INTERNATIONAL HELLENIC UNIVERSITY
DEPARTMENT OF INFORMATION AND ELECTRONIC ENGINEERING

BACHELOR THESIS

Emulating x86-64 Applications on RISC-V Environments

Student:
Paris Oplopoios
Student ID: 185246

Supervisor:
Antonis Sidiropoulos

25 Jan 2025

Title of Dissertation: Emulating x86-64 Applications on RISC-V Environments
Code of Dissertation: 24222
Student's full name: Paris Oplopoios
Supervisor's full name: Antonis Sidiropoulos
Date of undertaking: 30-08-2024
Date of completion: 25-01-2025

We hereby affirm the authorship of this paper as well as the acknowledgement and credit of whichever assistance We received in its composition. We have, furthermore, noted the various sources from which We extracted data, ideas, visual or written material, in paraphrase or exact quotation. Moreover, we affirm the exclusive composition of this paper by myself only, for the purpose of it being a dissertation, in the Department of Information and Electronic Engineering of the I.H.U.

This paper constitutes the intellectual property of Paris Oplopoios, the student that composed it. According to the open-access policy, the author/composer offers the International Hellenic University authorisation to use the right to reproduce, borrow, publicly present and digitally distribute the paper globally, in electronic form and media of all kinds, for teaching or research purposes, voluntarily. Open access to the full text by no means grants the right to trespass the intellectual property of the author/composer, nor does it authorise the reproduction, republication, duplication, selling, commercial use, distribution, publication, downloading, uploading, translation, modification of any kind, in part or summary of the paper, without the explicit written consent of the authors.

The approval of this dissertation by the Department of Information and Electronic Engineering of the International Hellenic University, does not necessarily entail the adoption of the author's views, on behalf of the Department.

Dedication

To my family, whose unwavering support, sacrifices, and guidance have been the foundation of my growth. This work is a testament to your love and belief in me.

To my friends in the emulator development community who have motivated me and helped me countless times throughout the years.

Prologue

As time passes, humanity makes great scientific advancements. New technologies appear and replace the old ones, rendering them outdated and leading to their gradual abandonment. It wasn't too long ago when incandescent light bulbs were widely used, while nowadays they see little use as they've been replaced with a better alternative. The progress of science has been of great help to the world.

In many cases however, people like keeping around relics of the past even if they are not as useful as they once were. Eventually, a lot of items that are no longer produced become scarce and even increase in value. This, in turn, makes them harder to acquire for the average person. Most of them can also see some physical degradation and lose their functionality.

The same problem applies to software. While software does not suffer from physical degradation, the required hardware to use said software does, and may become uncommon and costly. This is where emulation comes in. An emulator is a program that aims to replicate the functionality of a specific piece of hardware. Through the use of emulators, countless pieces of software have become usable again without the need of the hardware they were originally meant to run on.

The following chapters describe the implementation of an emulator for programs originally written for x86-64 hardware, allowing them to run on RISC-V hardware. It will unfold the history of these architectures, the design of the emulator, our implementation, and the results. It will tackle machine code Just-in-Time recompilation and Linux kernel emulation. It is my hope that this work results in a useful thesis on the topic of CPU emulators and a valuable emulator that will upsurge x86-64 software preservation.

Abstract

This thesis presents the design and development of a user-space x86-64 Linux emulator for RISC-V systems with the use of just-in-time machine code recompilation. It aims to achieve good runtime performance and minimize the recompilation penalty.
Our primary objective is to explore different approaches to recompilation, analyze their benefits and compilation overhead, and create an emulator that can execute full programs. This thesis offers a comprehensive analysis of our implementation, including instruction decoding, generating a static single-assignment form intermediate representation, optimizations, register allocation, and Linux user-space emulation such as syscall and signal handling.
Finally, this thesis presents the performance results of our emulator and compares them with other emulators, as well as with different algorithms and degrees of optimization.

Περίληψη

Αυτή η πτυχιακή εργασία παρουσιάζει τον σχεδιασμό και την ανάπτυξη ενός user-space Linux x86-64 εξομοιωτή σε RISC-V περιβάλλον με την χρήση μεταγλώττισης κώδικα μηχανής κατα την εκτέλεση (just-in-time recompilation). Στοχεύει να επιτύχει καλή απόδοση και να ελαχιστοποιήσει την επιβάρυνση της μεταγλώττισης.
Ο κύριος στόχος μας είναι να εξερευνήσουμε διαφορετικές προσεγγίσεις μεταγλώττισης, να αναλύσουμε τα οφέλη τους και τα αρνητικά τους, καθώς και να δημιουργήσουμε έναν εξομοιωτή που μπορεί να εκτελεί πλήρη προγράμματα. Σε αυτή την πτυχιακή προσφέρουμε μια ολοκληρωμένη ανάλυση της υλοποίησης μας, συμπεριλαμβανομένης της αποκωδικοποίηση κώδικα μηχανής, της δημιουργίας μιας ενδιάμεσης αναπαράστασης, των βελτιστοποιήσεων, της κατανομής καταχωρητών, και την εξομοίωση του Linux user-space, όπως η διαχείριση syscall και signal.
Τέλος, παρουσιάζουμε τα αποτελέσματα απόδοσης του εξομοιωτή μας και τα συγκρίνουμε με άλλους εξομοιωτές, καθώς και με διαφορετικούς αλγόριθμους και βαθμούς βελτιστοποίησης.

Contents

1 Introduction **1**

1.1 Central Processing Units 1
1.2 Modern Instruction Sets 1
1.2.1 x86-64 1
1.2.2 RISC-V 4
1.3 Linux 5
1.4 Introduction to Emulation 6
1.4.1 Memory and State 6
1.4.2 Interpreters 7
1.4.3 Recompilation 8
1.5 Linux user-space emulation 11
1.5.1 Root filesystem 11
1.5.2 Syscalls 12
1.5.3 Signals 12
1.6 Objectives 12

2 Related work **14**

2.1 Projects 14
2.1.1 QEMU 14
2.1.2 FEX 14
2.1.3 Box86 and Box64 15
2.1.4 Rosetta 2 15
2.1.5 Intel Houdini 15
2.1.6 Comparison 16
2.2 Compiler literature 16
2.2.1 Fundamental work 16
2.2.2 Static Single Assignment form 16
2.2.3 Optimizations 17
2.2.4 Register allocation 17

2.3 Emulation 19
2.4 Conclusion 19

3 Design 20
3.1 Program Loading 20
3.1.1 State 20
3.1.2 Loading Into Memory 21
3.1.3 Dynamic Linker 21
3.1.4 The Stack 22
3.1.5 Root Filesystem 23
3.1.6 Dispatcher 23
3.1.7 CPUID 24
3.2 Recompilation 24
3.2.1 Decoding 24
3.2.2 Intermediate Representation 28
3.2.3 Optimizations 32
3.2.4 Register Allocation 35
3.2.5 RISC-V concerns 38
3.3 Emulation 41
3.3.1 Syscalls 42
3.3.2 Signals 42
3.4 Conclusion 43

4 Results 44

5 Discussion 51

6 Conclusion 54

Bibliography 55

List of Figures

1.1 Different ways rax, one of the GPRS, can be accessed 3
1.2 Synthesizing two's complement negation in RISC-V 5
1.3 An array of 32 uint32_t variables is enough to emulate the registers in this CPU . . . 6
1.4 Allocating 8 megabytes of memory to emulate the RAM in this system 7
1.5 MIPS machine code in hexadecimal and its equivalent assembly 7
1.6 C code for emulating a hypothetical OR instruction 8
1.7 A basic recompilation of x86-64 code to RISC-V 9
1.8 Fitting x86-64 code into basic blocks 10
1.9 Intermediate representations can be very helpful in compilers 10
1.10 Integer multiplication by 32 is equivalent to shifting left by 5 11
1.11 Felix86 emulating x86-64 programs on a RISC-V development board 13

3.1 Our ThreadState struct in C++ 20
3.2 Reading the second x86-64 register, rcx, into the RISC-V register t2 21
3.3 Stack layout according to x86-64 System-V ABI 22
3.4 An effective address in x86-64 24
3.5 Effective addresses can perform complex operations and reduce binary size 25
3.6 x86-64 instructions and their machine code 26
3.7 Some instructions change their operation based on the prefix 26
3.8 Translating x86-64 to intermediate representation 28
3.9 A handler function for an x86-64 logical or instruction 29
3.10 Renaming variables 30
3.11 SSA conversion example 30
3.12 The first assignment here is unnecessary 31
3.13 Phi operations are replaced with moves in the predecessor blocks 32
3.14 Replacing an add with an addi, reducing instruction count and register pressure . . . 32
3.15 A critical edge from Block_1 to Block_4 34
3.16 The critical edge has been broken 34
3.17 The structure of a Briggs-styled register allocator 35

3.18 Conversion of an x86-64 SIMD operation to RISC-V 39
3.19 The price we pay for 8-bit and 16-bit atomics 40

4.1 Results with graph coloring register allocation and an upper limit of 15 blocks 45
4.2 Execution time goes down when compiling more blocks at a time 46
4.3 Results with linear scan register allocation and an upper limit of 10 blocks 46
4.4 Execution times were not affected by our optimizations 47
4.5 Graph coloring produced slightly better code 47
4.6 Our linear scan implementation performs better in hot loops 48
4.7 Compilation overhead is significantly reduced with static allocation and no optimizations 49
4.8 Emulating python3 and executing Python code 50

List of Tables

3.1 Common instruction prefixes and example assembly 25
3.2 Bit fields in ModR/M 26
3.3 Bit fields in SIB 27

Chapter 1

Introduction

1.1 Central Processing Units

Computers rely on processors to perform computations, with the main processor, known as the **Central Processing Unit (CPU)**, coordinating the system. A CPU cannot understand human language, and relies on electrical signals to process information. These signals take the form of instructions that are relevant to that specific CPU. An instruction may be something we want the CPU to perform, such as a numerical addition, or a modification to memory. Different CPU models understand different types of instructions. The set of instructions a CPU understands is called its **Instruction Set Architecture (ISA)**. Software developers write code in high level languages such as C which is then translated to a lower level representation by a program called the compiler. The compiler parses, analyses and optimizes this code and finally emits human readable assembly code. Finally an assembler translates this assembly code to machine code according to the ISA. Processing units are programmed to understand this machine code and perform operations such as arithmetic and logic operations or interacting with memory and peripheral devices.

1.2 Modern Instruction Sets

1.2.1 x86-64

History and Origins

The most common instruction set used today in personal computers is AMD's **x86-64** ISA, first introduced in 2003. It is a 64-bit version of the x86 instruction set. Back in the late 1970s, Intel created

their first 16-bit microprocessor, the Intel 8086, which featured 16-bit registers, addresses, and a 16-bit data bus. As demands for computing power grew, so did the demand for larger registers and addresses, capable of addressing more data in the ever-growing memory capacities. Thus, in 1985, the 32-bit version of the 8086 ISA was introduced, which is now called x86. This instruction set came with backwards compatibility with the 8086 ISA. The processor would begin execution in 16-bit mode capable of running 8086 code, and with the appropriate instructions, the programmer could switch to 32-bit mode and continue execution. The x86 architecture was a success and was widely used in computers around the world.

Eventually, demands grew once again. Memory grew to sizes that a processor could no longer address with 32 bits. Hence, around the late 1990s there was the need for a 64-bit successor to x86, capable of facing the 21st century. CPUs were also getting more advanced with the introduction of pipelines, where multiple instructions are dispatched for execution at once and might even execute in parallel. This lead to a difficult conundrum regarding instruction set design, the problem of instruction scheduling. Instruction scheduling is the reordering of instructions to optimize for better instruction-level parallelism in the CPU's pipeline. A bad ordering can cause stalls in the pipeline, where an instruction might have to wait for the previous instruction to finish before it can start execution.

Two companies stepped forth to create a suitable 64-bit replacement to x86. Intel created a new instruction set architecture that bared no resemblance to x86, called Itanium. Itanium is a Very Long Instruction Word (VLIW) architecture, which takes a static scheduling approach. Compilers for Itanium were responsible for creating an optimal ordering of machine code so that the pipeline is always busy with work and stalls as little as possible. An instruction set designed with better static scheduling in mind to account for modern pipelines would, in theory, lead to better performance.

AMD took a different approach. Their instruction set architecture, which was then called AMD64, would extend x86 and be backwards compatible with it, introducing new instructions and bigger registers. As for scheduling, they decided to rely on dynamic scheduling, splitting the complex instructions into simpler ones and reordering them to keep the multi-scalar pipeline busy for as long as possible while the program is running. This put less significance on the compiler, but required a bit of runtime overhead for reordering the instructions as well as space in the physical chip.

The two instruction sets co-existed for a while, but eventually AMD64 won and Itanium was discontinued. This settled the debate of dynamic versus static scheduling, with dynamic scheduling being the undisputed winner in execution speed [1], used in almost every modern high performance CPU to a significant level. When Itanium was being designed there were no CPUs that used dynamic scheduling so when the first ones were introduced in the mid 1990s it was far too late for the design of the architecture to change. Eventually it became apparent that compilers found it very difficult to create an optimal scheduling of instructions statically, and in some cases it was impossible without runtime

information, which dynamic scheduling had access to.

Finally, x86-64 became the sole successor to x86 and is now the most widely used instruction set architecture for personal computers.

Instruction Set Details

At its core, x86-64 is comprised of 16 **General Purpose Registers (GPR)**, eight more compared to its predecessor. The GPRs may be aliased, which means accessed in parts or as a whole. **Figure 1.1** showcases the different ways the 64-bit GPR `rax` may be accessed. This allows modifications to be made to different segments of the 64-bit register in the case that the entire register is not needed.

The base x86-64 instruction set also defines an extension for **Single Instruction Multiple Data (SIMD)** operations, the Streaming SIMD Extension (SSE). This extension defines 16 128-bit vector registers that can be used both for packed and scalar operations. Later extensions such as Advanced Vector Extensions (AVX) and AVX-512 increase the vector register size to 256 and 512-bit respectively, with the latter introducing 16 extra vector registers.

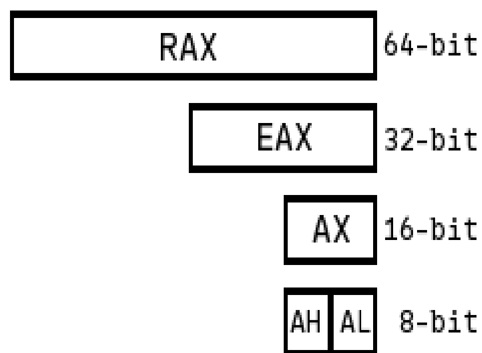

Figure 1.1: Different ways rax, one of the GPRS, can be accessed

Both integer and SIMD instructions may use the flags register, named RFLAGS, as output, and some integer instructions can use it as input. Most of the bits in this register are unused; the important ones being the Zero, Negative, Overflow and Carry bits. Operations such as Add overwrite all of those, and operations like Add With Carry take the carry flag's value as input and also set the appropriate flags.

Some instructions can thus use the value of these flags to either change the control flow of execution, such as Jump If Not Carry, or conditionally move a value to a register. These can be converted from higher level programming constructs such as loops and ifs, making x86 a compact architecture.

Additionally, processors have access to eight 80-bit registers for floating-point operations. These were part of the original floating-point implementation known as x87. The operation philosophy differs

from SSE, with x87 using its registers as a stack. Ultimately x87 fell out of favor and is nowadays rarely used, apart from cases where the extra precision is needed.

1.2.2 RISC-V

History and Origins

RISC-V is an open standard **Reduced Instruction Set Computer (RISC)** ISA, originating from University of California, Berkeley. It was originally designed to support computer architecture research, but has evolved to become a standard suitable for industry implementations. It aims to build on established RISC concepts while avoiding defining implementation details, resulting in a very flexible ISA that can be freely implemented by anyone that abides by its open-source license.

As the name suggests, it is the fifth iteration of RISC architectures published at University of California, Berkeley. The oldest iteration, RISC-I, was introduced in 1980 by David Patterson. Patterson noticed that ISAs of the time had increasingly complex instruction sets. Due to the complex nature of the instructions, they would be internally decoded into microcode which would increase the amount of transistors needed and spend significant time on interpretation. An architecture with simple instructions that run in a single cycle each thus removes the need for microcode which appeared to enhance performance while reducing chip size.

Three years later the second iteration, RISC-II, was introduced. The SOAR and SPUR architectures followed, which were named RISC-III and RISC-IV by Patterson. Finally, in 2010 RISC-V was created. Nowadays several entities produce RISC-V compatible CPU cores that can be used for many purposes including general computing.

Instruction Set Details

RISC-V builds on established RISC principles. It is a load-store architecture, which means it divides operations to memory accesses and ALU operations. The base RISC-V ISA defines a small set of 47 integer instructions with a fixed length of 32-bits per instruction. The ISA also supports variable-length encoding in chunks of 16-bits.

To meet the demands of more complicated processing, a number of official extensions are defined that implementations may choose to include. Extensions are usually named by a single letter, and include 'M' for multiplication support, 'A' for atomics support, 'F' and 'D' for single and double precision floating-point support, 'V' for vector operations and others.

In similar fashion to ARM's Thumb mode, RISC-V also has compact versions of some instructions

via the 'C' extension, optimizing code density and energy efficiency. Unlike ARM however, which required a special instruction to switch between modes, programs may combine 2-byte and 4-byte instructions in a program freely if the 'C' extension is available

Combinations of these extensions can forge implementations of the RISC-V architecture that are suitable for a variety of uses. Different extension combinations can satisfy different needs, thus RISC-V has applications in all sorts of fields, such as embedded hardware or personal computing.

The base RISC-V integer ISA is currently defined in four architectural variants; RV32I, RV32E - a reduced version of RV32I for embedded systems, RV64I and RV128I. The 64-bit variant RV64 received official support in the Linux kernel in 2022. Linux distributions such as Debian and Ubuntu are readily available to be installed on RV64 systems that support the 'I', 'M', 'A', 'F' and 'D' extensions - this combination being commonly referred to as the 'G' extension for convenience.

There are 32 available GPRs and with the 'F' or 'D' extension there are also 32 floating-point Registers (FPRs) for scalar floating-point math operations. The 'V' extension also defines 32 Vector Registers. The length of these vector registers is implementation defined, as the RISC-V 'V' extension is very flexible to adjust to different vector register lengths.

The first GPR, x0, has a hardcoded value of zero. This is similar to other architectures like MIPS, where zero is a useful constant in a RISC context and can be used to synthesize other instructions. For example, there is no two's complement negation instruction in RISC-V, as it can be synthesized by using `sub`, thus reducing the amount of necessary unique instructions in the ISA.

```
sub Rd, x0, Rs
```

Figure 1.2: Synthesizing two's complement negation in RISC-V

1.3 Linux

Linux is a family of free and open source operating systems based on the Linux kernel, an operating system kernel created in 1991 by Linux Torvalds. While Linux was originally developed for x86, it has been ported to countless architectures, more than any other operating system. RISC-V is one of the architectures that has official Linux support.

As with most operating systems, there is a hierarchical separation of resources. The kernel has the highest privilege on the system and applications generally run on a lower privilege level and don't have direct access to system devices and I/O. This lower privilege mode is called user-space. To access higher privilege operations, such as opening files or interacting with devices, programs on Linux use

syscalls. A syscall is a function that is implemented by a special instruction which is different for each architecture, being the `syscall` instruction on x86-64 or the `ecall` instruction on RISC-V.

Linux has more than 400 syscalls, each defined by a separate identifier number, and can take up to 6 different arguments passed in integer registers. For example, on x86-64 the syscall id is passed in the register `rax` and the arguments are passed in `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9`. When the `syscall` instruction is ran, execution is passed to the kernel and the kernel handles the defined operation.

The **Application Programming Interface (API)** and the **Application Binary Interface (ABI)** are stable and compatible across multiple Linux versions, making code portable and binaries compatible across multiple Linux versions.

However, this doesn't mean that syscalls are compatible across architectures. Not only are the syscall ids different between architectures, but some syscalls are different. For example, x86-64 is the only architecture that defines the `arch_prctl` syscall. Similarly, newly added architectures like RISC-V exclude old syscalls that have been replaced with better alternatives, such as `open` which is superseded by `openat`. Such an exclusion is not possible on already established supported architectures for backwards compatibility reasons.

1.4 Introduction to Emulation

Emulation is the process that enables a computer (the **host**) to behave like another (the **guest**). Emulating a CPU involves decoding the machine code of a program and imitating the behavior the CPU. Machine code follows a specific encoding as defined by its instruction set architecture. This encoding can be parsed and understood by software and the expected behavior can be reproduced.

1.4.1 Memory and State

Every CPU is comprised of one or more cores that can execute instructions independently. Each of these cores has some internal state. This includes registers, flags and configurations. Additionally, there may be some global state shared across all cores, such as system registers. Most of the CPU state needs to exist in the emulator, usually stored in memory. As an example, if a system has 32 registers that are 32-bit in size each, that could be represented as the array shown in **Figure 1.3**. Then, if an instruction wanted to access a general purpose register, the emulator can simply index the array.

```
uint32_t gprs[32];
```

Figure 1.3: An array of 32 uint32_t variables is enough to emulate the registers in this CPU

Most systems also have some sort of memory. This could be represented in an emulator by allocating heap memory in the host. For example, if we wanted to emulate a system that had access to 8 megabytes of RAM, we could simply allocate it with `malloc` as demonstrated in **Figure 1.4**.

```
ram = malloc(8 * 1024 * 1024);
```

Figure 1.4: Allocating 8 megabytes of memory to emulate the RAM in this system

As for accessing this memory, the load and store instructions can be emulated to access memory according to the system's memory map or address translation procedure. Some emulators may need to emulate the memory management unit of the CPU, while others may be able to just access the memory that the instruction is trying to access directly.

In the case of user-space emulators, an emulator doesn't have to worry about dealing with addresses that potentially point to memory mapped I/O, as that is handled in the kernel, and can usually emulate guest memory accesses by doing a regular host memory access.

1.4.2 Interpreters

One way of emulating the CPU instructions is an interpreter. An interpreter is a piece of software that decodes instructions of the guest software one by one and immediately reproduces their behavior in code.

```
01 4b 48 25                             or $t1, $t2, $t3
```

Figure 1.5: MIPS machine code in hexadecimal and its equivalent assembly

Generally, interpreters start with fetching bytes from memory at the location of the instruction pointer. Depending on the architecture, this could be a fixed amount of bytes, like 4 bytes per instruction in MIPS, or a variable amount of bytes depending on the instruction, like 1 to 15 bytes in x86-64.

Next, the bytes are decoded to figure out the instruction opcode and registers used. For example, an interpreter for the MIPS instruction set would read the bytes in **Figure 1.5** and recognize it as a logical OR operation.

After recognizing the instruction, an interpreter would execute some piece of code to emulate its functionality. **Figure 1.6** shows an example of what that might look like in C.

```
void handle_logical_or(int rd, int rs1, int rs2) {
    u32 result = gprs[rs1] | gprs[rs2];
    gprs[rd] = result;
}
```

Figure 1.6: C code for emulating a hypothetical OR instruction

At that point the interpreter has finished execution of this instruction and is ready to advance to the next instruction by incrementing the instruction pointer.

This loop of fetching, decoding and executing instructions is costly. Some software expects to be executing on processors that perform hundreds of millions of operations per second, while an interpreter does a lot of work just for a single operation. It requires several cycles to emulate an instruction with an interpreter, and these costs add up quickly.

A few optimizations can be made to interpreters. One involves decoding multiple instructions at once and combining their decoded info into a batch that corresponds to a specific address. This batch is then cached so that further hits of that address no longer require decoding. An example implementation of this technique could involve creating arrays of function pointers to the instruction handler functions. Of course, in the case that the code in that address changes, the batch of pre-decoded information needs to be discarded. This interpreter variety is usually referred to as a cached interpreter.

Another optimization that can improve performance is adding an inline control-flow instruction at the end of every instruction that dispatches to the next instruction [2]. This better utilizes branch predictor units in modern CPUs by duplicating the dispatch code at the end of every instruction, thus giving the branch predictor a better chance of detecting patterns and making better guesses. Usually in machine code there's patterns in the order of instructions, for example an `ADDI` instruction might usually follow a `LUI` instruction in MIPS to load an immediate into a register. By adding a dispatch at the end of every instruction handler patterns like these can be observed, improving branch prediction and keeping the pipeline full for longer.

1.4.3 Recompilation

Recompilation is another approach to CPU emulation. It involves translating the guest machine code to host machine code. This means that, while the process of fetching and decoding instructions still needs to happen, it only happens once as the translated code can be reused. Furthermore, the translated code usually requires far fewer cycles to emulate the behavior of the guest instruction.

On the other hand, recompilation is a lot more difficult to implement and has higher memory requirements to store the translated machine code. Furthermore, it can be computationally expensive

depending on the desired quality of the emitted host code. Better recompiled code requires more time to produce but executes faster, while worse code can be emitted relatively quickly but has greater execution latency.

Figure 1.7 shows what a trivial x86-64 to RISC-V recompiler may produce. This example assumes a pointer to the guest register state is stored in the `s0` register.

```
add rcx, rax
```

```
ld t0, 8(s0)
ld t1, 0(s0)
add t2, t1, t0
sd t2, 8(s0)
```

Figure 1.7: A basic recompilation of x86-64 code to RISC-V

Loading and storing the emulated state on each instruction is simple, but not optimal. A more sophisticated recompiler can assign host registers to guest registers, in a process known as register allocation.

A recompiler usually combines chunks of code into **basic blocks**. A basic block is defined as a straight-line code sequence with a single entry and a single exit. Basic blocks are useful because they are highly amenable to analysis and optimization [3]. Multiple basic blocks may be combined to form a larger chunk of the executable program. By mapping out relations between different basic blocks in a graph, the machine code can be analyzed more thoroughly and across multiple block boundaries. **Figure 1.8** shows what conversion from assembly to a control-flow graph of basic blocks would look like visually.

In many cases, recompilers don't directly translate to machine code. An **Intermediate Representation (IR)** can be used as an initial translation target. This can have multiple benefits. First, it can be a common compilation target for multiple guest languages, and a common intermediary step before emitting machine code for different host architectures. Multiple intermediate representations can be combined to suit a compiler's needs. **Figure 1.9** showcases the translation process in a compiler with an intermediate representation. Having a common intermediate representation can thus reduce code duplication, as any machine independent optimization can be applied on the shared IR.

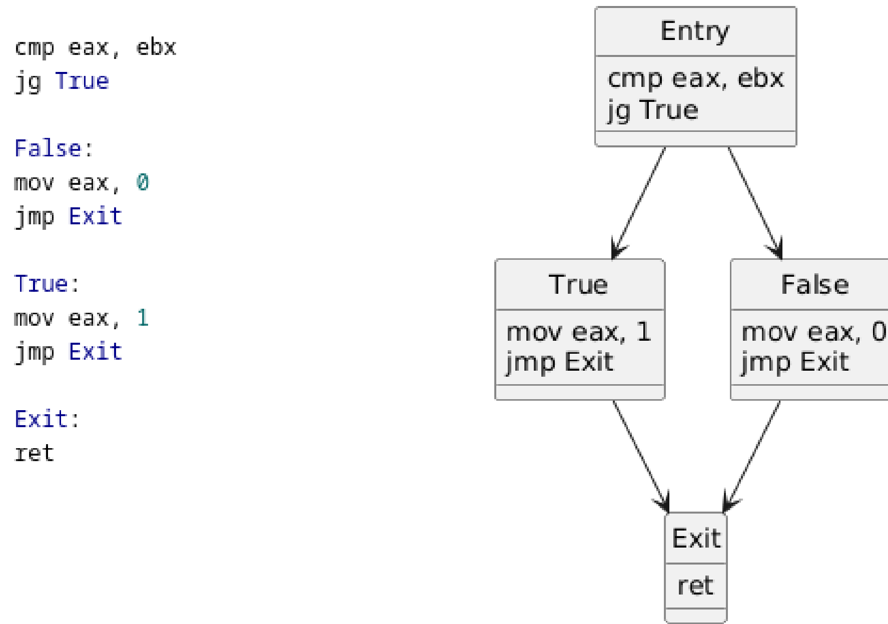

Figure 1.8: Fitting x86-64 code into basic blocks

Another benefit of having an intermediate representation is that complex instructions can be broken up to simpler ones that allow for easier analysis and optimization. This is particularly helpful in **Complex Instruction Set Computer (CISC)** architectures like x86-64, where instructions can do multiple operations such as memory load and stores along with arithmetic operations in a single instruction.

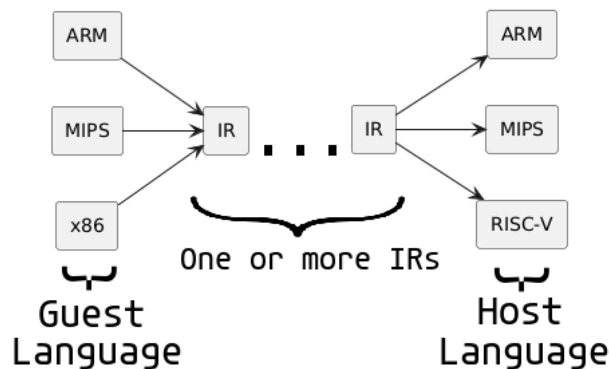

Figure 1.9: Intermediate representations can be very helpful in compilers

Optimizations can be applied to the intermediate representation, such as removing unused code, coalescing copies, and reducing operations to computationally simpler alternatives, as shown in **Figure 1.10**.

Once the intermediate representation has been optimized it can be used to generate host machine code using an emitter. An emitter is a tool or library that creates machine code as programmed by the developer for each IR opcode.

```
x1 <- y1 * 32                    x1 <- y1 << 5
```

Figure 1.10: Integer multiplication by 32 is equivalent to shifting left by 5

Machine code recompilation is usually split between Just-in-Time and Ahead-of-Time recompilation. Just-in-Time recompilation means that as the program is running the segments that are about to execute are recompiled on the spot if they haven't already. Ahead-of-Time compilation means that as much of the executable code as possible is recompiled to host machine code before execution begins.

While Ahead-of-Time compilation may seem superior it can result to longer waiting times before execution begins, and is not always possible. In order to Ahead-of-Time recompile a program, the recompiler would have to find the possible *jump locations* of the program, meaning the locations at which execution might jump to using some control-flow instruction like a branch. It's not always possible to find all jump locations of a program, so a combination of Ahead-of-Time and Just-in-Time compilation might be necessary. For example, Apple's Rosetta 2 emulator Ahead-of-Time recompiles the x86-64 code to AArch64 code that can run natively on their chips. However, it too has a Just-in-Time recompiler fallback in case some code is missed or is generated on the fly and needs to be recompiled.

1.5 Linux user-space emulation

Due to differences between the guest and host architectures, some Linux features need to be emulated.

1.5.1 Root filesystem

Software usually isn't statically linked. This means it needs to link to dynamic libraries at runtime. Since there's differences between guest and host architecture, the host dynamic libraries cannot be used. This means that for Linux user-space emulation there is a need for the equivalent guest root filesystem to be available to the emulator. This includes the necessary libraries like `libc.so.6` or the dynamic linker itself, a program responsible for finding dynamic libraries and loading them in the process's address space.

1.5.2 Syscalls

Due to the differences in Linux syscalls between different architectures described earlier, syscalls also need to be emulated. For some syscalls, the only difference between architectures is their identifier number so emulation is simpler. Others may not exist in the host system and need to be entirely emulated in code.

1.5.3 Signals

In Linux, signals are software interrupts used to communicate with or control processes. Processes can handle signals by installing a signal handler, ignore them or take default actions. Signal handlers will be in guest code, so they need to be recompiled. Some challenges arise with signals as we will see in a later chapter.

1.6 Objectives

The primary goal of this thesis is to present an in-depth analysis of our emulator, Felix86.

Felix86 is an x86-64 Linux user-space emulator, designed to run on RISC-V host systems.

It is open-source and its source code is freely available under the MIT license:

`https://github.com/OFFTKP/felix86`

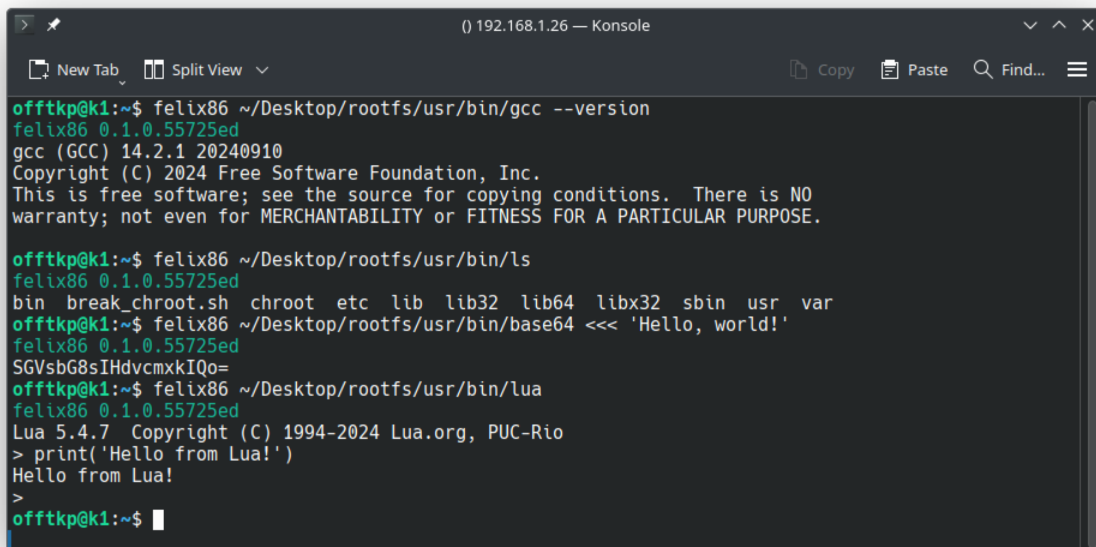

Figure 1.11: Felix86 emulating x86-64 programs on a RISC-V development board

Chapter 2

Related work

2.1 Projects

2.1.1 QEMU

QEMU [4] is an open-source generic machine and user-space emulator and virtualizer. It is capable of emulating an entire machine in software while utilizing machine code recompilation to achieve good performance. QEMU is also capable of providing user-space emulation, translating Linux and BSD guest syscalls to host counterparts. QEMU supports the emulation of many architectures, such as x86, ARM, PowerPC, RISC-V and MIPS. A recompiler known as the Tiny Code Generator (TCG) is used to translate guest machine code into TCG operations, a machine-independent intermediate representation. This IR can be then optimized and translated into host machine code or interpreted as a slower fallback. It is the most comprehensive user-space emulator currently available, with many developers as part of its team.

2.1.2 FEX

FEX [5] is an open-source user-space emulator for x86 and x86-64 on AArch64 hosts. It aims to achieve good performance for software that requires 3D acceleration such as video games. This is partially done via *thunking*, which is the process of replacing guest function calls with host function calls for better performance. This can be particularly useful for functions in libraries like `libGL.so`, the OpenGL API library. Software that utilize computer graphics make frequent calls to this library, which when replaced with host calls can improve performance. FEX also utilizes recompilation for translation of x86-64 code to AArch64, while also making use of the AArch64 flags for emulation of

the x86-64 ones.

2.1.3 Box86 and Box64

Box86 and Box64 [6], [7] are open-source user-space emulators for x86 and x86-64 respectively on AArch64 and RISC-V hosts. Like FEX, it also features a recompiler and takes advantage of thunking commonly used libraries. It can achieve near native speeds in graphics intensive apps as most work is done in native libraries.

2.1.4 Rosetta 2

Rosetta 2 is a closed-source user-space emulator developed by Apple to aid with their transition from x86-64 to AArch64. Rosetta 2 shines in x86-64 to AArch64 translation due to multiple factors. First, it can utilize an Apple-only, non-standard ARM extension that enables hardware Total Store Ordering (TSO) support. The x86-64 architecture has a strong memory model and emulating it in weak memory model architectures like AArch64 or RISC-V normally is expensive, because it involves the use of memory fences near memory reads and writes. Another Apple-only extension enables the hardware calculation of the Parity and Auxiliary flags. Normally x86-64 and AArch64 have four flags that function similarly and a recompiler can take advantage of that, namely the Negative, Overflow, Carry and Zero flags. However, there's also the less frequently used Auxiliary and Parity flags, that also require emulation. With this extension, the results of these flags are calculated during specific instructions and don't require software emulation on Rosetta. Finally, Rosetta 2 takes advantage of ahead-of-time translation of the binary's text segment. Most of the executable code can be discovered and translated, while the rest can be emulated with a just-in-time fallback [8].

2.1.5 Intel Houdini

Intel Houdini is a closed source ARM on x86 user-space emulator primarily used for Android on x86 emulation. It does not include a recompiler and instead interprets ARM instructions and reproduces their behavior in code [9]. It is used on x86 smartphones and x86 Chromebooks. Not a lot is known about it as there is no public documentation and little mention of it by Intel.

2.1.6 Comparison

Tests [10] have demonstrated that while QEMU covers a vast assortment of systems, it is bested by other emulators like FEX or Box64 that focus on better runtime performance when it comes to applications that use 3D acceleration.

FEX provides optional software emulation of TSO by using memory fences, ensuring accurate behavior and thread safety. Enabling this mode introduces some expected performance degradation. On the other hand Rosetta 2 can enjoy the TSO extension found in Apple's chips and thus does not emulate TSO in software.

2.2 Compiler literature

2.2.1 Fundamental work

The work of Frances E. Allen [11] lays the groundwork for several key concepts in compiler theory, including the introduction of basic blocks and control-flow graphs. These concepts are fundamental in most compilers and facilitate analysis and optimization.

Kildall [12] contributes important theoretical foundations for data flow analysis, specifically the introduction of iterative algorithms for solving data flow equations which came to be known as ”Kildall's algorithm”. It demonstrated how multiple optimization problems can be formulated within this framework.

2.2.2 Static Single Assignment form

Rosen et al. [13] propose an intermediate representation form that can simplify optimization named **Static Single Assignment (SSA)** form. In this form, every variable has a single unique assignment. If a variable previously had multiple assignments each assignment is renamed to accommodate the rule, uses of the variable being replaced with the newly renamed counterpart. At join points of the program special *pseudo-assignments* named phi are used to select a variable based on control flow.

Cytron et al. [14] presents a novel algorithm for computing SSA form for arbitrary control flow graph. This algorithm, recognized as the first method that efficiently constructs SSA form, has been widely adopted by many sophisticated optimizing compilers such as LLVM. It computes dominance relations between basic blocks and spots the dominance frontiers of the control flow graph, the places where phi nodes need to be inserted.

Choi et al. [15] extends the aforementioned algorithm to construct pruned SSA form, which includes no dead phi functions. This is done by computing liveness information and only introducing phi instructions for live variables. Due to the high computational cost of computing liveness information, Briggs et al. [16] suggests a different way of eliminating unnecessary phi instructions, by constructing semi-pruned SSA form which prevents dead phi instructions locally to a basic block.

Braun et al. [17] argues that the algorithm proposed by Cytron et al. has two major drawbacks: The input program needs to represented as a control flow graph and that the minimality of phi function placement is guaranteed by relying on several other analysis. In turn, they present a different algorithm for SSA construction that doesn't require dominance frontiers, making it suitable for use directly on an abstract syntax tree, and can compute pruned SSA form for all programs.

2.2.3 Optimizations

Kildall [18] presents a technique for global analysis of code in order to perform various optimizations such as constant folding and common subexpression elimination. It extends previous work on straight-line sequence optimizations to include branching structure.

Cocke and Kennedy [19] propose an algorithm for strength reduction, which is the reduction of computationally expensive operations to simpler but identical counterparts.

Morel and Renvoise [20] constructed an optimization that can eliminate expressions that are only available on some paths of a program. This is done by inserting the common subexpression on the path that is missing it, thus allowing the subexpression to be eliminated at the join point. It is an optimization that performs common subexpression elimination and loop-invariant code motion at the same time, making it one of the most important optimizations in optimizing compilers.

Wegman and Zadeck [21] introduce a contemprorary algorithm for constant propagation on SSA form. It can find opportunities for constant folding better than traditional approaches by simultaneously eliminating unreachable code, making it a crucial optimization in many compilers that convert to SSA.

2.2.4 Register allocation

Graph Coloring

The problem of assigning an arbitrary amount of variables to a fixed number of registers is called **Register Allocation**. Chaitin et al. [22] showed that the problem can be abstracted as a graph-coloring problem along with proof that register allocation is an NP-complete problem. Nodes in the

graph represent variables to be allocated and edges represent interferences in the live ranges of the variables. In order to allocate the variables the graph needs to be colored in a way that no neighboring nodes share the same color, colors in this case being the set of available host registers.

Not all graphs are colorable, so a later paper by Chaitin [23] incorporates spilling into the algorithm. Spilling is the process of storing a host register to memory in order to make it allocatable again. A spilled variable is stored to memory after every definition and loaded from memory on every use. Heuristics are used in order to select the appropriate variable for spilling.

Briggs et al. [24] describes two improvements to the Chaitin-style register allocator. One is a stronger heuristic for finding an optimal interference graph coloring. It decreases the number of procedures that require spilling and aims to reduce the amount spill code when it is unavoidable. Chaitin's algorithm fails to find optimal colorings in some cases, such as a diamond graph, where as Briggs's does not. Secondly, Briggs et al. extends Chaitin's treatment of rematerialization to handle a larger class of values. Rematerialization describes the recomputing of a variable as an alternative to spilling. While optimizations like common subexpression elimination aim to reduce common expressions this increases register pressure as expressions now have larger live ranges. Rematerialization can introduce new previously computed expressions for the sake of decreasing register pressure.

Traditionally graph coloring is applied on non-SSA form, however, Hack et al. [25] showcases that interference graphs in SSA form are *chordal* which results in a couple appealing properties. One, that their chromatic number is equal to the largest clique in the graph. Two, that they can be colored in quadratic time.

While graph coloring's computational overhead is usually unattractive to JIT recompilers, it's possible to trade-off some overhead for slightly worse spilling decisions. Cooper and Dasgupta [26] demonstrate such a method.

Some just-in-time recompilers choose to use graph coloring as their register allocation algorithm, such as Hotspot's c2 compiler.

Linear Scan

While graph coloring register allocation can produce very good results it is often associated with significant computational overhead. This is especially detrimental in situations where code generation speed is important, like just-in-time recompilers.

Due to this, Poletto et al. [27] describe an algorithm that can allocate registers to variables globally in linear time. It can produce register allocations significantly faster than graph coloring, while maintaining good results. It is favored in several JIT recompilers that need fast compilation speed to avoid

compilation stutter.

Traub et al. [28] describe refinements to the linear scan algorithm that improve code quality near to that produced by a graph coloring allocator.

Mössenböck et al. [29] show that linear scan can be applied in SSA form to simplify data flow analysis and produce shorter intervals.

2.3 Emulation

Bellard [30] discusses QEMU and its capabilities along with difficulties in its creation. The user mode emulation section specifies that the memory management unit does not need to be emulated, as the host manages memory mappings. QEMU also operates on basic block granularity, which means that only one block gets translated and executed at a time.

Baraz et al. [31] presents a user-space emulator that allows x86 programs to run on Itanium hardware. This emulator is also tiered, as it carefully examines blocks that are used too frequently and marks them as optimization candidates. Since optimization algorithms may slow down compilation, a tiered approach like this that only optimizes hot code could produce good results while keeping compilation overhead low. This translation layer would also operate on a single basic block at a time, except for its high tier. When multiple blocks have been marked as hot for optimization, they may be combined and optimized together to achieve greater results.

2.4 Conclusion

While vast research on compilers and related algorithms exists, the same cannot be said for papers focusing on user-space emulation. Nevertheless, numerous projects, both open-source and proprietary, serve as user-space emulators. Documentation detailing their development processes however is typically limited. In this paper we aim to document the complete development process of a user-space emulator that leverages recompilation. Additionally we want to explore the viability of compilation strategies that are less commonly explored in just-in-time machine code recompilation, such as optimizations and global register allocation.

Chapter 3

Design

3.1 Program Loading

3.1.1 State

As mentioned in the introduction, the emulator functions like a virtual machine – it needs to keep the CPU state in memory. Not only that, but recompiled code needs a way to readily access this state. Additionally, CPU state is unique to every thread, so if there's multiple threads there's going to be multiple states, and each thread needs to know where in memory their state is.

```
struct ThreadState {
    u64 gprs[16]{};
    u64 rip{};
    u64 fp[8]{};
    XmmReg xmm[16]{};
    bool cf{};
    bool pf{};
    bool af{};
    bool zf{};
    bool sf{};
    bool of{};
    bool df{};
    u64 gsbase{};
    u64 fsbase{};
}
```

Figure 3.1: Our ThreadState struct in C++

In our emulator, we allocate the general purpose register `s1` to have the sole purpose of holding the CPU state pointer. This way, if recompiled RISC-V code needs to access a guest register, it can use a read or write relative to `s1` as seen in **Figure 3.2**. Since the RISC-V ABI specifies that s1 is a callee-saved register we don't need to preserve it when calling C functions from our recompiled code, such as our syscall handlers.

```
ld t2, 8(s1)
```

Figure 3.2: Reading the second x86-64 register, rcx, into the RISC-V register t2

3.1.2 Loading Into Memory

Linux binaries are in Executable and Linkable Format (ELF). This format contains a few headers that specify various information, and the rest of the data is split into segments and sections. Segments and sections can overlap with each other – generally segments are important to program loading and sections are relevant during program linking.

Segments can be marked as loadable, which means they need to be loaded into memory. Such segments include `.text` which is the executable code segment, `.data` which is the readable and writeable data segment, and `.rodata` which is the read-only data segment. These need to be loaded into memory at specific offsets, which they define in the format.

Our emulator first finds out how much memory it needs to allocate to fit the entire program. This is done by taking the sum of all loadable segment sizes and aligning it to a 4KB page size. It then allocates the memory and copies the loadable segments over from the file. It also clears the memory for the `.bss` segment, which serves as a zero-initialized `.data` segment. At the end of program memory resides the start of the heap memory. Any future `brk` syscalls may increase the size of the heap.

3.1.3 Dynamic Linker

Software nowadays is almost always dynamically linked, due to the many benefits of dynamic linkage. This means that software is not packed with the necessary libraries, but instead a different software called the *interpreter* (not to be confused with CPU interpreters) or **dynamic linker** is responsible for finding and loading the necessary libraries into memory. If a library is not found an exception occurs before the program has a chance to start. When every library is loaded and the interpreter has finished, it passes execution to the entrypoint of the program.

Interpreters need some information on the program they are about to run. On Linux, this is provided via ELF sections like `.dynamic`. Since the kernel is responsible for loading the program and the interpreter into memory, the interpreter also needs to know where in memory the program resides to read its sections. This information is passed to the interpreter via the **auxiliary vector**, an array that resides in the stack. The interpreter can parse the contents of the stack to find this array, and it can use its entries, like `AT_PHDR` to find the program header, or `AT_ENTRY` to get the entrypoint address.

A program can specify which interpreter it wants to use via the `.interp` ELF section. For x86-64 Linux executables it is almost always the case it uses the `ld-linux-x86-64.so.2` interpreter.

Our implementation first checks if the program is statically or dynamically linked. Statically linked programs start execution at their entrypoint and aren't linked to shared libraries. For dynamically linked programs, it loads the interpreter specified by the `.interp` section into memory and starts execution at the interpreter's entrypoint, only after creating the stack first.

3.1.4 The Stack

Before execution can begin, the program's stack needs to be created and initialized. The structure of the stack needs to follow the rules defined in the x86-64 System-V ABI [32].

Purpose	Start Address	Length
Unspecified	High Addresses	
Information block, including argument strings, environment strings, auxiliary information ...		varies
Unspecified		
Null auxiliary vector entry		1 eightbyte
Auxiliary vector entries ...		2 eightbytes each
0		eightbyte
Environment pointers ...		1 eightbyte each
0	`8+8*argc+%rsp`	eightbyte
Argument pointers	`8+%rsp`	argc eightbytes
Argument count	`%rsp`	eightbyte
Undefined	Low Addresses	

Figure 3.3: Stack layout according to x86-64 System-V ABI

Our emulator, following the System-V ABI, allocates memory for the stack and begins by inserting strings that need to be pointed to later. These include environment variable strings such as `HOME=/home/user` or any arguments that will end up being pointed to by `argv`. After this data

is added, we need to add the null-terminated auxiliary vector entries to convey data to the interpreter. Next follows another null-terminated array of `char` pointers for environment variables. Finally, the argument pointers need to be added followed by the argument count. These are going to be available to the main function of the program via `int argc` and `char* argv[]`. After the stack is ready, the register `rsp` needs to be set to point to what it pushed last, in this case the argument count, and execution can begin.

3.1.5 Root Filesystem

Programs that run on Linux usually expect a specific filesystem structure. For example, shared libraries can be usually found in the `/lib` or `/usr/lib` directories, while executables are commonly stored in `/bin` or `/usr/bin`. For an emulator to successfully run dynamically linked programs, the necessary guest libraries need to be available.

Our emulator provides a flag to configure the root filesystem path. Upon start-up, the root filesystem is sanity-checked for having the correct file structure. During execution, the guest process cannot open, create, modify or delete files that are not inside the root filesystem. An attempt to do so via a syscall returns an `EACCES` error to the guest, forbidding access. This effectively sandboxes the application to avoid unwarranted modifications to the host's files.

3.1.6 Dispatcher

In recompilers, a dispatcher is a piece of code that is responsible for checking if the code at the instruction pointer has been recompiled, and either running the subroutine responsible for recompiling or passing execution to the already recompiled code. When the chunk of recompiled code has finished execution it returns to the dispatcher to repeat the process.

Dispatchers can be implemented in high-level language code. However, for performance reasons, they are sometimes implemented in assembly. Our emulator implements the dispatcher in RISC-V assembly. This gives us better control of the dispatch code, allowing us to prepare registers such as the thread state pointer register and push the callee-saved registers to the stack before calling on the recompiled code. By storing these registers they are free to be used by the recompiler, as they will be restored to their previous state in the event that the dispatcher needs to exit.

3.1.7 CPUID

The `cpuid` instruction is a special instruction that provides info about the CPU. The dynamic linker uses it to setup tunables for `libc`. Using it, the dynamic linker can determine presence of specific x86-64 extensions such as SSE or AVX and choose algorithms that utilize them on runtime. Commonly used functions such as `memcpy` have multiple execution paths and the best path is chosen for better performance.

A call to `cpuid` will read the `rax` and `rcx` registers and return a result in the `rax`, `rbx`, `rcx`, `rdx` registers, usually in the form of bitfields. Our emulator implements extensions up to SSE2 at this time, so the `cpuid` instruction needs to reflect an appropriate CPU.

3.2 Recompilation

3.2.1 Decoding

The first step in recompilation, commonly referred to as the frontend, needs to decode guest machine code and call an appropriate function. x86-64 is a variable-length instruction set architecture. This means that, unlike fixed-length architectures such as MIPS, each instruction can have a variable length of 1 to 15 bytes.

Addressing modes

x86-64 instructions usually operate between two registers or a register and memory. Instructions can access memory by using an **effective address**. An effective address is an address calculated by a base register, a scaled index register and a displacement. An example is shown in **Figure 3.4**. Scale can be 1, 2, 4 or 8. This sort of memory addressing is useful when accessing arrays of objects in object oriented languages, while the displacement can be used to access individual fields in an object. Addresses can also use the instruction pointer as a base to do rip-relative addressing.

```
[base + index * scale + displacement]
```

Figure 3.4: An effective address in x86-64

Effective addresses are powerful and can perform relatively complex operations. For example, the instruction `lea` (Load Effective Address) can load an effective address into a register, which means

it can perform multiplication by a scale and addition at the same time. **Figure 3.5** shows one such example, compared to the equivalent without using `lea`.

```
mov rax, rbx
shl rcx, 3
add rax, rcx
add rax, 1234
```

```
lea rax, [rbx + 8 * rcx + 1234]
```

Figure 3.5: Effective addresses can perform complex operations and reduce binary size

Effective addresses can also be relative to the special registers `fsbase` and `gsbase` via the use of a couple of specific prefixes. These registers don't have a defined purpose in the ISA, however Windows and Linux use them for thread local storage.

Prefixes

Decoding starts by scanning for prefixes. An instruction is allowed to have as many prefixes as it wants as long as it doesn't exceed 15 bytes in total. In most cases however, the amount of prefixes in an instruction is 0 to 3. Prefixes are used to change various properties of the instruction, and in some cases change the instruction as a whole. For example, `0xf0` is the lock prefix which specifies that a read-modify-write operation should happen atomically. **Table 3.1** shows some of the most common prefixes encountered on programs.

Table 3.1: Common instruction prefixes and example assembly

Prefix	Byte	Example Assembly
Lock	0xf0	`lock add [rax], ecx`
Operand Size Override	0x66	`sub bx, cx`
Address Override	0x67	`mov [eax + ebx], edx`
REX	0x40 - 0x4f	`xor r8, r15`

Opcodes

Prefixes end when decoding reaches an opcode byte. An opcode byte is a byte which is defined by the ISA to correspond to a specific operation. Each instruction can have 1 to 3 opcode bytes. **Figure 3.6** shows instructions with one, two and three bytes respectively.

```
add eax, ebx                 ; 01 d8, opcode: 01
cmovg esp, ebp               ; 0f 4f e5, opcodes: 0f 4f
movbe [rax], ebx             ; 0f 38 f1 18, opcodes: 0f 38 f1
```

Figure 3.6: x86-64 instructions and their machine code

The `0x0f` byte signals that a secondary byte will follow. Instructions with a `0x38` or `0x3a` secondary byte also have a third byte.

On some instructions a REPNZ, REPZ or Operand Override prefix means that the instruction takes a different form. For example, the `addps` instruction which is for 32-bit floating-point packed addition can become `addpd`, `addss` or `addsd` for packed double, single scalar and double scalar operations accordingly, as shown in **Figure 3.7**.

```
addps xmm0, xmm1  ;    0f 58 c1
addpd xmm0, xmm1  ; 66 0f 58 c1
addss xmm0, xmm1  ; f3 0f 58 c1
addsd xmm0, xmm1  ; f2 0f 58 c1
```

Figure 3.7: Some instructions change their operation based on the prefix

ModR/M

The addressing mode is specified by the ModR/M byte. This is a byte that may follow the opcode bytes. Instructions that deal with 2 operands, which is most instructions before Advanced Vector Extensions (AVX), have a `reg` operand and an `rm` operand. The `reg` operand is always a register while the `rm` operand is a register or memory. This is specified by the ModR/M byte. It contains 3 fields as shown in **Figure 3.2**.

Table 3.2: Bit fields in ModR/M

Bit	Size	Field
0	3 bits	`rm`
3	3 bits	`reg`
6	2 bits	`mod`

The `mod` field specifies the addressing mode. A value of `0b11` indicates a register-to-register operation, while any other value indicates different kinds of register-to-memory operations. The `reg` field usually indicates one of the registers used. Note that while the `reg` field only has 3 bits, it can be extended by another bit from a REX prefix to reach 4 bits – being able to address all 16 general purpose registers. In some cases, `reg` is used to encode an opcode extension instead of a register. For

example, instructions that only require 1 operand like `neg` are usually packed in groups with other similar instructions. This means that the same opcode byte is used to encode multiple instructions, and the `reg` field in ModR/M is solely used to differentiate between them.

The `rm` field usually indicates a register if the addressing mode is register-to-register, or a base register for an effective address. Some specific `mod` and `rm` combinations however indicate that another byte is used to aid with providing more information about the effective address. That is the Scale Index Base (SIB) byte, with its bit fields shown in **Figure 3.3**. When a SIB byte is used, the `base` field is the base register in the effective address, and the `index` field is the index register that is multiplied by `scale`.

Table 3.3: Bit fields in SIB

Bit	Size	Field
0	3 bits	`base`
3	3 bits	`index`
6	2 bits	`scale`

Some instructions also operate with an immediate, such as some forms of the `imul` instruction, that multiply `rm` by an immediate and store the result in `reg`. Some addressing modes require a displacement, which is also encoded right after the ModR/M and SIB bytes. Additionally, some instructions operate implicitly on the `rax` register. Furthermore, some instructions don't require a MorR/M byte at all, as they deal with fixed registers.

Implementation

Our emulator collects the prefix bytes until it stumbles upon an opcode byte. It then checks if there's a need to parse a secondary or tertiary byte by using instruction metadata tables. Additionally it examines whether a ModR/M and SIB byte are present in this instruction and whether or not there's an immediate. The instruction metadata also signal whether the `reg` or `rm` operand of this instruction are an xmm register and whether this is an instruction that deals with the byte form of registers such as `al`.

Once all the information about the instruction has been gathered an appropriate function pointer is called to handle the implementation of this instruction. It then advances to the next instruction until it hits a control-flow instruction like `jmp` or `ret`. At that point the basic block needs to end. If the jump target is known at compile-time, the basic block ends with a jump to a new basic block at that location. Otherwise, the block needs to end and return to the dispatcher, because the jump address will only be known at run-time.

3.2.2 Intermediate Representation

The job of the instruction handler is to generate intermediate representation that emulates the x86-64 instruction. Our intermediate representation is three-address code form that closely matches RISC-V instructions. An example conversion is shown in **Figure 3.8**. Remember that since RISC-V doesn't have hardware flags we need to manually calculate their value. As we will see later, optimizations take care of unused flag calculations.

```
mov rax, 5                    rax ← 5
mov rbx, 3                    rbx ← 3
add rbx, rax                  rbx ← rbx + rax
                              zf ← rbx == 0
                              sf ← rbx < 0
                              cf ← ...
                              pf ← ...
                              af ← ...
                              of ← ...
```

Figure 3.8: Translating x86-64 to intermediate representation

While decoding instructions, our frontend fills a vector of instruction structs with these IR instructions. Each IR instruction has an index which serves as its name, up to 4 operands that are pointers to other instructions, a 64-bit integer that can hold immediate info, its opcode and some other information. In **Figure 3.9** you can see an example of an instruction handler function. This specific handler is for instructions like `or rax, [rbx + rcx + 20]`. Some helper functions are used to reduce code duplication.

These handler functions produce the initial intermediate representation. In theory, the `GetReg` and `SetReg` instructions load and store the x86-64 state. In practice however, reading and writing to state doesn't need to happen on every instruction. In fact, reading only needs to happen at the start of the function, and writing only needs to happen right before exiting back to the dispatcher. Between entry and exit values can reside in registers which is faster. Thus, this intermediate representation gets converted to a form where reads from state only happen at the entrance of the function and writes to state only happen at the exit. This happens during conversion to Static Single Assignment form.

```
IR_HANDLE(or_reg_rm) { // or r16/32/64, rm16/32/64 - 0x0B
    Instruction* reg = ir.GetReg(inst->operand_reg);
    Instruction* rm = ir.GetRm(inst->operand_rm);
    Instruction* result = ir.Or(reg, rm);
    ir.SetReg(inst->operand_reg, result);

    Instruction* zero = ir.Imm(0);
    Instruction* p = ir.Parity(result);
    Instruction* z = ir.IsZero(result, size);
    Instruction* s = ir.IsNegative(result, size);

    ir.SetCPAZSO(zero, p, nullptr, z, s, zero);
}
```

Figure 3.9: A handler function for an x86-64 logical or instruction

Static Single Assignment form

To make optimizations easier we convert our IR to SSA form. This is done using the Cytron et al. [14] algorithm. This algorithm describes SSA conversion to happen in three steps:

- The dominance frontier mapping is constructed from the control flow graph
- Using the dominance frontiers, the locations of the phi functions for each variable in the original program are determined
- The variables are renamed by replacing each mention of an original variable V with an appropriate mention of a new variable Vi

There's multiple algorithms for finding dominance frontiers. We are going to be using Cooper et al.'s [33] algorithm for its simplicity and efficiency. It provides us with a way of calculating dominance in our control-flow graph and finding dominance frontiers. After calculating dominance frontiers phi instructions can be placed. This is explained in Cytron et al.'s paper, it is done by iterating the dominance frontiers and using a worklist algorithm. Each node X in the worklist ensures that each node Y in DF(X) receives a phi instruction.

With the first two steps finished, variable renaming can happen. This process replaces variables with multiple definitions to multiple variables, such that each variable has only a single definition. In **Figure 3.10** we can see variable renaming in action.

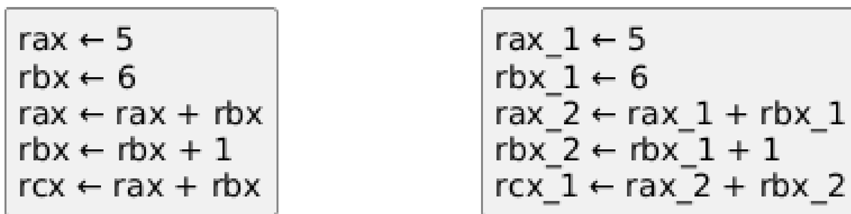

Figure 3.10: Renaming variables

And with that our SSA conversion is finished. An example of complete SSA conversion is shown in **Figure 3.11**.

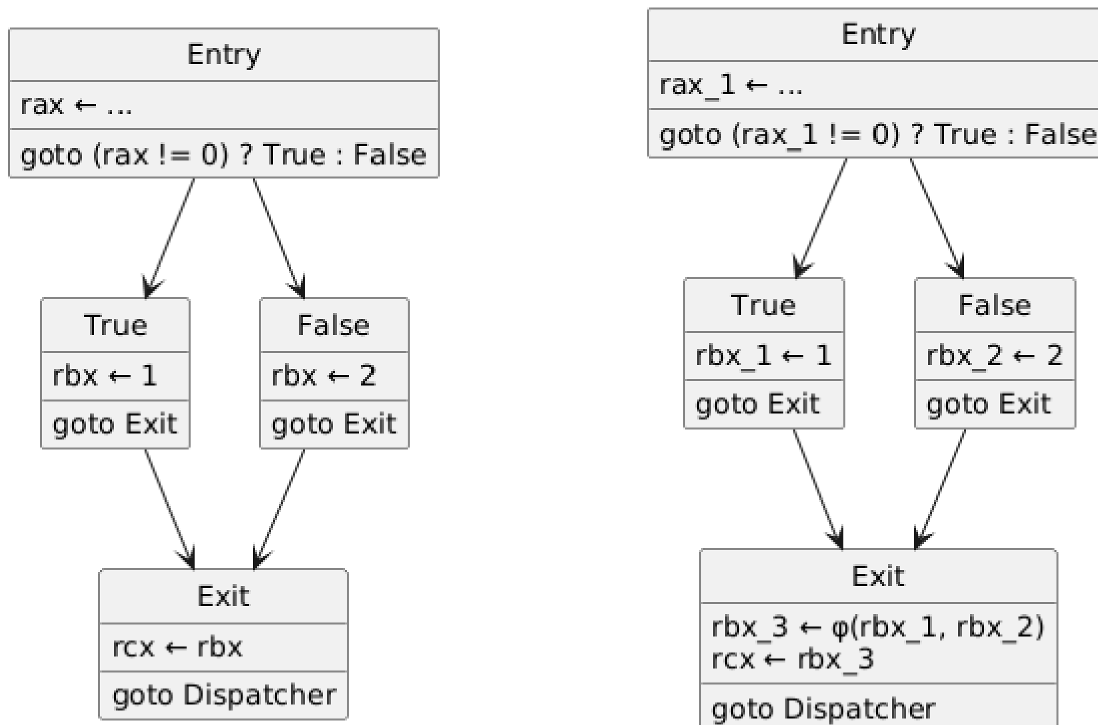

Figure 3.11: SSA conversion example

Static single assignment form makes many optimizations easier. For example, consider the piece of code in **Figure 3.12**. The first assignment is unnecessary because the variable `rax` is immediately given a new value right after, with no uses in between. For a human this is obvious, however a program would have to perform reaching definition analysis to figure this out.

```
rax ← 1                  rax_1 ← 1
rax ← 2                  rax_2 ← 2
rbx ← rax                rbx_1 ← rax_2
```

Figure 3.12: The first assignment here is unnecessary

However, after renaming variables we can deduce that the first assignment can be removed as it is not used anywhere. In the previous example, `rax_1` has no uses while `rax_2` has one use. The optimization of removing variables with no uses is called dead-code elimination and is one of the optimizations we perform. Dead-code elimination is also used to remove redundant flag calculations. While x86-64 flags are calculated in hardware, RISC-V needs to emulate them in software so it is in our best interest to remove flag calculations that produce an unused result. This is trivially done with dead-code elimination, as those flags will have no uses, and after they are removed their calculations will also have no use.

Phi instructions select a value based on control flow. While they are useful for using the correct version of the renamed variables in join points, they need to be eliminated before the host machine code emitting step. Essentially, SSA form is an IR form we enter that aids with optimization, but then needs to be exited before final code emission, as phi instructions aren't generally kept around during code emission. To break up phi instructions, copy operations are inserted in the predecessor blocks. In **Figure 3.13** you'll see the example from earlier and the resulting IR after exiting SSA form.

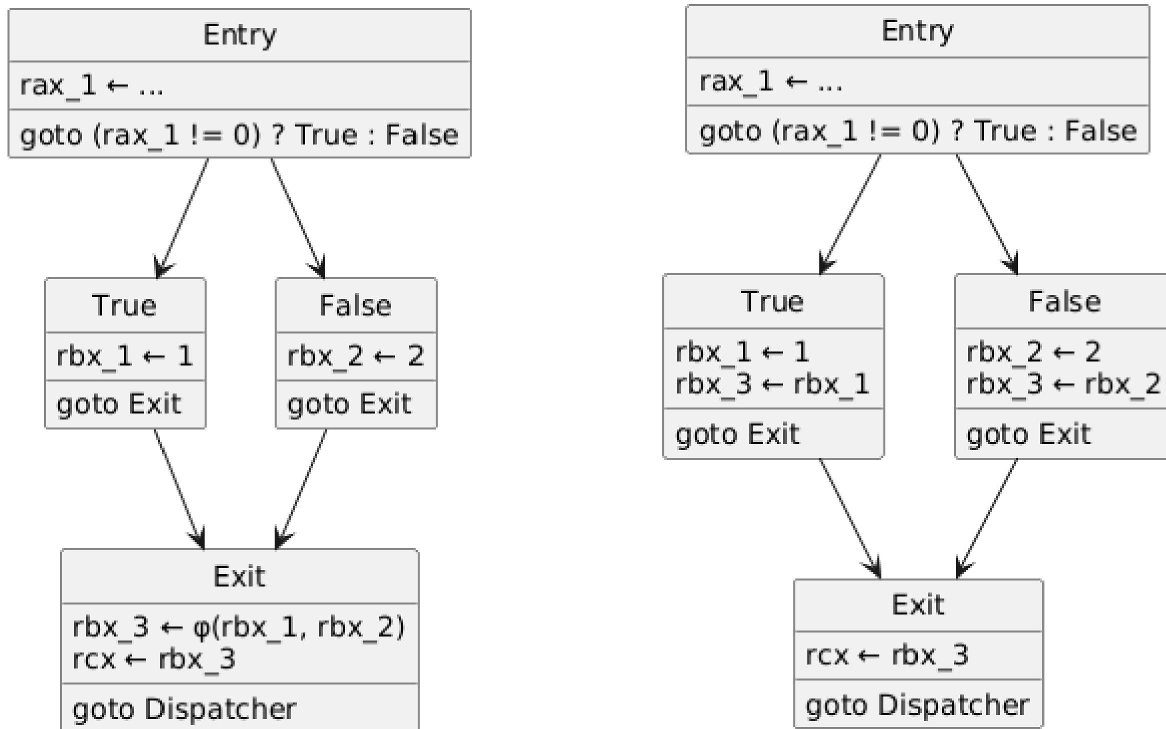

Figure 3.13: Phi operations are replaced with moves in the predecessor blocks

3.2.3 Optimizations

Our emulator applies relatively light optimizations for the sake of quick compilation speed. Other than dead-code elimination mentioned earlier, it applies several strength reductions. There are multiple rules that the recompiler checks for to best prepare instructions for the host architecture.

For example, some RISC-V instructions like `add` have variants that can operate with an immediate, in this case `addi`. Due to limited encoding space, immediates in RISC-V are in most cases 12-bits and signed. This means that, the strength reduction seen in **Figure 3.14** is possible, since the immediate is in the 12-bit signed boundary.

```
t0 ← ...
t1 ← 1234
t2 ← t0 + t1
```

```
t0 ← ...
t2 ← t0 + 1234
```

Figure 3.14: Replacing an add with an addi, reducing instruction count and register pressure

Other strength reductions include finding identity operations, such as `x + 0` or `y * 1`. These operations don't change the value of what they are operating on, and can thus be converted to a copy,

that can be eliminated by copy propagation. Strength reduction rules are also responsible for detecting variables whose value is known at compile time and folding them. For example, if we have the operation `x + 1234` and we know at that point in the code `x` has the value of `4321` the entire expression can be folded to `5555`.

Sometimes certain combinations of instructions can be combined on specific hardware that has the necessary extensions. For example, on RISC-V hardware with the `B` extension we can combine an addition with a shifted operand with the `sh1add, sh2add, sh3add` instructions, which is especially helpful for emulating effective addresses.

Another optimization we perform on SSA form is common subexpression elimination. This optimization scans the IR for expressions that have been previously calculated and replaces them with copies. In order to minimize potential performance penalties, this optimization only happens locally to a basic block.

Since optimizations sometimes produce redundant copies, our emulator also performs copy propagation. Copy propagation removes copy operations by propagating the right hand side of an instruction to the uses of the left hand side. For example, in SSA form, the instruction `t1 ← t0` can be removed as long as uses of `t1` are replaced with `t0`.

These optimizations happen until fixpoint, which means in a loop until the program can't be optimized anymore. Some optimizations enable new optimizations so several passes may be necessary.

The emulator also takes advantage of a specific machine-dependent opportunity. If there's immediates close to each other that satisfy the following rule: $-2048 \leq \|x - y\| \leq 2047$, the second immediate can be replaced with an `addi` instruction using the first immediate and a 12-bit signed integer. This is sometimes beneficial because in RISC-V, loading large 64-bit immediates to a register can take several instructions, while in x86-64 it only takes one instruction. Thus an optimization like that can reduce code size in RISC-V. However, it also increases the interferences of the first immediate so benchmarking needs to be performed to verify if such an optimization is worth keeping around.

Finally, before exiting SSA form the recompiler makes sure to break *critical edges*. A critical edge is one that starts at a block with multiple successors and leads to a block with multiple predecessors. Keeping these around when the phi instructions need to be broken is wasteful and sometimes incorrect and can lead to buggy behavior. Consider **Figure 3.15**.

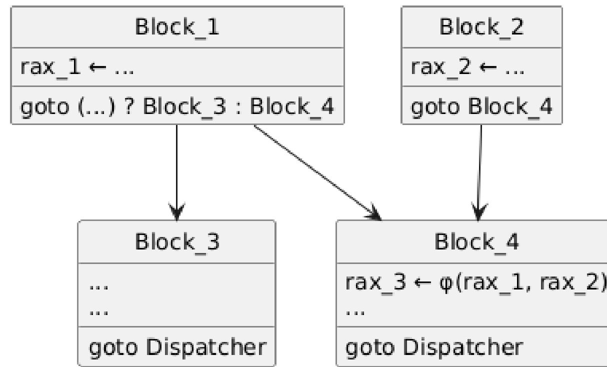

Figure 3.15: A critical edge from Block_1 to Block_4

Naively breaking the phi instruction here will lead to a `rax_3 ← rax_1` instruction in `Block_1`. However, this instruction would be redundant if execution flows from `Block_1` to `Block_3`. This contributes to unnecessary register pressure. Blocks usually require multiple phi instructions too, which exacerbates the problem. Critical edges can be split by inserting a block between them. This is demonstrated in **Figure 3.16**. This way when breaking the phi the copy will be inserted in `Block_5` and will no longer be applying register pressure.

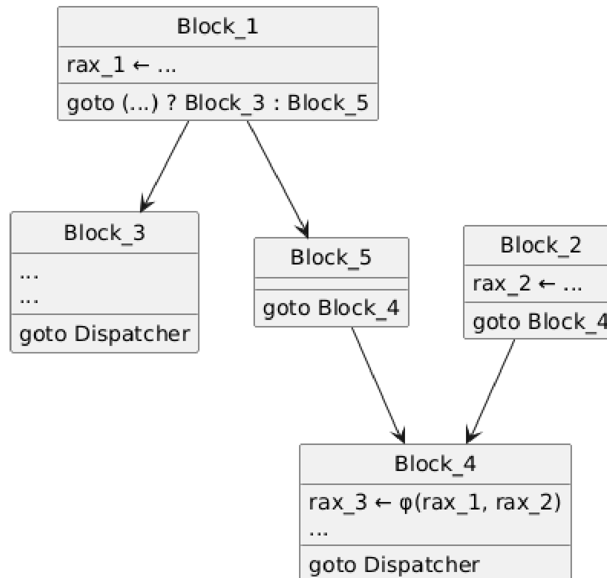

Figure 3.16: The critical edge has been broken

3.2.4 Register Allocation

Graph Coloring

Graph coloring register allocation was picked initially for register allocation. It can produce better results while being slower than the alternative, a linear scan algorithm. However, as it was later realized, graph coloring register allocation was determined to be too slow to be the main allocation strategy. Nevertheless, the implementation is kept around to use in tiered recompilation in the future.

Specifically, the Briggs et al. [24] algorithm was chosen. Its general structure is showcased in **Figure 3.17**.

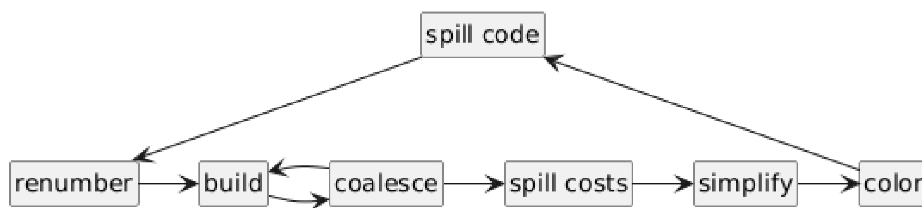

Figure 3.17: The structure of a Briggs-styled register allocator

The *renumber* and *build* stages involve calculating the live ranges of the variables across blocks and creating an interference graph. First, liveness information is collected at the boundaries of basic blocks using data-flow analysis. With it, it's simple to calculate liveness in points inside the basic blocks. Once liveness has been calculated for every point in the graph we can map out the interferences using a two-dimensional array where each dimension size is the count of variables in the graph.

The *coalesce* step is a form of copy propagation. Unlike regular copy propagation it is done after the build stage, because it uses interference information to deduce whether eliminating a copy will produce a worse graph – one that will be harder to color. Whenever two variables are coalesced the resulting variable has the combined live range of both variables, so we need to verify that it won't lead to excessive spilling. This is done via the use of heuristics. There's several heuristics for this purpose, such as ones that aggressively coalesce variables, or ones that do more conservative coalescing in order to preserve the integrity of the graph. We want to use a conservative coalescing algorithm to avoid having to recreate the interference graph more times than necessary. George et al. [34] propose such an algorithm that fits our case. This heuristic suggests that coalescing X and Y will cause no harm to our graph as long as for each neighbor T of Y either:

- degree of T is less than the number of available registers, or
- T interferes with X

During coalescing we coalesce each node that matches our heuristic. If we coalesced at least once, we need to rebuild the interference graph.

Up next comes calculating the *spill costs*. This calculation is done in order to be able to select the best instruction for spilling. A good way of calculating spill cost is counting the number of definitions and uses of a variable and dividing it by the number of interferences it has. The smaller the value, the more likely to be spilled.

This way, if an instruction has few uses but many interferences it is a better candidate for spilling. Likewise, if an instruction has a lot of uses and not as many interferences it should be kept in a register and not spilled.

Simplification follows, and this step involves removing nodes that have less interferences than the number of available registers from the graph and pushing them to a stack, as those can be trivially colored. After removing a node its neighboring nodes are going to have fewer interferences so the check is repeated until there is no more nodes that fit the rule.

If the graph is not empty, the algorithm picks a node using a heuristic and removes it – pushing it to the stack. If it results in some nodes having less interferences than the number of available registers, we go back to the previous step. Otherwise we keep removing nodes and pushing them to the stack until the graph is empty.

Once the graph is empty we can attempt to *color* it. This involves going through the nodes in the stack and assigning them a register that is not used by any neighbor. If a node can't be colored, we continue with the other nodes. At the end, if any nodes are uncolored that means one of them must be spilled. We can use our calculated spill costs from before to choose the best candidate.

Inserting *spill code* involves going to each place our spilled variable is assigned and storing the new value to memory right after. Then, each use of the spilled variable needs to load it from memory first. This essentially splits the live range of the spilled variable to only the places it is used, reducing interferences and making the graph more easily colorable.

Once we spill our variable, the process needs to be repeated until the graph is fully colorable after simplification. At that point every instruction has registers allocated to it and we are free to move on to code emission. This process needs to be repeated for SIMD instructions, as they use a different set of registers. At that point, the graph coloring register allocator has successfully allocated registers to all the variables, spilling whenever necessary.

Linear Scan

Linear scan register allocation was also implemented as an alternative. Graph coloring will be used in the future for tiered recompilation, in order to better allocate registers for hot code in a background thread.

Our linear scan implementation follows the process described in the original paper [27]. First, we reuse the liveness analysis algorithm to calculate variables live on block entries and exits. Then the blocks are linearized, so that we have a straight-line sequence of instructions that lacks control-flow. Since we know the variables that were live on each block entry and exit, the order of the blocks doesn't matter for correctness. To induce less spilling however, the blocks are laid out in reverse post-order.

With a flat array of instructions we then need to calculate liveness intervals for each variable. A liveness interval starts at the variable's first definition and ends at its last use. Note that while variables may have holes in their liveness – places where their value is not used before being redefined – their liveness interval does not have holes. This would mean that the variable is live in places where it shouldn't be, which is one of the tradeoffs of this faster algorithm.

After liveness intervals have been computed, the algorithm can begin. A loop goes over the intervals, sorted in ascending order of starting position. Registers are allocated to each interval, and the intervals are added to an active interval array. At the start of the loop the currently active intervals are checked to see if they have expired, if any of them have an endpoint that is before the start of the current interval, and if so they are removed from the active array and the register becomes free to use again.

If at any point an interval needs to be allocated but none are available an interval needs to be spilled. We follow the original implementation and spill either the currently iterated interval or the last interval in the active array. Different heuristics using interval weight are also possible.

Static allocation

An even quicker way of allocating registers is if part or all of them are allocated statically. In our linear scan and graph coloring implementations we allocate the thread state pointer statically due to its frequent use. In our case however it's possible to allocate all the general purpose registers to RISC-V registers statically, since RISC-V has double the registers. Extra registers are usually needed to store temporaries, since the x86-64 instruction set is more complex in nature. Temporaries can be thus allocated using the more complex methods, while the frequently used registers are statically allocated.

Early tests in our implementations that use optimizations and register allocation had significant compilation overheads as we'll see in the next chapter. For this reason, a second implementation of the recompiler was made which employs almost no optimizations and static register allocation.

Code emission

Once the IR has been optimized and each variable is allocated to host registers or spill locations RISC-V code can be emitted. We use a library called *biscuit* [35] due to its completeness of the RISC-V set of extensions and simplicity. Most IR instructions directly translate to a RISC-V instruction. Some can optionally check if an extension is available and produce a better instruction.

Code gets emitted into a code cache and a pointer to the start of the function is returned. Guest pointers are also mapped to the host code pointer so the dispatcher can know where to jump. At the end of each function a jump back to the dispatcher is inserted.

The code cache may get occasionally full. Tracking and removing individual function entries that are not used is inefficient as it leads to memory fragmentation, and defragmentation can also be inefficient. Our recompiler does what a lot of recompilers do when the code cache is filled – it clears it along with the mappings.

3.2.5 RISC-V concerns

Translating from one architecture to another always has its difficulties due to the differences between the two.

Differences in SIMD operations

The way **Single Instruction Multiple Data (SIMD)** operations are implemented is vastly different between the two architectures. x86-64 has different instructions for different element access patterns. For example, the `addss` instruction does addition on a single 32-bit floating-point element, while a different instruction `addsd` does addition on a 64-bit floating-point element. Each of those has a packed instruction variant, `addps` and `addpd`, which operate on multiple 32-bit or 64-bit floating-point elements. This pattern repeats for most floating-point operations.

In RISC-V, things are different. In the case of floating-point addition, only one instruction exists: `vfadd`. To be able to operate with different element sizes and number of elements, RISC-V has a global state configurable via specific instructions. Most importantly, this global state defines the size of elements, the amount of elements each instruction operates on, whether or not it should use multiple registers grouped together and other less frequently used details. Instructions like `vsetivli` can configure this state. Additionally, masking can be used to only operate on part of a vector register. The mask is taken from the vector register `v0`, and most instructions can be encoded as a masked variant. An example conversion of the x86-64 instruction `mulpd` to RISC-V is shown in **Figure**

3.18. This conversion presupposes that `xmm0` is allocated to `v1` and `xmm1` is allocated to `v2`.

```
mulpd xmm0, xmm1                  vsetivli   zero, 2, e64, m1, tu, mu
                                  vfmul.vv   v1, v1, v2
```

Figure 3.18: Conversion of an x86-64 SIMD operation to RISC-V

The `vsetivli` instruction shown here sets the element count to 2 elements, since the x86-64 vector registers before AVX are 128-bit. It also sets the element width to 64-bit. The group multiplier is set to 1 to not use register grouping and the tail and mask policy are set to undisturbed. This means that any elements that are tail elements, meaning ones past the element count, or masked by the mask register are unchanged by the operation. Since we operate on the entire register this setting is not as important but comes useful in some scenarios where we want the unused bits to remain undisturbed.

Inserting a `vsetivli` instruction before every single vector instruction would be wasteful. For this reason, each IR vector instruction is marked with a *VectorState* variable. This takes the form of common element count and size combinations, such as `PackedWord` or `Float`. A later compilation step examines these fields and inserts `vsetivli` instructions only when necessary, so that multiple instructions can benefit from the same global state before it is changed.

While RISC-V has separate floating-point and vector registers and instructions, any FPU operation is done on the same registers as the SIMD registers on modern x86-64, as the x87 registers are rarely used. The FPU registers are not utilized. This is because switching context between FPU and vector registers would be costly at compilation time and runtime.

Unaligned vector memory access

Unaligned vector memory accesses trigger the `SIGBUS` exception. While it's possible to use 8-bit element sizes for every load and store, it may be suboptimal in some hardware. So, 64-bit and 32-bit element sizes are used, and a signal handler is installed to capture unaligned memory errors.

We ensure that each vector memory access is preceded with a `vsetivli` instruction or a `nop`. Then, in the signal handler we change this preceding instruction with a `vsetivli` that sets element width to 8-bits and an appropriate length. Each access is also followed by another `nop` instruction so we can insert another `vsetivli` which restores the vector state to what existed previously at that point, as further instructions may rely on this vector state.

Lack of 80-bit floating-point registers

The x87 instructions are rarely used, but they are available and need to be emulated. x87 makes available a feature that is rare across architectures, that is use of 80-bits for floating-point operations. For simplicity, these calculations are performed on the previously unused 64-bit FPU registers. This sacrifices some precision for much better performance, as the alternative would be emulating 80-bit operations in software. In the future, an optional high-accuracy mode could be added to opt for software emulation.

Lack of 8-bit, 16-bit and 128-bit atomics

The base `A` extension introduces 32-bit and 64-bit atomic support. While this is great for most operations one might encounter in a program, some x86-64 programs use 8-bit and 16-bit atomics, and some use 128-bit compare exchange via the `cmpxchg16b` instruction. With the 32-bit load-linked/store-conditional instructions that are available we can emulate 8-bit and 16-bit atomics. An example of this is shown in **Figure 3.19**. A RISC-V extension named `Zabha` is going to introduce 8-bit and 16-bit atomic instructions, however until then software emulation is our only choice.

```
lock add byte[rdi], 1
```

```
andi    a4,a0,3
slli    a4,a4,3
li      a5,255
sll     a5,a5,a4
li      a3,1
sll     a3,a3,a4
andi    a0,a0,-4
not     a4,a5
loop:
lr.w.aqrl a2, 0(a0)
add     a1, a2, a3
and     a1, a1, a5
and     a6, a2, a4
or      a6, a6, a1
sc.w.rl a1, a6, 0(a0)
bnez    a1, loop
```

Figure 3.19: The price we pay for 8-bit and 16-bit atomics

In this example, the address stored in `a0` is aligned to the 32-bit boundary. A mask is created based on which byte inside this 32-bit number we need to operate on. Then, inside the `ll/sc` loop the address is continuously read, the addition happens, the result is masked, and the unused bits are or'ed together with the result to produce a resulting 32-bit number that only operates on the relevant 8-bits.

Finally, `sc` makes an attempt to write the data. If the address has been overwritten before the loop is finished it needs to repeat until it is no longer the case. This results in correct emulation but greater overhead and register pressure, which `Zabha` will eventually alleviate.

Although emulating 8-bit and 16-bit operations is possible, 128-bit compare and exchange cannot be emulated with 32-bit or 64-bit operations. Until 128-bit atomics are introduced, such as 128-bit compare and exchange which is introduced in the `Zacas` extension, the only option would be to lock every thread on every memory access upon encountering a 128-bit atomic operation, which would be prohibitively slow. For this reason, the atomicity of `cmpxchg16b` cannot be currently respected in our emulator.

Total Store Ordering

x86-64 has a strong memory consistency model. It ensures that stores retain their order across different cores, which means that stores cannot be reordered after stores. This is not true in RISC-V's weak memory model. Not only can stores be reordered after stores, but any combination of loads/stores can be reordered as long as the behavior in a single core remains the same.

A RISC-V extension exists that disallows this reordering and only allows stores to be reordered after loads, just like in x86-64. This extension is called `Ztso`, however, it is not available on any publicly available RISC-V hardware at this time.

If total store ordering is not emulated, bugs can occur on multi-threaded code. While it is rare any individual store would rely on total store ordering, in x86-64 it is guaranteed for every store, so we cannot know which ones rely on it and which do not. Thus, in order to emulate total store ordering, memory fences would need to surround every load and store operation. This would lead to big performance losses. Additionally, other emulators such as Box64 also choose not to emulate TSO and rarely run into problems. For this reason, we choose to address this problem at a later time.

3.3 Emulation

CPU emulation is the predominant amount of work that goes into a user-space emulator. This is because we don't need to emulate other devices, peripherals or memory management, as those are provided by the host. For example, a syscall that communicates with the GPU is going to communicate directly with the host GPU, not an emulated GPU, like it is the case with many non user-space emulators. However, some components of the Linux operating system need to be emulated for compatibility.

3.3.1 Syscalls

Most syscalls can be trivially emulated. Syscall ids are different between x86-64 and RISC-V, so a translation is necessary. Some syscalls that exist on x86-64 do not exist on RISC-V. This could be the case due to them being depracated, and since x86-64 Linux needs to maintain backwards compatibility they remain available for programs to use. For example, the `open` syscall exists on x86-64 Linux but not on RISC-V Linux, as it was replaced with the more powerful `openat` syscall. x86-64 has both `open` and `openat` so newer programs can use the better one, however `open` needs to remain and old software may still use it. Syscalls like `open` need to be emulated in software, usually by calling the newer variety. Keep in mind that any syscall that interacts with the filesystem also needs its path sanitized so that it can only access files inside the root filesystem – if a path tries to ”escape” by having multiple double dots it should not be able to reach to folders on the host filesystem.

Some syscalls may also have slightly different arguments, which makes the arguments not trivially passable to host syscalls. The `uname` syscall for example has an argument which is a pointer to a struct called `utsname`. This struct can have slight differences between architectures, so the data needs to be copied from a host struct to the guest struct. Additionally, some of the data is filled by our emulator, such as the `utsname->machine` field which is set to ”x86-64”.

Other syscalls only exist on x86-64 because they are architecture specific to x86-64. One example is `arch_prctl`, which can do a few things such as setting the values of `fsbase` and `gsbase` or disabling `cpuid` instruction access. This syscall only exists in x86-64 and its functionality needs to be emulated in RISC-V.

3.3.2 Signals

In many non-trivial programs custom signal handlers are used for a variety of reasons. A signal handler is a piece of user-space code that executes upon a signal. Signals can be divided in two categories:

- Synchronous signals, which are the direct result of a process's actions
- Asynchronous signals, which are invoked by an external event or a different process

An example of a synchronous signal is `SIGFPE`, triggered by a mathematical error such as dividing by zero. It could also be a `SIGBUS` by accessing invalid memory locations.

Asynchronous signals include `SIGINT`, which could result by hitting `Ctrl+C` on a terminal window, or delivered by an outside process such as `SIGCHLD` or user-defined signals.

Signals pose one of the greatest challenges in Linux user-space emulation. Since asynchronous signals can happen at any point of executions, it may be the case they happen in the middle of block execution. Since a signal handler may modify guest state, reconstruction of the guest state needs to happen before returning to the point of execution. This is not trivial, as guest registers are allocated to host registers it is not enough to modify the state struct. One solution could be to defer asynchronous signals until a ”safe” point, such as when jumping back to the dispatcher. This way signals are not going to happen in the middle of recompiled code execution. Static register allocation makes state reconstruction easier, as we know at any point in time which host register holds a guest register value, and we can update it in the signal handler.

We also need to handle interference with signals that are meant for our emulator and not for the guest application. In any case, signals need to first go through our own handler as the signal handler the application sets contains guest machine code.

Currently, our emulator does not handle signals, but it will be something we have to tackle once we try to emulate more complex applications.

3.4 Conclusion

Two implementations of the recompiler were made for our emulator. One which converts code to SSA form and allocates registers using graph coloring and linear scan, and one that directly converts x86-64 code to RISC-V, applying very minimal optimizations such as not emitting unnecessary flag calculations and enforcing static register allocation. On the following chapter we will see the runtime performance and compilation overhead of both implementations with several programs.

Chapter 4

Results

The emulator was compiled and tested on a RISC-V machine with the vector and bit-manipulation extensions running a variation of Ubuntu.

Initial results showed that our implementation of graph coloring register allocation on big chunks of code exponentially increased compilation times. On some programs, 99% of the total execution time was spent on the register allocation alone. Some functions with hundreds of basic blocks could take in the order of multiple seconds to compile. Additionally, a lot of the paths that were getting compiled ended up not being used.

Two different approaches were employed to tackle this. The first one was limiting the basic block count that can exist in a function. This vastly reduced compilation times at the expense of more frequent jumps to the dispatcher, which degrades runtime performance. The second approach was a linear scan implementation. This proved to be much faster, even when dealing with multiple blocks.

A few different programs were used to benchmark the emulator's performance. The `prime` program aims to find the 10,000th prime with a naive primality test. The `pi` program approximates the value of π using the Monte Carlo method, employing a total of 5,000,000 randomly generated points. The `ls` program is one present in every Linux system and can be used to list the contents of the current directory. To observe the impact of syscall emulation, the `syscall` program executes the `time` function from `time.h`, which in turn calls the `time` syscall, 10 million times. Finally, `mem` copies 1GB of data from one pointer to another using `memcpy`.

Our first iteration of benchmark tests these programs with a fixed basic block upper limit of 15 blocks per function using graph coloring register allocation. **Figure 4.1** shows the execution time in seconds each program took under our emulator, QEMU, and native execution when the program was originally compiled for RISC-V.

Our emulator performs worse than QEMU on most measurements except on the `syscall` program, where it seems to be doing significantly better. Compilation times ranged from 2 to 4 seconds of total execution time in each case. The `pi` program performs the worst, taking three times as much time.

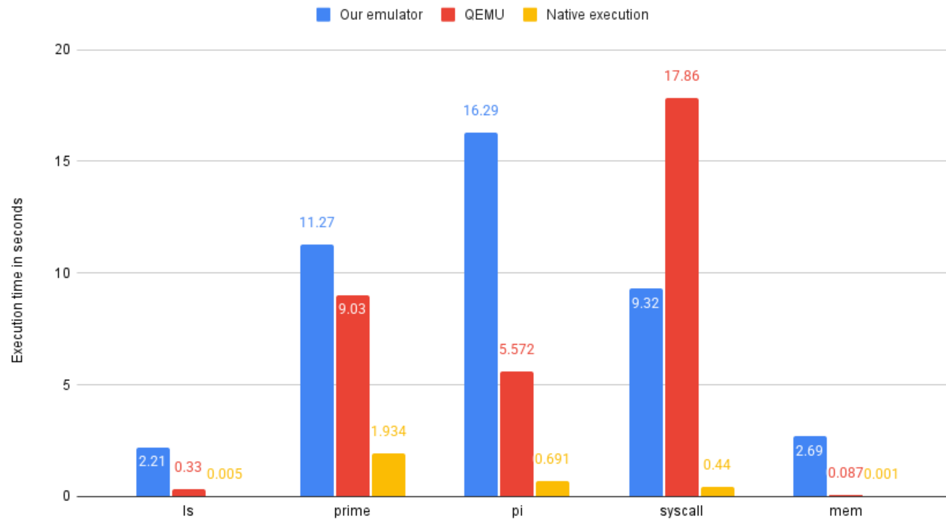

Figure 4.1: Results with graph coloring register allocation and an upper limit of 15 blocks

To test the effects of block count on compilation time, different block counts were tested with the `prime` program. This can be seen in **Figure 4.2**.

While higher block count usually correlates with better host code, it also correlates with higher compilation speeds. For example, we can see that performance actually worsened going from a limit of 4 blocks to a limit of 5, due to the benefits not being enough to outperform the increase in compilation time. Additionally, on higher block counts the execution time should go back up due to slow compilation.

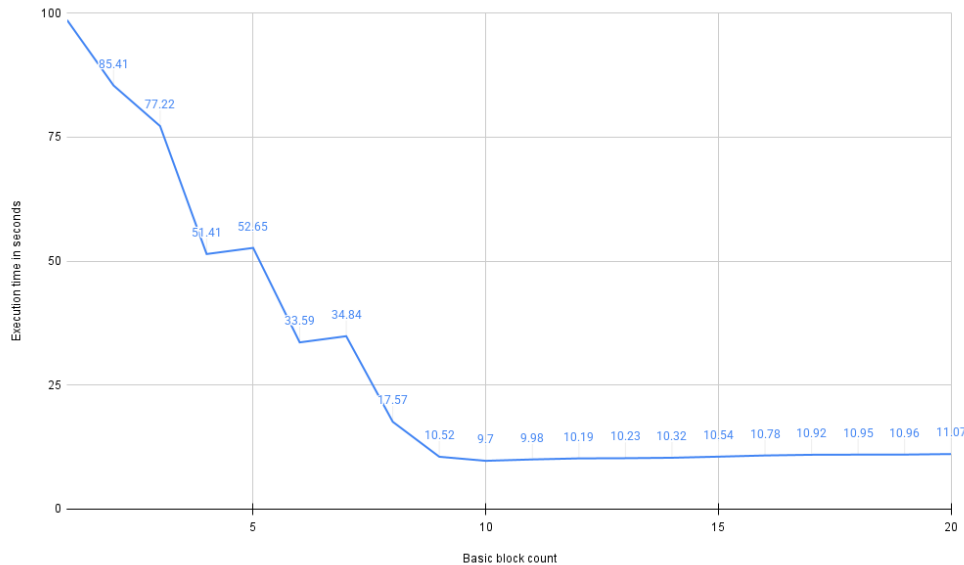

Figure 4.2: Execution time goes down when compiling more blocks at a time

The same benchmark binaries were tried with a linear scan implementation, at a fixed block boundary of 10 blocks. The results are seen in **Figure 4.3**. Total execution time seems to have been reduced by a few seconds in each case due to faster compilation speeds.

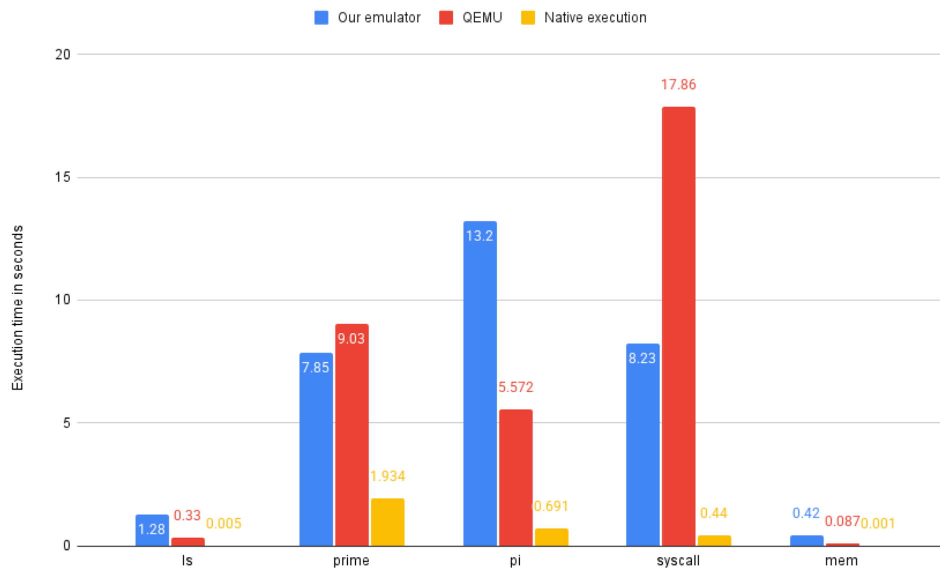

Figure 4.3: Results with linear scan register allocation and an upper limit of 10 blocks

Our optimizations did not seem to improve performance for the `prime` program. Execution times

stayed relatively the same, as seen in **Figure 4.4**.

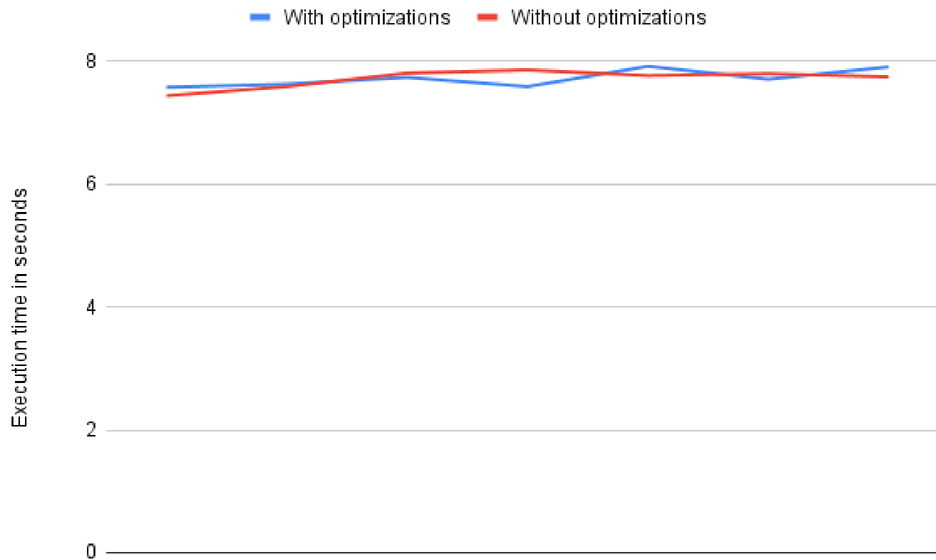

Figure 4.4: Execution times were not affected by our optimizations

If we subtract compilation time from our results, the difference in execution time between graph coloring and linear scan is vastly reduced. This can be seen in **Figure 4.5**.

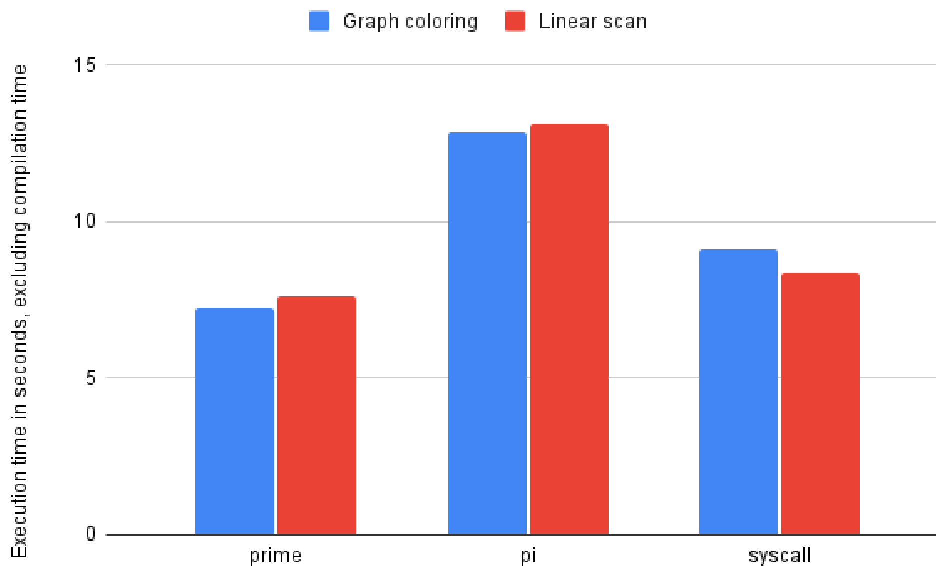

Figure 4.5: Graph coloring produced slightly better code

Our second implementation, which uses static register allocation on basic block level with block link-

ing, was compared to linear scan with a limit of 10 basic blocks. Three different programs were tried, that compute the 10,000th, 20,000th and 30,000th prime number respectively. In **Figure 4.6** we can see that our linear scan algorithm performs better than static allocation, due to having the context of multiple basic blocks.

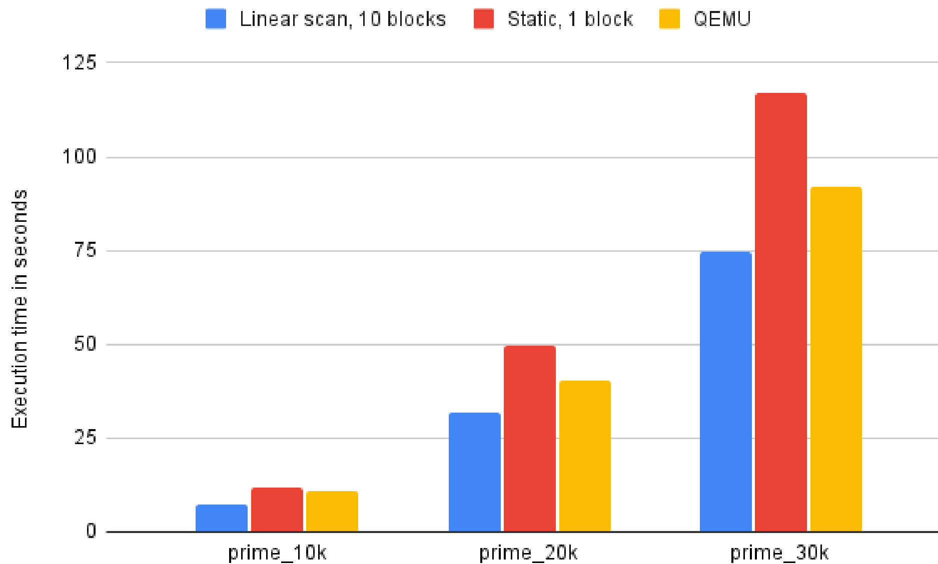

Figure 4.6: Our linear scan implementation performs better in hot loops

On the other hand, static register allocation is faster. While it doesn't perform optimizations and may produce worse code, the compile-time overhead is significantly reduced. This is seen in **Figure 4.7**. Programs like `ls`, `mem` that had significant compilation overhead can run at almost native speeds with the static register allocator implementation. Other programs like `prime` performed slightly worse. On the other hand, `pi` has performed better likely due to better SIMD instruction implementation in the second recompiler.

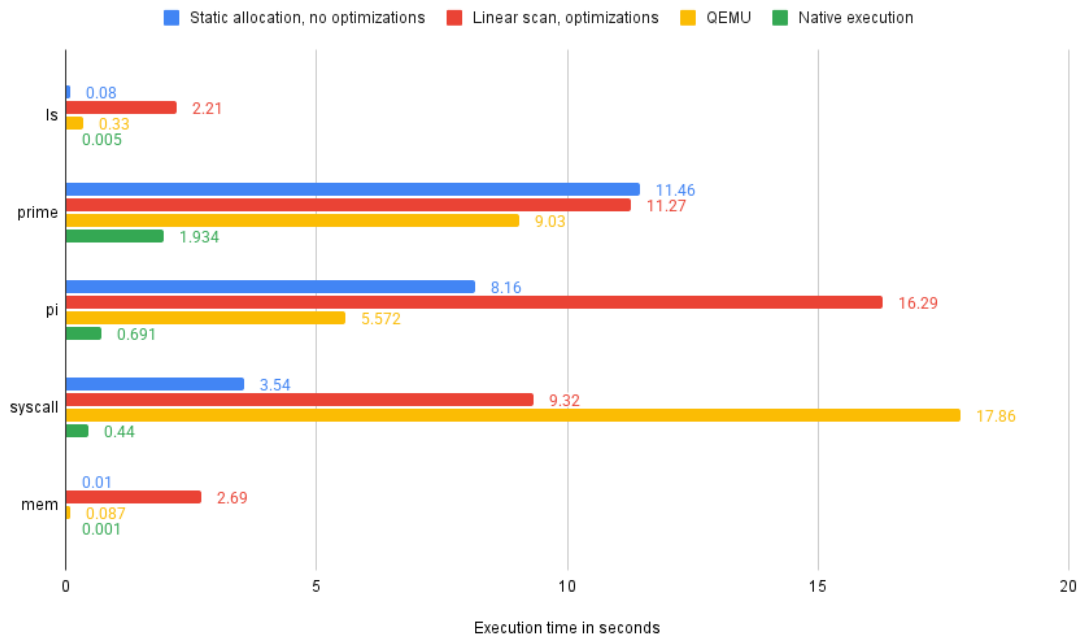

Figure 4.7: Compilation overhead is significantly reduced with static allocation and no optimizations

Multiple other programs commonly found in Linux environments were tested, such as `lua`, `gcc`, `git`, and `python3`, as well as simpler programs such as `echo`, `tar`, `pwd`, `env`, among others. The aforementioned programs all work properly to at least display the introductory help message. Program functionality seems to work as expected. In **Figure 4.8** we can see our emulator running x86-64 Lua and executing Lua code, as well as a script that computes the Fibonacci sequence for 15 iterations.

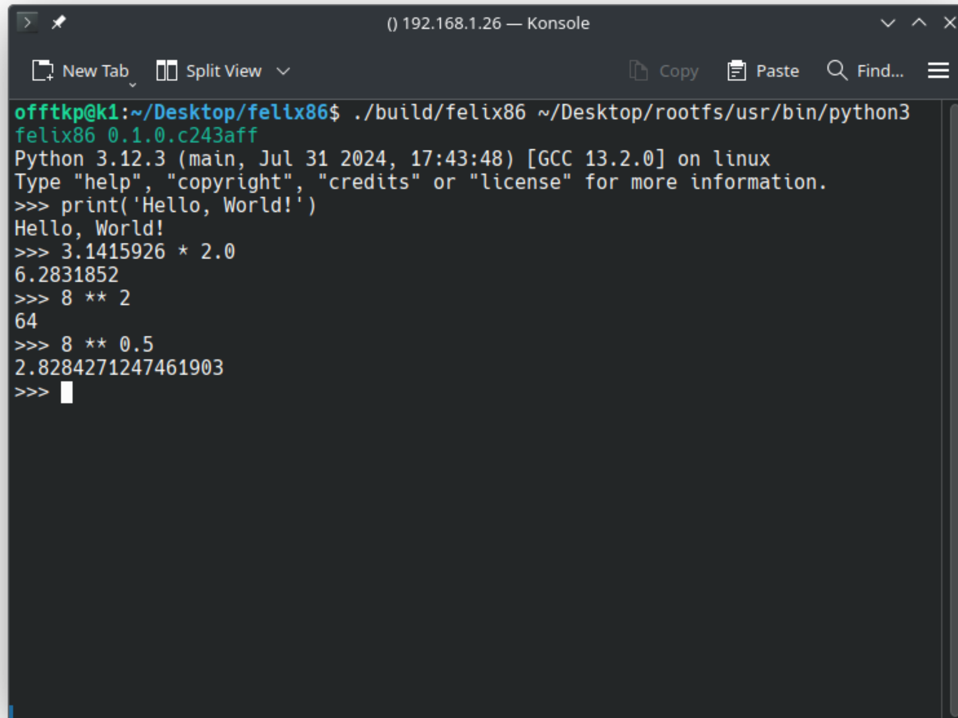

Figure 4.8: Emulating python3 and executing Python code

Chapter 5

Discussion

In this emulator, we attempted to perform heavier optimizations and register allocation techniques across multiple blocks. We found quickly that massively allocating all blocks we could collect in a control-flow graph would explode compilation times with graph coloring. An approach to generate simpler CFGs could be explored, for example *Extended Basic Blocks*, a collection of blocks where only the entry block has multiple predecessors. This would reduce our graphs to ones that we can perform quicker liveness analysis on.

Arbitrary block limits per control flow graph managed to reduce compilation times to acceptable times. Graph coloring was able to produce slightly faster execution times in programs with hot loops like `prime` and `pi`, but was heavily offset by its bigger compilation time. In **Figure 4.5** we can see that up to a second total was saved. This makes our graph coloring implementation not worth the compilation times, which were in the order of up to four seconds total. Ways to reduce the compilation time could be explored, such as [26], however they would still have to be performed in a reduced version of our control flow graph.

Slow compilation times from heavier algorithms were expected, however not to this level. For this reason, the second implementation of the recompiler became the primary strategy, building on quicker algorithms such as static register allocation and applying very minimal optimizations. Our straightforward approach with the newer implementation allowed complex programs such as the `lua` interpreter to start execution immediately, something which would take at least a few seconds even for some of the simplest programs.

In **Figure 4.4** we saw that our optimizations, such as dead code elimination, copy propagation and local common subexpression elimination did not seem to affect execution times. Most of the binaries we emulate are already heavily optimized by a compiler already, so not a lot of opportunities for optimization occur, if any. While these optimizations may help us produce better machine code, it

did not seem to matter in our implementation.

The performance degradation seen in low block counts in **Figure 4.2** are caused by the recompiled code exiting to the dispatcher too often. In hot loops like the ones in the `prime` benchmark the impact of exiting recompiled code is exaggerated, as every loop iteration has to perform a lookup to find the next block. In most emulators, this is mitigated by ”block linking”, which is hardcoding the jump to the next block whenever possible, to avoid exiting to the dispatcher and having to do a lookup to match the guest address to the recompiled code address. In our first implementation, this functionality does not exist and, after every function, execution returns to the dispatcher. While this isn't harmful to performance if functions have multiple blocks, when compiling and running a single block at a time the performance penalty of this adds up quickly.

Our second implementation works on the basic block level, but implements block linking. This eliminates the significant overhead our first implementation suffered from, as the vast majority of execution time in hot loop programs like `prime` is spent executing recompiled code, and no jumps to the dispatcher are made. While this performs way better compared to single block linear scan without block linking, it performs worse than linear scan with an upper limit of 10 blocks. This is seen in **Figure 4.6**. This opens the door for tiered recompilation, where code gets profiled, and hot code is heavily optimized. This would in theory combine the benefits of fast recompilation for rarely used code, and slower recompilation but better runtime performance for frequently used code. In multi-core CPUs, this could be offloaded to a different core which would allow the main thread to continue execution and smoothly integrate with the better code when compilation is finished. For this reason, graph coloring could be reconsidered. Since the compilation overhead would not affect our main thread, we should be able to enjoy the better allocations that are seen in **Figure 4.5** without affecting the execution time.

Compared to QEMU, our first implementation again suffered from slow start-up times. Our second implementation eliminated this problem, however as seen in **Figure 4.7** it still doesn't have better execution time than QEMU for programs with hot code. This is likely due to suboptimal code generation on our part, such as more state loads and stores than necessary. On the other hand, the second implementation performed significantly better than our heavier allocation schemes in the `pi` program. Likely cause for this is better vector instruction implementation or better vector state management.

Due to the quicker start-up time, the second implementation is more user-friendly and useful during debugging. Direct emission of host code during machine code interpretation also gives us finer control of the code we emit and the registers we use. Using a single block at a time also makes the program easier to debug. We can insert breakpoints on individual blocks based on their guest address, something which would be less trivial when compiling entire functions.

Both implementations seem to perform a lot better on programs that make heavy usage of syscalls.

All syscalls are currently handled by a function which the recompiler emits a jump to. In the future, the syscall id could be detected during recompilation and an `EBREAK` instruction could be used. This should make syscall emulation slightly faster due to better instruction cache usage, as the code doesn't have to jump to a different function.

Chapter 6

Conclusion

In this paper, we went through the development process of Felix86, a user-space x86-64 Linux emulator. We evaluated algorithms and techniques commonly used in compilers. We demonstrated differences between the host and target architecture, as well as the challenges they caused. We discussed the x86-64 and RISC-V architectures, the differences between them, our approach to emulating the CPU and the Linux environment.

We designed and implemented two different recompilers with different benefits. Our first recompiler was slow to compile, but was able to produce good code that took advantage of multiple basic blocks to make optimizations and allocate registers. On the other hand, our second recompiler produced code a lot quicker, but the code quality slightly lessened. Static register allocation allowed it to not have to make runtime decisions using heavier algorithms, which provided good run-time performance and even greater compile-time performance.

In the future, multi-threaded applications need to be explored, as well as those with a graphical user interface. More complex syscalls such as `clone3` need to be emulated to enable this. Additionally, different strategies to speed up emulation need to be experimented with, such as library thunking and tiered recompilation.

Overall, Felix86 demonstrates the feasibility of a user-space emulator for cross-architecture emulation, but further research and development are essential. While further research and development are needed to optimize performance and scalability, this project has been a great learning experience in the topics of emulation, the x86-64 and RISC-V architectures and recompilers.

Bibliography

[1] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W.-m. W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24, Albuquerque, New Mexico, Puerto Rico: Association for Computing Machinery, 1991, pp. 25–33, ISBN: 0897914600. DOI: `10.1145/123465.123471`. [Online]. Available: `https://doi.org/10.1145/123465.123471`.

[2] G. Savrun-Yeniçeri, W. Zhang, H. Zhang, *et al.*, "Efficient hosted interpreters on the jvm," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, Feb. 2014, ISSN: 1544-3566. DOI: `10.1145/2532642`. [Online]. Available: `https://doi.org/10.1145/2532642`.

[3] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970, ISSN: 0362-1340. DOI: `10.1145/390013.808479`. [Online]. Available: `https://doi.org/10.1145/390013.808479`.

[4] *Qemu*, `https://github.com/qemu/qemu`.

[5] *Fex*, `https://github.com/FEX-Emu/FEX`.

[6] *Box86*, `https://github.com/ptitSeb/box86`.

[7] *Box64*, `https://github.com/ptitSeb/box64`.

[8] J. Dougall, *Why is rosetta 2 fast?* Blog post on Dougall J's blog, 2022. [Online]. Available: `https://dougallj.wordpress.com/2022/11/09/why-is-rosetta-2-fast/`.

[9] B. Hong, "Sleight of arm: Demystifying intel houdini," in *DEF CON 29 Presentations*, 2021.

[10] ptitSeb, *Box86/box64 vs qemu vs fex (vs rosetta2)*, Accessed: 2025-01-12, 2022. [Online]. Available: `https://box86.org/2022/03/box86-box64-vs-qemu-vs-fex-vs-rosetta2/`.

[11] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, 1970, pp. 1–19.

[12] G. A. Kildall, "Global data flow analysis and iterative algorithms," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 97–119, 1973.

[13] B. Rosen, M. Wegman, and F. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88, New York, NY, USA: ACM Press, 1988, pp. 12–27.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," pp. 451–490, 1989.

[15] J. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '91, New York, NY, USA: ACM, 1991, pp. 55–66. DOI: 10.1145/99583.99594. [Online]. Available: https://doi.org/10.1145/99583.99594.

[16] P. Briggs, K. Cooper, T. Harvey, and L. Simpson, "Practical improvements to the construction and destruction of static single assignment form," *Software: Practice and Experience*, 1998.

[17] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form."

[18] G. A. Kildall, "A unified approach to global program optimization," pp. 194–206, 1973. DOI: 10.1145/512927.512945.

[19] J. Cocke and K. Kennedy, "An algorithm for reduction of operator strength," *Communications of the ACM*, 1977.

[20] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," *Communications of the ACM*, vol. 22, no. 2, pp. 96–103, 1979. DOI: 10.1145/359060.359069.

[21] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991. DOI: 10.1145/103135.103136.

[22] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981. DOI: 10.1016/0096-0551(81)90048-5.

[23] G. J. Chaitin, "Register Allocation & Spilling via Graph Coloring," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, ACM, 1982, pp. 98–105, ISBN: 0-89791-074-5. DOI: 10.1145/800230.806984.

[24] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 428–455, 1994. DOI: 10.1145/177492.177575.

[25] S. Hack, D. Grund, and G. Goos, "Register allocation for programs in ssa-form," in *Compiler Construction, 15th International Conference, CC 2006*, ser. Lecture Notes in Computer Science, vol. 3923, Springer, 2006, pp. 247–262. DOI: 10.1007/11688839_20. [Online]. Available: https://doi.org/10.1007/11688839_20.

[26] K. D. Cooper and A. Dasgupta, "Tailoring graph-coloring register allocation for runtime compilation," in *Proceedings of the 2006 International Symposium on Code Generation and Optimization (CGO)*, New York, NY, USA: IEEE, 2006. DOI: 10.1109/CGO.2006.35. [Online]. Available: https://doi.org/10.1109/CGO.2006.35.

[27] M. Poletto and V. Sarkar, "Linear scan register allocation," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA: ACM, 1999, pp. 246–257. DOI: 10.1145/330249.330250. [Online]. Available: https://doi.org/10.1145/330249.330250.

[28] O. Traub, G. Holloway, and M. D. Smith, "Quality and speed in linear-scan register allocation," *SIGPLAN Not.*, vol. 33, no. 5, pp. 142–151, May 1998, ISSN: 0362-1340. DOI: 10.1145/277652.277714. [Online]. Available: https://doi.org/10.1145/277652.277714.

[29] H. Mössenböck and M. Pfeiffer, "Linear scan register allocation in the context of ssa form and register constraints," in *Compiler Construction*, R. N. Horspool, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 229–246, ISBN: 978-3-540-45937-8.

[30] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, Berkeley, CA, USA: USENIX Association, 2005.

[31] L. Baraz, T. Devor, O. Etzion, *et al.*, "IA-32 Execution Layer: A Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, Washington, DC, USA: IEEE Computer Society, 2003, pp. 191–202. DOI: 10.1109/MICRO.2003.1253187.

[32] *System v application binary interface: Amd64 architecture processor supplement*. [Online]. Available: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf.

[33] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," Rice University, Tech. Rep. TR06-38870, 2006. [Online]. Available: https://hdl.handle.net/1911/96345.

[34] L. George and A. W. Appel, "Iterated register coalescing," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, pp. 300–324, 1996. DOI: 10.1145/229542.229546. [Online]. Available: https://doi.org/10.1145/229542.229546.

[35] *Biscuit*, `https://github.com/lioncash/biscuit`.