CHAIR OF COMPUTER ARCHITECTURE AND PARALLEL SYSTEMS

Großpraktikum Rechnerarchitektur

Static binary translation from RISC-V to x86_64

Summer semester 2021

Lukas Döllerer Jonathan Hettwer Johannes Maier Tobias Schwarz
Felix Solcher

Contents

1. Motivation and Problem Statement

In order to natively execute machine code, a processor which runs this machine code is needed. But there are many different processor families with different instruction set architectures (ISA) for which programs are developed. To run a program which is written for an ISA one does not have access to a system running this ISA, one can translate it and execute it on another system. There are three main approaches to do this:

The first option for executing programs written for a different ISA is to use an emulator. It emulates a different execution environment by interpreting the program's instructions just-in-time [2]. The development is simple, but it is a rather slow method.

The second option is to use dynamic binary translation (DBT). It works like an emulator but uses caching to store already translated instruction sequences for future use. This increases the execution speed compared to an emulator because often used instructions do not need to be translated multiple times. [2]

As the third option, one can statically translate the full binary. The program is rewritten for the target system's ISA. Translation and execution are separated phases, which means the binary only has to be translated once for running it multiple times. This increases execution speed compared to DBT. This approach is called static binary translation (SBT). [2]

The current rise of RISC-V processors makes it probable that we will see a need for RISC-V to x86_64 binary translation in the near future. Even though x86_64 is still a more widely used ISA than RISC-V, a lot of companies invest in a shift towards the RISC-V ISA. This revolution makes ISA incompatibilities a future problem we want to solve today. [9] Another important use case for the SBT is translating binaries where the source code was lost. Programs without source code can not be compiled for another target ISA, but they can be translated using SBT.

Therefore we developed a static binary translator from RISC-V to x86_64 which uses an intermediate representation (IR). The instruction decoding and code generation are separated by using as the IR as a abstract data structure. Our aim is to be faster than the popular dynamic binary translator QEMU.

The following chapters will provide some background information for the RISC-V architecture, the relevant differences to the x86_64 architecture (Section 2) and our general approach to static binary translation (Section 3). Afterwards, the distinct parts of the translator, namely our intermediate representation (Section 4), the RISC-V lifter (Section 5), the optimizer (Section 6), and the x86_64 code generator (Section 7) are further explained. Section 8 describes our approach to support floating point RISC-V instructions and the present difficulties. The last section (Section 9) evaluates the quality and performance of our implementation using benchmarks tests.

2. Background

This chapter provides important background information, which is required for our approach. This includes an overview over the RISC-V instructions, standard calling

convention [1] and registers, as well as relevant differences between the RISC-V and the x86_64 architecture.

2.1. Short Overview of RISC-V

According to the RISC-V specification, RISC-V is an open and freely accessible ISA, which consists of a base ISA for integer computations. It also offers general purpose software development supporting extensions, which are optional. There is also the option to implement custom extensions. The RISC-V ISA is available for building 32-bit and 64-bit based applications and operating system kernels. Because of its minimalistic nature, it is well suited for custom hardware implementations. [16]

Currently, there are standard extensions for multiplication and division (M), for atomic instructions (A), for floating point arithmetic (F,D), for compressed instructions to reduce the code size (C) and for control and status registers (Ziscr). Together with the RV64 Base ISA, they are called `rv64imacfd` or short `rv64g`. More standard extensions are planned. [16]

RV64I provides 31 64-bit wide integer registers (`x1` − `x31`) and a hard wired zero register (`x0`). The floating point extensions add instructions for 32 floating points registers (`f0` − `f31`). They also add a floating point control and status register (`fcsr`). The instruction words have a fixed width of 32-bit, with the C extension introducing instructions with 16-bit wide instruction words. The standard calling convention defines `x1` as the link register and `x5` as the alternative link register. Link registers store the return address of jumps which target a subroutine and intend to return the parent routine. [16]

2.2. Differences between RISC-V and x86_64

When comparing the RISC-V and the x86_64 ISAs, most of the differences can be derived from their different design ideologies. The x86_64 ISA is built to describe a CISC, a *Complex Instruction Set Computer*. This decision results in processors that have a limited number of registers, richer instruction sets and variable length instructions. Because of the complex instruction functions and formats, logic is often implemented in a mix between microcode and hard coded logic, executing instructions over a number of CPU clock cycles. [8]

As the name suggests, the RISC-V ISA describes a *Reduced Instruction Set Computer*. This kind of microprocessor is characterized by a larger number of registers, its load-/store architecture, fixed-length instruction words and a generally limited amount of instructions and instruction formats. [8]

These major design differences play an important role during the binary translation. The following paragraph describes two relevant differences in greater detail.

The RISC-V ISA defines a load/store architecture, meaning that a single instruction can either manipulate the memory or perform an operation that does not access the memory. The x86_64 ISA does not have this restriction and defines arithmetic and

[1] When referring to the *standard calling convention*, the RVG calling convention is meant.

logical instructions which operate directly on memory. Also, RISC-V base instructions are currently always 32-bit wide (16-bit for compressed instructions) which makes loading wide immediates difficult. These are either combined by two or more separate immediate loading operations or loaded relative to the current instruction pointer. [16] In comparison, x86_64 instructions can contain immediate values which are up to 64-bit wide, because of their variable instruction lengths [10].

Assemblers make this difference less complicated for developers by introducing so called *pseudoinstructions*. These are instructions which perform simple tasks like loading an immediate, jumping to an address, or calling a subroutine. Most of them can't be directly expressed as a single RISC-V instruction and are thus replaced with a common pattern. This gives a translator the power to detect these patterns and efficiently convert them to a more compact, x86_64 representation.

These differences are necessary knowledge for the translation. They give the opportunity to generate efficient and optimized code. For example, some RISC-V instruction sequences can be merged to a single x86_64 instruction.

2.3. Inspiration for the Intermediate Representation

Languages for intermediate representations (IR) are often used in compilers, which means that highly advanced concepts already exist. The LLVM assembly language is a established IR, known for its deep integration into the LLVM ecosystem with existing compiler and debugger backends and its well defined language reference. These advantages however have a price. The LLVM assembly language is a complex IR with a lot of features which are not required for the purposes of this project. The used IR is inspired by the LLVM assembly language, tailored to serve the use case of lifting binaries. The parallels between our IR and the LLVM assembly language are the static typing, single static assignment (SSA) variables and machine-instruction-like operations.

3. Approach

The translation process is divided into three main parts: instruction lifting, optimization, and code generation. This partition is made to structure the translator and to simplify the implementation of optimizations. This allows supporting other source or target ISAs in the future. Only some modular components need to be extended or replaced for extending the ISA support of this translator.

The IR is tailored to support efficient binary translation. It is the data structure which contains the logic of the input binary in an architecture-neutral form. Generic optimization passes can be applied on this structure. They reduce the number or complexity of operations without altering the program's behavior.

The lifter reads the instruction bytes from the binary file and decodes them using `frvdec` [7]. The resulting instruction objects are used to disassemble the RISC-V binary into basic blocks, operations, and variables in the IR. The code generator creates x86_64 gnu assembler assembly, based on the optimized program in the form of the IR.

A helper library is used to implement platform specific or runtime based functionality which is used by the translated binary. It includes, among other functions, an interpreter which can be used to translate instructions during runtime. The helper library is designed for specific source and target ISAs. System calls and the interpreter work in an architecture-specific way.

To create the translated, executable binary, the GNU assembler (AS) [15] and the GNU linker (LD) [15] are used to assemble the generated assembly and link it to the helper library.

Figure 1 depicts a schematic of the basic structure of the translator. It shows the operating parts of the translator and the processed data fragments, represented by arrows and squares respectively.

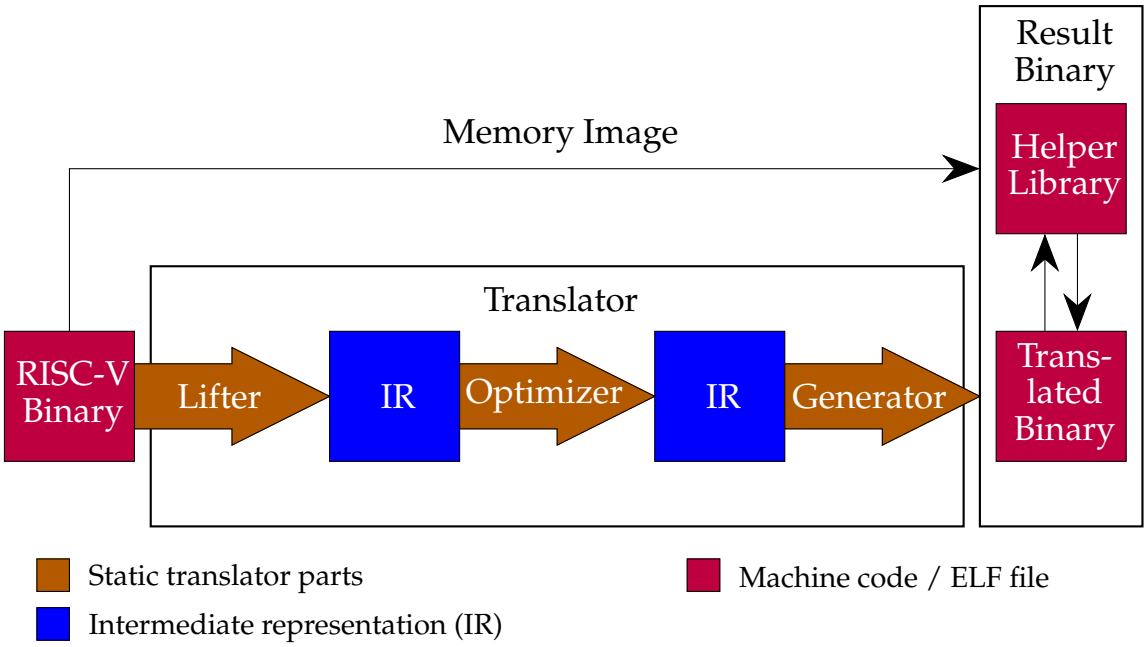

Figure 1: A schematic representation of the translator's internal structure as well as in- and output binaries.

Our implementation only supports 64-bit RISC-V ELF[2] binaries which are statically linked and in little endian format. Another requirement is that the binary conforms to the System V ABI[3]. This defines, among other things, the system call IDs and the calling convention. These prerequisites are checked, using the metadata contained in the ELF-file, before translating the file. This is required debugging and assertion purposes.

[2]Executable and Linking Format
[3]Application Binary Interface

4. Intermediate Representation

As explained in Section 3, an IR is used to create a platform-independent representation of the translated program's logic. This is not only useful to support different input and output languages in the future, but also to be able to optimize the more abstract representation. By parsing control flow and program structures, optimizations like Dead Code Elimination and Constant Folding and Constant Propagation can be applied to the IR code.

Integers	Floats	Special
i64	f64	mt
i32	f32	
i16		
i8		

Figure 2: Types defined for the IR variables. The numbers describe the width of each type in bits. The memory token (*mt*) is a purely virtual type and thus doesn't have a specified width.

The IR consists of variables which are statically typed. The types are shown in Figure 2. The variables follow a single static assignment form, which makes them immutable after they are initialized. Operations take variables as inputs and assign values to new output variables. Every operation performs one of the available instructions. These are abstract, assembly-like instructions defining mostly arithmetic and logical operations. A full list of all the supported instructions can be found in the Figure 19. Variables and their operations are grouped into basic blocks. These are blocks of code which do not contain a control flow changing operation. These special operations are jumps and system calls. Subroutine calls which are defined by many ISAs are also jump instructions with additional memory accessing operations. The control flow changing operations act as connectors between basic blocks. Figure 3 shows a list of all defined control flow changing operations.

Each basic block stores a list of its predecessors and successors. Predecessors are the basic blocks which jump to the current basic block, and successors are the basic blocks that are jumped to by the current basic block. This "jump" can be any control flow changing operation, including syscalls.

A basic block also stores a list of input parameters. These basic block parameters are used for transferring data between basic blocks. Every variable which is linked to a static mapper (4.2) is added to the basic block parameters. These have to be filled with the correct values with every jump to a new basic block.

IR Operation	Description
jump	Direct jump to a known target address.
ijump	Indirect jump to a possibly unknown target address.
cjump	Conditional jump to a known target address.
call	Subroutine call. Also encodes a continuation address to which the subroutine is expected to return to.
icall	Same behavior as a normal *call*, but with unknown subroutine target address.
return	Exit subroutine execution and continue at the continuation address of the *call* instruction which lead to the subroutine execution.
unreachable	This operation should never be reached. On reach, it terminates the program's execution.
syscall	A syscall in respect to the System V ABI.

Figure 3: Control flow changing operations defined by the IR.

4.1. Single Static Assignment Form

First described in 1988 for optimizing an intermediate program representation[17], the *single static assignment form* (SSA) is used for creating and optimizing explicit control flow graphs. It follows one basic rule: "[E]ach variable is assigned to exactly once in the program text." [17] In our IR, each variable stores its source operation (if it is the result of an operation). This enables us to efficiently backtrack variable origins and e.g. replace variables which can be evaluated statically with constant immediates (Section 6.2) or eliminate unused variables (Section 6.1).

We introduce a *memory token*, a required input for operations which read from memory and the result of operations which write to memory. It creates a dependency between operations that store and operations that load data. This should be used to keep the memory access operations in their correct order when reordering operations inside basic blocks and functions. A writing operation acts as a separator of reading operations. Because of the SSA variables used in the IR, correct memory access order is automatically achieved as long as no variable is referenced before its assignment. This insures a consistent memory access ordering and therefore the correctness of the program.

4.2. Static Mappers

To keep track of register contents between basic blocks, we introduced a concept called a *static mapper* or a *static* for short. Variables which link to them can be used as inputs in

basic blocks, They act as a signal to code generators that, if the control flow is transferred to the basic block from an unknown predecessor, the variable can be expected in this static. For control flow operations with an unknown target, a mapping from the output of the basic block to the static at which they should be stored is then created.

The IR itself does not enforce a fixed set of statics but instead leaves it to the lifter to create as many statics as it needs to lift the source ISA. For RISC-V, these are 31 general purpose registers, one memory token, 32 floating point registers and one control and status flag register. The register `x0` is always zero and thus not assigned to a static mapper.

5. RISC-V Lifter

This chapter describes how the RISC-V instructions are decoded and parsed, how they are lifted into the intermediate representation and which optimizations can be made during lifting. This includes the subroutine call optimization, indirect jump target backtracking and jump table detection.

5.1. General Method

The lifter creates the IR based on the RISC-V binary. This process is called lifting because the machine code is lifted up to a more abstract representation at a higher level, the IR.

5.1.1. Program Decoding

The translator starts with the ELF input binary file which is supplied by the user. After checking the validity of the binary file and whether it is suited for being translated by the program, the RISC-V lifter decodes all instructions.

ELF binaries store information about what data is stored where in the binary using *program headers* and *section headers*. Both header types store information about the virtual address spaces, allocation details and access flags. The ELF-file specification does not guarantee that every ELF binary defines *sections* [19]. The lifter therefore supports instruction detection using either *program headers* or *section headers*. However, *section headers* hold finer and more detailed information about program sections than the *program headers* and are therefore preferred for instruction decoding.

The ELF-file loader parses instructions from the sections of the binary which are marked as executable. In case there are no sections in the ELF file[4], we use the program loader information, stored in the *program headers*. They specify which bytes of the underlying binary are actually loaded to the final program.[5] We only parse instructions from the program headers which are marked as *readable* and *executable*.

Due to the memory architecture of modern computers, instructions and data are mostly stored in the same memory. This concept, which was first introduced by John

[4] e.g. memory dumps.
[5] Program header type `PT_LOAD`.

von Neumann in 1945[20], makes it hard to statically distinguish between instructions and data. That is why we rely on the additional information stored in the section- and program headers which identifies certain parts of the binary as executable instructions. This however means we rely on the compiler and the programmer to use executable sections only for storing instructions.

5.1.2. IR Generation

After loading the data form the binary, the lifter runs sequentially through the discovered instructions which are decoded using `frvdec`[7]. The resulting instructions are objects, filled with the parsed content of the binary instruction code. This is a more generic and easier processable form with direct access to involved register numbers and immediate values. For each of the instructions, a variable amount of operations and variables is added to the IR to replicate the original program's behavior.

Figure 4a shows an example RISC-V assembly sequence and Figure 4b shows the resulting IR parts.

As explained in the IR chapter (Section 4), our IR is subdivided into basic blocks which contain consecutive instructions without a change in control flow. As soon as a control flow changing instruction is discovered, it is associated with the basic block. A basic block can be associated to one or more closing control flow changing instructions, e.g. RISC-V branches which consist of a conditional jump and a direct jump to the next instruction (in the next block) if the branch is not taken. The current basic block is finished and a new one is created. If the jump address can be inferred from the instruction using Backtracking (Section 5.3) or if it is directly encoded, like in branch instructions, this address is used to register the start of a new basic block.

If the instructions at the virtual target address have not been parsed by the lifter yet[6], the basic block entry address is stored.

Otherwise, if the lifter already parsed the instruction at the target address without starting a new basic block, the block containing the jump address must be split up at the jump address. This is required because it is only possible to jump to the start of a basic block.

Splitting basic blocks on backwards jumps is required because of the way the lifter consumes the input binary. It runs over all instructions twice. Once for decoding them and a second time for lifting them into the IR. The second run includes detecting indirect jumps and parsing basic blocks at the same time. This causes every backwards jump which would have changed the basic block parsing to be detected too late. It then has to change the already existing basic blocks by splitting them at the jump's target address.

When lifting instructions, the lifter stores their original virtual address. This is used for splitting basic blocks. During splitting, all instructions with an original address lower than the split address are sorted into the first block, the other ones into the second. Both blocks are connected using a direct jump. The predecessor and successor lists as well as some operation inputs need to be adjusted.

[6]This is the case for forward jumps.

```
            start:
                [...]
            loop:
    0xb0:       addi a0, a0, -1
    0xb4:       bne a0, zero, loop
            continuation:
    0xb8:       sll a1, a1, a2
                [...]
```

(a) RISC-V assembly sequence used as an example for the lifter. Decrements `a0` by one in a loop until it is equal to zero, then shifts `a1` to the left by the amount stored in `a2`.

```
[...]

block b1(/*input list */) <= [/* predecessors */] { // loop
    [...]
    i64 v9 <- @10
    [...]
    imm v33 <- immediate -1
    i64 v35 <- add i64 v9, imm v33
    imm v36 <- immediate 0
    imm v37 <- immediate 0xb0
    imm v38 <- immediate 0xb8
} => [(cjump, [b1, /* target inputs */], eq, v35, v36, v37), (jump, [b2,
    /* target inputs */], v38)]

block b2(/* input list */) <= [b1, /* other predecessors */] { //
    continuation
    [...]
    i64 v10 <- @11
    i64 v11 <- @12
    [...]
    i64 v33 <- shl i64 v10, i64 v11
} => [/* control flow op */]

[...]
```

(b) The IR created by the lifter from the RISC-V assembly in Figure 4a.
Some duplicated or unnecessary parts, e.g. predecessors list, variables from statics, target inputs and control flow operations are omitted.

Figure 4: Exemplary lifting of a RISC-V assembly program.

If the jump address cannot be extracted from the instruction, the backtracking (Section 5.3) is used to infer possible addresses which are processed with the same procedure as described before. Using a command line flag, the user can control whether all possible addresses should be found or whether only the first one should be used. If no jump address can be inferred, a dummy basic block which indicates an unresolved jump target, is set as the target block. The jump address then needs to be evaluated at runtime.

If a jump target address is inside a decoded instruction, the further behavior of the program is undefined.

While lifting the instructions, a register file, called *mapping*, keeps track of the variables which represents the content of all registers, the memory token, and the control and status registers (csr). This is required to determine the inputs of the operations based on the input registers of the instructions. For example, when lifting a RISC-V instruction which has the registers `x13` and `x14` as inputs, the input variables are taken from the mapping at the corresponding indices.

At the start of a basic block the mapping is initialized with IR variables from statics (Section 4.2). The register with the index zero is a hard wired zero in RISC-V and therefore not initialized with a variable. Furthermore, each read from the mapping at index zero is handled by adding an immediate variable with value zero. Each write to the zero register is ignored, as it is specified for the `x0` register.[16]

5.1.3. Postprocessing

After all available instructions have been lifted, a post processing step is required to connect the basic blocks with each other. Because we lift the file by reading the instructions consecutively, a basic block does not know its successors during lifting. Vice versa, a new basic block does not know its full list of predecessors during lifting. During post processing, we iterate over all basic blocks, assign them their jump target basic blocks and adjust their predecessor and successor lists.

The last step in lifting is to add an entry basic block that sets the RISC-V stack up and jumps to the basic block at the ELF-file's entry address. This virtual address is stored in the ELF header and specifies where the execution of the binary starts.[19]

5.2. Atomic Instructions

To support programs which use atomic instructions from the "A" standard extension, the lifter converts every atomic instruction into a set of non-atomic instructions which emulate the original instruction's behavior. This does not change the logic or validity of the program as long as the instructions are executed consecutively, without the need for atomic ressource access.

5.3. Backtracking

Although indirect jump addresses can be evaluated at runtime, we can only jump to the beginning of detected basic blocks. This is the case, because we can optimize (Section 6.1)

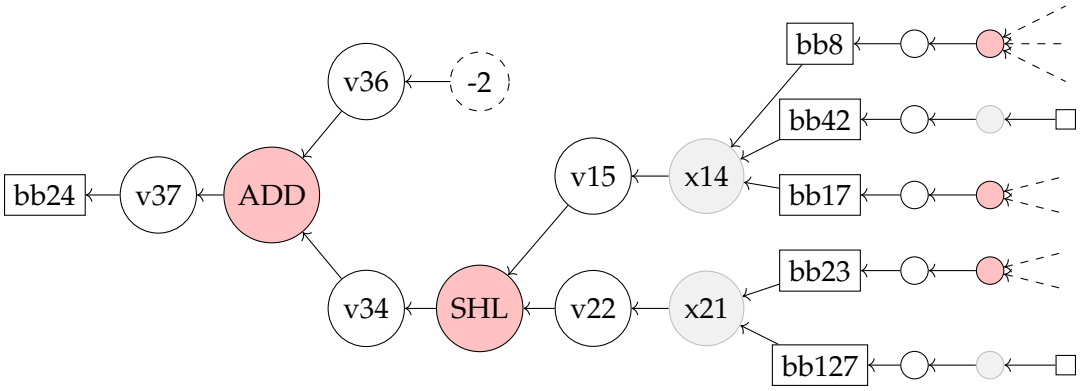

Figure 5: Schematic dependency tree of an indirect jump target address variable (**v37**). The basic block **bb24** ends with an indirect jump, jumping to the address stored in `v37`. `ADD` and `SHL` are operations, names starting with **x** are names of static mappers.

and fold instructions (Section 6.2) inside basic blocks without altering the control flow or logic of the program. That is why there might often not be a one-to-one correspondence of RISC-V to x86_64-Instructions. Creating a fast and runnable program thus means detecting most basic blocks with their corresponding entry points (including their exit points). Although determining all possible indirect jump targets statically is not possible (e.g. virtual function calls), backtracking register contents during lifting can provide us with an approximation of the indirect jump target set.

To statically determine indirect jump target addresses, we need to evaluate possible contents of the jump's input register operand. Each variable stores its origin, being either the result of an operation, an immediate value or from a static. This creates a dependency tree. To backtrack the value of a variable, this tree has to be traversed until a variable which can not be backtracked is reached or until all possible values have been evaluated.

Registers keep their values between jumps and calls. We model this behavior using static mappers which bind variables to their registers in between basic blocks. For backtracking, this means a variable which is bound to a static mapper can have any value that the variable mapped to the same static mapper in its basic block's predecessors has. This opens up a room of variables which influence the jump target, growing with the amount of predecessors and operation inputs.

Figure 5 shows an example for such a dependency tree.

5.4. Jump Table Detection

As described in [4], n-conditional case statements, like the C programming language's switch-case statement, are often implemented by compilers using jump tables. A jump

table is a pattern which enables efficient[7] branching based on a stored set of addresses which are possible jump targets. This means we need to detect jump tables to find all possible basic block entry points.

The jump table patterns observed in generated assembly always consist of a particular set of RISC-V instructions. At the beginning, a bounds check is executed. A branching instruction is used to either jump to the default branch or the code after the switch if the switch condition (an integer) is above the highest case value.

If this is not the case, the switch condition is multiplied by a factor of 4. Using a combination of a `LUI` and an `ADDI` instruction, it is possible to load a full 32-bit constant[16] representing the start address of the jump table. This address is added to the integer condition. The resulting value is the memory address at which the desired jump address is stored. A 32-bit load is executed to load the value into a register. An indirect jump to this register's content is the final instruction, executing the case label jump.

For the jump table detection, the previously described pattern is backtracked for every encountered indirect jump. If the jump turns out to be part of a jump table configuration, the jump target addresses can be extracted and used as entry points for new basic blocks. This also works for switch-statements which are compiled into a combination of jump tables and branch-based search trees.

5.5. Call and Return Optimization

The RISC-V architecture does not include special operations for handling subroutines. A function call is assembled as a jump which places the address of the next instruction into a *link*-register `x1` or `x5` (for the standard calling convention).[16] Returning from a subroutine is done by jumping to the address which is contained in either of those registers. This behavior is further optimized with the compressed instructions `C.JAL` and `C.JALR`. These operations implicitly store the address of the next instruction to the link register `x1`.[16]

Subroutine calls which jump to a dynamic address are so called *icalls*. They act similar to indirect jumps and are backtracked in the same way.

6. Optimizations

The IR produced by the lifter is not always optimal. For example, at the time of lifting, the lifter is not always able to know whether a variable will be used elsewhere, or whether some operation can be directly calculated or combined with another operation that follows later. For these purposes, separate optimization passes are performed once the full IR is available.

[7] Because the integer condition is used as the jump address selector. This is only efficient for switch-statements with small entry ranges (e.g. enum switching).

6.1. Dead Code Elimination

There are cases in compiled programs, where a value is written to a register, but never read again. When lifted, these register writes are found in the IR as unused variables, which can be removed through *Dead Code Elimination* (DCE). For this purpose, each variable has a reference count assigned to it. This is a number, which is increased each time the variable is referenced in another operation or control flow operation. If the reference count of a variable is zero, it is not used anywhere and can be removed.

By visiting the variables in reverse order, *cascading* deletes are taken care of, as the reference count for inputs of an operation is decreased once the operation is deleted.

Another purpose of the Dead Code Elimination pass is to remove unused inputs from basic blocks. For this, the reference-count does not suffice, since there might be circular control flow (like in loops). Instead, DCE marks each variable associated with side effects, i.e. variables used in `stores`, `syscalls` and jumps to unknown targets like `ijumps`. This marking is then propagated through the IR by visiting operation inputs and predecessors, until no more variables are reachable. Then, all unmarked variables and inputs can be deleted. This technique can be compared to a graph search algorithm, which detects connected components containing an operation with side effects.

```
1  block b1(i64 v0, i64 v1) <= [/* predecessors */] {
2      i64 v0 <- @0 (1)
3      i64 v1 <- @1 (3)
4      imm v2 <- immediate -1 (3)
5      i64 v3 <- add i64 v0, imm v2 (1)
6      imm v4 <- immediate 0 (2)
7      i64 v5 <- add i64 v1, imm v4 (0)
8  } => [(cjump, [b2, v2, v1], eq, v3, v4), (jump, [b1, v2, v1])]
9
10 block b2(i64 v0, i64 v1) <= [b1] {
11     /* content omitted for simplicity */
12     i64 v0 <- @0 (1)
13     i64 v1 <- @1 (0)
14 } => [/* control flow operations */];
```

Figure 6: Example IR before DCE: The value in parenthesis after each variable is its reference count.

```
block b1(i64 v0) <= [/* predecessors */] {
    i64 v0 <- @0 (1)
    imm v1 <- immediate -1 (1)
    i64 v2 <- add i64 v0, imm v1 (3)
    imm v3 <- immediate 0 (1)
} => [(cjump, [b2, v2], eq, v2, v3), (jump, [b1, v2])]
block b2(i64 v0) <= [b1] {
    /* content omitted for simplicity */
    i64 v0 <- @0 (1)
} => [/* control flow operations */];
```

Figure 7: Example IR after DCE.

Figure 6 shows an example IR which has not been optimized. Figure 7 shows a potential result of performing DCE on this example. To better visualize the operation of DCE, the variables have been annotated with their corresponding reference count.

The simplest change is variable `v5`, which is not referenced in any other variable or control flow operation, and can be eliminated. One can also see that `v1` is not used, but since it is still passed as a target input to other basic blocks, its reference count is not zero. In this case, the marking process finds that `v1` is not involved in any side effect, and consequently removes it. Since it is also a static, the corresponding input is removed from the basic block.

6.2. Constant Folding and Constant Propagation

Constant Folding and Constant Propagation are closely related optimizations, where operations with known input values (i.e. immediate values) are computed, and immediate variables are propagated throughout basic blocks. In our implementation, Constant Folding, Constant Propagation, and operation simplifications are grouped into the same pass. As such, they are further referred to collectively as Constant Folding.

This optimization pass is done by visiting each operation in a basic block in order, making the pass run in linear time. If the variable is an integer arithmetic, logic or conditional set operation, one of several actions can be performed.

In the simplest case, all inputs are immediates. The result of the operation can be computed, and the variable is replaced with the resulting immediate value. One exception to this are binary-relative immediates — immediates which are offset at runtime by the base address of the binary. Hence, only addition and subtraction with one binary-relative and one non-binary-relative immediate operand are evaluable, and evaluating subtraction is only sensible if the binary-relative immediate is the first operand. Another exception are instructions with side effects like store and load instructions, since these still need to be performed at runtime.

The next case is one immediate input and one non-immediate (static or operation) input. Generally, this case allows for simplification of identity operations, like addition with zero. If the non-immediate operand is an operation with an immediate operand

itself, the two immediates can be evaluated under the rules of associativity and commutativity (Exceptions for binary-relative immediates still apply).

This optimization is greatly simplified by the use of the SSA form in the IR, since variables can not be reassigned. This means that the value of a variable is known just by looking at the declaration of the variable.

```
block b1(i64 v0) <= [/* predecessors */] {
    i64 v0 <- @0

    imm v1 <- immediate 73
    i64 v2 <- add i64 v0, imm v1
    imm v3 <- immediate 64
    i64 v4 <- add i64 v2, imm v3

    imm v5 <- immediate 95
    imm v6 <- immediate 31
    i64 v7 <- sub imm v5, imm v6

    imm v8 <- immediate 0
    i64 v9 <- add i64 v7, imm v8
} => [/* control flow operations */]
```

Figure 8: Example IR before Constant Folding.

```
block b1(i64 v0) <= [/* predecessors */] {
    i64 v0 <- @0

    imm v3 <- immediate 137
    i64 v4 <- add i64 v0, imm v3

    i64 v7 <- immediate 64
} => [/* control flow operations */]
```

Figure 9: Example IR after Constant Folding.

Figures 8 and 9 show an example IR before and after applying Constant Folding. First, variables `v2` and `v4` are collapsed into one variable by evaluating the add with immediates (since addition is associative), making `v1` and `v2` dead code. Next, `v7` is directly evaluated, since both inputs are immediates. Last, since `v9` is an add with zero and thus an identity operation, it is removed, and references to it are replaced with `v7`.

6.3. Common Subexpression Elimination

Common Subexpression Elimination, or more specifically, *(Local) Value numbering* [17], is an optimization that removes redundant code. In principle, each variable is given a

symbolic value depending on its inputs and operation, variables with the same value are redundant. These redundant variables can thus be all replaced by the first occurrence. In our implementation, a hash set is employed (similar to how this optimization is described in the SSA proposal), where relevant properties, i.e. operation type, input variables, rounding mode, or in the case of immediate variables the value itself, are considered and compared.

Common subexpression elimination is mostly useful for eliminating duplicate immediates, which can also be produced by the Constant folding pass. Since input binaries are often already optimized, duplicate operations are rather uncommon.

```
block b1(i64 v0, i64 v1) <= [/* predecessors */] {
    i64 v0 <- @0
    i64 v1 <- @1

    imm v2 <- immediate 64
    i64 v3 <- add i64 v0, imm v2
    imm v4 <- immediate 64
    i64 v5 <- add i64 v0, imm v4

    i64 v6 <- add i64 v1, i64 v5
} => [/* control flow operations */]
```

Figure 10: Example IR before Common Subexpression Elimination.

```
block b1(i64 v0, i64 v1) <= [/* predecessors */] {
    i64 v0 <- @0
    i64 v1 <- @1

    imm v2 <- immediate 64
    i64 v3 <- add i64 v0, imm v2

    i64 v6 <- add i64 v1, i64 v3
} => [/* control flow operations */]
```

Figure 11: Example IR after Common Subexpression Elimination.

7. Code Generator

The code generator is used to produce assembly code for all basic blocks. This means assembling all variables with their associated operations, preparing the inputs for the next basic block, and creating the control flow operations for each block. Additionally, it needs to provide for a runtime environment. This runtime environment consists of the original binary, which makes relevant data for execution (e.g. the original `.data` section) accessible. Furthermore, a stack is part of the runtime environment, which is used as a

```
block b1(i64 v0) <= [/* predecessors */] {
    i64 v0 <- @0
    imm v1 <- immediate -1
    i64 v2 <- add i64 v0, imm v1
    imm v3 <- immediate 0
} => [(cjump, [b2, v2], v2, v3, eq), (jump, [b1, v2])]
```

Figure 12: Example IR used to show the difference between the simple code generator and the register allocator.

replacement for the original stack. This is required, because the x86_64-Stack is reserved for storing temporary variables when assembling a basic block and return addresses for the Call-Return-Optimization.

When generating assembly for the basic blocks, a user can choose between a simple approach that generates unoptimized assembly, and a more sophisticated approach that generates optimized assembly.

7.1. Simple Code Generation

The simple implementation is meant to ease debugging and to better evaluate gains from optimizations in the IR, as it very closely recreates the IR operations. Since the assembly it generates is not optimized, it should only be used for development purposes.

The simple code generator creates assembly for each basic block independently. It creates stack frames which hold space for every variable in the basic block. Then, it iterates all basic block operations and retrieves its sources, either from a static or from the stack frame. The generator applies the operation and saves the results back in the stack frame. At the end of a block, it iterates over all control flow operations and evaluates their condition if necessary. It then writes out the inputs for the next basic block from the stack frame and either calls a helper routine or directly jumps to the next block.

To show an example of the code produced by the naive code generator the IR shown in Figure 12 is used. This simple IR, which decrements a counter until it is 0, produces the assembly shown in Figure 13.

7.2. Advanced Code Generation

To achieve good performance and code density, a more sophisticated approach to generating assembly from the IR needs to be employed. This is done by using register allocation to keep frequently used variables in registers and on the stack without spilling them into the statics when transferring control flow to the next basic block. In addition, certain operation sequences in the IR are merged into single x86_64-Instructions to achieve better code density and performance.

```
b1:
    sub rsp, 32         ; create stack frame, 4 variables * 8 bytes for
        each
    mov rax, [s0]       ; calculate the val of v0
    mov [rsp], rax      ; mov v0 to the stack frame
    mov [rsp + 8], -1   ; immediates are loaded directly into the stack
        frame
    mov rax, [rsp]      ; operands are loaded from the stack again
    mov rbx, [rsp + 8]
    add rax, rbx        ; v2
    mov [rsp + 16], rax
    mov [rsp + 24], 0   ; v3 is a 0 immediate
    mov rax, [rsp + 16]
    mov rbx, [rsp + 24]
    cmp rax, rbx        ; test cjump condition
    jne b1_cf_1         ; jump to next cfop if not fulfilled
    mov rax, [rsp + 16]
    mov [s0], rax       ; Store input for the next basic block
    add rsp, 32         ; Destroy stack frame
    jmp b2
b1_cf_1:
    mov rax, [rsp + 16]
    mov [s0], rax
    add rsp, 32
    jmp b1
```

Figure 13: Assembly produced by the simple code generator from Figure 12.

7.2.1. Register Allocation

Basic blocks which closely represent functions of the original binary are grouped and compiled together. The register and stack state (i.e. the location of variables) is maintained between blocks. To achieve this, all blocks which are targets of a call control flow changing operation must first be marked. The register allocator then iterates over all basic blocks, checks if they are either the target of a call operation or have no known predecessors. It then starts allocating registers for the detected group of these basic blocks.

The allocator compiles the starting block by allocating registers and stack slots for the variables. Every basic block that sits at the beginning of a detected function is called a top-level block. The allocator evaluates the control flow changing operations and checks whether the target of such an operation is part of the current group. This basically means that it is not a target of a call operation. If the target is part of the current group, it creates an input mapping for the target which specifies where each input variable is located. However, if the block has already been compiled, the allocator emits a sequence of instructions which move the variables to the position at which the target block expects them. After this, the handling of the current block is finished and if any of the targets of the block's control flow changing operations are part of the current group they are

recursively compiled too.

The process of generating instructions for the basic block itself is also relatively straight forward. First, a rough estimate of the time-of-uses of each variable is created. This is required because control flow changing operations might need reshuffling. Afterwards, each operation in the basic block is then iterated over and its inputs are loaded into registers if they are located in a static or on the stack frame. A destination register is chosen too. If possible, a destination register other than the input registers is used if the inputs are still required later. Otherwise, one of the input registers is overwritten. When choosing a register, empty or disused registers are preferred. In case all registers contain variables which are still needed, the variable which is the farthest away from being used again is chosen to be evicted to the stack. This is evaluated using the stored time of next use.

A possible improvement here is to create a preference for a register for each variable if it is required at a fixed location. This could be the case if it is used as an input for an already compiled basic block. This preference could then be taken into account when choosing a new register for storing a variable.

As an example of how this works, we assume that only `rax` and `rbx` are available for allocation in the following example. We also assume that an `add` of a variable located in `rax` and an immediate 10 is compiled. It is also assumed that the variable in rax is accessed later in the same block again. If `rbx` contains no variable which is needed anymore the following code is compiled:

```
1 lea rbx, [rax + 10] ; rbx is chosen as the destination since it is empty
```

If `rbx` contains a variable and it is used in the operation following the `add`, `rax` is chosen as the destination, because it is used after rbx:

```
1 mov [rsp], rax      ; rax is saved to the stack frame to be retrievable
      later on
2 add rax, 10         ; then overwrite it
```

If `rbx` contains a variable but it is only used after the variable contained in `rax`, it is instead chosen as the destination:

```
1 mov [rsp], rbx      ; rbx is saved to the stack frame to be retrievable
      later on
2 lea rbx, [rax + 10] ; then overwrite it
```

After all blocks in a group are compiled and the final stack-frame size is known, the control-flow operations for each block can be assembled. The difference to the simple generator is that the inputs for the next block might not be written to statics but instead need to be placed in certain register or stack slots while not overwriting values that

are currently stored in them and might be needed later. This is why the initial time-calculation might be wrong for these cases as it is hard to predict when such conflicts might occur. The current approach taken is to place the input into statics first, then the stack slots, then the registers while storing values which may be overwritten in temporary stack slots. Also, when compiling the control flow operations and the target is a block not part of the current group the stack frame size for it might be different and a such it needs to be adjusted as well to prevent stack over- or underflows.

As an example of the input relocating we again assume only rax and rbx to be available which both hold variables used as inputs for the next blocks. We assume to compile the control flow operations for a block which conditionally jumps to one block with the inputs needed in rbx and the second stack slot respectively. The other other block that is jumped to assumes the variables in rax and rbx.

```
cmp rax, 0       ; evaluate cjump condition
jne b1_cf_1
; cf 0, adjust rax and rbx to rbx and the second stack slot
mov [rsp + 8 * 2], rbx ; variables needed in the stack frame are written
    out first
mov rbx, rax           ; now rbx is safe to be overwritten
jmp b2
b1_cf_1:
; no need for adjustment as the locations for the input match
jmp b3
```

To show an example of the generated assembly, the same IR as in Figure 12 is used. The produced assembly is shown in Figure 14.

```
b1:
    mov rax, [s0]    ; Input is in s0 so load it from there
    add rax, -1      ; the immediate operand can be encoded directly into
        the instruction
    cmp rax, 0       ; evaluate the eq-condition of the cjump control flow
         operation
    jne b1_cf_1
    mov [s0], rax    ; move the modified var to the location the block
        expects it at
    jmp b1
b1_cf_1:
    ; b2 expects the input in rax so no need to move it
    jmp b2
```

Figure 14: Assembly produced by register allocation from Figure 12.

7.2.2. Translation Blocks

One problem that is encountered when employing register allocation on groups of basic blocks is that the programm cannot simply jump to them using indirect jumps anymore. If the basic block is not a top-level block, it might expect variables to be outside of statics and the stack frame, which is set up at the first block of a group. These would be missing if the basic block is reached via a jump.

To still make it possible for indirect jumps to jump to basic blocks which are register-allocated, translation blocks need to be generated. These translation blocks set up the stack frame and move the inputs for the basic block to the registers or stack slots at which they are expected. They afterwards jump to the register-allocated block.

A problem with these blocks is that they need to be generated for a lot of basic blocks and take up a lot of space in the finished binary.[8] They are also rarely used, since most indirect jumps will target the entry basic block of detected functions. The solution is to only generate translation blocks for targets of function returns. This way, the binary size can be drastically reduced while also having a negligible impact on performance.

7.2.3. Merging Operations

Because the x86_64 architecture is a CISC-architecture, it offers more sophisticated instructions than RISC-V or our IR. As such, multiple operations in the IR can be merged into single x86-Instructions if the intermediate results of them are not required later. Mergeable sequences include:

- `add`, (`cast`), `store`: Can be merged to a single store instruction with the `add` embedded in the memory operand and the `cast` expressed as the store width.
- `add`, `load`, `sign-` / `zero-extend`: Can be merged into a single `movsx` / `movzx` instruction with the `add` embedded in the memory operand
- `and` with 0x1f / 0x3f, (`cast`), `shr` / `shl` / `sar`: Can be merged into a single shift instruction since they mask the count operand themselves

7.3. Call and Return Optimization

To support the IR's `call` and `return` instructions, the x86_64-Stack is used. When the code generator encounters a `call`, it first destroys the stack frame of the current basic block, leaving only the return addresses on the stack. It then pushes the return address from the original binary, which is encoded in the IR-`call` as the continuation address. Afterwards, a x86_64 `call` instruction to the target is emitted.

A return is assembled to do a comparison of the actual return address computed and the previously pushed, expected return address. This expected return address is pushed by a previously executed `call` instruction, as explained in the paragraph above. If both return addresses match, an x86_64 `ret` instruction is executed.

[8] This required space was more than 30% of the text section, in some extreme cases.

If they do not match, the stack pointer is reset to the original stack pointer before execution of any basic block and the indirect jump handler is invoked. To prevent underflow, 16 bytes of zeroes are pushed onto the stack before the first basic block is executed. This will cause the compare at a `return` instruction without a previous `call` instruction to fail and the indirect jump handler to be used instead. To prevent overflow, the stack pointer is compared to a maximum allowed size every time a call control flow operation is executed and the stack is reset if it exceeds this size.

7.4. Helper Library

The ABI that is exposed by Linux at runtime is different on x86_64 compared to RISC-V. Some aspects are specified by the System V ABI, like the ELF file format and x86_64 calling convention, but the system call ABI used by Linux differs from platform to platform, mostly due to backwards compatibility or architectural differences [13].

Because of these incompatibilities, some system calls need a thin wrapper, which is defined in the Helper Library. In most cases, the only difference is the ID used to perform the system call, but in some cases, arguments need to be modified before the system call can be performed or to be handled specially. However, system calls related to signals were stubbed to do nothing, since signal support was not necessary for running the benchmarks and implementing these requires invasive changes.

At startup, Linux puts information on the x86_64 stack, for example the program arguments and auxiliary vectors. Before executing the translated code, these values need to be copied to the emulated Stack, and in some cases transformed.

For example, the auxiliary vectors contain information on where to locate the Linux vDSO [14]. This is an optional ELF object that provides user-space wrappers for some system calls to make them faster (e.g. by not invoking the rather expensive kernel system call procedure), and is automatically linked by Linux (if present). The application can use the auxiliary vectors to locate this object and call the provided functions; however, since the ABI is very different, we opted to remove any reference to the vDSO from the auxiliary vectors. In the future, a vDSO wrapper could be implemented to provide the same speed benefits.

Additionally, the helper library houses more required components for execution, like helpers for indirect jump lookup (7.5) and the interpreter (7.6).

7.5. Indirect Jump Resolution

When encountering an indirect jump, the jump's input variable contains the jump target address in the original RISC-V binary address space. The lifter tries to statically identify most of the indirect jump target addresses during lifting and is thus able to start new basic blocks at such addresses. This however means that the translated binary needs a method to translate the original binary's addresses to basic block start addresses in the translated binary, at runtime. This is the case, because address calculations still refer to the original binary's addresses. This method also needs a way to identify indirect jump target addresses which are not the start address of any lifted basic block.

We can only jump to the start of a (translation) basic block, because this is the only reference point where the emulated RISC-V registers are stored at a known location (in the form of statics).

In case we encounter such an unknown jump target address, we switch to the interpreter (Section 7.6) to interpret the original RISC-V code just-in-time until the next jump-able start of a basic block is reached.

Mapping the RISC-V indirect jump target addresses to translated basic blocks requires a dictionary-like data structure. The implementation has to have fast access times as we need to access entries with every indirect jump and every interpreted control flow changing operation in the interpreter. It also has to be able to identify invalid keys, to switch to the interpreter if the jump target address is not a valid basic block start address. We developed two data structure and compared their strengths and weaknesses:

1. For the simple implementation, we store a specific 64-bit wide number for each lifted RISC-V address in the `.ro_data` section. This recreates part of the original binary's address space with a known offset. The numbers are either 0 if no basic block with the given start address exists or the address of the basic block which starts with instructions from this RISC-V address. The latter is done by using the basic block labels, with the assembler calculating the actual address numbers.

   This method has the advantage of having a constant access time with barely no calculation overhead. It however means that we need to store additional information in our output assembly file in the size of the lifted instructions section of the original binary. This could be part of the problem of the assembler AS requiring a lot of memory for assembling the output assembly.

2. To reduce this size overhead, we implemented a method for creating a perfect hash table with a process called "Hash, displace, and compress"[3]. The process consist of a two-stage hash function which first sorts the items into buckets and later generates a hash function for each bucket. The hashed values are stored in one common hash table. The calculation starts with a load factor (α) of 100% and a bucket size of 10 elements (λ) per bucket in average. These values are lowered incrementally by 10% and 1 element respectively if no valid bucket-level hash function could be found for the set constraints. While a lower average bucket size decreases the translation time (because hash functions hash fewer items which generally results in fewer collisions), a higher average bucket size decreases the translated binary's size (because every bucket needs to store its own hash function). The bucket-level hash functions only need a 16-bit index to be reconstructed and can thus be stored memory efficiently. These bucket hash function indices are stored in a separate table in the `.ro_data` section, together with the hash table.

   For the hash function generation, we did not use truly random hash functions. The authors of the "Hash, displace, and compress" perfect hashing method suggest using a "heuristic hash function"[3], one of Bob Jenkins' first published hash functions.[12] For our implementation, we use one of his latest hash functions which is supposed to be faster than his first proposals.[11] Because we are only

interested in hashing 64-bit addresses, we only need part of his hash function which generates hashes for byte arrays of variable sizes.

The hashing algorithm is used in the generator for building the hash table and finding the required hash functions. It is also implemented in the helper library (Section 7.4) for providing hashes for addresses which are only available during runtime.

With the generated hash table and hash functions, every key is assigned an index in the hash table, no matter if it is a valid key which was used to build the hash table or not. That is why we store both the key and the value, which is the target basic block label in this case, in the hash table. For key validation, we first load the key from the calculated index and compare it to the input key. If they do not match, we know that no basic block exists for the supplied RISC-V address. Otherwise, we load the value from the table which is placed just 64-bits after the key.

The hashing helps reduce the required storage space to the hash table size and the hash function indices table size. (Figure 17) The storage size optimizations mean however that every access takes, although its complexity is still constant, the additional time of calculating the hash table index.

7.6. Interpreter

An emulator is the slowest method of translating a binary [2]. Our interpreter is an non optimized emulator.[9] This means that the interpreter is comparatively slow as it decodes the instructions at runtime using frvdec [7]. In order to access the original RISC-V instructions at runtime, the original binary has to be stored as data in the result binary.

We did not optimize the interpreter much because it is only a fall back system for the translated binary and not the focus of this project. Additionally the simple implementation reduces the interpreter code's size and therefore the translated binary size.

The interpreter is called if an indirect jump or indirect call target was not resolved and therefore the generator does not know where to jump. The interpreter is used to fill this gap and to jump back to the translated instructions when possible. This can be done when the start of a top-level basic block or the address of a translation block is reached. If the assembly is generated using the simple code generator, every basic block can be considered to be a top-level basic block.

When the interpreter is entered and left, the RISC-V register values (represented by the static values) need to be in the statics. This is important as the generator stores the variable values in registers or in the stack frame, but the interpreter works on the statics. This needs to be done as the interpreter needs to know a location where the variable values are stored. Therefore, the values are transferred to the statics before calling the interpreter.

[9] Our emulator is called *interpreter* in the implementation and therefore we use this name here too.

This is also a reason why we only leave the interpreter at the beginning of a new basic block. The naive generator (Section 7.1) takes the basic blocks inputs from the statics at this point. When using the register allocator (Section 7.2.1), the translation blocks (Section 7.2.2) move the values to the correct registers and stack slots and are therefore — as well as the top level basic blocks — the addresses where we can leave the interpreter. These addresses are stored in the indirect jump lookup table (Section 7.5).

Usually, a translated basic block will be found after a few interpreted instructions, resulting in almost negligible slow down from entering the interpreter. However, in some extreme cases where no return basic block is found, practically the entire program is emulated resulting in extreme slow downs. This can be observed in rare cases if no translation blocks are created resulting in less possible return targets.

The interpreter fully emulates `rv64imacfd` and therefore supports the same standard extensions as the translator. The translator can be instructed to translated no basic blocks, so that all code is interpreted. This can be used to verify the correctness of the interpreter using the benchmark tests. This method however is, as explained before, rather slow and is therefore not suitable for interpreting entire binaries.

The interpreter theoretically supports the execution of RISC-V JIT compilers. Using the interpreter alone, self-modifying code should also be able to be executed. Such programs were not tested though and therefore this is only a theoretical feature.

At runtime, the CPUID [10] is checked for support of the *FMA3* extension. The interpreter uses it, if it is present, for faster floating point emulation. Otherwise, the fused multiply add instructions are assembled using separate multiplication and addition or subtraction instructions.

8. Floating Point Support

Floating point arithmetic is special as it is affected by rounding and therefore depends on the implementation of the architecture. We decided to use the x86_64 SSE instructions if possible and otherwise emulate them, because a bitwise correct implementation is even slower. Due to rounding, the results of the translated floating point operations may not be the same as if they were run using a RISC-V architecture.

8.1. Lifting and Generating

To support the `F` and `D`[10] standard extensions, the mapping as well as the amount of registered statics has to be extended to include the floating point registers and the floating point control and status register (fcsr). To lift the RISC-V floating point instructions, some instructions are reused, e.g. *add*, *sub*, and some floating point specific instructions are defined, e.g. *fmul*, *fsqrt*. The floating point variables have the types *f32* and *f64* which represents single and double precision numbers, respectively.

The code generator (Section 7) uses the x86_64 floating point instructions provided by the *Streaming SIMD Extensions* (SSE) and *Streaming SIMD Extensions 2* (SSE2) which

[10]The Q standard extension is not supported due to its complexity and lack of usage.

operate on the `xmm` registers. As SSE2 is part of x86_64, it can be assumed that SSE2 instructions are usable. However, the usage of other x86_64 extensions like SSE4 or *Fused Multiply Add 3* (FMA3) in the translated binary has to be enabled using a command line flag for the translator.

For performance reasons, the occurring x86_64 floating point exceptions are not transferred to the register / variable representing the *fcsr*. This register contains the floating points exceptions on RISC-V processors. It is too slow to check after each operation which exceptions were raised and then set the according flags. A read from the *fcsr* value can be redirected to a read from the *MXCSR* register, but this would also take an implementation and performance overhead. However, this does not result in a big problem, because only very few programs depend on floating point exceptions.

The floating point support can be deactivated using a command line flag for the translator in order to prevent the lifter from adding the additional statics. The advantage is a decreased memory usage, because less static variables have to be stored. The lifter then ignores all floating point instructions instead of throwing an error. This is required, because the ELF parser and instruction decoder is not able to distinguish instructions from other program data, as emphasized in Section 5.1.1. Data being wrongly decoded as a floating point instruction does not cause the translation to abort.

8.2. Rounding and Accuracy Problems

A known problem when dealing with different floating point implementations is that especially rounding effects will lead to slightly different results. As we want to use the *SSE2* floating point instructions to provide an acceptable performance, some compromises have to be made.

Firstly, for each RISC-V floating point instruction a rounding mode can be specified, which is used if the number cannot be expressed in the given number format. Because x86_64 does not support such a feature, it is quite hard to implement this in a fast way. With x86_64 floats, the results are already rounded and therefore it is not possible to round the result of each operation afterwards.

Furthermore, a bitwise correct implementation, as provided by the dynamic translator `QEMU-user`, is challenging to implement and comparatively slow. We decided to ignore the rounding mode during lifting instructions. This does not apply to conversions between integers and floating point numbers as well as between single and double precision floating point numbers. We made this exception because it makes a great difference when rounding to full integer numbers. For example, the difference in rounding *25.5* towards positive infinity or towards negative infinity, resulting in *26* respectively *25*, makes a noticeable difference. Furthermore, the used benchmarks (Section 9) require the translator to respect the rounding mode of conversion instructions.

There are two ways of implementing this per-instruction specific rounding.

1. The first one is to use the `roundss` instruction (respectively `roundsd` for double precision) and the selected rounding mode as an immediate input. However, these

instructions are part of *SSE4* which is not supported by all currently used, x86_64-based CPUs [10]. Therefore the usage of *SSE4* instructions must be explicitly enabled for this method.

2. The alternative is to change the rounding mode using the *MXCSR* register. The x86_64 *convert* instructions round according to the state of this register. The disadvantage of this method is that the only way to access the MXCSR register is using `STMXCSR` and `LDMXCSR` instructions which only operate on memory operands. This makes this method slower than the first method. Additionally, this method uses more instructions to change the rounding mode in our implementation. [10]

A second problem is that there can be differences in rounding when assembling some instructions. For example, the fused multiply add instructions can be translated using the x86_64 FMA3 instructions. However, as this extension is not supported by every x86_64 CPU, it needs to be enabled explicitly. The alternative is to emulate the instruction using multiplication and addition or subtraction [11]. This can result in a different result due to rounding errors.

8.3. Implementation Problems and Solutions

Some RISC-V instructions do not have an x86_64 equivalent, especially unsigned conversions, sign injection and the floating point classification instruction `FCLASS`. These instructions are emulated using sequences of x86_64 operations.

The x86_64 ISA does not define an unsigned conversion instruction. Although the *Advanced Vector Extensions 512* (AVX512) provide such instructions, they are relatively new and implemented only in a few CPUs. Therefore, the translator does not provide an optimization flag for using these instructions.

The RISC-V unsigned conversion instruction's behavior is emulated using a combination of the signed x86_64 conversion instructions and logical operations for correcting the results.

For unsigned integer to floating point number conversion, the corrections are applied based on the signed representation of the unsigned integers. 32-bit wide integer values can always be zero extended to 64-bits, which leads to their signed representation to be non-negative. If the signed representation is greater than, or equal, to zero, the 64-bit signed conversion can be applied directly. Otherwise, the input and output values of the signed conversion instruction need to be adjusted. This is done using bit arithmetic, which is also used by *gcc*. The RISC-V floating point conversion to unsigned integers is emulated by converting to a signed integer. If the input floating point value is negative, the result is set to zero, as specified by the RISC-V manual [16].

The sign injection instruction is emulated by arithmetically manipulating the sign bit of the floating point numbers. The floating point classify instructions categorize the input floating point number. Possible categories include sNaN, qNaN, negative infinity,

[11] For some fused multiply adds, negating a floating point value is also required.

negative normal number, and more. The x86_64 architecture does not feature a similar instruction, therefore the classification is done using custom bit arithmetic.

9. Evaluation

To measure the performance we used the SPEC CPU®2017 [18] benchmark package. We only ran the integer and not the floating point suite, because we did not optimize floating point operations. However, the integer suite includes tests that not only cover integer instructions, but also many floating point operations, allowing us to verify our implementation.

The integer benchmark suite covers many actual use cases, including compilers, video compression, artificial intelligence and more. This allows us to stress test the performance under conditions similar to a real production environment.

The benchmark results were compared to the results using QEMU, a popular dynamic binary translator which supports many different architectures. They are also compared to RIA-JIT [1], another dynamic binary translator that only supports RISC-V to x86_64 translation. This translator also claims to be faster than QEMU in its project description paper [6].

The effectiveness and drawbacks of the implemented optimizations were also measured, with special attention to the effects of floating point operation accuracy.

9.1. Benchmark Setup

All benchmarks where performaned on the `time-x` server provided by the Chair of Computer Architecture and Parallel Systems. [12] The Server has a `AMD Ryzen Threadripper PRO 3955WX 16-Cores` CPU and `32 GB` memory. It is running `Ubuntu 20.04.2 LTS` with kernel release `5.10.0-1044-oem` and version `#46-Ubuntu SMP Wed Aug 11 09:50:57 UTC 2021`. The benchmarks where executed on the `NFS` filesystem.

For native benchmarks the GNU Compiler provided by the operating system was used. It had the gcc version `9.3.0 (Ubuntu 9.3.0-17ubuntu1 20.04)` with glibc version `2.31-0ubuntu9.2`. For RISC-V benchmarks, the gnu toolchain provided by the RISC-V foundation was used, version 2021.09.21 [13], it was configured with `-with-arch=rv64imafdc -with-abi=lp64d`.

Our own translator (SBT) was build from the latest master [14] at the time in release mode. The QEMU version used was `1:4.2-3ubuntu6.17`. For RIA-JIT, a modified version by Alexis Engelke was used, adding support for RV64C over the original implementation. It was further modified to fix a bug with compressed instructions and to implement an additional system call required by the benchmarks. [15]

[12] `https://www.in.tum.de/caps/hw/caps-cloud/`

[13] Repository `https://github.com/riscv-collab/riscv-gnu-toolchain`, Commit hash b39e36160aa0649ba0dfb9aa314d375900d610fb

[14] Commit hash 3c341ea7298555b7c34572bd355ca4765d7650df

[15] Commit hash cff2c15e4b646dca8d839f4a5ee9c3ec89231ac6

Due to lack of development tools on the benchmark system, the toolchain, RIA-JIT and SBT were compiled on another host in a container similar to `time-x` and uploaded. Version 1.1.0 of the SPEC CPU®2017 benchmarks was used. The example configuration of the benchmark suite was modified to make it compatible with the newer `gcc` version used, `OpenMP` support was disabled and all test binaries were statically linked. All benchmarks were run with 2 iterations and the slower result was selected. For our static binary translation, only the execution speed of the resulting binary and not the translation time was measured.

9.2. Correctness

To verify that our translator produces correct binaries, we used the testsuites of the SPEC CPU 2017 benchmarks. Especially the tests for the `600.perlbench` benchmark are ideal for this purpose, as they they test the perl interpreter and some popular perl packages. As a result, they cover a wide range of operations, including many different system calls, floating point and string operations. Other benchmark tests cover compilation and compression. All together, the tests cover many real use cases. Any binary passing the benchmarks' testsuites can be assumed to be correct in almost all cases.

Furthermore, the code base was tested for implementation and regression faults using unit tests and integration testing. The unit tests were written using the test framework *Google Test*. Additional integration tests were applied, for example a test where the translator translates a simple zip utility tool once with full optimizations and once with only the interpreter enabled. The resulting translated zip tools are used to compress and decompress sample files. The output of each operation is then compared to the output of a natively compiled version of the same zip utility.

Using this methodology, we tested the interpreter for correctness. The translator also passes all tests when compiled without floating point support. However, with floating point support, additional optimizations [16] need to be enabled to achieve correct floating point results.

9.3. Comparison to QEMU and RIA-JIT

We used the SPECSpeed®2017 Integer benchmark suite to compare the performance of our SBT, QEMU and RIA-JIT. Because the individual benchmarks total execution time varies widely, all results are normalized to a native benchmark. This clearly shows the overhead caused by the binary translation and can be used to compare SBT, QEMU and RIA-JIT. Our own translator (SBT) was run with all optimizations, except the removal of *Translation Blocks*, as it proofed to be highly detrimental for execution speed. This will be further discussed in Section 9.5. QEMU and RIA-JIT where run without any additional configuration.

In Figure 15 the results are visualized. It can be seen that we consistently outperformed QEMU in every benchmark. On average SBT was 18% faster than QEMU, in some cases,

[16] `fma3` and `sse4` need to be enabled.

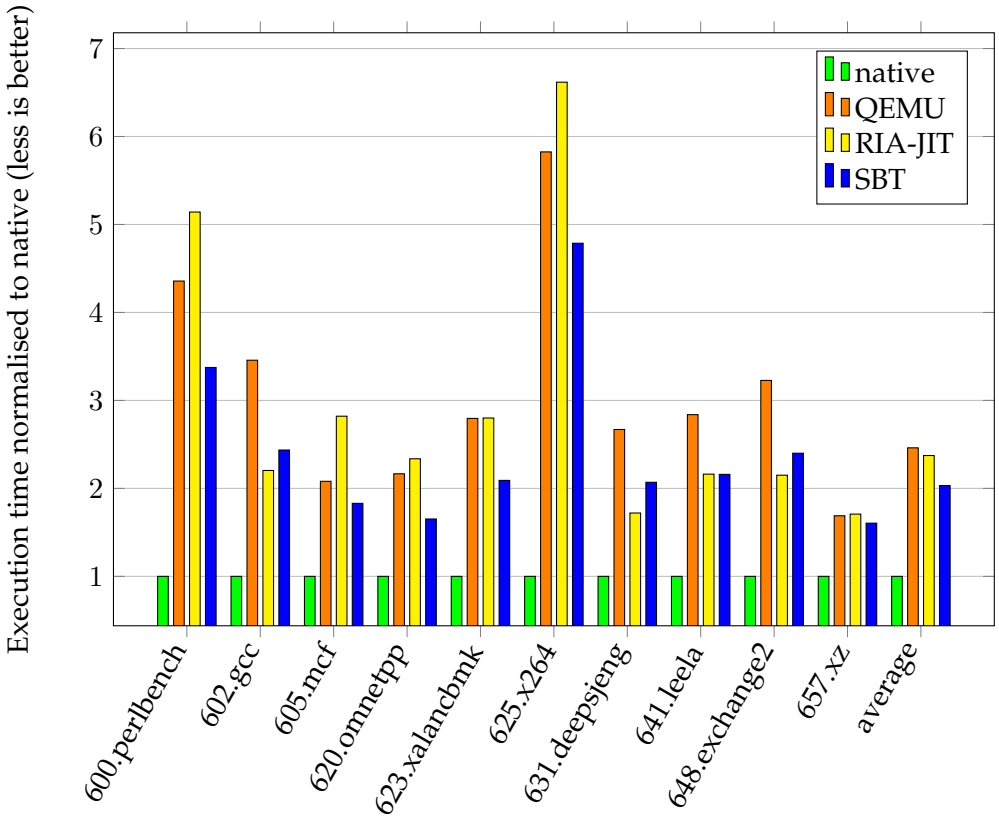

Figure 15: SPECSpeed®2017 Integer benchmark suite results compared. SBT was executed with all optimizations except the removal of translation blocks and the hash table. RIA-JIT encountered a floating point error in the benchmark 625.x264, so this single benchmark was repeated with a toolchain using `-with-arch=rv64iamc -with-abi=lp64`.

namely *658.xz*, the difference was very small, but in others it was as much as 41% when comparing the *602.gcc* benchmark.

SBT is on average faster than RIA-JIT, but when looking at individual benchmarks the results are mixed. In some benchmarks SBT is faster by a large margin, and in some other cases it performs similarly, being slower in some cases. Because of a bug in the floating point operations a toolchain without hard floating point support had to be used for the *625.x264* benchmark for RIA-JIT, in practice this could mean worse performance.

Comparing our results for QEMU and RIA-JIT to those presented in the RIA-JIT paper [6], QEMU has become significantly faster, from 3.2x to 2.5x native speed, while RIA-JIT has become slightly slower.

9.4. Effectiveness of different Optimizations

To better understand the performance of our translator we compared the effect of the implemented optimizations on the runtime performance. In Figure 16 the performance of optimizations applying to the `IR`, applying to the generator and the `Call and Return` optimizations are compared to a baseline of no additional optimizations.

On average any optimization improves the performance, however in some cases individual optimizations can result in performance loss.

On average the biggest performance increase is achieved by optimizations targeting the IR, most importantly `Constant Folding`. It also shows that static binary translation can perform more extensive anaylsis of the target program than dynamic binary translation, resulting in more speed up at runtime.

In benchmarks with many instructions following the load, modify, store pattern, the `Merging Operations` optimizations yields the biggest results, in some edge cases it can cause a minor slow down.

In most cases `Call and Return Optimizations` improves performance, but in some cases the additional cost incured by miss predictions causes a slow down, here in the `648.exchange2` and `657.xz` benchmarks.

9.5. Performance of Translation Blocks

The translation blocks introduced in Section 7.2.2 are responsible for a large part of the resulting binary. To save space in the translated executable we added an option to remove them.

The removal of these blocks also means that any indirect jump that can't be resolved needs to enter the interpreter, which is inherintly slow. The performance impact is made worse due to the lack of points where the interpreter can jump back to compiled code. For example the `600.perlbench` benchmark was heavily affected by this, because not all indirect jump targets where detected at translation, resulting in misses at runtime.

Because of the large impact on performance we ran our benchmarks with translation blocks available. In the future this could be mitigated by detecting more indirect jump targets and keeping translation blocks in these cases.

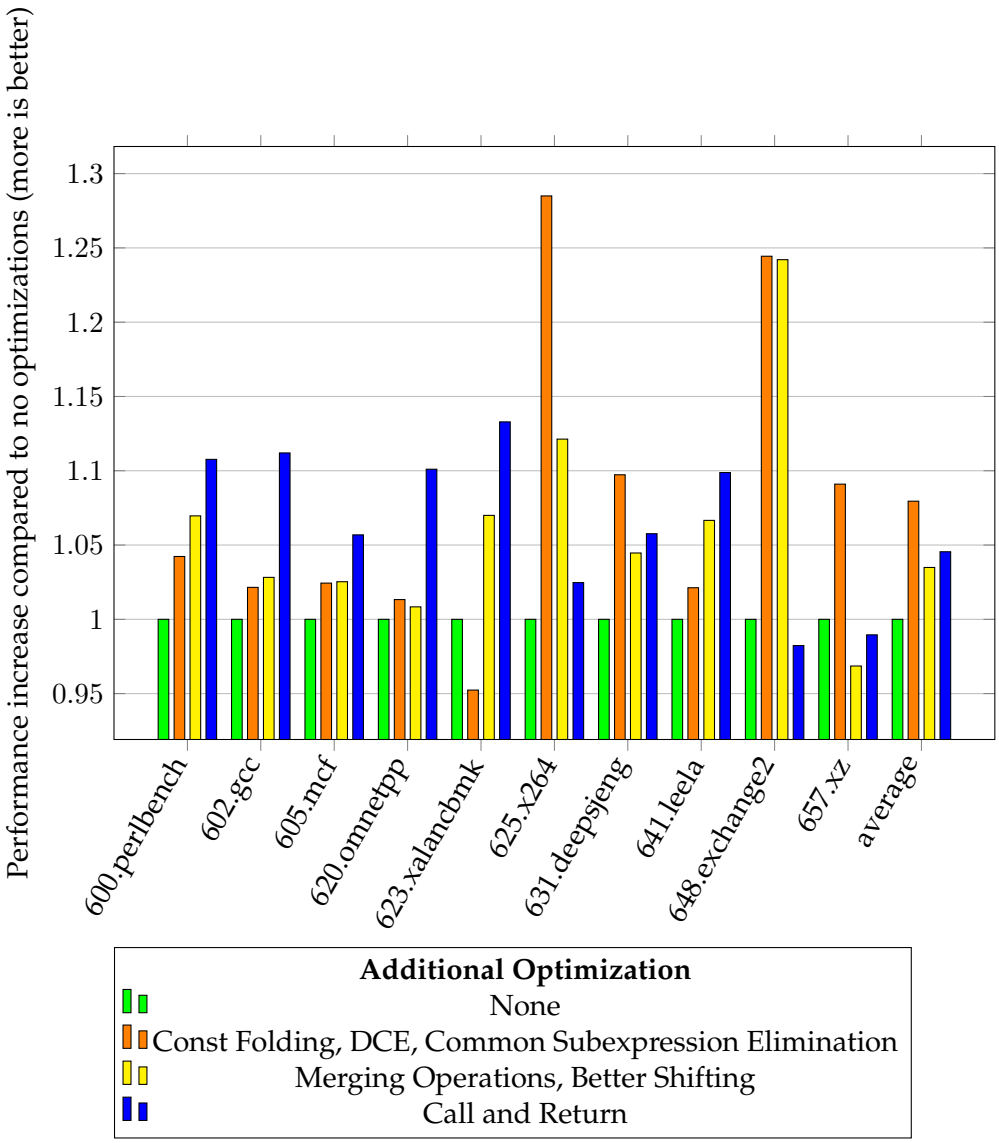

Figure 16: SPECSpeed®2017 Integer benchmark suite results of different optimizations compared using the register allocator with FMA3 and SSE4 enabled. For IR the Constant Folding, Dead Code Elimination and Variable Deduplication where enabled, for Generator the Merging Operations and BMI2 instruction set extension.

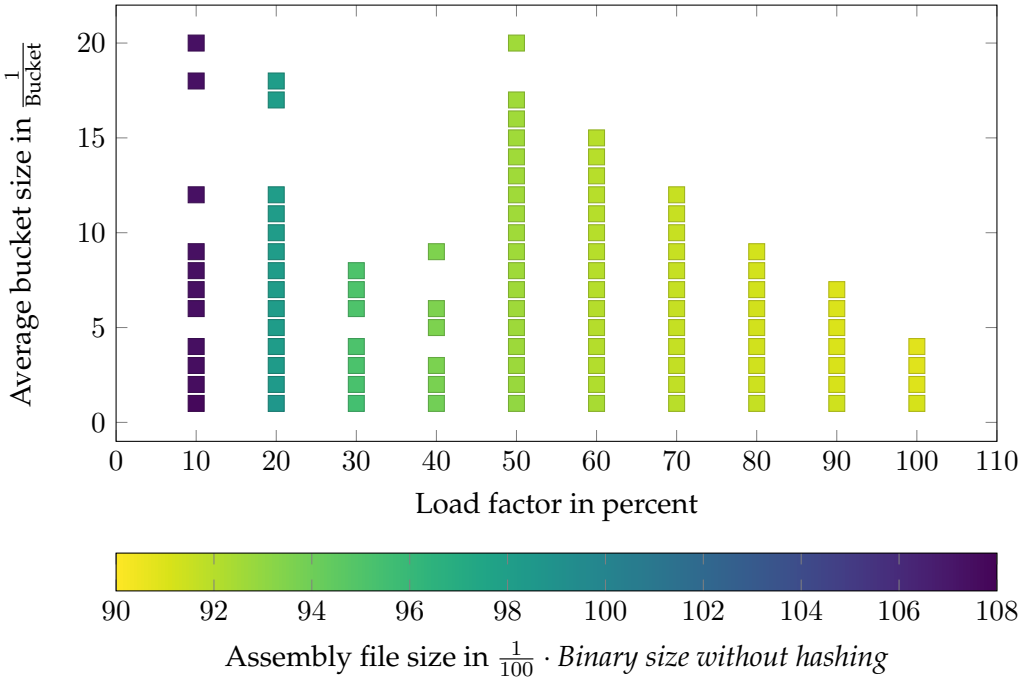

Figure 17: Effects of average bucket size and hash table load factor on final assembly file size.

9.6. Basic Block Start Address Hashing

In Figure 17 we show the effects of the two major parameters for the used hashing algorithm. Missing data points mean that the hash function algorithm was not able to find all valid bucket-level hash functions within the 16-bit per-bucket storage limit. The load factor is the fraction (in percent) of the final hash table which is filled with hashed entries. The average bucket size is used to calculate the number of buckets and therefore directly effects the number of bucket-level hash functions required.

The test was conducted on the translated binary of a simple zip program. Sources are located in the `tests` project folder. The test was only performed once because the hashing algorithm and the translator itself are both strictly deterministic.

The analysis shows that the amount of storage required for bucket-level hash functions does not have a noticeable impact on the final assembly size. It also shows that finding all valid bucket-level hash functions is more likely with a lower average number of elements per bucket. This leads to the conclusion that using smaller buckets sizes can improve bucket-level hash function generation times while not noticeably increasing the output assembly size.

However a hash table is noticeably slower than a direct lookup table. This caused a significant slow down in some of the benchmarks. Even with the `Call and Return Optimization` reducing the amount of lookups required, the performance impact was too big. Indirect jump hashing was therefore disabled for performance testing, but kept

for use cases that require a smaller translated binary result. It is an optimization which the user can manually enable using a command line flag.

9.7. Floating point accuracy

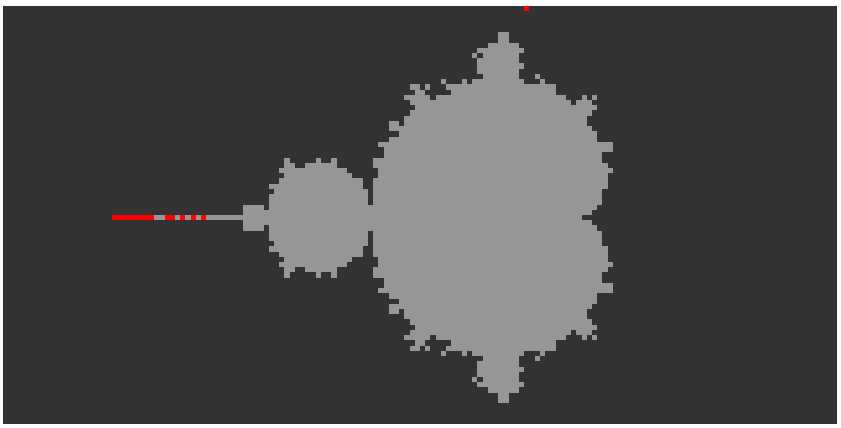

Figure 18: Differences in results of the mandelbrot set visualizer [5] when comparing the translator to QEMU. light grey = mandelbrot set, dark grey = remainder, red = differences

To depict the occuring rounding differences, a simple mandelbrot set visualizer [5] was used. Figure 18 shows that the translator without any optimizations produces different results than the RISC-V binary emulated with QEMU. The comparison of the emulated RISC-V program result and the picture calculated by the program nativley compiled on x86_64 shows the same differences.

Another problem can be observed when activating the usage of *FMA3*. It allows the translation of RISC-V fused multiply add instruction to x86_64 equivalents rather than emulating it. This again has an impact on the accuracy, because the intermediate result in fused multiply add instructions is not rounded [10]. The mandelbrot test showed that the usage of *FMA3* instructions causes the result to be more similar to the RISC-V result and therefore should be used if available.

10. Summary

The static binary translation program (SBT) developed in this project statically translates binaries compiled for the RISC-V ISA to x86_64 binaries. It is able to translate every benchmark from the SPECSpeed®2017 integer benchmark suite. It fully supports the *RV64I*, *RV32I* RISC-V base integer modules. It also supports the *Zifencei*, *Zicsr*, *M*, *F*, *D*, *A* and *C* standard extension for single threaded execution.

The SBT is written in a modular manner to support repurposing parts of the project for e.g. translating other source or target ISAs. It consists of a RISC-V lifter, an abstract

intermediate representation and an x86_64 code generator. The translator additionally supplies a runtime environment for translated binaries. This is used for translating OS-specific features and performing a fallback just-in-time interpretation of the original RISC-V code.

As shown in the evaluation section, the SBT reached its goal to be consistently faster than QEMU. It is able to outperform the dynamic binary translator in every executed benchmark. The SBT is, on average, approximately 18% faster than QEMU. The SBT was also compared to a dynamic binary translator specifically optimized for translating RISC-V binaries. The static translator was able to outperform the dynamic translator by 16% on average over all executed benchmarks.

In order to increase the execution speed even further, more optimizations to the translated binary are possible. Some of them have already been introduced in previous chapters. Such features are not yet implemented due to the limited time span of this project.

Further improvements of the translator include instruction reordering beyond basic block borders, support for dynamically linked binaries and support for more RISC-V ISA extensions. Another milestone project would be to utilize the modularity of the project and reuse the IR and either the RISC-V lifter or the x86_64 code generator to introduce support for different source or target ISAs.

References

[1] Lightweight and performant dynamic binary translation for risc–v code on x86–64. https://github.com/ria-jit/ria-jit, visited 2021-10-14.

[2] E.R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *Computer*, 33(3):40–45, 2000. https://ieeexplore.ieee.org/document/825694, visited 2021-09-28.

[3] Djamal Belazzougui, Fabiano Botelho, and Martin Dietzfelbinger. Hash, Displace, and Compress. pages 682–693, 09 2009. http://cmph.sourceforge.net/papers/esa09.pdf, visited 2021-10-09.

[4] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2):171–188, 2001. https://www.sciencedirect.com/science/article/pii/S0167642301000144, visited 2021-09-28.

[5] Chris Domas. https://github.com/xoreaxeaxeax/movfuscator/blob/master/validation/mandelbrot.c, visited 2021-10-05.

[6] Noah Dormann, Simon Kammermeier, Johannes Pfannschmidt, and Florian Schmidt. Großpraktikum rechnerarchitektur - dynamic binary translation for risc–v code on x86–64summer term 2020. 2020. https://raw.githubusercontent.com/ria-jit/ria-jit/master/documentation/paper/paper.pdf, visited 2021-10-14.

[7] Alexis Engelke. frvdec, 2021. https://git.sr.ht/~aengelke/frvdec, visited 2021-09-28.

[8] A.D. George. An overview of risc vs. cisc. In *[1990] Proceedings. The Twenty-Second Southeastern Symposium on System Theory*, pages 436–438, 1990.

[9] Jeremy Hsu. Risc-v star rises among chip developers worldwide. *IEEE Spectrum*, April 2021. https://spectrum.ieee.org/riscv-rises-among-chip-developers-worldwide, visited 2021-09-20.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, April 2016. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, visited 2017-08-19.

[11] Bob Jenkins. Spookyhash: a 128-bit noncryptographic hash. https://www.burtleburtle.net/bob/hash/spooky.html, visited 2021-10-08.

[12] Bob Jenkins. *Algorithm alley: Hash functions*. Dr. Dobb's Journal of Software Tools, 1997.

[13] Linux man-pages project. *SYSCALLS(2) Linux Programmer's Manual*, 5.13 edition, August 2021. `https://man7.org/linux/man-pages/man2/syscalls.2.html`, visited 2021-10-13.

[14] Linux man-pages project. *VDSO(7) Linux Programmer's Manual*, 5.13 edition, August 2021. `https://man7.org/linux/man-pages/man7/vdso.7.html`, visited 2021-10-10.

[15] GNU Projekt. Gnu binutils. `https://www.gnu.org/software/binutils/`, visited 2021-09-28.

[16] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, December 2019. `https://github.com/riscv/riscv-isa-manual/releases/`, visited 2021-09-19.

[17] Barry Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, 1988. `https://www.cse.wustl.edu/~cytron/531Pages/f07/Resources/Papers/valnum.pdf`, visited 2021-09-28.

[18] The Standard Performance Evaluation Corporation (SPEC). Spec cpu®2017. `https://www.spec.org/cpu2017/`, visited 2021-10-14.

[19] TIS Committee. *Executable and Linking Format*. TIS Committee, 1995. `https://refspecs.linuxfoundation.org/elf/elf.pdf`, visited 2021-09-28.

[20] J. von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

A. Appendix

Instruction	Description
store	Store value at supplied address. Consumes memory token and generates a new one.
load	Loads value at supplied address and returns it. Reads as many bits from memory as the output variable is wide.
add	Addition.
sub	Subtraction.
mul_l	Multiplication. Returns the lower half of the result.
ssmul_h	**Signed** multiplication. Returns upper half of the result.

uumul_h	**Unsigned** multiplication. Returns upper half of the result.
sumul_h	Multiplication of one **signed** and one **unsigned** value. Returns upper half of the result.
div	**Signed** division. Returns both the result and the remainder.
udiv	**Unsigned** division. Returns both the result and the remainder.
shl	Logical shift left.
shr	Logical shift right.
sar	Arithmetic shift right.
or	Logical *or*.
and	Logical *and*.
not	Logical *not*.
xor	Logical *xor*.
cast	C-style typecast. Defined for shrinking variables and for bit-wise cast from integer to floating point numbers.
slt	Ternary operator, set if less than: IR: `dst ← slt v1, v2, v3, v4` Pseudocode: `dst = (v1 < v2) ? v3 : v4`
sltu	Ternary operator, set if less than: IR: `dst ← sltu v1, v2, v3, v4` Pseudocode: `dst = (v1 <u v2) ? v3 : v4`
sle	Ternary operator, set if less or equal: IR: `dst ← sle v1, v2, v3, v4` Pseudocode: `dst = (v1 <= v2) ? v3 : v4`
seq	Ternary operator, set if equal: IR: `dst ← seq v1, v2, v3, v4` Pseudocode: `dst = (v1 == v2) ? v3 : v4`
sign_extend	Enlarge value to bigger type using sign extension.
zero_extend	Enlarge value to bigger type using zero extension.
setup_stack	Meta instruction used for stack setup at program start.
umax	Returns larger one of two values, **unsigned** comparison.
umin	Returns smaller one of two values, **unsigned** comparison.
max	Returns larger one of two values, **signed** comparison.
min	Returns smaller one of two values, **signed** comparison.
fmul	**Signed** floating pointer number multiplication.

fdiv	**Signed** floating pointer number division, only returns the result.
fsqrt	Takes the square root of two floating point values.
fmadd	Fused multiply add of three floating point values: IR: `dst ← fmadd v1, v2, v3` Pseudocode: `dst = v1 * v2 + v3`
fmsub	Fused multiply sub of three floating point values: IR: `dst ← fmsub v1, v2, v3` Pseudocode: `dst = v1 * v2 - v3`
fnmadd	Fused negate multiply add of three floating point values: IR: `dst ← fnmadd v1, v2, v3` Pseudocode: `dst = -(v1 * v2) + v3`
fnmsub	Fused negate multiply sub of three floating point values: IR: `dst ← fnmsub v1, v2, v3` Pseudocode: `dst = -(v1 * v2) - v3`
convert	Conversion between integer and floating point numbers or between single and double precision floating point numbers.
uconvert	Unsigned conversion between integer and floating point numbers or between single and double precision floating point numbers.

Figure 19: Defined IR instructions.