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. In our case, we are dealing with…

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.

The stack must also 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.

2.3 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 and interrupt/exception handling.

Thus, in our case, a basic block for the purpose of translation will be ended by any dynamic control-flow altering instruction or a system call. Static (or direct) jumps can be followed in translation, as the jump target is known without executing the binary; the resulting block can be seen as an extended basic block covering both the origin and target blocks.

2.4 Translation of the partitioned code

2.5 Code cache and block handling

2.6 Context switching and register handling

2.7 System call handling

3 Implementation Details

4 Results and Performance

5 Summary

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.