Xuan Guo

Dynamic Binary Translator for RISC-V

Computer Science Tripos – Part II

Peterhouse

May 14, 2018

Proforma

Name:	**Xuan Guo**
College:	**Peterhouse**
Project Title:	**Dynamic Binary Translator for RISC-V**
Examination:	**Computer Science Tripos – Part II, June 2018**
Word Count:	**11394**
Project Originator:	Xuan Guo
Supervisor:	Dr Timothy Jones

Original Aims of the Project

RISC-V is an open and modular ISA. It is a recent innovation, so for the ecosystem to grow, software running on RISC-V ISA has to be developed or ported from other ISAs. Therefore, there exists a need for a fast RISC-V emulator that can run on commodity AMD64 hardware. This project aims to develop an emulator that is capable of running unmodified RISC-V Linux binaries directly on an AMD64 Linux system, and it shall be utilising dynamic binary translation techniques to achieve better performance than simple interpreters.

Work Completed

All success criteria were met, and the optional extension to support dynamically linked binaries is implemented. The emulator produced passes all test cases set, and significantly surpassed the original performance goal, achieving 30.5x speedup in Dhrystone and 18.0x in CoreMark when compared to `spike`, the reference RISC-V interpreter. When compared against QEMU, a well-known multi-platform emulator, it achieves 9.06x speedup in Dhrystone, 2.60x in CoreMark, and 2.49x in SPECint. This project applies traditional compiler optimisations to achieve high performance. As far as I am aware, no other open-source emulators do the same.

Special Difficulties

None

Declaration

I, Xuan Guo of Peterhouse, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date May 14, 2018

Contents

List of Figures

List of Algorithms

List of Tables

List of Listings

Acknowledgements

I would like to express my sincere gratitude to the following individuals. This project would not be as successful as it was without their encouragement and suggestions.

- **Dr Timothy Jones**, for supervising my project and giving invaluable advice, guidance and suggestions for the project and this dissertation.
- **Dr Robert Mullins**, for his encouragement and valuable feedbacks for this dissertation.
- **Junwei Yuan and Ranjeev Menon**, for proofreading my dissertation and giving suggestions.
- **My family and friends**, for their understanding, support and encouragement.

Chapter 1

Introduction

Over the past decade, the open-source software community has expanded rapidly. Besides individual open-source enthusiasts, numerous technology companies also contribute code or funding to the community. The hardware platforms that software runs on, however, are mostly proprietary. The RISC-V Foundation [16] is one notable exception, bringing open-source concepts to the hardware world, aiming to develop an open microprocessor architecture standard that everyone can use without royalty.

The concept about open-source hardware has attracted interest and popularity in the open-source community. However, as a new instruction set architecture (ISA), RISC-V is disadvantageous compared to established architectures like AMD64 (also called x86-64) in terms of ecosystems. Most modern desktops and servers are powered by processors running the AMD64 ISA, and these systems cannot run RISC-V binaries natively. This essentially creates a barrier for developing software for or porting software to the RISC-V ISA. For the ecosystem to grow, there exists a need for a RISC-V emulator that has both good performance and interoperability with existing environments.

Emulators usually come with huge performance penalties. Dynamic binary translation (DBT) is a technique to reduce performance overhead by translating binaries compiled for one architecture to another instead of performing interpretation, whereby each instruction is executed in software in isolation. Existing emulators either use no DBT or use a very simple DBT that translates code based on templates only. QEMU uses an intermediate representation (IR), a high-level description of each instruction that can be used to run analysis and transformations on, but in QEMU its usage is mainly for portability instead of optimisation concerns. These existing tools share very little in terms of structure and optimisation passes with traditional compilers. On the other hand, just-in-time (JIT) compilers, which compile source code down to machine code immediately before execution, are more closely related to traditional ahead-of-time compilers, using similar techniques or even sharing the same codebase (e.g. LLVM JIT, GCC's libjit). Therefore, utilising traditional compiler optimisations within a DBT emulator is an interesting area to investigate.

This project aims to develop an emulator that is capable of running unmodified RISC-V

Linux binaries directly on an AMD64 Linux system, utilising dynamic binary translation techniques and compiler optimisations to achieve better performance than simple interpreters.

There are many challenges in an ambitious project like this. This project requires a deep understanding of architecture, compilers and system programming. The goal to achieve binary compatibility with RISC-V Linux binaries mandates that all components in this project be equivalent in behaviour to a native RISC-V Linux machine. Working at the binary level also makes the project extremely difficult to debug, adding more risks to the project. The biggest challenge is the tight time budget, given that this is a Part II project. Another binary translator targeting RISC-V to AMD64, `rv8`, took around 2 years to reach its current status, and it still cannot execute many RISC-V Linux binaries. Despite all the risks, I have managed to achieve and exceed the original success criteria and beat all existing emulators by a margin. The following chapters of this dissertation will discuss the planning, design, implementation and evaluation of this project.

Chapter 2

Preparation

2.1 RISC-V

RISC-V is an open, general-purpose ISA. It began as a research project in 2010 at the University of California, Berkeley when researchers failed to find a suitable ISA to use for new research projects [4]. Commercial ISAs, such as AMD64 or ARM, are too complex in design and too restrictive due to intellectual property issues. Existing open-source ISAs often have no clear specifications and are usually released under copyleft licences, whose requirement to make modifications publicly available prevents commercial adoption. RISC-V was developed to address these issues and to support computer architecture research and education within Berkeley, but it also attracts attention from industry. In 2015, the non-for-profit RISC-V Foundation was created to maintain the stability of the ISA and its wider adoption.

RISC-V is a universal ISA. It is designed to work well with existing software stacks and languages, to suit all sizes of processor, from the smallest micro-controllers to the largest super-computers, and to allow extensive specialization for different workloads. To achieve these goals the ISA is designed with modularity and simplicity in mind, so it can be efficiently implemented for all microarchitectures and fabrication technologies.

The instruction set is divided into base integer ISAs and optional extensions. At the time of writing, stable base ISAs include RV32I and RV64I, namely 32-bit and 64-bit base integer ISAs. RV128I, the 128-bit ISA, and RV32E, a 32-bit integer ISA with a reduced number of registers for embedded micro-controllers are under development. Extensions can be vendor-specific, and there also exists standardised extensions for commonly implemented features. Current stable standard extensions include the “M” extension for integer multiplication extension and division, the “A” extension for atomic instructions, the “F” extension for single-precision floating-point, the “D” extension for double-precision floating point and the “C” extension for compressed instructions. IMAFD are collectively known as the “G” (general-purpose) ISA. General-purpose RISC-V processors intended to run Linux/Unix are usually expected to also support the compression extension to reduce the code size.

2.2 Binary Translation

To run a program compiled for a particular ISA and application binary interface (ABI) on a different platform, an emulator is needed. The architecture emulated is called the guest architecture, and the architecture that an emulator runs on is called the host architecture. An emulator can be further classified as an interpreter or a binary translator. An interpreter is an emulator that works by decoding and emulating instructions one at a time. Binary translators can achieve significant speedup by translating binaries from the guest ISA to the host ISA and executing the translated binaries, avoiding the repetitive decoding and dispatching costs in interpreters. Binary translators can be further divided into static binary translators, which translate all code ahead-of-time, or dynamic binary translators which translate code at runtime when the code is actually used.

Binary translation has a wide range of applications. Emulators such as QEMU utilise binary translation for fast emulation, and virtual machines, such as VMware, VirtualBox, are using binary translation if hardware virtualisation is unavailable. Microsoft uses binary translation to support x86 applications on its ARM version of its Windows 10 operating system. Apple used binary translation a few times to aid its architecture changes, once from M68K to PowerPC and once from PowerPC to x86. Binary translation is also useful for other purposes. Program analysis tools such as Valgrind utilise binary translation to instrument the code while retaining the original semantics and reasonable performance.

Dynamic binary translation is also a type of JIT compilation. A conventional JIT translates a compile abstract syntax tree (AST) or bytecode into native code for immediate execution, instead of producing a binary. JIT techniques are widely used in products such as Oracle's Java Hotspot VM, Microsoft's .NET Runtime, Google's V8 JavaScript Engine, etc. Implementations of JIT engines provide very useful insights for the design of this project.

2.3 Existing Emulators

There were already a number of RISC-V emulators in existence at the start of this project. I have evaluated them case-by-case to determine whether they can be used as the baseline or as the starting point of this project.

The RISC-V Foundation maintains an emulator `riscv-isa-sim` [22]. It is also called `spike`, as it is the name of the executable file of the project. In the rest of this dissertation, I will use `spike` to reference it. It is a cycle-accurate simulator that can either perform whole-system simulation or user-space simulation via a so-called proxy kernel `riscv-pk`. Fundamentally it is an interpreter, but with various tricks applied it is much faster than naïvely implemented interpreters.

QEMU [5] is an emulator that uses binary translation. It is a mature emulator that supports both system-level emulation and user-space emulation of both Linux and BSD

binaries. By utilising an IR, QEMU supports emulation of multiple guest ISAs on multiple host platforms. A port of QEMU is maintained by the RISC-V Foundation, and was upstreamed on 9 Mar 2018.

`rv8` [7], developed by Michael Clark from SiFive, is another RISC-V to AMD64 binary translator. It has inferior or similar performance compared to QEMU when I evaluated it for the draft project proposal, but it has improved significantly since then, outperforming QEMU in some benchmarks tested. `rv8` relies on handcrafted templates to perform binary translation. Its codebase is very immature compared to QEMU, and it fails to run many Linux binaries.

All of these tools would be ideal candidates for performance comparison. I have chosen `spike` as the primary performance target, as outperforming the reference interpreter is a significant and non-risky result. QEMU is selected to be the secondary performance target, as its maturity allows multiple benchmarks to be used for comparison.

I made the decision to not use any of the existing emulators as the starting point. `spike` is too specialised on its cycle-accuracy model and its optimisation tricks make it very hard to make large-scale changes, such as adding binary translation. QEMU is licensed under GPL, a copyleft license. I believe in permissive open-source licenses, so I would prefer my work not to be polluted by the GPL, therefore I have also ruled out QEMU. `rv8` makes heavy use of C++ templates, so it is hard to understand and modify. Its immaturity also means it is likely that I would end up debugging existing bugs, rather implementing my own code. The risk of starting from scratch is low. Even though I might have to spend a few weeks writing a decoder and interpreter, I would have first-hand knowledge about the codebase and therefore save time parsing the existing codebase and potentially achieve better code re-use. Combing all these factors I have decided to start fresh.

2.4 Requirement Analysis

The mandatory functional requirement for this project is that it must be able to execute unmodified, statically-linked, single-threaded 64-bit RISC-V Linux binaries that use only common system calls.

- RISC-V Linux binaries are formally defined as ELF files [10] with ELF type "EXEC" (executables) or "DYN" (dynamic executables), ELF machine type "RISC-V", and ELF OS/ABI "GNU/Linux". As RISC-V Linux binaries assume the processor to support at least RV64GC, the base 64-bit instruction set and all five standard ISA extensions MAFDC need to be supported.
- Statically-linked ELF binaries in this dissertation are defined to be ELF files does not contain the "INTERP" program header, and the ELF type is not "DYN".
- Single-threaded in this dissertation means that the program being emulated does not use the "clone" system call, shared memory map, or otherwise, to share writable code/data memory locations with other processes.

- Being able to execute means the program should execute as if it were executing under a native RISC-V Linux kernel. The program is assumed to be well-behaved: it should not rely on a particular layout, should not request excessive memory pages, should not exceed other potential system limits, and should not otherwise try to detect and workaround measures in the emulator. For example, the requirement does not mean that the emulator must execute a binary that is set to detect the existence of an emulator and crash.
- Unmodified means that if a binary satisfying other requirements can execute natively on RISC-V Linux, no special modifications are needed to make it execute under the binary translator.
- Common system calls are defined to be any necessary system calls to be able to use fopen, fread, fwrite, fseek and fclose implemented in GNU C Library (glibc).

The following requirements are important but are not mandatory for the success of this project:

- The ability to support dynamically-linked binaries. This lifts the restriction on the ELF file to not contain the "INTERP" header and allows the ELF type to be "DYN".
- Support all system calls necessary to run all benchmarks and other test suites.

The following requirements are listed as optional extensions in the project proposed, but have been dropped due to high risks of failure:

- The ability to support multi-threaded programs. This imposes high requirements on atomic instruction implementation. I dropped this extension early because it is an unknown field to perform binary translations between strong and weak memory consistency models and between compare-and-swap and load-linked stored-conditional atomic models.
- The ability to support full-system emulation. As full-system emulations require device emulation and address space translation, this extension is dropped as it is not manageable within the time budget of this project.

The project also has non-functional requirements on its performance.

- The emulator produced by this project shall have statistically significant performance improvement compared to `spike`. In this dissertation, the statistical significance level α is set to be 5% (i.e. 95% confidence interval).
- Ideally, the emulator should have better performance than QEMU or rv8, two other binary translators.

2.5 Development Methodology

I selected the spiral model of software development for this project. Each iteration of the project consists of planning, risk analysis, implementation, testing and evaluation. Initial iterations will implement basic functionality, while later iterations gradually improve upon previous iterations, e.g. by adding more analyses and optimisation passes. The risk analysis is important as there are many potential optimisations to implement but the time budget is very limited. Testing would be carried out in each iteration to ensure that at the end of each iteration the project is in a working state, passing all the unit test cases. Simple evaluation would be performed at the end of each iteration to guide the next iteration.

2.6 Choice of Tools

2.6.1 Programming Language

As mentioned in Section 2.3, I made the decision to start from scratch. As a result, I am not restricted in choice of programming languages.

As the project is very low-level, requiring manipulation of machine code and access to arbitrary memory locations, only system programming languages could be used. I have assessed C, C++ and Rust, and eventually decided to use C++.

There are very few reasons why C should be preferred to C++, including:

- Working on an embedded system with limited memory or storage;
- Need to interact with other languages;
- Existing codebase uses C for historical reasons.

As none of the above reasons apply for this project, C is ruled out.

Rust is a modern system programming language that does not have many historical pitfalls like C++. Its type safety and functional programming patterns also appeal to me. However, as it is likely that the emulator needs to handle signals, and signal handling in Rust is still missing, I decided not to adopt Rust.

C++ is the only choice remaining. I have used C++ for a few years already, and I consider myself reasonably experienced in C++, so choosing C++ is less risky than a new language such as Rust. C++'s high-level paradigms can make algorithm implementations easier and more readable, while it also has low-level abilities such as using C libraries, performing signal handling, and accessing arbitrary memory. Overall I believe C++ is the best fit for the project. C++17, the latest C++ revision, was chosen to make use of new language and library features.

Scripting languages and domain-specific languages, such as Makefile, bash, Python, JavaScript and Matlab will be used wherever suitable to simply development and evaluation. Matlab requires a license to use, and I obtained mine from the University's Software Distribution site.

2.6.2 Toolchain

The RISC-V Foundation has ported a few toolchains to RISC-V, including Binutils and GCC, and the ports are subsequently upstreamed. A cross-compilation toolchain is necessary to produce RISC-V Linux binaries to test and evaluate the project. I compiled GCC 7.2 and Binutils 2.29 with target `riscv64-unknown-linux-gnu` for this propose. Even though GCC released 7.3 and Binutils released 2.30 in January 2018, I decided not to upgrade the tools to maintain consistency of evaluation results.

The toolchain used to compile this project is GCC 7.2 and Binutils 2.26.1. GCC 7.2 is used as GCC 7+ is required for C++17, and 7.2 is the latest version when I started the project. Binutils 2.26.1 is used as it is the default version installed using the package manager in my compilation environment. GNU Make is used for build automation.

2.6.3 Test Suite

I elected to use `riscv-qemu-test` and Berkeley TestFloat for unit testing and Dhrystone, CoreMark and SPECint together for functional testing and performance benchmarking. Details are covered in Chapter 4.

The SPEC CPU benchmark suite requires a license. The University of Cambridge holds such a license and I have used the license for the benchmarking.

2.6.4 Development Environment

Any Linux environment on AMD64 can be used as a development and evaluation environment. I decided to use my own laptop that runs Microsoft Windows as the development environment for convenience. Window subsystem for Linux was enabled to provide a Linux compatible environment. In case that fails, I made plans to use the MCS Linux machines as a backup. Microsoft Windows requires a license to use and I obtained mine from the Microsoft Imagine programme for students.

Having said that, it is inconvenient to run time-consuming benchmarks on a personal laptop, as I frequently need to carry it around to attend lectures, supervisions, etc. Dr Timothy Jones kindly provided a workstation for running benchmarks.

2.6.5 Backup and Version Control

All paperwork and experiment results were periodically synchronized to OneDrive for Business storage using rsync. I used my personal OneDrive for Business tenant instead of the one provided by the university.

All source code were version-controlled using Git and the local repository was kept in sync with a private GitHub repository. The master branch always compiled free of warnings and errors, and was free from obvious bugs after any commits. All commits on master have meaningful commit messages, and contain all changes required for a functionality. Usage of private GitHub repository requires subscription and GitHub has kindly provided free subscriptions for all students.

2.6.6 Related Courses

Knowledge of computer architecture, compiler construction and optimisation techniques was required to complete the project. This project is highly relevant to Part II Optimising Compilers, Part IB Compiler Construction, and is relevant to Part IB Computer Design and Part II Comparative Architecture.

Chapter 3

Implementation

This project implements three execution engines for the emulator, shown in Figure 3.1. This includes an interpreter, a simple binary translator and an optimising binary translator. The three engines, although using different approaches for execution, share the same RV64GC instruction decoder and the same memory layout of process states (e.g. registers). Two binary translation engines share the same AMD64 assembler.

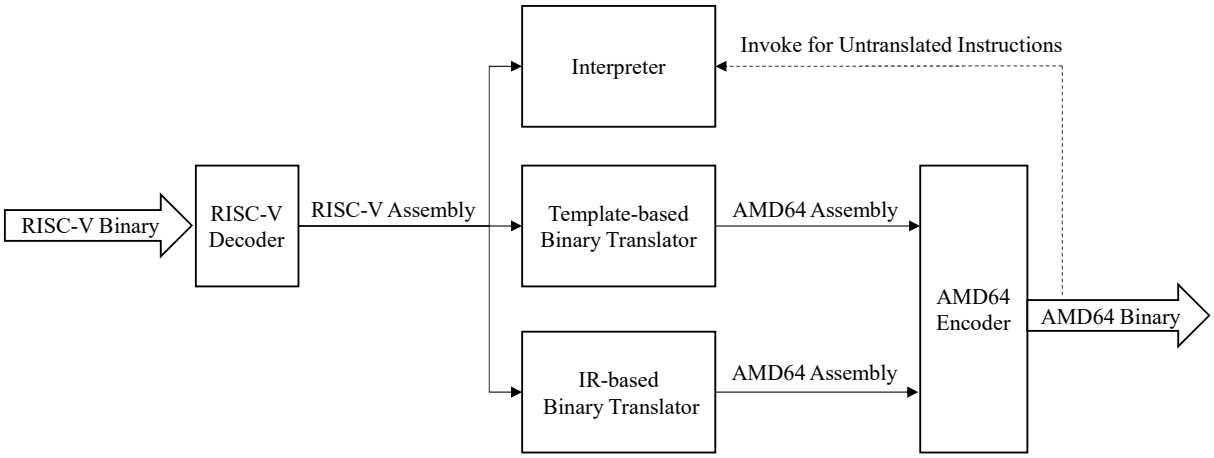

Figure 3.1: Overview of interpretation and translation stages

3.1 Interpreter

Despite the ultimate goal of the project being to create a binary translator, an interpreter is an important part of the codebase. In the design goal of this project, RV64IMC instructions will be binary translated, while AFD extensions still need to be supported for ABI compatibility. The interpreter exists in the codebase to support these instructions. When such instructions are encountered by the binary translator, a function call to the interpreter is generated.

The interpreter also serves as a tool for validation and debugging purposes. Interpreters are very simple, so they can be relatively easily tested, debugged and validated. On the

other hand, it is very difficult to test whether the actual behaviour of the program matches the expected behaviour, when the program is transformed and the native AMD64 code is generated and executed. Execution traces of the interpreter and the binary translator can be compared to discover issues in the binary translator. This technique helped to pinpoint many bugs.

3.1.1 Software Floating-point Library

When trying to support floating point instructions in the interpreter, it turns out RISC-V requires an additional rounding mode. IEEE 754 [3] defines 5 rounding modes, namely towards 0, towards $-\infty$, towards $+\infty$, to nearest and ties to even, to nearest and ties away from 0. The last rounding mode is added in the 2008 revision and exists in neither C/C++ standard nor AMD64, but is required by RISC-V [23]. The difference requires software floating point support. I decided to implement a software floating library from scratch using modern C++ for better integration with the interpreter.

3.1.2 Trap Handling

The emulated application can access arbitrary memory locations, therefore it is possible that a load/store instruction will trigger synchronous signals, or traps, such as SIGSEGV or SIGBUS. If they are not handled properly, the entire emulator will abort, which is not desirable. Traditional emulators solve the issue by using setjmp on the interpreter's main loop and longjmp in the signal handler. The approach is not ideal as setjmp/longjmp have no guarantees on executing destructors, breaking C++'s resource acquisition is initialization (RAII) paradigm in which a resource is tied to an object's lifetime.

I use C++'s zero-overhead exceptions for handling signals. When signal is received, a C++ exception is thrown from the signal handler. This needs special compiler support which is fortunately available in commonly used compilers, such as GCC, Clang and ICC, via the `-fnon-call-exceptions` compilation flag [15]. As turning on non-call exceptions will negatively impact performance, only a single file is compiled with the flag which exposes my functions `util::safe_read<T>`, `util::safe_write<T>`, and all guest memory accesses that can potentially cause traps are replaced with calls to these two functions.

3.1.3 Environment Emulation

In order to execute unmodified Linux binaries, the emulator needs to also emulate environment setup, load binaries and translate all system calls made by the guest program. When the emulator starts, the following happens in order:

- Environment variables and command line arguments are pushed onto the stack.
- Random seeds are pushed onto the stack. Dynamically linked binaries rely on the random seeds to perform randomization.

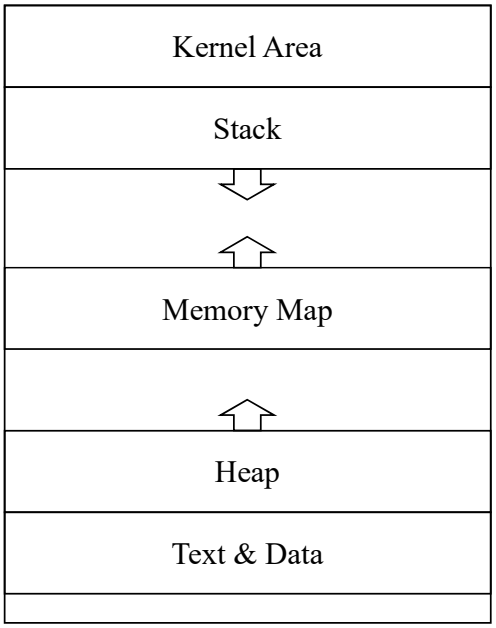

Figure 3.2: Memory layout of native application

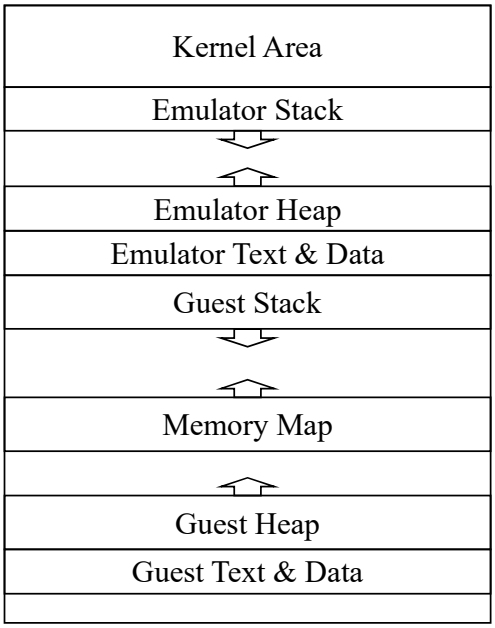

Figure 3.3: Memory layout of this project

- Specified file is loaded and its program segments are mapped into memory.
- If the specified file is dynamically linked, its interpreter is also loaded and mapped.
- Auxiliary vectors are set up to pass UID, GID, location of random seed and information about ELF to the emulated program [14]. These vectors are required by dynamically linked binaries.
- Registers are initialised and control is handed to the emulated program.

To allow the emulated program to freely place itself, the emulator text and data is placed at `0x7fff00000000`. The kernel will therefore place the heap and stack on higher addresses. The guest application can rarely see any differences in execution as it makes no assumptions about stack addresses. The chance of address space collision is negligible as normal applications rarely ask themselves to be placed in such a high location. Figure 3.2 and Figure 3.3 illustrate the comparison between memory layouts of native and emulated application.

Dynamically linked binaries in addition need to access shared libraries in the system root. As the emulated program cannot use libraries in the host OS's system root, system root translation is also implemented by redirecting accesses to files if the path exists in the specified system root (default to `/opt/riscv/sysroot`).

System calls made by the guest program are translated and then delegated to the actual kernel. Most system calls require only data structure translation (as the memory layout may be different), flag and error code translation. Some file accesses need to be treated specially, e.g. access to files under `/proc/self/`, in addition to aforementioned system root access redirection. For `mmap` and `mprotect`, `PROT_EXEC` is translated into `PROT_READ` as the guest program is not directly executed.

`brk`, the system call to adjust the size of the heap, is entirely emulated by maintaining a brk pointer and uses `mmap` to allocate new pages if the heap has to be expanded. One of the major obstacles encountered during implementation was related to the behaviour of `brk` when the heap is shrunk and then expanded. According to the legacy POSIX standard [1] (the newer version of POSIX removes specification about `brk`) the memory content is unspecified, while examining glibc's source code reveals that glibc assumes the area to be zeroed, therefore not calling `memset`s for `calloc`s in certain scenarios. Glibc's expectation matches the behaviour of Linux, as Linux will reclaim pages when the heap is shrunk, and therefore zeroed pages will be allocated when the heap is expanded again. I modified the implementation of `brk` to mimic Linix's behaviour.

To allow diagnosing system call emulation of guest programs, I have also implemented a single system call tracer which logs each guest system call. The tracer can be turned on using a runtime option `--strace`, regardless of the execution engine being used.

3.2 Simple Binary Translation

I implemented a simple binary translator for RISC-V using templates. All guest architectural registers are stored in memory, and each instruction is translated independently using the predefined templates. A whole basic block is translated at a time, where a basic block is a continuous sequence of instructions that ends with a branch instruction, so if the first instruction is executed, all instructions in the block are executed.

All translated code is cached in memory to avoid recompilation when the same program counter (PC) is executed again, unless there is an explicit `fence.i` instruction to flush the instruction cache. There are two levels of cache. A tagged, direct-mapped cache is used for fast cache lookup, and a hash table is used for slow cache lookup. Each time a PC is given, the lowest significant bits (excluding the last bit, which is always 0 as RISC-V instructions are 16-bit aligned) are used to index into the cache. If the tag matches, then the cached code will be executed, otherwise the PC will be used to look up in the hash table. If both lookups fail, the translator is invoked.

In the binary translator the INSTRET counter increment is disabled by default. INSTRET is the control status register (CSR) representing the number of instructions retired in RISC-V. It can be used for performance monitoring, however it is usually unused by normal applications. It can be enabled by a runtime flag `--with-instret`. This matches the default behaviour of rv8 and QEMU.

3.2.1 Division Exception Handling

On AMD64, division by zero or quotient overflow will result in a division exception being raised [12]. RISC-V differs from AMD64 and many other instruction sets by not having division exceptions. This difference must be addressed in order to preserve the semantics of the translated program.

Two cases may cause a division exception: the divisor being zero, or in case of a signed division, the dividend being the signed minimum value and the divisor being -1. Existing emulators, including rv8 and QEMU port, generate code to check the operands before the division is executed. However, as the C/C++ standard mandates that divide by zero or signed arithmetic overflow is an undefined behaviour, in pure C/C++ code it is extremely unlikely that the checks will ever fail. In some other languages, division exceptions will trigger language-level exceptions, which are also considered rare cases. By checking the operand each time before execution, performance penalties are introduced to the common case.

This project seeks an alternative strategy. It relies on the fact that the division exception will be caught by the operating system and then be turned into SIGFPE, the floating-point exception signal, and the fact that in POSIX-compliant operating systems, signal handlers can inspect and modify the program's context at the point the exception was raised using `sigaction`. When a SIGFPE is received, the signal handler will disassemble the instruction at the current RIP (AMD64's name for the program counter) and read out the value of the dividend and the divisor from the execution context. It will then place the correct quotient and remainder value defined per the RISC-V standard into the RAX and RDX registers, increment RIP to the next instruction and ask the operating system to resume execution. For the application code, it will behave as if the DIV or IDIV instruction succeeds without faulting. This approach eliminates the overhead of range checks while preserving the correctness of the semantics defined by the ISA.

3.2.2 Exception Handling Frames

As mentioned in 3.1.2, traps will be converted into userspace exceptions, so any memory accesses in the dynamically generated code may throw exceptions. Calls to interpreters might also throw exceptions. The thrown exception must be able to propagate through the generated code. In Linux and Unix, exception handling in C++ is powered by a language-agnostic exception handling and stack unwinding library and language-specific parts [2]. A DWARF derived format is used to describe how to restore registers saved in stack frames and handle exceptions, allowing the unwinding library to correctly unwind the stack.

In DWARF [9], the common information entry (CIE) contains a pointer to the personality routine. The personality routine is a language-specific routine that will determine whether an exception should be caught, and how to react with stack unwind. The frame description entry (FDE) contains a 64-bit value called the language specific data area (LSDA), usually a pointer to metadata that assists the personality routine. Usually one FDE is generated per function, so the personality routine can efficiently acquire the corresponding LSDA. Figure 3.4 gives an overview of the data structures. Normally these structures are placed in the `.eh_frame` section, but it is also possible to register/deregister additional ones in the runtime using `__register_frame` and `__deregister_frame`.

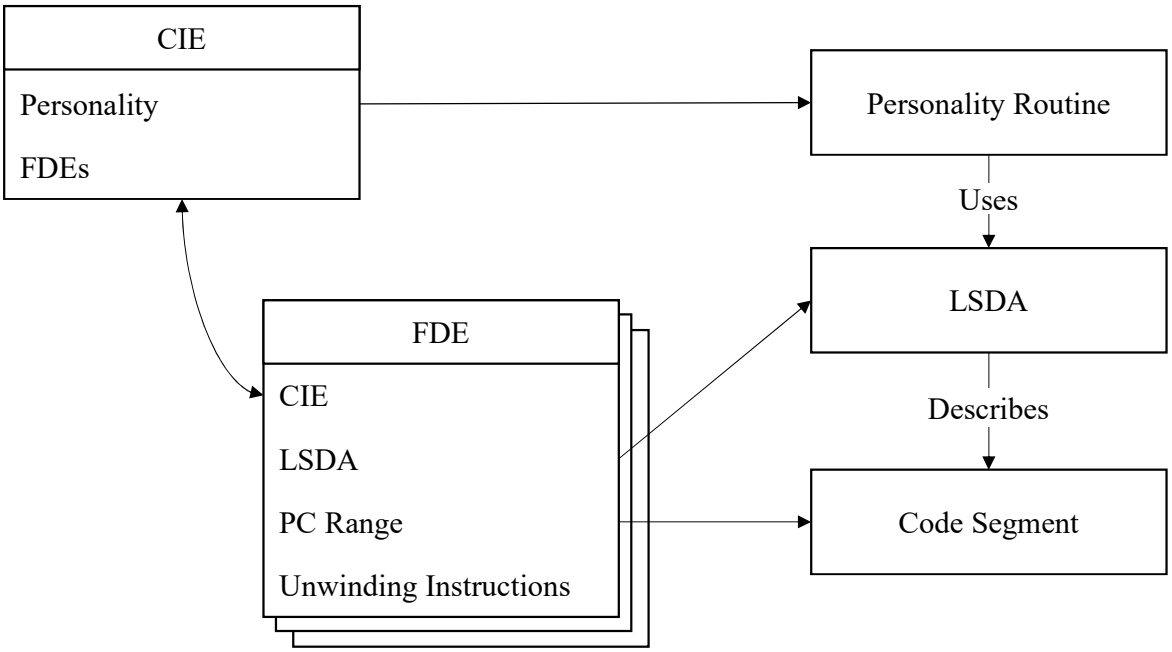

Figure 3.4: Simplified view of exception handling structures

When an exception happens, the unwinding library can search the corresponding FDE using the PC. It can therefore also acquire the LSDA and call the personality routine. If the personality routine decides to catch the exception, it will also set up the contexts.

As there are no high-level concepts like exception handlers and destructors in machine code, a manually crafted CIE and FDE template suffices. A personality routine is implemented to provide precise exception support: in the simple DBT, the PC register and the INSTRET CSR (if enabled) are updated only once per basic block, so when the translated code is unwound, the correct values of both PC and INSTRET CSR need to be determined by the current host RIP register, and written back into the memory-backed guest architectural register file.

3.3 Optimising Binary Translation

A drawback of the template-based binary translation approach is lack of portability. If the binary translator intends to support m guest architectures on n host architectures, mn sets of templates need to be crafted. An alternative approach, which is also used by QEMU, is to first translate into an IR, and then translate the IR to the host architecture. The optimising binary translator implemented in this project additionally performs IR transformations to improve performance. Figure 3.5 gives an overview of stages of the optimising binary translator.

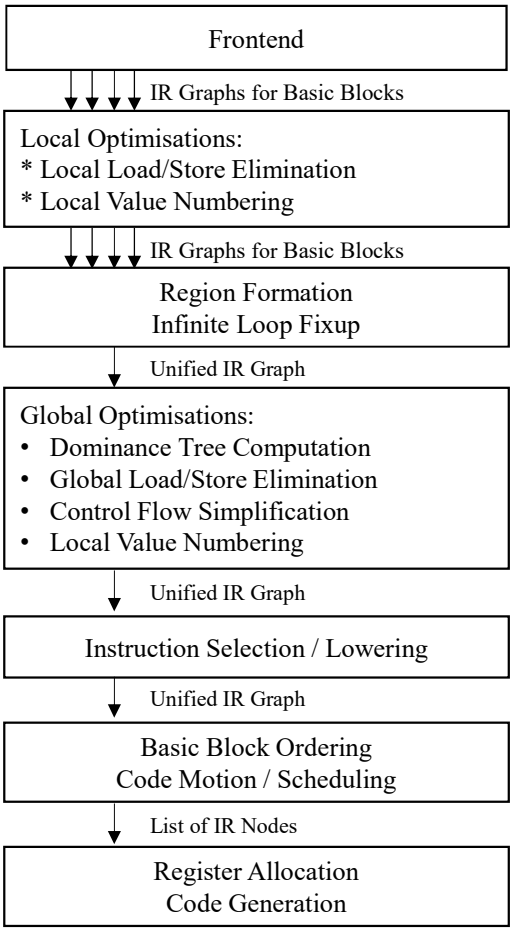

Figure 3.5: Overview of stages of the optimising binary translator

3.3.1 Intermediate Representation

This project uses a a graph-based intermediate representation (also called sea of nodes). The design of the IR is inspired by libFIRM [6] and the C2 compiler in Hotspot JVM [8].

Figure 3.6 shows an IR graph for calculating Fibonacci sequence. Nodes in the graph represent computations and edges represent dependencies. All types of dependence are explicitly modelled as edges, including control-flow dependencies (red edges in the figure), memory dependencies (blue edges) and data-flow dependencies (black edges). Each node accepts one or more operands as input (except for the constant node and entry node), and produces one or two values as output (except for the exit node). All values in the IR are typed with one of i1, i8, i16, i32 and i64, or `control` or `memory` if they represent control-flow or memory dependence. Each value of type `control` or `memory` can be only used once, except for keepalive edges (see Section 3.3.4).

The graph nature of the IR means that there are no variables, so it is naturally in SSA form. The graph form makes transformation easier by not having to worry about where to insert new instructions, but it makes code generation harder as the graph has to be converted into a linear list of instructions first.

Nodes include:

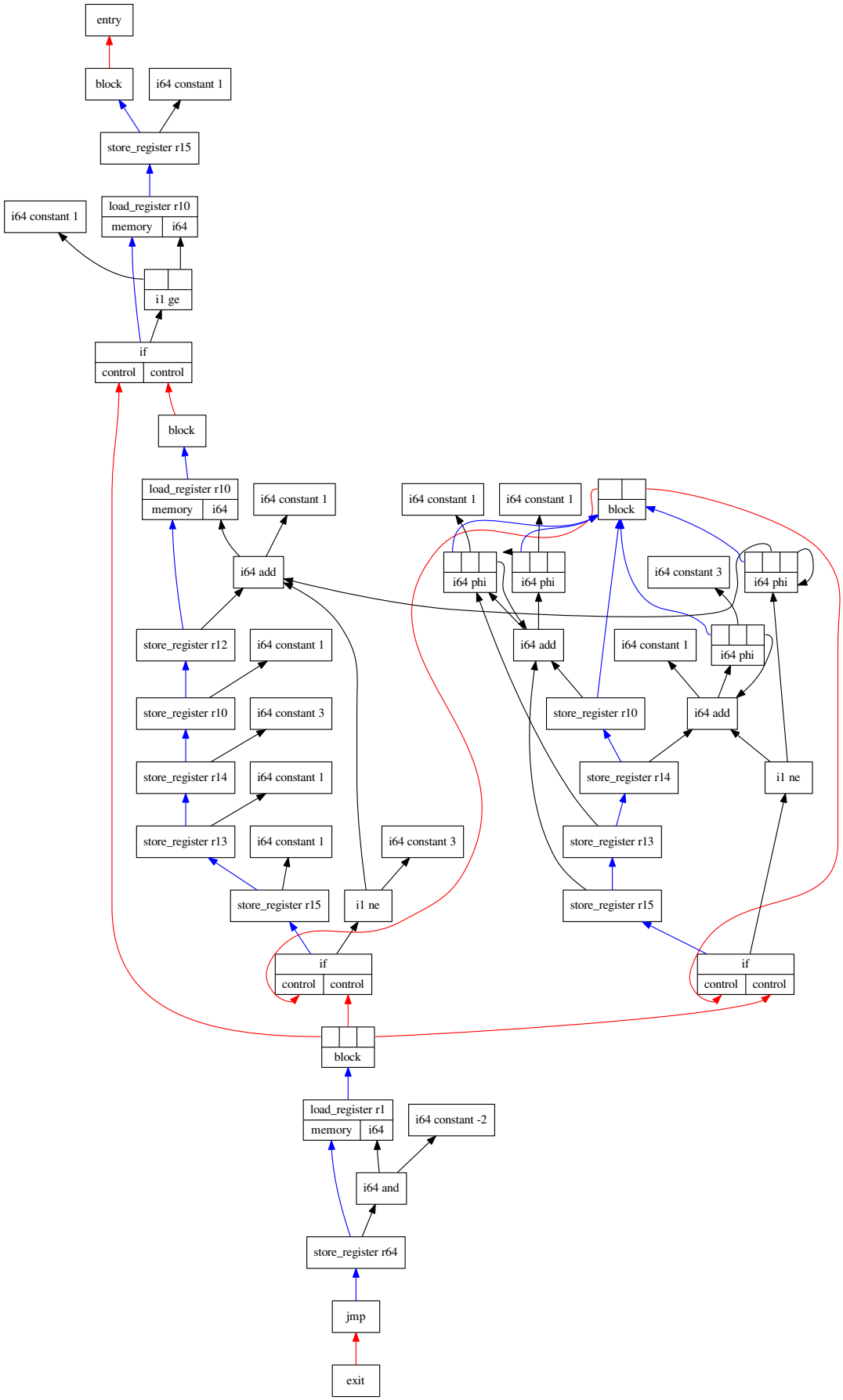

Figure 3.6: The optimised IR graph from a typical Fibonacci function

- entry, exit: special nodes, there is exactly one copy of each node in each graph.
- block: starting of a basic block. The block node takes one ore more control inputs, denoting successors in the CFG.
- jmp, if: end of a block. A jmp node produces a control output while if takes an i1 input and produces two control outputs.
- phi: merge values when merging control flow.
- call: call a helper function. Produced when the instruction cannot be binary translated.
- constant: refers to a constant value.
- cast: casting between different integer types. Upcasts can either be sign-extended or zero-extended.
- neg, not, add, sub, xor, or, and, shl, shr, sar: unary and binary arithmetic nodes that produce one output.
- eq, ne, lt, ge, ltu, geu: comparison nodes that produce an i1.
- mux: ternary node that takes an i1 and selects one of the two other operands as appropriate.
- mul, mulu, div, divu: arithmetic nodes that take two input and produce two outputs. For multiplication they are the lower bits and higher bits of the product, and for division they are the quotient and remainder.
- copy: produce a copy of a value. This is redundant in a graph-based SSA, but it can ease register allocation.
- Other backend-specific nodes. The AMD64 backend implemented defines an address node to represent complex address modes, and a lea node to represent AMD64's load effective address instruction.

As most nodes takes fewer than two operands and produce fewer than two values, I implemented a `util::Small_vector<T, N>` to place values on the stack (or in the containing structure) instead of allocating them on the heap. The optimisation reduces memory allocation and deallocation by approximately one third.

To facilitate debugging and visualisation, I have implemented a pass to print out Graphviz's .dot representations of the IR graphs. An visualiser is also implemented to automatically retrieve the printed Graphviz code from the log and present an user interface for easy speculation.

3.3.2 Frontend

A minimal frontend is implemented to translate RV64IM instructions into the IR. Compressed opcodes defined in the C-extension are expanded into their full form during de-

coding, so the frontend does not have to handle them specially. The frontend is minimal as it performs virtually no optimisations: e.g. `li a0, 1`, which is a shorthand for `addi a0, x0, 1`, will be translated to Figure 3.7.

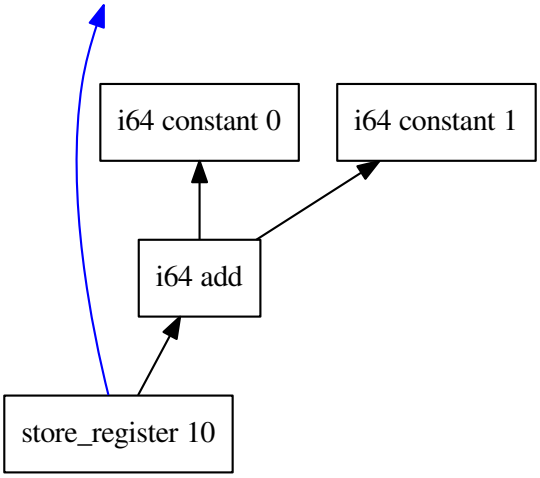

Figure 3.7: The unoptimised IR of `li a0, 1` generated by the frontend

Similarly `mv` will be translated to a load_register node, a constant 0 node, an add node, and a store_register node. After generating the IR graph, a simplified version of load and store elimination (see 3.3.6) is applied followed by local value numbering (see 3.3.7), which takes care of constant folding. It is essential that the basic optimisations are applied before region formation, so that indirect jumps such as `auipc t0, offset; call t0` can be treated as a direct jump.

3.3.3 Region Formation

Performing transformations and optimisations only on a single basic block is insufficient to achieve high performance. However unlike traditional compilers, which know about the control flow in the program, at the machine code level it is difficult to precisely recover the control flow graph, especially when the binary is compiled with optimisations enabled. A very simple method is used in this project to combine blocks together, described by Algorithm 1.

This essentially means that known reachable blocks will be decoded speculatively. A size limit (measured as a number of basic blocks) is imposed to prevent the binary translator from forming a region containing a huge number of blocks that may never be executed. The limit is tunable in runtime via a flag `--region-limit=<n>`, and details about tweaking it are described in Section 4.3. Special treatment is made to ensure that the foreach loop processes newly added blocks later, so the control flow graph is explored in a breadth-first manner. This prevents the algorithm from following a deep but unlikely branch. As loop bodies are usually small, a region is usually able to contain all blocks of the loop bodies, resulting in significant speedup.

```
repeat
    foreach b ∈ pred(exit) do
        if PC is set to a constant p then
            if p is already added then
                modify b to jmp to the block for p instead of exit
            else if region size limit is not reached then
                decode p and add it to the graph
                break
until no changes are made
```

Algorithm 1: Region formation

3.3.4 Infinite Loop Handling

The IR of this project is represented in a graph structure. There is a single entry node and a single exit node. Two desired properties for the graph are that for every block in the program,

- there is a control path flowing from the entry node to the block, and
- there is a control path flowing from the block to the exit node.

The first property is always true considering how region formation works, however the second is not if the code contains infinite loops. Without the second property it is difficult to perform certain program transformations and analyses:

- Dead code elimination via garbage collection will recycle blocks within the infinite loop, therefore breaking the semantics of the program.
- The post-dominator tree will not exist, as some blocks will not have successors, causing some algorithms depending on post-dominator analysis to fail.

The issue is dealt with by inserting keepalive edges, which are pseudo control-flow edges from a block to the exit node. Algorithm 2 describes how keepalive edges are inserted. The keepalive edges are inserted in a way that when these pseudo-edges are considered, the second property holds. These edges can be easily skipped in transformations that do not require the second property to work.

3.3.5 Dominance Tree Computation

Dominator and post-dominator trees are very helpful for transformations and analysis in SSA form. Immediate dominator computation using data-flow analysis has time complexity $O(mn)$ where n is the number of blocks and m the number of control flow edges. The Lengauer-Tarjan algorithm [18] improves the time complexity to $O(m\alpha(m, n))$ with α being the inverse Ackermann function.

```
unseen_blocks ← {all blocks reachable from entry}
worklist ← pred(exit)
while unseen_blocks ≠ ∅ do
    while worklist ≠ ∅ do
        remove a block b from worklist
        if b ∈ unseen_blocks then
            unseen_blocks ← unseen_blocks \ {b}
            worklist ← worklist ∪ succ(b)
    if unseen_blocks ≠ ∅ then
        pick a block b from unseen_blocks
        add a keepalive edge from b to exit
        worklist ← worklist ∪ {b}
```

Algorithm 2: Keepalive edge insertion

This project uses a simple version of the Lengauer-Tarjan algorithm, described by Algorithm 3. The algorithm is adapted from [20] with worse time complexity $O(m \log n)$ but performing better when number of blocks is small. Immediate post-dominators are computed similarly, except that the $succ(i)$ in the algorithm is replaced with $pred(i)$ and the depth-first search operates on the reverse graph.

Dominance frontiers are also computed, using the straight-forward algorithm [11]. Post-dominance frontiers computation is similar, using immediate post-dominators instead and replacing $pred(i)$ with $succ(i)$. Algorithm 4 gives the algorithm for computing dominance frontiers.

3.3.6 Load and Store Elimination

The IR of this project mandates all guest architectural registers to be placed in memory and makes register load and store explicit. This allows the frontend to produce SSA form directly by generating a register load node before a computation is carried out, and a register store node after the computation. To allow other data-flow analysis to extract useful information and to achieve reasonable performance in the generated code, redundant loads and stores need to be eliminated.

Load elimination can be performed using an SSA construction algorithm, e.g. LLVM's `mem2reg` pass. The classic SSA construction algorithm [11] assumes variables will not be clobbered by other operations, which is not true in this project, as interpreter calls may access or modify memory-backed registers. A modification is made to account for the difference. Algorithm 7 describes the modified SSA-construction-based load elimination algorithm, featuring the following 4 stages:

- The standard ϕ-insertion algorithm is executed.

```
number all blocks from 1 to n using depth-first search
foreach i ← 1 to n do
    parent(i) = DFS tree parent of i
    semi(i), best(i) ← i
    idom(i), ancestor(i) ← ⊥
    bucket(i) ← ∅
procedure eval(i)
    a ← ancestor(i)
    if a = ⊥ then return i
    if ancestor(a) ≠ ⊥ then
        u ← eval(a)
        if semi(best(i)) > semi(u) then best(i) ← u
        ancestor(i) ← ancestor(a)
    return best(i)
for i ← n to 2 by −1 do
    p ← parent(i)
    foreach v ∈ pred(i) do
        u ← eval(v)
        if semi(i) > semi(u) then semi(i) ← semi(u)
    add i to bucket(semi(i))
    ancestor(i) ← p
    foreach v ∈ bucket(p) do
        u ← eval(v)
        if semi(u) < semi(v) then
            idom(v) ← u
        else
            idom(v) ← p
    bucket(p) ← ∅
for i ← 2 to n do
    if idom(w) ≠ semi(w) then idom(w) ← idom(idom(w))
```

Algorithm 3: Lengauer-Tarjan algorithm

```
foreach block i do df(i) ← ∅
foreach block i do
    d ← idom(i)
    foreach v ∈ pred(i) do
        r ← v
        while v ≠ d do
            df(r) ← df(r) ∪ {i}
            r ← idom(r)
```

Algorithm 4: Computing dominance frontier

procedure `fill_phi(`b`)`
- **if** $b \in$ phis **then**
  - *push*(value_stack, phis(b))
- **foreach** operation m in block b **do**
  - **if** m is a load of r **then**
    - *push*(value_stack, output of m)
  - **else if** m is a store of r **then**
    - *push*(value_stack, input of m)
  - **else if** m may define r **then**
    - *push*(value_stack, $\perp$)
- **foreach** successor s of b **do**
  - **if** $s \in$ phis **then**
    - fill corresponding operand of phi(s) to be *top*(value_stack)
- **foreach** s immediately dominated by b **do**
  - `fill_phi(`s`)`
- pop out all values pushed into the stack in this call

Algorithm 5: Filling ϕ-nodes in load elimination

- The renaming algorithm is executed once to populate operands of the SSA nodes, without touching any other nodes. If a node may clobber the register value, $\perp$ is pushed to the value stack to indicate invalid value. Algorithm 5 gives an overview.
- $\perp$ is propagated, replacing all ϕ-nodes containing $\perp$ with $\perp$.
- The renaming algorithm is executed again, and load elimination is performed if the replacing value is not $\perp$. Algorithm 6 gives an overview.

The actual load elimination implemented handles all architectural registers at once instead of doing so separately for performance concerns.

Store elimination will eliminate all store nodes that will always be followed by another store without intervening reads. This could be accomplished by very-busy expression analysis, but I chose to use an algorithm similar to load elimination on the reversed graph, described by Algorithm 8, Algorithm 9 and Algorithm 10. The ϕ-nodes created in store elimination are transient - they will never be added to the actual graph.

After store elimination, some blocks may become empty. Such blocks are identified and removed.

For simplicity, by default the store elimination assumes that (guest) memory operations will not read register values. This assumption is true for most programs, but not if the guest program will ever try to resume from a trap (as this project does for division handling). Reliance on the assumption can be turned off by using `--strict-exception`.

```
procedure rename(b)
    if b ∈ phis then
        push(value_stack, phis(b))
    foreach operation m in block b do
        if m is a load of r then
            v = top(value_stack)
            if v ≠ ⊥ then
                replace all references to output of m with reference to v
            else
                push(value_stack, output of m)
        else if m is a store of r then
            push(value_stack, input of m)
        else if m may define r then
            push(value_stack, ⊥)
    foreach s immediately dominated by b do
        rename(s)
    pop out all values pushed into the stack in this call
```

Algorithm 6: Renaming in load elimination

```
Input: architectural register number r
phis = {}
worklist = {blocks containing accesses to (or may clobber) r}
while worklist ≠ ∅ do
    remove a block b from worklist
    foreach dominance frontier f of b do
        place a φ-node n for r at entry of f
        worklist ← worklist ∪ {f}
        phis ← phis ∪ {f ↦ n}
value_stack = {⊥}
fill_phi(successor of entry)
repeat
    repalce all φ-nodes containing ⊥ with ⊥
until no changes are made
rename(successor of entry)
```

Algorithm 7: Load elimination

```
procedure fill_phi(b)
    if b ∈ phis then
        push(value_stack, phis(b))
    foreach operation m in block b in reverse order do
        if m is a store of r then
            push(value_stack, ⊤)
        else if m is a load of or may use r then
            push(value_stack, ⊥)
    foreach predecessor s of b do
        if s ∈ phis then
            fill corresponding operand of phi(s) to be top(value_stack)
    foreach s immediately post-dominated by b do
        fill_phi(s)
    pop out all values pushed into the stack in this call
```

Algorithm 8: Filling ϕ-nodes in store elimination

```
procedure rename(b)
    if b ∈ phis then
        push(value_stack, phis(b))
    foreach operation m in block b in reverse order do
        if m is a store of r then
            v = top(value_stack)
            if v ≠ ⊥ then remove m
            push(value_stack, ⊤)
        else if m is a load of or may use r then
            push(value_stack, ⊥)
    foreach s immediately post-dominated by b do
        rename(s)
    pop out all values pushed into the stack in this call
```

Algorithm 9: Renaming in store elimination

Input: architectural register number r
phis = {}
worklist = {blocks containing accesses to (or may clobber) r}
while worklist $\neq \emptyset$ **do**
    remove a block b from worklist
    foreach post-dominance frontier f of b **do**
        create a ϕ-node n for r, without adding to the graph
        worklist $\leftarrow$ worklist $\cup \{f\}$
        phis $\leftarrow$ phis $\cup \{f \mapsto n\}$
value_stack = $\{\bot\}$
foreach b immediate post-dominated by exit **do** `fill_phi(`b`)`
repeat
    replace all ϕ-nodes containing $\bot$ with $\bot$
until no changes are made
foreach b immediate post-dominated by exit **do** `rename(`b`)`
destroy all ϕ-nodes

Algorithm 10: Store elimination

3.3.7 Local Value Numbering

Hash-based local value numbering [11] is implemented to perform local common expression elimination. Constant folding and arithmetic identity simplification are also performed. Notable transformations made in local value numbering pass include:

- Multiple instances of arithmetic nodes with the same operands are replaced with their first occurrence.
- Arithmetic nodes whose both operands are constant are evaluated and replaced with constant.
- Casts followed by another cast are folded to a single cast or folded away if possible. This is notable as the frontend will generate casts for instructions like `addiw`, relying on this transformation to simplify the graph.
- Commutative arithmetic nodes are normalized so the constant operand is always the second operand.
- Adding 0 is folded away. This simplifies the graph generated for `mv`.
- Subtracting a number from 0 is replaced with negation, and xor-ing with -1 is replaced with bitwise not. This is important as RISC-V encodes negation/not as subtract/xor.
- Anding with 0xFF is replaced with cast to i8, and left shifting 48 bits followed by right shifting 48 bits is replaced with cast to i16. This optimisation is vital for code quality as RISC-V does not provide 16-bit arithmetic directly but AMD64 does.

- mul and mulu nodes are merged if higher bits of the product of one of such node is unused.

3.3.8 Basic Block Ordering

Before generating code, the graph-based IR needs to be converted into a linear list of instructions. In this project this is done in two steps: basic blocks are ordered into a linear list, then other nodes are scheduled into basic blocks.

When ordering basic blocks, the following properties are desired:

- the successor of entry node is ordered as the first block;
- dominators of a block are ordered before the block.

The second property is essential to guarantee that when instructions are scheduled into basic blocks, definitions will always appear before usages. A simple pre-order depth-first search would produce an ordering to satisfy both of these properties.

In addition, we also want to maximise the number of fall-throughs (i.e. minimise the number of jump instructions needed). Formally speaking, for each ordering $\sigma : \text{Block} \to \mathbb{N}$ and control flow edge $a \to b$, define penalty p as:

$$p(\sigma, a, b) = \begin{cases} 0 & \text{if } \sigma(b) = \sigma(a) + 1 \\ 1 & \text{otherwise} \end{cases}$$

Define the total penalty as

$$p(\sigma) = \sum_{\text{all control flow edge } a \to b} p(\sigma, a, b)$$

The target is to find $\operatorname{argmin}_{\sigma} p(\sigma)$. One way to explore the search space of σ is to start with a guess (in this case the outcome of depth-first search) and try to swap pairs (without violating the two properties) to see if the penalty decreases. However as the penalty is not varying continuously, it is likely the algorithm will get stuck in plateaus. This project uses distances between blocks to guide minimisation to smooth the penalty change.

$$\mathit{distance}(\sigma, a, b) = \begin{cases} \sigma(b) - \sigma(a) - 1 & \text{if } \sigma(b) > \sigma(a) \\ \sigma(a) - \sigma(b) & \text{otherwise} \end{cases}$$

$$\tilde{p}(\sigma, a, b) = p(\sigma, a, b) + \mathit{distance}(\sigma, a, b)$$

It is debatable whether the reordering is worthwhile. As the optimisation eliminates only unconditional jumps, if the branch target buffer in modern CPUs hits then the number of clock cycles saved by the optimisation is very limited.

3.3.9 Code Motion/Scheduling

The next task is to assign each node into one of the basic blocks (except special ones such as constant nodes). This step is necessary as the sea-of-nodes IR design permits nodes to float around. The scheduling algorithm is inspired by HotSpot JVM's approach [19]. Conceptually the assignment takes place in two stages. The first stage identifies the earliest legal block for each node and establishes a topological ordering of the nodes. described by Algorithm 11.

visited $\leftarrow$ {all constant nodes}
order $\leftarrow$ empty list
foreach block b in topological order of dominator tree **do**
  visited $\leftarrow$ visited $\cup$ {b and all associated ϕ-nodes}
  repeat
    foreach node $n \notin$ visited **do**
      if depedencies of $n \subseteq$ visited **then**
        visited $\leftarrow$ visited $\cup \{n\}$
        early(n) $\leftarrow b$
        push_back(order, n)
  until no changes are made

Algorithm 11: Find the earliest legal block

The algorithm will produce a legal scheduling, though it is inefficient as ideally computations should be as close to their usages as possible. The second stage therefore tries to schedule these nodes into later blocks using the order, described by Algorithm 12.

foreach block b **do**
  let n be the ending node of b
  late(n) $\leftarrow b$
  list(b) $\leftarrow$ empty list
foreach node n in reversed order **do**
  $t \leftarrow \bot$
  foreach node r that references n **do**
    if r is not ϕ-node **then**
      `// assert that` late(r) `is already assigned`
      $t \leftarrow$ lease common dominator of t and late(r)
    else
      let b be the block associated with n in ϕ-node r
      $t \leftarrow$ lease common dominator of t and b
  late(n) $\leftarrow t$
  push_front(list(t)) $\leftarrow n$

Algorithm 12: Schedule nodes as late as possible

The assertion in the second stage is guaranteed by the particular ordering chosen in the first stage. After the second stage, we have a linear list of all the blocks and a linear list of instructions for each block, making it possible to proceed to register allocation and code generation.

3.3.10 Backend

The backend consists of three parts, instruction selection, register allocation and code generation. Instruction selection happens before scheduling, and register allocation happens after scheduling and before code generation.

The instruction selection for AMD64 is very simple, as there are AMD64 instructions for all arithmetic node types in the IR. The only transformation made in the AMD64 instruction selector is to map $a + b * c + d$ to a lea node or an address node, if it is used as a memory address.

A very simple register allocation algorithm is used. Registers are allocated when the value is first encountered, and spilling will occur if none of the registers are free. New "copy" nodes are inserted into the graph when values change location, e.g. spill, unspill or moved to a different register. This allows each value to be associated with only a single register/memory location, simplifying code generation.

The code generation is a simple linear scan over the ordered and scheduled IR, generating assembly code using the allocated registers. The code generator also reorders ϕ-nodes to preserve the semantics that all ϕ-nodes moves are simultaneous. The IR graph and all intermediate analysis results will be discarded, and the output AMD64 code will be cached and executed similar to the simple binary translator described in Section 3.2.

3.3.11 Chaining

In region formation we identified all blocks that set PC to a constant before jumping to the exit. In cases where region formation does not eliminate this pattern completely or it is freshly introduced by optimisations after region formation, it is desirable that the DBT can generate

```
    jmp translated_address
```

which has less overhead than returning control back to the binary translator's main loop. However as it is possible that the target is not yet translated, the DBT will instead generate

```
.trampoline:
    mov rax, .trampoline
    ret
```

and when the target address is known, the trampoline will be replaced with a jump directly to the translated address.

3.3.12 Compilation Threshold

Compiling a region using the optimising DBT is slow. Therefore, if a block is executed only a few times, it is not worthwhile to have that block compiled. To reduce worthless compilations, I have introduced a tunable compilation threshold to guide the optimising DBT, specified by a runtime flag `--compile-threshold=<n>`. A block will trigger region formation and compilation only if the number of times it is executed exceeds the threshold, otherwise it will be interpreted.

When running benchmarks, compilation time is only a negligible proportion of the overall time, therefore I set the default compilation threshold to 0, i.e. all blocks are compiled before execution.

3.4 Summary

I have demonstrated most techniques and algorithms that I have used when implementing the three execution engines. A few novel approaches, including trap and exception handling, are used in this project for better performance and maintainability. I have also implemented a few well-known optimisations, with tweaks made to account for differences in binary translation. This project also includes a huge amount of boilerplate code for decoding and encoding RISC-V and AMD64 assemblies, but these code pieces are not described in this chapter for their lack of technicality and novelty. Directory structures and lines of code are available in Appendix A.

Chapter 4

Evaluation

This chapter evaluates the project to determine whether the success criteria was met. The evaluation is split into four aspects: correctness, performance, tuning of parameters and analysis of overhead. The functional goal of the original success criteria focuses on producing a working emulator, which is evaluated in the correctness section. The performance section illustrates how this project met and surpassed the original non-functional goal. The final two section explains how default runtime parameters are selected, and how the selection of parameters influences compilation overhead.

4.1 Correctness

It is vital that the emulator developed in this project can execute RISC-V programs correctly. A number of techniques are used to test the correctness, and together they provide a broad coverage of the codebase.

4.1.1 RISC-V ISA Tests

A userspace port [21] of the official RISC-V ISA unit tests that was originally intended to test the QEMU port is used to test ISA correctness. It contains hardcoded test cases for most instructions available in the instruction set. I added a test case to cover the `fence.i` instruction, which is used to flush the instruction cache to support self-modifying code. All three execution engines, interpreter, simple DBT and optimising DBT, passed all test cases.

4.1.2 Testing Floating Point Arithmetic

Floating point arithmetic may contain more edge cases than integer instructions, so I decided that the available tests in the unit test suite were insufficient to guarantee the correctness of the software floating point library implemented with the project. I therefore

ported the Berkeley TestFloat [17] suite to perform additional tests on the floating point arithmetic. Berkeley TestFloat can generate randomized test samples that can test all common cases along with known edge cases. Both double precision and single precision floating point arithmetic implemented in this project passed all test cases generated by Berkeley TestFloat.

4.1.3 Benchmark Suites

Correctness was also tested by letting the project execute several real-world applications, checking whether the output was as expected. I was able to compile and run Dhrystone, CoreMark and SPECint benchmarks in all three execution engines without errors. All execution engines passed CoreMark's built-in checksum check. In SPECint runs, the output of execution was compared with the expected result distributed with the test suites. All benchmarks produced exactly the same output, except for `464.h264ref`. Inspecting the output of `464.h264ref` indicates that the output differs from the expected result only by least significant bits, and QEMU's execution output matches the output of this project, so I believe the difference might be attributable to precision differences in floating point units, as AMD64 internally has 80-bit precision where this project has 64-bit precision only. As the output matches QEMU's output, I decided that the mismatch could be ignored. For SPECint benchmarks, both statically-linked and dynamically-linked binaries were tested and they produced identical output.

4.1.4 Cross-validation Between Execution Engines

All three execution engines in this project are implemented separately – they share a very limited amount of code, namely decoding and non-integer instructions. As decoding and floating-point were thoroughly tested via unit tests, it is unlikely that the same bug will occur on all execution engines. Therefore another useful test is to check whether all three execution engines' behaviours are functionally equivalent. The execution engines were instrumented so that they would print out execution traces with register contents, and their outputs were then compared to see whether there were any differences. This was performed on both Dhrystone and CoreMark binaries, and the output was exactly the same for all three engines. Therefore it is reasonable to assert that the engines are functionally equivalent.

Overall, passing all four mechanisms of validation, this project is considered able to run unmodified single-threaded RISC-V Linux binaries that use only common system calls. Therefore the success criteria has been met.

4.2 Performance

The goal of the project was to develop a fast binary translator, so performance is the most important metric to measure the success of this project. All performance evaluations were performed on a workstation with two AMD Opteron™ Processor 6376 (2x16x2.3GHz cores), 128GiB of RAM, running Ubuntu 16.04.4 LTS. The workstation was shared with one other user, however given abundant resources available on the machine, interference should have been negligible. All RISC-V binaries were compiled with GCC 7.2.0.

4.2.1 Dhrystone and CoreMark

Dhrystone and CoreMark are simple benchmarks that can be run in a relative short time and are fair indicators of performance. Their simplicity allowed them to be used to guide the design directions of this project, because experimenting using larger benchmarks for each design decision would have been intractable.

I ran both benchmarks 10 times for each implementation. Dhrystone ran for 10000000 passes (the default), and the Dhrystone's DMIPS (Dhrystone MIPS) was recorded. CoreMark ran using seed 0,0,0x66 (the default) and 100000 iterations, and the number of iterations per second was recorded. I decided not to let CoreMark decide the number of iterations dynamically to reduce the number of variables and therefore reduce variance.

Tested emulators include `spike`, QEMU, `rv8` and all three execution engines implemented in this project. Native performance is also included for intuition about performance. Note that as `spike` simulates an internal wallclock and does not use the actual wallclock, `gettimeofday` does not produce correct result. Its performance was therefore estimated using the overall execution time instead of benchmarks' self-reported results. The setup and clean-up time for `spike` was estimated and removed from the execution time before calculation.

Figure 4.1 shows the 95% confidence interval of DMIPS. The confidence interval is calculated using Student's t-distribution with degree of freedom of 9, therefore is

$$\bar{x} \pm 2.262 \times \frac{s}{\sqrt{10}}$$

where $\bar{x}$ is the sample mean and s the sample standard deviation. To assert an emulator performs better than another, confidence intervals must not overlap, or two-sample one-tailed t-test should be used. In this case, non-overlapping confidence intervals in the chart suggest that the optimising DBT beats all other emulators significantly. It achieves 29.7x speedup relative to `spike` and 8.75x speedup relative to QEMU in Dhrystone.

Figure 4.2 shows the 95% confidence interval for CoreMark. The chart shows that the optimising DBT outperforms all other emulators. It achieves 16.4x speedup relative to `spike` and 2.39x speedup relative to QEMU in CoreMark.

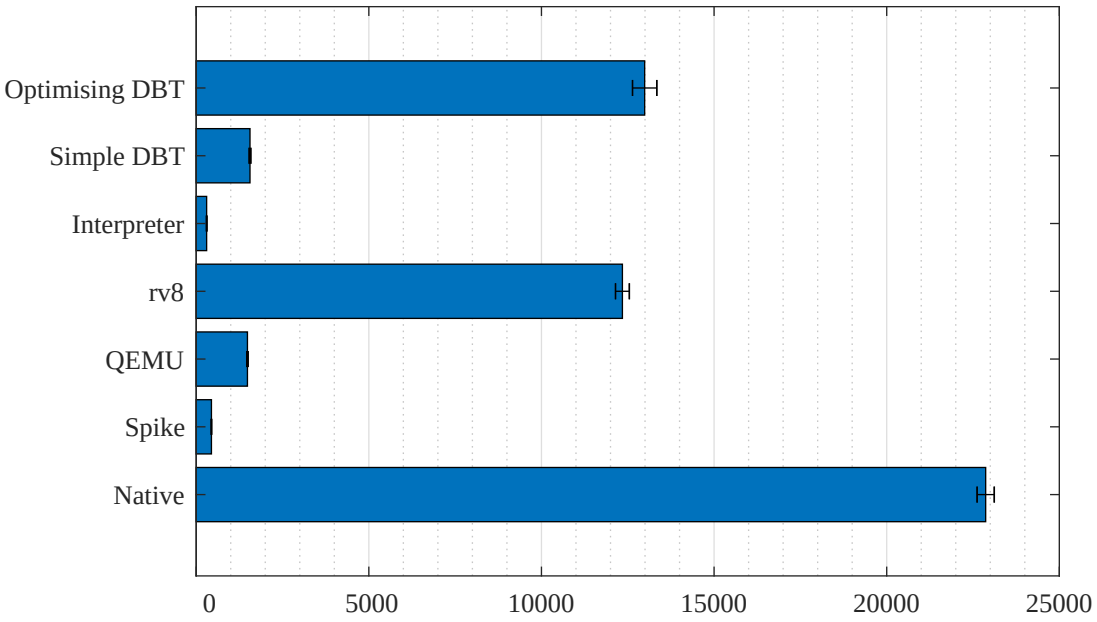

Figure 4.1: Dhrystone performance in DMIPS

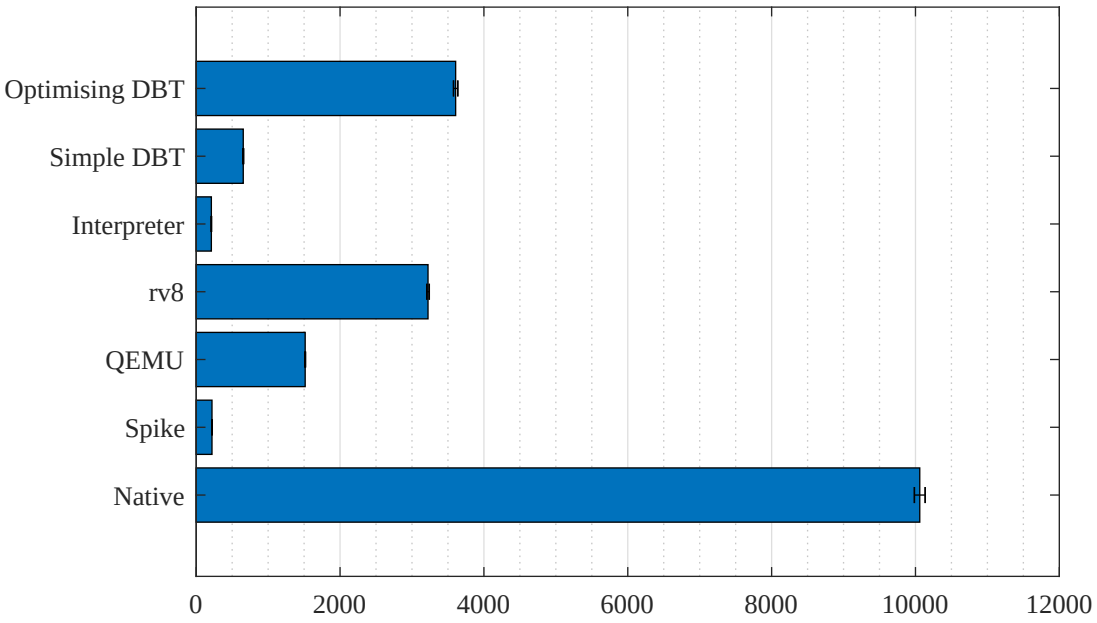

Figure 4.2: CoreMark performance in iterations per second

4.2.2 SPECint

Both Dhrystone and CoreMark are synthetic benchmarks, meaning they cannot reflect the performance in real-life situations. Therefore I picked SPECint to be the main metric for the performance evaluation, which is a compilation of practical applications that test integer performance exclusively. Benchmarks include a compiler, word processor, scientific computing application, etc.

The benchmark run uses SPEC CPU 2006's reportable configuration and with only "base" metric. Reportable configurations enforce that each benchmark is run on a large dataset

three times [13]. Execution time was recorded, and median was used for each benchmark item. SPEC uses a reference machine, Sun's Ultra Entriprise 2 produced in 1997, to normalize the performance metrics. The normalised score for each benchmark item is the ratio between the execution time and the execution time on the reference machine. The overall score is the geometric average of individual benchmark scores.

SPECint was ran once natively, once using QEMU, and once using this project's optimising DBT engine. `rv8` is not compared, as it failed to execute most of benchmarks. `spike` is not compared as it is too slow and would require days to finish one iteration.

Figure 4.3 and Figure 4.4 show the comparison between runs for each individual benchmark, in relative to the reference machine and native performance. The vertical lines denote the final score (geometric mean) achieved by each run. For consistency with SPECint's convention, error bars are not displayed. Numeric scores can be found in Appendix D.

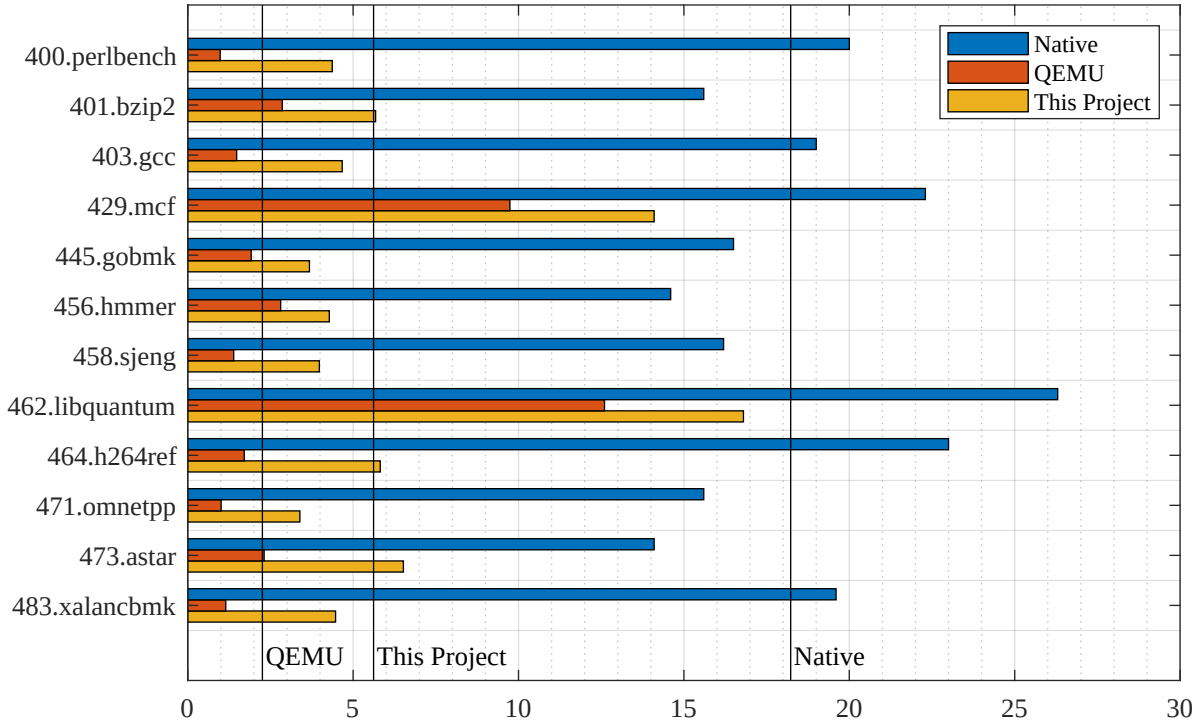

Figure 4.3: SPECint scores relative to the reference machine

Comparing the scores suggests that QEMU obtains 12.4% of native performance, while this project obtains 30.8% of native performance. Setting QEMU as the baseline, this project outperforms QEMU on every single benchmark, and an overall speedup of 2.49x is achieved.

4.3 Tuning Region Size Limit

One important tunable parameter in the optimising binary translator is the region size limit, mentioned in Section 3.3.3. If the limit is too small, then only a few blocks will

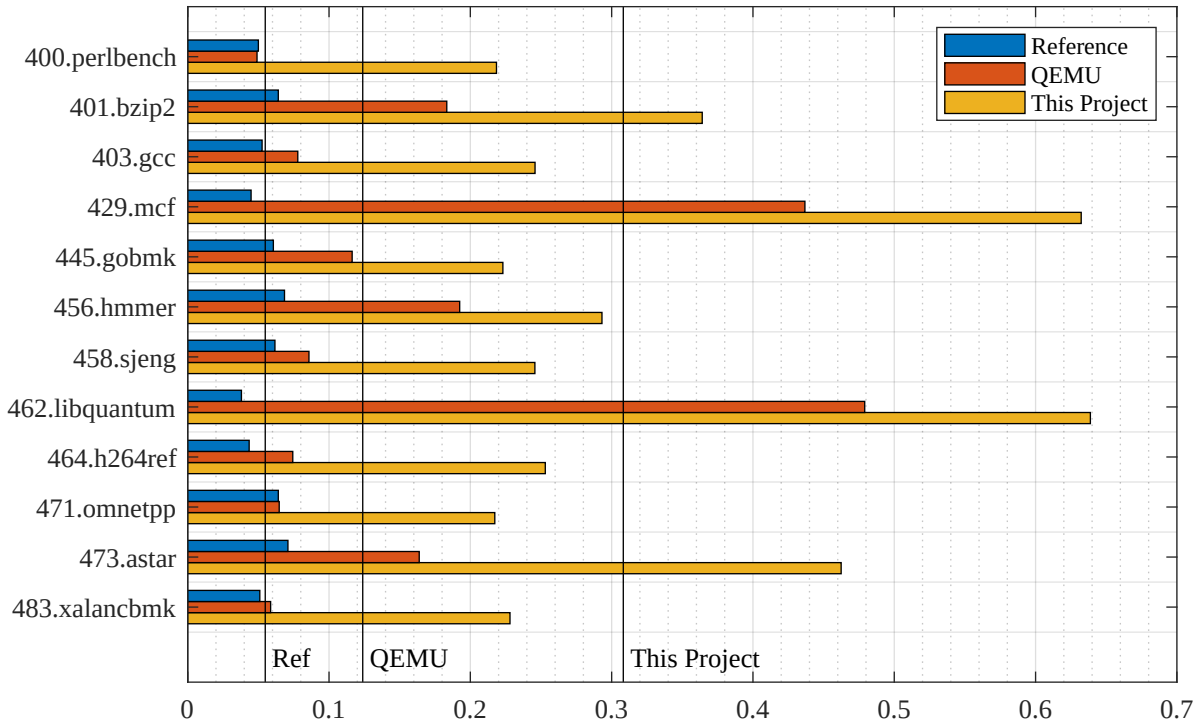

Figure 4.4: SPECint scores relative to the native performance

be included in the region, so transformations and optimisations that can be applied to the region will be limited. An extreme case is when the region size limit is 1, in which case the global load/store elimination is effectively a no-op, so performance will be poor. If the limit is too large, then the program will end up spending a lot of time processing blocks that may never be executed. An extreme case is when the region size limit is ∞, in which case all reachable blocks will be added to the region, causing a significant increase in compilation time and memory usage. It is necessary to find a trade-off limit that balances the compilation time and performance.

I decided to find the ideal region size limit experimentally. For each region size limit, I ran CoreMark 3 times with both the compilation time and the execution time recorded. The execution time can be calculated by subtracting the compilation time from the overall runtime.

It can be observed from Figure 4.5 that the execution time reduces when the region size limit increases. The reduction is very significant when the region size limit ≤ 8, then marginal improvement diminishes when the region size increases beyond 8. It is expected that when region size limit continues to increase, the execution time will continue to drop until convergence.

Figure 4.6 shows that the compilation time increases when the region size limit increases. To choose a default region size limit requires balancing compilation time and execution time. 16 is chosen to be the default value, as beyond 16 the benefit of decreasing execution time can be outweighed by increasing compilation time.

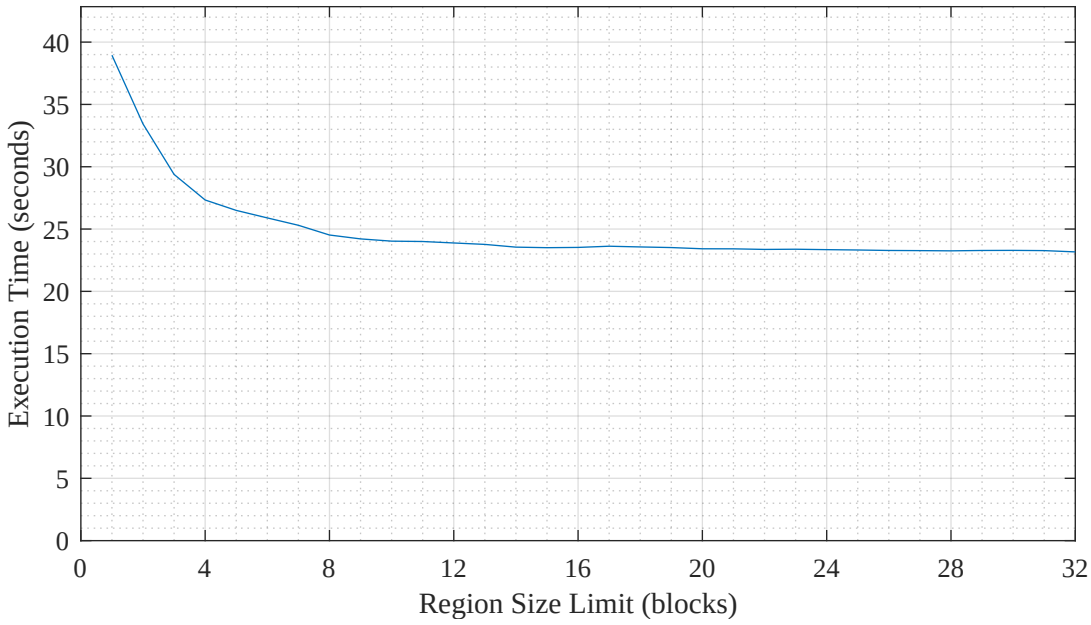

Figure 4.5: Execution time versus region size limit

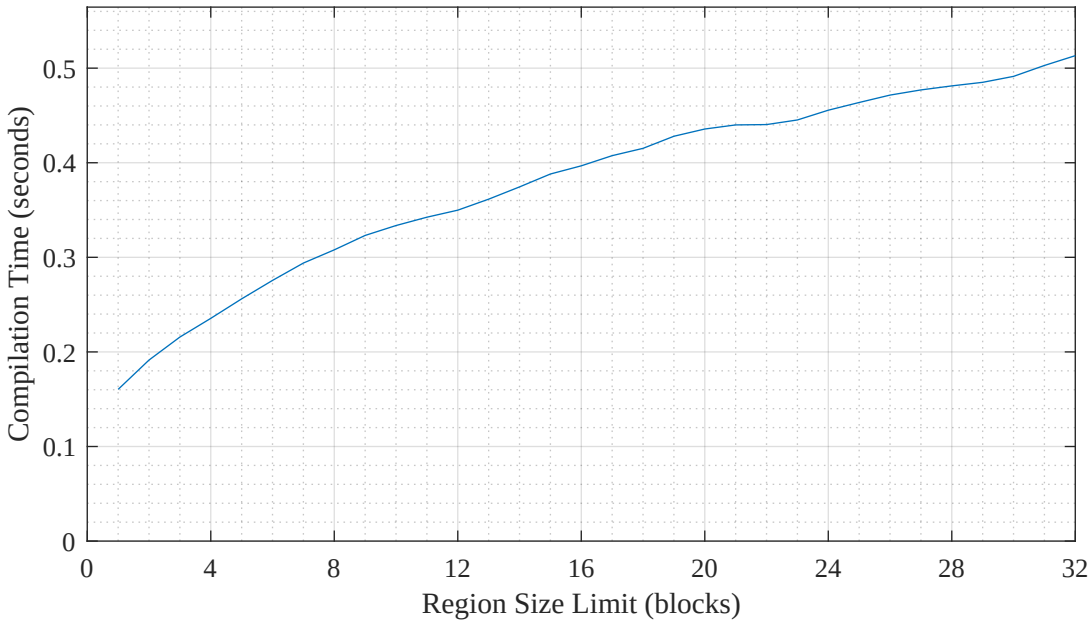

Figure 4.6: Compilation time versus region size limit

4.4 Analysis of Compilation Overhead

Though compilation time is negligible in benchmarking, it is nevertheless interesting to understand why it varies in the pattern shown in Figure 4.6. Therefore, I modified the binary translator to generate compilation statistics, seeking for explanations about the increase in compilation time when the region limit increases.

From Figure 4.7, the average region size (number of blocks per region) increases when region size limit increases. Figure 4.8 suggest that the reduction in the number of regions is much slower than the growth of region size, therefore leading to an overall increase in

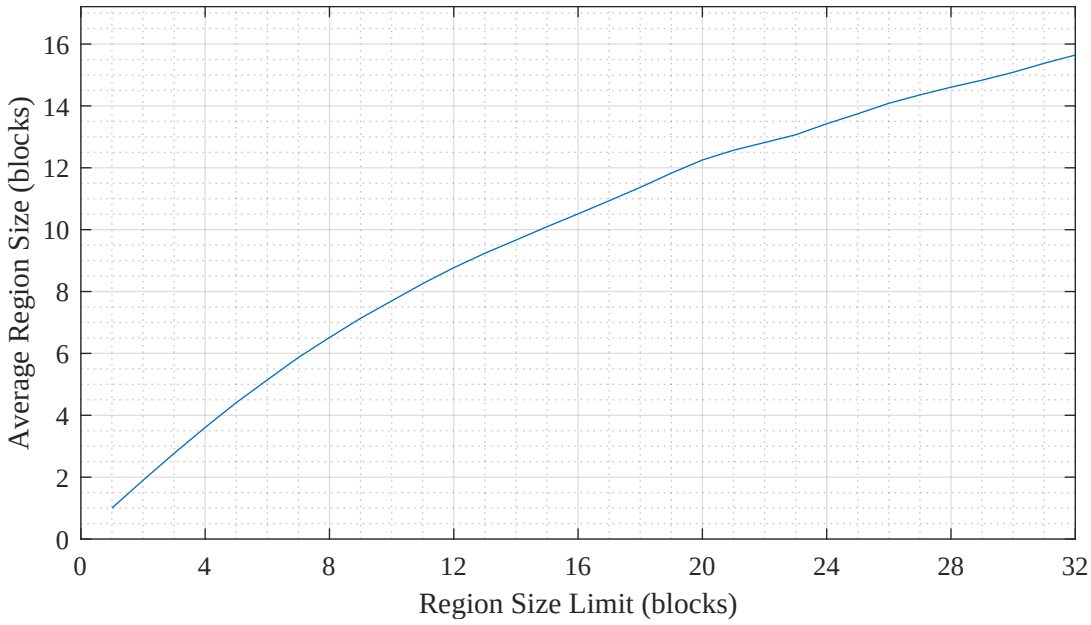

Figure 4.7: Average region size versus region size limit

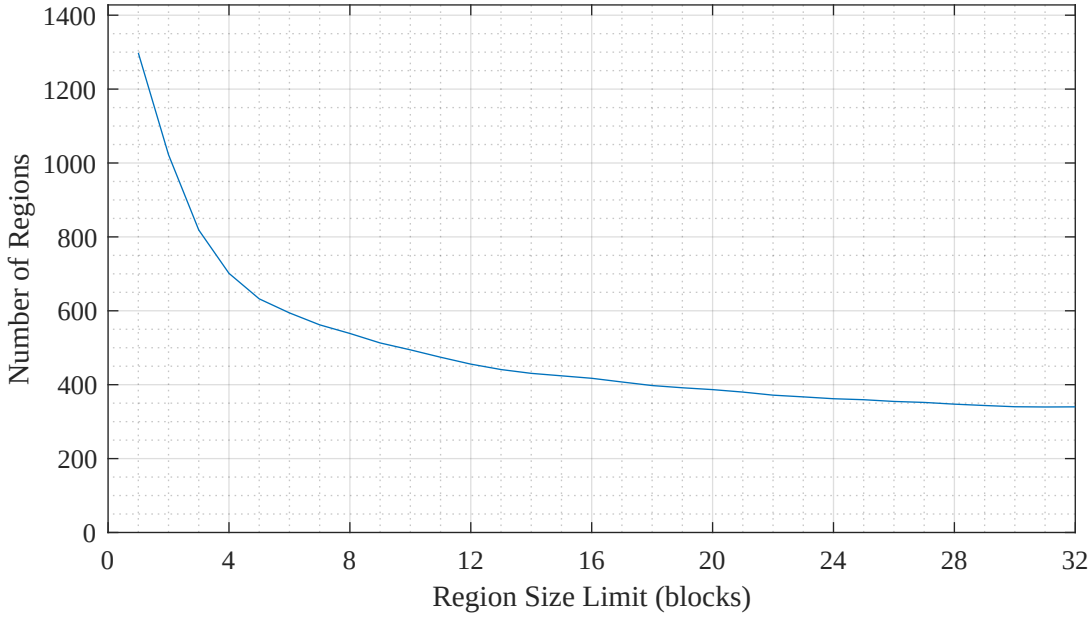

Figure 4.8: Number of regions compiled versus region size limit

the number of blocks compiled. Comparing Figure 4.9 and 4.6 shows a strong correlation between the total number of all blocks compiled and the compilation time.

The increase in the total number of blocks compiled can be explained by two factors. First, raising the region size limit will result in more blocks that are not actually executed being compiled. When the region size is 1, only executed blocks are fetched and translated. From Figure 4.10 it is clear that an increase in the region size limit leads to an increase in the number of unique blocks translated, indicating more blocks are translated but never executed.

Secondly, each block can be translated more than once. As regions are not mutually

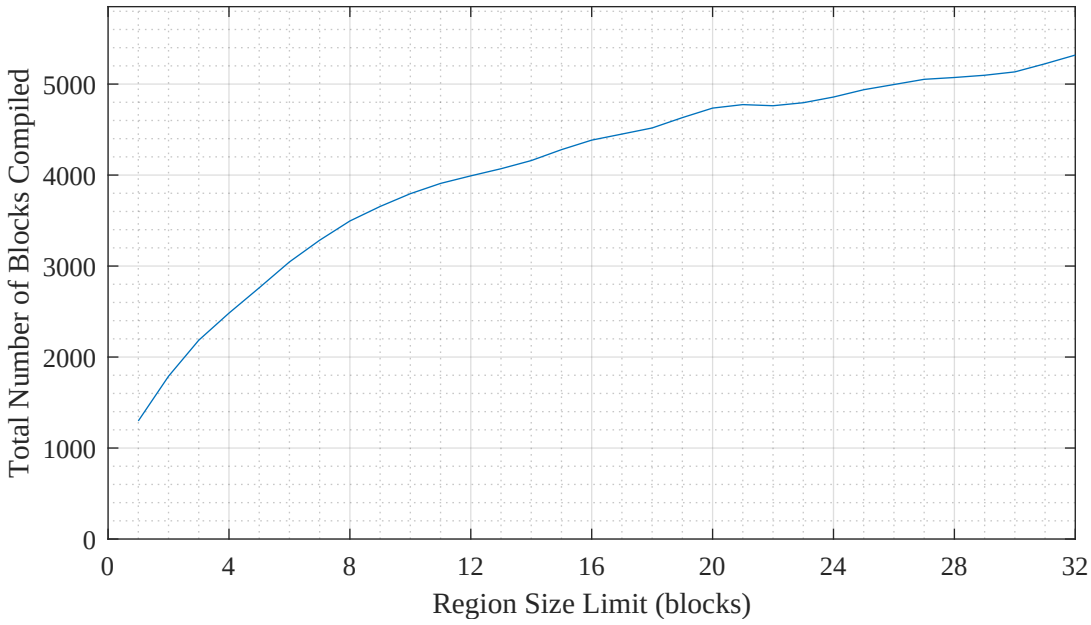

Figure 4.9: Total number of blocks compiled versus region size limit

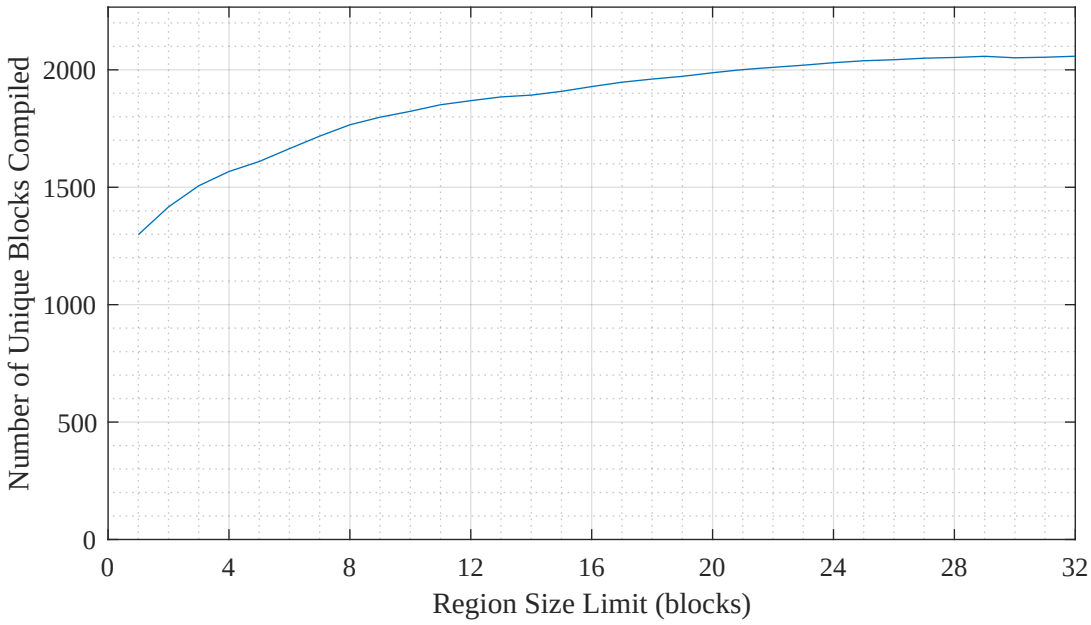

Figure 4.10: Number of unique blocks compiled versus region size limit

exclusive with each other, it is very likely that a block might be reachable from multiple paths. Therefore a block may belong to multiple regions. As each region is separately compiled and optimised, appearances in different regions must be separately translated and compiled, therefore the total number of blocks increases. Figure 4.11 shows that most blocks are compiled only once, but some blocks are compiled multiple times. When the number of times they are compiled is accounted for, their contribution towards the total number of blocks compiled can be visualised via Figure 4.12. Even a small number of blocks getting translated multiple times can significantly increase the average shown in Figure 4.13.

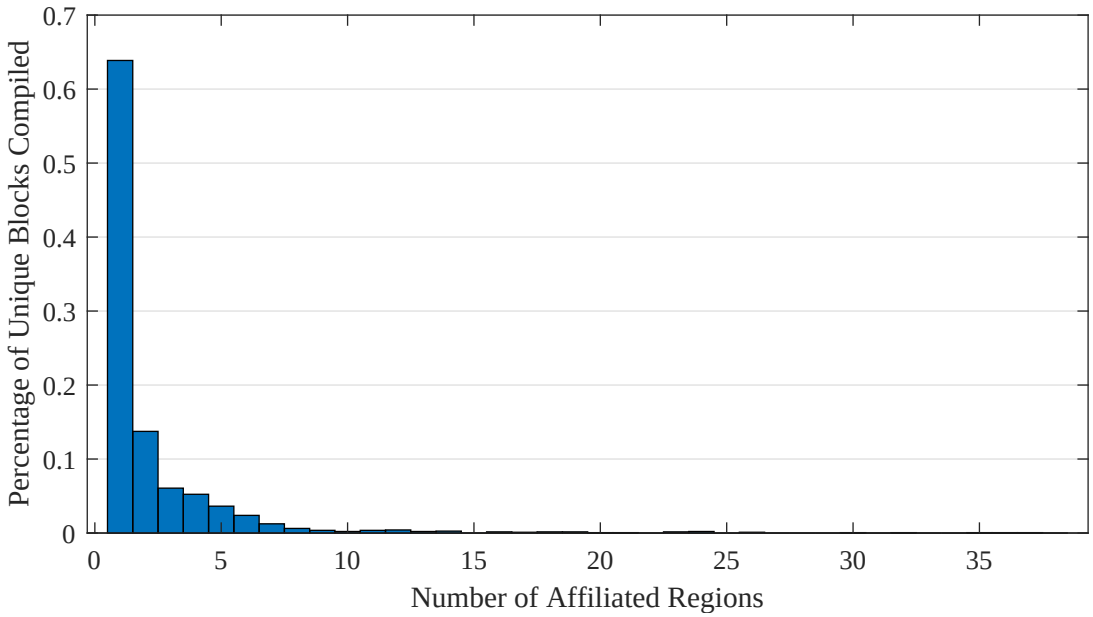

Figure 4.11: Distribution of number of affiliated regions regarding unique blocks, with region size limit 16

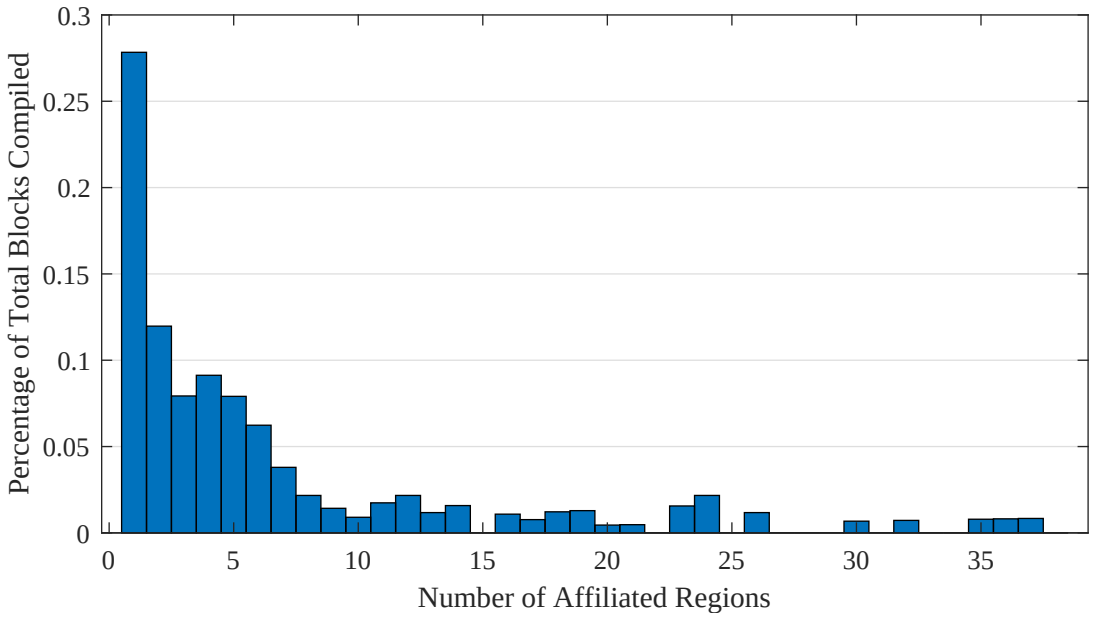

Figure 4.12: Distribution of number of affiliated regions regarding all blocks, with region size limit 16

The effect of the first factor can be possibly suppressed by utilising profile-based compilation, only adding blocks on the most probable path to the region. Profile-based compilation is listed as a possible extension but is not implemented due to time constraints. The second factor is harder to mitigate as optimisations will prevent previously translated blocks from being re-used.

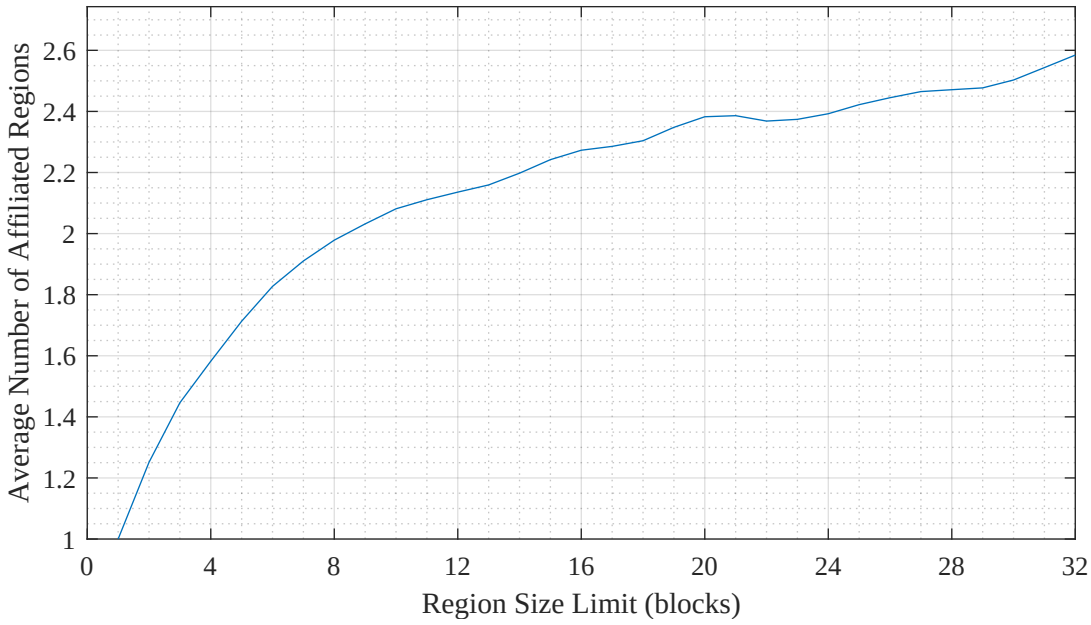

Figure 4.13: Average number of affiliated regions per block versus region size limit

4.5 Summary

Through various tests and benchmarks, I have demonstrated that the project has met its original success criteria, and significantly surpassed the original performance goals. I have also demonstrated how I tuned the default region size limit, mentioning a few interesting observations and trying to explain the underlying reasons behind the observations.

Chapter 5

Conclusion

The dissertation has described the planning, design, implementation and evaluation of a binary translator from RISC-V to AMD64. The emulator delivered surpassed the pre-set goal, outperforming all competing emulators. This project also shows that applying traditional compiler techniques in binary translators is not only viable, but also worthwhile.

5.1 Achievements

The initial functional project goal was met. The extension to make the emulator support dynamically linked binaries was implemented. The final emulator is able to execute unmodified single-threaded 64-bit RISC-V Linux binaries that use a selected subset of system calls. The binary compatibility of the emulator produced is of a comparable level with QEMU, much better than another fast binary translator rv8.

The project has also greatly exceeded the initial performance goal. Significant speedup is achieved compared to `spike`, with a 30.5x speedup in Dhrystone and 18.0x in CoreMark. During the evaluation the comparison baseline was therefore revised to be QEMU, which is much faster than `spike`. Even with the new baseline, this project achieves 9.06x speedup in Dhrystone, 2.60x in CoreMark, and 2.49x in SPECint.

5.2 Further Directions

There are a number of ways that could potentially improve this project but are not implemented due to project time and complexity constraints.

- A proper global register allocation could be implemented to replace the current naïve first-come-first-serve register allocator.
- Load and store elimination could also perform partial redundancy elimination.

- Store eliminations could be modified to not rely on the assumption that memory operations do not fault, but instead can use exception landing pads to fix register states properly.
- Trace or profile-based optimisations can be used. For example, region formation can use the frequency information to select the block to add, and compilation can be triggered only on hot paths to reduce compilation overhead.
- More frontends and backends can be implemented. The IR-based approach should allow the project to be ported easily.

Bibliography

[1] IEEE Standards Interpretations for IEEE Standard Portable Operating System Interface for Computer Environments. *IEEE Std 1003.1-1988/INT, 1992 Edition*, 1992.

[2] Itanium C++ ABI: Exception Handling. `https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html`, 2005.

[3] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, Aug 2008.

[4] Krste Asanović and David A Patterson. Instruction sets should be free: The case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[5] Fabrice Bellard. QEMU: the FAST! processor emulator. `https://www.qemu.org/`, 2005.

[6] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. *FIRM-A graph-based intermediate representation*. KIT, Fakultät für Informatik, 2011.

[7] Michael Clark. rv8: RISC-V simulator for x86-64. `https://rv8.io/`, 2017.

[8] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *ACM Sigplan Notices*, 30(3):35–49, 1995.

[9] DWARF Standards Committee. DWARF version 5 debugging format standard. `http://dwarfstd.org/`, 2017.

[10] Tool Interface Standards Committee et al. Executable and Linkable Format (ELF). *Specification, Unix System Laboratories*, 1(1):1–20, 2001.

[11] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[12] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 2: Instruction set reference*, 2016.

[13] Standard Performance Evaluation Corporation. SPEC CPU 2006 Documentation. `https://www.spec.org/cpu2006/Docs/`.

[14] David Drysdale. How programs get run: ELF binaries. `https://lwn.net/Articles/631631/`, 2015.

[15] Free Software Foundation. Using the GNU Compiler Collection (GCC): Code Gen Options. `https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html`, 2018.

[16] RISC-V Foundation. RISC-V ISA. https://riscv.org/risc-v-isa/, 2017.

[17] John Hauser. Berkeley TestFloat. http://www.jhauser.us/arithmetic/TestFloat.html, 2017.

[18] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[19] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association, 2001.

[20] Martin Richards. The Lengauer Tarjan algorithm for computing the immediate dominator tree of a flowgraph. http://www.cl.cam.ac.uk/~mr10/lengtarj.pdf, 2017.

[21] Alex Suykov. riscv-qemu-tests. https://github.com/arsv/riscv-qemu-tests, 2016.

[22] Andrew Waterman and Yunsup Lee. RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim, 2011.

[23] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.2. https://riscv.org/specifications/, 2017.

Appendix A

Source Directory Tree

This project consists of about 17400 line of codes (LoC), spread across 87 files. The directory tree of the source directory is shown below, along with lines of code associated with each file/directory.

/ (17400 LoC)

- include (5131 LoC)
  - emu (247 LoC)
    - mmu.h (54 LoC)
    - state.h (79 LoC)
    - typedef.h (14 LoC)
    - unwind.h (100 LoC)
  - ir (832 LoC)
    - analysis.h (177 LoC)
    - builder.h (89 LoC)
    - node.h (420 LoC)
    - pass.h (61 LoC)
    - visit.h (85 LoC)
  - main (141 LoC)
    - dbt.h (45 LoC)
    - executor.h (10 LoC)
    - interpreter.h (26 LoC)
    - ir_dbt.h (46 LoC)
    - signal.h (14 LoC)

- riscv (984 LoC)
  - abi.h (497 LoC)
  - basic_block.h (27 LoC)
  - context.h (32 LoC)
  - csr.h (24 LoC)
  - decoder.h (34 LoC)
  - disassembler.h (25 LoC)
  - frontend.h (15 LoC)
  - instruction.h (90 LoC)
  - opcode.h (223 LoC)
  - typedef.h (17 LoC)
- softfp (1244 LoC)
  - float.h (1244 LoC)
- util (1012 LoC)
  - assert.h (72 LoC)
  - bit_op.h (70 LoC)
  - bitfield.h (82 LoC)
  - code_buffer.h (48 LoC)
  - format.h (78 LoC)
  - functional.h (30 LoC)
  - int128.h (62 LoC)
  - int_size.h (39 LoC)
  - macro.h (11 LoC)
  - memory.h (26 LoC)
  - multiset.h (79 LoC)
  - reverse_iterable.h (29 LoC)
  - safe_memory.h (21 LoC)
  - scope_exit.h (43 LoC)
  - select_int.h (83 LoC)
  - small_vector.h (239 LoC)

  - x86 (660 LoC)
    - backend.h (162 LoC)
    - builder.h (206 LoC)
    - decoder.h (35 LoC)
    - disassembler.h (22 LoC)
    - encoder.h (60 LoC)
    - instruction.h (104 LoC)
    - opcode.h (71 LoC)
  - config.h (11 LoC)
- src (12047 LoC)
  - emu (1129 LoC)
    - elf_loader.cc (278 LoC)
    - mmu.cc (43 LoC)
    - state.cc (33 LoC)
    - syscall.cc (775 LoC)
  - ir (2448 LoC)
    - block_analysis.cc (252 LoC)
    - dominance.cc (308 LoC)
    - dot_printer.cc (221 LoC)
    - load_store_elimination.cc (517 LoC)
    - local_load_store_elimination.cc (74 LoC)
    - local_value_numbering.cc (588 LoC)
    - lowering.cc (74 LoC)
    - node.cc (208 LoC)
    - scheduler.cc (182 LoC)
    - visit.cc (24 LoC)
  - main (2484 LoC)
    - dbt.cc (1700 LoC)
    - interpreter.cc (62 LoC)
    - ir_dbt.cc (361 LoC)

        * main.cc (246 LoC)
        * signal.cc (115 LoC)
    - riscv (3002 LoC)
        * decoder.cc (1093 LoC)
        * disassembler.cc (454 LoC)
        * frontend.cc (414 LoC)
        * step.cc (1041 LoC)
    - softfp (11 LoC)
        * float.cc (11 LoC)
    - util (260 LoC)
        * assert.cc (9 LoC)
        * code_buffer.cc (37 LoC)
        * format.cc (165 LoC)
        * safe_memory.cc (49 LoC)
    - x86 (2679 LoC)
        * code_generator.cc (647 LoC)
        * decoder.cc (156 LoC)
        * disassembler.cc (219 LoC)
        * encoder.cc (804 LoC)
        * lowering.cc (152 LoC)
        * register_allocator.cc (701 LoC)
    - feature.cc (34 LoC)
- visualizer (131 LoC)
    - generate.js (29 LoC)
    - template.html (102 LoC)
- Makefile (91 LoC)

Appendix B

Sample Code

B.1 Trap Handling

As mentioned in Section 3.1.2, traps are turned into exceptions. This is achieved by the following code:

```
void handle_fault(int sig) {
    ASSERT(sig == SIGSEGV || seg == SIGBUS);

    sigset_t x;
    sigemptyset(&x);
    sigaddset(&x, sig);
    sigprocmask(SIG_UNBLOCK, &x, nullptr);
    throw Segv_exception {sig};
}

void setup_fault_handler() {
    struct sigaction act;
    memset(&act, 0, sizeof(act));
    act.sa_handler = handle_fault;
    sigaction(SIGSEGV, &act, NULL);
    sigaction(SIGBUS, &act, NULL);
}
```

Listing B.1: Trap handling in main/signal.cc

The following code is (the only code) compiled with `-fnon-call-exceptions`, and is referenced by other parts of the project when guest memory needs to be accessed.

```
template<typename T>
T safe_read(void* pointer) {
    T ret;
    memcpy(&ret, pointer, sizeof(T));
    return ret;
```

```
}

template<typename T>
void safe_write(void *pointer, T value) {
    memcpy(pointer, &value, sizeof(T));
}

void safe_memcpy(void *dst, const void *src, size_t n) {
    std::byte *c_dst = reinterpret_cast<std::byte*>(dst);
    const std::byte *c_src = reinterpret_cast<const std::byte*>(
       src);
    for (size_t i = 0; i < n; i++, c_dst++, c_src++) {
        *c_dst = *c_src;
    }
}

void safe_memset(void *memory, int byte, size_t size) {
    unsigned char data = static_cast<unsigned char>(byte);
    unsigned char* pointer = reinterpret_cast<unsigned char*>(
       memory);
    for (unsigned char* end = pointer + size; pointer < end;
       pointer++) {
        *pointer = data;
    }
}

template uint8_t safe_read<uint8_t>(void*);
template uint16_t safe_read<uint16_t>(void*);
template uint32_t safe_read<uint32_t>(void*);
template uint64_t safe_read<uint64_t>(void*);
template void safe_write<uint8_t>(void*, uint8_t);
template void safe_write<uint16_t>(void*, uint16_t);
template void safe_write<uint32_t>(void*, uint32_t);
template void safe_write<uint64_t>(void*, uint64_t);
```

Listing B.2: Excerpt from util/safe_memory.cc

B.2 Environment Emulation

As mentioned in Section 3.1.3, `brk` is entirely emulated. The emulation code for `brk` is attached below.

```
if (arg0 < state::original_brk) {
    // Cannot reduce beyond original_brk
} else if (arg0 <= state::heap_end) {
    if (arg0 > state::brk) {
```

```
        zero_memory(state::brk, arg0 - state::brk);
    }
    state::brk = arg0;
} else {
    reg_t new_heap_end = std::max(state::heap_start, (arg0 +
       page_mask) &~ page_mask);

    // The heap needs to be expanded
    reg_t addr = guest_mmap(
        state::heap_end, new_heap_end - state::heap_end,
        PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, -1, 0
    );

    if (addr != state::heap_end) {
        // We failed to expand the brk.
        guest_munmap(addr, new_heap_end - state::heap_end);

    } else {

        // Memory should be zeroed here as this is expected by
           glibc.
        zero_memory(state::brk, state::heap_end - state::brk);
        state::heap_end = new_heap_end;
        state::brk = arg0;
    }
}

reg_t ret = state::brk;
if (state::strace) {
    util::log("brk({}) = {}\n", pointer(arg0), pointer(ret));
}
return ret;
```

Listing B.3: `brk` emulation in emu/syscall.cc

B.3 Exception Handling Frames

As mentioned in Section 3.2.2, a manually crafted exception handling frame template is used. The listing below shows the template in a format used by `objdump`. The actual template in the source code is encoded in binary.

```
00000000 000000000000001C 00000000 CIE
  Version:               1
  Augmentation:          "zPL"
  Code alignment factor: 1
  Data alignment factor: -8
```

```
  Return address column: 16
  Augmentation data:     0a 00 $(8 byte address of personality)
     00

  DW_CFA_def_cfa: r7 (rp) ofs 8
  DW_CFA_offset: r16 (rip) at cfa-8

00000020 0000000000000028 00000024 FDE cie=00000000 pc=$(address
   of pc start)..$(address of pc end)
  Augmentation data:     08 $(8 byte address of LSDA)

  DW_CFA_advance_loc: 1
  DW_CFA_def_cfa_offset: 16
  DW_CFA_offset: r6 (rbp) at cfa-16
  DW_CFA_def_cfa_offset: $(stack_size + 16)
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
```

Listing B.4: Exception handling frame template

Appendix C

Sample Output

The following C code is the function used to calculate Fibonacci numbers:

```
long fib(long v) {
    if (v <= 1) return v;
    long prev = 0, curr = 1;
    for (long i = 2; i <= v; i++) {
        long next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

Listing C.1: Fibonacci number calculation function `fib`

It produces the following RISC-V assembly code:

```
    li      a5,1
    ble     a0,a5,.2
    addi    a2,a0,1
    li      a4,2
    li      a3,0
.1:
    add     a0,a3,a5
    addi    a4,a4,1
    mv      a3,a5
    mv      a5,a0
    bne     a2,a4,.1
.2:
    ret
```

Listing C.2: RISC-V assembly of `fib`

The optimising binary translator decodes the RISC-V binary and produces an optimised IR graph shown in Figure 3.6. After register allocation and code generation, the following

AMD64 assembly is produced.

```
    push    rbp
    mov     rbp, rdi
    mov     qword [rbp+0x78], 0x1
    mov     rax, qword [rbp+0x50]
    cmp     rax, 0x1
    jle     .2
    mov     rax, qword [rbp+0x50]
    add     rax, 0x1
    mov     qword [rbp+0x60], rax
    mov     qword [rbp+0x50], 0x1
    mov     qword [rbp+0x70], 0x3
    mov     qword [rbp+0x68], 0x1
    mov     qword [rbp+0x78], 0x1
    cmp     rax, 0x3
    jz      .2
    mov     rcx, rax
    mov     rdx, 0x1
    mov     rsi, 0x3
    mov     rax, 0x1
.1:
    add     rdx, rax
    mov     qword [rbp+0x50], rdx
    add     rsi, 0x1
    mov     qword [rbp+0x70], rsi
    mov     qword [rbp+0x68], rax
    mov     qword [rbp+0x78], rdx
    cmp     rcx, rsi
    jz      .2
    xchg    rax, rdx
    jmp     .1
.2:
    mov     rax, qword [rbp+0x8]
    and     rax, -0x2
    mov     qword [rbp+0x200], rax
    pop     rbp
    xor     eax, eax
    ret
```

Listing C.3: AMD64 assembly generated for `fib`

Appendix D

SPECint Scores

The tables below show the recorded execution time and computed scores from each run. The ratio is displayed to 3 significant places.

	First Run		Second Run		Third Run		Median	
Benchmark	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench	**489**	**20.0**	481	20.3	491	19.9	489	20.0
401.bzip2	616	15.7	621	15.5	**620**	**15.6**	620	15.6
403.gcc	420	19.2	**423**	**19.0**	424	19.0	423	19.0
429.mcf	421	21.7	408	22.4	**409**	**22.3**	409	22.3
445.gobmk	**636**	**16.5**	637	16.5	633	16.6	636	16.5
456.hmmer	638	14.6	**639**	**14.6**	644	14.5	639	14.6
458.sjeng	750	16.1	746	16.2	**749**	**16.2**	749	16.2
462.libquantum	786	26.4	810	25.6	**789**	**26.3**	789	26.3
464.h264ref	961	23.0	981	22.6	**964**	**23.0**	964	23.0
471.omnetpp	400	15.6	418	15.0	**401**	**15.6**	401	15.6
473.astar	**499**	**14.1**	510	13.8	497	14.1	499	14.1
483.xalancbmk	345	20.0	**352**	**19.6**	353	19.3	352	19.6
							Score	18.2

Table D.1: SPECint results of native run

	First Run		Second Run		Third Run		Median	
Benchmark	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench	9652	1.01	10818	0.903	**9958**	**0.981**	9958	0.981
401.bzip2	3536	2.73	**3379**	**2.86**	3167	3.05	3379	2.86
403.gcc	5888	1.37	**5449**	**1.48**	5328	1.51	5449	1.48
429.mcf	994	9.17	**937**	**9.74**	924	9.87	937	9.74
445.gobmk	**5454**	**1.92**	5761	1.82	5306	1.98	5454	1.92
456.hmmer	3227	2.89	3419	2.73	**3322**	**2.81**	3322	2.81
458.sjeng	**8701**	**1.39**	8708	1.39	8345	1.45	8701	1.39
462.libquantum	1598	13.0	1707	12.1	**1645**	**12.6**	1645	12.6
464.h264ref	13737	1.61	**12919**	**1.71**	12248	1.81	12919	1.71
471.omnetpp	**6196**	**1.01**	6676	0.936	5998	1.04	6196	1.01
473.astar	2925	2.40	3131	2.24	**3045**	**2.31**	3045	2.31
483.xalancbmk	5782	1.19	6361	1.08	**6018**	**1.15**	6018	1.15
							Score	2.26

Table D.2: SPECint results of QEMU

	First Run		Second Run		Third Run		Median	
Benchmark	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench	**2237**	**4.37**	2254	4.34	2174	4.49	2237	4.37
401.bzip2	1588	6.08	**1700**	**5.68**	1751	5.51	1700	5.68
403.gcc	1698	4.74	**1723**	**4.67**	1765	4.56	1723	4.67
429.mcf	597	15.3	653	14.0	**645**	**14.1**	645	14.1
445.gobmk	**2854**	**3.68**	2969	3.53	2661	3.94	2854	3.68
456.hmmer	2351	3.97	**2182**	**4.28**	2122	4.40	2182	4.28
458.sjeng	**3038**	**3.98**	3063	3.95	2904	4.17	3038	3.98
462.libquantum	**1235**	**16.8**	1200	17.3	1249	16.6	1235	16.8
464.h264ref	**3803**	**5.82**	3966	5.58	3746	5.91	3803	5.82
471.omnetpp	**1845**	**3.39**	1887	3.31	1813	3.45	1845	3.39
473.astar	1079	6.50	1042	6.73	**1077**	**6.52**	1077	6.52
483.xalancbmk	1552	4.45	**1542**	**4.47**	1431	4.82	1542	4.47
							Score	5.62

Table D.3: SPECint results of this project

Appendix E

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Dynamic Binary Translator for RISC-V

Xuan Guo, Peterhouse

Originator: Xuan Guo

10 October 2017

Project Supervisor: Dr Timothy Jones

Director of Studies: Dr Robert Mullins

Project Overseers: Dr Sean Holden & Dr Neel Krishnaswami

Introduction

RISC-V is a recent innovation in general-purpose instruction set architecture (ISA). Originally designed to support computer architecture research and education, it was soon embraced by the industry and eventually became an open architecture standard for use of everyone. It is designed with modularity and simplicity in mind, and is designed to serve all devices: ranging from embedded systems to datacenter-scale computers.

Even though RISC-V is open and embraced by the industry, it is still disadvantageous compare to existing architectures, e.g. AMD64 or ARM in terms of ecosystems. Most modern desktops and servers are powered by processors running AMD64 ISA, and these systems cannot run RISC-V binaries natively. This essentially creates a barrier for developing for or porting software to RISC-V ISA. For the ecosystem to grow, there exists a

need for a RISC-V emulator that has both good performance and interoperability with existing environments.

This project aims to develop an emulator that is capable of running unmodified RISC-V Linux binaries directly on an AMD64 Linux system, and it shall be utilizing dynamic binary translation techniques to achieve better performance than simple interpreters.

Terminology

An **emulator** is a program that can run program compiled for a particular ISA. The architecture emulated is called the **guest** architecture, and the architecture that an emulator runs on is called the **host** architecture. An emulator can be further classified as an interpreter or a binary translator.

An **interpreter** is an emulator that works by decoding and emulating instruction once at a time.

A **binary translator** is an emulator that works by translating binaries from the guest ISA to host ISA a block or multiple blocks at a time, and executing the translated binaries. It can be further divided to static binary translator, which translates all code ahead-of-time, or **dynamic binary translator** which translates code in runtime when the code is actually used.

RISC-V is a modular ISA, divided into base integer ISA and optional extensions. At the time of writing, stable base ISAs include **RV32I** and **RV64I**, which uses different register widths, and stable standard extensions include **M**, **A**, **F**, **D**, **C**, which stand for multiplication, atomic, floating-point, double precision floating-point and compressed instruction extensions. IMAFD is collectively called **G**.

Starting point

I already have basic computer architecture knowledge from Part IB Computer Design course, and basic compiler construction knowledge from Part IB Compiler Construction course and my previous hobby projects. I consider myself experienced C++ user, as I regularly use C++11 and above for my hobby projects.

The following projects are state-of-art RISC-V emulators that could serve as the basis of comparison and evaluation:

- Currently a reference interpreter `riscv-isa-sim` is implemented by the RISC-V Foundation. With various coding tricks applied, it is the fastest known interpreter.
- A `QEMU` port of RISC-V is also available, which utilizes dynamic binary translation, and it is the fastest known emulator.

- `rv8` is another RISC-V to AMD64 binary translator. However it is still immature and under development.

The aim of this project is to implement binary translation, so starting from existing binary translators is not considered. Also, tricks used in `riscv-isa-sim` make it hard to perform large-scale changes such as implementing binary translation. Considering all these factors, this project intends to start from scratch without extending from any of existing codebases.

The RISC-V Foundation also ported toolchains to RISC-V, including binutils, gcc and gdb. These tools will be used throughout the project, and they are helpful for generating binaries for testing.

Resources required

This project requires no special resources. Any Linux environment on AMD64 will suffice. For the sake of convenience, I shall mainly use my own laptop that runs Microsoft Windows. Windows subsystem for Linux will be used as development and evaluation environment. In case that it fails, I shall be using MCS Linux as the backup.

All documents, including source code and any paper work will be synchronized to OneDrive for Business. Source code will in addition be version-controlled using git, and backed up to a private GitHub repository.

Work to be done

The project is split into three implementation phases, each phase is ended with a milestone. Evaluation should be constantly performed while implementing to determine the standard conformance performance bottlenecks and hint optimisations directions. At the end of phase 3, a systematic evaluation should be performed to compare performance benchmarks in different configurations and against existing emulators.

Phase 1 – Interpreter

In the first phase, a basic interpreter will be implemented. Even though the project aims to build a binary translator and targets better performance than simple interpreters, building an interpreter first can serve as a basis to build the binary translator on.

The interpreter will accept a statically-linked RISC-V Linux binaries that use most fundamental system calls (e.g. fstat, open, read, write, brk, exit). Binaries will be compiled targeting RV64GC (i.e. RV64I + MAFDC extensions). Decoder and disassembler will be implemented as part of building an interpreter.

Phase 2 – Basic Block Binary Translation

The next phase is building the binary translator. The binary translator needs only to work on RV64IMC instructions, and instructions in AFD extensions can remain interpreted. The early stage of the binary translator will work on basic blocks, i.e. contiguous instructions with no control flow changes. By building upon the interpreter, instructions can be incrementally implemented by calling into the interpreter on un-implemented instructions. Register allocation and optimisations are not necessary at this stage, and all emulated registers can be placed in heap or stack for the ease of implementation.

Phase 3 – Inter-block Binary Translation, Register Allocation, Optimisations

The binary translator should be extended further so that nearby basic blocks can be translated together, e.g. branch targets of the same `if` should be translated together for better optimisation opportunities. Register allocations and local or global optimisations techniques could then be applied to increase the performance. At the end of this phase, the binary translator should be almost feature-complete.

Success criteria

The project will be a success if I have implemented a dynamic binary translator that can run unmodified statically-linked, single-threaded RISC-V Linux binaries that uses only common system calls. Ideally it would have better performance than the reference interpreter implementation.

Possible extensions

If I achieve my main result early I shall try the following extensions:

1. Implement binary translation for RISC-V standard extension AFD. This is placed as an extension since RISC-V and AMD64 have different memory models and floating-point register models, and having these as the main goal is highly risky.
2. Implement parallel code generation. With optimisations added, code generation is likely to be slow. Therefore, the task could be offloaded to a different thread, and the main thread can either be operating in interpreter mode or run a less optimised binary before the translation completes.
3. Implement dynamic profile-guided re-compilation. Hot code can be spotted via tracing, and the code could be re-compiled by applying more optimisations to boost

performance. Some aggressive optimisations can also be made if de-optimisation is implemented.

4. Implement signals, multi-threading, dynamic-linking and other system calls. This depends on the A extension, so it will be blocked by extension 1.
5. Implement full system emulation in addition to user space emulation.

Timetable

Planned starting date is 22/10/2017.

1. **Michaelmas weeks 3–4** Compile and get experience on RISC-V toolchains. Read RISC-V specifications, books about optimising compilers and papers about binary translation.
2. **Michaelmas weeks 5–6** Implement RV64GC interpreter and necessary system calls to run the benchmark binary.
3. **Michaelmas weeks 7–8** Implement RV64IMC basic binary translator on basic blocks, based on the interpreter.
4. **Christmas vacation** Start implementing binary translation across basic blocks. Experiment on optimisation techniques.
5. **Lent weeks 0–2** Write progress report. Continue working on optimisations.
6. **Lent weeks 3–4** Evaluate the quality of generated code, run experiments in different configurations and achieve main project goal.
7. **Lent weeks 5–6** Write dissertation main chapters.
8. **Lent weeks 7–8** Extensions.
9. **Easter vacation** Extensions and dissertation chapters about extensions.
10. **Easter weeks 0–2** Further evaluation and complete dissertation.
11. **Easter weeks 3** Proofreading and submission.