1. INTRODUÇÃO

Tolerância à falhas (TF) é a capacidade de um sistema computacional continuar oferecendo qualidade de serviço mesmo na presença de defeitos e interferências inesperadas, falhas podem ser encontradas comumente nos contextos de sistemas distribuídos, onde os canais de comunicação podem sofrer degradação ou total inoperabilidade devido à interferência eletromagnética, falta de energia e eventos climáticos. Sistemas embarcados encontram os mesmos problemas ao utilizar um canal com ruído ou instável, mas também podem dados em memória ou registradores diretamente afetados por causas externas como radiação ionizante, flutuações súbitas de temperatura ou voltagem e colisões físicas. A existência desses fenômenos necessitam que dispositivos estejam preparados, especialmente em aplicações aero-espaciais(que necessitam ser confiáveis e tolerar seu ambiente volátil).

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**, seja de um sistema operacional ou de um runtime, é crucial na execução concorrente de diversas tarefas, sendo então um candidato interessante para aprimorar sua resiliência com foco em reduzir desperdício dos nós computacionais. O processo de detecção das falhas e seu impacto no grafo de execução assim como nas métricas quantitativas de tempo de CPU e uso de memória portanto deve ser considerado, particularmente no contexto de escalonamento, pois a reação rápida e correta às falhas requer previamente a detecção e elaboração das rotinas de escalonamento de forma adequada.

Este trabalho visa portanto fazer uma análise do impacto de diferentes técnicas de detecção durante o escalonamento de tarefas e seu impacto na performance e no fluxo de execução do sistema, para que se possa melhor compreender e evidenciar os custos e benefícios ao tornar um sistema mais resiliente.

1.1. Problematização

A presença de sistemas embarcados em contextos críticos como na exploração espacial, automobilística e tecnologia médica, assim como a ubiquidade de dispositivos móveis e de baixo consumo energético no mercado consumidor (Celulares, Notebooks, equipamentos IoT) e a existência do mercado de cloud e computação distribuída, entende-se que manter um 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.

1.2. Formulação do Problema

O custo de utilizar técnicas de tolerância é sensível ao contexto da aplicação e ao nível de tolerância desejado, para desenvolvedores de software o custo e as limitações impostas para que o sistema

seja mais resiliente não são imediatamente evidentes, a criação de software resiliente se torna uma tarefa potencialmente mais difícil do que o necessário.

> Um fator político que dificulta a aquisição de dispositivos com tolerância é a regulação 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 aumenta mais ainda a necessidade de compradores de outros países adquirirem alternativas comerciais mais comuns (COTS).

1.3. 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 resiliência do sistema, particularmente no contexto de sistemas de tempo real, dado que algoritmos que consigam satisfazer a restrição de tempo real também podem ser utilizados em outros contextos com restrições temporais mais relaxadas.

1.4. Objetivos

1.4.1. Objetivo Geral

Explorar o uso de técnicas de escalonamento de tempo real com detecção de erros.

1.4.2. Objetivos Específicos

- Identificar métodos de detecção de erro de execução em nível de software:
  - CRCs para mensagens
  - Heartbeat simples (timer)
  - Heartbeat com Proof of Work
  - Replicação
  - Asserts
- Aplicar como prova de conceito em um RTOS a técnica selecionada
- Avaliar por meio de métricas a técnica durante a execução em um RTOS
- Avaliar por meio de métricas a técnica em nível de memória em um RTOS

1.5. 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.

Uma implementação dos algoritmos na forma de uma prova de conceito será realizada dentro de um contexto com um host para facilitar prototipação e realizar uma pré análise e validar se a implementação está correta.

1.5.1. Métodos

1.5.2. Materiais

1.6. Estrutura do Trabalho

2. Fundamentação teórica

2.1. Falhas e Tolerância

Uma **falha** pode ser compreendida como um evento fora do controle do sistema que provoca uma degradação na sua qualidade de serviço, seja ao afetar a validade dos resultados ou com uma degradação na forma de aumento de latência. Falhas podem ser classificadas em 3 grupos referentes ao seu padrão de ocorrência:

- Falhas **Transientes**: Ocorrem aleatoriamente e possuem um impacto temporário.
- Falhas **Intermitentes**: Assim como as transientes possuem impacto temporário, porém re-ocorrem periodicamente.
- Falhas **Permanentes**: Causam uma degradação permanente no sistema da qual não pode ser recuperada, potencialmente necessitando de intervenção externa.

Existem diversas fontes de falhas que podem afetar um sistema, exemplos incluem: Radiação ionizante, Interferência Eletromagnética, Harmônicas, Impacto Físico, Oscilação elétrica (picos de tensão e/ou corrente).

Para que o sistema possa ser **Tolerante à Falhas**, isto é, ser capaz de manter uma qualidade de serviço aceitável mesmo na presença de falhas são necessários 2 principais mecanismos:

1. **Detecção de Falhas**: Capacidade de perceber a ocorrência de uma falha e executar a rotina de tratamento. Métodos comuns de detecção incluem: Bits de paridade, Funções hash, Sinais heartbeat e Limites de tempo (**timeouts** ou **deadlines**).
2. **Tratamento de Falhas**: Capacidade de reagir as falhas, com uma correção, re-execução ou rotina de mitigação. Falhas permanentes podem necessitar de um desligamento gracioso do sistema ou reorganização para manter o máximo de qualidade de serviço possível.

A detecção e o tratamento podem ser implementados em hardware ou em software, implementações em hardware conseguem fazer garantias físicas mais fortes com melhor revestimento e redundância implementada diretamente no circuito, e prover transparência de execução para o programador, a desvantagem é custo elevado de espaço no silício possível degradação de performance geral e menor flexibilidade. O processo de tornar o design e a implementação de um hardware com estas características é chamado de **hardening**.

Implementações em software não são capazes de fornecer todas as garantias fortes do hardware, em contrapartida, não ocupam espaço extra no chip e são mais flexíveis, com software é possível

implementar lógica de detecção recuperação e estruturas de dados mais complexas, e até mesmo realizar atualizações remotas com o sistema ativo (**live patching**).

Para que um sistema possa ser resiliente à falhas, ambas soluções de hardware e software precisam ser consideradas, é possível utilizar hardware com **hardening** para um núcleo que realiza atividades críticas, e delegar núcleos menos protegidos para atividades em que o tratamento em software é suficiente. Balancear a troca de espaço em chip, uso de memória, flexibilidade de implementação, tempo de execução, vazão de dados e garantias de transparência é indispensável para a criação de um sistema que seja resiliente à falhas e que forneça uma boa qualidade de serviço pelo menor custo possível.

2.2. Mecanismos de Detecção

Os mecanismos de **detecção**, permitem que um sistema detecte uma inconsistência em seus dados, causada por falha externa ou por erro lógico de outra parte no caso de sistemas distribuídos. Os mecanismos de detecção são essenciais para a tolerância à falhas. Ao detectar uma falha o sistema deve tomar uma ação corretiva para o tratamento da falha, mecanismos de tratamento serão abordados posteriormente.

2.2.1. CRC (Cyclic Redundancy Check)

Os CRCs são códigos de detecção de erro comumente utilizados em redes de computador e armazenamento não volátil para detectar falhas. Para cada segmento de dado é concatenado um valor (denominado **check value** ou simplesmente o valor CRC) que é calculado com base no resto da divisão de um polinômio previamente acordado entre remetente e destinatário (chamado de "polinômio gerador").

Ao receber o segmento, o receptor calcula seu próprio valor CRC com base nos dados do segmento (sem incluir o CRC do destinatário), caso ocorra diferença entre os CRCs isso indica a ocorrência de um erro. CRCs são comumente utilizados devido à serem simples de implementar, ocuparem pouco espaço adicional no segmento e serem resilientes à "**burst errors**", falhas transientes que alteram uma região de bits próximos.

2.2.2. Heartbeat signals

É possível determinar se uma falha ocorreu com um nó de execução através de um critério temporal, os sinais de **heartbeat** ("batimento cardíaco") são sinais periódicos para garantir se um nó computacional está ativo. Basta enviar um sinal simples e verificar se uma resposta correta chega em um tempo pré determinado. Sinais heartbeat são extremamente baratos porém não garantem um detecção ou correção de erro mais granular, portanto são usados como um complemento para detectar falhas de forma concorrente aos métodos mais robustos.

O custo de memória de um sinal heartbeat tende a ser pequeno, porém possui o custo temporal de tolerância limite no pior caso e o custo da viagem ida e volta no melhor caso. Este método é aplicado

em datacenters, também chamado de “health signal” ou “health check”, o sinal e sua resposta desejada podem conter outros metadados para análise de falhas, caso desejado.

Também é possível usar os próprios prazos de execução como um mecanismo de detecção, porém isso pode não ser viável em sistema com prazos curtos, especialmente quando se opera em um contexto hard real time.

2.2.3. Pré e Pós condições e asserts

A utilização de asserts é um mecanismo simples que é particularmente útil, um assert trata-se de checar se uma condição é verdadeira, caso não seja, o programa é interrompido e entra um estado de pânico. Utilizar asserts automáticos na entrada e saída de funções é denominado pré/pós-condições. Asserts não previnem erros do hardware ou geram reexecuções, mas tratam-se de um mecanismo de uso extremamente fácil que pode ser inserido pelos desenvolvedores para detectar falhas de design cedo, o uso de asserts podem detectar um defeito externo, mas por serem mecanismos exclusivamente de fluxo de controle, não são muito robustos em suas garantias, mas ainda assim, seu custo baixo e fácil inserção/ deleção os fazem um mecanismo que não deve ser ignorado.

2.3. 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.

2.3.1. 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.

Uma técnica de redundância comum é o uso de TMR (Triple Modular Redundancy) onde essencialmente a tarefa é executada 3 vezes em paralelo, e uma porta de consenso utiliza a resposta gerada por pelo menos 2 das unidades. O uso de TMR é elegante em sua simplicidade e consegue atingir um bom grau de resiliência, porém com o custo adicional de triplicar a superfície.

Sistemas distribuídos também podem aproveitar de sua redundância natural por serem sistemas com múltiplos nós computacionais, falhas transientes 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, o uso de sistemas capazes de auto reparo é vital para a existência de telecomunicação em larga escala e computação em nuvem.

2.3.2. 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.

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 improvável que múltiplas falhas ocorram repetidamente em **N** execuções)

2.3.3. Correção de Erro

Existem também algoritmos que permitem detectar e corrigir erros dentro de um payload, em troca de um custo de espaço e tempo para a detecção, dentro da família de algoritmos que possibilitam detecção e correção, são encontrados os códigos como os de: Reed-Solomon, Turbo Codes, LDPCs.

Este trabalho não abordará algoritmos de correção de forma aprofundada pois foge do escopo de foco nas técnicas de escalonamento (execução) e detecção, mas se trata de um tópico importante que complementa qualquer implementação de sistemas resilientes particularmente no processo de envio e recebimento de mensagens.

2.4. 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. Um exemplo de um sistema embarcado são microcontroladores encontrados em dispositivos como mouses, teclados e eletrodomésticos.

Limitação de recursos: Um corolário da natureza especialista destes sistemas, é que recursos alocados para o sistema são definidos previamente. No caso de microcontroladores tanto o poder computacional quanto a disponibilidade de memória são restritas. Importante notar que existem sistemas embarcados com acesso maior à recursos, como equipamentos de rede e hardware aceleradores que podem ter acesso a quantias maiores de poder computacional ou memória, 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. A importância

do tempo de execução de uma tarefa de um sistema pode ser classificada em duas categorias: Soft real time, e Hard real time, a distinção entre estas categorias é explicada na seção **Sistemas Operacionais de Tempo-Real**.

2.5. Sistemas Operacionais de Tempo-Real

Um **sistema operacional** é um conjunto conjunto de software que permitem o gerenciamento e interação com os recursos da máquina através de uma camada de abstração, no contexto deste trabalho, o componente fundamental é o **kernel**, a parte sistema operacional que sempre está executando, o trabalho principal do kernel é permitir a coexistência de diferentes tarefas no sistema que precisam acessar as capacidades do hardware, especialmente tempo na CPU e memória, o kernel pode ser descrito de maneira simplificada como a "cola" entre a aplicação(software) e os recursos físicos(hardware). (GAGNE, 2018)

Já um **sistema operacional de tempo real** (RTOS) é um tipo de SO mais especializado, tipicamente pequeno, que possui como característica central cumprir o requisito temporal, que divide-se 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. O atraso de uma tarefa em um sistema soft real-time não é desejável, mas não constitui um erro. **Exemplos**: Player de DVD, videogames, kiosks de atendimento.
- **Hard Real Time**: Precisam garantir as propriedades de soft real time, além disso, o atraso de uma tarefa de seu prazo (**deadline**), 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 aparacem em contextos críticos. **Exemplos**: Software para sistema de frenagem, Sistemas de navegação em aplicações aeroespaciais, software de trading de alta frequência, broker de mensagens de alta performance.

Como sistemas Hard Real Time cumprem os requisitos de sistemas Soft Real Time, os sistemas operacionais de tempo real tem seu design orientado a serem capazes de cumprir o critério Hard Real Time.

Em contraste com sistemas operacionais focados em uso geral que são encontrados em servidores e computadores pessoais (como Windows, Linux e OSX), o objetivo do primário de um RTOS não é dar ao usuário a sensação de fluidez dinamicamente escalonando os recursos da máquina, sistemas em tempo real buscam ser simples, confiáveis e determinísticos. É essencial que um RTOS execute as tarefas do sistema com um respeito estrito aos prazos de execução fornecidos e que faça de maneira resiliente à flutuações de tempo causadas por IO e outras interrupções.

Drivers em RTOSes são adicionados previamente de maneira **ad-hoc**, não há necessidade de carregamento dinâmico de drivers ou de bibliotecas pois na maioria das aplicações que necessitam de um RTOS, o hardware ja é conhecido e definido de antemão.

Devido à suas características de simplicidade, baixo custo e previsibilidade, os sistemas operacionais de tempo real são extensivamente usados em aplicações de sistemas embarcados. Exemplos incluem: FreeRTOS, VxWorks, Zephyr e LynxOS.

2.6. Escalonador

O escalonador (**scheduler**) é o componente do sistema operacional responsável por gerenciar múltiplas tarefas que desejam executar (GAGNE, 2018), sendo um componente extramente crucial, a implementação do escalonador deve garantir que tarefas de alta prioridade executem antes e que a troca entre tarefas (**context switching**) seja o mais rápido possível, o algoritmo de escalonamento é o fator central para o comportamento do escalonador, sendo categorizados em 2 grandes 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" (**yield**) ou implicitamente ao utilizar uma rotina assíncrona do sistema, como ler arquivo, receber pacotes de rede ou aguardar uma variável de condição.
- **Preemptivos**: Além de poderem transferir a CPU manualmente, o escalonador forçará trocas de contexto caso a tarefa exceda um limite de tempo definido para sua execução, o processo de interromper e trocar de tarefa forçadamente chama-se **preempção**, e a quantia de tempo máximo alocada para execução contínua da tarefa é tipicamente denominada como **time slice** ("Fatia de tempo"). Tarefas ainda podem possuir relações de prioridade, e **time slices** podem também serem alteradas.

Sistemas operacionais de tempo-real são comumente executados no modo totalmente preemptivo, mas o uso cooperativo também é viável e possui vantagem de possuir o controle mais previsível e não necessitar de tantas interrupções de timer, 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 fatia longa de tempo.

2.7. Escalonamento tolerante à falhas

Durante a execução de um sistema tolerante à falhas, existem alguns tipos principais de overheads que independente da presença de uma falha vão ocorrer e precisam ser considerados pelo escalonador.

1. **Mudança de contexto**: Trocar entre tarefas possui um custo inerente pois é necessário salvar o estado da máquina e fazer alterações no TCB (**Task control block**) da tarefa.
2. **Envio de mensagens**: Para comunicar entre tasks ou entre componentes fisicamente distintos do sistema, seja por bus ou por mecanismo de rede, existe um custo inerente à serialização e ao meio de transmissão da mensagem.

3. **Detecção de Erro**: É necessário um overhead fixo para detectar a presença de falhas, um bom algoritmo de detecção possui um equilíbrio entre minimizar esse custo e conseguir detectar falhas com uma alta taxa de acerto, sem presença de falsos positivos.

Na ocorrência de uma falha com uma política de re-execução, existe um overhead extra, similar à de uma mudança de contexto, para restaurar o estado anterior da tarefa.

Uma consequência natural de possuir diversos processos se comunicando com até **k** falhas, é uma explosão combinatória de possíveis caminhos de execução e reexecução, além de drasticamente aumentar o tempo de execução de algoritmos de escalonamento (seja online ou offline), o sistema se torna excessivamente complicado, afetando negativamente duas das características desejáveis de sistemas de tempo real, como o determinismo e as fortes garantias de prazo de execução.

Pode-se reduzir o grau de possíveis combinações e garantir maior previsibilidade do sistema utilizando-se de pontos de **transparência**, também chamados de **freezing**. Para uma tarefa qualquer, considera-se que a tarefa é transparente se para uma deadline especificada e dado um limite de até **k** falhas, se sua execução é finalizada no prazo independente do número de falhas que ocorreram. Para o caso onde nenhuma falha ocorra, existe a presença de um tempo (potencialmente ocioso) extra onde a tarefa está "congelada". Pontos de transparência podem ser estrategicamente escolhidos para garantir o tempo de execução entre as principais macro etapas sem a necessidade de redundância de replicação. É importante ressaltar que a troca fundamental que ocorre na inserção de um ponto de transparência é a troca de maior gasto **temporal** para o caso sem falhas de uma tarefa, em troca de uma garantia sistêmica de sua conclusão, outras tarefas ou nós no sistema são capazes de confiar na conclusão de uma tarefa transparente dado que seu prazo esteja cumprido.

2.7.1. Grafos de execução tolerantes à falha

Para melhor visualização de um fluxo de execução com falhas, é possível utilizar de um mecanismo de diagramação denominado **grafos resilientes à falhas**, que descrevem o comportamento do sistema na presença de falhas. Neste contexto, a distinção entre processo, thread e tarefa não é importante, os termos processo e tarefa serão utilizados de forma intercambiável, e correspondem simplesmente a uma unidade de execução com um espaço de pilha dedicado.

Dado um processo qualquer, será utilizado a notação `PX (N)`, onde X é o número identificador do processo, e N corresponde à sua N-ésima re-execução, por exemplo `P2 (1)` indica a primeira execução do processo P2, enquanto `P1 (3)` indica a terceira reexecução do processo P1. Uma notação similar será utilizada para mensagens entre processos, `mX (N)`, mensagens, assim como processos, estão sujeitas à falhas e overheads de detecção, mas ao invés de re-execução, mensagens são re-enviadas ou restauradas na caso algoritmos de recuperação de erro estejam disponíveis.

No representação de grafo, nós são processos, que podem estar rodando na mesma CPU ou não, arestas indicam o fluxo de execução, uma aresta não marcada indica execução incondicional, já arestas

demarcadas com notação de mensagem, representam execução que depende de uma transmissão de mensagem. Mensagens e processos indicados com um símbolo circular representam pontos ordinários no grafo, já pontos com símbolos quadrados indicam as condições de transparência.

3. >> Grafo simples aqui <<

4. >> Grafo com múltiplas mensagens aqui <<

O escalonamento tolerante à falhas é a combinação de métodos que permitem que o escalonador reaja à ocorrência de falhas e agende as tarefas de forma a minimizar tempo ocioso e overhead de recuperação e detecção. A rotina de escalonamento pode ser executada **online**, onde existe a possibilidade de criar e suspender tarefas dinamicamente ou **offline**, onde o número e prazos das tarefas são determinados previamente. Este trabalho será focado na execução **offline**, pois fornece garantias mais fortes de transparência e previsibilidade, é importante mencionar que um método **offline** de boa qualidade também pode ser adaptado para um contexto **online**.

4.1. Trabalhos Relacionados

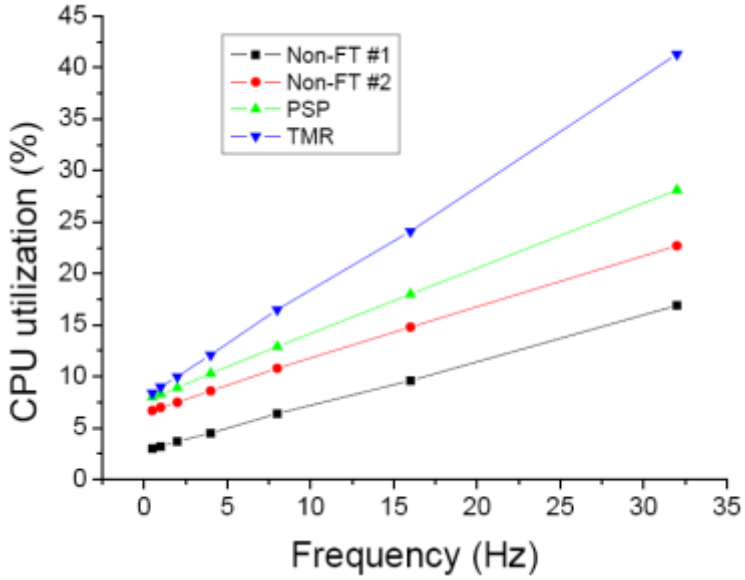
4.1.1. Reliability Assessment of Arm Cortex-M Processors under Heavy Ions and Emulated Fault Injection

Neste trabalho conjunto de pesquisadores da USP e UFRGS utilizam de um sistema COTS e criam um perfil de falhas com exposição a íons pesados assim como injeção artificial de falhas para posteriormente realizar uma adição de formas de detecção de falhas para melhorar a confiabilidade do sistema. Os autores foram capazes de detectar metade das falhas funcionais apenas com técnicas de software no banco de registradores.

Dentre as conclusões extraídas foram que com o uso de técnicas de software foi possível reduzir a quantia de falhas funcionais em mais de 50%, outra questão observada foi que a quantia de falhas injetadas para ocasionar um erro de funcionalidade é 2 ordens de magnitude maior ao comparar registradores em relação à memória, indicando que existe uma necessidade real de poder detectar e mitigar erros de memória mais rapidamente (AZAMBUJ, 2024).

4.1.2. Application-Level Fault Tolerance in Real-Time Embedded System

No artigo de Afonso, Silva, Tavares e Montenegro um framework de execução na para o sistema operacional BOSS é criado, o trabalho apresenta técnicas de escalonamento mas não entra em detalhamento profundo na parte de detecção, mas sim de prover uma biblioteca na forma de classes representando **threads** resilientes. O trabalho possui um caso de estudo com sistema de filtragem de radar, o trabalho demonstra 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).



O trabalho demonstra também a viabilidade de prover interfaces mais abstratas que ainda sejam capazes de rodar em sistemas de recursos restritos, os pesquisadores realizam uso amplo de herança e padrões orientados à objetos com chamadas virtuais. Uma possível otimização em termos de memória e coerência do cachê da CPU é reduzir o uso de despache dinâmico em favor de técnicas de despache em tempo de compilação, como **typeclasses** (Presentes em linguagens como Haskell e Rust) que podem ser também emuladas em C++ com o sistema de `concepts`.

4.1.3. A Software Implemented Comprehensive Soft Error Detection Method for Embedded Systems

No trabalho realizado pelos pesquisadores Asghari, Marvasti e Daneshtalab propõem um método de detecção e reação à erros de controle fluxo juntamente com correção de payloads de dados, o trabalho demonstra resultados positivos e conclui que a aplicação de técnicas de software podem aprimorar drasticamente a tolerância de um sistema. O trabalho possui um foco na análise do grafo de execução do programa, utilizando de IDs para a detecção de jumps errôneos entre blocos básicos. (SEYYED AMIR ASGHARI, 2020)

Esse trabalho relacionado possui similaridade na avaliação da troca de overhead em relação à resiliência com o que será proposto neste artigo, com a principal diferença sendo o enfoque na análise fina dos grafos de controle de fluxo. O trabalho de Asghari et. al serve como um exemplo de uma possível extensão futura da pesquisa apresentada aqui, servindo como uma fonte compreensiva de diversas técnicas de análise e detecção de fluxo defeituoso.

5. Projeto

A etapa mais fundamental do projeto é a implementação dos algoritmos e da API de resiliência, dado o contexto de real time, cuidados devem ser tomados no quesito da performance e uso de memória (que pode indiretamente degradar a CPU na presença de erros de cachê). Dado estas restrições, o uso de

despache dinâmico será mantido baixo, para reduzir o tamanho do executável, não será utilizado mecanismo de exceção com stack unwinding, ao invés, erros de validação devem ser cuidados explicitamente ou através de **callbacks**. Será também assumido que o sistema tenha ao menos uma quantia de memória tolerante (ROM ou não) para guardar os dados necessários para disparar o tratamento de falhas.

A arquitetura será primariamente orientada à passagem de mensagens, pois permite uma generalização para mecanismos de I/O assíncrono e distribuição da arquitetura, permite também um desacoplamento entre a lógica de detecção e transporte das mensagens, potencialmente permitindo otimizações na diminuição da ociosidade dos núcleos. O estilo de implementação orientado à mensagens naturalmente oferece um custo adicional em termos de latência quando comparado à alternativas puramente baseadas em compartilhamento de memória, apesar deste custo poder ser amortizado com a utilização de filas concorrentes bem implementadas e com a criação de um perfil de uso para melhor **tuning** do sistema.

> NOTE: Mencionar que sistemas como o QNX usam isso tbm?

- Criar teste sintetico (stress alto, fault rate alta)
- ? Implementar gerador de tabela de escalonemento ?

5.1. Algoritmos e Técnicas

- CRC: Será implementado o CRC32 para a checagem do payload de mensagens.
- Heartbeat Signal (simples): Um sinal periódico será enviado para a tarefa em paralelo, apenas uma resposta sequencial será necessária.
- Heartbeat Signal (com proof of work): Um sinal periódico juntamente com um payload com um comando a ser executado e devolvido, para garantir não somente a presença da task mas seu funcionamento esperado.
- Replicação espacial: Uma mesma task será disparada diversas vezes, em sua conclusão, será realizado um consenso. A replicação tripla servirá como um controle.
- Replicação temporal: Uma mesma task será re-executada N-vezes, tendo suas N respostas catalogadas, a resposta correta será decidida por consenso.
- Asserts: Não é um algoritmo propriamente dito, mas sim a checagem de algum invarante necessária dentro do código, que caso seja falsa, é tratada como uma falha crítica, espera-se que esse seja um método barato (porém menos robusto) de detectar estados inconsistentes.

5.2. Interface

Uma tarefa (task) é uma unidade de trabalho com espaço de stack dedicado e uma deadline de conclusão.

O “corpo” de um tarefa é simplesmente a função que executa após a task ter sido inicializada. Será utilizado uma assinatura simples permitindo a passagem de um parâmetro opaco por referência. Este parâmetro pode ser o argumento primordial da task ou um contexto de execução.

```
type FT_Task = record
  id: uint,
  body: func(parameter: address),
  param: address,
  stack_base: address,
  stack_size: uint,
  fault_policy: Policy, // Re-exec, Replication, None..
  fault_handlers: []FT_Handler,
  pre_execution: ?Task_Hook,
  post_execution: ?Task_Hook,

  injectors: []Fault_Injector, /* Apenas para testes sinteticos
end
```

- Implementar as rotinas para a interface de resiliencia (spawn_watchdog, check_crc, attach_handler, reexec)

5.3. Visão Geral e Premissas

5.3.1. Premissas

Será partido do ponto que ao menos o processador **watchdog** terá registradores que sejam capazes de mascarar falhas, apesar de ser possível executar os algoritmos reforçados com análise de fluxo do programa e redundância de registradores, isso adiciona uma extra de overhead e como mencionado na seção de trabalhos relacionados, a memória fora do banco de registradores pode ser 2 ordens de magnitude mais sensível à eventos disruptivos, portanto, todos os testes subsequentes assumirão ao menos uma quantia mínima de tolerância do núcleo monitor. Ao invés focando em detecção de falhas de memória, I/O (passagem de mensagem) e resultados dos co-processadores.

Outra necessidade indutiva para a realização do trabalho é 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 overheads relativos introduzidos, mesmo que não necessariamente reflitam as medidas absolutas do produto final.

Uma outra característica sobre falhas, é que tipicamente ocorrem numa fração pequena do tempo de operação do sistema, a maioria das operações ocorrem em um estado correto. Portanto, pode-se

testar um sistema em uma situação de falhas elevadas, de tal forma que consiga o grau necessário de confiabilidade mesmo em uma situação adversa, no caso de sistemas que possuem um impacto crítico ou catastrófico, é melhor optar por ter um excesso de resiliência.

Será assumido que os resultados extraídos de injeção de falhas emuladas, apesar de menos condizentes com os valores absolutos da aplicação e não sendo substitutos adequados na fase de aprovação de um produto real, são ao menos capazes para realizar uma análise quanto ao overhead proporcional introduzido, devido à sua facilidade de realização e poder extrair diversas métricas em paralelo, serão priorizados inicialmente neste projeto.

5.4. Análise de Requisitos

> NOTE: Isso aqui é regra de negocio? O projeto deve ser capaz de executar em um kernel RTOS, se o componente será acoplado diretamente ao kernel ou implementado como uma extensão trata-se de um detalhe de implementação. Além disso, deve ser possível utilizar em um sistema COTS, isto é, não deve estar associado à um hardware particular e deve ser portável na medida em que necessita apenas de uma camada HAL para poder realizar a funcionalidade adequada.

5.4.1. Requisitos Funcionais

5.4.2. Requisitos Não-Funcionais

5.5. Delimitação de Escopo

5.6. Plano de Verificação

- Teste inicial virtualizado -> Provar corretude e projetar overhead dos algoritmos
- Teste final em placa (ESP32?) rodando um RTOS com injeção de falhas

6. Bibliografia

AZAMBUJ, L. R. G. A. F. B. A. N. A. A. S. G. A. A. E. L. A. M. A. V. A. P. A. A. N. H. M. A. F. L. K. A. J. R. "Reliability Assessment of Arm Cortex-M Processors under Heavy Ions and Emulated Fault Injection", 2024.

GAGNE, A. S. A. P. B. G. A. G. **Operating System Concepts**. Tradução:. [S.l: s.n.], 2018.

SEYYED AMIR ASGHARI, M. D., Mohammadreza Binesh Marvasti. "A Software Implemented Comprehensive Soft Error Detection Method for Embedded Systems", 2020.