CHAIR OF COMPUTER ARCHITECTURE AND PARALLEL SYSTEMS

Großpraktikum Rechnerarchitektur

Dynamic Binary Translation for RISC–V code on x86–64

Summer term 2020

Noah Dormann Simon Kammermeier Johannes Pfannschmidt Florian Schmidt

Contents

1. Introduction **2**

2. Background **3**
2.1. Comparison of the RISC–V and x86–64 ISAs 3
2.2. Environment setup and memory layout 4
2.3. Partitioning the input code 5

3. Approach **5**
3.1. Translating the partitioned code 5
3.2. Code cache and block handling 6
3.3. Register handling and context switching 7
3.4. System call handling 8
3.5. Floating point extension 9
3.6. Optimisations 10

4. Implementation Details **12**
4.1. System architecture and execution control flow 12
4.2. Translation process 13
4.3. Static hybrid register mapping 14
4.4. Detailed system call overview 17

5. Results and Performance **19**
5.1. Verification 19
5.2. SPEC CPU 2017 Benchmark Suite 20
5.3. Evaluation of translator optimisations 23
5.4. Data compression via gzip 25
5.5. Discussion 26

6. Summary **27**

Appendices **29**
A. Download and installation instructions 29
B. Executable program requirements 29
C. Using the translator 29
D. Version history 30

1. Introduction

RISC–V is an open ISA first conceptualised in 2010 with the initial goals of research and education in mind. In contrast to the widespread x86–64 ISA [1] it employs the RISC (Reduced Instruction Set Computer) scheme by providing fewer and less powerful instructions, addressing modes and cycle-heavy features in favour of a simplified micro architecture. Its development takes the lessons learned in terms of backwards compatibility and future-proofing from other widespread ISAs like x86 into account, and aims to provide an open interface for the architecture, rather than strict implementation details. This grants a large freedom to the implementors and greatly increases the flexibility and ease of working with the architecture [2, S. 1f]. As such, it looks to be open to future extensions by already defining a basis for future 128-bit integer instructions and instruction length encodings of up to 176 bits (22 bytes). The possibility to expand further when widespread technology developments would require such an expansion is also remained open.

There is already some hardware available for RISC–V[1], but it is not yet widespread. This means developers usually do not have access to real hardware yet, so they must instead rely on emulation to test their code written for the RISC–V platform.

When attempting to execute programs compiled for a foreign architecture on a different native one, there are essentially three distinct approaches:

- **Interpretation**, where, much alike interpreted programming languages (such as JavaScript, Python, or Ruby), the assembly instructions located in the binary are examined while emulating the execution of the program, and equivalent actions are taken on the native system in order to simulate the guest ISA.

  While likely being the easiest to actually be implemented, this comes with a significant performance penalty mainly because every single assembly instruction will have to be interpreted for every execution of that program part, potentially causing a lot of redundant work.

- **Static Binary Translation**, where the executable is statically reverse-engineered and translated to the other architecture as a whole. After this translation step, it can be executed as if it were a native binary, without the need for any further special treatment. In theory you could reach near native speeds for the generated binary using this technique. There some hurdles with this though, one example is register indirect branches, which require some way to convert the foreign addresses to native at runtime. Any program that produces or edits assembly at runtime would also generally prove difficult to translate statically.

- **Dynamic Binary Translation** (DBT), which serves as a middle ground between interpreting and statically translating the executable. It aims to translate the program on the fly, while only focussing on the parts that are actually needed for

[1] The biggest name here is probably SIFive [3], which already produce multi-core CPUs with super scalar out-of-order pipelines reaching multi-GHz clock speeds.

> execution. Therefore, it can save some of the overhead of a static translator by not spending execution time on unused code paths. The other aforementioned issues are also fairly easily resolved. Unlike an interpreter, every instruction only has to be translated once and can then be run without any unnecessary overhead. Of course, this assumes that the translation routines are relatively swift in performing their functions, so as not to introduce any more overhead than necessary [4, S. 1f.].

One of the most popular software emulators in general is QEMU [5]. While QEMU is a portable DBT that supports a wide variety of architectural combinations and ISAs, this also makes it hard to optimise it for a specific guest/host combination and therefore the program execution will be slower than necessary.

Our aim is to provide a faster emulator, allowing the execution of RISC–V 64 bit code on an x86–64 machine by means of dynamic binary translation. We want to provide a tool to execute binaries compiled for RISC–V Linux on Linux for x86–64, what is conventionally called *user space emulation*, full system emulation is not supported. Currently only single threaded applications are supported, since managing the execution environment and guaranteeing atomicity for the respective instructions would be rather difficult.

The rest of this paper is structured as follows: Section 2 will define concepts and constraints relevant for our undertaking, after which section 3 will show our approach and describe the rationale for major design decisions taken during the implementation. Select parts of the translator will then further be elaborated on in section 4. Finally, sections 5 and 6 will evaluate the success and quality of our developed solutions by using standardised benchmark suites, as well as provide a short summary and future perspective of the project at hand.

2. Background

In the following, the term *host* will refer to the system of the native architecture the binary translator is built for (in our case, x86–64), and the term *guest* will designate the foreign system we are attempting to emulate (RISC–V).

2.1. Comparison of the RISC–V and x86–64 ISAs

By its very nature, executing code compiled for one architecture on a different one is not an easy task. It is obvious that there are major differences in the two architectures, brought forth by RISC–V being a reduced instruction set computer (RISC) architecture and x86–64 a complex instruction set computer (CISC) architecture.

The most relevant distinction between RISC–V and x86–64 for the development of our DBT is the different address format and mismatching numbers of general purpose and floating point registers. The ISA design of RISC–V with a load-store architecture and three-operand instruction format allows for a more flexible register usage but requires

more instructions due to the explicit load/store operations. x86–64 however is a register-memory architecture with a two-operand instruction format. Thus, memory accesses are reduced by implicit loads in instructions using a memory operand.

RISC assembler code includes pseudo-instructions that are translated into multiple instructions by the assembler, due to the nature of a reduced instruction set. One example is the absence of a `mov` instruction in the RISC–V ISA; the assembler translates the pseudo-instruction `mv` into an `addi rd, rs1, 0` instruction.

Similarly, the ISA offers no instruction to load a 64 bit immediate into a register, as the fixed-width 32 bit instruction encoding does not allow for it. Contrary to x86, where this problem would be solved by a single `mov` instruction, the RISC–V assembler has to build up the immediate in the register by using a specific combination of addition and shifting operations as well as program-counter-relative arithmetic.

In an ideal world, translating a RISC binary for execution on a CISC system would lead to a size reduction of the generated code. However, in practice, this is nearly impossible. Efficient fusion of multiple RISC–V instructions into fewer x86–64 instructions is a very difficult endeavor. When the DBT is only presented with partial block-wise assembly code snippets, some optimisations are impossible to undertake, as there are many unknowable parameters at play. Those challenges will be further elaborated on in section 3.

2.2. Environment setup and memory layout

As the DBT is responsible for managing the execution environment of the guest binary in the shared address space, it must also handle the setup of said environment.

The header of the ELF-file (*Executable and Linkable Format* [6]) specifies which section(s) of the program need to be loaded, and where in memory they must reside. The DBT must take care to map the file into memory correctly, while not compromising its own memory region.

Furthermore, the guest registers (see section 3.3) and stack must be initialised in accordance with the architecture specification and calling convention, which necessitates a specific layout of environment and auxiliary parameters as well as command line arguments to be present [4, S. 2].

The stack is set up exactly like it would have been by the linux kernel. As such, the stack pointer needs to point at the argument count, followed by (towards higher addresses) the zero terminated argument, environment and auxiliary vectors. Finally some alignment bytes need to be added, so the stack pointer is 16 byte aligned and ABI-compliant. All of the information can generally be copied from the host in our case.

The memory is laid out as follows: The translator is linked to a high address at `0x780000000000` so the range that typical programs use at the bottom of the address space is kept free and guest addresses do not need to be converted and can directly be used. The following region (sized slightly below the max size of a 32 bit signed integer) is reserved for the translated code, so the global variables of the translator can be reached with a 32 bit `RIP`-relative offset from our generated code. The lower limit for the guest stack is obtained by using the kernel stack limit and adding a guard page.

Address range	Usage
`0x780000000000+`	Translator address region
`0x77ff81000000+`	JIT generated code
`0x77ff807fe000+`	guest stack
`(last mapped address + 1)+`	guest heap
defined by ELF file	mapped guest binary

Table 1: The layout of the memory space.

2.3. Partitioning the input code

Logically, upon facing the task of translation, the DBT must somehow divide the code into chunks it can then process for translation and execution. The natural choice here is for the translator to partition the code into basic blocks.

Basic blocks, by definition, have only a single point of entry and exit; all other instructions in a single block are executed sequentially and in the order that they appear in the code. (Of course, this does not take into account mechanisms such as out-of-order execution or system calls as well as interrupt- and exception handling).

So, for our purposes, a basic block will be terminated by any control-flow altering instruction like a jump, call or return statement, or a system call[2].

3. Approach

The following section will elaborate on our approach to the problem at hand and lay out basic principles related to the process of dynamic binary translation.

3.1. Translating the partitioned code

Generating equivalent code

The most basic idea for translating the now partitioned basic blocks is to have a fixed association that maps every instruction in the guest ISA to a sequence of instructions native to the host.

The quality of the code, that can be generated here, strongly depends on the properties of the host and guest architectures in question. Difficulties can arise due to differences in the instruction operand formats and the types of instruction set architectures the DBT is dealing with.

In our case, as outlined in section 2.1, challenges stem from the fact that we are translating code from a load-store architecture using a three-operand instruction format into a register-memory architecture in which (generally) one of the source operands is also the implicit destination operand. This, for example, means that the single arithmetic

[2]These may or may not have control-flow altering effects; they in any case need to be handled this way due to the reasons laid out in section 3.4.

instruction `sub rd, rs1, rs2` in RISC–V assembly language generally can not be translated via a single instruction, but rather requires two instructions: moving `rs1` to `rd`, then subtracting the value of `rs2` from `rd`.

Opportunities for optimisation lie wherever there is a way to shorten the execution time of the translation, possibly by employing semantically equivalent native instructions that run in a shorter timespan. The RISC–V pseudo-instructions (as mentioned in section 2.1) are also of some help here [2, S. 139], along with discoverable patterns in the input assembly. It is clear, for example, that an instruction like `xori x10, x10, -1` can be directly translated as a `not x10`, without needing to resort to `mov` and `xor`. The same principle applies to combinations of multiple instructions. An `lui rd, imm1` followed by `addi rd, rd, imm2` may for example be translated as directly loading the result of the computation `imm1 + imm2` into `rd`.

Translation procedure

The central element of the translator is the transcode loop, which controls both translation and execution: first, it is checked whether the block to be executed next has already been translated (by doing a cache lookup, see 3.2), and if it has not, translation is started. The translated block is then executed, and, before returning to the transcode loop, sets the address where program execution should continue.
Translation itself is done in two steps:

1. Parsing code of the guest program into easy to work with internal instruction representations, until we hit the end of the basic block.

2. Translation is then done by dispatching the parsed instructions to special single-instruction translator functions, which emit the matching x86-64 code into memory allocated for the block.

When translation of the block is complete, a `ret` instruction is appended for returning to the translator after execution.

3.2. Code cache and block handling

Naturally, the DBT aims to store the translated code in a semi-permanent way, as it is the goal to not have to translate a required section more than once.

For this, we allocate a region of memory reserved for the basic block translations, also called a *code cache*. Additionally, an index to this memory section is required, since there needs to be a way to quickly reference the blocks residing in the cache and associate them with both the host and guest instruction pointers that identify them during execution.

The efficiency of the index lookup is increased by a *translation lookaside buffer* (TLB). This technique, also used by the CPU to optimise memory page lookup, reduces the lookup time for the most recently used code blocks. Still, it is desirable to avoid cache lookups when possible, as they are relatively costly and can cause significant slowdown to the whole system. Repeatedly retrieving a higher number of blocks than the capacity

of the TLB will also decrease the performance, because entries will be overwritten before they are read again. Several of the optimisations we employ thus aim to reduce the number of cache lookups by avoiding having to return to the transcode loop.

It is possible that this code cache might fill up during the execution of a large guest program. If it does, there are two different strategies to handle this issue: One can either invalidate and purge some or all of the blocks currently residing in the cache, or dynamically resize the cache according to the needs of the guest program [4, S. 3].

Purging the entire cache would require the translator to restart translation on older blocks that might be needed again, introducing a performance overhead that needs to be weighed against the higher memory usage of enlarging the cache.

On the other hand, selective deletion of some of the blocks in the cache is very difficult due to optimisations taken in the context of chaining. As any chained jumps located in another cached block are dependent on the target block residing in the cache, removal of the target would invalidate these jumps. It would thus only be possible to either remove all blocks with jump references to the candidate up for removal, or to leave all blocks with jump references in the cache altogether.

3.3. Register handling and context switching

As outlined in section 2.1, the RISC–V and x86–64 architectures have differing amounts of general purpose registers. In some way, the state of the 32 general purpose registers `x1`[3] to `x31` and the `pc` needs to be stored and available to the translations of the identified basic blocks.

3.3.1. Handling of guest registers

As x86–64 only provides 16 general-purpose registers (`rax`–`rdx`, `rsp`, `rbp`, `rsi`, `rdi` and `r8`–`r15`), it is impossible to directly and statically map all guest registers to native host registers. Adding to the above, due to the fact that some x86–64 registers have special or implicit purposes in some instructions like `(i)mul` or `(i)div`, care must be taken in choosing the registers that can be used for such a mapping. Keeping a guest register file exclusively in memory, and loading them into native registers when needed within the translations of single instructions is technically possible, especially in light of the ability to extensively use memory operands in the instructions on x86–64. However, this necessitates a large number of memory accesses for both memory operands in the instructions as well as local register allocation within the translated blocks. Due to the very large performance gain connected to using register operands instead of memory operands, this is also not feasible at scale [4, S. 8f.].

Accordingly, the solution for this problem would be an approach that employs parts of both of these extremes [4, S. 9]. We utilise the tools we designed to discover the most-used registers in the guest programs, and statically map these to general purpose x86–64 registers. The remaining operands are then dynamically allocated into reserved

[3] `x0` is hardwired to a constant zero. All reads will return 0, all writes will be ignored. Hence, this register needs special handling in the DBT, as there is no equivalent construct on x86–64.

host registers inside the translation of a single block. The loaded values are then lazily kept in the temporary registers for as long as possible in order to avoid unnecessary memory accesses. In case the translator requires a value not currently present in a replacement register, the oldest value is written back to the register file in memory and the now free space is utilised for the requested value. The final write-backs then need to be performed on the block boundaries.

The most-used registers are relatively invariant in between RISC–V executables and their basic blocks, however it might be the case that a single block in such an executable requires a few unique registers fairly often. By dynamically allocating these into temporaries and statically mapping the most-used registers in general, we save much overhead otherwise spent on memory access to the register file, but do not unnecessarily occupy native register space with seldom accessed guest registers.

3.3.2. Context switching during execution

When the code translated by the DBT is executed, it will behave as if it were an independent x86–64 executable. With the static register mapping in place, these values will thus need to be loaded before any of the translated blocks are called, and stored back before the execution is returned to the DBT.

This is called a *context switch*, as we are switching from the host's program state made up of the current register values to that of the guest. Evidently, preserving both the host and guest state during execution is critical for the correct program behaviour.

3.4. System call handling

System calls are also a very important part of enabling the guest program execution. Thus, every ISA must offer some way to switch the execution context in the kernel mode for the system call to be handled.

For RISC–V, the instruction `ECALL` (for *environment call*, formerly `SCALL`) handles these requests, with the system call number residing in register `a7` and the arguments being passed in `a0` – `a6`.

However, the DBT generally cannot just reorganise the guest argument values and system call identifier according to the calling convention of the host and relay the system call directly. The RISC–V guest program expects a different operating system kernel than is present natively on the host; with that, the system call interface also differs [4, S. 2f.].

In order to handle the `ECALL` instruction correctly, the translator must thus build the translated instruction to call a specific handler routine not too dissimilar from one that may be found in a kernel. There, system calls that exist natively on the host architecture as well (like `write` or `clock_gettime`) can usually be passed along to the host kernel directly.

Care must be taken for system calls that would enable the guest to change the state or context of the host – an `mmap` into the memory region of the translator, for example, or a call to `exit` – these calls must be emulated accordingly to prevent these faults. In cases where the data structure layout used by the kernels differs, the DBT must also perform

necessary actions to adapt the formats to each other. Some system calls may not exist at all on the native architecture of the host, it is up to the DBT to emulate the required functionality [4, S. 2f.].

3.5. Floating point extension

Floating point support is a vital part of modern processors, enabling fast computation of real world problems like physics simulations. While a standard C compiler like *GCC* [7] is able to emulate floating point arithmetic using integer arithmetic, using the native support of the x86–64 SSE extensions is evidently a lot faster.

The main difficulties (and their resolutions) that arise by using the x86–64 SSE extensions to translate the RISC–V F- and D-extensions are listed below:

- **Register handling** is similar to the integer register management laid out in section 3.3. As mentioned before, the RISC–V architecture consists of 32 floating point registers (`f0`–`f31`) which can hold a single precision (F-extension) or double precision (D-extension) floating point value, whereas the SSE-extensions only provide 16 registers `XMM0`–`XMM15`. We utilise the tools we designed to discover the most-used registers in the guest programs, and statically map these to x86–64 SSE registers `XMM2`–`XMM15`. One could use the same dynamic mapping approach for the remaining registers as is being used for general purpose registers, but for simplicity reasons registers `XMM0` and `XMM1` are reserved for use as replacements and missing registers are loaded into them from memory temporarily.

- **Missing equivalent SSE instructions** can lead to a huge instruction overhead, as emulation often needs to use bit manipulation operations instead. For this constants or masks need to be loaded in from either memory or the general purpose registers because the SSE extensions do not support immediate operands. The instructions that need to be emulated are unsigned conversion instructions e.g. `FCVT.WU.S`, sign-injection instructions e.g. `FSGNJ.S`, compare instructions e.g. `FEQ.S`, fused multiply-add instructions e.g. `FMADD.S` and the `FCLASS.S` instruction that classifies a floating point value. These instructions are not supported by the SSE extension natively. One could use the FMA–extension [1, S. 141] to implement the fused multiply-add instructions natively, but these instructions require AVX, which is not generally available on x86–64 hardware. As an implementation reference for these instructions, the assembly generated by GCC was used.

- **Rounding modes** are handled differently in the RISC–V architecture, as the rounding mode can be set individually for every instruction. The rounding mode of the SSE extension however is controlled by the state of the `MXCSR` control and status register. Thus in case a instruction with explicit rounding mode is encountered, the rounding mode is temporarily changed in the `MXCSR` register.

- **Exception handling** in RISC–V is realized by reading the `fcsr` floating-point control and status register, traps are not supported. The CSR instructions used

to read this register are thus emulated to instead read and translate the `MXCSR` exception flags. x86–64 exceptions are meanwhile disabled by masking them in the `MXCSR` register.

3.6. Optimisations

The following four sections aim to provide an overview on the optimizations we employ to mitigate the performance penalties imposed by architecture differences and the concept of dynamic binary translation. The first three methods optimize control flow to avoid leaving guest context, while the fourth increases execution efficiency of the generated arithmetic instructions by means of macro opcode fusion.

3.6.1. Recursive jump translation

Returning from translated guest code to the main transcode loop comes with big performance penalties. These are imposed by the first context switch, code cache lookup, and second context switch necessary for starting execution of the next basic block. To avoid these negative performance impacts, we employ, among others, the method of recursive translation: When the parser arrives at an unconditional jump, translation of the jump target is started recursively. That way, the jump target will always be translated before the jump itself. Because the host code address of the jump target is now already known at translate time, we can make the emitted host code jump to the target directly instead of returning to the transcode loop. The blocks containing jump and target are thereby chained.

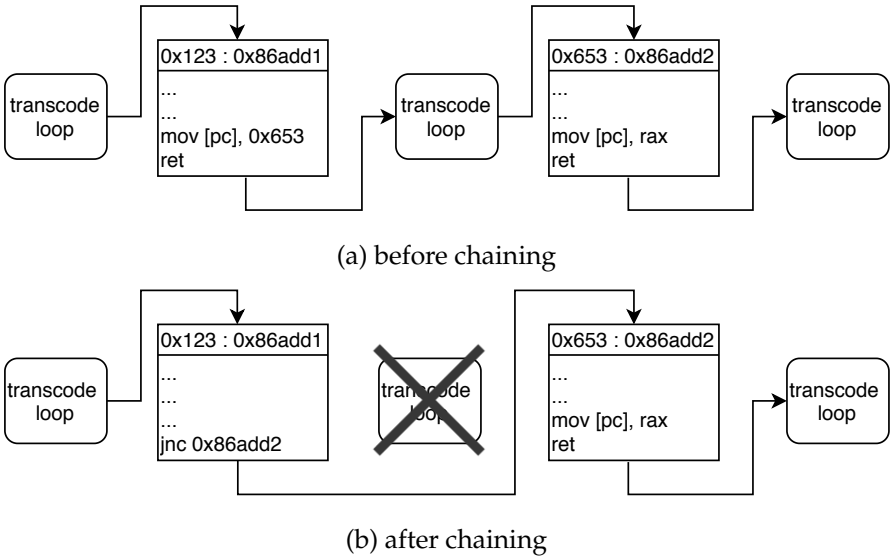
3.6.2. Retroactive block chaining

Translation in general is done lazily. This especially applies to the handling of conditional jumps, as it is nearly impossible to know at parsing time whether a branch will actually be taken or not. In cases where the translated conditional jump is not taken during execution, translating the target would cause unnecessary overhead. For that reason, we do not recursively translate conditional jumps.

If a branch is reached during translation, its two sides can only be chained if the respective target blocks have already been translated and their host code addresses are thus known. If that is not the case, the targets can instead be chained retroactively after the respective sides of the branch are first taken: following translation of the target block, the host code of the branch is modified to jump directly to the target address, which is now known, instead of returning to the transcode loop. Further cache lookups and context switching are thus avoided for this side of the branch.

3.6.3. Return address stack

Dynamic jumps can not be statically chained, because their target address may vary. Still, there is potential for optimisation, as the majority of dynamic jumps found in a typical



(a) before chaining

(b) after chaining

Figure 1: An illustration of retroactive block chaining.

program are function returns. By keeping track of function calls, it is possible to predict the return addresses. In order to do so, we use a stack that holds entries consisting of pairs containing both the guest return address and its corresponding host code address. The stack is implemented as a ring buffer to prevent over- and underflow.

Calls are first detected at parsing time and have their return targets recursively translated. After that, host code will be emitted that pushes the corresponding return address pair onto the stack at every execution without having to leave guest context.

Following dynamic jumps will compare their target guest-address with the entry in the top stack element. If the values match, the address pair is popped, and the target block is jumped to directly. Thus, returning to the transcode loop is avoided. If the values do not match, control is handed back to the host. In this case the stack will not be changed, as it is assumed that a return corresponding to the top stack element will still follow.

3.6.4. Macro operation fusion by pattern matching

As RISC–V is a RISC and x86–64 a CISC architecture, programs often need more instructions on RISC–V than on x86–64 to achieve the same effect. We employ a technique known as macro operation fusion to translate specific patterns of RISC–V instructions into shorter, equivalent x86–64 code, thereby increasing performance of the generated code. The pattern matcher will detect these patterns after parsing, and replace them with special pseudo-instructions, whose translator functions then emit the corresponding sequence of x86–64 instructions at translate time.

Care must be taken when defining the patterns, as instructions in the pattern might

RISC-V	**x86-64**
`LUI r1, imm; LW r2, imm(r1); ADDI r2, r2, imm; SW r2, imm(r1);`	`ADD m32, imm32`
`LUI r1, imm; LD r2, imm(r1); ADDI r2, r2, imm; SD r2, imm(r1);`	`ADD m64, imm32`
`AUIPC r1, imm; ADDI r1, r1;`	`MOV r64, imm64`
`AUIPC r1, imm; LW r1, imm(r1);`	`MOVSX r64, m32`
`AUIPC r1, imm; LD r1, imm(r1);`	`MOV r64, imm64`
`SLLI r1, r1, 32; SLRI r1, r1, 32;`	`MOV r32, r32`
`ADDIW r1, r1, imm; SLLI r1, r1, 31; SRLI r1, r1 32;`	`ADD r32, imm32`

Table 2: Some examples of patterns currently fused by the translator, the last five taken from the RISC–V emulator project rv8 [8].

set other register values as a side-effect. These side-effects must be preserved when replacing the instruction sequences, as later instructions might or might not rely on these values being set correctly. Thus, either the emitted x86–64 code has to set these registers as well, or the pattern has to only be applicable if no such writes into later used registers occur, thereby constraining the register usage in the instruction combination. An exemplary excerpt from the patterns we implemented is shown by table 2.

4. Implementation Details

The following section aims to provide an in-depth overview of select components central to the translator's functionality and describe the reasoning behind them.

4.1. System architecture and execution control flow

Initialisation

After the command line arguments are parsed, the guest executable is mapped into memory, as specified by the ELF header. The next step then is to initialise the guest environment. This includes setting up the guest environment as specified in section 2.2.

Some of the optimisation systems also have to initialised, namely the code cache, the return address stack, the register file, and the context switcher.

Transcode Loop

To begin guest program translation and execution, the transcode loop is started with the address next to be executed set to the entry point of the guest program.

After entering the loop (see figure 2), it is first checked if the block next to be executed has already been stored in the code cache.

If this is not the case, translation will start by parsing the blocks instructions, after which the pattern matcher detects and replaces optimisable patterns. The parsed instructions are then translated, and the block is stored in the code cache.

Next it is checked if the last executed block ended in a jump that can be statically chained. If yes, the chainer is called to link the new block to the last one.

The last step is to switch into guest context and start executing the block. Due to block chaining and the return address stack, multiple blocks might be executed before switching back to host context, setting the address next to be executed, and returning to the transcode loop.

The loop will break if the guest program terminated, after which the translator will terminate as well, returning the guest exit code.

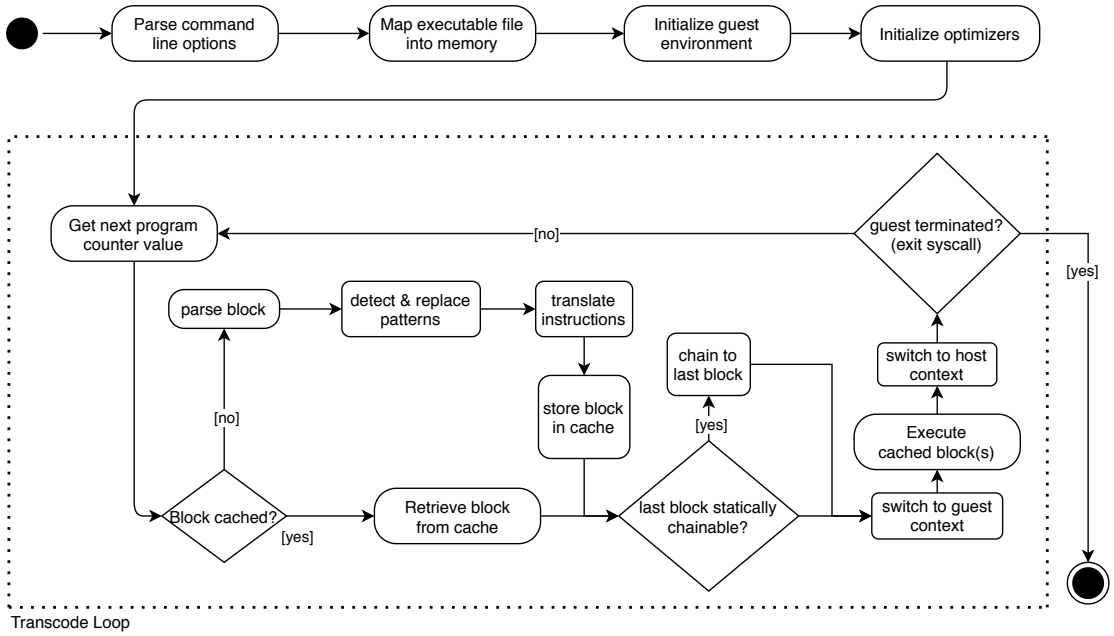


Figure 2: Control flow overview of the translation and execution process.

4.2. Translation process

Instruction parsing

The first step to translating a basic block is to parse its instructions. The decoding of the RISC–V assembly is relatively straight-forward, as we have a fixed instruction length of 32 bits. Thus, the assembly is parsed in blocks of 4 bytes with the information being extracted in an intermediate instruction format holding the mnemonic, operands and immediates in uncompressed form. Parsing continues until the first possibly control flow altering and thus block ending instruction. If this last instruction is a call, the called block as well as the next block to be executed after the return are translated recursively and stored in the code cache.

The pattern matcher (see section 3.6.4) now iterates over the parsed instructions and replaces detected patterns.

Instruction translation

The instructions in their uncompressed format are then dispatched to special per-mnemonic translation functions, which generate and emit the x86–64 byte code using faenc [9]. Because the block ended on a possibly control flow altering instruction, it is this last instruction which has to ensure execution is continued correctly.

If it is known at translation time what block will have to be executed next, and if this block has already been translated, then the last instruction can be made to directly jump to the next block. Dynamic jumps will check the return stack and also jump directly if possible, as described in section 3.6.3. If the next block is known but not yet translated, the last instruction will instead signal its ability to be chained to the translator before returning to the transcode loop.

Should none of two mentioned methods avoiding returning to the transcode loop be applicable, the last instruction will be made to write the address next to be executed to the program counter, which is then read by the transcode loop to initiate further execution.

4.3. Static hybrid register mapping

In order to achieve the best performance with the hybrid approach to the register mapping described in section 3.3 on page 7, we must decide how to make best use of the limited number of host GPRs we have available.

Principally, we chose to statically map into as many registers as we could, without compromising on flexibility in handling instructions that require specific registers.

4.3.1. Register priority analysis

There are two main ways of determining the priority of registers when considering them as candidates for a mapping.

It is, on the one hand, possible to assess the priority statically, by performing an analysis of the binary in question. Essentially, the hereby produced metric counts the number of times the register is used in the assembly instructions listed in the guest program and thus delivers an idea of how important each register is to this specific executable. We have built the tools required for this effort directly into the translator's analyser function, accessed via the `-a` flag.

However, this approach does not take into account that a single instruction may be executed many times while the program is running. Accordingly, the other approach is to assess the register priority dynamically by analysing and profiling the execution of the testing program, thereby gaining an insight into how often each register is actually used during the execution. The translator is also capable of performing such an analysis, commanded by the `-p` flag. A dynamic analysis, of course, delivers a largely more accurate idea of the priority of the registers in question, but has the decided and obvious disadvantage that it cannot be performed without actually executing the binary.

For the average results of such an analysis performed on a range of programs, including *gzip* [10] and several benchmarks of the `intspeed`-Suite of *SPEC CPU 2017* [11], see

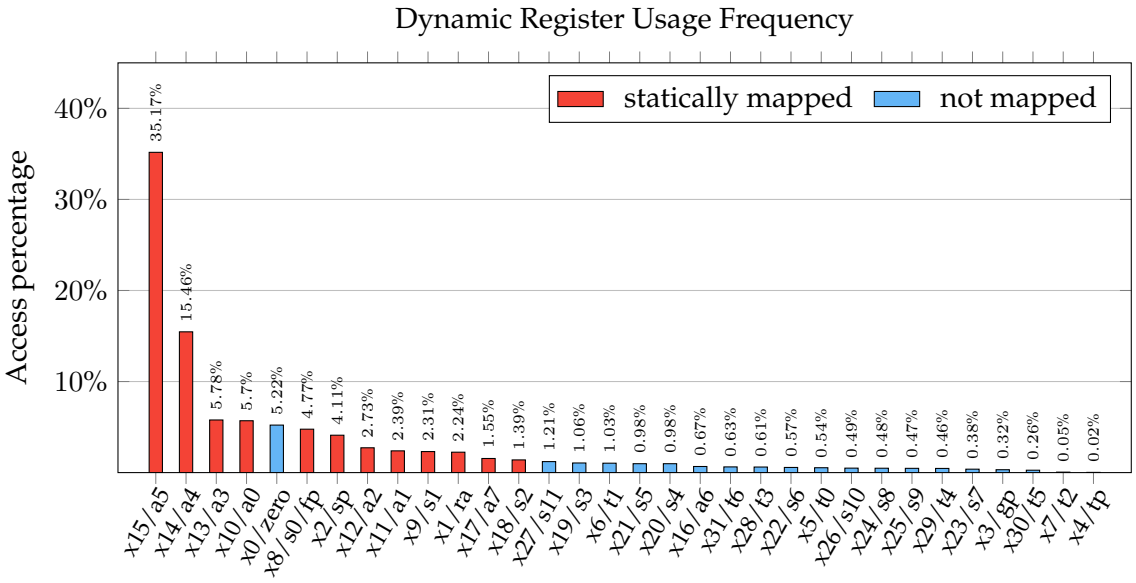


Figure 3: The average results of a dynamic register usage analysis, ordered by frequency.

figure 3.

Primarily, we gain interesting insights into the differences between the static and dynamic results yielded by the analysis. While the static ranked hit list does not differ greatly between the different executables and the top 12 entries are identical for every one of the tested programs, the dynamic results are far more variable. This makes creating a register mapping that fits well to every executable very difficult.

The benchmarks `605.mcf` (route planning workload) and `620.omnetpp` (discrete event simulation for computer networking) [12] of the *SPEC CPU* suite can serve as examples here. For programs like `605.mcf` that only lightly use the stack, holding the stack pointer `sp/x2` in a native register when only $1{,}20\,\%$ of accesses actually utilise it would not be necessary. However, other programs like `620.omnetpp` may rely heavily on the stack, and thus log very frequent accesses to `sp/x2`; when statistically every ninth access is to the stack pointer, it is absolutely essential to map the register to a native GPR.

If a static analysis yielded results of similar quality to the dynamic counterpart, the DBT could analyse the binary prior to execution and run every program with a best-fit static register mapping. However, evidently, this is impossible with dynamic profiling.

4.3.2. Structure of the mapping

From the 16 general-purpose registers x86–64 has to offer, we may use the 12 registers `rbx`, `rbp`, `rsi`, `rdi` and `r8`–`r15`. The remaining registers have either implicit or exclusive functions in some instructions (`rax` and `rdx` for multiplication/division, `cl` for shifting), or, like `rsp`, are impractical to use in combination with block chaining and function calls.

In keeping with our goal to statically map as many registers as possible, we chose to

RISC–V register	a5	a4	a3	a0	fp	sp	a2	a1	s1	ra	a7	s2
x86–64 mapping	rbx	rbp	rsi	rdi	r8	r9	r10	r11	r12	r13	r14	r15

Table 3: The static register mapping in use by the translator, ordered by the RISC–V register usage frequency (descending).

use all 12 of these registers. When we structure our mapping into the chosen slots by the average case of the insights gained (see table 3), we statistically capture about $83{,}59\,\%$ of register accesses, initially leaving the remaining $16{,}41\,\%$ to read from the register file in memory. The following section will touch on our approach to optimise the accesses to the remaining not-statically-mapped registers.

4.3.3. Dynamically allocated replacement registers

In order to implement the desired temporary register replacement behaviour detailed in section 3.3, we need to keep track of the age of the values currently situated in the replacement registers. With that, we can implement a *least recently used* register replacement policy. To do this, we track a metric of *replacement recency* (essentially serving as an inverse value age) for each block during translate-time. This recency gets incremented for every access not captured by the static register mapping.

An access to a register that is not statically mapped then first checks the contents of all temporary registers – if the value is already present, the DBT may use that register. If it is not already present, the DBT selects any free replacement register if able, and otherwise selects the register with the oldest value (or minimal recency) for write-back. The value is then loaded into the selected register from the register file in memory and marked as being the youngest in order to prevent it from being discarded in following mapping calls.

The dynamic mapping must also be able to receive requests for specific target registers for a load in order to support shifting and multiplication/division instructions that require arguments in implicitly defined registers. In order to support this efficiently, the DBT is able to shuffle the values in the replacement registers around accordingly.

With this behaviour, we manage to greatly save on memory accesses to the register file compared to simply loading and storing the values on an instruction-by-instruction basis. This style of register mapping can also not perform worse than accessing memory for each instruction, as in the worst case the DBT will perform the same memory accesses as with the other strategy, just possibly at a different time.

4.3.4. Discussion of other solutions

Apart from the above-mentioned solution, two more approaches are available in order to optimise register usage within basic blocks:

- **Allocating more registers dynamically**, and thereby reducing the amount of statically-mapped registers could theoretically improve the performance in some

```
1
2
3
4
5  ; add x6, x6, x7
6  mov rax, [gp_file + 8 * 6]
7  mov rdx, [gp_file + 8 * 7]
8  add rax, rdx
9  mov [gp_file + 8 * 6], rax
10
11 ; slli x6, x6, 3
12 mov rax, [gp_file + 8 * 6]
13 shl rax, 3
14 mov [gp_file + 8 * 6], rax
15
16 ; xori x7, x7, -1
17 mov rax, [gp_file + 8 * 7]
18 xor rax, -1
19 mov [gp_file + 8 * 7], rax
20
21
22
23
```

(a) Without dynamic register mapping

```
1  ; load registers
2  mov rdx, [gp_file + 8 * 6]
3  mov rcx, [gp_file + 8 * 7]
4
5  ; add x6, x6, x7
6
7
8  add rdx, rcx
9
10
11 ; slli x6, x6, 3
12
13 shl rdx, 3
14
15
16 ; xori x7, x7, -1
17
18 xor rcx, -1
19
20
21 ; write back registers
22 mov [gp_file + 8 * 6], rdx
23 mov [gp_file + 8 * 7], rcx
```

(b) With dynamic register mapping

Figure 4: A comparison of the code generated for example assembly by the DBT prior to and after introducing dynamic register allocation. The translated instructions are listed as a comment.

workloads. However, we dismissed this approach due to the fact that the most frequently used registers in the analysed workloads were largely invariant. Hence, these values are then preserved context-wide and do not need to be written back and restored on block boundaries.

- **Not mapping statically at all**, resorting to a dynamic allocation unique to every block. This optimisation opportunity is left as future work.

4.4. Detailed system call overview

As described in section 3.4 on page 8, we must assume the role of the kernel by handling system calls during the execution of the guest program. We achieve this by translating the `ECALL` instruction as a context switch and jump to the `emulate_ecall` routine in the DBT, which can then take the appropriate action.

As we stored the guest's registers before jumping to the handler, the requested system call index is now available to the DBT in the register file as entry `a7`, as per the RISC–V standard calling convention. We may now handle the system calls based on that index and the arguments passed in the registers `a0` through `a6`, and write the return value to

entry `a0` of the register file prior to switching the context back to the guest.

As previously mentioned, some system calls require special handling when encountered by the DBT (see table 4 on the next page for details). The following will describe the specifics of these issues with system calls that are either not present on the x86–64 host architecture, or may influence or break the state of the DBT.

Adapting structure data format. There are system calls like `fstat` and `fstatat` that exist both on RISC–V as well as x86–64, but use different data structure layouts in their return values. Thus, the DBT must adapt the data returned by the host to the required format prior to passing it back to the guest.

Emulation required. The DBT captures the `exit` and `exit_group` calls. Passing them through would immediately terminate the DBT – an action that is undesirable as it prevents any form of clean-up or post-execution profiling and analysis to take place. Thus, the DBT uses these system calls to set a flag which stops the main loop from executing the next iteration.

The `brk` system call must also be entirely emulated, as it would otherwise allow the guest program to modify the data segment (*program break*) belonging to the DBT, thus potentially deallocating some of the critically required memory.

Ignoring system calls. The `rt_sigaction` system call is ignored by the DBT. Due to the fact that the DBT and guest binary are running within the confines of the same process, any signal handler installed by the guest binary through `rt_sigaction` would also capture the respective signal sent to the translator. As it is impossible to distinguish the DBT from the guest program in inter-process communication, we must ignore this call in order to avoid undefined behaviour on signal handling.

Guarded pass-through to host. Essentially, any system call that has the possibility to influence the state or memory of the translator needs to have respective safe-guards in place. A good example of this behaviour is the `mmap` system call, the handling of which also reflects the memory layout scheme discussed in section 2.2 on page 4.

In any case, we must prevent a memory mapping into a memory region reserved by the translator. Mappings that do not interfere with the memory of the DBT can be passed along to the host directly. In case a hinted mapping would conflict with the memory of the translator, we may just re-hint the mapping to the top of the guest address space. When the call is not hinted (the `MAP_FIXED` or `MAP_FIXED_NOREPLACE` flag commands the mapping at exactly the specified address), we are unable to provide the guest with the requested mapping; thus we simulate an existing mapping in the location in question by returning `EEXIST` for `MAP_FIXED_NOREPLACE` and failing the call with `EINVAL` for `MAP_FIXED`.

Similarly, we fail a guest `munmap` with `EINVAL` in cases where the memory region of the translator would be compromised by the deallocation.

The other supported system calls may be directly passed through to the host after performing the necessary index mapping. With this strategy, we are able to support the following system calls:

System Call (index)	Handling	x86–64 base (index)
`fstatat` (79)	data reformat	`newfstatat` (262)
`fstat` (80)	data reformat	`fstat` (5)
`exit` (93)	emulate	n/a
`exit_group` (94)	emulate	n/a
`rt_sigaction` (134)	ignore	n/a
`brk` (214)	emulate	n/a
`munmap` (215)	guarded pass-through	`munmap` (11)
`mmap` (222)	guarded pass-through	`mmap` (9)

Table 4: An overview of the system calls we support that require special handling by the binary translator.

- `getcwd`
- `fcntl`
- `ioctl`
- `unlinkat`
- `ftruncate`
- `faccessat`
- `chdir`
- `fchmod`
- `fchown`
- `pipe2`
- `openat`
- `close`
- `getdents64`
- `lseek`
- `read`
- `write`
- `writev`
- `readlinkat`
- `utimensat`
- `set_tid_address`
- `futex`
- `set_robust_list`
- `clock_gettime`
- `tgkill`
- `rt_sigprocmask`
- `uname`
- `gettimeofday`
- `getpid`
- `getuid`
- `geteuid`
- `getgid`
- `getegid`
- `gettid`
- `sysinfo`
- `execve`
- `wait4`
- `prlimit64`
- `renameat2`
- `getrandom`

5. Results and Performance

The following section will give an overview of the methods used to test the programs correctness and performance, as well as lay out the results of these tests.

5.1. Verification

The correctness of our translator is checked by extensive unit tests, which are used to check if the translations of a the RISC–V instruction perform the operations that are expected as per the architecture specification.

Using the parameterised tests provided by the Google Test [13] framework, we are able to test a variety of different input values and combinations of statically mapped or not mapped register operands. Different combinations of using the same register and zero register `x0` for in- and output are also tested. Apart from this, correctness is also confirmed by being able to successfully run the *SPEC CPU 2017* benchmark suite as described in section 5.2 below.

SPECspeed Benchmark	Workload
`600.perlbench`	Perl interpreter
`602.gcc`	GNU C compiler
`605.mcf`	Route planning
`620.omnetpp`	Discrete Event simulation – computer network
`623.xalancbmk`	XML to HTML conversion via XSLT
`625.x264`	Video compression
`631.deepsjeng`	Artificial Intelligence: alpha-beta tree search (Chess)
`641.leela`	Artificial Intelligence: Monte Carlo tree search (Go)
`648.exchange2`	Artificial Intelligence: recursive solution generator (Sudoku)
`657.xz`	General data compression

Table 5: A description of the workloads covered by the *SPEC CPU 2017* `intspeed` suite [12].

5.2. SPEC CPU 2017 Benchmark Suite

Measuring the performance of the DBT was accomplished by using the tools in *SPEC CPU 2017* `intspeed` suite of benchmarks. This not only generates reproducible and widely accepted results in the industry, it also validates the results produced during the run, thus ruling out any errors in the translations of the benchmarks.

The `intspeed` suite also presents a variety of different workloads to the translator that are based on real-life scenarios, thus producing an accurate and understandable overview of the performance in a non-controlled environment. An overview of the workloads covered by the aforementioned suite can be found in table 5. Further context is provided by performance testing using the data compression utility *gzip* [10], where compression time is compared between runs on a native machine, in QEMU and in the DBT.

All testing was performed on an x86–64 8-core *Intel Xeon Bronze 3106* system clocked at 1,70 GHz base with 78 GiB of physical memory, running *Ubuntu 18.04.3 LTS*, kernel version *4.15.0-70-generic*. The DBT was compiled via `CMAKE_BUILD_TYPE` set to `Release` and `CMAKE_INTERPROCEDURAL_OPTIMIZATION` enabled, which implies `-O3` and `-flto -fno-fat-lto-objects`.

The benchmarks were compiled using compiler optimisation level `-O3` and linked statically. For the native run, `-march=x86-64` was used on GCC version 7.5.0 from the default Ubuntu package repository. The RISC–V binaries for our translator and QEMU were compiled using `-march=rv64ima` and `-mabi=lp64`. For this, a self-compiled GCC with the sources taken from the official toolchain repository at version 10.1.0 was used. This made it necessary to also specify `-fcommon -fallow-argument-mismatch` to stay fully compatible.

5.2.1. Results

Figure 5 shows normalized performance results of the *SPEC CPU 2017* `intspeed` benchmarks, effectively showing how much overhead QEMU and our translator caused versus the same benchmark compiled and run natively. Some of the overhead must of course be attributed to the architectural differences between x86 and RISC–V resulting in needing more instructions in RISC-V assembly than x86. This means these results do not directly measure the overhead vs. native that the whole translator infrastructure (parsing, translation, code cache etc.) causes. What we can compare though, is the relative results of QEMU and our translator, since both use the same compiler and thus get the same binary. This means the results are a measure for the relative efficiency of the infrastructure and the quality of the generated code.

Through the various performance optimisations mentioned in section 3.6, we are able to reach our goal of consistently outperforming QEMU. In some cases the advantage is only slight, but in other workloads like the `602.gcc` compiler benchmark the advantage grows to a comfortable 80 %.

Most benchmarks show runtimes of about 1.9x native with `600.perlbench` and `625.x264` being the outliers.

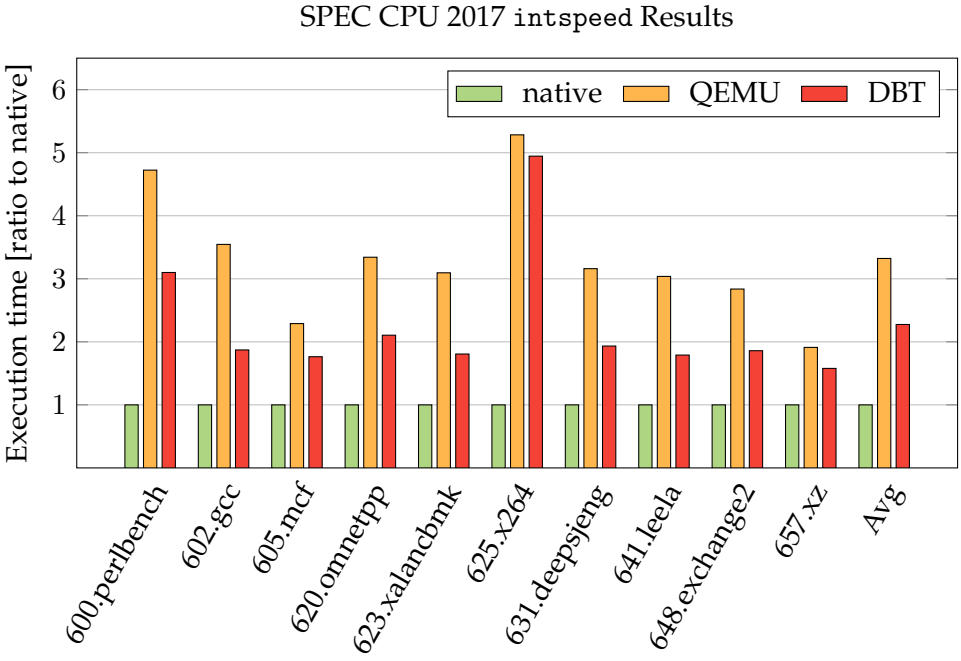


Figure 5: Results of `ref`-workload runs of *SPEC CPU 2017* `intspeed` (normalised, lower is better).

5.2.2. Analysis

`625.x264` on x86 heavily takes advantage of vectorisation, which RISC-V does not yet support, meaning that the compiler will have to generate loops that run more often, thus needing significantly more instructions for the same result. Retrospective vectorisation on the translator side is not easy since this would involve detecting the vectorisable loop patterns on assembler level, a task that even the compiler often isn't very effective at, even though it has the knowledge of the entire program. There are also instances where it relies heavily on 32 bit integer arithmetic, which in RISC-V always causes the results to be sign extended to the 64 bit register width, in comparison to x86 which zero extends in these cases. Thus many consecutive 32 bit instructions on the same values cause a lot of redundant sign extensions. A future version of the translator could do the sign extensions lazily to save on a bunch of redundant work in some cases. Another problem this benchmark causes is, that the hot sections use a lot of different not statically mapped registers. Resulting form this there is little benefit in the static register mapping, since they are not used a lot, or the dynamic allocation of registers as it is currently done, since the temporarily loaded registers will not be reused anyways. QEMU always loads register inputs from memory into temporary registers and then does the needed computation. Output registers may be reused as inputs for the following instructions and only written back when the used temporary is needed for another value. This means that the generated code for the part of the program mentioned above is effectively very similar to the one that QEMU generates preventing us from having a big advantage over QEMU.

The `600.perlbench` on the other hand has a lot of conditional branches and jumps in the hot functions. Many of these are not existing in the compiled x86–64 binary since they are replaced by conditional move instructions there. RISC–V does not have this instruction type and therefore these are replaced by branches in the RISC–V code. This already is a big performance penalty since branches are prone to misprediction resulting in needing to rollback the processor state while conditional moves simply cause some delay until the value is ready. If the compiler arranges code in a way that the outputs of the conditional moves are only used later, then this delay is basically no problem, while a jump misprediction might cause a major performance hit. Initially these jumps cause a lot of context switches since recursive translation currently is only employed for unconditional branches/calls, so for every one of the branches it initially is necessary to switch back for translating the taken paths. Recursively translating the path that is considered hot by the compiler might improve performance there by a bit. Combined with unconditional jumps, which are also very abundant in this section, this also causes a lot of unneeded retranslation, since jumping to the middle of another block is currently handled by treating it like translating a new block beginning at the jump target. These re-translated sections also might end with an unconditional jump which will cause more retranslation and so on. Apart from causing unnecessary parsing and translation overhead this also pollutes our code cache index increasing the possibility for hash collisions and thus increasing cache lookup time. It also severely hurts code density, making the CPU instruction cache less effective. This also means that the same

Option	Description
`no-ras`	Disable the return address stack
`no-chain`	Disable block chaining
`no-jump`	Disable recursive jump target translation
`no-fusion`	Disable macro operation fusion
`none`	All of the above

Table 6: The options for translator optimisations, as seen in `--optimize=help`.

conditional branch instruction might be copied at multiple places which means that the CPU branch predictor treats them as multiple branches which means it has less information for a right prediction.

5.3. Evaluation of translator optimisations

In order to evaluate the optimisations built into the translator, we ran the *SPEC CPU 2017* suite with various combinations of the available optimisation options in the same translator version (`v1.3.1`, the final release in the project's main development cycle).

The results of these runs can be seen in figure 6, and an overview of the specified switches can be found in table 6.

Benchmark	no-fusion	no-ras	no-jump, no-ras	none
600.perlbench	1.04	1.26	1.30	7.32
602.gcc	1.01	1.34	1.55	7.89
605.mcf	1.02	1.23	1.22	3.67
620.omnetpp	1.02	1.46	1.63	5.22
623.xalancbmk	1.02	1.45	1.59	7.66
625.x264	1.00	1.06	1.06	2.46
631.deepsjeng	1.01	1.56	1.56	7.40
641.leela	1.00	1.49	1.51	5.54
648.exchange2	1.00	1.00	1.00	8.71
657.xz	1.02	1.03	1.04	4.09
Avg	1.01	1.29	1.35	6.00

Table 7: Optimisation results data (including `--optimize=none` run), normalised to base.

Macro operation fusion does not seem to provide a large performance benefit, in most benchmarks the numbers do not even suggest any performance increase above natural deviation of benchmark runs. This means the implemented pattern matching does not give the desired effect of a good performance increase. Further tweaking of the checked patterns might make this optimisation more worthwhile.

The return address stack provided for a significant advantage in some benchmarks. Especially the function call heavy `620.omnetp`, `623.xalancbmk`, `631.deepsjeng`, `641.leela`

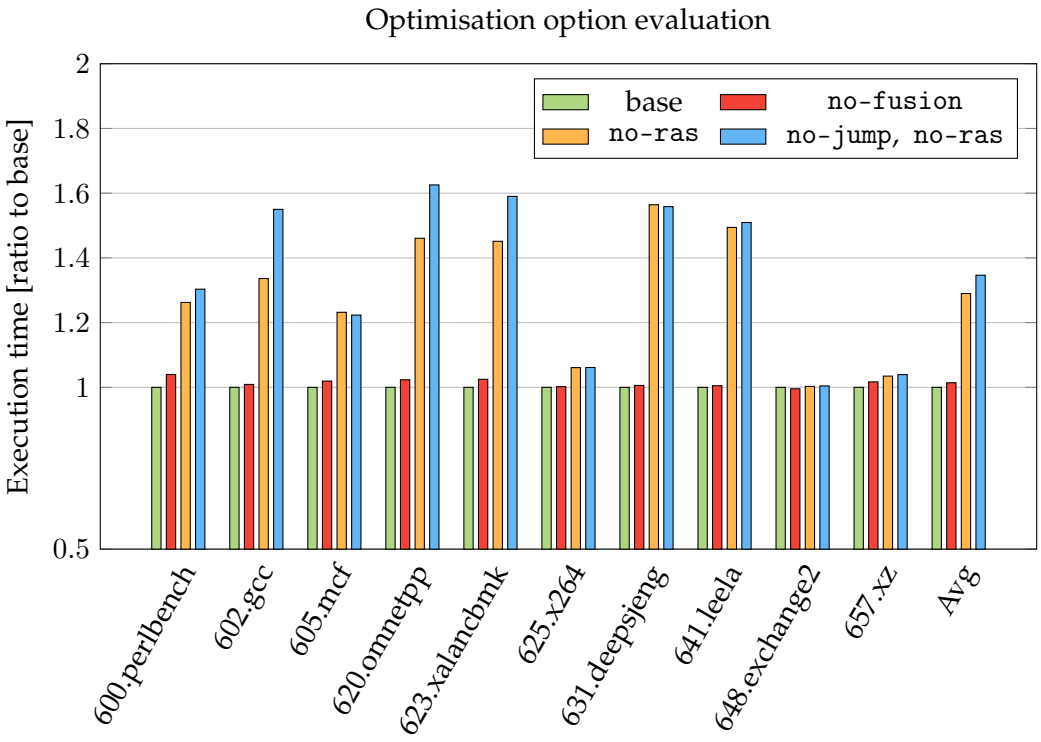


Figure 6: Results of `ref`-workload runs of *SPEC CPU 2017*s `intspeed` with various optimisation option combinations (normalised, lower is better).

benchmarks showed good performance gains of over $50\,\%$. The `600.perlbench`, as well as the `648.exchange2` and `657.xz` benchmarks where most of the runtime is spent in only a couple loops naturally could not benefit a lot.

Recursive jump translation without also utilising the return address stack only provided a performance increase over disabling both in some benchmarks. The main reason for this might be that this also makes context switches necessary on unconditional jumps that aren't function calls or returns. This makes jump-heavy benchmarks take a performance hit while jump-light benchmarks are almost unaffected.

Expectedly, the highest performance penalty was incurred by disabling chaining as well. This makes a context switch back to the translator necessary for every executed basic block. The benchmarks that are less impacted by disabling block chaining are the ones where fewer basic blocks were executed relative to their runtime. This correlates with the fact that the most executed blocks of these benchmarks contain more instructions and hence execute for a longer time.

Furthermore, the lazy replacement register handling described in section 3.3.1 had a high impact in some benchmarks, most notably `657.xz`, providing for a roughly $45\,\%$ performance increase – more than any other optimisation apart from block chaining. Any workload that frequently accesses registers that are not statically mapped as per table 3 on page 16 benefits significantly from this style of register handling, as these

registers will essentially behave as if they were statically mapped within the confines of that single basic block.

Of course, our chosen *least recently used*-approach to register replacement into three temporary slots suffers from the same issue known from caching: If a program accesses the same four not-statically-mapped registers in order in a loop, the algorithm will always replace and write back the value that would be needed next. Preventing this issue, however, is not a trivial task even when presented with the entirety of the guest program. So, as this approach can not perform worse than accessing the register file in memory for each instruction and has lead to significant performance increases in some workloads, this is a very worthwhile optimisation.

5.4. Data compression via gzip

Next to the results of the *SPEC CPU 2017* suite, it is also valuable to measure the performance of the translator in real-world workloads by running data compression via *gzip*.

For better comparability, both the native and RISC-V *gzip* binaries were compiled manually with the compiler optimisation level `-O3` alongside the linker flag `-static`. The RISC-V ABI was setup with `-march=rv64ima` and `-mabi=lp64`.

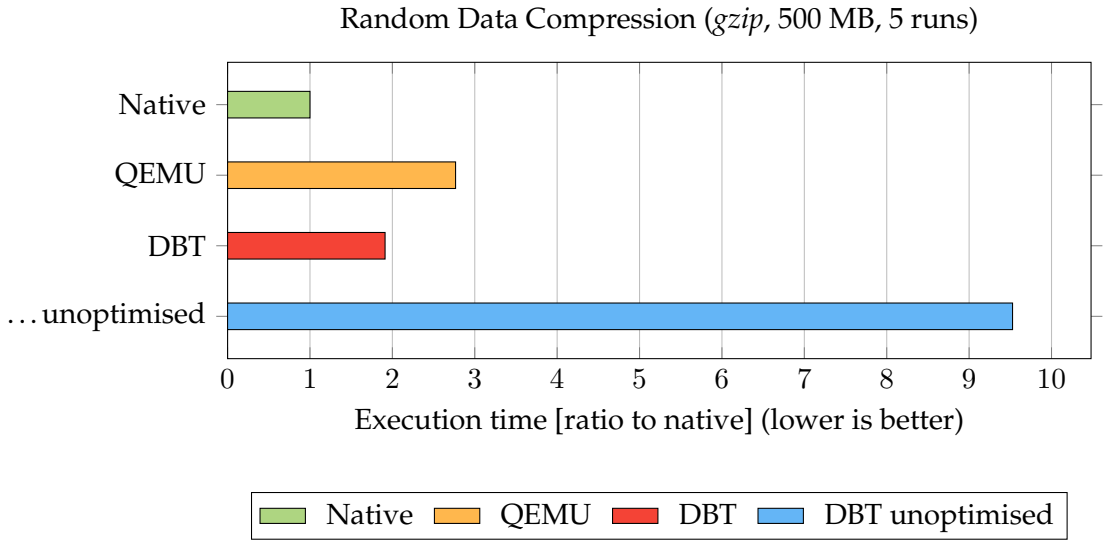


Figure 7: Execution time of gzip file compression (500 MB of random data, 5 runs) in seconds (normalised, lower is better).
Unoptimised run executed with `--optimize=none`.

Figure 7 lists the execution times of *gzip* compressing a pseudo-random 500 MB file sourced from `/dev/urandom`[4].

[4]Reproducible via `base64 /dev/urandom | head -c 524288000 > random.txt;`

Through our very efficient return address stack, recursive jump target translation, macro operation fusion and, most importantly, block chaining we are able to significantly outperform QEMU in random data compression by nearly 45 %. The achieved performance of approximately two times the execution time of a native run is in line with the *SPEC CPU 2017* results shown in figure 5.

As mentioned in the caption, the unoptimised run was performed with the command line option `--optimize=none`, which disables all of the optimisation features mentioned above. The translator will then have to translate every block one-by-one, jump back into the main loop on every block end and fetch the next position based on the current program counter.

5.5. Discussion

We could reach the goal of outperforming QEMU by employing a direct translation from RISC–V to x86–64 and using various optimisations. QEMU first translates the instructions into micro instructions in a plattform and ISA independent intermediate representation. While this allows easy implementation of new host and guest architectures, it also means that specific advantages of both guest and host can not be used. This leads to needing potentially more host instructions than necessary for a simple task. QEMU also does not employ a Return Address Stack, an optimisation which proved to be very worthwhile. One of the biggest disadvantages of QEMU though is, that it does not use a static register mapping. Instead it holds all registers in memory and only loads them into one of a couple of temporary registers, when needing them as input operands. The loaded output register may then be reused as input for the next instruction, if this instruction needs different inputs though, the register is written back and needs to be reloaded next time it is needed. This obviously leads to a big overhead which we can avoid by statically mapping the most used registers and dynamically allocating temporaries for the rest.

There is probably still some room for improvement to get closer to native compiled binaries. One possibilities here would be to further increase the quality of the generated code. As mentioned in section 5.3 on page 23 there is still room to increase the efficiency of the pattern matcher and the quality of the patterns. Something that might also provide a benefit is increasing the number of dynamically allocated register by decreasing the number of statically mapped registers. This could prevent unnecessary loads and write backs in case a block needs more than three different not mapped registers. Something that also could be done is treating jumps into the middle of block not as jumps to new blocks but redirecting them to the already translated block at the specific offset. For this you would need to keep track of the offsets of the specific translated instructions in the block, so it is possible to determine the jump target address. Overall though the biggest disadvantage to native compilation can be attributed to the greatly differing architectures. The compiler of course takes advantage of both architectures. On x86–64 this means using for example vectorisation. If a future RISC–V expansion offers support for this as well, there is a potential for a big performance increase. This means that while we are producing code that is effectively equivalent to what the compiler does for the

native binary, we do not have the possibility to generate code that is as efficient.

6. Summary

Looking back at the motivation for the project, we are able to achieve our goal of consistently outperforming QEMU [5]. The performance is measured using *SPEC CPU 2017* [11], an industry-standardised, CPU-intensive package of benchmark suites for measuring and comparing compute-intensive performance. In most cases, we can achieve an execution time of around twice the native one, but the tests also show areas with further optimisation possibilities. The quite significant performance increases in comparison to QEMU can be attributed to our specialisation for RISC–V to x86–64 translation, leading to a straight-forward design of the translation process and enabling specific optimisations for this guest-host pairing.

The benchmark suites not only helped to ensure the performance of our translator, but also tested the implementation in various scenarios and aided in discovering bugs due to their verification of the program outputs. In this sense, the *SPEC* suite also served as an integration test for us.

Furthermore, many unit tests were designed to assure the correct translation of (nearly) every supported RISC–V instruction in various contexts. Overall, nearly 30.000 parameterised test cases are being executed by our own test suites based on *Google Test* [13], covering the RISC–V parser as well as the emitted x86–64 assembly.

The development mainly focused on supporting and optimising the core instruction set and integer extensions so far. Support for floating point extensions, however, is also already implemented and tested. Our preliminary tests already show a significant performance increase in comparison to QEMU as a consequence of their use of emulation in order to offer floating point arithmetic support, contrasted by our use of native SSE–extension instructions.

Currently, our translator supports all base instruction sets and extensions needed for executing single threaded programs apart from the *C-* and *Q standard extensions*. We can easily add support for future standard extensions such as the *B standard extension* for bit manipulation when they come available, by adding decoding of this instructions to the parser and writing tailored translator functions. Implementing support for multi-threading would be a bigger undertaking, as it would require a redesign of our core architecture to support the necessary memory consistency and atomicity of memory accesses (see the *A standard extension*). Thus, we are not planning on adding this in the near future.

Moreover, some features related to the operating system are not provided yet, as only a subset of the existing system calls is implemented and loading of dynamically linked libraries at runtime is not yet supported.

The next steps for general improvement of the project would be implementing techniques used for integer arithmetic (e.g. dynamically allocated register replacement) for floating point arithmetic as well and supporting even more system calls. Besides this, one could invest into one of the aforementioned areas specifically, for example applying

a more rigid memory consistency model. The most development time concerning performance optimisations has been invested on optimising the emitted code and reducing the number of necessary context switches.

The current version of our project is capable of appreciably outperforming QEMU whilst providing almost the same functionality. In some aspects, e.g. floating point arithmetic, we are even ahead of QEMU by using native instructions from the SSE-extension instead of relying on emulation. Overall, the results we have demonstrated clearly show the usability and flexibility of our translator for executing general-purpose RISC–V binaries on an x86–64 processor.

Appendices

A. Download and installation instructions

The source code for the translator can be downloaded by checking out the project's git repository. Take care to either `git clone` the repository with the option flag `-recursive`, or to run the command

```
git submodule update --init
```

as the repository contains submodules that are required for compilation.

Then, the translator can be built by exeucting

```
sudo apt-get -y install gcc g++ cmake make autoconf meson
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INTERPROCEDURAL_OPTIMIZATION=true ..
make
```

in the root directory of the repository. Note that the build requires CMake version 3.15 or above. This will build two artifacts:

translator The dynamic binary translator. For details on the usage, see section C, or execute `./translator -h`.

test The unit test binary. It can be executed via `./test` and performs extensive unit testing of the RISC–V instruction implementations, the cache, register file, as well as the parser.

B. Executable program requirements

We can execute binaries compiled via the tools provided in the RISC–V GNU toolchain[5]. The executables need to be linked statically (pass the flag `-static` to GCC when compiling), as the translator does not support dynamically linked files.

We currently support binaries compiled for the architecture specifier `rv64imafd` (also called `rv64g`), meaning the compiler is free to utilise the base integer instruction set (`i`), as well as the instructions provided by the multiplication (`m`), atomic (`a`) and floating point standard extensions (`f`, `d`). This can be achieved by passing `-march=rv64g` to GCC[6].

C. Using the translator

```
./translator [translator option(s)] -f <filename> [guest options]
```

[5] For further information as well as download and usage instructions, see `https://github.com/riscv/riscv-gnu-toolchain` (last accessed on 25.09.2020).

[6] Note that some architecture strings require recompilation of the toolchain. Also, excluding the floating point instructions in the architecture string implies `-mabi=lp64`.

Seen above is the syntax for executing the translator with a guest program. All possible translator options are described in the help text, as seen by executing `./translator -h`. Every option after the filename specified via the `-f` flag is passed along to the guest in its `argv`, so all options intended for the translator must be passed before `-f`.

The command line options also include the ability to analyse (`-a`) the binary to produce a detailed breakdown of which instruction mnemonics and registers the guest will use when executed. Furthermore, it includes the ability to time the execution of only the guest program (`-b`) and profile the register and cache usage (`-p`).

Logging can be controlled by passing the requested category to the `--log` flag as detailed in the help, and can provide insights into the state of the translator during execution or debugging. Lastly, it is also possible to selectively disarm optimisation features like the return address stack, block chaining, recursive jump target translation or macro operation fusion via `--optimize`.

D. Version history

The following mirrors the version control change log of the translator over the course of its development.

Version 1.3.1 (latest)

- fix a bug where the replacement register recency was not reset correctly when loading a non-mapped-register that was already present
- prevent redundant writes to x0 in the A-extension translation
- reorder patterns for more efficient translations
- various minor cleanups

Version 1.3.0

- implement support for F-/D-extension including a static register mapping
- expand unit testing coverage to test combinations for floating point instructions
- optimise chaining for conditional branches
- cleanup and fix various small issues in the code base

Version 1.2.4

- implement a lazy runtime register replacement strategy, keeping the not-statically-mapped values in the replacement registers as long as possible to prevent redundant memory operations inside blocks
- add logging for static and dynamic register mapping
- implement patterns for MV and LI, ADDI with rs1 == x0 and several shifting combinations as well as zero-extensions via ANDI and 0xff
- use the x86 LEA instruction for faster translations of various input instructions
- rewrite and optimise the return address stack

- fix the `-m` short option to include `--optimize=no-fusion`
- use the output of `git describe` for the version string to include the commit hash in the build
- add inline logging for generated assembly with `--log=asm-out`
- split up the analyser command line flags to specify what to analyse
- bump C standard to C11

Version 1.2.3

- do not page-align the generated code
- implement pattern matching to apply macro operation fusion of multiple RISC–V instructions
- optimise various instructions and clean up legacy code
- gain performance to approx. 1.4x–1.7x faster than QEMU

Version 1.2.2

- expand the static register mapping for better performance overall
- reallocate the code cache index and rehash all values when it is 50 % full for better lookup performance on capacity overflow
- rewrite command line options parsing to allow for long options (see `./translator -h`)
- allow finer control of specific optimisation features via `--optimize`
- add instruction pattern matching to the binary analyser to gather data for macro operation fusing
- add a profiler for counting register accesses
- implement emulation for syscalls `faccessat`, `getrandom`, `renameat2`
- remove emulation for the syscall `clone`
- fix crash when the code cache fills up by increasing the memory space available for translated blocks

Version 1.2.1

- add implementations for `AMOMIN` and `AMOMAX` instructions
- add extensive unit testing for atomic and arithmetic instructions, as well as the parser
- fix several issues to enable the SPEC CPU 2017 benchmark suite to run
- implement emulation for syscalls `chdir`, `pipe2`, `getdents64`, `munmap`, `clone`, `execve`, `wait4`
- fix `ORI` instruction being parsed as `XORI`
- fix instruction semantics for `SUB(W)`
- finalize implementation of the return address stack

Version 1.2.0

- enable register mapping for translated instructions
- add context switching from host to guest programs
- rework instruction translator function for flexibility
- implement a return address stack
- implement a TLB for cache lookup of blocks
- flip `-m` translation optimiser flag (enabled by default, flag now turns off optimisations)

Version 1.1.0

- cleanup and refactor project files
- remove all C++ usage from translator code
- eliminate standard library usage
- add performance measuring flags to the translator

Version 1.0.1

- fix an issue with the `read` system call that causes blocking problems with gzip

Version 1.0 The initial release of the translator capable of executing gzip.
This release supports

- the RISC–V integer instruction set
- the multiplication extension instructions (M)
- the atomic extension instructions (A).

The latter are not yet implemented atomically, however they make the translator compatible with binaries compiled for the architecture `rv64ima`, with the ABI `lp64`.

List of Tables

1. Memory layout 5
2. Patterns used for macro-op-fusion 12
3. Active static register mapping 16
4. Specially handled system calls overview 19
5. SPEC CPU 2017 workload description 20
6. Translator optimisation options 23
7. Optimisation results 23

List of Figures

1. Block chaining illustration 11
2. Translator execution control flow 13
3. Dynamic Register Usage Analysis 15
4. Example assembly for dynamic register allocation 17
5. SPEC CPU 2017 Results 21
6. Translator optimisation evaluation results 24
7. Execution time of gzip compression 25

References

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

[2] Editors Andrew Waterman and Krste Asanović, *The RISC-V Instruction Set Manual, Volume I: User Level ISA, Document Version 20191213*. RISC-V Foundation, Dec. 2019.

[3] "The sifive home page." `https://www.sifive.com/` (last visited 23.10.2020), 2020.

[4] M. Probst, "Dynamic binary translation," in *UKUUG Linux Developer's Conference*, vol. 2002, 2002.

[5] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

[6] "Linux elf man page." `https://linux.die.net/man/5/elf` (last visited 26.10.2020) or by executing `man elf` on any linux machine, 2020.

[7] "Gcc, the gnu compiler collection." `https://gcc.gnu.org/` (last visited 23.10.2020), 2020.

[8] M. Clark and B. Hoult, "rv8: a high performance risc-v to x86 binary translator," *1st CARRV*, 2017.

[9] Alexis Engelke, M.Sc., "Fadec – fast decoder for x86-32 and x86-64." `https://github.com/aengelke/fadec`, 2020.

[10] "The gzip home page." `https://www.gzip.org/` (last visited 02.10.2020), 2020.

[11] "SPEC CPU 2017." `https://www.spec.org/cpu2017/` (last visited 02.10.2020), 2020.

[12] "SPEC CPU 2017 Documentation." `https://www.spec.org/cpu2017/Docs/overview.html` (last visited 02.10.2020), 2020.

[13] Google, "Google test." `https://github.com/google/googletest`, 2020.