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

The aim of this project is to create an emulator capable of executing code compiled for the RISC-V instruction set architecture on an x86-64 system.

RISC-V is an open ISA first conceptualised in 2010 with the initial goals of research and education in mind. 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].

1.1. Problem description

Because there is as yet no real hardware available for the RISC-V ISA, developers must rely on emulation in order to execute their software on a foreign architecture.

We aim to provide such an 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.

1.2. Comparison of the RISC-V and x86-64 ISAs

Continued here (compare ISAs and note challenges)…

2. Approach

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. 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 two extremes of this spectrum 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.
- **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.

However, it turns out both of these approaches have their downsides. Interpretation, on the one hand, comes with a significant performance penalty. Just as programs written for interpreted languages like Python will always execute more slowly than equivalent programs created with a compiled language like C++, there is a large overhead to having to interpret every single instruction in an executable binary. Especially if the program contains certain parts of the code that are executed many times – bodies of large loop statements, for example – the interpreter has to do a lot of redundant work in simulating the same instruction operations for as many times as that section is called.

Static binary translation, on the other hand, is very difficult to achieve correctly. If done well, it would produce a binary native to the host's environment, and could thus run without any wrapper programs or other utilities, enjoying the same performance benefits as other native executables do.

However, there are certain programming concepts that have proven difficult for a static binary translator to deal with. Register indirect branching, for example, is one of them, where the jump address is not selected statically, but rather computed at runtime. Deducing the jump target would thus again require (in some ways) emulating the program execution in order to know the values needed for the computation. In light of the project at hand, a dynamic binary translator would also be a very difficult program to translate statically, due to the fact that it generates some the code it executes at runtime.

The solution to these issues, and our chosen approach for this project, is

- **Dynamic Binary Translation**, which serves as a middleground 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.

By only translating the parts of the binary that are actually executed, the dynamic binary translator (DBT) can save the overhead a static translator would have spent on translating unused code paths. Also, as the DBT is actually stepping through and executing the program, the other limitations of static binary translators essentially do not apply.

It can also perform significantly better than an interpreter might, as it has to translate a single piece of code only once, which can then be called many times; unlike the above-mentioned behaviour of an interpreter. Furthermore, as it produces blocks of code that are native to the host platform, it can be reasonably expected that a single block's execution will be faster than the respective emulation.

Of course, this assumes that the translation routines of the DBT are relatively swift in performing their functions, so as not to introduce any more overhead than necessary [2, S. 1f.].

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 2.6 on the following page) 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].

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].

2.4. 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 easily 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 1.2 on page 2, 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 1.2) are also of some help here [1, S. 139], as it is clear that an instruction like `xori x10, x10, -1` can be directly translated as a `not x10`, without needing to resort to `mov` and `xor`.

[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 2.7 on page 6.

2.5. 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.

2.6. Register handling and context switching

As outlined in section 1.2, 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 would be an approach that employs parts of both of these extremes [2, S. 9]. We aim to employ the tools we design to discover the most-used

[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.

registers in the guest programs, and statically map these to general purpose x86-64 registers. The remaining operands can either be used from memory directly, or dynamically allocated into free host registers inside a single block's translation. Thus, we save much overhead otherwise spent on memory access to the register file, but do unnecessarily occupy native register space with seldomly accessed guest registers.

2.7. System call handling

3. Implementation Details

4. Results and Performance

5. Summary

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 .. && make
```

in the root directory of the repository. Note that the build requires CMake version 3.16 or above. This will build two artifacts:

translator The actual dynamic binary translator. For details on the usage, see the section below, 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[3]. 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 `rv64ima`, meaning the compiler is free to utilise the base integer instruction set (`i`), as well as the instructions provided by the multiplication (`m`) and atomic standard extensions (`a`). This can be achieved by passing `-march=rv64ima` to gcc[4].

C. Using the translator

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

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

[4] Note that some architecture strings require recompilation of the toolchain. Also, the aforementioned option implies `-mabi=lp64`.

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

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

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 or recursive jump target translation via `--optimize`.

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, December 2019.

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