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**

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

2.1 Common terms

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 ability to avoid service failures in the presence of faults.

3 Types of faults

In order to be able to talk about fault tolerance, 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 the hardware and the software, which is why we can further classify them as **hardware faults** and **software faults** [Avi+04].

3.1 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 solid fault.

3.2 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 4.2.

3.3 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 single-event upsets (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.

Memory corruption is not always catastrophic. While it can result in complete system failure and unrecoverable states, it is just as likely to manifest as small, hard-to-detect errors. These subtle corruptions might not immediately disrupt the software's functioning but can lead to unpredictable behavior over time.

3.4 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]. Both of these sources are under the control of the software developers, and therefore are prime target of software implemented fault tolerance techniques.

To ensure that software remains robust and as free of errors as possible, there are several effective strategies. One promising approach we will look at is the use of memory-safe programming languages, specifically Rust. Rust is a modern language that has gained traction for its safety features, particularly in system programming and embedded applications. It ensures high performance through zero-cost abstractions and introduces a memory ownership model, which reduces memory-related errors such as null pointer dereferencing and data races. This model makes Rust particularly well-suited for low-level and resource-constrained environments, where reliable memory management is crucial. (https://docs.rust-embedded.org/book/)

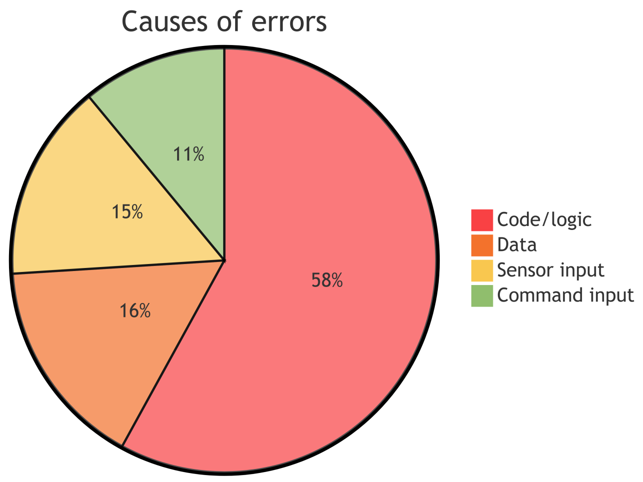

Figure 1: Causes of errors [Pro23]

4 Analysis

4.1 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 [Rei+05].

Drawbacks of single version techniques are primarily the lack of alternatives and fallbacks, should the version 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 sometimes enough for noncritical parts of the system.

4.1.1 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, or be relied upon by other modules.

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

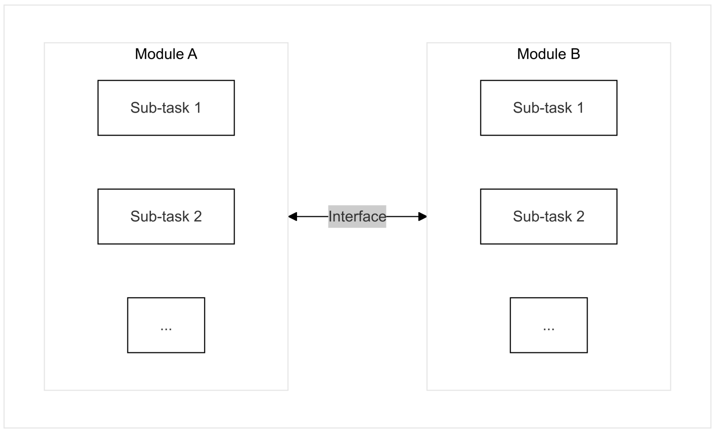

Figure 2: Horizontal partitioning

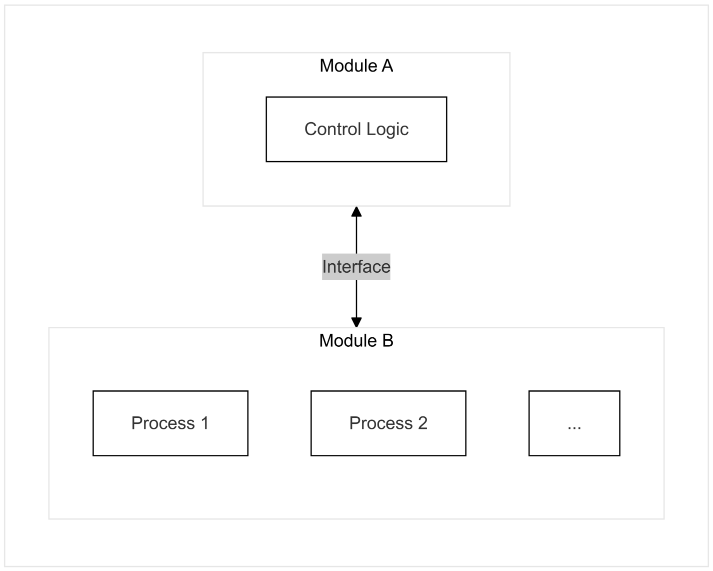

Figure 3: Vertical partitioning

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.

4.1.2 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.

4.1.3 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.

4.1.4 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

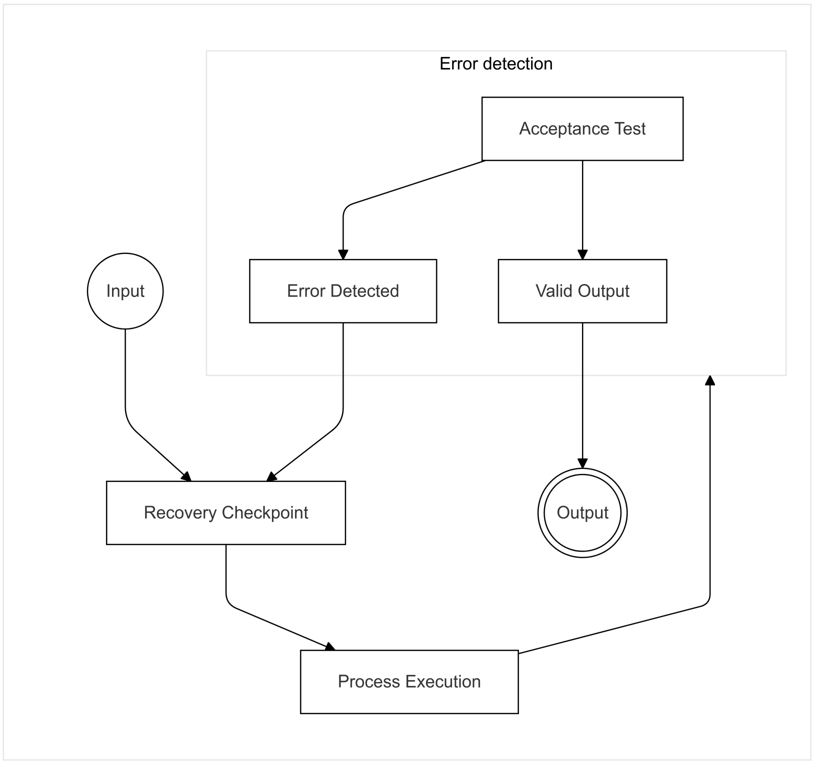

Figure 4: Checkpoint and restart

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.

4.2 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.

4.2.1 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.

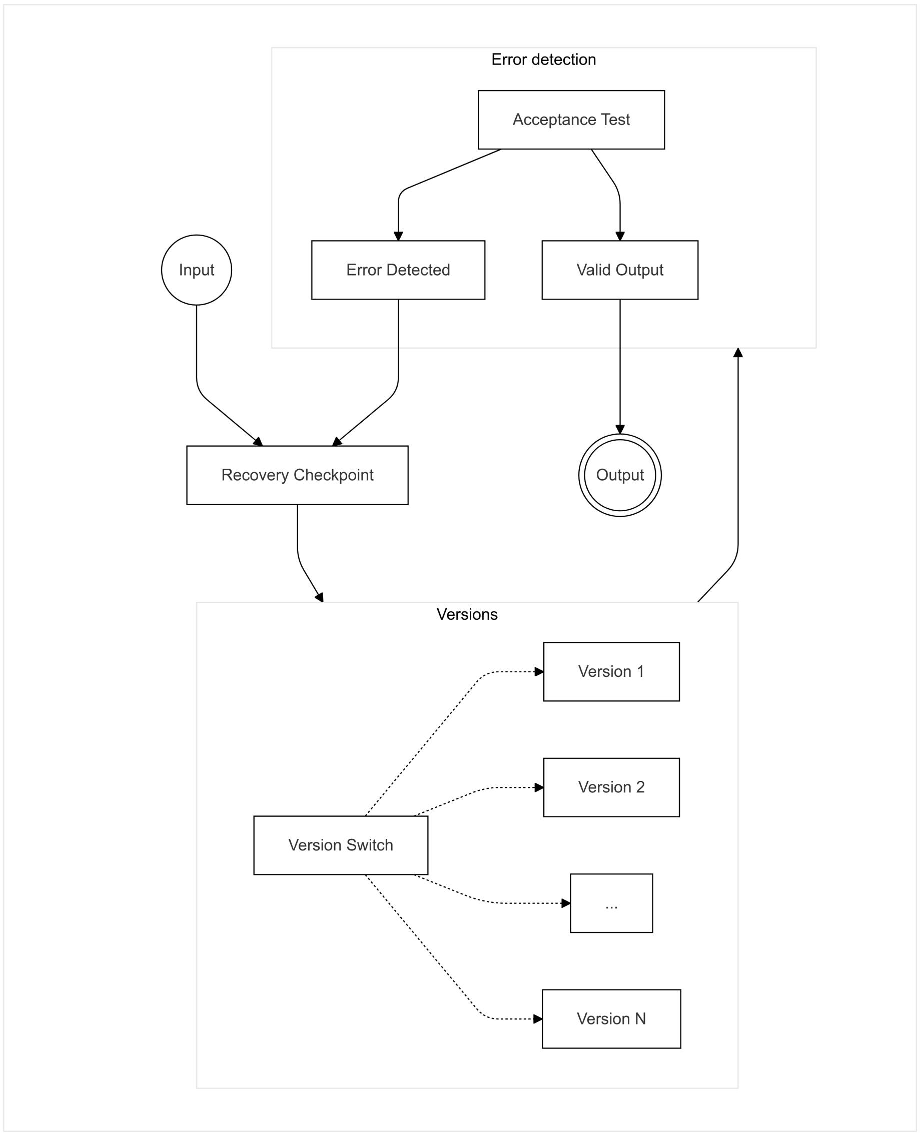

Figure 5: Recovery Blocks

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.

4.2.2 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.

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

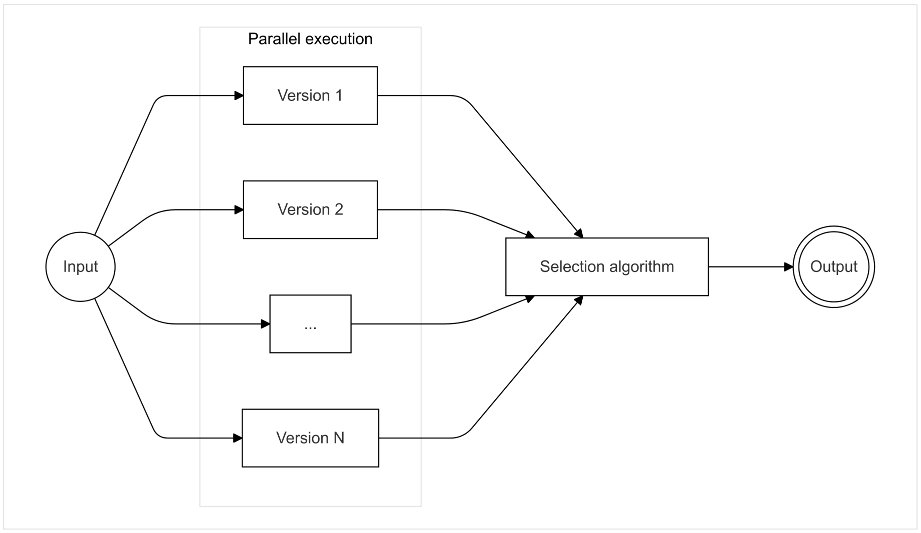

Figure 6: N-Version Programming

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.

4.2.3 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.

4.2.4 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.

subsectionData 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.

4.2.5 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.

4.2.6 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]

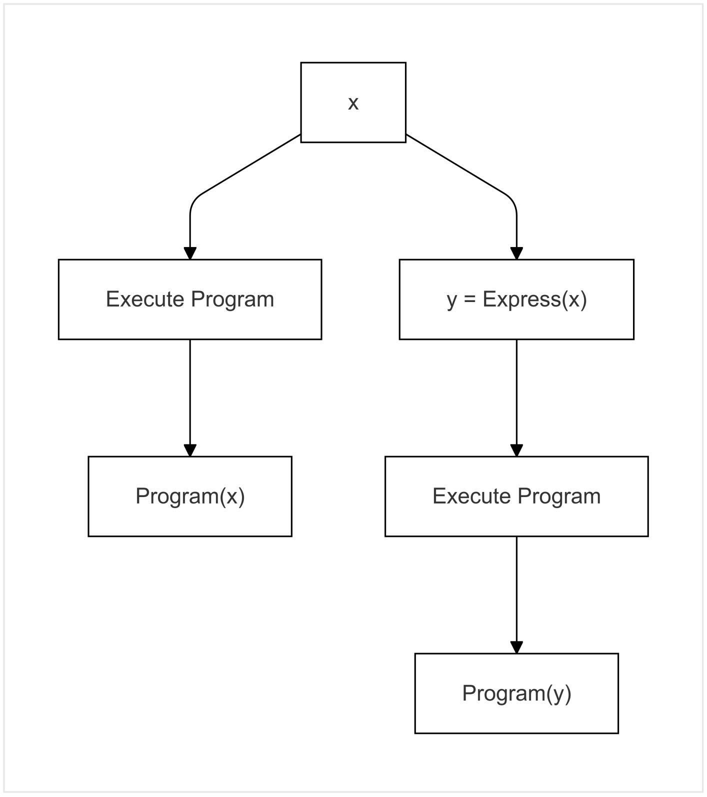

Figure 7: Data Reexpression

4.3 Miscellaneous

(Name is subject to change.) This category is dedicated to fault tolerance methods which do not strictly fall into any of the aforementioned categories but are still worth consideration.

Framework can be considered any fault tolerance technique which instead of aiming to directly create fault tolerant software provides a wrapper or tooling to help create fault tolerant software. Benefit of frameworks is the fact that the software developers do not have to concern themselves with fault tolerance, since the overarching framework is responsible for that. Example might be an OS designed to be fault tolerant.

Another interesting example is specialized compiler which embeds fault tolerance into the software during compilation, independently of the actual software's purpose. Example of this approach is **"Control flow checking by software signature"** published in [OSM02].

5 Motivation for Rust

One of the main 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.

5.1 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.

5.2 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].

```rust
fn foo() -> Result<u32, ()> {
    ...
}
fn main() {
    let Ok(res) = foo() else {
        // Function returned with an error
    }
}
```

Figure 8: 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 8). This can act as a very quick and rudimentary form of fault tolerance, since it can be easily applied to any function.

5.3 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.

5.4 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 9). 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.

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

Figure 9: Rust - Using C function within Rust

6 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. Some examples of the implementation code are also provided where fitting.

6.1 Environment

6.1.1 RISC-V

The implementation was designed to run on a 32-bit simulation of RISC-V core. This posed an issue with

6.1.2 Interrupt and exception handling

```rust
#[riscv_rt::exception(Exception::InstructionFault)]
fn instruction_fault(trap: &mut riscv_rt::TrapFrame) -> ! {
    unsafe { return_to_checkpoint("InstructionFault") }
}
```

Figure 10: Rust - Exception handler example

6.1.3 FreeRTOS integration

In order to integrate Rust with FreeRTOS an open-source library called FreeRTOS-Rust was used. This library provided basic...

6.2 Checkpoint and restart

Checkpoint and restart system is the backbone of our fault-tolernace framework. 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. A lot of other methods, namely ones outlined in the Multiversion section use checkpoint and restart at their core.

6.2.1 Creating a 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 11.

```
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 11: Rust - Creating checkpoint

In order to set the return address, we utilize a trick (shown in figure 12) whereby calling a function we now have access to the instruction immediatelly following the function call (figure 11 line 11) saved in the return address register (RA). We store the value of this register along with the registers in the checkpoint variable.

```
fn set_checkpoint() {
    unsafe {
        asm!(
            "sw ra, ({0})",
            in(reg) &(CHECKPOINT.ret_addr),
        );
    }
}
```

Figure 12: Rust - Set checkpoint

6.2.2 Storing the checkpoint

As mentioned in section 6.2.1, 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.

6.2.3 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 outlined in section 6.1.2 is triggered. Stack pointer is restored to the pre-checkpoint block state and the program jumps 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 13: Rust - Using the checkpoint system

6.2.4 Performance impact

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.

6.2.5 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 6.4.

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.

6.3 CFCSS

TODO: Explain why CFCSS is important (e.g. how branching errors occur)

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.

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

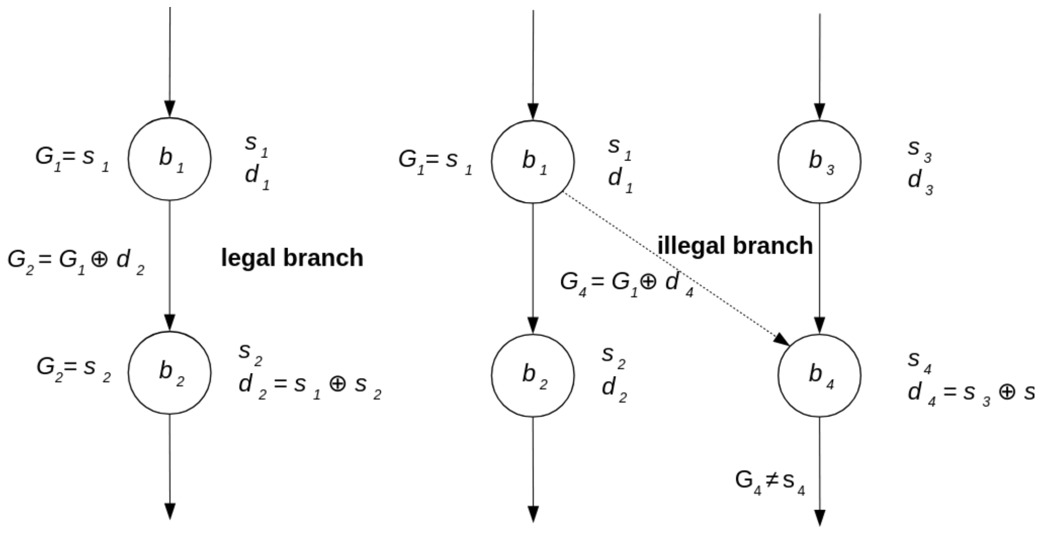

Figure 14: Basic CFCSS [Jef]

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 $G_4 = G_3 \oplus d_4$.

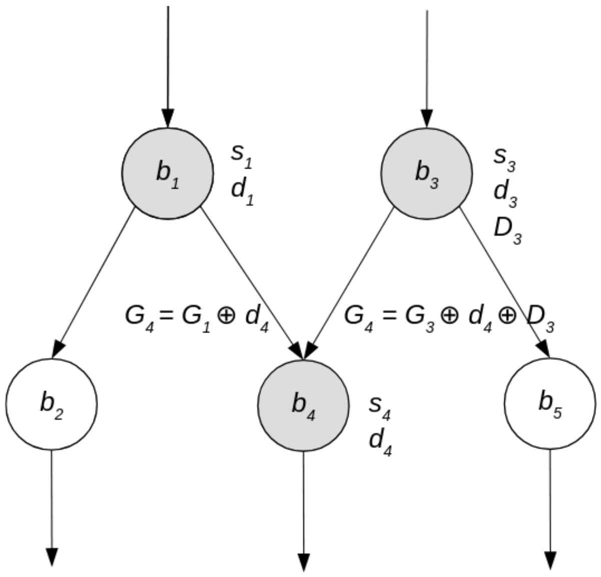

Figure 15: CFCSS with fan-in block [Jef]

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.

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

6.3.1 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 16). 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 16: Rust - CFCSS instructions inserted before function block

This macro can be applied as shown in figure 17 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 17: Rust - Using the CFCSS macro

6.3.2 Performance impact

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.

6.3.3 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.

6.4 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 variiying degree of protection depending on the use case.

6.4.1 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 18, 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.

```rust
fn bar(mut a: CopyCommit<u32>) {
    *a += 1;
    a.commit();
}

fn foo() {
    let mut a = 1;
    bar(CopyCommit::new(&mut a));
}
```

Figure 18: 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.

6.4.2 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.

```rust
// 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 19: 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 19. 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.

6.4.3 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.

6.4.4 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.

6.5 Multiversion

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

6.5.1 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 20, 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.

```rust
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 20: 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.

7 Testing

Testing of the implementation was done on a custom RISC-V core (1MB RAM, 8KB stack, 8KB heap) simulation with the ability to statically insert faults during runtime of the program. The frequency of the fault insertion as well as the memory 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 memory 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.

Predicate Registers (PRED) Used in prediction of conditional execution (error inserted here should not actually have any impact on the program functioning, as the prediction logic is assumed to be possibly innacurate by design).

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 6.4.3, 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.

7.1 Common errors

Despite the fact that the timing and location of error manifestation is unpredictable, most errors usually result in similar program states. We can check for the occurence of these states to determine what kind of error we are dealing with.

7.1.1 Exception

7.1.2 Infinite loop

7.1.3

7.2 Fault coverage

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.

[OSM02] 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.

[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`.