Appendices

Reg. No.: FIIT-100241-117028

A Work plan

Week	Activity
Week 1	Familiarization with the topic
Week 2	Familiarization with the topic
Week 3	Reading up general information about the topic
Week 4	Gathering and reading articles and papers about the topic
Week 5	Gathering and reading articles and papers about the topic
Week 6	Gathering and reading articles and papers about the topic
Week 7	Gathering and reading articles and papers about the topic
Week 8	Writing the analysis
Week 9	Writing the analysis
Week 10	Writing the analysis
Week 12	Writing the analysis
Week 12	Polishing up BP1

Table 9: Work plan - BP1

The first semester of working on the bachelor's thesis was mostly dedicated to reading and gathering of materials. This phase consisted of finding relevant articles describing software-implemented fault-tolerance methods that would be feasible to implement. This was surprisingly difficult as the problem space mostly revolves around hardware-implemented fault-tolerance methods.

Once a general idea of the thesis became clear, we moved onto the analysis writing stage. Here, general ideas were outlined, this served as a springboard to aid in implementation during the second phase.

Overall, the work plan was mostly adhered to and the first part of the thesis was submitted without much issue.

Week	**Activity**
Week 1	Familiarization with the technology
Week 2	Setting up the development environment
Week 3	Setting up Rust integration
Week 4	Debugging
Week 5	Creating checkpoint and restart
Week 6	Creating variable protection
Week 7	Creating CFCSS framework
Week 8	Implementing multi-version programming
Week 9	Setting-up benchmarks and testing enviornment
Week 10	Testing the benchmarks + thesis writing
Week 12	Polishing up the benchmarks and testing + thesis writing
Week 12	Final touches on BP2

Table 10: Work plan - BP2

The second semester was mostly focused on the implementation of previously analyzed software-tolerance methods. Before implementation, however, the development environment had to be set up. Having to work with a brand new architecture - RISC-V, and unusual operating system - FreeRTOS, the environment setup took a lot of time and debugging, especially considering the need to integrate it with Rust programming language.

After the environment setup, implementation of the select techniques began. Once the software was able to run in the simulated environment, we began setting up the fault-insertion and constructing benchmarks to execute.

The testing and polishing of the benchmarks was also accompanied with writing down the findings and implementation details into the thesis.

The plan was overall adhered to quite well and work on the thesis went along smoothly.

B Digital Files

The digital submission that comes with this thesis includes a complete Rust project, organized and structured to support the development, testing, and simulation of the implemented fault tolerance methods. Below is an overview of the main components and folders in the project:

- **.cargo**/ - Contains the Rust compiler build target.
- **FreeRTOS-Kernel** - Included as a Git submodule. This provides the real-time operating system used by the project.
- **libs**/ - Contains cloned Rust libraries for working with RISC-V:
  - `riscv` - Low-level access to RISC-V-specific features.
  - `riscv-rt` - Runtime support for RISC-V embedded programs.

  These libraries were included directly in the project so they could be modified to fit our specific needs.
- **link_files**/ - Contains the linker scripts and memory layout files required to build the project for our target platform. These files were copied from Rust RISC-V library[13] and modified, unless stated otherwise within the file.
- **macros**/ - Holds the macro-based implementation of the CFCSS fault tolerance technique (Control Flow Checking by Software Signatures).
- **rng**/ - A small utility module for generating pseudo-random numbers, used mainly for generation of signatures for CSFCSS and for testing purposes.
- **src**/ - This is where the main Rust application code lives, including the core logic for all fault tolerance methods.

  This folder further splits into several important subfolders:
  - **benchmarks**/ - Contains code for protected and unprotected benchmarks used for testing.

[13] `https://github.com/rust-embedded/riscv`

  - **checkpoint**/ - Contains the implementation of the Checkpoint and Restart system.
  - **events**/ - Contains hooks for RISC-V and FreeRTOS system events.
  - **utils**/ - Contains helper functions.
  - **vars**/ - Contains variable protection implementation.
  - Base directory contains the program entry point - *main.rs* as well as some helper and configuration files.

- **build.rs** - A build script that handles some of the behind-the-scenes configuration during compilation.

- **Cargo.toml** and **Cargo.lock** - Standard Rust files that define the project's dependencies, configuration, and build settings.

- **FreeRTOSConfig.h** - Configuration file for FreeRTOS, where system-level settings are defined.

- **Makefile** - A helper script to make building and running the simulation easier and more consistent. When running the project locally, set the MODELSIM and HARDISC variables to your corresponding installation paths.

- **examples**/ - Folder which contains compiled binary and object dump of an example benchmark (Fibonacci), both with and without protection. This is included so the basic functionality of the project can be verified without the need to compile the entire project from scratch.

Altogether, this project structure was designed to be modular, easy to navigate, and flexible enough to support low-level embedded development in Rust with FreeRTOS on a RISC-V target.

C Technical Documentation

This section provides detailed instructions on how to configure, build, and run the project included in the digital submission. The project is designed for simulation and testing in a RISC-V environment using Rust, FreeRTOS, and the Hardisc fault injection framework.

C.1 Prerequisites

Before building or running the project, ensure the following software and toolchains are installed and properly configured:

- **Rust** with `nightly` toolchain support. Install via `https://www.rust-lang.org/`.
- **RISC-V GNU Toolchain** compiled with support for:
  - `abi=ilp32`
  - `arch=rv32imac_zicsr`

  Toolchain can be downloaded and compiled from `https://github.com/riscv-collab/riscv-gnu-toolchain`.
- **ModelSim - Intel FPGA Edition** (tested with version 18.1), available at:
  `https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html`
- **Hardisc** fault injector cloned, configured, and compiled. See its documentation for detailed setup instructions (`https://github.com/janomach/the-hardisc`).
- **Make** tool installed (`https://www.gnu.org/software/make/`).

C.2 Configuration and Setup

1. Open the provided `Makefile` in the project files and update the following paths:

- `MODELSIM` - Set to the installation path of ModelSim.
- `HARDISC` - Set to the path where Hardisc is cloned and built.

2. Set Rust to use the nightly toolchain:

   ```
   rustup default nightly
   ```

3. Install the RISC-V target for Rust:

   ```
   rustup target add riscv32imac-unknown-none-elf
   ```

4. Download the FreeRTOS-Kernel submodule:

   ```
   git submodule update --init --recursive
   ```

C.3 Compilation and Simulation

1. To choose which benchmarks to compile and whether to include fault protection, edit the Cargo.toml file by settings the *features* section.

   ```
       features = ["fib", "no_prot"]
   ```

   This will compile the Fibonacci benchmark without any protection.

2. To compile the project:

   ```
   make
   ```

   The compiled binary and object dump can be found in the /target folder.

3. To run the simulation without fault injection:

   ```
   make sim
   ```

4. To run the simulation with fault injection using Hardisc follow the steps listed in `https://github.com/janomach/the-hardisc`.