

DETECÇÃO DE ERROS EM SISTEMA OPERACIONAL DE TEMPO REAL

Universidade do Vale do Itajaí (UNIVALI)

Escola Politécnica - Ciência da computação

Aluno: Marcos Augusto Fehlauer Pereira

Orientador: Felipe Viel

Introdução / Contexto

- Sistemas embarcados estão presentes diversas áreas, tipicamente utilizando de um sistema operacional de tempo real
- É provável que a adoção destes sistemas, particularmente sistemas COTS continue a crescer

Introdução / Problematização

Problematização

- Existem diversas técnicas para tornar um sistema tolerante à falhas, mas seus tradeoffs nem sempre são claros.
- Pode ser vantajoso de um ponto de vista competitivo e social, que estes sistemas apresentem melhor dependabilidade.
- Portanto, é necessário conhecer os tradeoffs de performance em relação ao seu ganho de tolerância.

Solução Proposta

- Implementar técnicas de tolerância à falhas próximas do escalonador do sistema operacional, analisar o impacto de performance causado e criar uma interface para o uso das técnicas.

Objetivos

Geral

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

Específicos

- Selecionar técnicas de tolerância à falhas em software
- Implementar técnicas escolhidas com uma interface para uso
- Realizar testes com de injeção de falhas e coletar métricas de performance das técnicas
- Produzir uma análise comparativa das técnicas, seus custos e eficácia

Definições Principais

Dependabilidade: Propriedade do sistema que pode ser sumarizada pelos critérios RAMS

- **Reliability** (Confiabilidade): Probabilidade de um sistema executar corretamente em um período
- **Availability** (Disponibilidade): Razão entre o tempo em que o sistema não consegue prover seu serviço (downtime) e o seu tempo total de operação
- **Maintainability** (Capacidade de Manutenção): Probabilidade de que um sistema em um estado quebrado consiga ser reparado com sucesso, antes de um tempo t .
- **Safety** (Segurança): Probabilidade do sistema funcione ou não sem causar danos à integridade humana ou à outros patrimônios

*Tolerar falhas influencia positivamente nos critérios **R** e **A**, melhorando a dependabilidade.*

Definições Principais

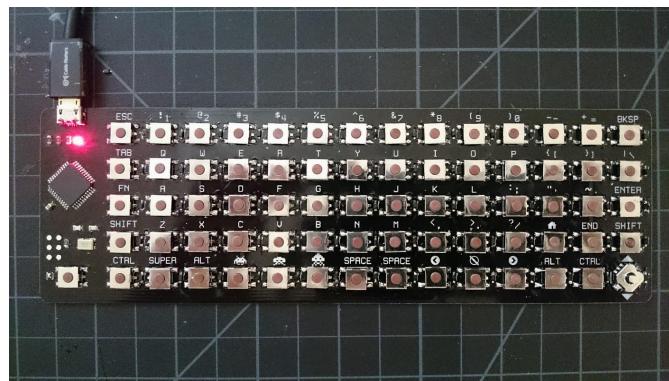
Definições em português segundo a IEEE:

- **Erro (Error)**: A diferença entre um valor esperado e um valor obtido.
- **Defeito (Fault)**: Estado irregular do sistema, que pode provocar (ou não) erros que levam à falhas
- **Falha (Failure)**: Incapacidade observável do sistema de cumprir sua função designada, constituindo uma degradação total ou parcial de sua qualidade de serviço.

Para os propósitos deste trabalho, o termo “Falha” será utilizado como um termo mais abrangente, representando um estado ou evento no sistema que causa uma degradação da qualidade de serviço.

Sistemas Embarcados

- Família vasta de sistemas computacionais que capacitam um dispositivo maior.
- Características comuns: Especificidade, Limitação de Recursos, Critério temporal (Soft ou Hard real time)

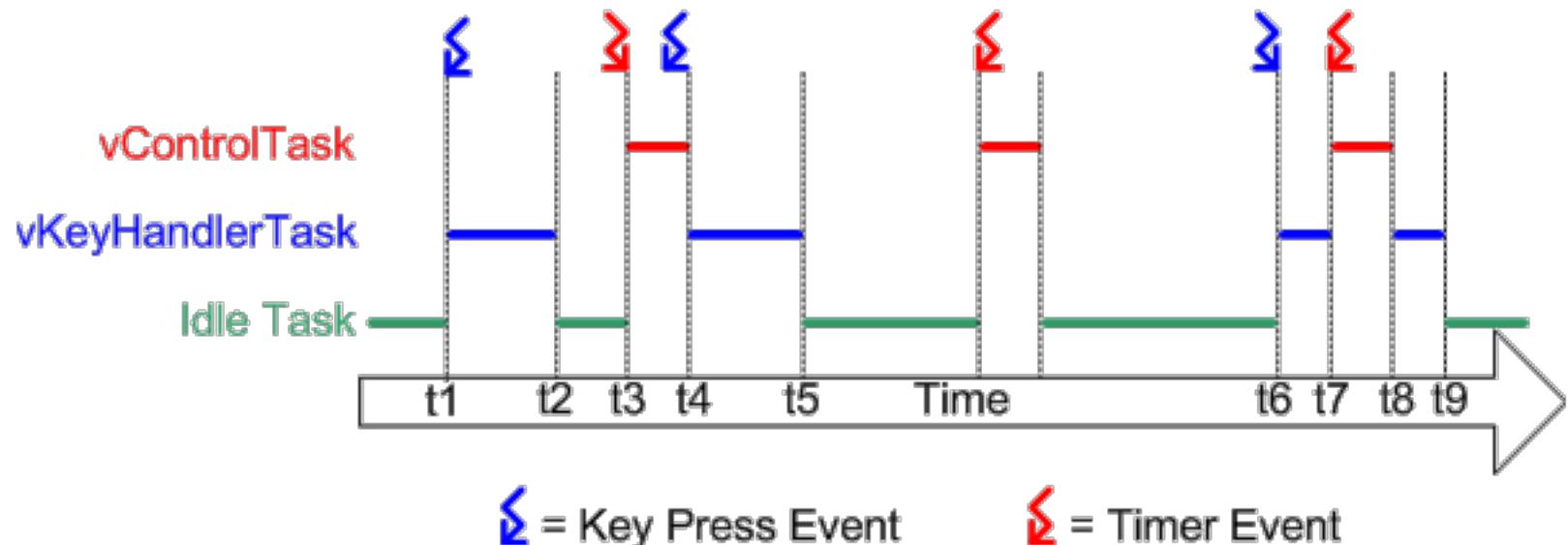


Sistemas Operacionais de Tempo Real

Sistemas comumente usados para diversos tipos de sistemas embarcados, possuem escalonadores totalmente preemptivos. Tipicamente possuem poucas funcionalidades, dependendo apenas de uma HAL (*Hardware Abstraction Layer*)

Escalonador

Componente do Sistema Operacional responsável por gerenciar o tempo da CPU entre das tarefas.



Falhas

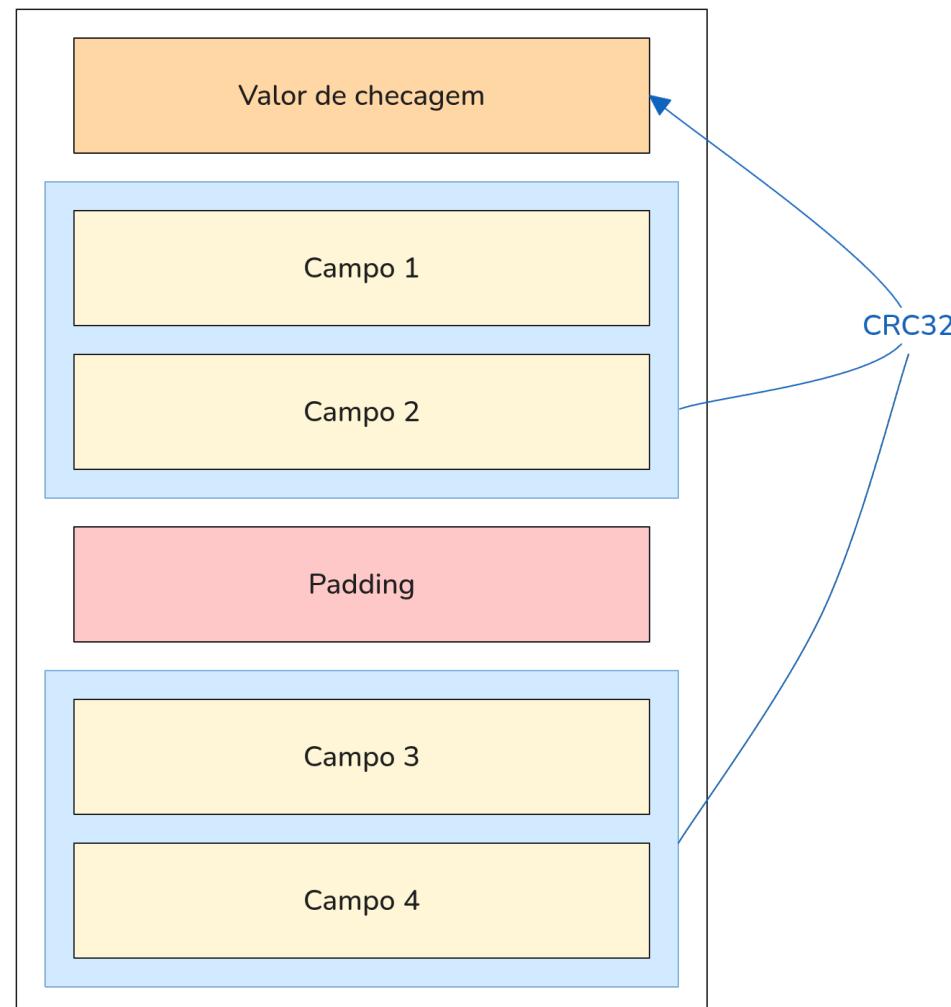
Falhas podem ser classificadas em 3 grupos de acordo com seu padrão de ocorrência:

- 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.

Este trabalho focará na tolerância à falhas transientes.

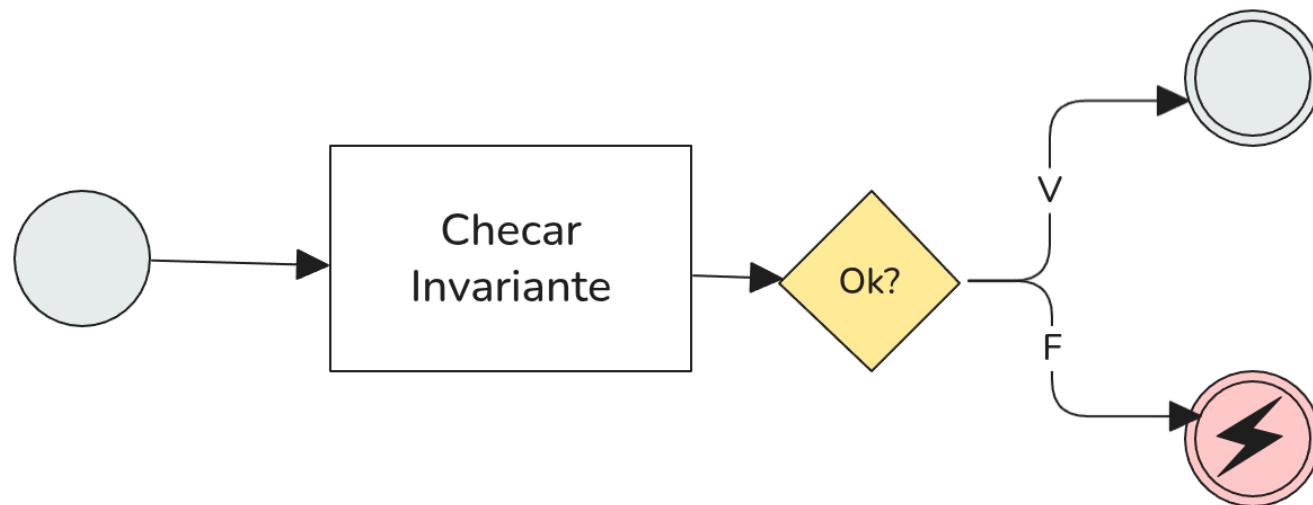
Mecanismos de Detecção / CRC

Cyclic Redundancy Check: Um valor de checagem é criado com base em um polinômio gerador e verificado, utilizado primariamente para verificar integridade dos dados.



Mecanismos de Detecção / Asserts

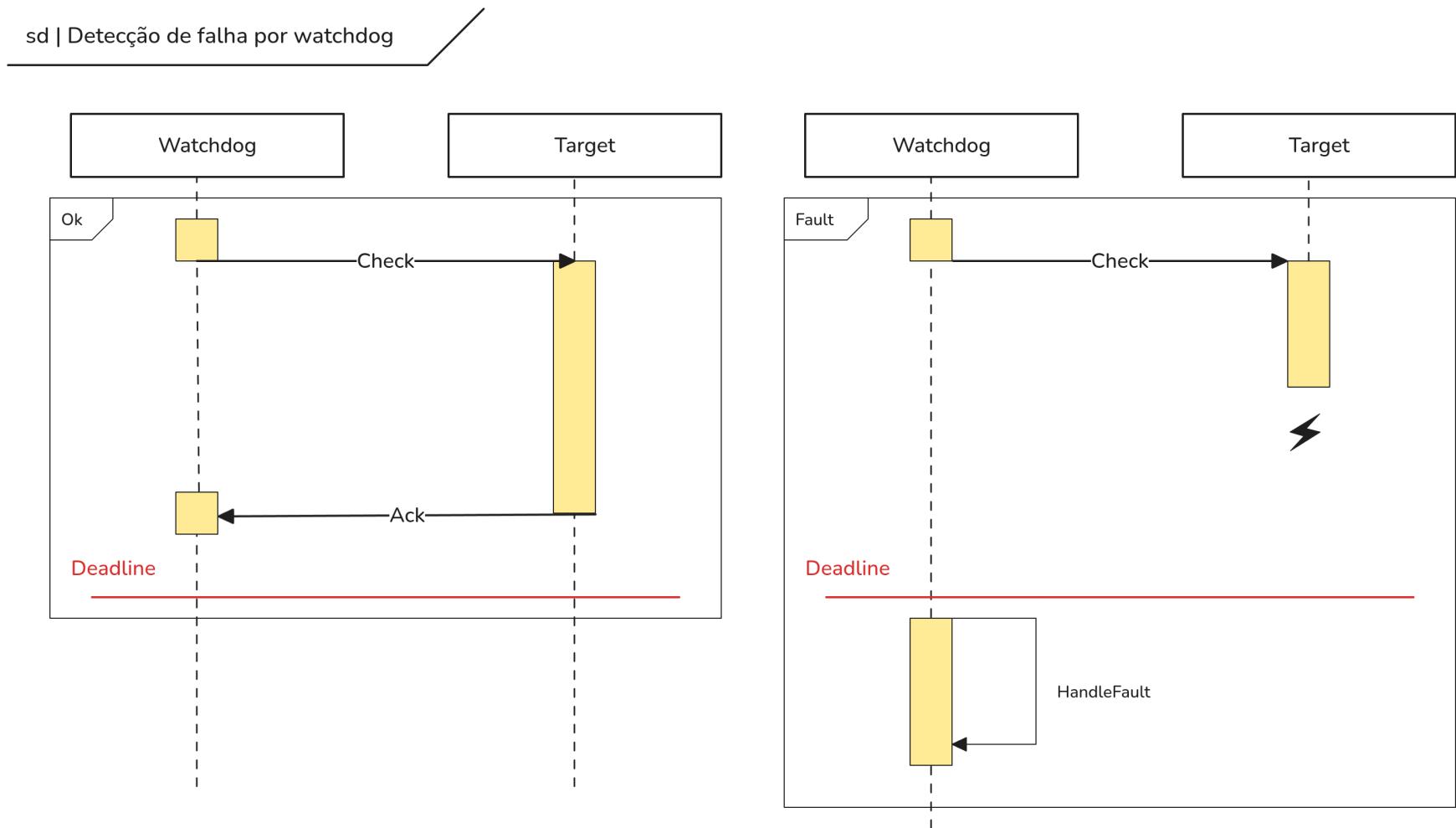
Asserts: Checagem de uma condição invariante que dispara uma falha, simples e muito flexível, pode ser automaticamente inserido como pós e pré condição na chamada de funções



Asserts são uma forma rápida e simples de garantir pré e pós condições

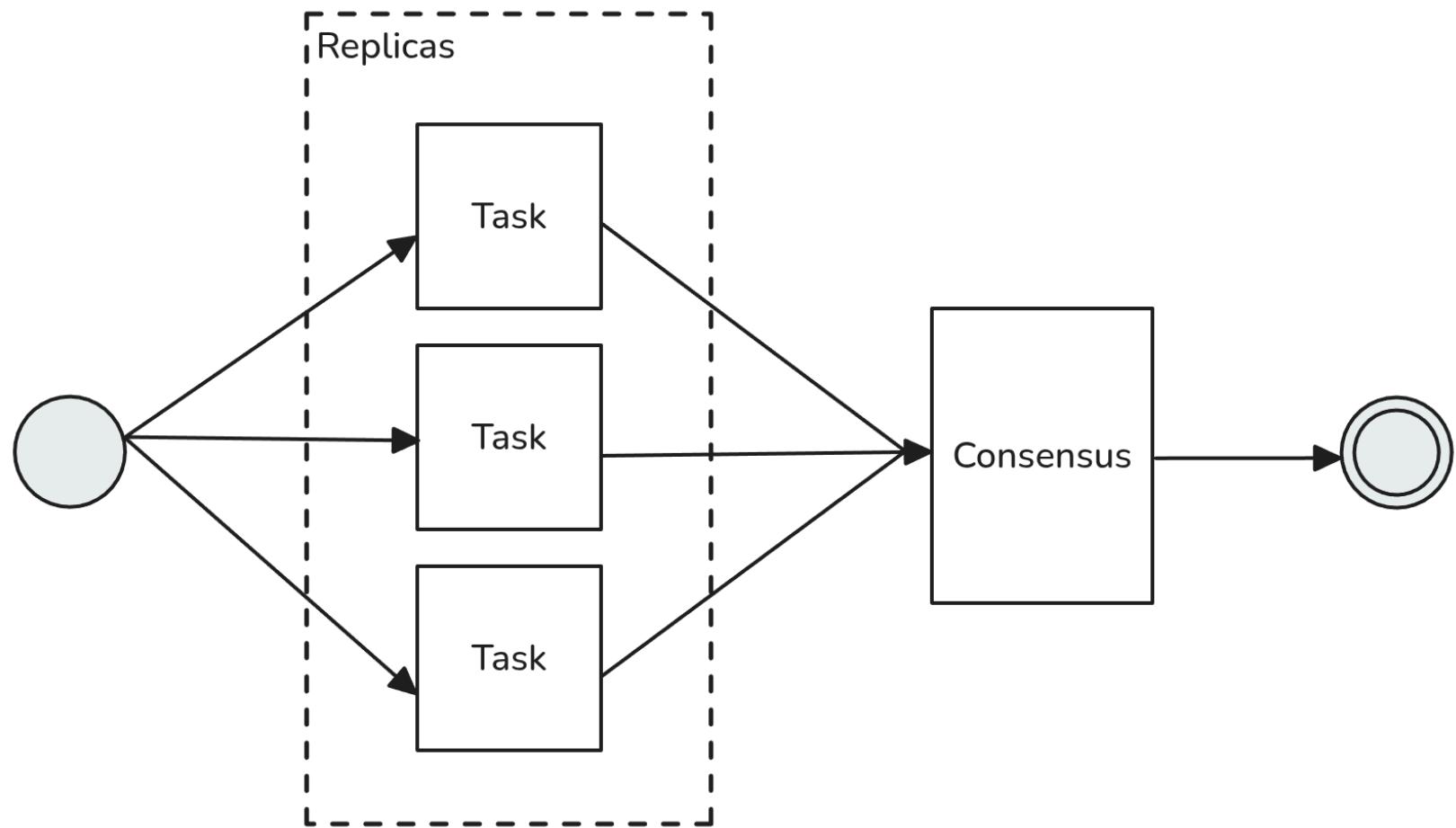
Mecanismos de Detecção / Sinal Heartbeat ou Verificador de Deadline

Envia-se um “sinal” periodicamente para checar o estado do sistema, neste trabalho, foi utilizado na forma de uma DeadlineWatcher que realiza a deteção da violação de prazos.



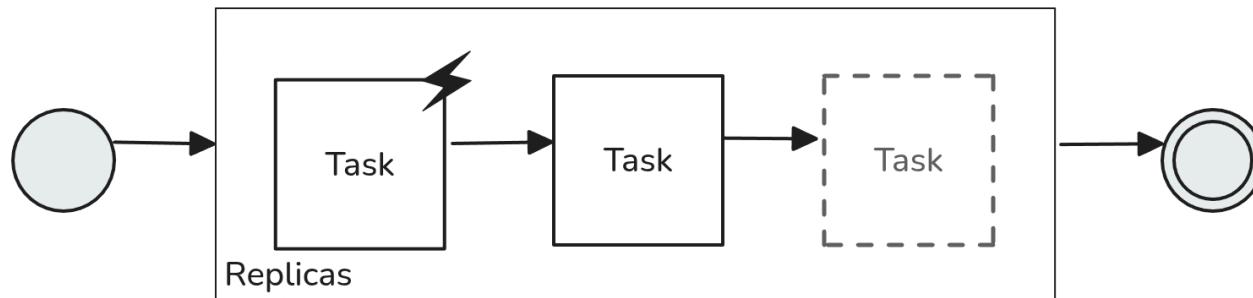
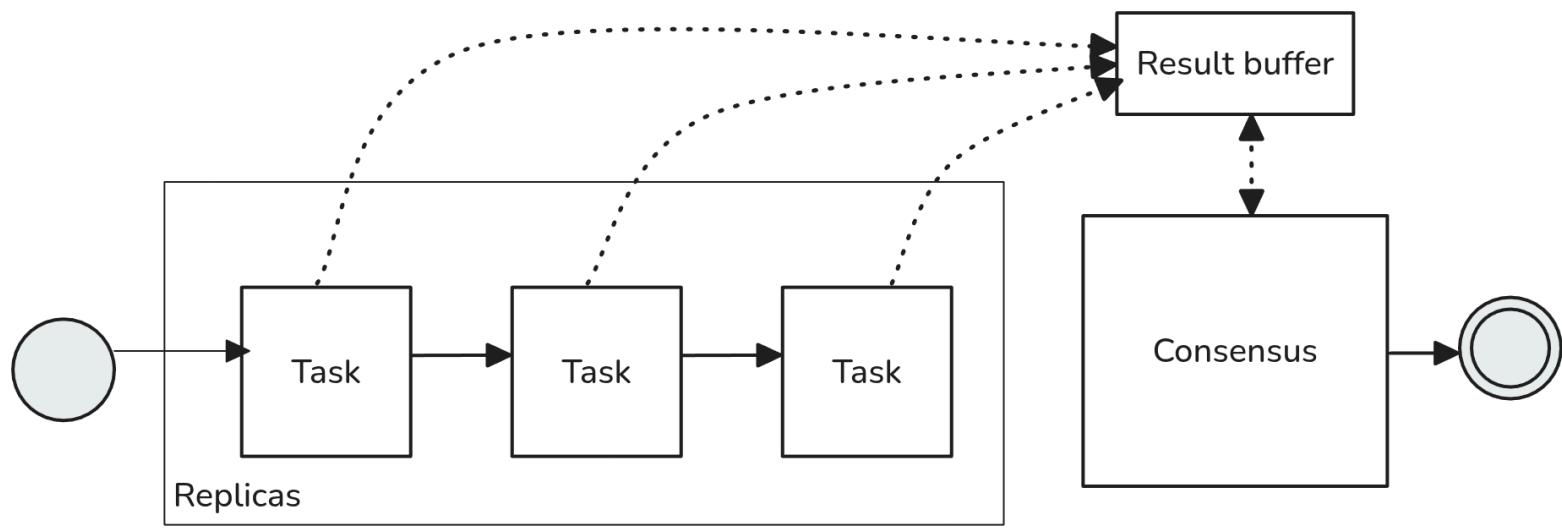
Mecanismos de Tolerância / TMR de Tarefa

Tarefa é replicada N vezes e seus resultados são verificados por consenso majoritário, neste trabalho, foi utilizado TMR (*Triple modular redundancy*), ou seja: $N = 3$



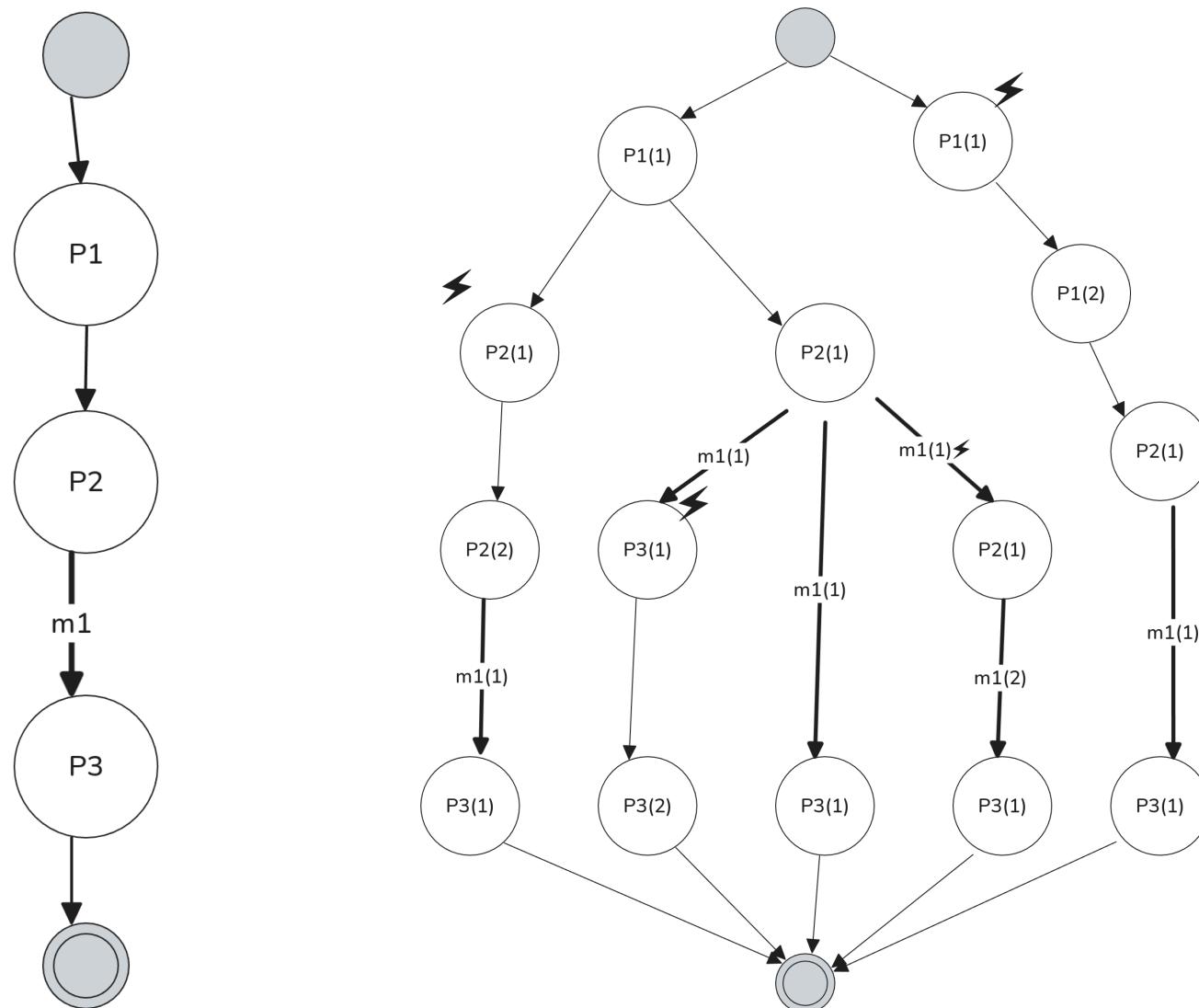
Mecanismos de Tratamento / Reexecução de Tarefa

edundância temporal, pode ser utilizado para estabelecer condições de transparência, depende do fato que é improvável que uma falha transitória ocorra N vezes em sucessão. O resultado um consenso majoritário das reexecuções ou apenas o primeiro resultado sem erros. Este trabalho utiliza do primeiro estilo.

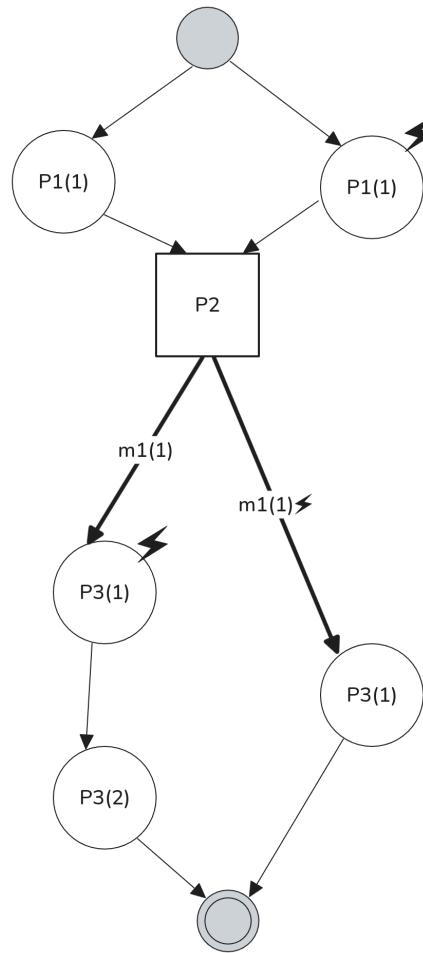


Escalonamento Tolerante à Falhas

Grafo tolerante à falhas de um programa simples (3 processos, 1 mensagem) e sua versão que tolera até uma falha transiente.



Escalonamento Tolerante à Falhas / Condição de Transparência



Condições de transparência podem drasticamente reduzir a complexidade do grafo de execução.

Injeção de Falhas

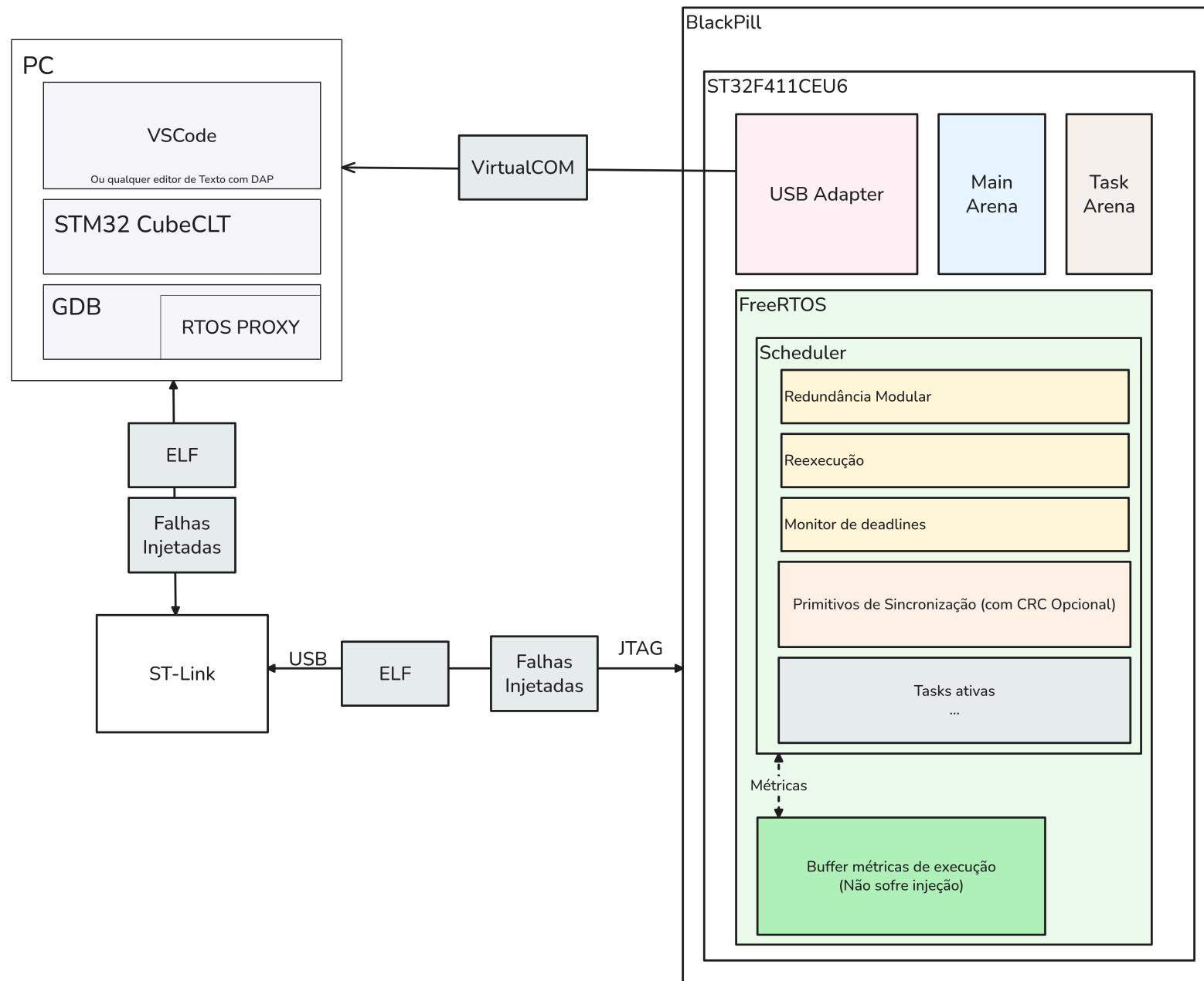
Tipos de injeção e suas desvantagens (Mamone, 2018)

Técnica	Vantagens	Desvantagens
Física	<ul style="list-style-type: none">• Alta fidelidade à falhas reais• Possível injetar em partes específicas do chip	<ul style="list-style-type: none">• Alto custo• Menos controle sobre o tipo particular de falha• Especialistas para lidar com equipamentos
Lógica em Hardware	<ul style="list-style-type: none">• Boa aproximação de falhas reais• Altamente precisa e oferece controle sobre dados injetados• Overhead pequeno no tempo de execução	<ul style="list-style-type: none">• Necessita de uma unidade extra(depurador/injetor)• Necessita de um sistema de comunicação• Pode necessitar de pinos ou modificações adicionais no sistema alvo
Lógica em Software	<ul style="list-style-type: none">• Baixo custo• Flexível e precisa• Altamente portável	<ul style="list-style-type: none">• Overhead no tamanho do código e no tempo de execução
Simulada	<ul style="list-style-type: none">• Zero intrusividade• Hardware alvo não necessário• Flexível e precisa	<ul style="list-style-type: none">• Custo de ferramenta de simulação pode ser alto• Nem sempre uma descrição HDL do sistema está acessível

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	Microkernel de Mezger et al. (2021)	RISC-V Emulado no QEMU	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 Watcher, Asserts, Reexecução e Redundância de Tarefas

Visão Geral

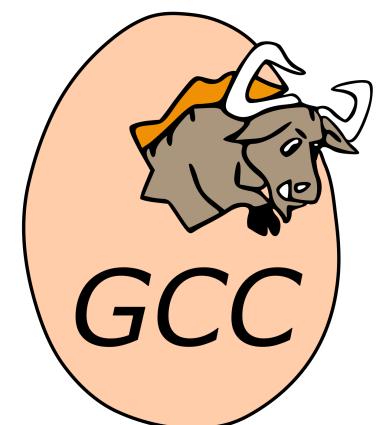
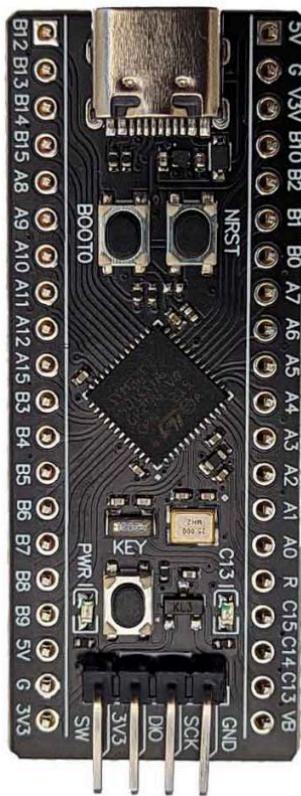


Premissas

- Registradores de controle (Stack Pointer, Return Address, Program Counter, Thread Pointer) serão isentos de falhas diretas.
- Será assumido que testes sintéticos possam ao menos aproximar a medição de um cenário com falhas físicas.
- Não será utilizado RTTI ou exceções baseadas em stack unwinding.

Materiais

- STLink: Depurador de hardware
- STM32F411CEU6 “BlackPill”: Microcontrolador para a execução do código
- GCC: Compilador para a linguagem C++ (Versão 20)
- STMCubeMX e STMCubeCLT: Ferramentas do fabricante



Métodos

- Técnicas de detecção e tolerância baseadas em software
- Injeção lógica em software durante desenvolvimento
- Injeção lógica em hardware para teste final com depurador de hardware
- Medição de memória e tempo de CPU
- Programa de teste que incorpore as técnicas
- Estratégia de alocação em arena

Requisitos Funcionais

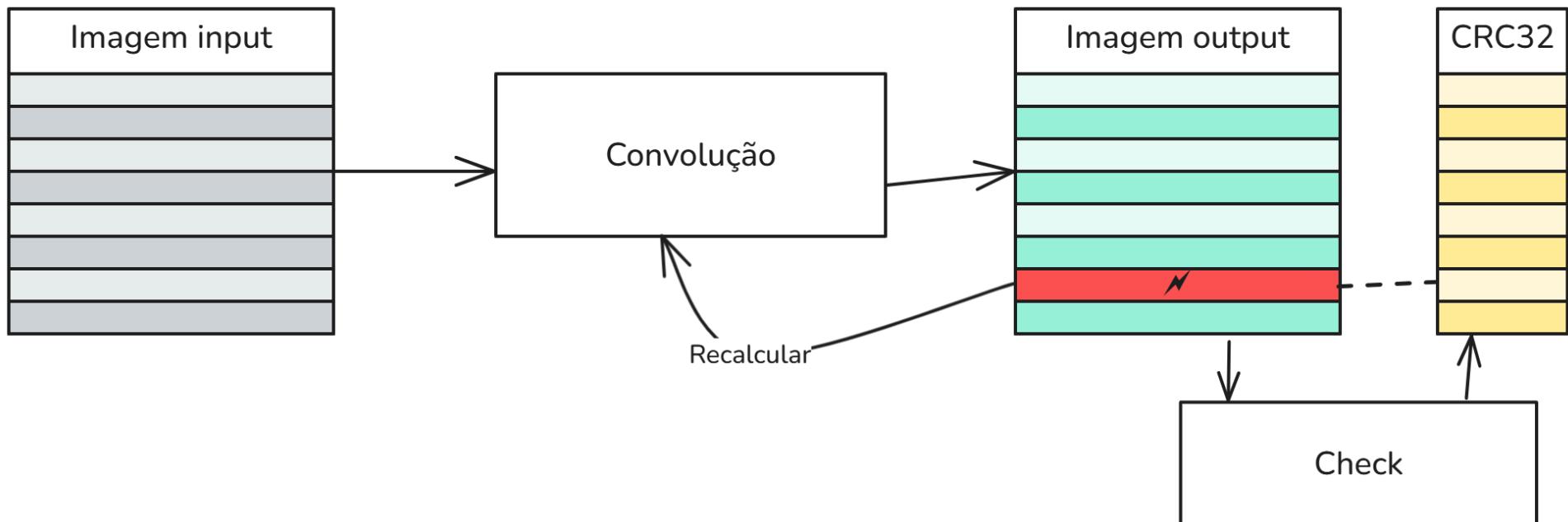
Requisito	Descrição
RF01	Implementação das técnicas propostas
RF02	Configuração do mecanismo de tolerância, prioridade e prazo de execução da tarefa
RF03	Cumprimento do prazo estipulado no momento de criação da tarefa caso não exista presença de falhas
RF04	Dependabilidade superior à versão do sistema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

Algoritmos e Técnicas

- CRC
- Heartbeat Signal / Monitor de Deadline
- Redundância Modular
- Reexecução
- Asserts

Programa Teste

Realiza convolução de uma imagem linha à linha, aplicando um filtro Sobel. Linhas com falhas detectadas são recomputadas mais uma vez.



Campanha de Injeção de Falhas

- Sujeitos à falhas: Maioria da memória RAM
- Falhas injetadas: Bit flips com XOR, números aleatórios.
- Método de injeção: Task auxiliar durante testes e desenvolvimento, Comandos via sessão GDB para alterar valores no controlador via STLink

Combinações de técnicas a serem usadas:

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

- 2 “Rodadas” de injeção
 - Injeção fixa em variáveis conhecidas: Linha computada e kernel do filtro
 - Injeção aleatória no stack frame(variável sorteada + offset aleatório entre 0 e 16)

Desenvolvimento / Tarefas

```
struct RawTask {
    RawTaskFunc func;
    Arena* arena;
    void* args;
    DeadlineSlot* deadline;
    u32 stack_size;
    u32 args_size;
    u32 id{};
    Atomic<TaskStatus> _status;
    TaskCancelCallback on_cancel;
    RawTaskPlatformSpecificData _specific;

    TaskStatus status();

    void join(CALLER_LOCATION);
    void cancel(CALLER_LOCATION);

    ~RawTask() {}

    bool _platform_init(Arena* a, usize
stack_size, RawTaskFunc func, void* args);
    bool _platform_join();
    bool _platform_cancel();
};
```

```
template<
    typename Output,
    Callable<Output, TaskContext> TaskFunc,
    Callable<void, TaskContext> OnCancel
>
struct BasicTask {
    RawTask _task;
    TaskFunc _func;
    OnCancel _on_cancel;
    Option<Output> _result;

    Option<Output> result();
    bool has_result() const;
    TaskStatus status() const;
    RawTask* raw_task();
    u32 id();
    void join();
    void cancel();
};
```

Desenvolvimento / Tarefas

```
auto numbers = Array<i32, 7>{-4, -2, 0, 6, 9, 2, 1};
auto task = make_basic_task(&arena, [numbers](TaskContext* ctx) -> i32 {
    i32 acc = 0;
    for(i32 x : numbers){
        acc += x;
    }
    return x;
});

task.join();
auto sum = task.result().unwrap();
printf("Sum = %d\n", sum);
```

Desenvolvimento / Deadlines

```
struct DeadlineWatcher {
    Slice<DeadlineSlot> slots;
    Spinlock _lock{};
    Atomic<u32> _count;
    char name[12] = {};

    [[nodiscard]]
    bool add(void* key,
             SlotCancellationCallback cancel,
             Duration limit
    );
    bool reset_deadline(void* key);

    DeadlineSlot* get(void* key);

    void remove(DeadlineSlot* node);

    void remove_key(void* key);

    void clear();

    bool scan();
};
```

```
using SlotCancellationCallback =
    void (*) (void* data);

struct DeadlineSlot {
    TimeTick last_tick;
    Duration limit;
    void* key;
    SlotCancellationCallback cancel;

    void reset(){
        last_tick = tick_now();
    }
};
```

Desenvolvimento / CRC32

- Baseado em lookup table (gerado pelo script de build para o polinômio)
- Concept `CRC_Checkable` restringe aplicação apenas para tipos compatíveis

```
template<typename T>
struct CRC32 {
    u32 get(T const& data) = delete;
};

template<>
struct CRC32<Slice<u8>> {
    u32 get(Slice<u8> data); // Base specialization
};

template<typename T>
concept CRC32_Checkable = requires(T const& obj) {
    { CRC32<T>{}.get(obj) } -> SameAs<u32>;
};
```

Desenvolvimento / Asserts

- Asserts globais

```
static inline
void ensure(bool pred, char const* msg, CALLER_LOCATION){
    ensure_ex(pred, msg, caller_location.file_name(), caller_location.line());
}
```

- Asserts locais que cancelam tasks

```
void TaskConext::ensure(bool pred, cstring msg, CALLER_LOCATION){
    if(!pred){
        error_printf(caller_location.file_name(), caller_location.line(),
                     "[Task %d] Assertion failed: %s\r\n",
                     int(task->id), msg
        );
        task->cancel();
    }
}
```

Desenvolvimento / Outros detalhes

Dependências Adicionais

- USB_DEVICE do fabricante do hardware (para output VirtualCOM)
- stb_sprintf de Sean Barrett para formatação consistente (Não essencial, usado para debug)
- Utilitários base providos pelo autor

Build do Projeto

- Build realizada com utilitário build.lua junto com o projeto (Flash + Injetar imagem no ELF final) e toolchain embarcada da ARM

Resultados / Execução

The screenshot shows a debugger interface with two main panes. The left pane displays the source code for `main.cpp`, with a red arrow pointing from the text "Breakpoint no 'main'" to the line where the breakpoint is set. The right pane shows the assembly code for `do_sobel_reexec`. A red box highlights the instruction at address `00aba6`, which is part of a conditional jump. The tooltip for this instruction says "Click to add a breakpoint" and "Condition: row == 30". The terminal window at the bottom shows the command `sudo microcom -f -p /dev/ttyACM0` being run.

Breakpoint no "main"

Extensão da STM, permite ver periféricos e tasks do RTOS. Importante!

```
src > main.cpp 9+, M x
src > main.cpp > do_sobel_reexec
310
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     #ifndef 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             if(consensus(output_rows[0], output_rows[1], ou
336
337             Condition: row == 30
338             if(output_rows[cons].len != image.width){
339                 panic("Row error\r\n");
340             }
341         }
342     }
343 }
```

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

Resultados / Execução

A screenshot of a debugger interface. At the top, there is a memory dump window showing a structure with fields: output_rows, 0, 1, and data. The data field is highlighted with a red box and contains the value 0x2000eb10. A red arrow points from this value to the assembly code below. The assembly code is in Rust syntax, showing a loop that iterates 120 times. The current instruction being executed is highlighted in orange.

```
output_rows = [3]
> 0 = {...}
> 1 = {...}
< data = 0x2000eb10 <main_arena_memory+144> "c\t...
| *data = 99 'c'
| len = 120
333
334
335
336
337
338
339
line_time_a
i32 cons =
ensure(cons
if(output_r
panic("
```

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

A screenshot of a terminal window displaying the execution logs of a task. The logs include information about the Task Arena Size, Address Width, Tick Frequency, and RawTask size. It also shows the execution of a Sobel Filter, which resulted in a warning: "WARNING: Consensus without full agreement.". The logs then show performance metrics like Took and ms/row, task arena sizes, main arena size, and image dimensions. Finally, it indicates that EOF was received from a port.

```
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
```

Resultados / CRC32

The screenshot shows a debugger interface with several windows:

- RUN AND DEBUG**: Shows "STLink launch" selected.
- BREAKPOINTS**: Lists two breakpoints: "main.cpp src" at line 292 and "main.cpp src" at line 555.
- WATCH**: Shows a watchlist for "row_crcs.data" at address 0x2000f3b8, which is a pointer to a memory area. It also shows "output.pixel_data" as a local variable with a value of 0x2000b993.
- VARIABLES**: Shows local variables: "image", "output", "begin" (value 3013), "row_arena" (value 0x2000f240), "line_time_acc", "row_crcs", and "end".
- Code Editor**: Two tabs are open: "main.cpp 9+" and "main.c 9+". The code is as follows:

```
src > main.cpp > do_sobel_simple
276
277     if(ok != image.width){
278         panic("Row error");
279     }
280     auto dest = output.pixel_data;
281
282 #ifdef FT_USE_CRC
283     row_crcs[row] = crc32(output);
284 #endif
285
286     copy(dest, output_row);
287     row_arena->offset = 0;
288 }
289
290
291 #ifdef FT_USE_CRC
292     println("[CRC Output check]");
293     if(!crc_row_check(output, row_c)
294         for(i32 row = 0; row < image.
295             if(row_crcs[row]){
296                 continue;
297             }
298             printin("[RECOMPUTE: %d]", row);
299             auto output_row = make_
300             i32 ok = 0;
301             Duration line_time = ti
302             ok = sobel_row(imag
303         );
304         line_time_acc = line_tir
305
306         if(ok != image.width){
307             printf("Row error\r\n");
308             auto dest = output.pixe
309 }
```

- Memory Dump**: Two panes show memory dumps. The top pane shows memory starting at 0x2000f3b8, and the bottom pane shows memory starting at 0x2000b993.

RUN AND DEBUG STLink 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> ... 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+ ...

src > main.cpp > do_sobel_simple

```

276
277     if(ok != image.width){
278         panic("Row error");
279     }
280     auto dest = output.pixel_data;
281
282 #ifdef FT_USE_CRC
283     row_crcs[row] = crc32(output);
284 #endif
285
286     copy(dest, output_row);
287     row_arena->offset = 0;
288 }
289
290
291 #ifdef FT_USE_CRC
292     println("[CRC Output check]");
293     if(!crc_row_check(output, row_c
294         for(i32 row = 0; row < image
295             if(row_crcs[row]){
296                 continue;
297             }
298             auto output_row = make_s
299             i32 ok = 0;
300
301             Duration line_time = tim
302             ok = sobel_row(imag
303             line_time_acc = line_time
304
305             if(ok != image.width){
306                 printf("Row error\r
307             }
308             auto dest = output.pixel_
309             copy(dest, output_row);
310             row_arena->offset = 0;
311

```

memory.bin x

0x2000f3b8 > memory.bin

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 7A 0E 88 C0 42 BE DB 00 4D 13 BC 00 66 F4 4A
00 B4 72 9F 60 F0 D4 B1 A0 1A 73 6E A0 54 DA 08
C0 F1 BE D2 E0 D5 2E 17 C0 55 D6 C9 80 5F 42 4A
20 FC 82 80 60 B1 9D C2 00 58 F4 4A 20 D6 0E 49
A0 74 3A 16 E0 1B E8 80 E0 FC BF A1 C0 59 86 67
40 29 EC 19 40 91 14 19 20 69 A3 AB C0 6C DD 2A
80 70 3E C1 E0 8C CD DA 40 F5 DB CF C0 71 39 D9
60 A1 51 2C 40 C9 F7 0A 60 19 B4 C9 80 58 00 39
20 56 BE 0F C0 31 4D A4 40 95 AD 9B 40 F6 6C D1
20 DE B5 D4 40 00 04 57 80 42 7E EF C0 7C 2C 78
C0 80 42 A3 A0 18 6A AC A0 BB 93 D6 A0 B8 83 DA
60 3C 49 A1 40 D2 67 EA A0 6B 7D 86 C0 9A C3 DE
E0 E9 18 40 60 68 00 8C 40 BC 6E 18 20 F0 E4 43

memory.bin x

0x2000b993 > memory.bin

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
69 69 69 69 FF E5 B8 96 A2 98 8C 6E 4C 38 6A 68
4E 2E 4C 83 53 37 23 79 7A 51 30 4B 90 77 63 63
8A 94 C4 FF FF E7 D7 FF FF F7 E3 E1 FF EC D6 C2
FF FF FC F3 E5 DE C6 90 68 69 62 7B DF DF BF F4
F5 CA 8C 5E 49 43 6E 73 96 CC A7 9C 9A A4 8E 6F
62 84 71 66 5D 81 73 60 51 7F 92 A5 B6 A5 A8 CE
E6 CD B2 8D 99 A3 B8 C3 B4 EB FF FD FD FD FD
FC FF FD FD FD FD BE E1 00 02 4E 12 00 00 00
15 02 00 00 00 00 37 18 02 06 38 57 10 28 37 6F
2C 00 00 38 60 15 00 06 29 22 42 6C 30 00 00 3A
1A 00 00 06 2D 00 00 00 4D 31 00 00 00 00 00 00
00 0B 00 1B 71 3A 00 39 1A 00 00 01 03 03 2D 1D

RUN AND DEBUG STLink launch

BREAKPOINTS

- main.cpp src (292)
- main.cpp src (296)
- main.cpp src (555)

WATCH

- row_crcs.data = 0x2000f3b8 <main_arena_memo...> X
- *row_crcs.data = 0
- output.pixel_data = {...}
- data = 0x2000b993 <image_output_storage+15> "iii..." len = 14400

VARIABLES

Local

- output_row = {...} ok = 536933304
- dest = {...} row = 0
- image = {...}
- output = {...}

CALL STACK

Paused on breakpoint

```

src > main.cpp > do_sobel_simple
    panic("Row error");
}
auto dest = output.pixel_data.skip(row * ou
281
282 #ifdef FT_USE_CRC
283 row_crcs[row] = crc32(output_row);
284 #endif
285
286 copy(dest, output_row);
287 row_arena->offset = 0;
}

#endif
291
292 if(!crc_row_check(output, row_crcs)){
293     for(i32 row = 0; row < image.height; row
294         if(row_crcs[row]) continue;
295         println("[RECOMPUTE: %d]", row);

296         auto output_row = make_slice<u8>(row_ar
297         i32 ok = 0;

298         Duration line_time = time_it([&](){
299             ok = sobel_row(image, row, output_
300         });
301         line_time_acc = line_time_acc + line_t
302
303         if(ok != image.width){
304             printf("Row error\r\n");
305         }
306         auto dest = output.pixel_data.skip(row
307         copy(dest, output_row);
308         row_arena->offset = 0;
309     }
310 }
311
312 }
#endif

```

memory.bin

0x2000f3b8	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000	00 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000010	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000020	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000030	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000040	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000050	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000060	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000070	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000080	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
00000090	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
000000A0	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
000000B0	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00
000000C0	01 00 00 00 01 00 00 00 01 00 00 00 01 00 00 00

memory.bin

0x2000b993	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000	69 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 48 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 FD FD
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
000000B0	00 0B 00 1B 71 3A 00 39 1A 00 00 01 03 03 2D 1D
000000C0	36 53 05 04 0B 0A 00 00 06 26 04 0F 16 30 06 00

Resultados / Confiabilidade e Desempenho I

Resultados sem injeção de falhas

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

Legenda

T_{total} Tempo total de execução

T_{linha} Tempo execução médio por linha da imagem

M_{task} Pico de alocação na arena de tasks

M_{extra} Pico de alocação na arena de memória extra

Resultados / Confiabilidade e Desempenho II

Técnicas	T_{total}	T_{linha}	M_{task}	M_{extra}	Detecção	Resultado	Obs.
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
Reexec + 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

Resultados / Confiabilidade e Desempenho III

Técnicas	T_{total}	T_{linha}	M_{task}	M_{extra}	Detecção	Resultado	Obs.	Var
N/A	1100ms	60ms	2768B	374B	1	Erro	Prazo de linha expirada	row 0:8
CRC32	1100ms	60ms	2768B	856B	1	Erro	Prazo de linha expirada	row 0:8
Reexec	<100ms	0ms	2768B	1074B	1	Erro	Assert: row.width != output.width	output_rows 4:12
Reexec + CRC32	2265ms	38ms	2768B	1556B	1	Erro	Hard Fault do FreeRTOS	output_rows 4:12
TMR	4625ms	38ms	8336B	1126B	0	OK	Evitou corrupção através de indireção	tmr0 0:4
TMR + CRC32	4628ms	38ms	8336B	1608B	0	OK	Evitou corrupção através de indireção	tmr0 0:4

Resultados / Impacto das falhas no Output

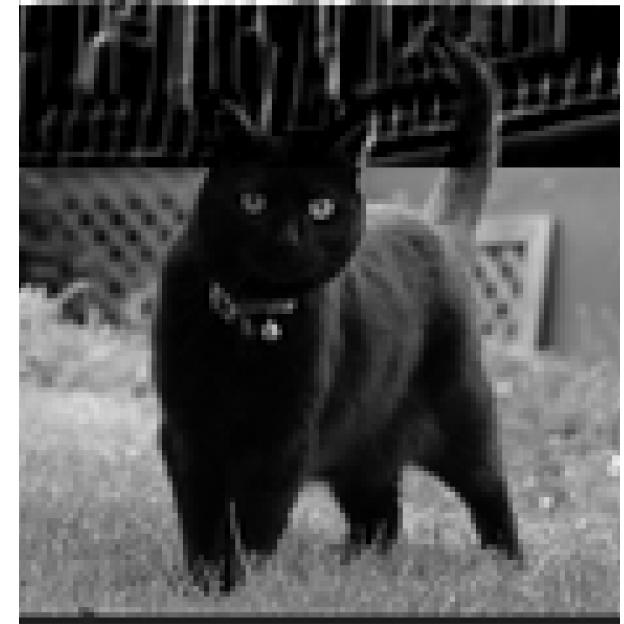
Resultado correto (Input → Output)



Resultados / Impacto das falhas no Output

Exemplos do impacto no output quando nenhuma técnica foi aplicada

1. Corrupção no kernel do filtro Sobel
2. Corrupção sutil em uma linha
3. Corrupção causou interrupção na execução



Verificação

Requisito	Descrição	Validação
RF01	Implementação de todos os algoritmos descritos na seção de projeto	Os algoritmos foram implementados e utilizados
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. Foi capaz de detectar violações

Resultados

Confiabilidade

- Impacto **positivo** na confiabilidade do sistema
- Melhor combinação: TMR/Reexecução + CRC32

Custos das Técnicas

- Custo de reexecução e TMR: 3x mais lento (sem multiprocessamento)
- Memória TMR: >3x overhead
- CRC32 Adicionaram overhead moderado em troca de boa detecção

Resultados

TMR vs Reexecução

- TMR evitou erros fatais em injeções no stack frame
- Redundância de estado adicional mascara padrões de falha complexos
- Trabalhos futuros: avaliar TMR com multiprocessamento simétrico

Validação de Deadline

- Pouco acionada durante testes (asserts e outros mecanismos interceptaram falhas primeiro)
- Útil no desenvolvimento: detectou deadlock de sincronização
- Necessita testes com maior sensibilidade temporal para evidenciar benefícios

Conclusões

Trabalho Realizado

- Exploradas técnicas de execução, monitoramento e checagem de integridade
- Implementação em C++20 no STM32F411CEU6 com FreeRTOS
- Campanha de injeção de falhas transientes para análise de impacto

Resultados Obtidos

- Reexecução/TMR: Mascarou falhas dentro do critério de tempo real
- CRC32: recomputação de dados corrompidos
- Monitoramento de deadlines: detecção de violações temporais e auxílio no desenvolvimento

Trabalhos Futuros

- Avaliar técnicas em multiprocessamento simétrico (múltiplos núcleos)
- Expandir detecção com análise de fluxo de controle em nível de compilador
- Automatizar injeção de falhas (ex: PyOCD)

DETECÇÃO DE ERROS EM SISTEMA OPERACIONAL DE TEMPO REAL

Universidade do Vale do Itajaí (UNIVALI)

Escola Politécnica - Ciência da computação

Aluno: Marcos Augusto Fehlauer Pereira

Orientador: Felipe Viel