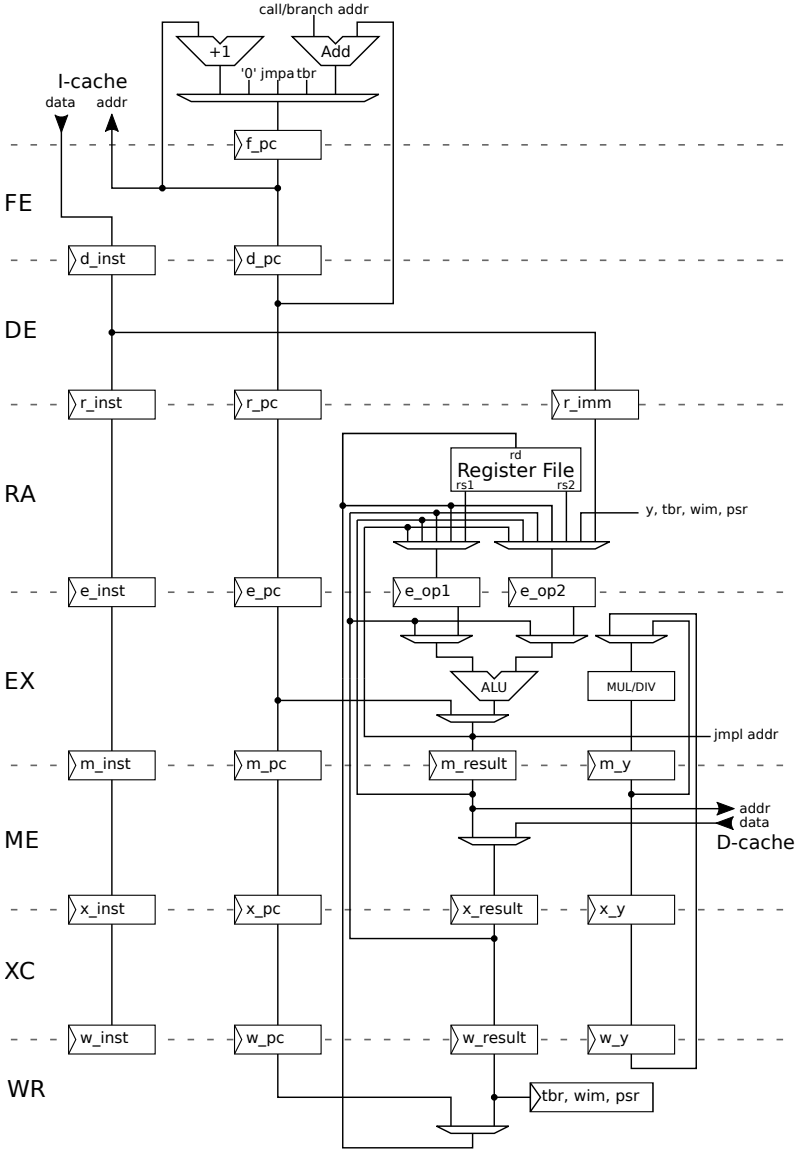

Figure 24 – LEON3 Pipeline
Source: (Aeroflex Gaisler, 2015a)

The GRLIB provides several designs for different FPGA vendors. These designs have a common characteristic of a single VHDL file for the top entity (`leon3mp.vhd`) and another file for the configuration of the processor (`config.vhd`). The top entity contains the instantiation of the `leon3s` that comprises the processor and its internal components (Fig. 23). The VHDL code is very modular, with each component within separate file. To better understand the hardware modifications needed to implement the CR technique, the relationship between these components are presented in Fig. 25. Note that the LEON3's Integer Unit it is in the same level of the Cache Controller/AMBA Interface, and the Register File is in another level of modularity.

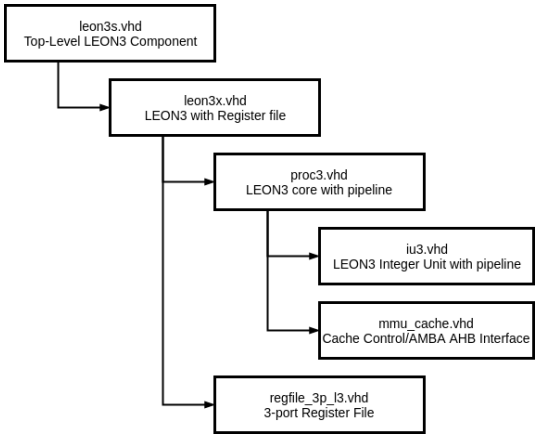

Figure 25 – LEON3 Entities Overview

Since the same entity responsible for the cache it is also responsible for the AMBA interface, it will always be instantiated inside the LEON3 processor. When the cache memory is disabled, the internal FSMs bypasses the cache memory access. The main components comprising the `proc3` entity and the relationship between the Integer Unit (IU3) and the Cache Controller/AMBA Interface are depicted in Fig. 26.

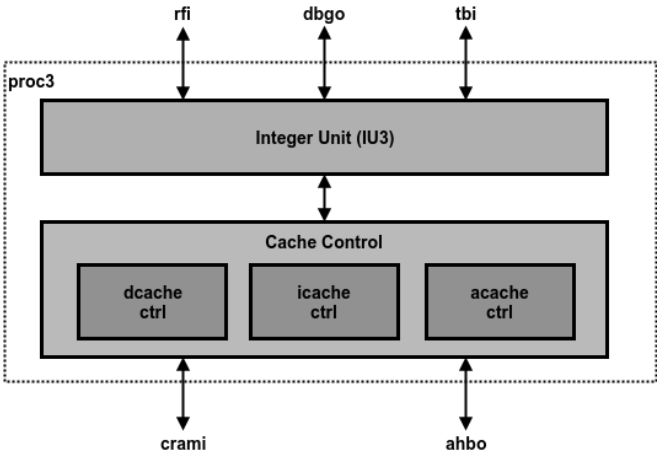

Figure 26 – PROC3 Connections Overview

3.3 IMPLEMENTED ERROR-DETECTION APPROACHES

To perform the rollback in the processor, the CR hardware needs to be aware of the error, meaning an error-detection must be implemented. There are several error-detection techniques in the literature. This study does not primarily aim at the detection of a SEU (i.e., error detection), as it can be considered another field of study by itself. Instead, we used fault tolerance techniques, which have fault-detection as their starting point. Three techniques have been used: the classical TMR (MARTINS et al., 2015); a bus-based DMR approach (FERLINI et al., 2012); and a time-redundant execution. The Fig. 27 presents the three architectures used in the experiment.

Fig. 27(a) uses a bus-based DMR to detect errors and inform to the CR module to perform the rollback on both processors. Fig. 27(b) is a classic TMR where it always detects single errors and masks single-faults using a majority voter. Fig. 27(c) employs the time redundant approach that executes twice every slice of code. In this case, the CR module saves the address and data that is going to be written on the main memory on the first attempt. After, rollback is performed, and the second address and data generated are compared with the ones

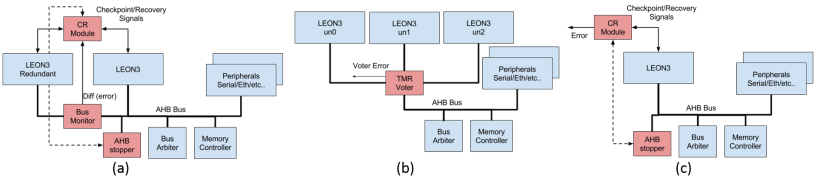

Figure 27 – Different Architectures Used for Error Detection: (a) bus-based DMR, (b) bus-based TMR, and (c) single-processor time-redundant.

saved in the first execution. When there is a match, the memory write operation is performed and a new checkpoint is saved, advancing the code execution to the next slice. If the values do not match, the second execution address and data are also stored by the CR hardware and another rollback is done to have a third execution of the code. This way, the CR hardware can use the result of three executions to perform a simple majority vote (similarly to the TMR) and write to the main memory the correct value. In the case of three different executions, an error signal is raised, similar to the voter error on TMRs approach, bringing the processor to a halt.

3.4 IMPLEMENTED CHECKPOINT RECOVERY APPROACH

During its normal operation, the processor creates checkpoints, which represent consistent states that can be restored. The checkpoints are a copy of the current processor state, more specifically the content of the pipeline registers. Any changes to the register file since the last consistent checkpoint are saved. The granularity of the checkpoints was designed, in such way that one checkpoint is created every time the processor executes an instruction that performs writes in the main memory. Since the main memory is the reference, instruction and data caches were disabled on the processor configuration. Even though the absence of cache in the processor degrades overall performance (regarding execution time), it also introduces another point of failure for SEUs.

To implement the CR technique, the LEON3 hardware was

modified. The first step was to find all the registers on the pipeline that holds the current state of the processor. In more detail, the IU3 unit has VHDL processes, comprising the entire pipeline that needed to be saved. Despite the fact that the instruction and data caches were disabled, there are FSMs that control the communication between the *Integer Unit* (IU) and the AMBA bus, and need to be checkpointed as well. A single checkpoint signal is connected to all modules involved. When the main memory write is detected, it causes the checkpoint by copying all the data to redundant registers.

In Fig. 28 the modified `proc3` unit is presented with the internal connections to the IU3 and Cache Control units, the requests to perform the checkpoint or recovery is done through a dedicated set of signals (indicated in the chkp/recov signal on the figure).

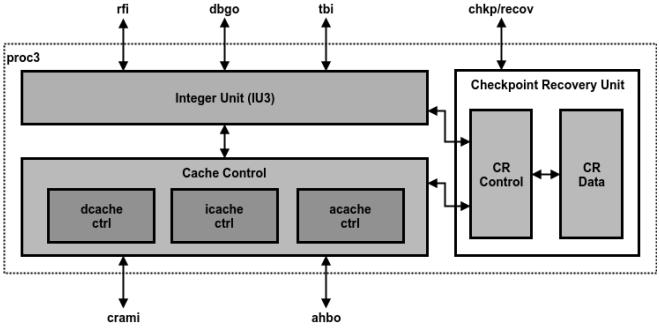

Figure 28 – Modified PROC3 unit with CR Control Unit.

The register file, likewise, needs to be taken into consideration when recovering the processor state. In order to do so, a fourth port was added to the register file to perform a read on the register that is currently being written. This way the old value can be saved in a memory stack. On the recover event, the stack is dumped back into the register file, bringing it back to its safe state (last checkpoint). This process was made inside the `leon3x` unit and is presented in Fig. 29. The Register File Checkpoint Unit is responsible for multiplexing the connections between the `proc3` unit and the modified 4-port register file. In the normal operation, the fourth-port address bus is connected to the

write port address, meaning that when a write operation is performed, the fourth port data output the register value being overwritten. This data value, along with the address, is then pushed into the stack by the Checkpoint Unit. In the event of a new checkpoint, the stack memory is flushed since all values inside the register file are supposed to be correct. If an error is detected, the recovery process is activated and the Checkpoint Unit initiates to perform writes to the Register-file. The address and data are pushed out of the stack and written to the Register-file. When the stack is empty, the recovery process of the Register-file is finished.

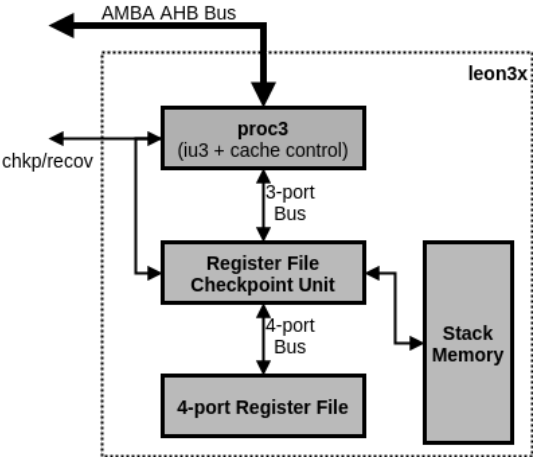

Figure 29 – Modified LEON3X unit with Register-file Checkpoint Unit and Stack Memory.

To perform a recovery on the aforementioned system, the processor needs to be halted for a period of time. This time is required to write the registers back into the Register-file, and recover the IU3's pipeline. In order to do so, a second AHB-master unit is connected to the AMBA bus. Its function is to request the AMBA-bus, through an write request, forcing the LEON3 processor into a halt state. While the second AHB-master owns the AMBA bus, the recovery process is done. This unit is part of the CR implementation.

Going into detail, the top-level VHDL file (`leon3mp.vhd`) of

the design on the GRLIB instantiates the unit `leon3s`. This unit is a wrapper to the aforementioned `leon3x` unit, with a few connections to `gnd` and `vcc`.

3.5 DMR AND TIME-REDUNDANT IMPLEMENTATION

The implementation of the DMR and Time-redundant approaches have different fault-detection schemes, while the former is based on transactions on the AMBA bus, the latter compares the pair address/data being written to the main memory.

For the DMR implementation, there is a module that compares transactions on the AMBA bus. Fig. 30 depicts the main connections of the LEON3 in order to achieve the same results presented by (FERLINI et al., 2012). The modification here are the ones presented to get the CR technique running inside of each LEON3 processor (presented in the section above). Note that the controller of the CR technique is in the top-level, along with the instantiation of both processors. Whenever the outputs do not match, a signal error is raised, the controller request the AMBA bus, and when it granted, it sends a recovery signal to both processors. After the recovery, both processors continue to run the program.

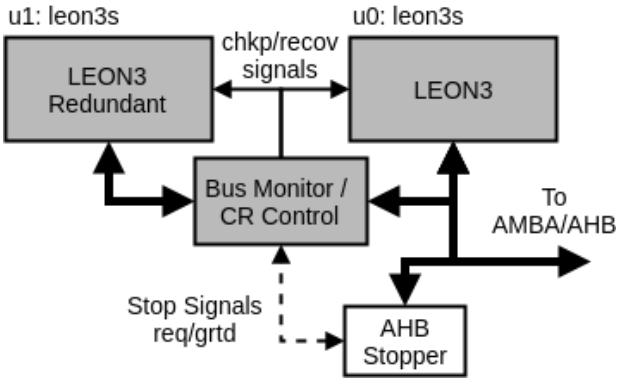

Figure 30 – Detailing of the LEON3 DMR connections.

The time redundancy is obtained by using the CR mechanism,

to run each interval between checkpoints twice. In order to do so, the main connections of the LEON3 Time-redundant are depicted on Fig. 31. Fig. 31(a) presents the top level instantiation of the LEON3 and the AHB unit to the AMBA bus, and Fig. 31(b) presents the modifications made inside the already modified `proc3` unit (Fig. 28). Note that Fig. 31(b) is a detailing of the LEON3 unit in Fig. 31(a).

On the first run, the processor saves the information of the memory write instruction, but does not allow it to proceed, bypassing the memory write enable signal (Write Mux on Fig. 31(b)). Then, a rollback is performed, and the processor executes all instructions from the last checkpoint. When the second run reaches the memory write instruction, the CR mechanism compares address and data to the ones stored from the first run, if they are equal, the main memory is written and the process repeats, otherwise, the fault is detected and a mismatch is signalised.

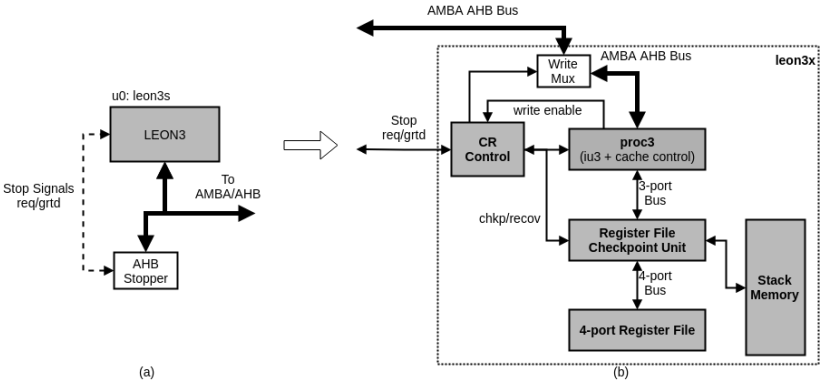

Figure 31 – Detailing of the LEON3 time redundant connections.

Since the checkpoints are based on memory write, both techniques presented here (DMR and Time-redundant) monitor the AMBA write signal coming from the processor, whenever it raises to high, the checkpoint is performed overwriting the old one.

3.6 CHECKPOINT RECOVERY HARDWARE CONSIDERATIONS

Each checkpoint is an image of the state of the system considered safe. Such checkpoint is a form of data redundancy. In our case, the data redundancy is comprised of the processor pipeline and register file modifications. As aforementioned, not only the `iu3` unit and the register file that contain information but also, the units `icache`, `dcache`, and `acache`. Table 2 presents the amount of data in bits for each unit and the size of the stack for the register file address and data.

Table 2 – Checkpoint storage data size

Component	**Bits**
iu3	2502
icache	323
dcache	830
acache	34
stack data	2048
stack addr	512
Total	6249

The `iu3` unit individually has the major quantity of data (2502b) since it is a copy of the entire LEON3 processor pipeline. The stacks summed account for 2560 bits since they are 64 positions of 8 bits for the address and 64 positions of 32 bits for the data. Both stacks can be easily protected against errors using an ECC based on the requirement, ranging from parity to extended-hamming or *Cyclic Redundancy Check* (CRC).

The other components' checkpoint data can be protected in a similar form, but preferentially with the use of signatures (such as checksums) since there are different registers widths. It would be possible to read the entire checkpoint as a string of bits and calculate a signature to confirm integrity.

Another weak point is the checkpoint hardware control and its components. This hardware is also susceptible to SEUs that could cause the system to malfunction. The checkpoint hardware is mostly

comprised of combinational logic and the amount of data stored is relatively when compared to the entire SoC. Since combinational logic is not affected by SEUs and the stored data can be further mitigated, at this stage, we consider that it would not be affected.

Lastly, it is important to mention that there are no modifications outside the LEON3 RTL code. This means the same code, compiled to the original LEON3, can be run seamlessly on our architecture. The only difference is how the code is going to be executed and recovered (in case of an error).

4 EXPERIMENTAL RESULTS

In this chapter, we describe the adopted simulation method, test setup, and benchmarks used in our tests to obtain simulation results. We use the fault definition according to (AVIZIENIS et al., 2004). All results here described, have been based on premises from Section 3.1.

4.1 SIMULATION METHOD

To run our tests, the LEON3 processor was simulated using the Modelsim tool. The fault injection was performed according to the pseudo-algorithm in Fig. 32. The fault injection script reads all LEON3's IU registered signals (memory elements). For each signal, a new simulation is run (line 2). In each simulation, a random time is picked (line 3) and ran. After the runtime, the current signal value is read (line 4), and a SEU is simulated by inverting one bit inside the signal value (line 5) and applying it to the current signal using a force command (line 6). Note that this `force` command modifies the signal until it gets overwritten, known as *deposit* on the simulator tool. Finally, the simulation is run until its end. This means that the program comes to its final state, by raising a stop signal, or an error signal (if detected by the simulation script). In line 10 we make sure we have enough samples to fulfill a confidence interval of 95% and a margin of error less than 5% (since it is a simple random sample: $0.98/\sqrt{n}$, or at least 400 runs).

The simulation results were classified according to Fig. 33. After fault injection, there are three possible results (outcomes): Correct, Detected, or Failure. A correct result implies in either no error detected, and/or a latent error detection. This means that the fault in that signal, at a given time did not affect the execution. A failure means that the fault causes a failure in the processor without being possible to detect it. Lastly, the detected fault is the result of an error, which can be further classified in three possible situations according to the fault-tolerant technique used: Recovered, Not-recovered, and Recovered incorrectly. A

```
1  do{
2    foreach(signal current in L3.IU3){
3      run rand();
4      value = read(current);
5      seu(&value);
6      force(current, value);
7      run all;
8      runs++;
9    }
10 }while(runs < CONFIDENCE);
```

Figure 32 – Simulation Steps Pseudo-algorithm

recovered case is when after detect, the recovery process acts accordingly, and the program finishes its execution with the expected result. A not-recovered error happens when the recovery process fails to complete the program, either without the expected result or a time-out. The last case is when the recovery process is performed, and the program reaches its final state with an incorrect result. This can happen when the error occurs on the variable that controls a loop, for example.

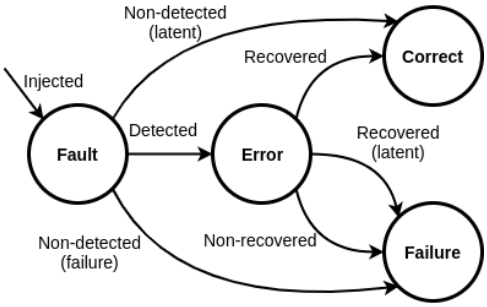

Figure 33 – Fault states diagram

4.2 EXPERIMENTAL SETUP

For each architecture of our tests a set of four programs were used to stress the processor instruction set as follows:

1. Basic: a simple arithmetic operation that is executed 50 times and checked against the correct value. This program aims at repetition
2. Bubble sort: classic bubble sort algorithm, performs five times the procedure of filling a ten element vector, sort it and verify the vector order.
3. NMEA: calculate the checksum (bitwise *xor*) of ASCII codes on a message string five times.
4. Hamming: calculate a hamming encoded message using matrices five times.

It is important to note that we did not use a more classic test program (such as *dhry*, *stanford*, or *whetstone*) since the simulation time was prohibitive, *e.g.* over a day on a high-end computer for a single execution. In order to circumvent this issue, the above programs were written in standard C language trying to comprise some of the classic code flow execution. Nonetheless, the chosen workload applied to the four variations of the LEON3 took over a week of computer simulation. This translates into over 2GiB of raw data logs.

Fig. 34 presents a breakdown of the number of instructions executed in the unmodified architecture for our workload. These figures were obtained from the simulator by enabling the console disassembly feature. This means that these instructions were executed inside the LEON3's pipeline. This approach is more accurate to observe the dynamic execution of each program, differently from dumping the instructions from the source code (which is static).

All programs, except for `basic`, have the major incidence of `ld` instructions. `bsort` and `hamming` have a very similar profile, but the former has 11% of memory write instructions (`st` and `stb`) since the vector is in the main memory. It is interesting to note that all programs have a high incidence of `nop` instructions. Indeed, once the cache is disabled, the processor pipeline needs to be stalled until the information arrives from the main memory. `basic` and `nmea` have a higher incidence of arithmetic operations, being the addition and shifting respectively.

The *General Purpose Input Output* (GPIO) pins are used to signalize the external world when it began and finish. These signals are used to assert the correctness of the execution and/or error states. For instance, if the program `bsort` on its verify state find an unordered value, an error signal is raised to communicate the simulation script.

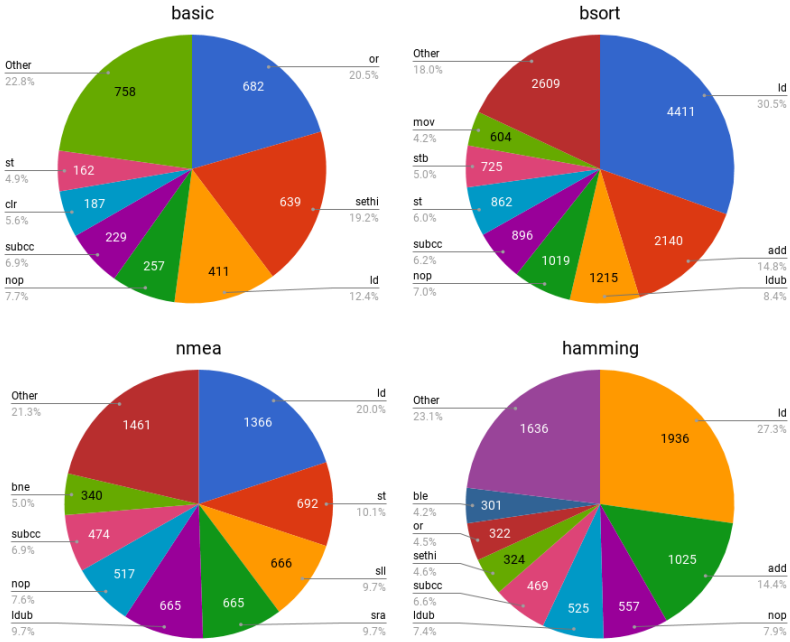

Figure 34 – Instruction breakdown for used workload

Source code for this workload can be found in Appendix B. The compiler used is standard sparc-elf-4.4.2 toolchain. The following flags have been used on compilation and linking:

```
CFLAGS=-msoft-float -Wall -O0
LDFLAGS=-qsvt -qnoambapp -lsmall
```

4.3 DETECTION AND RECOVERY CAPABILITY ANALYSIS

Results from the simulation were analyzed and compiled according to Section 4.1. This section presents a comparative analysis of the four variations of the LEON3 processor using the workload mentioned above.

Fig. 35 presents the detection analysis for the three architectures used in the experiment with the inclusion of the LEON3 original (unmodified) configuration. The Y axis on the left shows the total of executions in the simulation ran, and on the right Y axis the percentage of these figures. Note that for the original configuration there is no detection available. Therefore only the Correct/Failure results are presented.

For the architectures of the TMR and the DMR, the correct rate were similar, in the order of 79% on average, which means that the fault is either latent, or not detected. The failure rate of the original is slightly lower than the detected figures in the TMR and DMR approaches. This is due to a detected error not always becoming a failure.

Interestingly the time redundant approach shows the higher percentage of corrected results (in the order of 95% on average). This is due to the re-execution of the code slice since the injected fault can be overwritten before it manifests itself during the program execution.

The error classified as failure appears on LEON3 original and time redundant implementations. After a closer look into the simulation results, it is possible to note that some signals have an immediate effect on processor execution. For instance, internal signals of the pipeline stage `EX→nerror` and `ALU→Ticc`, are responsible for the general processor error and Trap interruption control, respectively. These signals can cause a failure every time they suffer a simulation SEU. Still, the time redundant presents an improvement over the original implementation for this type of error.

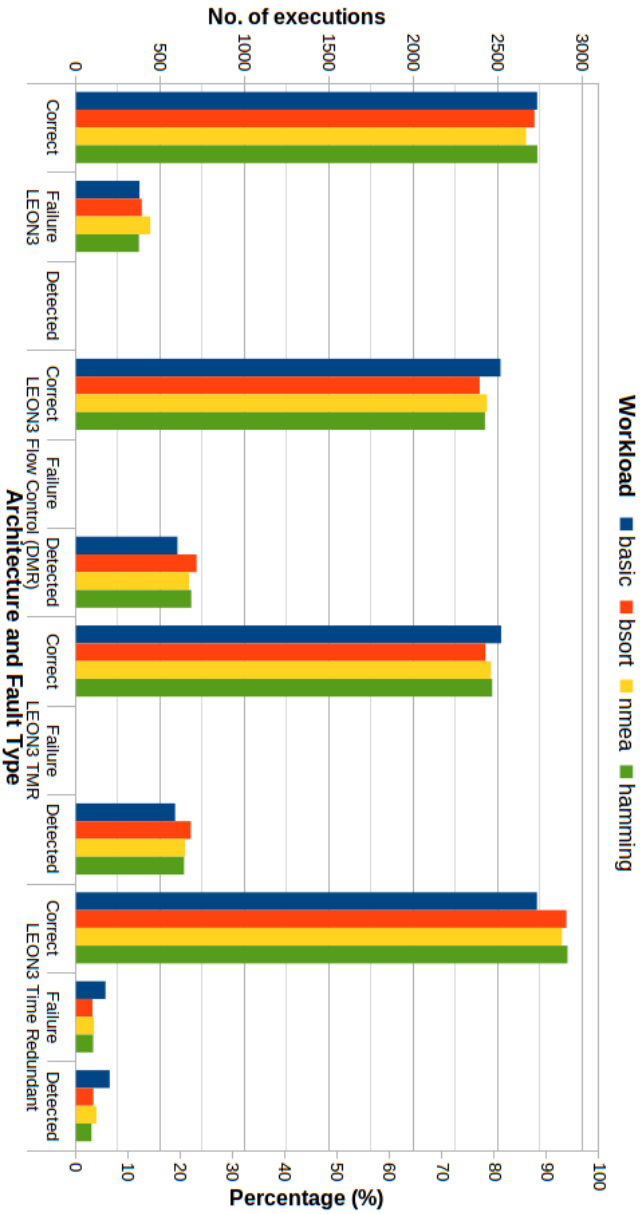

Figure 35 – Detection analysis comparison of different LEON3 architectures

Fig. 36 shows an analysis for the recovery process on the detected errors for the time redundant and DMR approaches. Note that these charts are based on the absolute number of errors detected, consequently the breakdown of the values are presented on stacked percentages, so it would be possible to compare both techniques. The TMR is not shown since it has 100% correction for single faults. However, TMR would have to, somehow, recover the faulty processor, otherwise, the error gets accumulated on the system. The original configuration is not presented once there are no detection/correction mechanisms.

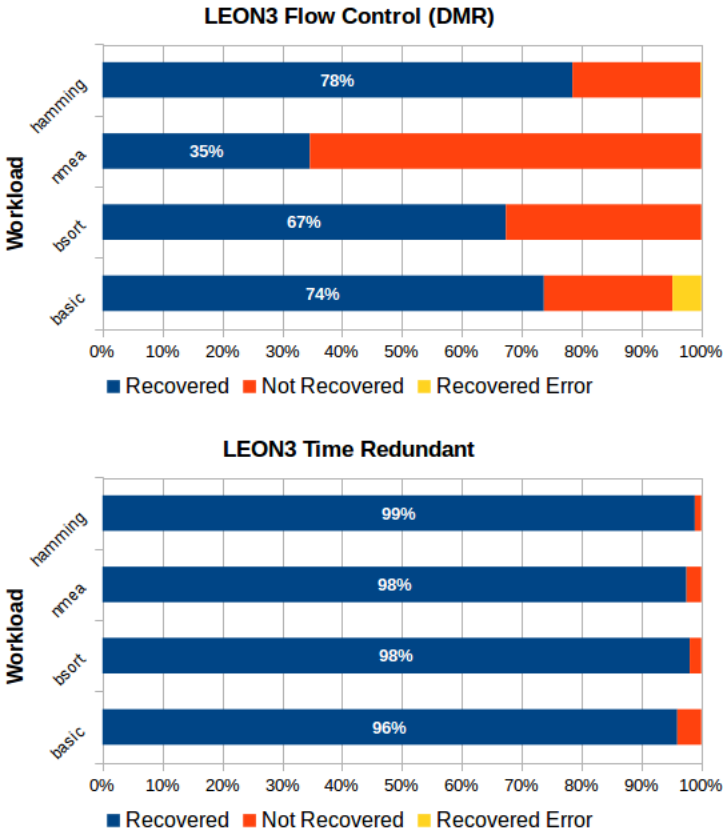

Figure 36 – Recovery analysis for the DMR and Time Redundant approaches

For the time redundancy approach, the average errors that were corrected, is near the 98% mark, while the average for the DMR is a little over 63%. The main problem with the DMR, for our tests, is to recovery both processors correctly.

The overall performance comparison for the DMR and time redundant approaches is depicted in Fig. 37. These charts present the total percentage of executions, for each program, which finished with success, including those detected and corrected.

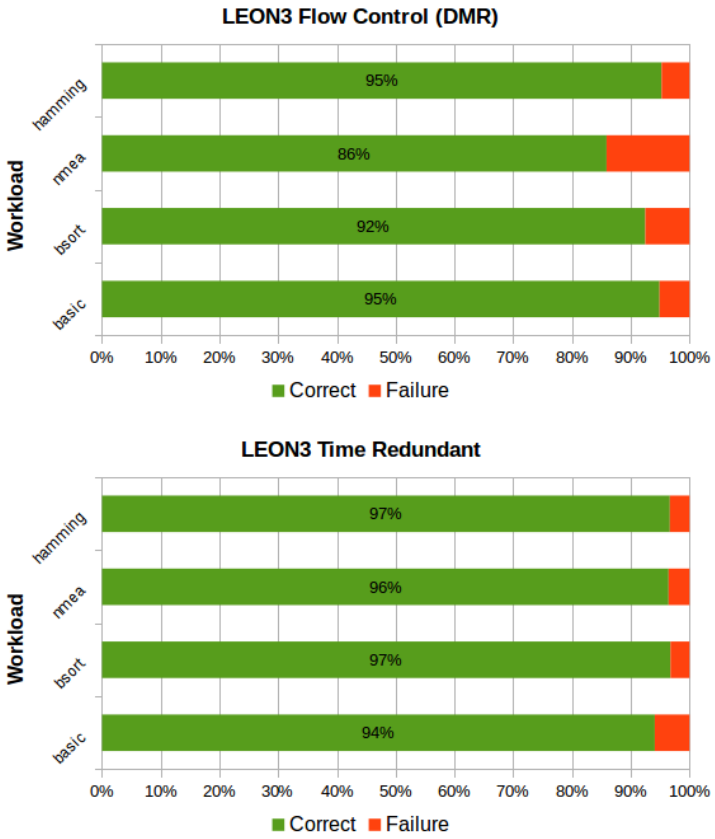

Figure 37 – Overall comparison of LEON3 DMR and time redundant approaches

The averages of correctness are 92% and 95% for DMR and time redundant approaches, respectively. For the time redundant, the average 5% of failures could be further mitigated due to the signal sensibility of the LEON3. This topic is discussed on Chapter 5.

4.4 EXECUTION OVERHEAD ANALYSIS

Although the CR technique presents a competitive recovery capability, it introduces time overhead on the program execution. Whenever a recovery is made, the execution needs to be halted for, at least, one clock cycle, allowing the recovery of the IU pipeline registers and an additional clock cycle for each register in the register file used since the last safe checkpoint. Table 3 presents the increase percentage on the workload execution against the original implementation of the LEON3 processor.

Table 3 – Execution time overhead against baseline

	LEON3 Flow Control		**LEON3 Time Redundant**	
Program	**Correct**	**Recovered**	**Correct**	**Recovered**
basic	0.00%	4.39%	112.88%	113.61%
bsort	0.00%	0.25%	104.90%	105.00%
nmea	0.00%	1.09%	104.90%	105.12%
hamming	0.00%	1.07%	106.71%	106.99%
Average	0.00%	1.70%	107.35%	107.68%

Each implementation of the LEON3 has different time overhead on the execution. The LEON3 DMR does not add time to perform the checkpoints since the checkpoint procedure is done in parallel. Therefore, no overhead is noticed when there are no errors detected on the execution. Although, once an error is detected, the recovery procedure takes a few clock cycles to occur, hence the average value of 1.70% of time increase against the baseline execution.

The cost of executing twice each slice of code out-stands on the LEON3 time redundant approach. On average it adds 107.35% for correct execution and 107.68% when the error is detected.

The impact of the time redundant approach can be isolated to analyze how long it takes to perform a rollback (recovery process). Table 4 presents the simulation time that the CR hardware needs to act on the system and its mean occupation of the stack of the register file. In our simulation, the clock cycle was configured to $25ns$ and the stack has 64 positions (Section 3.6). Each rollback process takes, on average, 17 clock cycles to finish and rewrite seven registers on the register file.

Table 4 – Recovery impact on time redundant approach

Program	**Simulation Time (avg - ns)**	**Clock Cycles**	**Stack Usage (avg)**
basic	456	18.24	8
bsort	474	18.96	8
nmea	405	16.2	7
hamming	400	16	7
Average	433.75	17.35	7.5

4.5 CACHE INFLUENCE ANALYSIS

Since we have disabled instruction and data caches, it is possible to analyze the time overhead due to this decision. Table 5 presents the simulation time (in ns) and increase ratio. The time increment due to the removal of caches and keeping the unmodified architecture has an average of almost seven times slower than with caches. Also, without cache and adopting the time redundant approach has an average of 14 times the original time.

It is a high price to pay in exchange for reliability. Nonetheless, the primary goal is to have a fault-free execution instead of the fastest possible execution.

Table 5 – Effect of caches on execution time

Program	With cache	Without cache	Increase	Without cache + TR	Increase
basic	133825	494000	3.69	1051630	7.86
bsort	601180	5064100	8.42	10376346	17.26
nmea	286770	2057875	7.18	4216682	14.70
hamming	275291	2219775	8.06	4588508	16.67
		Average	6.84	**Average**	14.12

4.6 FPGA AREA OVERHEAD ANALYSIS

Although we have not performed the test on the FPGA hardware, it would be interesting to compare how the different architectures influence on the area occupied. To do so, we ran the design flow for a Xilinx FPGA (Spartan-3 1500, model xc3s1500-4-fg456). The simulation was made using the Xilinx FPGA since the LEON3's GRLIB does not provide models for simulation of the main memory for the Microsemi ProASIC3e. Nonetheless, all modifications were made inside the LEON3 core architecture. Hence no difference should be found when synthesized to the Microsemi device.

Table 6 shows the total of the main device resources used for the different architectures compared to the baseline (no modifications) and also the available resources on the FPGA. For all resources presented, the lower overhead obtained can be noticed on the time redundant variant. Indeed it was expected since no processor replication was made. Respectively, the DMR and TMR have higher occupation rates when compared with the time redundant approach. The TMR approach can be fit on this FPGA since it uses more Slices and LUTs than it is available on the device.

Table 6 – Area Overhead Comparison for a Xilinx Spartan-3 1500 FPGA

Resource Type	Available	Baseline	Time Redundant	Increase	DMR	Increase	TMR	Increase
Slices	13312	6483	8208	26.61%	9873	52.29%	13702	111.35%
4-input LUT	26624	12017	12598	4.83%	18424	53.32%	25836	115.00%
BRAM	32	6	7	16.67%	10	66.67%	14	133.33%
MULT18x18	32	4	4	0.00%	8	100.00%	12	200.00%

The Time-Redundant version has a significant increase in Slices due to logic implementation, but it is the most abundant resource on the FPGAs. Nonetheless, when coupled with the results of time overhead, this technique presents a competitive approach when compared to the DMR and TMR versions. Moreover, the spare logic on the FPGA could be used to implement other functions or improve even further the technique by protecting more elements on the processor.

Additionally, all implementations went through the synthesis tool on the Microsemi design flow. At this stage, preliminary results can be obtained for the target FPGA. The data from Core (VersaTiles) and RAM for a Microsemi ProASIC3E-1500 FPGA are presented in Table 7 along with the increase percentages for each variation. Since no optimizations were made, the area increase is different than those presented for the Spartan-3 FPGA when compared to the baseline implementation.

Table 7 – Area Overhead Comparison for a Microsemi ProASIC3E-1500 FPGA

Resource Type	Available	Baseline	Time Redundant	Increase	DMR	Increase	TMR	Increase
Core	38400	15599	30147	93.26%	41852	168.30%	52243	234.91%
RAM/FIFO	60	52	54	3.85%	60	15.38%	68	30.77%

Once again, the TMR could not be implemented on this device. The same goes for the DMR approach, which cannot be fit in the device at the current stage of the design.

It is possible to note that both results of area match the order of footprint increase. The lower is the time redundant, followed by the DMR and lastly the TMR. This confirms the consequence of replicating the processor unit inside the SoC.

4.7 FPGA POWER ANALYSIS

Another critical figure when designing space applications is power consumption. Microsemi offers a spreadsheet (Microsemi Inc., 2018) that can estimate power consumption of its devices on very early stages of development. At synthesis, it is possible to use the quantities

of VersaTiles and RAM used, along with the operating frequency of the system to estimate dynamic and static power.

Table 8 shows the power consumption estimation results for a ProASIC3E-3000 FPGA. The following configurations were used on the power estimation tool:

- Device: A3PE3000
- Range: Commercial
- Condition: Typical
- Mode: Active

The decision to estimate values on a larger device was based on the spreadsheet limitations. The calculator spreadsheet does not allow to enter with a number higher of Cores/RAMs than the available on the chosen device. Since this is an estimation for comparison, the differences on the dynamic power figures[6] are negligible. The major difference is the static power, which is the amount of power that the device consume independently of the implemented circuit on the FPGA. For the -1500 variant this value is 18mW for the same settings.

Table 8 – Power Consumption Comparison for a Microsemi ProASIC3E-3000 FPGA

Power Source	**Original**	**Time Redundant**	**DMR**	**TMR**
Dynamic Power (mW)	39.58	72.94	100.12	124.44
Static Power (mW)	37.5	37.5	37.5	37.5
Total (mW)	77.08	110.44	137.62	161.94
Total increase (%)	-	43%	78%	110%

The time redundant approach shows the lower increase in power consumption, followed by the DMR and TMR approaches. Since the redundant processors are fed with the main clock source, their dynamic power are proportional to the occupied resources of the FPGA.

On a more practical example, a 1,000mAh/1.5V battery have a 1500mWh capacity. If we consider this capacity as the main power source

[6] Experimenting on the spreadsheet, less than 0.5mW difference on the dynamic power was noticed for the 1500 and 3000 device.

and ignoring losses, we can calculate the runtime using the Equation 4.1.

$$Runtime(h) = Capacity(mWh)/PowerConsumed(mW) \quad (4.1)$$

In this case, the theoretical runtime are:

- Original: ∼19.4 hours
- Time Redundant: ∼13.5 hours
- DMR: ∼10.9 hours
- TMR: ∼9.2 hours

4.8 CHAPTER REMARKS

This chapter presented experimental results for the CR technique applied to the LEON3 soft-core processor. The proposed technique was evaluated against the major FT techniques, namely DMR and TMR.

Our simulation results show that the time redundant based on CR have lower overhead on area and power while sustaining reasonable numbers on the detection and recovery process. The major drawback is time overhead due to its nature of re-execution of code slices.

As a final comparison, in order to obtain a more solid number, we use an adapted formula from (ARGYRIDES; PRADHAN; KOCAK, 2011) and (CASTRO et al., 2016), to get a metric on the technique total cost. The total cost formula is presented in Equation 4.2. The calculated total cost is dimensionless once it represents a relationship between proportions.

$$TotalCost = \frac{DetectionRate * RecoveryRate * RunTime}{TimeOverhead * AreaOverhead} \quad (4.2)$$

Table 9 shows the results of the total cost for the three implemented techniques for a Microsemi ProASIC3E-3000 FPGA. Values used in detection/correction rates and overheads columns are percentages (i.e., 1.00 means 100%) compared to the original implementation.

These figures show us that the time redundancy is in between the DMR and TMR technique. Although, the time redundancy approach

Table 9 – Total cost analysis for Microsemi ProASIC3E-3000 FPGA

Approach	**Detection Rate**	**Recovery Rate**	**Runtime**	**Overhead**		**Total Cost**
				Time	**Area**	
Time Redundant	0.95	0.98	0.70	2.07	1.93	0.16
DMR+CR	1.00	0.92	0.56	1.01	2.68	0.19
TMR	1.00	1.00	0.48	1.00	3.35	0.14

have a better $Detection * Recovery$ factor, 0.93 against 0.92 for the DMR+CR.

Regarding the TMR technique, its major drawback is $\frac{Runtime}{Area}$ factor. Even though it has 100% detection and recovery rates, without time overhead, the spatial redundancy compromise it's application for low-power applications.

5 IMPROVING MICROPROCESSOR RELIABILITY

In order to obtain full microprocessor reliability, some of the drawbacks mentioned above on the implementation should be addressed. In this work, there are two main issues that can be resolved with ongoing work of which this author has been participating. Additionally, the adoption of a SRAM FPGA for the based hardware is discussed.

5.1 MICROPROCESSOR CRITICAL SIGNALS

The work in (TRAVESSINI et al., 2018) presents an analysis of the results of a fault injection campaign targeting the LEON3 processor core, which comprises both the pipeline execution unit and the cache controllers. The study investigates the effects of the injected faults, and how they manifest in the processor interfaces with other modules such as the caches, main memory, and register file. Fig. 38 shows the major registers that caused errors in the entire LEON3 SoC.

The LEON3 most vulnerable registers have been protected with a partial TMR technique. The results of the fault injection campaign are presented in Fig. 39. The `no effect` and `latent` faults account for approximate 99.25% executions. Synthesis results show an increase of around six percent on area for the FPGA implementation.

For processor critical signals, mentioned in Section 4.3, this approach can be easily integrated to the time redundant LEON3. This process is expected to significantly improve the mitigation of failures not detected by the CR technique.

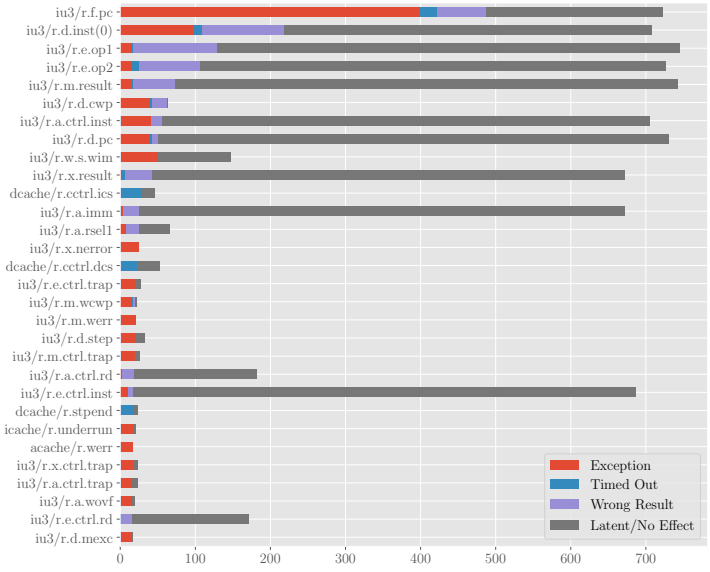

Figure 38 – Individual Register Performance
Source: (TRAVESSINI et al., 2018)

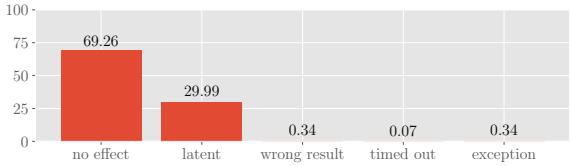

Figure 39 – Protected LEON3 overall performance
Source: (TRAVESSINI et al., 2018)

5.2 REGISTER-FILE RELIABILITY

Since the time redundant technique reads data from the register file to form the checkpoint, once the register file is affected, the checkpoint can be compromised as well. For that matter, the work in (GOERL et al., 2018) presents an approach to detect and correct MBU occurrence in memory arrays. The EDAC technique is based on spatial redundancy allied to a simple parity scheme per byte to guarantee

memory reliability, namely *Parity per Byte and Duplication* (PBD). Fig. 40 presents the block diagram of the proposed EDAC approach.

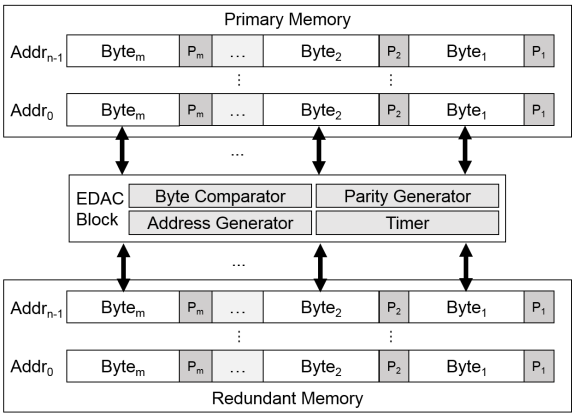

Figure 40 – PBD block diagram
Source: (GOERL et al., 2018)

The obtained results demonstrated the effectiveness of the proposed approach to detect and correct errors in memory systems, either running in a stand-alone mode or connected to the register file of the LEON3 soft-core processor. This approach has the advantage that it can be used either in the register file or the main memory. Though it has a high footprint, EDAC techniques with this level of reliability are not far from this implementation.

5.3 SRAM FPGA CONSIDERATIONS

When using SRAM FPGAs for space applications, additionally to the user configuration (i.e., implemented hardware), the configuration memory can also suffer from SEUs. Unwanted modification of configuration memory bits can compromise the entire system. A well-known and established technique to mitigate this problem is with the use of a memory scrubber (BERG et al., 2008). Xilinx provides an IP to implement scrubbing on configuration memory of their FPGAs, being a practical solution to integrate into the system. It is also possible to

perform configuration memory scrubbing via JTAG interface from an external device.

However, the scrubbing process implies on halting the system for a period of time. Usually, the scrub process is done using the *Partial Reconfiguration* (PR) feature available on SRAM FPGAs. PR allows to read and write to a portion of configuration memory inside the FPGA. The works in (KELLER; WIRTHLIN, 2017) and (LINDOSO et al., 2017) presents solutions based on out-of-the-box scrubbing techniques.

Additionally, the configuration memory can suffer from permanent faults. Permanent faults are critical and can occur due to radiation effects in the device as a function of TID and TID-Imprinted Effect (BENFICA et al., 2016). Note that permanent faults on memory elements are not destructive to the FPGA and therefore can be isolated and/or reallocated.

Accordingly, once the PR feature is available, the developed work in (MARTINS et al., 2018) presents a technique to mitigate permanent faults on SRAM FPGAs. If the permanent fault is not destructive, there is a high probability that it affects only its location inside the FPGA. To deal with the faulty region, the work proposes a modified design flow, based on *Reconfigurable Partition*s (RPs) with the same interface, using a set of design rules to maintain compatibility. Whenever a permanent fault is detected inside a RP, it can be switched to another RP (if available) free of permanent faults. Fig. 41 shows an overview of the RPs compatibility scheme needed to perform the swap.

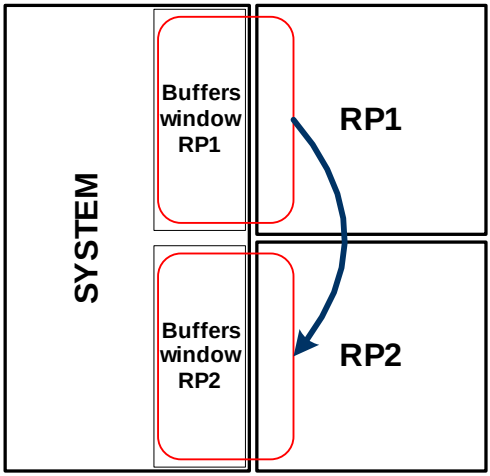

Figure 41 – Overview of the RP compatibility
Source: (MARTINS et al., 2018)

6 CONCLUSIONS AND FUTURE WORK

In this thesis, we presented a fault-tolerant architecture using the checkpoint recovery technique for soft-core processors aimed at space-applications using FPGAs. The related work on the area shows there is room for improvement on time-redundancy FT techniques.

From our design premises, we picked the LEON3 soft-core processor as the test vehicle. The LEON3 is already used on space missions with its paid fault-tolerant version (LEON3FT - (Aeroflex Gaisler, 2015c)).

We named this technique as LEON3 **C**heckpoint **Re**covery **F**ault-**T**olerant (LEON3CReFT). All modifications made to the GR-LIB (Aeroflex Gaisler, 2015a) are available under this link[7], as required by the GPL-3.0.

The fault injection campaign was described in detail and the results for three different architectures were compared for a set of programs.

From our experimental results, it was shown that the CR technique is a valid alternative to TMR and even DMR. This conclusion is valid also for the limited logic area and power budget, subjects of interest in satellites. The constraints are allied to comparable levels of reliability. In our approach, there is no need to perform modifications to the software source code or compiler.

Some of the minor shortcomings of the presented technique can be easily addressed with on-going works, to improve the reliability of the microprocessor to a space-grade level.

As for future works, the designed system must be validate with a faster fault-injection mechanism, such as FTUNSHADES (GUZMAN-MIRANDA; AGUIRRE; TOMBS, 2009; MOGOLLON et al., 2011). Also, we aim to perform analysis of SEU-susceptibility for combined effects of EMI and TID as a continuation of the work in (VILLA et al., 2017). Fig. 42 shows the test setup of Pelletron Accelerator (MEDINA et al., 2014) in USP/Brazil with the *Device Under Test* (DUT) Microsemi

[7] https://github.com/prcvilla/leon3creft

ProASIC3e-1500 FPGA used for heavy-ion experimentation. For the X-ray TID, Fig. 43 presents the setup used for the same DUT on FEI University.

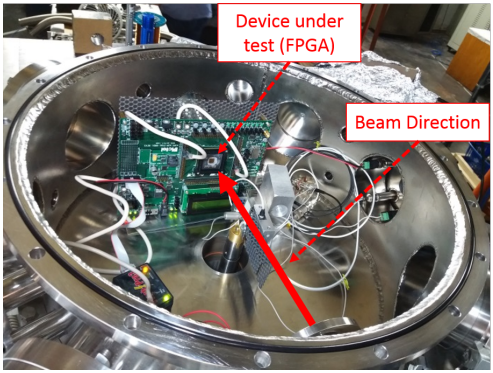

Figure 42 – Heavy-Ion test chamber in Pelletron Accelerator facilities

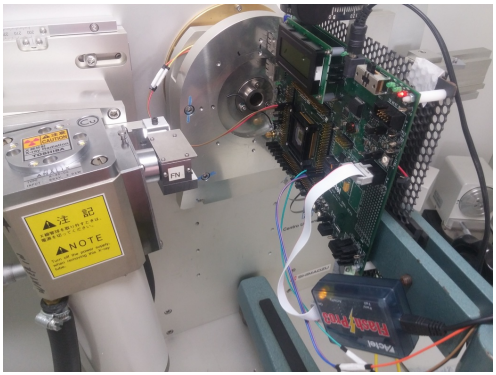

Figure 43 – X-ray test in FEI university facilities

BIBLIOGRAPHY

Aeroflex Gaisler. *GRLIB IP Library*. 2015. Disponível em: <http://www.gaisler.com/index.php/products/ipcores/soclibrary>. 65, 66, 68, 101

Aeroflex Gaisler. *LEON3 Processor*. 2015. Disponível em: <http://www.gaisler.com/index.php/products/processors/leon3>. 33, 65

Aeroflex Gaisler. *LEON3FT-RTAX Fault-tolerant Processor*. 2015. Disponível em: <http://www.gaisler.com/index.php/products/components/leon3ft-rtax>. 32, 101

AHMED, R.; FRAZIER, R.; MARINOS, P. Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems. In: *Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*. IEEE Comput. Soc. Press, 1990. p. 82–88. ISBN 0-8186-2051-X. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=89338>. 55

ALKHAFAJI, F. S. M. et al. Robotic Controller: ASIC versus FPGA - A Review. *Journal of Computational and Theoretical Nanoscience*, v. 15, n. 1, 2018. 31

ARGYRIDES, C.; PRADHAN, D. K.; KOCAK, T. Matrix Codes for Reliable and Cost Efficient Memory Chips. *IEEE Trans. VLSI Syst.*, v. 19, n. 3, p. 420–428, 3 2011. ISSN 1063-8210. Disponível em: <http://ieeexplore.ieee.org/document/5352255/>. 92

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, p. 11–33, 1 2004. ISSN 1545-5971. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1335465>. 40, 41, 79

BARBOZA, S. H. I. et al. SAMPA chip: a new ASIC for the ALICE TPC and MCH upgrades. *Journal of Instrumentation*, IOP Publishing, v. 11, n. 02, p. C02088, 2016. 31

BARNABY, H. J. Total-Ionizing-Dose Effects in Modern CMOS Technologies. *IEEE Transactions on Nuclear Science*, v. 53, n. 6, p. 3103–3121, 12 2006. ISSN 0018-9499. Disponível em: <http://ieeexplore.ieee.org/document/4033191/>. 39, 40

BATTEZZATI, N.; STERPONE, L.; VIOLANTE, M. *Reconfigurable field programmable gate arrays for mission-critical applications*. Springer, 2010. Disponível em: <https://books.google.com.br/books?hl=en&lr=&id=iVScPZCgp_EC&oi=fnd&pg=PP5&dq=reconfigurable+field+programmable+gate+arrays+for+mission-critical+applications&ots=fjBZtSQupc&sig=jx3fKoLJ61msyfpPwqZJVTtT5lo>. 37, 41, 43

BAUMANN, R. *Impact of Single-Event Upsets in Deep-Submicron Silicon Technology*. Cambridge University Press, 2003. 117–120 p. Disponível em: <http://journals.cambridge.org/abstract_S0883769400017516>. 32

BENFICA, J. et al. Analysis of SRAM-Based FPGA SEU Sensitivity to Combined EMI and TID-Imprinted Effects. *IEEE Transactions on Nuclear Science*, IEEE, v. 63, n. 2, p. 1294–1300, 4 2016. ISSN 0018-9499. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7454850>. 98

BERG, M. et al. Effectiveness of Internal Versus External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA: Design, Test, and Analysis. *IEEE Transactions on Nuclear Science*, v. 55, n. 4, p. 2259–2266, 8 2008. ISSN 0018-9499. Disponível em: <http://ieeexplore.ieee.org/document/4636940/>. 97

BERNARDESCHI, C.; CASSANO, L.; DOMENICI, A. SRAM-Based FPGA Systems for Safety-Critical Applications: A Survey on Design Standards and Proposed Methodologies. *Journal of Computer Science and Technology*, v. 30, n. 2, p. 373–390, 3 2015. ISSN 1000-9000. Disponível em: <http://link.springer.com/10.1007/s11390-015-1530-5>. 32, 34

BOUHALI, M. et al. FPGA Applications in Unmanned Aerial Vehicles - A Review. In: WONG, S. et al. (Ed.). *Applied Reconfigurable Computing*. Cham: Springer International Publishing, 2017. p. 217–228. ISBN 978-3-319-56258-2. 31

CASTRO, H. d. S. et al. A correction code for multiple cells upsets in memory devices for space applications. In: *2016 14th IEEE International New Circuits and Systems Conference (NEWCAS)*. IEEE, 2016. p. 1–4. ISBN 978-1-4673-8900-6. Disponível em: <http://ieeexplore.ieee.org/document/7604783/>. 92

CETIN, E. et al. Overview and Investigation of SEU Detection and Recovery Approaches for FPGA-Based Heterogeneous Systems. In:

FPGAs and Parallel Architectures for Aerospace Applications. Cham: Springer International Publishing, 2016. p. 33–46. Disponível em: <http://link.springer.com/10.1007/978-3-319-14352-1_3>. 46

EEJournal. *The Biggest SoC/FPGAs*. 2017. Disponível em: <https://www.eejournal.com/article/the-biggest-socfpgas/>. 31

Enshan Yang et al. HHC: Hierarchical hardware checkpointing to accelerate fault recovery for SRAM-based FPGAs. In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, 2013. p. 193–198. ISBN 978-1-4799-0664-2. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6604078>. 54, 55, 61

FERLINI, F. et al. Non-intrusive fault tolerance in soft processors through circuit duplication. In: *2012 13th Latin American Test Workshop (LATW)*. IEEE, 2012. p. 1–6. ISBN 978-1-4673-2355-0. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6261264>. 70, 74

FOUAD, S. et al. Context-aware resources placement for SRAM-based FPGA to minimize checkpoint/recovery overhead. In: *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, 2014. p. 1–6. ISBN 978-1-4799-5944-0. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7032506>. 54, 55

FRIEND, R. B.; ARROYO, C.; HANSEN, J. Big Missions, Small Solutions Advances and Innovation in Architecture and Technology for Small Satellites. In: *AIAA SPACE 2016*. Reston, Virginia: American Institute of Aeronautics and Astronautics, 2016. ISBN 978-1-62410-427-5. Disponível em: <http://arc.aiaa.org/doi/10.2514/6.2016-5229>. 31

GARDENYES, R. B. *Trends and patterns in ASIC and FPGA use in space missions and impact in technology roadmaps of the European Space Agency*. Tese (Doutorado), 2012. 31, 32

GLEIN, R. BRAM Radiation Sensor for a Self-Adaptative SEU Mitigation. In: *SpacE FPGA Users Workshop*. [S.l.: s.n.], 2014. 32, 34

GOERL, R. C. et al. An Efficient EDAC Approach for Handling Multiple Bit Upsets in Memory Array. In: *29th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis - Unpublished*. [S.l.: s.n.], 2018. 96, 97, 113

GOLOUBEVA, O. et al. *Software-implemented hardware fault tolerance*. Springer, 2006. Disponível em: <https://books.google.com.br/books?hl=en&lr=&id=qX9GAAAAQBAJ&oi=fnd&pg=PA1&dq=software+implemented+hardware+fault-tolerance&ots=owaXCAdHzD&sig=G5Ql7eRDVfTwvyZRIrP4zYxQsw8>. 42, 43, 44, 45, 47

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. IEEE Computer Society, p. 3–14, 12 2001. Disponível em: <http://dl.acm.org/citation.cfm?id=1128020.1128563>. 49, 57

GUZMÁN, D. et al. A Low Power Processors for Cubesat Missions. In: *8th Annual Cubesat Developer's Workshop 2011*. [S.l.: s.n.], 2011. 32, 34

GUZMAN-MIRANDA, H.; AGUIRRE, M.; TOMBS, J. Noninvasive Fault Classification, Robustness and Recovery Time Measurement in Microprocessor-Type Architectures Subjected to Radiation-Induced Errors. *IEEE Transactions on Instrumentation and Measurement*, v. 58, n. 5, p. 1514–1524, 5 2009. ISSN 0018-9456. Disponível em: <http://ieeexplore.ieee.org/document/4787115/>. 101

HENKEL, J. et al. Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends. In: *Proceedings of the 50th Annual Design Automation Conference*. New York, NY, USA: ACM, 2013. (DAC '13), p. 99:1–99:10. ISBN 978-1-4503-2071-9. Disponível em: <http://doi.acm.org/10.1145/2463209.2488857>. 46

HENNESSY, J.; PATTERSON, D. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2002. Disponível em: <https://books.google.com.br/books?hl=en&lr=&id=v3-1hVwHnHwC&oi=fnd&pg=PP2&dq=patterson&ots=H1SiJWb4xK&sig=T9qPPtq6KSFb0h4sBlLbHPwYBpM>. 57, 67

INPE. *Plataforma Multimissão (PMM)*. 2018. Disponível em: <http://www3.inpe.br/amazonia-1/sobre_satelite/pmm.php>. 33

JOHNSON, M. *Superscalar multiprocessor design*. Prentice-Hall, Inc., 1991. ISBN 0-13-875634-1. Disponível em: <http://dl.acm.org/citation.cfm?id=102832>. 56

KANG, S.-h. et al. Optimal Checkpoint Selection with Dual-Modular Redundancy Hardening. *IEEE Transactions on Computers*, PP, n. 99, p. 1–1, 2014. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6880324>. 21, 52, 53, 61

KASTENSMIDT, F. L. et al. TID in Flash-Based FPGA: Power Supply-Current Rise and Logic Function Mapping Effects in Propagation-Delay Degradation. *IEEE Transactions on Nuclear Science*, IEEE, v. 58, n. 4, p. 1927–1934, 8 2011. ISSN 0018-9499. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5752883>. 64

KELLER, A. M.; WIRTHLIN, M. J. Benefits of Complementary SEU Mitigation for the LEON3 Soft Processor on SRAM-Based FPGAs. *IEEE Transactions on Nuclear Science*, v. 64, n. 1, p. 519–528, 1 2017. ISSN 0018-9499. Disponível em: <http://ieeexplore.ieee.org/document/7763831/>. 48, 98

KLETZING, C. A. et al. The Electric and Magnetic Field Instrument Suite and Integrated Science (EMFISIS) on RBSP. *Space Science Reviews*, v. 179, n. 1-4, p. 127–181, 6 2013. ISSN 0038-6308. Disponível em: <http://link.springer.com/10.1007/s11214-013-9993-6>. 32, 34

KOCH, D.; HAUBELT, C.; TEICH, J. Efficient hardware checkpointing. In: *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays - FPGA '07*. New York, New York, USA: ACM Press, 2007. p. 188. ISBN 9781595936004. Disponível em: <http://dl.acm.org/citation.cfm?id=1216919.1216950>. 21, 53, 54, 61

KOREN, I.; KRISHNA, C. *Fault-tolerant systems*. Morgan Kaufmann, 2010. Disponível em: <https://books.google.com.br/books?hl=en&lr=&id=o_Pjbo4Wvp8C&oi=fnd&pg=PR11&dq=fault+tolerant+systems+koren&ots=RYPEQBzbyA&sig=pMKkYxL70ahe4U4U3hTKWLlrR3Y>. 34, 43, 44, 63

LESAGE, L.; MEJIAS, B.; LOBELLE, M. A software based approach to eliminate all SEU effects from mission critical programs. In: *2011 12th European Conference on Radiation and Its Effects on Components and Systems*. IEEE, 2011. p. 467–472. ISBN 978-1-4577-0586-1. Disponível em: <http://ieeexplore.ieee.org/document/6131353/>. 64

LI, T. et al. Fine-Grained Checkpoint Recovery for Application-Specific Instruction-Set Processors. *IEEE Transactions on Computers*, v. 66, n. 4, p. 647–660, 4 2017. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/document/7562290/>. 48, 49, 61

LI, T. et al. CSER: HW/SW configurable soft-error resiliency for application specific instruction-set processors. In: *DATE '13 Proceedings of the Conference on Design, Automation and Test in Europe*. EDA

Consortium, 2013. p. 707–712. ISBN 978-1-4503-2153-2. Disponível em: <http://dl.acm.org/citation.cfm?id=2485288.2485460>. 58, 59, 60

LI, T. et al. DHASER: dynamic heterogeneous adaptation for soft-error resiliency in ASIP-based multi-core systems. In: *ICCAD '13 Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013. p. 646–653. ISBN 978-1-4799-1069-4. Disponível em: <http://dl.acm.org/citation.cfm?id=2561828.2561955>. 59

LINDOSO, A. et al. A Hybrid Fault-Tolerant LEON3 Soft Core Processor Implemented in Low-End SRAM FPGA. *IEEE Transactions on Nuclear Science*, v. 64, n. 1, p. 374–381, 1 2017. ISSN 0018-9499. Disponível em: <http://ieeexplore.ieee.org/document/7776886/>. 48, 98

LLC, S. *SimpleScalar*. 2015. Disponível em: <http://www.simplescalar.com/>. 57

MARTINS, V. M. G. et al. Soft Errors Analysis on FPGAs for CubeSat Missions. In: *II Latin American IAA CubeSat Workshop*. [S.l.: s.n.], 2016. 113

MARTINS, V. M. G. et al. A TMR Strategy with Enhanced Dependability Features Based on a Partial Reconfiguration Flow. In: *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015. v. 07-10-July, p. 161–166. ISBN 978-1-4799-8719-1. ISSN 21593477 21593469. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7309556http://ieeexplore.ieee.org/document/7309556/>. 70, 114

MARTINS, V. M. G. et al. The experience of designing and developing the on-board electronics of a Cubesat in Brazil. In: *I Latin American IAA CubeSat Workshop*. [S.l.: s.n.], 2014. 114

MARTINS, V. M. G. et al. A dynamic partial reconfiguration design flow for permanent faults mitigation in FPGAs. *Microelectronics Reliability*, v. 83, p. 50–63, 4 2018. ISSN 00262714. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0026271418300118>. 98, 99, 113

MCCOLLUM, J. ASIC versus antifuse FPGA reliability. In: *2009 IEEE Aerospace conference*. IEEE, 2009. p. 1–11. ISBN 978-1-4244-2621-8. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4839526>. 64

MEDINA, N. H. et al. First Successful SEE Measurements with Heavy Ions in Brazil. In: *2014 IEEE Radiation Effects Data Workshop (REDW)*. IEEE, 2014. p. 1–3. ISBN 978-1-4799-5884-9. Disponível em: <http://ieeexplore.ieee.org/document/7004571/>. 101

Microsemi Inc. *ProASIC3 FPGA*. 2017. Disponível em: <https://www.microsemi.com/products/fpga-soc/fpga/proasic3-overview>. 64

Microsemi Inc. *Power Estimators and Calculators*. 2018. Disponível em: <https://www.microsemi.com/products/fpga-soc/design-resources/power-calculator>. 90

MOGOLLON, J. et al. FTUNSHADES2: A novel platform for early evaluation of robustness against SEE. In: *2011 12th European Conference on Radiation and Its Effects on Components and Systems*. IEEE, 2011. p. 169–174. ISBN 978-1-4577-0586-1. Disponível em: <http://ieeexplore.ieee.org/document/6131392/>. 101

NASA/JPL-Caltech/SwRI. *PIA16938: Sources of Ionizing Radiation in Interplanetary Space*. 2018. Disponível em: <https://photojournal.jpl.nasa.gov/catalog/PIA16938>. 38

NORTON, C. D. et al. An evaluation of the Xilinx Virtex-4 FPGA for on-board processing in an advanced imaging system. In: *2009 IEEE Aerospace conference*. IEEE, 2009. p. 1–9. ISBN 978-1-4244-2621-8. Disponível em: <http://ieeexplore.ieee.org/abstract/document/4839460/http://ieeexplore.ieee.org/document/4839460/>. 31

OH, N.; SHIRVANI, P.; MCCLUSKEY, E. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, v. 51, n. 1, p. 111–122, 3 2002. ISSN 00189529. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=994926>. 56

OH, N.; SHIRVANI, P.; MCCLUSKEY, E. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, v. 51, n. 1, p. 63–75, 3 2002. ISSN 00189529. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=994913>. 56

OLIVEIRA, A. B. de; TAMBARA, L. A.; KASTENSMIDT, F. L. Applying lockstep in dual-core ARM Cortex-A9 to mitigate radiation-induced soft errors. In: *2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 2017. p. 1–4. ISBN 978-1-5090-5859-4. Disponível em: <http://ieeexplore.ieee.org/document/7948063/>. 50, 51, 61

PETKOV, M. The effects of space environments on electronic components. 2003. Disponível em: <https://trs.jpl.nasa.gov/handle/2014/7193>. 63

RAGEL, R.; PARAMESWARAN, S. IMPRES: integrated monitoring for processor reliability and security. In: *2006 43rd ACM/IEEE Design Automation Conference*. IEEE, 2006. p. 502–505. ISBN 1-59593-381-6. ISSN 0738-100X. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1688849>. 57

RAGEL, R.; PARAMESWARAN, S. Reli: Hardware/software Checkpoint and Recovery scheme for embedded processors. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012. p. 875–880. ISBN 978-1-4577-2145-8. ISSN 1530-1591. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6176621>. 49, 57, 58, 59, 64

REIS, G. et al. SWIFT: Software Implemented Fault Tolerance. In: *International Symposium on Code Generation and Optimization*. IEEE, 2005. p. 243–254. ISBN 0-7695-2298-X. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1402092>. 56, 59, 61

REORDA, M. et al. A low-cost SEE mitigation solution for soft-processors embedded in Systems on Programmable Chips. *2009 Design, Automation & Test in Europe Conference & Exhibition*, European Design and Automation Association, p. 352–357, 4 2009. ISSN 1530-1591. Disponível em: <http://dl.acm.org/citation.cfm?id=1874620.1874704>. 64

RODRIGUEZ-ANDINA, J. J.; VALDES-PENA, M. D.; MOURE, M. J. Advanced Features and Industrial Applications of FPGAs—A Review. *IEEE Transactions on Industrial Informatics*, v. 11, n. 4, p. 853–864, 8 2015. ISSN 1551-3203. Disponível em: <http://ieeexplore.ieee.org/document/7104117/>. 31

SABENA, D. et al. Reconfigurable high performance architectures: How much are they ready for safety-critical applications? In: *2014 19th IEEE European Test Symposium (ETS)*. IEEE, 2014. p. 1–8. ISBN 978-1-4799-3415-7. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6847820>. 32, 34

SIEGLE, F.; VLADIMIROVA, T. Mitigation of Radiation Effects in SRAM-Based FPGAs. *ACM Computing Surveys (CSUR)*, v. 47, n. 2, 2015. ISSN 0360-0300. 38, 39

SIEWIOREK, D.; SWARZ, R. *Reliable Computer Systems: Design and Evaluatuion*. [S.l.]: Digital Press, 2017. 46

STUIJK, S.; GEILEN, M.; BASTEN, T. SDFˆ3: SDF For Free. In: *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*. IEEE, 2006. p. 276–278. ISBN 0-7695-2556-3. ISSN 1550-4808. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1640245>. 53

T. Li et al. ReCoRD: Reducing Register Traffic for Checkpointing in Embedded Processors. In: *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. San Jose, CA, USA: EDA Consortium, 2016. (DATE '16), p. 582–587. ISBN 978-3-9815370-6-2. Disponível em: <http://dl.acm.org/citation.cfm?id=2971808.2971945http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7459379>. 49

TANG, H. H.; OLSSON, N. Single-Event Upsets in Microelectronics. *MRS Bulletin*, Cambridge University Press, v. 28, n. 02, p. 107–110, 2 2003. ISSN 1938-1425. Disponível em: <http://journals.cambridge.org/abstract_S0883769400017498>. 32

TORRENS, G. FPGA-SRAM Soft Error Radiation Hardening. In: *Field - Programmable Gate Array*. InTech, 2017. Disponível em: <http://www.intechopen.com/books/field-programmable-gate-array/fpga-sram-soft-error-radiation-hardening>. 31

TRAVESSINI, R. et al. Processor core profiling for SEU effect analysis. In: *2018 IEEE 19th Latin-American Test Symposium (LATS)*. IEEE, 2018. p. 1–6. ISBN 978-1-5386-1472-3. Disponível em: <https://ieeexplore.ieee.org/document/8347235/>. 95, 96, 113

U.S. State Department. *Directorate of Defense Trade Controls*. 2018. Disponível em: <http://pmddtc.state.gov/index.html>. 33

VELAZCO, R.; FOUILLAT, P.; REIS, R. (Ed.). *Radiation Effects on Embedded Systems*. Dordrecht: Springer Netherlands, 2007. ISBN 978-1-4020-5645-1. Disponível em: <http://link.springer.com/10.1007/978-1-4020-5646-8>. 37

VILLA, P. et al. Analysis of COTS FPGA SEU-sensitivity to combined effects of conducted-EMI and TID. In: *Proceedings of the 2017 11th International Workshop on the Electromagnetic Compatibility of Integrated Circuits, EMCCompo 2017*. [S.l.: s.n.], 2017. ISBN 9781538626894. 101, 113

VILLA, P. et al. A reconfigurable hardware platform for power converter control systems. In: *Proceedings of the IEEE International Conference on Industrial Technology*. [S.l.: s.n.], 2015. v. 2015-June. 113

VILLA, P. R. C. et al. Analysis of single-event upsets in a Microsemi ProAsic3E FPGA. In: *2017 18th IEEE Latin American Test Symposium (LATS)*. IEEE, 2017. p. 1–4. ISBN 978-1-5386-0415-1. Disponível em: <http://ieeexplore.ieee.org/document/7906772/>. 64, 113

VILLA, P. R. C. et al. A complete CubeSat mission: the Floripa-Sat experience. In: *I Latin American IAA CubeSat Workshop*. [S.l.: s.n.], 2014. 114

VILLA, P. R. C. et al. Processor checkpoint recovery for transient faults in critical applications. In: *2018 IEEE 19th Latin-American Test Symposium (LATS)*. IEEE, 2018. p. 1–6. ISBN 978-1-5386-1472-3. Disponível em: <https://ieeexplore.ieee.org/document/8349674/>. 113

VIOLANTE, M. et al. A Low-Cost Solution for Deploying Processor Cores in Harsh Environments. *IEEE Transactions on Industrial Electronics*, v. 58, n. 7, p. 2617–2626, 7 2011. ISSN 0278-0046. Disponível em: <http://ieeexplore.ieee.org/document/5740344/>. 21, 51, 52, 61, 64

WILSON, D. S. Cubesat Flight Software Development. In: *2011 Workshop on Spacecraft Flight Software (FSW11)*. Baltimore: [s.n.], 2011. 32, 34

WU, K.-L.; FUCHS, W.; PATEL, J. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems*, v. 1, n. 2, p. 231–240, 4 1990. ISSN 10459219. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=80134>. 55

APPENDIX A – PUBLICATIONS

A.1 JOURNAL PAPER

- A dynamic partial reconfiguration design flow for permanent faults mitigation in FPGAs. In: Microelectronics Reliability, 2018. (MARTINS et al., 2018)

A.2 CONFERENCE PAPERS

1. Processor Checkpoint Recovery for Transient Faults in Critical Applications. In: 19th IEEE Latin-American Test Symposium (LATS), 2018. (VILLA et al., 2018)

2. Processor Core Profiling for SEU Effect Analysis. In: 19th IEEE Latin-American Test Symposium (LATS), 2018. (TRAVESSINI et al., 2018)

3. An Efficient EDAC Approach for Handling Multiple Bit Upsets in Memory Array. In: 29th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis (ESREF), 2018. *Unpublished.* (GOERL et al., 2018)

4. Analysis of COTS FPGA SEU-sensitivity to combined effects of conducted-EMI and TID. In: 2017 11th International Workshop on the Electromagnetic Compatibility of Integrated Circuits (EMCCompo), 2017. (VILLA et al., 2017)

5. Analysis of SEU Sensitivity in a Commercial FPGA. In: 18th IEEE Latin-American Test Symposium (LATS), 2017. (VILLA et al., 2017)

6. Soft Errors Analysis on FPGAs for CubeSat Missions. In: II Latin American IAA CubeSat Workshop, 2016. (MARTINS et al., 2016)

7. A reconfigurable hardware platform for power converter control systems. In: 2015 IEEE International Conference on Industrial Technology (ICIT), 2015. (VILLA et al., 2015)

8. A TMR Strategy with Enhanced Dependability Features Based on a Partial Reconfiguration Flow. In: 2015 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2015. (MARTINS et al., 2015)

9. A complete CubeSat mission: the Floripa-Sat experience. In: 1st Latin American IAA CubeSat Workshop, 2014. (VILLA et al., 2014)

10. The experience of designing and developing the on-board electronics of a Cubesat in Brazil. In: I Latin American IAA CubeSat Workshop, 2014. (MARTINS et al., 2014)

APPENDIX B – WORKLOAD SOURCE CODE

B.1 BASIC

```
#include <stdio.h>
#include "gpio.h"

#define CALC 6
#define FIN 7
#define ERR 8

volatile int x,a=9,b=8,c=4,d=5,i=0;

int main(){
        GPIO_SET_OUTPUTS;
        GPIO_WRITE(0x0);
        while(i<IRUNS){
                GPIO_SETPIN(CALC);
                x=(a+b)-(c+d);
                if(x!=8){GPIO_SETPIN(ERR);}
                GPIO_WRITE(0x0);
                i++;
        }
        GPIO_SETPIN(FIN);
        return 0;
}
```

B.2 BSORT

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#ifndef BUBBLESORT_H
#define BUBBLESORT_H

int checkarr(uint8_t *);

void fillArray (uint8_t *);

void bubblesort(uint8_t* arr);

#endif
#include "bubblesort.h"

int checkarr(uint8_t *arr){
        int i;
        for(i=0;i<BSIZE-1;i++){
                if(arr[i] > arr[i+1]) return -1;
        }
        return 0;
}

void fillArray (uint8_t *arr){
        int i=0;
        for(i=0;i<BSIZE;i++){
                //~ arr[i] = (uint8_t)(rand()%255);
                arr[i] = (uint8_t)(9-i);
        }
}
```

```
void bubblesort(uint8_t* arr){
        int swapped = 1, x, y, index2;
        uint8_t tmp;
        for (x = 0; (x < BSIZE) && swapped; x++) {
                swapped = 0;
                for (y = 0; y < BSIZE - x - 1; y++) {
                        index2 = y + 1;
                        if (arr[y] > arr[index2]) {
                                tmp = arr[y];
                                arr[y] = arr[index2];
                                arr[index2] = tmp;
                                swapped = 1;
                        }
                }
        }
}

#include <stdio.h>
#include "gpio.h"
#include "bubblesort.h"

#define CALC 6
#define FIN 7
#define ERR 8

 volatile int i=0;
uint8_t bsArray0[BSIZE];

int main(){
        GPIO_SET_OUTPUTS;
        GPIO_WRITE(0x0);
        while(i<IRUNS){
                GPIO_SETPIN(CALC);

                 fillArray (bsArray0);
                bubblesort(bsArray0);

                if(checkarr(bsArray0)!=0){
                        GPIO_SETPIN(ERR);
                }

                GPIO_WRITE(0x0);
                i++;
        }
        GPIO_SETPIN(FIN);
        return 0;
}
```

B.3 NMEA

```
#include <stdio.h>
#include "gpio.h"

#define CALC 6
#define FIN 7
#define ERR 8

int checksum(const char*);

int checksum(const char *s) {
```

```
        int c = 0;

        while(*s)
                c ^= *s++;

        return c;
}

 volatile int i=0;
//correct checksum = 0x76 = 118 decimal
const char *nmeamsg = "GPGGA,092750.000,5321.6802,N,00630.3372,W,1,8,1.03,61.7,M,55.2,M,,";

int main(void) {
        int chks;
        GPIO_SET_OUTPUTS;
        GPIO_WRITE(0x0);

        while(i<IRUNS){
                GPIO_SETPIN(CALC);
                chks=checksum(nmeamsg);

                if(chks!=118) {
                        GPIO_SETPIN(ERR);
                }

                GPIO_WRITE(0x0);
                i++;
        }
        GPIO_SETPIN(FIN);
        return 0;
}
```

B.4 HAMMING

```
#include <stdio.h>
#include "gpio.h"

#define CALC 6
#define FIN 7
#define ERR 8

/*Code Generator Matrix*/
const unsigned char G[7][4] = {
        {1,1,0,1},
        {1,0,1,1},
        {1,0,0,0},
        {0,1,1,1},
        {0,1,0,0},
        {0,0,1,0},
        {0,0,0,1}
};

/*Data to be transmitted*/
const unsigned char data[4] = {1,0,1,0};

/*Encoded msg*/
unsigned char msg[7] = {0,0,0,0,0,0,0};

/*Verify*/
unsigned char verify[7] = {1,0,1,1,0,1,0};
```

```
void encode(unsigned char*,const unsigned char*);

void encode(unsigned char *encoded, const unsigned char *dt){
        int _i,_j;
        for(_i=0;_i<7;_i++){
                for(_j=0;_j<4;_j++){
                        encoded[_i] += G[_i][_j] & dt[_j]; /*it is possible to either AND it or multiply*/
                }
                encoded[_i]&=1;
        }
}

volatile int i=0;

int main(void){
        int x, error=0;
        GPIO_SET_OUTPUTS;
        GPIO_WRITE(0x0);

        while(i<IRUNS) {
                GPIO_SETPIN(CALC);
                encode(msg,data);

                for(x=0;x<7;x++){
                        if(msg[x]!=verify[x]) error=1;
                }
                if(error) {
                        GPIO_SETPIN(ERR);
                }

                for(x=0;x<7;x++) msg[x]=0; //msg initial state
                error=0;
                GPIO_WRITE(0x0);
                i++;
        }
        GPIO_SETPIN(FIN);
        return 0;
}
```

B.5 GPIO.H

```
#include <stdio.h>
#include <stdint.h>

#ifndef __GPIO_H__
#define __GPIO_H__

#define GPIO_IN *((volatile unsigned int *)0x80000800)
#define GPIO_OUT *((volatile unsigned int *)0x80000804)
#define GPIO_DIR *((volatile unsigned int *)0x80000808)

#define GPIO_SET_OUTPUTS GPIO_DIR=0xffffffff
#define GPIO_WRITE(_v) GPIO_OUT=_v
#define GPIO_SETPIN(_p) GPIO_OUT|=(1<<_p)
#define GPIO_CLEARPIN(_p) GPIO_OUT&=~(1<<_p)
#define GPIO_READ (int32_t)GPIO_IN

#endif
```