Towards a High-Performance RISC-V Emulator

Leandro Lupori[1], **Vanderson Martins do Rosario**[1], **Edson Borin**[1]

[1]Institute of Computing – UNICAMP – Campinas – SP – Brasil

`leandro.lupori@gmail.com, {vanderson.rosario, edson}@ic.unicamp.br`

Abstract. *RISC-V is an open ISA which has been calling the attention worldwide by its fast growth and adoption, it is already supported by GCC, Clang and the Linux Kernel. Moreover, several emulators and simulators for RISC-V have arisen recently. However, none of them with good performance. In this paper, we investigate if faster emulators for RISC-V could be created. As the most common and also the fastest technique to implement an emulator, Dynamic Binary Translation (DBT), depends directly on good translation quality to achieve good performance, we investigate if a high-quality translation of RISC-V binaries is feasible. To this, we used Static Binary Translation (SBT) to test the quality that can be achieved by translating RISC-V to x86 and ARM. Our experimental results indicate that our SBT is able to produce high-quality code when translating RISC-V binaries to x86 and ARM, achieving only 12%/35% of overhead when compared to native x86/ARM code. A better result than well-known RISC-V DBT engines such as RV8 or QEMU. Since DBTs have its performance strongly related with translation quality, our SBT engine evidence the opportunity towards the creation of RISC-V DBT emulators with higher performance than the current ones.*

1. Introduction

RISC-V is a new, open and free ISA initially developed by the University of California [Waterman et al. 2014] and now maintained by the RISC-V foundation [RV-foundation 2018a] with a handful of companies supporting its development. It is a small RISC-based architecture divided in multiples modules that support floating-point computation, vector operations, and atomic operations, each one focusing on different future computing targets such as IoT embedded devices and cloud servers. RISC-V is calling attention worldwide by its fast growth and adoption. By now, it is supported by the Linux Kernel, GCC, Clang, not to mention several RISC-V simulators [Ta et al. 2018, Ilbeyi et al. 2016] and emulators [Clark and Hoult 2017, Bartholomew 2018]. However, at the current time, all emulators for RISC-V suffer from poor performance.

Having a high-performance RISC-V emulator for common architectures, i.e. x86 and ARM, would not only facilitate its adoption and testing but also would show it as a useful virtual architecture to ease software deployment. One approach to implement a high-performance emulator is by using Dynamic Binary Translation (DBT)

Acknowledgements: the authors would like to thank CAPES (PROEX - 0487/2018), CNPq, Microsoft and Instituto de Pesquisas Eldorado for their financial support and LMCAD for its infrastructure and technical support.

[Smith and Nair 2005], a technique that selects and translates regions of code dynamically during the emulation. This technique has been used to implement fast virtual machines (VMs), simulators, debuggers, and high-level language VMs. For example, it has been used to facilitate the adoption of new processors and architectures, such as the Apple's PowerPC to x86 migration Rosetta software [Apple 2006], to enable changes in microarchitecture without changing the architecture itself, as with the Transmeta VLIW processor [Dehnert et al. 2003] that implements x86, or in the deployment of high-level languages in several platforms such as with the Java VM [Häubl and Mössenböck 2011]. A DBT engine usually starts by interpreting the code and then, after heating (translating all hot regions), it spends most of the time executing translated regions instead. Thus, the quality and performance of these translated regions are responsible for most of the DBT engine performance [Borin and Wu 2009] and there are two DBT design choices which affect most of the quality of translation: (1) the DBT's Region Formation Technique (RFT) which defines the shape of the translation units [Smith and Nair 2005] and (2) the characteristics of the guest and host ISA which can difficult or easy the translation [Auler and Borin 2017]

While RFT design choice is well explored in the literature, the translation quality of each pair of guest and host ISA needs to be researched and retested for every new ISA. One approach to understanding the quality and difficulty of code translation for a pair of ISAs is by implementing a Static Binary Translation (SBT) engine [Auler and Borin 2017]. SBTs are limited in the sense that they are not capable of emulating self-modifying code and may have difficulty differentiating between data and code, however, its design and implementation is usually much simpler than a DBT. Since the translation mechanisms in a DBT and an SBT are very similar and creating an SBT engine which can emit high-quality code implies that the same could be done with a DBT engine, we implement and evaluate a LLVM-based SBT to investigate whether or not it is possible to produce high-quality translations from RISC-V to x86 and ARM. Our SBT is capable of producing high-quality translations, that execute almost as fast as native code, with only 1.21x/1.39x slowdown in x86/ARM. These results suggest that it is possible to design and implement a high-performance DBT to emulate RISC-V code on x86 and ARM platforms. The main contributions of this paper are the following:

- We show that it is possible to perform a high-quality translation of RISC-V binaries to x86 and ARM.
- We compare the performance of our SBT with the performance of state-of-the-art RISC-V emulators and argue that there is a lot of room for performance improvement on RISC-V emulators.

The rest of this paper is organized as follows. Section 2 further describes DBT and SBT techniques, the challenges to implement both and discusses ISA characteristics that are difficult to translate. Section 3 describes the main characteristics of the RISC-V ISA and Section 4 presents other emulators for RISC-V. Then, in Sections 6 and 7 we discuss our SBT for RISC-V and the results we have obtained with it. Finally, Section 8 presents our future work and conclusion.

2. Binary Translation and Challenges

Interpretation, SBT and DBT are well known methods used to implement ISA emulators. Interpretation is a technique that relies on a fetch-decode-execute loop that mimics the behavior of a simple CPU, a straightforward approach. Nonetheless, it usually requires the execution of tens (or hundreds) of native instructions to emulate each guest instruction. On the other hand, dynamic and static translators translate (maps) pieces of guest code into host code and usually obtain greater performance with the cost of being more complex and hard to implement. Because of this, DBT is commonly used on high-performance emulators, such as QEMU [Bellard 2005]. A DBT engine uses two mechanisms to emulate the execution of a binary, one with a fast-start but slow-execution and another with a fast-execution but a slow-start. The former is used to emulate cold (seldom executed) parts of the binary, normally implemented using an interpreter. The latter is used to emulate hot (frequently executed) parts of the code by translating the region of code and executing it natively. A translated region of code normally executes more than 10x faster than an interpreter [Böhm et al. 2011]. It is important to notice that the costs associated with the translation process impacts directly on the final emulation time. As a consequence, DBTs usually employ region formation techniques (RFTs) that try to form and translate only regions of code that the execution speedup (compared to interpretation) pays off the translation time cost.

The majority of a program execution is spent in small portions of code [Smith and Nair 2005], thus DBT engines spend most of their time executing small portions of translated code. Therefore, the quality of these translated regions is crucial to the final performance of a DBT engine. In fact, this is evidenced by the low overhead of Same-ISAs DBT engines [Borin and Wu 2009], also known as binary optimizers, as they always execute code with the same or better quality than the native binary (this happen because same-ISA do not actually impose translations, but only optimizations). Designing and implementing high-performance Cross-ISA DBT engines, on the other hand, is more challenging because the performance of translated code depends heavily on the characteristics of the guest (source) and the host (target) ISA. For instance, the ARM has a conditional execution mechanism that enables instructions to be conditionally executed depending on the state of the status register, however, since x86 does not have this feature, it may require several instructions to mimic this behavior in x86 [Salgado et al. 2017]. Experience has shown that emulating a guest-ISA which is simpler than the target-ISA is normally easier [Auler and Borin 2017].

In the end, implementing a high-performance DBT would be the final proof of concept of whether an ISA is simple to translate, but implementing a DBT is a complex project and a challenge by itself to construct. Another possibility, which we use in this paper, is to implement an SBT engine to translate the binary. An SBT engine translates statically the whole binary at once. SBT is not usually used to emulate binaries in industry, despite being easier to implement than a DBT engine, because an SBT cannot execute all kind of applications. Self-modifying code, code discovery problem and indirect branches, are some problems that cannot be addressed statically [Cifuentes and Malhotra 1996]. However, for the purpose of testing the difficulty of translating code with high-quality, an SBT is enough. This same approach was used by Auler and Borin [Auler and Borin 2017] to test the OpenISA emulation performance.

3. RISC-V

In terms of ISA design, RISC-V is reaching a mature and stable state only by now [Waterman et al. 2014]. RISC-V was developed in 2010, but the user-level ISA base and extensions MAFDQ (Multiply/divide, Atomic, single-precision Floating-point, Double-precision floating-point and Quadruple-precision floating-point: the main standard extensions) were frozen only in 2014 [RV-foundation 2018a]. For the privileged ISA, at the time of this writing, it is still a draft, albeit at an advanced stage. For the physical implementations, there are several open-sourced RISC-V CPU designs available. While these open-sourced designs are a great step towards having plenty RISC-V CPU chips available, that is not the case for now, given the research/experimental state of such CPU designs. Therefore, at the present time, there are few platforms that implement the RISC-V architecture. It usually takes some time until hardware implementing a new ISA becomes widely available. Until then, emulation plays a crucial role, because it enables the use of a new ISA while there are no (or few) physical CPUs available for it. The main job of an ISA emulator is to translate guest instructions to host instructions, with the goal of making the host perform a computational work that is close enough to what would be achieved by the guest instructions being executed on the guest platform. As the majority of available hardware is either X86 or ARM based, targeting these ISAs as host platforms for RISC-V emulators seems reasonable. While there are already some RISC-V emulators available for those, e.g., Spike and QEMU, they are unable to achieve near-native performance — as shall be discussed in the next section — which limits their scope by excluding them from use cases where performance plays a major role. Thus, this work aims at designing mechanisms for fast RISC-V emulation. It shows the viability of running RISC-V applications with near-native performance on X86 and ARM-based processors, proving that RISC-V is an easy to translate ISA, at least to these architectures.

4. Related Work

According to Auler and Borin [Auler and Borin 2017], it is possible to achieve near-native performance in cross-ISA emulation if the guest architecture is easy to be emulated. They showed this to be possible with OpenISA, an ISA based on MIPS but modified with emulation performance in mind. Using SBT to emulate OpenISA on X86 and ARM, they were able to achieve less than 1.2x of slowdown on most benchmarks. RISC-V has most of the characteristics pointed by the authors to be easy to emulate: it is simple, it hardly uses the status register and it has a small number of instructions, with 66% of them being equivalent to the OpenISA instructions, all indicating that RISC-V is also an easy to emulate ISA. For same-ISA emulation, the best performance achieved is from StarDBT by Borin and Wu [Borin and Wu 2009]: 1.09x slower than native (x86) emulation.

Spike [RV-foundation 2018b], a RISC-V ISA simulator, is considered by the RISC-V Foundation to be their “golden standard” in terms of emulation correctness. As expected from an interpreted simulator, its performance is not very high, although quite higher than other simulators in some cases, varying from 15 to 75 times slower than native on SPECINT2006 benchmarks. This performance is due to several DBT-like optimizations, such as instruction cache, software TLB, and unrolled PC-indexed interpreter loop to improve host branch prediction. ANGEL [RV-foundation 2018b] is a Javascript RISC-V ISA (RV64) Simulator that runs riscv-linux with BusyBox. Our simple run achieved $\approx$ 10 MIPS in Chrome, on an Intel Core i7-2630QM CPU running at 2.0GHz, or about 200 times slower than native.

In the case of RISC-V DBT engines, B. Ilbeyi *et al.* [Ilbeyi et al. 2016] showed that Pydgin can achieve better performance than Spike, by means of more sophisticated techniques, mainly, DBT. Pydgin is able to achieve a performance between 3.3x to 4x slower than native. Clark and Hoult [Clark and Hoult 2017] presented the RV8 emulator a RISC-V high-performance DBT for x86. Using optimizations such as macro-op fusion and trace formation and merge, RV8 is able to achieve a performance 3.16x slower than native. QEMU [Bellard 2005], a famous DBT with multiple sources and targets, also gained support for RISC-V. QEMU is 7x slower than a native execution and one of its main performance disadvantages comes from floating-point emulation, as its Intermediate Representation (IR) does not have any instruction of that kind and it needs to simulate them by calling auxiliary functions.

5. RISC-V SBT

Static translation of a RISC-V binary into a native binary for other architecture, such as ARM and x86, involves several steps. As mentioned earlier, the goal is to translate RISC-V machine instructions into machine instructions for another ISA, that performs equivalent computational work. Modern compilers, such as LLVM/Clang, compile from source to machine code in several, independent, steps. It helps to modularize, organize and decouple the parts that compose a compiler. That is why we chose to take advantage of an existing modular compiler: LLVM 7.0. This enabled us to reuse several parts of it, write only the missing parts and assemble everything to perform the complete translation.

Our Static Binary Translator starts by reading the RISC-V executable file and disassembling each instruction in it, with the help of LLVM libraries. Then, for each RISC-V instruction, our translator emits equivalent, target independent, LLVM Intermediate Representation (IR) instructions or bitcode. This is practically the same as the IR produced by Clang after compiling a source code file. After that, the produced LLVM IR is written to a file, concluding the first translation stage.

The remaining steps are performed with existing software. LLVM tools are used to optimize the IR and to generate assembly code for x86 or ARM. After that, a standard assembler and linker for the target platform, such as GNU as and ld, can be used to produce the native binary for the host architecture. The code generation flows used in our experiments are further detailed in Section 6. Finally, all these steps for SBT are summarized in the diagram from Figure 1.

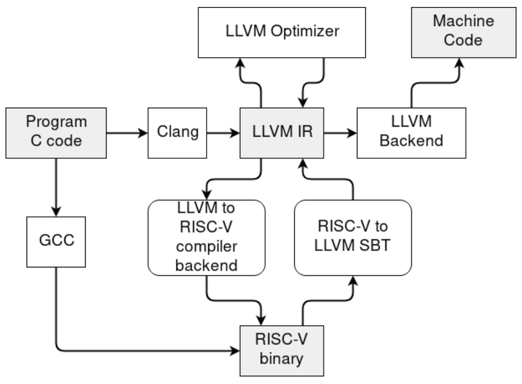


Figure 1. Our RISC-V SBT Architecture.

5.1. Unlinked Objects as Input

Instead of translating final linked binaries, we choose to translated object binaries before linking to avoid having to deal with some issues. It enables us to translate only the benchmark code, leaving C runtime out. In this way, we avoid C runtime translation quality from interfering in benchmark performance measurements and also save a considerable amount of work that would be required if the SBT needed to be able to translate all C runtime libraries.

However, with this approach, the translator must now be able to identify C library calls in guest code and forward these to the corresponding ones on native code. This was done by listing all C functions needed by the benchmarks we used, together with their types and arguments and then, at the call site, copying RISC-V registers corresponding to arguments to the appropriate host arguments' locations, as defined by their ABIs.

5.2. Register Mapping

Regarding register mapping between architectures during the translation, our SBT implements two techniques:

- **Globals** – RISC-V registers are translated to global variables. The main advantages of this approach is that it is simple and it does not need any kind of inter-function synchronization. The main disadvantage of it, however, is that the compiler is unable to optimize most accesses to global variables.
- **Locals** – RISC-V registers are translated to function's local variables. The main advantage of this approach is that the compiler is able to perform aggressive optimizations on those. The main disadvantage is that the values of these local variables need to be synchronized with those of other functions at function calls and returns, what can impact performance significantly on hot spots. We implement the synchronization by copying local register variables from/to global register variables when entering or leaving functions.

6. Experimental Setup and Infrastructure

In order to quantify the performance overhead introduced by the SBT, we compare the performance of benchmarks emulated with SBT against the performance of their native execution. Also, we designed several experiments to investigate the performance overhead on both x86 and ARM platforms and the effect of different compilers on the performance of the SBT. As a consequence, we employed multiple compilation flows in our experiments. These compilation flows are depicted on Figure 2.

The first compilation flow, Native (GCC), was used to produce native x86 and ARM binaries using the GCC compiler. The second compilation flow, Native (Clang), was used to produce native x86 and ARM binaries using the Clang compiler. In this case, the assembly code was generated by Clang 7.0 and the final binary was assembled and linked by GCC 6.3.0. We used this combination because LLVM's assembler and linker do not support RISC-V binaries. However, as we discuss later, differences in libc versions do not matter in our experiments because we factor out time spent in it. Now, in order to measure the performance of our SBT engine, we combine the following flows: three to produce RISC-V binaries (RISC-V OBJ) from benchmarks' source code (Clang soft-float

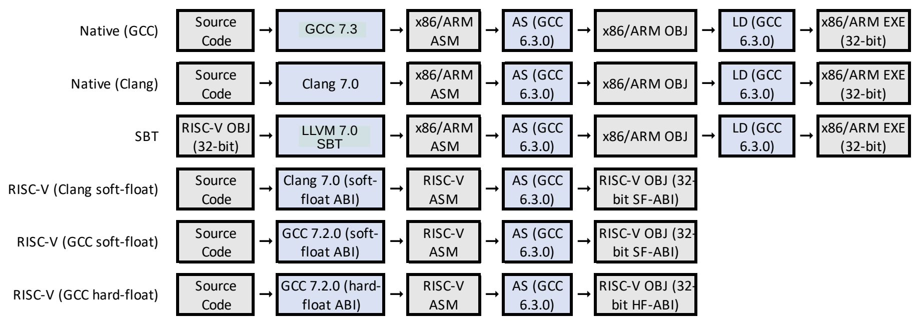


Figure 2. Code generation flows.

and GCC Soft or Hard float) with another to translate the RISC-V binaries to native code, using our SBT based on LLVM 7.0 (SBT).

To minimize performance differences that may be introduced by using different compiler versions and flags, we have used the same compilers and optimization flags (-O3 was used in all cases) for flows and experiments. Moreover, currently, Clang supports generating only RISC-V assembly code, not the full linked binary. Thus, for all targets, we used the same approach: use Clang (7.0) or GCC (7.3) to compile the source code (C) to ASM and then GCC (6.3) to assemble and link. Furthermore, for x86, the AVX extensions were enabled and, to avoid issues with legacy x86 extended precision (80-bit) floating point instructions, we also used the -mfpmath=sse flag. As for ARM, we targeted the armv7-a processor family, with vfpv3-d16 floating point instructions, as this is a perfect match with Debian 9 distribution for *armhf*.

6.1. Measurement Technique

To perform the experiments, after compiling and translating all needed binaries, each one was run 10 times. Their execution times were collected using Linux Perf and summarized by their arithmetic mean and standard deviation (SD). The execution times showed to follow a normal distribution with a very small SD. Thus, as it would not be relevant and in order to have clear graphics, we choose to not present the SD data.

Moreover, we also decided to factor out from the results the time spent on libc functions. We followed the same methodology aforementioned, executing the benchmarks 10 times and calculating the libc portion of the execution time arithmetic mean (percentage of the execution time). The final runtime of each benchmark is then multiplied by this percentage, so that time spent in parts other than the main benchmark code, such as libc, libm, and dynamic loader, are factored out.

6.2. GCC vs Clang and Soft vs Hard Float backend

To compile the benchmarks, our initial plan was to use Clang for every target: ARM, RISC-V and x86. However, during the experiments, we found out that Clang's support for RISC-V is still incomplete and considerably behind GCC's. For instance, some of LLVM optimizations need to be performed in collaboration with the target back-end or they may otherwise be skipped. But the major inefficiency we have noticed so far is that LLVM does not support RISC-V hard-float ABI. Although it is able to generate code that makes use of floating point instructions, function arguments are always passed through integer registers and stack, instead of using floating point registers whenever possible. This causes unnecessary copies from floating point registers to integer registers and vice-versa. This is further aggravated by the fact that, on RISC-V 32-bit, there is no instruction to convert a double value to a pair of 32-bit integer registers or to do the opposite conversion; this needs to be done in multiple steps, using the stack. Because of this, we also performed the same experiments using GCC to compile the code, so that we could have a higher quality RISC-V input code, especially on benchmarks that make heavy use of floating point operations.

As mentioned above, we observed that Clang produced RISC-V code with a considerably worse quality than GCC in some cases, especially on floating point benchmarks. Because of this, for x86, we performed the same experiments with both compilers. For ARM, we used only GCC and hard-float ABI in the experiments, as these gave the best results.

6.3. RISC-V Configuration

In our experiments and SBT implementation, we chose to use the RISC-V 32-bit instruction set, mainly to: facilitate comparisons with OpenISA, that is also 32-bit; facilitate translation to ARM 32-bit. The standard RISC-V extensions used by us were: M (integer multiplication/division instructions), F and D (single and double precision floating point instructions). Except for the A (atomic instructions) extension that we left out, these extensions compose the general-purpose RISC-V instructions. The reason for leaving the A extension out is that we have used only single-threaded benchmarks, in which case atomic instructions are not needed.

7. Experimental Results

In this section, we present the performance of our RISC-V SBT in terms of slowdown when compared to native execution (GCC RISC-V compared to GCC native and Clang RISC-V compared to Clang native). Hence, the higher the value the worse the emulation performance. A slowdown equal to 1 means that the translated binary is as fast as the native. In all cases, the guest binaries were translated using both the Globals and the Locals translation schemes.

7.1. GCC vs Clang

Figure 3 shows the performance of the SBT when emulating RISC-V binaries produced by Clang and GCC soft-float ABI on x86. In general, the performance of the Locals translation scheme produces better code than the Globals one. The only exception is bitcount. In this case, we profiled the code using *perf* and observed that what caused the poor emulation performance with Locals was the register synchronization overhead. This synchronization occurs at bitcount's main loop, when entering and leaving the main benchmark functions through indirect calls, making it harder for the compiler to optimize/inline these. On average, the Locals translation scheme achieved a slowdown with GCC/Clang of 1.20x/1.34x while the Globals scheme produced a slowdown of 2.06x/2.18x when emulating RISC-V code produced with soft-float ABI.

Figure 3 also indicates that in some cases (adpcm-decode, rijndael-decode, and sha) the SBT performs better when emulating code produced by Clang while in others (dijkstra, crc32, adpcm-encode, stringsearch, basicmath, and patricia) it performs better with GCC. Now, for benchmarks that do not use floating point operations (from dijkstra to blowfish-decode), we can see that emulation performance stays close to native performance, at least in Locals mode. Basicmath and FFT performed poorly, but the following figures make it clear that this is due to the lack of a hard-float ABI for RISC-V code in Clang, as these two benchmarks are among the ones that make the heaviest usage of floating point operations and have their performance improved when using the hard-float ABI.

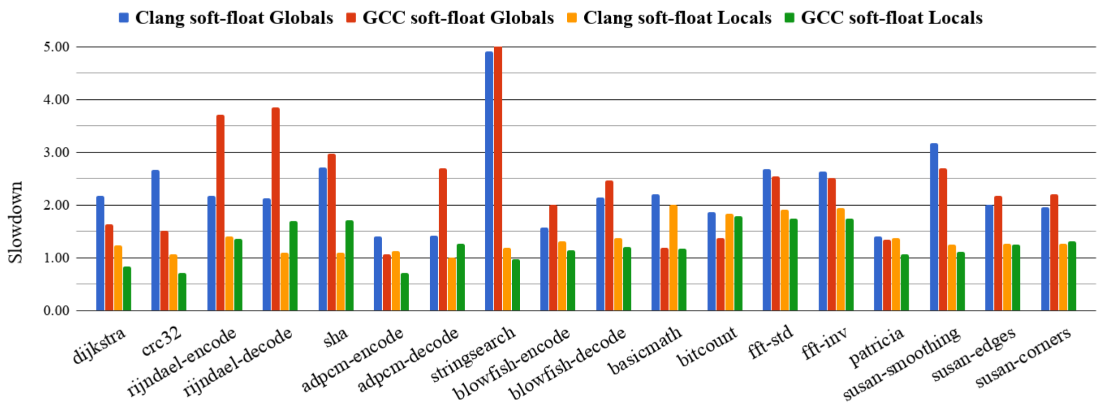


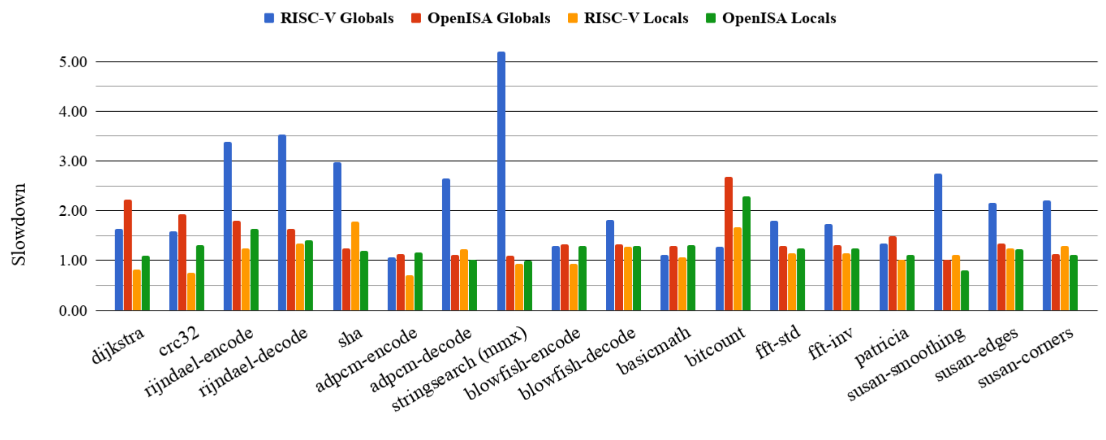
Figure 3. Slowdown benchmarks compiled with Clang and GCC soft-float RISC-V backends. Both Globals and Locals register mapping results are presented.

This GCC vs Clang experiment showed that although GCC has a more mature backend than Clang, our SBT performance when translating RISC-V binaries with soft-float ABI produced by both compilers was close. However, as the Clang RISC-V backend does not have, until the date, support to hard-float ABI, we are going to use solely GCC in the next experiments. Moreover, the Locals performance outstands the Globals performance in almost 2-fold, showing the importance of the register mapping approach.

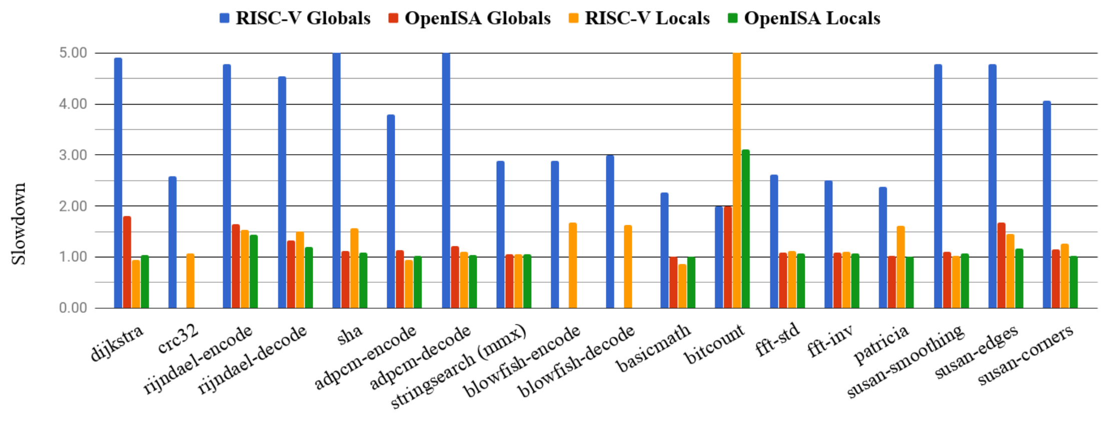
7.2. RISC-V vs OpenISA

In both Figure 4(a) and 4(b) we compared the performance of our SBT with the ones obtained by the OpenISA SBT translating OpenISA to x86 and ARM [Auler and Borin 2017] using the hard-float ABI. Translating RISC-V for x86, we obtained an average of 1.99x slowdown using Globals mapping and 1.12x using Locals, even better than the one obtained by the OpenISA SBT translating OpenISA to x86, which was 1.41x and 1.23x respectively. The ARM support for our SBT was the last added, so it still needs improvements, but its performance was close to the OpenISA: 3.49x slowdown with Globals and 1.35x with Locals, while with OpenISA it was 1.21x with Globals and 1.17x with Locals.

Figure 4(a) shows the results obtained using GCC hard-float ABI to compile RISC-V binaries and translating them to x86. Besides the results already discussed above for soft-float ABI (Figure 3), we can see a great improvement in FFT and some smaller improvements in basicmath and patricia when using hard-float ABI. Figure 4(b) shows the same binaries translated to ARM. One thing that stands out is that Globals results are much worse for the ARM and this needs further investigation. Moreover, we can see good performance for a handful of benchmarks, similar to x86's, such as dijkstra, crc32, adpcm, stringsearch, basicmath, FFT, and susan when using Locals. Others, on the other hand, suffered a much greater slowdown than x86, such as rijndael, blowfish, bitcount, and patricia. Further analysis is required to understand such differences. Nevertheless, despite having slightly worse results than when translating to x86, ARM results were only 23% worse than the results from OpenISA.



(a) x86



(b) ARM

Figure 4. Slowdown for our RISC-V SBT and for the OpenISA SBT. All binaries were compiled using the hard-float ABI.

7.3. Our SBT engine vs DBT engines available

Finally, we compared our best approach, translating binaries generated with hard-float from GCC to x86, with the two most known DBT engines for RISC-V available: RV8 (github 1d4d1ee commit) and QEMU (v2.12.0-835-g360a7809d2-dirty). We can clearly see in the chart from Figure 5 that our RISC-V SBT was the one with the best performance for all tested programs. Our RISC-V SBT achieved, on average, 1.12x slowdown, while RV8 and QEMU achieved 3.16x and 7.07x slowdown, respectively. However, we were not able to run all the benchmarks with RV8 and QEMU. With RV8, during the emulation of some benchmarks, it finished with a segment fault. With QEMU, the benchmarks which manipulated files did not run because some of the syscalls were missing.

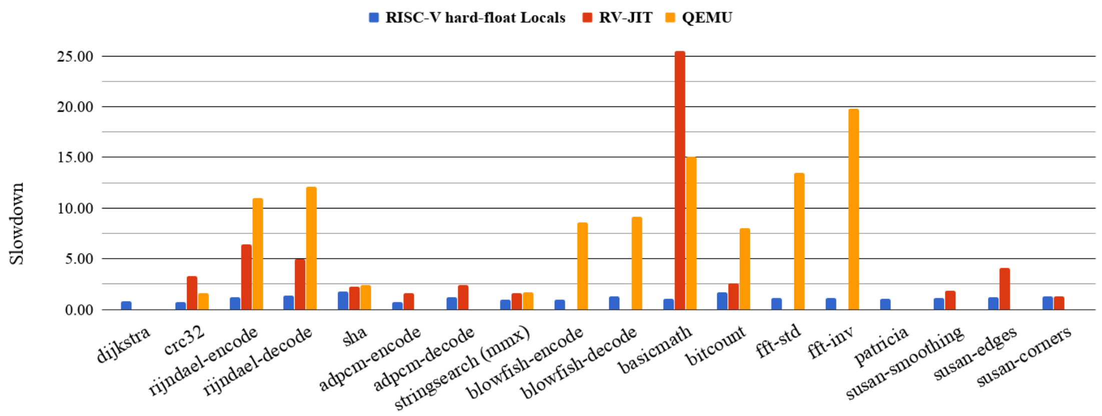


Figure 5. Slowdown comparison between our RISC-V SBT, RV8 DBT and QEMU-RISCV DBT. All tests emulating RISC-V binaries on a x86 processor.

Table 1 lists emulation results presented so far in the literature. As can be noticed, other RISC-V emulators suffer from high overheads (more than 3x) when emulating RISC-V code on x86 and ARM platforms. Also, the low overheads achieved by our SBT suggest that it is possible to design and implement high-performance DBTs to emulate RISC-V code on x86 and ARM platforms.

Table 1. Comparison Between Binary Translator Approaches

Name	Guest-ISA	IR	Target-ISA	Technique	Avg. Slowdown
StarDBT	x86	None	x86	DBT	1.09x
OpenISA-SBT	OpenISA	LLVM 3.7	x86 & ARM	SBT	1.23x/1.17x
ANGEL	RISC-V	None	x86	Interpreter	200x
Spike	RISC-V	None	x86	Interpreter	50x
Pydgin	RISC-V	None	x86	DBT	4x
QEMU	RISC-V	QEMU IR	x86, ARM...	DBT	7.07x
RV8	RISC-V	None	x86	DBT	3.16x
Our SBT	RISC-V	LLVM 7.0	x86 & ARM	SBT	**1.12x/1.35x**

8. Conclusion

RISC-V is having the attention globally from the industry and academia. Thus, it is probable that RISC-V is going to have a significant impact in the future of IoT and cloud. However, by now, there is no RISC-V emulation with low overhead available. In this work, we demonstrated that RISC-V is an architecture that enables its code to be translated into high-quality x86 and ARM code. An strong evidence that DBT engines with high-performance can be built for RISC-V. We did this by building a RISC-V static translator which is able to translate RISC-V to x86 and ARM with an execution overhead lower than 12% in the former and 35% in the latter, being the fastest RISC-V emulator presented so far in the literature.

In future work, we plan to modify the OpenISA DBT (OI-DBT) to also support RISC-V, enabling high-performance emulation of RISC-V code with DBT and accelerating its adoption. We also expect to show that RISC-V can be used as an IR for easy software deployment.

References

Apple (2006). Rosetta: the most amazing software you'll never see. *Archived. Url: http://www.apple.com/asia/rosetta/. Last Accessed 07/2018.*

Auler, R. and Borin, E. (2017). The case for flexible isas: unleashing hardware and software. In *SBAC-PAD*, pages 65–72. IEEE.

Bartholomew, D. (2018). Risc-v qemu. *Github. Url: https://github.com/riscv/riscv-qemu. Last Accessed 07/2018.*

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46.

Böhm, I., Edler von Koch, T. J., Kyle, S. C., Franke, B., and Topham, N. (2011). Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *ACM SIGPLAN Notices*, volume 46, pages 74–85. ACM.

Borin, E. and Wu, Y. (2009). Characterization of dbt overhead. In *IISWC 2009.*, pages 178–187. IEEE.

Cifuentes, C. and Malhotra, V. M. (1996). Binary translation: Static, dynamic, retargetable? In *icsm*, pages 340–349.

Clark, M. and Hoult, B. (2017). rv8: a high performance risc-v to x86 binary translator.

Dehnert, J. C., Grant, B. K., Banning, J. P., Johnson, R., Kistler, T., Klaiber, A., and Mattson, J. (2003). The transmeta code morphing$^{\text{TM}}$ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO*, pages 15–24. IEEE Computer Society.

Häubl, C. and Mössenböck, H. (2011). Trace-based compilation for the java hotspot virtual machine. In *Proceedings of the 9th PPPJ*, pages 129–138. ACM.

Ilbeyi, B., Lockhart, D., and Batten, C. (2016). Pydgin for risc-v: A fast and productive instruction-set simulator. In *Extended Abstract for Presentation at the 3rd RISC-V Workshop*.

RV-foundation (2018a). Risc-v foundation — instruction set architecture (isa).

RV-foundation (2018b). Risc-v software tools. *Url: https://riscv.org/software-tools. Last Accessed 07/2018.*

Salgado, F., Gomes, T., Pinto, S., Cabral, J., and Tavares, A. (2017). Condition codes evaluation on dynamic binary translation for embedded platforms. *IEEE Embedded Systems Letters*, 9(3):89–92.

Smith, J. and Nair, R. (2005). *Virtual machines: versatile platforms for systems and processes*. Elsevier.

Ta, T., Cheng, L., and Batten, C. (2018). Simulating multi-core risc-v systems in gem5.

Waterman, A., Lee, Y., Patterson, D., and Asanovic, K. (2014). The risc-v instruction set manual. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*.