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 **3**
1.1. Problem description 3

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

3. Approach **6**
3.1. Translating the partitioned code 6
3.2. Code cache and block handling 6
3.3. Register handling and context switching 7
3.4. System call handling 8

4. Implementation Details **8**
4.1. System architecture and execution control flow 9
4.2. Instruction translation process 9
4.3. Code cache and TLB for block lookup 9
4.4. Static hybrid register mapping 9
4.5. Context switching details 11
4.6. Optimisation of the generated code 11
4.7. Detailed system call overview 11

5. Results and Performance **13**
5.1. SPEC CPU 2017 Results 13
5.2. Evaluation of translator optimisations 13
5.3. Data compression via gzip 15

6. Summary **16**

Appendices **17**
A. Download and installation instructions 17
B. Executable program requirements 18
C. Using the translator 18
D. Version history 18

References **22**

1. Introduction

RISC-V is an open ISA first conceptualised in 2010 with the initial goals of research and education in mind. Follows the RISC (Reduced Instruction Set Computer) Scheme (like ARM) in contrast to x86-64... Its development took the lessons learned in terms of backwards compatibility and future-proofing from other widespread ISAs like Intel x86 into account, and aimed 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 [1, 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) already defined and the possibility to expand further.

1.1. Problem description

There is already some hardware available for RISC-V (see/maybe cite SIFive), but it is not yet widespread and a lot of developers won't have access to real hardware, so they must rely on emulation to test their code.

1.1.1. Modes of binary translation

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

The main possibilities 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 host system in order to simulate the guest ISA.

  While being probably 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 guest executable is statically reverse-engineered and translated to the guest 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 guest addresses to native at runtime. Any program that produces or edits assembly at runtime would also 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 worrying about unused code paths and also the other mentioned issues are 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 [2, S. 1f.].

1.1.2. Motivation

One of the most popular emulators is QEMU. While QEMU is a portable DBT that supports a wide variety of guest and host architectures and ISAs, this also makes it hard to optimize 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.

By its very nature, executing code compiled for one architecture on a different one is not an easy task.

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

It is obvious that there are major differences in the two architectures, implied 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 this DBT is the different address format and mismatch of general purpose and floating point registers in number.

RISC-V's load-store architecture with a three-operand instruction format allows for a better reuse of data but more instructions due to the explicit load/store operations.

x86-64 however has a register-memory architecture with a two-operand instruction format leading to more implicit/fused memory accesses being used in optimized code.

Also the very nature of translating a RISC architecture into a CISC architecture might seem like it would lead to less instructions. In praxis the efficient fusion of multiple RISC-V instructions into single x86-64 instructions is difficult considering the fact that we are only presented with the assembly. Even more a naive implementation leads to an instruction overhead due to the mismatch in supported operands per instruction.

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 on page 7) 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 [2, S. 2].

The stack is set up exactly like the linux kernel would. As such the stack pointer needs to point at the argument count and then towards higher addresses in order there are the zero terminated argument, environment and auxiliary vector. Finally some alignment bytes need to be added, so the stack pointer is ABI-conformant 16 Byte aligned. All of the information is basically just copied from the host in our case.

The memory is laid out as follows:

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[1].

[1] 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.

3. Approach

3.1. Translating the partitioned 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 `add 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 adding the value of `rs2` to `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 pseudoinstructions (as mentioned in section 2.1) are also of some help here [1, 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 `auipc 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`.

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 [2, 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`[2] 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, rsi, rdi, rsp, rbp 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 (`rax` and `rdx` in `(i)mul`, `cl` in shifting, etc.), 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 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 [2, S. 8f.].

Accordingly, the solution for this problem would be an approach that employs parts of both of these extremes [2, 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 seldomly accessed guest registers.

[2] `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.

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 [2, 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 [2, S. 2f.].

4. Implementation Details

The following section aims to provide an in-depth overview of the system's architecture as well as the rationale for major design decisions taken during the implementation.

4.1. System architecture and execution control flow

4.2. Instruction translation process

The decoding of the RISC-V assembly is quite straight forward as we have a fixed instruction length of 32bit. Thus, the assembly is parsed in 4byte blocks with the information beeing extracted in an intermediate instruction format, holding the instruction 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.

4.3. Code cache and TLB for block lookup

4.4. Static hybrid register mapping

4.4.1. 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 7, 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 results of such an analysis performed on a range of programs, including *gzip* [3] and several benchmarks of the `intspeed`-Suite of *SPEC CPU 2017* [4], see table 2 on the following page. 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_s` (route planning workload) and `620.omnetpp_s` (discrete event simulation for computer networking) [5] of the *SPEC CPU* suite can serve as

Register analysis results

*: for SPEC benchmarks: test workload was chosen
**: workload: approx. 150 MB text file sourced from /dev/urandom

Static: Number of times the register is mentioned in the binary. Computed statically by parsing and analysing the **Run via:** ./translator -a -f <binary>
Dynamic: Number of register accesses during execution. Computed dynamically by logging each read/write to a pro **Run via:** ./translator -p -f <binary> [guest arguments]

Static analysis

Ranked hitlist	sort		gzip		602.gcc_s		605.mcf_s		620.omnetpp_s		631.deepsjeng_s		Average	Ranked hitlist	Average Static Hits
Access frequency														Access frequency	
1	x15/a5	16,79%	x15/a5	22,71%	x15/a5	17,92%	x15/a5	16,79%	x15/a5	13,80%	x15/a5	16,87%	x15/a5	17,12%	1	
2	x0/zero	12,16%	x0/zero	11,50%	x0/zero	13,24%	x0/zero	12,20%	x2/sp	12,09%	x0/zero	11,82%	x0/zero	12,23%	2	
3	x14/a4	8,79%	x14/a4	10,06%	x14/a4	9,25%	x14/a4	8,93%	x10/a0	11,35%	x14/a4	9,33%	x10/a0	9,19%	3	
4	x10/a0	8,16%	x10/a0	7,09%	x10/a0	8,78%	x10/a0	7,71%	x0/zero	9,41%	x10/a0	7,91%	x14/a4	8,44%	4	
5	x2/sp	8,11%	x8/s0/fp	6,98%	x1/ra	8,56%	x2/sp	7,48%	x1/ra	9,29%	x2/sp	7,36%	x1/ra	8,02%	5	
6	x8/s0/fp	5,84%	x2/sp	6,77%	x2/sp	6,80%	x13/a3	6,01%	x8/s0/fp	7,38%	x13/a3	6,24%	x2/sp	8,01%	6	
7	x13/a3	5,52%	x13/a3	5,38%	x8/s0/fp	4,90%	x8/s0/fp	5,79%	x14/a4	5,45%	x8/s0/fp	5,44%	x8/s0/fp	5,58%	7	
8	x11/a1	4,21%	x11/a1	3,73%	x13/a3	4,46%	x12/a2	4,32%	x11/a1	5,27%	x12/a2	4,38%	x11/a1	4,42%	8	
9	x12/a2	4,20%	x12/a2	3,72%	x11/a1	4,20%	x11/a1	4,26%	x9/s1	4,59%	x11/a1	4,22%	x13/a3	4,31%	9	
10	x1/ra	3,54%	x1/ra	3,26%	x9/s1	3,74%	x9/s1	3,24%	x12/a2	3,22%	x1/ra	3,13%	x9/s1	3,83%	10	
11	x9/s1	3,21%	x9/s1	2,84%	x12/a2	3,71%	x1/ra	3,24%	x18/s2	3,09%	x9/s1	3,08%	x12/a2	3,66%	11	
12	x18/s2	2,23%	x18/s2	2,03%	x18/s2	2,61%	x18/s2	2,34%	x13/a3	2,94%	x18/s2	2,32%	x18/s2	2,66%	12	
13	x19/s3	1,99%	x19/s3	1,72%	x19/s3	2,06%	x19/s3	2,06%	x19/s3	2,23%	x19/s3	2,05%	x19/s3	2,08%	13	
14	x20/s4	1,52%	x20/s4	1,22%	x20/s4	1,54%	x20/s4	1,47%	x20/s4	1,62%	x20/s4	1,53%	x20/s4	1,54%	14	
15	x21/s5	1,42%	x21/s5	1,18%	x21/s5	1,27%	x21/s5	1,46%	x21/s5	1,22%	x21/s5	1,48%	x21/s5	1,27%	15	
16	x16/a6	1,40%	x16/a6	1,13%	x22/s6	1,10%	x16/a6	1,36%	x22/s6	0,90%	x16/a6	1,39%	x22/s6	1,06%	16	
17	x22/s6	1,21%	x23/s7	1,01%	x23/s7	0,98%	x23/s7	1,25%	x23/s7	0,85%	x26/s10	1,22%	x23/s7	0,97%	17	
18	x25/s9	1,20%	x22/s6	0,97%	x24/s8	0,85%	x26/s10	1,22%	x6/t1	0,83%	x17/a7	1,22%	x24/s8	0,84%	18	
19	x26/s10	1,15%	x26/s10	0,96%	x25/s9	0,80%	x22/s6	1,19%	x24/s8	0,74%	x22/s6	1,22%	x25/s9	0,81%	19	
20	x23/s7	1,15%	x17/a7	0,95%	x26/s10	0,74%	x17/a7	1,18%	x25/s9	0,68%	x23/s7	1,17%	x26/s10	0,75%	20	
21	x17/a7	1,15%	x25/s9	0,94%	x27/s11	0,70%	x25/s9	1,15%	x16/a6	0,64%	x27/s11	1,10%	x27/s11	0,70%	21	
22	x24/s8	1,00%	x27/s11	0,86%	x16/a6	0,53%	x27/s11	1,07%	x26/s10	0,55%	x25/s9	1,09%	x16/a6	0,65%	22	
23	x27/s11	0,94%	x24/s8	0,84%	x17/a7	0,34%	x24/s8	1,02%	x27/s11	0,51%	x24/s8	1,02%	x6/t1	0,49%	23	
24	x6/t1	0,72%	x6/t1	0,62%	x6/t1	0,34%	x6/t1	0,77%	x17/a7	0,42%	x6/t1	0,80%	x17/a7	0,46%	24	
25	x3/gp	0,61%	x28/t3	0,44%	x28/t3	0,14%	x28/t3	0,57%	x3/gp	0,34%	x28/t3	0,59%	x3/gp	0,21%	25	
26	x28/t3	0,52%	x29/t4	0,29%	x3/gp	0,13%	x3/gp	0,54%	x28/t3	0,18%	x3/gp	0,56%	x28/t3	0,20%	26	
27	x29/t4	0,32%	x4/tp	0,21%	x29/t4	0,11%	x29/t4	0,37%	x29/t4	0,13%	x29/t4	0,42%	x29/t4	0,15%	27	
28	x30/t5	0,23%	x30/t5	0,20%	x30/t5	0,08%	x30/t5	0,26%	x30/t5	0,08%	x30/t5	0,27%	x30/t5	0,10%	28	
29	x4/tp	0,23%	x31/t6	0,18%	x31/t6	0,07%	x4/tp	0,23%	x31/t6	0,06%	x31/t6	0,24%	x31/t6	0,09%	29	
30	x31/t6	0,22%	x5/t0	0,13%	x5/t0	0,05%	x31/t6	0,23%	x4/tp	0,05%	x4/tp	0,21%	x5/t0	0,06%	30	
31	x7/t2	0,13%	x7/t2	0,09%	x7/t2	0,04%	x5/t0	0,16%	x5/t0	0,04%	x5/t0	0,17%	x7/t2	0,05%	31	
32	x5/t0	0,12%	x3/gp	0,00%	x4/tp	0,01%	x7/t2	0,12%	x7/t2	0,04%	x7/t2	0,16%	x4/tp	0,05%	32	

77,32% mapped | **22,68% not mapped**

Dynamic

Ranked hitlist	sort		gzip**		602.gcc_s*		605.mcf_s*		620.omnetpp_s*		631.deepsjeng_s*		Average	Ranked hitlist	Average Dynamic Hits
Access frequency														Access frequency	
1	x15/a5	40,92%	x15/a5	57,77%	x15/a5	20,03%	x15/a5	15,38%	x15/a5	15,21%	x15/a5	22,02%	x15/a5	35,17%	1	
2	x8/s0/fp	21,43%	x14/a4	19,35%	x14/a4	10,44%	x14/a4	10,11%	x2/sp	11,63%	x14/a4	14,97%	x14/a4	15,46%	2	
3	x14/a4	20,04%	x8/s0/fp	5,53%	x2/sp	10,41%	x0/zero	9,11%	x10/a0	10,19%	x10/a0	10,22%	x13/a3	5,78%	3	
4	x13/a3	7,59%	x13/a3	4,42%	x0/zero	7,88%	x10/a0	6,73%	x14/a4	9,71%	x13/a3	8,36%	x10/a0	5,70%	4	
5	x2/sp	2,73%	x0/zero	3,07%	x13/a3	7,01%	x17/a7	6,67%	x0/zero	8,17%	x2/sp	5,59%	x0/zero	5,22%	5	
6	x0/zero	1,90%	x9/s1	2,56%	x10/a0	6,55%	x8/s0/fp	6,53%	x13/a3	5,87%	x0/zero	5,03%	x8/s0/fp	4,77%	6	
7	x1/ra	1,31%	x2/sp	2,51%	x8/s0/fp	4,89%	x13/a3	4,36%	x8/s0/fp	5,45%	x12/a2	4,19%	x2/sp	4,11%	7	
8	x10/a0	0,99%	x18/s2	1,05%	x12/a2	4,41%	x11/a1	4,33%	x1/ra	5,25%	x11/a1	3,24%	x12/a2	2,73%	8	
9	x12/a2	0,98%	x1/ra	0,89%	x1/ra	3,85%	x6/t1	3,99%	x11/a1	5,13%	x1/ra	2,83%	x11/a1	2,39%	9	
10	x11/a1	0,78%	x10/a0	0,87%	x11/a1	3,78%	x12/a2	3,97%	x12/a2	4,15%	x8/s0/fp	2,53%	x9/s1	2,31%	10	
11	x9/s1	0,21%	x12/a2	0,79%	x9/s1	3,42%	x27/s11	3,81%	x9/s1	3,66%	x9/s1	1,84%	x1/ra	2,24%	11	
12	x16/a6	0,14%	x21/s5	0,36%	x18/s2	2,66%	x1/ra	2,81%	x18/s2	2,44%	x18/s2	1,74%	x17/a7	1,55%	12	
13	x18/s2	0,11%	x11/a1	0,33%	x16/a6	2,41%	x31/t6	2,73%	x19/s3	1,91%	x20/s4	1,64%	x18/s2	1,39%	13	
14	x19/s3	0,11%	x20/s4	0,25%	x19/s3	2,36%	x5/t0	2,54%	x16/a6	1,57%	x27/s11	1,64%	x27/s11	1,21%	14	
15	x26/s10	0,09%	x19/s3	0,24%	x20/s4	1,68%	x21/s5	2,02%	x17/a7	1,18%	x19/s3	1,53%	x19/s3	1,06%	15	
16	x25/s9	0,08%	x16/a6	0,02%	x21/s5	1,22%	x9/s1	1,85%	x20/s4	1,15%	x16/a6	1,36%	x6/t1	1,03%	16	
17	x20/s4	0,08%	x17/a7	0,00%	x22/s6	1,13%	x19/s3	1,70%	x21/s5	1,07%	x21/s5	1,20%	x21/s5	0,98%	17	
18	x27/s11	0,07%	x4/tp	0,00%	x17/a7	1,07%	x28/t3	1,46%	x22/s6	0,87%	x22/s6	1,16%	x20/s4	0,98%	18	
19	x17/a7	0,06%	x25/s9	0,00%	x23/s7	0,87%	x20/s4	1,43%	x29/t4	0,71%	x24/s8	1,14%	x16/a6	0,67%	19	
20	x24/s8	0,06%	x23/s7	0,00%	x25/s9	0,79%	x26/s10	1,21%	x6/t1	0,70%	x28/t3	0,99%	x31/t6	0,63%	20	
21	x22/s6	0,05%	x22/s6	0,00%	x26/s10	0,69%	x2/sp	1,20%	x25/s9	0,62%	x17/a7	0,89%	x28/t3	0,61%	21	
22	x6/t1	0,05%	x24/s8	0,00%	x24/s8	0,58%	x25/s9	1,08%	x23/s7	0,56%	x3/gp	0,88%	x22/s6	0,57%	22	
23	x28/t3	0,05%	x26/s10	0,00%	x27/s11	0,49%	x18/s2	1,02%	x28/t3	0,51%	x6/t1	0,87%	x5/t0	0,54%	23	
24	x21/s5	0,05%	x27/s11	0,00%	x6/t1	0,33%	x29/t4	0,82%	x24/s8	0,48%	x26/s10	0,81%	x26/s10	0,49%	24	
25	x23/s7	0,03%	x6/t1	0,00%	x28/t3	0,19%	x30/t5	0,79%	x3/gp	0,44%	x29/t4	0,79%	x24/s8	0,48%	25	
26	x29/t4	0,03%	x28/t3	0,00%	x3/gp	0,19%	x22/s6	0,69%	x27/s11	0,34%	x25/s9	0,70%	x25/s9	0,47%	26	
27	x31/t6	0,02%	x29/t4	0,00%	x5/t0	0,18%	x23/s7	0,65%	x26/s10	0,28%	x23/s7	0,67%	x29/t4	0,46%	27	
28	x30/t5	0,02%	x30/t5	0,00%	x31/t6	0,16%	x16/a6	0,52%	x4/tp	0,25%	x31/t6	0,42%	x23/s7	0,38%	28	
29	x7/t2	0,01%	x7/t2	0,00%	x29/t4	0,13%	x24/s8	0,46%	x30/t5	0,22%	x30/t5	0,33%	x3/gp	0,32%	29	
30	x3/gp	0,01%	x5/t0	0,00%	x7/t2	0,07%	x7/t2	0,03%	x31/t6	0,22%	x5/t0	0,29%	x30/t5	0,26%	30	
31	x5/t0	0,01%	x31/t6	0,00%	x4/tp	0,06%	x3/gp	0,00%	x7/t2	0,03%	x7/t2	0,14%	x7/t2	0,05%	31	
32	x4/tp	0,00%	x3/gp	0,00%	x30/t5	0,06%	x4/tp	0,00%	x5/t0	0,02%	x4/tp	0,00%	x4/tp	0,02%	32	

83,59% mapped | **16,41% not mapped**

Table 2: The results of a static and dynamic register usage analysis of several *SPEC CPU 2017* benchmarks, *gzip* as well as a merge sort utility program.

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.

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.

4.4.2. 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, leaving the remaining $16{,}41\,\%$ to read from the register file in memory.

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.

4.4.3. Register file memory access

Continued here…

4.5. Context switching details

4.6. Optimisation of the generated code

4.6.1. Block chaining

4.6.2. Recursive jump translation

4.6.3. Return address stack

4.6.4. Macro operation fusion by pattern matching

4.7. 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 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, as any signals received by the process will be handled by the translator due to the fact that the guest program's execution is emulated in the DBT's process.

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 5.

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 deallocation.

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.

The other supported system calls may be directly passed through to the host after performing the necessary index mapping (see table 5 on the next page).

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. Further context is provided by performance testing using the data compression utility *gzip* [3], 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

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

RISC-V system call	**... index**	**x86-64 index**
`getcwd`	17	→ 79
`fcntl`	25	→ 72
`ioctl`	29	→ 16
`unlinkat`	35	→ 263
`ftruncate`	46	→ 77
`faccessat`	48	→ 269
`chdir`	49	→ 80
`fchmod`	52	→ 91
`fchown`	55	→ 93
`pipe2`	59	→ 293
`openat`	56	→ 257
`close`	57	→ 3
`getdents64`	61	→ 217
`lseek`	62	→ 8
`read`	63	→ 0
`write`	64	→ 1
`writev`	66	→ 20
`readlinkat`	78	→ 267
`utimensat`	88	→ 280
`set_tid_address`	96	→ 218
`futex`	98	→ 202
`set_robust_list`	99	→ 273
`clock_gettime`	113	→ 228
`tgkill`	131	→ 234
`rt_sigprocmask`	135	→ 14
`uname`	160	→ 63
`gettimeofday`	169	→ 96
`getpid`	172	→ 39
`getuid`	174	→ 102
`geteuid`	175	→ 107
`getgid`	176	→ 104
`getegid`	177	→ 108
`gettid`	178	→ 186
`sysinfo`	179	→ 99
`execve`	221	→ 59
`wait4`	260	→ 61
`prlimit64`	261	→ 302
`renameat2`	276	→ 316
`getrandom`	278	→ 318

Table 5: An overview of the system calls handled via pass-through to the host.

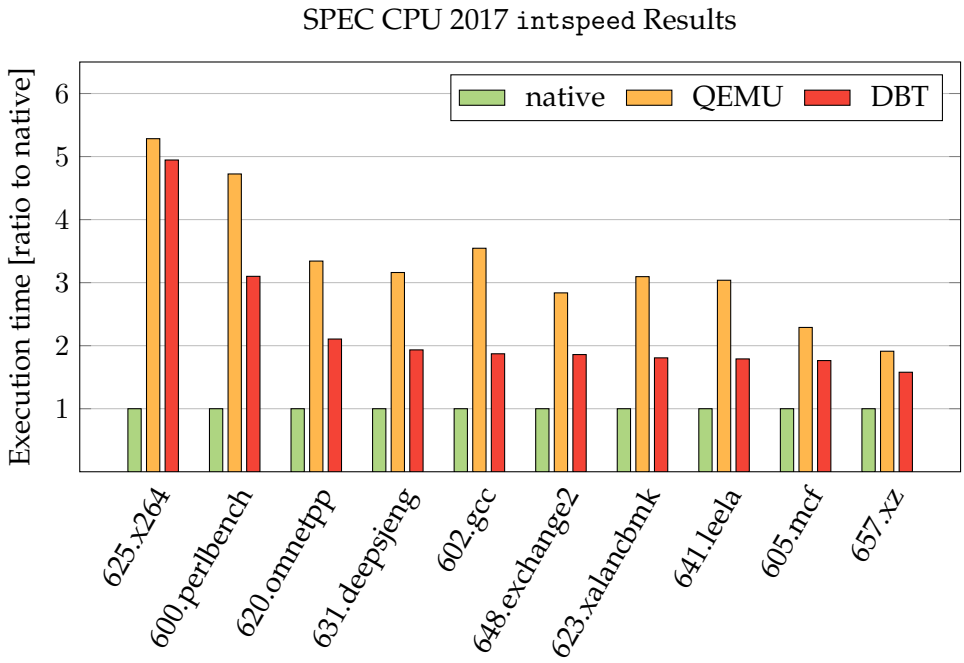

Figure 1: Results of `ref`-workload runs of *SPEC CPU 2017*'s `intspeed` (normalised, lower is better).

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 3 on page 17 lists the execution times of *gzip* compressing a pseudo-random 500 MB file sourced from `/dev/urandom`[3].

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

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`.

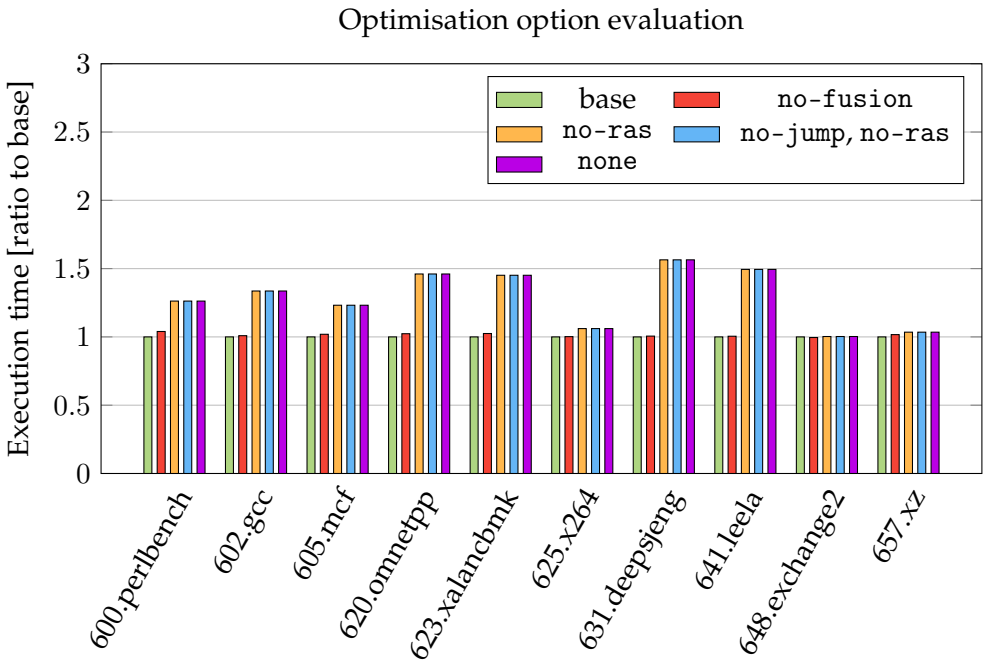

Figure 2: Results of `ref`-workload runs of *SPEC CPU 2017*'s `intspeed` with various optimisation option combinations (normalised, lower is better).

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 1.

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…

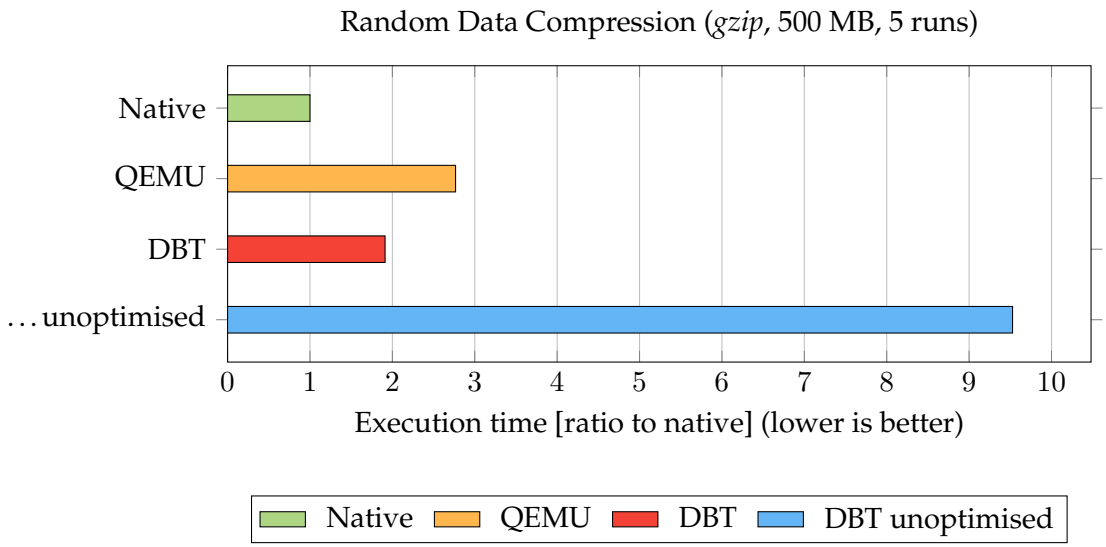

Figure 3: Execution time of gzip file compression (500 MB of random data, 5 runs) in seconds (normalised, lower is better).
Unoptimised run executed with `--optimize=none`.

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[4]. 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[5].

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

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

[5] 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

1. Memory layout 5
2. Register usage analysis results 10
3. Active static register mapping 11
4. Specially handled system calls overview 13
5. General system call overview 14
6. Translator optimisation options 15

List of Figures

1. SPEC CPU 2017 Results 15
2. Translator optimisation evaluation results 16
3. Execution time of gzip compression 17

References

[1] 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.

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

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

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

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