Universidade Estadual de Campinas
Instituto de Computação

Leandro Lupori

High-Performance RISC-V Emulation

Emulação de RISC-V com Alto Desempenho

CAMPINAS
2019

Leandro Lupori

High-Performance RISC-V Emulation

Emulação de RISC-V com Alto Desempenho

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Edson Borin

CAMPINAS
2019

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

L973h

Lupori, Leandro, 1983-
High-performance RISC-V emulation / Leandro Lupori. – Campinas, SP : [s.n.], 2019.

Orientador: Edson Borin.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Compiladores (Programas de computador). 2. Arquitetura de computador. 3. Computação de alto desempenho. I. Borin, Edson, 1979-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Emulação de RISC-V com Alto Desempenho
Palavras-chave em inglês:
Compilers (Computer programs)
Computer architecture
High performance computing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Anderson Faustino da Silva
Lucas Francisco Wanner
Data de defesa: 14-03-2019
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Leandro Lupori

High-Performance RISC-V Emulation

Emulação de RISC-V com Alto Desempenho

Campinas, 14 de março de 2019

Acknowledgements

I would like to take this opportunity to first and foremost thank God, for giving me health and wisdom, and for enabling me to finish this dissertation.

I would like to express my special thanks to my supervisor, Prof. Edson Borin, and to his Ph.D student, Vanderson Martins do Rosário, for their guidance and support throughout this master's program.

I would also like to thank the Institute of Computing of UNICAMP and LMCAD for their support and infrastructure and Eldorado Research Institute for granting me weekly hours to dedicate to this work.

Last but not least, I would like to thank my wife, Leda Maria Horta Lupori, for all her support during the years of this work.

Resumo

RISC-V é uma ISA aberta que tem chamado a atenção ao redor do mundo por seu rápido crescimento e adoção. Já é suportado pelo GCC, Clang e Kernel Linux. Além disso, vários emuladores e simuladores para RISC-V surgiram recentemente, mas nenhum deles com desempenho próximo ao nativo. Nesta dissertação, nós investigamos se emuladores mais rápidos para RISC-V podem ser criados. Como a técnica mais comum e também a mais rápida para implementar um emulador, Tradução Dinâmica de Binários (TDB), depende diretamente de boa qualidade de tradução para alcançar bom desempenho, nós investigamos se uma tradução de alta qualidade de binários RISC-V é plausível. Desta forma, neste trabalho nós implementamos e avaliamos um motor de Tradução Estática de Binários (TEB) baseado no LLVM, para investigar se é ou não possível produzir traduções de alta qualidade de RISC-V para x86 e ARM. Nossos resultados experimentais indicam que nosso motor de TEB consegue produzir código de alta qualidade quando traduz binários RISC-V para x86 e ARM, com sobrecargas médias em torno de 1.2x/1.3x quando comparado à código nativo x86/ARM, um resultado melhor do que motores de TDB de RISC-V bem conhecidos, como RV8 e QEMU. Além disso, como motores de TDB tem seu desempenho fortemente relacionado à qualidade de tradução, nosso motor de TEB evidencia a oportunidade na direção da criação de emuladores RISC-V de TDB com desempenho superior aos atuais.

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, but none of them with near-native performance. In this work, 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. Thus, in this work we implemented and evaluated a LLVM-based Static Binary Translation (SBT) engine to investigate whether or not it is possible to produce high-quality translations from RISC-V to x86 and ARM. Our experimental results indicate that our SBT engine is able to produce high-quality code when translating RISC-V binaries to x86 and ARM, with average overheads around 1.2x/1.3x when compared to native x86/ARM code, a better result than well-known RISC-V DBT engines such as RV8 and QEMU. Moreover, since DBT engines have its performance strongly related to translation quality, our SBT engine evidences the opportunity towards the creation of RISC-V DBT emulators with higher performance than the current ones.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

RISC-V is a new, open and free Instruction Set Architecture (ISA), initially developed at the University of California [1] and now maintained by the RISC-V foundation [2], with a handful of companies supporting its development. It is a small RISC-based architecture divided into multiple modules that support integer computation, floating-point, atomic operations, and compressed instructions, besides others that are being developed for vector, bit manipulation, transactional memory and packed SIMD instructions [1], 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 [3, 4, 5] and emulators [6, 7].

In terms of ISA design, RISC-V is reaching a mature and stable state only by now [1]. 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 [1]. For the privileged ISA, at the time of this writing, the latest released specification [8] was still a draft, albeit at an advanced stage. For the physical implementations, there are several open-sourced RISC-V CPU designs available [9, 10, 11] and also some off-the-shelf RISC-V development boards [12, 13]. These open-sourced designs and development boards are great steps towards making RISC-V CPU chips easily available, an stage that was still not reached, as 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 emulate guest instructions using host instructions, with the goal of making the host perform an equivalent computation to what would be achieved by the guest instructions being executed on the guest platform. However, not only mimicking the computation is important, but normally performance also plays a crucial role. For RISC-V, to the best of our knowledge, at the current time, no emulator can achieve near-native performance — which in this work we consider to be around 1.20x slower than native (s.t.n.) — as the best RISC-V emulators performances are more than 2 times s.t.n. This limits the scope of RISC-V emulators, by excluding them from use cases where performance plays a major role.

Having a high-performance RISC-V emulator for common architectures, i.e. x86 and

ARM, would not only facilitate RISC-V adoption and testing but also would show it as an useful virtual architecture to ease software deployment. One approach to implement a high-performance emulator is by using Dynamic Binary Translation (DBT) [14], 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 Apple's PowerPC to x86 migration software Rosetta [15], to enable changes in microarchitecture without changing the architecture itself, as with the Transmeta Crusoe and Efficeon processors [16] that implement x86, or in the deployment of high-level languages in several platforms such as with the Java VM [17].

A DBT engine usually starts by interpreting the code and then, after warming-up (translating all hot regions), it spends most of the time executing translated regions. Thus, the quality and performance of these translated regions are responsible for most of the DBT engine performance [18] 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 [19] and (2) the characteristics of the guest and host ISA which can hinder or facilitate the translation [20].

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 [20]. 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, but its design and implementation are usually much simpler than those of a DBT. Since the translation mechanisms in a DBT and an SBT are very similar, if one is able to create an SBT engine which can emit high-quality code for a pair of architectures, it implies that the same can be done for a DBT engine. Thus, in this work we implemented and evaluated 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 was capable of producing high-quality translations, that execute almost as fast as native code, with around 1.2x/1.3x slowdown in x86/ARM. In this way, the main contributions of this work are the following:

- We present a novel Open Source RISC-V Static Binary Translator based on LLVM.[1]
- We show with our RISC-V SBT 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 engine with the performance of state-of-the-art RISC-V emulators and argue that there is still a lot of room for improvement on dynamic RISC-V emulators.
- We show that it is not trivial to make LLVM be able to vectorize code when translating optimized code from an ISA that does not support vectorization to another one that does.

[1]https://github.com/OpenISA/riscv-sbt

- We show that the aggressive use of registers when performing loop unrolling in guest code can have a significant performance impact in translated code when the host ISA has considerably less registers than the guest.

Furthermore, two other works were developed in the context of this dissertation. In the first, Uma Análise da Facilidade de Emulação de Binários RISC-V [21], presented at ERAD-SP 2018, we investigated in more depth the ease of emulation of RISC-V binaries, comparing it with OpenISA [20], an ISA designed to allow high-performance emulation. This revealed many similarities between RISC-V and OpenISA and indications that RISC-V could also be emulated with low overhead. In the second, Towards a High-Performance RISC-V Emulator [22], presented at WSCAD 2018, we have evaluated some of the fastest RISC-V DBT engines available, comparing them with an initial version of our SBT engine, that was able to translate RISC-V binaries with considerably less overhead, evidencing the opportunity towards the creation of faster DBT engines.

The rest of this dissertation is organized as follows. Chapter 2 further describes ISA emulation techniques, the challenges to implement them, discusses ISA characteristics that are difficult to translate and presents other emulators for RISC-V. Then, in Chapter 3 we discuss our SBT engine for RISC-V, in Chapter 4 we describe our experimental setup and how the compiler settings that gave the best results were found, and in Chapter 5 we discuss the results we have obtained with our RISC-V SBT engine. Lastly, Chapter 6 presents our future work and conclusions.

Chapter 2

ISA Emulation and Related Work

Interpretation and DBT are well-known methods used to implement ISA emulators. In this section, we examine them in more details, along with SBT and works that achieved good performance results with each method.

2.1 Interpretation

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. For instance, Bochs [23] is a well-known and mature x86 interpreted emulator, able to emulate the entire system and boot operating systems. But, by emulating x86 over x86, its performance varies from 31 to 95 host cycles per instruction emulated (or about 31 to 95x slower than native) on average, measured using the SPEC CPU2006(int) benchmark [24]. Therefore, we conclude that even high-performance interpreted emulators such as Bochs are not good enough when compared to native execution performance.

2.1.1 RISC-V Interpreters:

The gem5 simulator [3, 25] is a modular platform for computer-system architecture research, supporting multiple distinct CPU architectures, such as x86, ARM, SPARC, and now also RISC-V (gem5 for RISC-V is a.k.a. RISC5). While a strong point of it is its ability to perform accurate CPU simulation and capture microarchitectural details, this ends up resulting in a much slower emulation speed — around 175 KIPS (Thousand Instructions per Second) on RISC-V [3] — that while being well above other in-depth simulators, such as the Chisel C++ RTL simulator, is well below other RISC-V emulators not trying to capture microarchitectural details, such as Spike and QEMU.

TinyEMU [26] is a system emulator for RISC-V and x86 architectures. Its purpose is to be small and simple while being complete. It even supports the 128-bit RISC-V variant and quadruple-precision floating-point (Q extension). While we found no performance data available for it yet, it should be similar to that of purely interpreted emulators. On x86 it makes use of KVM, which in general achieves a performance well above that of interpretation due to hardware acceleration, but that doesn't help on improving RISC-V performance.

ANGEL [27] is a Javascript RISC-V ISA (RV64) Simulator that runs RISC-V 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.

Spike [28], 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 emulators in some cases, varying from 15 to 75 times slower than native on SPECINT2006 benchmarks [4]. 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.

As expected, RISC-V emulators that use mainly interpretation are far from near-native performance.

2.2 Dynamic Binary Translation (DBT)

Dynamic Binary Translators translate (map) pieces of guest code into host code and usually obtain greater performance with the cost of being more complex and harder to implement. Because of this, translation is commonly used on high-performance emulators, such as QEMU [29]. 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 [30]. It is important to notice that the costs associated with the translation process impact 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.

In most programs, the majority of their execution is spent in small portions of code [19]. Thus, when emulating these programs, DBT engines also spend most of their time executing small portions of translated code. This implies that the translation quality of these portions of code is crucial to the final performance of a DBT engine. In fact, this is evidenced by the low overhead of same-ISA DBT engines [18], also known as binary optimizers, as they always execute code with the same or better quality than the native binary (this happens 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 as the quality of the translated code depends heavily on the characteristics of the guest (source) and the host (target) ISA. For instance, 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 on x86 [31]. Experience has shown that when emulating a guest-ISA which is simpler than the target-ISA it is normally easier to obtain high quality translation [20]. Next, we examine some well-known DBT engines with high-performance emulation.

Hong et al. created HQEMU [32] in an effort to enhance QEMU [29] with LLVM and try to achieve near-native performance. It uses QEMU standard Tiny Code Generator (TCG) for fast translation of cold regions, while LLVM runs on another core to aggressively optimize traces of hot regions. The geometric mean of the overhead compared to native execution is 2.5x for x86 emulation on x86-64 (almost same-ISA emulation) and 3.4x for ARM emulation on x86-64 (cross-ISA setup), with an i7 3.3 GHz as the host machine. This same work also evaluates the performance of QEMU as a baseline, reporting 5.9x on the same-ISA emulation setup and 8.2x on the cross-ISA setup.

In a more recent work, HERMES [33] proposes to drop the architecture of QEMU in favor of a host-specific data dependency graph, which allows exploring optimizations at a representation that is closer to the host instead of the generic IR of QEMU. HERMES achieves the performance of, on average, 2.66x slower than native for SPEC CPU2000 programs, which is very competitive for a cross-ISA translation.

One of the best performances we see in literature is achieved by IA32-EL, by Baraz et al. [34], an ISA translator that runs the x86 guest programs on the discontinued Itanium architecture. They built a specialized DBT engine that runs x86 programs, on average, 1.35x slower than native Itanium programs, albeit their DBT is focused on only a specific guest and host machine pair.

For same-ISA emulation, the best performance achieved is that of StarDBT by Borin and Wu [18]: 1.1x slower than native (x86) emulation.

These works show that it is possible to achieve near-native performance by means of DBT techniques, even with cross-ISA emulation, as shown by IA32-EL. In the cross-ISA scenario, however, the performance of the DBT is highly dependant on guest and host ISA characteristics, as we mentioned earlier.

2.2.1 RISC-V Dynamic Binary Translators:

Ilbeyi et al. [4] showed that the Pydgin Instruction Set Simulator can achieve better performance than Spike, by means of more sophisticated techniques, mainly, DBT. Pydgin with DBT is able to achieve between 4x to 33x slower than native performance. While achieving a better result than Spike, Pydgin is slower than QEMU for RISC-V, that was still unavailable at the time Pydgin work was published.

QEMU [29], a famous DBT with multiple sources and targets, also gained support for RISC-V. QEMU is 4.57x slower than a native execution [6] 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.

OVP Simulator for RISC-V [5] implements the full functionality of RISC-V User and Privileged specifications. It is developed and maintained by Imperas Software [35], being fully compliant to the OVP open standard APIs. As we show in Chapter 5, it is on average 4.92x s.t.n., a result close to that of QEMU.

Clark and Hoult [6] presented the RV8 emulator, a high-performance RISC-V DBT for x86. Using optimizations such as macro-op fusion and trace formation and merge, RV8 is able to achieve a performance 2.62x slower than native, on average, overcoming QEMU and being the fastest known RISC-V DBT engine. Note, however, that RV8 is currently more limited than QEMU, as it does not support running as many program types as QEMU and it can run only on x86. Besides, although 2.62x s.t.n. is a good performance result, it is still far from near-native performance.

2.3 Static Binary Translation (SBT)

Having a high-performance DBT for a pair of ISAs would prove that, for these ISAs, it is possible to achieve good translation quality. However, implementing a DBT is a complex project and a challenge by itself. Another possibility 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 SBT cannot execute all kinds of applications. Self-modifying code, code discovery problems and indirect branches are some of the emulation problems that cannot be handled statically [36]. However, for the purpose of testing the difficulty of translating code with high-quality, an SBT is enough.

A remarkable SBT engine we see in literature is LLBT [37], a static binary translator based on LLVM that achieves cross-ISA translation (ARM to an Intel Atom) with 1.40x of overhead, on average, for the EEMBC benchmark.

Going further, according to Auler and Borin [20], 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 an overhead of less than 1.10x for the majority of programs.

In fact, among the main motivations for this work was the near-native performance that Auler and Borin were able to achieve with their OpenISA emulator, and the similarities between OpenISA and RISC-V, that suggested that RISC-V could also be emulated very efficiently. OpenISA's work [20] discusses several characteristics that may ease or difficult an ISA emulation. RISC-V has most of the characteristics pointed by the authors to be easy to emulate: it is simple, it hardly uses status registers and it has a small number of instructions — RISC-V has 107 and OpenISA 139, with 66% of them being equivalent — all indicating that RISC-V is also an easy to emulate ISA. This is the reason why we use the same approach as that used by Auler and Borin to test OpenISA emulation performance and this is the methodology that we use to test if RISC-V translation can achieve good performance.

Table 2.1 summarizes the ISA emulation results presented in this section.

Name	**Guest**	**Host**	**Benchmark**	**Performance (x s.t.n.)** [1]
Bochs [24]	x86	x86	SPEC CPU2006 (int)	31 to 95 [2]
QEMU [32]	x86	x86	SPEC CPU2006 (int)	5.9
QEMU [32]	ARM	x86	SPEC CPU2006 (int)	8.2
HQEMU [32]	x86	x86	SPEC CPU2006 (int)	2.5
HQEMU [32]	ARM	x86	SPEC CPU2006 (int)	3.4
Hermes [33]	x86	MIPS	SPEC CPU2000	2.66
LLBT [37]	ARM	x86	EEMBC	1.40
IA32-EL [34]	x86	Itanium	SPEC CPU2000 (int)	1.35
StarDBT [18]	x86	x86	SPEC CPU2000	1.10
OpenISA SBT [20]	OpenISA	x86	Mibench and SPEC CPU2006	1.10
OpenISA SBT [20]	OpenISA	ARM	Mibench and SPEC CPU2006	1.10
RISC5 [3]	RISC-V	x86	Ligra	17142 [3]
ANGEL	RISC-V	x86	boot linux	200 [4]
Spike [4]	RISC-V	x86	SPEC CPU2006 (int)	15 to 75 [5]
Pydgin [4]	RISC-V	x86	SPEC CPU2006 (int)	4 to 33 [6]
OVP	RISC-V	x86	MiBench	4.92
QEMU [6]	RISC-V	x86	other [7]	4.57
RV8 [6]	RISC-V	x86	other [7]	2.62

[1] s.t.n. = slower than native
[2] host cycles per emulated instruction
[3] 175 KIPS, on a unspecified x86 machine (estimating 3000 MIPS)
[4] 10 MIPS, on an Intel i7 2.0GHz ($\approx$ 2000 MIPS)
[5] 40 to 200 MIPS, on a "contemporary server-class host" (estimating 3000 MIPS)
[6] 90 to 750 MIPS, on a "contemporary server-class host" (estimating 3000 MIPS)
[7] AES cipher, dhrystone v1.1, miniz compression and decompression, the NORX cipher, prime number generation, qsort and the SHA-512 digest algorithm

Table 2.1: Summary of ISA emulation related work and their performances.

Chapter 3

A Static Binary Translator for RISC-V

Static translation of RISC-V binaries into native form for other architectures, such as ARM and x86, involves several steps. Our Static Binary Translator starts by reading a RISC-V object 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 (a.k.a. bitcode). Instruction translation is covered in more details in later subsections. Note that this is very similar to what Clang does when compiling a source file to bitcode. 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. All these steps for SBT are summarized in the diagram from Figure 3.1. The code generation flows used in our experiments are further detailed in Chapter 4.

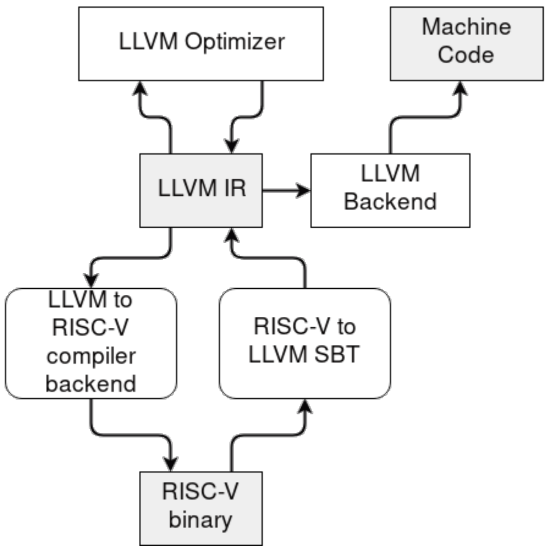


Figure 3.1: Our RISC-V SBT engine architecture.

In the following subsections, we first discuss our approach of using unlinked objects as input, then we describe in details how register mapping is implemented in our SBT engine, and how the different approaches to it impact in performance. Finally, we explain in more depth how each RISC-V instruction is translated to LLVM IR.

3.1 Unlinked Objects as Input

Instead of translating final linked binaries, we chose to translate relocatable object files to avoid dealing with some issues inherent to SBT, such as differentiating code from data. It also enables us to translate only the benchmark code, leaving the C runtime out, which simplifies the implementation and debugging of the SBT engine, saving a considerable amount of work that would otherwise be required. With this approach, however, the translator must 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.

It is worth mentioning that, from the 19 benchmarks profiled and evaluated by us, 8 (Dijkstra, ADPCM-Encode, ADPCM-Decode, Susan-Smooth, Susan-Edges, Susan-Corners, BitCount and LAME) spent more than 80% of the time in the main binary, 2 (Rijndael-Encode and Rijndael-Decode) spent more than 60%, 2 (BlowFish-Encode and BlowFish-Decode) spent more than 40% on x86 and more than 25% on ARM, 1 (SHA) spent around 25%, and only 6 (CRC32, StringSearch, Patricia, BasicMath, FFT-Standard and FFT-Inverse) spent less than 20% of the time in the main binary. While most benchmarks spent most time in the main binary, considerable time was spent in non-translated *libc* code, that must not be considered when calculating the slowdown of the translated binaries. To handle this and improve the accuracy of the results, we also measure and factor out the time that the binaries spend in *libc*, as shall be explained in more details in Chapter 4. With this measure, the results obtained by our SBT engine are not benefited by the non-translated native *libc* code that is excuted.

3.2 Register Mapping

Regarding register mapping between architectures during the translation, our SBT engine implements 3 techniques:

- **Globals** – RISC-V registers are translated to global variables. In this technique, the translator emits load/store instructions to read/write from/into these global variables whenever registers are used/modified by guest instructions. The main advantages of this approach are 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. In this technique, the translator emits load/store instructions to read/write from/into these local variables whenever registers are used/modified by guest instructions. 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 variables, when entering and leaving functions.

- **ABI** – this technique is very similar to Locals, with the main difference being that only registers that are specified as non-volatile in RISC-V ABI are preserved through function calls. This reduces the synchronization overhead considerably, but limits the translatable programs to those that conform to RISC-V ABI.

3.3 Code Translation

In general, most RISC-V instructions have a direct or close enough LLVM IR instruction, that eases the task of implementing a binary translator for it. However, some classes of instructions are difficult to translate by nature, such as branches and jumps, but that is the case for most architectures, not a RISC-V particularity. Next, we first go through some general implementation decisions and then give an overview of how the main RISC-V instruction classes were implemented in our SBT engine.

3.3.1 Shadow Image

Before translating the code present in the `.text` section, all other sections are processed and a *Shadow Image* of the binary is built, that is, a copy of the sections of the original binary. This copy is then modified, by translating most of the guest addresses, during the relocation process, adding some helper data, such as the global variables used to simulate or synchronize RISC-V registers, and some other adjustments.

3.3.2 Handling Relocations

As we use unlinked object files as input, our translator must be able to handle relocations properly. Our approach to implement this is to process all relocation entries and make the corresponding address point to the emulated guest memory area. This way, memory accesses to data are performed by using the guest address as an offset that is relative to the base address of the binary *Shadow Image*. In the case of relocations that point to the `.text` section, however, we need to defer the relocation, until the corresponding code location is translated and its host address or label is known. This way, for addresses that point to instructions inside a function, that are potential indirect branch targets, pending relocations are created, and resolved during code translation.

3.3.3 Arithmetic Logic Operations

For the arithmetic and logic instructions, in the vast majority of cases the translation is trivial — not considering the register mapping code — as there is a direct correspondent instruction in LLVM IR. Examples are *add*, *sub*, and *xor*. Some instruction forms were optimized, such as *xor a,b,-1*, that is used in RISC-V to perform a logical *not*, that does not have a distinct instruction for it, but is present in other ISAs, that usually execute it faster than a *xor*. In such cases the translator emits the optimal LLVM IR instruction, that would be a *not* in this case. In a few cases, however, there is no direct equivalence in LLVM IR, such as in the *mulh* instruction, that returns the upper 32 bits of a multiplication. In this case, this instruction is translated to a 64-bit multiplication, an arithmetic shift right of 32 bits and a truncate instruction. Here it would be possible to optimize the case where both *mul* and *mulh* instructions were performed with the same inputs, but we left this optimization out, as we did not detect this to have a significant impact in our benchmarks.

3.3.4 Load/Store

The load and store instructions were also straightforward to implement, mainly because the code that handles relocations, explained above, already performs the necessary guest to host address translation. Then it was just a matter of calculating the resulting address, by adding the base and offset parts, performing the load or store of the correct amount of bytes, and, for loads, performing the necessary zero or sign extension.

3.3.5 Direct Branches and Jumps

Branch and jump instructions, on the other hand, are considerably more complex to translate. This, however, is not due to hard to translate RISC-V instructions, but due to the inherent difficulty of translating branch and jump instructions in general. Excluding the instruction forms that are used to perform function calls, all branch and jump instructions are translated in the same way. First, the correct condition test is emitted, if the instruction is a branch. Next, a conditional or an unconditional jump is emitted. To compute the target guest address, the displacement operand of the branch or jump instruction is added to the current guest address, that in our case is equal to the address of the instruction being translated. If the resulting address is greater than the current one, a new basic block is created at the target address and added to the basic blocks' map. Otherwise, a lookup by address is perfomed at the function's basic blocks, to check if there is already one starting at the target guest address. If not, the basic block that contains the instruction at the target address is located and split in two at that point. The branch to the correct basic block can then be emitted.

3.3.6 Indirect Jumps

The translation of indirect jumps is performed in three steps: the binary relocation phase; the processing of pending relocations during instruction translation; and the addition of all possible targets of each indirect jump, after all function instructions have been translated.

During the binary relocation phase, explained in Subsection 3.3.2, pending relocations against the `.text` section are produced. These relocations refer to host addresses that are unknown before the translation of the functions they point to. Instead, they are resolved during the translation of guest instructions, where the guest address of each instruction is checked against pending relocations' guest addresses. When a match occurs, the corresponding pending relocation is resolved to the now available host address of the translated code.

With this approach, each RISC-V indirect jump can be translated directly to an LLVM indirect branch instruction. This is possible because of the use of relocation information, that enables our SBT engine to replace guest code addresses on jump tables by host addresses, so that indirect jump instructions can simply jump to already translated addresses. But note that our translation of indirect jumps handles only common cases: indirect jumps emitted by compilers, with a limited number of targets, usually loaded from jump tables. For the general case, it can be handled efficiently with known DBT techniques [38].

The last step is performed after all function's instructions have been translated. After that, the SBT engine goes through each LLVM indirect branch instruction emitted, adding all possible destinations — as they are known at this stage — to them. This is needed in order to make LLVM correctly build the Control Flow Graph (CFG) of the function. A final remark about indirect jumps is that they were not very common in our benchmarks, and thus their translation quality contributed little to increase or decrease the measured performance.

3.3.7 Function Calls

The translation of function calls can be divided in 3 parts: direct calls to functions inside (internal) the benchmark binary (the main binary); direct calls to functions outside (external) the main binary, (e.g. *libc* functions); and indirect function calls. There are however some translation aspects that are common to all 3 types. One of them is that at function calls and returns, mapped registers are synchronized between the involved functions, if the Locals or ABI register mapping approach is used. Another one, is that, as a simplification, we take advantage of the regular/ABI RISC-V return form, that is *jalr zero, ra*, and emit a LLVM return instruction in this case. Tail calls are identified and handled appropriately by our translator, but more rare types of function returns, such as loading the return address to an arbitrary register and jumping to there would fail. Note, however, that despite the fact that SBT is not always viable, such as in this case, this does not invalidate our experiment — that shows that it is possible to generate high-quality code for x86 and ARM — because our SBT engine was able to translate several benchmark programs, indicating that it is able to translate the most used code constructions, even though it would require a more sophisticate technique in some corner cases.

3.3.8 Direct Internal Function Calls

For direct calls to internal functions, the implementation is similar to that of jumps. First, the guest target address is calculated. Next, the target function is looked up by address, at the functions map. If it is not found, then a new function is created, initially empty.

Then, in order to mimic the behavior of RISC-V *jump and link* instruction, the address of the next instruction is saved to the emulated RISC-V output register specified in the instruction (a.k.a. the link register) and, finally, the call is made.

3.3.9 Direct External Function Calls

For external calls, that can be identified by relocation entries pointing to undefined symbols, our *libc* call forwarding mechanism is used. Our translator looks up the function by name, from an LLVM IR file that lists all *libc* functions used by our benchmarks. This file provides the SBT engine almost all information that is needed to perform a *libc* call, including its parameters and types. The "almost" here is due to functions with variable number of arguments, that, at compile time, make it difficult to discover the correct number and types of the arguments that need to be passed on each call. For them, we pass up to four extra integer arguments, for the variadic part. However, special handling must be done for *printf()* and similar functions, in cases where there are more than 4 arguments or that mix floating-point and integer arguments. These special cases are not handled in our current SBT engine implementation, instead, some complex calls with variable number of arguments were broken into simpler ones in the benchmarks source code. Although in most cases (those that use a literal format string) this could be done, by inspecting the format string to find out the number of arguments and their types, this was left out due to the amount of work that would be required, and the limited benefits of it in our research. In any case, the arguments for the external function calls are loaded from the mapped registers where RISC-V ABI expects to find them and then an LLVM IR call instruction is emitted. Analogously, the return value is written to the return value register, as specified by RISC-V ABI. It is worth noting that this *libc* call forwarding mechanism is used by us more as a mean of isolating and avoiding having to translate several complex *libc* function calls in our experiments, and not as a suggested high-performance translation technique for general cases. Besides, it is important to remember that, as discussed in Subsection 3.1, most of the benchmarks' execution time is spent in the main binary and not in *libc* and also that the time spent on it is not considered by us when calculating the slowdown of the translated binaries.

3.3.10 Indirect Function Calls

To handle indirect calls, we implemented two translation mechanisms. The more efficient one is able to handle only indirect calls to internal functions, that is, functions in the main binary. It works almost the same way as the indirect jump implementation, where the translation of guest to host address is performed in address relocation, but a call is made instead of a jump, which involves saving the return address to the link register and synchronizing mapped RISC-V registers. The more complex mechanism, that must be able to handle indirect calls to external functions too, whose addresses are unknown before the link phase, works as follows. References to each external function are replaced by distinct, invalid, addresses in a given range. Then, when an indirect call is made, an indirect caller function is invoked, that consists basically of a big switch with the target address as input, where each case makes the appropriate internal function call or external *libc* function call forwarding. This extra indirection incurs some overhead, that may be significant when inside loops with a small body but many iterations, such as one present in BitCount, in which the indirection overhead is about 0.1x, in Locals mode. In its case, we took advantage of the fact that it does not make indirect calls to *libc* functions to use only the first translation mechanism.

3.3.11 Floating-Point

In floating-point emulation, there are also some RISC-V instructions that are straightforward to translate, such as additions and subtractions, and others that have no direct equivalent in LLVM IR, such as the sign-injection instructions. Load and store implementation is very similar to the integer equivalents. For some of the more complex instructions, such as *fsqrt* (square root) and *fmadd* (fused multiply-add), LLVM provides intrinsic calls, that implement it in the most efficient way for the target architecture. Other fused operations, however, are not so common and have no direct LLVM correspondent and so they must be decomposed in multiple instructions, such as in the fused multiply-subtract case, but are otherwise straightforward to translate. Conversions and casts from floating-point to integer and vice-versa also do not present much problem to translate, except for out-of-range and some special inputs, such as ∞ and NaN, that require a distinct conversion method, mostly by requiring a specific result value instead of leaving it undefined. While the check and handling of these special cases is implemented in our SBT engine, it sometimes introduces considerable overhead. As these never occur in our benchmarks and would otherwise look more like compiler or hand-written assembly issues, these checks are turned off by default.

The sign-injection instructions, on the other hand, are more tricky to implement. While *fsgnj* (sign-injection) can be mapped to LLVM *copysign* intrinsic, and *fsgnjn* (negated sign-injection) can be translated to *copysign* with the second operand negated, LLVM IR has nothing close to *fsgnjx* (xor sign-injection). In this last case, the result must be the same value of the first operand, but with the sign bit replaced by the result of an *xor* of the sign bits of the first and second input operands. This is implemented as follows: cast the floating-point input operands to integers, a and b; *xor* a and b, producing c, that has the correct sign bit; mask all but the sign bit from c with an *and*; mask none but the sign bit from a with an *and*; perform an *or* of a and c; cast the result to floating-point. Thus, it can be noted that translating the general form of *fsgnjx* results in many LLVM IR instructions. But, by analysing generated code, we discovered that most uses of *fsgnjx* are made to perform the *abs* operation. This way, by identifying and optimizing this common case, replacing the slower general implementation above by LLVM IR *abs* intrinsic, the cost of excuting the translated *fsgnjx* instruction becomes low on average.

Another point worth of mentioning about RISC-V floating-point instructions is the rounding mode operand, present in several of them, allowing the operation result to be rounded in a few distinct ways. The fixed rounding modes did not present a considerable translation challenge, as they may only introduce one or more extra instructions, in cases where they differ from the standard LLVM IR or target host rounding mode. Also, we verified that most instructions translated by our SBT engine use either the default rounding mode, that rounds to the nearest value, ties to even (RNE), or use the dynamic rounding mode, that, while specified in a separate register, defaults to RNE too, meaning no extra overhead due to the rounding mode feature, in most cases. While all rounding modes seen in our benchmarks are supported by our translator, we have ignored the dynamic rounding mode feature, assuming that the register that specifies it (*fcsr*) is always equal to RNE. This approach worked well in all of our benchmarks, as none tried to change the dynamic rounding mode during execution. Emulating this feature correctly, however, can be costly and complex, because of the extra check to RISC-V *fcsr* register that must be performed. On the other hand, the fact that none of our benchmarks relied on this feature seems to indicate that its use is quite rare.

3.4 Translation Example

To illustrate several of the translation steps discussed above, consider the following C code in Listing 3.1:

```
int dot_prod(int a[2], int b[2])
{
    return a[0] * b[0] + a[1] * b[1];
}
```

Listing 3.1: Dot Product in C

Function *dot_prod()* performs the dot product of two vectors, *a* and *b*, with 2 elements each, and returns the result. Listing 3.2 shows the RISC-V code generated by GCC 7.3.

```
        .text
        .align  2
        .globl  dot_prod
        .type   dot_prod, @function
dot_prod:
        lw      a5, 4(a0)
        lw      a4, 0(a0)
        lw      a2, 0(a1)
        lw      a3, 4(a1)
        mul     a0, a4, a2
        mul     a5, a5, a3
        add     a0, a0, a5
        ret
```

Listing 3.2: Dot Product in RISC-V

As specified by RISC-V ABI, the two arguments that this function take are passed in registers *a0* and *a1*. Therefore, the first four instructions in Listing 3.2 load all vector elements from *a* and *b* into RISC-V registers. Next, two multiplications and one addition compute the dot product, writing the result into *a0* — the RISC-V ABI register used to return word-sized integer values — and the function returns. Next, the RISC-V code for

Dot Product is used as the input of our SBT engine, that translates it to (unoptimized) LLVM IR, using the Locals register mapping mode, as shown in Listing 3.3.

```
define void @dot_prod() {
bb0:
  ; RISC-V registers are translated to local variables
  %lrv_x1 = alloca i32
  %lrv_x10 = alloca i32
  %lrv_x11 = alloca i32
  %lrv_x12 = alloca i32
  %lrv_x13 = alloca i32
  %lrv_x14 = alloca i32
  %lrv_x15 = alloca i32

  ; At function entry, the local variables that emulate RISC-V
  ; registers are synchronized (or initialized in this case)
  ; with the values of global RISC-V register variables
  %0 = load i32, i32* @rv_x1
  store i32 %0, i32* %lrv_x1
  ; ...
  %9 = load i32, i32* @rv_x10
  store i32 %9, i32* %lrv_x10
  %10 = load i32, i32* @rv_x11
  store i32 %10, i32* %lrv_x11
  %11 = load i32, i32* @rv_x12
  store i32 %11, i32* %lrv_x12
  %12 = load i32, i32* @rv_x13
  store i32 %12, i32* %lrv_x13
  %13 = load i32, i32* @rv_x14
  store i32 %13, i32* %lrv_x14
  %14 = load i32, i32* @rv_x15
  store i32 %14, i32* %lrv_x15
  %15 = load i32, i32* @rv_x16
  ; ...
  %30 = load i32, i32* @rv_x31
  %31 = load double, double* @rv_f0
  %32 = load double, double* @rv_f1
  ; ...
  %62 = load double, double* @rv_f31

  ; Next, each RISC-V instruction is translated to LLVM IR

  ; lw   a5, 4(a0)
  %a0_ = load i32, i32* %lrv_x10
  %63 = inttoptr i32 %a0_ to i8*
  %64 = getelementptr i8, i8* %63, i32 4
  %65 = bitcast i8* %64 to i32*
```

```
%66 = load i32, i32* %65
store i32 %66, i32* %lrv_x15

; lw  a4, 0(a0)
%a0_1 = load i32, i32* %lrv_x10
%67 = inttoptr i32 %a0_1 to i8*
%68 = getelementptr i8, i8* %67, i32 0
%69 = bitcast i8* %68 to i32*
%70 = load i32, i32* %69
store i32 %70, i32* %lrv_x14

; lw  a2, 0(a1)
%a1_ = load i32, i32* %lrv_x11
%71 = inttoptr i32 %a1_ to i8*
%72 = getelementptr i8, i8* %71, i32 0
%73 = bitcast i8* %72 to i32*
%74 = load i32, i32* %73
store i32 %74, i32* %lrv_x12

; lw  a3, 4(a1)
%a1_2 = load i32, i32* %lrv_x11
%75 = inttoptr i32 %a1_2 to i8*
%76 = getelementptr i8, i8* %75, i32 4
%77 = bitcast i8* %76 to i32*
%78 = load i32, i32* %77
store i32 %78, i32* %lrv_x13

; mul a0, a4, a2
%a4_ = load i32, i32* %lrv_x14
%a2_ = load i32, i32* %lrv_x12
%79 = mul i32 %a4_, %a2_
store i32 %79, i32* %lrv_x10

; mul a5, a5, a3
%a5_ = load i32, i32* %lrv_x15
%a3_ = load i32, i32* %lrv_x13
%80 = mul i32 %a5_, %a3_
store i32 %80, i32* %lrv_x15

; add a0, a0, a5
%a0_3 = load i32, i32* %lrv_x10
%a5_4 = load i32, i32* %lrv_x15
%81 = add i32 %a0_3, %a5_4
store i32 %81, i32* %lrv_x10

; jalr zero, 0(ra)  (a.k.a. ret)
%ra_ = load i32, i32* %lrv_x1
```

```
  ; At function exit, the local variables that emulate RISC-V
  ; registers are synchronized with the global RISC-V
  ; register variables
  %82 = load i32, i32* %lrv_x1
  store i32 %82, i32* @rv_x1
  %83 = load i32, i32* %lrv_x10
  store i32 %83, i32* @rv_x10
  %84 = load i32, i32* %lrv_x11
  store i32 %84, i32* @rv_x11
  %85 = load i32, i32* %lrv_x12
  store i32 %85, i32* @rv_x12
  %86 = load i32, i32* %lrv_x13
  store i32 %86, i32* @rv_x13
  %87 = load i32, i32* %lrv_x14
  store i32 %87, i32* @rv_x14
  %88 = load i32, i32* %lrv_x15
  store i32 %88, i32* @rv_x15

  ; return
  ret void
}
```

Listing 3.3: Dot Product routine translated from RISC-V to LLVM IR (unoptimized)

The first LLVM IR instructions declare local variables, used to emulate local copies of RISC-V registers inside the function, that may be allocated in the stack or promoted to LLVM registers later. The load and store instructions that follow initialize these local variables from the global variables that emulate RISC-V registers, that in the Locals mode are used only to synchronize local variables of functions, whenever the execution is switched from one to another. Note that, although the values of all registers are loaded from global variables, the stores occur only to those local variables that are used in the function. This is because the SBT engine removes all loads and stores to emulated registers that are only used in synchronization points, except for the initial load that is left behind, but that is easily removed by liveness analysis during the optimization step.

After the local variables have been initialized, the translation of the first function instructions begin, that in this case is the load of the vector elements of arguments *a* and *b*. In this paragraph we explain the translation of the first RISC-V instruction (*lw a5, 4(a0)*), as the translation of the other 3 RISC-V *lw* instructions are nearly identical. First, the value of the base address register, *a0* (a.k.a. *x10*), is loaded from the local variable that emulates it. Next, the loaded 32-bit integer value is casted to an 8-bit integer pointer, to which the byte address offset specified in the RISC-V instruction is added. The resulting address is then casted to a 32-bit integer pointer, that can then be used to perform a 32-bit integer load from memory, that is then stored into the output register *a5* (a.k.a. *x15*), as specified by the RISC-V instruction being translated. An interesting thing to note is that the translation of these RISC-V *lw* instructions produces some unecessary LLVM loads, as in the case of *a0*, that is loaded in the translations of the first and second *lw* instructions, even though it has already been loaded during a previous register synchronization instruction. This happens because our SBT engine, in most cases, limits its scope to only the instruction being translated. But these kinds of redundant operations do not hurt performance, because they can be easily eliminated by LLVM *opt*, used in the sequence to optimize the translated IR — as we shall see later — before proceeding to actual target code generation.

The translation of the *mul* and *add* instructions are straightforward: the input register operands are loaded, the arithmetic operation is performed and the result is stored into the output register. The last RISC-V instruction translated is *ret*, that is actually a pseudo-instruction, mapped to *jalr zero, 0(ra).* As this is the function exit point, all registers that were modified during its execution, using local variables, must be written to their global counterparts before the function can actually return. This is done by the loads from local variables and the stores to global ones, and then the function returns. Note that, as specified by RISC-V ABI, the caller expects the result to be returned in *a0*, which is why the LLVM *ret* instruction returns no value.

Next, in Listing 3.4, we present the result of the LLVM optimization step, using LLVM *opt* with the -O3 flag, in the LLVM IR just shown.

```
; Function Attrs: norecurse nounwind
define void @dot_prod() local_unnamed_addr #1 {
bb0:
  ; register synchronization (function entry)
  %0 = load i32, i32* @rv_x10, align 4
  %1 = load i32, i32* @rv_x11, align 4

  ; lw  a5, 4(a0)
  %2 = zext i32 %0 to i64
  %3 = inttoptr i64 %2 to i8*
  %4 = getelementptr i8, i8* %3, i64 4
  %5 = bitcast i8* %4 to i32*
  %6 = load i32, i32* %5, align 4

  ; lw  a4, 0(a0)
  %7 = inttoptr i64 %2 to i32*
  %8 = load i32, i32* %7, align 4

  ; lw  a2, 0(a1)
  %9 = zext i32 %1 to i64
  %10 = inttoptr i64 %9 to i8*
  %11 = inttoptr i64 %9 to i32*
  %12 = load i32, i32* %11, align 4

  ; lw  a3, 4(a1)
  %13 = getelementptr i8, i8* %10, i64 4
  %14 = bitcast i8* %13 to i32*
  %15 = load i32, i32* %14, align 4
  ; dot product
  %16 = mul i32 %12, %8
  %17 = mul i32 %15, %6
  %18 = add i32 %17, %16

  ; register synchronization (function exit) and ret
  store i32 %18, i32* @rv_x10, align 4
  store i32 %12, i32* @rv_x12, align 4
  store i32 %15, i32* @rv_x13, align 4
  store i32 %8, i32* @rv_x14, align 4
  store i32 %17, i32* @rv_x15, align 4
  ret void
}
```

Listing 3.4: Dot Product routine translated from RISC-V to LLVM IR (optimized)

We can observe that LLVM was able to optimize several parts of the initial IR and reduce considerably the code size. Unnecessary loads from global variables were eliminated, all local variables used to emulate RISC-V registers were promoted to LLVM virtual registers, redundant loads emitted by the SBT engine were removed and stores to local variables overwritten before their use were removed.

The last block of code performs register synchronization, and since it involves writing to global variables, that is a side effect visible outside the function, these writes cannot be omitted. In ABI mode, however, we take advantage of RISC-V ABI information, followed by current RISC-V compilers, to omit the synchronization of volatile registers not used to transfer information between functions, such as *a1* to *a5* registers.

Finally, in Listing 3.5, we have the final x86 assembly code, produced from the optimized LLVM IR above.

```
        .globl  dot_prod                    # -- Begin function dot_prod
        .p2align        4, 0x90
        .type   dot_prod,@function
dot_prod:                                   # @dot_prod
# %bb.0:                                    # %bb0
        pushl   %esi
        # reg sync
        movl    rv_x10, %eax
        movl    rv_x11, %ecx
        # loads
        movl    (%eax), %edx
        movl    (%ecx), %esi
        movl    4(%ecx), %ecx
        movl    4(%eax), %eax
        # reg sync
        movl    %esi, rv_x12
        # dot product
        imull   %edx, %esi
imull    %ecx, %eax
addl     %eax, %esi
# reg sync
movl     %esi, rv_x10
movl     %ecx, rv_x13
movl     %edx, rv_x14
movl     %eax, rv_x15
# ret
popl     %esi
retl
```

Listing 3.5: Dot Product routine translated from RISC-V to x86

Besides the additional prologue and epilogue instructions, the remaining, non-omitted instructions map almost directly to the LLVM IR instructions above. That is not always the case, however, especially in more complex programs. In this example, three types of instructions were omitted. The first were the LLVM cast instructions, that did not generate extra x86 instructions, as x86, like most architectures, does not have multiple pointer types and does not differentiate integers from pointers. The second were the zero extensions, that are implicitly performed by several x86 instructions. The third were the *getelementptr* instructions, that were performed using x86 base-plus-offset addressing mode, as part of the *movl* instructions.

It is worth to recall the need of the register synchronization instructions, present in Listing 3.5. As the guest instructions operate on RISC-V registers and, in Locals mode, they are mapped to local function variables, at function entry/exit points they must be loaded/saved from/to somewhere. In our SBT engine implementation, we use a global register file for this purpose. In this way, the caller saves its local register file to the global one, from where the callee initializes its local register file, and, before returning, the callee writes all RISC-V registers that it modified to the global register file, in order for them to become visible to other functions. Thus, the register synchronization instructions represent an overhead inherent to the Locals register mapping technique employed by our translator. In ABI mode, however, only non-volatile RISC-V registers are saved, which in the case of Listing 3.5 would eliminate the need to write back registers *x12* to *x15*.

Chapter 4

Experimental Setup and Infrastructure

In order to quantify the performance overhead introduced by the SBT, we compared the performance of benchmarks emulated with SBT against the performance of their native execution. For the benchmarks, we have used MiBench [39], which provides a reasonable set of programs with sufficient variation to cover most CPU emulation aspects. The experiments were performed on two host ISAs: x86 and ARM.

The x86 machine used was an Intel Core i7-6700K, running at 4.0GHz, with 32GiB of DDR4 memory, in two modules of 16GiB, one in each memory bank. The operating system (OS) used was Debian 9, with Linux Kernel 4.9.0-8-amd64. Although a 64-bit OS was used, it also supports running 32-bit binaries, through kernel and multilib support, so that in our experiments only 32-bit binaries and libraries were used.

As for the ARM machine, we used a Raspberry Pi 3 Model B, that has a Quad Core ARM 64-bit CPU running at 1.2GHz and with 1GiB of RAM. The operating system used was Raspbian 9, with Linux raspberrypi 4.14.62-v7+ (armv7l) kernel. Note that although the ARM CPU supported 64-bit, the OS and all programs and libraries used 32-bit only.

As the selected benchmarks are designed to stress mainly the CPU and memory, the rest of the hardware should not interfere with the results. However, after noticing considerable variation between each run in some benchmarks, we decided to move all input and output files to a RAM disk, in order to mitigate what seemed to be an interference caused by disk buffers in the operating system. As we have confirmed, this resulted in a much smaller variation in the execution times of some benchmarks.

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 4.1.

The first compilation flow, Native (GCC), was used to produce native x86 and ARM binaries using the GCC 7.3 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[1] and the final binary was assembled and linked by GCC 6.3.0. We used this combination because LLVM's assembler and linker did not support RISC-V binaries during our experiments. 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

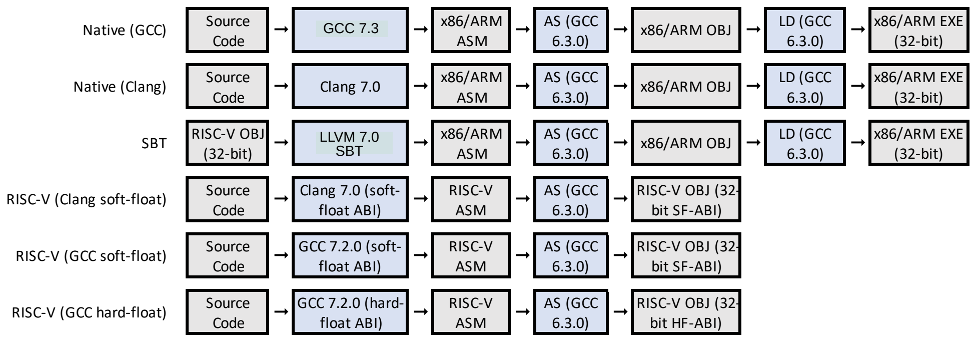


Figure 4.1: Code generation flows.

the following flows: three to produce RISC-V binaries (RISC-V OBJ) from benchmarks' source code (Clang soft-float and GCC soft and hard-float) with another to translate the RISC-V binaries to native code, using our SBT engine based on LLVM 7.0[1].

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 (7.3.0) to compile the source code (C) to ASM and then GCC (6.3.0) 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, that instructs the compiler to use SSE or better (e.g. AVX) instructions when emitting floating-point code, but not legacy 387 instructions. As for ARM, we targeted the *armv7-a* processor family, with *vfpv3-d16* floating-point instructions, as this is a perfect match for Raspbian 9 distribution for *armhf*.

4.1 Measurement Methodology

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 small SD.

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 arithmetic mean of the percentage of execution time spent in the main binary, thus excluding the time spent in shared libraries, such as *libc*. The final run time of each benchmark was then multiplied by this percentage.

Our experimental results are presented in terms of the slowdown that the translated binary shows in relation to the native one. In other words, it is the number of times that the translated binary is slower than the native one. It is calculated by dividing

[1]When the experiments were performed, LLVM/Clang 7.0 had not been released yet, so 7.0 here actually refers to LLVM/Clang master Git branch as it was on July 09 of 2018, commit ae0f1dc9280. We have used this version because LLVM 6 lacked many RISC-V back-end improvements.

the execution time of the translated binary, as explained previously, by the execution time of the native binary. In this way, the slowdown calculation can be summarized by the following formula: $x = tt/tn$, where x is the slowdown, tt the execution time of the translated binary and tn the execution time of the native binary. Thus, the higher the slowdown the worse is the performance of the translated program. Also, note that a slowdown of 1 means that the translated binary is as fast as the native one.

Beyond the execution time (our main metric), we also collected other performance metrics, such as: task-clock, context-switches, cpu-migrations, page-faults, cycles, instructions retired, number of branches and branch-misses. These are the default performance counters used by Linux *perf*, and, according to Bitzes and Nowak [40], the overhead of using up to 8 *perf* events in counting mode, as in our case, is negligible for most purposes, staying under 0.5% in their experiments. When more events were needed, such as cache-misses in different cache levels, these were collected in separate runs, that were not considered when calculating the slowdowns. These performance metrics helped us in our qualitative analysis of emulation performance and in understanding good and bad results of individual benchmarks. Coupled with code generation and translation analysis, it enabled us to identify code constructions and compiler optimizations that resulted in low quality translated code, and thus presented slower execution times, when compared to native.

4.2 GCC vs Clang and Soft-Float vs Hard-Float ABI

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-precision 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, for x86, 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. For ARM, we used only GCC and hard-float ABI in the experiments, as these gave the best results.

4.3 RISC-V Configuration

We chose to use the RISC-V 32-bit IMFD variant, composed by the instruction sets of the integer base I (mandatory), standard extensions M (integer multiplication and division), F and D (floating-point operations with single and double precision). This choice was made mainly because:

- 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.
- To make it easier to compare RISC-V with OpenISA — remembering that this work uses the same approach and methodology used in OpenISA's work — that is also 32-bit, and ARM 32-bit.

In future works we intend to experiment with the 64-bit variant of RISC-V.

4.4 Impact of Compilers and ABIs on SBT Performance

Before going deeper into the experimental results, first it is important to explain how we arrived at the experimental setup that gave the best results. Although part of it was already explained in the previous section, some other choices were guided by experimentation, as we shall see now.

4.4.1 GCC vs Clang

After having observed a couple of issues with LLVM RISC-V back-end and, especially after noticing that it did not support the use of a hard-float ABI, our interest in using the more mature GCC for RISC-V compiler instead of Clang/LLVM for RISC-V grew. In this setup, our translator still uses LLVM infra-structure and libraries to perform the translation, but now taking a RISC-V input binary produced by RISC-V GCC instead.

It is important to highlight that, as discussed in Section 4.2, when we use the terms *hard-float ABI* and *soft-float ABI* in the text, we are referring only to how floating-point arguments are passed/returned to/from called functions, as floating-point instructions are used in both ABIs. Also note that this applies only to RISC-V code, as native x86 and ARM binaries always use a hard-float ABI.

Figure 4.2 shows the slowdown caused by our SBT, when compared to native execution, of the translation of input binaries produced by Clang and GCC soft-float ABI on x86. In StringSearch case, most values were so high that they did not fit the graph's upper limit. For Clang, StringSearch's measured slowdowns were 16.59/3.76/4.18x (Globals/Locals/ABI), while for GCC, they were: 15.23/2.75/2.78x. Note that, the higher the value, the worse the emulation performance. It can be seem that in some cases our SBT engine performs better when emulating code produced by Clang while in others it performs better with GCC, although for benchmarks that do not use floating-point operations (from Dijkstra to Blowfish-Decode), emulation performance stays close to native performance in Locals and ABI modes, with a few exceptions (e.g. StringSearch) that we analyze later on Chapter 5. Among others, FFT performed poorly, but as we discuss next, this is due to the lack of a hard-float ABI for RISC-V code in Clang, as this benchmark is one of the heaviest users of floating-point operations, having its performance greatly improved when using the hard-float ABI. The high overhead of other benchmarks are explained further, in Chapter 5.

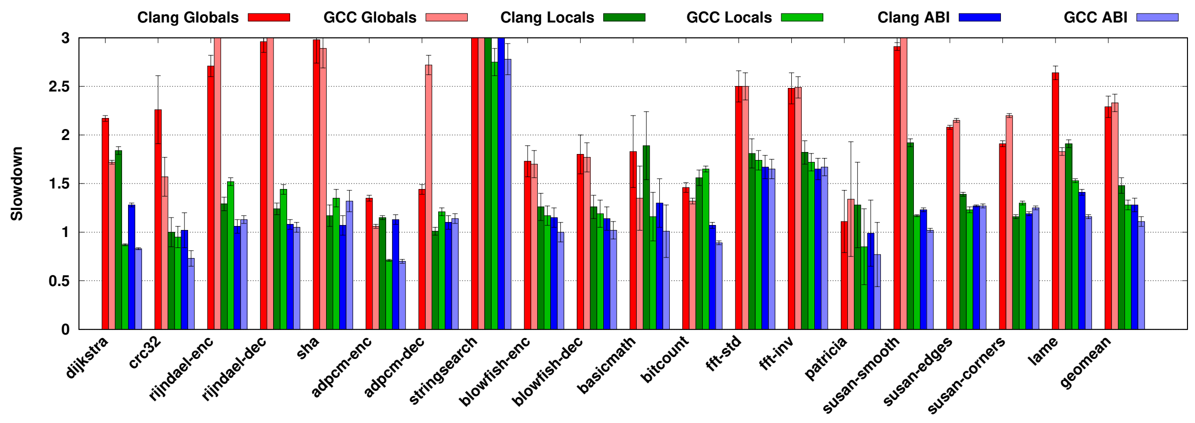


Figure 4.2: Slowdown of benchmarks compiled with Clang and GCC soft-float RISC-V back-ends.

This GCC vs Clang experiment showed that although GCC has a more mature back-end than Clang, our SBT engine performance when translating RISC-V binaries with soft-float ABI produced by both compilers was close, although we can see a small improvement in the average performance of the benchmarks, just by using GCC to produce RISC-V code instead of Clang.

4.4.2 Soft-Float ABI vs Hard-Float ABI

After comparing Clang with GCC previously, both using soft-float ABI, it is interesting to observe how much performance is gained by switching to the hard-float ABI. Remembering that, as discussed in Section 4.2 and in previous subsection, the hard-float ABI differs from the soft-float ABI only in how floating-point arguments are passed to functions, where hard-float ABI allows floating-point registers to be used and soft-float do not. Figure 4.3 shows the slowdown caused by our SBT, when compared to native execution, of the translation of input RISC-V binaries produced by GCC soft-float ABI vs GCC hard-float ABI.

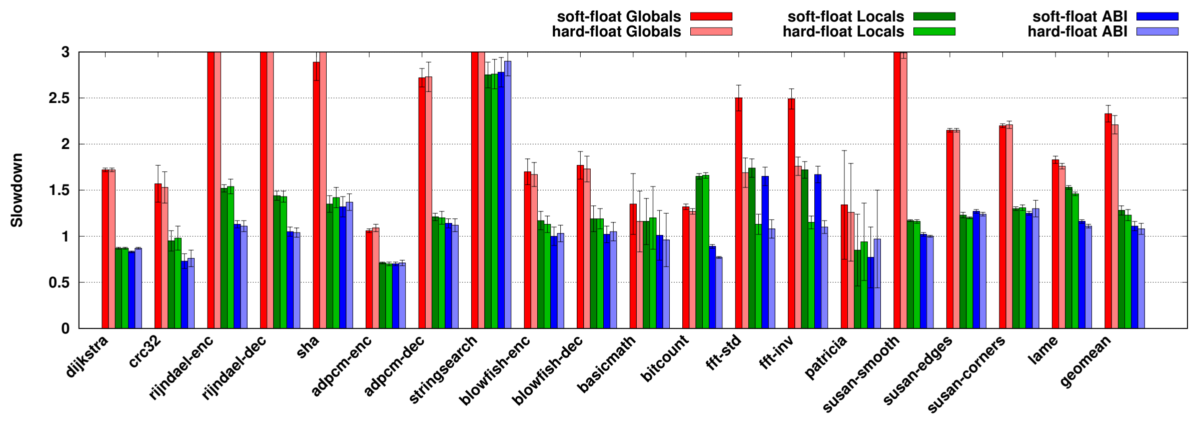


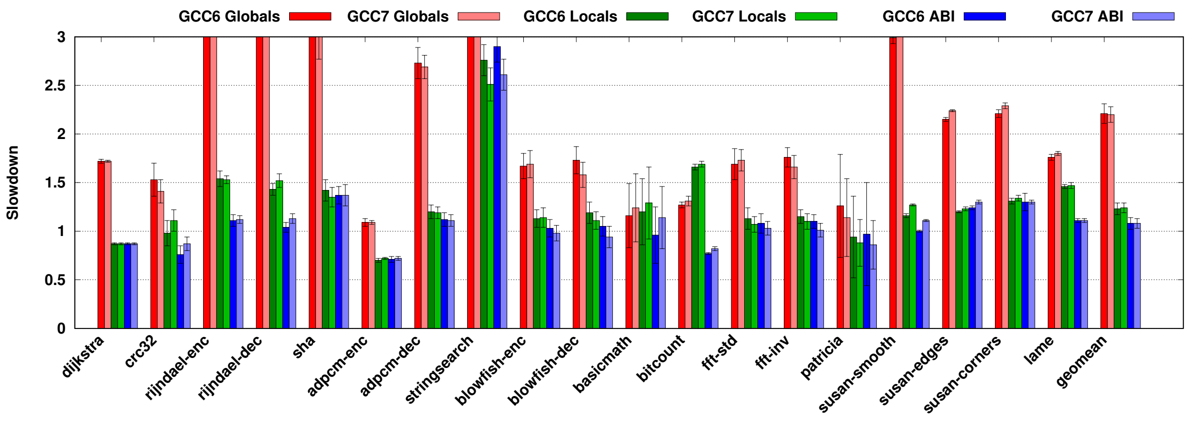
Figure 4.3: Slowdown of benchmarks compiled with GCC soft-float ABI vs hard-float ABI.

We can see a major improvement in FFT and some smaller improvements in a few other benchmarks, such as LAME and BitCount, when using RISC-V hard-float ABI, remembering that native binaries always use a hard-float ABI. This experiment indicates that RISC-V soft-float ABI is indeed less efficient than the hard-float ABI, as our SBT engine is able to produce higher quality code with the latter, and, even though in our benchmarks only FFT is heavily affected by this, no benchmark is impaired by using hard-float ABI instead of soft-float ABI. Furthermore, as the Clang RISC-V back-end does not have, until the date, support to hard-float ABI, we are going to use solely GCC in the next experiments.

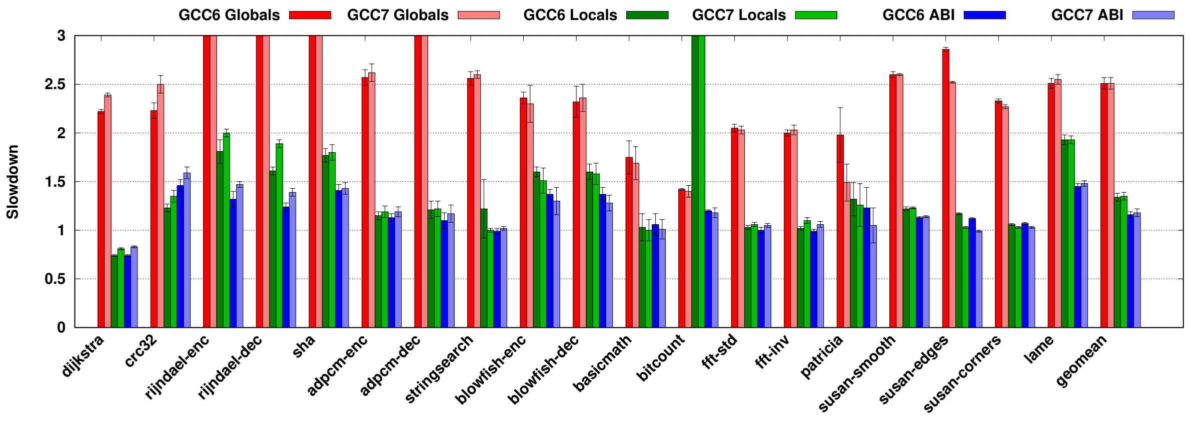
4.4.3 GCC 6.3.0 vs GCC 7.3.0

In this chapter we mention the usage of GCC 6.3.0 to build native binaries, while GCC 7.3.0 was used for RISC-V, as there was practically no support for RISC-V in GCC 6.3.0. This however leads to some uncertainty about different compiler versions affecting the performance results. To rule this out, we have compiled and translated all benchmarks for x86 and ARM using GCC 7.3.0 and compared it with the results obtained with GCC 6.3.0. As in the time of this writing GCC 7.3.0 was not available for Debian 9 (stretch) — the OS used in our hosts — but only for Debian 10 (buster) — that was not released as stable yet — we choose to build and run the native benchmark binaries using Debian 10 toolchain and libraries, instead of building our own GCC 7.3.0 toolchains for x86 and ARM from source. To simplify this task, we have used Docker to be able to run Debian 10 from our Debian 9 hosts. Also note that, at the time of this writing, Debian 10 used Linux 4.18, which was then installed and used on our Debian 9 hosts when performing the GCC 7.3.0 experiments in the containerized Debian 10 environment. Figure 4.4 shows the results of this experiment. The slowdowns of BitCount on ARM, in Locals mode, that were greater than the graph limit, were of 3.16x and 3.10x, for GCC 6.3.0 and GCC 7.3.0, respectively.

It can be seen that the differences in performance due to using GCC 6.3.0 for compiling native binaries instead of GCC 7.3.0 are minimal, as the differences in the geometric means



(a) x86



(b) ARM

Figure 4.4: Comparison of slowdowns obtained when compiling native binaries with GCC 6.3.0 vs GCC 7.3.0.

are, at most, 0.02x. For this reason, and to ease the experimentation, all other experiments were performed solely on Debian 9 using GCC 6.3.0 as the native compiler.

Chapter 5

Experimental Results

In this chapter we present the performance of our RISC-V SBT engine in terms of slowdown when compared to native execution (GCC based RISC-V binaries translated by our SBT engine compared to GCC based native binaries). 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 the Globals, Locals and ABI translation schemes.

In Figure 5.1 we can see the slowdowns obtained by translating MiBench benchmarks from RISC-V code to x86 and ARM. When translating RISC-V to x86, we obtained an average slowdown of 2.21x, 1.23x, and 1.08x, for Globals, Locals, and ABI, respectively. On ARM, the average slowdowns were 2.51x, 1.34x, and 1.16x. Moreover, Locals and ABI performance outstands the Globals performance in almost 2-fold, showing the importance of the register mapping approach. The highest slowdown seen in Locals mode is that of BitCount on ARM, of 3.16x. In the following paragraphs we analyze the results of each benchmark.

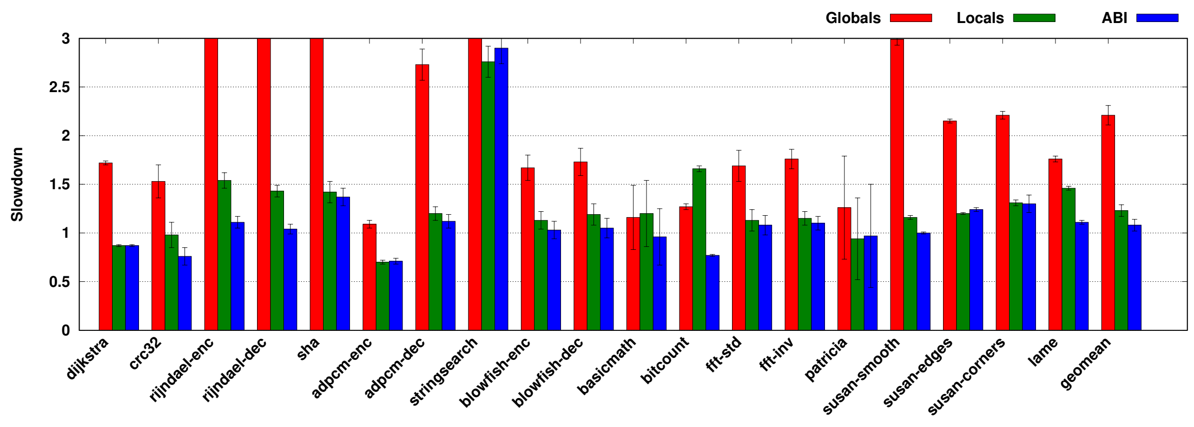
ADPCM-Decode, FFT-Standard, and FFT-Inverse show near-native performance, both on x86 as on ARM, in Locals and ABI modes. On ARM, ADPCM-Encode also shows near-native performance.

ADPCM-Encode on x86 and Dijkstra had better performance than native. Our analysis indicates that this is due to a better optimization from the LLVM infrastructure used by our translator when compared to GCC, for these specific benchmarks.

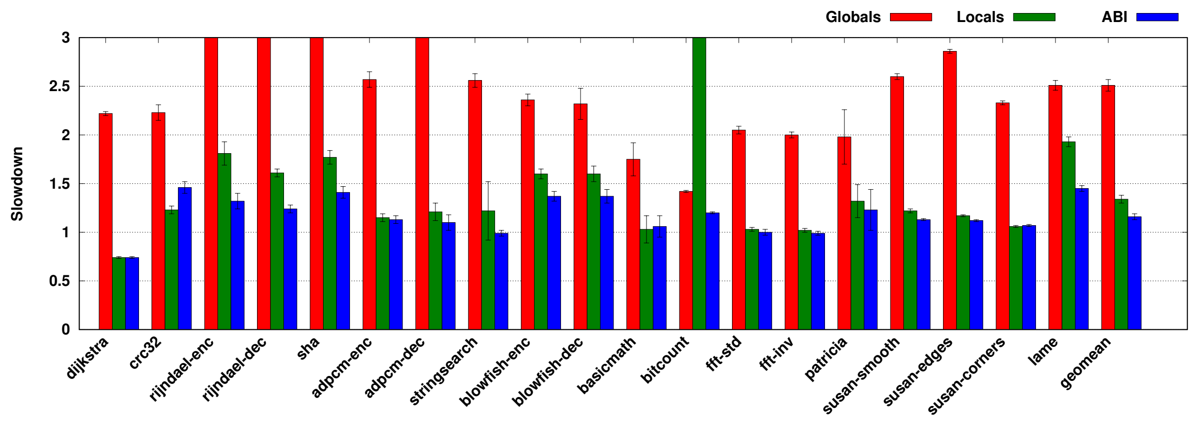
CRC32 performs very well on x86, while on ARM it reaches 1.23x in Locals mode. What stands out in this case, however, is that ABI mode is considerably worse than Locals. This was unexpected, as the ABI mode is basically an optimization of Locals, that takes advantage of RISC-V ABI to reduce the number of registers copied in synchronization points. Analyzing the generated code, we found out that, in CRC32 case, this mode somehow made LLVM optimization phase produce IR with different ordering, which made the subsequent code generation phase produce more code in the main loop (the hottest spot of this benchmark) for ABI mode, when compared to Locals.

StringSearch has a good result on ARM, but a very high overhead on x86. As we shall see next, however, this is caused by the compiler not being able to vectorize the translated code.

Rijndael-Encode, Rijndael-Decode, and SHA show a high slowdown on both x86 and



(a) x86



(b) ARM

Figure 5.1: MiBench slowdown of RISC-V translated binaries.

ARM. Blowfish-Encode and Blowfish-Decode present a good result on x86 but high overheads on ARM. We investigate the causes of these high overheads in the next sections.

Overall, BasicMath and Patricia benchmarks show good performance results in both x86 and ARM, except for Patricia Locals on ARM, that reaches 1.32x. But their high error range, especially on x86, calls the attention. The problem is that the percentage of time these benchmarks spend in the main binary is very low: oscillating from 2% to 4% on x86 and ARM. Instead, most of the time is spent in *libc* calls. As our slowdown calculation takes into account only time spent in the main binary, small variations in this low percentage result in large variations in the execution time considered. This gets worse on x86 because it is able to execute the benchmarks much faster than our ARM host. Note, however, that BasicMath and Patricia are the two benchmarks who spend the most time in *libc*, which is not the case for most benchmarks, as discussed in Section 3.1.

In BitCount we can see that Locals overhead is very high, even worse than Globals mode. By comparing it with ABI mode however, it becomes clear that register synchronization represents a large portion of its execution time. We investigate this case in more details later.

Susan-Smooth and Susan-Edges show good performance on both x86 and ARM. Susan-Corners has a very good result on ARM, while being a bit high on x86. As we shall see further, this is caused by missed vectorization optimizations by the compiler, as in StringSearch case.

For LAME on x86, we observed a high overhead in Locals mode, but comparing it with ABI indicates that most of it is caused by register synchronization. On ARM, however, even the ABI result is bad, indicating that there are other overhead sources besides register synchronization. We investigate this further in a later section.

5.1 Translated Code not Vectorized

While investigating the cause of some major overheads, like that seen in StringSearch, generated code analysis revealed that several loops were not being vectorized from the IR produced by our SBT engine, while they were when compiling the program from source. Further investigation is needed to fully understand the causes of this issue, but by comparing native IR with translated IR, it seems that LLVM is only capable of vectorizing code if the corresponding IR is at a given format it supports/expects. In order to evaluate the performance impact of code not being vectorized, we have performed a native run of MiBench with vectorization disabled and compared it to previous results.

It turns out that completely disabling vectorization in LLVM and GCC can be tricky. For this reason, we chose to limit the x86 instructions that code generation can use up to MMX only, and compared it with our previous setup that used up to AVX extensions, as shown in Figure 5.2.

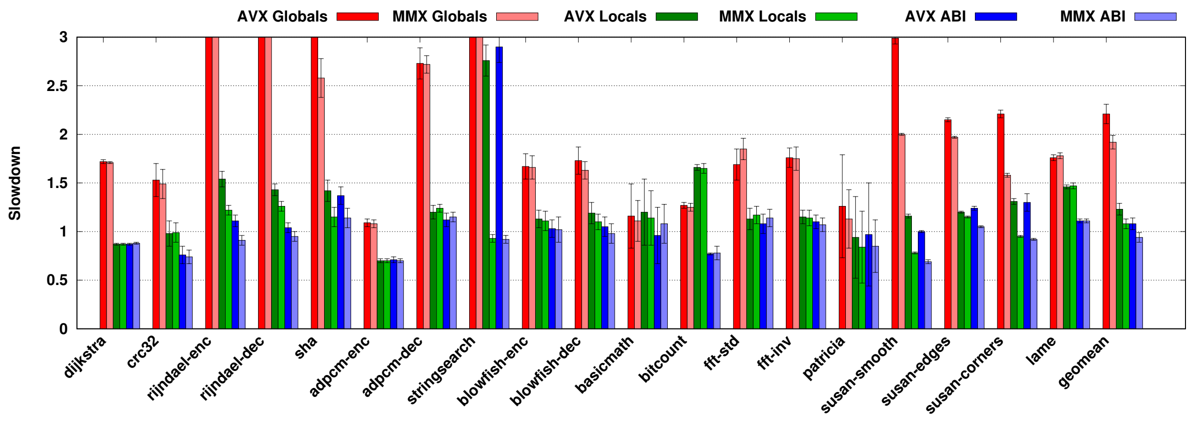


Figure 5.2: Comparison of slowdowns for x86 with AVX extensions enabled vs limited to MMX.

We can see that, in StringSearch case, practically all overhead was caused by this missed vectorization opportunity. This is due to its hottest spot being a loop that was completely vectorized in native compilation while not vectorized at all when translated. We can also note significant differences in Rijndael-Encode, SHA, Susan-Smooth, Susan-Edges, and Susan-Corners, while for the remaining benchmarks the performance stayed almost the same.

We expect that, the planned RISC-V vectorization extension will help with this, when it is ready, because then the SBT engine could just translate RISC-V vector instructions to LLVM IR vector instructions.

5.2 The Rijndael Case

The main overheads of Rijndael on x86 are:

- Register synchronization.
- Missed LLVM vectorization. There is an important loop in this benchmark that performs the load, *xor* and store of a 16-byte buffer. The code generated when compiling it to RISC-V completely unrolls the loop, performing multiple loads, *xors* and stores. On x86, instructions *vload*, *vxor* and *vstore* are used. The problem is that LLVM *opt* and *llc* are unable to infer that the multiple loads, *xors* and stores, emited by our SBT engine when translating RISC-V code, could be replaced by vector instructions, when generating code for x86.

On ARM, the main overheads are:

- Register synchronization.
- At *encfile()/decfile()* functions, the loop to *xor* 16 positions of *inbuf* with *outbuf* is unrolled, using a large number of registers. When translating from RISC-V to ARM, the resulting code ends up performing a lot of spills, because it is unable to promote all emulated registers on local variables to host registers, as ARM has a smaller register set.

5.3 The SHA Case

On both x86 and ARM, SHA's main source of overhead is somewhat similar to that of Rijndael: too many spills when a large number of registers is used. In SHA's case, it happens at *sha_final()*, at the 2 calls to *byte_reverse()*. On RISC-V, *byte_reverse()* code generation produces a series of loads followed by stores, from/to registers directly to an offset at *SHA_INFO*'s data array, using most of the 32 RISC-V registers. When the code is translated to x86 or ARM, however, that have a much smaller register set, there is a huge number of spills and reloads. It is worth to note that we did not explicitly specify the register allocator to be used, but, according to LLVM documentation, the greedy allocator is used for optimized code. Native x86/ARM code performs better because code generation limits more the loop unrolling, to make better usage of the number of host registers available.

To test this hypothesis, we have performed an experiment that consists of adding some *pragmas* around *byte_reverse()* function, to disable loop unrolling in it, as shown in Listing 5.1.

This change greatly improved performance results on x86, in which Locals slowdown went from 1.61x to 1.19x and ABI went from 1.24x to 1.09x. All results were measured with vectorization already disabled (using only up to MMX instructions). Note that the results presented in this section were measured from separate runs, and thus they are slightly different from those of the previous graphs. On ARM, the performance did not improve much on Locals mode, going from 1.64x to 1.59x, because with loop unrolling disabled the compiler did not *inline byte_reverse()* in *sha_final()*, which increased register synchronizations. In ABI mode however, a substantial improvement can be seen, with the slowdown going from 1.48x to 1.18x.

```
#pragma GCC push_options
#pragma GCC optimize(
    "no-unroll-loops,no-peel-loops,no-tree-loop-optimize")

void byte_reverse(...) {
/* ... */
}

#pragma GCC pop_options
```

Listing 5.1: Disabling loop unrolling at byte_reverse()

Additionally, on x86, the missed vectorization optimization is also among the main overhead causes. Another one is that translated code uses more instructions at *sha_transform()*. On native compilation, x86 seems to move pointer values directly to registers and makes memory accesses through them, taking full advantage of x86 more complex addressing modes, while RISC-V code needs to break these accesses in more parts, performing calculations that could be done directly in a single *mov* instruction for x86. This way, when translating RISC-V to LLVM IR and then back to x86, LLVM is unable to deduce that those broken up memory accesses address calculations could be grouped into fewer instructions/accesses, by taking advantage of x86 more complex addressing modes.

5.4 The BlowFish Case

Blowfish performs well on x86, so we will focus our analysis on ARM. On it, the biggest overhead is at *BF_cfb64_encrypt()*. There is a couple of optimizations that the compiler performs when emitting code directly to ARM that are lost when translating from RISC-V to ARM:

- Combine several loads and reloads (from spills) into a single *ldmia* (load multiple registers) instruction.
- Preserve contents loaded before calling *BF_encrypt()*, used in *n2l()* calls, to reuse them in *l2n()* calls later.
- Implement 4 calls to *l2n()*, shown in Listing 5.2, with 4 pairs of *lsr* (logical shift right) plus *bfi* (bit field insert) and a single 32-bit store of the result. RISC-V has nothing similar to *bfi* to manipulate bits, so it ends up using more instructions and more memory accesses to arrange data correctly.

```
/* char *a; char b, c; */
#define l2n(a, b, c)  *(a++) = (b>>c) & 0xff
```

Listing 5.2: BlowFish's *l2n()* macro.

Together, these optimizations are responsible for most of the slowdown measured in the translated binary.

5.5 The BitCount Case

On ARM, register synchronization overhead of BitCount has a much higher impact than on x86. Even on ABI mode, register synchronization is responsible for 42% of the time spent running BitCount on ARM. On native mode, the compiler is smart enough to use very few registers in the main loop and, above that, load/reload only the registers that it will need inside the loop and are used to pass arguments to the indirect function called. When translated, all registers that had any modification and may transfer data to/from the called function are synchronized. ABI mode reduces this number drastically, but still *syncs* many more registers than native compilation, causing an overhead of more than 0.2x. This, however, is not due to hard to emulate aspects of RISC-V, as this overhead could be eliminated by improvements in the SBT engine, by making it move register *syncs* of unchanged registers inside the loop to the outside, an optimization similar to loop invariant code motion, but that in this case need to be performed by the translator.

5.6 The LAME Case

On ARM, the biggest LAME overhead is at *window_subband.constprop.28()* function. However, compared to the other benchmarks of MiBench, LAME is considerably larger, and other parts of it also present significant overhead. But, because we had not enough time to perform a full analysis of LAME, we have analyzed only *window_subband()*.

First, let's consider the experimental data collected from a typical run of LAME on ARM, translated using the ABI mode:

```
            time spent at LAME  time spent at window_subband  slowdown
ARM:        91.68%              13.71%
RV32-ARM:   93.70%              15.86%                        1.46x

Perf samples:   pre-loop    loop1   loop2   total
ARM:            10          241     288     539
RV32-ARM:       17          421     863     1301
```

RV32-ARM means RISC-V 32-bit translated to ARM using our SBT engine, while ARM is the native binary. Note that although the percentage of time spent at *window_subband()* on RV32-ARM is not much higher than that of ARM, this percentage refers to a bigger total execution time. Linux *perf* samples show how the time spent on *window_subband()* is distributed. Little time is spent before the first loop, both in native and in translated mode. At the first function loop, the translated code is almost 2x slower than native, but at the second (nested) loop, it is more than 3x slower than native. It is this part of the function that is going to be analyzed next, inspecting how code was generated for it.

The *hot spot* source code has the following format:

```
for 15..0:
    for 14..0:
        s0 += *wp++ * *in++;
        s1 += *wp++ * *in++;
```

Where, *wp* points to the *mm* variable, that corresponds to uninitialized static data (but that was already initialized by previous code at this point) and *in* is a function parameter that here points to the *win* variable, that also corresponds to uninitialized static data. *s0* and *s1* are local variables. The fact that the main data used in the analyzed code is static global data helps us to exclude other potential sources of inefficiency, such as local variables, emulated stack, and spills.

Inspecting the generated code, we can see that, in all cases, the inner loop was completely unrolled, consisting of (roughly) the instructions presented in Listing 5.3, for each iteration, on each architecture:

```
4x vldr
2x vmla.f64



total: 6 instructions
```

(a) ARM

```
4x fld offs(a4/a5)
2x fmadd



total: 6 instructions
```

(b) RV32

```
2x
1- movw r0, %lo(addr)
2- movt r0, %hi(addr)
3- add r0, ip
4- vldr dx, [r0]
5- add.w rx, r2, #imm
6- vldr dy, [rx]
7- vmla.f64

total: 14 instructions
```

(c) RV32-ARM

Listing 5.3: LAME hot spot loop body, for ARM, RV32 and RV32-ARM.

We can see that, for both ARM and RISC-V, the generated code is optimal. It performs only the minimum required operations: 4 loads and 2 fused multiply adds of the loaded values. When translating from RISC-V to ARM however, the number of instructions more than double. We can divide the inefficiencies in two parts:

1. On RV32-ARM, instructions 1 to 3 are used to load an immediate address, that is then used by instruction 4 to load a value from there. On RV32, these addresses are relative to the *a4* register, that at the inner loop has a known value, which explains why these are translated to immediate addresses on ARM. The main problem here seems to be that, when translating the optimized/unrolled RV32 loop, the optimizer/code generator fails to infer that all these addresses are relative to a common

base, with an offset added only. Add to that the fact that, when RISC-V is translated to ARM, an immediate address in Position Independent Code (PIC) requires 3 instructions to be loaded into a register: load the lower part (*movw*), higher part (*movt*) and add the position independent base (*ip*). On native ARM *codegen*, loads are performed using a base register plus an offset. Going even further, the compiler adds an offset to the base register, to make it possible to use the limited immediate offset field of *vldr* instruction (-1020..1020), saving a register and an add instruction on each load. Thus, for native ARM, the compiler is able to maintain the whole view of the hottest nested loop and generate the most efficient code for it. That is not the case when translating the already unrolled and optimized RV32 loop code. Performance could possibly be increased if the generated ARM code did not need to be position independent or if we forced RISC-V code to be position independent too, so both optimizers would match on this feature. Our ARM toolchain however requires PIC and substantial time and effort would be needed to change the experiments to use RISC-V PIC. Besides, LAME was the only benchmark where translating RISC-V non-PIC to ARM PIC had a significant performance impact.

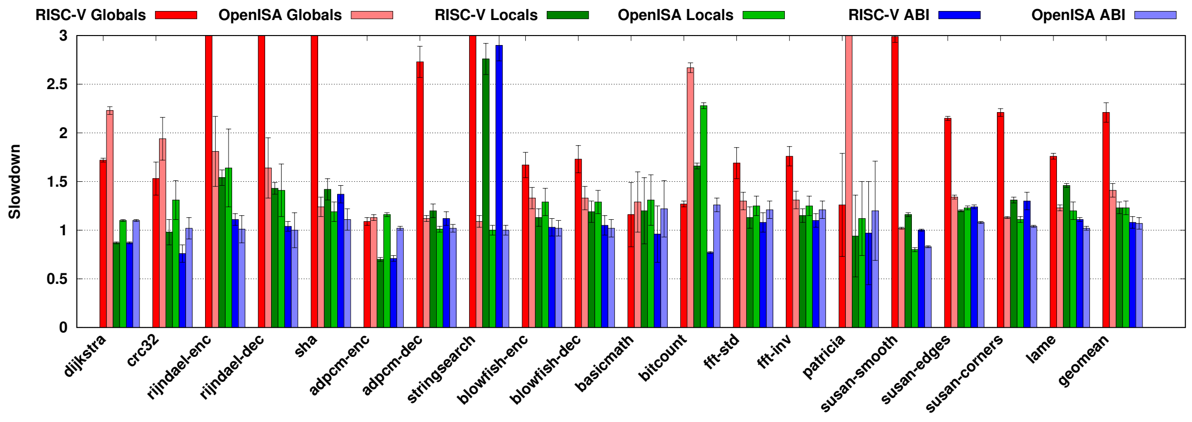
2. On RV32, the value of the *a5* register inside the main basic block varies (it is assigned from a *phi* node). That is why addresses derived from it cannot be converted into immediate values, as in case 1. When generating ARM code, *a5* based loads results in 2 instructions: adding an offset to *r2*, that corresponds to *a5* and then loading the value (instructions 5 and 6). In this case, the base register adjustment optimization performed in ARM— to make the offsets fit in *vldr* immediate field — was missed, as in case 1.

5.7 RISC-V vs OpenISA

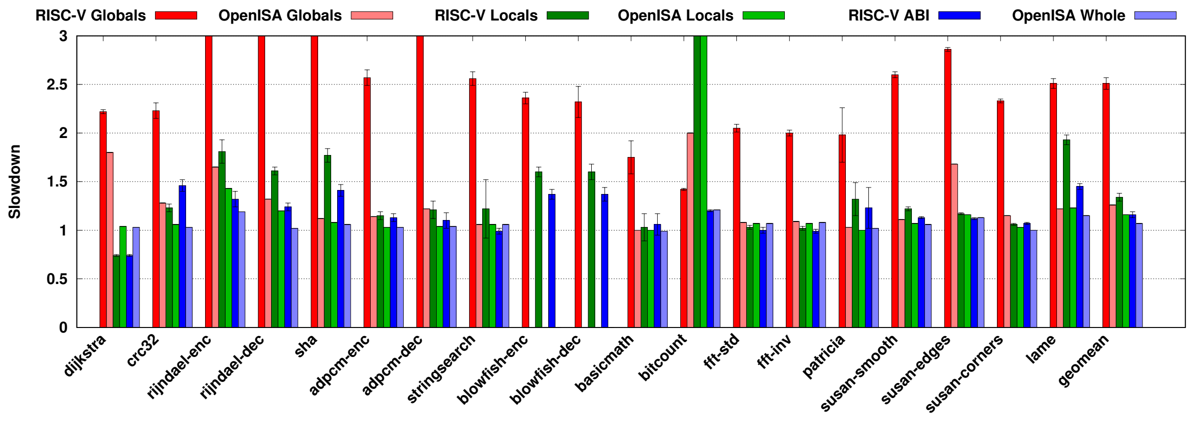
Now that we have discussed issues related to different compilers, versions and ABIs, and investigated the sources of the major overheads in our MiBench translation, we move on to compare the results of our RISC-V SBT engine with that of OpenISA [20]. This comparison is relevant because, as seen earlier, this work was based on that of OpenISA, using the same approach to investigate the quality of translated code, and thus the results of our RISC-V SBT engine were expected to be similar to those obtained by OpenISA's SBT engine. In fact, Figure 5.4, that compares the performance of our SBT engine with that obtained by OpenISA's, when translating RISC-V/OpenISA to x86 and ARM using the hard-float ABI, shows that, when considering the geometric mean, the results obtained by both are indeed similar. On the other hand, when each benchmark is considered separately, considerable performance differences can be noticed in some cases, such as in StringSearch, where the older LLVM version used by OpenISA SBT engine was apparently unable to vectorize native code, and in BitCount for x86, where RISC-V SBT engine performed better.

For x86, the result is practically the same as that of OpenISA, excluding Globals. On ARM, the results were a little worse than OpenISA's. The slowdowns of BitCount in Locals mode, that cannot be seen in the graph, were of 3.16x and 3.11x, for the RISC-

V SBT engine and OpenISA SBT engine, respectively. Note however that in Figure 5.4(b) we compared RISC-V SBT engine's ABI mode with OpenISA SBT engine's Whole mode — that translates the whole program at once, considering it a single, huge region — because, currently, OpenISA's infrastructure does not support ABI mode on ARM, while RISC-V SBT engine does not support the Whole mode. Also, it is important to mention that OpenISA SBT engine's results for ARM were extracted from Auler's PhD thesis [41] — and not reproduced as in the x86 case — because we had no easy access to the infrastructure needed to reproduce his experiments. That is also the reason why the performance results for Blowfish-Encode and Blowfish-Decode are missing for OpenISA's results for ARM, as they were not available in Auler's thesis.



(a) x86



(b) ARM

Figure 5.4: Slowdown for our RISC-V SBT engine and for the OpenISA SBT engine. All binaries were compiled using the hard-float ABI.

Because OpenISA's results [41] were obtained using LLVM 3.7 and ours used LLVM 7.0, we have compiled the benchmarks with both versions and compared the geometric means of the slowdowns obtained by each LLVM version. The differences were small, less than 2%, indicating that the results that we have obtained were not caused by improvements in LLVM.

5.8 Our SBT engine vs DBT engines available

Finally, we compared our best approach — translating binaries generated with hard-float ABI from GCC to x86 — with the most known DBT engines for RISC-V available: QEMU (v2.7.50, from github.com/riscv/riscv-qemu, commit ff36f2f77e), RV8 (github.com/rv8-io/rv8, commit 8342590) and Imperas OVP Simulator for RISC-V (github.com/riscv/riscv-ovpsim, commit 0b8b51a). The QEMU version we used was the same as the one in riscv-gnu-toolchain, that, although a bit older, was very stable and could run all benchmarks without errors, unlike other recent QEMU versions. For RV8 and OVP, however, we were not able to run any benchmark that makes use of floating-point instructions, as they either crash, hang or produce wrong results. Further investigation is needed to understand the causes of these failures, and if they are due to limitations in RV8 and OVP or due to some incompatibility between the toolchain and libraries used to produce the binaries and what is supported/expected by these emulators. Another difference between QEMU and RV8/OVP was that only the former was able to run the dynamic binaries produced by our default RISC-V GCC Linux toolchain. For the latter, we used a RISC-V GCC Newlib toolchain, that consists of the same compiler, but uses Newlib instead of GNU *libc* and produces statically linked binaries only.

Note that, with QEMU, RV8, and OVP it was not viable for us to use the same measurement approach as that of RISC-V SBT engine and OpenISA SBT engine, where we could easily factor out time spent at *libc*. Because of this, in the following chart, the slowdowns were all measured considering only the execution time, including those of our SBT engine. That is why its slowdowns are a little different than that of previous graphs.

We can clearly see in the chart from Figure 5.5 that our RISC-V SBT engine was the one with the best performance for all tested programs. Our RISC-V SBT engine achieved, on average, a 1.11x slowdown in Locals mode, while QEMU, RV8 and OVP achieved 6.13x, 2.85x and 4.92x, respectively.

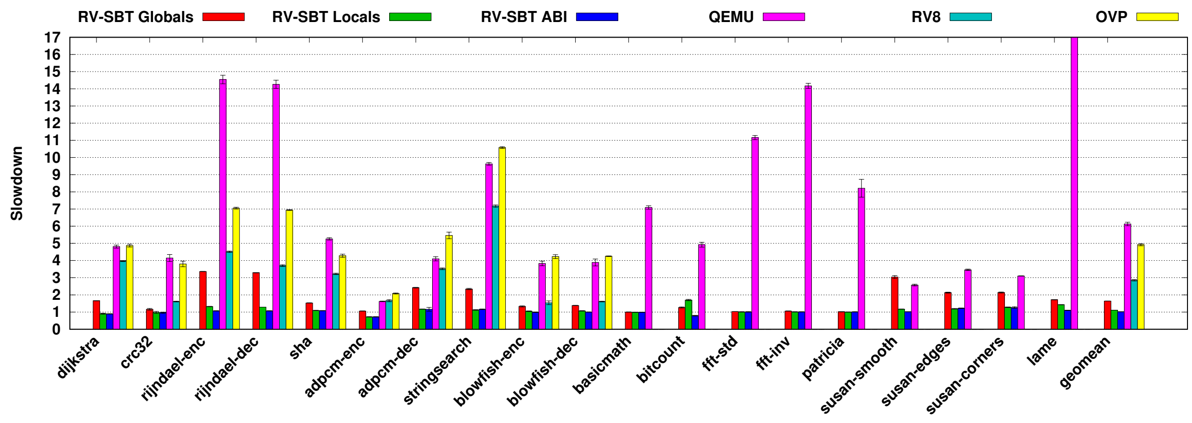


Figure 5.5: Slowdown comparison between our RISC-V SBT, QEMU-RISCV DBT, RV8 DBT and Imperas OVP Simulator for RISC-V, on a x86 processor.

In general, the performance of the Locals translation scheme produced better code than the Globals one, while ABI mode further improved Locals performance by reducing the register synchronization overhead, at the cost of being unable to translate programs that do not follow RISC-V ABI.

5.9 SBT vs DBT code quality

Comparing only the execution times of binaries translated with SBT against those translated by DBT is not very fair, as DBT engines have many other overhead sources that are absent in SBT engines, as we discussed previously. As this work's main concern is code quality, in this section we take a brief look at code generated by QEMU, RV8 and our SBT engine, when translating RISC-V code to x86, using the Dijkstra benchmark as example.

This comparison, however, raises an issue: the binary translators being compared each use different Region Formation Techniques (RFT). In QEMU, the regions are dynamic basic blocks, while in RV8 they are traces and in our RISC-V SBT each region corresponds to a function. For Dijkstra benchmark's main function, *dijkstra()*, QEMU generates 793 instructions in 29 regions, RV8 generates 2230 instructions in 16 regions and our SBT engine generates 325 instructions in 1 region. Figures 5.6 and 5.7 illustrate these differences.

As can be seen, QEMU and RV8 end up fragmenting the translated function in several regions. This fragmentation has two issues:

- It hinders the application of optimizations, as the optimizations do not see whole loops or code cycles, but only parts of it.
- DBT engines add prologue and epilogue in the regions, that are instructions not related to the translated RISC-V instructions but only to the DBT engine working. These instructions result in overhead whenever there is a transition between regions.

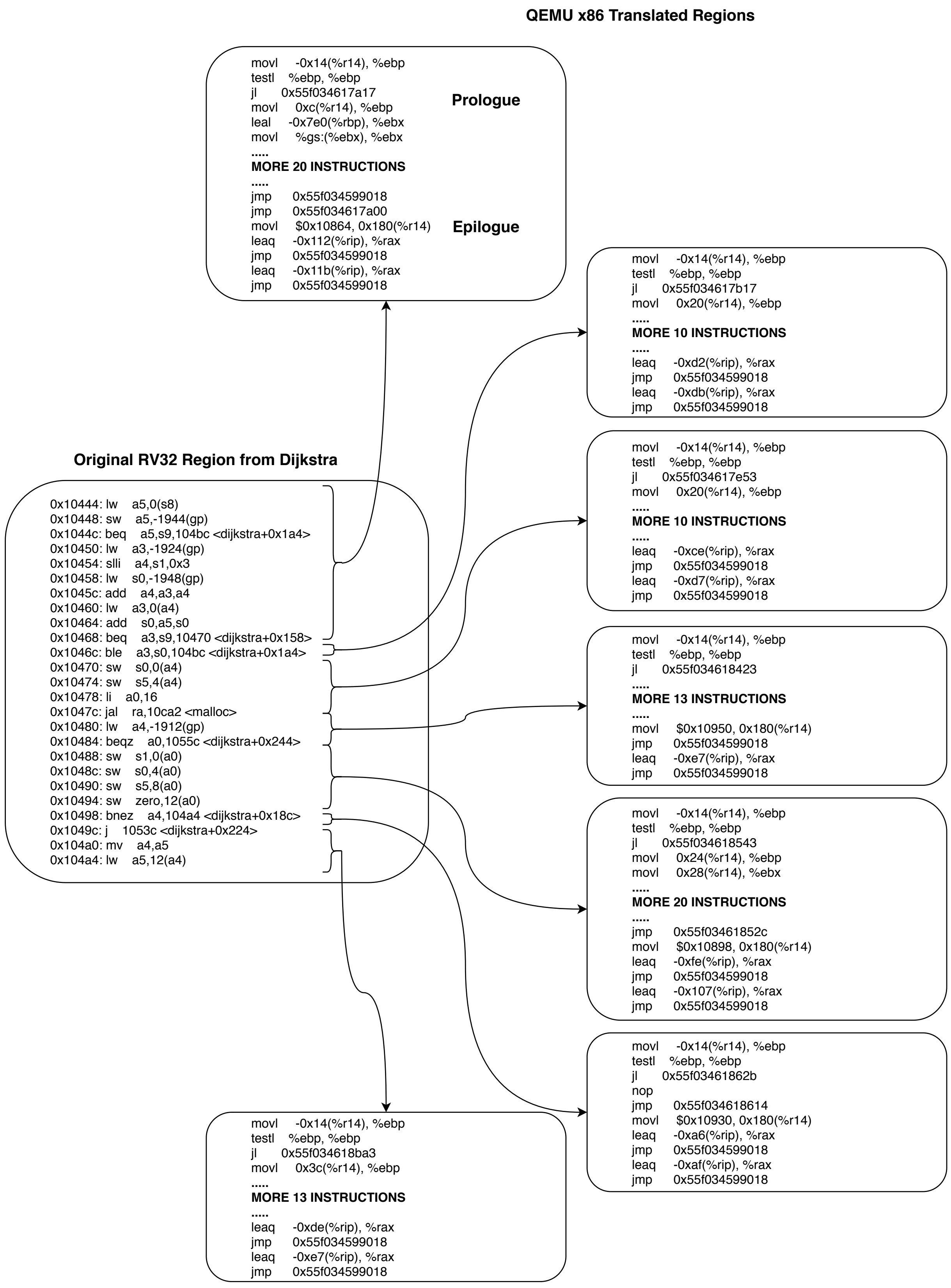
The SBT engine has the advantages of not fragmenting the code and not necessarily needing prologues and epilogues between distinct regions, although our SBT engine — in Locals or ABI mode — inserts extra instructions to perform register synchronization when switching between functions. In a sense, saving registers on function calls and restoring them on returns is analogous to DBT engine's prologues and epilogues, by executing some code needed by the translation engine internal working only, adding overhead when switching between regions, that corresponds to functions in our SBT.

Now that the general structure of translated code has been discussed, let us take a closer look at a specific part of the code and how each engine translates it. In order to avoid having huge code listings, only a small, relevant, part of the code was selected. It corresponds to the hottest loop in Dijkstra benchmark. Listing 5.3 shows the C code for this region.

```
while (qLast->qNext)
    qLast = qLast->qNext;
```

Listing 5.3: Dijkstra hot spot in C

It is important to note that the code fragment above is inside two loops. It walks through a linked list, from head to tail, to find the last element in it, before adding a



QEMU x86 Translated Regions
Prologue
Epilogue
Original RV32 Region from Dijkstra

```
movl    -0x14(%r14), %ebp
testl   %ebp, %ebp
jl      0x55f034617a17
movl    0xc(%r14), %ebp
leal    -0x7e0(%rbp), %ebx
movl    %gs:(%ebx), %ebx
.....
MORE 20 INSTRUCTIONS
.....
jmp     0x55f034599018
jmp     0x55f034617a00
movl    $0x10864, 0x180(%r14)
leaq    -0x112(%rip), %rax
jmp     0x55f034599018
leaq    -0x11b(%rip), %rax
jmp     0x55f034599018
```

```
0x10444: lw    a5,0(s8)
0x10448: sw    a5,-1944(gp)
0x1044c: beq   a5,s9,104bc <dijkstra+0x1a4>
0x10450: lw    a3,-1924(gp)
0x10454: slli  a4,s1,0x3
0x10458: lw    s0,-1948(gp)
0x1045c: add   a4,a3,a4
0x10460: lw    a3,0(a4)
0x10464: add   s0,a5,s0
0x10468: beq   a3,s9,10470 <dijkstra+0x158>
0x1046c: ble   a3,s0,104bc <dijkstra+0x1a4>
0x10470: sw    s0,0(a4)
0x10474: sw    s5,4(a4)
0x10478: li    a0,16
0x1047c: jal   ra,10ca2 <malloc>
0x10480: lw    a4,-1912(gp)
0x10484: beqz  a0,1055c <dijkstra+0x244>
0x10488: sw    s1,0(a0)
0x1048c: sw    s0,4(a0)
0x10490: sw    s5,8(a0)
0x10494: sw    zero,12(a0)
0x10498: bnez  a4,104a4 <dijkstra+0x18c>
0x1049c: j     1053c <dijkstra+0x224>
0x104a0: mv    a4,a5
0x104a4: lw    a5,12(a4)
```

```
movl    -0x14(%r14), %ebp
testl   %ebp, %ebp
jl      0x55f034617b17
movl    0x20(%r14), %ebp
.....
MORE 10 INSTRUCTIONS
.....
leaq    -0xd2(%rip), %rax
jmp     0x55f034599018
leaq    -0xdb(%rip), %rax
jmp     0x55f034599018
```

```
movl    -0x14(%r14), %ebp
testl   %ebp, %ebp
jl      0x55f034617e53
movl    0x20(%r14), %ebp
.....
MORE 10 INSTRUCTIONS
.....
leaq    -0xce(%rip), %rax
jmp     0x55f034599018
leaq    -0xd7(%rip), %rax
jmp     0x55f034599018
```

```
movl    -0x14(%r14), %ebp
testl   %ebp, %ebp
jl      0x55f034618423
.....
MORE 13 INSTRUCTIONS
.....
movl    $0x10950, 0x180(%r14)
jmp     0x55f034599018
leaq    -0xe7(%rip), %rax
jmp     0x55f034599018
```

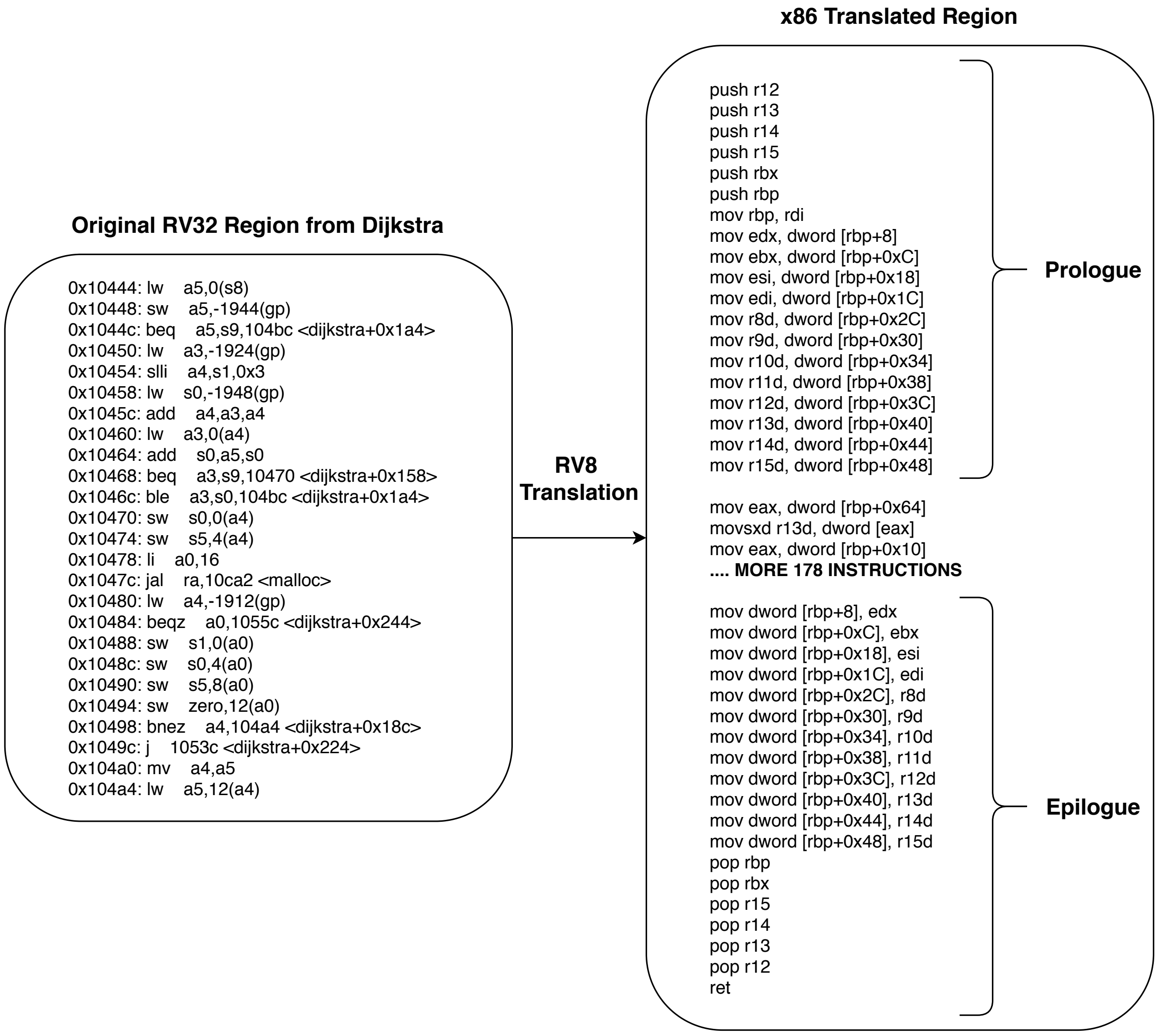
```
movl    -0x14(%r14), %ebp
testl   %ebp, %ebp
jl      0x55f034618543
movl    0x24(%r14), %ebp
movl    0x28(%r14), %ebx
.....
MORE 20 INSTRUCTIONS
.....
jmp     0x55f03461852c
movl    $0x10898, 0x180(%r14)
leaq    -0xfe(%rip), %rax
jmp     0x55f034599018
leaq    -0x107(%rip), %rax
jmp     0x55f034599018
```

```
movl    -0x14(%r14), %ebp
testl   %ebp, %ebp
jl      0x55f03461862b
nop
jmp     0x55f034618614
movl    $0x10930, 0x180(%r14)
leaq    -0xa6(%rip), %rax
jmp     0x55f034599018
leaq    -0xaf(%rip), %rax
jmp     0x55f034599018
```

```
movl    -0x14(%r14), %ebp
testl   %ebp, %ebp
jl      0x55f034618ba3
movl    0x3c(%r14), %ebp
.....
MORE 13 INSTRUCTIONS
.....
leaq    -0xde(%rip), %rax
jmp     0x55f034599018
leaq    -0xe7(%rip), %rax
jmp     0x55f034599018
```

Figure 5.6: Translated code regions produced by QEMU.

new one. Here in Dijkstra benchmark this list is actually used to implement a queue, and this code fragment is part of the enqueue function. Listing 5.4 shows the corresponding



```
0x10444: lw    a5,0(s8)
0x10448: sw    a5,-1944(gp)
0x1044c: beq   a5,s9,104bc <dijkstra+0x1a4>
0x10450: lw    a3,-1924(gp)
0x10454: slli  a4,s1,0x3
0x10458: lw    s0,-1948(gp)
0x1045c: add   a4,a3,a4
0x10460: lw    a3,0(a4)
0x10464: add   s0,a5,s0
0x10468: beq   a3,s9,10470 <dijkstra+0x158>
0x1046c: ble   a3,s0,104bc <dijkstra+0x1a4>
0x10470: sw    s0,0(a4)
0x10474: sw    s5,4(a4)
0x10478: li    a0,16
0x1047c: jal   ra,10ca2 <malloc>
0x10480: lw    a4,-1912(gp)
0x10484: beqz  a0,1055c <dijkstra+0x244>
0x10488: sw    s1,0(a0)
0x1048c: sw    s0,4(a0)
0x10490: sw    s5,8(a0)
0x10494: sw    zero,12(a0)
0x10498: bnez  a4,104a4 <dijkstra+0x18c>
0x1049c: j     1053c <dijkstra+0x224>
0x104a0: mv    a4,a5
0x104a4: lw    a5,12(a4)
```

```
push r12
push r13
push r14
push r15
push rbx
push rbp
mov rbp, rdi
mov edx, dword [rbp+8]
mov ebx, dword [rbp+0xC]
mov esi, dword [rbp+0x18]
mov edi, dword [rbp+0x1C]
mov r8d, dword [rbp+0x2C]
mov r9d, dword [rbp+0x30]
mov r10d, dword [rbp+0x34]
mov r11d, dword [rbp+0x38]
mov r12d, dword [rbp+0x3C]
mov r13d, dword [rbp+0x40]
mov r14d, dword [rbp+0x44]
mov r15d, dword [rbp+0x48]

mov eax, dword [rbp+0x64]
movsxd r13d, dword [eax]
mov eax, dword [rbp+0x10]
.... MORE 178 INSTRUCTIONS

mov dword [rbp+8], edx
mov dword [rbp+0xC], ebx
mov dword [rbp+0x18], esi
mov dword [rbp+0x1C], edi
mov dword [rbp+0x2C], r8d
mov dword [rbp+0x30], r9d
mov dword [rbp+0x34], r10d
mov dword [rbp+0x38], r11d
mov dword [rbp+0x3C], r12d
mov dword [rbp+0x40], r13d
mov dword [rbp+0x44], r14d
mov dword [rbp+0x48], r15d
pop rbp
pop rbx
pop r15
pop r14
pop r13
pop r12
ret
```

Figure 5.7: Translated code region produced by RV8.

RISC-V code, produced by GCC, and Listing 5.8 shows the translations of QEMU, RV8 and our RISC-V SBT, respectively.

```
107b8:     mv     a4, a5
107bc:     lw     a5, 12(a4)
107c0:     bnez   a5, 107b8
```

Listing 5.4: Dijkstra hot spot in RISC-V

Notice that, although the RISC-V addresses in translated code differ, they correspond to the same guest RISC-V instructions. There are two reasons for the address differences. The first is that RV8 takes as input a statically linked binary from RISC-V Newlib toolchain, while QEMU and RISC-V SBT take a dynamically linked binary from RISC-V Linux toolchain, as discussed in the previous section. The second is that it seems like QEMU is able to identify duplicated dynamic basic blocks and keep only a single copy. Thus, in the sample above it holds only the region from the non-inlined *enqueue()* function.

It can also be noted that QEMU's prologue and epilogue code are considerably shorter than that of RV8. The cause seems to be that QEMU maps RISC-V registers to memory locations only, while RV8 statically maps the most used RISC-V registers to x86 registers, but then needs to save/restore those when switching from/to the translator's code. This is analogous to our RISC-V SBT Globals and Locals modes, where in Globals no register synchronization is needed, but every register write means a memory write.

In the code generated by QEMU, each RISC-V register maps to a memory location. But, as x86 does not support moving data between two memory locations in a single instruction, the RISC-V move from *a5* to *a4* needs to be performed in two instructions, using a temporary register. Next, QEMU is smart enough to avoid reloading *a4*, that is already in *ebp*, but on the other hand it performs the address calculation and the load in two instructions, which on x86 could be performed in a single instruction, using a different addressing mode. The translation of the last RISC-V instruction is straightforward.

In RV8 code, first the prologue loads all x86 registers used for direct mapping of RISC-V registers. Note that all registers are loaded in the prologue — and later saved in the epilogue — even those not used in the region. Here an optimization similar to that performed on our SBT should be possible, to avoid restoring/saving unused RISC-V registers. But, except for the prologue and epilogue, the translated RISC-V code in this case presents very high-quality, the same as that of our RISC-V SBT. Note, however, that this is not always the case, as the other, not so oftenly used RISC-V registers, are not mapped to x86 registers — because x86 does not have as many registers as RISC-V— and then end up resulting in extra memory accesses. Besides, RV8 takes advantage of x86-64 extra registers, even when emulating RISC-V 32-bit code, that is not the case for our SBT, that uses only those registers available in IA-32.

In both QEMU and RV8, while the code shown in Listing 5.8 invokes the translator at the branch point, both seem to leave room for hot-patching, where QEMU can replace the jump-to-next-instruction for a jump to beginning of the loop and RV8 can replace the *mov* to *rbp* for a jump to *L1*.

```
# prologue
movl  −0x14(%r14), %ebp
testl %ebp, %ebp
jl    L4

# 0x10894: mv    a4, a5
#           a4: 0x38(%r14)
#           a5: 0x3c(%r14)
movl  0x3c(%r14), %ebp
movl  %ebp, 0x38(%r14)

# 0x10898: lw    a5, 12(a4)
addl  $0xc, %ebp
movl  %gs:(%ebp), %ebp
movl  %ebp, 0x3c(%r14)

# 0x1089c: bnez a5, −8
#              <0x10894>
testl %ebp, %ebp
jne   L2

# epilogue1
nop
jmp   L1
L1:
movl  $0x108a0, 0x180(%r14)
leaq  −0xc1(%rip), %rax
jmp   0x55f034599018

# epilogue2
L2:
jmp   L3
L3:
movl  $0x10894, 0x180(%r14)
leaq  −0xde(%rip), %rax
jmp   0x55f034599018

# epilogue3
L4:
leaq  −0xe7(%rip), %rax
jmp   0x55f034599018

──────────
7 instructions
4 memory accesses
```

(a) QEMU

```
# prologue
push r12
push r13
push r14
push r15
push rbx
push rbp
mov rbp, rdi
mov edx, dword [rbp+8]
mov ebx, dword [rbp+0xC]
mov esi, dword [rbp+0x18]
mov edi, dword [rbp+0x1C]
mov r8d, dword [rbp+0x2C]
mov r9d, dword [rbp+0x30]
mov r10d, dword [rbp+0x34]
mov r11d, dword [rbp+0x38]
mov r12d, dword [rbp+0x3C]
mov r13d, dword [rbp+0x40]
mov r14d, dword [rbp+0x44]
mov r15d, dword [rbp+0x48]

# 0x104a4: lw    a5, 12(a4)
#           r12d: a4
#           r13d: a5
L1:
movsxd r13d, dword [r12d+0xC]

# 0x104a8: bnez a5, 104a0
cmp r13d, 0
je L2

# 0x104a0: mv    a4, a5
L3:
mov r12d, r13d

# epilogue
mov qword [rbp], 104A4
L0:
mov dword [rbp+8], edx
mov dword [rbp+0xC], ebx
mov dword [rbp+0x18], esi
mov dword [rbp+0x1C], edi
mov dword [rbp+0x2C], r8d
mov dword [rbp+0x30], r9d
mov dword [rbp+0x34], r10d
mov dword [rbp+0x38], r11d
mov dword [rbp+0x3C], r12d
mov dword [rbp+0x40], r13d
mov dword [rbp+0x44], r14d
mov dword [rbp+0x48], r15d
pop rbp
pop rbx
pop r15
pop r14
pop r13
pop r12
ret

.align 16
L2:
mov qword [rbp], 104AC
jmp 7FFF00000000
──────────
4 instructions
1 memory access
```

(b) RV8

```
L1:
# 0x107b8: mv    a4, a5
#           a4: ecx
#           a5: ebp
mov    %ebp, %ecx

# 0x107bc: lw    a5, 12(a4)
mov    0xc(%ebp), %ebp

# 0x107c0: bnez a5, 107b8
test   %ebp, %ebp
jne    L1

──────────
4 instructions
1 memory access
```

(c) SBT

Listing 5.8: Dijkstra hot spot translation, produced by QEMU, RV8 and RISC-V SBT.

Chapter 6

Conclusions

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 near-native performance 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. A strong evidence that DBT engines with high-performance can be built for RISC-V. We did this by building a RISC-V static binary translator which is able to translate RISC-V to x86 and ARM with an execution overhead lower than 1.23x in the former and 1.34x in the latter, being the fastest RISC-V emulator presented so far in the literature.

During our experiments, we have seen that one of the major obstacles that prevented us from achieving near-native performance in some benchmarks was vectorized code in native binaries. Currently, RISC-V does not have vector instructions, so it uses more instructions to perform operations that can be performed with vector instructions in other ISAs. Thus, when RISC-V is translated to the LLVM IR, LLVM is not able to deduce that some instructions can be grouped and replaced by a vector instruction. We see two possible approaches to this issue. The first is to wait for (or create) RISC-V vector extension to become ready and make use of it when generating code. The second is to improve the SBT engine, possibly by means of more sophisticated data analysis techniques, in order to make it emit or reorganize IR instructions in a way that enables LLVM to generate vectorized code.

Besides the difficulties due to code vectorization, we have observed that, in some cases, translating RISC-V code optimized to make use of most of its 32 general purpose registers can result in considerable performance loss when the host ISA does not have as many registers, as in x86 and ARM cases. We saw this high number of registers used specially in RISC-V loops that were completely unrolled by the compiler. On x86 and ARM, the compiler limits the unrolling depth, apparently using the number of available host registers as a parameter, to avoid spills. In this case, a possible approach could be to reorder loads, stores and operation on data, in an effort to try to reduce the number of live registers.

One last source of translation difficulty that we highlight occurs when the compiler performs optimizations that make use of some complex, ISA specific, instructions. When translating from RISC-V to LLVM IR and then native ISA, these optimizations are usually lost. On ARM, this was seen in LAME and Blowfish. In the latter, ARM makes use of instructions to manipulate bits, saving some memory accesses, and loads multiple registers at once. On x86, this was seen in SHA, where the native compiler was able to take advantage of some complex x86 addressing modes. At least some of these cases could be handled by improvements in the SBT engine, such as combining simple instruction patterns and replacing them by more powerful LLVM IR instructions, that could map to more complex instructions present in the host ISA. But this could involve reordering instructions, which would complicate the implementation.

In this work, the performance of our RISC-V SBT was compared to that of OpenISA SBT, the best cross-ISA SBT known, and to the best RISC-V DBTs available, as summarized in Table 6.1. The low overheads achieved by our SBT engine suggest that it is possible to design and implement high-performance DBTs to emulate RISC-V code on x86 and ARM platforms.

Name	**Guest-ISA**	**IR**	**Target-ISA**	**Technique**	**Avg. Slowdown**
OpenISA-SBT	OpenISA	LLVM 3.7	x86/ARM	SBT	1.23x/1.16x
QEMU	RISC-V	QEMU IR	x86 (and others)	DBT	6.13x
OVP	RISC-V	Unknown	x86	DBT	4.92x
RV8	RISC-V	None	x86	DBT	2.85x
Our SBT	RISC-V	LLVM 7.0	x86/ARM	SBT	**1.23x/1.34x**

Table 6.1: Comparison between binary translator approaches.

In future works, we intend to experiment with the 64-bit variant of RISC-V, to check if it can also be translated to high-quality code. We also intend to investigate further the causes of missed vectorizations by LLVM and propose ways to handle it. Finally, in a future work we plan to build a RISC-V DBT engine that makes use of the translation techniques discussed in this work, or else modify an existing DBT engine, such as the OpenISA DBT, to also support RISC-V.

Bibliography

[1] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0. Technical report, EECS Department, University of California, Berkeley, 2014.

[2] RISC-V Foundation. RISC-V Foundation. http://riscv.org/risc-v-foundation. Acessed: 2018-10-21.

[3] Tuan Ta, Lin Cheng, and Christopher Batten. Simulating multi-core RISC-V systems in Gem5. In *Workshop on Computer Architecture Research with RISC-V*, 2018.

[4] Berkin Ilbeyi, Derek Lockhart, and Christopher Batten. Pydgin for RISC-V: A fast and productive instruction-set simulator. In *Extended Abstract for Presentation at the 3rd RISC-V Workshop*, 2016.

[5] Imperas Software. OVP simulator for RISC-V. https://github.com/riscv/riscv-ovpsim. Accessed: 2018-12-02.

[6] Michael Clark and Bruce Hoult. rv8: a high performance RISC-V to x86 binary translator. In *First Workshop on Computer Architecture Research with RISC-V (CARRV). Boston, MA, USA*, 2017.

[7] RISC-V Foundation. RISC-V QEMU. https://github.com/riscv/riscv-qemu. Accessed: 2018-10-21.

[8] Andrew Waterman and Krste Asanovic. The RISC-V instruction set manual, volume ii: Privileged architecture, version 1.10. https://riscv.org/specifications/privileged-isa, 2017. Accessed: 2018-12-15.

[9] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical report, EECS Department, University of California, Berkeley, 2016.

[10] Berkeley Architecture Research. BOOM: Berkeley out-of-order machine. https://github.com/riscv-boom/riscv-boom. Accessed: 2018-12-15.

[11] SiFive. Freedom. https://github.com/sifive/freedom. Accessed: 2018-12-15.

[12] SiFive. Hifive1. https://www.sifive.com/boards/hifive1. Accessed: 2018-12-15.

[13] SiFive. Hifive unleashed. https://www.sifive.com/boards/hifive-unleashed. Accessed: 2018-12-15.

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

[15] Apple. Rosetta: the most amazing software you'll never see. https://www.apple.com/rosetta/index.html. Accessed: 2018-10-21.

[16] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta code morphing^{TM} software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, 2003.

[17] Christian Häubl and Hanspeter Mössenböck. Trace-based compilation for the java hotspot virtual machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 2011.

[18] Edson Borin and Youfeng Wu. Characterization of DBT overhead. In *IEEE International Symposium on Workload Characterization, 2009, Austin, TX. Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, 2009.

[19] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[20] Rafael Auler and Edson Borin. The case for flexible ISAs: unleashing hardware and software. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2017 29th International Symposium on*, 2017.

[21] Leandro Lupori, Vanderson Martins do Rosario, and Edson Borin. Uma análise da facilidade de emulação de binários RISC-V. In *ERAD-SP 2018*, 2018.

[22] Leandro Lupori, Vanderson Martins do Rosario, and Edson Borin. Towards a high-performance RISC-V emulator. In *WSCAD 2018*, 2018.

[23] Kevin P Lawton. Bochs: A portable PC emulator for Unix/x. *Linux Journal*, 1996(29es):7, 1996.

[24] Divino Cesar, Rafael Auler, Rafael Dalibera, Sandro Rigo, Edson Borin, and Guido Araujo. Modeling virtual machines misprediction overhead. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[25] Alec Roelke and Mircea R Stan. RISC5: Implementing the RISC-V ISA in Gem5. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[26] Fabrice Bellard. TinyEMU. https://bellard.org/tinyemu. Accessed: 2018-10-27.

[27] RISC-V Foundation. riscv-angel. https://github.com/riscv/riscv-angel. Accessed: 2019-01-24.

[28] RISC-V Foundation. Spike RISC-V ISA simulator. https://github.com/riscv/riscv-isa-sim. Accessed: 2019-01-24.

[29] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[30] Igor Böhm, Tobias JK Edler von Koch, Stephen C Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. 46(6):74–85, 2011.

[31] Filipe Salgado, Tiago Gomes, Sandro Pinto, Jorge Cabral, and Adriano Tavares. Condition codes evaluation on dynamic binary translation for embedded platforms. *IEEE Embedded Systems Letters*, 9(3):89–92, 2017.

[32] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012.

[33] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. HERMES: a fast cross-ISA binary translator with post-optimization. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, 2015.

[34] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[35] Imperas Software. Imperas RISC-V solutions. https://www.imperas.com/imperas-riscv-solutions. Accessed: 2018-12-02.

[36] Cristina Cifuentes and Vishv M Malhotra. Binary translation: Static, dynamic, retargetable? In *icsm*, 1996.

[37] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: an LLVM-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, 2012.

[38] Gabriel Ferreira Teles Gomes and Edson Borin. *Indirect branch emulation techniques in virtual machines.* Dissertation, University of Campinas, 2014.

[39] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4*, 2001.

[40] Georgios Bitzes and Andrzej Nowak. The overhead of profiling using PMU hardware counters. Technical report, CERN, 2014.

[41] Rafael Auler and Edson Borin. *OpenISA, a hybrid ISA*. PhD thesis, University of Campinas, 2016.