UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

KLEBER PORTO DOS SANTOS

# Análise de eficiência, cobertura de erros e custos da técnica de TMR Heterogêneo em um processador VLIW

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Antonio C. S. Beck Filho
Co-orientador: Doutorando Anderson Luiz Sartor

Porto Alegre
2016

## AGRADECIMENTOS

Agradeço aos meus pais e demais familiares pelo suporte oferecido durante o meu desenvolvimento como estudante, aos meus Orientadores que estiveram sempre dispostos a sanar dúvidas e a ajudar com ideias que contribuíram para o desenvolvimento deste trabalho. E um agradecimento à minha namorada pelo apoio que me foi dado durante a graduação.

## RESUMO

O aumento da integração de componentes eletrônicos torna-os também mais suscetíveis a falhas transientes causadas por radiação, até mesmo quando tais componentes se encontram dentro da atmosfera terrestre. Assim, é necessário o uso de técnicas para prevenir danos causados em sistemas críticos, como missões espaciais, extração de petróleo, aviação civil, carros inteligentes, entre outras. Entretanto, existem aplicações que atendem nichos específicos e possuem diferentes características, tendo um foco maior na redução do consumo de potência ou uma necessidade maior de tolerar falhas, ou seja, algumas aplicações precisam um maior nível proteção que outras. Devido aos problemas citados acima, este trabalho visa aplicar técnicas de tolerância a falhas em um processador VLIW, arquitetura que utiliza ILP (*Instruction Level Paralallelism*) para atingir maior desempenho. Mais especificamente, implementa, em VHDL, a técnica de Redundância Modular Tripla Heterogênea ao processador VLIW ρ-VEX, utilizando diferentes variações de sua microarquitetura. Sempre considerando um conjunto de três aplicações: é capaz de proteger a mais crítica com a técnica TMR, uma segunda aplicação menos crítica com a técnica DMR e uma aplicação executando sem proteção. Foram realizadas análises de custo em termos de performance, consumo energético e cobertura de erros utilizando ferramentas de validação e simuladores.



**Palavras-chave**: TMR. VLIW. Tolerância a Falhas. Softcore. ρ-VEX.

**Efficiency, Fault Coverage and Cost analysis of the Heterogeneous TMR Technique Applied to a VLIW Processor**


## ABSTRACT

The increasing integration of electronic components also increases the likelihood of single event upsets due to radiation, even when said components are within the Earth's atmosphere. Therefore, techniques are needed to prevent damage caused to critical systems, such as space missions, oil extraction, aviation, smart cars, among others. However, there are applications with distinct characteristics, aimed to reduce power consumption or that need higher fault tolerance, meaning that some applications may need a higher level of protection against faults than others. This paper makes use of fault tolerance techniques in a VLIW processor, an architecture that uses ILP (Instruction Level Parallelism) to achieve a higher performance. More precisely, we implemented, in VHDL, the Heterogeneous Triple Modular Redundancy technique to the ρ-VEX softcore processor, using different variations of its micro architecture. It always works with a set of 3 applications, by protecting the most critical one using the TMR technique; the second, which demands less protection, using DMR; and the last, which executes without any protection. We analyze the cost in terms of performance, energy consumption and fault coverage, using specific tools for validation and wave simulation.



**Keywords**: TMR. VLIW. Fault Tolerance. Softcore ρ-VEX.

## LISTA DE FIGURAS

# LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ASIC | Application Specific Integrated Circuits |
| ECC | Error Correction Code |
| FPGA | Field Programable Gate Array |
| HP | Hewllet-Packard |
| ISA | Instruction Set Architecture |
| LUT | Lookup Table |
| SEU | Single Event Upset |
| TMR | Triple Modular Redundancy |
| UF | Unidade Funcional |
| VEX | VLIW Example |
| VLIW | Very Long Instruction Word |

**SUMÁRIO**

# 1 INTRODUÇÃO

O constante avanço e desenvolvimento tecnológico na área da microeletrônica permitiu o desenvolvimento de transistores cada vez menores e assim, aumenta-se a densidade de componentes nos processadores para aproveitar melhor a área disponível, chegando a integração de bilhões de transistores (TOTONI et al., 2012). Isto permitiu a construção de chips com maior capacidade de processamento ocupando a mesma área.

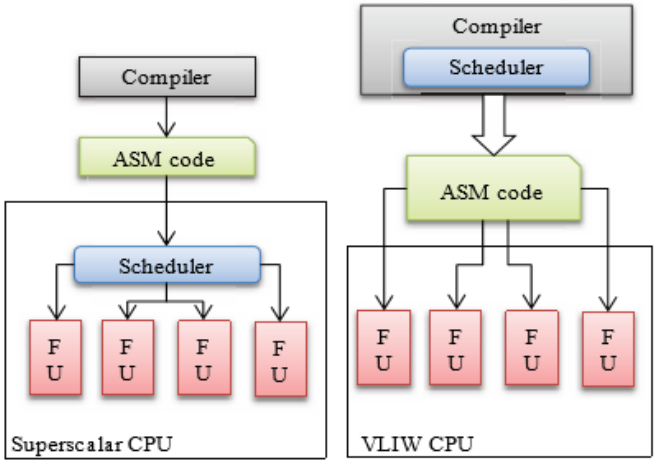
Programas espaciais, bem como sistemas de extração de petróleo no fundo do oceano, são grandes beneficiários do desenvolvimento e aplicação de técnicas que traga maior confiabilidade para seus sistemas de tempo real. Além disso, é necessária uma arquitetura capaz de suprir a demanda pela alta vazão de instruções processadas. Para tanto, são empregadas arquiteturas com grande capacidade de processamento paralelo, como as arquiteturas Superescalar e VLIW (*Very Long Instruction Word*), que precisam ser protegidas através de técnicas de tolerância a falhas para evitar catástrofes e a geração de dados incorretos causados por falhas.

Com a alta densidade de transistores por área, circuitos que operam em faixas reduzidas de tensão e a alta frequência operacional dos processadores, os dispositivos tornam-se suscetíveis a falhas causadas por partículas radioativas, modificando os valores de bits em posições de memória, *flip-flops, latches*, registradores ou dados em processamento nas unidades funcionais (BAUMANN, 2005). Falhas em hardware podem representar até 50% dos defeitos em um sistema computacional (WEBER, 2003). Torna-se então necessária a utilização de técnicas de tolerância a falhas para prevenir que falhas se propaguem e tornem-se erros, causando graves consequências em sistemas críticos.

Em processadores Superescalares, as instruções são analisadas em tempo de execução e despachadas para as diferentes Unidades Funcionais (UFs). Após o término do estágio de execução do pipeline, as instruções são novamente reordenadas, através do buffer de reordenamento, e os dados necessários são gravados na memória e banco de registradores. Isto aumenta a complexidade de hardware, bem como o consumo de energia, devido à necessidade dos processadores Superescalares possuírem fila de instruções, buffer de reordenamento e o hardware dedicado a verificar as dependências de instruções e dados.

A arquitetura VLIW também explora o paralelismo de instruções e, da mesma forma, utiliza diferentes unidades funcionais do processador para executar duas ou mais instruções ao mesmo tempo. Entretanto, ao contrário dos processadores Superescalares, a análise do código para determinar instruções que podem ser executadas concomitantemente (i.e.: não possuem

Figura 1.1: Comparação entre a arquitetura Superscalar e VLIW.




Fonte: (SABENA; REORDA; STERPONE, 2014)

dependências entre si) é feita em tempo de compilação. Após esta análise, o compilador gera uma única palavra, bastante larga (que justifica o nome do processador do processador – VLIW), muitas vezes também chamada de *bundle*, composta por instruções que podem ser executadas simultaneamente. Sem a necessidade de hardware para analisar as instruções em tempo de execução, verificar dependências e reordena-las depois da execução, o hardware de um processador VLIW torna-se mais simples que o de um processador Superescalar. A Figura 1.1 faz um comparativo entre as duas arquiteturas, deixando claro onde ocorre o *Scheduling* de instruções, durante a execução no processador Superescalar, e durante a compilação no processador VLIW.

Neste trabalho, analisou-se viabilidade, em termos de consumo de energia, área e cobertura de falhas, de um sistema provido de um processador VLIW protegido utilizando a técnica de Redundância Modular Tripla Heterogênea, que consiste em utilizar três processadores com diferentes microarquiteturas, mas que implementam a mesma organização (isto é, são capazes de executar o mesmo binário) para realizar a execução das mesmas aplicações e decidir, através de um votador, qual o resultado correto. Além disso, como aplicações possuem diferentes características e diferentes níveis de necessidade de proteção contra falhas, como dito anteriormente, foi desenvolvido então um sistema que executa sempre grupos de três aplicações: sendo uma podendo ser protegida por TMR, executando em três processadores; uma protegida por DMR, executando em apenas dois e uma aplicação rodando em apenas um processador e sem proteção nenhuma.

Devido aos fatores citados acima, foi desenvolvido neste trabalho, o que foi chamado de TMR Heterogêneo, utilizando microarquiteturas 2-*issue*, 4-*issue* e 8-*issue*. Desta forma, é possível também balancear o tempo de execução das aplicações, distribuindo as tarefas uniformemente, e preencher lacunas de tempo em que o hardware esteja ocioso geradas pela diferença de capacidade de processamento entre os processadores, atingir o nível de proteção de um TMR para a aplicação mais crítica, e proteger com DMR uma segunda aplicação, fazendo um equilíbrio entre área ocupada pelo sistema, desempenho e proteção das aplicações.

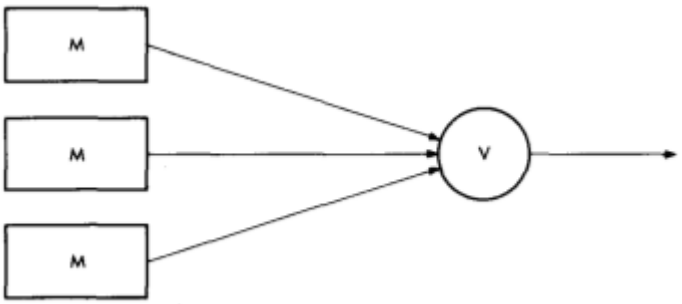
## 1.1 Técnicas de Tolerância a Falhas

Abaixo, serão descritos alguns conceitos, pontos positivos e falhas das técnicas de tolerância a falhas empregadas neste trabalho, sendo elas o TMR (*Triple Modular Redundancy*, ou Redundância Modular Tripla em português) homogêneo, o TMR Heterogêneo e a técnica de DMR (*Dual Modular Redundancy*, ou Redundância Modular Dupla em português).

### 1.1.1 TMR Homogêneo

Como já foi citado, o TMR padrão – ou homogêneo, para diferenciar da técnica proposta – é uma técnica, criada por Von Neumann (LYONS; VANDERKULK, 1962), altamente difundida entre os pesquisadores e a indústria. Consiste na replicação de componentes de hardware – podendo ser no nível das unidades funcionais, *datapaths*, processadores ou até mesmo sistemas inteiros – ou software – executando a mesma instrução, tarefa ou aplicação – com o objetivo de utilizar a redundância para mascarar falhas, como é possível ver na Figura 1.2 feita por Von Neumann. Altamente eficiente contra falhas do tipo *single event upset* (SEU) – erros não permanentes causados por radiação como *bitflips*, por exemplo – faz o uso de um elemento votador para selecionar o resultado com o maior número de votos, dois ou mais, no caso da utilização de três componentes e, em algumas situações, obtém-se como resultado o valor médio das três saídas. Quando utilizado o TMR por maioria de votos, caso ao final da execução não houver um resultado com maioria de votos (isto é, as entradas do votador são diferentes), considera-se que o TMR não foi capaz de mascarar a falha.

Figura 1.2: Redundância Tripla como prevista por Von Neumann




Fonte: (LYONS; VANDERKULK, 1962)

Assumindo-se que o votador não está sujeito a falhas, podemos obter o cálculo da confiabilidade do TMR através de uma função, onde $R_M$ é a confiabilidade individual de cada componente. O sistema como um todo apresentará uma falha apenas se mais de um componente apresentar um valor errado em sua saída. Assim, devido ao fato de que a probabilidade de um componente apresentar falhas é mutualmente exclusiva em relação aos demais, chegamos à seguinte formula para a confiabilidade total R do sistema (LYONS; VANDERKULK, 1962):

$$R = {R_M}^3 + 3{R_M}^2(1 - R_M) = 3{R_M}^2 - 2{R_M}^3$$

Com o modelo visto na imagem acima, é fácil notar que o votador, por estar em série com os demais componentes, é um *Single Point of Failure*, ou seja, se o votador apresentar uma falha, todo o sistema é prejudicado. Por isso, é necessário buscar formas de aumentar a confiabilidade do módulo votador, seja através da utilização de componentes mais confiáveis; utilizando redundância de votadores e um comparador; ou através da implementação do votador em software (WEBER, 2003), que será explicado nos capítulos seguintes deste trabalho.

### 1.1.2 TMR Heterogêneo

Para este trabalho, foi adotado o conceito de TMR Heterogêneo, que foi definido pela utilização de diversidade entre os módulos, de software ou hardware, utilizados para implementar o TMR. O principal problema do TMR Homogêneo é a triplicação total de seus componentes, fazendo com que um sistema protegido por TMR ocupe três vezes mais área que a versão desprotegida. Com isso, triplica-se também a potência dissipada pelo sistema. Com a diversidade do TMR Heterogêneo, busca-se reduzir tais custos de área e potência dissipada

através da utilização de processadores menores, mas evitar ao máximo a degradação do desempenho do sistema.

Após implementar o TMR Homogêneo, busca-se obter medidas de dissipação de potência, consumo energético e área ocupada. Realizando um estudo comparativo entre os pontos positivos, ou negativos, de realizar a implementação do TMR Heterogêneo que ocupa menos área em troca de uma queda no nível de proteção. Com isso, é possível determinar uma relação custo x benefício (redução de área e potência com a diminuição da cobertura de falhas) quando se emprega módulos com maior processamento, porém com maior área, através da comparação entre as duas versões

Para fins de comparação, primeiramente foi implementado um TMR clássico, com três instâncias idênticas do processador, para que fossem obtidos resultados conhecidos para comparar as diferentes versões. Com um processador *softcore*, como o ρ-VEX, é possível alterar a sua configuração a fim de implementar o TMR Heterogêneo realizado neste trabalho através de alterações em seu core. Assim, cada processador pode ter um conjunto diferente de unidades funcionais como ULAs, multiplicadores, unidades de memória e unidades de *branch*, bem como número de *issue-slots* diferentes. Desta forma, o TMR Heterogêneo foi desenvolvido utilizando três processadores, um 2-*issue*, um 4-*issue* e um terceiro com microarquitetura 8-*issue*. Através da utilização de processadores de tamanhos diferentes, busca-se atingir uma redução da área ocupada, bem como da potência dissipada, tentando evitar a queda de desempenho do sistema.

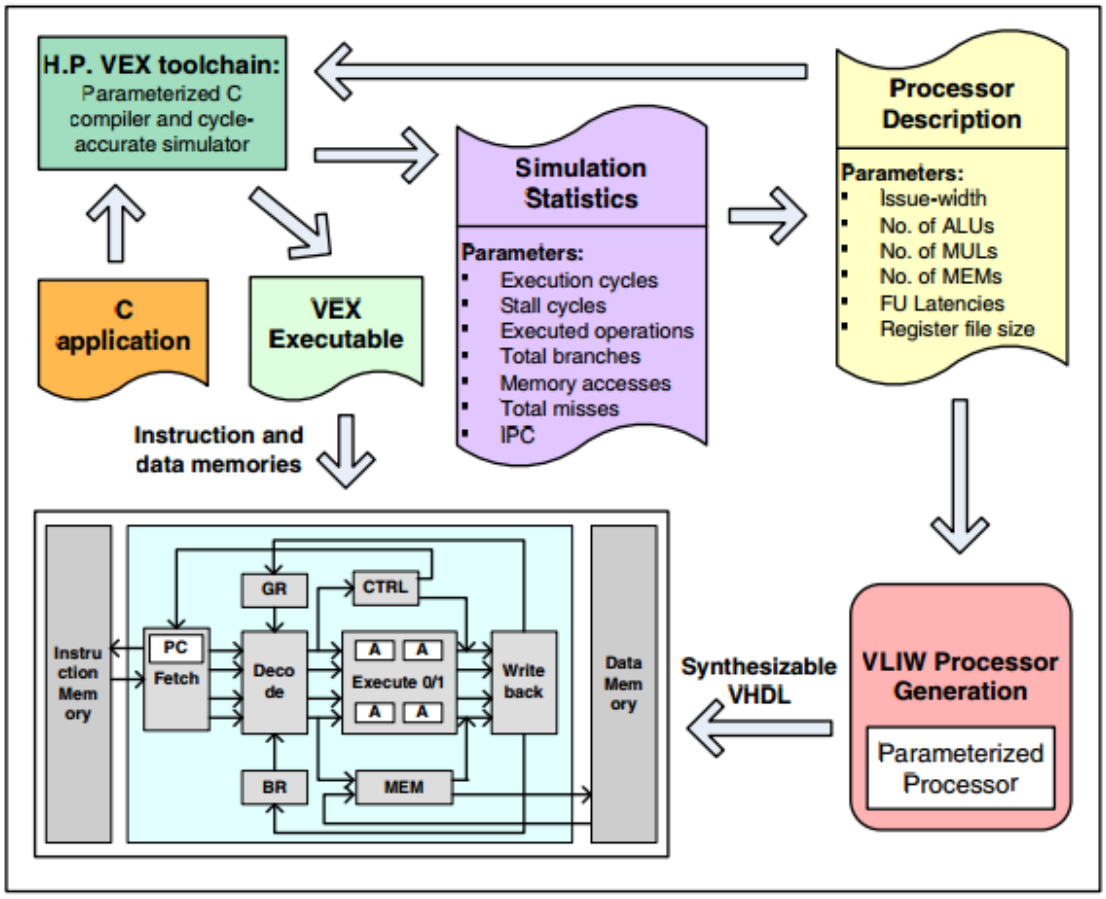
### 1.1.3 DMR

A técnica de DMR (*Dual Modular Redundancy*) possuí o mesmo princípio do TMR: buscar o aumento da confiabilidade através do emprego de redundância. A grande diferença está no fato do DMR utilizar apenas dois módulos redundantes, enquanto o TMR utiliza três. Obviamente, isto reduz a área em relação ao TMR (no caso da redundância ser aplicada em componentes de hardware), porém em caso de falhas, é impossível decidir qual é o resultado correto quando os resultados chegam ao comparador. Por isso o DMR é capaz apenas de sinalizar falhas, fazendo necessária uma nova execução da aplicação para determinar o valor correto ou o envio de um sinal de erro para parar todo o sistema.

## 1.2 Plataforma ρ-VEX

O processador VLIW configurável ρ-VEX, utilizado como processador base do TMR, é um softcore desenvolvido na *Delft Technical University* (*TUDelft*). Trata-se de uma implementação em VHDL de 32 bits, com uma ISA VLIW desenvolvida pela HP, e altamente customizável (ANJAM; WONG, 2013). O compilador da HP, o VEX C *compiler*, baseia-se na técnica de *trace scheduling* para gerar o código VLIW para rodar no ρ-VEX. Tal técnica permite o escalonamento de instruções com um alcance além de *basic blocks* (Blocos formados por instruções limitadas por duas instruções de desvio) (LOWNEY et al., 1993). O VEX C compiler possuí como parâmetros o código C da aplicação, que deve seguir o padrão ISSO/C89 e um arquivo de configuração do processador para qual o código está sendo compilado. Neste arquivo de configuração, chamado *Machine Model*, descreve-se todas as unidades funcionais e quantas elas são, além do *Issue-Width* da microarquitetura. Com esta informação, o compilador

Figura 1.3: Metodologia de Geração de um processador ρ-VEX.



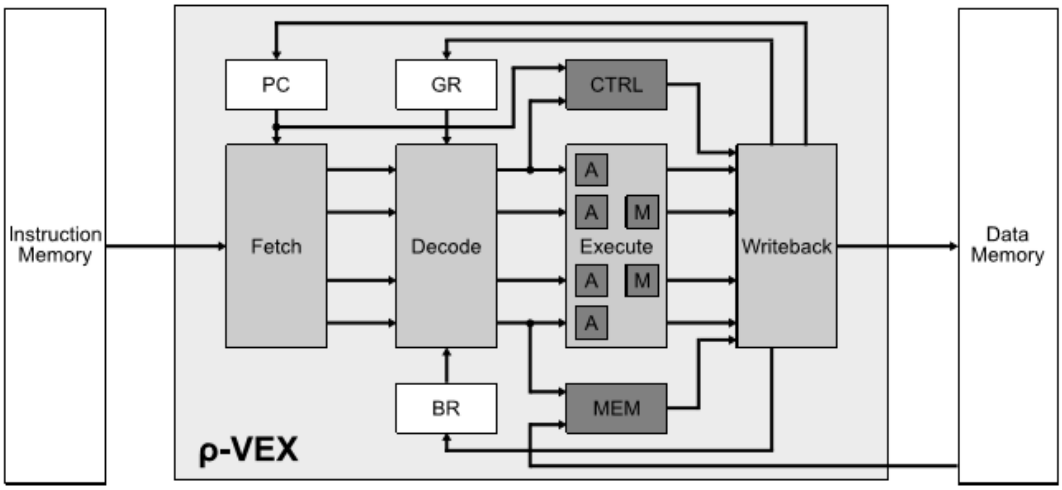

Fonte: (AJAM; WONG, 2013)

consegue realizar o *schedule* das instruções de forma a aumentar o paralelismo e a eficiência do ρ-VEX. A Figura 1.3 mostra o processo completo para a execução de uma aplicação no processador ρ-VEX desde seu código C, até a geração do código *assembly* utilizando o *toolchain* VEX da HP que é distribuído pela própria HP (HP LABS, [s.d.]). A partir do código *assembly*, gera-se o código executável – memória de instruções e memória de dados – para o ρ-VEX através do *Binutils* desenvolvido por pesquisadores da *TUDelft*.

Com o seu *core* em VHDL altamente manipulável, é possível alterar o *issue-width*, quantas e quais unidades funcionais o processador possuirá, instruções suportadas, largura de memória e tamanho do *register file*, tornando possível, assim, realizar as modificações necessárias para desenvolver o TMR Heterogêneo.

Além de basear-se na arquitetura VLIW, o processador possui um pipeline de cinco estágios, sendo eles: busca, decodificação, execução 0, execução 1/memória, e o estágio de *writeback*. O processador também dispões de 64 registradores de 32 bits para propósitos gerais e 8 registradores de *branch* (BR) de 1 bit. A memória é baseada na Arquitetura Harvard, possuindo uma memória de dados e uma de instruções de 32kB cada(WONG; VAN AS; BROWN, 2008). A Figura 1.4 representa a arquitetura geral do processador com ambas as memórias – de instruções e de dados – e os 5 estágios de pipeline.

Figura 1.4: Arquitetura completa do processador ρ-VEX




Fonte: (WONG; VAN AS; BROWN, 2008)

# 2 REFERÊNCIA BIBLIOGRÁFICA

Neste capítulo que segue, será discutido sobre pesquisas e artigos realizados na área e que, de alguma forma, contribuíram para o projeto e desenvolvimento deste trabalho de graduação. Entre os artigos pesquisados, encontram-se diferentes formas de implementação e validação de diversas formas de TMR, tanto homogêneo, como heterogêneos. Foram estudadas pesquisas que utilizavam *softcores*, processadores com arquitetura VLIW e implementações que utilizam FPGAs e ASICs. Assim, foi possível reunir conhecimentos e desafios enfrentados por outros autores, a fim de embasar as decisões de projeto tomadas neste trabalho.

## 2.1 Trabalhos Relacionados

É possível encontrar diversos artigos onde os autores utilizam TMR ou outras técnicas, em hardware ou software, para a proteção de diferentes arquiteturas de processadores utilizando diferentes abordagens. O TMR clássico, cujo a maioria dos trabalhos mencionados a seguir baseia-se, é uma técnica bem difundida na área de Tolerância a Falhas e consiste na realização de uma tarefa (e.g. uma instrução, um conjunto de instruções, ou até uma aplicação inteira) por três módulos de hardware, ou até mesmo de software. Ao final da execução, os três resultados devem passar por um módulo adicional, usualmente chamado de votador, que exibe em sua saída o resultado que possuir a maioria dos votos. Como já citado anteriormente, os trabalhos relacionados descritos a seguir, partem do princípio do TMR clássico e realizam modificações de acordo com a situação-problema e o tipo de processador que almejam proteger.

Em (SCHÖLZEL, 2007), o autor propõe o que foi chamado de Redundância Modular Tripla Reduzida (*Reduced* TMR) como uma forma de autocorreção embutida em um processador VLIW. Esta técnica realiza instruções duplicadas e compara o resultado entre elas. Caso o resultado seja diferente, a instrução é executada uma terceira vez. Com isto, reduz-se o tempo ocioso do hardware, utilizando a cópia do processador, que atuaria apenas em casos de falha, para processar instruções em paralelo. Entretanto, em alguns sistemas críticos, pode ser fundamental a sinalização de uma falha o mais cedo possível. Com a abordagem proposta por este autor, será necessário que a instrução gaste o dobro do tempo para executar, duas em paralelo e a terceira após o término destas, para que seja detectado um erro que poderá se propagar.

A proposta em (SAMUDRALA; RAMOS; KATKOORI, 2004), dispõe de diversos módulos, alguns ativos e rodando a aplicação, enquanto outros permanecem em *standby*. Assim, quando um dos módulos ativos apresenta uma falha permanente, os recursos físicos são remanejados para um dos módulos em *standby*. Com isso, o autor pretende estender o tempo de vida do processador em missões de longa duração. A desvantagem é que isto requer um grande número de módulos de processamento, o que aumenta o custo do projeto. Além disso, é necessário implementar a técnica realizada para remapear os recursos dos processadores que apresentaram falhas para os módulos em *standby*, aumentando o hardware e tornando necessário um tipo de comunicação entre tais módulos. Ainda que a técnica TMR seja capaz de mascarar falhas permanentes, desde que ocorra apenas em um processador, tais decisões vão contra a proposta deste trabalho de buscar uma solução de baixo custo e consumo energético para tratar, como principal enfoque, falhas transientes.

O artigo (TAMBARA et al., 2014) realizou um estudo sobre a técnica DDR (*Design Diversity Redundancy*) que explora o uso de diferentes arquiteturas rodando uma mesma aplicação, de forma assíncrona, e seleciona o resultado correto ao término da execução através de um votador majoritário. Através da implementação utilizando um circuito FPGA independente para cada uma das arquiteturas, previne-se que uma falha cause um curto-circuito entre módulos redundantes, o que dificultaria o mascaramento das falhas pelo TMR. É importante ressaltar que as arquiteturas utilizadas apresentam uma grande diferença em termos de desempenho e, portanto, arquiteturas com maior performance eram limitadas pelas arquiteturas que levam mais tempo para executar a mesma aplicação, causando desperdício de recursos. Outro fator importante é a comparação entre os resultados apenas ao término da execução completa da aplicação. Isso ocorre pois existe uma diferença entre a capacidade de processamento das arquiteturas. Assim, seria necessário tomar medidas para aproveitar melhor o tempo de ociosidade dos processadores com maior poder computacional.

Como exemplo de TMR implementado em software aplicado a um processador VLIW, temos o trabalho do autor (SABENA; REORDA; STERPONE, 2014), que utilizou o conceito de Tolerância a Falhas de Hardware Implementada em Software (SIHFT, do inglês *Software-Implemented Hardware Fault Tolerance*). A técnica consiste em alterar o código *assembly* gerado pelo compilador VLIW de modo a realizar cada computação três vezes, em paralelo, utilizando recursos diferentes. Então, um conjunto de instruções, chamado pelos autores de instruções votadoras, realizam uma votação majoritária entre os três resultados obtidos, porém, isto reduz a eficiência do processador em relação ao processamento das aplicações, pois serão

acrescentadas diversas instruções, em vários pontos do código, aumentando o tempo necessário para obter os resultados necessários. Apesar de ser uma solução com um menor custo de hardware, pois não precisaria modificar o processador no qual a aplicação irá executar, uma camada de testes a ser realizada é adicionada. Além da verificação do código da aplicação, é necessário analisar o código em *assembly* gerado após as alterações para a introdução das instruções votadoras da técnica de TMR.

O trabalho dos autores (PRATT et al., 2006) analisa o fato de que grande maioria das células de memória de um FPGA são responsáveis por armazenar dados sobre a configuração do circuito implementado, e propõe o que foi chamado por eles de TMR Parcial. Esta técnica propõe proteger, contra SEU, elementos responsáveis por dados persistentes contidos no FPGA, como a configuração de suas LUTs (*lookup tables*), por exemplo. Através disto, propõe-se a proteção através de TMR de apenas parte do circuito, diminuindo a área ocupada quando comparado à um TMR completo do processador. Entretanto, tais componentes responsáveis pela persistência variam conforme a aplicação devido à diferença de roteamento de sinais internos, portas de I/O, e principalmente a diferença entre os próprios circuitos como um todo e, por isso, é necessária a análise caso a caso. O autor cita a utilização de uma série de simulações com injeção de falhas e utilização de inteligência artificial para determinar quais partes do circuito eram as consideradas mais críticas. A necessidade de analisar cada novo sistema desenvolvido acaba dificultando o uso deste TMR em larga escala, como em COTS (*Commercial off-the-shelf*), por exemplo.

Como trabalho que foca mais em aumentar a confiabilidade do votador e a re-execução da aplicação quando o votador não é capaz de decidir o resultado, pode ser citado o trabalho de (SHIN; KIM, 1994). Para determinar a forma que a aplicação, cuja falha não pode ser mascarada pelo TMR, acontece através da utilização de duas técnicas, sendo elas a RSWH (*Re-execution of the task on the Same HardWare*) e RHWR (*Replace the faulty Hardware, reload, and Restart*). Com apenas um votador tradicional, por ser um *single-point of failure*, caso ocorra uma falha, perde-se toda a computação realizada pelos processadores. Por isso, os autores do artigo implementam uma solução com dois módulos votadores onde cada um realiza a comparação de dois processadores. O resultado destes dois votadores passa por um comparador, que verifica a validade dos resultados obtidos. Em caso de diferença de resultados na entrada de um dos votadores, é feita uma análise de custo e, baseado nesta análise, a técnica menos custosa entre RSWH e RHWR é executada a fim de obter o resultado correto. Apesar de ser

uma técnica com capacidade de recuperação e uma maior cobertura de falhas, pois cobre falhas inclusive no módulo votador, tem-se um aumento no hardware – devido ao votador extra e o módulo comparador – e também na complexidade para executar as técnicas de decisão de como será realizada a recuperação do sistema.

A ideia desenvolvida por (ICHINOMIYA et al., 2010) desenvolve um TMR utilizando o softcore MicroBlaze e, através da técnica de Reconfiguração Parcial, recuperar-se de falhas permanentes que atingem a memória de configuração do FPGA onde o processador está executando. Neste TMR, triplicaram-se apenas os processadores, *timers* e controladores de interrupções, pois a memória, assim como a deste trabalho de graduação, é protegida através de ECC (*Error Correction Code*) – códigos de correção utilizados para garantir a integridade de dados. Com a ajuda de votadores e módulos detectores, para indicar qual dos três processadores apresentou a falha, realiza-se a reconfiguração da memória onde está armazenada a configuração do processador que apresentou a falha. A reconfiguração ocorre enquanto os dois processadores que apresentaram o resultado correto continuam sua execução e, quando o processador que apresentou falha está pronto para voltar a executar, aciona-se uma interrupção e ocorre a sincronização dos processadores. Os processadores não defeituosos salvam na memória todos os dados que estavam sendo utilizados em sua execução e então os três processadores carregam novamente este último estado salvo em memória. Isto garante que os três processadores possuem o mesmo estado em seus registradores internos. Esta solução cobre um ponto importante do TMR tradicional que é a recuperação de falhas quando um dos três módulos sofre um SEU capaz de tornar o módulo persistentemente defeituoso. Entretanto, os autores do artigo citado não possuem nenhum compromisso em reduzir área e consumo de energia e, além disso, a técnica de Reconfiguração Parcial funciona apenas em FPGAs, enquanto o TMR desenvolvido neste trabalho possui foco maior em falhas transientes e pode ser implementado em FPGAs e ASICs.

Neste trabalho de graduação, foi desenvolvido um projeto similar ao de (TAMBARA et al., 2014), porém, diminui-se o tempo ocioso do sistema através da execução de mais de uma aplicação em processadores de maior desempenho. O artigo de (SAMUDRALA; RAMOS; KATKOORI, 2004) utiliza diversos módulos extras, que permanecem em *stand by*, e substituem módulos que apresentem falhas. Isso, obviamente, aumenta consideravelmente a área ocupada pelo sistema e vai contra uma das propostas deste trabalho de graduação, que é buscar a redução de área do TMR. A implementação feita por (ICHINOMIYA et al., 2010) não visa realizar a redução de área do TMR – visto que foi empregada a total triplicação do

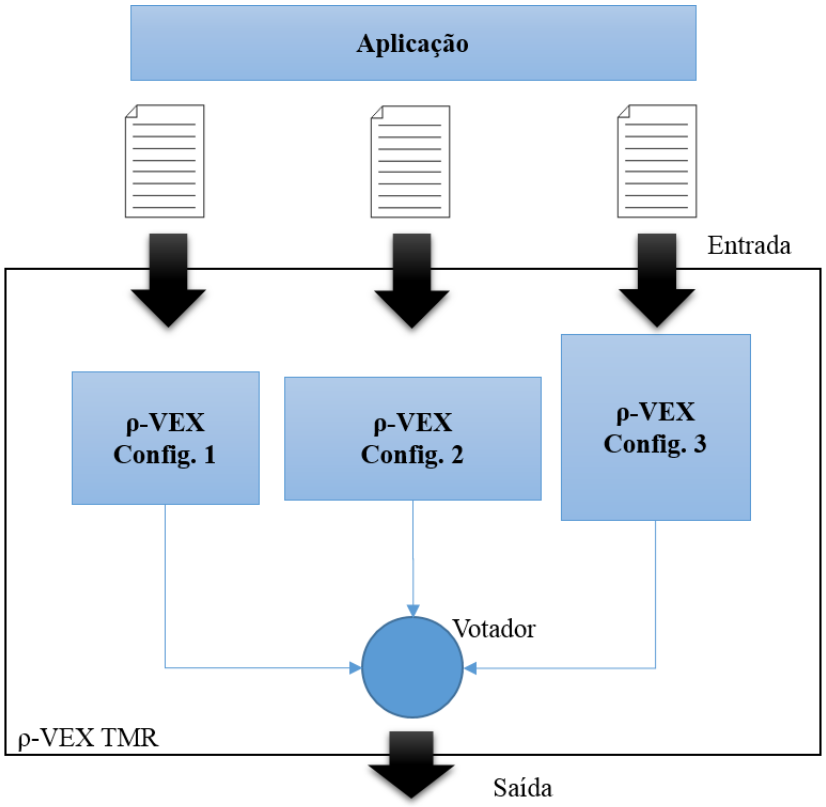
processador – sendo isso um fator importante no desenvolvimento do projeto desse trabalho de graduação.

# 3 ALTERNATIVAS PARA IMPLEMENTAÇÃO DO TMR HETEROGÊNEO

Como já foi citado nos trabalhos relacionados, existem maneiras diferentes de implementar um TMR Heterogêneo: através de software ou hardware; analisando o resultado final da aplicação; o resultado final de um grupo de instruções; ou então comparando cada instrução. Cada uma das três maneiras diferentes é descrita abaixo, citando seus pontos negativos e positivos. Embora fosse interessante realizar as três implementações e comparar entre elas para determinar suas vantagens em relação a energia, área e cobertura de falhas, apenas uma será escolhida para a implementação durante a segunda etapa do Trabalho de Graduação.

A Figura 3.1 representa uma visão geral da arquitetura de um TMR Heterogêneo com três instâncias diferentes do processador ρ-VEX e um votador que coleta o resultado dos processadores ao final da execução e realiza uma comparação, semelhante à solução implementada em (TAMBARA et al., 2014), que utiliza arquiteturas diferentes, ao contrário do projeto desenvolvido neste trabalho, no qual o TMR é desenvolvido utilizando a mesma ISA. Na saída, haverá o resultado com maioria de votos na saída dos processadores ρ-VEX configuração 1, microarquitetura 2-*issue*, ρ-VEX com a configuração 2, o processador 4-*issue*, e ρ-VEX com a configuração 3, com a microarquitetura 8-*issue*. Através de diferentes configurações, será possível realizar uma análise para determinar como atingir o melhor custo-benefício. Como prós, este modelo possui a vantagem de manter o código da aplicação inalterado e dispensa qualquer tipo de sincronização entre os processadores, tornando o hardware mais simples. Em contrapartida, com os processadores executando durante um período muito grande sem nenhuma verificação de seus resultados, aumentam as chances de que outro processador apresente uma falha, tornando o TMR limitado ao processador mais lento. Por isso, executa-se mais de uma aplicação, reduzindo os problemas da diferença de desempenho dos processadores, o que será discutido adiante neste mesmo capítulo. Entretanto, caso ocorra um SEU enquanto o processador não estiver operando, está falha não causará danos à execução do processador. Como as falhas serão injetadas apenas no processador, considera-se que as memórias estão protegidas por ECC (*Error Correction Code*) – codificações utilizadas em palavras de memória para detectar e corrigir bits inconsistentes.

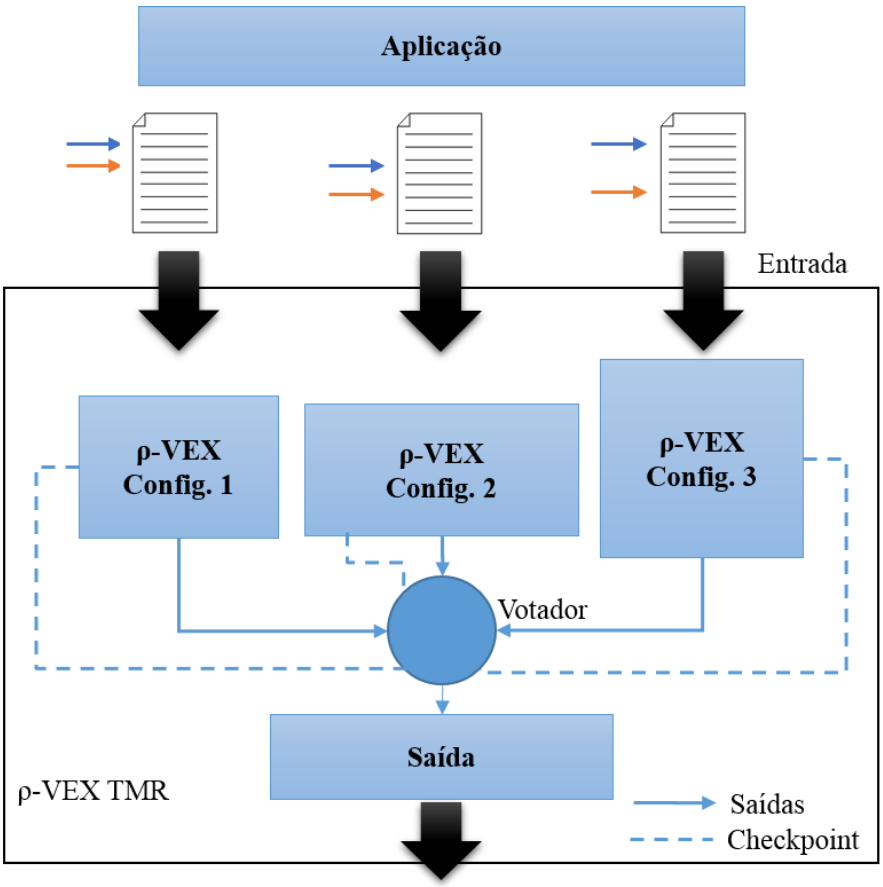
Figura 3.1: Arquitetura de TMR Heterogêneo com comparação ao final da execução.




Fonte: O autor

Outra possível abordagem seria a utilização de checkpoints introduzidos no código *assembly* da aplicação. Ao atingir um checkpoint, o votador recebe uma sinalização de checkpoint atingido, informando valores contidos nas saídas dos processadores estão sincronizados e disponíveis para comparação. Esta abordagem é semelhante à realizada em (SABENA; REORDA; STERPONE, 2014), porém, neste caso, o autor adicionava instruções que atuavam como votador. A Figura 3.2 ilustra como seria a arquitetura desta segunda abordagem, mostrando os checkpoints destacados no código da aplicação. Um dos pontos fortes desse modelo é a ausência de uma sincronização em hardware dos processadores, o que facilita sua implementação, com processadores podendo possuir tamanho de *issue-slots* diferentes, por exemplo. Porém, uma das dificuldades seria avaliar a quantidade de *checkpoints* necessária em cada caso. Checkpoints excessivos podem tornar o código muito extenso, enquanto que a subutilização de checkpoints pode nos levar à uma menor cobertura de falhas como discutido no modelo anterior.

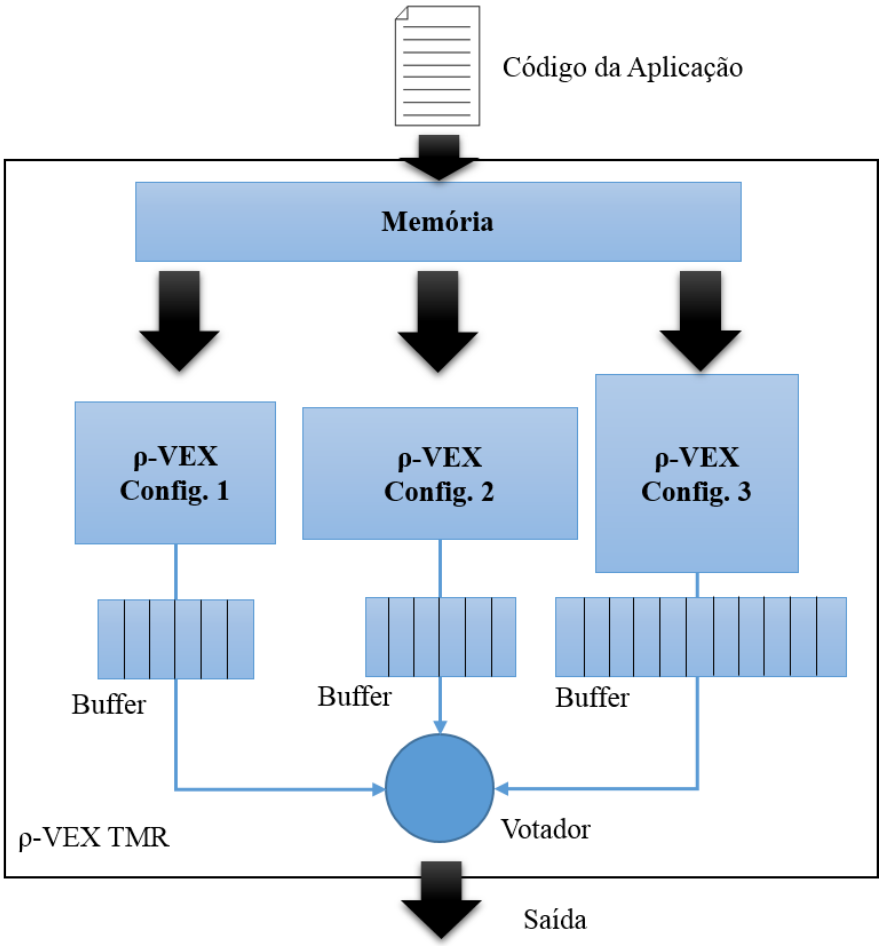
Figura 3.2: TMR Heterogêneo com checkpoints de sincronização.




Fonte: O autor

A terceira abordagem para o problema de sincronização das instruções e realizar a comparação no votador seria através da utilização de *buffers*. A ideia principal é utilizar o compartilhamento de memória e bancos de registradores entre os processadores, enquanto o número de *issues* e as unidades funcionais seriam diferentes para cada processador. Porém, devido ao uso de microarquiteturas com *issue*-slots de tamanhos diferentes, os processadores facilmente perderiam a sincronia, impossibilitando a avaliação das instruções pelo votador. Por isto, adiciona-se um *buffer* na saída de cada processador; assim, o votador lê e compara as instruções diretamente de cada buffer. Também, é necessário analisar e dimensionar corretamente cada buffer a fim de evitar possíveis *overflows*. A Figura 3.3 ilustra a arquitetura geral desta solução. Embora esta solução pareça a melhor em um primeiro momento, pois realiza o TMR no nível das instruções e com baixa latência de sinalização em caso de falhas não mascaradas, bem como a redução do hardware devido ao uso de memória compartilhada e, portanto, menor dissipação de potência, há um grande desafio em como realizar a quebra de

Figura 3.3: TMR Heterogêneo com buffers de sincronização.



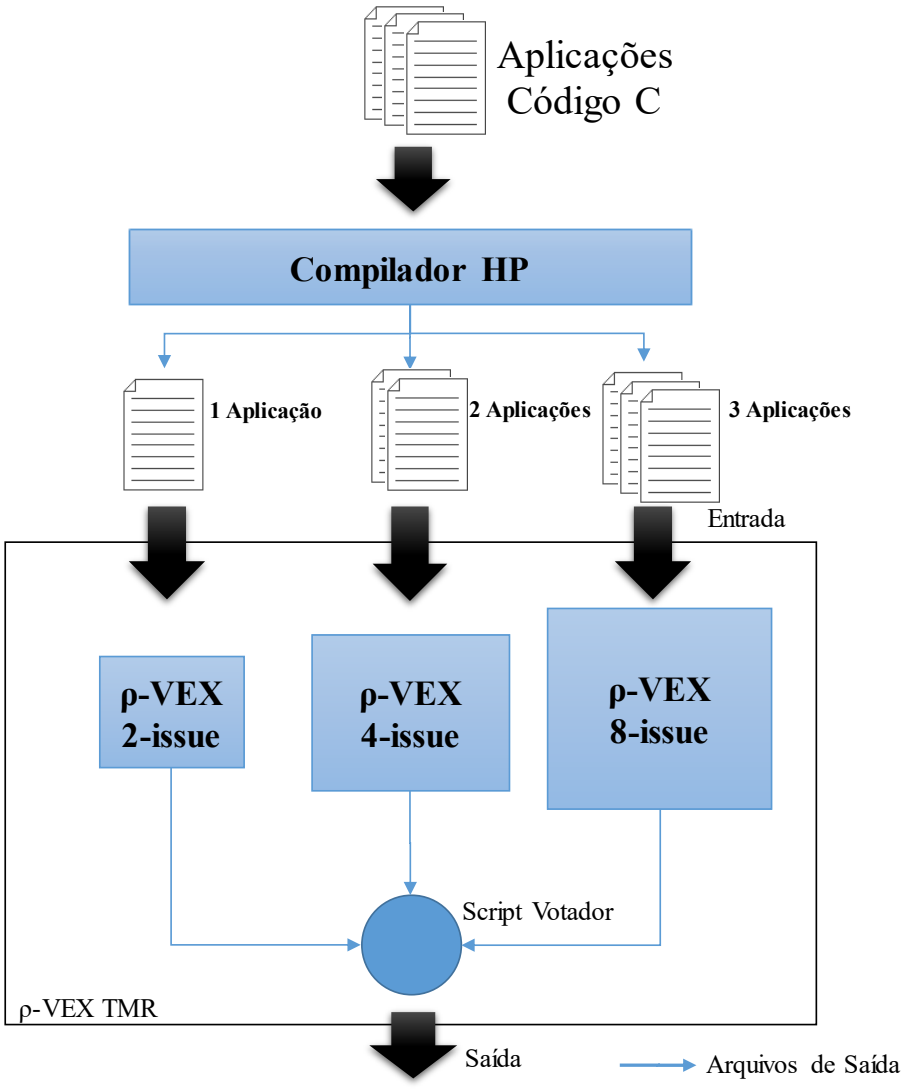

Fonte: O autor

*bundles* de instruções, de modo que este possa ser executado em todas as microarquiteturas do processador, independentemente do número de *issues*. Os autores em (BRANDON; WONG, 2013) realizaram uma análise sobre requisitos para realizar o particionamento de *bundles* através da análise de um grafo gerado a partir das instruções contidas no *bundle* a ser fracionado. Desta forma, como resultado do fracionamento, foi observado um aumento de até 30% de *overhead* quando comparado ao código compilado diretamente para o seu devido *issue-width*. Outro problema nesta implementação seria a introdução de um ponto único de falhas ao utilizar apenas uma memória e um banco de registradores.

## 3.1 Ideia Desenvolvida

Após analisar estas três propostas, optou-se por desenvolver a primeira, com a votação ao final da execução da aplicação. Assim, é possível utilizar arquiteturas com desempenho diferente, não sendo necessário parar completamente a execução de um processador, que possua maior capacidade de processamento, para aguardar o término do processador de menor vazão de instruções. Outro fator importante é que não será necessário alterar o código da aplicação para adicionar instruções específicas do sistema TMR, evitando assim, a perda de desempenho em relação à execução da aplicação de forma desprotegida. Além disso, não é necessário nenhum tipo de buffer de instruções, como no terceiro modelo, que aumentaria área do sistema,

Figura 3.4 Visão geral da Ideia Desenvolvida




Fonte: O autor

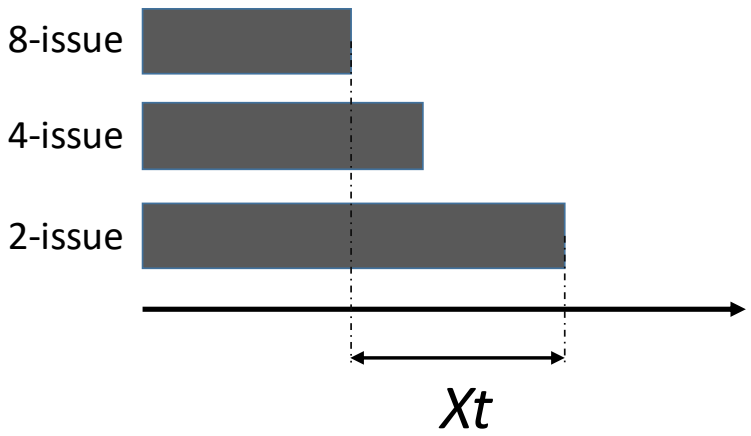
bem como a potência dissipada, sendo a otimização destes dois fatores a principal motivação deste trabalho.

### 3.1.1 Arquitetura Geral

Como no TMR comum, também são utilizadas três unidades de processamento, porém, para o TMR Heterogêneo, como citado anteriormente, utiliza-se três unidades diferentes, como pode ser melhor visualizado na Figura 3.4. O TMR é gerado a partir de três instâncias do *softcore* ρ-VEX onde a primeira possui dois *lanes* de execução, ou seja, 2-issue, o segundo *softcore* é um 4-issue e o terceiro é um ρ-VEX 8-issue. Entretanto, devido à diferente taxa de instruções processadas por cada instância ser diferente, por questões de diferença de poder de processamento, um TMR simples deixaria as instâncias 4-issue e 8-issue ociosas aguardando pelo término da execução da aplicação na instância 2-issue. Por isso, decidiu-se tomar uma abordagem diferente: executar três aplicações na microarquitetura 8-issue, 2 aplicações na arquitetura 4-issue e uma aplicação na arquitetura 2-issue. Com isso, busca-se alcançar um equilíbrio no tempo total de execução das aplicações.

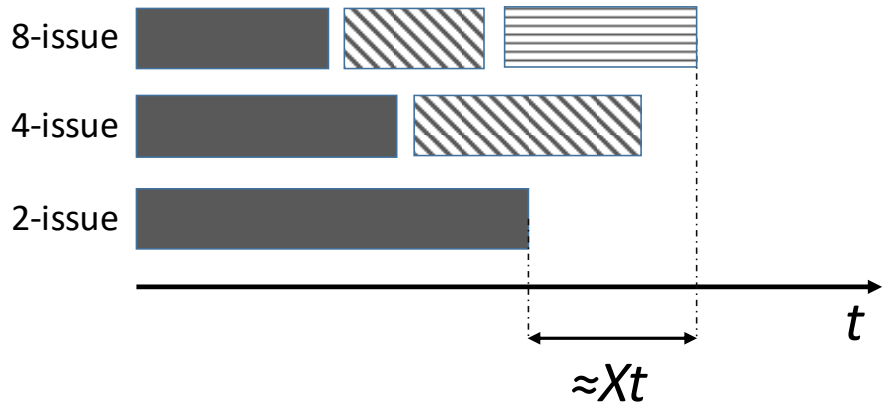
Devido à essa preocupação com a ociosidade de processadores, foi criado um modelo onde uma aplicação roda com um TMR com mascaramento de falhas – no ρ-VEX 2, 4 e 8-issue –, uma aplicação roda um DMR com sinalização de falhas – nos ρ-VEX 4 e 8-issue – e uma aplicação roda de maneira desprotegida no ρ-VEX 8-issue. Com isso, mesmo que o tempo para

Figura 3.5: Execução de uma tarefa no decorrer do tempo nas diferentes microarquiteturas




Fonte: O autor

Figura 3.6: Execução de três tarefas no decorrer do tempo nas diferentes microarquiteturas




Fonte: O autor

executar três aplicações no ρ-VEX 8-issue não seja o mesmo para executar uma aplicação no ρ-VEX 2-issue, aproveita-se melhor o tempo pois ao invés de executar uma aplicação e aguardar um certo período *Xt* de forma ociosa, executa-se três aplicações, para só então ocorrer um período de tempo de aproximadamente *Xt* ocioso novamente. Através das Figura 3.5 é possível visualizar como ocorre normalmente a execução de uma mesma aplicação entre as diferentes microarquiteturas, onde *Xt* representa o tempo que o processador de maior desempenho precisa esperar para que a aplicação seja finalizada no processador que leva mais tempo para terminar sua execução. A Figura 3.6 mostra como uma melhor distribuição e balanceamento de cargas é capaz gera um melhor aproveitamento dos processadores. Os blocos completamente preenchidos em cinza representam uma mesma aplicação sendo executada nos três processadores, sendo assim, um TMR completo; os blocos com listras diagonais representam uma segunda aplicação que executa nos processadores 8-*issue* e 4-*issue*, ou seja, um DMR; e, por fim, o bloco com listras horizontais representa uma terceira aplicação que é executada de maneira desprotegida no processador 8-*issue*. Desta forma, é possível reduzir – ao final de uma longa missão, por exemplo – o tempo total de ociosidade do sistema, tornando mais eficiente a execução de três aplicações mesmo quando o tempo necessário para terminar as tarefas no processador 8-*issue* seja maior que o tempo necessário para terminar a aplicação no processador 2-*issue*.

### 3.1.2 Modificações no Código Base VHDL

Para realizar a execução das aplicações nos três ρ-VEX de forma paralela, o arquivo onde o sistema é instanciado, que inicialmente instanciava apenas um sistema do ρ-VEX, foi modificado para que fosse criada três instâncias do sistema. Com isso, foram necessárias uma série de modificações para que, em fase de síntese, o compilador reconhecesse qual o tipo de ρ-VEX está sendo instanciado – 2, 4 ou 8-issue – e gerar os demais sinais, vetores, componentes, entidades, etc de forma correta. Isso foi feito através da adição de um valor do tipo *Generic* passado ao *Component* do sistema ρ-VEX.

Outra modificação necessária no código base do ρ-VEX foi transferir o nome dos arquivos de memória para a declaração do componente ρ-VEX. Com isso, é possível passar diferentes nomes de arquivos, uma vez que são gerados arquivos diferentes para cada *issue-widths*. Assim, ao declarar o ρ-VEX 2-issue, por exemplo, atribui-se aos arquivos de memória (de dados e de instruções) o nome referente aos arquivos compilados para tal configuração do ρ-VEX, o mesmo ocorre para os casos 4-*issue* e 8-*issue*. Sendo tal modificação muito importante no momento em que uma aplicação é finalizada, pois torna possível o carregamento das diferentes aplicações. Assim, cada versão do ρ-VEX instancia o número de memórias necessárias – três memórias de dados e instruções por processador, quando o processador precisa executar duas aplicações, duas memórias de instruções e duas de dados quando o processador executa duas aplicações, e um módulo de memória de dados e um módulo de memória de instruções quando o processador executa apenas uma aplicação.

O controle de execução das aplicações também sofreu alterações. Como anteriormente apenas uma aplicação era executada em apenas um core, o contador de ciclos e o processo responsável por supervisionar o término da simulação era mais simples. Com dois *cores* a mais e o aumento do número de aplicações durante a simulação, o processo de controle utiliza contadores para cada *core* e sinalizar, quando um dos processadores finaliza as aplicações que lhe são atribuídas. Assim, com essa modificação, garante-se que a simulação será encerrada apenas quando os três processadores rodarem todas as suas aplicações. Além disso, o processo responsável por contar os ciclos, que era encerrado quando o único processador finalizava sua execução, também foi alterado para continuar contando o número total de ciclos até o termino da execução de todas as aplicações, garantindo o controle de quantos ciclos cada processador leva para executar o seu batch de tarefas, bem como o número de ciclos necessários para

executar cada uma de suas aplicações. O controle do número de ciclos é importante pois uma falha pode interferir no sistema de controle do processador e fazer com que o processador execute indeterminadamente, sem atingir o término de sua execução. Com o controle do número total de ciclos é possível finalizar a execução quando algum processador extrapolar o tempo esperado para a execução do *benchmark*.

Para que fosse possível gerar os arquivos de memória para realizar a comparação, foram adicionados *breakpoints* nos pontos onde cada aplicação é terminada para cada processador. Com isso, é possível que a simulação determine quando deve pausar a execução do processador e gerar o arquivo contendo o estado final da memória para futura comparação. Vale ressaltar que é importante a diferenciação entre qual das três aplicações e em qual dos três processadores a execução foi finalizada para que possa ser gerado arquivos de memória independentes de cada aplicação e compará-los corretamente no módulo responsável pela votação.

A definição do *issue-width* de cada processador é determinada ao criar a instância do processador. Assim, dentro do módulo ρ-VEX, foi necessária uma adaptação para que os seus componentes internos fossem gerados dinamicamente de acordo com o tamanho do *issue-width*. Por isso, foram feitas alterações no sistema ρ-VEX, bem como nas memórias de dados e instruções para que tais módulos instanciassem seus componentes de acordo com o tamanho *issue-width* durante a síntese, não sendo mais necessário altera-los manualmente.

# 4 METODOLOGIA

Para desenvolver este trabalho, foram utilizadas diversas ferramentas para simulação, síntese e extração de informações sobre o circuito. Para síntese e teste das versões do TMR, foi utilizado a ferramenta Modelsim. O ISE, da Xilinx, também foi necessário para gerar parte dos scripts de injeção, que serão explicados abaixo. Como ferramenta de medida de área e potência, foi utilizado o Cadence® Encounter RTL. O benchmark executado nas três instâncias do ρ-VEX é um produto vetorial, devido à problemas discutidos no capítulo 5.

## 4.1 Configuração das microarquiteturas

Para a realização dos testes e medições, foram utilizadas, como dito anteriormente, duas versões de TMR: Uma Homogênea, onde as três microarquiteturas são 8-issue e uma versão Heterogênea, com microarquiteturas 2-issue, 4-issue e 8-issue. Além das variações no número de sílabas por instrução VLIW, também há modificações significativas nas Unidades Funcionais, aumentando o desempenho nas arquiteturas com maior grau de paralelismo. Abaixo segue a descrição das microarquiteturas utilizadas nas duas versões de TMR desenvolvidas para este trabalho.

### 4.1.1 Microarquitetura TMR Homogêneo

Na microarquitetura do TMR Homogêneo foram utilizadas três instâncias do ρ-VEX 8-issue com duas Unidades de Multiplicação e uma Unidade de Memória. Com isso, foi possível obter um padrão para a comparação entre os dois tipos de TMR. Com as três microarquiteturas com o máximo de paralelismo possível, alcança-se o maior nível de proteção com o preço de aumentar consideravelmente a área ocupada pelo sistema.

### 4.1.2 Microarquitetura TMR Heterogêneo

No TMR heterogêneo, como dito anteriormente, há uma instancia 2-issue do ρ-VEX que, por ser mais simples, possui apenas uma Unidade de Multiplicação. Já a instância 4-issue, segue

exatamente igual à versão encontrada no TMR Homogêneo, com duas Unidades de Multiplicação. A instância 8-issue possui quatro Unidades. Devido à uma limitação do processador ρ-VEX, todas as instâncias possuem apenas uma Unidade de Memória e apenas uma Unidade de *Branch*.

## 4.2 Scripts

Abaixo são descritos dois grupos de scripts, dos quais um é utilizado para injetar falhas no sistema através de *bitflips*; e outro responsável por realizar a votação dos resultados da aplicação. Os scripts injetores de falhas foram fornecidos pelo grupo do Orientador deste trabalho, sendo necessárias a realização de algumas adaptações que serão descritas abaixo. O script votador foi desenvolvido a partir de outro script, também fornecido pelo grupo, que realizava apenas a verificação de uma memória, sendo necessário adicionar as votações do TMR e do DMR para a realização deste trabalho.

### 4.2.1 Injetor de Falhas

O script injetor de falhas é o responsável por gerar anomalias na execução das aplicações através da inversão do valor de um bit. Antes de cada simulação, o script injetor seleciona um instante de tempo aleatório e um bit, também aleatório, de uma lista de sinais contidos nas instâncias do ρ-VEX. Após escolher o instante de tempo e o bit a ser modificado, a simulação é iniciada. Ao atingir o instante de tempo selecionado, a simulação é pausada e o bit selecionado tem o seu valor invertido – se antes o valor deste bit era 1, passa a ter o valor 0 e vice-versa – e, após isto, a execução da aplicação segue normalmente. Desta forma, é possível simular a ocorrência de falhas transientes, como as geradas por radiação, e testar como o sistema se comporta na presença destas mudanças nos valores dos bits através da comparação – da memória de dados e número de ciclos totais das aplicações – após a execução na ausência de falhas e a execução com injeção de falhas. É importante ressaltar que o instante de tempo selecionado e o bit a ser invertido são gerados randomicamente antes de cada simulação do sistema executando as três aplicações, garantindo que cada injeção de falha seja completamente independente das demais. Outro fator importante e que também deve ser ressaltado, é que as falhas são injetadas em qualquer um dos *piplanes* de qualquer um dos processadores podendo,

assim, atingir um processador durante a execução de uma aplicação protegida por TMR, ou uma aplicação protegida por DMR ou a aplicação que executa de maneira desprotegida. Por esse motivo, considera-se que as memórias estão protegidas com ECC pois não são afetadas pelas falhas injetadas. As listas de sinais, necessárias para a utilização do injetor, são obtidas de arquivos Verilog intermediários gerados através da síntese dos *pipelanes* (*issue-slots*) utilizando o ISE *Design Suite* da Xilinx. Durante a simulação, cada instância do ρ-VEX possui um processo que monitora se houve o término da aplicação. Quando a aplicação é finalizada, é gerado um arquivo contendo todo o conteúdo presente na memória de dados do processador e, quando necessário, carrega-se as próximas memórias de dados e instruções que devem ser executadas.

### 4.2.2 Votador

O módulo votador, como descrito anteriormente, é responsável por determinar se a aplicação executou de maneira correta ou se houve uma falha. Para determinar se houve alguma falha, ao término de cada execução, o votador realiza a comparação das três memórias da aplicação 1, que rodou nos três processadores, e assume que a execução foi correta quando duas ou mais memórias estão com os valores iguais. O votador então compara o resultado das duas aplicações que rodaram nas instâncias do ρ-VEX 4-issue e 8-issue e sinaliza caso elas estejam erradas. Por fim, o votador analisa a aplicação que rodou desprotegida no ρ-VEX 8-issue e, para fins de cálculo da cobertura de falhas, indica se houve uma falha nesta aplicação. É importante relembrar que, para evitar ociosidade de processadores, cada processador executa suas aplicações de maneira independente. Ou seja, ao terminar sua primeira aplicação, o ρ - VEX 8-issue, por exemplo, passa a executar sua segunda aplicação imediatamente, sem aguardar que a votação ocorra.

# 5 RESULTADOS

## 5.1 Comparações entre TMR Heterogêneo e o Homogêneo

A seguir será realizado o comparativo entre os resultados obtidos. Como padrão, será adotado como parâmetro o sistema do TMR Homogêneo e então será feita a comparação com o TMR Heterogêneo de Área, Potência e Desempenho.

### 5.1.1 Área

Buscar a redução da área ocupada pelo sistema TMR foi um dos principais propósitos desse trabalho e foi realizado através da utilização de microarquiteturas com hardware menor, em troca do nível de proteção atingido. Através do Encounter RTL, da Cadence, e do ISE, da Xilinx, foi possível chegar aos valores de área ocupadas em número de células ASIC, número de LUTs e número de Registradores, em FPGA, como descritos na tabela Tabela 5.1 para cada configuração.

A partir dos dados mostrados na Tabela 5.1, foi possível calcular a área total de cada TMR, Homogêneo e Heterogêneo, através da soma do número de Registradores, LUTs e ASICs referente a cada uma das microarquiteturas que compõem cada TMR, e realizar um comparativo entre eles, tomando o TMR Homogêneo como referência de comparação, como visto na Tabela 5.2.

Tabela 5.1 – Comparação de Área do entre as diferentes microarquiteturas.

| | FPGA | | ASIC |
|---|---|---|---|
| | Registradores | LUTs | Cells |
| 2-Issue | 2652 | 7709 | 17228 |
| 4-Issue | 3047 | 15931 | 28049 |
| 8-Issue | 3922 | 34984 | 65281 |

Fonte: O autor

Tabela 5.2 – Comparação de Área do TMR Heterogêneo e Homogêneo.

| | FPGA | | | ASIC | |
|---|---|---|---|---|---|
| | Registradores | LUTs | Overhead | Cells | Overhead |
| TMR Homogêneo (8-Issue) | 11766 | 104952 | 1 | 195843 | 1 |
| TMR Heterogêneo (2-,4-,8-issue) | 9621 | 58624 | 0,5585 | 110558 | 0,5645 |

Fonte: O autor

### 5.1.2 Potência

Em teoria, diminuindo-se a área de um processador, diminui-se também a potência dissipada, pois haverá menos transistores e, consequentemente, menos consumo de corrente. Assim, através das ferramentas da Cadence® Encounter RTL, pode-se chegar ao comparativo total da potência dissipada por ambos TMR. Primeiramente, foram realizadas as medidas de cada configuração do ρ-VEX. Chegando aos resultados contidos na Tabela 5.3.

Tabela 5.3 – Medidas de Potência de cada configuração do ρ-VEX.

| Configuração | Potência (nW) |
|---|---|
| 2-Issue | 7256991,415 |
| 4-Issue | 13763346,318 |
| 8-Issue | 31373640,353 |

Fonte: O autor

Tabela 5.4 – Medidas de Potência do TMR Homogêneo e Heterogêneo.

| Configuração | Potência (nW) | Proporção |
|---|---|---|
| TMR Homogêneo (8-Issue) | 94120921,059 | 1 |
| TMR Heterogêneo (2-, 4-, 8-Issue) | 52393978,086 | 0,5566 |

Fonte: O autor

Com isso, pode-se calcular a potência total dissipada e, tomando o TMR Homogêneo como parâmetro, calcular qual a taxa de redução de potência atingida, mostrada na Tabela 5.4.

### 5.1.3 Desempenho

Com a redução do tamanho do Hardware através da redução do número de unidades funcionais e também do nível de paralelismo utilizado, é de se esperar que ocorra uma queda de desempenho do sistema. Foi possível observar que, utilizando aplicações que demandam um tempo de execução muito alto, o tempo total de execução das aplicações é determinado pelo processador que executa três aplicações. O benchmark executado para a injeção de falhas, cálculo do produto interno, precisou de 2325 ciclos no ρ-VEX 8-issue, 2328 no 4-issue e 2864 ciclos no 2-issue.

## 5.2 Nível de Proteção

Nesta seção será discutido os resultados e comparativos entre o nível de proteção atingido com o TMR Homogêneo e o TMR Heterogêneo. Para tanto, executou-se o injetor de falhas nos dois TMRs durante um período de tempo e anotou-se os números de falhas injetadas, quantas falhas foram protegidas pelo TMR completo, quantas foram sinalizadas pelo DMR e quantas falhas não foram detectadas, pois incidiram no processador desprotegido. É importante discutir sobre os três tipos de erros detectados pelo injetor. O primeiro tipo é o erro de dados e ocorre quando o algum dado contido na memória do processador após sua execução está diferente do dado que deveria possuir após uma execução sem a incidência de falhas transientes, ou seja, ocorreu um *bitflip* em algum dado no *Datapath* que foi propagado e guardado na memória incorretamente. Outro tipo de erro é o erro de controle, que ocorre quando o *bitflip* altera o fluxo natural de execução do programa – alterando o endereço alvo de um desvio, por exemplo – fazendo com que o processador passe a executar indeterminadamente. Esse tipo de erro é considerado como erro tratado pela existência de um *process* que atua como *Watchdog Timer* – um *timer* que sinaliza sempre que o processador parar de responder, realizando a reinicialização quando necessário. O terceiro tipo de erro é o erro de simulação. Este erro ocorre quando o ModelSim apresenta um comportamento inesperado e seu processo é finalizado abruptamente, sendo estes erros irrelevantes para a análise de resultados.

Tabela 5.5 – Dados obtidos a partir da injeção de falhas no TMR Homogêneo.

| TMR Homogêneo | | | |
|---|---|---|---|
| | App1 (TMR) | App2 (TMR) | App3 (TMR) |
| Total de Falhas Injetadas | 1341 | | |
| Erros de Controle | 3 | | |
| Erros de Simulação | 0 | | |
| Erros de Dados | 0 | | |
| Taxa de Defeitos | 0% | 0% | 0% |

Fonte: O autor

### 5.2.1 Nível de Proteção TMR Homogêneo

Para determinar o nível de proteção do TMR Homogêneo, após a injeção das falhas, foi feita uma análise de quantas falhas não foram mascaradas pelo TMR. Como o injetor afeta apenas um processador por vez, foi observado o comportamento esperado de um TMR Homogêneo onde seus processadores são completamente independentes e todas os erros foram mascarados, assim, com o número de falhas injetadas, foi possível comprovar essa hipótese. A Tabela 5.5 mostra a relação de falhas injetadas, erros de controle, erros de simulação e o número de erros de dados que não foram mascarados pelo TMR. Assim, como o esperado, pode-se observar que, na saída do TMR sob efeitos de falhas transientes, observa-se uma taxa de defeitos de 0%.

### 5.2.2 Nível de Proteção TMR Heterogêneo

Abaixo segue o estudo do nível de proteção alcançado com o TMR Heterogêneo, onde foi injetado um total de 12071 falhas, obtendo-se uma média estável dos dados analisados. Além dos tipos de erros já citados (erro de dados, simulação e controle), o TMR Heterogêneo possui alguns casos particulares. Pelo fato do TMR Heterogêneo ser composto de um TMR, um DMR

Tabela 5.6 – Dados obtidos a partir da injeção de falhas no TMR Heterogêneo.

| TMR Heterogêneo | | | |
|---|---|---|---|
| Nível de Proteção | Aplicação 1 (TMR) | Aplicação 2 (DMR) | Aplicação 3 (Desprotegida) |
| Total de Falhas Injetadas | 12071 | | |
| Erros de Controle | 65 | | |
| Erros de Simulação | 1 | | |
| Erros de Dados | 0 | 472 | 465 |
| Taxa de Defeitos | 0% | 0% | 3,85% |

Fonte: O autor

e uma aplicação desprotegida, foi necessário analisar tais casos. Assim, a Tabela 5.6 mostra a relação de falhas injetadas, falhas mascaradas (Erros de dados que foram mascarados pelo TMR), falhas sinalizadas pelo DMR, ou seja, uma memória do DMR é diferente da outra e essa falha foi sinalizada pelo DMR, o número de falhas apresentadas pela aplicação desprotegida e os demais tipos de erros discutidos anteriormente. Assim, como o é injetada apenas uma falha por vez em apenas um processador, o TMR, como esperado, é capaz de mascarar todas as falhas e, além disso, o DMR também é capaz de sinalizar as falhas, sendo o único caso de falhas propagadas para a saída, aquelas afetam o processador desprotegido do TMR.

### 5.2.3 Comparativo do nível de proteção TMR Homogêneo x TMR Heterogêneo

Para finalizar a análise de falhas, a Tabela 5.7 mostra um comparativo entre a porcentagem de proteção contra falhas foi atingida com cada uma das versões do TMR. Nesta tabela, é possível observar que a execução de ambos os TMRs possuem, como o esperado, a mesma taxa de defeitos. A aplicação executada em DMR também apresenta a sinalização de 100% das falhas que prejudicariam a execução da aplicação.

Tabela 5.7: Comparação do nível de proteção do TMR Homogêneo e Heterogêneo.

| | TMR Heterogêneo | | | TMR Homogêneo |
|---|---|---|---|---|
| Proteção da Aplicação | TMR | DMR | Desprotegido | TMR |
| Total de Falhas Injetadas | 12071 | | | 1341 |
| Erros de Controle | 65 | | | 3 |
| Erros de Simulação | 1 | | | 0 |
| Erros de Dados | 0 | 472 | 465 | 0 |
| Taxa de Defeitos | 0% | 0% | 3,85% | 0% |

Fonte: O autor

## 5.3 Problemas encontrados durante o desenvolvimento

Segue a seguir uma discussão sobre problemas encontrados durante o desenvolvimento do trabalho. Obviamente, ocorrem muitas dificuldades em questão de desenvolvimento do código, porém serão listadas apenas dificuldades mais críticas e que exigiram um maior estudo sobre o caso e tomadas de decisões para que o projeto pudesse ser concluído.

### 5.3.1 Pilha

Como problemas durante o desenvolvimento é necessário citar uma certa dificuldade em relação à pilha dos processadores. Com processadores de diferentes *issue-widths*, o binário gerado pela compilação é diferente para cada versão e, com isso, a pilha se comporta de maneira diferente. Assim, ao comparar as memórias entre as diferentes versões do ρ-VEX, no TMR Heterogêneo, foi possível notar que a única diferença na memória de dados era a região utilizada pela pilha. Para que fosse possível realizar a comparação entre as diferentes versões das microarquiteturas, cada memória após a injeção de falhas foi comparada com uma memória

previamente gerada (chamada de Memória Gold, pois foi gerada sem a injeção de falhas), de acordo com o *issue-width* do processador em questão.

### 5.3.2 Memória RAM

Durante o desenvolvimento do TMR, percebeu-se um considerável aumento no consumo de memória RAM durante a simulação da versão em que há processadores que executam mais de uma aplicação. Isso foi causado pela existência de três módulos de memória para armazenar instruções e três módulos de memória de dados por processador, totalizando nove memórias de instruções e nove memórias de dados, no caso do TMR Homogêneo. Isso foi necessário devido às trocas de aplicações, ou seja, cada aplicação possui a sua memória de dados e sua memória de instruções. Devido a esse problema com o consumo de memória RAM, os módulos de memória do ρ-VEX foram dimensionados de maneira que o tamanho total de cada memória seja apenas o suficiente para conter toda a aplicação, evitando que as memórias fossem subutilizadas - com instruções vazias, por exemplo - reduzindo o consumo total de memória RAM ocupada pelo Modelsim e tornando possível a simulação.

# 6 CONCLUSÕES E TRABALHOS FUTUROS

A seguir, neste capítulo são apresentadas as conclusões obtidas durante a realização deste trabalho de graduação. Também são apresentadas algumas propostas para trabalhos futuros baseados na experiência e ideias obtidas durante o desenvolvimento e que, por algum motivo, não pôde ser desenvolvida nesse projeto.

## 6.2 Conclusões

Conforme foi observado, é possível concluir que se atinge níveis satisfatórios de área e redução de potência, reduzindo ambas a praticamente metade. Porém, a taxa de erros na aplicação desprotegida é extremamente alta para os padrões mais altos de sistemas críticos. Porém, para aplicações em missões espaciais onde, por exemplo, coleta-se imagens constantemente, a perda de alguma dessas imagens, ou algum pixel com a cor levemente alterada não causaria grande impacto frente à redução de dissipação de energia e área atingidas. Sobre o desempenho, é necessário observar que, para os experimentos finais, não foi possível reduzir completamente a ociosidade dos processadores mais lentos, como discutido em 3.1.1, devido ao problema com a memória RAM discutido em 5.3.2. Porém, os resultados obtidos continuam validos pois a simulação tratou cada grupo de aplicações (protegidas por TMR, DMR e desprotegida) de maneira independente. Durante este trabalho também foi possível compreender melhor as diversas etapas de um projeto de hardware, desde o desenvolvimento de uma ideia abstrata, até as simulações e extração de dados que possibilitam determinar pontos fortes e fracos do novo sistema desenvolvido.

## 6.2 Trabalhos Futuros

Para possíveis trabalhos futuros, podem ser realizadas melhorias neste modelo apresentado, bem como a simulação do comportamento deste sistema sob a influência de um ambiente com maior radiação, gerando um número maior de *bitflips*.

Uma das principais propostas seria transformar o módulo votador, que neste trabalho foi realizado em software, em um componente de hardware que possa ser implementado

juntamente no mesmo circuito, ou até mesmo em um circuito a parte, garantindo uma maior autonomia para o sistema.

Outra possibilidade seria a de modificar a forma como as aplicações são carregadas, tornando possível decidir em tempo de execução qual aplicação irá executar. Assim, seria possível implementar uma solução mais dinâmica e com um maior mascaramento de falhas. A ideia seria executar primeiramente a aplicação do TMR no processador 2-*issue*, a aplicação do DMR no 4-*issue* e no 8-*issue*. A segunda aplicação dos processadores 4 e 8-*issue*, então, seria a aplicação protegida com TMR. Assim, caso ocorresse uma diferença de resultado entre a execução das aplicações do DMR, seria possível carregar novamente a aplicação DMR no processador 8-*issue* e então realizar uma nova comparação com este novo resultado. Com isso, mascara-se a falha sinalizada pelo DMR e não haveria a necessidade de descartar toda a computação realizada e nem reexecutar a aplicação aumentando o nível de proteção do sistema. Durante a próxima execução, caso o DMR não apresente diferença entre as duas execuções, o processador 8-issue seguiria com a execução da terceira aplicação normalmente. Esta seria apenas uma das características que poderiam ser exploradas utilizando este TMR Heterogêneo. Sendo possível atingir o nível de proteção do TMR tradicional, porém ocupando muito menos área.

A partir da Tabela 5.1, que mostra a área ocupada por cada microarquitetura, é possível notar que 6 instâncias do processador 2-issue ocupam, aproximadamente, a mesma área do TMR Heterogêneo 2-,4-,8-issue. Como proposta para atingir um maior desempenho, poderia ser realizado um estudo sobre a seguinte abordagem: Utilizar seis instâncias do ρ -VEX 2-*issue* e executar as três aplicações de forma paralela, com o objetivo de obter os mesmos níveis de proteção atingidos neste trabalho, com uma aplicação protegida por TMR, uma aplicação protegida por DMR e uma aplicação desprotegida. Sendo assim, três processadores executariam em paralelo uma aplicação, sendo essa protegida por TMR, dois processadores executando a segunda aplicação, protegida por DMR, e o sexto processador executando a terceira aplicação de maneira desprotegida. Apesar desta abordagem, a princípio, aumentar o desempenho do sistema, um dos problemas dessa abordagem seria a quantidade total de memórias, seis memórias de instruções e seis memórias de dados, o que poderia aumentar consideravelmente o tamanho final do sistema.

Seguindo ainda a ideia do TMR Heterogêneo, seria interessante utilizar o trabalho realizado por (BRANDON; WONG, 2013), que torna possível a execução do mesmo código binário entre as diferentes microarquiteturas, sendo possível a utilização de apenas uma memória e um banco

de registrador, reduzindo a área total do sistema TMR. Porém, como foi observado, isso cria um ponto único de falha que precisaria ser analisado e, caso as simulações mostrassem necessárias, desenvolver uma técnica de proteção da memória e do banco de registradores.

## REFERÊNCIAS

ANJAM, F.; WONG, S. **Configurable fault-tolerance for a configurable VLIW processor**. International Symposium on Applied Reconfigurable Computing. **Anais**...2013

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

BRANDON, A.; WONG, S. **Support for dynamic issue width in VLIW processors using generic binaries**. Proceedings of the Conference on Design, Automation and Test in Europe. **Anais**...2013

HP LABS. **VEX Toolchain**. Disponível em: <http://www.hpl.hp.com/downloads/vex/>.

ICHINOMIYA, Y. et al. **Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration**. Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on. **Anais**...2010

LI, Y.; SKADRON, K. TMR: A Solution for Hardware Security Designs. [s.d.].

LOWNEY, P. G. et al. The multiflow trace scheduling compiler. **The journal of Supercomputing**, v. 7, n. 1–2, p. 51–142, 1993.

LYONS, R. E.; VANDERKULK, W. The use of triple-modular redundancy to improve computer reliability. **IBM Journal of Research and Development**, v. 6, n. 2, p. 200–209, 1962.

PRATT, B. et al. **Improving FPGA design robustness with partial TMR**. 2006 IEEE International Reliability Physics Symposium Proceedings. **Anais**...2006

SABENA, D.; REORDA, M. S.; STERPONE, L. **Soft error effects analysis and mitigation in VLIW safety-critical applications**. Very Large Scale Integration (VLSI-SoC), 2014 22nd International Conference on. **Anais**...2014

SAMUDRALA, P. K.; RAMOS, J.; KATKOORI, S. Selective triple modular redundancy

(STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. **Nuclear Science, IEEE Transactions on**, v. 51, n. 5, p. 2957–2969, 2004.

SCHÖLZEL, M. **Reduced Triple Modular redundancy for built-in self-repair in VLIW-processors**. Signal Processing Algorithms, Architectures, Arrangements and Applications, 2007. **Anais**...2007

SHIN, K. G.; KIM, H. A time redundancy approach to TMR failures using fault-state likelihoods. **IEEE Transactions on Computers**, v. 43, n. 10, p. 1151–1162, 1994.

TAMBARA, L. A. et al. **Decreasing FIT with diverse triple modular redundancy in SRAM-based FPGAs**. Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on. **Anais**...2014

TOTONI, E. et al. **Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs**. Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on. **Anais**...2012

WEBER, T. S. Tolerância a falhas: conceitos e exemplos. **Apostila do Programa de Pós-Graduação--Instituto de Informática-UFRGS. Porto Alegre**, 2003.

WONG, S.; VAN AS, T.; BROWN, G. **ρ-VEX: A reconfigurable and extensible softcore VLIW processor**. ICECE Technology, 2008. FPT 2008. International Conference on. **Anais**...2008