Técnica de confiabilidade em nível de sistema operacional para a arquitetura RISC-V

Nicole Migliorini Magagnin
Universidade do Vale do Itajaí, Brasil
nicole.magagnin@edu.univali.br

Benjamin William Mezger
Universidade do Vale do Itajaí, Brasil
ben@edu.univali.br

Douglas Rossi de Melo
Universidade do Vale do Itajaí, Brasil
drm@univali.br

ABSTRACT

Embedded systems are applications used for specific tasks or as part of a general purpose, having constraints and design metrics. These systems are implemented using microprocessors, with a favorable option being the RISC (Reduced Instruction Set Computer), which has a reduced number of instructions and shorter execution time. A highlight within the RISC architecture is the RISC-V, an open and stable instruction set. Since the increase in reliability is essential for embedded systems operating in critical environments, there is a need for fault tolerance provision in hardware and software. Thus, considering design options for operating systems aimed at embedded systems, the Rust programming language is a safe alternative due to its efficient memory management. Therefore, this work proposes implementing a fault tolerance technique in a pre-existing operating system for the RISC-V architecture, aiming to increase reliability in critical environments. The technique chosen for implementation was the triple modular redundancy applied to the operating system's processes. After the implementation, an error injection was performed through an emulator to validate the provision of reliability. Finally, we evaluated the cost and performance metrics, consolidating the implementation and the reliability improvement with a justifiable increase in costs and performance.

PALAVRAS-CHAVE

Sistemas Operacionais, RISC-V, Rust, Tolerância a Falhas.

1 INTRODUÇÃO

Um sistema embarcado é uma aplicação criada para corresponder a um propósito específico ou parte de algo maior, sendo implementada em microprocessadores, onde atualmente se destacam os de arquitetura RISC (Reduced Instruction Set Computer). Os computadores RISC são posteriores às arquiteturas CISC (Complex Instruction Set Computer), simplificados por possuírem menos instruções e um tempo de execução por instrução reduzido [1].

Os processadores da linha RISC fizeram com que o padrão de desempenho de mercado fosse maximizado e seu custo-benefício auxiliou para que a empresa ARM se tornasse dominante ao fim dos anos 1990 [2]. Diante da arquitetura RISC, teve origem o processador RISC-V, uma ISA (Instruction Set Architecture) RISC desenvolvido pela Universidade de Berkeley e que permite alterações em sua arquitetura. Sua acomodação a todas as tecnologias de implementação, como FPGAs (Field Programmable Gate Array), e a estabilidade do processador são pontos relevantes na arquitetura [3].

Para que se tenha um uso completo e facilitado de um processador embarcado, é recomendável que haja um sistema operacional servindo como intermediário para o uso dos dispositivos oferecidos pelo processador ao seu usuário [1]. Os sistemas embarcados, por sua vez, possuem características próprias, sendo geralmente sistemas de funcionamento único e rodando o mesmo programa repetidamente, enquanto um sistema operacional desktop executa diversas aplicações simultaneamente [4].

Diversos projetos já utilizam processadores RISC-V com aplicações em nível de hardware e software. Uma das aplicações em destaque é o processador HArdened RISC-V (HARV) [5], o primeiro processador RISC-V projetado para uso em aplicações espaciais. Enquanto isso, em nível de software, foi proposto por [6] um microkernel para execução em processadores RISC-V, implementado em linguagem C, para que fossem atendidas as necessidades acadêmicas para estudo dos processadores que utilizam o conjunto de instruções RISC-V.

Considerando os sistemas operacionais e as linguagens suportadas pela arquitetura de processadores RISC, a linguagem Rust pode ser considerada uma opção de alto desempenho, produtividade e confiabilidade, o que faz com que esta se torne uma boa opção para a implementação em sistemas embarcados que demandem de requisitos de segurança ou confiabilidade [7]. Dado esse contexto, alguns sistemas operacionais já contam com a linguagem Rust em sua implementação, como o Redox [8], um sistema com foco em confiabilidade e segurança e o Linux [9]. Além disso, o trabalho desenvolvido por [6] foi reprojetado utilizando Rust, de forma a tornar o microkernel básico apto a operar em sistemas críticos.

Apesar de sua maior confiabilidade, a linguagem Rust não se torna imune a faltas. Por decorrência disso, para que um sistema seja considerado confiável, é necessário que este implemente alguma técnica de confiabilidade. Desta forma, este artigo explora técnicas de confiabilidade em nível de software e que integrem soluções computacionais que podem ser utilizadas em ambientes críticos. Foi implementada a técnica de redundância modular tripla (TMR - Triple Modular Redundancy) no escalonador de processos do microkernel em linguagem Rust [6]. Após a implementação, a técnica foi validada a partir da injeção de erros com o software QUick EMUlator (QEMU), juntamente ao depurador GNU Debugger (GDB).

Este artigo possui a seguinte organização. A Seção 1 apresentou a introdução deste trabalho. Na Seção 2 é apresentada a fundamentação teórica, seguida dos trabalhos relacionados. A Seção 3 detalha o desenvolvimento e aplicação da técnica de tolerância a falhas em software. Por fim, na Seção 4 são apresentados e discutidos os resultados da implementação e a Seção 5 contém as conclusões e considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta a base teórica utilizada para o desenvolvimento deste trabalho, consistindo de sistemas operacionais embarcados, arquitetura RISC-V e técnicas de tolerância a falhas, bem como o levantamento dos trabalhos relacionados.

2.1 Sistemas operacionais embarcados

Um sistema operacional tem o papel de oferecer aos programas do usuário um modelo de computador simplificado. Além disso, esse permite gerenciar os componentes de rede, interfaces, periféricos, memória, entradas e saídas de um computador. Sistemas operacionais são geralmente complexos e possuem um ciclo de vida longo, devido a sua dificuldade de implementação.

Os sistemas operacionais podem ser implementados não apenas em computadores de uso pessoal e comercial, mas também em computadores que controlam dispositivos específicos e não aceitam intervenções de instalação de software pelo usuário. Esses sistemas, denominados embarcados, são encontrados em diversos ambientes, podendo também estar presentes em microcontroladores [1].

Um exemplo de sistema operacional de uso embarcado e com capacidade para operar em ambientes críticos é o FreeRTOS, que oferece suporte de tempo real com *timer* de hardware, visando atender métricas de preemptividade e prioridade. Além disso, utiliza o método de fila de tarefas, possui fácil implementação e seu limite de *timers* é correspondente a memória disponível [10].

2.2 RISC-V

O RISC-V é uma arquitetura recente e aberta, que possui menos instruções em relação a arquiteturas anteriores e também um tempo de execução reduzido. Seu objetivo é tornar-se um conjunto de instruções universal e para isso busca atender a todos os tamanhos de processadores. Outro fator importante para o RISC-V é atender a todas as tecnologias de implementação, como FPGAs, ASICs (Application-specific integrated circuit), e chips customizados [3].

Em seu projeto, o RISC-V possui 32 registradores inteiros e 32 registradores de ponto flutuante opcionais, além de possuir uma variação de arquitetura com apenas 16 registradores. O RISC-V possui diversas extensões, além de implementações em uso acadêmico e comercial [11].

2.3 Técnicas de tolerância a falhas

Para que um sistema exposto a ambientes críticos possa ser considerado confiável, são necessárias técnicas de tolerância a falhas. Uma técnica popular na literatura é a redundância modular.

A técnica de redundância modular ou NMR (N-Modular Redundancy) consiste na replicação de um módulo N vezes, utilizando a mesma entrada, a fim de comparar as saídas obtidas em busca de saídas iguais [12]. Dessa forma, ao gerar uma saída, é selecionado o resultado de maior incidência. Caso haja um resultado diferente e não majoritário, isso determina que houve um erro [13].

Para a técnica NMR, é necessária a implementação de um módulo do mesmo componente repetido em N vezes. A saída de maior incidência desses componentes é selecionada como saída verdadeira através de um votador, conforme demonstrado na Figura 1 [14].

2.4 Trabalhos relacionados

O trabalho de [15] realizou a injeção de faltas em registradores não visíveis ao programador, de um sistema *baremetal* em arquitetura RISC-V. Os autores colocam o microcontrolador em comportamentos inesperados aos modelos típicos de tolerância a falhas, para gerar impacto em sua segurança a nível de software. Todas as faltas injetadas possuem as opções de contramedida tanto em hardware,

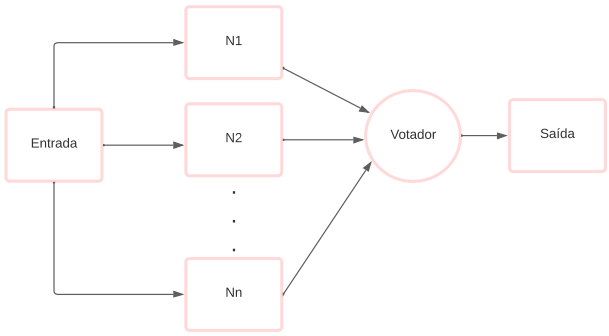

Figura 1: Funcionamento da redundância modular

visando as falhas em seu projeto, quanto em software onde propõem medidas planejadas para o fluxo de dados e de controle, com foco em garantir que os dados sejam manipulados corretamente e sejam evitados saltos de instruções que possam corromper o sistema. Os autores indicam que a melhor maneira de corrigir os erros gerados é a aplicação de redundância dupla ou tripla em nível de instrução.

A pesquisa de [16] aplica uma técnica de tolerância a falhas em um sistema *baremetal* implementado em arquiteturas RISC-V e também em ARM de 32 e 64 bits. A técnica consiste na replicação de operações computacionais e operações de *load* e *store*, deixando um único fluxo de operações de controle e então aplicando proteções TMR (Triple Modular Redundancy) e DWC (Duplicate With Compare). A ferramenta proposta pelos autores é denominada COAST (Compiler Assisted Software fault Tolerance) e ainda pode contabilizar os erros detectados.

O trabalho de [17] trata da injeção de faltas em um sistema operacional FreeRTOS, implementado em um microcontrolador STM32F303VCT6 de arquitetura ARM. As injeções de faltas foram feitas de maneira aleatória e verificou-se a sensibilidade a erros no sistema FreeRTOS. Os erros, conforme sugerido pelos autores, poderiam ser corrigidos duplicando ou triplicando os dados sensíveis, porém o sistema de votação poderia afetar o funcionamento em tempo real do sistema.

Por fim, o trabalho realizado por [18] abordou a implementação de técnicas de tolerância a falhas em nível de sistema operacional no FreeRTOS, com porte para o processador HARV da arquitetura RISC-V. O autor implementou a técnica de TMR aplicada aos processos do sistema, com o objetivo de garantir a execução correta mesmo em caso de falhas transitórias. Além disso, foi implementado um suporte para notificação de falhas detectadas pelo processador para o sistema operacional. A validação da técnica foi realizada pela inversão de bits nas variáveis utilizadas.

Neste contexto, este trabalho utiliza uma arquitetura RISC-V para a implementação da técnica de redundância modular tripla de processos no microkernel em Rust proposto por [6]. Esta abordagem difere dos trabalhos relacionados pelo uso da linguagem Rust para implementação de uma técnica de confiabilidade em nível de software, além do uso de um microkernel desenvolvido em meio acadêmico. A escolha da linguagem Rust traz benefícios a abordagem em questão de confiabilidade, devido ao seu bom gerenciamento de memória, evitando a perda de dados.

3 DESENVOLVIMENTO

Este trabalho propõe a implementação de uma técnica de confiabilidade pré-existente em microkernel implementado em linguagem Rust. Foi selecionada uma estrutura dentro do microkernel apta a ser replicada para a aplicação de redundância tripla.

No diagrama da Figura 2 estão dispostos o kernel e seus processos, ponto alvo da implementação da técnica de confiabilidade. Uma vez que uma técnica de redundância tripla é aplicada, os processos são replicados em três vezes e cada uma de suas saídas é encaminhada a um votador que escolhe a saída de maior incidência como a saída correta.

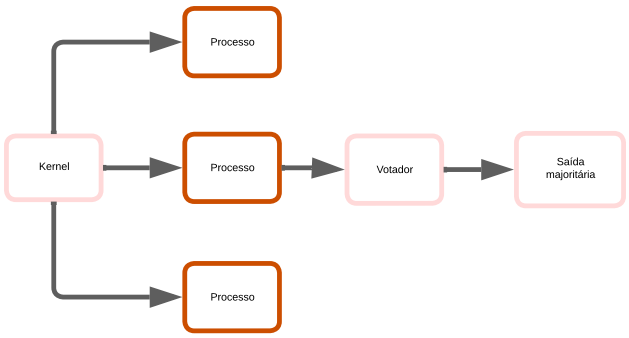

Figura 2: Visão geral da implementação

3.1 Materiais e Métodos

Para a implementação da técnica de confiabilidade foi utilizado o microkernel proposto por [6] em sua versão mais recente, escrita na linguagem Rust. A linguagem Rust também foi utilizada para a implementação da técnica de confiabilidade, tendo como arquitetura-alvo o RISC-V.

O funcionamento da técnica foi validado com a utilização do ambiente de emulação QEMU (Quick Emulator). O ambiente de emulação é capaz de virtualizar uma arquitetura RISC-V com todos os componentes necessários para a execução do microkernel. Para essa validação, uma injeção de falhas foi realizada diretamente no emulador através do *console* do ambiente de desenvolvimento VScode e um roteiro foi gerado a partir dessa prática.

A TMR (Redundância Modular Tripla) foi a técnica de confiabilidade escolhida. Essa escolha se deve a capacidade de detecção e mitigação de erros dessa técnica. Isso é possível devido a sua implementação seguindo as normas da técnica NMR com um número ímpar de clones e garantindo a presença de um votador. Sendo três componentes iguais, o votador sempre irá escolher a saída mais incidente, dessa forma, em caso de erro em uma das saídas, esse será automaticamente mascarado pelo votador, retornando a saída correta.

Para essa aplicação, um mesmo processo é clonado e escalonado para execução em três instâncias. Assim, mediante uma votação, é apresentada a saída de maior incidência. Com a tripla repetição é esperado que pelo menos duas das três saídas estejam corretas, resultando em uma melhoria na confiabilidade do microkernel.

3.2 Arquitetura do microkernel

O microkernel proposto por [6] tem foco em seu tamanho reduzido, possuindo linhas de código limitadas, o que o torna adequado para sistemas embarcados com pouca memória disponível. Esse conta com modo usuário e modo supervisor, sendo o modo usuário separado para a execução de processos. O mesmo mantém apenas o básico em gerenciamento de memória e processos de modo a simplificar a arquitetura.

Para a inicialização de componentes necessários da CPU e configuração do processador, o microkernel utiliza a linguagem assembly. A implementação utiliza o kit personalizado de desenvolvimento SiFive, sendo esse o sistema utilizado para a implementação original. O microkernel é compilado sem o SDK (Software Development Kit) e sem o uso da biblioteca padrão GCC, gerando um tamanho binário menor.

A arquitetura do sistema é dividida entre dois modos, o modo de supervisor e o modo de usuário, ambos representados na Figura 3. O modo de usuário é responsável pela chamadas de sistema privilegiadas e gerenciamento de *drivers*. Enquanto isso, o modo supervisor tem como objetivo o controle de processos relacionados ao usuário, interrupções de hardware, chamadas de sistemas acionadas pelo modo usuário e a manutenção de controle de componentes e estrutura de contexto.

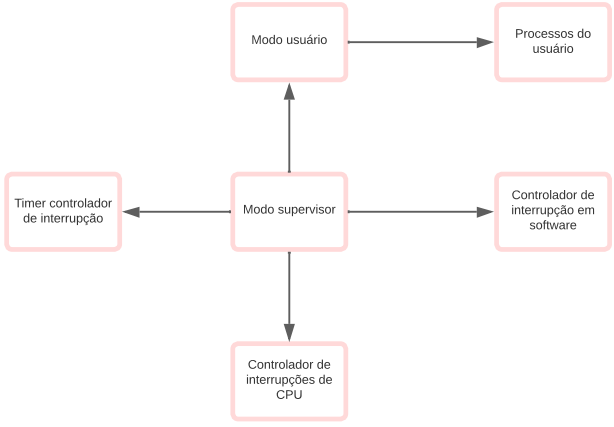

Figura 3: Hierarquia do microkernel

3.3 Processos e escalonamento

Os processos são os serviços a serem executados por um sistema. Neste caso, o microkernel possui um processo principal para administração de processos com o qual realiza as chamadas que fazem ligação entre o início ou uma parada de processos. A arquitetura utilizada pelo microkernel possui a funcionalidade mínima para que o sistema funcione e é dependente da plataforma do processador RISC-V, com possível extensão para outras arquiteturas [6].

Esses processos demandam de um escalonador, que está demonstrado na Figura 4. O escalonador determina quais processos estão prontos e devem ser executados, mediante seu estado. Um processo pode ser considerado pronto quando foi recentemente criado ou é fruto de uma interrupção e está em execução quando estiver de

posse da CPU. Há também o estado de espera, ocorrido quando o processo está esperando o fim ou o resultado de processos anteriores. O processo está terminado quando já foi executado ou falhou.

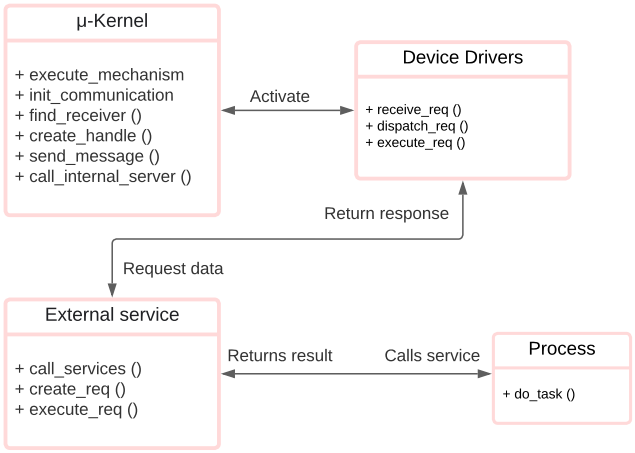

Figura 4: Escalonador de processos

3.4 Sistema operacional confiável

Originalmente, o microkernel em questão foi proposto em linguagem C e cumpriu com os requisitos determinados por [6], possuindo dois modos de execução e permitindo troca de contexto entre eles, interrupções funcionais, um gerenciador de interrupções e um escalonador. Visando uma versão do microkernel com características de segurança e confiabilidade, foi desenvolvida uma versão em linguagem Rust. A escolha da linguagem se deu devido ao seu gerenciamento de memória, reduzindo falhas e perdas de dados, além de ser uma linguagem em constante crescimento no mercado.

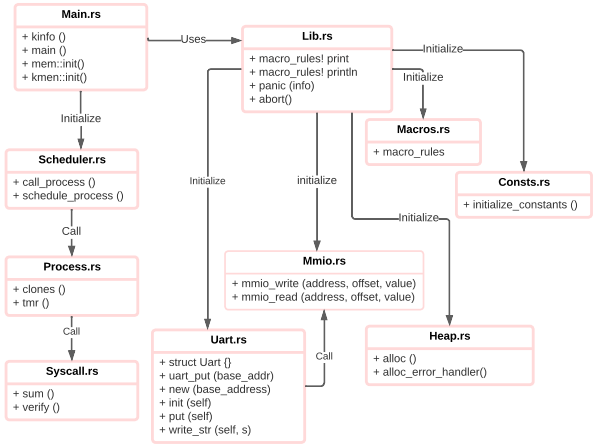

Figura 5: Diagrama de classes do sistema

Uma vez que as técnicas de confiabilidade são implementadas no próprio sistema, a tolerância a falhas é proposta em nível de aplicação. Na Figura 5 estão demonstradas as interfaces desse núcleo onde é possível observar a inicialização do microkernel, suas bibliotecas e componentes Rust através da classe `main`, também com vistas ao escalonador que é objeto-alvo deste trabalho.

A Figura 6 demonstra a implementação da técnica de confiabilidade TMR no microkernel. A técnica está projetada para implementação de maneira que o escalonador (*scheduler*) reconheça o processo onde deverá ser aplicada a replicação e chame o processo com sua função de clone ativa. O processo é agendado e tem sua alocação na memória definida de maneira triplicada, alocando cada um dos processos em um espaço de memória diferente. Então, as saídas de processo passam por uma interface de validação para realizar a decisão de qual retorno foi o mais frequente entre as chamadas de processo. Essa aplicação visa reduzir os erros de execução, detectando saídas diferentes da majoritária como falha e aumentando a confiabilidade em nível aplicação no microkernel.

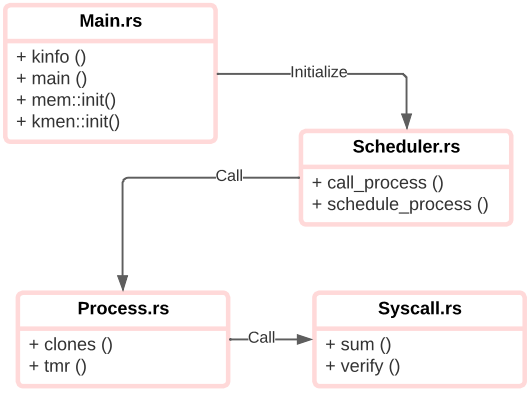

Figura 6: Implementação da técnica TMR

No diagrama de sequência da Figura 7 estão representadas as interfaces de `main`, `scheduler`, `process` e `validation`. A sequência determinada para o funcionamento da aplicação da TMR se dá de forma que o escalonador verifica se o processo está com a técnica ativada. Em caso positivo, esse envia o processo triplamente, o alocando em posições diferentes da memória. Após a programação e execução, o processo retorna sua saída como resposta. Posteriormente, o escalonador envia as respostas para verificar qual delas apareceu com maior frequência e assim determinar a saída por meio de um algoritmo votador.

3.5 Avaliação

A primeira injeção de erros para validação realizada foi na versão do microkernel contendo a soma de escalares. Para isso, foram realizadas as seguintes etapas:

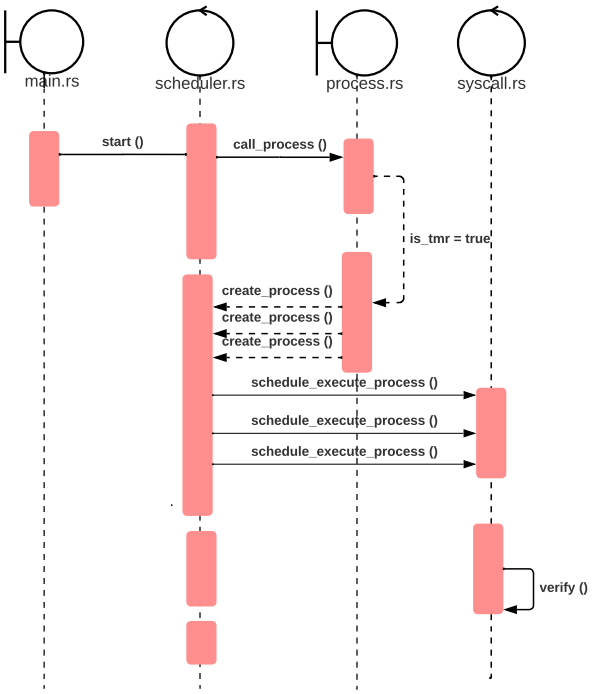

Figura 7: Diagrama de sequência da implementação

(1) Em um terminal é disparado o comando `make qemu`;
(2) Em um segundo terminal, o comando `make gdb` é executado;
(3) No terminal em que está sendo executado o GDB é definido um `breakpoint`, utilizando o comando `b syscall.rs:84`. Esse comando significa que o depurador irá parar quando chegar na linha 84 da classe `syscall`;
(4) Posteriormente, o código é percorrido usando `c` para continuar e `n` para ir até a próxima linha, até o momento adequado a injeção de erros;
(5) Uma vez que o erro será injetado no resultado da soma, é utilizado o comando `p sum` para imprimir o valor dessa variável;
(6) O comando `p &sum` é então utilizado para imprimir o endereço dessa variável;
(7) Obtendo o endereço, é possível utilizar o comando `x/16xw` (endereço) para localizar esse endereço na memória;
(8) Tendo o endereço localizado, basta utilizar o comando `set sum = 123` e o QEMU juntamente ao GDB injetarão o valor definido (123) na posição desejada, a da variável responsável pelo resultado da soma.

Na Figura 8 estão representados os comandos de injeção de erros na função de soma de escalares. Primeiramente, é realizada a impressão do valor da soma e em seguida de seu endereço. Dessa forma, é possível imprimir em console sua localização na memória e visualizar o valor `0x0000004` na primeira posição. Então, esse valor é alterado para 123 através da variável `sum` e pode-se observar a mudança de valor na memória para `0x000007b`, destacada na imagem.

```
(gdb) p sum
$1 = 4
(gdb) p &sum
$2 = (*mut usize) 0x80034048 <strail::syscall::sum>
(gdb) x/16xw 0x80034048
0x80034048 <_ZN6strail7syscall3sum17h0cfb0782e8d490bdE>:      0x00000004      0x00000000
0x80034058 <_ZN6strail7process5total17h557e937d7636b9e1E>:    0x00000000      0x00000000
0x80034068 <_ZN6strail3cpu17KERNEL_TRAP_FRAME17hcfb49177802487b7E>:    0x00000000
0x80034078 <_ZN6strail3cpu17KERNEL_TRAP_FRAME17hcfb49177802487b7E+16>:  0x00000000
(gdb) set sum = 123
(gdb) x/16xw 0x80034048
0x80034048 <_ZN6strail7syscall3sum17h0cfb0782e8d490bdE>:      0x0000007b      0x00000000
0x80034058 <_ZN6strail7process5total17h557e937d7636b9e1E>:    0x00000000      0x00000000
0x80034068 <_ZN6strail3cpu17KERNEL_TRAP_FRAME17hcfb49177802487b7E>:    0x00000000
0x80034078 <_ZN6strail3cpu17KERNEL_TRAP_FRAME17hcfb49177802487b7E+16>:  0x00000000
```

Figura 8: Injeção de erros na soma de escalares

Para a soma de vetores, a injeção de erros foi realizada diretamente na verificação da função TMR, fazendo com que uma das comparações entre uma mesma posição de dois vetores diferentes fosse afetada. Para isso, foram seguidos os mesmos passos da injeção de erros em soma de escalares, apenas alterando a posição do `breakpoint` e a variável para `x`. Na Figura 9 é possível observar a impressão do endereço da variável `x`, seguido pela sua localização na memória. Essa variável também foi alterada para 123 e é possível localizar o valor inicial de `0x0000006` e o valor hexadecimal de `0x000007b` após a alteração, destacados.

```
(gdb) p x
$1 = 6
(gdb) p &x
$2 = (*mut usize) 0x8003f918
(gdb) x/16xw 0x8003f918
0x8003f918:     0x00000006      0x00000000      0x00000004      0x00000000
0x8003f928:     0x00000000      0x00000000      0x00000003      0x00000000
0x8003f938:     0x00000001      0x00000000      0x00000003      0x00000000
0x8003f948:     0x00000001      0x00000000      0x00000000      0x00000000
(gdb) set x = 123
(gdb) x/16xw 0x8003f918
0x8003f918:     0x0000007b      0x00000000      0x00000004      0x00000000
0x8003f928:     0x00000000      0x00000000      0x00000003      0x00000000
0x8003f938:     0x00000001      0x00000000      0x00000003      0x00000000
0x8003f948:     0x00000001      0x00000000      0x00000000      0x00000000
```

Figura 9: Injeção de erros na soma de vetores

4 RESULTADOS

Nesta seção são apresentados os resultados obtidos através da implementação e injeção de erros, além da comparação dos resultados de custo e desempenho.

4.1 Validação

Na Figura 10 é apresentada a saída da execução da soma de escalares com TMR e injeção de erros. Observa-se que a primeira soma tem como resultado 4, enquanto a segunda tem o erro injetado e seu resultado é 123. Na soma seguinte, o resultado já é obtido corretamente. Ainda assim, para a saída, a validação faz com que a técnica de TMR seja executada e o valor de maior incidência se torne a saída, mascarando o erro injetado e retornando a saída como 4.

Um resultado semelhante é visto na soma de vetores após a injeção de erros. Na Figura 11 está disposta a saída da execução com soma de vetores utilizando TMR e a injeção de erros. É possível observar que, ao se deparar com uma comparação errada, todo o vetor torna-se um erro e a palavra `error` é impressa. É válido ressaltar também que a contagem da variável `count_queue` não é incrementada nesse caso, fazendo com que ela não chegue ao tamanho do vetor. Da mesma maneira que na soma de escalares, a função TMR mascarou o erro dessa saída e trouxe como saída mais comum o vetor gerado em duas das três somas do TMR.

```
Init process created at address 0x8000d534
Frame Address: 2160427008
Environment call from M-mode
Environment call from M-mode
Sum: 4
TMR_VALUES_LIST size: 1

Frame Address: 2160455680
Environment call from M-mode
Environment call from M-mode
Sum: 123
TMR_VALUES_LIST size: 2

Frame Address: 2160398336
Environment call from M-mode
Environment call from M-mode
Sum: 4
TMR_VALUES_LIST size: 3

Environment call from M-mode
Correct output: 4
```

Figura 10: Soma de escalares com TMR aplicado

Após a execução dos códigos implementados e a injeção de erros em ambas as situações, foram obtidas as informações de custo em tamanho do arquivo binário e o tempo de execução em cada um dos cenários.

4.2 Custo

Para a validação dos custos foi utilizado o comando `size -A -d` para obter-se um detalhamento de tamanho binário de cada um dos segmentos do sistema em *bytes*, representados na Tabela 1. O segmento `.text` corresponde ao tamanho do código, `.rodata` ao tamanho da área de memória de dados de somente leitura, `.bss` ao detalhamento das variáveis alocadas de forma estática e `.data` as variáveis estáticas, locais e globais que foram inicializadas. Por fim, é apresentado o custo total entre essas partes do sistema, em cada uma de suas versões.

Segmento	Original (bytes)	Escalares com TMR (bytes)	Vetores com TMR (bytes)
`.text`	15456	37952	44108
`.rodata`	5733	12717	13685
`.bss`	4544	4608	4640
`.data`	3424	2616	3680
`Total`	29157	57893	66113

Tabela 1: Custo do binário do microkernel

Correspondente a soma das frações do sistema, tem-se um sobrecusto de aproximadamente 98,53% na versão com soma de escalares e TMR. Enquanto isso, para a versão com soma de vetores e aplicação de TMR, esse número sobe para 126,7% de sobrecusto em relação ao sistema sem tolerância a falhas aplicada.

```
Environment call from M-mode
x is 2, y is 2, count_queue is 1
x is 4, y is 4, count_queue is 2
x is 6, y is 6, count_queue is 3
x is 2, y is 2, count_queue is 4
x is 4, y is 4, count_queue is 5
x is 6, y is 6, count_queue is 6
x is 2, y is 2, count_queue is 7
x is 4, y is 4, count_queue is 8
x is 6, y is 6, count_queue is 9
x is 2, y is 2, count_queue is 10
equal vector
x is 2, y is 2, count_queue is 1
x is 4, y is 4, count_queue is 2
x is 6, y is 6, count_queue is 3
x is 2, y is 2, count_queue is 4
x is 4, y is 4, count_queue is 5
x is 6, y is 6, count_queue is 6
x is 2, y is 2, count_queue is 7
x is 4, y is 4, count_queue is 8
x is 123, y is 6, count_queue is 8
x is 2, y is 2, count_queue is 9
error
x is 2, y is 2, count_queue is 1
x is 4, y is 4, count_queue is 2
x is 6, y is 6, count_queue is 3
x is 2, y is 2, count_queue is 4
x is 4, y is 4, count_queue is 5
x is 6, y is 6, count_queue is 6
x is 2, y is 2, count_queue is 7
x is 4, y is 4, count_queue is 8
x is 6, y is 6, count_queue is 9
x is 2, y is 2, count_queue is 10
equal vector
Most common value: Some([2, 4, 6, 2, 4, 6, 2, 4, 6, 2])
```

Figura 11: Soma de vetores com TMR aplicado

4.3 Desempenho

Na Tabela 2 estão dispostos os resultados da avaliação de desempenho a partir do tempo de execução, gerado a partir da leitura da quantidade de ciclos de *clock* e com base na frequência de 100 MHz.

Dados	Confiabilidade	Tempo de execução (ms)
Escalares		1106,62
Escalares	TMR	3979,77
Vetores		2357,37
Vetores	TMR	6516,42

Tabela 2: Desempenho do microkernel modificado

Tendo em vista os resultados obtidos quanto ao tempo de execução, tem-se um aumento de 3,59x ao utilizar a técnica TMR em comparação com o tempo de execução da soma de escalares sem a aplicação dessa técnica. Enquanto isso, para o uso de vetores, o tempo de execução é 2,76x maior para a execução com o uso de TMR em relação a execução sem a técnica.

4.4 Discussão

Avaliando os resultados obtidos na métrica de custo, é possível observar que há um sobrecusto significativo nas versões onde a técnica de TMR foi implementada. Esse sobrecusto pode ser justificado pelo aumento de confiabilidade gerado, uma vez que a prioridade para o microkernel apresentado é tornar-se seguro para uso em ambientes críticos.

Quanto ao desempenho do microkernel, é importante ressaltar que a sua versão original [6] possui um tempo de execução de cerca de um segundo. Esse tempo se deve ao sobrecusto de inicialização dos processos necessários ao funcionamento, uma vez que ao iniciar o sistema, o mesmo demonstra em *console* seus registradores e memória, além do detalhamento de operações. Com base nessas informações, é possível observar que o aumento de tempo de execução se deu de maneira proporcional ao número de processos e funções a serem executados. Em ambas as versões esse tempo aumentou em cerca de 3x, sendo essa a quantidade de processos a serem executados pela técnica TMR. O tempo necessário para a função de votação da técnica também gerou certo aumento no tempo de execução, porém pouco significativo.

5 CONCLUSÃO

Neste trabalho, buscou-se avaliar uma técnica de confiabilidade em nível de software para implementação em um microkernel em linguagem Rust e voltado à arquitetura RISC-V. Foi selecionada a técnica de redundância modular tripla para implementação. Após o desenvolvimento, foram injetados erros através do software QEMU para a validação, tendo como resultado um aumento de confiabilidade e a mascaração dos erros, com posterior análise de custo e desempenho.

Para trabalhos futuros, pretende-se validar a técnica implementada por meio de uma campanha de injeção de faltas aleatórias. Essa abordagem permitirá uma validação mais abrangente da técnica e seu potencial para mascarar erros. Além disso, pretende-se realizar uma análise do impacto da aplicação da técnica de tolerância a falhas em sistemas de tempo real, levando em consideração o aumento de tempo de execução gerado pela implementação do TMR. Por fim, sugere-se implementar o microkernel em um processador de referência e realizar um estudo comparativo do provimento de confiabilidade em nível de hardware e software.

AGRADECIMENTOS

Este trabalho foi financiado em parte pelo Programa de Bolsas Universitárias de Santa Catarina (UNIEDU) e pela Fundação de Amparo à Pesquisa e Inovação de Santa Catarina (FAPESC-2021TR001907).

REFERÊNCIAS

[1] Andrew S Tanenbaum. *Sistemas operacionais modernos*, volume 4. Pearson, 2015.
[2] David A Patterson and John L Hennessy. *Arquitetura de computadores: uma aboradagem quantitativa: tradução da 5a edição*. Elsevier, 2014.
[3] David Patterson and Andrew Waterman. *Guia pratico RISC-V*. 2017.
[4] Frank Vahid and Tony D Givargis. *Embedded system design: a unified hardware/-software introduction*. John Wiley & Sons, 2001.
[5] Douglas Almeida Santos, Lucas Matana Luza, Luigi Dilillo, Cesar Albenes Zeferino, and Douglas Rossi Melo. Reliability analysis of a fault-tolerant risc-v system-on-chip. *Microelectronics Reliability*, 125:114346, 2021.
[6] Benjamin Mezger, Fabricio Bortoluzzi, Cesar Zeferino, Paulo Valim, and Douglas Rossi Melo. A basic microkernel for the risc-v instruction set architecture. pages 057–063, 04 2021. doi: 10.14210/cotb.v12.p057-063.
[7] Rust, 2022. URL https://www.rust-lang.org/.
[8] Jeremy Soller. The redox operating system, 2022. URL https://doc.redox-os.org/book/ch00-00-introduction.html.
[9] Diolinux. Rust: a linguagem de programação que está conquistando o mercado, dec 2022. URL https://diolinux.com.br/sistemas-operacionais/linux/rust-conquista-mercado.html.
[10] José Morais. Rtos: Software timer no freertos, Mar 2020. URL https://embarcados.com.br/rtos-software-timer-no-freertos.
[11] Rodolfo Labiapari Mansur. Introdução ao processador risc-v - laboratório imobilis - ufop, 2016. URL http://www2.decom.ufop.br/imobilis/o-risc-v/.
[12] Jimmy Fernando Tarrillo Olano. Exploring the use of multiple modular redundancies for masking accumulated faults in sram-based fpgas. 2014.
[13] Kleber Kruger. Programação de microcontroladores utilizando técnicas de tolerância a falhas. 2014.
[14] Wanderson Ricardo de Medeiros. Tolerância a falhas em sistemas embarcados baseados em microcontroladores. B.S. thesis, Universidade Federal do Rio Grande do Norte, 2018.
[15] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a risc-v processor. *Microprocessors and Microsystems*, 71:102862, 2019.
[16] Benjamin James, Heather Quinn, Michael Wirthlin, and Jeffrey Goeders. Applying compiler-automated software fault tolerance to multiple processor platforms. *IEEE Transactions on Nuclear Science*, 67(1):321–327, 2019.
[17] Dario Mamone, Alberto Bosio, Alessandro Savino, Said Hamdioui, and Maurizio Rebaudengo. On the analysis of real-time operating system reliability in embedded systems. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE, 2020.
[18] Benjamin William Mezger, Douglas Almeida dos Santos, Luigi Dilillo, and Douglas Rossi de Melo. Hardening a real-time operating system for a dependable risc-v system-on-chip. In *DFT 2023-36th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, 2023.