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

RISC-V is an open ISA first conceptualised in 2010 with the initial goals of research and education in mind. In contrast to Intel's x86 [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.

1.1. Problem description

There is already some hardware available for RISC-V[1], but it is not yet widespread. Developers usually don't have access to real hardware, so they must instead rely on emulation to test their code written for the RISC-V platform.

Modes of binary translation When attempting to execute programs compiled for a foreign architecture on a different native one, there are essentially three distinct approaches at one's disposal.

The main principles to achieve this are:

- **Interpretation**, where, much alike interpreted programming languages (e.g. 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

[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.

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 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.].

1.2. Motivation

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 code on an x86-64 machine by means of dynamic binary translation.

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. RISC-V's load-store architecture with a 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 hardware 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 (e.g. by vectorisation) is difficult even for compilers presented with the entire program context. 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*) 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:

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

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.

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

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 type of instruction set architecture 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 a single arithmetic `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 translation's amount of CPU clock cycles, 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`

[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.

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

3.2. Code cache and block handling

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

For that, 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.

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, the target's removal 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

3.3.1. Handling of guest registers

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.

As x86-64 only provides for 16 general-purpose registers (`rax`–`rdx`, `rbx`, `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

[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.

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 x86-64's ability to extensively use memory operands in the instructions. 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 host registers inside a single block's translation. 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's 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's 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 the host's calling convention 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's 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 translator's memory region, 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 the *gcc* [6] is able to emulate floating point arithmetic using integer arithmetic, using the native support of the x86-64 SSE extension is evidently a lot faster.

The main difficulties (and their resolutions) that arise by using the x86-64 SSE extension 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-extension only provides 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 register `XMM0`, `XMM1` are just reserved as replacements and missing registers are moved into them from memory temporally.

- **Missing equivalent SSE instruction** can lead to a huge instruction overhead, as emulation often use bit manipulation operations instead. Therefore constants or masks need to be moved in from either memory or the general purpose registers because the SSE extension does not support immediates. 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. 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. One could use EVEX-encoding [1, S. 374] to set the rounding mode on a per instruction level for x86-64 as well, but faenc [**?**] does not support EVEC-encoding so far. Thus in case a instruction with explicit rounding mode is encountered, the rounding mode is temporally 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

3.6.1. Recursive jump translation

Returning from translated guest code to the translator's main transcode loop comes with performance penalties 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 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 jump target's host code address is now already known at translate time, we can emit a direct jump to the target block 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 the translator reaches a branch, its two sides can only be chained if the target block has already been translated and its host code address is known. If that is not the case, the target can instead be chained retroactively after the respective side of the branch is first taken: Following translation of the target block, the branch's host code is modified to jump to the targets now known address directly, instead of returning to the transcode loop. Further cache lookups and context switching are thus avoided for this branch.

`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 [7].

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 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 holding entries for the calls' respective return addresses, that consist 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. The call's then emitted host code will push its return address pair onto this stack at every execution without having to leave guest context. Following dynamic jumps will compare their target address with the value in the top stack entry, and, if it matches, pop the address pair and jump to the target block directly, thus avoiding having to return to the transcode loop. In case of a mismatch the stack will not be changed and control is handed back to the host.

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 the generated code's performance. 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 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

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 4 byte blocks with the information being extracted in an intermediate instruction format holding the mnemonic, operands and immediates in uncompressed form. This intermediate format can then be used by special per-mnemonic translation functions which, in the end, generate the x86-64 byte code using faenc.

4.2. Static hybrid register mapping

4.2.1. Register priority analysis

In order to achieve the best performance with the hybrid approach to the register mapping described in section 3.3 on page 6, we must decide which registers of the RISC-V guest to map into the host's limited number of available GPRs. 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* [8] and several benchmarks of the `intspeed`-Suite of *SPEC CPU 2017* [9], see figure 1.

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.

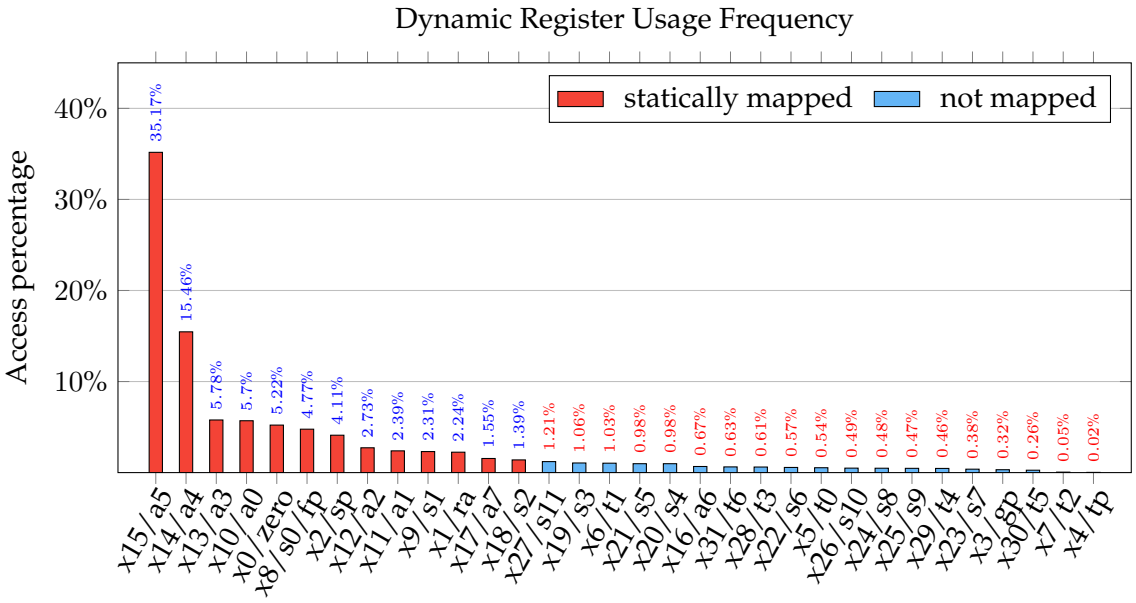

Figure 1: The average results of a dynamic register usage analysis, ordered by frequency.

The benchmarks `605.mcf_s` (route planning workload) and `620.omnetpp_s` (discrete event simulation for computer networking) [10] of the *SPEC CPU* suite can serve as examples here. For programs like `605.mcf_s` 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_s` 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.

4.2.2. Structure of the mapping

When we structure our mapping by the average case of the insights gained, 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.

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.

Taking the 12 registers that are most accessed on average, the mapping structure is as seen in table 3 on the next page.

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).

4.2.3. 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. 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 value's 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. Detailed system call overview

As described in section 3.4 on page 7, 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 page 16 for details). The following will describe the

```
1  mov    rax,PTR <gp_file+24>
2  movsxd rax,PTR [rax-0x3dc]
3  mov    PTR <gp_file+128>,rax
4  mov    rbx,0xf0f68
5  mov    ebp,PTR <gp_file+128>
6  add    rbx,rbp
7  mov    rax,PTR <gp_file+128>
8  add    eax,0x1
9  movsxd rax,eax
10 mov    PTR <gp_file+128>,rax
11 mov    BYTE PTR [rbx],r11b
12 mov    rax,PTR <gp_file+24>
13 mov    rdx,PTR <gp_file+128>
14 mov    PTR [rax-0x3dc],edx
15 mov    rax,PTR <gp_file+24>
16 movzx  ebx,BYTE PTR [rax-0x3e8]
17
18
19 movsxd r14,PTR <gp_file+128>
20 test   rdi,rdi
21 je     no_match
22 mov    QWORD PTR <gp_file+256>,
23        0x19224
24 jmp    end
25 no_match:
26 mov    QWORD PTR <gp_file+256>,
27        0x1913c
28 end:
29 ret
```

(a) Without dynamic register mapping

```
1  mov    rax,PTR <gp_file+24>
2  movsxd rdx,PTR [rax-0x3dc]
3
4  mov    rbx,0xf0f68
5  mov    ebp,edx
6  add    rbx,rbp
7
8  add    edx,0x1
9  movsxd rdx,edx
10
11 mov    BYTE PTR [rbx],r11b
12
13
14 mov    PTR [rax-0x3dc],edx
15
16 movzx  ebx,BYTE PTR [rax-0x3e8]
17 mov    PTR <gp_file+24>,rax
18 mov    PTR <gp_file+128>,rdx
19 movsxd r14,PTR <gp_file+128>
20 test   rdi,rdi
21 je     no_match
22 mov    QWORD PTR <gp_file+256>,
23        0x19224
24 jmp    end
25 no_match:
26 mov    QWORD PTR <gp_file+256>,
27        0x1913c
28 end:
29 ret
```

(b) With dynamic register mapping

Figure 2: A comparison of the code generated by the DBT prior to and after introducing dynamic register allocation.

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 host's returned data 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 translator's 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 endpoint of the DBT's data segment (*program break*), thus potentially deallocating some of the translator's 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 the translator's memory region. Mappings that do not interfere with the DBT's memory can be passed along to the host directly. In case a hinted mapping would conflict with the translator's memory, we may just re-hint the mapping to the top of the guest's 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 the guest's `munmap` with `EINVAL` in cases where the translator's memory would be compromised by the de-allocation.

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:

- `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`

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.

SPECspeed Benchmark	Workload
`600.perlbench_s`	Perl interpreter
`602.gcc_s`	GNU C compiler
`605.mcf_s`	Route planning
`620.omnetpp_s`	Discrete Event simulation – computer network
`623.xalancbmk_s`	XML to HTML conversion via XSLT
`625.x264_s`	Video compression
`631.deepsjeng_s`	Artificial Intelligence: alpha-beta tree search (Chess)
`641.leela_s`	Artificial Intelligence: Monte Carlo tree search (Go)
`648.exchange2_s`	Artificial Intelligence: recursive solution generator (Sudoku)
`657.xz_s`	General data compression

Table 5: A description of the workloads covered by *SPEC CPU 2017*'s `intspeed` suite [10].

5. Results and Performance

Measuring the performance of the DBT was accomplished by using the tools in *SPEC CPU 2017*'s `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 benchmark's translation.

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 DBT's 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* [8], 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`.

5.1. SPEC CPU 2017 Results

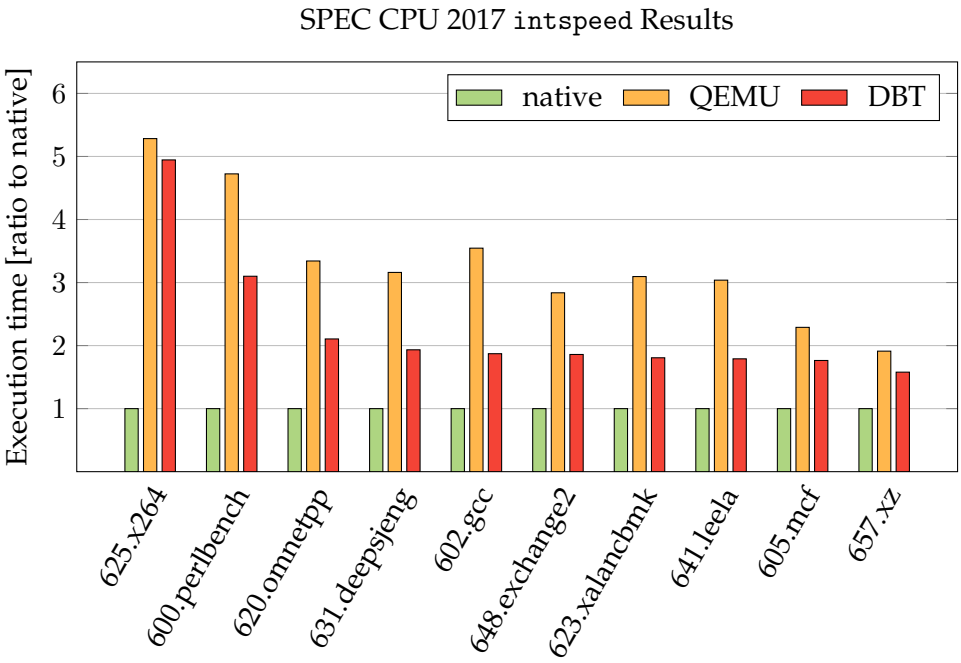

Figure 3: Results of `ref`-workload runs of *SPEC CPU 2017*'s `intspeed` (normalised, lower is better).

5.1.1. Analysis

Figure 3 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 on page 9 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 GCC compiler benchmark the advantage grows to a comfortable $80\,\%$.

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

`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.

The `perlbench` on the other hand has a lot of conditional branches and jumps in the hot blocks. This causes a lot of context switches since recursive translation currently is only employed for unconditional branches/calls. Recursively translating the path that is considered hot by the compiler could improve performance by a bit. It also potentially causes redundant work for the translator, since jumps to the middle of a basic block currently are handled by treating it like a new block beginning at the jump target.

5.2. 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 4, and an overview of the switches specified in the figure's legend can be found in table 6.

Macro operation fusion does not seem to provide a lot of a performance benefit, in

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`.

most benchmarks the numbers do not even suggest any performance increase above natural deviation of benchmark runs. This means the implemented pattern matching did not give the desired effect of a good performance increase. Further tweaking of the checked patterns might make this optimisation more worth it.

The return address stack gave a significant advantage in some benchmarks. Especially the function call heavy 620, 623, 631, 641 benchmarks showed good performance gains of over $50\,\%$. The 600, 648, 657 benchmarks where most of the runtime is spent in only a couple loops naturally could not benefit a lot.

Recursive jump translation without also utilizing 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 had lower performance losses are the ones were less basic blocks were executed relative to their runtime. This can be justified by seeing that the most executed parts of the these benchmarks are blocks that are very long.

Something that also needs to be noted that the lazy replacement register handling described in section 3.3.1 on page 6 had a high impact in some benchmarks, most notably 657, providing a roughly $45\,\%$ performance increase there more than any other optimisation apart from chaining.

5.3. 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 5 on page 21 lists the execution times of *gzip* compressing a pseudo-random

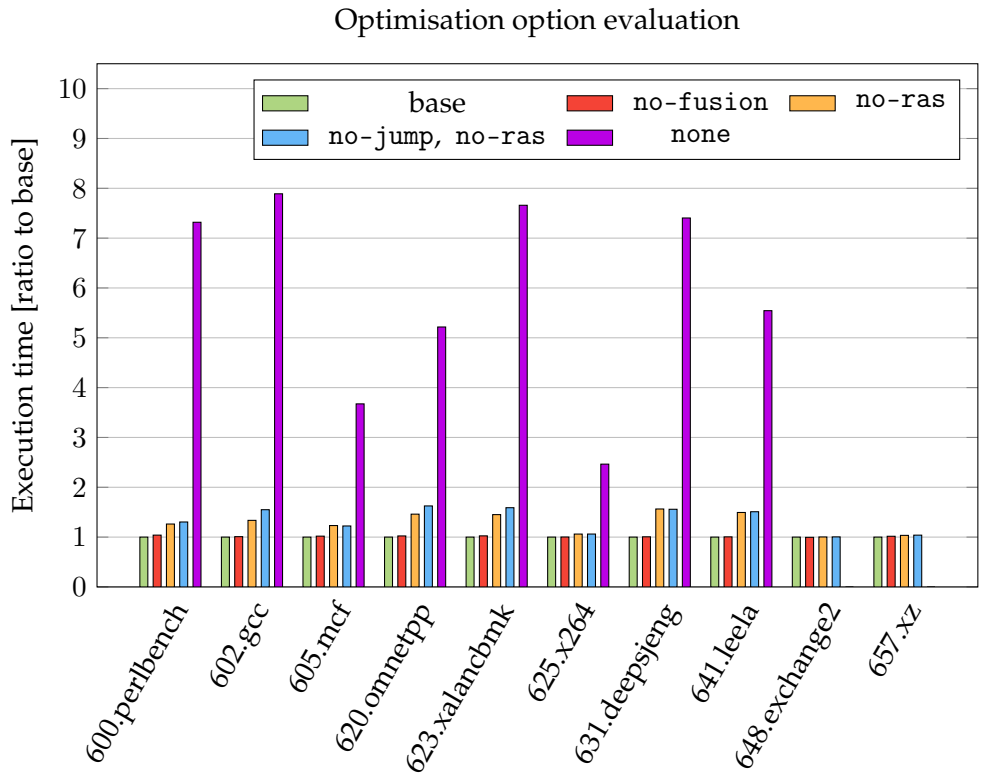

Figure 4: Results of `ref`-workload runs of *SPEC CPU 2017*'s `intspeed` with various optimisation option combinations (normalised, lower is better).

500 MB file sourced from `/dev/urandom`[4].

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 3.

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.

6. Summary

Summary here…

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

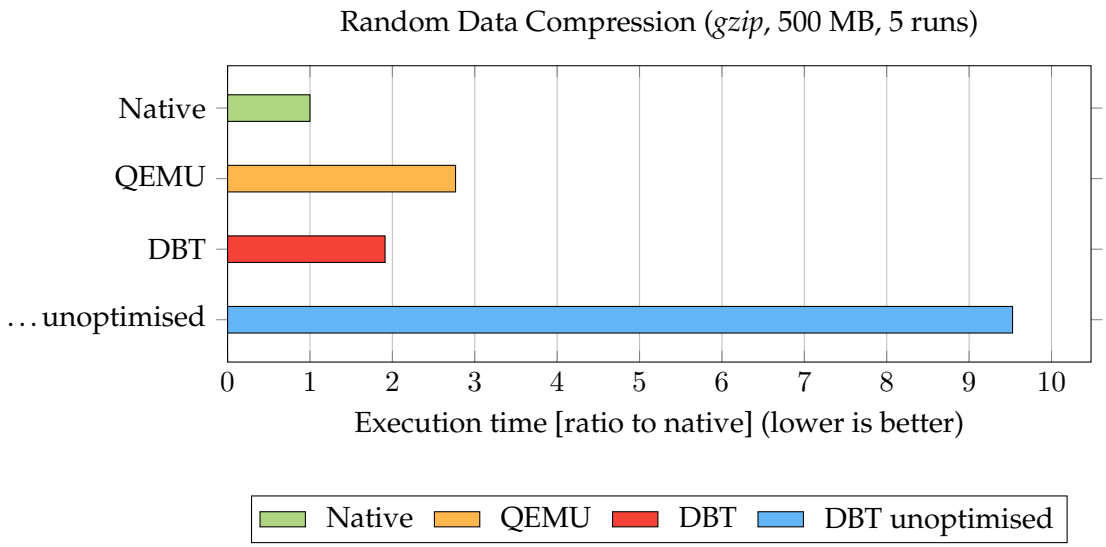

Figure 5: Execution time of gzip file compression (500 MB of random data, 5 runs) in seconds (normalised, lower is better).
Unoptimised run executed with `--optimize=none`.

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]
```

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

[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`.

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

List of Figures

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

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

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

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

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