

**UNIVERSIDADE DO VALE DO ITAJAÍ
ESCOLA POLITÉCNICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO**

**DETECÇÃO DE ERROS EM SISTEMA OPERACIONAL DE TEMPO
REAL**

por

Marcos Augusto Fehlauer Pereira

Itajaí (SC), Junho de 2025

**UNIVERSIDADE DO VALE DO ITAJAÍ
ESCOLA POLITÉCNICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO**

**DETECÇÃO DE ERROS EM SISTEMA OPERACIONAL DE TEMPO
REAL**

Área de Sistemas Operacionais

por

Marcos Augusto Fehlauer Pereira

Relatório apresentado à Banca Examinadora do Trabalho Técnico-científico de Conclusão de Curso de Engenharia de Computação para análise e aprovação.

Orientador(a): Felipe Viel, MSc.

Itajaí (SC), Junho de 2025

RESUMO

Sistemas embarcados são sistemas especializados tipicamente encontrados como um componente lógico de um dispositivo maior, estes sistemas utilizam-se com frequência um tipo especializado de sistema operacional: Sistemas operacionais de tempo real, que permitem que múltiplas tarefas executem de forma concorrente. Em diversas situações é necessário que esses dispositivos operem em condições adversas (como radiação ionizante e interferência eletromagnética) que alteram seu comportamento esperado e degradam sua qualidade de serviço, para operar dentro de tais condições, técnicas de tolerância à falhas são aplicadas, que visam permitir a operação razoável do sistema mesmo na presença de falhas. Para viabilizar tolerância à falhas é possível utilizar de diversos mecanismos, dentre eles, aqueles que operam em conjunto com o escalonador do sistema operacional de tempo real podem permitir um grau de resiliência maior enquanto visando reduzir a ociosidade dos núcleos da máquina. Este trabalho visa explorar e aplicar técnicas de tolerância e detecção de falhas (Redundância Modular, Reexecução, Heartbeat Signal, CRCs e Asserts) com uma interface voltada ao escalonador do sistema operacional, com o objetivo de fornecer uma análise dos custos e vantagens associados à cada combinação de técnicas.

Palavras-Chave: Sistemas Embarcados. Sistemas Operacionais. Tolerância a Falhas. FreeRTOS. Escalonador.

ABSTRACT

Embedded Systems are specialized systems that are typically found as a logical component of a greater device, these systems are frequently equipped with a special kind of operating system: a Real Time Operating System, that allow for multiple tasks to be executed concurrently. In many situations, it is required that such devices operate in adverse or volatile conditions (e.g. ionizing radiation and electromagnetic interference) that alter its behavior, causing a degradation in its quality of service. Thus, to be able to operate within these contexts, fault tolerance techniques are applied, with the goal of allowing reasonable system operation even within the presence of faults. Many mechanisms may be used to achieve fault tolerance, among them, there are those that operate in conjunction with the real time operating system's scheduler to offer a greater degree of reliability while also decreasing idle time of the machine's cores. This work shall explore and apply techniques of fault tolerance (Modular Redundancy, Re-execution, Heartbeat Signal and Asserts) that have an interface focused on the scheduler's capabilities, with the main objective of providing an analysis over the tradeoffs attached to permutations of those techniques.

Keywords: *Embedded Systems. Operating Systems. Fault Tolerance. FreeRTOS. Scheduler.*

LISTA DE ILUSTRAÇÕES

Figura 1 – Sequência de um Heartbeat Signal	18
Figura 2 – Fluxograma de um Assert	18
Figura 3 – Exemplo de reexecuções	20
Figura 4 – Exemplo de execução com redundância	21
Figura 5 – Hierarquia de latência de acessos	26
Figura 6 – Grafo com 3 processos e uma mensagem	27
Figura 7 – Mesmo grafo, mas tolerante à uma falha transitória	28
Figura 8 – Introdução de transparência em P_2	29
Figura 9 – Análise de resiliência, dividida por categoria	30
Figura 10 – Principais componentes do projeto	33
Figura 11 – Diagrama da STM32F411CEU6 ("BlackPill")	35
Figura 12 – ST-LINK/V2	35
Figura 13 – STMCubeIDE	36
Figura 14 – Injeção lógica em hardware	37
Figura 15 – Layout de uma mensagem	38
Figura 16 – Objeto que implementa a interface de Tarefa	39
Figura 17 – Diagrama de bloco de Redundância modular	40
Figura 18 – Estados de uma reexecução	41
Figura 19 – Sinal Heartbeat	41
Figura 20 – Fluxo do programa de exemplo	43
Figura 21 – Inicializar sessão de depuração no ST-Link	53
Figura 22 – Conectar via VirtualCOM	53
Figura 23 – Posicionar Breakpoint	54
Figura 24 – Causar upset	54
Figura 25 – Resultado no VirtualCOM	55
Figura 26 – Resultado do Filtro Sobel	55
Figura 27 – Estado Inicial	56
Figura 28 – Convolução aplicada e CRCs de cada linha calculados	57
Figura 29 – Inserção de um valor upset	57
Figura 30 – Comparação dos CRCs resulta em máscara de execução	58
Figura 31 – Estado inicial - 3 tarefas	59
Figura 32 – Upset inserido na segunda tarefa	59
Figura 33 – Consenso majoritário seleciona opção que mascara o upset	60
Figura 34 – Imagem com o parâmetro de filtro corrompido	62
Figura 35 – Imagem com Corrupção sutil	63
Figura 36 – Imagem com filtro Sobel interrompido	63

LISTA DE QUADROS

Quadro 1.	Comparação dos trabalhos relacionados	32
Quadro 2.	Requisitos funcionais	37
Quadro 3.	Requisitos não funcionais	38
Quadro 4.	Combinações de técnicas utilizadas	43
Quadro 5.	Análise de riscos	44
Quadro 6.	Declaração da estrutura RawTask	46
Quadro 7.	Declaração da estrutura BasicTask (Implementação omitida)	47
Quadro 8.	Utilizando uma BasicTask para calcular uma soma	48
Quadro 9.	Declaração da estrutura DeadlineWatcher	49
Quadro 10.	Resultados sem injeção	60
Quadro 11.	Resultados com Injeção em variáveis fixas	61
Quadro 12.	Resultados com Injeção no Stack Frame	62
Quadro 13.	Requisitos funcionais	64
Quadro 14.	Validação dos Requisitos não funcionais	64

LISTA DE EQUAÇÕES

Equação 1. Qualidade de Serviço	15
Equação 2. Confiabilidade	16
Equação 3. Disponibilidade	16
Equação 4. Capacidade de Manutenção	17
Equação 5. Polinômio CRC-32C	39

LISTA DE ABREVIATURAS E SIGLAS

COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
CPU	Central Processing Unit
FT	Fault Tolerance
FFT	Fast Fourier Transform
GDB	GNU Debugger
iFFT	Inverse Fast Fourier Transform
IoT	Internet of Things
ITAR	International Traffic in Arms Re-gulations
iSCSI	Internet Small Computer Systems Interface
PCB	Printed Circuit Board
QEMU	Quick EMULATOR
RTOS	Real Time Operating System
RAMS	Reliability, Availability, Mantainability, Safety
SCTP	Stream Control Transmission Protocol
TCC	Trabalho de Conclusão de Curso

SUMÁRIO

1	INTRODUÇÃO	12
1.1	Problematização	12
1.1.1	Solução Proposta	13
1.2	Objetivos	13
1.2.1	Objetivo Geral	13
1.2.2	Objetivos Específicos	13
1.3	Metodologia	14
1.4	Estrutura do Trabalho	14
2	FUNDAMENTAÇÃO TEÓRICA	15
2.1	Definições Principais	15
2.1.1	Qualidade de Serviço	15
2.1.2	Falhas	15
2.1.2.1	Padrões de Ocorrência	15
2.1.3	Dependabilidade	16
2.1.3.1	Confiabilidade	16
2.1.3.2	Disponibilidade	16
2.1.3.3	Capacidade de manutenção	16
2.1.3.4	Segurança	17
2.2	Tolerância à Falhas	17
2.2.1	Mecanismos de Detecção	17
2.2.1.1	CRC (Cyclic Redundancy Check)	17
2.2.1.2	Heartbeat signals	17
2.2.1.3	Asserts	18
2.2.2	Mecanismos de Tratamento	19
2.2.2.1	Re-execução	19
2.2.2.2	Redundância	20
2.2.2.3	Loop Unrolling e Function Inlining	21
2.2.3	Injeção de falhas	22
2.3	Sistemas embarcados	23
2.3.1	Sistemas Operacionais de Tempo-Real	23
2.3.2	Escalonador	24
2.3.3	Concorrência e Assincronia	25
2.3.4	Execução de Tarefas na Presença de Falhas	26
2.4	Trabalhos relacionados	30
2.4.1	Reliability Assessment of Arm Cortex-M Processors under Heavy Ions and Emulated Fault Injection	30

2.4.2	Técnica de confiabilidade em nível de sistema operacional para a arquitetura RISC-V	30
2.4.3	Application-Level Fault Tolerance in Real-Time Embedded System	31
2.4.4	Análise Comparativa dos trabalhos relacionados	31
3	PROJETO	33
3.1	Visão Geral	33
3.2	Premissas	34
3.3	Metodologia	34
3.3.1	Materiais	34
3.3.2	Métodos	36
3.4	Análise de requisitos	37
3.4.1	Interface	38
3.4.2	Algoritmos e Técnicas	39
3.4.2.1	CRC: Cyclic Redundancy Check	39
3.4.2.2	Redundância Modular	39
3.4.2.3	Reexecução	40
3.4.2.4	Sinal Heartbeat / Deadline	41
3.4.2.5	Asserts	41
3.5	Plano de Verificação	42
3.5.1	Programa de Teste	42
3.5.2	Campanha de Injeção de Falhas	43
3.6	Análise de Riscos	44
4	DESENVOLVIMENTO	45
4.1	Estrutura da Implementação	45
4.1.1	Estratégia de Alocação	45
4.1.2	Tarefas	45
4.1.3	Deadlines e Heartbeat	48
4.1.4	CRC32	49
4.1.5	Asserts	50
4.1.6	Dependências Adicionais	50
4.1.7	Compilação do Projeto	50
4.1.8	Programa de Teste	51
5	RESULTADOS	52
5.1	Execução	52
5.1.1	Injeção de falhas	52
5.1.2	Demonstração do processo de Injeção	52
5.1.3	CRC em Detalhes	56

5.1.4	Consenso (TMR) em Detalhes	58
5.2	Dependabilidade e Performance	60
5.3	Impacto de falhas no Output	62
5.4	Verificação	64
5.5	Discussão dos Resultados	65
6	CONSIDERAÇÕES FINAIS	66
	REFERÊNCIAS	67

1 INTRODUÇÃO

A tolerância a falhas (TF) refere-se à capacidade de um sistema computacional de manter sua qualidade do serviço mesmo na presença de estados adversos. Falhas são comumente observadas em sistemas distribuídos, onde os canais de comunicação estão sujeitos a degradação ou inoperabilidade total devido à interferência eletromagnética, falta de energia e eventos climáticos (Krishna; Koren, 2020). Sistemas embarcados enfrentam problemas semelhantes ao operar com canais sujeitos a ruído ou instabilidade, mas também podem ter seus dados em memória ou registradores diretamente corrompidos por causas externas, incluindo: radiação ionizante, flutuações repentinas de temperatura ou voltagem, e colisões físicas (Solouki; Angizi; Violante, 2024). A presença desses fenômenos exige que os dispositivos estejam adequadamente preparados, especialmente em aplicações críticas que operam em ambientes voláteis e enfrentam consequências potencialmente catastróficas caso ocorra um erro.

Tornar um sistema tolerante à falhas é um problema multi facetado, ambas soluções em hardware e software necessitam ser abordadas para garantir a qualidade de serviço desejada. O escalonador é o componente crucial de um sistema operacional para a execução concorrente de diversas tarefas (Galvin; Gagne; Silberschatz, 2018). O processo de detecção das falhas e seu custo de tempo de CPU e uso de memória deve ser considerado pois a reação rápida e correta às falhas requer previamente a detecção e elaboração das rotinas de escalonamento de forma adequada (Solouki; Angizi; Violante, 2024).

Este trabalho visa portanto fazer uma análise do impacto de diferentes técnicas de tolerância à falhas durante o escalonamento de tarefas e seu impacto na performance de um sistema, para que se possa melhor compreender e evidenciar os custos e benefícios ao tornar um sistema mais resiliente.

1.1 PROBLEMATIZAÇÃO

Sistemas embarcados estão presentes em diversos contextos como: exploração espacial, indústria automotiva, tecnologia médica, dispositivos de baixo consumo energético e sistemas distribuídos no contexto IoT. Portanto, entende-se que manter alto grau da qualidade de serviço com o mínimo de degradação de performance e aumento de custo (monetário ou energético), pode prover uma vantagem econômica para fabricantes e provedores assim como um benefício social na maior confiabilidade no caso de aplicações críticas. (Solouki; Angizi; Violante, 2024).

Ademais, ocorreu nos últimos anos uma maior adoção de sistemas COTS (Commercial off the shelf), por poderem ser economicamente mais viáveis e fornecerem uma solução "genérica" para problemas que anteriormente necessitariam de hardware especializado (Nussbaum; Berg, 2020). E mesmo caso pretenda-se utilizar um design especializado para o produto, estes

sistemas são úteis para a fase de prototipação e validação do projeto, dado sua facilidade de acesso e flexibilidade.

Um outro fator que influencia na adoção do uso de COTS para certas aplicações que necessitam de tolerância à falhas são as regulações ITAR (International Traffic in Arms Regulations) imposta pelos Estados Unidos que restringe a exportação de diversos tipos de tecnologia de cunho potencialmente militar. Neste caso, o impedimento da exportação certos tipos tecnologias de PCBs e firmware aumenta mais ainda a necessidade de compradores de outros países adquirirem alternativas comerciais mais comuns que já são complacentes com a regulação (Freeman, 2025).

O custo de utilizar técnicas de tolerância é sensível ao contexto da aplicação e ao nível de tolerância desejado, e no caso dos sistemas COTS, técnicas robustas de resiliência em hardware nem sempre estão disponíveis, sendo necessário delegar esta funcionalidade para a aplicação. É necessário escolher a técnica de tolerância mais adequada de uma multidão de técnicas. Para que a escolha seja informada, é essencial que as trocas em termos de performance e grau de confiabilidade adquiridos sejam compreendidos para minimizar o custo de manter uma qualidade de serviço desejada.

1.1.1 Solução Proposta

Implementar e comparar técnicas de escalonamento com detecção de erros, com o objetivo de esclarecer o impacto de performance em relação ao ganho de dependabilidade do sistema, particularmente no contexto de sistemas com restrição *Hard Real Time*, pois se uma técnica é capaz de satisfazer o critério de tempo real mais rígido, também poderá ser usada em contextos com critério temporal mais relaxado.

1.2 OBJETIVOS

Esta seção formaliza os objetivos do trabalho, conforme descrito a seguir.

1.2.1 Objetivo Geral

Analisar o impacto de técnicas de tolerância à falhas em software num sistema operacional de tempo real.

1.2.2 Objetivos Específicos

1. Selecionar técnicas de tolerância à falhas em software
2. Implementar técnicas escolhidas com uma interface para uso

3. Realizar testes com de injeção de falhas e coletar métricas de performance das técnicas
4. Produzir uma análise comparativa das técnicas, seus custos e eficácia

1.3 METODOLOGIA

O objetivo do trabalho é descritivo e exploratório, as métricas coletadas são de caráter quantitativo e conclusões e observações derivadas do trabalho serão realizadas de maneira indutiva baseadas nas métricas de performance coletadas e comparadas.

Foi realizado uma pesquisa bibliográfica para a fundamentação e escolha das técnicas e dos materiais do trabalho, sendo esta primariamente focada em autores com obras associadas ao tema de tolerância assim como temas adjacentes relevantes como sistemas operacionais e interface hardware-software.

Após isso será realizado uma implementação e testes das técnicas escolhidas para validação, e uma campanha de injeção de falhas será realizada em um microcontrolador para a coleta final das métricas no Capítulo 3.

1.4 ESTRUTURA DO TRABALHO

No Capítulo 2 serão explorados os tópicos centrais que constituem a premissa do trabalho, primariamente conceitos de detecção de falhas e escalonamento nos sistemas operacionais. Sistemas operacionais são um tópico particularmente vasto, portanto sua abordagem será focada apenas nos aspectos mais essenciais para o trabalho. No Capítulo 3 é definido o projeto e o plano de verificação para a implementação e análise das técnicas descritas seguido de sua implementação no Capítulo 4. No Capítulo 5 são expostos os resultados dos testes realizados e no Capítulo 6 é realizado uma recapitulação do que foi abordado no trabalho e quais temas podem ser expandidos por trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 DEFINIÇÕES PRINCIPAIS

Para melhor esclarecer os assuntos abordados, é importante que seja primeiramente definido alguns dos termos centrais para a fundamentação do trabalho.

2.1.1 Qualidade de Serviço

Qualidade de serviço é uma métrica sistêmica que sumariza o quanto bem o sistema provisão suas funcionalidades em um determinado momento. É possível definir e modelar essa métrica de diversas maneiras, mas para os propósitos deste trabalho, será utilizada uma definição simples que resume a funcionalidade geral numa escala de 0 a 1.

A qualidade de serviço Q do sistema pode ser aproximada pela média ponderada de seus serviços $S_0 \dots S_n$ com os pesos de seus fatores de contribuição para a qualidade total $q_0 \dots q_n$ (Izosimov, 2006).

$$Q = \frac{\sum_{i=0}^n S_i q_i}{\sum_{i=0}^n q_i} \quad (1)$$

2.1.2 Falhas

De acordo com a definição de anormalidades de software da IEEE: um erro (*error*) é a diferença entre um valor esperado e o valor obtido. Um defeito (*fault*) é um estado irregular do sistema que pode (ou não) provocar erros que resultem em falhas. Já uma falha (*failure*) é uma incapacidade observável do sistema de cumprir sua função designada, constituindo uma degradação total ou parcial de sua qualidade de serviço (IEEE, 2010).

Neste trabalho, o termo "falha" será utilizado de forma mais geral, representando um estado ou evento no sistema que cause uma degradação na sua qualidade de serviço.

2.1.2.1 Padrões de Ocorrência

Falhas podem ser classificadas em 3 grupos principais quanto ao seu padrão de ocorrência (Krishna; Koren, 2020).

- Falhas Transientes: Ocorrem aleatoriamente e possuem um impacto temporário.
- Falhas Intermitentes: Assim como as transientes possuem duração e impacto temporários, porém ocorrem periodicamente.

- Falhas Permanentes: Causam uma degradação permanente no sistema da qual não pode ser recuperada, potencialmente necessitando de intervenção externa.

2.1.3 Dependabilidade

Será utilizado o termo dependabilidade como uma propriedade que sumariza os atributos: confiabilidade, disponibilidade, capacidade de manutenção e segurança (conhecidos em inglês como critérios RAMS). Os critérios serão definidos na seção seguinte.

A tolerância à falhas impacta positivamente os critérios confiabilidade e disponibilidade, e pode em alguns casos melhorar a capacidade de manutenção, portanto a tolerância à falhas é um aspecto importante para sistemas com dependabilidade.

2.1.3.1 Confiabilidade

Confiabilidade (*Reliability*), é a probabilidade de um sistema executar corretamente no período $[t_0, t]$. Para modelar essa métrica é necessário um modelo estatístico que é particular da aplicação. A confiabilidade R é uma função do tempo t , a taxa de falhas λ e quaisquer sejam os outros parâmetros do modelo (Mamone, 2018).

$$R(t) = f(t, \lambda, \dots) \quad (2)$$

2.1.3.2 Disponibilidade

Disponibilidade (*Availability*) é a razão entre o tempo em que o sistema não consegue prover seu serviço (*downtime*) e o seu tempo total de operação (Mamone, 2018). A disponibilidade A pode ser modelada em termos do tempo disponível t_{up} e do tempo indisponível t_{down} :

$$A = t_{up}/(t_{up} + t_{down}) \quad (3)$$

2.1.3.3 Capacidade de manutenção

Capacidade de manutenção (*Maintainability*) é a probabilidade de um sistema em um estado inválido ser reparado com sucesso antes de um tempo t (Mamone, 2018).

A modelagem deste atributo necessita de conhecimento particular sobre a aplicação e sobre a disponibilidade de equipamentos ou especialistas humanos para a realização do reparo.

Pode ser definida como uma função probabilidade do tempo t , taxa de falhas λ e os outros parâmetros do modelo.

$$M(t) = f(t, \lambda, \dots) \quad (4)$$

2.1.3.4 Segurança

Segurança (*Safety*) é a probabilidade do sistema não causar danos à integridade humana ou à outros patrimônios, independentemente da presença falhas. Por ser um critério muito particular da natureza da aplicação e seu contexto de operação, uma estimativa analítica necessita de um modelo estatístico que não é facilmente sumarizado com apenas uma equação (Mamone, 2018).

2.2 TOLERÂNCIA À FALHAS

2.2.1 Mecanismos de Detecção

Mecanismos de detecção são responsáveis por identificar a presença de uma falha no sistema, um bom mecanismo de detecção deve ser capaz de detectar uma classe grande de falhas sem introduzir uma penalidade grande ao tempo de execução do sistema. Dentre as possíveis escolhas de mecanismos, os subsequentes são utilizados neste trabalho.

2.2.1.1 CRC (Cyclic Redundancy Check)

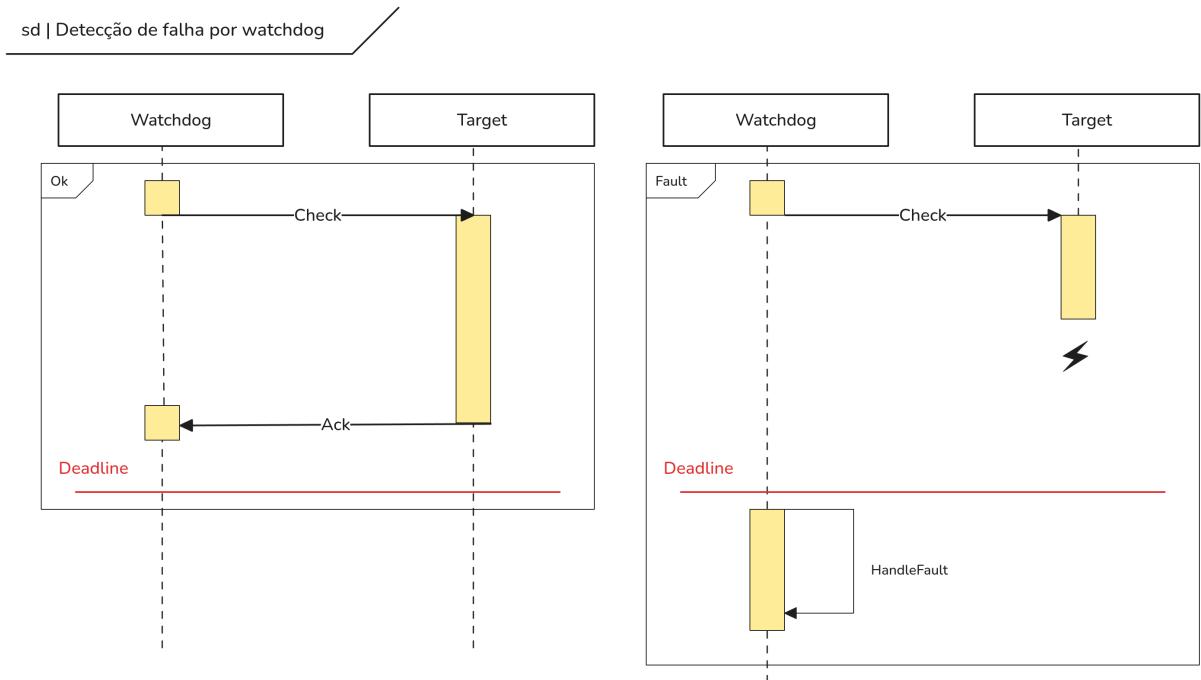
Os CRCs são códigos de detecção de erro comumente utilizados em redes de computador e armazenamento não volátil. Para cada segmento de dado é concatenado um valor de checagem que é calculado com base no resto da divisão com um polinômio gerador pré definido (Krishna; Koren, 2020).

CRCs são comumente utilizados devido à serem simples de implementar, ocuparem pouco espaço adicional no segmento de dados e serem resilientes à erros em rápida sucessão, falhas transientes que alteram uma região de bits próximos.

2.2.1.2 Heartbeat signals

Os *heartbeat signals* (sinais de batimento cardíaco) são sinais periódicos para garantir a atividade de um nó computacional com o recebimento de uma resposta, são também chamados de *watchdog timers* (Solouki; Angizi; Violante, 2024). Uma das aplicações desta técnica em conjunto com reexecução ou replicação é a possibilidade de detectar uma demora excessiva e preventivamente cancelar uma das unidades de execução.

Figura 1 – Sequência de um Heartbeat Signal



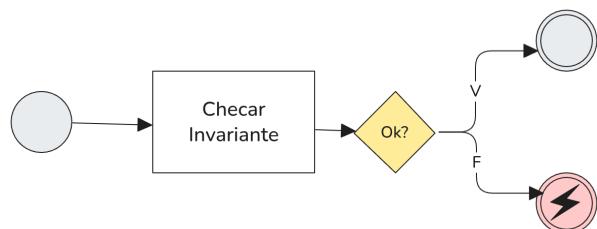
Fonte: Elaborada pelo autor

Na Figura 1, é utilizado a resposta tardia para deduzir a presença de uma falha na tarefa ou no canal de transmissão. Estes sinais também podem ser utilizados no contexto de tempo real para a validação de um prazo ou sub prazo da tarefa (Krishna; Koren, 2020).

2.2.1.3 Asserts

Asserts são mecanismos simples e flexíveis para a detecção de falhas, consistindo na verificação de uma condição que, durante uma execução normal do programa, deve permanecer invariavelmente verdadeira (denominada "invariante"). Caso a invariante seja falsa, detecta-se a presença de uma falha. A Figura 2 ilustra o fluxograma de um assert.

Figura 2 – Fluxograma de um Assert



Fonte: Elaborada pelo autor

Apesar de sua simplicidade, quando usados em conjunto com simulações determinísti-

cas e ferramentas de *fuzzing*, asserts podem detectar erros durante a execução assim como revelar erros de design durante a fase de desenvolvimento (TigerBeetle, 2024) (Holzmann, 2006).

Durante a execução de um sistema tolerante à falhas, asserts servem como uma forma de saber rapidamente que algo errado aconteceu. Porém não são robustos o suficiente para detectar corrupção silenciosa de dados ou pulos inesperados de maneira consistente. Quando asserts são inseridos na entrada ou saída de um procedimento são denominados como pré-condições e pós-condições respectivamente. Alguns compiladores são capazes de automaticamente inserir estas condições para assegurar contratos da interface de um programa. Algumas linguagens de programação, utilizam de uma forma automática de asserts chamados de "contratos", que podem ser usados para garantir certas pós e pré condições, certos compiladores como o da linguagem SPARK fazem uso destas capacidades para realizar verificação formal de um programa (AdaCore, 2025).

2.2.2 Mecanismos de Tratamento

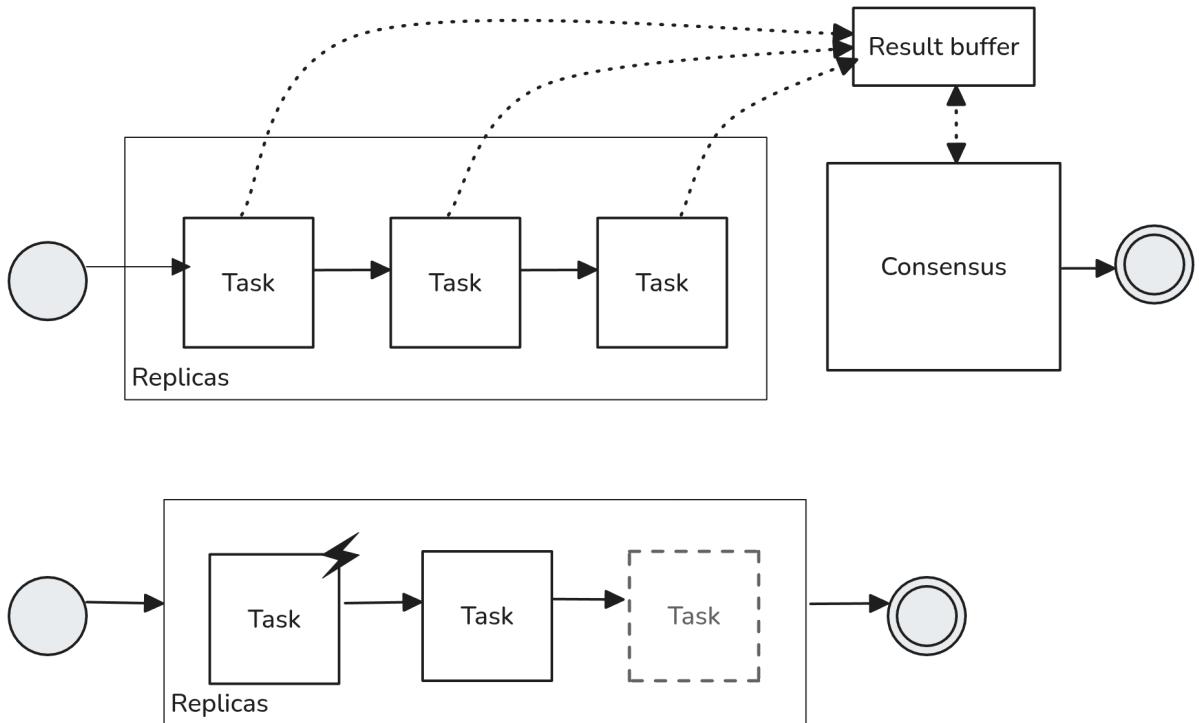
Uma vez que uma falha tenha sido detectada o sistema precisa tratar a falha o mais rápido possível para manter a qualidade de serviço, alguns mecanismos de detecção também fornecem a possibilidade de correção dos dados, como é o caso dos códigos Reed-Solomon, nestes casos, fica à critério da aplicação se a correção deve ser tentada ou outro tratamento deve ser usado.

2.2.2.1 Re-execução

Re-executar uma tarefa é uma outra forma simples de recuperar-se de uma falha, a probabilidade de k falhas intermitentes ocorrem em sequência é menor do que a probabilidade de apenas ocorrer $k - 1$ vezes no intervalo de execução. Ao re-executar, espera-se que a falha não ocorra novamente na N-ésima tentativa (Solouki; Angizi; Violante, 2024) (Izosimov, 2006).

Portanto, é sacrificado um tempo maior de execução caso a falha ocorra, em troca de um tempo menor de execução médio sem necessitar de componentes extras. Em contraste com a técnica de redundância tripla, é possível entender que a redundância tripla ou "tradicional", depende de uma resiliência espacial "É improvável que uma falha ocorra em vários lugares ao mesmo tempo", enquanto a re-execução depende de uma resiliência temporal: É menos provável que múltiplas falhas ocorram repetidamente em N execuções (Krishna; Koren, 2020).

Figura 3 – Exemplo de reexecuções



Fonte: Elaborada pelo autor

Na Figura 3 é possível observar a reexecução de uma tarefa em duas modalidades: Na primeira é realizado um consenso entre os resultados das execuções, e na segunda a tarefa é apenas reexecutada até N vezes (podendo então tolerar até N falhas transientes), encerrando sua execução caso nenhuma falha seja detectada. A segunda modalidade é particularmente útil para a implementação de condições de transparência (Izosimov, 2006) que serão abordadas posteriormente.

2.2.2.2 Redundância

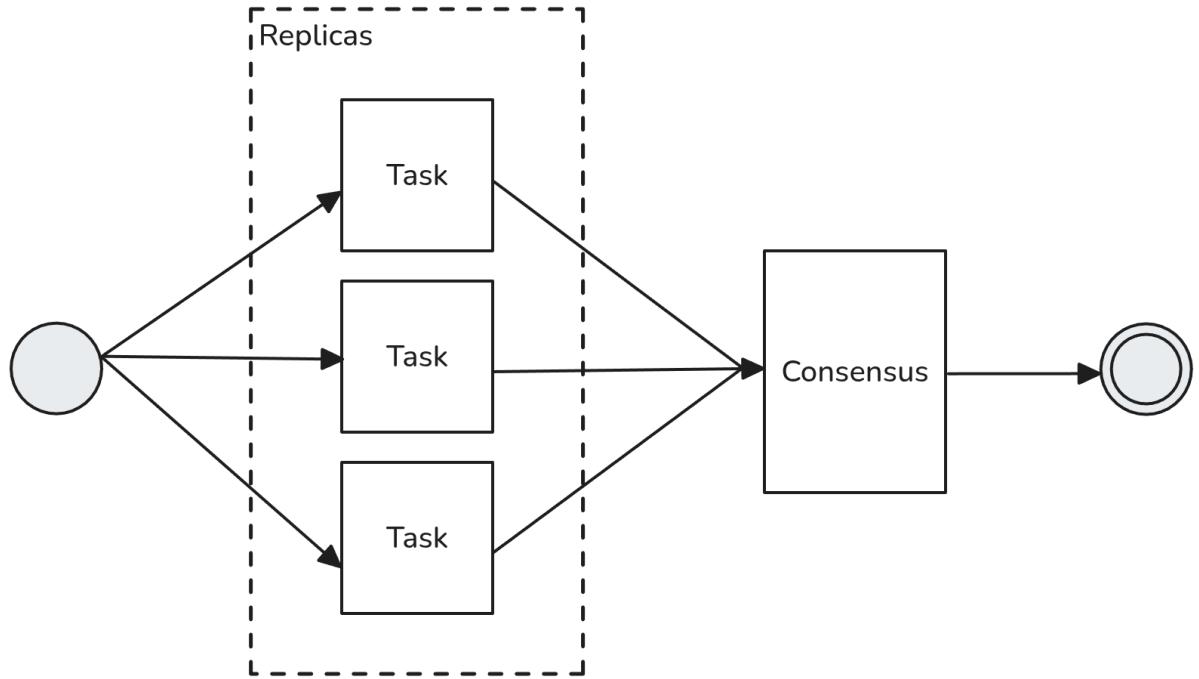
Adicionar redundância ao sistema é uma das formas mais intuitivas e mais antigas de aumentar a tolerância à falhas, a probabilidade de N falhas transientes ocorrendo simultaneamente em um sistema é mais baixa do que a probabilidade de apenas 1 falha (Izosimov, 2006).

Uma técnica de redundância comum é o uso de redundância modular, tipicamente com 3 instâncias replicadas (neste caso chamada de TMR ou "Triple Modular Redundancy"), a Figura 4 ilustra a execução concorrente seguida de um votador.

O uso de TMR para uma tarefa pode consistir na reexecução concorrente das três instâncias, tendo seus resultados decididos por um votador. No contexto de tempo-real é importante que caso alguma das tasks no processo de execução concorrente viole sua deadline, que o votador

ainda escolha um resultado para garantir o critério de tempo-real. O uso de TMR é elegante em sua simplicidade e consegue atingir um bom grau de resiliência, porém com o custo adicional de triplicar o custo em termos de memória e execução (Solouki; Angizi; Violante, 2024), e potencialmente necessitar de hardware mais poderoso para manter a mesma performance esperada.

Figura 4 – Exemplo de execução com redundância



Fonte: Elaborada pelo autor

Sistemas distribuídos, sejam estes embarcados ou não, também podem aproveitar de sua redundância natural para ter maior dependabilidade (Armstrong, 2003). Falhas em um nó podem ser propagadas e no caso de falhas permanentes em um nó, os outros podem suplantar a execução de suas tarefas mantendo a qualidade média de serviço (Armstrong, 2003) (Izosimov, 2006), o uso de sistemas capazes de auto reparo é vital para a existência de telecomunicação em larga escala e computação em nuvem.

2.2.2.3 Loop Unrolling e Function Inlining

Uma otimização comum que compiladores realizam é desenrolar laços de repetição (Loop Unrolling) com a finalidade de reduzir erros no preditor de desvios da CPU, no contexto de tolerância à falhas, é possível utilizar dessa otimização como uma forma de redundância espacial, ao reduzir a possibilidade de pulos dependentes de um valor, torna-se menos provável um salto baseado em uma versão corrompida do mesmo. O desenrolamento pode também ser feito caso exista um limite superior conhecido no laço durante a compilação. (ARM, 2024).

Outra transformação comum é o inlining de funções, onde o corpo de uma função é copiado como se o código tivesse sido diretamente escrito em seu ponto de chamada. Ao reduzir a quantidade de pulos é possível melhorar a coerência do cache de instruções além de permitir outras otimizações durante os passes otimizantes do compilador (Torczon; Cooper, 2007), causando uma melhora na performance. No caso de tolerância à falhas, ao reduzir a quantia de jumps e prover redundância de instruções, o inlining pode também reduzir a chance de um salto inadequado (Armstrong, 2003).

Importante ressaltar que aplicar *function inlining* e *loop unrolling* de forma excessiva pode resultar no oposto do que se deseja no quesito de performance, quando aplicadas de forma agressiva, essas otimizações saturam o cache de instruções, utilizam de mais registradores e ocupam espaço desnecessário no executável (Torczon; Cooper, 2007). Portanto, é importante que estas técnicas não sejam aplicadas de forma arbitrária e que seu uso seja acompanhado de medição para confirmar sua efetividade.

2.2.3 Injeção de falhas

Para adequadamente testar a dependabilidade do sistema, é possível deliberadamente causar falhas com o propósito de catalogar e validar se o sistema atinge as métricas necessárias. Dentre os tipos de teste que podem ser realizados, é possível categorizá-los em quatro grupos principais:

Injeção Física: Envolve utilizar um ambiente físico genuíno para causar as falhas, o principal benefício desta técnica é replicar eventos reais que possam causar falhas, assim como poder injetar falhas em superfícies reais do dispositivo (Mamone, 2018). O principal problema é que esta técnica é particularmente cara e requer auxílio de equipamentos e profissionais especializados, também não é possível injetar um tipo específico de dado para testar um caso específico.

Injeção Lógica em Hardware: Utiliza-se de um dispositivo adicional para injetar as falhas que controla o dispositivo alvo, possui como vantagem ser menos intrusivo e ainda permitir um algo grau de controle e simulação dos fenômenos físicos, desvantagens incluem uma área maior de circuito necessária, implementação de uma unidade extra e criação de canais de comunicação com o dispositivo alvo (Mamone, 2018).

Injeção Lógica em Software: Funções são executadas em software para injetar falhas em outras partes do programa, o método é pouco invasivo, de baixo custo, alta portabilidade e permite um controle muito elevado sobre os pontos de injeção e estilo de falha (Mamone, 2018). Possui a desvantagem de aumentar o tempo médio de execução ao introduzir um custo extra de memória para armazenar o código de injeção, e nem sempre reproduz precisamente fenômenos

físicos.

Injeção Simulada: O dispositivo é executado em um ambiente totalmente simulado, tem como vantagem não ser invasivo, altamente flexível e nem sequer necessitar de uma versão física do dispositivo, porém tipicamente requer software de simulação potencialmente caro assim como uma descrição do chip na forma de alguma linguagem de descrição de hardware, que raramente é disponibilizada (Mamone, 2018).

2.3 SISTEMAS EMBARCADOS

Sistemas embarcados são uma família vasta de sistemas computacionais, algumas das principais características de sistemas embarcados são:

Especificidade: Diferente de um sistema de computação mais generalizado como um computador pessoal ou um servidor, sistemas embarcados são especializados para uma solução de escopo restrito. Exemplos de sistemas embarcados variam de microcontroladores encontrados em carros, televisões e dispositivos IoT até sistemas sofisticados de navegação de aeronaves e navios de grande porte (Hennessy; Patterson, 2020).

Limitação de recursos: Um corolário da natureza especialista destes sistemas, é que recursos alocados para o sistema são definidos previamente. No caso de microcontroladores o poder de processamento e quantidade de memória podem ser restritos para satisfazer uma necessidade de baixo custo de fabricação e menor consumo energético (Hennessy; Patterson, 2020). Importante notar que existem sistemas embarcados com acesso maior à recursos, como certos equipamentos de rede e aceleradores, mas os recursos do sistema continuam estaticamente delimitados para cumprir sua função específica.

Critério Temporal: Sistemas embarcados, por serem parte de um todo maior, devem realizar sua função com o mínimo de interrupção para a funcionalidade geral do contexto externo (Galvin; Gagne; Silberschatz, 2018). A importância do tempo de execução de uma tarefa de um sistema pode ser classificada em duas principais categorias: Soft Real-Time , e Hard Real-Time , a distinção entre estas categorias é explicada na seção seguinte.

2.3.1 Sistemas Operacionais de Tempo-Real

Um sistema operacional é um conjunto de software que permite o gerenciamento e interação com os recursos do hardware através de uma camada de abstração. O componente essencial de um sistema operacional é o kernel, que sempre está executando, A função primordial do kernel é viabilizar a coexistência de diversas tarefas no sistema, as quais demandam acesso às capacidades do hardware, notadamente o tempo de processamento da CPU e o espaço

de memória. De forma simplificada, o kernel pode ser descrito como a "cola" entre as aplicações e os recursos de hardware (Galvin; Gagne; Silberschatz, 2018).

Um sistema operacional de tempo real (RTOS) é um tipo de sistema operacional mais especializado, tipicamente pequeno, que possui como característica central cumprir um critério temporal Real-Time , que é dividido em 2 categorias:

- **Soft Real-Time:** Um sistema que garante essa propriedade precisa sempre garantir que tarefas de maior importância tenham prioridade sobre as de menor importância. Sistemas Soft Real-Time tipicamente operam na escala de milissegundos, isto é, percepção humana (Izosimov, 2006). O atraso de uma tarefa em um sistema Soft Real-Time não é desejável, mas não constitui uma falha. Exemplos: Player de DVD, videogames, kiosks de atendimento automáticos.
- **Hard Real-Time:** Precisa garantir as propriedades de Soft Real-Time , além disso, o atraso de uma tarefa de seu prazo, é inaceitável, para um sistema Hard Real-Time uma resposta com atraso é o mesmo que resposta nenhuma. Cuidado adicional deve ser utilizado ao projetar sistemas Hard Real-Time , pois muitas vezes aparecem em contextos críticos (Bos; Tanenbaum, 2023). Exemplos: Software para sistema de frenagem, sistemas de navegação em aplicações aeroespaciais, software de negociação de alta frequência, fila de mensagens de alta performance

Como sistemas que cumprem o critério Hard Real-Time também cumprem os requisitos Soft Real-Time, os sistemas operacionais de tempo real tipicamente tem sua arquitetura orientada a serem capazes de cumprir o critério Hard Real-Time (Izosimov, 2006).

Diferentemente de sistemas operacionais focados em uso geral como Windows, Linux e OSX, os RTOSes não priorizam fornecer ao usuário uma sensação de fluidez e adaptabilidade. Devido à seus requisitos temporais rígidos, um RTOS é feito com um foco significativo em determinismo, confiabilidade e simplicidade, para garantir que tarefas sejam executadas com um respeito estrito de seus prazos (Galvin; Gagne; Silberschatz, 2018). Exemplos de RTOS disponíveis no mercado incluem: FreeRTOS, VxWorks, Zephyr e LynxOS.

2.3.2 Escalonador

O escalonador é o componente do sistema operacional responsável por gerenciar múltiplas tarefas que desejam executar (Galvin; Gagne; Silberschatz, 2018), sendo um componente extremamente crucial, a implementação do escalonador deve garantir que tarefas de alta prioridade executem antes e que a troca de contexto seja o mais rápido possível. O algoritmo de escalonamento é o fator central para o comportamento do escalonador, sendo categorizados em dois principais grupos:

- **Cooperativos:** Tarefas precisam voluntariamente devolver o controle da CPU (com exceção de certas interrupções de hardware) para que as outras tarefas possam executar, isso pode ser feito explicitamente por uma função de largar ou implicitamente ao utilizar uma rotina assíncrona do sistema, como ler arquivos, receber pacotes de rede ou aguardar um evento (Galvin; Gagne; Silberschatz, 2018).
- **Preemptivos:** Além de poderem transferir a CPU manualmente, o escalonador fará trocas de contexto caso uma condição para a troca seja satisfeita. O algoritmo mais comum que serve de base para diversos escalonadores preemptivos é o Round-Robin onde tarefas possuem uma quantia de tempo máximo alocada para sua execução contínua, nomeada "fatia de tempo" ou "quantum" (Bos; Tanenbaum, 2023). Tarefas ainda podem possuir relações de prioridade, alterando a ordem que o escalonador realiza seu despache assim como o tamanho de sua fatia de tempo.

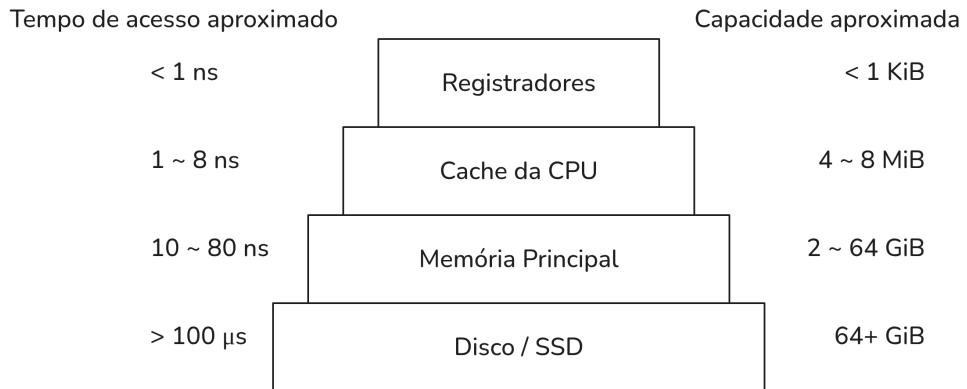
Sistemas operacionais de tempo real são tipicamente executados no modo totalmente preemptivo, mas o uso cooperativo também é viável e possui a vantagem de possuir um controle mais granular da execução das tarefas, mas é importante que seja tomado o cuidado adequado para que nenhum prazo de execução Hard Real-Time seja violado por uma tarefa inadvertidamente utilizando a CPU por uma quantidade longa de tempo.

2.3.3 Concorrência e Assincronia

Será utilizado a definição de concorrência como a habilidade de um sistema de lidar com múltiplas tarefas computacionais dividindo seus recursos (particularmente tempo de CPU e memória). Isto é, um sistema não necessariamente precisa ser paralelo (execuções múltiplas simultâneas) para possuir concorrência, mas para tornar paralelismo viável, o sistema necessita de mecanismos de concorrência (Armstrong, 2003).

Uma característica central para a utilidade de concorrência mesmo em situações em que paralelismo é limitado ou impossível vai além da pura expressividade do programador, existem assimetrias grandes na velocidade de acesso de disco, memória, rede, e caches da CPU como demonstrado na Figura 5. Para acessar recursos de forma eficaz, é necessário lidar com suas características inherentemente assíncronas. O uso de concorrência permite que uma tarefa seja suspensa e resumida (voluntariamente ou não) o que permite que o sistema não fique excessivamente ocioso (Galvin; Gagne; Silberschatz, 2018).

Figura 5 – Hierarquia de latência de acessos



Fonte: Adaptado de Bos; Tanenbaum, 2023

Implementar os mecanismos de concorrência adequados também permite lidar com interrupções de forma mais estruturada, um problema clássico de lidar com uma interrupção é restaurar a memória de pilha e registradores de forma adequada. Interrupções introduzem um fluxo de programa não local , violando as garantias fortes de escopo e ponto de entrada fornecidas por funções.

É uma tendência atual aumentar o número de núcleos em dispositivos devido à velocidades de relógio das CPUs possuirem ganhos marginais em relação ao impacto térmico. A maioria dos computadores de propósito geral (smartphones, tablets, desktops) tipicamente possuem 2 núcleos ou mais (Hennessy; Patterson, 2020). Essa tendência não se restringe apenas à computadores gerais, sistemas embarcados comerciais também podem se beneficiar tremendamente das possibilidades de paralelismo providas por mais de um núcleo, porém, é importante ressaltar que o uso de estado compartilhado se torna muito mais sensível à erros em um ambiente com múltiplos fluxos de execução, e medidas devem ser tomadas para evitar condições de corridas e deadlocks (Galvin; Gagne; Silberschatz, 2018).

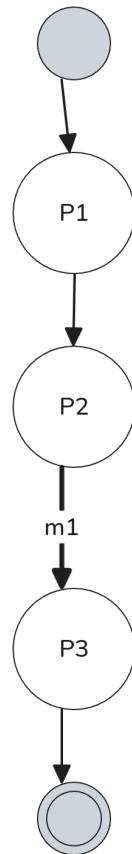
2.3.4 Execução de Tarefas na Presença de Falhas

Para uma representação mais clara e eficaz de um fluxo de execução sujeito a falhas, é possível utilizar de grafos resilientes a falhas como um mecanismo de diagramação. Nesta representação, os nós correspondem a tarefas, as quais podem ser executadas na mesma unidade de processamento ou não. As arestas do grafo representam o fluxo de execução, uma aresta não demarcada indica execução incondicional, enquanto arestas demarcadas com notação de mensagem designam uma execução condicional à transmissão de uma mensagem. Mensagens e tarefas representadas por símbolo circular indicam pontos ordinários no grafo, ao passo que símbolos quadrados denotam condições de transparência. (Izosimov, 2006).

Um grafo de processos tolerantes é falhas é definido como um grafo não ponderado direcionado acíclico com seus nós representando tarefas/processos, arestas representam o fluxo de execução e arestas nomeadas representam fluxo dependente da entrega de mensagens. Será utilizado a notação $P_X(N)$, onde X é o número identificador da tarefa, e N corresponde à sua N -ésima re-execução, por exemplo: $P_2(1)$ indica a primeira execução da tarefa P_2 , enquanto $P_1(3)$ indica a terceira reexecução da tarefa P_1 . Uma notação similar será utilizada para mensagens entre tarefas, $m_X(N)$, mensagens, assim como tarefas, estão sujeitas à falhas e custos adicionais de detecção, mas ao invés de re-execução, mensagens são re-enviadas (Izosimov et al., 2008).

Para melhor exemplificar a importância da detecção das falhas, será tomado como exemplo um grafo simples, com apenas 3 tarefas e uma mensagem. O grafo precisará tolerar uma falha transitória. O fluxo "ideal" é demonstrado na Figura 6.

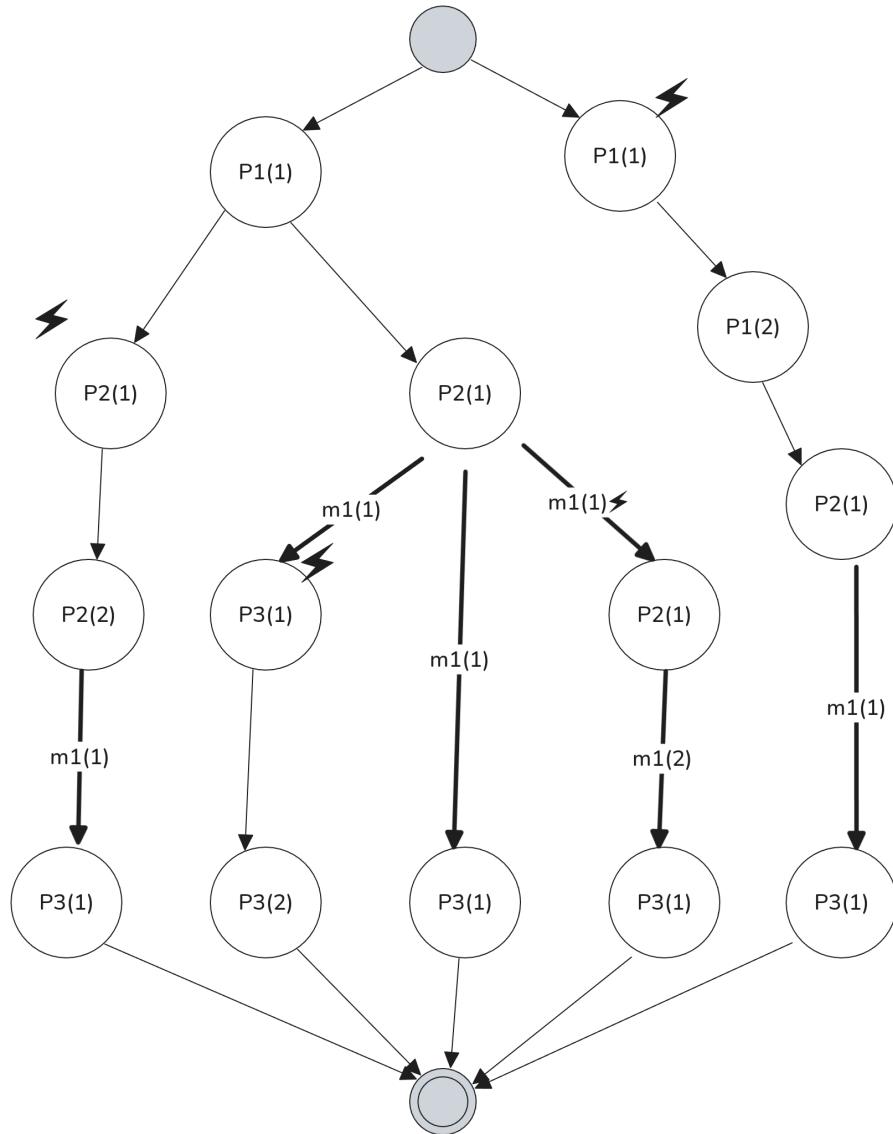
Figura 6 – Grafo com 3 processos e uma mensagem



Fonte: Elaborada pelo autor

Ao incluir os diferentes desvios possíveis na presença de apenas uma falha, o grafo de execução da Figura 7 é obtido. A incidência de falhas no processamento ou passagem de mensagens é indicada com um ícone de raio.

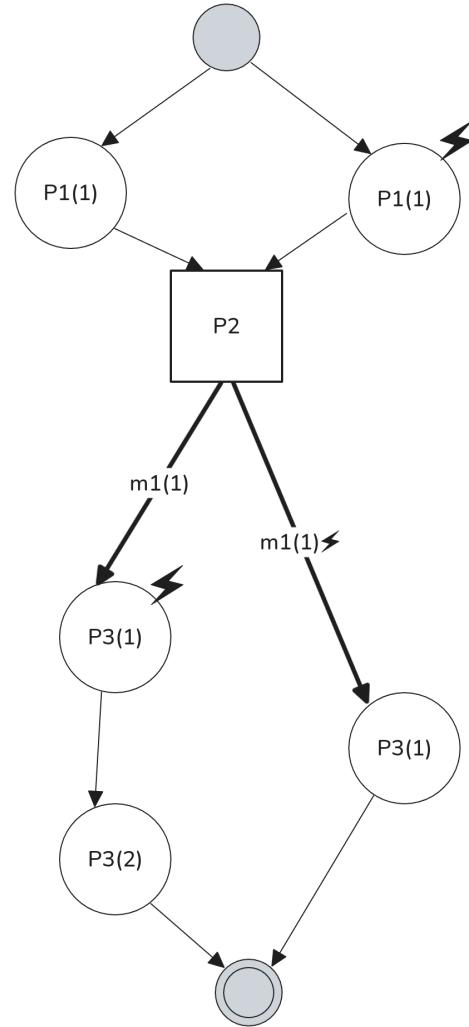
Figura 7 – Mesmo grafo, mas tolerante à uma falha transiente



Fonte: Elaborada pelo autor

Será introduzido transparência na tarefa P_2 , isto é, será executada com redundância temporal ou modular de tal forma que as tarefas subsequentes pudessem assumir "como se" uma falha nunca tivesse acontecido em P_2 após sua deadline ter sido completa, o grafo na Figura 8 demonstra a condição de transparência com um ícone retangular.

Figura 8 – Introdução de transparência em P_2



Fonte: Elaborada pelo autor

Introduzindo apenas um ponto de transparência é possível reduzir significativamente as possibilidades de execução do sistema, isso é particularmente benéfico para escalonadores ou algoritmos de tratamento de falhas baseados em máquinas de estado finito. Um grafo mais compacto é benéfico a previsibilidade do sistema, estabelecendo uma relação forte de pré-conclusão com sucesso ao respeitar o prazo da tarefa transparente (Izosimov, 2006).

Este exemplo é simples e tolera apenas uma falha transitória, porém processos complexos com múltiplas mensagens entre si causam um aumento exponencial de complexidade, especialmente caso seja necessário tolerar até k falhas transitórias.

Pode-se adicionar pontos de transparência através de reexecução ou redundância modular (se a deadline conjunta das N tarefas for determinística) para aumentar a confiabilidade e previsibilidade do sistema. Essa transparência não é gratuita: há um troca entre maior con-

fiabilidade no escalonador e menor imprevisibilidade na execução, com o custo de maior uso de CPU e memória. Todos os pontos precisam ser verificados, o que pode aumentar o tempo ocioso dos núcleos e exigir a extensão do prazo da tarefa para permitir reexecuções suficientes.

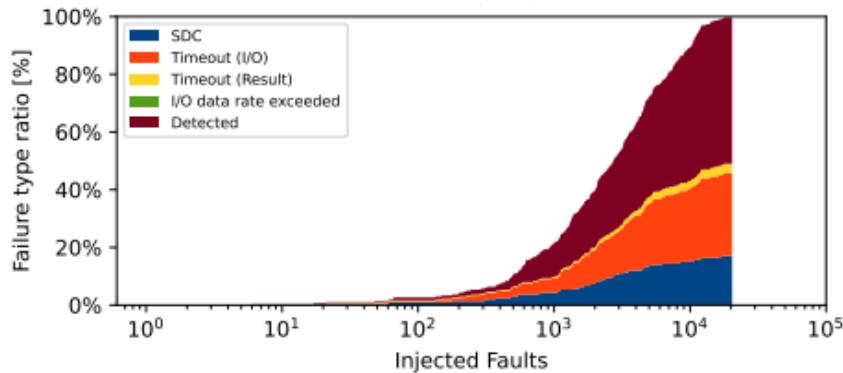
2.4 TRABALHOS RELACIONADOS

Durante a pesquisa bibliográfica para a fundamentação deste trabalho, foram selecionados três artigos que abordassem temas de tolerância à falhas em software.

2.4.1 Reliability Assessment of Arm Cortex-M Processors under Heavy Ions and Emulated Fault Injection

No trabalho de Gobatto et al. (2024) utilizam de um sistema COTS e criam um perfil de falhas com exposição a íons pesados assim como injeção artificial de falhas em software para posteriormente realizar uma adição de formas de detecção de falhas para melhorar a confiabilidade do sistema. Foi possível detectar mais da metade das falhas funcionais apenas com técnicas de software, como indicado na figura Figura 9.

Figura 9 – Análise de resiliência, dividida por categoria



Fonte: Gobatto et al., 2024

Uma outra observação foi que a memória foi duas ordens de magnitude maior em relação ao banco de registradores, indicando que é necessário um foco maior na detecção de falhas na memória (Gobatto et al., 2024).

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

O trabalho de Magagnin (2023) aborda uma aplicação de TMR (Replicação com $N = 3$) dos processos em um microkernel para a arquitetura RISC-V, para a validação da implementação é utilizado injeção de falhas lógicas em software emuladas com o depurador GDB e o com QEMU como emulador, o sistema é escrito na linguagem Rust que é capaz de fornecer

garantias de segurança de acessos à memória. Foi observado um aumento de confiabilidade do sistema em troca de um custo adicional de memória e tempo de execução.

2.4.3 Application-Level Fault Tolerance in Real-Time Embedded System

No trabalho de Afonso et al. (2008) são apresentadas técnicas de tolerância à falhas em um sistema operacional chamado BOSS, é utilizado uma interface de thread com a implementação de tolerância conformando à interface. O trabalho naturalmente explora o escalonador mas não entra em detalhamento profundo na parte de detecção, mas sim de prover uma biblioteca na forma de classes representando tarefas resilientes (Afonso et al., 2008). Um caso de estudo de um sistema de filtragem de radar é utilizado como projeto.

O trabalho demonstra também a viabilidade de prover interfaces mais abstratas que ainda sejam capazes de executar em sistemas de recursos restritos. Demonstraram-se resultados favoráveis para uma forma híbrida de tolerância com menor uso de CPU em relação à redundância tripla utilizando de técnicas em software combinado com um par de processadores com auto checagem (PSP).

2.4.4 Análise Comparativa dos trabalhos relacionados

Após comparar os trabalhos, o de natureza mais similar em termos de arquitetura é o de Afonso et al. (2008) e o mais similar em termos de técnicas de injeção e materiais utilizados a técnica de confiabilidade apresentada em Magagnin (2023) . O Quadro 1 demonstra as principais diferenças entre os trabalhos.

Quadro 1 – Comparação dos trabalhos relacionados

Trabalho	Sistema	Hardware	Injeção	Técnicas
Gobatto et al., 2024	Bare Metal, FreeRTOS	CY8CKIT-059	Física e Lógica em Software	Redundância de Registradores, Deadlines, Redução de Registradores, Asserts
Magagnin, 2023	RISC-V Emulado no QEMU	Microkernel de Mezger et al. (2021)	Emulada em Software	Redundância Modular, Segurança de memória extra (Borrow checker do Rust)
Afonso et al., 2008	BOSS	Máquinas PowerPC 823 e um PC x86_64 não especificado	Simulada em Software	Redundância Modular, Deadlines, Rollback/Retry
Este Trabalho	FreeRTOS	STM32 Blackpill	Lógica em Software e Hardware	Heartbeat/Deadline, Asserts, Reexecução e Redundância de Tarefas

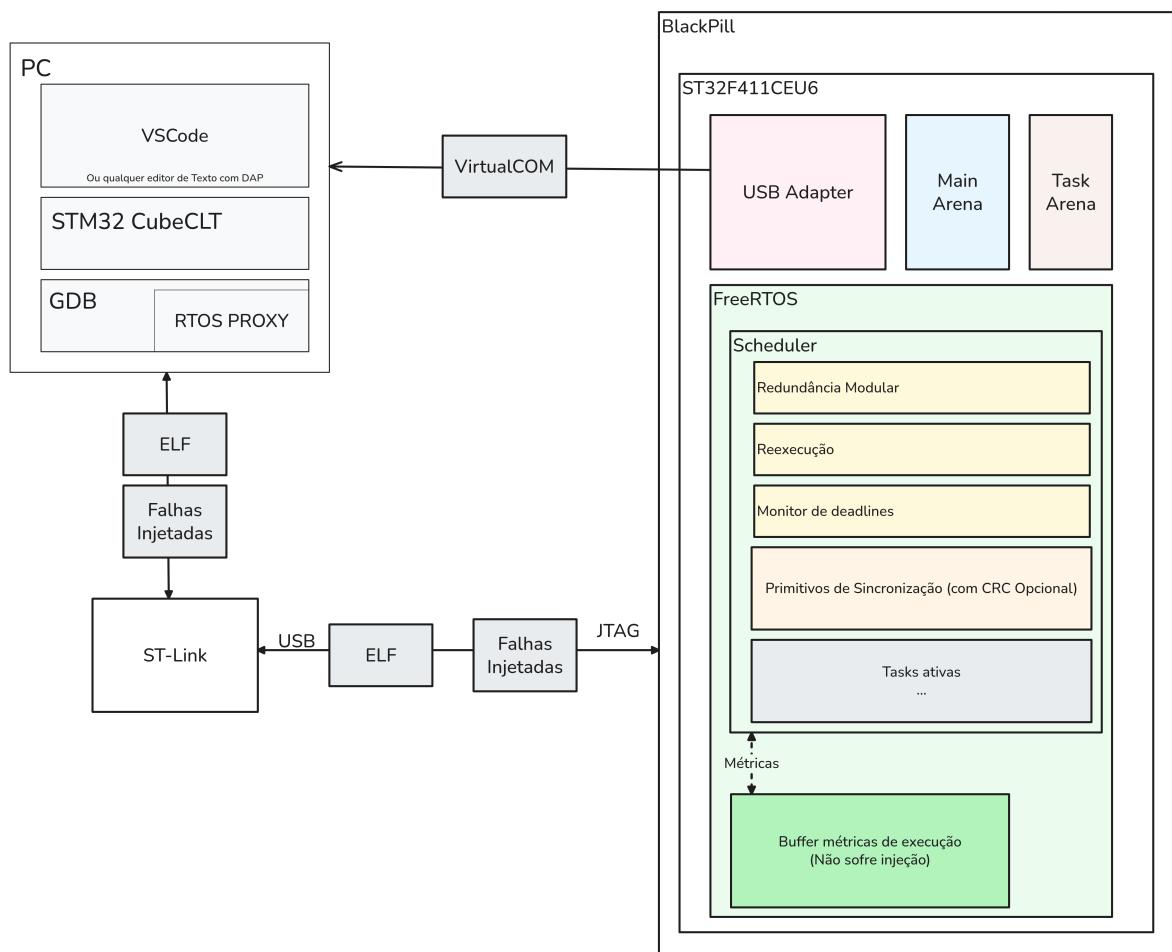
3 PROJETO

3.1 VISÃO GERAL

Serão implementados mecanismos de tolerância utilizando do FreeRTOS como base, as técnicas implementadas serão discutidas em maior detalhe na subseção 3.4.2. Para a coleta das métricas de eficácia e custo computacional será utilizado um cenário de injeção de falhas lógicas em hardware utilizando do depurador em chip ST-Link, os detalhes da campanha de injeção de falhas são abordados na subseção 3.5.2.

A Figura 10 sumariza a relação entre os principais componentes, as técnicas de tolerância são implementadas como complementos ao FreeRTOS e o processo de injeção é controlado por um computador externo que envia comandos para o depurador em chip com o propósito de simular uma falha.

Figura 10 – Principais componentes do projeto



Fonte: Elaborada pelo autor

3.2 PREMISSAS

Será partido do ponto que ao menos o processador que executa o escalonador terá registradores de controle (ponteiro de pilha, contador de programa, endereço de retorno) que sejam capazes de mascarar falhas. Apesar de ser possível executar os algoritmos reforçados com análise de fluxo do programa e adicionar redundância aos registradores, isso adiciona um grau grande de complexidade que foge do escopo do trabalho. Como mencionado na seção 2.4, a memória fora do banco de registradores pode ser duas ordens de magnitude mais sensível à eventos disruptivos (Gobatto et al., 2024).

Durante os testes, será assumido um critério Hard Real-Time sem presença de falhas, isto é, não é aceitável o descumprimento do prazo de execução quando não houver falhas. Na presença de falhas, o prazo também deve ser mantido porém será considerado preferível que falhas sejam detectadas e causem um atraso ao invés de causar uma corrupção silenciosa.

Com o fim de reduzir o tamanho do executável e manter o fluxo de mais previsível não serão utilizados mecanismos de exceção com desenrolamento (unwinding) da pilha. Também não será utilizado de RTTI (Runtime Type Information). Todos os erros devem portanto ser tratados como valores ou como falhas lógicas.

Necessariamente, é preciso também presumir que testes sintéticos possam ao menos aproximar a performance do mundo real, ou ao menos prever o pior caso possível com grau razoável de acurácia. O uso de testes sintéticos não deve ser um substituto para a medição em uma aplicação real, porém, uma bateria de testes com injeção artificial de falhas pode ser utilizada para verificar as tendências e custos relativos introduzidos, mesmo que não necessariamente reflitam as medidas absolutas do produto final.

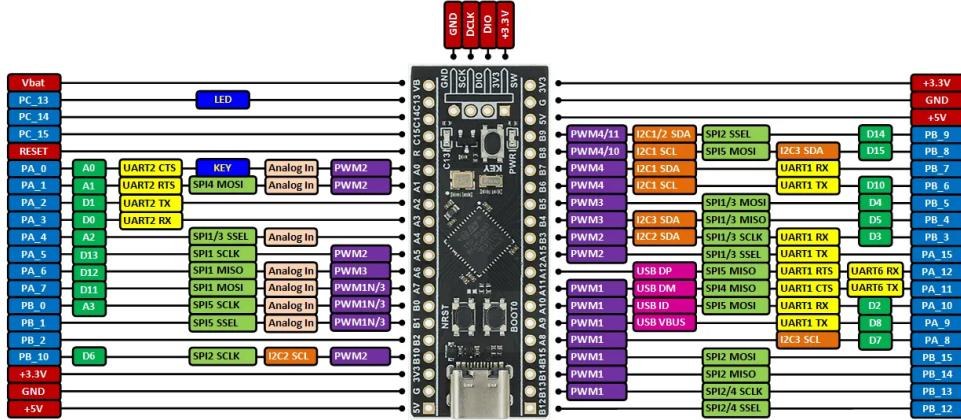
Portanto, será assumido que os resultados extraídos de injeção de falhas artificiais, apesar de menos condizentes com os valores absolutos de uma aplicação e não sendo substitutos adequados na fase de aprovação de um produto real, são ao menos capazes para realizar uma análise quanto ao custo proporcional introduzido, e devido à sua facilidade de realização e profundidade de inspeção possível, serão priorizados inicialmente neste projeto.

3.3 METODOLOGIA

3.3.1 Materiais

Será utilizada a linguagem C++ com o compilador GCC (ou Clang), o alvo principal do trabalho será um microcontrolador STM32F411CEU6 "BlackPill"32-bits da arquitetura ARM, como visto na Figura 11.

Figura 11 – Diagrama da STM32F411CEU6 ("BlackPill")



Fonte: STMicroelectronics, 2025c

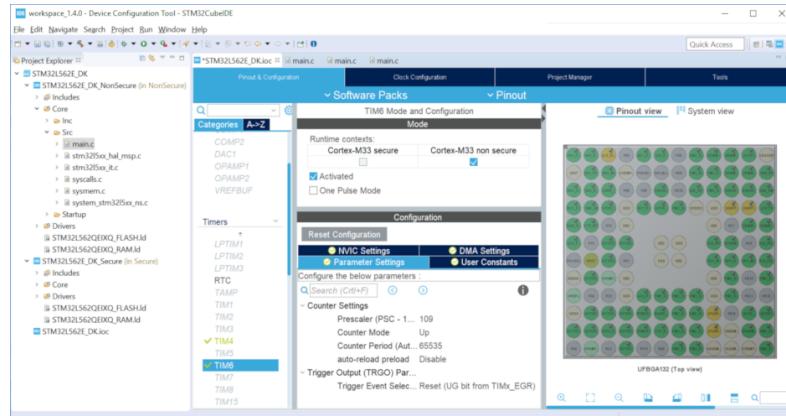
Para a injeção de falhas será utilizado o depurador GDB em conjunto com uma ferramenta de depuração de hardware ST-LINK (Figura 12), a comunicação do ST-LINK é feita via USB com o computador e via JTAG com o microcontrolador alvo, também será usado em conjunto ferramentas do fabricante como o CubeIDE, CubeMX e CubeCLT.

Figura 12 – ST-LINK/V2



Fonte: STMicroelectronics, 2025a

Figura 13 – STMCubeIDE



Fonte: STMicroelectronics, 2025b

Durante a fase de desenvolvimento dos algoritmos será utilizado o QEMU juntamente com as ferramentas anteriormente citadas, assim como sanitizadores de memória e condições de corrida (ASan, TSan, UBSan) para auxiliar na detecção de erros mais cedo durante o desenvolvimento.

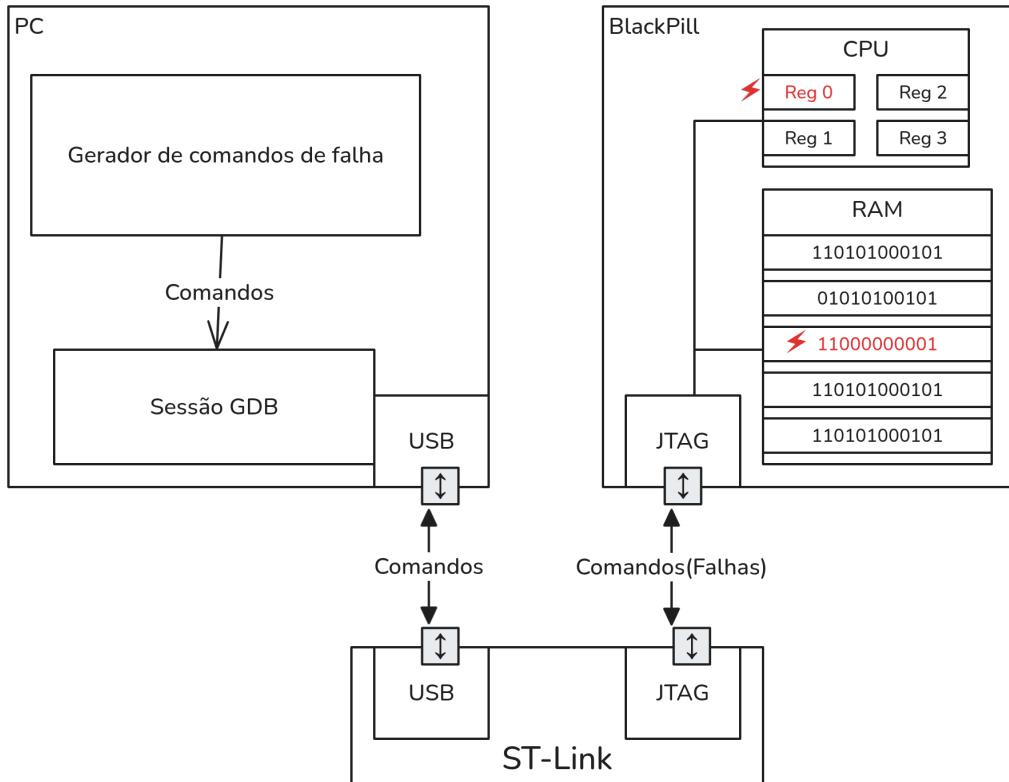
O sistema operacional de tempo real escolhido foi o FreeRTOS, por ser extensivamente testado e documentado e prover um escalonador totalmente preemptivo com um custo espacial relativamente pequeno, além disso, os contribuidores do FreeRTOS mantém uma lista grande de versões para diferentes arquiteturas e controladores, facilitando drasticamente o trabalho ao não ter que criar uma HAL do zero.

3.3.2 Métodos

Serão utilizadas as seguintes técnicas de tolerância à falhas implementadas em software: CRCs para dados, redundância modular, reexecução, sinal heartbeat e asserts. O detalhamento específico de cada técnica é abordado em maior detalhe na subseção 3.4.2.

Para a criação da análise, serão realizados testes com injeção lógica em hardware utilizando-se do ST-Link em combinação com um computador que emitirá os comandos para injeção via depurador, as falhas serão de natureza transiente e afetarão valores na memória (corrupção silenciosa). A Figura 14 detalha de forma mais específica o fluxo de gerar uma falha. As combinações específicas de falhas e técnicas escolhidas são abordadas na subseção 3.5.2.

Figura 14 – Injeção lógica em hardware



Fonte: Elaborada pelo autor

A coleta de métricas será realizada com os contadores de incremento atômico, o tempo de execução das tarefas, seu espaço de memória utilizado e o número de falhas detectadas será armazenado em uma estrutura que residirá em um segmento de memória que é deliberadamente isento de falhas. Ao fim da execução, será utilizado o depurador para ler estes valores.

Com o objetivo de promover a reutilização de código, será criada uma interface que abstrai noções comuns de tarefa. A interação da interface com o resto do sistema é abordada em maior detalhe na subseção 3.4.1.

3.4 ANÁLISE DE REQUISITOS

Quadro 2 – Requisitos funcionais

Requisito	Descrição
RF01	Implementação de todos os algoritmos descritos na subseção 3.4.2
RF02	Configuração do mecanismo de tolerância, prioridade e prazo de execução da tarefa
RF03	Cumprimento do prazo estipulado no momento de criação da tarefa caso não exista presença de falhas
RF04	Dependabilidade superior à versão do sistema sem técnicas
RF05	Monitoramento do número de falhas detectadas e violações de prazos

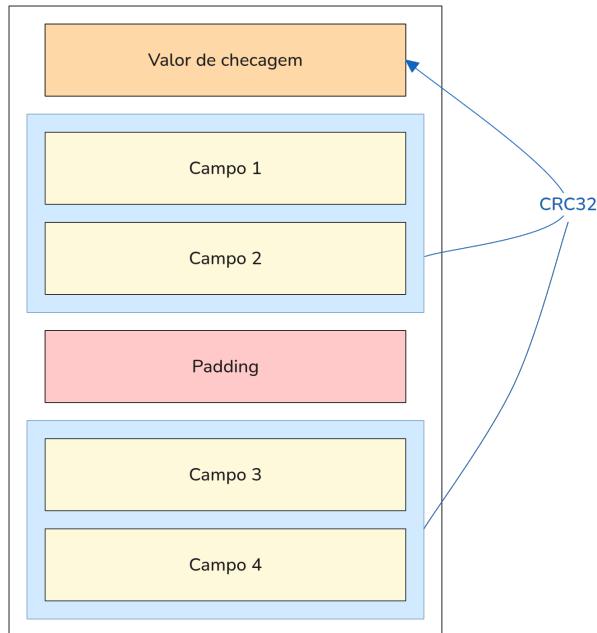
Quadro 3 – Requisitos não funcionais

Requisito	Descrição
RNF01	O consumo de memória deve ser pré determinado em tempo de compilação ou na inicialização do sistema
RNF02	A interface deve ser construída sobre o escalonador preemptivo do FreeRTOS
RNF03	Deve ser compatível com arquitetura ARMv7M ou ARMv8M
RNF04	Implementação realizada em C++ (versão 20 ou acima)
RNF05	Código Fonte disponível sob licença permissiva

3.4.1 Interface

Para melhor generalizar o uso das técnicas, utiliza-se de uma abstração da estrutura de tarefa juntamente com um mecanismo de validação de payload via CRC32. A Figura 15 demonstra a estrutura de um envelope de dados que inclui um valor de checagem, note que bytes de enchimento necessitam ser ignorados.

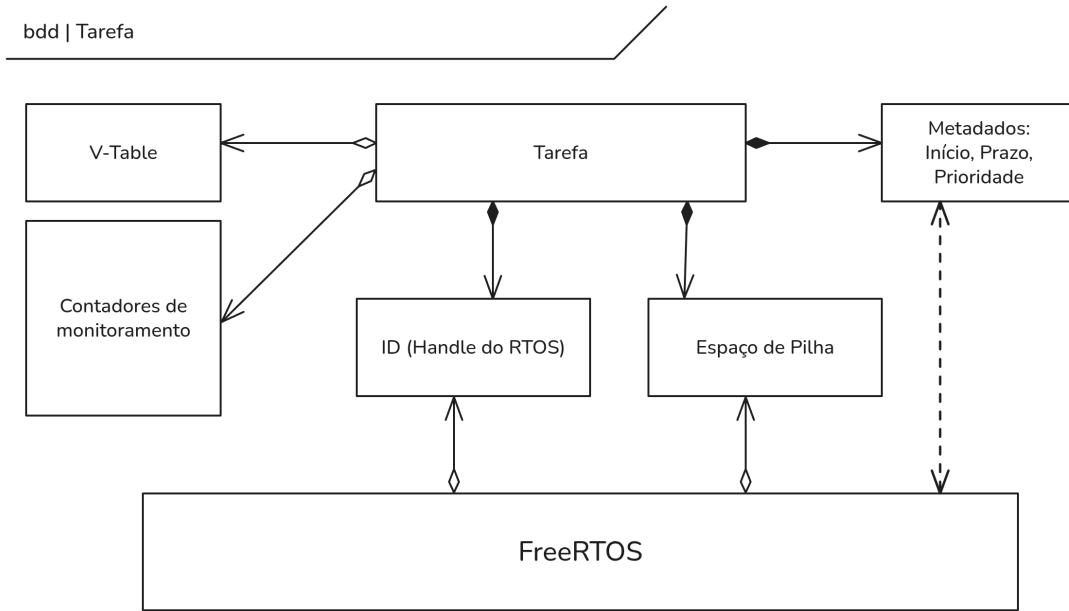
Figura 15 – Layout de uma mensagem



Fonte: Elaborada pelo autor

A tarefa é um objeto de interface que abstrai parte do estado utilizado pelo RTOS e provê métodos para sua inicialização, término e cancelamento. Uma tarefa possui um espaço de pilha dedicado, e uma V-Table que inclui os métodos providos para sua execução. A identificação da tarefa se dá pelo seu ID, que diretamente mapeia o recurso que encapsula o estado da tarefa no RTOS. O diagrama na Figura 16 demonstra a relação de uma tarefa e os outros componentes do sistema.

Figura 16 – Objeto que implementa a interface de Tarefa



Fonte: Elaborada pelo autor

3.4.2 Algoritmos e Técnicas

Para a implementação da funcionalidade de tolerância à falhas, algumas das técnicas abordadas no Capítulo 2 serão utilizadas. O detalhamento sobre a implementação será abordado nesta seção.

3.4.2.1 CRC: Cyclic Redundancy Check

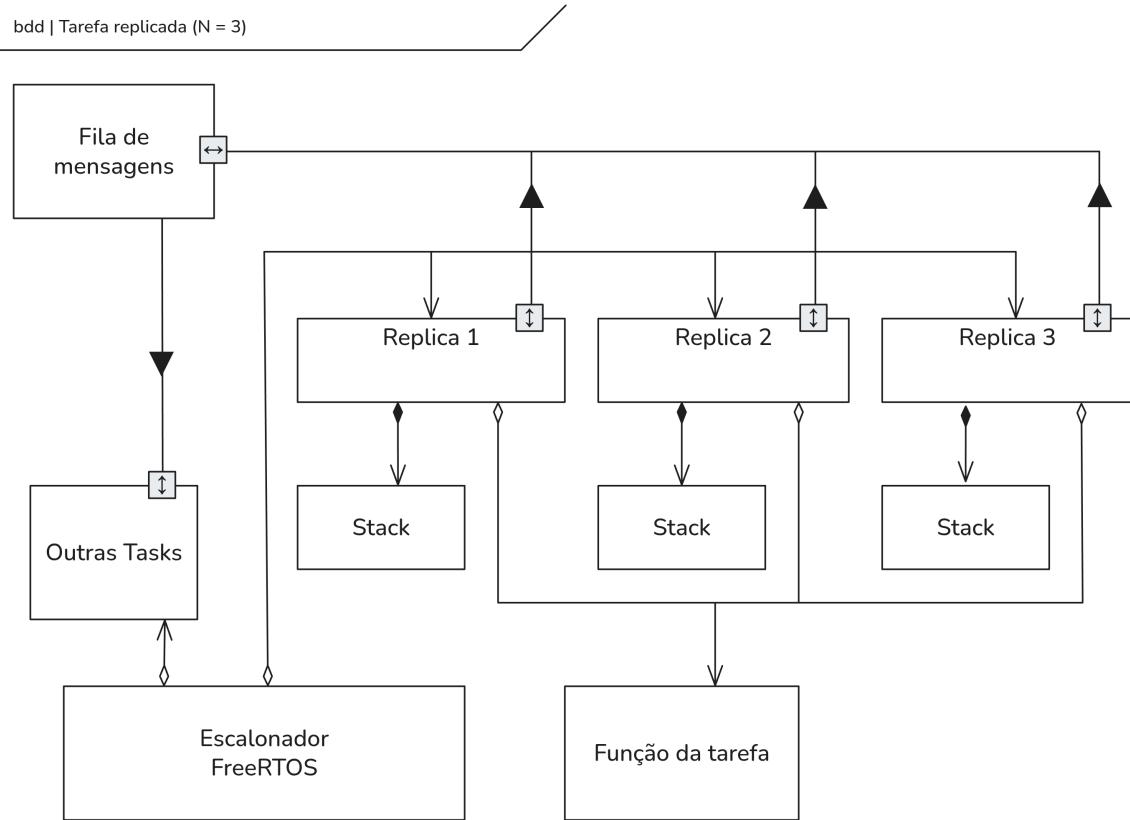
Será implementado o CRC-32C, que já é aplicado em sistemas de arquivos como o Btrfs e o ext4, assim como em protocolos de rede como iSCSI e SCTP. Seu Polinômio gerador P é:

$$\begin{aligned}
 P = & x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} \\
 & + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + 1
 \end{aligned} \tag{1}$$

3.4.2.2 Redundância Modular

Para a aplicação da redundância modular, neste caso a redundância modular tripla, será feito a replicação concorrente da tarefa, cada tarefa possui um espaço de pilha próprio e são escalonadas de forma convencional pelo FreeRTOS. O corpo das tarefas não é replicado, e continua como parte de memória para apenas leitura e execução, um exemplo da relação de réplicas de tarefas executando em relação ao resto do sistema pode ser observado na Figura 17.

Figura 17 – Diagrama de bloco de Redundância modular



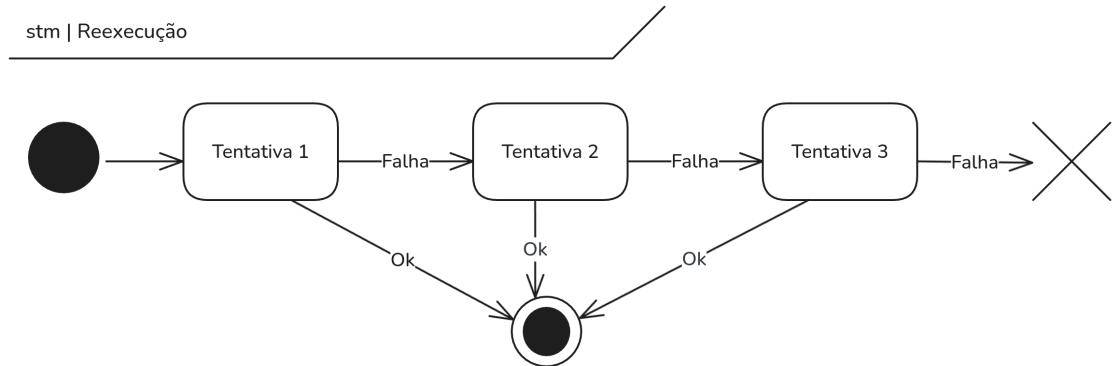
Fonte: Elaborada pelo autor

3.4.2.3 Reexecução

A implementação de tarefas com reexecução é baseada no uso de execuções consecutivas que reutilizam do mesmo espaço de pilha, uma tarefa pode ser sempre reexecutada N vezes, servindo um propósito similar à técnica de redundância, ou executada *até* N vezes, encerrando a execução imediatamente após não encontrar nenhuma falha. Para os propósitos deste trabalho, será utilizada a segunda técnica, pois permite mais oportunidade para o escalonador encaixar trabalho no tempo ocioso, e também por ser um exemplo mais bem estudado na fundamentação teórica deste trabalho por Isosimov et. al.

Assumindo $N = 3$, o diagrama na Figura 18 descreve uma máquina de estado finito para a execução de uma tarefa com reexecução, espera-se que o caso médio seja a execução diretamente para um estado correto. Para a criação de uma condição de transparência, o prazo da tarefa deve ser o pior caso possível de N execuções.

Figura 18 – Estados de uma reexecução

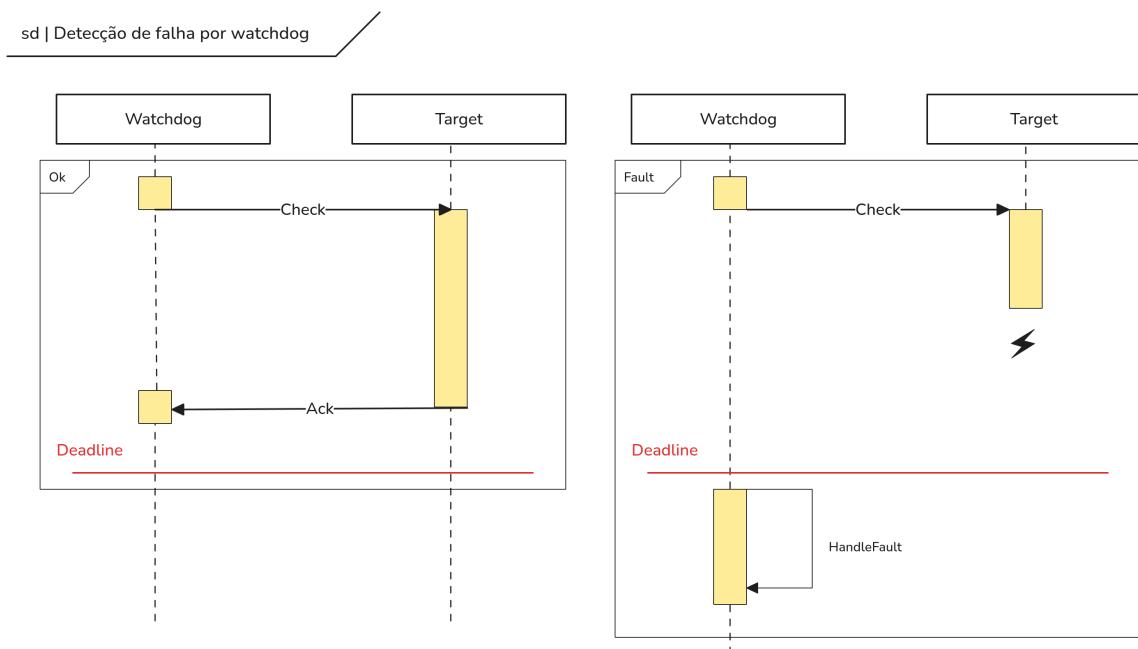


Fonte: Elaborada pelo autor

3.4.2.4 Sinal Heartbeat / Deadline

Para a implementação dos sinais de heartbeat será utilizado uma tarefa que servirá como um monitor que associa uma chave (como o ID da tarefa) a um prazo específico. Será utilizado uma escrita atômica de um número e o temporizador do sistema no momento da escrita, se houver uma violação do prazo combinado na criação da tarefa e o prazo apresentado, é considerado que ocorreu uma falha. Este fluxo é visualmente representado na Figura 19.

Figura 19 – Sinal Heartbeat



Fonte: Elaborada pelo autor

3.4.2.5 Asserts

Asserts serão utilizados para verificar invariantes, qualquer quebra de contrato de função ou invariante que é coberta com um assert deve resultar em uma falha.

3.5 PLANO DE VERIFICAÇÃO

Para a validação dos algoritmos e técnicas utilizadas(**RF01**, **RF02**) serão feitos testes unitários das técnicas isoladas.

A validação da detecção de falhas e vencimento de prazos (**RF04**, **RF05**) serão preliminarmente testadas com injeção lógica em software e será validado definitivamente durante o teste com injeção lógica em hardware. Importante notar que a priorização de tarefas e parte dos algoritmos de comunicação entre tarefas já são implementados no FreeRTOS.

Como o produto final do trabalho requer uma análise de resiliência e do custo das técnicas, a seção seguinte aborda a campanha de injeção utilizada, que será aplicada com método lógico em hardware para a análise final.

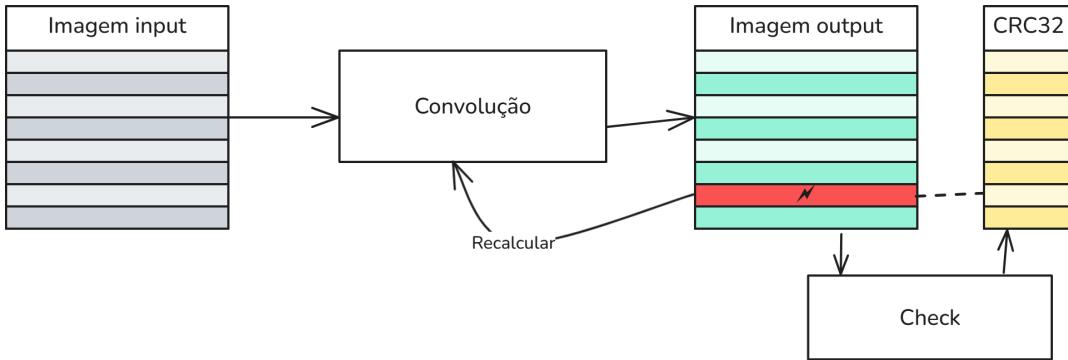
3.5.1 Programa de Teste

Para testar a viabilidade e o impacto das técnicas implementadas, será utilizado um programa de teste que lerá uma imagem e aplicará um filtro de detecção de bordas Sobel utilizando convolução, o resultado é salvo num arquivo após ser enviado via VirtualCOM. O output é construído linha por linha. Cada linha possui um prazo hard real-time de 125ms para ser computada.

O programa requer um uso de CPU alto e realiza uma grande quantidade de acessos à memória, o processo de convolução também necessita de valores estáveis de kernel e offsets para calcular o valor de cada pixel. A fundamentação teórica da aplicação destes algoritmos é primariamente extraídos de um livro de processamento digital de sinais por Smith (1999) .

A estrutura do programa com CRC é demonstrada na figura Figura 20, caso o check de CRC não seja bem sucedido, a linha é recomputada com a mesma técnica de tolerância mais uma vez. Caso CRC esteja desabilitado, este passo é ignorado.

Figura 20 – Fluxo do programa de exemplo



Fonte: Elaborada pelo autor

3.5.2 Campanha de Injeção de Falhas

Para testar a injeção de falhas serão utilizados mecanismos lógicos em software e em hardware com auxílio do depurador ST-Link. As falhas serão de natureza transiente e focarão no segmento de memória com leitura e escrita.

A primeira rodada de injeção consiste em símbolos conhecidos do programa de exemplo (linha do output e imagem resultado) com o objetivo de demonstrar as técnicas em uma situação de falha previsível, onde o valor e posição exata do upset não são conhecidos, mas não ocorre corrupção direta do estado do stack frame.

A segunda rodada consiste em execução e causar um upset de N bytes a partir do endereço de uma variável local. O objetivo deste teste é avaliar como as tarefas irão se comportar na presença de uma falha que diretamente corrompe seu stack frame.

As combinações listadas no Quadro 4 visam observar o impacto das duas técnicas que causam alteração significativa no fluxo de execução (TMR e Reexecução) e da técnica de detecção de corrupção CRC. Todos os testes terão acompanhamento de sinal Heartbeat no início e no final para monitorar cumprimento do prazo de execução global assim como uma instância local.

Quadro 4 – Combinações de técnicas utilizadas

Reexecução	Redundância modular	CRC	Deadline/Heartbeat	Asserts
-	-	-	X	X
-	-	X	X	X
X	-	-	X	X
X	-	X	X	X
-	X	-	X	X
-	X	X	X	X

3.6 ANÁLISE DE RISCOS

O trabalho é de risco baixo, dado que constrói em cima de fundações técnicas previamente exploradas, porém dentro dos principais riscos que possam alterar ou causar problemas durante a produção da análise encontram-se:

Quadro 5 – Análise de riscos

Risco	Probabilidade	Impacto	Gatilho	Mitigação
Funcionalidades e API do RTOS é incompatível com a interface proposta pelo trabalho	Baixo	Alto	Implementar interface no RTOS	Utilizar outro RTOS, modificar o FreeRTOS, adaptar a interface
Problemas para injetar falhas com depurador em hardware	Baixa	Alto	Realizar injeção no microcontrolador	Utilizar de outro depurador, depender de falhas lógicas em software como última alternativa
Não conseguir coletar métricas de performance	Baixa	Médio	Teste em microcontrolador ou ambiente virtualizado	Inserir pontos de medição manualmente

4 DESENVOLVIMENTO

4.1 ESTRUTURA DA IMPLEMENTAÇÃO

Esta seção aborda os detalhes técnicos mais importantes para a implementação das técnicas de tolerância à falhas

4.1.1 Estratégia de Alocação

Com exceção de algumas alocações internas do FreeRTOS e da biblioteca do sistema, a maioria das alocações dinâmicas realizadas é feita com o uso de alocadores do tipo arena de memória. Este padrão de alocador comporta-se similarmente à uma pilha, o incremento e decremento de um ponteiro determina a barreira entre memória disponível e alocada. Portanto alocações são rápidas e simples, assim como liberação total ou reset para um estado anterior, a principal desvantagem destes alocadores é não serem capazes de expressar ordens de alocação e liberação granulares muito distintas (Fleury, 2022).

Pela estrutura do alocador ser simples, a coleta de métricas e injeção de falhas foi simplificada dado que é possível acessar a maioria dos componentes do sistema apenas em termos de seu deslocamento em relação à sua arena que é facilmente identificável no depurador ao pesquisar por seu símbolo.

Cada arena serve para encapsular um lifetime conjunto de N alocações, todas as alocações estão vivas, ou todas estão mortas. Para o gerenciamento de tarefas essa técnica mostrou-se satisfatória e fácil de utilizar, pois tarefas e seu estado local são inseparáveis no comportamento normal do sistema, também é possível facilmente recolher memória de tarefas replicadas canceladas durante a aplicação de técnicas de TMR e Reexecução. A estrutura de arena falha em ser adequada no gerenciamento de múltiplos objetos com lifetimes distintos, o único caso que isso ocorre na aplicação é na alocação de observadores de deadline, onde o problema é mitigado com o uso de uma lista intrusiva.

4.1.2 Tarefas

A classe RawTask é uma abstração fina que encapsula a interação com o escalonador e fornece mecanismos unificados para gerenciamento do ciclo de vida das tarefas. Esta estrutura foi projetada para oferecer uma interface consistente que abstrai as especificidades da plataforma subjacente.

Quadro 6 – Declaração da estrutura RawTask

```

1 struct RawTask {
2     RawTaskFunc func = nullptr;
3     Arena* arena = nullptr;
4     void* args = nullptr;
5     DeadlineSlot* deadline = nullptr;
6     u32 stack_size = 0;
7     u32 args_size = 0;
8     u32 id{};
9     Atomic<TaskStatus> _status = TaskStatus_Initialized;
10    TaskCancelCallback on_cancel = nullptr;
11    RawTaskPlatformSpecificData _specific{};
12
13    TaskStatus status();
14
15    void join(CALLER_LOCATION);
16
17    void cancel(CALLER_LOCATION);
18
19    ~RawTask() {}
20
21    bool _platform_init(Arena* a, usize stack_size, RawTaskFunc func, void*
22                         args);
23    bool _platform_join();
24    bool _platform_cancel();
25};

```

Os atributos centrais da classe demonstrados no Quadro 6 incluem o ponteiro para o corpo da tarefa (`func`) que é envelopado por uma função `task_wrapper` para permitir a inicialização e saída adequada. Os argumentos da função (`args`), o tamanho da pilha (`stack_size`) e o identificador único da tarefa (`id`). O macro `CALLER_LOCATION` é apenas uma expansão para injetar a localização do código que chamou a função utilizando a funcionalidade de `std::source_location` do C++ 20 para facilitar diagnóstico de problemas.

O atributo opcional `deadline` referencia um estrutura `DeadlineSlot` que serve como um handle para um monitor, a tarefa pode utilizar este monitor para validar seu prazo de execução que é monitorado externamente. Já o callback `on_cancel` é uma rotina opcional que pode ser executada na deleção da tarefa, primariamente para garantir a limpeza de certos recursos mesmo no evento em que o cancelamento impeça a execução dos destrutores inseridos pelo compilador.

O estado da tarefa é mantido através do atributo atômico `status` que indica o ponto geral de execução da tarefa (Não-Inicializado, Inicializado, Executando, Finalizada, Finalizada com Erro). A utilização de operações atômicas neste caso é essencial para execução em ambientes concorrentes e cuidado extra foi tomado na escolha do ordenamento de memória correto para evitar dupla desalocação de recursos. O mecanismo de alocação em arenas auxilia, já que

liberação repetida de memória é idempotente.

A classe genérica `BasicTask` demonstrada no Quadro 7 incrementa a classe `RawTask` com conveniência adicional do uso de funções anônimas com grupos de captura e garantia de tipagem de seu retorno. O tipo `TaskContext` é passado como argumento obrigatório pois permite que a tarefa sendo executada sinalize cancelamento ou conclusão prematura, permitindo um controle superior no caso de tarefas redundantes.

Quadro 7 – Declaração da estrutura `BasicTask` (Implementação omitida)

```

1 template<
2     typename Output,
3     Callable<Output, TaskContext> TaskFunc,
4     Callable<void, TaskContext> OnCancel
5 >
6 struct BasicTask {
7     RawTask _task;
8     TaskFunc _func;
9     OnCancel _on_cancel;
10    Option<Output> _result;
11
12    static void _basic_task_wrapper(RawTask* t);
13
14    static void _basic_task_cancel_wrapper(RawTask* t);
15
16    Option<Output> result();
17
18    bool has_result() const;
19
20    TaskStatus status() const;
21
22    RawTask* raw_task();
23
24    u32 id();
25
26    void join();
27
28    void cancel();
29 };

```

Apesar da implementação mais complexa o uso desta classe é relativamente ergonômico, sendo similar em seu uso à classe `std::thread` introduzida no C++11 como demonstrado no Quadro 8.

Quadro 8 – Utilizando uma BasicTask para calcular uma soma

```

1 auto numbers = Array<i32, 7>{-4, -2, 0, 6, 9, 2, 1};
2 auto task = make_basic_task(&arena, [numbers] (TaskContext* ctx) -> i32 {
3     i32 acc = 0;
4     for(i32 x : numbers) {
5         acc += x;
6     }
7     return x;
8 });
9
10 task.join();
11 auto sum = task.result().unwrap();
12 printf("Sum = %d\n", sum);

```

4.1.3 Deadlines e Heartbeat

A classe `DeadlineWatcher` descrita no Quadro 9 implementa um vigilante responsável pelo gerenciamento centralizado de deadlines ativas, também podendo ser utilizado como um sinal de heartbeat ao utilizar uma deadline deliberadamente mais lenta para utilizar como uma flag de progresso periódico. Internamente, baseia-se em um vetor de descritores que mantém informações temporais de todas as tarefas sob monitoramento. Um spinlock é utilizado para sincronização pois as seções críticas são pequenas.

É possível também disparar uma tarefa que executa um monitor que é monitorado por outro monitor, permitindo a criação de sub-deadlines arbitrariamente aninhadas, este padrão é uma versão rudimentar da técnica de Árvores de Supervisão que é comumente encontrada em sistemas distribuídos com tolerância à falhas (Armstrong, 2003).

Quadro 9 – Declaração da estrutura DeadlineWatcher

```

1 struct DeadlineWatcher {
2     Slice<DeadlineSlot> slots;
3     Spinlock _lock{};
4     Atomic<u32> _count;
5
6     usize count() const;
7
8     bool watch(RawTask* task, Duration limit);
9
10    void remove(DeadlineSlot* node);
11
12    void clear();
13
14    bool scan();
15
16    auto lock_guard(){
17        return _lock.guard();
18    }
19};
20
21 struct DeadlineSlot {
22     TimeTick last_tick;
23     Duration limit;
24     RawTask* task;
25
26     void reset(){ last_tick = tick_now(); }
27 };

```

Para a leitura e comparação de temporizadores, são utilizadas 2 funções específicas para cada plataforma: `tick_now` para observar o estado temporal atual e `tick_frequency` para ler a frequência do temporizador.

A funcionalidade de `scan` é responsável por validar as deadlines e limpar descritores que foram encerrados, o seu retorno indica um estado de sucesso. No caso do uso do temporizador embutido da blackpill para watchdog independente (IWDG), é possível utilizar o output do monitor principal para decidir se o watchdog deve ser reiniciado ou não.

4.1.4 CRC32

A verificação de integridade via CRC32 utiliza a técnica de LUT (Look-up table) para todos os possíveis bytes de entrada considerando o polinômio gerador escolhido, armazenando-os em uma tabela estática. Durante o cálculo do CRC cada byte dos dados é processado através de uma operação de indexação na tabela seguida de uma operação XOR com o estado acumulado, eliminando a necessidade de cálculos bit a bit. O valor final também pode ser incrementalmente computado, que é essencial para estruturas que possuem descontinuidades em memória ou que sejam de tamanho muito grande.

Cuidado adicional deve ser tomado ao calcular o CRC de uma estrutura arbitrária, pois compiladores frequentemente adicionam bytes de enchimento (padding) para garantir alinhamento adequado dos dados na memória. O conteúdo destes bytes é indefinido, portanto calcular o CRC de duas estruturas idênticas pode resultar em números diferentes devido à inclusão de bytes de enchimento com valores distintos. Para remediar isso, uma restrição de tipo `CRC_Checkable` deve ser implementada por todas as estruturas que desejam fornecer essa funcionalidade, é parte do contrato do conceito que o implementador adequadamente lide com bytes de enchimento.

4.1.5 Asserts

Foram utilizados 2 níveis de assert, globais (críticos) e locais (contextuais). Para evitar conflito com o macro da biblioteca padrão, a função de assert é nomeada com o sinônimo `ensure`. A função `ensure` global verifica um predicado, caso seja falso, dispara uma exceção do processador para causar uma reinicialização forçada, indicando uma falha catastrófica no sistema. A função membro `TaskContext::ensure` não causa uma falha catastrófica, mas apenas cancela a tarefa.

Asserções locais são utilizadas para detectar a violação de invariantes dentro de um contexto recuperável, já asserções globais são reservadas para propriedades fundamentais que não podem ser violadas.

4.1.6 Dependências Adicionais

A camada de abstração de hardware é fornecida pela STMicroelectronics através de sua ferramenta de geração de código STMCubeMX. Para a comunicação VirtualCOM por USB foi utilizado o pacote de software `USB_DEVICE` da STMicroelectronics. Uma biblioteca utilitária geral (`base`) que é compatível com alvos freestanding é fornecida pelo autor. A biblioteca de domínio público `stb_sprintf` por Sean Barrett foi usada para permitir uma implementação consistente de formatação textual entre todas as plataformas, apesar de não ser parte análise do trabalho, foi extremamente útil para diagnosticar e reportar dados do sistema durante o desenvolvimento.

4.1.7 Compilação do Projeto

Apesar do alvo de compilação do projeto ser um microcontrolador com o FreeRTOS com CMSISv2, o design transparente permitiu também a implementação para testes e um ciclo de debug mais rápido em sistemas operacionais completos (Linux e Windows 10/11) sem necessitar de um simulador. Os detalhes de plataforma são consolidados em uma única unidade de tradução e condicionalmente compilados dependendo do sistema alvo. Todo código específico de plataforma pode ser encontrado em suas respectivas unidades `platform_PLATAFORMA.cpp`.

4.1.8 Programa de Teste

O programa de teste escolhido foi desenvolvido em 3 versões (Controle, TMR, Reexecução), o algoritmo de convolução é o mesmo como demonstrado no Capítulo 3, tendo como alteração apenas o setup das tarefas adicionais (no caso de TMR) e a alocação inicial de memória. Para carregar a armazenar a imagem embutida, foi escrito um pequeno carregador de imagem PGM do tipo binário (P5).

5 RESULTADOS

5.1 EXECUÇÃO

O projeto é compilado com o compilador GCC da toolchain oficial para embarcados da ARM (arm-eabi-none). O script `build.lua` provê uma interface CLI simples para compilar o projeto para a plataforma escolhida.

5.1.1 Injeção de falhas

Após compilar o projeto em um arquivo ELF com informação de debug é encontrar quais os símbolos seus endereços para relizar a injeção. Para a fase de injeção em locais conhecidos, os símbolos escolhidos foram:

- `kernel`: Filtro (kernel) do contexto de convolução usado
- `output_rows`: Linha(s) utilizadas como espaço de resultado para a convolução
- `output`: Resultado final

Para a fase de injeção com offsets do stack, foram sorteados uma variável local da função e um tamanho entre 4 a 32 bytes (1 a 4 palavras).

5.1.2 Demonstração do processo de Injeção

Com o microcontrolador conectado via ST-Link e com VirtualCOM por USB é inicializada uma sessão de depuração, o primeiro passo é inicializar a sessão com o breakpoint padrão na função `main` (Figura 21) que é definida pelo código de plataforma da HAL fornecida pelo fabricante do microcontrolador. É inserido um breakpoint na função `entrypoint` que o ponto de entrada real da aplicação que ocorre após o setup de periféricos e do RTOS.

Figura 21 – Inicializar sessão de depuração no ST-Link

```

RUN AND DEBUG  STLink ...  ...
BREAKPOINTS
main.cpp src  529
  ● main.cpp src
  ● main.cpp src
VARIABLES
Breakpoint no "main"
WATCH
Extensão da STM, permite ver periféricos e tasks do RTOS. Importante!
src > main.cpp > do_sobr
516
517     auto end = tick();
518     auto elapsed =
519     return {int(end - start)};
520 }
521 #else
522 static inline Pai
523     panic("TODO: 1");
524 }
525 #endif
526 static inline
527 void entrypoint()
528 {
529     main_arena = i
530     task_arena = i
531
532     #if defined(F1)
533         for(int i
534             | sleep
535

```

Fonte: Elaborada pelo autor

O programa é resumido Até alcançar o breakpoint, neste momento, é recomendado iniciar a conexão VirtualCOM para observar o sistema, aqui foi usado o microCOM pelo terminal do VSCode como demonstrado na Figura 22.

Figura 22 – Conectar via VirtualCOM

```

arc ~ /src/mf/ft_sched
○ > sudo microcom -f -p /dev/ttyACM0
connected to /dev/ttyACM0
[Press any character... Ctrl + C]

```

Fonte: Elaborada pelo autor

É selecionado um local para a injeção, neste caso será realizado uma injeção fixa nas linhas de output no exemplo com reexecução. Recomenda-se usar a funcionalidade de breakpoints condicionais para avançar para uma iteração desejada, na Figura 23 foi escolhido uma iteração específica.

Figura 23 – Posicionar Breakpoint

```

src > main.cpp > do_sobel_reexec
335
341
311     auto begin = tick_now();
312     auto row_arena = main_arena.make_sub(350 * 3);
313     Duration line_time_acc = {0};
314     Slice<u8> output_rows[3];
315     #ifdef FT_USE_CRC
316     auto row_crcs = make_slice<u32>(&main_arena, image.height);
317     ensure(row_crcs.data, "Failed to allocate CRC space");
318     #endif
319
320     for(i32 row = 0; row < image.height; row += 1){
321         output_rows[0] = make_slice<u8>(row_arena, image.width);
322         output_rows[1] = make_slice<u8>(row_arena, image.width);
323         output_rows[2] = make_slice<u8>(row_arena, image.width)
324
325         ensure(output_rows[0].data && output_rows[1].data && ou
326
327         Duration line_time = time_it([&](){
328             sobel_row(image, row, output_rows[0], row_arena);
329             sobel_row(image, row, output_rows[1], row_arena);
330             sobel_row(image, row, output_rows[2], row_arena);
331         });
332
333         line_time_acc = line_time_acc + line_time;
334
335         Click to add a breakpoint
336         Condition: row == 30
337
338         if(output_rows[cons].len != image.width){
339             panic("Row error\r\n");

```

Fonte: Elaborada pelo autor

Para realizar um upset, é selecionado o endereço do local escolhido e aplicado pelo console do debugger. Neste caso foi realizado um XOR com uma máscara de bits aleatória. A Figura 24 demonstra este processo.

Figura 24 – Causar upset

```

output_rows = [3]
> 0 = {...}
✓ 1 = {...}
< 2 = {...}
< data = 0x2000eb10 <main_arena_memory+144> "c\tt... 01
| *data = 99 'c'
| len = 120
333
334
335 i32 cons =
336 ensure(cons
337
338 if(output_r
339 panic("

> set {int}0x2000eb10 = {int}0x2000eb10 ^ 0xf27789f1

```

Fonte: Elaborada pelo autor

Após feitas as injecções desejadas, o programa é resumido e seu resultado inspecionado. Neste caso, como demonstrado na Figura 25, o erro foi mascarado via consenso majoritário.

Figura 25 – Resultado no VirtualCOM

```

Task Arena Size: 24576B
Address Width: 32-bit
Tick Frequency: 1000 Hz
RawTask size: 56

[Sobel Filter]
WARNING: Consensus without full agreement.

-----
Took: 4647ms
ms/row: 38ms
Task Arena: 8336B
Main Arena: 1126B
Image dimensions: 120x120
-----
Got EOF from port

```

Fonte: Elaborada pelo autor

Para observar o resultado, basta definir a macro DUMP_BITMAP e redirecionar a saída do microCOM para um arquivo PGM. Em um sistema POSIX seria algo como `microcom -f -p /dev/ttyACM0 > out.pgm`

Refazendo o processo acima com a saída redirecionada é possível observar o output da imagem com o filtro Sobel aplicado à direita, como desmonstrado na figura Figura 26.

Figura 26 – Resultado do Filtro Sobel



Fonte: (Original) <https://www.flickr.com/photos/evapro/539719106>

5.1.3 CRC em Detalhes

Esta seção foca especificamente na aplicação do CRC. Para essa demonstração em particular não será utilizado Reexecução ou TMR.

A Figura 27 demonstra estado inicial do cálculo do CRC de cada linha. Na janela superior direita estão os conteúdos da memória do vetor de CRCs e na parte inferior os dados de pixel da imagem output.

Na Figura 28 é demonstrado a imagem após ter as linhas filtradas e seus valores de checagem calculados. É introduzido uma corrupção na primeira linha do output (para facilitar visualização) na Figura 29.

Após a inserção, é realizado a checagem dos valores CRC com suas linhas correspondentes, os valores são modificados e viram um vetor que serve como uma máscara booleana para decidir quais linhas necessitam ser recomputadas, linhas marcadas com 0 indicam necessidade de recomputação pois a checagem falhou. A Figura 30 demonstra que a linha que sofreu um upset é corretamente identificada. Cores indicam a relação do valor e sua região de memória.

Figura 27 – Estado Inicial

The screenshot shows a debugger interface with several panes:

- RUN AND DEBUG**: Shows breakpoints at main.cpp:268, main.cpp:292, and main.cpp:555.
- BREAKPOINTS**: Lists the same breakpoints.
- WATCH**: Shows a watchpoint on row_crcs.data at 0x2000f3b8 and output.pixel_data.
- VARIABLES**: Shows local variables: ROW = -1515870811, image, output, and pixel_data.
- CALL STACK**: Shows the call stack: do_sobel_simple@0x0800b59a, main.cpp:268, operator()@0x0800b884, main.cpp:576.
- main.cpp 9+, M**: Code editor for main.cpp with breakpoints at lines 268, 292, and 555.
- main.c 9+**: Code editor for main.c.
- base.cpp**: Code editor for base.cpp.
- imz ...**: Code editor for imz ... (partially visible).
- memory.bin**: Memory dump for row_crcs.data at 0x2000f3b8. The first few bytes are 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F followed by a large sequence of zeros.
- memory.bin**: Memory dump for output.pixel_data at 0x2000b993. The first few bytes are 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F followed by a sequence of values: FF E9 B6 E4 FF F9 BC 8A 97 9B 93 71 4D 33 40 65, 55 1E 12 4F 65 38 11 31 67 5F 27 15 4C 70 65 55, etc.
- assembly**: Assembly code for the do_sobel_simple function, showing the logic for calculating CRCs and filtering rows.

Fonte: Elaborada pelo autor

Figura 28 – Convolução aplicada e CRCs de cada linha calculados

The screenshot shows a debugger interface with three main panes:

- Left pane (Breakpoints and Watch):** Displays breakpoints at line 292 and 555, and a watch point for `row_crcs.data` at address `0x2000f3b8`.
- Middle pane (Source code):** Shows the `main.cpp` and `main.c` files. The `main.c` file contains code for Sobel convolution and CRC calculations. A yellow box highlights the `printf` statement at line 292.
- Right pane (Memory dump):** Displays the memory dump for address `0x2000f3b8`, showing the calculated CRC values for each row.

```

RUN AND DEBUG SLink launch ...
BREAKPOINTS
main.cpp src 292
main.cpp src 555
WATCH
row_crcs.data = 0x2000f3b8 <main_arena_memory>
output.pixel_data = {...}
data = 0x2000b993 <image_output_storage+15> "\37...
len = 14400

VARIABLES
Local
image = {...}
output = {...}
begin = 3013
row_arena = 0x2000f240 <main_arena_memory>
line_time_acc = {...}
row_crcs = {...}
end = <optimized out>

main.cpp 9+, M x main.c 9+ ... memory.bin
src > main.cpp > do_sobel_simple
276
if(ok != image.width){
| panic("Row error");
}
auto dest = output.pixel_data;
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311

#define FT_USE_CRC
println("[CRC Output check]");
if(!crc_row_check(output, row_ci
for(i32 row = 0; row < image
if(row_crcs[row]) { conti
println("RECOMPUTE: %d\n");
auto output_row = make_s
i32 ok = 0;
Duration line_time = ti
| ok = sobel_row(imag
}););
line_time_acc = line_tin
if(ok != image.width){
| printf("Row error\r\n");
}
auto dest = output.pixe
0x2000f3b8 > memory.bin
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 00 7A 0E 88 C0 42 BE DB 00 4D 13 BC 00 66 F4 4A
00000010 00 B4 72 9F 60 F0 D4 B1 A0 1A 73 6E A0 54 DA 08
00000020 C0 F1 BE D2 E0 D5 2E 17 C0 55 D6 C9 80 5F 42 4A
00000030 20 FC 82 80 60 B1 9D C2 00 58 F4 4A 20 D6 0E 49
00000040 A0 74 3A 16 E0 1B E8 80 E0 FC BF A1 C0 59 86 67
00000050 40 29 EC 19 40 91 14 19 20 69 A3 AB C0 6C DD 2A
00000060 80 70 3E C1 E0 8C CD DA 40 F5 DB CF C0 71 39 D9
00000070 60 A1 51 2C 40 C9 F7 0A 60 19 B4 C9 80 58 00 39
00000080 20 56 BE 0F C0 31 4D A4 40 95 AD 9B 40 F6 6C D1
00000090 C0 80 42 A3 A0 18 6A AC A0 BB 93 D6 A0 B8 83 DA
000000A0 60 3C 49 A1 40 D2 67 EA A0 6B 7D 86 C0 9A C3 DE
000000B0 E0 E9 18 40 60 68 00 8C 40 BC 6E 18 20 F0 E4 43
000000C0 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
FF DA C7 FF FF E5 B8 96 A2 98 8C 6E 4C 38 6A 68
00000010 4E 2E 4C 83 53 37 23 79 7A 51 30 4B 90 77 63 63
00000020 8A 94 C4 FF FF E7 D7 FF FF F7 E3 E1 FF EC D6 C2
00000030 FF FF FC F3 E5 DE C6 90 68 69 62 7B DF DF BF F4
00000040 F5 CA 8C 5E 49 43 6E 73 96 CC A7 9C 9A A4 8E 6F
00000050 62 84 71 66 5D 81 73 60 51 7F 92 A5 B6 A5 A8 CE
00000060 E6 CD B2 8D 99 A3 B8 C3 B4 EB FF FD FD F0 D0 F0
00000070 FC FF FD FD FD FD BE E1 00 02 4E 12 00 00 00
00000080 15 02 00 00 00 00 37 18 02 06 38 57 10 28 37 6F
00000090 2C 00 00 38 60 15 00 06 29 22 42 6C 30 00 00 3A
000000A0 1A 00 00 06 2D 00 00 00 4D 31 00 00 00 00 00 00 00
000000B0 00 0B 00 1B 71 3A 00 39 1A 00 00 01 03 03 2D 1D

```

Fonte: Elaborada pelo autor

Figura 29 – Inserção de um valor upset

The screenshot shows a debugger interface with three main panes:

- Left pane (Breakpoints and Watch):** Displays breakpoints at line 292 and 555, and a watch point for `row_crcs.data` at address `0x2000f3b8`.
- Middle pane (Source code):** Shows the `main.cpp` and `main.c` files. The `main.c` file contains code for Sobel convolution and CRC calculations. A yellow box highlights the `printf` statement at line 292.
- Right pane (Memory dump):** Displays the memory dump for address `0x2000f3b8`, showing the calculated CRC values for each row. A red box highlights the value `69 69 69 FF` at memory location `0x2000f3b8`.

```

RUN AND DEBUG SLink launch ...
BREAKPOINTS
main.cpp src 292
main.cpp src 555
WATCH
row_crcs.data = 0x2000f3b8 <main_arena_memory+376>
output.pixel_data = {...}
data = 0x2000b993 <image_output_storage+15> "\37...
len = 14400

VARIABLES
Local
image = {...}
output = {...}
begin = 3013
row_arena = 0x2000f240 <main_arena_memory>
line_time_acc = {...}
row_crcs = {...}
end = <optimized out>

CALL STACK Paused on breakpoint

main.cpp 9+, M x main.c 9+ ... memory.bin
src > main.cpp > do_sobel_simple
276
if(ok != image.width){
| panic("Row error");
}
auto dest = output.pixel_data;
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311

#define FT_USE_CRC
println("[CRC Output check]");
if(!crc_row_check(output, row_ci
for(i32 row = 0; row < image
if(row_crcs[row]) { conti
println("RECOMPUTE: %d\n");
auto output_row = make_s
i32 ok = 0;
Duration line_time = ti
| ok = sobel_row(imag
}););
line_time_acc = line_tin
if(ok != image.width){
| printf("Row error\r\n");
}
auto dest = output.pixe
0x2000f3b8 > memory.bin
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 00 7A 0E 88 C0 42 BE DB 00 4D 13 BC 00 66 F4 4A
00000010 00 B4 72 9F 60 F0 D4 B1 A0 1A 73 6E A0 54 DA 08
00000020 C0 F1 BE D2 E0 D5 2E 17 C0 55 D6 C9 80 5F 42 4A
00000030 20 FC 82 80 60 B1 9D C2 00 58 F4 4A 20 D6 0E 49
00000040 A0 74 3A 16 E0 1B E8 80 E0 FC BF A1 C0 59 86 67
00000050 40 29 EC 19 40 91 14 19 20 69 A3 AB C0 6C DD 2A
00000060 80 70 3E C1 E0 8C CD DA 40 F5 DB CF C0 71 39 D9
00000070 60 A1 51 2C 40 C9 F7 0A 60 19 B4 C9 80 58 00 39
00000080 20 56 BE 0F C0 31 4D A4 40 95 AD 9B 40 F6 6C D1
00000090 C0 80 42 A3 A0 18 6A AC A0 BB 93 D6 A0 B8 83 DA
000000A0 60 3C 49 A1 40 D2 67 EA A0 6B 7D 86 C0 9A C3 DE
000000B0 E0 E9 18 40 60 68 00 8C 40 BC 6E 18 20 F0 E4 43
000000C0 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
FF DA C7 FF FF E5 B8 96 A2 98 8C 6E 4C 38 6A 68
00000010 4E 2E 4C 83 53 37 23 79 7A 51 30 4B 90 77 63 63
00000020 8A 94 C4 FF FF E7 D7 FF FF F7 E3 E1 FF EC D6 C2
00000030 FF FF FC F3 E5 DE C6 90 68 69 62 7B DF DF BF F4
00000040 F5 CA 8C 5E 49 43 6E 73 96 CC A7 9C 9A A4 8E 6F
00000050 62 84 71 66 5D 81 73 60 51 7F 92 A5 B6 A5 A8 CE
00000060 E6 CD B2 8D 99 A3 B8 C3 B4 EB FF FD FD F0 D0 F0
00000070 FC FF FD FD FD FD BE E1 00 02 4E 12 00 00 00
00000080 15 02 00 00 00 00 37 18 02 06 38 57 10 28 37 6F
00000090 2C 00 00 38 60 15 00 06 29 22 42 6C 30 00 00 3A
000000A0 1A 00 00 06 2D 00 00 00 4D 31 00 00 00 00 00 00 00
000000B0 69 69 69 FF E5 B8 96 A2 98 8C 6E 4C 38 6A 68
00000010 4E 2E 4C 83 53 37 23 79 7A 51 30 4B 90 77 63 63
00000020 8A 94 C4 FF FF E7 D7 FF FF F7 E3 E1 FF EC D6 C2
00000030 FF FF FC F3 E5 DE C6 90 68 69 62 7B DF DF BF F4
00000040 F5 CA 8C 5E 49 43 6E 73 96 CC A7 9C 9A A4 8E 6F
00000050 62 84 71 66 5D 81 73 60 51 7F 92 A5 B6 A5 A8 CE
00000060 E6 CD B2 8D 99 A3 B8 C3 B4 EB FF FD FD F0 D0 F0
00000070 FC FF FD FD FD FD BE E1 00 02 4E 12 00 00 00
00000080 15 02 00 00 00 00 37 18 02 06 38 57 10 28 37 6F
00000090 2C 00 00 38 60 15 00 06 29 22 42 6C 30 00 00 3A
000000A0 1A 00 00 06 2D 00 00 00 4D 31 00 00 00 00 00 00 00
000000B0 00 0B 00 1B 71 3A 00 39 1A 00 00 01 03 03 2D 1D

```

Fonte: Elaborada pelo autor

Figura 30 – Comparação dos CRCs resulta em máscara de execução

The screenshot shows a debugger interface with several windows:

- RUN AND DEBUG**: Shows breakpoints at main.cpp:292, main.cpp:296, and main.cpp:555.
- BREAKPOINTS**: Lists the same breakpoints.
- WATCH**: Shows variables: row_crcs.data = 0x2000f3b8, output.pixel_data = {...}, and data = 0x2000b993.
- VARIABLES**: Shows local variables: output_row = {...}, ok = 53693384, dest = {...}, row = 0, image = {...}, and output = {...}.
- CALL STACK**: Shows the call stack: do_sobel_simple@0x0000b766, main.cpp:296, main.cpp:313.
- main.cpp 9+, M**: Source code for main.cpp, showing the do_sobel_simple function.
- main.c 9+**: Source code for main.c, showing the main function.
- base.cpp**: Source code for base.cpp.
- im ...**: Source code for im.c.
- memory.bin**: Memory dump showing two blocks of data at addresses 0x2000f3b8 and 0x2000b993. The first block has a CRC error (highlighted in red). The second block has a CRC check (highlighted in green).

Fonte: Elaborada pelo autor

5.1.4 Consenso (TMR) em Detalhes

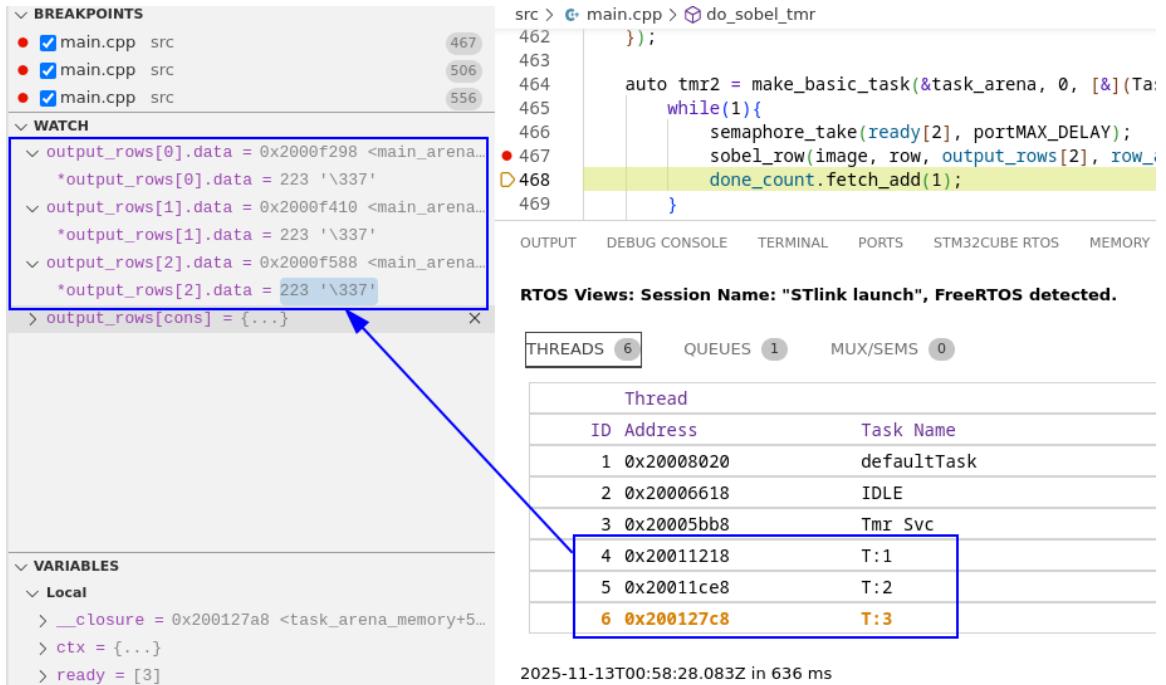
Nas aplicações de reexecução e TMR é necessário aplicar um consenso majoritário dentro as 3 execuções, esta seção aborda o funcionamento do consenso em detalhes. Será utilizado TMR como um exemplo, pois é mais complexo e requer sincronização e o funcionamento do algoritmo majoritário é o mesmo aplicado na aplicação de Reexecução.

Na Figura 31 é observado o estado inicial, as 3 tarefas replicadas são criadas e aguardam seus semáforos binários para iniciar um novo ciclo de trabalho.

Uma corrupção de um NOT subtraído por 1 é realizada na segunda tarefa (índice 1), a discrepância entre o conteúdo das linhas das tarefas é demonstrado em Figura 32.

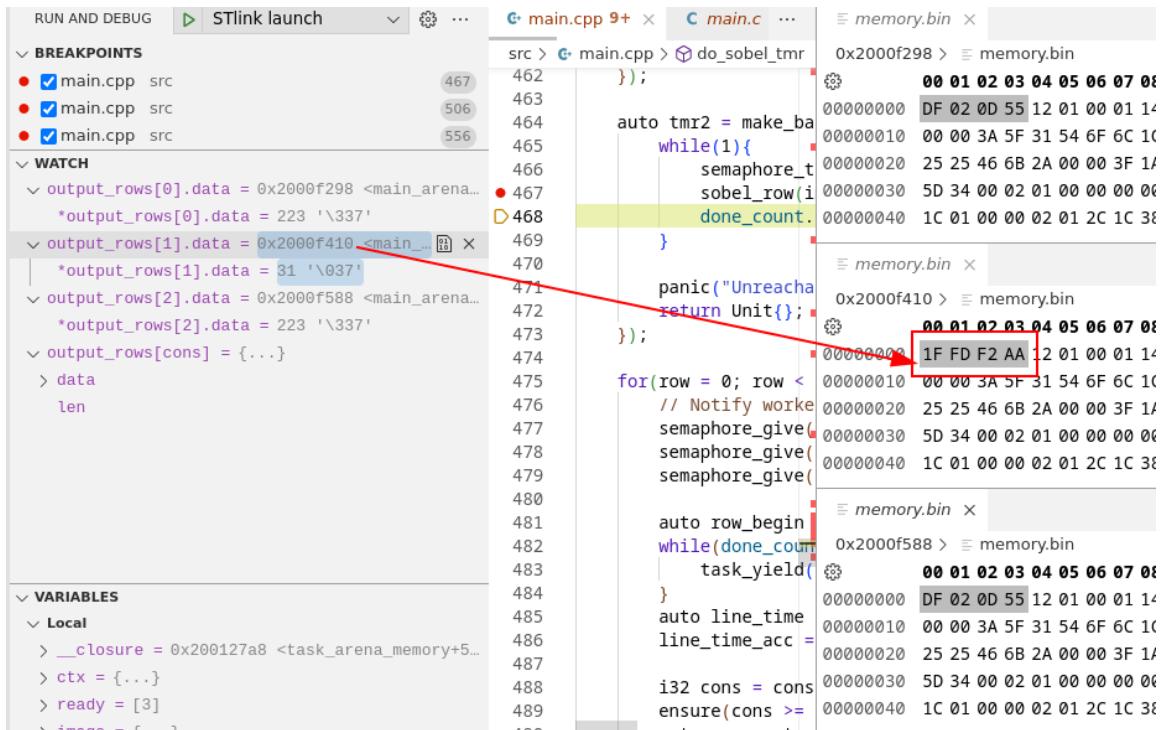
Na seguinte iteração (Figura 33), a função de consenso detecta e mascara o erro, selecionado o resultado majoritário da terceira tarefa (índice 2) que está em concordância com a primeira.

Figura 31 – Estado inicial - 3 tarefas



Fonte: Elaborada pelo autor

Figura 32 – Upset inserido na segunda tarefa



Fonte: Elaborada pelo autor

Figura 33 – Consenso majoritário seleciona opção que mascara o upset

```

output_rows[0].data = <-var-create: unable t...
output_rows[1].data = <-var-create: unable t...
output_rows[2].data = <-var-create: unable t...
> output_rows[cons] = {...}           x
majority = 2

auto end = tick_now();
return tick_diff(end, begin);

__attribute__((noinline))
i32 consensus(Slice<u8> a, Slice<u8> b, Slice<u8> c){
    volatile u32 ab = a.equals(b);
    volatile u32 bc = b.equals(c);
    volatile u32 ca = c.equals(a);

    volatile u32 agreement = ab + bc + ca;

    int majority = -1;
    if(ab) majority = 0;
    if(bc) majority = 1;
    if(ca) majority = 2;

    if(agreement != 3){           yellow box
        println("INFO: Consensus without full agreement.");
    }
}

return majority;

```

Fonte: Elaborada pelo autor

O processo de consenso é idêntico para reexecução, porém ao invés de 3 estados provenientes de tarefas distintas que sincronizam, é realizado uma aplicação à um buffer com os resultados das 3 execuções sucessivas.

5.2 DEPENDABILIDADE E PERFORMANCE

Após repetir o processo de injeção da seção 5.1 para as combinações de técnicas, foi possível coletar os dados representados na Quadro 10. O pico de memória da arena para o armazenamento das tarefas é denominado M_{task} , o pico da arena de memória adicional é denominado M_{extra} . As variáveis T_{total} e T_{linha} representam o tempo total de execução e o tempo médio por linha, respectivamente. A variável D indica o número de detecções de erros bem-sucedidas.

Quadro 10 – Resultados sem injeção

Técnicas	T_{total}	T_{linha}	M_{task}	M_{extra}	Detecção	Resultado
N/A	1542ms	12ms	2768B	374B	0	OK
CRC32	1549ms	12ms	2768B	856B	0	OK
Reexec	4631ms	38ms	2768B	1074B	0	OK
Reexec + CRC32	4635ms	38ms	2768B	1556B	0	OK
TMR	4644ms	38ms	8336B	1126B	0	OK
TMR + CRC32	4651ms	38ms	8336B	1608B	0	OK

Os resultados neste caso servem para estabelecer uma linha base das latências envolvidas. O método de CRC32 é relativamente rápido, adicionando apenas alguns milisegundos e um pouco menos de 500 bytes para manter o CRC de cada linha da imagem juntamente com buffers temporários para processamento.

As técnicas de reexecução e TMR possuem um grande impacto no tempo de execução de aproximamente 3.02x, já a técnica de TMR em particular necessita do triplo de memória de tarefas para manter o stack de todas suas sub-tarefas assim como triplo de memória de trabalho para garantir o isolamento entre as sub-tarefas. O tempo de execução entre a Reexecução e TMR não é muito grande, o sistema foi executado sem multiprocessamento simétrico, portanto não foi possível observar as vantagens potenciais causadas pelo uso de paralelismo. O custo das trocas de contexto do kernel e das sincronizações pode ser observado no valor das tarefas TMR serem tipicamente apenas alguns millisegundos mais lentas que as versões de reexecução.

Injetando falhas nos filtros e no output como descrito no Capítulo 3, foi possível observar o comportamento das técnicas na presença de falhas que não envolvam corrupção de seu stack frame.

Quadro 11 – Resultados com Injeção em variáveis fixas

Técni-cas	T_{total}	T_{linha}	M_{task}	M_{extra}	D	Resultado	Observação
N/A	1544ms	12ms	2768B	374B	0	Corrupção	N/A
CRC32	1565ms	12ms	2768B	856B	1	OK	Corrupção detectada no output apenas
Reexec	4634ms	38ms	2768B	1074B	1	Corrupção	Mascarou corrupção na linha, mas não no output
Ree-xec + CRC32	4676ms	38ms	2768B	1556B	2	OK	Corrupção mascarada e output recomputado
TMR	4647ms	38ms	8336B	1126B	1	Corrupção	Mascarou corrupção na linha, mas não no output
TMR + CRC32	4654ms	38ms	8336B	1608B	2	OK	Corrupção mascarada e output recomputado

No Quadro 11 é possível observar um padrão, a correção de erro ao final (output) só pode ser concluída no casos em que se possui a presença de um algoritmo de checagem, as técnicas de TMR e Reexecução conseguiram com sucesso mascarar a falha local de uma linha da corrupção com um uma função de consenso majoritário simples, porém uma vez que o valor já tenha sido calculado não é mais possível validar sua integridade. Para cada técnica foi escolhida uma variável do stack frame e aplicado um upset a partir de um offset e um tamanho. A notação $V + O : N$ indica que foi aplicado uma corrupção de tamanho N a partir do endereço da variável V deslocado por O bytes.

Quadro 12 – Resultados com Injeção no Stack Frame

Técni-cas	T_{total}	T_{linha}	M_{task}	M_{extra}	D	Resultado	Observação	Var
N/A	1100ms	60ms	2768B	374B	1	Erro	Deadline de li-nha expirada	row 0:8
CRC32	1100ms	60ms	2768B	856B	1	Erro	Deadline de li-nha expirada	row 0:8
Reexec	<100ms	0ms	2768B	1074B	1	Erro	Assert: row.width != output.width	out-put_rows 4:12
Ree-exec + CRC32	2265ms	38ms	2768B	1556B	1	Erro	Hard Fault do FreeRTOS	out-put_rows 4:12
TMR	4625ms	38ms	8336B	1126B	0	OK	Evitou corrup-ção através de indireção	tmr0 0:4
TMR + CRC32	4628ms	38ms	8336B	1608B	0	OK	Evitou corrup-ção através de indireção	tmr0 0:4

5.3 IMPACTO DE FALHAS NO OUTPUT

Foram selecionadas execuções que resultam em um output que possa ser visualizado para demonstrar o efeito dos upsets no resultado final. A Figura 26 demonstra o output desejado, com a imagem de um gato com um filtro Sobel aplicado. A Figura 34 mostra o impacto de um upset causado no filtro (kernel) da convolução. Na Figura 35 a corrupção é bem mais sutil, manifestando-se como uma pequena linha horizontal em $y = 6$. Por fim, a figura Figura 36 demonstra o efeito da convolução sendo interrompida devido à um assert, deixando o resultado incompleto.

Figura 34 – Imagem com o parâmetro de filtro corrompido



Fonte: Elaborada pelo autor

Figura 35 – Imagem com Corrupção sutil



Fonte: Elaborada pelo autor

Figura 36 – Imagem com filtro Sobel interrompido



Fonte: Elaborada pelo autor

5.4 VERIFICAÇÃO

Quadro 13 – Requisitos funcionais

Requisito	Descrição	Validação
RF01	Implementação de todos os algoritmos descritos na subseção 3.4.2	Os algoritmos foram implementados e utilizados para os experimentos no Capítulo 5
RF02	Configuração do mecanismo de tolerância, prioridade e prazo de execução da tarefa	A configuração do mecanismo é feita de forma imperativa, já o prazo, prioridade e tamanho da pilha são feitos declarativamente no momento de instanciação da tarefa
RF03	Cumprimento do prazo estipulado no momento de criação da tarefa caso não exista presença de falhas	Validado com os testes sem injeção de falha
RF04	Dependabilidade superior à versão do sistema sem técnicas	O sistema foi capaz de mascarar algumas falhas e detectar outras. Portanto possui uma dependabilidade superior
RF05	Monitoramento do número de falhas detectadas e violações de prazos	Implementado em DeadlineWatcher em conjunto com o IWDG. Foi capaz de detectar violações

Os requisitos não funcionais também foram analisados conforme o Quadro 14.

Quadro 14 – Validação dos Requisitos não funcionais

Requisito	Descrição	Validação
RNF01	O consumo de memória deve ser pré-determinado em tempo de compilação ou na inicialização do sistema	O pico de memória é estável e determinado no startup da aplicação
RNF02	A interface deve ser construída sobre o escalonador preemptivo do FreeRTOS	Os mecanismos de tarefa são implementados sobre o escalonador
RNF03	Deve ser compatível com arquitetura ARMv7M ou ARMv8M	O projeto compila e executa na arquitetura
RNF04	Implementação realizada em C++ (versão 20 ou acima)	O projeto compila na versão C++20 com GCC e Clang
RNF05	Código Fonte disponível sob licença permissiva	Validado parcialmente. O módulo de USB_DEVICE não é compatível com uma licença BSD2 porém não é essencial para a demonstração das técnicas. O resto do código cumpre este requisito.

5.5 DISCUSSÃO DOS RESULTADOS

A aplicação das técnicas resultou em um impacto positivo na dependabilidade do sistema, em particular técnicas de execução (TMR e Reexecução) em combinação com a checagem por CRC32 demonstraram a melhor dependabilidade.

O custo das técnicas de execução é significativo, triplicando o tempo computacional no caso de um sistema sem multi processamento simétrico e adicionando um overhead de memória alto (mais de 3x) no caso da aplicação de TMR. Para o sistema apresentado, a reexecução em combinação com o CRC32 possui bons resultados dado seu consumo menor de memória quando comparado à técnica de TMR.

Durante injeções aleatórias no stack frame, a redundância de estado adicional do TMR contribuiu para evitar um erro fatal, indicando que talvez outros padrões de falha sejam melhor capturados e mascarados pela técnica de TMR, justificando seu custo adicional de memória. É também interessante que trabalhos futuros avaliem a técnica de TMR no contexto de multiprocessamento simétrico para analisar o potencial impacto positivo da execução paralela, dado que a técnica de TMR possui uma grande vantagem ao prover maior isolamento de estado mutável.

A validação de deadline não foi ativa com frequência durante os testes, mesmo nos casos de injeções aleatórias um erro fatal via assert foi capaz de interromper a execução primeiro. Isso não significa que os testes de deadline não possuiram nenhum impacto positivo, pois foi possível detectar um deadlock de sincronização durante o desenvolvimento mais cedo graças a um disparo prematuro, porém os resultados apresentados não foram capazes de causar uma mudança significativa no tempo de execução que ativasse o monitor. Injeções de falhas que alterem mais o fluxo de controle, ou a aplicação de um programa de teste com características temporais mais sensíveis à upsets de memória talvez seja capaz de gerar mais oportunidades para a contribuição desta técnica.

6 CONSIDERAÇÕES FINAIS

Sistemas embarcados são sistemas computacionais com propósito específico e são tipicamente parte da qualidade de serviço de um todo. Tendo essas características em mente, é importante que o custo dos recursos necessários para a execução correta de um projeto sejam considerados para a viabilidade do produto final com o máximo de funcionalidade possível com o mínimo de custo de produção e custo energético.

O uso de sistemas operacionais de tempo real em sistemas embarcados é popular por fornecer garantias em relação à prioridade de tarefas assim como permitir um modelo de aplicação assíncrono orientado à eventos.

A dependabilidade é uma propriedade geral do sistema que combina diversos critérios (RAMS) e o aumento da dependabilidade em sistemas de natureza crítica é importante para a segurança dos usuários, outros sistemas também podem se beneficiar ao oferecer uma melhor qualidade de serviço mesmo na presença de adversidades.

Com o fim de melhorar a propriedade de dependabilidade de um sistema operacional de tempo real, foram exploradas técnicas de execução, monitoramento de prazo e de checagem de integridade em cenários de falhas transientes.

Para analisar o impacto da aplicação das técnicas, foi projetado uma aplicação de teste e uma campanha de injeção de falhas. As técnicas foram implementadas na linguagem C++20 e executadas no microcontrolador STM32F411CEU6 com o sistema operacional FreeRTOS e tiveram suas métricas de performance coletadas.

As técnicas demonstraram um efeito positivo da dependabilidade com diferentes custos de memória e tempo de execução. As técnicas que alteram o fluxo do programa (Reexecução e TMR) foram capazes de mascarar certas falhas em tempo real, já o uso de CRC foi capaz de permitir recomputar dados após um upset ter alterado um resultado. O monitoramento de deadlines foi importante para detectar violações do critério temporal assim como auxiliou durante o processo de desenvolvimento para o diagnóstico de erros lógicos.

Para elaboração de trabalhos futuros, é interessante que a presença de múltiplos núcleos em um sistema com multi processamento simétrico, assim como expandir a detecção de erros para incluir mecanismos de análise de fluxo de controle a nível de compilador. A campanha de injeção de falhas também poderia ser melhor automatizada com uma ferramenta de instrumentalização como o PyOCD.

REFERÊNCIAS

- ADACORE. 2025. Disponível em: <https://docs.adacore.com/spark2014-docs/html/ug/en/source/subprogram_contracts.html>.
- AFONSO, F. et al. Application-level fault tolerance in real-time embedded systems. In: **2008 International Symposium on Industrial Embedded Systems**. [S.l.: s.n.], 2008. p. 126–133.
- ARM. **ARM Compiler v5.06 for uVision armcc User Guide**. [S.l.], 2024. Disponível em: <<https://developer.arm.com/documentation/dui0375/g/Compiler-Coding-Practices/Loop-unrolling-in-C-code>>.
- ARMSTRONG, J. **Making reliable distributed systems in the presence of software errors**. Tese (Doutorado) — Swedish Royal Institute of Technology, 2003.
- BOS, H.; TANENBAUM, A. S. **Modern Operating Systems**. 5. ed. [S.l.]: Pearson, 2023.
- FLEURY, R. **Untangling Lifetimes: The Arena Allocator**. 2022. Disponível em: <<https://www.rfleury.com/p/untangling-lifetimes-the-arena-allocator>>.
- FREEMAN, O. J. **ITAR PCB Compliance: Regulations, Challenges, and Solutions**. Altium 365, 2025. Disponível em: <<https://resources.altium365.com/p/itar-pcb-compliance-regulations-challenges-solutions>>.
- GALVIN, P. B.; GAGNE, G.; SILBERSCHATZ, A. **Operating System Concepts**. 10. ed. [S.l.]: Wiley, 2018.
- GOBATTO, L. R. et al. Reliability assessment of arm cortex-m processors under heavy ions and emulated fault injection. 2024.
- HENNESSY, J. L.; PATTERSON, D. A. **Computer Organization and Design: The Hardware Software Interface**. [S.l.]: Morgan-Kaufmann, 2020.
- HOLZMANN, G. J. The power of 10: Rules for developing safety-critical code. 2006.
- IEEE. Ieee standard classification for software anomalies. **IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)**, p. 1–23, 2010.
- IZOSIMOV, V. **Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems**. Dissertação (Mestrado) — Linköping University, 2006.
- IZOSIMOV, V. et al. Scheduling of fault-tolerant embedded systems with soft and hard timing constraints. In: **2008 Design, Automation and Test in Europe**. [S.l.: s.n.], 2008. p. 915–920.
- KRISHNA, C. M.; KOREN, I. **Fault-Tolerant Systems**. 10. ed. [S.l.]: Morgan-Kaufmann, 2020.
- MAGAGNIN, N. M. **Técnica de confiabilidade em nível de sistema operacional para a arquitetura RISC-V**. 65 p. Monografia (Trabalho Técnico-científico de Conclusão de Curso (Graduação em Engenharia de Computação)) — Universidade Do Vale Do Itajaí Escola Politécnica, Itajaí, SC, 2023.
- MAMONE, D. **Fault injection techniques for Real-Time Operating Systems**. Tese (Doutorado) — Politecnico di Torino, 2018.

MEZGER, B. W. et al. A basic microkernel for the risc-v instruction set architecture. **Anais do Computer on The Beach**, p. 57–63, 2021.

NUSSBAUM, B.; BERG, G. Cybersecurity implications of commercial off the shelf (cots) equipment in space infrastructure. 2020.

SMITH, S. W. **The Scientist and Engineer's Guide to Digital Signal Processing**. 2. ed. [S.l.]: California technical Publishing, 1999.

SOLOUKI, M. A.; ANGIZI, S.; VIOLANTE, M. Dependability in embedded systems: A survey of fault tolerance methods and software-based mitigation techniques. 2024.

STMICROELECTRONICS. **ST-LINK/V2**. 2025. Disponível em: <<https://www.st.com/en/development-tools/st-link-v2.html>>.

STMICROELECTRONICS. **STM32Cube IDE**. 2025. Disponível em: <<https://www.st.com/en/development-tools/stm32cubeide.html>>.

STMICROELECTRONICS. **STM32F103C8T6**. 2025. Disponível em: <<https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html>>.

TIGERBEETLE. **TigerBeetle Concepts: Safety**. 2024. Disponível em: <<https://docs.tigerbeetle.com/concepts/safety/>>.

TORCZON, L.; COOPER, K. **Engineering A Compiler**. 2. ed. San Francisco, CA, USA: Morgan Kaufmann, 2007.