SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

Faculty of Informatics and Information Technologies

Reg. No.: FIIT-100241-117028

Software-implemented fault tolerance

Bachelor thesis

Study programme: Informatics
Study field: Computer Science
Training workplace: Institute of Informatics, Information Systems and Software Engineering
Thesis supervisor: Ing. Ján Mach

Bratislava 2025 **Filip Ďuriš**

Annotation

Slovak University of Technology Bratislava
Faculty of Informatics and Information Technologies
Degree Course: Informatics

Author: Filip Ďuriš
Diploma Thesis: Software-implemented fault tolerance
Supervisor: Ing. Ján Mach
Máj 2025

Anotácia

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Študijný program: Informatics

Autor: Filip Ďuriš
Diplomová práca: Software-implemented fault tolerance
Vedúci diplomového projektu: Ing. Ján Mach
Máj 2025

Contents

1 Introduction

With the increasing demand for high-performance embedded software comes the inevitable, difficult task of ensuring fault-free functioning of ever-more complex systems. The increased complexity, both in terms of hardware components and software features, increases the number of failure points where errors can manifest.

Errors can be caused by both external factors beyond our control such as radiation in space, but also as a result of human error while designing the software. Due to the incredibly complex nature of this problem, it is unlikely that we will be designing completely error-free software and hardware in the near future. This simple fact makes fault-tolerance an important aspect of software design, especially when designing critical systems whose failure could endanger human life. However, as with most things, fault-tolerance and reliability is a trade-off, which usually comes at the cost of performance and development time.

Historically, speciallized hardware was the go-to choice for implementing fault tolerance, specifically by hardening and or duplicating the components. This approach requires designing and producing non-standard computure components, exponentially increasing the development cost. Lately, a more economical solution began gaining traction for non-critical missions. Namely, using off-the-shelf components without specialized hardening, while using software reduncies to implement fault tolerance.

This thesis will aim to analyze various common software-implemented fault-tolerance methods. We will look at the benefits and drawbacks of the utilized methods, as well as construct a working demo based on FreeRTOS running on RISC-V core that implements and tests the effectiveness of some selected methods using the Rust programming language.

2 Nomenclature

For the sake of consistency in the terms used in this document, we will refer to the naming conventions and definitions as outlined in [Avi+04]. Below are the definitions of the most crucial terms used.

Failure is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. A service failure is a transition from correct service to incorrect service, i.e., to not implementing the system function [Avi+04].

Error is a deviation of the service from its correct state. It is a part of the system's state that may lead to service failure [Avi+04]. Errors are the observable result of issues within the software and or hardware.

Fault is the actual or hypothesized cause of an error. Faults are usually considered dormant until manifested, causing an error [Avi+04]. An example of a fault might be hardware issue causing an I/O device to send corrupted data as input. If the software is not designed to deal with incorrect input this would lead to an internal error, possibly causing a service failure.

Fault Tolerance is the system's ability to continue operating correctly in the presence of faults. Rather than eliminating faults entirely, fault-tolerant systems are designed to detect, isolate, and recover from faults before they cause service failures.

System boundary defines what is considered part of the system and what is not. It separates the system from everything outside it. This helps us know which components, functions, or behaviors are included when analyzing or designing the system. Understanding the system boundary is important for knowing where problems might happen and who or what is responsible for handling them.

3 Requirements Specification

The primary objective of this project is to analyze, implement, and evaluate selected software-based fault tolerance techniques. The focus is on embedded systems operating in environments where faults, particularly single event upsets (SEUs), can compromise system behavior. These issues are especially relevant in resource-constrained systems where safety and reliability are paramount.

This section outlines the functional and non-functional requirements that the proposed system must fulfill to meet its objectives. The requirements are derived from practical scenarios and target use cases involving typical embedded workloads, including mathematical computations, sorting algorithms, and data integrity verification. These tasks will be executed under both normal and fault-injected conditions to evaluate the system's behavior and robustness.

Functional Requirements

- The system shall implement multiple software-based fault tolerance techniques.
- The system shall be capable of detecting and reporting transient faults at runtime.
- The system shall provide a mechanism for recovering from detected faults where applicable (e.g., via checkpoint rollback or redundant execution).

Non-Functional Requirements

- The system shall introduce minimal performance overhead relative to the unprotected baseline.
- The solution shall be implemented entirely in software.
- All fault tolerance methods shall be implemented in the Rust programming language to leverage its memory safety guarantees and low-level control.

- The system shall be portable and capable of running on a 32-bit RISC-V platform using the FreeRTOS operating system.
- The implementation shall be maintainable and modular, enabling future extension or substitution of fault tolerance mechanisms.

Ultimately, the project aims to deliver a practical and reproducible evaluation framework for software-based fault tolerance, enabling developers to assess the trade-offs between fault coverage, runtime overhead, and system complexity in embedded contexts.

4 Types of faults

In order to be able to analyze fault-tolerance techniques, we first need to understand the various types of faults that we might encounter. This list is not exhaustive and only covers fault types relevant to this thesis, for more complete list see [Avi+04]. It is also important to understand two different classifications are not always mutually exclusive. As we will see later on, a solid fault, for example, can be caused during development, but can also be a product of external factors.

Faults can be further split into various categories, the ones mostly relevant to us are **external faults** which originate from outside the system boundaries and propagate into the system. Which also includes **natural faults** that are caused by natural phenomena without human participation. These faults can affect the hardware and the software, which is why we can further classify them as **hardware faults** and **software faults** respectively [Avi+04].

4.1 Transient fault

is a temporary fault which results in an error. Transient faults are usually caused by random, external events (e.g. radiation) interfering with the system. A key characteristic of transient faults is the inability to accurately reproduce it, since the cause is external to the system. Retrying the same process multiple times is usually enough to deal with a transient fault, since it is extremely unlikely a transient fault will occur repeatedly within the same process at successive time intervals. If a fault persists across multiple attempts, it is possible we are dealing with a permanent fault.

4.2 Permanent fault

also known as solid fault, is a lasting fault within the system that requires maintenance to remove. This could come in various forms, such as damaged hardware, corrupted memory or missing code. When dealing with a solid fault, retrying the same process is usually not enough as the same error will keep reoccurring. This approach requires alternatives and fallbacks within

the system, which is outlined in more detail in section 5.2.

4.3 Hardware faults

Hardware errors are caused by external factors beyond our control as software developers. They are usually caused by environmental influences, such as cosmic radiation in space, electro-magnetic fields or adverse weather conditions.

One of the most common hardware errors is memory corruption, which can appear in many forms and result from a wide range of causes. For instance, radiation exposure in space can cause SEUs, flipping individual bits in memory and altering data unpredictably. Similarly, physical damage to storage media, such as hard drives or SSDs, might corrupt specific regions of the file system, making certain data inaccessible or incorrect.

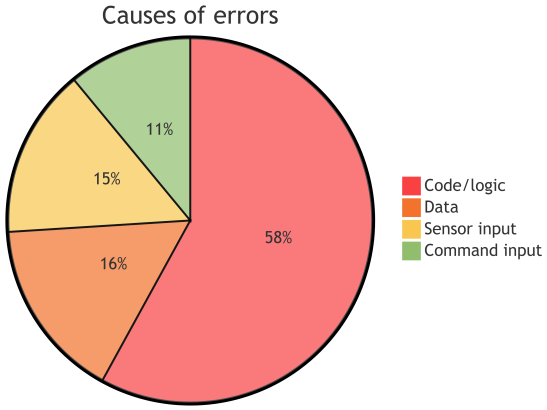

Figure 1: Causes of errors [Pro23]

4.4 Development faults

Development faults is a category which includes all the various faults whose causes are introduced during software development stage. According to research conducted by NASA, majority of errors stem from faults within the code and logic of the afflicted software, followed closely by faults in data [Pro23] (see Figure 1). Both of these sources are under the control of the

software developers, and therefore are prime target of software implemented fault tolerance techniques.

5 Fault-tolerance techniques

This section focuses on the analysis and comparison of theoretical aspects of fault-tolerance, notes certain methods that were already implmenented and tested in practice and outlines methods chosen for our own implementation.

5.1 Single-version

Single-version is a category of techniques which focus on creating a singular, robust implementation of software by integrating safety checks and redundancies directly into the software design process.

This technique emphasizes the detection of errors within the software and the ability to recover from them. Error detection typically involves monitoring the system for unexpected behaviors or inconsistencies, which could signal the presence of a fault. Recovery mechanisms then act to mitigate the effects of these faults.

Handling an error can be done in various ways. Most common would be to backtrack to a saved checkpoint and retry the part of the application where an error was discovered in the hopes of getting the correct result on the subsequent retries. This usually works well with transient faults, but it is likely to fail in the presence of a solid fault.

Drawbacks of single version techniques are primarily the lack of alternatives and fallbacks, should the program fail. Single-version techniques heavily rely on error detection and recovery, which might not always work in practice. The reason single-version techniques are viable is an observation that transient errors are way more common than solid errors [SS87], meaning that single-version is usually enough for noncritical parts of the system.

5.1.1 Modularity

Perhaps the simplest way we can create a more resilient software is to structure it into independent modules. Each module should handle one task and, when possible, not directly rely upon any other modules for its functionality.

A technique commonly utilized to achieve modularity is partitioning, which can be divided into horizontal and vertical partitioning. Horizontal partitioning aims to split the software into independent structural branches communicating through interfaces. Vertical partitioning splits the software in a top-down fashion, where higher level modules are tasked with control logic, while lower level modules do most of the processing [Tor00].

Benefit of partitioning is the ability of software to isolate errors. Provided the software is correctly structured, an error occurring in a single module should not propagate to other modules. Meaning we can use modularity as a way to pinpoint the erroneous parts of software and attempt recovery. If recovery is not possible, the software should still be able to partially function, given that other parts of the software are not influenced by the fault. In most situations, partial functioning of a software is preferable to a complete shutdown.

5.1.2 Error detection

Fault-tolerant single-version application should meet two main criteria: self-protection and self-checking. Self-protection means that the application should be able to protect itself from external corruption by detecting errors in information being passed into the application. Self-checking means that the application component must be able to detect errors within itself and prevent propagation of these errors into other components. These two traits combined can be together considered as the ability of "error detection".

Error detection covers a wide range of techniques used to locate errors and mitigate them. Some common approaches include:

a) checksums and error correction codes (ECC), which embed additional metadata with the actual data in order to verify integrity and attempt to correct corrupted data. This approach allows for some degree of memory corruption mitigation but comes at the cost of memory overhead and additional processing per data-chunk which uses checksum or ECCs.

b) assertion and runtime checks, which perform independent checks on the data during execution which ensures the data matches the expected outcomes at certain checkpoints. This approach also carries with it the additional processing overhead without guarantees that we will be able to catch all errors.

c) watchdog timers, whose main purpose is to catch deadlock states by giving a task a certain amount of time to execute before aborting it.

Error detection is a crucial aspect of single-version application, since we have no alternate version to fall back upon.

5.1.3 Checkpoint and restart

Checkpoint and restart could be considered the basis of some of the more advanced fault tolerance techniques. It involves creating a data checkpoint which the software can be rolled back to before the execution of a process. At the end of the execution, an acceptance check is performed on the process output, if an error is detected rollback is initiated and the process is restarted.

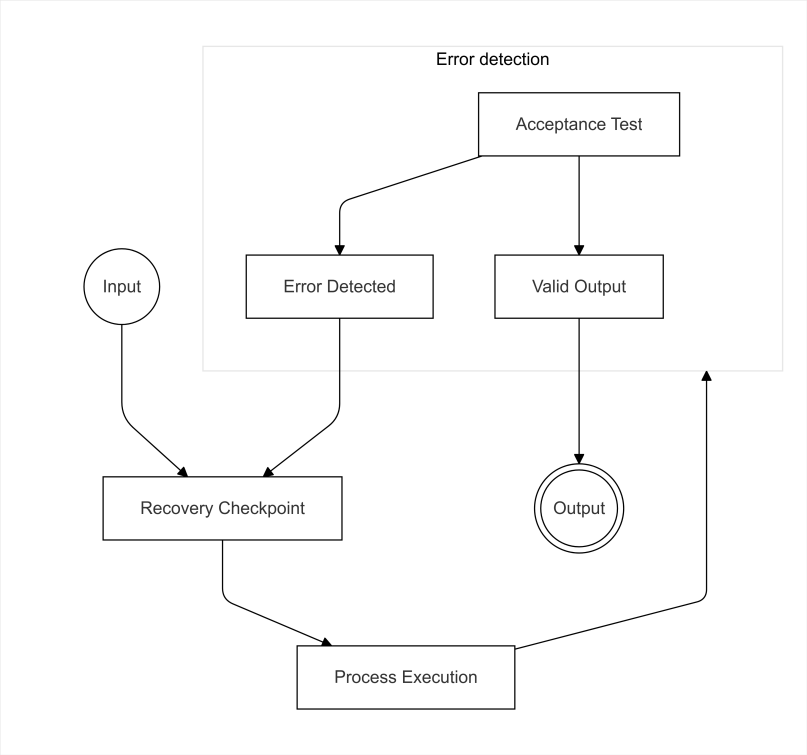

Figure 2: Checkpoint and restart

5.1.4 Considerations for single-version techniques

Single-version fault tolerance techniques are largely easier to develop when compared to the ones discussed in the following section. With this relative

simplicity comes the drawback of being ill-suited for some situations. While these techniques should work well for transient faults that are unlikely to reoccur, a single version fault tolerant system will fail when attempting to deal with persistent faults. Even if an error is successfully detected, single version techniques provide no clear way of dealing with errors that stem from, or are heavily influenced by the design of the system. As an example, if the memory region containing a critical function has been permanently damaged, it matters not how many times we attempt to rerun said function, it will always cause an error. For this reason, single version techniques should be reserved for parts of software which are not mission critical and allow system to function even in their absence.

In order to effectively deal with solid faults, we need to consider multi-version techniques covered in the following section. A lot of these techniques take inspiration from single version techniques, or even directly build off of them.

5.2 Multi-version

Multi-version techniques build on the idea of multiple software versions which all meet the same specifications. These versions are interchangeable in terms of their output, but each version executes differently from the rest, ensuring that no two version share the same failure regions. Multiple versions of the same software are executed either in sequence or in parallel, each utilizing different error detection and recovery methods, to have the highest probability of at least one completing the task successfully.

5.2.1 Recovery blocks

Recovery blocks is a simple form of multi-version programming, expanding upon the idea of "checkpoint and restart". Unlike its single version counterpart, however, recovery blocks does not re-execute the same code again, but instead chooses a different version to try next.

A key advantage of the recovery blocks technique is that, in most cases, the initial version will execute successfully, allowing subsequent versions to prioritize redundancy and safety over performance. This enables the design of backup versions with gradually reduced performance requirements, ensuring robust fallback options without excessive resource consumption.

Since errors are relatively rare compared to normal execution, this approach often achieves an optimal balance of performance and reliability. By prioritizing efficiency in the primary execution path while incorporating progressively resilient alternatives, recovery blocks can provide dependable fault tolerance without compromising system performance in typical operating conditions. This balance makes recovery blocks a practical solution for systems requiring high availability and reliability.

A consideration for recovery blocks is the utilization of a single shared acceptance check for all the versions. This means that the acceptance check must be implementation agnostic and only consider the inputs and outputs of the version. Although this means easier development and potentially less opportunities for faults given less software design, it also fails to take advantage of version-specialized ways errors could be detected in theory.

A considerable drawback of recovery blocks approach is its inherent complexity which creates numerous failure points. Namely, during error detection and state recovery. Since we are performing error detection on a singular execution of a version, we have no way to detect errors which coincide with normal functioning of the software, e.g. random bit flips in used variable. Errors such as these would result in incorrect output from a version, but would be undetectable provided the memory corruption is minimal. Even if we do detect an error, we have no guarantee that the state which was saved is not corrupted as well. We would need to employ various techniques to detect a corrupted recovery checkpoint and ideally implement redundancies to ensure the state can be restored.

5.2.2 N-version programming

N-version programming extends the multi-version technique by running "N" independent versions in parallel or in sequence, hence "N-version" programming. In this approach, each version meeting the same specifications independently performs the task, and the final outcome is determined through a consensus mechanism that evaluates the results from all N executions.

This consensus is usually achieved through a voting algorithm, which aggregates the outputs from each version and selects the result agreed upon by the majority. Selection algorithms are an entire topic of its own covered well by [Alj21].

This voting approach to handling errors is sometimes referred to as **fault masking**, since we are not necessarily concerned with detecting an error,

but rather getting an acceptable output even in the presence of a fault.

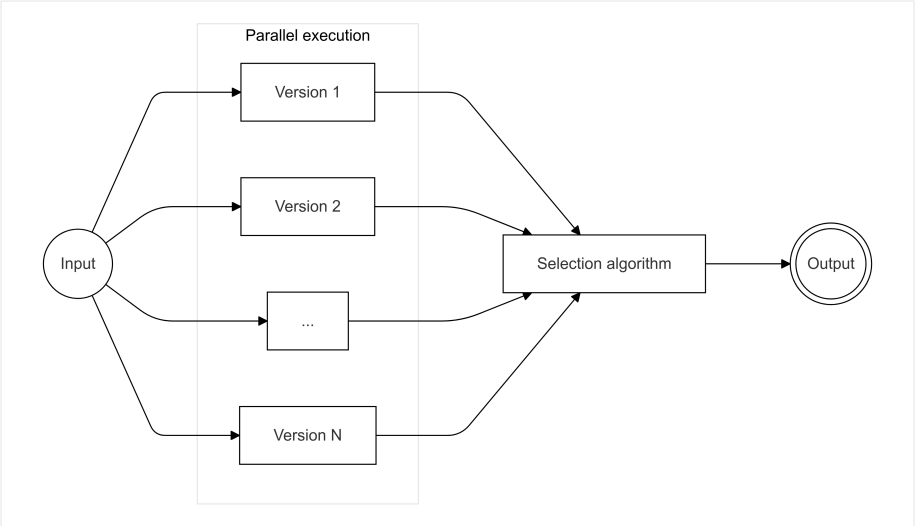

Figure 3: N-Version Programming

The primary drawback of N-version programming is its requirement to execute all of the versions before determining the final output. This can be highly resource-intensive, especially for large or complex tasks, as it requires significant computational power and memory to run multiple versions simultaneously.

For systems with limited resources, such as embedded systems, this approach can be particularly inefficient. The need to allocate resources for each version can strain the system's capabilities, potentially reducing its overall performance and responsiveness. As a result, while N-version programming enhances fault tolerance and reliability, it may not be suitable for applications where resource constraints are a priority or where processing efficiency is critical.

A consideration for N-version programming is the possibility of an error not being a random event, but rather a function of the input variables [LA88]. Therefore, even multiple versions running in parallel could all fail and give erroneous results. This makes the selection algorithm a critical failure point which N-version programming on its own does not address.

5.2.3 N Self-checking programming

N Self-checking programming is an extension of the classic N-version programming, where on top of executing multiple versions, each version also contains its own independent acceptance test or recovery block, before the results are passed to the selection logic. The selection logic then selects the "topmost" possible version that reports a correct output.

A version-specialized acceptance check is an interesting addition, as it provides the opportunity to take advantage of the version implementation details. We can specifically tailor the check to consider the inner workings of the version to detect errors and possibly even correct them before proceeding to the selection stage.

The drawback here is the increase in complexity over the more simple recovery blocks which uses a shared acceptance check, or the simple N-version approach which opts for masking instead. By creating more acceptance checks we are introducing more opportunities for errors to manifest, while also spend more resource on development.

5.2.4 Considerations for multi-version programming

The primary challenge associated with multi-version programming is the significant effort required to develop, test, and maintain several versions of software that perform the same function. This process can be resource-intensive, leading to increased costs, making it unfeasible for smaller projects or for teams with a limited budget.

To achieve effective multi-version programming, each version must be carefully designed to execute the same task while incorporating distinct failure mechanisms. Ensuring that no two versions fail in the exact same way is very difficult and in practice not always possible.

Research has been conducted into other methods that improve upon the aforementioned multi-version techniques, such as the "t/(n-1)-Variant programming" [XR97]. However, the findings do not conclusively prove that the sharp increase in complexity justifies the marginal benefits this or other improved techniques provide.

5.3 Data Diversity

Faults within the data are the second largest cause of errors [Pro23], with that in mind, it is sometimes not enough to execute different version of a software on the same data in order to get an acceptable output. Instead, data diversity might be required to ensure correct execution. Data diversity is an orthogonal method to the previously mentioned design diversity methods. It can be used on its own, or in combination with other fault tolerance methods.

5.3.1 Failure Regions

In many situations only very conditions will result in an error, we call these specific condition edge-cases. Even with thorough testing, there is no guarantee that all edge-cases will be caught during development, since the failure domain can be extremely small. We can take advantage of this, however, by using "reexpressed" input on a subsequent execution of a procedure, since even small adjustments of the input are likely to move it away from the failure domain.

5.3.2 Data Reexpression

Data reexpression is generation of logically equivalent data sets. Any mapping of a program's data that preserves the information content of the data is a valid reexpression algorithm. A simple approximate data reexpression algorithm for a floating-point quantity might alter its value by a small percentage. The allowable percentage by which the data value could be altered would be determined by the application. In applications that process sensor data, for example, the accuracy of the data is often poor and deliberate small changes are unlikely to affect performance. [Kni91]

5.4 Previous work

While the analysis so far was mostly theoretical, this section outlines methods already implemented and tested in practice, giving an insight into what we can expect when implemnting our own fault-tolerance methods.

5.4.1 EDDI

Error Detection by Duplicated Instructions [OSM02b] (EDDI) is a pure software, single-version, technique that enhances runtime error detection by duplicating program instructions during compilation. Each original instruction (master instruction) is paired with a duplicate (shadow instruction) that operates on separate registers or variables. At key synchronization points, such as before memory writes or control transfers, the system compares the results of the master and shadow instructions. Any mismatch indicates an error, triggering a fault handler. EDDI leverages idle resources in superscalar processors by interleaving these redundant instructions to maximize instruction-level parallelism while minimizing performance overhead. It is particularly effective in detecting transient faults, memory corruption, and control-flow deviations without requiring any hardware modifications, achieving over 98% fault coverage in benchmark programs.

5.4.2 CFCSS

Control Flow Checking by Software Signatures (CFCSS), originally proposed in [OSM02a] is a pure software method that embeds control-flow checking logic into a program at compile time. It is capable of detecting faulty branching or jumps within the program.

The program is first divided into basic blocks, each representing a unique node in the control-flow graph. A unique signature is assigned to each node, and additional instructions are inserted to monitor the program's control flow during execution.

At runtime, the control flow is validated using a designated general-purpose register (GSR), which holds the expected signature. When control transfers to a new basic block, a new runtime signature is generated using a lightweight XOR-based function that combines the previous and current signatures. This updated signature is then compared with the expected one at the destination block. If a mismatch is detected, it indicates an illegal control transfer, and execution is redirected to an error handler.

CFCSS is particularly advantageous because it requires no dedicated hardware (e.g., watchdog processors) and can operate even in environments without multitasking support. Its effectiveness was validated through fault injection experiments, which showed that while 33.7% of branching faults went undetected in unprotected programs, only 3.1% remained undetected

when CFCSS was applied, demonstrating an order-of-magnitude improvement in error detection coverage. Although CFCSS introduces moderate code size and execution time overhead (especially in branch-heavy programs).

5.4.3 SWIFT

SWIFT [Rei+05] (Software Implemented Fault Tolerance) is a compiler-based technique for detecting transient faults using software-only transformations, without requiring any specialized hardware. It builds upon earlier methods like EDDI by duplicating instructions and inserting validation checks, but improves upon them with several key optimizations to reduce memory and performance overhead. Unlike EDDI, SWIFT excludes memory from the sphere of replication, assuming protection is already provided by ECC in modern systems. It introduces an enhanced control-flow checking mechanism using software signatures, allowing it to detect illegal control transfers more reliably. Additionally, SWIFT incorporates optimizations such as restricting signature checks to store-containing blocks and eliminating redundant branch validation. These design decisions enable SWIFT to achieve high fault coverage with lower execution and code size overhead. SWIFT demonstrated fault detection rates comparable to hardware-based methods while incurring significantly less overhead than previous software-only techniques.

5.5 Chosen methods

When considering the fault-tolerance methods to implement, the most crucial factor was the feasibility of implementation. Since we are working with a high-level language, certain methods such as EDDI are not possible to effectively implement, since they require implmenentation at assembly instruction level, however, modified or simplified version of these methods may still be implemented.

Ultimately, checkpoint and restart was chosen as the backbone of fault recovery, as it can be easily combined with other methods. CFCSS, with various modifications to make its implmentation fasible in Rust, was chosen to secure the control flow of the program. Finally, recovery blocks was chosen to facilitate multi-version functionality. Additional miscallenouse fault-tolerance methods were also use, such as variable protection through duplication to support the aforementioned methods.

6 Motivation for Rust

The secondary purposes of this thesis is to analyze the suitability of Rust language for the implementation of fault-tolerant software. Historically, C has been the most dominant language in this problem space, but Rust has numerous benefits which make it a strong candidate.

6.1 Safe by design

The goal of the Rust language is to eliminate the most common software bugs at the compiler level. An example of this is Rust's *ownership and borrowing model* - a static code analysis done by the compiler [Foua]. It ensures memory safety of the program by explicity disallowing references to deallocated values, which prevents null pointer dereferencing, and also prevents race conditions by limiting the access to mutable variables. Rust's ownership model is a complex topic and explaning it in full is not within the scope of this thesis, more information can be found in the offical Rust book.

6.2 Robust error handling

Another major benefit of Rust is its approach to error handling. Rust splits errors into two categories: *Unrecoverable* errors immediately terminate the execution of the program using the *panic!()* macro and *recoverable* errors which are return values of a function using the Result enum [Foub].

```
1 fn foo() -> Result<u32, ()> {
2     ...
3 }
4 fn main() {
5     let Ok(res) = foo() else {
6         // Function returned with an error
7     }
8 }
```

Figure 4: Rust - error as a value

Using the approach of *error-as-a-value* we can determine if a function executed successfully by simply checking its return value (see figure 4). This can act as a very quick and rudimentary form of fault tolerance, since it can be easily applied to any function.

6.3 Macro system

Rust has a powerful macro system which in essence works as a code preprocessor acting over the abstract syntax tree. Unlike C macros, which only work as direct text replacement, Rust macros allow for arbitrary modification and generation of code before the compilation step. Using Rust macros, various fault tolerance techniques can be implmeneted. An example of utilizing macros is highlighted in this very thesis, where a Control Flow Checkign using Signature Streams (CFCSS) technique is implemented using Rust macro system.

6.4 Low-level control and C integration

Rust demonstrates comparable functionality to C when it comes to low-level control - example being the ability to define custom heap allocator or the ability to use in-line assembly from within Rust. Since Rust uses LLVM under the hood, its compiler is able to link against existing C code. We can also easily use C functions within Rust by declaring the function signature (see figure 5). This makes for an easy integration with existing C projects making Rust well suited as a higher level extension. This thesis demonstrates the use of Rust with FreeRTOS codebase written in C.

```
1 unsafe extern "C" {
2     fn puts(string: *const c_char, len: usize) -> i32;
3 }
```

Figure 5: Rust - Using C function within Rust

7 Environment

This section outlines the implementation environment, including hardware constraints, system configuration, and software integration. It also describes key modifications made to support our specific use case.

7.1 RISC-V

The implementation was developed for a 32-bit RISC-V core, running in a simulated environment. The simulation features a single RAM region with a total size of 1MB. Of this, 8KB was allocated for the stack and another 8KB for heap memory. These parameters can be adjusted as needed for testing and experimentation.

A key feature of the RISC-V architecture is its interrupt handling mechanism, which distinguishes between two primary types of interrupts. *Software interrupts* are triggered by the program itself, often to perform system calls, while *hardware interrupts* are initiated by hardware events, such as timers or peripherals.

7.2 FreeRTOS and Interrupt Management

The system runs on the FreeRTOS kernel, a lightweight real-time operating system widely used in embedded systems. FreeRTOS is designed for minimal resource usage, making it suitable for constrained environments like our RISC-V simulation.

FreeRTOS uses *tasks* to run user code. Task is a wrapper around user code which include the specific code's context. Tasks can be instantiated at any point during the program's lifetime. FreeRTOS *scheduler* handles the execution of tasks as well as switching between them using the RISC-V hardware interrupt - *Machine Timer Interrupt*.

In addition to timer-based preemption, FreeRTOS also makes use of software interrupts, such as those triggered by the RISC-V ecall instruction. These interrupts are typically used for system-level operations and are routed, along with hardware interrupts, through a shared interrupt handler. In our case, this handler is implemented using a vectored interrupt table to efficiently dispatch control to the correct service routine.

7.3 Rust Integration with FreeRTOS

To enable Rust development on top of FreeRTOS, we used the open-source FreeRTOS-Rust library[1]. This library provides Rust bindings to the FreeRTOS C ABI. Modifications had to be done to ensure compatibility with our simulated environment. Notably, the used library required atomic instructions for certain functionality, however, our simulation did not support atomic operations. Since we are only simulating single core, multithreading is not a concern, as such, we could safely replace all atomic operations with their non-atomic counterparts.

On top of this, RISC-V Rust library[2] was used to facilitate the compilation of Rust into a RISC-V acceptable binary. This library provided functionality such as designating the program entry point, designating the exception and interrupt handler functions and provided functions to work with RISC-V specific registers. However, the used library also added significant performance overhead in the form of BSS clearing segment before the main program start. Clearing BSS is useful to ensure all variables are initialized to 0 if a default value is not assigned. However, because Rust does not fundamentally allow for the use of non-instantiated variables this step was not necessary for our use case. By removing this segment we saved significant computation time, reducing the possible area for faults to manifest.

[1] `https://github.com/lobaro/FreeRTOS-rust`
[2] `https://github.com/rust-embedded/riscv`

8 Implementation

The implementation section outlines some selected methods which were implemented using Rust and FreeRTOS and tested on a RISC-V core simulation. The implemented methods were modified to make them fasible to implement in a high level language.

8.1 Checkpoint and restart

Checkpoint and restart system is the backbone of our fault-tolernace framework, facilitating recovery. It can be used on its own to provide basic level of redundancy, or it can be combined with other techniques to boost their fualt-tolerance potential.

8.1.1 Creating a checkpoint

```
asm!(
    // Storing registers in a static variable
    "lui   t0, %hi(CHECKPOINT)",
    "addi  t0, t0, %lo(CHECKPOINT)",
    "sw x1,   4(t0)",
    "sw x2,   8(t0)",
    ...
    "sw x31,124(t0)",
    "call set_checkpoint",
    // Restoring registers to their previous state
    "lui   t0, %hi(CHECKPOINT)",
    "addi  t0, t0, %lo(CHECKPOINT)",
    "lw x1,   4(t0)",
    "lw x2,   8(t0)",
    ...
    "lw x31,124(t0)",
)
```

Figure 6: Rust - Creating checkpoint

Checkpoint is generated by creating a *checkpoint context* during which the current state of the application is stored. Namely, the general purpose registers (x1 - x31 in RISC-V 32bit) are stored into a static memory location before the execution of a *checkpoint block* (the part of code protected by the

checkpoint). Additionally, a return address is stored along with the general purpose registers.

Higher-level programming languages do not directly provide the functionality to manipulate register values. This means assembly instructions have to be used to facilitate the low level access as seen in figure 6.

In order to set the return address, we utilize a trick (shown in figure 7) whereby calling a function we now have access to the instruction immediatelly following the function call (figure 6 line 11) saved in the return address register (RA). We store the value of this register along with the registers in the checkpoint variable.

```
1 fn set_checkpoint() {
2     unsafe {
3         asm!(
4             "sw ra, ({0})",
5             in(reg) &(CHECKPOINT.ret_addr),
6         );
7     }
8 }
```

Figure 7: Rust - Set checkpoint

8.1.2 Storing the checkpoint

As mentioned in section 8.1.1, the checkpoint information is stored in a static memory location. This might seem as an arbitrary decision given that storing the checkpoint on the stack or the heap are also an option. Each of the listed approaches comes with its own issues. Using a static memory location limits the number of checkpoints that can be stored. Using the stack for checkpoint storage can be very unreliable as stack corruption can easily occur should the stack pointer manifest a fault. And using the heap is generally slow and prone to error stemming from pointer corruption or premature memory freeing which could occur in the prsence of faults.

Our implementation uses a hybrid approach of using a static memory location for the storage of the *primary checkpoint* (the last checkpoint that was set), while pushing any older checkpoints on the stack. This heightens the chance that at least the primary checkpoint will be available. Once the primary checkpoint context ends, the previous checkpoint is popped from the

stack and can be used again. This allows for infite nesting of checkpoints, provided we have enough memory.

8.1.3 Returning to the checkpoint

After setting the checkpoint, the program execution continues as per usual, until either the checkpoint block finishes successfuly, or an error is detected. An error can be detected in two way, an exception is detected by the processor, or an error is detected via checks inserted into the checkpoint block.

If the processor detects an error the exception handler subroutine is called. We can hook into this subroutine and use it to restore the stack pointer to the pre-checkpoint block state and then perform a jump back to the saved return address.

If an error is detected via checks inserted by the programmer, checkpoint restart can be triggered by simply returning an error (Err()) from within the checkpoint block.

Both forms of error detection and returning to checkpoint result in the restoration of the program context to the program state in which the checkpoint function was called. After the context was restored, checkpoint block will be attempted again if retries are allowed, otherwise the entire checkpoint context returns an erroneous value.

```
if let Err(Error::MaxRetriesReached(n)) = checkpoint(2, || {
    // Checkpoint block here
    ...
    Ok(())
}) {
    println(&format!("Max retries reached: {}", n));
}
```

Figure 8: Rust - Using the checkpoint system

8.1.4 Overhead

The implemented checkpoint system incurs minimal instruction overhead. Additional instructions are only inserted when creating the checkpoint context and just before the end of the checkpoint context. This comes out to 79 additional instructions to save and restore program state, plus approximately

300 instructions to save and restore old checkpoint and perform various error checks, without factoring in any compiler optimizations.

This overhead is constant for the entire checkpoint block, irrespective of the checkpoint block size. As such, the ratio of overhead instructions to the checkpoint block instructions is solely determined by the size of the protected checkpoint block.

8.1.5 Considerations

Checkpoint system only facilitates basic restart and retry functionality. It can be used on its own when only basic level of protection is required, however, in most cases, checkpoint and restart system is meant to be used as a framework to build upon. Namely, the multi-version techniques outlined in later sections all use checkpoints to implement version switching.

Our implmenetation of the checkpoint and restart system only saves the minimal required state, which in the case of RISC-V, is the general registers. This limits the use of the checkpoint system as other memory regions such as stack or heap are not stored. Careful consideration needs to be given to the code block protected by the checkpoint context. If the checkpoint body has any side-effects, such as modifying global variables, variables outside of the checkpoint block scope or writing to heap, these alterations will not be reversed in the case of an error and subsequent checkpoint return.

This problem can be avoided by explicitly creating a copy of any outside variable which needs to be modified within the checkpoint block. The copy will then be used within the checkpoint block and the original variable will only be overrided if the checkpoint context returns successfully. More on this in section 8.3.

Another important thing to mention is that our implementation does not save registers other than the general purpose registers. This means special registers such as *mtvec*, *mcause* or *mstatus* are not protected by the checkpoint system. The fault-insertion simulation does not insert faults into these registers, as such their directly protection is not needed. However, errors can still manifest (as if they occuerd in the special registers) in ALU after the special registers are read for processing.

8.2 CFCSS

Unlike the usual implementation, which works over assembly instructions, or LLVM's intermediate representation (IR), our implementation takes advantage of Rust's macro system to insert CFCSS directly into program code. An algorithm originally designed for LLVM IR is used, with modifications to make it suitable for our purposes [Jef].

Macro is used to designate a *module* as CFCSS protected, each function within the module being designated as a *block* (b_n). A graph is then generated at compile time representing the allowed control flow within the protected module. Each block within the graph is assigned a random unique signature (s_n) and signature difference (d_n), which is calculated by XORing ($\oplus$) the current block's signature and the signature of its predecessor as seen in equation 1.

$$d_n = s_n \oplus s_{n-1} \tag{1}$$

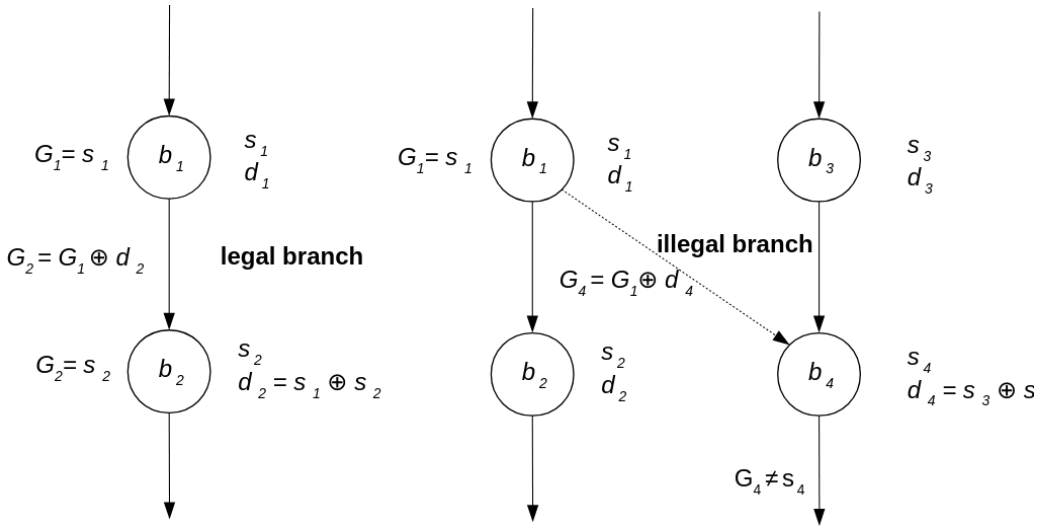

Figure 9: Basic CFCSS [Jef]

A runtime signature G_n is also created for the given protected module, initially set to $G_0 = 0$. This signature is updated at runtime before the execution of every block by XORing it with the signature difference of the destination block (equation 2). Using XOR to update the runtime signature

is favorable, since XOR operation can be undone by simply being repeated, which is desirable when returning from a block.

$$G_n = G_{n-1} \oplus d_n \tag{2}$$

To ensure correct control flow, the newly calculated runtime signature must be equal to the function signature, otherwise an invalid branching occured.

$$G_n = s_n \tag{3}$$

This approach is, however, insufficient in the presence of so-called *fan-in* block. If both b_1 and b_3 have an edge to b_4 a problem arises where (provided that $s_1 \neq s_3$) either $G_4 = G_1 \oplus d_4$ is true or $G_4 = G_3 \oplus d_4$.

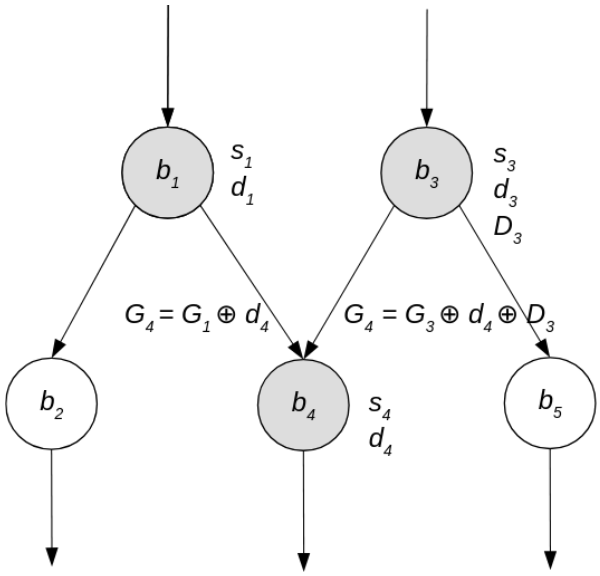

Figure 10: CFCSS with fan-in block [Jef]

We need to introduce another constant called signature adjuster (D_n) caluclated for each predecessor of a fan-in node, and apply it while updating the runtime signature. This gives us the actual equation (4) for updating runtime signature.

$$G_n = G_{n-1} \oplus d_n \oplus D_n \tag{4}$$

8.2.1 Macro implementation

The Rust macro generates static variables (G_n, D_n, s_n, d_n) during compile time and inserts additional code before and after the actual function code witin the protected module (see figure 11). The instructions inserted after the function block are equivalent to the ones inserted before, with the exception of restoring *last adjuster* from stack instead of updating and storing it.

```
    RUNTIME_SIGNATURE ^= #dif_name;
    if is_fan_in {
        RUNTIME_SIGNATURE ^= RUNTIME_ADJUSTER;
    }
    if #sig_name != RUNTIME_SIGNATURE {
        return_to_checkpoint(ERROR_MSG);
    }
    if func_has_adjuster {
        RUNTIME_ADJUSTER = #adj_name;
    }

```

Figure 11: Rust - CFCSS instructions inserted before function block

This macro can be applied as shown in figure 12 by specifying a module as *#[cfcss_root]*. An entry funciton is designated with a special macro *#[entry]*, this macro embeds additional instruction that reset the runtime signature and adjuster of the module, allowing for repeated call of the entry point. An entry function can only be a function that is not called from within another function in the protected module.

```
#[cfcss_root]
pub mod protected_block {
    fn function_0();
    fn function_1();
    #[entry]
    pub fn entry_function() {
        function_0();
        function_1();
    }
}
```

Figure 12: Rust - Using the CFCSS macro

8.2.2 Overhead

Embedding new instruction during compile time will understandably have performance impact on the execution of the program. Each function will incur a execution overhead, as multiple operations and checks must be carried out before the actual function body executes. Approximately 50 additional assembly instructions are inserted before the function block and 50 more after the function block, without using any compiler optimizations.

However, due to the nature of the implementation, the developer is in full control of just how much overhead is incurred. Since the number of additional instructions is constant per function definition, we can either split our protected module into fewer function to have less overhead, but at the cost of less fault-tolerance. Or we can fragment our code into as many function blocks as possible, and thereby increasing the number of control flow checks performed.

8.2.3 Considerations

Since our implementation of CFCSS utilizes a purely code based approach it natuarally comes with several limitations. Unlike the traditional approach, which implements CFCSS at IR level by extending the LLVM pipeline we do not have access to the underlying instructions, which limits the possible density of the control flow checks. An IR approach is able to precisely insert instructions after every branch and jump operations, and also has more granular control over the actual instructions being inserted. This means our implementation is more prone to control flow errors. Although not within the extent of this thesis, further improvements could be made by directly extending Rust's LLVM pipeline and integrating a CFCSS method at IR level.

Additionally, CFCSS is incompatible with the aforementioned checkpoint and restart system. As explained in section 8.1.5, our checkpoint and restart system does not restore context of static variables, which CFCSS relies on for proper functioning. As such, checkpoints cannot be used within a CFCSS protected module, since if a checkpoint restart were to happen, the runtime signature and adjuster would not be reset to their proper pre-checkpoint values.

8.3 Protection of variables

Various unforseeable circumstances can lead to the corruption of program variables, from direct corruption of registers storing the variables to indirect side-effect stemming from errors within function that work with said variables. This section focuses on various techniques implemented to ensure the reliability of critical program variables with variying degree of protection depending on the use case.

8.3.1 Copy and commit

Copy and commit is a method inspired by a common practice in database transaction handling, where a temporary copy of the data is modified, and changes are only applied to the original data upon an explicit commit. This pattern is particularly useful when modifications must be validated or selectively applied, reducing the risk of unintended side effects. In the Rust example shown in Figure 13, a CopyCommit wrapper is used to hold a temporary copy of the value. Inside the bar function, the value is incremented, but the change only takes effect on the original data when commit() is called. This allows for greater control over when and how mutations are finalized.

```
1 fn bar(mut a: CopyCommit<u32>) {
2     *a += 1;
3     a.commit();
4 }
5
6 fn foo() {
7     let mut a = 1;
8     bar(CopyCommit::new(&mut a));
9 }
```

Figure 13: Rust - Copy and commit example

While this method gives us more control over when variables are modified and helps prevent unwanted side-effects, it inherently does not provide any fault tolerance against direct corruption of variables. As such, it should only be reserved for non-critical parts of the program, and or combined with other variable protection techniques.

8.3.2 Multiple redundant variables

Another way we can secure a variable is creating multiple copies of it and applying any modification to all the copies. The modified copies are then copared against each other to detect any mismatch. With two copies of a variable, we can detect an error but cannot correct it, since we cannot tell which one of the copies is faulty. With three copies, we can detect an error but also perform correction, if at least two copies of the variable are equal. The more copies we create the more likely we are to have a majority of the variables be equal, therefore the more confident we can be in our fault-tolerance.

```
// Creates 3 identical copies
let mut a = MultiVar::new(0);

// Applies the edit function to each of the copies and
    performs equality check
if let Err(e) = a.edit(|ptr_a| {
    *ptr_a += 1;
}) {
    // Error while updating variable
    ...
}
```

Figure 14: Rust - Multiple variable redundancy

In our implementation, a MultiVar wrapper around a variable is used to automatically instantiate three separate copies of the variable as seen in Figure 14. By calling the edit function on the wrapper, we map any modification to all the copies of the variable. The edit function also includes an internal checks which tries to correct the variables in a case of a mismatch. If correction is not possible this function returns an error.

8.3.3 Checksum

Checksum is a technique to ensure a long-lived variable has not been corrupted during the execution of the program. If we need to store a variable for prolonged amount of time we can store a checksum along with it. The checksum is calculated by splitting the stored data into chunks of 32-bits each

and XORing these chunks with one another, thereby generating an unique signature.

Every time the protected variable is modified the checksum is updated by recalcuating the signature. Before the variable data is read again at a later point in the program, a checksum is recalculated and compared with the stored checkum. If the checksums match the variable has not been changed since the last proper modification. In the case of a checksum mistmatch, however, the variable is likely to have been unexpected modified or directly corrupted.

This technique is best utilized for global variables, since they live for the duration of the program and their correctness is paramount for the proper application functioning.

8.3.4 Considerations

The implemented methods for variable protection can give us more confidence in the correctness of the program variables, however, no matter the degree of protection and fault-tolerance implemented, faults can always manifest. Since data corruption can happen on the ALU directly, even if our stored variable is correct, the actual value being read and processed by the CPU can appear to be faulty. As such, the methods listed above are not sufficient on their own to ensure we get the correct result even if the program executes seemingly without fault.

8.4 Multiversion

The implemented multiversion techniques are clumination of most of the techniques discuessed so far.

8.4.1 Recovery blocks

Recover blocks have been implemented by taking advantage of the checkpoint system. By wrapping each call to a different version function in a checkpoint context we ensure that the function will return even in case of an error. We can then check the return value of the checkpoint block to determine if the version executed successfully. As seen in Figure 15, if the version returns successfully, we immediatelly return the value, otherwise we try another version. The checkpoint is set to 0 retries, meaning each one of the versions (fib_v1, fib_v2, fib_v3) will only be tried once.

```
pub fn fib_rec_blocks(x: u8) -> Result<u32, &'static str> {
    // Version 1
    if let Ok(res) = checkpoint(0, || fib_v1(x)) {
        return Ok(res);
    }
    // Version 2
    if let Ok(res) = checkpoint(0, || fib_v2(x)) {
        return Ok(res);
    }
    // Version 3
    if let Ok(res) = checkpoint(0, || fib_v3(x)) {
        return Ok(res);
    }
    Err("Versions exhausted")
}
```

Figure 15: Rust - Recovery blocks

Important thing to mention is the lack of any verification of the result. In the case of recovery blocks, we only care about the function executing without obvious errors. The return value could be corrupted by an undetectable bit flip somewhere in the function variables. Recovery blocks alone does not deal with this issue.

9 Testing

Testing of the implementation was done on Hardisc[3] (simulated using ModelSim) - a hardened RISC-V core with the ability to statically insert faults during runtime of the program. The form of the fault is a single bit-flip. The frequency of the fault insertion as well as the regions where fault are inserted can be changed to simulate different environments and circumstances. For the purpose of narrowing down the scope of the thesis, faults were only inserted in the following regions:

General Purpose Registers (GPR) Registers used for general computation or temporary storage during instruction execution.

Register File Control (RFC) Controls access to the register file; involved in reading/writing registers.

Arithmetic Logic Unit (ALU) The part of the CPU that performs arithmetic and logical operations.

Multiply-Divide Unit (MDU) Specialized unit for performing multiplication and division operations.

Data Path (DP) The part of the CPU where data is processed, including registers, ALUs, and buses.

Trap Interrupt Registers (TP) Handles traps (exceptions) and interrupts in the CPU.

Notably, RAM was exempt from fault insertions during our testing. Should a non-transient error occur in the part of RAM which holds the program instructions there would be no way to properly recover from it using software-only fault tolerance. Usually, RAM and CPU duplication is used, effectively running the same program twice in parallel with set synchronization points. This method is outside of the scope of software implemented fault tolerance.

The methods outlined in the implementation section, however, are in theory capable of detecting and recovering from certain faults in RAM. As mentioned in section 8.3.3, checksums are capable of detecting corruption in variables, this includes variables stored in the RAM. Similarly, using multi-version programming can provide tolerance even in the case of non-transient error occuring in one of the versions. Due to general unreliability of this approach, however, the testing on RAM fault insertions was not conducted.

[3] `https://github.com/janomach/the-hardisc`

9.1 Benchmarks

A set of benchmarks was selected to evaluate the effectiveness of the implemented fault tolerance techniques. These benchmarks primarily consist of common arithmetic operations and data manipulation tasks representative of typical embedded system workloads.

In addition to standard benchmarks, one unique benchmark - *Nested Function Calls* (NFC) was included. This benchmark involves a series of functions invoking one another in a nested manner, creating intertwined control-flow graph. It was specifically chosen to assess the performance impact of CFCSS, which is particularly sensitive to complex function call hierarchies.

Benchmark programs report results in one of two ways, successful execution (0) and unrecoverable error (1). Any other return status code is considered an unknown error. The reliability of the fault tolerance was measured as a sum of successful executions and correctly reported unrecoverable errors divided by the overall number of executions (25 per benchmark). Each benchmark was given equal amount of time to execute (100 000 CPU cycles), if the time thresholdt is reached the program ends in a timeout. Notably, a timeout does not necessarily means the benchmark itself did not execute correctly. It is very common for the FreeRTOS scheduler to be the source of an error leading to a timeout. As such, in the case of a timeout, the output logs are examined to determine if the benchmark succeeded.

Fib sequence	**Correct**	**Errors Reported**	**Fault tolerance**	**Timeouts**
Unprotected	10	5	60%	8
Protected	14	9	92%	3

Table 1: Fibonacci sequence benchmark statistics

As seen in Table 1 the overall fault tolerance of the protected benchmark went up significantly. The increase in correct results was not exceptionally high likely due to the fact that errors mostly occurred outside of the variables and registers that would directly alter the computation result. Rather, most faults resulted in errors manifesting in various supporting subroutines. Evidence of this is the nearly doubled number of reported errors when fault tolerance was enabled.

Similar trend can be observed in the case of Matrix benchmark (see Table 2). The correct results difference is even lower, likley due to the fact that

Matrix dot product	Correct	Errors Reported	Fault tolerance	Timeouts
Unprotected	16	0	64%	9
Protected	18	7	100%	0

Table 2: Matrix multiplication benchmark statistics

the matrix dot product test requires less CPU instructions, and therefore is a relatively smaller area for faults to manifest. However, the overall error reporting went up significantly, proving our fault tolerance methods are good at catching error that would otherwise result in unexepcted program termination.

TODO: More benchmarks...

9.2 Performance impact

Performance impact of the implemneted fault-tolerance methods was measured without employing any compiler optimizations on a fault-free version of the simulated core. In the presence of faults, the effective performance could vary unexpected, influenced by the randomness of the fault insertion. As such, there is not much point in measuring performance in the presence of faults.

The test were measured in CPU cycles using the RISC-V *mcycle* and *mcycleh* registers.

Test	Unprotected (cycles)	Protected (cycles)	Increase (%)
Fibonacci	30734	34983	13.82%
Matrix	9871	11133	12.78%
NFC	9889	13126	32.73%
Bubblesort	15228	16773	10.14%
CRC	8278	10614	28.21%

Table 3: Execution time comparison between unprotected and protected tests

As expected, looking at Table 3 we see an increase in CPU cycles it takes to execute the protected version as opposed to the unprotected one. Fibonacci, Matrix and Bubblesort (group one) test only incurred rougly 12% increase, while both NFC and CRC (group two) incurred much more substantial increase. The main difference between these two groups of tests is their flow graph complexity. Group one has linear control flow and mostly consists

of 1-4 function call, meaning very little time is spent on CFCSS checking. Group two has more complex control flow resulting in a lot of CFCSS checks.

9.3 Size Impact

To evaluate the memory overhead introduced by the fault tolerance mechanisms, we compared the size of the compiled binaries with and without protection under two conditions: with no compiler optimizations and with standard release optimizations enabled. As expected, enabling protection introduces additional instructions and data structures, resulting in larger binary sizes. The impact is more pronounced in the unoptimized build, while optimized builds show a more moderate increase due to compiler optimizations reducing overall code size.

Compiler optimizations?	**Unprotected**	**Protected**	**Size increase**
NO	23164	32847	41.8%
YES	8467	9216	8.84%

Table 4: Binary size increase with and without protection

Final size of the binary is an important consideration for fault tolerance. The larger the binary the more instructions will be executed, on average, to run our program. This increases the execution time and the probability of errors manifesting.

10 Conclusion

11 Resumé

References

[Alj21] A Aljarbouh. "Selection of the optimal set of versions of N-version software using the ant colony optimization". In: *Journal of Physics: Conference Series* 2094.3 (Nov. 2021), p. 032026. DOI: 10.1088/1742-6596/2094/3/032026. URL: https://dx.doi.org/10.1088/1742-6596/2094/3/032026.

[Avi+04] A. Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[Foua] The Rust Foundation. URL: https://doc.rust-lang.org/1.8.0/book/ownership.html.

[Foub] The Rust Foundation. URL: https://doc.rust-lang.org/book/ch09-00-error-handling.html.

[Jef] Benjamin James Jeffrey Goeders. URL: https://coast-compiler.readthedocs.io/en/latest/cfcss.html.

[Kni91] John C. Knight. *Software fault tolerance using data diversity*. Work of the US Gov. Public Use Permitted. 1991. URL: https://ntrs.nasa.gov/citations/19910016332.

[LA88] J.H. Lala and L.S. Alger. "Hardware and software fault tolerance: a unified architectural approach". In: *[1988] The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1988, pp. 240–245. DOI: 10.1109/FTCS.1988.5326.

[OSM02a] N. Oh, P.P. Shirvani, and E.J. McCluskey. "Control-flow checking by software signatures". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 111–122. DOI: 10.1109/24.994926.

[OSM02b] N. Oh, P.P. Shirvani, and E.J. McCluskey. "Error detection by duplicated instructions in super-scalar processors". In: *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75. DOI: 10.1109/24.994913.

[Pro23] Lorraine Prokop. *Historical Aerospace Software Errors Categorized to Influence Fault Tolerance*. Work of the US Gov. Public Use Permitted. 2023. URL: https://ntrs.nasa.gov/citations/20230001295.

[Rei+05] G.A. Reis et al. "SWIFT: software implemented fault tolerance". In: *International Symposium on Code Generation and Optimization*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[SS87] Schuette and Shen. "Processor Control Flow Monitoring Using Signatured Instruction Streams". In: *IEEE Transactions on Computers* C-36.3 (1987), pp. 264–276. DOI: 10.1109/TC.1987.1676899.

[Tor00] Wilfredo Torres-Pomales. *Software Fault Tolerance: A Tutorial*. Work of the US Gov. Public Use Permitted. 2000. URL: https://ntrs.nasa.gov/citations/20000120144.

[XR97] Jie Xu and B. Randell. "Software fault tolerance: t/(n-1)-variant programming". In: *IEEE Transactions on Reliability* 46.1 (1997), pp. 60–68. DOI: 10.1109/24.589928.