peRISCVcope: A Tiny Teaching-Oriented RISC-V Interpreter

Darío Suárez Gracia, Alejandro Valero, Rubén Gran Tejero, María Villarroya, and Víctor Viñals
Department of Computer Science and Systems Engineering
Universidad de Zaragoza
Zaragoza, Spain
{dario,alvabre,rgran,mvg,victor}@unizar.es

***Abstract*—The fast advances of computer systems translate into a growing demand of methodologies and tools to introduce those novelties into classes. Among the plethora of those advances, virtualization has become an essential technology in almost every relevant system stack, from connected cars to hyperscaled cloud servers. However, introducing those technologies into the classroom remains a challenging task because of the huge complexity of their software components that may hinder the learning process of students. peRISCVcope aims to help in this area by proposing a tiny yet powerful interpreter to dig into virtualization technologies, such as the implementation of trap&emulate hypervisors.**

With less than 2,000 lines of code, and thanks to the conciseness of the RV32I base instruction set of RISC-V, peRISCVcope enables students to make virtualization knowledge their own. This paper presents our experiences developing and testing a virtualization laboratory where students implement parts of an interpreter. After the practical experience, peRISCVcope has been proved as a useful pedagogical tool, and, most importantly, students have positively rated the experience.

***Index Terms*—Education, interpreter, RISC-V, virtualization.**

I. Introduction

Teaching everyday computing technologies requires the combination of multiple class activities to ensure the learning progress of the students. Laboratories are one key activity where students implement theoretical concepts with adapted tools. There exist a deluge of tools and frameworks for many topics, including programming and digital design. However, for certain advanced topics such as virtualization, the pool of available tools is very scarce. As a result, students often lack hands-on experiences on how binary interpreters and hypervisors work.

This paper presents peRISCVcope, a work-in-progress RISC-V pedagogical interpreter aimed at teaching virtualization and computer architecture concepts. Its main goal is to provide a simple yet powerful environment where teaching implications take precedence over performance in all design decisions.

RISC-V is an open Instruction Set Architecture (ISA) designed at the University of California, Berkeley [14]. While originally designed for research and education, RISC-V is currently evolving into a contender to the mayor players, ARM and AMD64/x86-64, as a general-purpose ISA in almost every application domain from embedded to high-performance computing. RISC-V offers an extensible architecture with customization capabilities that no other ISA provides. In addition, public and private investments are fueling its advancement, which, in turn, will soon require a large workforce for its development.

For students, knowing RISC-V is a must-have skill, and, from the pedagogical point of view, RISC-V offers an orthogonal and small base set of instructions, RV32I, with an excellent trade-off between capabilities and complexity. As a consequence, RISC-V attracts the interest of students and can help making assignments more appealing to them.

Virtualization has established itself as a key technology in almost every computing segment; e.g., automotive or cloud computing. However, from a teaching point of view, virtualization presents many challenges because it requires knowledge from operating systems, ISAs, and so forth. In addition, it demands strong abstraction capabilities of students to understand how a software layer presents a virtual device; e.g., a processor to another software such as an operating system [9].

This work presents a learning-by-doing experience focused on specific laboratories that implement some of the key aspects of hypervisors and interpreters. Designing these laboratories has required to propose the peRISCVcope pedagogical interpreter, which is powerful enough for students to experiment many of the real issues of hypervisors and interpreters. In addition, peRISCVcope streamlines all the side aspects of complex software projects. To assist students, it also provides build support and a set of compiled RISC-V binaries for testing.

Our initial experiences with peRISCVcope have been successful, and, after three 2-hour laboratory sessions, students were able to complete a RV32I interpreter.

The rest of this paper is organized as follows. Section II introduces the academic environment of the project. Section III surveys the related work. Section IV describes the proposed peRISCVcope interpreter. Section V presents the proposed laboratories and students feedback, and finally, Section VI concludes the article and discusses future directions.

II. Academic Context and Rationale

This section describes the contents of the two main courses where the proposed peRISCVcope interpreter is expected to be used.

978-1-6654-5950-1/22/$31.00

The University of Zaragoza offers all levels of computer science and engineering studies, from a common 4-year bachelor's degree in Computer Science and Engineering (CSE) to a specific master's degree in Robotics, Graphics, and Computer Vision (RGCV) together with PhD training.

During the last semester of the CSE program, students take a mandatory course on resilience and security, which consists of six ECTS[1]. These credits correspond to an average of 150 student working hours, and, in the University of Zaragoza, to 60 face-to-face class hours, including lectures, problems, and laboratory work. In this course, students dive into the details on the internals of a virtual machine. The syllabus includes topics from the implementation of the required software to virtualize a machine, to the hardware extensions that enhance virtualization. While Smith and Nair's book on virtual machines covers most of the theoretical course contents [11], there is a lack of a simple tool that assists students in hands-on labs. Therefore, using a tool with a shallow learning curve is an appealing methodology to strengthen virtualization concepts.

The RGCV Master is a completely different program that aims to provide specific education in research, innovation, and development on robotics, graphics, and computer vision. During the first semester of the degree, students take a mandatory course on programming and architecture of computing systems, organized in the same way as the previous CSE course with six ECTS credits. In this course, students learn advanced concepts on computer architecture and gain some parallel programming skills with the C++ programming language. From the teaching perspective, the main challenge of this course is to accommodate students with diverse backgrounds, including Computer Science, Mathematics, Telecommunications Engineering, Physics, or Electrical Engineering. Since the schedule of the course is very tight, activities that combine learning of different topics such as programming and computer architectures are very appealing. One of these activities that could combine multiple topics could be crafting an interpreter. Students would be exposed to the details of an instruction set while they write code, establishing a relationship between computer architecture and programming skills.

III. Related Work

There are plenty of tools to interpret, simulate, and/or virtualize the RISC-V architecture. For example, rv8 provides a suite of tools, including a RISC-V to x86-64 binary translator, a user-mode simulator, and a full-system emulator [2]. The eXtensible Versatile hypervISOR, XVISOR, provides a pure monolithic hypervisor oriented toward embedded virtualization [7, 6]. For some years, the standard Linux virtualization software stack of QEMU/KVM provides RISC-V support [3].

However, none of the above tools are teaching oriented teaching. They include a large base of source code, usually with complex patterns, and tend to prioritize performance over code readability when required. Therefore, they are not suited for an academic laboratory. On the contrary, Palicherla *et al.* offer an excellent tool to learn how to build a hypervisor following a hands-on approach with several labs. They propose HOSS, a small OS that can virtualize itself [5]. HOSS is based on MIT's JOS teaching-oriented OS[2] and runs on x86 and AMD64 architectures. Ideally, to strengthen learning, students should use JOS during their studies on operating systems, and, later on in advanced courses, HOSS to learn virtual machines and interpreters, imposing a strong requirement between courses. peRISCVcope aims to avoid having any previous requirement at the cost of providing less architectural details than HOSS. In addition, HOSS does not provide support for the RISC-V architecture.

Other authors have focused on providing sources to present a vertical approach into systems, from binary data to parallelism, but excluding virtualization [8, 4]. Often, virtualization concepts only appear as short chapters or appendices in many traditional operating system books [10, 12, 1].

IV. Proposed peRISCVcope Architecture and Implementation

This section describes the internals and taken implementation decisions during the development of the current version of peRISCVcope.

RISC-V is a highly modular architecture. Originally designed for research and education, it is currently evolving toward becoming a competitor to the dominant ARM and x86/x86-64 architectures. The original education goal facilitates its adoption as a reference ISA for developing interpreters and virtual machines oriented to teaching. In addition, the fact that the ISA is fully virtualizable also helped in its choice for this project.

Within RISC-V modules, and, for the sake of simplicity, peRISCVcope only targets the version 2.1 of the RV32I Base Integer Instruction Set, which was ratified in 2019. Since RV32I has only 40 unique instructions, which were selected as a minimum base to support a modern operating system, we found this reduced set the perfect substrate for implementing a teaching oriented interpreter. Furthermore, there is a strong RISC-V tool-chain support for both bare-metal and Linux systems, ensuring the ability to create binaries and the availability of tools like `readelf` or `objdump` to assist the programmer during the development process.

While the choice of RISC-V was clear since the very beginning of this project, selecting the proper programming language took several iterations. C, C++, and Rust made the short list of candidates. The safe memory guarantees of Rust were very appealing, but the fact that students have not been previously exposed to such a language removed Rust from the list of candidates. Between C and C++, the latter was chosen for several reasons. First, in the master course (see Section II), students already use C++ for many labs because this programming language has been embracing concurrency and parallelism since its C++11 version. Second,

[1]ECTS refers to European Credit Transfer and accumulation System: http://ec.europa.eu/education/resources-and-tools/european-credit-transfer-and-accumulation-system-ects_en.

[2]https://pdos.csail.mit.edu/6.828/2018/overview.html

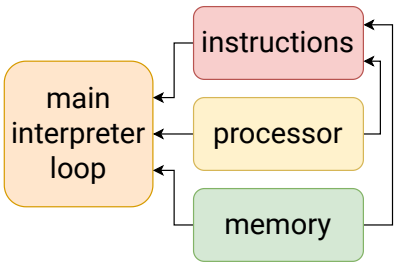

Fig. 1. Principal peRISCVcope software modules. The arrows represents the dependencies among them. The right modules correspond to the virtual machine state and interpretation routines, while on the left side, the interpreter loop fetches, decodes, and executes the instructions with the help from other modules.

undergraduate students experience with C during many courses of the computer engineering program, so having more practice with C++ would strengthen their object-oriented programming skills and help learning how to perform low-level programming with a high-level language. Last but not least, object-oriented programming offers a straightforward mapping between the virtualized elements and their code.

peRISCVcope design is based on simplicity to support student learning. One of the first design decisions was excluding input/output virtualization to simplify the implementation, since it eliminated many asynchronous operations. We think Virtual I/O, although important in any system, can be left out for a first contact with virtual machine development.

Figure 1 presents the main software modules, which correspond to the main virtualized resources: processor and memory. First, the processor module stores the main architectural state of the processor, including all the registers of the system. Second, the memory module provides two main capabilities: i) supporting binary load into the virtualized memory, and ii) accessing the virtualized memory, which requires the translation from the binary physical to the interpreter virtual address spaces. Third, the instructions module deals with instruction decoding and semantics. This division would facilitate the interpretation of other architectures, if required, because the memory module could remain almost unchanged. Finally, the main module contains the main function of the executable and the main interpreter loop.

Listing 1. Main interpreter loop.

```
1  size_t exec_instrs{0};
2  address_t pc{0x0}, next_pc{0x0};
3  do {
4  // fetch
5  pc = proc.read_pc();
6
7  // decode
8  uint32_t raw_instr{mem.read<uint32_t>(pc)};
9  instruction instr{raw_instr};
10
11 // dispatch & execute
12 next_pc = dispatch[instr.opcode()]
13       (mem, proc, raw_instr);
14 exec_instrs++;
15 // look for while(1) in the code
16 } while (next_pc != pc);
```

Listing 1 shows a code snippet of the main interpreter loop with the fetch, decode, and dispatch & execute stages in just a few lines. This conciseness was a goal to help students from non computer engineering degrees, in the master course, to understand how a processor operates and match the *function blocks* from theory classes with the hands-on labs. For example, the `mem` and `proc` objects correspond to the memory and processor modules from Figure 1, and the dispatch table, line 12, calls the corresponding interpreter function of each instruction from the instruction module. In this particular implementation, a `std::map` associative container with the opcode of the instruction as the key and `std::function` target as the values perform the actual dispatch. This implementation avoids the use of a virtual table in the base instruction class.

The tiny footprint of peRISCVcope, around 2,000 lines of code, has some important limitations; e.g., the current version does not provide memory protection mechanisms, or proper support for system calls. However, since the software architecture is easily extensible, we plan to add these features in the future.

Lastly, working with peRISCVcope only requires the `cmake` building tool, so students can build the interpreter on many operating systems, assuming the system includes the `elf.h` header. Also, there are no dependencies on external libraries. To compile the tool, the only requirement is the support of C++14 that includes binary literals, simplifying the decoding programming. Finally, peRISCVcope accepts RV32I binary programs compiled with the standard GNU gcc compilation tool-chain in bare-metal mode. The source code is available upon request, and to ease adoption, peRISCVcope has adopted a non-restrictive MIT license.

V. Example of a Virtualization Laboratory

For the sake of brevity, we only describe a single proposed laboratory using peRISCVcope for the resilience and security course and do not include the ongoing materials for the RGCV master course.

The proposed virtualization laboratory comprises three 2-hour sessions. In lectures, students learn how a trap&emulate hypervisor works, without digging into the emulation details, which are covered in this laboratory.

The first session deals with the loading of an Executable and Linkable Format (ELF) binary into memory. The second session describes how to decode instructions, whereas the third session completes the implementation of the interpreter. Besides the description of the lab, in all three sessions, students receive a skeleton of the code with the missing parts that they have to implement.

The following subsections summarize each laboratory session. Finally, this section concludes with a qualitative assessment of the proposed laboratory.

A. Session 1: Loading a Binary

In many Spanish CSE curricula, ELF binaries are often overlooked, and students do not experience how a loader and a linker operate. Fortunately, crafting an interpreter requires to

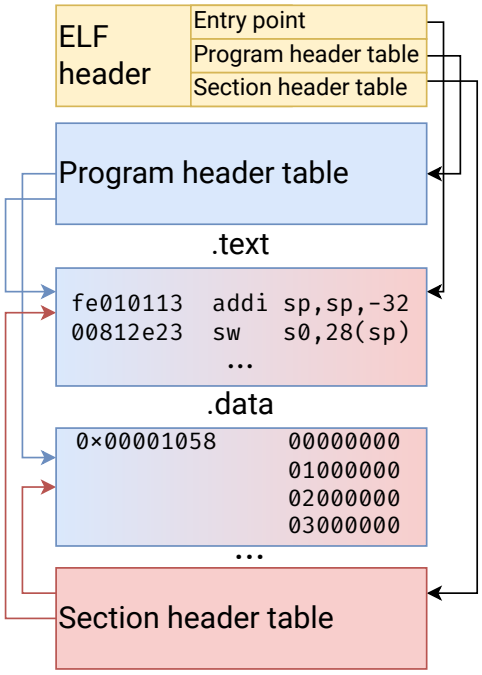

Fig. 2. Simplified ELF binary layout for the array reduction program example.

load a binary into the interpreter memory, so this experience provides an excellent opportunity to introduce students to the process of loading a binary.

Figure 2 shows a simplified view of an ELF binary. Namely, the content inside the `.text` and `.data` sections corresponds to the array reduction code of Listing 2. The first bytes of an ELF binary, top of the figure, correspond to the ELF header that, among others, contains pointers to the following elements: the entry point (first instruction to execute), the program header table, which is an array storing all the information required by the loader, and the section header table, storing all the information required by the linker. In other words, Program and Section header tables provide all the metadata required by the loader and the linker, respectively. In our example, the loader will copy into memory two binary sections: i) `.text`, which contains all the instructions of the program[3], and ii) `.data`, which contains all the global variables not initialized to zero; i.e., the 4-element `array` vector with values 0, 1, 2, and 3.

Understanding the meta-information encoded in a binary for both the loader and the linker can be challenging for many students. In this way, before starting the implementation of this part, reading the ELF man page[4], which contains all the required information, should be a mandatory prerequisite. To reduce the effort of this laboratory session, students focus on the loader meta-information, leaving aside the linker, and receive the code that reads the binary from the file system and stores the content into an array of bytes. Therefore, students only have

[3]The real binary encoding is shown in the left side of the `.text` box in Figure 2, whereas the assembly code is shown in the right side.

[4]https://man7.org/linux/man-pages/man5/elf.5.html

Listing 2. Example of a test program: array reduction.

```
1  enum { N = 4 };
2  int array[N] = {0, 1, 2, 3};
3
4  int main()
5  {
6    int res = 0;
7
8    for(size_t i = 0; i < N; ++i) {
9      res+=array[i];
10   }
11   // signaling perRISCVcope termination
12   while(1);
13 }
```

to traverse the program header table and write into memory the loadable segments. Internally, the virtualized memory also stores the initial virtual address where the segment should reside in memory, which will be different from the actual position in the system where the interpreter runs. In addition, students experience that the ELF binary entry point corresponds to the first executed instruction.

To validate this activity, students also have to write a hexadecimal dump method that prints the content of a segment (e.g., `.text`) and verify that is the same as in the original binary obtained with the `objdump` utility.

B. Session 2: Decoding Instructions

Decoding instructions is a challenging task for the students. Programmatically, decoding requires to study the RISC-V ISA manual and understand the format of the six instruction types (r, i, s, b, u, and j). To simplify this lab, students receive code with a base class representing any instruction and a derived class of the r-type. Thus, students have some reference code to minimize the chances of proposing an instruction representation that later on hinders the interpretation. In addition, this implementation helps strengthen object-oriented programming concepts because of the use of inheritance. Notice too that this representation assumes an indirect threaded interpreter [13], but advanced students can extend the code to support more sophisticated versions such as predecoding.

From the practical experience, many students have difficulties handling immediate values, which in turn causes many bugs; e.g., understanding the ISA rules for extending the precision of a number by adding zeros or replicating the sign, depending on whether it is unsigned or two's complement integer. In this session, the online RISC-V interpreter from the University of Cornell can help debugging the content of the registers and the memory [15].

Finally, in this activity, students begin to figure out the importance of how to store the architectural processor state, which is one of the main goals of the next laboratory session.

C. Session 3: Interpreting Instructions

Implementing the actual operations encoded in the instructions requires to perform the computations and correctly update the processor and the memory state. At this stage, students connect all the principal modules of peRISCVcope (memory,

instructions, and processor) with the main interpreter loop. Basically, they have to read one-by-one all the 4-byte instructions. And, for each one, find out its type with the help of the code from the previous lab, and call the suitable interpreting method for the decoded instructions, which in turn will update the processor and/or memory state. At this step, students grasp how the function fields operate and identify, for example, the different ALU operations. Also, they need to figure out how to advance the pc register value, specially for branch and jump instructions. Also, it is worth to mention that the last step in this session consists on writing an equivalent code of Listing 1.

Since the session only lasts for two hours, the provided code includes the implementation of load and store instructions. Otherwise, students would have some difficulties completing the task in time.

To test their implementation, the repository includes toy RV32I compiled programs; e.g., reduce the elements of an array into a scalar variable as shown in Listing 2. For the sake of simplicity, and due to the lack of system calls and operating system support, peRISCVcope assumes that a `while(1)` statement at the end of the program represents the termination condition and ends its execution.

D. Students Survey

Before and after the completion of the lab, students filled an optional form with questions with the aim to evaluate the quality of the proposed laboratory. The number of participants was limited to 11 because there were only 12 students enrolled in the course. Such a small population does not enable to provide very general conclusions. Nevertheless, in general, the response was very positive, and students appreciated the lab. For example, more than 60% of the participants fully agree that the laboratories helped them to understand how interpreters and hypervisors operate. Also, students consider that after completing the lab, their knowledge on the RISC-V architecture has improved.

Most importantly, the forms have provided specific information on aspects that could be improved:

- The use of modern C++ features like polymorphic function wrappers was not clear, and the laboratory could offer additional materials on C++ to ease the task. On the other hand, more than 60% of the students considered that as a side effect, the laboratory improved their C++ coding skills.
- All but one student provided an affirmative answer when asked if the interpreter run in a different architecture than RISC-V, even implementing and running peRISCVcope on AMD64 computers.
- Surprisingly, when asked how well they understand how a hypervisor stores the state of the machine before and after the lab, the responses were very similar with an average score of 2.6 over 5.

According to the received feedback and the overall good impression of the results, we plan to continue the development of peRISCVcope to support learning about interpreters and virtualization with the RISC-V architecture.

VI. Conclusions and Future Work

This paper proposes peRISCVcope, a learning-by-doing experience to study interpreters, hypervisors, and RISC-V computer architecture. The presented approach is a student-friendly tool that favors simplicity over performance and has been validated in an advanced Computer Science and Engineering undergraduate course on virtual machines.

For interpreters and virtual machines, peRISCVcope helped students to understand several system key aspects such as how a loader operates, how instruction decoding works, or how modern C++ techniques simplify the implementation of interpreters.

Currently, we are extending peRISCVcope for a programming and architecture course belonging to a master study on Robotics, Graphics, and Computer Vision. Our future work directions include: i) providing some extra C++ exercises before coding the interpreter in the virtual machine laboratories, ii) integrating dual instructions into the standard compiler work-flow, so students can implement its support within the interpreter and implement the H virtualization extension, and iii) provide a mature debugging environment to support the learning of the RISC-V architecture, which is key for the master course. All the assignments and the source code are available at https://github.com/dariosg/periscvcope, and solutions are available upon request.

Acknowledgments

All authors acknowledge support from grants (1) PID2019-105660RB-C21/AEI/10.13039/501100011033 from *Agencia Estatal de Investigación* (AEI), and (2) gaZ: T5820R research group from Dept. of Science, University and Knowledge Society, Government of Aragon. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018. ISBN: 198508659X.

[2] Michael J. Clark. *RISC-V simulator for x86-64*. URL: https : / / github . com / michaeljclark / rv8. (accessed: 05.28.2022).

[3] Sagar Karandikar. "QEMU Support for the RISC-V Instruction Set Architecture". In: *KVM Forum*. 2016.

[4] Suzanne J. Matthews, Tia Newhall, and Kevin C. Webb. "Dive into Systems: A Free, Online Textbook for Introducing Computer Systems". In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1110–1116. ISBN: 9781450380621. DOI: 10.1145/3408877.3432514. URL: https://doi.org/10.1145/3408877.3432514.

[5] Abhinand Palicherla, Tao Zhang, and Donald E. Porter. "Teaching Virtualization by Building a Hypervisor". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: Association for Computing Machinery, 2015, pp. 424–429. ISBN: 9781450329668. DOI: 10.1145/2676723.2677254. URL: https://doi.org/10.1145/2676723.2677254.

[6] Anup Patel. "Xvisor: Embedded Hypervisor for RISC-V". In: *Open Source Summit Europe*. 2019.

[7] Anup Patel et al. "Embedded Hypervisor Xvisor: A Comparative Analysis". In: *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015, pp. 682–691. DOI: 10.1109/PDP.2015.108.

[8] Yale N. Patt and Sanjay J. Patel. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. 2nd. McGraw-Hill, 2004. ISBN: 978-0-07-246750-5.

[9] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Communications of the ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[10] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 10th. Wiley Publishing, 2018. ISBN: 9781119320913.

[11] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[12] William Stallings. *Operating Systems: Internals and Design Principles*. 9th. Person Education Limited, 2017. ISBN: 978-0134670959.

[13] Nguyen T. Thanh and E. Walter Raschner. "Indirect Threaded Code Used to Emulate a Virtual Machine". In: *SIGPLAN Not.* 17.5 (May 1982), pp. 80–89. ISSN: 0362-1340. DOI: 10.1145/947923.947932. URL: https://doi.org/10.1145/947923.947932.

[14] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual. Volume I: Unprivileded ISA*. Tech. rep. University of California, Berkeley, 2019.

[15] Hakim Weatherspoon. *RISC-V Interpreter*. URL: https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/. (accessed: 05.28.2022).