Evaluation of a high performance ECU running embedded Linux

Utvärdering av en högpresterande ECU som kör embedded Linux

Oskar Larsson
Erik Magnusson

Faculty of Health, Science and Technology

Computer Science

C-uppsats 15hp

Supervisor: Per Hurtig

Examiner: Kerstin Andersson

Date: 2023-05-30

Acknowledgements

We want to thank Volvo Autonomous Solutions for giving us the opportunity to do this thesis work, and specifically our supervisors Jörlèn Teo, Takkoush Mohamed and Drenth Edo who supported and gave valuable feedback throughout the work.

We also want to thank our university supervisor Per Hurtig for providing help and feedback on the report during the project.

Finally, we would like to thank this thesis examiner, Kerstin Andersson.

Abstract

Autonomous vehicles rely on real-time embedded systems called Electronic Control Units (ECUs) and it is crucial for safety and behavioural correctness that ECUs run fast enough, in other words, within their deadlines. In this thesis a high-performance ECU used at Volvo Autonomous Solutions (V.A.S.) running PetaLinux will be evaluated. To ensure that the ECU is able to run in real-time, it will be evaluated by simulating physical models in the form of Functional Mock-up Units (FMUs) that are cross compiled to aarch64. The ECU execution time will be measured to evaluate whether the models run in real-time and to compare it with the performance of a desktop computer. With the use of profiling tools, possible bottlenecks in the utilization of ECU will be identified. The results show that the ECU is capable of running the FMU simulation models faster than real-time and that the ECU runs at less than 10% of the speed of the desktop computer. FMU communication frequency, underutilization of available instructions, unnecessary allocation and unnecessary use of synchronization primitives where identified as the key possible bottlenecks. We conclude that the ECU is likely capable of running significantly more computationally demanding models in real-time if they are optimized for the ECU.

Contents

List of Figures

List of Tables

Listings

Glossary

aarch64	64 bit extension of ARM.
ABI	Application Binary Interface.
API	Application Programming Interface.
BSP	Board Support Package.
C	The C programming language.
CISC	Complex Instruction Set Computer.
CPU	Central Processing Unit.
CS	Co-Simulation.
CSV	Comma-separated values.
ECU	Electronic Control Unit.
FMI	Functional Mock-up Interface.
FMIL	FMI Library.
FMU	Functional Mock-up Unit.
FP	Function Pointer.

GCC GNU Compiler Collection.

I/O Input and Output.

ISA Instruction Set Architecture.

LUT Look Up Table.

ME Model Exchange.

OMEdit OpenModelica Connection Editor.

OS Operating System.

POSIX Portable Operating System Interface.

RISC Reduced Instruction Set Computer.

RTOS Real-Time Operating System.

SD Standard Deviation.

V.A.S. Volvo Autonomous Solutions.

x86-64 64 bit version of the x86 ISA.

XML Extensible Markup Language.

Chapter 1

Introduction

This chapter introduces the topic, background and motivation for the thesis work. Section 1.1 introduces the background for the topic. Section 1.2 describes the main problems in the thesis topic. Section 1.3 describes the purpose of the work. Section 1.4 describes the ethical considerations of the work. Section 1.5 describes how tests will be performed. Section 1.6 describes the company the thesis was done for. Section 1.7 describes how the work was divided between the students. Section 1.8 describes the main deliminations of the work. Section 1.9 describes how the rest of the report is structured.

1.1 Background

Autonomous vehicles are a quickly emerging technology with great potential impact on society and Volvo Autonomous Solutions (V.A.S.) is one of the companies developing the technology. The vehicles rely on embedded computer systems called Electronic Control Units (ECUs) that need to run complex models within strict performance requirements in order to facilitate the control systems. ECUs typically run specialized Real-Time Operating Systems (RTOSs) or run bare-metal software, since a standard

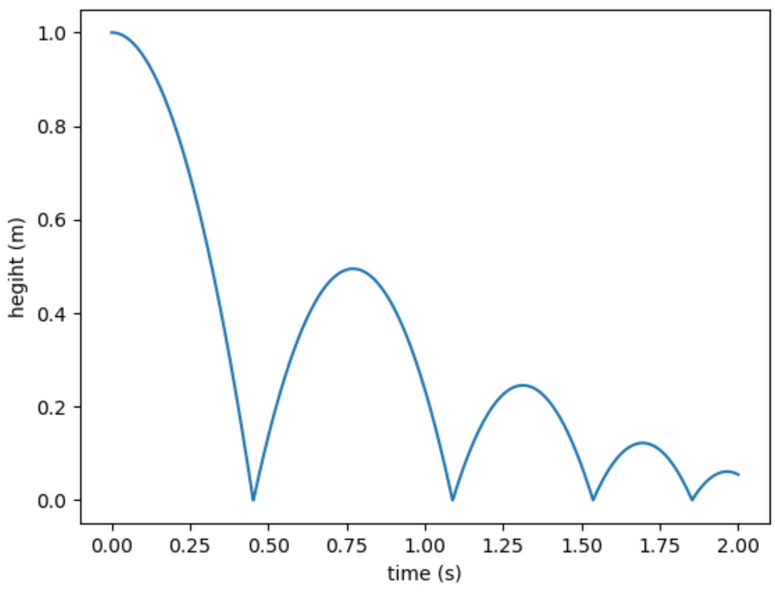


Figure 1.1: Results from a FMU simulating a bouncing ball.

desktop operating system has unpredictable or unbounded delays and resource usage. While bare-metal coding gives developers more control over a system, it also makes it harder to use, debug and develop for. High performance ECUs have a higher performance margin, and therefore, using a RTOS may be viable. To evaluate this, one of the ECUs used at V.A.S. will be benchmarked by running a vehicle simulation in the form of a Functional Mock-up Unit (FMU), which consists of C source code or binaries along with an Extensible Markup Language (XML) description of the model, that contains for example variable names and units [1]. Figure 1.1 is an example of the output of a FMU, it shows how the height variable changes over time. The ECU will be running the PetaLinux [2] Operating System (OS), requiring a cross compiled program to execute the various FMUs. The ECU simulation speed will be compared with simulation time and with a desktop computer to understand the ECU performance.

1.2 Problem Description

It is common for ECUs to utilize RTOSs in order to remove interference from unwanted processes that may be present in an OS that is not designed for real-time applications. RTOSs reduces the risk of missing deadlines, and therefore the risk of a failure in the system. To determine the available performance capacity of the system is enough for the real-time application, it is important to be able to run test directly on the ECU. The evaluation will be performed by running a FMU simulation model. Because the ECU and the Linux desktop computers are using different Instruction Set Architectures (ISAs) the programs and FMUs needs to be cross compiled to the ECU ISA, instead of compiling natively. This is because code compiled for the wrong Central Processing Unit (CPU) architecture or Application Binary Interface (ABI) will not run due to the machine instructions being incompatible. The Linux desktop computers use the more common x86-64 ISA and the ECU uses aarch64. We aim to answer how well the embedded ECU can run a FMU with the PetaLinux OS [2].

1.3 Purpose and Goals

To evaluate the performance of the ECU, the PetaLinux [2] OS and associated tools will be used to be able to run a FMU simulation model and check whether it runs faster than real-time. The performance of the ECU will be compared with the result from the same FMU running on a regular Linux desktop computer. To get the FMU on to the ECU a program that interfaces with the FMU needs to be created that uses FMI Library (FMIL), a C library that can load and run FMUs. In order to run the FMU model on the ECU, it needs to be cross-compiled along with the program that will be interfacing with the FMU, which is going to add an additional challenge. In this thesis we aim to answer:

- Is the ECU able to run the models faster than real-time?
- How does the ECU perform compared to a desktop computer?
- What are some of the possible performance bottlenecks?

1.4 Ethics

As autonomous vehicles become more common there is a need to decide who is responsible for the actions of the vehicle. Depending on the level of autonomy of the vehicle a number of different entities may be responsible. When a vehicle that would be classified as level three or lower on the SAE J3016 scale, meaning that the driver is required to take control of the vehicle at any time [3], the driver of the vehicle would be responsible. But when the level vehicle is on a higher level on the SAE J3016 scale, being classified as level four or five, the vehicle does not need to rely on a driver at all [3]. Since there are no driver that can take responsibility, the producer of the autonomous system will be responsible. For the developers to be able to ensure that safety systems that they have put in place will be activated in time, they need to make sure that the computer responsible for the system will be able to keep within its deadlines. If calculations on the system are not finished in time and a deadline is missed, the safety and functionality of a vehicle can be severely compromised. It is therefore vital that the performance characteristics are very well understood and that there is a significant performance margin.

1.5 Method

To run the simulations on the ECU, PetaLinux [2] tools will be used to setup the OS. The simulations use FMU models for the vehicles. The FMU will be run with a C program using FMI Library (FMIL) [4]. We will use a version of GCC to cross-compile the FMU, C program and FMIL in order to run it on the ECU, since its running a different CPU architecture than the development devices. `perf` [5] will be used to look for possible bottlenecks during execution. All tools and hardware were provided by the stakeholder.

1.6 Stakeholder

This work was done for Volvo Autonomous Solutions (V.A.S.), who specializes in self driving transport [6]. Volvo produced their first car in 1927 [7]. Since then Volvo Group have expanded into most of the automotive industry, producing trucks, buses, military vehicles [8] and autonomous haulage systems [6]. In 1999 Volvo group sold the Volvo Cars division of the company to focus on bigger vehicles [7]. V.A.S. was founded in 2020 to accelerate and commercialize autonomous transport [6]. This work will provide a tool for running simulations directly on one of their ECUs. This allows the company to compare the performance of the ECU to the simulation performance on a regular desktop in order to get a better understanding of the performance characteristics of the ECU.

1.7 Work distribution

The report has been worked on iteratively and all parts of the report have been revised by both of us, every section is therefore a shared responsibility. The code was also worked on iteratively by both of us. Some of the practical parts of the work where not overlapping, simply because it did not make sense for multiple people to work on it at the same time.

Oskar

Analysis of benchmarks, images in the report, graph generation and making FMUs in OpenModelica Connection Editor (OMEdit).

Erik

Setting up PetaLinux and FMIL, ECU communication, cross compiling and profiling.

1.8 Delimitations

The functional safety or correctness of the models will not be considered. The FMUs used will be simulation models representing mechanical systems, in order to keep the result as relevant as possible. A Co-Simulation (CS) FMU will be used because a Model Exchange (ME) FMU adds unnecessary complexity and is not used by the provided FMU models. While useful as a comparison, the application will not be running on the ECU bare-metal since that would be more complex and make the scope of the project too big. FMIL will be used to interface with the FMU which means that it will be dynamically linked at runtime instead of statically compiling the FMU into the application. All code will be compiled using GCC, both to reduce the number of variables that may affect the result, but also to reduce the number of different FMUs in the tests. There are a large number of embedded OSs, but only PetaLinux will be used for this project. Profiling tools will not be run on the target ECU, since that would be too complex.

1.9 Disposition

In Chapter 2 the technical background is described, notably compilation, embedded systems and FMI/FMU. Implementation decisions and how these decisions changed during the project are described in Chapter 3. Details about the implementation, for example cross compiling and running the FMU is described in Chapter 4. The results of the performance evaluation are described in Chapter 5. Conclusions and future work are discussed in Chapter 6.

Chapter 2

Background

This chapter describes the technical background needed to understand this work. Section 2.1 describes embeded and real-time systems. Section 2.2 describes the Electronic Control Unit (ECU). Section 2.3 describes Linux. Section 2.4 describes different processor architectures. Section 2.5 describes compilation and linking. Section 2.6 describes the Functional Mock-up Interface (FMI) standard. Section 2.7 describes the FMI Library (FMIL). Section 2.8 describes what OpenModelica is and Section 2.9 describes benchmarking.

2.1 Embedded and Real-time Systems

Embedded systems are computer systems that exist within other systems to perform a specific function, for example the computers in washing machines, microwave ovens or network switches [9]. Contrast this with a general purpose computer which can operate independently of other systems and is able to do general purpose tasks. Note that a general purpose computer contains many small embedded systems, such as the memory controller. Since they are made to do a very specific task, they often only have enough resources such as processor speed, memory and Input and Output (I/O) needed to complete the task in order to reduce for example production cost and power usage [9].

Embedded systems are typically also real-time systems, where their work needs to be reliably completed within some time constraint, called deadlines [9]. Deadlines can be put into the spectrum from soft to hard deadlines. A soft deadline may be a video stream, where missed or skipped deadlines for packets has a low consequence. A hard deadline is for example self-driving cars, train brake systems or airplane autopilots, where a missed deadline may cause harm or death to humans or severe economic costs. If a system is embedded it does not necessarily have to be real-time and visa versa, for example a desktop computer running a Real-Time Operating System (RTOS) or a calculator.

2.2 Electronic Control Unit (ECU)

An Electronic Control Unit (ECU) as can be seen in Figure 2.1, is an embedded system for automotive electronics, that is responsible for controlling a specific function in a vehicle. Generally, the ECU controls essential functions like the engine or the steering control, but it could also be tasks as simple as controlling the motor in the seats or windows [10]. Figure 2.2 shows some of the common uses for an ECU in a car, for example controlling the airbags, anti-lock breaking and controlling the engine. An ECU is typically low performance since the tasks it performs are quite computationally simple, but in autonomous applications, the computations are more demanding and therefore a high performance ECU is needed.

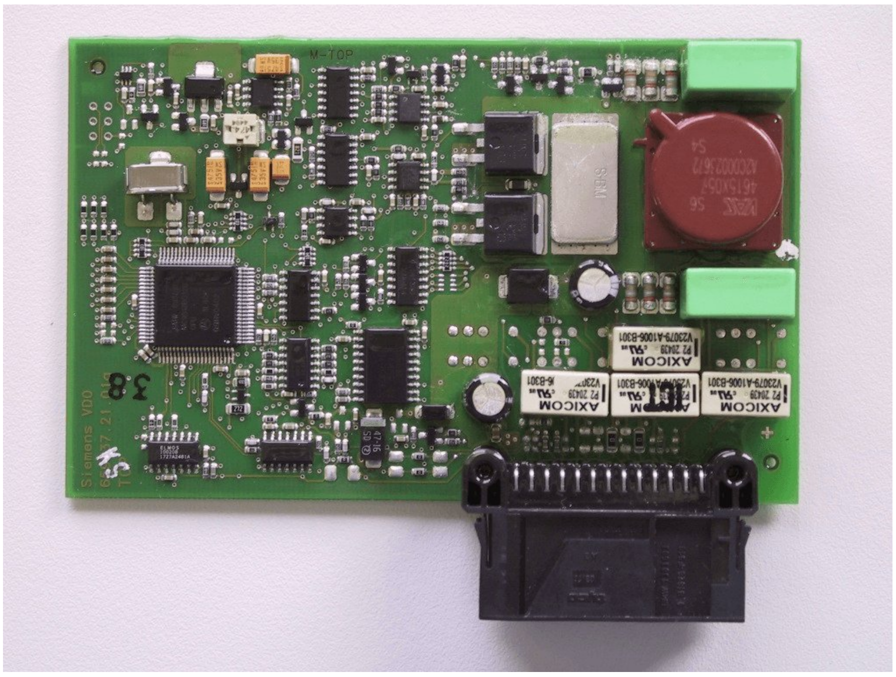


Figure 2.1: Example of an ECU. (Image source: [11])

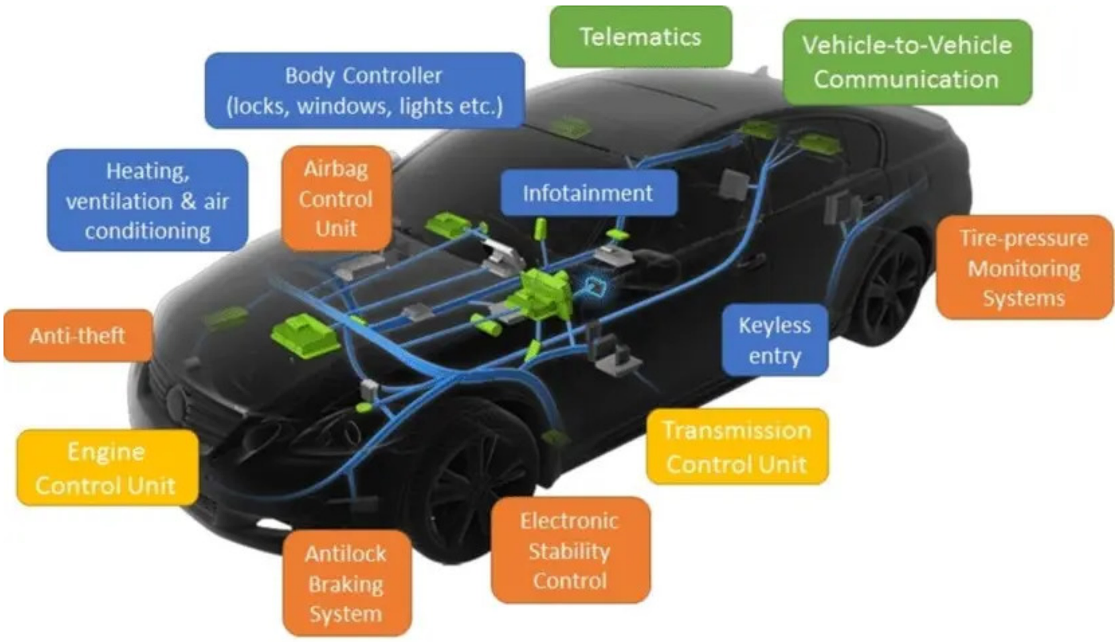

Telematics
Vehicle-to-Vehicle Communication
Body Controller (locks, windows, lights etc.)
Heating, ventilation & air conditioning
Airbag Control Unit
Infotainment
Tire-pressure Monitoring Systems
Anti-theft
Keyless entry
Transmission Control Unit
Engine Control Unit
Electronic Stability Control
Antilock Braking System

Figure 2.2: Example uses of ECUs in a car. (Image source: [12])

2.3 Linux

Linux is a group of open source Operating Systems (OSs) that are typically used for desktop and server applications [13]. The Linux kernel is by default not a real-time kernel and there is therefore a potentially a significant overhead when running it compared to bare-metal programming, where code is compiled to be run directly on the processor. Because autonomous applications often need to run in real-time with hard or firm deadlines, the unpredictable performance overhead of running an OS that is not a RTOS can impact the performance and reliability significantly. PetaLinux is used for the ECU and it is an OS based on the Yocto-Linux distribution designed to ease development for Xilinx boards.

2.4 Instruction Set Architecture (ISA)

An Instruction Set Architecture (ISA) or computer architecture is a model of how a computer executes machine code, for example aarch64 and x86-64 are Instruction Set Architectures (ISAs). Machine code is the actual 0s and 1s that the Central Processing Unit (CPU) interprets and assembly is a human readable form of it. Using `objdump` it is possible to see what instructions are used, to the left is machine code and to the right is assembly.

For example this is what a recursive fibonacci function looks like on x86-64 and aarch64:

```
objdump main --disassemble=fib -M intel
00000000000044b0 <fib>:
f3 0f 1e fa         endbr64
83 ff 01            cmp     edi,0x1
76 27               jbe     44e0 <fib+0x30>
55                  push    rbp
31 ed               xor     ebp,ebp
53                  push    rbx
89 fb               mov     ebx,edi
48 83 ec 08         sub     rsp,0x8
8d 7b ff            lea     edi,[rbx-0x1]
83 eb 02            sub     ebx,0x2
e8 e2 ff ff ff      call    44b0 <fib>
01 c5               add     ebp,eax
83 fb 01            cmp     ebx,0x1
77 ee               ja      44c3 <fib+0x13>
48 83 c4 08         add     rsp,0x8
8d 45 01            lea     eax,[rbp+0x1]
5b                  pop     rbx
5d                  pop     rbp
c3                  ret
90                  nop
b8 01 00 00 00      mov     eax,0x1
c3                  ret
```

Listing 2.1: x86-64 code.

```
aarch64-linux-gnu-objdump main_arm --disassemble=fib
00000000000033a0 <fib>:
71 00 04 1f        cmp   w0, #0x1
54 00 02 c9        b.ls  33fc <fib+0x5c>
a9 bd 7b fd        stp   x29, x30, [sp, #-48]!
51 00 08 01        sub   w1, w0, #0x2
12 1f 78 21        and   w1, w1, #0xfffffffe
91 00 03 fd        mov   x29, sp
f9 00 13 f5        str   x21, [sp, #32]
51 00 0c 15        sub   w21, w0, #0x3
4b 01 02 b5        sub   w21, w21, w1
a9 01 53 f3        stp   x19, x20, [sp, #16]
51 00 04 13        sub   w19, w0, #0x1
52 80 00 14        mov   w20, #0x0
2a 13 03 e0        mov   w0, w19
51 00 0a 73        sub   w19, w19, #0x2
97 ff ff f2        bl    33a0 <fib>
0b 00 02 94        add   w20, w20, w0
6b 15 02 7f        cmp   w19, w21
54 ff ff 61        b.ne  33d0 <fib+0x30>
11 00 06 80        add   w0, w20, #0x1
a9 41 53 f3        ldp   x19, x20, [sp, #16]
f9 40 13 f5        ldr   x21, [sp, #32]
a8 c3 7b fd        ldp   x29, x30, [sp], #48
d6 5f 03 c0        ret
52 80 00 20        mov   w0, #0x1
d6 5f 03 c0        ret
```

Listing 2.2: aarch64 code.

Note that x86-64 instructions are variable length and the aarch64 instructions are all 32-bit. x86-64 is a Complex Instruction Set Computer (CISC) ISA and aarch64 is a Reduced Instruction Set Computer (RISC) ISA. Generally, CISC architectures have many instructions that do several operations, and typically has many addressing modes or many uncommon operations and RISC architectures have few simple orthogonal instructions and are typically load-store architectures, which means that memory is only loaded and stored using explicit instructions. In the execution time formula:

$$\text{total execution time} = \frac{\text{instructions}}{\text{program}}\frac{\text{cycles}}{\text{instruction}}\frac{\text{time}}{\text{cycle}}$$

CISC ISAs tends to focus on instructions/program and programmer convenience while RISC ISAs tends to focus on cycles/instruction and time/cycle. x86-64 is common in personal computers because most software is made for x86-64, but RISC architectures tend to be more power efficient and faster when implemented, and therefore RISC architectures are more common in high performance computing and embedded systems [14].

2.5 Compilation and Linking

Compilation is the process of translating between programming languages, typically from source code to assembly or machine code, and is done by a compiler. C and Fortran are examples of languages that typically compile to machine code, they were made at a time where parts of programs had to be separately compiled and then linked together, because of memory constraints. In the case of C, each file is the compilation unit that is separately compiled. There are many compilers for the same language, they mainly vary in what architectures they support, what optimizations they perform on the code after it has been parsed before generating the machine code and how fast they can compile.

The compilation unit is compiled into an object file with a given Application Binary Interface (ABI) that describes for example calling conventions and is specific to the ISA, OS and compiler. For example the code for the ECU in this report is compiled to the `aarch64-linux-gnu` ABI. If the ABIs are the same between different object files, they can be linked together using a linker that combines all the object files resulting in a single executable binary file. If there are any references to symbols in another object file, such as global variables or functions, these get resolved by the linker. A common linker error is missing symbols, where an object file references a non existent symbol, which is typically caused by referencing a function that was never implemented or the object file containing the function was never created. Linking does not require the compilers to be the same and therefore, for example a Fortran compiler could produce object files that are linked with object files generated by GCC, as long as the ABIs match.

Linking can also be performed dynamically at runtime, called dynamic linking. A set of object files is compiled into a shared library and then it can be loaded into memory at runtime, and a program can query the OS to obtain function pointers to the requested symbols [15]. Sometimes, compiling a program on the same system that will run it (native compiling) is not possible, impractical, or the program is expected to run on a variety of different systems. In this case the code needs to be cross compiled to target a different ISA from the host system (see Figure 2.3).

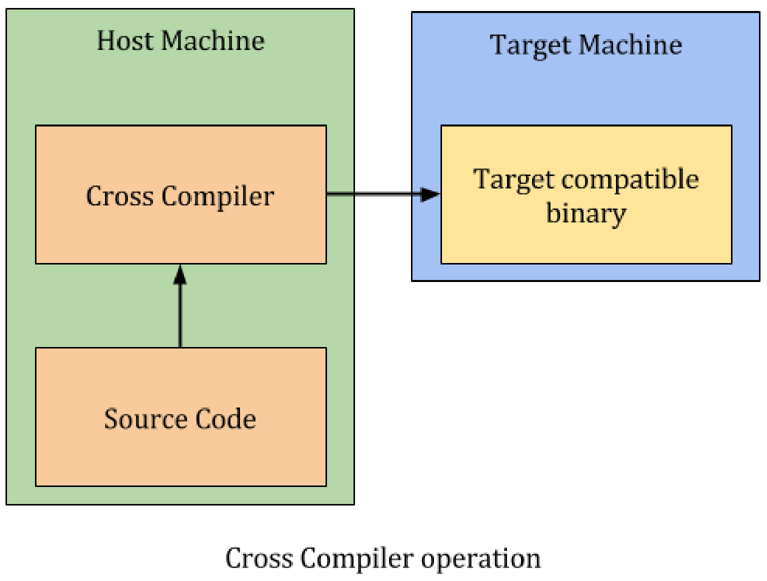


Figure 2.3: Cross compiling to a different ISA. (Image source: [16])

2.6 Functional Mock-up Interface (FMI) and Functional Mock-up Unit (FMU)

Functional Mock-up Interface (FMI) is a standard for sharing models between different software and tools. A model is packaged into an Functional Mock-up Unit (FMU) which is a zip-file but with a `.fmu` filename extension, containing an Extensible Markup Language (XML) description of the capabilities, header files, binaries and/or C files, [1]. Because of this, a modeling environment can generate a processor agnostic FMU and it is possible to treat the internals of a FMU as a black box when compiling it, making reusing and distributing FMUs easy [1]. Because the simulation model will be compiled into a FMU, it can be treated as a black box. The XML file contains static information such as variable names, attributes, units and default values. A FMU may contain several different binaries compiled to be able to run on different systems without needing to be recompiled.

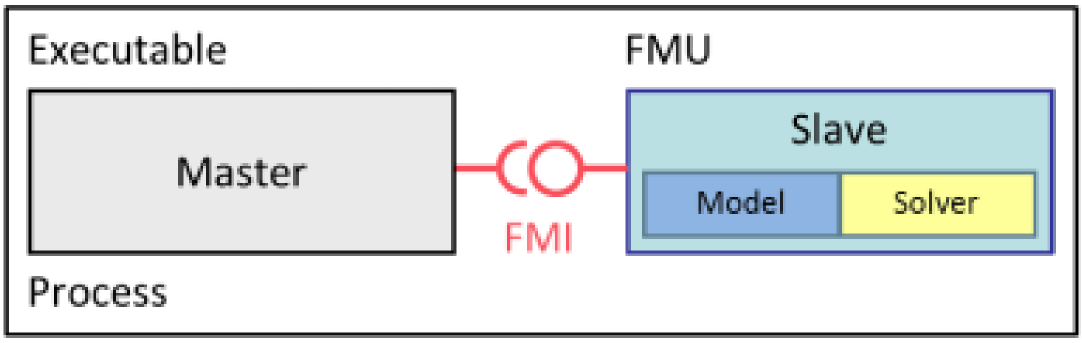


Figure 2.4: Use of a Co-Simulation (CS) FMU. (Image source: [1])

FMI is divided into two types of FMUs, Model Exchange (ME), and Co-Simulation (CS) [1] With ME, the FMU does not contain the solver, instead it has Ordinary Differential Equations (ODEs) in state space form [1]. The solver is instead provided by the simulation environment. A CS FMU, as can be seen in Figure 2.4, is more general than ME, the FMU instead has it's own solver, and is able to communicate with other simulation models [1]. Variable communication occurs at specific time intervals, and can be different to time step of the internal solver [1]. For numerical accuracy, if the internal solver is fixed step, the communication interval should be the same as the internal time step.

2.7 FMI Library (FMIL)

FMI Library (FMIL) is a C library intended as a tool for integration of FMUs [4]. It is capable of handling FMUs implemented using FMI 1.0 and FMI 2.0 versions and both CS and ME FMUs. The handling of the different FMI versions and simulation types is up to the user of library to implement. Since most of the time only the FMI version and simulation type used will be implemented, it is possible to check the FMI version and simulation type of the provided FMU in order to only proceed if the FMU fulfills the required specification.

When the FMU is prepared by the library (see function *prepare_fmu* in Appendix C), the internal variables defined in the XML file can be set to initialize the FMU if necessary. To then run a CS FMU, the user needs to step through the simulation using `fmi2_import_do_step`, this will run the simulation according to the time specified in the parameter `communicationStepSize` (see function *run_simulation* in Appendix C). If the user want to read or write to any of the internal variables of the FMU during the simulation, this can be done between each simulation step.

2.8 OpenModelica

OpenModelica is an open-source modeling and simulation environment [17]. One part of OpenModelica is the program OMEdit. OMEdit is a tool that can give a graphical view of the internal structure of the mechanical FMUs that are provided by Modelica. This is shown in Figure 2.5 where a number of mechanical components can be seen such as gears, inertial masses and motors. Another important aspect of this tool is the integrated compiler that is able to cross compile a given FMU to several different ISAs when exporting the FMU.

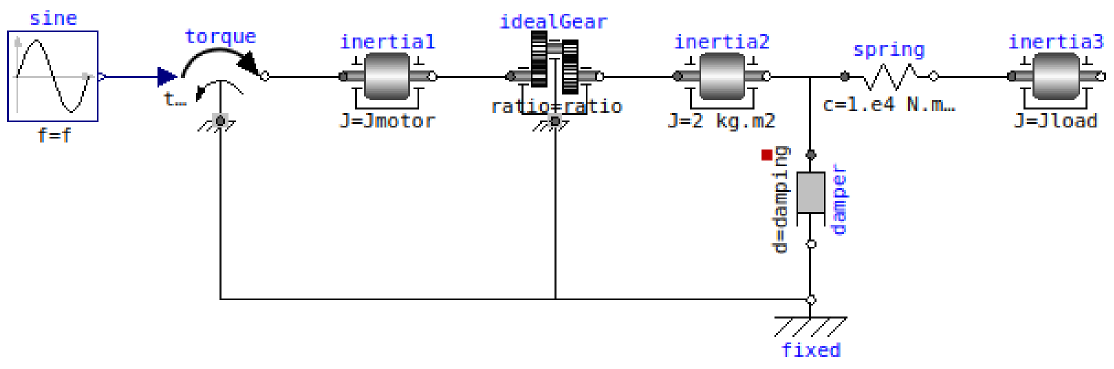


Figure 2.5: View from OpenModelica Connection Editor (OMEdit) of the OpenModelica Mechanics Rotational First FMU [17].

2.9 Benchmarks

Benchmarking measures the relative performance of a computer. Benchmarks can be divided into two groups: synthetic benchmarks and real-world benchmarks [18]. The benefit of synthetic benchmarks are that they can be easily automated and they can give a standard for evaluating performance, independently of the difference in hardware or other software on the system. The test can consist of different simulated tasks like 3D rendering, file compression and floating-point calculations [18]. The term synthetic comes from the fact that the test result are gathered from simulations, unlike the real-world benchmarks. In the real-world benchmarks the actual program that is expected to run on the system is tested [18].

Chapter 3

Design

This chapter explains the decisions about the implementation and how the overall work will be structured. Section 3.1 describes the goals of the project. Section 3.2 describes the setup of the Electronic Control Unit (ECU) and the tests. Section 3.3 describes how testing will be done. Section 3.4 describes what models will be used and how that changed.

3.1 Design goals

In order to evaluate the performance of the ECU, the goal have been split into three distinct sub-goals that will be used in order to get a better understanding of the result. The sub-goals will aim to find different aspects of the ECUs performance. The first sub-goal will answer the question: *Is the ECU able to run the models faster than real-time?* The second: *How does the ECU perform compared to a desktop computer?* The third: *What are some of the possible performance bottlenecks?* The first sub-goal will tell if the ECU is capable of running a specific model in a real-time setting. The second will give a reference of how the execution time of the same Functional Mock-up Unit (FMU) will differ on the ECU and on a Linux desktop in order for a developer to be able to estimate the ECU performance without running tests directly on the ECU. And the final sub-goal will allow for future optimization by identifying bottlenecks that can be removed in future works.

3.1.1 Is the ECU able to run the models faster than real-time?

Running the models faster than real-time means that the execution time is smaller than the simulation time. Execution time is how many seconds the Central Processing Unit (CPU) has spent running the simulation and simulation time is how many seconds have elapsed in the simulation. For example, if the execution time was 4 seconds and the simulation time was 5 seconds, then the model is running at 5/4 of real-time and has 1.25 times real-time performance. This is vital to ensure that it is possible for the ECU to finish within its deadlines.

3.1.2 How does the ECU perform compared to a desktop computer?

The ECU performance will be compared to the performance of a desktop computer, specifically how long it takes to simulate the different models. This is important to get an understanding of how much work is performed when simulating the models and to see if the available performance of the ECU is utilized reasonably well. The Standard Deviation (SD) can give an indication about how reliably the different systems perform when runing the simulations.

3.1.3 What are some of the possible performance bottlenecks?

Throughout testing, it is expected that some possible performance bottlenecks will be identified. Whether something is a performance bottleneck is hard to empirically verify, so this will be estimated based on program behavior, our code and results from profiling tools run on the desktop. The bottlenecks found may then be mitigated or removed in future work on the ECU.

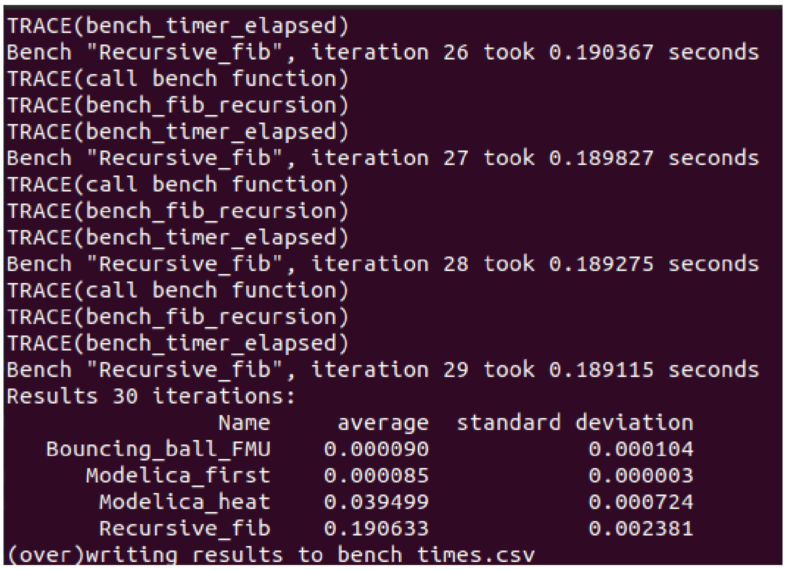
3.2 Setting up Benchmarks

To keep Operating System (OS) setup simple, the ECU will be running PetaLinux on a SD card with a pre-built Board Support Package (BSP). Communication with the ECU will be done using serial USB and directly modifying the file system by reading from and writing to the SD card while the ECU is turned off. All software on the ECU, (application and FMUs) will be cross compiled with GCC. While it would be possible to do native compiling, it would require that there was a working native compiler for the ECU, and would require an ECU just for compiling or require emulation, it is therefore simpler to use cross compiling. GCC will be used because it can cross compile, has aarch64 code generation, and it works on Linux. The FMUs will be running on the ECU and therefore both the FMUs and FMI Library (FMIL) will need to be cross-compiled. FMUs are used because they are a standard way of exchanging dynamic simulation models. FMIL will be used because it is one of the only public C libraries for loading FMUs, otherwise the Functional Mock-up Interface (FMI) functions would have to be interfaced with directly. C will be used because it has low overhead and a GCC can generate ECU compatible code. A C program will be created to run and interface with the FMUs which will store execution times for each FMU. This will allow for time comparisons with the length of the simulation in order to answer goal in *Is the ECU able to run the models faster than real-time?* and to compare the ECU and Linux desktop times in order to answer the question in *How does the ECU perform compared to a desktop computer?*. The available FMUs are source based or compiled for x86-64 and therefore they either need to be cross compiled to aarch64 or aarch64 FMUs need to be directly generated from the tool that creates the FMUs in order to run them on the ECU. To make this process easier a Python script and a makefile will be created to invoke GCC and package the compiled FMU.

3.3 Executing Benchmarks and Measure Results

In order to answer *How does the ECU perform compared to a desktop computer?* the C program will run on a Linux x86-64 desktop computer and the ECU to measure the execution times. The execution of Appendix C Benchmarking application is displayed in Figure 3.1 The final line of the program shows that the execution times of each individual test of the benchmarking application will be stored in a Comma-separated values (CSV) file. By saving the execution times to a file the results can be moved from the SD card to the Linux desktop, and then be analyzed and compared with the mesured times from the Linux desktop.

In order to find the answer to *What are some of the possible performance bottlenecks?*, `perf` [5] will be used along with memory usage measurements on the desktop computer to understand if there are any obvious performance bottlenecks. Since the Appendix C Benchmarking application loads the FMU before each test case a big part of the findings in `perf` will reflect this. To make the results from `perf` show more of the relevant code, the simulation time will be increased to 100 seconds instead of the 1 second simulation time that will be used in the other tests. No profiling tools will be used on the ECU since making them work on it was judged to take too long to setup.



```
TRACE(bench_timer_elapsed)
Bench "Recursive_fib", iteration 26 took 0.190367 seconds
TRACE(call bench function)
TRACE(bench_fib_recursion)
TRACE(bench_timer_elapsed)
Bench "Recursive_fib", iteration 27 took 0.189827 seconds
TRACE(call bench function)
TRACE(bench_fib_recursion)
TRACE(bench_timer_elapsed)
Bench "Recursive_fib", iteration 28 took 0.189275 seconds
TRACE(call bench function)
TRACE(bench_fib_recursion)
TRACE(bench_timer_elapsed)
Bench "Recursive_fib", iteration 29 took 0.189115 seconds
Results 30 iterations:
                Name     average  standard deviation
   Bouncing_ball_FMU    0.000090            0.000104
      Modelica_first    0.000085            0.000003
       Modelica_heat    0.039499            0.000724
       Recursive_fib    0.190633            0.002381
(over)writing results to bench times.csv
```

Figure 3.1: Output of Appendix C Benchmarking application in the terminal.

3.4 Choice of FMU Models

Ideally, the models used should reflect actual autonomous applications, and therefore at least some physical system should be simulated. Initially, it was planned to run a Volvo Autonomous Solutions (V.A.S.) FMU of a vehicle dynamics model of an entire vehicle to get a good understanding of the goal *Is the ECU able to run the models faster than real-time?*. However the model contained binary dependencies on proprietary third party x86-64 libraries, and obtaining the aarch64 versions of them was not possible due to time and monetary constraints. Another similar vehicle FMU was tried, but it had the same issue.

To still run a FMU that represents a physical system, a couple of the OpenModelica example models will be used (shown in Figure 3.2 and Figure 2.5). Initially, these where generated using Modelon Impact, a proprietary program, which can generate x86-64 and source based FMUs. Although it runs fine on the desktop, the generated aarch64 FMUs produce segmentation faults in the generated runtime when running it on the ECU. Since the source based FMU runtime was machine obfuscated, it was difficult to debug and it might not be solvable by some simple change to it, so the effort to find the root cause of it was abandoned.

There exist another way to generate these FMUs, by using OpenModelica Connection Editor (OMEdit) from OpenModelica. OMEdit is an open source program capable of generating FMUs, in a similar way to Modelon Impact. It can generate x86-64 and aarch64 FMUs directly, so that manually compiling them is not necessary. These where able to run on the ECU.

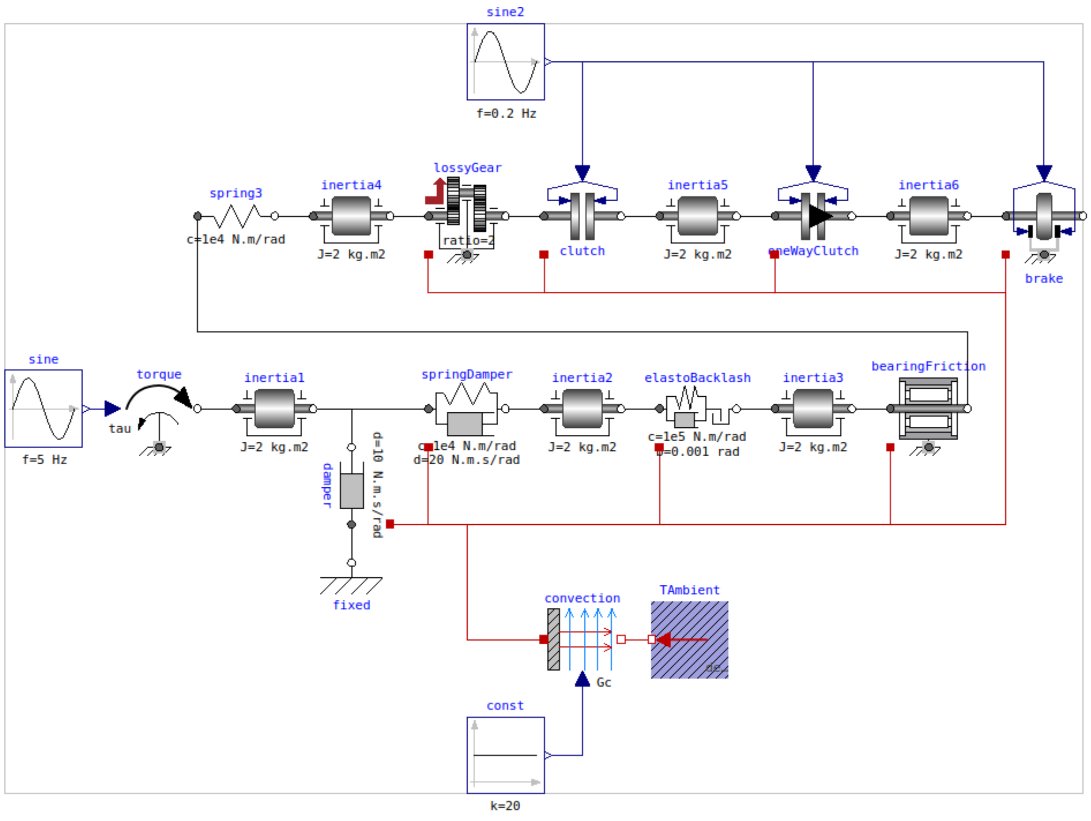


Figure 3.2: View from OMEdit of the OpenModelica Mechanics Rotational Heatlosses FMU [17].

In addition to the aforementioned models, a trivial example FMU called Bouncing-Ball is also used to get a baseline and to make sure that any potential issue is with the FMU and not the C program. A simple recursive Fibonacci function is also tested to check that the performance differences are to some extent reasonable. To summarise, these are the benchmarks that will be run:

1. A simple FMU simulating a bouncing ball that calculates the height of a ball durring three seconds. The first two seconds of the result can be found in Figure 1.1 [19].

2. Modelica Rotational First is a FMU simulating a mechanical system. In Figure 2.5) [17] the internal parts of the FMU are displayed. It consists of a motor that is controlled by a sine-wave generator. Several other components like a gearbox, a spring and a dampener is connected to the motor and transporting the rotational energy [20].

3. Modelica Rotational HeatLosses is more complex mechanical FMU. Figure 3.2 [17] displays the internal parts of the FMU. The drive train is more complex than in the Modelica Rotational First FMU and in addition to this this model takes the generated heat from each component and disperses it to the environment [21].

4. A simple recursive Fibonacci found in Appendix C Benchmarking application. The function takes an integer and calculates the number at that position in the Fibonacci series. This is done by calling the function recursively to generate the numbers at position *n-1* and *n-2*.

Chapter 4

Implementation

This chapter describes the details of how the design was implemented. Section 4.1 describes Petalinux installation. Section 4.2 describes usage of FMIL. Section 4.3 describes booting the ECU. Section 4.6 and Section 4.7 describe FMU generation.

4.1 PetaLinux

The PetaLinux tools along with the Board Support Package (BSP) for the ECU was obtained from the Xilinx website [22] and required dependencies documented in the releases page [23] were installed using the command listed in Appendix G. PetaLinux requires using `bash` instead of `dash`, which is default on Ubuntu, so it was changed. Using the PetaLinux tools we created a project, built the image and packaged it into a binary

```
petalinux-create -t project -s <BSP file>; cd <project
   folder>
petalinux-build
petalinux-package --boot --u-boot
```

Listing 4.1: Build PetaLinux.

The ECU will boot from an SD card, `fdisk` was used to create the partitions. One partition contains the file system and the other partition contains the boot loader (`BOOT.BIN`) and image (`image.ub`).

4.2 FMI Library

FMIL is the library that will be used for loading and running FMUs. Since the library is written in C it needs to be compiled and cross-compiled for the ECU in order to be utilized. This section describes the compiling, cross-compiling and setup of the FMIL

4.2.1 Installing and Compiling library

FMIL compiles into a shared library file (.so) using `cmake`. If FMIL source is located in a folder named `fmi-library`, it can be compiled with the following script:

```
#!/bin/sh
mkdir build-fmil; cd build-fmil
cmake ../fmi-library -DFMILIB_BUILD_TESTS="OFF"
make install
```

Listing 4.2: Compile FMIL.

This compiles into a shared library which is dynamically linked at runtime. This allows multiple programs to use it without increasing the size of their binary. Since the FMI standard is written in C, FMIL shares some of the FMI Application Programming Interface (API) by using wrapper functions.

4.2.2 Building Application Binary (x86-64)

In order to use FMIL, a small application named `main.c` that interfaces with the library was written (Appendix C Benchmarking application). GCC was used and the path to the library header and the path to the shared library was included when compiling the program. The makefile that was used can be found in Appendix D Application makefile.

```
gcc main.c -o main
    -I<fmil install folder>/include
    -L<fmil install folder>/lib
    -l:libfmilib_shared.so
```

Listing 4.3: Compile the C program that interfaces with FMIL.

4.2.3 Running Binary with shared library

Because the shared library is loaded at runtime, the OS needs to know about it. This can be achieved on Ubuntu by adding the shared library folder to the environment variable `LD_LIBRARY_PATH`. A wrapper script can be used to achieve this.

```
#!/bin/sh
if [ -n "$LD_LIBRARY_PATH" ]; then
  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<fmil install folder>>/
     lib
else
  LD_LIBRARY_PATH=<fmil install folder>/lib
fi
export LD_LIBRARY_PATH
exec ./main
```

Listing 4.4: Wrapper to add FMIL to `LD_LIBRARY_PATH`.

4.2.4 Cross Compiling FMIL to the ECU

Compiling using the PetaLinux version of GCC did not work because it needed a number of missing headers. The ECU uses aarch64, so the following package is installed to get a version of GCC with aarch64 code generation.

```
sudo apt install gcc-aarch64-linux-gnu
```

Listing 4.5: Install GCC `C` compiler with aarch64 codegen.

This works on the ECU to compile a simple hello world program:

```
#include<stdio.h>
int main() {
    printf("Hello␣World!\n");
    return 0;
}
```

Listing 4.6: Simple compiler test program.

```
aarch64-linux-gnu-gcc hello.c
./a.out
Hello World!
```

Listing 4.7: Output from compiler test program.

Note that if the program is compiled with the wrong architecture and executed, `sh` will report the vague error "file/directory not found" instead of reporting that the CPU architecture is wrong. A c++ compiler is also needed which exists as a similar package

```
sudo apt install g++-aarch64-linux-gnu
```

Listing 4.8: Install GCC `C++` compiler with aarch64 codegen.

Trying to cross-compile FMIL by using a `cmake` toolchain file now results in a linker error from `ld`, because object files have the wrong format. Turns out it was compiled for x86-64.

```
file ./ExpatEx/CMakeFiles/expat.dir/lib/xmlparse.c.o
./ExpatEx/CMakeFiles/expat.dir/lib/xmlparse.c.o: ELF 64-bit
   LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

Listing 4.9: Filetype of one of the object files.

Expat is the Extensible Markup Language (XML) library used by FMIL. Turns out cmake arguments are insufficient to change the active compiler, so the following environment variables have to also be set in addition to setting the correct compiler in the toolchain file.

```
export CC='aarch64-linux-gnu-gcc'
export CXX='aarch64-linux-gnu-g++'
export CMAKE_C_COMPILER='aarch64-linux-gnu-gcc'
export CMAKE_CXX_COMPILER='aarch64-linux-gnu-g++'
```

Listing 4.10: Set environment variables.

The full build script can be found in Appendix E FMIL `aarch64` build script, and the toolchain file can be found in Appendix F FMIL `aarch64` `cmake` toolchain file.

4.3 Boot and Communication with ECU

This section describes the boot switches and serial communication with the ECU. The documentation for the boot mode switches were unclear since the sequence of up/down may be reversed and correspond to either on/off or off/on depending on orientation. The documentation from diffrent sources also contradicts each other. We tried booting without setting the proper boot mode and it worked, so it seems to boot from the SD card by default.

minicom was used to do serial communication, in our case we used /dev/USB1 with the baud rate 115200 because that is a baud rate supported by the ECU.

```
sudo apt-get install minicom
sudo minicom -D /dev/ttyUSB1 -b 115200
```

Listing 4.11: Install and start minicom.

Before starting the ECU, hardware access flow control needs to be disabled which is done by pressing CTRL-A, Z, O, Serial Port setup and then toggling hardware access flow control. Note that not disabling hardware access flow control makes it impossible to send keypresses to the board. The ECU can now be started and it should display a login prompt.

4.4 Statically linking FMU

This is described in Appendix H

4.5 Application performance profiling

To profile the application perf [5] is used, in this case, the version of perf for the current kernel has to be specified:

```
sudo apt install linux-tools-common linux-tools-generic
   linux-tools-‘uname -r‘
sudo perf record ./run.sh
sudo perf report --hierarchy -M intel
```

Listing 4.12: Install and run perf

To try to avoid profiling FMU parsing, the simulation time for the HeatLosses was increased from 1 second to 100 seconds. The output of this can be found in Section 5.3 Performance bottlenecks and profiling. We were only able to get `perf` running on the desktop and not the ECU, although there is not a technical reason making it impossible to run it on the ECU.

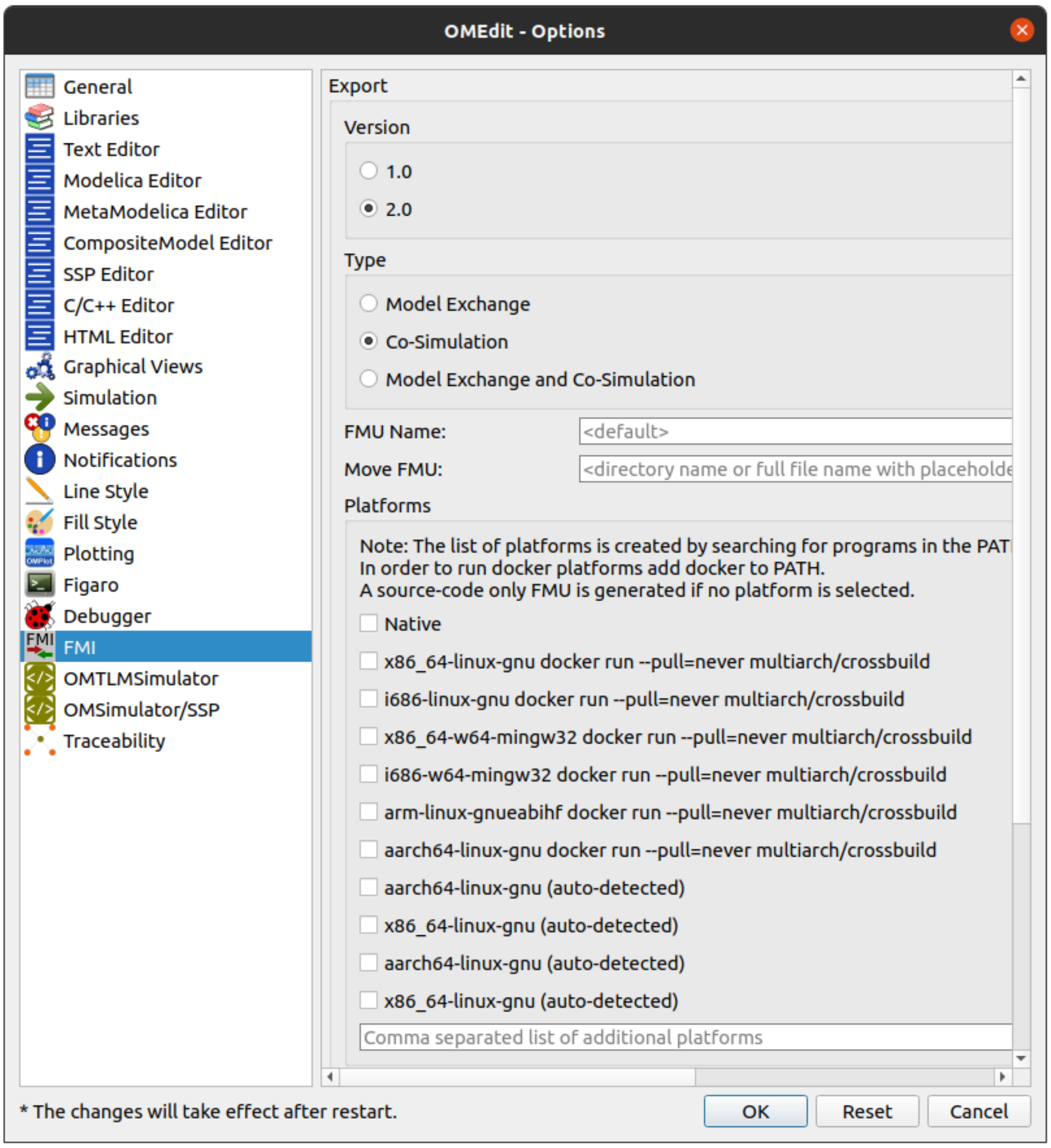
4.6 Binary FMU Creation for `aarch64`

A source based FMU needs to be compiled in order to run on the ECU. A python program was created to invoke `make` which in turn invokes aarch64 or x86-64 versions of GCC to compile the FMU and then finally package it again. The python program and makefile can be found in Appendix A FMU Python compilation program and Appendix B FMU makefile.

4.7 Use of OpenModelica to generate FMUs

When the FMU models were generated using Modelon Impact, they were not able to run on the ECU because they triggered a segmentation fault (see section 3.4). Therefore, the same FMUs was instead generated using OpenModelica. In Figure 4.1 the different compile options available in OMEdit are listed and there are compile options both for aarch64 and x86-64. This meant that the FMU models from OpenModelica did not need to use the `Python` compilation program.

When running the HeatLosses FMU with a low communication frequency, the internal solver of the FMU could not run the simulation properly. Therefore the communication frequency was set to match the internal step size of the FMU. Due to the high communication frequency, the FMU now communicated with the runtime environment 10000 times per second, resulting in a high amount of function calls. This increased



Export

Version

( ) 1.0
(●) 2.0

Type

( ) Model Exchange
(●) Co-Simulation
( ) Model Exchange and Co-Simulation

FMU Name: <default>

Move FMU: <directory name or full file name with placeholde

Platforms

Note: The list of platforms is created by searching for programs in the PAT
In order to run docker platforms add docker to PATH.
A source-code only FMU is generated if no platform is selected.

- [ ] Native
- [ ] x86_64-linux-gnu docker run --pull=never multiarch/crossbuild
- [ ] i686-linux-gnu docker run --pull=never multiarch/crossbuild
- [ ] x86_64-w64-mingw32 docker run --pull=never multiarch/crossbuild
- [ ] i686-w64-mingw32 docker run --pull=never multiarch/crossbuild
- [ ] arm-linux-gnueabihf docker run --pull=never multiarch/crossbuild
- [ ] aarch64-linux-gnu docker run --pull=never multiarch/crossbuild
- [ ] aarch64-linux-gnu (auto-detected)
- [ ] x86_64-linux-gnu (auto-detected)
- [ ] aarch64-linux-gnu (auto-detected)
- [ ] x86_64-linux-gnu (auto-detected)

Comma separated list of additional platforms

* The changes will take effect after restart.

Figure 4.1: Compilation options in the OMEdit program.

the execution time significantly. To reduce the overhead, the communication frequency was lowered until the solver no longer could process the simulation, resulting in 1000 communications times per second.

Since V.A.S. still were interested in FMUs compiled using GCC, the `python` compilation program will still be used on another FMU. The Bouncing Ball FMU compiled with GCC using the `Python` compilation program will be part of the tests in the C program.

Chapter 5

Results

This chapter presents and discusses the results of the work. Section 5.1 discusses the real-time execution results. Section 5.2 discusses the results regarding system differences. Section 5.3 discusses profiling and bottleneck results.

5.1 Real-time performance margin

To be able to determine if the ECU is capable of running a given FMU model in real-time the execution time is compared to the simulation time. In Table 5.1 the simulation time and execution time of the FMUs used are compared and a performance margin is calculated. The performance margin is calculated by the formula:

$$\text{Performance margin} = 1 - \frac{\text{Execution time}}{\text{Simulation time}}$$

Benchmarks (30 samples)	**Desktop SD (ms / %)**	**ECU SD (ms / %)**
Bouncing Ball FMU	0.039 49%	0.015 1.4%
Rotational First FMU	0.021 23%	0.031 2.0%
Rotational HeatLosses FMU	10 22%	7.5 1.0%
Recursive Fibonacci	1.7 0.8 %	1.2 0.046%

Table 5.2: Standard Deviation (SD) from benchmarks on the desktop computer and ECU for each FMU and what proportion of the average execution time the SD is.

FMU	**Simulation Time (ms)**	**Execution time (ms)**	**Performance margin**
Bouncing Ball FMU	3000	1.1	99.96%
Rotational First FMU	1000	1.6	99.84%
Rotational HeatLosses FMU	1000	750	25%

Table 5.1: Simulation time, execution time and the performance margin on the ECU for each FMU.

As seen in Table 5.1, the lowest performance margin was for the Rotational Heat-Losses FMU and it was 25%, showing that the ECU is capable of running the FMU faster than real-time with a significant margin. In Table 5.2 it can be seen that the SD from the ECU for this test is 1.0%, indicating that the ECU is running the model with low variance in execution speed. When taking the SD into account it is clear that the performance margin is well above what is required.

FMU	**Simulation Step Size (s):**	**Simulation Step Frequency (Hz):**
Bouncing Ball	0.01	100
Rotational First	0.01	100
Rotational Heatlosses	0.001	1000

Table 5.3: Simulation step sizes and the communication frequency for the used FMUs.

Due to limitations in the internal solver of the Modelica HeatLosses FMU the size of the simulation time step needed to be limited. In Table 5.3 the different simulation step sizes are listed. The step size of the Modelica Rotational HeatLosses are 10 times smaler than the other FMUs listed in the table, resulting in 10 times as many function calls to `fmi2_import_do_step`. This will have a negative impact on the execution time and a model without this restriction may give a better result, but the choice or design of a solver in the FMU is outside the scope of this thesis.

5.2 System performance differences

The results from running the Appendix C Benchmarking application on the ECU and the desktop computer can be found in Table 5.4. In the table, the average execution times of the two systems is shown along with the multiplicative difference. Execution times are between 12-17 times faster on the desktop computer. The lowest difference is found in the Fibonacci benchmark, indicating that there are factors outside CPU performance that are affecting the execution time of the FMUs on the ECU.

To get an approximation of what the execution speed on the ECU would be the CPU clock frequency were used as a baseline. In Figure 5.1, Figure 5.2, Figure 5.3 and Figure 5.4 the y-axis shows the time in *ms* and the x-axis show from left to right the ECU, the expected time from the ECU when adjusting for the difference in CPU clock speed and the desktop. In the figures the expected execution time is displayed, along

Test Name	Desktop Average (ms)	ECU Average (ms)	Multiplicative difference
Bouncing Ball FMU	0.080	1.1	14
Rotational First FMU	0.091	1.6	17
Rotational HeatLosses FMU	46	750	17
Recursive Fibonacci	210	2600	12

Table 5.4: Time averages from benchmarks on the desktop computer and ECU for each FMU. A bar plot of the data can be found in Figure 5.1, Figure 5.2, Figure 5.3 and Figure 5.4.

with the the measured execution time of the ECU on the left and the execution time of the desktop computer on the right. To get the expected time the following formula was used:

$$\text{Expected time} = \text{Desktop time} \cdot \frac{\text{Desktop CPU clock frequency}}{\text{ECU CPU clock frequency}}$$

The clock frequency of the desktop computer CPU is 4.8 GHz and the ECU CPU is 1.3 GHz, resulting in a multiplicative difference of approximately 3.7. This is a fairly inaccurate way of comparing CPUs, since it entirely ignores architectural differences, but is still indicates to some extent what performance can be expected. Table 5.4 confirms that clock speed differences accounts for only a small part of the total execution time differences between the ECU and desktop computer since the multiplicative difference is significantly higher than what would be expected from just clock speed differences.

When comparing the SD in Table 5.2 of the desktop computer and the ECU, the difference is not very significant. But when adjusting for the average execution time, which is shown by the percentage bellow the SD in Table 5.2, there are major differences. When only looking at the different FMUs the consistency of the ECU is down to 1-2% of the execution time, where the desktop is above 20%. This consistency is an important aspect when aiming to run real-time software on the ECU.

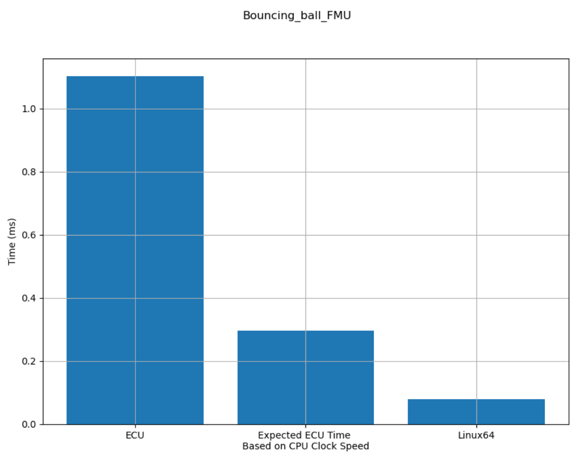


Figure 5.1: Bar plot of the execution times on the desktop, ECU and an the expected execution time on the ECU from the Bouncing Ball FMU

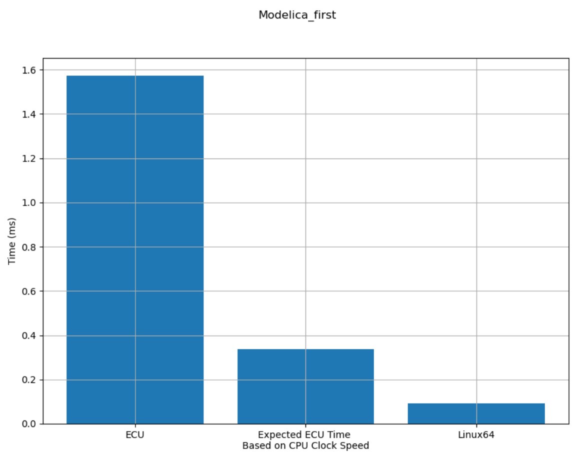


Figure 5.2: Bar plot of the execution times on the desktop, ECU and an the expected execution time on the ECU from the Rotational First FMU

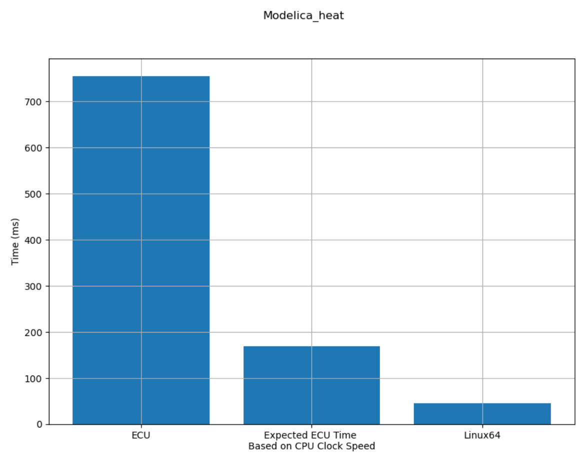


Figure 5.3: Bar plot of the execution times on the desktop, ECU and an the expected execution time on the ECU from the Rotational HeatLosses FMU

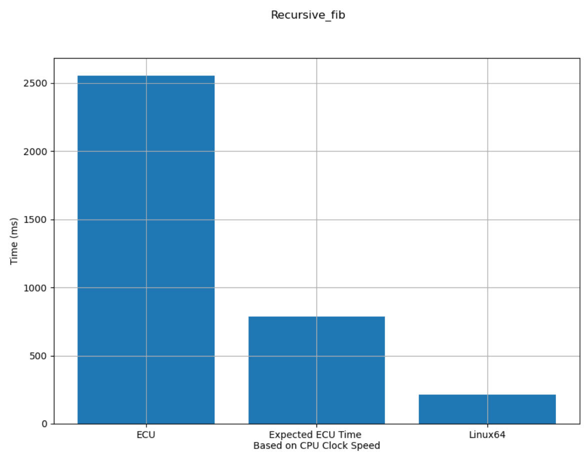


Figure 5.4: Bar plot of the execution times on the desktop, ECU and an the expected execution time on the ECU from the Fibonacci benchmark

5.3 Performance bottlenecks and profiling

In order to find bottlenecks `perf` was used to analyze what the program does when running a FMU simulation. The Modelica Rotational HeatLosses FMU was run for 100 seconds when profiling. This is in order to maximize the simulation time relative to the overhead time of loading and unloading the FMU.

5.3.1 Findings from `perf`

`perf` [5] prints the following output (this information is drawn in Figure 5.5):

```
100.00%       main
    55.48%
       Modelica_Mechanics_Rotational_Examples_HeatLosses.so
                     [.] ...
    25.11%        libpthread-2.31.so
       13.50%        [.] __pthread_mutex_lock
       11.52%        [.] __pthread_mutex_unlock
        0.09%        [.] __pthread_setspecific
    09.60%        [kernel.kallsyms]
                     [k] ...
     7.58%        libc-2.31.so
        4.58%        [.] __memset_avx2_unaligned_erms
        0.82%        [.] __memmove_avx_unaligned_erms
        0.52%        [.] _int_free
        0.45%        [.] _int_malloc
        0.32%        [.] __libc_calloc
        0.24%        [.] __strncmp_avx2
        0.22%        [.] malloc
        0.18%        [.] __sigsetjmp
        0.15%           [.] cfree@GLIBC_2.2.5
                        [.] ...
    1.85%           libfmilib_shared.so
                        [.] ...
    0.26%           libm-2.31.so
        0.26%           [.] __sin_fma
    0.12%           main
                        [.] ...
                    ...
```

Listing 5.1: perf output.

Note that perf [5] counts cycles across all threads, so a 4 core CPU at full utilization would display 400%, the total in this case is slightly above 100%.

Figure 5.5 shows how much time the program is using to interact with the FMU. It shows that about half of the CPU time is spent in the FMU itself, which is not ideal. A significant amount of cycles are spent on locking and unlocking mutexes, memory allocation, deallocation and kernel calls. Mutexes are used in multi-threaded applications to synchronize threads, mostly to have sections of code where only a single thread may execute in order to remove data races. High quality mutex implementations generally make calls to the kernel in order to make sure that the overall system is not wasting CPU cycles, instead of using spinlocks which may cause priority inversion if the locked thread is put to sleep by the CPU scheduler. While that is good in the general case, it becomes an expensive operation when called frequently, so high performance software generally tries to use lock-free data structures where possible.

Since the FMU seems to be running in a single thread, it does not really make sense for it to be using synchronization primitives in the first place. Additionally, since the physical system is closed, the amount of state should be constant, and therefore no memory allocation or deallocation should occur during simulation.

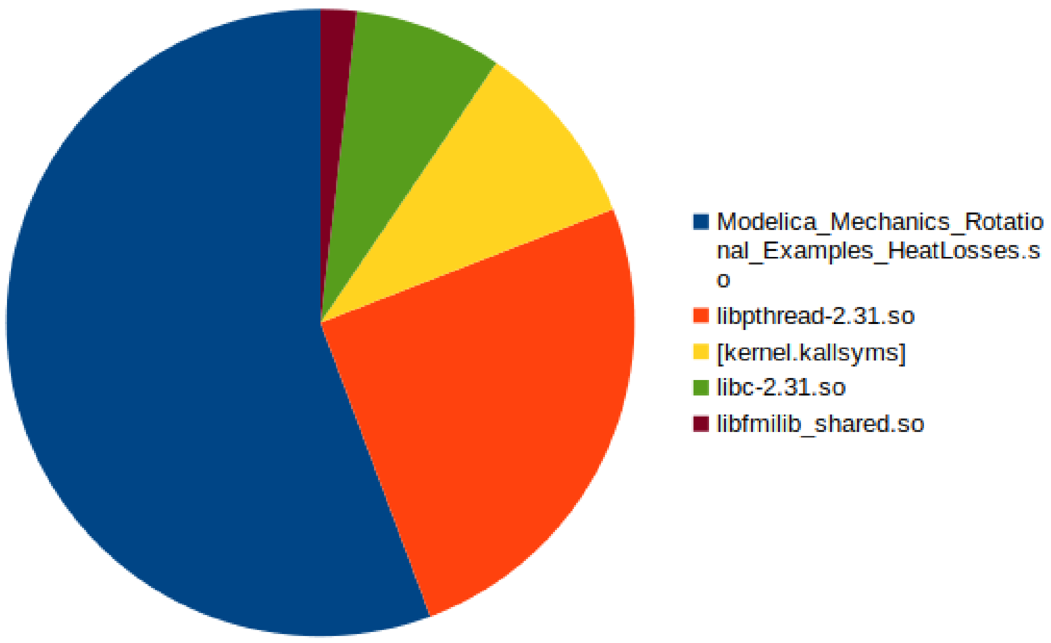


Figure 5.5: Modelica Heatlosses FMU profiling output from `perf` as a pie-chart.

5.3.2 CPU Utilization and Memory Usage

The memory usage shown in Figure 5.6 increased approximately linearly throughout execution. Since the FMU is using a custom memory arena to manage memory, this is a bit misleading, but at least shows that allocations are constantly made and that the arena memory is not released often enough. As previously mentioned, these allocations do not need to be made in the first place. The CPU usage shown in Figure 5.7 was mostly constant, with a single thread, but sometimes switched between physical cores.

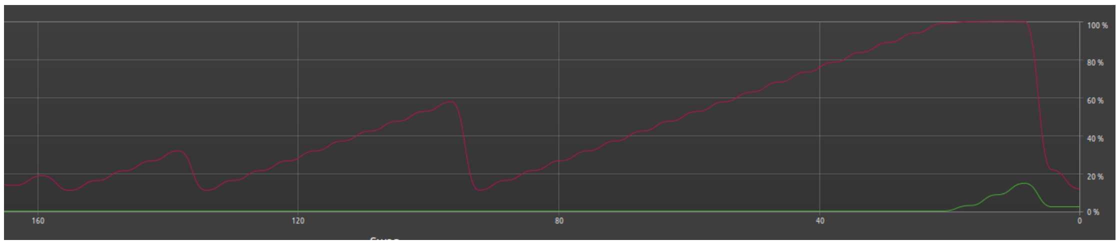


Figure 5.6: Memory usage when running Modelica Rotational HeatLosses FMU.

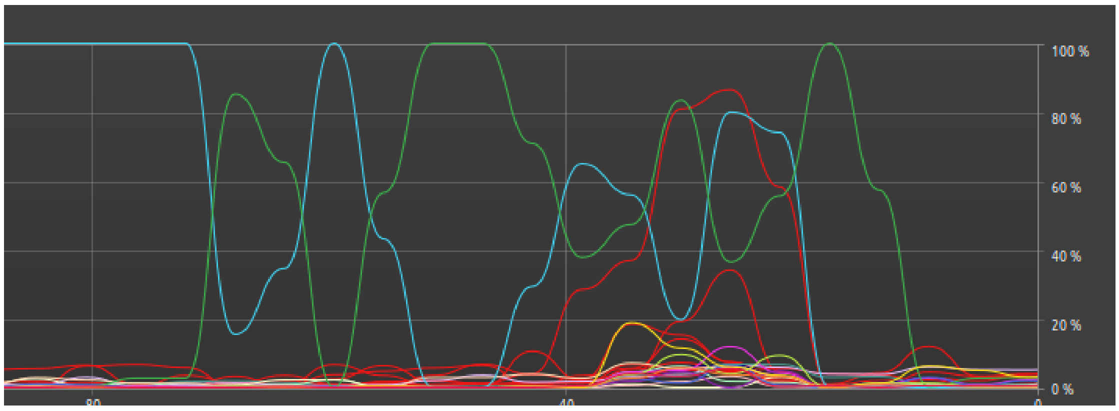


Figure 5.7: CPU usage when running Modelica Rotational HeatLosses FMU.

Chapter 6

Conclusion

This chapter discusses the conclusions that can be made from the project. Section 6.1 discusses how well the goals of the project are achieved. Section 6.2 discusses conclusions of the thesis. Section 6.3 mentions potential areas of future work.

6.1 Discussion

It is clear that the ECU is able to run the provided FMUs significantly faster than real-time, however it was also shown that a significant amount of time was not spent on running the actual simulation. There were also a clear difference in execution time where the ECU runs 12-17 times slower than the desktop and 3-5 times slower if the difference in clock frequency is accounted for. This indicates that the available performance of the ECU is not utilized well.

The FMU communication frequency, allocation, compilation flags, Function Pointer (FP) overhead and the use of mutexes are potential performance bottlenecks that we have identified. The high communication frequency results in overhead because the communication is not free since the internal representation has to be ready to be sent through the FMI API, and execution momentarily switches from the FMU to the main application. The results show that a higher communication frequency increases execution time, showing that it has a significant overhead. The FMU models from OMEdit allocated memory throughout execution, but it should be possible to eliminate this. Because of time constraints, hardware specific compilation flags where not experimented with, even though they could have taken better advantage of the available instructions on the ECU. Because both FMIL and the FMUs are dynamically linked, there are several layers of pointer indirection when using the FMI API, and compiler inlining optimizations are not possible. Pointer indirection causes pipeline stalls and requires more memory operations that depend on each other. The OMEdit FMUs used unnecessary threading and mutexes, this was shown to account for a significant amount of execution time and is not necessary since only one threads worth of clock cycles was utilized at a time.

Initially we intended to use the V.A.S. models for a full vehicle, but that was not possible due to dependencies not being available for aarch64. The models that where actually used where quite simple, making it hard to generalize the results to more complex models. The results still give an indication of the performance of the ECU.

6.2 Conclusion

The work on compiling FMUs to the ECU Instruction Set Architecture (ISA) can still be applied in future work when compiling new FMUs. The results indicate that the ECU is able to run the tested models and potentially more complex models if they are better optimized to suit the ECU, that it runs significantly slower than expected compared to a desktop computer and that there are significant unresolved bottlenecks.

6.3 Future works

There are various areas to explore to extend the work done in this project. More advanced FMUs with lower overhead could be used, for example with a lower communication frequency but with more active systems. Various ways of compiling could be used, for example static linking, using different compilers and enabling flags for hardware specific instructions and optimizations to be used. The installation of PetaLinux was not changed much but it could be modified to optimize specific workloads, or to include profiling and debugging tools to get a better understanding of the ECU performance characteristics. Additionally, for hard real-time systems the maximum execution time would be valuable.

Bibliography

[1] Modelica Association Project. *Functional Mock-up Interface for Model Exchange and Co-Simulation*, 2.0 edition, July 2014.

[2] Petalinux tools and operating system [accessed may 10 2023]. `https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html`.

[3] Sae standards news: J3016 automated-driving graphic update [accessed may 10 2023]. `https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic`.

[4] Fmi library [accessed may 10 2023]. `https://github.com/modelon-community/fmi-library`.

[5] perf - performance analysis tools for linux [accessed may 10 2023]. `https://www.man7.org/linux/man-pages/man1/perf.1.html`.

[6] Volvo autonomus solutions [accessed may 10 2023]. `https://www.volvoautonomoussolutions.com/en`.

[7] Volvo cars [accessed may 10 2023]. `https://www.media.volvocars.com/se/sv-se/corporate/this-is-volvo`.

[8] Volvo group [accessed may 10 2023]. https://www.volvogroup.com/se/about-us/what-we-do/our-brands.html.

[9] Embedded system [accessed may 10 2023]. https://www.techtarget.com/iotagenda/definition/embedded-system.

[10] What is an electronic control unit (ecu)? [accessed may 10 2023]. https://www.autopi.io/blog/what-is-electronic-control-unit-definition/.

[11] Electronic control units (ecu) information [accessed may 10 2023]. https://www.globalspec.com/learnmore/specialized_industrial_products/transportation_products/automotive_equipment_products/electronic_control_units_ecu.

[12] What is an ecu? electronic control unit (ecu) explained [accessed may 10 2023]. https://www.carandbike.com/news/what-is-an-ecu-electronic-control-unit-ecu-explained-2703492.

[13] What is linux? [accessed may 10 2023]. https://opensource.com/resources/linux.

[14] The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v [accessed may 10 2023]. https://arxiv.org/abs/1607.02318.

[15] dlopen(3) - linux man page [accessed may 10 2023]. https://linux.die.net/man/3/dlopen.

[16] Cross compilation toolchain for arm example with raspberry pi [accessed may 10 2023]. https://microcontrollerslab.com/cross-compilation-toolchain-for-arm-example-with-raspberry-pi/.

[17] Openmodelica [accessed may 10 2023]. `https://openmodelica.org/`.

[18] Cpu benchmarks are crucial in determining whether your processor can run the games and applications you like. heres what you need to get started. [accessed may 10 2023]. `https://www.intel.com/content/www/us/en/gaming/resources/read-cpu-benchmarks.html`.

[19] modelica/reference-fmus/bouncingball/ [accessed may 10 2023]. `https://github.com/modelica/Reference-FMUs/tree/main/BouncingBall`.

[20] Modelica.mechanics.rotational.examples.first [accessed may 10 2023]. `https://build.openmodelica.org/Documentation/Modelica.Mechanics.Rotational.Examples.First.html`.

[21] Modelica.mechanics.rotational.examples.heatlosses [accessed may 10 2023]. `https://build.openmodelica.org/Documentation/Modelica.Mechanics.Rotational.Examples.HeatLosses.html`.

[22] Petalinux tools installer [accessed may 10 2023]. `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html`.

[23] 73686 - petalinux 2020.1 - product update release notes and known issues [accessed may 10 2023]. `https://support.xilinx.com/s/article/73686?language=en_US`.

Appendices

Appendix A

FMU Python compilation program

```
# Script to compile source based FMUs into binary FMUs

import os
import subprocess
import sys
import xmltodict

import zipfile
import shutil
def mkdir_clear(directory):
    try:
        os.mkdir(directory)
        print("creating", directory)
    except FileExistsError:
        print("clearing", directory)
        shutil.rmtree(directory)
        os.mkdir(directory)

def mkdir(directory):
    try:
        os.mkdir(directory)
        print("creating", directory)
    except FileExistsError:
        pass
        print("already␣created", directory)


def main(args):
    compile_only = False

    if len(args) < 1:
        print(len(args), args)
        print("Usage:␣./compile_fmu.py␣[-s␣(=␣compile␣only)]
           ␣[source␣fmus]")
        print("Example:␣./compile_fmu.py␣BouncingBall.fmu␣
           foobar.fmu")
    else:


        for fmu_str in args:
            if fmu_str == "-s":
                compile_only = True
                continue

            # [(compiler, folder)]
            gcc_arch = [("aarch64-linux-gnu-gcc", "linux64")
                , ("aarch64-linux-gnu-gcc", "aarch64-linux-
                gnu")]

            tmp_dir = "tmp_compile_fmu"


            if not compile_only:
                mkdir_clear(tmp_dir)

                os.altsep = "\\"
                with zipfile.ZipFile(fmu_str, 'r') as
                   zip_ref:
                    zip_ref.extractall(tmp_dir)

                for (folder, _, _) in os.walk(tmp_dir):
                    fixed_name = folder.replace("\\", "/")
                    print("rename:", folder, "->",
                       fixed_name)
                    os.rename(folder, fixed_name)

            os.chdir(tmp_dir)

            if not compile_only:
                for filename in os.listdir():
                    fixed_filename = filename.replace("\\",
                       "/")
                    if filename != fixed_filename:
                        print("rename:", filename, "->",
                           fixed_filename)
                        os.rename(filename, fixed_filename)
```

```
mkdir("binaries")
os.chdir("binaries")

for (gcc_str, arch_str) in gcc_arch:

    mkdir_clear(arch_str)
    os.chdir(arch_str)

    mkdir_clear("tmp_build")
    os.chdir("tmp_build")

    os.environ["CC"] = gcc_str

    subprocess.run(["ls", "-l"])

    print(gcc_str)
    print(arch_str)
    doc = xmltodict.parse(open("../../../
       modelDescription.xml").read())["
       fmiModelDescription"]["CoSimulation"]
    libname = doc["@modelIdentifier"]
    print(libname)
    print("Compiling...")
    print(subprocess.run("make␣-f␣../../../../
       make_fmu", shell=True, stdout=subprocess.
       PIPE, stderr=subprocess.STDOUT, text=True
       ).stdout)

    os.rename("libthing.so", "../" + libname + "
       .so")

    os.chdir("..") # build dir
    shutil.rmtree("tmp_build")
    os.chdir("..") # arch dir

os.chdir("..") # binaries dir
os.chdir("..") # tmp dir

name = "COMPILED_"+os.path.basename(fmu_str)
name_zip = name + ".zip"
print(name)
```

```
            print(tmp_dir)
            shutil.make_archive(name, "zip", tmp_dir)
            os.rename(name_zip, name)
            if not compile_only:
                input("files␣extracted/compiled.␣Clearing␣
                   tmp␣dirs...␣(press␣enter␣or␣C-c␣to␣cancel
                   ...)")
                print("Removing␣folder:", tmp_dir)
                shutil.rmtree(tmp_dir)


if __name__ == "__main__":
    main(sys.argv[1:])
```

Listing A.1: compile_fmu.py

Appendix B

FMU makefile

```
# Follow build instructions, do not run directly.

# This was based on https://github.com/Leandros/Generic-
   Makefile/blob/master/Makefile

LD = $CC
RM                          = rm -rf
RMDIR              = rmdir
DEBUG              = -ggdb -O0 -ftrapv

TARGET             = foo
SRCDIR             = ../../../sources
OBJDIR             = obj
BINDIR             = bin

# CFLAGS, LDFLAGS, CPPFLAGS, PREFIX can be overriden on CLI
CFLAGS             += $(DEBUG) -I$(SRCDIR)
CPPFLAGS           :=
LDFLAGS            :=
PREFIX             := /usr/local
TARGET_ARCH :=

# Compiler Flags
ALL_CFLAGS                  := $(CFLAGS)
ALL_CFLAGS                  += -Wall -Wextra -pedantic -ansi
ALL_CFLAGS                  += -fno-strict-aliasing
ALL_CFLAGS                  += -Wuninitialized -Winit-self -
   Wfloat-equal
ALL_CFLAGS                  += -Wundef -Wshadow -Wc++-compat -
   Wcast-qual -Wcast-align
```

```
ALL_CFLAGS                += -Wconversion -Wsign-conversion -
   Wjump-misses-init
ALL_CFLAGS                += -Wno-multichar -Wpacked -Wstrict-
   overflow -Wvla
ALL_CFLAGS                += -Wformat -Wno-format-zero-length
   -Wstrict-prototypes
ALL_CFLAGS                += -Wno-unknown-warning-option

# Preprocessor Flags
ALL_CPPFLAGS    := $(CPPFLAGS)

# Linker Flags
ALL_LDFLAGS               := $(LDFLAGS)
ALL_LDLIBS                := -lc

# Source, Binaries, Dependencies
SRC                       := $(shell find $(SRCDIR) -type f -
   name '*.c')
OBJ                       := $(patsubst $(SRCDIR)/%,$(OBJDIR)
   /%,$(SRC:.c=.o))
DEP                       := $(OBJ:.o=.d)
BIN                       := $(BINDIR)/$(TARGET)
-include $(DEP)

# Verbosity Control, ala automake
V                         = 0

# Verbosity for CC
REAL_CC         := $(CC)
CC_0            = @echo "CC␣$<"; $(REAL_CC)
CC_1            = $(REAL_CC)
CC                        = $(CC_$(V))

# Verbosity for LD
REAL_LD         := $(LD)
LD_0            = @echo "LD␣$@"; $(REAL_LD)
LD_1            = $(REAL_LD)
LD                        = $(LD_$(V))

# Verbosity for RM
REAL_RM         := $(RM)
RM_0            = @echo "Cleaning..."; $(REAL_RM)
RM_1            = $(REAL_RM)
RM                        = $(RM_$(V))
```

```
# Verbosity for RMDIR
REAL_RMDIR      := $(RMDIR)
RMDIR_0         = @$(REAL_RMDIR)
RMDIR_1         = $(REAL_RMDIR)
RMDIR           = $(RMDIR_$(V))

# Build Rules
.PHONY: clean
.DEFAULT_GOAL := all

all: setup $(BIN)
setup: dir
remake: clean all

dir:
        @mkdir -p $(OBJDIR)
        @mkdir -p $(BINDIR)

$(BIN): $(OBJ)
        echo "Make␣shared␣library"
        $(CC) $(ALL_CFLAGS) -shared -o libthing.so obj/*.o -
           lpthread $(CFLAGS_POST)

$(OBJDIR)/%.o: $(SRCDIR)/%.c
        $(CC) $(ALL_CFLAGS) $(ALL_CPPFLAGS) -fPIC -c -MMD -
           MP -lpthread -o $@ $<


clean:
        $(RM) $(OBJ) $(DEP) $(BIN)
        $(RMDIR) $(OBJDIR) $(BINDIR) 2> /dev/null; true
```

Listing B.1: make_fmu

Appendix C

Benchmarking application

```
// NOTE: This program was only used to test performance.
// DO NOT assume correctness nor memory safety

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
#include <time.h>
#include <limits.h>
#include <fmilib.h>
#include <math.h>
#include <unistd.h>
#include <execinfo.h>
#include <signal.h>
#include <stdint.h>

#define BALL_SIM_TIME 3
#define BALL_SIM_STEPS 0.1

#define CSV_BALL_FILE_PATH "ball_res.csv"
#define ARR_LEN(x) (sizeof(x) / sizeof((x)[0]))
#define TRACE(s) if (1) {printf("TRACE(%s)\n", s);}


void handler(int sig) {
  fprintf(stderr, "main.c:␣handler()");
  printf("main.c:␣handler()");
  fflush(stdout);
  void *array[10];
  size_t size;
  size = backtrace(array, 10);
  fprintf(stderr, "Error:␣signal␣%d:\n", sig);
  printf("Error:␣signal␣%d:\n", sig);
  fflush(stdout);
  backtrace_symbols_fd(array, size, STDERR_FILENO);
  exit(1);
}

void prepare_fmu(fmi_import_context_t** context /*out*/,
   fmi2_import_t** fmu /*out*/,jm_callbacks* callbacks /*out
    (lifetime)*/,const char * fmu_path, const char *
   tmp_path);
void cleanup_fmu_context(fmi_import_context_t* context,
   fmi2_import_t* fmu);

clock_t bench_time_start;
clock_t bench_time_end;
void bench_timer_start() { bench_time_start = clock(); }
void bench_timer_end() { bench_time_end = clock(); }

void importlogger(jm_callbacks* c, jm_string module,
   jm_log_level_enu_t log_level, jm_string message) {
    printf("module␣=␣%s,␣log␣level␣=␣%d:␣%s\n", module,
       log_level, message);
}
void fmi2logger(fmi2_component_environment_t env,
   fmi2_string_t instanceName, fmi2_status_t status,
   fmi2_string_t category, fmi2_string_t message, ...) {
#define BUFFER 1000
    char msg[BUFFER];
    va_list argp;
    va_start(argp, message);
    jm_vsnprintf(msg, BUFFER, message, argp);
    printf("fmiStatus␣=␣%s;␣␣%s␣(%s):␣%s\n",
       fmi2_status_to_string(status), instanceName, category
       , msg);
#undef BUFFER
}
void stepFinished(fmi2_component_environment_t env,
   fmi2_status_t status) {
    printf("stepFinished␣is␣called␣with␣fmiStatus␣=␣%s\n",
       fmi2_status_to_string(status));
}
void check_ok_jm(jm_status_enu_t j) {
    switch (j) {
        case jm_status_error:
            {
                printf("jm␣error,␣exiting...\n");
                exit(1);

            } break;
        case jm_status_warning:
            {
                printf("jm␣warning,␣exiting...\n");
                exit(1);
            } break;
        case jm_status_success:
            break;
    }

}

#define assert_ok_jm(j) assert((j)==jm_status_success)
#define assert_ok_fmi2_status(f) assert((f)==fmi2_status_ok)

void set_real_by_name_fmi2(fmi2_import_t* fmu, const char *
   name, fmi2_real_t real_val) {
    fmi2_import_variable_t* variable_pointer =
       fmi2_import_get_variable_by_name (fmu, name);
    assert(variable_pointer);
    fmi2_value_reference_t variable_reference =
       fmi2_import_get_variable_vr (variable_pointer);
    const size_t nvr = 1;
    fmi2_value_reference_t vr[1] = {variable_reference};
    fmi2_real_t value[1] = {real_val};
    assert_ok_fmi2_status(fmi2_import_set_real(fmu, vr, nvr,
        value));
}

// Sets "real_val" to value with reference name "name"
// Returns fmi2_status_t
// for details, see fmiGetReal documentation
fmi2_status_t get_real_by_name(fmi2_import_t* fmu, const
   char * name, fmi2_real_t* real_val) {
    fmi2_import_variable_t* variable_pointer =
       fmi2_import_get_variable_by_name (fmu, name);
    assert(variable_pointer);
    fmi2_value_reference_t variable_reference =
       fmi2_import_get_variable_vr (variable_pointer);
    const size_t nvr = 1;
    fmi2_value_reference_t vr[1] = {variable_reference};
    return fmi2_import_get_real(fmu, vr, nvr, real_val);
}


// Returns the fmi2 type of the fmu kind
fmi2_fmu_kind_enu_t get_fmu_kind(fmi2_import_t* fmu) {
    TRACE("get_fmu_kind");
    fmi2_fmu_kind_enu_t fmu_kind = fmi2_import_get_fmu_kind(
       fmu);
    assert(fmu_kind);
    switch (fmu_kind) {
        case fmi2_fmu_kind_unknown:
            {
                printf("unknown␣fmu␣kind\n");
                exit(1);
            } break;
        case fmi2_fmu_kind_me:
            {
                printf("model␣exchange␣is␣not␣suported\n");
                exit(1);

            } break;
        case fmi2_fmu_kind_cs:
            {
                return fmi2_fmu_kind_cs;
            } break;
        case fmi2_fmu_kind_me_and_cs:
            {
                printf("FMU␣supports␣co-simulation␣AND␣model
                   ␣exchange\n");
                printf("Using␣Co␣Simulation\n");
                return fmi2_fmu_kind_cs;
            } break;
        default:
            printf("Error␣in␣reading␣FMU␣Kind...␣Exiting\n")
               ;
            exit(1);
    }
}

fmi_version_enu_t get_fmu_version(fmi_import_context_t*
   context, const char* FMUPath, const char* tmpPath) {
    TRACE("get_fmu_version");
    fmi_version_enu_t version;
    version = fmi_import_get_fmi_version(context, FMUPath,
       tmpPath);
    if (version != fmi_version_2_0_enu) {
        printf("The␣code␣only␣supports␣version␣2.0\n");
        exit(1);
    }
    return version;
}


bool csv_add_header(FILE* fpt, char** value_list, int
   value_number) {
    TRACE("csv_add_header");
    int i;
    for (i=0;i<value_number;i++) {
        if (fprintf(fpt, "%s", value_list[i]) < 0) {
            return false;
        }
        if (i == value_number-1) {
            fprintf(fpt, "\n");
        } else {
            fprintf(fpt, ",");
        }
    }
    return true;
}

bool csv_add_line(FILE* fpt, fmi2_real_t* value_list, int
   value_number) {
    TRACE("csv_add_line");
    int i;
    for (i=0;i<value_number;i++) {
        if (fprintf(fpt, "%g", value_list[i]) < 0) {
            return false;
        }
        if (i == value_number-1) {
            fprintf(fpt, "\n");
        } else {
            fprintf(fpt, ",");
        }
    }
    return true;
}
```

```
void run_ball_simulation(fmi2_import_t* fmu, fmi2_real_t
   tstart, fmi2_real_t tend, fmi2_real_t hstep) {
    TRACE("run_ball_simulation");
    fmi2_real_t ball_h, ball_v, time;
    fmi2_boolean_t newStep = fmi2_true;

    printf("\nStarting␣simulation:\n");
    fmi2_real_t tcur = tstart;

    FILE* fpt = fopen(CSV_BALL_FILE_PATH, "w+");
    if (fprintf(fpt, "%s,%s,%s\n", "Time", "Ball_Height", "
       Ball_Speed") < 0) {
        printf("%10s␣%15s␣%15s\n", "Time", "h", "v");
    }

    bench_timer_start();
    while (tcur < tend) {
        if (fmi2_import_do_step(fmu, tcur, hstep, newStep)
           != fmi2_status_ok) {
            printf("Simulation␣error:␣%s\nStopping␣
               simulation\n", fmi2_import_get_last_error(fmu
               ));
            exit(1);
        }

        assert_ok_fmi2_status(get_real_by_name(fmu, "time",
           &time));
        assert_ok_fmi2_status(get_real_by_name(fmu, "h", &
           ball_h));
        assert_ok_fmi2_status(get_real_by_name(fmu, "v", &
           ball_v));

        if (fprintf(fpt, "%g,%g,%g\n", time, ball_h, ball_v)
            < 0) {
            printf("%10g␣%15g␣%15g\n", time, ball_h, ball_v)
               ;
        }

        tcur += hstep;
    }
    bench_timer_end();
    if (fclose(fpt) != 0) {
        printf("Could␣not␣close␣the␣.csv␣file\n");
    }
}

void run_simulation(fmi2_import_t* fmu, fmi2_real_t tstart,
   fmi2_real_t tend, fmi2_real_t hstep) {
    TRACE("run_simulation");
    fmi2_boolean_t newStep = fmi2_true;

    printf("\nStarting␣simulation:\n");
    fmi2_real_t tcur = tstart;

    bench_timer_start();
    while (tcur < tend) {
        if (fmi2_import_do_step(fmu, tcur, hstep, newStep)
           != fmi2_status_ok) {
            printf("Simulation␣error:␣%s\nStopping␣
               simulation\n", fmi2_import_get_last_error(fmu
               ));
            exit(1);
        }
        tcur += hstep;
    }
    bench_timer_end();
}



void start_simulation(const char * fmu_path, const char *
   tmp_path, fmi2_real_t tstart, fmi2_real_t tend,
   fmi2_real_t hstep) {
    TRACE("start_simulation");
    fmi_import_context_t* context = NULL;
    fmi2_import_t* fmu = NULL;
    jm_callbacks callbacks;

    fmi2_string_t instanceName = fmu_path;
    fmi2_string_t fmuLocation = NULL;
    fmi2_boolean_t visible = fmi2_false;
    fmi2_real_t relativeTol = 1e-4;

    fmi2_boolean_t StopTimeDefined = fmi2_false;

    prepare_fmu(
                &context /*out*/,
                &fmu /*out*/,
                &callbacks,
                fmu_path,
                tmp_path
                );

    assert_ok_jm(fmi2_import_instantiate(fmu, instanceName,
       fmi2_cosimulation, fmuLocation, visible));
    assert_ok_fmi2_status(fmi2_import_setup_experiment(fmu,
       fmi2_true, relativeTol, tstart, StopTimeDefined, tend
       ));

    printf("fmu(%p)␣", fmu);
    TRACE("fmi2_import_enter_initialization_mode")
    assert_ok_fmi2_status(
       fmi2_import_enter_initialization_mode(fmu));
    TRACE("fmi2_import_exit_initialization_mode")
    assert_ok_fmi2_status(
       fmi2_import_exit_initialization_mode(fmu));

    TRACE("run_simulation")
    run_simulation(fmu, tstart, tend, hstep);

    cleanup_fmu_context(context, fmu);
}


void prepare_fmu(
        fmi_import_context_t** context /*out*/,
        fmi2_import_t** fmu /*out*/,
        jm_callbacks* callbacks /*out (lifetime)*/,
        const char * fmu_path,
        const char * tmp_path
        ) {
    TRACE("prepare_fmu");
    fmi2_callback_functions_t callBackFunctions;
    jm_callbacks c = {
        .malloc = malloc,
        .realloc = realloc,
        .calloc = calloc,
        .free = free,
        .logger = importlogger,
        .log_level = 1, // jm_log_level_debug,
        .context = 0
    };
    *callbacks = c;

    jm_status_enu_t status;

    printf("Creating␣context\n");
    *context = fmi_import_allocate_context(&(*callbacks));
    assert(*context!=NULL);

    fmi_version_enu_t version = get_fmu_version(*context,
       fmu_path, tmp_path);
    printf("FMU␣version:␣%d\n", version);

    printf("Parsing␣XML\n");
    *fmu = fmi2_import_parse_xml(*context, tmp_path, 0);
    assert(*fmu!=NULL);

    fmi2_fmu_kind_enu_t fmu_kind = get_fmu_kind(*fmu);
    printf("FMU␣kind:␣%s\n", fmu_kind == 2 ? "co-simulation"
       : "model-exchange");

    callBackFunctions.logger = fmi2_log_forwarding;
    callBackFunctions.allocateMemory = calloc;
    callBackFunctions.freeMemory = free;
    callBackFunctions.componentEnvironment = *fmu;

    printf("Creating␣dll\n");
    status = fmi2_import_create_dllfmu(*fmu, fmu_kind, &
       callBackFunctions);
    if (status != jm_status_success) {
        printf("Could␣not␣create␣the␣DLL␣loading␣mechanism(C
           -API)␣(error:␣%s).\n", fmi2_import_get_last_error
           (*fmu));
        exit(1);
    }
    printf("dll␣created␣without␣errors/warnings\n");
}

void cleanup_fmu_context(
        fmi_import_context_t* context,
        fmi2_import_t* fmu
        ) {

    assert_ok_fmi2_status(fmi2_import_terminate(fmu));
    fmi2_import_free_instance(fmu);
    fmi2_import_destroy_dllfmu(fmu);
    fmi2_import_free(fmu);
    fmi_import_free_context(context);
}

// Handles initialisation and cleanup in FMIL and calls the
   simulation stage
void start_ball_fmu_simulation(const char * fmu_path, const
   char * tmp_path) {
    TRACE("start_ball_fmu_simulation");
    fmi2_real_t tstart = 0.0;
    fmi2_real_t tcur = tstart;
    fmi2_real_t hstep = BALL_SIM_STEPS;
    fmi2_real_t tend = BALL_SIM_TIME;
    fmi2_boolean_t StopTimeDefined = fmi2_false;

    fmi_import_context_t* context = NULL;
    fmi2_import_t* fmu = NULL;
    jm_callbacks callbacks;

    fmi2_string_t instanceName = "Test␣Bouncing␣Ball␣CS␣mode
       ";
    fmi2_string_t fmuLocation = NULL;
    fmi2_boolean_t visible = fmi2_false;
    fmi2_real_t relativeTol = 1e-4;


    prepare_fmu(
            &context /*out*/,
            &fmu /*out*/,
            &callbacks,
            fmu_path,
            tmp_path
            );

    assert_ok_jm(fmi2_import_instantiate(fmu, instanceName,
       fmi2_cosimulation, fmuLocation, visible));
    assert_ok_fmi2_status(fmi2_import_setup_experiment(fmu,
       fmi2_true, relativeTol, tstart, StopTimeDefined, tend
       ));

    /* set vars here */
    assert_ok_fmi2_status(
       fmi2_import_enter_initialization_mode(fmu));
    assert_ok_fmi2_status(
       fmi2_import_exit_initialization_mode(fmu));

    run_ball_simulation(fmu, tstart, tend, hstep);

    cleanup_fmu_context(context, fmu);
}

const char * bouncing_ball_fmu_path;
const char * fmil_tmp_dir;

__attribute__((noinline)) int fib(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}

struct BenchInfo {
    char* name; // name of bench
    void (*bench)(void);
    double real_time; // real time bench is simulating
};

void bench_ball_fmu_simulation() {
    TRACE("bench_ball_fmu_simulation");
    start_ball_fmu_simulation(bouncing_ball_fmu_path,
       fmil_tmp_dir);
}
void bench_fib_recursion() {
    TRACE("bench_fib_recursion");
    bench_timer_start();
    fib(40);
    bench_timer_end();
}
void bench_mechanics_examples_first() {
    TRACE("bench_mechanics_examples_first");
    start_simulation("
       COMPILED_Modelica_Mechanics_Rotational_Examples_First
       .fmu", fmil_tmp_dir, 0.0, 1.0, 0.01);
}
void bench_mechanics_examples_heat_losses() {
    TRACE("bench_mechanics_examples_heat_losses");
    start_simulation("
       COMPILED_Modelica_Mechanics_Rotational_Examples_HeatLosses
       .fmu", fmil_tmp_dir, 0.0, 1.0, 0.001);
}
double bench_timer_elapsed() {
    TRACE("bench_timer_elapsed");
    double elapsed = ((double) (bench_time_end -
       bench_time_start))/CLOCKS_PER_SEC;
    bench_time_end = clock();
    bench_time_start = bench_time_end;
    return elapsed;
}

// check that size of c types are what we expect.
void sanity_check() {
    int size = 1<<24; // 16 MiB
    uint8_t* a = (uint8_t*) malloc(size);
    *a = 3;
    *(a+size-1) = 3;
    free(a);

    assert(sizeof(void*) == 8);
    assert(sizeof(unsigned int) == 4);
    assert(sizeof(int) == 4);
    assert(sizeof(double) == 8);
    assert(sizeof(char) == 1);
    assert(sizeof(char*) == 8);
    printf("sanity_check():␣ok");
    fflush(stdout);

}

int main(int argc, char *argv[]) {
    signal(SIGSEGV, handler);
    sanity_check();

    int expected_argc = 2 + 1;
    if (argc != expected_argc) {
        printf("Usage:␣./main␣[fmil␣temp␣dir]␣[bouncing␣ball
           ␣fmu]␣(argc␣=␣%d,␣expected␣%d)\n", argc,
           expected_argc);
        exit(1);
    }
    bouncing_ball_fmu_path          = argv[2];
```

```
fmil_tmp_dir           = realpath(argv[1], NULL);


struct BenchInfo tests[] = {
    { .bench = bench_ball_fmu_simulation,                .
       name = "Bouncing_ball_FMU", .real_time = -1 },
    { .bench = bench_mechanics_examples_first,           .
       name = "Modelica_first",     .real_time = -1 },
    { .bench = bench_mechanics_examples_heat_losses,     .
       name = "Modelica_heat",      .real_time = -1 },
    { .bench = bench_fib_recursion,                      .
       name = "Recursive_fib",      .real_time = -1 },
};

printf("NUM_TESTS␣%ld\n", ARR_LEN(tests));
#define BENCH_ITERATIONS 30

double times[ARR_LEN(tests)][BENCH_ITERATIONS];


for (int i = 0; i<ARR_LEN(tests); i++) {
    for (int j = 0; j<BENCH_ITERATIONS; j++) {
        TRACE("call␣bench␣function");
        (*(tests[i].bench))();
        double elapsed = bench_timer_elapsed();
        times[i][j] = elapsed;
        printf("Bench␣\"%s\",␣iteration␣%d␣took␣%f␣
           seconds\n", tests[i].name, j,  elapsed);
    }
}
printf("Results␣%d␣iterations:\n", BENCH_ITERATIONS);
printf("%20s%12s%20s\n","Name", "average", "standard␣
   deviation");
for (int i = 0; i<ARR_LEN(tests); i++) {
    double sum = 0;
    double sum2 = 0;
    for (int j = 0; j<BENCH_ITERATIONS; j++) {
        double time = times[i][j];
        sum += time;
        sum2 += time*time;
    }
    double average = sum/BENCH_ITERATIONS;
    double variance = (BENCH_ITERATIONS * sum2 - sum *
       sum) / (BENCH_ITERATIONS * BENCH_ITERATIONS);
    double sd = sqrt(variance);
```

```
        printf("%20s%12f%20f\n", tests[i].name, average, sd)
           ;
    }
    char * csv_filename = "bench_times.csv";
    printf("(over)writing␣results␣to␣%s\n", csv_filename);

    FILE* fp = fopen(csv_filename, "w+");

    for (int i = 0; i<ARR_LEN(tests); i++) {
        fprintf(fp, "%s,", tests[i].name);
    }
    fseek(fp, -1, SEEK_CUR);
    fprintf(fp, "\n");


    for (int j = 0; j<BENCH_ITERATIONS; j++) {
        for (int i = 0; i<ARR_LEN(tests); i++) {
            fprintf(fp, "%f,", times[i][j]);
        }
        fseek(fp, -1, SEEK_CUR);
        fprintf(fp, "\n");
    }
    fclose(fp);

    return 0;
}
```

Listing C.1: main.c

Appendix D

Application makefile

```
all: main main_arm main_arm_debug main_arm_debug_san

main: main.c makefile
	gcc \
		-O3\
		-Wall\
		-I../install/include main.c\
		-L../install/lib\
		-l:libfmilib_shared.so\
		-o main\
		-lm\
		-pthread

main_arm: main.c makefile
	aarch64-linux-gnu-gcc\
		-O3\
		-Wall\
		-I../build-fmil-arm/aarch64-staging/include
		   main.c\
		-L../build-fmil-arm/aarch64-staging/lib\
		-l:libfmilib_shared.so\
		-o main_arm\
		-lm\
		-pthread

main_arm_debug: main.c makefile
	aarch64-linux-gnu-gcc\
		-ggdb3\
		-Wall\
                    -I../build-fmil-arm/aarch64-staging/include
                       main.c\
                    -L../build-fmil-arm/aarch64-staging/lib\
                    -l:libfmilib_shared.so\
                    -o main_arm_debug\
                    -lm\
                    -pthread

main_arm_debug_san: main.c makefile
          aarch64-linux-gnu-gcc\
                    -ggdb3\
                    -Wall\
                    -I../build-fmil-arm/aarch64-staging/include
                       main.c\
                    -L../build-fmil-arm/aarch64-staging/lib\
                    -l:libfmilib_shared.so\
                    -o main_arm_debug_san\
                    -lm\
                    -pthread\
                    -fsanitize=address\
                    -static-libasan
```

Listing D.1: main_makefile

Appendix E

FMIL `aarch64` build script

```
#!/bin/sh
export CC='aarch64-linux-gnu-gcc'
export CXX='aarch64-linux-gnu-g++'
export CMAKE_C_COMPILER='aarch64-linux-gnu-gcc'
export CMAKE_CXX_COMPILER='aarch64-linux-gnu-g++'

echo "CC:␣$CC"
echo "CXX:␣$CXX"
echo "CMAKE_C_COMPILER:␣$CMAKE_C_COMPILER"
echo "CMAKE_CXX_COMPILER:␣$CMAKE_CXX_COMPILER"

mkdir build-fmil-arm && cd build-fmil-arm
#cmake -DFMILIB_INSTALL_PREFIX="../install-arm" -
   DCMAKE_TOOLCHAIN_FILE="../build_arm.cmake" ../fmi-library
   / -DFMILIB_BUILD_TESTS="OFF" -DFMILIB_FMI_PLATFORM="
   aarch64-linux-gnu"
cmake -DFMILIB_INSTALL_PREFIX="../install-arm" -
   DCMAKE_TOOLCHAIN_FILE="../build_arm.cmake" ../fmi-library
   / -DFMILIB_BUILD_TESTS="OFF" -DFMILIB_FMI_PLATFORM="
   linux64"
make install

echo "␣"
echo "If␣the␣output␣did␣not␣contain␣any␣errors,␣fmil␣was␣
   compiled␣for␣arm␣correctly"
```

Listing E.1: build_fmi_library_arm.sh

Appendix F

FMIL `aarch64` `cmake` toolchain file

```
# the name of the target operating system
set(CMAKE_SYSTEM_NAME Linux) # Petalinux is using the Linux
   kernel

# set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR aarch64)

#set(CMAKE_SYSROOT ???)
set(CMAKE_FIND_ROOT_PATH ${CMAKE_SYSROOT})
set(CMAKE_LIBRARY_ARCHITECTURE aarch64-linux-gnu)
set(CMAKE_STAGING_PREFIX aarch64-staging)

set(GCC_TRIPPLE "aarch64-linux-gnu")

# which compilers to use for C and C++
set(CMAKE_C_COMPILER aarch64-linux-gnu-gcc)
set(CMAKE_CXX_COMPILER aarch64-linux-gnu-g++)
#set(CMAKE_CXX_FLAGS "-m64")

set(ARCH_FLAGS "-mcpu=cortex-a53") # add more specific flags
    for ideal performance
set(CMAKE_C_FLAGS_INIT ${ARCH_FLAGS})
set(CMAKE_CXX_FLAGS_INIT ${ARCH_FLAGS})
set(CMAKE_Fortran_FLAGS_INIT ${ARCH_FLAGS})

set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)


# prevent cmake from making a test binary
# since it obviusly can't run it when cross compiling
set(CMAKE_C_COMPILER_WORKS 1)
set(CMAKE_CXX_COMPILER_WORKS 1)
```

Listing F.1: build_arm.cmake

Appendix G

Petalinux dependencies

```
sudo apt-get install gawk python build-essential gcc git
     make net-tools libncurses5-dev tftpd zlib1g-dev
    libssl-dev flex bison libselinux1 gnupg wget diffstat
     chrpath socat xterm autoconf libtool tar unzip
    texinfo zlib1g-dev gcc-multilib build-essential-dev
    zlib1g:i386 screen pax gzip

sudo dpkg-reconfigure dash
```

Appendix H

Statically linking FMU

Integrating the FMU and FMIL with the application would be desirable for performance and distribution reasons. While it would be technically possible, it was judged would not be worth the time investment. This section only exists to document the attempt.

In FMIL on Linux, the Portable Operating System Interface (POSIX) APIs defined in `dlfcn.h` are used [15]. For portability they are abstracted away in the "jm" part of FMIL.

```
grep -r dlopen .
grep -r dlsym .
```

Listing H.1: Search for dlopen and dlsym.

`dlopen()` loads a dynamic library file and it's result used by `dlsym()` to obtain a function pointer to a library function at runtime [15]. These are abstracted away in jm_portability.c:

```
jm_portability_load_dll_handle_with_flag (abstracts dlopen)
jm_portability_load_dll_function (abstracts dlsym)
```

Listing H.2: jm_portability function names.

With some small modifications, linking a FMU here may be possible by overriding this function. A Look Up Table (LUT) can be created from a list of FMI api functions. This does not solve the issue of XML parsing, but is a start. The library loads DLL functions using the macro `LOAD_DLL_FUNCTION` in `fmi2_capi.c` in the function `fmi2_capi_load_common_fcn` Model Exchange (ME)/Co-Simulation (CS) is loaded conditionally based on the capabilities. In order to statically link FMU into single binary, we could have used FMIL to parse and then generate a header file, which is then used to compile the entire thing again statically. Alternatively, the FMU could be compiled directly into the application, but then raw FMI functions would have to be used.