Universiteit Leiden
The Netherlands

Bachelor Computer Science

Optimizing RISC-V Binaries using LLVM-based Same-Architecture Binary Translation

Tim de Jong

Supervisors:
Dr. K.F.D. Rietveld & Prof.dr. R.V. van Nieuwpoort

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 17, 2024

Abstract

In recent decades, binary translation has been widely used for migrating and emulating legacy ISAs. A recent study investigated the efficacy of static binary translation using MCTOLL, an open-source LLVM-based tool for raising x86 and ARM binaries to LLVM IR. They found that, in some cases, same-architecture binary recompilation had positive effects on the size of the binary. This would be especially interesting for RISC-V, which is currently mainly used in resource-constrained embedded systems and IoT devices. However, MCTOLL only supports raising x86 and ARM binaries. In this thesis, we expanded MCTOLL by implementing the RISC-V binary raiser to be able to analyze the effectiveness of same-architecture binary recompilation for 64-bit RISC-V ELF binaries. Our findings indicate that same-architecture binary recompilation can, in some cases, lead to a reduction in the size of ELF sections and an improvement in runtime performance.

Contents

1 Introduction

Binary translation plays a crucial role for the migration and emulation of legacy instruction set architectures (ISAs). It is essential for emulating systems that are not easily accessible, such as emerging architectures or specialized hardware platforms. Additionally, binary translation enables the migration of software to emerging architectures, which often suffer from a lack of available binaries. Despite the advancements in binary translation tools, there remains a significant gap in support for RISC-V. This gap likely arises because RISC-V, being a modern and open-source architecture, is typically the target ISA for binary translation rather than the source. Consequently, there is little demand for translating RISC-V binaries to other architectures such as ARM.

A recent study investigated the efficacy of static binary translation using MCTOLL, a prominent LLVM-based static binary translation tool [Fin22]. They found that raising an x86 binary to the LLVM Intermediate Representation (IR) and recompiling the optimized IR to an x86 binary (i.e., same-architecture binary translation) had positive effects on the runtime performance and in some cases on the size of the x86 binary. This would be particularly interesting in the context of RISC-V, which is mainly used in resource-constrained systems. However, the lack of binary translation support for RISC-V within existing frameworks hampers the ability to leverage same-architecture binary translation techniques that have shown promising results for x86 and ARM.

This thesis aims to address this gap by expanding the capabilities of MCTOLL by adding support for raising 64-bit RISC-V ELF binaries to LLVM IR. By doing so, we aim to analyze the effectiveness of same-architecture binary translation for RISC-V, potentially leading to optimizations that could benefit embedded systems and IoT devices. By introducing a RISC-V binary raiser in MCTOLL, this research contributes a valuable tool for the community, possibly stimulating further studies on RISC-V binary translation.

This thesis is organized as follows. In Chapter 2 we will describe the necessary background information required for this thesis. Chapter 3 describes the implementation of our RISC-V binary raiser within MCTOLL, which is evaluated in Chapter 4 alongside the effectiveness of same-architecture binary translation for 64-bit RISC-V ELF binaries. Finally, in Chapter 5 we discuss the results of the evaluation and Chapter 6 identifies the current limitations of our RISC-V binary raiser.

2 Background

In this chapter, we will describe the necessary background required for the remaining chapters of this thesis. Furthermore, we will describe the LLVM tool MCTOLL, how it functions, and its current state and limitations.

2.1 RISC-V

RISC-V is an open standard Instruction Set Architecture (ISA). An ISA defines a set of available instructions which the processor can execute, as well as the available registers and main memory on which they operate. Examples of ISAs are x86, ARM, 6502, SPARC, MIPS, PowerPC, OpenRISC, and many more. Some ISAs are proprietary, such as x86, ARM, and MIPS, while others such as PowerPC and OpenRISC are open-source and royalty-free. RISC-V is also open-source and was initially designed for research and education of computer architecture [WLP+14]. They considered adopting OpenRISC, but decided against it because of technical limitations, such as limited space for expansion and branch delay slots complicating implementations for higher performance. Refer to Listing 1 for an example of a RISC-V machine function.

Some of the mentioned architectures are RISC (Reduced Instruction Set Computer), while others are CISC (Complex Instruction Set Computer). RISC ISAs typically have a small number of simple instructions and a large register set on which the instructions operate. Load and store instructions are needed to read from and write to memory. Examples of RISC ISAs are ARM, PowerPC, and RISC-V. CISC ISAs typically have many complex instructions and they typically operate on memory directly. Examples of CISC ISAs are x86 and 6502.

The fact that RISC-V is freely available for use, combined with the simplicity and extensibility of the RISC-V ISA make it attractive for manufacturers to use RISC-V to design their processors. At the moment, RISC-V is mainly used for embedded and IoT (Internet of Things) devices.

In recent years, the RISC-V ecosystem has matured significantly, which led to RISC-V also slowly being used for High Performance Computing (HPC) [Bro24]. However, RISC-V HPC is not quite there yet, as performance of x86 and ARM architectures is still an order of magnitude ahead [FRT+23]. Last year, a world's first RISC-V laptop was released by DeepComputing [Dee23b, Dee23a]. This shows that many companies are contributing to the RISC-V ecosystem. However, RISC-V is still slightly behind its competitors.

2.2 Binary Translation

In the past decades, binary translation has been extensively used for the migration and emulation of legacy ISAs. In general, the goal of binary translation is to translate binary code of a legacy or older ISA to equivalent binary code of a newer ISA [CM96]. In dynamic binary translation, the translation of the binary code occurs at runtime, providing the translator context about the execution of the program. This approach overcomes issues such as dynamic linking and indirect jumps at the cost of a significant performance overhead, similar to interpreted languages [Pro02]. In contrast, a static binary translator translates the binary code ahead-of-time by reconstructing the Control Flow Graph (CFG) of the program [Fin22]. This has the advantage of being able to perform more optimizations and only needing to translate the binary once. Additionally, statically translated binaries typically use less memory, less cycles and less power when compared to a dynamic variant [SCHY12]. However, issues such as indirect jumps become more difficult to address, because the target address is only known during runtime.

There is a slight semantic difference between the terms binary raising and binary translation. The term *binary raising* typically refers to the process of transforming machine code into a higher-level abstraction, whereas *binary translation* typically maintains the same level of abstraction between input and output. The term *decompilation* typically refers to the process of recovering the original source code from a binary, whereas binary raising focuses on transforming machine code into a higher-level abstraction for analysis, instrumentation, and recompilation, rather than aiming to recover the exact original source code.

2.3 Basic Blocks and Control Flow Graphs

A machine function can be divided into basic blocks. A basic block is *"a straight-line code sequence with no branches in except to the entry and no branches out except at the exit"* [HP11], i.e., a sequence of instructions which are executed sequentially. Identifying the basic blocks of a machine function involves determining the entry points, also known as *leaders*. An instruction is a leader if, and only if:

1. It is the first instruction.
2. It is the target of a jump/branch instruction.
3. It is the instruction immediately after a jump/branch instruction.

A jump/branch instruction at the exit point of a basic block is also called a *terminator* instruction.

Listing 1 illustrates a simple RISC-V function containing a loop. In this case, every label represents an entry point to a basic block. The instruction after the branch instruction (`bge`) is also a leader. This instruction is not a target of a jump/branch instruction, so the compiler did not mark it with a label. The label `.T1` has been added to represent this basic block. The program can be divided into six basic blocks: *Entry* (`loop`), *Loop Check* (`.L1`), *Pre-Loop Body* (`.T1`), *Loop Body* (`.L2`), *Loop Increment* (`.L3`), and *Exit* (`.L4`). Using these basic blocks, a Control Flow Graph (CFG) can be constructed, where each basic block is represented as a node. See Figure 1.

2.4 Compilers

In general, a compiler translates high-level source code into low-level machine code that can be executed by the processor. This compilation process usually consists of five phases: Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, and Machine

```
loop:
    addi    sp, sp, -32
    sd      ra, 24(sp)
    sd      s0, 16(sp)
    addi    s0, sp, 32
    sw      a0, -20(s0)
    li      a0, 1
    sw      a0, -24(s0)
    li      a0, 0
    sw      a0, -28(s0)
    j       .L1
.L1:
    lw      a0, -28(s0)
    lw      a1, -20(s0)
    bge     a0, a1, .L4
.T1:
    j       .L2
.L2:
    lw      a1, -20(s0)
    lw      a0, -24(s0)
    mulw    a0, a0, a1
    sw      a0, -24(s0)
    lw      a0, -20(s0)
    addiw   a0, a0, -1
    sw      a0, -20(s0)
    j       .L3
.L3:
    lw      a0, -28(s0)
    addiw   a0, a0, 1
    sw      a0, -28(s0)
    j       .L1
.L4:
    lw      a0, -24(s0)
    ld      ra, 24(sp)
    ld      s0, 16(sp)
    addi    sp, sp, 32
    ret
```

Listing 1: A RISC-V machine function containing a loop.

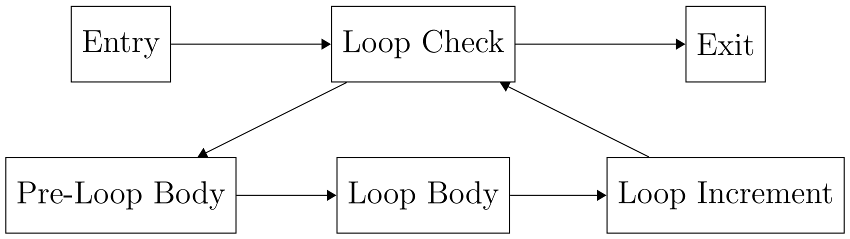

Figure 1: A Control Flow Graph representing the basic blocks of the RISC-V machine function of Listing 1.

Code generation. Compilers are usually divided into a *frontend* and a *backend* using an intermediate representation. The frontend translates the high-level source to the Intermediate Representation (IR), while the backend translates the intermediate representation to machine code. This division enables a modular approach, allowing multiple frontends to use the same backend, which can be seen in Figure 2. Another advantage is that language-indepedent optimizations can be applied to the intermediate representation.

Binary translation is essentially the inverse of the standard compilation process, the low-level machine code is translated to a higher-level intermediate representation or to a high-level target and from there to low-level machine code.

2.5 LLVM

LLVM, originally a research project at the University of Illinois [Pro24], is an open-source compiler infrastructure that provides libraries and tools for building compilers and other

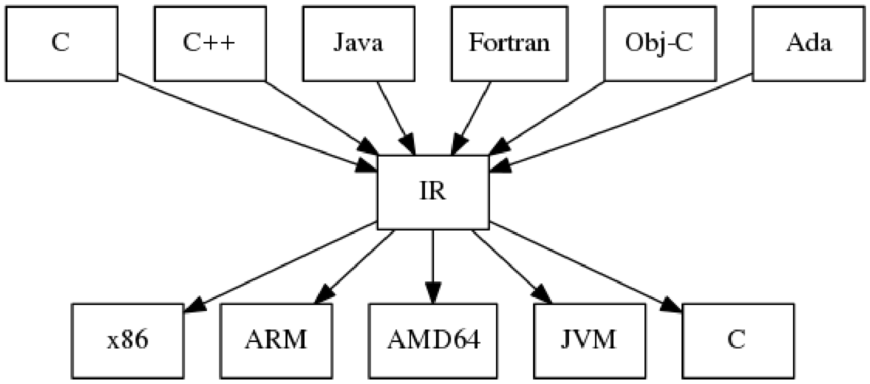

Figure 2: A compiler infrastructure with multiple frontends and backends compiling to and from a common IR, adopted from https://liucs.net/cs664s16/ir.html

programming language-related software [Lat02]. Binary translators have been used in combination with LLVM to utilize its optimizations and compiler backends. At the core of the LLVM project lies its intermediate representation (IR): a low-level, Static Single Assignment (SSA)-based instruction set [LA04] that serves as a common language for compilation and optimization. Variables of an SSA-based language are required to be assigned exactly once and must dominate all its uses [Fin22]. The LLVM IR is targeted by multiple compiler front ends, most notably the Clang compiler for the C language family, the Rust compiler, and the Swift compiler. Other significant sub-projects within the LLVM project include LLD, a faster alternative to system linkers; BOLT, a post-linking optimizer; and, more recently, the MLIR initiative.

LLVM IR mainly comes in two forms: a human-readable text-based representation of the IR, and a machine-friendly binary representation known as bitcode (not to be confused with bytecode). LLVM IR supports a wide range of types, for example integral types (i1, i8, i64), floating point types (float, double), and also arrays, pointers, and structures. LLVM IR instructions include binary operations (add, mul, xor), comparison instructions (icmp sge, icmp ult), control flow instructions (br, ret), and memory operations (load, store, alloca, getelementptr). The alloca instruction allocates memory on the stack, and the getelementptr (usually abbreviated to GEP) instruction can be used to compute an address.

The top-level container for an LLVM IR program is called a module. The module contains a list of functions, global variables, the symbol table, and all other LLVM IR objects [YS19]. A function consists of basic blocks and each basic block contains instructions. An example of an LLVM IR program can be seen in Listing 22.

2.6 ELF

The Executable and Linkable Format (ELF) is a common file format for executables, object code, and shared libraries on Unix and Unix-like systems [Com95]. An ELF file consists of a header and data. The header contains metadata about the ELF file, for example its class (32-bit or 64-bit), the endianness, or the file type (executable, shared object, etc.). The data of the ELF file is divided into sections. These sections include executable code, initialized data, uninitialized data, read-only data, symbol tables, relocation information, and more. For example, the .text section contains the executable code, the .plt section contains a jump table used for dynamic linking, the .rodata section contains read-only data, and the .data section contains modifiable data such as global and static variables.

2.7 MCTOLL

S. B. Yadavalli and A. Smith mentioned that new ISAs are constantly being designed, resulting in the need for (re)engineering binary translators for the resulting legacy ISAs [YS19]. They argued that as compiler backends are irrespectively developed for these new ISAs, a static binary translator should be presented that leverages those backends. This was the motivation for MCTOLL: an open source LLVM-based tool created and maintained by Microsoft that raises ELF binaries to LLVM IR and extensively uses data structures, algorithms, and the code generation from the LLVM infrastructure [Fin22].

At the moment, MCTOLL mainly supports raising x86 binaries, with support for raising ARM binaries being in its early stages. The raising process of machine code to LLVM IR consists of six phases. The ELF binary is disassembled to a list of `MCInsts`, which essentially are encoded instructions in a generic LLVM data structure. This list is then used to build a CFG of `MachineBasicBlocks` containing `MachineInstrs`, which represent instructions prior to encoding. Multiple walks over the CFG are necessary for fully forming the LLVM IR. These walks identify function prototypes and jump tables and raise instructions to LLVM IR. After the LLVM IR is fully formed, it is emitted. The different phases are shown in Figure 3. The first two phases are already implemented in a generic way for every architecture adopted by LLVM.

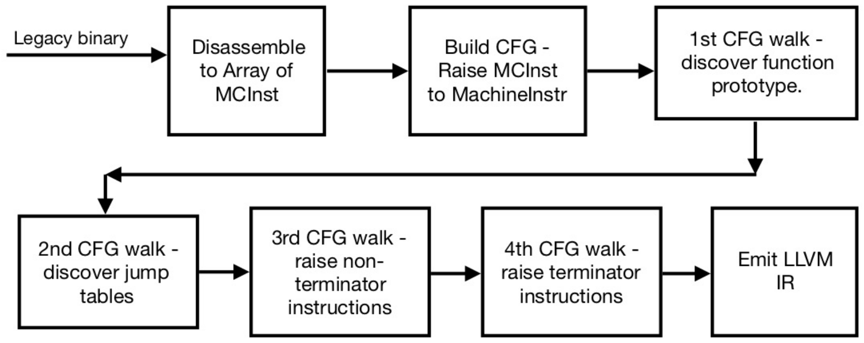
Legacy binary
Disassemble to Array of MCInst
Build CFG - Raise MCInst to MachineInstr
1st CFG walk - discover function prototype.
2nd CFG walk - discover jump tables
3rd CFG walk - raise non-terminator instructions
4th CFG walk - raise terminator instructions
Emit LLVM IR

Figure 3: The phases of the raising process of MCTOLL, adopted from [YS19].

To be able to raise an ELF binary to LLVM IR using MCTOLL, it needs to be supplied with the header files used by the program. For example, when one wants to raise a program that uses the puts function, MCTOLL needs to be supplied with the full path to the stdio.h header file. At the moment, MCTOLL is not able to raise some C++-related features such as virtual tables and exception handling.

2.8 Related Work

In a recent study [Fin22], the efficacy of static binary translation was investigated using MCTOLL. Their experiments consisted of both cross-architecture binary translation (x86 to ARM) and same-architecture binary translation (x86 to x86), comparing the runtime performance of the raised and recompiled binary to that of the native binary using the phoenix-2.0 benchmark [RRP+07]. They found that same-architecture binary translation (raising the x86 binary to LLVM IR and re-compiling the optimized IR to a x86 binary) had positive effects on the runtime performance and in some cases on the size of the binary. The last finding is especially interesting in the context of RISC-V binaries, where optimization of binary size holds significant implications for resource-constrained embedded systems and IoT devices. However, the LLVM-based tool MCTOLL currently only supports raising x86 and ARM binaries [YS19]. It is also worth mentioning that Fink exclusively conducted experiments using Clang, but not with the C-compiler of the GNU Compiler Collection (GCC). Furthermore, a *dynamic* binary translator that is able to translate RISC-V binaries to LLVM IR has been presented [Eng21]. However, because of the dynamic nature of this library, it is not suitable for our research.

Several static binary translators that lift machine code to LLVM IR have been proposed in the past, such as LLBT [SCHY12], rev.ng [DFFA18], and RetDec [KMZ17]. However, none of the proposed static binary translators support lifting RISC-V binaries. This likely stems from the fact that RISC-V is usually the target ISA of binary translation, because of it being a more modern architecture, and binary translation is mostly used for the migration of legacy software.

This research aims to expand the MCTOLL tool by implementing the RISC-V binary raiser to be able to analyze the effectiveness of same-architecture binary recompilation for 64-bit RISC-V binaries.

3 Implementation

This chapter describes the implementation of the binary translation process for RISC-V ELF binaries using the RISC-V binary raiser within MCTOLL. It describes how the first, third, and fourth CFG walks (corresponding to the third, fifth, and sixth phases in Figure 3) are implemented in our RISC-V binary raiser. The first and second phases were already generically implemented using the LLVM code generators. Furthermore, the second CFG walk is skipped, because discovering jump tables has not yet been implemented. Some boilerplate code was already present for the RISC-V binary raiser in the MCTOLL repository, which enables the use of the RISC-V binary raiser. Refer to Appendix A for an example of the full raising process.

It is important to note that the binaries are assumed to be unoptimized when raising and use the standard RISC-V calling convention. A lot of information, which may be needed for discovering the function prototypes or raising the machine functions, is lost otherwise.

Additionally, it is essential to differentiate between a *machine* function, *machine* basic block, and *machine* instruction, and their counterparts: function, basic block, and instruction. A *machine* entity refers to elements in the assembly or binary code, whereas the other refers to elements in the LLVM IR.

Finally, for all implemented features, the x86 binary raiser within MCTOLL has been used as a reference. However, there are some differences in implementations due to the different natures of RISC-V and x86. In general, the same algorithmic ideas have been used as those used by the x86 binary raiser, for example discovering function prototypes and promoting registers to the stack, but some changes and simplifications were made for the RISC-V binary raiser.

3.1 Representing pointer types

Before we begin with describing the discovery of function prototypes and the raising of machine functions, it is important to note a key issue regarding pointer types and 64-bit integers. In general, there is no foolproof method of determining whether a 64-bit word represents a pointer (`ptr`) or a 64-bit integer (`i64`). Consider Listing 2, which illustrates a machine function that accepts a 64-bit word argument. This argument could be either a `ptr` or a `i64`. In this case, it is not possible to determine whether the 64-bit integer contained in register `a0` represents an address or an integral value.

```
func:
    sd a0, -24(s0)
    ld a0, -24(s0)
    call <func>
    ret
```

Listing 2: A RISC-V machine function that uses the `sd` instruction to store its first argument on the stack. Prologue and epilogue instructions have been omitted.

Due to this ambiguity, we will not use the ptr type to represent addresses. Instead, we will always use the i64 type for both addresses and 64-bit integers. This approach does not lead to immediate issues, since addresses and 64-bit integers are essentially the same at the machine level. However, certain LLVM IR instructions such as loads, stores, and GEP instructions do require a ptr value. External functions not part of the ELF can also have functions with ptr arguments.

To address this problem introduced by always using the i64 type for 64-bit word instructions, we will convert between i64 and ptr types where needed by adding additional inttoptr and ptrtoint instructions. While this will significantly increase the size of the LLVM IR for some binaries, these instructions will be optimized out during subsequent optimization steps.

This approach results in the LLVM IR snippet seen in Listing 3 frequently occurring, which shows an i64 being converted to a ptr via an inttoptr instruction. The pointer is then used as an operand for a getelementptr instruction together with an index, in this case effectively calculating the address of the second element of an array.

```
%0 = ...
%1 = inttoptr i64 %0 to ptr
%2 = getelementptr inbounds i64, ptr %1, i32 1
```

Listing 3: An LLVM IR snippet representing an address calculation.

3.2 Discovering function prototypes

The first CFG walk is for discovering the function prototypes of the machine functions. A function prototype is a declaration of a function, consisting of an identifier and a type signature. To correctly raise a binary to LLVM IR, we need to discover the type signatures of the functions, as these are used when raising call machine instructions and when referencing argument values. This includes discovering the return type, the amount of parameters, and their respective types.

3.2.1 Discovering return types

For discovering the return type of a machine function, we differentiate between three cases: the function does not return a value (i.e., void), the function returns a 32-bit integer (i32), or the function returns a pointer type or 64-bit integer (i64). In RISC-V, the return register is a0. To determine whether the machine function returns a value, we must establish whether this register is defined in the machine function. However, a0 is also used as the first argument of a function call. To accurately determine whether a machine function returns a value, we must only consider the machine instructions from the last call instruction to the end of the machine function. See Listing 4a and Listing 4b for a comparison between a machine function that does not return a value and one that does.

Listing 4a shows that func defines register a0, however, this register is used as an argument and not as a return value. When we only consider the machine instructions after the call

```
func:
    li a0, 0
    call <func>
    ret
```

(a)

```
func:
    li a0, 7
    ret
```

(b)

```
func:
    auipc a5, 2
    addi a5, a5, <X>
    mv a0, a5
    ret
```

(c)

Listing 4: RISC-V machine functions with void (a), i32 (b), and i64 (c) return types.

instruction, we can conclude that this function does not return anything. In Listing 4b, func does define a0 after the last function call (in this case no function call takes place), so this machine function does have a return type.

To differentiate between a return type of i32 and i64, we must examine the instruction that defines the register used for defining a0. In general, the return value is defined through a move register instruction (e.g., mv a0, a5), through a binary operation (e.g., addw a0, a5, 1), or through a move immediate instruction (e.g., li a0, 7). A li instruction always indicates an i32 return type. For the other two, we need to determine how the register of the second operand is defined. When this register is defined via a 64-bit word load instruction (i.e., ld) or via an PC-relative or absolute access (i.e., auipc or lui respectively, both followed by an accompanying addi/ld), we can conclude that the return type of the machine function is i64. A return type of i32 is assumed otherwise. Listing 4c shows a machine function that returns the address of a global variable X, in which case we deduce the return value to be i64.

3.2.2 Discovering argument types

For discovering the arguments of the machine function and their types, we again differentiate between i32 and i64. The arguments are passed via the argument registers a0 to a7. When a function has more than eight arguments, the arguments are passed via the stack[1]. In unoptimized RISC-V programs, the argument registers are usually moved to a local register or stored to the stack after the prologue. To identify the instructions that represent the storing or moving of the passed arguments, we need to check two conditions:

1. The register that is being moved or stored is one of the argument registers (a0 - a7).
2. The register has not yet been defined in the current basic block.

Only the entry machine basic block of the machine function is considered, as all argument registers are moved or stored in the entry machine basic block. See Listing 5a and Listing 5b for a comparison between a function that has two i32 arguments and a function that has two i64 arguments.

Listing 5a shows that three argument registers are being moved or stored. However, the register a5 of the second move instruction is already defined in the basic block. Furthermore, this second move instruction would most likely not be in the entry basic block of the machine

[1]At the moment, the discovery of such arguments is unsupported.

```
func:
    mv a5, a0
    sw a1, -20(s0)
    ...
    mv a0, a5
    ret
```

(a)

```
func:
    sd a0, -24(s0)
    sd a1, -32(s0)
    ...
    mv a0, a5
    ret
```

(b)

Listing 5: RISC-V machine functions with two i32 arguments (a) and two i64 arguments (b).

function. Therefore, the a5 register is not classified as an argument and it is concluded that the machine function has two arguments: a0 and a1, both of i32 type. Similarly, Listing 5b shows two argument registers being stored to the stack using a double word store, indicating that the machine function has two arguments, both of i64 type.

3.3 Raising machine instructions to LLVM IR

Once the function prototypes are discovered, the third CFG walk is performed[2]. The machine functions are raised to LLVM IR, with each machine function being raised individually. The machine basic blocks within the machine function are traversed in loop traversal order, ensuring that the machine basic blocks are processed in a manner that respects the control flow of the program. With loop traversal order, some machine basic blocks are passed twice. We will only process the primary pass of each machine basic block. During this traversal, a corresponding basic block is created and stored for future reference. All raised instructions are inserted into the basic block associated with the machine basic block. Additionally, these basic blocks also serve as operands for branch instructions.

Every machine instruction within a machine basic block is traversed and will be raised to one or more LLVM IR instructions. Certain instructions, such as those involving the loading or storing of the return address or stack pointer, are skipped as they are unnecessary to raise. In the initial pass, terminator instructions are also skipped since the target basic block might not yet be raised. Information necessary for raising these instructions in a second pass is recorded.

3.3.1 Tracking register values

Because of the SSA characteristic of LLVM IR, the infinitely available virtual registers can only be assigned once. This is not the case for machine registers, which are reused many times. Therefore, a one-to-one mapping between machine register and a virtual register of LLVM IR is not possible. Similar to the x86 implementation, we will maintain a mapping from each machine register to the current SSA value that is assigned to it. This mapping is used whenever a register's value needs to be set or retrieved in response to a machine instruction. An example which illustrates this mapping can be seen in Listing 6a and Listing 6b.

[2]The second CFG walk is skipped, because discovering jump tables is not yet supported

```
func:
    mv a4, a0
    li a5, 5
    bge a5, a4, <target>
    ...
```

(a)

```
define void @func(i32 %0) {
  %2 = icmp sge i32 5, %0
  br i1 %2, <target>, <fall-through>
  ...
}
```

(b)

Listing 6: A RISC-V machine function containing move instructions (a) and the corresponding LLVM IR function (b) raised using our RISC-V binary raiser.

Listing 6a defines the registers a4 and a5 and subsequently uses those registers in a comparison instruction. Instructions such as mv or li do not have a corresponding instruction in the LLVM IR. The register mapping is updated with the new definition and following instructions can use the current definitions for each register. This can be seen in Listing 6b, which does not contain any instructions for moving the values. The values have been directly retrieved from the register-value map.

3.3.2 Promoting registers to the stack

The register-value map is maintained separately for each machine basic block to handle branching appropriately. If a register value is not locally defined within the current machine basic block, we search for its definition in the immediate predecessors of the machine basic block. There are two scenarios to consider[3]:

1. **Exactly one predecessor defines the register**: We simply use this definition as the value for the current machine basic block.
2. **All predecessors define the register**: When all predecessors define the register, we must promote the register to the stack using an alloca instruction. Each branch will then store to this designated memory address and the current machine basic block will load from this memory address.

When the referenced register number is not defined locally and also not defined by any of its predecessors, it is attempted to retrieve the value from the function arguments. This will only be attempted if the referenced register is an actual argument register, i.e., registers a0 to a7. This algorithm more or less matches the algorithm used by the x86 implementation.

3.3.3 Tracking stack values

To be able to raise machine operands that reference memory on the stack, we introduce an alloca instruction which will represent a specific stack slot. This is maintained in the form of a mapping from stack offset to the current alloca instruction representing that stack slot. This mapping is used whenever a load or store instruction from or to the stack is raised. This offset-slot map is not maintained separately for every machine basic block, because the stack slots should be available for all machine basic blocks of the machine function. This approach

[3]The scenario where more than one predecessor, but not all, defines the register is undefined.

is a limitation of the current state of the RISC-V binary raiser, as not all "stack slots" are explicitly defined. For example a stack load with offset −40 might be accessing the pointer stored at offset −44 with an offset of 4 bytes.

3.3.4 Type coercion and widening

For function calls, the argument types must exactly match the types specified in the function's type signature. Similarly, for return statements, the type of the return value must match the function's return type exactly. For example, calling a function that expects an i64 value with an i32 value is not allowed. Therefore, it is necessary to coerce the types of the arguments or return value to match the expected type of the type signature in case of a type mismatch. For integer values, this coercion will be done using a trunc or sext instruction. When the expected type is a pointer type and the actual type is an i64 (or vice-versa), we will generate an inttoptr (or ptrtoint) instruction, as described in Chapter 3.1. If the expected type is a pointer but the actual type is a constant zero, the zero will be replaced with a constant null pointer. Any other type mismatches will not be coerced and will result in an error.

Similarly, binary operations and comparison instructions also require matching types. In cases of mismatching types, we will widen the smaller type to match the bit width of the larger type. This widening is only applicable to integer types.

3.3.5 Raising return instructions

Return instructions are, by far, the simplest machine instructions to raise to LLVM IR. First, the return type of the discovered function prototype is consulted. In the case of a void return type, an LLVM IR return instruction is created without an SSA value. For all other return types, an LLVM IR return instruction is created with the value currently assigned to the return register, i.e., a0. In the case of a type mismatch between the value currently assigned to the return register and the return type of the discovered prototype, the return type of the discovered prototype is coerced upon the return value. An example illustrating the raising of a return instruction can be seen in Listing 7a and Listing 7b.

```
func:
    li a5, 3
    mv a0, a5
    ret
```

(a)

```
define dso_local i32 @func() {
  ret i32 3
}
```

(b)

Listing 7: A RISC-V machine function returning the value 3 (a) and the corresponding LLVM IR function (b) raised using our RISC-V binary raiser.

3.3.6 Raising binary operations

To raise an instruction that represents a binary operation, we must determine the SSA values for the left-hand side and the right-hand side of the machine instruction. In the case of binary operations, the left-hand side is a register and the right-hand side can be either a register or an immediate. For the addition instruction shown in Listing 8a, the SSA value assigned to the register a0 will be determined (in this case the first argument value is used, because a0 is not defined yet and it is an argument register) and a constant integer value representing the immediate value 1 will be created. These values will then be used to create an LLVM IR binary operator instruction, which will be appended to the current basic block. It is possible that a type widening is needed for either the left-hand side or the right-hand side. All binary operators are raised in this manner. Listing 8b shows the resulting raised LLVM IR function corresponding to Listing 8a.

```
func:
    addiw a0, a0, 1
    ret
```

(a)

```
define dso_local i32 @func(i32 %0) {
    %2 = add i32 %0, 1
    ret i32 %2
}
```

(b)

Listing 8: A RISC-V machine function containing an addition instruction (a) and the corresponding LLVM IR function (b) raised using our RISC-V binary raiser.

A single exception is addition instructions that are used to compute an address using a stack offset[4]. For example, consider the instruction add a5, s0, -20, which computes the address of the stack slot at offset -20. When raising these kinds of instructions we will simply set the value of the destination register to be equal to the alloca instruction currently functioning as the specified stack slot, no LLVM IR instruction will be generated for these addition instructions. Adding an offset to an actual pointer is not a problem, because of pointers being represented as 64-bit integers.

3.3.7 Raising load and store instructions

Both load and store instructions are raised using a similar approach. The process begins by determining the pointer to load from or store to, which involves using the second and third operands of the load/store instruction. The second operand is a register containing an address, and the third operand is an immediate value serving as an offset to that address. After determining the pointer and applying any necessary offset, an LLVM IR load or store instruction is created using this pointer.

For store instructions, the current value assigned to the register specified by the first operand will be used to create the LLVM IR store instruction. For load instructions, the created LLVM IR load instruction's result is assigned to the register specified by the first operand. This LLVM IR load or store instruction is then appended to the current basic block.

For determining the pointer to load from or store to, we consider three scenarios:

1. The load/store is a stack load/store.
2. The load/store is not a stack load/store and the offset is zero.

[4]This exception arises solely due to the limitations in advanced stack access.

3. The load/store is not a stack load/store and the offset is non-zero.

In the first scenario, the pointer to load from or store to is the alloca functioning as the specified stack slot, similar to the situation described in Chapter 3.3.6. For example, consider Listing 9a. This unoptimized RISC-V program first stores its argument to the stack, followed by immediately loading this value again, moving it to the return register, and returning to the caller. Listing 9b shows how this machine function is raised. An alloca instruction is created for the stack slot at offset -20. The argument is stored at this address, then loaded, and finally returned.

```
func:
    mv a5,a0
    sw a5,-20(s0)
    lw a5,-20(s0)
    mv a0,a5
    ret
```

(a)

```
define dso_local i32 @func(i32 %0) {
    %2 = alloca i32, align 4
    store i32 %0, ptr %2, align 4
    %3 = load i32, ptr %2, align 4
    ret i32 %3
}
```

(b)

Listing 9: A RISC-V machine function that loads from and stores to the stack (a) and the corresponding LLVM IR function (b) raised using our RISC-V binary raiser.

For the second scenario, the pointer to load from or store to is simply the address contained in the specified register. This is handled similarly as the previous example, minus the alloca instruction.

In the case of the third scenario, we need to compute the address using a getelementptr instruction using the immediate value as the index. However, the immediate value of the machine instruction represents the offset in bytes, whereas getelementptr instructions work with indices. We need to determine the alignment of the operation represented by the opcode of the machine instruction (usually either 4 or 8 bytes) and use that to compute the index. Listing 10a shows an example of such a load instruction, in this case loading an element of an array. Listing 10b illustrates how such offsetted loads are raised. An inttoptr instruction is needed, because getelementptr instructions can only be used with pointers. The offset of 8 bytes is converted to an index of 2, because the alignment of the load instruction is a single word, i.e., 4 bytes.

3.3.8 Raising call instructions

To raise call instructions, we first need to determine the function prototype of the target function. This is achieved by using the offset of the jal instruction as a relative offset to the current instruction to compute the offset of the target machine function. If the target machine function is found in this manner, it is locally defined, and we can simply use the function prototype created in the second CFG walk.

When the target function is not locally defined, the offset refers to an entry of the PLT section of the ELF. Listing 11a illustrates such an entry. The instructions in this entry are disassembled and the PC-relative offset is computed using the values of the auipc and ld

```
func:
    sd  a0,-24(s0)
    ld  a5,-24(s0)
    lw  a5,8(a5)
    mv  a0,a5
    ret
```

(a)

```
define dso_local i32 @func(i64 %0) {
  %2 = alloca i64, align 8
  store i64 %0, ptr %2, align 8
  %3 = load i64, ptr %2, align 8
  %4 = inttoptr i64 %3 to ptr
  %5 = getelementptr inbounds i32, ptr %4, i32 2
  %6 = load i32, ptr %5, align 4
  ret i32 %6
}
```

(b)

Listing 10: A RISC-V machine function that loads from the address argument with an offset (a) and the corresponding LLVM IR function (b) raised using our RISC-V binary raiser.

instructions. This offset should point to a dynamic relocation record with an associated symbol, an example of which can be seen in Listing 12. Finally, the name of this symbol is used to locate the function using the included files described in Chapter 2.7. Listing 11b shows what an LLVM IR function calling an externally defined function would look like.

```
printf@plt:
    auipc t3, 2
    ld t3, t3, -1408
    jalr t1, t3
    nop
```

(a)

```
define dso_local i32 @main() {
    %1 = call i32 (ptr, ...) @printf(ptr @.str)
    ret i32 0
}

declare dso_local i32 @printf(ptr, ...)
```

(b)

Listing 11: A part of the PLT section of a RISC-V ELF binary (a) and an LLVM IR function illustrating the use of such an externally defined function (b).

```
DYNAMIC RELOCATION RECORDS
OFFSET           TYPE               VALUE
0000000000002018 R_RISCV_JUMP_SLOT  __libc_start_main@GLIBC_2.34
0000000000002020 R_RISCV_JUMP_SLOT  printf@GLIBC_2.27
```

Listing 12: The dynamic relocation records of a RISC-V ELF binary.

After the function prototype is determined, an arguments vector will be constructed based on the current values assigned to the argument registers. The amount of registers used is determined by the number of parameters of the prototype of the target function. For example, if the target function only accepts two arguments, only registers `a0` and `a1` will used to construct the arguments vector. In the case of a variadic function, exclusively the argument registers that are defined locally are added to the arguments vector to prevent using too many arguments. In some cases the types of the arguments must be coerced to that of the prototype of the target function, as was described in Chapter 3.3.4. Finally, we create an

LLVM IR call instruction using the target function and the constructed arguments vector, whose value will be assigned to the return register.

Listing 13a shows a machine function that calls another function. In this case, it is a locally defined function. From the discovered function prototype, it is known that this function accepts two arguments, both of i32 type. The values assigned to registers a0 and a1 are used as arguments and an LLVM IR call instruction is created. Listing 13b shows the raised result.

```
main:
    li a1, 5
    li a0, 3
    call func
    ret
```

(a)

```
define dso_local i32 @main() {
  %1 = call i32 @func(i32 3, i32 5)
  ret i32 %1
}
```

(b)

Listing 13: A RISC-V machine function that arranges two argument registers and calls a function (a) and the corresponding LLVM IR function (b) raised using our RISC-V binary raiser.

3.3.9 Raising PC-relative and absolute accesses

PC-relative accesses are characterized by the auipc instruction paired with another instruction, typically an addi instruction. Importantly, the addi instruction does not necessarily immediately follow the auipc instruction. This instruction pair computes an address by shifting the immediate value of the auipc instruction 12 bits to the left and adding the 20-bit immediate value of the addi instruction to form a relative offset from the current instruction's position. The final address is obtained by adding the address of the current instruction and the address of the text section. In general, this results in an address within the dynamic relocations, read-only data, or data section of the ELF binary, which may correspond to a global variable or read-only data such as a string.

To raise PC-relative accesses, the described offset is used to resolve the global variable as either a dynamic relocation, read-only data, or modifiable data, in that specific order. The accessed symbol is used to create an LLVM global variable, using its linkage type (i.e, external, internal, etc.), size, and contents. The size determines the type and alignment of the global variable. For instance, a symbol with a size of 4 bytes results in a global variable of type i32. This only applies to dynamic relocations and modifiable data. In contrast, for read-only data no symbol is consulted. The data is directly extracted from the read-only data section and is treated as an array of bytes. Global variables are instantiated upon the first access, subsequent accesses reuse the created global variable. Finally, the created or retrieved global variable is assigned to the register specified by the first operand of the addi instruction. Listing 14 and Listing 15 show how such PC-relative accesses are raised.

Absolute accesses are characterized by a lui instruction paired with a complimentary addi instruction and are raised in a nearly identical manner. The only difference lies in the offset computation. For absolute accesses, the computed offset is used directly without adding the

```
main:
    auipc a5, 2
    addi a5, a5, -1640 # 2008 <C>
    lw a5, a5, 0
    mv a1, a5
    auipc a0, 0
    addi a0, a0, 36 # 6a0 <_IO_stdin_used+0x8>
    jal ra, 5a0 <printf@plt>
    li a5, 0
    mv a0, a5
    ret
```

Listing 14: A RISC-V machine function loading the address of both a global variable and read-only data.

lst:implementation:global-variable.riscv

```
@C = dso_local global i32 8, align 4
@.rodata13 = private unnamed_addr constant [12 x i8] c"↩
   \01\00\02\00\00\00\00\00%d\0A\00", align 8

define dso_local i32 @main() {
  %1 = load i32, ptr @C, align 4
  %2 = call i32 (ptr, ...) @printf(ptr getelementptr inbounds ([12 x i8↩
      ], ptr @.rodata13, i64 0, i32 8), i32 %1)
  ret i32 0
}

declare dso_local i32 @printf(ptr, ...)
```

Listing 15: An LLVM IR function corresponding to Listing 14, raised using our RISC-V binary raiser.

address of the current instruction and the base address of the text section, as it represents an absolute address.

3.3.10 Raising terminator instructions

Most machine basic blocks end with a terminator instruction. As was described in Chapter 2.3, a terminator instruction is an instruction that *terminates* a basic block, e.g., a jump or branch instruction. It is possible that the target basic block of such a terminator instruction is not yet raised. For this reason, all terminator instructions are skipped in the initial pass and necessary information – such as the machine instruction and the register values at that time – is recorded for use at the fourth CFG walk[5]. It is also possible that a machine basic block does *not* end with a terminator instruction. In this case, a fall-through unconditional branch instruction will be added, with the next basic block as its target. The fourth CFG walk can be divided into two different scenarios: unconditional branches and conditional branches.

[5]This is not really a CFG walk, as only the recorded terminator instructions are processed.

The process of raising unconditional branches is quite straightforward. First, the target machine basic block is identified using the offset operand of the jump machine instruction. As mentioned in Chapter 3.3, a mapping between the machine basic block and its associated basic block is maintained. Using this mapping, the basic block associated with the machine basic block is determined. Finally, an unconditional LLVM branch instruction targeting the appropriate basic block is created and inserted at the end of the current basic block.

Raising conditional branches is a bit more complicated. Similar to binary operation instructions, the SSA values for the left-hand side and right-hand side of the comparison are determined. The left-hand side is a register, while the right-hand side can be either a register or an immediate value. Some comparison instructions implicitly compare against zero, which requires slightly different handling. Types are once again widened as needed. Using these values, an LLVM IR comparison instruction is created. Next, the fall-through basic block and the destination basic block are identified. These, along with the created comparison instruction, are used to create an LLVM IR branch instruction, which is then inserted at the end of the current basic block.

```
func:
    addi    sp,sp,-48
    sd      s0,40(sp)
    addi    s0,sp,48
    mv      a5,a0
    sw      a5,-36(s0)
    sw      zero,-20(s0)
    lw      a5,-36(s0)
    sext.w  a4,a5
    li      a5,2
    blt     a5,a4,68e <func+0x26>
    li      a5,1
    sw      a5,-20(s0)
    j       694 <func+0x2c>
    li      a5,3
    sw      a5,-20(s0)
    lw      a5,-20(s0)
    mv      a0,a5
    ld      s0,40(sp)
    addi    sp,sp,48
    ret
```

Listing 16: A RISC-V machine function containing both an unconditional branch and a conditional branch.

Listing 16 and Listing 17 show how both unconditional and conditional branches are raised. For the conditional branch machine instruction, an LLVM IR comparison instruction (icmp) is created, whose result is used in the LLVM IR branch instruction (br) right after it. The unconditional branch machine instruction is raised to an LLVM IR branch instruction without a comparison.

```
define dso_local i32 @func(i32 %0) {
  %2 = alloca i32, align 4
  store i32 %0, ptr %2, align 4
  %3 = alloca i64, align 8
  store i64 0, ptr %3, align 8
  %4 = load i32, ptr %2, align 4
  %5 = add i32 %4, 0
  %6 = icmp slt i32 2, %5
  br i1 %6, label %8, label %7

7:                                                ; preds = %1
  store i32 1, ptr %3, align 4
  br label %9

8:                                                ; preds = %1
  store i32 3, ptr %3, align 4
  br label %9

9:                                                ; preds = %7, %8
  %10 = load i64, ptr %3, align 8
  %11 = trunc i64 %10 to i32
  ret i32 %11
}
```

Listing 17: An LLVM IR function corresponding to Listing 16, raised using our RISC-V binary raiser.

4 Evaluation

This chapter evaluates the quality of the RISC-V binary raiser by comparing supported features between the RISC-V and the more mature x86 binary raisers within MCTOLL. Furthermore, the effectiveness of same-architecture binary recompilation for RISC-V binaries is evaluated by assessing the binary size and runtime performance of the recompiled RISC-V binaries. Refer to Appendix B for information on how to reproduce to results of the evaluation.

4.1 Qualitative Comparison of Feature Support

To compare the supported features of both binary raisers, we conducted a qualitative comparison based on a list of relevant language features, mostly corresponding to the tests written for our RISC-V binary raiser. The test programs will be raised with both our RISC-V binary raiser and the x86 binary raiser and the outputted IR will be evaluated, if the raising process did not fail. The comparative results are summarized in Table 1.

Feature	x86-64	RISC-V
Return Statements	✓	✓
Binary Operations	✓	✓
Internal Functions	✓	✓
External Functions	✓	✓
Vararg Functions	✓	✓
Local Variables	✓	∼
Global Variables	∼	✓
Strings	✓	∼
Arrays	✓	✓
Matrices	✓	✓
Loops	✓	✓
Branches	✓	✓
Switch Statements	✓	∼
Structures	✓	∼
Pointers	✓	✓
Memory Allocation	✓	✓
File I/O	✓	×
Vector Instructions	✓	×

Table 1: A qualitative comparison of supported features between the x86 and RISC-V binary raisers within MCTOLL, where a checkmark represents full support, a tilde represents partial support, and a cross represents no support.

It is evident that the x86 binary raiser is more mature. It offers strong feature support across various functionalities. The only incorrect behaviour observed in the x86 binary raiser was related to initialized global arrays and matrices. This resulted in IR with instructions that did not dominate all their uses.

The RISC-V binary raiser is capable of raising most basic programs and shows promising potential. Despite this, the RISC-V binary raiser is not able to raise all benchmarks, due to some identified limitations, as will be discussed in the next section. For a description of all limitations of the RISC-V binary raiser, refer to Chapter 6.

4.2 Setup

To evaluate the binary size and runtime performance, we will use the same `phoenix-2.0`[6] benchmarks that M. Fink used in his research. However, due to time constraints, our RISC-V binary raiser is not yet fully capable of raising the benchmarks without issues. For this reason, some benchmarks have been slightly modified, while others have been entirely excluded. For more details on the current limitations, refer to Chapter 6. In particular:

- Our RISC-V binary raiser currently lacks support for vector instructions. Consequently, we were unable to raise the `linear_regression` benchmark, and it is excluded from the evaluation.
- The binary raiser also does not support arrays and/or structs on the stack, leading to issues with functions such as `fstat` and `gettimeofday`. Additionally, functions like `pread` and `mmap` still present some challenges. As a result, reading from files has been completely removed, and all test data is hard-coded into the benchmarks. The `word_count` and `string_match` benchmarks use a *Lorem Ipsum* text and the `pca` and `matrix_multiply` use randomly generated data using a user-defined seed.
- Although our RISC-V binary raiser is able to raise the `kmeans` benchmark, the output is incorrect for unknown reasons. To avoid any misleading conclusions, this benchmark is omitted from the evaluation.
- The `histogram` benchmark originally involved reading from a BMP file, which could not be easily replaced with hard-coded data. Additionally, this benchmark did not provide significant added value, as its covered language features are already addressed by other benchmarks. Therefore, the `histogram` benchmark is not included in the evaluation.

It is worth noting that the x86 binary raiser within MCTOLL is, with the latest commit of MCTOLL, also not able to raise the benchmarks without problems. This most likely has something to do with environment, which illustrates the very sensitive nature of binary translation.

For the runtime performance and binary size experiments, we will consider four different binaries:

1. A binary compiled from source using GNU, without optimizations.
2. A binary compiled from source using GNU, with optimizations.
3. A raised binary, recompiled using LLVM without optimizations.
4. A raised binary, recompiled using LLVM with optimizations.

[6] https://github.com/kozyraki/phoenix

Both raised binaries use the native unoptimized binary (1) as the input. For optimization, the flag `-O3` has been used instead of `-Oz`, because the latter did not have any different effect for these specific programs.

All benchmarks were run on an `x86` machine running `Zorin OS 17.1` with an `Intel Core i7-13700H` (14 cores, 20 threads), and 2×16 GB of DDR5 RAM. The native binaries have been compiled with `GNU Compiler Collection 11.4.0` and the raised binaries have been compiled with `Clang 15.0.4`, the version which will be build along side the LLVM project tree mentioned in Appendix B.

4.3 Binary Size

To evaluate the effectiveness of recompilation on the size of RISC-V ELF binaries, we will measure the size of the four binaries described in Section 4.2. However, because ELF sections can be removed or re-ordered during raising and optimization, alignment and padding can be added to the ELF, resulting in a skewed impression. For this reason, we will additionally measure the total size of the various sections of the binary, as well as the size of the text section on its own. The results can be seen in Figure 4, Figure 5, and Figure 6 respectively.

To count the sizes, we utilized `GNU size`[7] with the Berkeley output style. In this context, the *text* section encompasses sections related to program code and constant read-only static or global variables, such as `.interp`, `.text`, `.rodata`, `.plt`, and others. Similarly, the *data* section includes sections associated with non-zero initialized global and static variables, such as `.data`, `.dynamic`, etc. These two groups, along with the `.bss` section (zero-initialized global and static variables), are used to determine the combined size of the sections. An example output of GNU size can be seen in Listing 18.

```
text     data      bss      dec      hex filename
4089      704       16     4809     12c9 scripts/build/word_count
```

Listing 18: An example of output from the GNU size program, displaying the sizes of the text sections, the data sections, the bss section, and the total size in both decimal and hexadecimal.

Figure 4 shows that the binary size generally increases after raising and optimizing, with the exception of the `string_match` benchmark. This increase is likely due to alignment and padding of the sections. Although the size of the text-related sections of the `string_match` benchmark decreases, this decrease is not as significant as Figure 4 might suggest. So this significant decrease in size might also be attributed to padding and alignment of the ELF sections.

An interesting observation can be made for the optimized native binary of the `pca` benchmark, which significantly increases in size. However, this increase is not seen in the optimized raised binary. This increase in size can also be seen in Figure 5 and Figure 6, so this can not be explained via padding and alignment of sections. While an increase in size is expected for this

[7] https://www.man7.org/linux/man-pages/man1/size.1.html

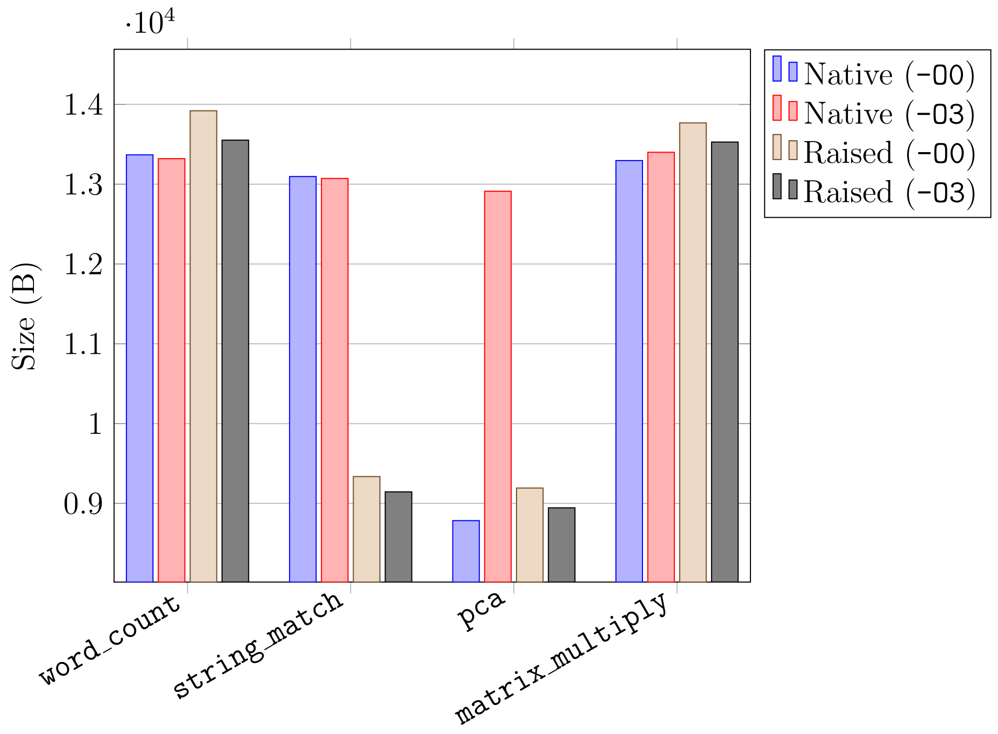

Figure 4: The binary size of the four benchmarks for both the native and raised binaries.

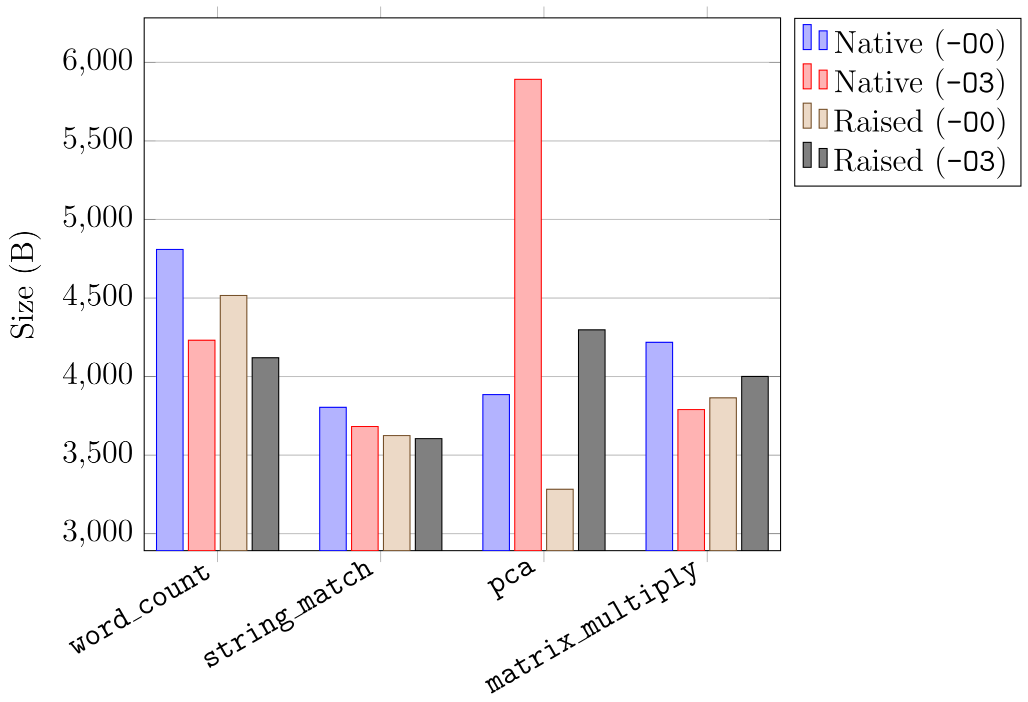

Figure 5: The combined size of the text, data, and bss sections for the four benchmarks for both the native and raised binaries.

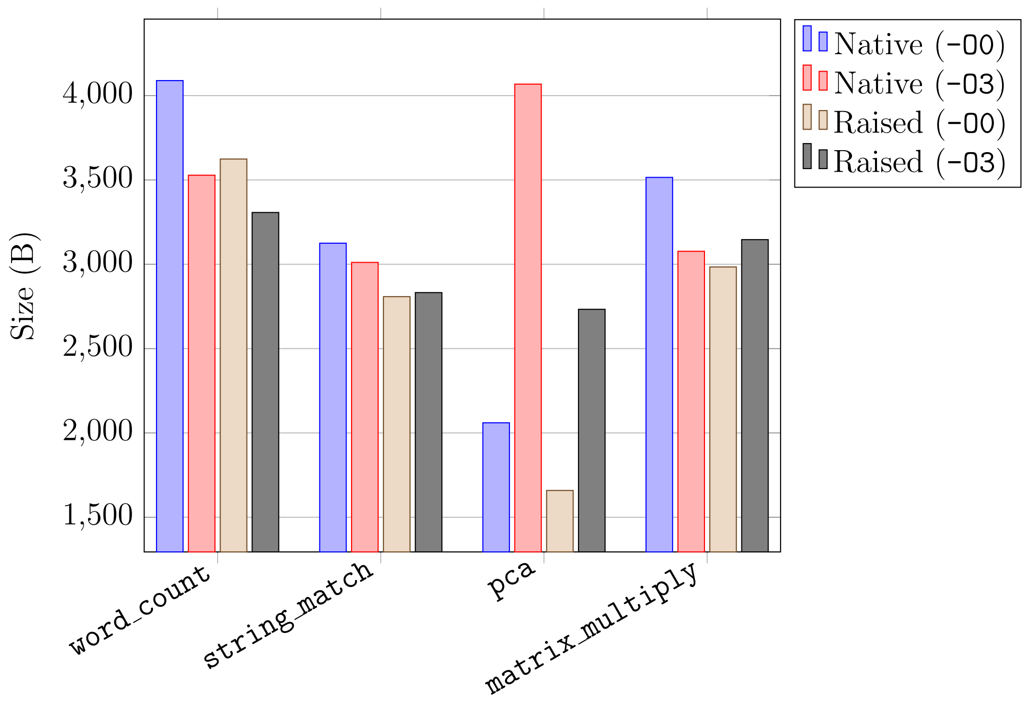

Figure 6: The size of the text section for both the native and raised binaries of the four benchmarks.

program, due to optimizations like loop unrolling, the extent of the increase is unusually large. The most plausible explanation is the difference in optimization and instruction selection between the GNU Compiler Collection and Clang.

Figure 5 and Figure 6 show that binary recompilation does have an effect on the size of the binary. Both the `word_count` and `string_match` benchmarks decrease in size when compared to both the unoptimized and optimized native binaries. Unexpectedly, the `matrix_multiply` benchmark decreases in size when compared to the unoptimized native binary. Excluding the optimized native binary discussed earlier, the `pca` benchmark shows an increase in size. This increase can be attributed to optimizations that do not always reduce the amount of instructions. For example, loop unrolling adds instructions to improve instruction scheduling.

The results of Figure 4 roughly match the results of the experiments conducted by M. Fink for the x86 binary raiser, with the exception of `string_match`. They concluded that recompiling to improve the size of the binary was not beneficial. However, Figure 5 and Figure 6 show that their experiment of just measuring the total size of the binary might give the wrong impression, because we measured a significant impact on the size of the ELF sections.

4.4 Runtime Performance

Although we did have access to a RISC-V development board[8], the board was rather old and it was not able to run our binaries because of an outdated `glibc` version. To still be

[8]HiFive Unleashed Developemnt Kit

able to measure the runtime performance, we will observe the amount of dynamic guest instructions for each binary using the RISC-V emulator of QEMU[9]. QEMU has multiple logging capabilities. We will be utilizing CPU logging, which logs the CPU state after every guest instruction. Using a simple script, we can determine the amount of occurrences of the PC register, which indicates the amount of instructions being executed. It is worth noting that QEMU, by default, tries to optimize the execution of guest instructions by chaining together sequences of instructions. This means it translates blocks of guest instructions into host instructions and then executes them as a unit, which can improve performance. However, this leads to an under count of instructions, because it executes blocks of instructions as a unit and skips intermediate states of the guest CPU. That optimization has been disabled for this experiment. The total amount of dynamic quest instructions for each binary can be seen in Figure 7.

The results illustrate that the raw overhead introduced by raising is significant, as can be seen from the unoptimized raised binaries ("Raised (`-O0`)") in Figure 7. This is in line with the results of M. Fink's research. The generated instructions are far from optimal and also contain numerous unnecessary casts and sign extensions. However, after optimizing the raised LLVM IR, the resulting binary outperforms the unoptimized native binary. It does not outperform the optimized native binary, except for loop-heavy programs such as the `pca` and `matrix_multiply` benchmarks. Once again, this can most likely be attributed to differences between the optimization passes of GCC and Clang.

[9]https://github.com/qemu/qemu

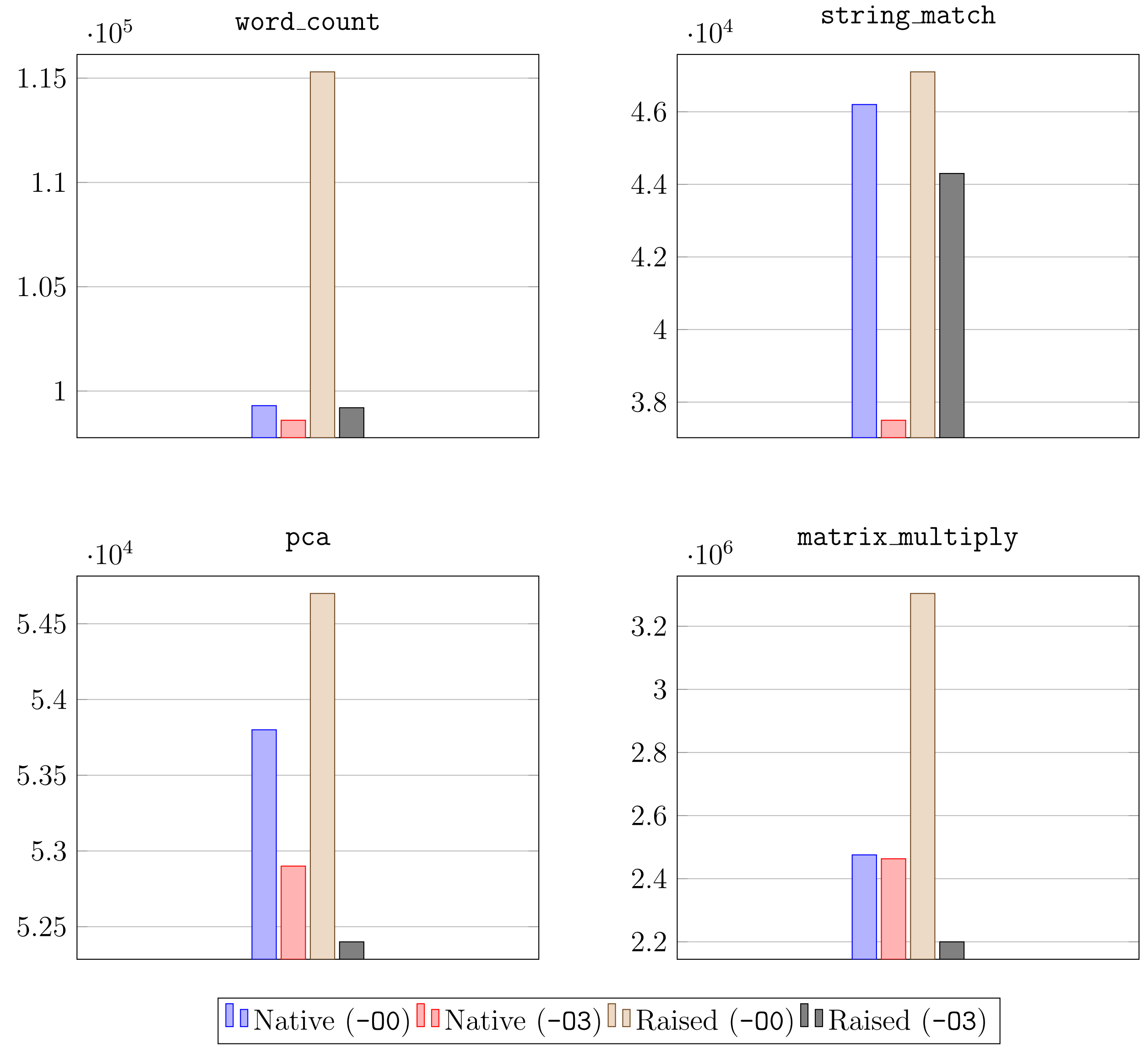

Figure 7: The total amount of dynamic guest instructions for each of the four benchmarks for both the native and raised binaries.

5 Conclusions

In this thesis, we have described the implementation of our RISC-V binary raiser and evaluated the effectiveness of same-architecture binary recompilation on the binary size and runtime performance of 64-bit RISC-V ELF binaries. Our findings indicate that same-architecture binary recompilation positively impacts the size of the sections of the ELF, though the total binary size may increase due to padding and alignment adjustments.

The raising process initially introduces substantial raw overhead in runtime performance. However, optimizing the raised LLVM IR effectively mitigates this overhead. For loop-heavy programs, we observed a significant improvement in runtime performance, outperforming both the unoptimized and optimized native binaries. No significant positive effect was found for programs that are not loop-heavy.

Finally, binary translation is highly sensitive to the environment. Variations in compiler or enabled RISC-V extensions can substantially influence the resulting machine code, thereby complicating the raising process. Even so, the RISC-V binary raiser demonstrates promising potential despite some identified limitations. Future efforts should aim to address these limitations, thereby improving the overall functionality and compatibility of the RISC-V binary raiser within MCTOLL.

6 Limitations and Future Work

This chapter describes the limitations of the RISC-V binary raiser within MCTOLL and the required future efforts for improving the overall functionality and compatibility of the RISC-V binary raiser.

In particular, the binaries are assumed to have been compiled with GCC and with the *"Compressed"* extension enabled. The reason for this is that GCC uses the compressed jump instruction `c.j` for unconditional branches, while Clang uses the pseudo-instruction `j`, which is an alias for a `jal` instruction with the `zero` register as the link register. Due to a presumed bug in the CFG building algorithm of MCTOLL, `jal` instructions with the `zero` register are not recognized as terminator instructions, resulting in incorrect basic blocks and ultimately an incorrect CFG.

Due to this limitation, we were unable to perform certain experiments. These included comparing binary size and runtime performance for binaries compiled with GCC versus those compiled with Clang, as well as testing the impact of enabling compressed instructions on binaries that were originally compiled without them.

The following is a list of *identified* unsupported features:

- Raising **32-bit binaries** is not supported and was outside the scope of this thesis.
- Raising **vector instructions** is not supported and was outside the scope of this thesis.
- Raising **optimized binaries** is not fully supported. Some limited testing has been done with optimized binaries, however, this might not always work correctly.
- Raising binaries compiled with **Clang** or binaries compiled **without the "Compressed" extension** is not supported.
- Raising binaries compiled **without the fno-stack-protector** flag is not supported, because otherwise functions such as `__printf_chk` would be included in the binary, which we were not able to find a corresponding header file for to include.
- **Discovery of jump tables**, the second CFG walk, is not implemented. As a result, complex switch statements that utilize jump tables are not accurately raised at this time. However, simple switch statements are supported.
- Support for **initialized global strings** is not implemented. In these cases, the symbol is stored in the data section of the ELF file, while the actual value resides in the read-only data section. This separation adds a layer of complexity, and a solution has not yet been devised due to time constraints. Nonetheless, addressing this issue should not be overly difficult.
- **Advanced stack accesses**, such as accessing properties of a local struct or elements of a local array, are not supported. This is because the struct or array is stored at a base offset on the stack (e.g., -40), and accesses occur at an offset from this base offset (e.g., -52). Our offset-slot map is not designed to handle these cases. As a result, local

variables representing structs or arrays are not supported. This also means that passing structs by value is not yet supported, as this typically occurs via the stack. However, passing structs by reference is supported.

- The limitations related to structs and stack accesses also affect **file I/O** functions. Operations such as using `fstat` or reading into a local buffer are currently not supported, due to use of local structs. For unknown other reasons, file I/O functions such as `mmap` and `pread` are also not supported.

References

[Bro24] Nick Brown. Risc-v for hpc: Where we are and where we need to go. *arXiv preprint arXiv:2406.12398*, 2024.

[CM96] Cifuentes and Malhotra. Binary translation: Static, Dynamic, Retargetable? In *1996 Proceedings of International Conference on Software Maintenance*, pages 340–349. IEEE, 1996.

[Com95] TIS Committee. Executable and Linkable Format (ELF) Specification, 1995. Accessed: 2024-05-29.

[Dee23a] DeepComputing. DC-ROMA RISC-V LAPTOP II. https://deepcomputing.io/product/dc-roma-risc-v-laptop-ii/, 2023. Accessed: 2024-06-26.

[Dee23b] DeepComputing. World's First RISC-V Pad with LTE Launched by DeepComputing at RISC-V Summit 2023. https://riscv.org/blog/2023/12/worlds-first-risc-v-pad-with-lte-launched-by-deepcomputing-at-risc-v-summit-2023/, 2023. Accessed: 2024-06-26.

[DFFA18] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. rev. ng: A multi-architecture framework for reverse engineering and vulnerability discovery. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5. IEEE, 2018.

[Eng21] Alexis Friedrich Engelke. *Optimizing performance using dynamic code generation*. PhD thesis, Technische Universität München, 2021.

[Fin22] Martin Fink. Translating x86 Binaries to LLVM Intermediate Representation. Bachelor's thesis, Technische Universität München, 2022.

[FRT$^+$23] William Fornaciari, Federico Reghenzani, Federico Terraneo, Davide Baroffio, Cecilia Metra, Martin Omana, Josie E Rodriguez Condia, Matteo Sonza Reorda, Robert Birke, Iacopo Colonnelli, et al. Risc-v-based platforms for hpc: Analyzing non-functional properties for future hpc and big-data clusters. In *International Conference on Embedded Computer Systems*, pages 395–410. Springer, 2023.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[KMZ17] Jakub Křoustek, Peter Matula, and Petr Zemek. RetDec: An open-source machine-code decompiler. In *July 2018*, 2017.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[Lat02] Chris Arthur Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[Pro02] Mark Probst. Dynamic binary translation. In *UKUUG Linux Developer's Conference*, volume 2002, 2002.

[Pro24] LLVM Project. The LLVM Compiler Infrastructure. https://llvm.org/, 2024. Accessed: 2024-05-16.

[RRP+07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.

[SCHY12] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: an LLVM-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 51–60, 2012.

[WLP+14] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2:1–79, 2014.

[YS19] S. Bharadwaj Yadavalli and Aaron Smith. Raising Binaries to LLVM IR with MCTOLL (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, page 213–218, New York, NY, USA, 2019. Association for Computing Machinery.

A Example

This chapter illustrates the various phases of the RISC-V binary raiser using a program that calculates the factorial sequence recursively. Listing 19 illustrates the input source code written in C. The source is compiled to RISC-V without optimizations, which can be seen in Listing 20. The resulting ELF binary is used as input for MCTOLL, which results in the LLVM IR seen in Listing 21. Directly compiling the source code to LLVM IR results in Listing 22. Listing 23 represents the optimized and recompiled version of Listing 21.

```c
#include <stdio.h>

int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}

int main() {
    int f = factorial(5);
    printf("%d\n", f);
    return 0;
}
```

Listing 19: An example C program that will calculate the factorial of 5 recursively.

```
factorial:
    addi    sp,sp,-32
    sd      ra,24(sp)
    sd      s0,16(sp)
    addi    s0,sp,32
    mv      a5,a0
    sw      a5,-20(s0)
    lw      a5,-20(s0)
    sext.w  a5,a5
    bne     a5,zero,.L2
    li      a5,1
    j       .L3
.L2:
    lw      a5,-20(s0)
    addiw   a5,a5,-1
    sext.w  a5,a5
    mv      a0,a5
    call    factorial
    mv      a5,a0
    lw      a4,-20(s0)
    mulw    a5,a4,a5
    sext.w  a5,a5
.L3:
    mv      a0,a5
    ld      ra,24(sp)
    ld      s0,16(sp)
    addi    sp,sp,32
    jr      ra
```

```
main:
    addi    sp,sp,-32
    sd      ra,24(sp)
    sd      s0,16(sp)
    addi    s0,sp,32
    li      a0,5
    call    factorial
    mv      a5,a0
    sw      a5,-20(s0)
    lw      a5,-20(s0)
    mv      a1,a5
    lla     a0,.LC0
    call    printf@plt
    li      a5,0
    mv      a0,a5
    ld      ra,24(sp)
    ld      s0,16(sp)
    addi    sp,sp,32
    jr      ra
```

Listing 20: An example RISC-V program corresponding to Listing 19

```
@.rodata13 = private unnamed_addr constant [12 x i8] c"↩
   \01\00\02\00\00\00\00\00%d\0A\00", align 8

define dso_local i32 @factorial(i32 %0) {
  %2 = alloca i32, align 4
  store i32 %0, ptr %2, align 4
  %3 = load i32, ptr %2, align 4
  %4 = add i32 %3, 0
  %x15_stack_slot = alloca i32, align 4
  %5 = icmp ne i32 %4, 0
  br i1 %5, label %7, label %6

6:                                                ; preds = %1
  store i32 1, ptr %x15_stack_slot, align 4
  br label %15

7:                                                ; preds = %1
  %8 = load i32, ptr %2, align 4
  %9 = add i32 %8, -1
  %10 = add i32 %9, 0
  %11 = call i32 @factorial(i32 %10)
  %12 = load i32, ptr %2, align 4
  %13 = mul i32 %12, %11
  %14 = add i32 %13, 0
  store i32 %14, ptr %x15_stack_slot, align 4
  br label %15

15:                                               ; preds = %7, %6
  %16 = load i32, ptr %x15_stack_slot, align 4
  ret i32 %16
}

define dso_local i32 @main() {
  %1 = call i32 @factorial(i32 5)
  %2 = alloca i32, align 4
  store i32 %1, ptr %2, align 4
  %3 = load i32, ptr %2, align 4
  %4 = call i32 (ptr, ...) @printf(ptr getelementptr inbounds ([12 x i8↩
     ], ptr @.rodata13, i32 0, i32 8), i32 %3)
  ret i32 0
}

declare dso_local i32 @printf(ptr, ...)
```

Listing 21: An example LLVM IR program corresponding to Listing 20, raised using our RISC-V binary translator. Some information is left out for brevity.

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @factorial(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  %4 = load i32, ptr %3, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %1
  store i32 1, ptr %2, align 4
  br label %13

7:                                                ; preds = %1
  %8 = load i32, ptr %3, align 4
  %9 = load i32, ptr %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call i32 @factorial(i32 noundef %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, ptr %2, align 4
  br label %13

13:                                               ; preds = %7, %6
  %14 = load i32, ptr %2, align 4
  ret i32 %14
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %3 = call i32 @factorial(i32 noundef 5)
  store i32 %3, ptr %2, align 4
  %4 = load i32, ptr %2, align 4
  %5 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %4)
  ret i32 0
}

declare i32 @printf(ptr noundef, ...) #1
```

Listing 22: An example LLVM IR program corresponding to Listing 20, compiled using Clang. Some information is left out for brevity.

```
main:
    addi    sp,sp,-16
    sd      ra,8(sp)
    lui     a0,0x10
    addi    a0,a0,1472 # 105c0 <_IO_stdin_used+0x10>
    li      a1,120
    jal     ra,104e0 <printf@plt>
    li      a0,0
    ld      ra,8(sp)
    addi    sp,sp,16
    ret

factorial:
    mv      a1,a0
    sext.w  a0,a0
    beqz    a0,1058e <factorial+0x16>
    li      a0,1
    addiw   a2,a1,-1
    mulw    a0,a0,a1
    mv      a1,a2
    bnez    a2,10580 <factorial+0x8>
    ret
    li      a0,1
    ret
```

Listing 23: An example RISC-V program corresponding to Listing 21, optimized recompiled with Clang.

B Reproducibility

In this chapter, we describe how to setup an environment that is able to reproduce the results of the experiments described in Chapter 4. The source code for the RISC-V binary translator can be found in our fork of the MCTOLL repository[10], where the following information can also be found.

It is assumed that a Linux system is used. Both the RISC-V GNU Compiler Toolchain[11] and a RISC-V root file system[12] need to have been installed, and both the LLVM project[13] and MCTOLL tool[14] need to have been cloned. The MCTOLL repository documents the exact commit of the LLVM project that should be used for building MCTOLL. At the time of writing, this is commit `5c68a1cb123161b54b72ce90e7975d95a8eaf2a4`. For further details for installation, please refer to the README document of MCTOLL.

Once everything is installed and cloned, the bash scripts inside the *scripts* directory can be used to initialize the build files, build the projects, and run the experiments. Building the entire LLVM project can take a while, because of 3000+ objects needing to be build. It is *highly* recommended to use the LLVM linker `lld` instead of your system linker to *significantly* speed up the process. Please refer to the README document within the *scripts* directory for more information about the scripts.

[10] https://github.com/tdejong00/llvm-mctoll/tree/riscv64
[11] https://github.com/riscv-collab/riscv-gnu-toolchain
[12] https://toolchains.bootlin.com/downloads/releases/toolchains/riscv64-lp64d/tarballs/
[13] https://github.com/llvm/llvm-project/tree/5c68a1cb123161b54b72ce90e7975d95a8eaf2a4
[14] https://github.com/microsoft/llvm-mctoll