Faster dynamically instrumented programs: A look at floating-point emulation in ARM Linux

Bachelor of Science (Hons) in Computer Science

6CCS3PRJ

Final Project Report

Author: Matthew Beardwell
Student ID: 1681382
Supervisor: Dr. Stephen Kell

April 2023

Abstract

This project borrows existing dynamic program instrumentation techniques to propose a faster method of emulating floating-point instructions on Unix-like operating systems than what is provided by the kernel. The proposed method replaces floating-point instructions with branches that indirectly lead to emulation code resident in the same process' memory. This prevents some execution flow switching into kernel code to run the kernel's floating-point instruction emulator which theoretically reduces overhead for every instruction emulated.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

Matthew Beardwell
26 April 2023

Acknowledgements

Credit to many helpful workarounds and better ideas go to my supervisor Dr. Stephen Kell who frequently made the impossible possible. Thank you.

Contents

1 Introduction **4**

2 Background **7**
2.1 Motivation 7
2.2 Instrumentation and Emulation 8
2.3 Instrumentation Approaches 9

3 Literature Review **10**
3.1 A Trap-based Approach 10
3.1.1 Instruction emulation in the kernel 10
3.1.2 Instruction emulation in userspace 11
3.1.3 How a trap-based technique can be improved on . . . 11
3.2 A Jump-based Approach 13

4 Requirements and Specification **16**
4.1 Functional Requirements 16
4.2 Non-Functional Requirements 17
4.3 Specification 17

5 Implementation **19**
5.1 Development Environment 19
5.2 Search 20
5.3 Instrumentation 21
5.4 Emulation 24
5.5 Limitations 26
5.5.1 Trampoline placement 26
5.5.2 Thumb mode 27
5.5.3 Inflexible loading 27

6 Evaluation **28**
6.1 Performance Comparison 28
6.2 Requirements Evaluation 30

7 Conclusion and Future Work **32**

Chapter 1

Introduction

Operating systems can often optionally support the running of programs that shouldn't technically be able to run on your computer. These programs may contain code that was compiled for other similar hardware and where the differences lie, the kernel picks up the slack. One place this difference may occur is the presence or lack thereof of a Floating Point Unit (FPU). Computer processors will usually have an FPU to perform floating-point arithmetic which can accelerate the performance of programs requiring many fractional calculations. Without the FPU, a compiler generates machine code that performs the same mathematical calculations, but using only integer-arithmetic instructions which, on the whole, take longer to do the same thing as more CPU cycles is needed.

Although packaging a processor with floating-point acceleration may result in more performant programs when the compiler produces code that uses the FPU, it increases power consumption and the size of the design [11]. This is why some manufacturers opt to leave out the additional component.

Sometimes the operating system supports the emulation of floating-point instructions instead of stopping the process when one of them is encountered. This more flexible approach let's slightly incompatible programs and hardware work together. The way the kernel can emulate this is by catching, or 'trapping', exceptions raised by the processor when the unknown instruction is encountered and performing the calculation in software.

As discussed in further sections, trapping is slow. A better approach is to never send the instruction to the processor in the first place so the kernel doesn't become involved. You can hopefully decide beforehand where the instructions are and what needs emulating and then run the program, where it jumps into the binary packaged with it to perform the emulation. Redirecting execution in this way doesn't just permit single instruction emulation. Forms of userspace execution flow redirection like this support a wide range of use cases including program instrumentation such as memory usage profiling or counting method calls and other emulation techniques like

QEMU's userspace architecture translation emulator [8].

The aim of this project is to realise faster emulation of floating-point instructions on ARM machines running Linux. The original program is ran on the CPU and when an unsupported instruction is encountered, the kernel emulates what it would have done if the hardware supported it. A faster way would be to never execute these unsupported instructions by removing them from the program as it is loaded into memory and replacing them with branch instructions to an area of memory which contains the emulation code. Dynamically patching the program code at run-time initially slows the process, but it means that the program doesn't have to be recompiled.

My solution selectively scans and decodes process memory and instruments the found floating-point instructions. When execution enters the instrumented program code, control flow redirects at the addresses of the replaced instructions with a branch off of a run-time generated trampoline to a routine that emulates the original instruction. This tool is packaged as a library intended to be 'preloaded' using a mechanism provided by the dynamic linker/loader. This means that the dynamic linker will execute this library code before the main program while they exist in the same virtual address space, thus permitting the tool to instrument the program before it is ran.

I managed a working proof-of-concept for the proposed new approach, but due to the scale of work needed, a satisfying, full working solution wasn't achievable in the permitted time frame. So far, support for a single instruction was implemented - 'vadd.f32' - however I'd estimate what I have achieved represents the majority of what the final product should look like without including any support for the ARM Thumb-1 and Thumb-2 extensions. I did face some limitations which I discuss throughout and in 5.5.1. For example, trampoline placement could be impossible in albeit rare circumstances. I also made assumptions about which regions of memory do not contain floating-point instructions which could be worth rethinking whether they are valid.

This report has six further chapters. In the background section, I will talk about why instrumentation is needed in my project and also about two types of binary instrumentation: trapping and jump-based instrumentation. Trapping being the old approach and jump-based instrumentation hopefully being an improvement. In the literature review, I relate to existing methods that attempt machine-code level patching. ARMPatch [13] edited stored binary files to replace code segments on ARM, LiteInst [10] addresses some problems found in jump-based probe insertion, and Kessler's 'Fast breakpoints' [14] explains thoroughly how jump-based probe insertion works. There is subsequently a chapter on the functional and non-functional requirements for the project as well as its specification. Then in the chapter on implementation, I walk through precisely how I programmed the solution and discuss some of the design decisions I made and problems I faced.

The performance of the proposed method is evaluated in a dedicated chapter then in the conclusion I review the limitations of the project and what achievements were made. I talk about how future works could improve on my approach.

Chapter 2

Background

In this chapter, the foundation is first laid out by motivating the need for instruction emulation and then secondly describing emulation and how it is achieved.

2.1 Motivation

Many computers such as smartphones and smart watches need to be conservative with the energy they use. Computer chips with a small instruction set provide "high performance per watt for battery operated devices" [18] meaning that for the same performance, they consume less power. These 'RISC' chips are cheap and low power and may not support such a wide range of instructions. And even then, a CPU might only implement a portion of the instruction set.

Take ARMv7 for example. This is an instruction set architecture (ISA) that allows for an extension to support floating-point operations. The 'Vector Floating Point' (VFP) extension is often optional for some older and lower-power ARM CPUs. When a CPU doesn't implement these floating-point operations [17], programs compiled to be used with the ARMv7 architecture that contain those VFP instructions do not work.

The way the operating system can handle this is by emulating what those instructions would've done had the CPU been built to support the VFP extension. The floating-point arithmetic is done on the integer-only hardware through software floating-point emulation routines in the kernel.

Unfortunately, this is quite slow. In ways that I'll explain in the next chapter, a method that modifies the program to use code that is packaged with it to perform the emulation is much faster than relying on the kernel.

2.2 Instrumentation and Emulation

I will use Laurenzano et al.'s [15] definition for instrumentation: "Binary instrumentation facilitates the insertion of additional code into an executable in order to observe or modify the executable's behavior". Apart from timing differences, changing the program code doesn't necessarily mean changing the observable behaviour of its process. To count the frequency of execution of a code segment or the execution time, instrumentation tools might interrupt control flow at run-time by inserting specific instructions beforehand at a desired location. Some instrumentation routine would then be executed and control returned. Dynamic instrumentation frameworks like Valgrind [19] and DynamoRIO [9] are the basis of many profiling tools that, for example, analyse memory usage and code coverage or list library and system calls.

Laurenzano et al. go on to say that this instrumentation can be static or dynamic. Static being modifications made on disk and dynamic, those made in memory. The distinction between static and dynamic instrumentation is more specifically the distinction between modifying a stored program prior to running it and modifying the program after it has been loaded into the virtual address space of a process. The former permits redistribution of the modified file and the modification only needs to be made once. Dynamic instrumentation means performing the changes at run-time, potentially introducing a performance overhead, but allows better control over the programs run-time environment.

When modifying the behaviour, various mechanisms can be relied on. Inter-process communication via an operating system signaling mechanism allows for control flow redirection when a special instruction is encountered in the program. This relies on the CPU making an exception and executing the part of the operating system that handles that, subsequently executing code segments in memory that were intended to be reached from the point of instrumentation. Trapping exceptions is expensive but simpler to write programs with because it is a facility already managed by the operating system. Inserting branch instructions to redirect control flow is more complicated to do correctly but comes with the advantage of speed by foregoing generating exceptions and moving execution into the kernel.

Emulating parts of a program has a wide range of uses. The idea is fundamentally quite simple - you want to see what it does but you may not be willing or able to directly run it on the hardware. QEMU [8], a system emulator, converts instructions in chunks to a target architecture so that they can be emulated. QEMU facilitates running of non-native binaries or even emulating entire systems. DynamoRIO is a dynamic instrumentation framework that also emulates instructions by dissecting and changing code segments to maintain control over the flow of execution. Fundamentally, DynamoRIO intercepts and emulates branch instructions [12] in the instru-

mented program.

There are a wide range of uses for emulating one or many instructions. This report is about redirecting control flow by patching a program in memory such that a subset of the instruction set is emulated. The goal is simply faster execution than the existing approach.

2.3 Instrumentation Approaches

One way to redirect control flow is to just change the source code and recompile it. In interpreted languages for example, it's even easier as no recompilation is needed. Unfortunately, you may not have access to the original source code or you might be unprepared or unable to recompile it. Dynamic binary instrumentation tools let you pick at random the program that you would like to instrument without knowing what it's going to be beforehand.

Trap-based instrumentation seeks to insert trapping instructions. These instructions are defined to halt the CPU and start running a handler. By registering a custom handler, execution flow will always redirect to user-supplied code at any point in the binary executable where these special instructions have been placed. The advantage of this approach is that it is very easy to implement in comparison to other methods.

Jump-based instrumentation increases the complexity of the instrumentation framework significantly from the trap-based methods, but it can be faster. It's a little similar in that instructions will be inserted, but instead of a generic trapping instruction, a specially crafted branch instruction is placed. This means that the mechanism that redirects the execution flow stays in-process without relying on those CPU or kernel facilities required by trapping. This is what makes it faster. The complexity involved with a jump-based approach comes from where the branch actually jumps to. With trapping, by registering a handler with the kernel, the location of the instrumentation code will be calculated automatically with no programmer intervention. By moving the instrumentation code into the process memory, the instrumentation framework has to search and manage locations for this code to go at run-time. Beyond this and depending on the instruction set, it can be difficult or even impossible to fit the branch instruction at the probing site.

Hopefully, I have outlined the core concepts to lay the foundation for my approach. In the rest of the report, I will look at related works then go further in-depth about the specifics of the design environment and considerations.

Chapter 3

Literature Review

This project draws considerable inspiration from Kessler's [14] paper "Fast Breakpoints: Design and Implementation". Their intention is to write a facility for inserting breakpoints which means that the displaced instruction is eventually sent to the CPU. Slightly tweaking their method to remove this makes it very close to my approach at a high level. Kessler says that the goal is to improve on a breakpoint mechanism provided by the Unix kernel. I am working on Linux, a Unix-like operating system, and am similarly hoping to avoid kernel hand-offs.

Huang and Song's "ARMPatch: A Binary Patching Framework for ARM-based IoT Devices" [13] has goals further than instrumentation. They introduce a framework to replace contiguous code segments in a binary executable, not just a single instruction at a time. I look to this paper for its architectural similarities. Since I am working on ARM hardware, the paper provides insights that other binary instrumentation papers do not.

Another paper on instrumentation is Chamith et al.'s paper "Instruction Punning: Lightweight Instrumentation for x86-64" [10]. They introduced a complex mechanism to counter problems with the faster jump-based instrumentation present on x86-64 architectures. These problems won't be present on ARM, but similar issues are.

In this literature review I will go further in-depth about some dynamic instrumentation approaches, trap-based instrumentation and jump-based instrumentation, compare their benefits and flaws, and illustrate with examples from the aforementioned literature.

3.1 A Trap-based Approach

3.1.1 Instruction emulation in the kernel

If an unsupported instruction is given to the processor, the operating system can handle some type of 'unsupported instruction' exception by emulating

what the instruction would've done and giving the result to the responsible process.

This is sometimes what Linux does, for example. In my case, where I focus on Linux on ARM, the kernel sometimes contains emulation code for floating-point instructions. Some ARM chips support floating-point instructions and some do not. This is a matter of whether they have a Floating Point Unit (FPU) coprocessor. The FPU extends the CPU's capabilities to support floating-point instructions. If the ARM chip lacks this FPU, and so doesn't support the floating-point operations, it will give an illegal instruction exception and the Linux kernel can trap the exception to see what the responsible instruction was. If it was indeed a floating-point exception, the kernel sends it off to its own internal emulation library. For the ARM chips, Linux can contain various floating-point emulators. For older chips, the NetWinder Floating Point Emulator or sometimes the Fast Floating Point Emulator is used. On newer VFP CPUs, support for VFP emulation exists and is enabled through some kernel build configuration options.

Another use for this mechanism is debugging. Debuggers can deliberately place trapping instructions into programs, like INT3 or UD2 on x86, that are designed to always throw an exception. Because the debugger has volunteered itself to handle the 'illegal instruction' signal from the kernel, called 'SIGILL' in Unix-like operating systems such as Linux, the kernel runs the signal handler routine provided by the debugger. The debugger can then take its leisurely time at this breakpoint before emulating the instruction and returning control back to the kernel (which then returns back to the original program).

The problem with letting the kernel emulate unsupported instructions is that it is slow. There is substantial overhead from executing the instruction in userspace then trapping the exception and emulating it in the kernel to then return back to userspace again.

3.1.2 Instruction emulation in userspace

In cases where you want to handle the trap yourself, an application can request the kernel to allow it to handle the illegal instruction signal. The emulation in userspace with this trapping technique acquires even more overhead than the kernel-space emulation. Userspace to kernel, back again, back to kernel, then back once more.

3.1.3 How a trap-based technique can be improved on

Chamith et al. [10] warn that trap-based probes cause a "substantial slowdown ... each probe invocation incurs an interrupt and associated user/kernel-space transition." The need for a faster approach motivates this project.

Jump-based probes can be injected so as to not require a user/kernel

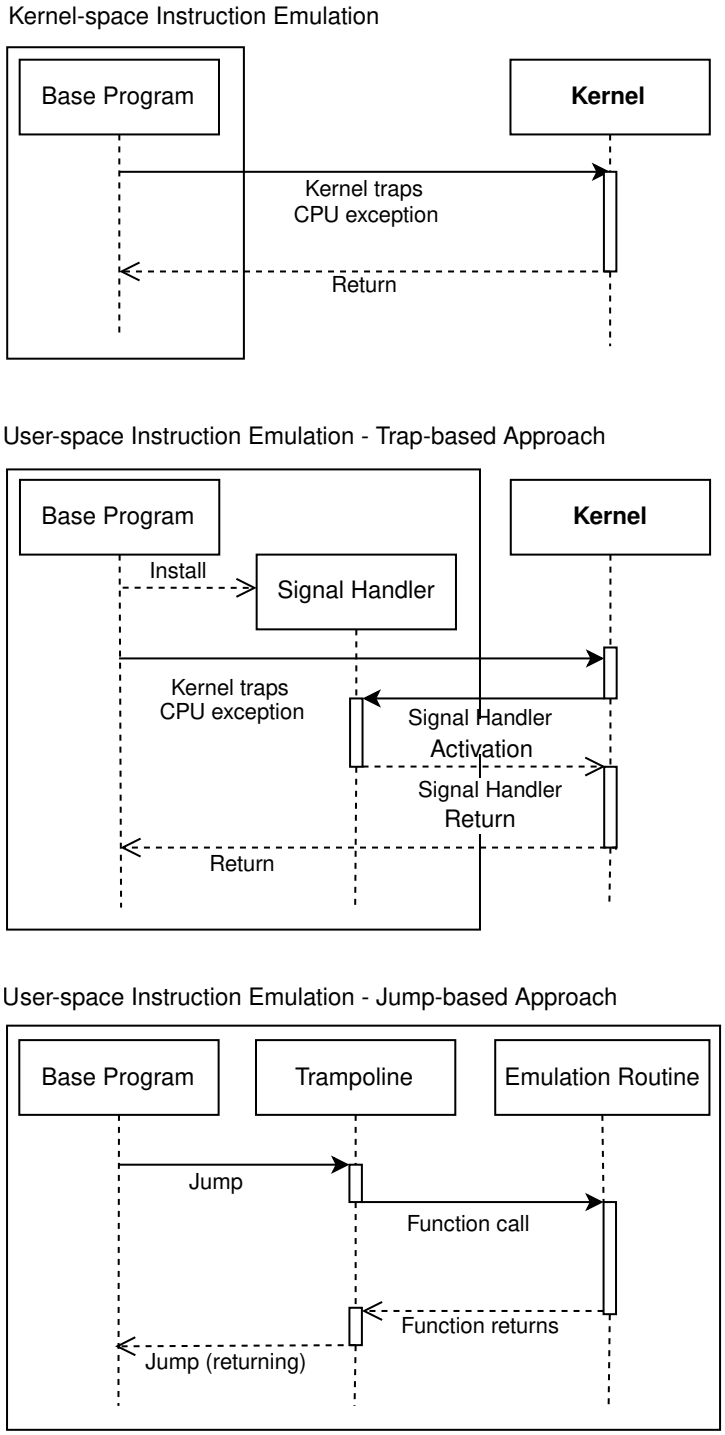

Figure 3.1: Comparing trap-based and jump-based instruction emulation. The box represents what execution is done in-process.

hand-off. Instead of waiting for the kernel to call into a signal handler, the need for any userspace to kernel transition can be foregone by not executing the instruction in the first place. Before execution, you can replace all the unsupported instructions in the binary with branches to a userspace emulation routine. The emulator is then packaged in the binary or, if this is done on-the-fly, memory can be allocated in the process's address space and the emulation code written there. The branches would then point to that memory region. Figure 3.1 illustrates different instrumentation methods and how they affect the process. As more of the execution is done in-process, kernel/userspace switching overhead is eliminated.

To follow calling conventions, a function call would need to take up multiple instructions worth of space. For example, setting up the stack and registers for passing arguments. You also still have to execute the instruction you replaced if the approach requires it. Chamith et al. [10] describe this process saying there "must [be] a jump target containing a run-time-generated trampoline, where the trampoline assumes responsibility for executing the displaced instructions ... as well as setting up a full function call to the function attached to the probe" [10]. The function call in this case being to the floating-point arithmetic emulation routine. Executing the full trampoline and the emulation routine instead of the single original instruction has a seemingly sizeable overhead but in cases where not very many instructions are replaced, the performance difference would likely be negligible. Clearly emulation is slower than executing the original instructions so perhaps it is more useful to compare types of emulation. Jump-based probe injection's relatively "low invocation overhead" [10] from bypassing the kernel makes it ideal for this project.

3.2 A Jump-based Approach

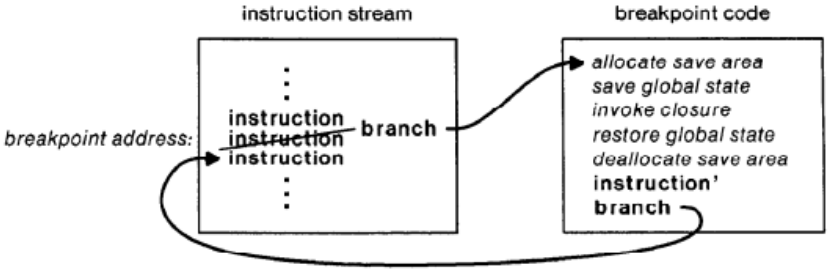

Figure 3.2: Diagram taken from Kessler's paper [14] illustrating how a breakpoint is inserted in their method.

Kessler [14] writes about a breakpoint utility that injects a function call at a given point in an instruction stream. As shown in figure 3.2, they adopt

jump-based instrumentation to insert breakpoints into the binary.

Because of the varying instruction lengths in some architectures, it may not be so easy just to place a branch at any point in the instruction stream. Kessler [14] didn't encounter this problem as "on the SPARC, all instructions are word-sized, [so] we never have any problem with branch instructions being larger than the instruction they are replacing".

The same cannot be said for x86-64, for example. Chamith et al. [10] introduce a technique they call "instruction punning" to dynamically insert instructions into x86-64 binaries on Linux. They point out that x86-64's five-byte jump instructions can only replace instructions that are also five bytes. If the displaced "instruction is smaller than the jump this technique will overwrite adjacent instructions" as well. Instead of overwriting the following instructions, they are instead interpreted as part of the address of the branch and the trampoline is placed wherever this pointer lands. In the case of ARM however, usually the instructions are all 4 bytes wide. This does place a limit on the distance a branch can go using immediate values but it gives more flexibility to where probes and trampolines can be placed. All probes will fit at any probe site because all instructions are the same width.

Since I'm focused on ARM's fixed four-byte wide instructions [16], the injection of breakpoints, or trampolines in the case of this project, will not need complex workarounds like instruction 'punning' and thus luckily meet the same lack of restriction as Kessler's SPARC breakpoints. Therefore it could be considerably easier to implement a jump-based probe injection approach like Kessler's. One key thing to note is that although the four-byte probe site instruction can always be replaced by another on ARM, the probe is still limited to whatever address can fit in the three-byte operand which limits the number of places the probe can branch to.

Huang and Song's [13] binary-patching tool makes changes to stored ARM ELF binary files to patch security flaws. They make changes to, or replace, entire functions. This is simpler to do in comparison to the single instruction replacement previously discussed. Patching regions of code means that a single branch instruction can almost always be placed in that region that points to code tagged onto the end of the binary file. I omit the practical details of adding code to the end of the file for simplicity. Namely, the updating of the file's metadata, i.e. the 'ELF header' in this case. When replacing a function entirely, they do consider inserting this branch like the previous two techniques and placing the new function elsewhere but "a lot of space is wasted if the function being replaced is large". The single branch would be put at the beginning of the patched function making the rest of the function unreachable, dead code and unnecessarily contributing to the size of the executable. For this reason, they only used this technique when it was necessary - the new function was larger than the one it was replacing.

Jump-based instrumentation does have the benefit of faster program ex-

ecution in comparison to trapping but it comes at the cost of higher memory usage and slower program start-up. You also have the additional complexity of the implementation. Finding appropriate locations for the trampolines and inserting the jumps present some difficulties. Chamith et al. [10] had to introduce a rather complex method to insert a jump instruction into small probing locations and on ARM, this project will encounter difficulties finding locations for trampolines because ARM instructions have a limited instruction width thus a limited range of offsets to branch to.

Chapter 4

Requirements and Specification

The goal of this project is to create a program designed to be loaded before a target ARMv7 ELF. It will edit the binary in memory to replace floating-point instructions with a branch to an emulation routine that performs those operations using only integer-arithmetic instructions. The altered binary should behave in the same way as the original unaltered binary if the unaltered one were ran on an ARMv7 processor that did indeed support the floating-point instructions.

The design of the program relies on the LD_PRELOAD mechanism of the dynamic linker. The dynamic linker is what loads a program into memory and runs it. The program itself will specify a specific dynamic linker such that when the operating system is asked to run the program, it looks for that dynamic linker, loads it into memory and runs the linker. The linker can do everything that a linker does - loading dependencies, making relocations, etc. - then it will execute the program. The environment variable LD_PRELOAD is read by the dynamic linker and used to determine what libraries to load before any else [6]. This can be taken advantage of to load my library before the main program and thus make the necessary changes to it while it's in memory.

Here I list the necessary functionalities the program should have and also some desirable qualities.

4.1 Functional Requirements

- The program can be injected into a binary using the LD_PRELOAD mechanism on Linux.
- The main program binary is patched in memory then run.

- The program can locate and replace any reasonably located floating-point instruction in memory. For example, it won't catch unpacked instructions or those generated at run-time.
- The program discriminates floating-point instructions and other instructions or data.
- The program replaces each identified floating-point instruction with a jump to a trampoline which calls its corresponding floating-point arithmetic emulation routine.
- The trampoline and emulation routine are are written to memory and branches are written to encode the address of trampolines and emulation routines at run-time.

4.2 Non-Functional Requirements

- Must be faster than an equivalent binary relying on traps to kernel floating-point emulation over a reasonable sufficiently-long run-time.
- The increased memory usage of the process must not exceed twice that which would occur with the original program.
- The functionality of the binary remains the same after patching.
- The program is reasonably compatible with any arbitrary ARMv7 binary on Linux.

4.3 Specification

Language The program is written in C.

Execution mechanism The program is injected through the LD_PRELOAD Linux environment variable.

Instruction replacement All floating-point instructions are replaced in the binary file with branch instructions.

Jump to emulator The branch instructions will jump to a trampoline and then to a floating-point arithmetic emulation routine.

Emulation method The floating-point arithmetic routine will carry out the original floating-point instruction using the integer-arithmetic ARM hardware.

Return from emulator Execution in the process after the routine call continues after where the floating-point instruction had been replaced.

No behaviour change The effect of the emulation routine call will be identical to the intention of the original floating-point instruction (e.g. the registers and stack will look as though the original floating-point instruction had been executed).

Chapter 5

Implementation

5.1 Development Environment

The most time-consuming difficulty I had throughout this project was understanding what the project was about. From the specification, I concluded that it was about how some ARM chips don't have floating-point coprocessors and how the operating system can have a mechanism to emulate floating-point instructions if they are encountered. My task was to find and build a hardware emulator and to find and build an operating system that matched this description.

I found that there were only a small number of ARM chips that lacked an FPU, but that also had a memory management unit (MMU) that was required for modern operating systems. 'QEMU' [8] is a popular full-system emulator. It has an emulator for some of the Versatile Express development boards, one of which uses ARM Cortex A9 CPUs in its System on Chip (SoC). The ARM Cortex A9 processor has an MMU and can optionally include a VFPv3 FPU. Unfortunately, there is no way to ask QEMU to emulate a version that doesn't have the FPU, so I had to slightly modify the QEMU source code and rebuild it. I then had an emulator for a system that could run a modern operating system which had a CPU with no floating-point hardware.

Linux has a distribution called Debian. "Debian is an operating system (OS), not a kernel (actually, it is more than an OS since it includes thousands of application programs)" [3]. This essentially means that Debian is a set of software packages provided with the Linux kernel. Debian has various ports to ARM, but the ones relevant to these older chips would be its Arm EABI port and newer Arm hard-float EABI port [2].

The difference being that the older port was built with a version of the application binary interface (ABI) that didn't rely on hardware floating-point support. The ABI doesn't just govern routine calling conventions, but a wide range of rules that dictate how programs should interact. The ABI,

called ‘EABI’ in this case, must be the non-floating-point variant simply because there is no floating-point hardware.

Programs can be compiled to use this non-float ABI and still contain VFP instructions. When compiling my test program for the project, I used the ‘-mfloat-abi=softfp’ compiler flag. This ensures that the VFP instructions are compiled into the binary file, but that also the binary has the non-float ABI. GCC documents that “‘softfp’ allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions” [1].

Although I expected it to be relatively simple setting up an emulator, operating system, and the relevant test program compiler flags, the difficulty emulating the niche hardware and not having a strong understanding of floating-point ABIs meant that I had to enumerate very many configurations to find a set that were compatible and functional. At one point, I compiled and tested 16 different Linux kernels on four emulated hardware configurations each to find out what types of compatibilities and incompatibilities existed taking days and reinforcing my understanding of ABIs. I then sought out support from a Debian online messaging group which members relatively authoritatively confirmed that my configuration was correct.

5.2 Search

The first difficult component is the instruction search; where are the instructions in memory? When a program is ran on Unix-like systems, the dynamic loader maps some of the sections of the ELF executable and its dynamically linked shared libraries into memory. I learned that one mechanism of discovering where in the process’ virtual address space those sections have been mapped to is reading the ‘/proc/self/maps’ file which is provided by the kernel. Each entry shows, for example, a range of addresses, the permissions of that memory region, and sometimes the path of the file that the memory was mapped from. This is all I knew at the beginning. Deciphering how to use this information to find the instructions was more or less educated guesswork. I can conclude that any entry that doesn’t have an executable permissions flag can safely be ignored because it is likely that these memory-mapped regions will not contain instructions that will be ran. Any region that relates to the operating system such as kernel-shared memory can also probably be ignored as it is unlikely that they will contain floating-point instructions. Beyond this, it becomes trickier and sometimes impossible to be sure about the location of all relevant instructions. Run-time-generated code can’t be instrumented in this way, for example, especially when there are many programs that compress their contents and unpack them at runtime. This is a problem that matters less in my case, since the goal of the method is to speed up the existing approach, the long-term efficiency of the

instrumented binary is a monotonic function of the number of floating-point instructions successfully instrumented. In other words: I can afford to miss out a few.

Using figure 5.1 as an example, various parts of the original binary file are loaded into each memory segment which makes it a little more complicated. The 'PT_LOAD' parts refer to each part of the file that is specified to be loaded into memory by the dynamic linker. The 'ELF Header' is packaged into the same segment as the first executable section, in many cases the program code. This means that when trawling through executable memory regions, I need to be careful to step over any unrelated portions like the ELF header, program headers, and PT_INTERP (that specifies which dynamic linker to use). I discovered this independently while bug-fixing by noticing that some of the code segments I was instrumenting started with the first few bytes of an ELF header. This wasn't too complex to work around as the header says where it (and other) headers end, so my code simply jumps to that offset.

I did face some programming challenges while writing this as when working with memory I had to be careful that the addresses I used were aligned with a page boundary when necessary and find the sections within the memory-mapped segment ranges.

The 'maps' file is parsed through library routines, but I did initially find the data structures overwhelming. As with most other things in this project, I spent much more time reading documentation and source code than programming.

After identifying likely instrumentation sites, a disassembler like 'libopcodes' [4] or the Capstone Engine [5] can be used to work out what each instruction does and so determine whether they are in the VFP ISA extension. These unsupported instructions can then be replaced with jumps to a trampoline. I chose the security-focused Capstone Engine because of its simple programming interface, but there was little to no documentation that I could find which meant I could only discover the interface by reading the source code. In the next section, I will talk about my implementation of the instrumentation phase.

5.3 Instrumentation

I have said that the instrumented sites will jump to a trampoline. Due to the instruction width constraint, the trampolines need to be placed reasonably close to the instrumented program in memory. At the moment, I deal with this by searching for unclaimed memory in the vicinity. I discuss this limitation further in subsection 5.5.1.

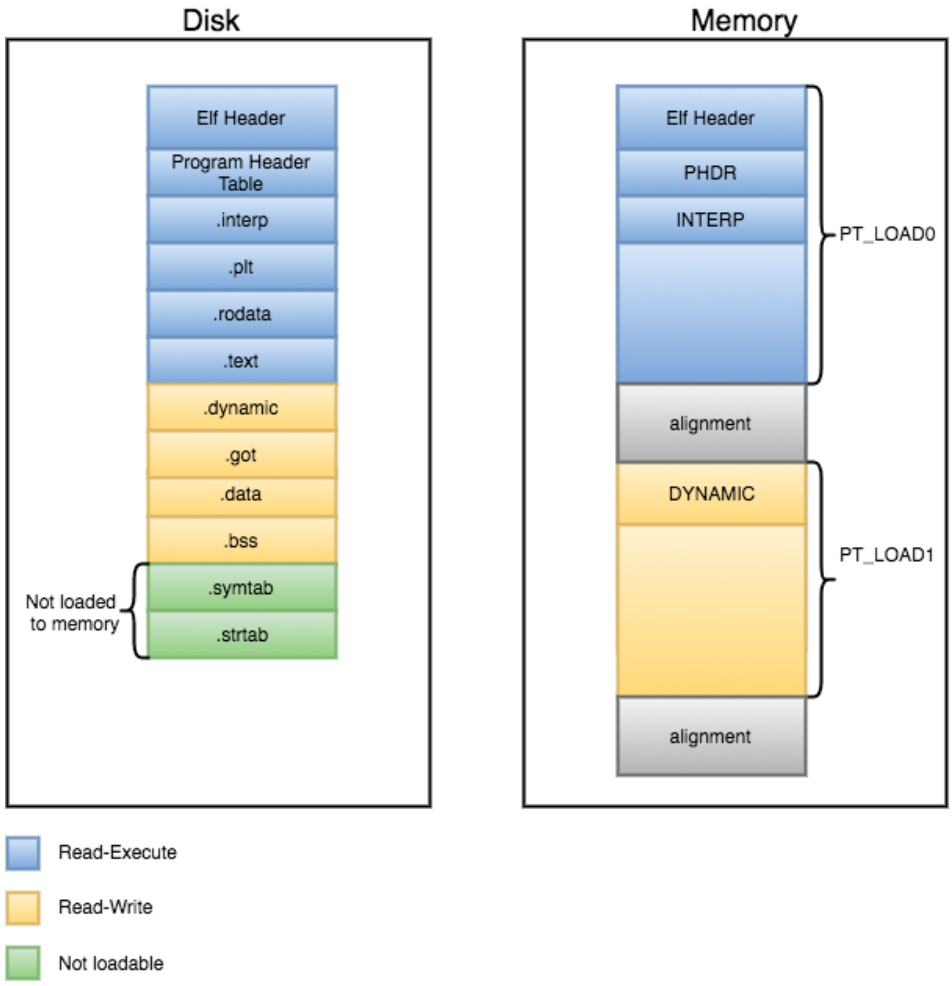

Figure 5.1: An example by Sanmillan [20] of how an ELF executable could be loaded into memory.

Onto designing the trampoline, the trampoline serves one function only - to call a method. The floating-point hardware emulator may trash registers that were important to the running of the program before it hit the probe. Therefore, it is necessary to save the state of the registers onto the stack and restore them after the function call. The assembly layout of the trampoline in memory is then:

- push {<most registers>}
- branch to emulator
- pop {<most registers>}
- branch back to probe site

This should look similar to Kessler's diagram in figure 3.2.

The trampoline exists only as a template in the C code and the addresses of the emulation routines are calculated at run-time and written into the template. A different trampoline is copied into memory for each floating-point instruction type which will have the address of the relevant emulation routine inserted in.

```
int8_t template[] = {
        0xFF, 0x5F, 0x2D, 0xE9, // push {r0-r12, r14}
        0xAD, 0x5E, 0x0D, 0xE3, // movw r5, #0xdead
        0x05, 0x58, 0xA0, 0xE1, // lsl r5, r5, #16
        0xEF, 0x6E, 0x0B, 0xE3, // movw r6, #0xbeef
        0x06, 0x50, 0x85, 0xE1, // orr r5, r5, r6
        0x00, 0x00, 0xA0, 0xE3, // mov r0, #0
        0x00, 0x10, 0xA0, 0xE3, // mov r1, #0
        0x00, 0x20, 0xA0, 0xE3, // mov r2, #0
        0x00, 0x30, 0xA0, 0xE3, // mov r3, #0
        0x35, 0xFF, 0x2F, 0xE1, // blx r5
        0xFF, 0x5F, 0xBD, 0xE8, // pop {r0-r12, r14}
        0xFE, 0xFF, 0xFF, 0xEA  // b #0
};
```

Above is a bare-bones example of what the trampoline could look like. One problem I struggled with was working out how the trampoline code would

know at run-time which instruction it should emulate. Since the instruction was removed during instrumentation, the only way to know what to emulate is to store the trampoline-instruction mappings beforehand. This could've been done by using a look-up table which would've meant a speed cost as a trade-off for only having to have one trampoline. I decided on using a separate trampoline for every single instruction which performed fine with programs I tested it on, but may be impractical for programs with many floating-point operations due to the memory overhead. A more sophisticated approach in a future iteration is needed to reduce this overhead.

When the trampoline is generated, the address of the emulation function is hard-coded into it. Calling conventions dictate that you have to first pass the arguments in registers and on the stack and then branch. Since I decided not to use run-time lookup, I would need to write the address in a branch in the trampoline. Again due to the fixed instruction width, I initially couldn't work out how to do this. I realised that if I branched to an address that was in a register, I could load the register with this fairly long four-byte address before the branch in two sections. I would split it into two two-byte chunks and load them separately into each half of the four-byte register as immediate values.

The arguments loaded into the registers are zero for this template which can then be filled in when the trampolines are loaded into memory depending on what type of instruction is jumping to it. Then there's the jump with a 'blx <scratch register>'. The branch, in this case to address 0xDEADBEEF, will be a C function so it returns using the link register. This was set using the 'blx' instruction, overwriting the link register from just before execution entered the trampoline. This is why the push and pop are used to restore the state of the registers, and specifically the link register (r14) in this example.

The second branch, the branch that returns to the site of the displaced floating-point instruction, is calculated and written in the same way as the branch that points to the beginning of the trampoline except the offset is slightly different and has a inverted sign. An small offset instead of the actual address is necessary here, but for a different than the probe site branch. The probe site branch is limited by the number of offsets that can fit in a single branch instruction's three-byte operand. The trampoline's return branch is limited to using small offsets instead of fixed addresses because writing the full address into a register would mean overwriting registers with no way to restore them later.

5.4 Emulation

The floating-point emulator should, in practice, match the exact specification of the hardware that it is being emulated, implementing different VFP extensions. The emulator should also check if the hardware already has the

floating-point registers since they are shared with the Neon extension. Neon instructions may expect to retrieve the floating-point values from the shared registers. In the case that these registers are not found, the register values will be stored in memory. Simple enough in C - just create an array.

It is more complicated transferring values between emulator registers and general purpose registers. In the FPU coprocessor, this would be achieved with instructions like VMOV. To emulate the VMOV, this is better done by making a different template that *doesn't* restore all the clobbered registers. Instead the registers that were changing in the emulation routine would not be restored with their original values from the stack.

All the changes were made in the preloaded library ran by the dynamic linker. Now that the instrumentation has occurred, the dynamic linker would then run the instrumented original program that will branch to trampolines and then emulation routines at every probed location.

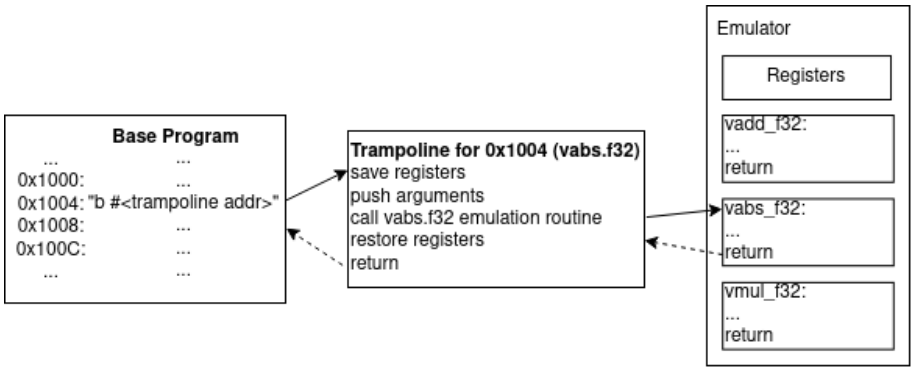

Figure 5.2: A rough idea of the memory layout and control flow at run-time. Each of the 'base program', 'trampoline', and 'emulator' boxes represent a contiguous region of memory.

Figure 5.2 shows what regions of memory in the process' virtual address space might look like. The dynamic loader has put library code and the main program at various addresses which are then patched with branch instructions before running. It is not necessarily only the 'base program' that is instrumented, but just used as an example in the diagram. The pseudo-address '0x1004' represents a location at which a floating-point instruction originally existed and was subsequently replaced by the branch. The branch points to the trampoline and somewhere in the trampoline there is a branch pointing to emulation code. The emulator 'registers' are stored in memory alongside the emulation routines but where the hardware already has these registers, the hardware should be used instead.

The design of an emulator for this project appears to be a mammoth task and, in turn, may also similarly be the bespoke design of a trampoline for each assembly instruction signature. I didn't reach a finished end-product

that I hoped for before starting, but I did learn about how userspace emulation might be done. At the beginning, all I knew was that the trampoline would 'jump' to an emulation routine. When deciding how to write the emulator, I didn't understand how I would connect the low-level assembly and the abstract high-level C language. C has at least some mechanism to retrieve the memory address at run-time of its methods through function pointers which makes the task nearly trivial. I am aware that it may be worth further research in determining a more reliable way of retrieving these addresses rather than the direct use of function pointers, however I haven't had a problem thus far. What seems obvious to me now that was not at the time was that I could call my methods in C from the assembly by just using the relevant calling convention and branching to the address of the function pointer. Therefore, in each trampoline all I had to do was ensure that the address of the function pointer matched the relevant emulation routine for the instruction that the trampoline was being written for.

5.5 Limitations

Here I describe some of the limitations of my approach that I have not already covered. Firstly, I talk about how the trampolines were limited in where they could be placed. Secondly, I note that the project's outcome only focuses on a portion of possible relevant binary formats. Lastly, I talk about a minor limitation of how the machine code is loaded into memory which limits its size.

5.5.1 Trampoline placement

Due to ARM's limited instruction width, a branch instruction only has room for a 24-bit operand [7]. Due to how this value is used, only about 67 megabytes of memory can be addressed directly. To branch to a larger range of addresses, typically the address should be resident in a register and the name of the register used as an operand. Since the instrumentation site is exactly the width of the original four-byte floating-point instruction, there is no space to load values into registers. Therefore, this emulation method is constrained by program size. As it currently stands, I have not implemented support for binary executable sections that, along with their trampolines, exceed 67 megabytes of memory. This is fine for most programs though as the restriction implies a considerable program binary size. There may be often enough room to search nearby to place a trampoline in, however this lack of reliability should be considered.

5.5.2 Thumb mode

Thumb instructions are a subset of the ARM ISA that are two bytes wide. Thumb-2 adds onto this to allow four byte instructions to mix with the two byte instructions. Thumb allows for a more condensed binary file and so you would expect it to be commonly output by compilers. The complexities of variable-length instructions with placing probes, among other considerations, made Thumb less accessible to support for this project. Presently, my program only supports ARM mode binary files. I am not aware of the relative frequency of non-Thumb programs in the wild especially for this niche use-case, but I would consider this as a major limitation of the project. I didn't find the time, but future works should certainly consider Thumb as well and should investigate what changes would need to be made to support it.

5.5.3 Inflexible loading

My method requires the whole binary file to be loaded into memory in order to instrument it before it's ran. This is not a limitation of dynamic instrumentation, as the sky is the limit with what modifications are made, but a limit of my implementation. I don't consider this a major limitation as the approach is scalable to a larger number of binary files and more flexible on-demand loading and instrumentation. The size of one binary file is usually not particularly large so it will be loaded entirely into memory before instrumentation anyway.

Chapter 6

Evaluation

6.1 Performance Comparison

To test whether the proposed method performed any better, I decided to measure the time for some programs to execute. The same program will have its floating-point instructions trapped and also separately instrumented with the jump-based probing technique.

Due to difficulties with setting up the environment correctly, I couldn't use the kernel floating-point emulator. My supervisor suggested using the 'getpid()' method in C which should act similarly to triggering the kernel emulation trap. The act of retrieving the process ID, or 'PID', should take negligible time and there is a very similar overhead and pattern of kernel-userspace transitions.

Although the source code for each these programs will be different, they should be very similar. Another difference is that the 'getpid()' method will obviously not perform any floating-point emulation, so in order to make a fair comparison I've also had to disable my emulator such that the emulator is branched to at run-time, but no actual emulation is performed.

Since the emulator currently only implements support for one instruction, it didn't make sense to test it with arbitrary real-world programs. Instead, I constructed a program that performed a block of 'vadd.f32' instructions in a loop. In the case of the trapped program, I replaced each in-line floating-point instruction with a 'getpid()' call in the source code.

Diagram 6.1 shows a comparison made for the programs when the loop's block contains only a single 'vadd.f32' instruction. For each data point, the program was re-run and re-instrumented ten times and then the mean taken. There was only one instruction to find and replace in the instrumentation search my method uses. I naively expected the proposed method to outperform trapping, represented by 'getpid()' here, in all cases, but the data shows that due to the slow start-up overhead of searching and replacing instructions, the jump-based method actually performed worse for programs

that did not run for very long. An improvement over trap-based emulation was only seen when the over about 20,000 floating-point instructions were emulated from the base program. At the scale shown in the diagram,

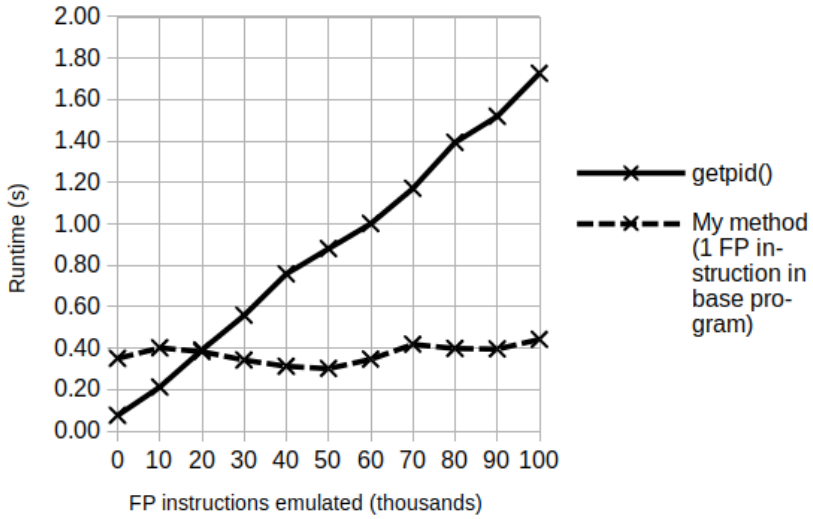

Figure 6.1: A comparison of my method with the previous approach in programs ran for different lengths of time. Each data point represents an average of ten re-instrumented program executions (i.e. the graph is not one run).

performance improved very roughly from linear to near constant time in comparison. The memory usage is also negligibly affected. I think that with a direct effort to improve the performance of my implementation, the threshold where the jump-based method reduces the total run-time could be brought significantly down.

I would expect this threshold to change depending on how many floating-point instructions were in the base program, so using the same methods, instead of using one floating-point instruction (or one 'getpid()' call), I used 10, 100 and 1000 instructions in the loop block. This would mean more trampolines in memory and a longer start-up delay.

Diagram 6.2 shows the trend continues as expected, but shows the limitation of my method. As the number of floating-point instructions in the base program increase to levels that can more reasonably be expected to be in any given program, the instrumentation stage takes significantly longer. For 1000 floating-point instructions in the base program, the jump-based method would only start out-performing trap-based emulation after around a million floating-point instructions had been emulated at run-time. Given

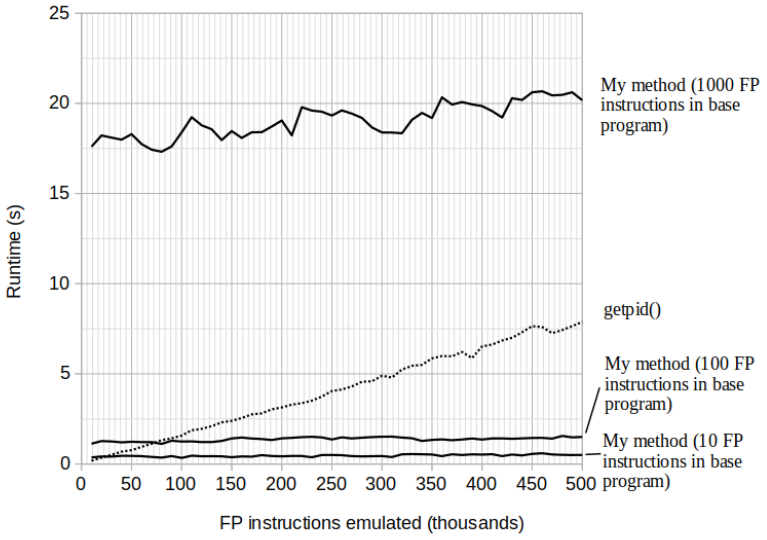

Figure 6.2: A comparison of my method with the previous approach in programs ran for different lengths of time. Tested with 10, 100, and 1000 floating-point operations in the base program before instrumentation.

the number of measurements I had already taken and thus the resolution of the graph, it was not feasible to continue gathering more data to this point.

While the jump-based method should inevitably outperform trap-based emulation, it could take time exceeding minutes to see any non-negligible difference for reasonably normal programs. Given that I had run out of time to make any additional improvements to performance, I'm pleased with the result. I would expect that the efficiency of instrumenting programs with many more floating-point operations can be reduced by potentially orders of magnitude by being more careful about which regions of memory are scanned, using a faster emulator or using Capstone in a more effective way by grouping instructions to be decoded, and implementing more thoughtful heuristics to search for and claim free space for trampolines.

6.2 Requirements Evaluation

All functional requirements were met. The third functional requirement of locating and replacing "any reasonably located floating-point instruction in memory" was a little vague which made it difficult to verify. I think this was due to a lack of understanding when writing the requirements at the beginning. The fourth was similarly difficult to verify as it is very challenging to distinguish instructions and data completely confidently. The last functional requirement, "emulation routine [is] written to memory",

doesn't quite match my implementation depending on how it's interpreted. I didn't understand what was possible when I wrote the requirement as the emulation routine is now written in C and forms part of the instrumentation library. This means that it is technically written to memory, but by normal trivial means, by the dynamic linker/loader, and not by any non-trivial process of reserving memory at run-time and writing emulation machine code.

In terms of non-functional requirements, two stand out as not being met. It was not fair to measure the memory overhead introduced as not all instructions were replaced and no strategy was used to share trampolines and locate them contiguously in memory. The requirement that "memory usage ... must not exceed twice that which would occur with the original program" was therefore not measured. My method is also not compatible with any ARMv7 binary as the solution was incomplete (see the limitation in section 5.5.2).

For the other non-functional requirements, the solution is faster than the trap-based method in long-running programs which I have shown at the start of the Evaluation chapter. The functionality does remain the same for programs that use only the supported ARM instruction, but does not if the program uses the VFP ISA normally as the emulator is incomplete. With a full ARM ISA emulator and minor changes made to connect it to the existing code, my implementation would functionally equivalent to the trap-based kernel emulator.

Chapter 7

Conclusion and Future Work

I have shown the proposed method to be eventually faster than the alternative that I have compared it to, but the project deliverable is fundamentally limited as a proof-of-concept given the required development hours to produce a satisfactory end product that accounts for hardware variations, implements the full range of floating-point instructions, and makes efforts to improve performance of the implementation. In terms of successes, all functional requirements were met and the specification matched. As the solution was incomplete, some non-functional requirements were not met.

The ambitions of this project were vast given what I knew before starting, so although the solution is not complete, it does represent considerable effort. If I were to continue this project in the future, I would find an ARM floating-point emulator library instead of trying to build my own from scratch and I would also fix the problems I have previously described such as with performance and lack of support for Thumb instructions and hardware floating-point registers.

One avenue to look into is that of trampoline placement. Because of limitations discussed in section 5.5.1, it would be good to find a way to support larger programs and dependencies. One strategy being to displace program code with trampolines and move them elsewhere by updating any branches that point in and out of the region. Since the proposed method doesn't support code generated at run-time, floating-point instructions could be intercepted such as while they are being written into memory.

Bibliography

[1] ARM Options. https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html.

[2] Debian ARM Ports. https://wiki.debian.org/ArmPorts.

[3] Debian Ports. https://www.debian.org/ports/index.en.html.

[4] GNU Binutils. https://www.gnu.org/software/binutils/.

[5] The 'Capstone' Disassembly Framework. http://www.capstone-engine.org/.

[6] ld.so(8) — Linux manual page. https://man7.org/linux/man-pages/man8/ld.so.8.html, Aug 2021.

[7] ARM Limited. Arm® architecture reference manual: Armv7-a and armv7-r edition. https://developer.arm.com/documentation/ddi0406/cd/.

[8] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *FREENIX Track: 2005 USENIX Annual Technical Conference*, pages 41–46, 2005.

[9] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[10] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–332, 2017.

[11] Syed Zohaib Gilani, Nam Sung Kim, and Michael Schulte. Virtual floating-point units for low-power embedded processors. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 61–68, 2012.

[12] Gabriel Ferreira Teles Gomes and E Borin. *Indirect branch emulation techniques in virtual machines*. PhD thesis, Dissertation, University of Campinas, 2014.

[13] Mingyi Huang and Chengyu Song. ARMPatch: A Binary Patching Framework for ARM-based IoT Device. *Journal of Web Engineering*, pages 1829–1852, 2021.

[14] Peter B Kessler. Fast breakpoints: Design and implementation. *ACM SIGPLAN Notices*, 25(6):78–84, 1990.

[15] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175–183, 2010.

[16] Arm Ltd. ARM Architecture Reference Manual Debug supplement: Breakpoint Debug Events. `https://developer.arm.com/documentation/ddi0379/a/Debug-Events/Software-Debug-events/Breakpoint-Debug-events`.

[17] Arm Ltd. ARM Compiler - Arm C and C++ Libraries and Floating-Point Support User Guide Version 6.11: About floating-point support. `https://developer.arm.com/documentation/100073/0611/floating-point-support/about-floating-point-support`.

[18] Arm Ltd. What is RISC? https://www.arm.com/glossary/risc.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[20] Ignacio Sanmillan. Executable and Linkable Format 101 - Part 1 Sections and Segments. `https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sections-segments/`.