





Static Global Register Allocation for Dynamic Binary Translators

Niko Zurstraßen , Nils Bosbach , Lennart M. Reimann , Rainer Leupers 

RWTH Aachen University, Institute for Communication Technologies and Embedded Systems

Abstract—Dynamic Binary Translators (DBTs) facilitate the execution of binaries across different Instruction Set Architectures (ISAs). Similar to a just-in-time compiler, they recompile machine code from one ISA to another, and subsequently execute the generated code. To achieve near-native execution speed, several challenges must be overcome. This includes the problem of *register allocation (RA)*. In classical compiler engineering, RA is often performed by global methods. However, due to the nature of DBTs, established global methods like graph coloring or linear scan are hardly applicable. This is why state-of-the-art DBTs, like QEMU, use basic-block-local methods, which come with several disadvantages. Addressing these flaws, we propose a novel global method based on static target-to-host mappings. As most applications only work on a small set of registers, mapping them statically from host to target significantly reduces load/store overhead. In a case study using our RISC-V-on-ARM64 user-mode simulator RISE-SIM, we demonstrate speedups of up to $1.4\times$ compared to basic-block-local methods.

Index Terms—Dynamic Binary Translation, Register Allocation, RISC-V, ARM, Virtual Platform, QEMU

I. INTRODUCTION

A Dynamic Binary Translator (DBT) is a software tool that translates binaries from a target Instruction Set Architecture (ISA) to a host ISA. Or in other words, DBTs enable the execution of binaries that were compiled for ISAs other than the system that executes them. It finds application in many different scenarios, reaching from industrial contexts to everyday use-cases. For example, as an integral part of so-called Virtual Platforms (VPs), which can be described as virtual twins of compute systems, DBTs enable early software development without requiring hardware prototypes. More on the user-oriented side, DBTs are also employed in user-mode simulators. Here, DBTs help to retain compatibility with legacy software, which cannot be ported to a new ISA. For example, Apple’s Rosetta 2 x86-on-ARM user-mode simulator played an important role in the company’s transition from the x86 ISA to ARM-based Apple silicon.

For all use-cases, the execution speed of a DBT is among its most important characteristics. Only a near-native execution speed allows for seamless interaction with the user. However, from translating the code itself to simulating floating-point instructions - there are many reasons why reaching native execution speeds may not be possible in all cases.

A ubiquitous challenge in DBT is Register Allocation (RA). Although RA is a well-explored field of research, established global methods from classical compiler engineering fail in DBT-based simulation for several reasons. Hence, state-of-the-

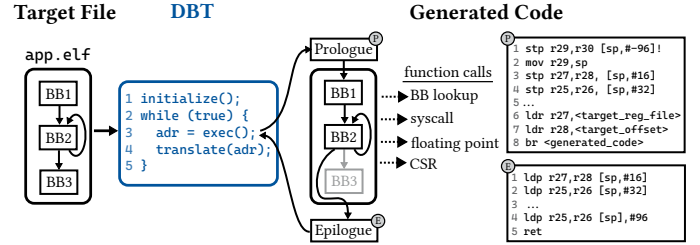


Fig. 1. High-level workflow of a Dynamic Binary Translator (DBT).

art DBT-based simulators, such as QEMU, employ simple local RA. This achieves an optimal solution per basic block, but requires loading and storing target registers at the beginning and end of each basic block.

To remedy this flaw of local RA, we propose a new global RA method based on static target-to-host mappings. The key idea is to exploit spatial locality of register usages in applications. Using an exhaustive set of benchmarks, we confirmed that most applications only work on a small set of registers. By keeping these frequently used target registers in static host registers, the number of additional loads and stores can be reduced by a significant amount. Using the RISE-SIM RISC-V-on-ARM user-mode simulator, we achieved performance improvements of up to $1.4\times$ using basic-block-local RA as a reference.

II. BACKGROUND

A. Dynamic Binary Translation

A Dynamic Binary Translator (DBT) is similar to a classical compiler in many regards. It translates a program from a source language to a target language, with the aim to eventually create an executable. While compilers usually translate from high-level languages (C, C++, etc.) to low-level languages (assembly, machine code, etc.), DBTs translate machine code from one ISA (e.g., RISC-V) to another ISA (e.g., ARM64). Opposed to a Static Binary Translator (SBT), a DBT executes and translates the binary in a coupled process. This allows to easily deal with self-modifying code and indirect branches with addresses only known at runtime. Fig. 1 gives a simplified overview of the typical workflow of a DBT like QEMU or RISE-SIM. As a first step, the target binary (e.g., an ELF file) is passed to the DBT. Subsequently, the initialization process starts, which includes translating the first basic block. To avoid repetitive recompilations, translated basic blocks are inserted into a cache. Then the main loop of

the DBT is entered, where code is executed and translated in an alternating fashion. The first part of the execution phase invokes a prologue, which saves call-preserved registers and sets up pointers, which are later explained in Section III. This prologue is similar to a prologue known from function calls. After the prologue, the generated code is executed. During the code execution, function calls to the host environment might occur, including cache lookups for basic blocks, system calls, handling of floating point instructions, and reading/writing architectural state through Control/Status Registers (CSRs). At some point, the program counter may reach parts of the Control Flow Graph (CFG) which were not yet translated. For example, BB3 in Fig. 1 is still not translated. This leads to a call of the epilogue, handing control back to the main loop. The main loop then repeats the translation/execution cycle until the process calls for an exit.

To account for self-modifying code, DBTs usually mark a host page write-protected if a translated basic block can be found within it. Thus, writes into executable code can be tracked by simply intercepting a SEGV signal. If a write is detected, already translated basic blocks are invalidated.

B. Register Allocation for DBT

As mentioned before, DBT shares many similarities with classical compilers. This also concerns the problem of *register allocation* (RA). During the RA phase, the code is already close to assembly, but instead of registers, the individual instructions use a possibly infinite number of automatic variables. Mapping these variables to a limited number of host registers is the key challenge of RA. In case the allocator cannot hold all used (also called “live”) variables in the host registers, so-called memory spills have to be inserted. This involves writing a variable to the main memory and fetching it once it is used again. To generate fast code, most allocators pursue the goal of minimizing the number of memory spills. However, generating an optimal solution quickly is far from trivial, as the underlying problem is NP-complete [14]. This is why numerous algorithms and heuristics have been developed in the past decades. Applying these methods to DBTs is possible, but they come with their own peculiarities, requiring a reevaluation of already beaten paths. Particularly RA for classical compilers and DBTs differ in the following regards.

Number of Variables: In classical compiler engineering, there can be an unlimited number of variables, which have to be mapped to the host registers. With DBTs, the number of variables (meaning the target registers) is limited. For standard RISC-V, there can be at most 32 general purpose registers/variables, which have to be mapped to the host register file.

Compilation Time: Code compilation and code execution are coupled phases in DBTs. That means, spending too much time on RA can compromise any resulting performance gains. In classical compiler engineering, such a situation only arises for just-in-time (JIT) compilers.

Control Flow Graph Uncertainty: For classical compilers, whole translation units are known during compilation time. DBTs faces a more opaque situation. As described in Subsection II-A, some jump addresses are only known during runtime, requiring alternating execute and translate phases to explore the CFG step-by-step. Moreover, self-modifying code may change the CFG during runtime.

Preprocessed Target Code: For DBT, the target code comprises target registers, which were determined by the target compiler’s register allocator. This creates inherent characteristics, which may be exploited by a DBT register allocator.

C. RISC-V/ARM64 Registers

This subsection provides greater detail about the register file structure and calling conventions on ARM64 and RISC-V. The information provided here is used in Section III for our RISC-V-on-ARM64 case-study.

The ARM64 ISA provides 32 general purpose registers. To facilitate interoperability between applications and the operating system, adhering to a defined interface, also called Application Binary Interface (ABI), is paramount. This includes defining special semantics for the register file. In case of ARM64, the individual meanings of each register are listed in Table II-B. The most important distinction concerns callee-saved and caller-saved registers. Callee-saved registers, also called call-preserved registers, are guaranteed to keep the same value before and after a function call. That means, if a callee-saved register is used in a function, a store in the function prologue and a load in the function epilogue have to be inserted. This is opposed to caller-saved registers, also called call-clobbered registers, where values might change due to the

TABLE I
ARM64 GENERAL PURPOSE REGISTERS ACCORDING TO AAPCS64 [9].

| Name | ABI Mnem. | Meaning | Call-preserved? |
|-----------|-----------|---------------------|-----------------|
| r0 - r7 | - | Argument | No |
| r8 | - | Indirect result | No |
| r9 - r15 | - | Temporary | No |
| r16 | IP0 | Temporary, IP | No |
| r17 | IP1 | Temporary, IP | No |
| r18 | | Temporary, platform | No |
| r19 - r28 | | Callee-saved | Yes |
| r29 | FP | Frame pointer | Yes |
| r30 | LR | Return address | Yes |
| SP | | Stack pointer | Yes |

TABLE II
RISC-V GENERAL PURPOSE REGISTERS AND SEMANTICS [24].

| Name | ABI Mnem. | Meaning | Call-preserved? |
|-----------|-----------|----------------|-------------------|
| x0 | zero | Zero | – (Immutable) |
| x1 | ra | Return address | No |
| x2 | sp | Stack pointer | Yes |
| x3 | gp | Global pointer | – (Unallocatable) |
| x4 | tp | Thread pointer | – (Unallocatable) |
| x5 - x7 | t0 - t2 | Temporary | No |
| x8 - x9 | s0 - s1 | Callee-saved | Yes |
| x10 - x17 | a0 - a7 | Argument | No |
| x18 - x27 | s2 - s11 | Callee-saved | Yes |
| x28 - x31 | t3 - t6 | Temporary | No |

function. If the caller wants to keep the value, a store and a load have to be inserted before and after the function call respectively.

Following the RISC approach, RISC-V’s general purpose register file is similar to ARM64. It provides 32 general purpose registers, which can be reduced to a number of 16 for its embedded version. The meanings of each register are listed in Table II-C.

III. METHODS

In this section, we introduce various RA methods for DBTs. This includes the methods used by QEMU [12] and SIM-V [21], as well as our own strategy.

A. Instruction-Local Register Allocation

In this subsection, we present an instruction-local RA. The method is simple and creates code with a maximum number of spills. Yet, it provides a fair starting point, and we will later use this method to assess how much RA impacts DBT performance.

As implicated by its name, the instruction-local RA performs RA per individual instruction. That means, all source operands are loaded before the translated instruction is executed and all destination operands are stored afterwards. An example for RISC-V-on-ARM64 is provided in Code 1. The example depicts a translation of two arithmetic RISC-V instructions (add and mul). Since both ARM64 and RISC-V addition/multiplication are similar, a RISC-V add/mul can be mapped to an ARM64 add/mul without further ado. However, before the instructions are executed, the corresponding operands from the target register file have to be loaded. Usually, the target register file is represented by a simple data structure in the DBT’s code (see RegisterFile in Code 1). If a pointer to the target register file is kept in a host register, addressing all target registers can be achieved by loads with corresponding offsets. In the given example, r27 contains this pointer. This pointer is initialized during the prologue as shown in Fig. 1. After loading the operands, the actual instruction is performed, and the result is written back to the target register file. This load and store procedure is

```

1 // C++ implementation of the register file.
2 struct RegisterFile {
3     std::array<u64, 32> x;
4     std::array<u64, 32> f;
5     u64 pc;
6 } rv_regfile;
7
8 // Target code: RISC-V
9 add x1, x2, x3 // x1 = x2 + x3
10 mul x5, x1, x4 // x5 = x1 * x4
11
12 // Host code: ARM64
13 ldr r20, [r27, #16] // Load x2 into r20
14 ldr r21, [r27, #24] // Load x3 into r21
15 add r22, r20, r21 // Perform add
16 str r22, [r27, #8] // Store x1 into struct
17 ldr r20, [r27, #8] // Load x1 into r20
18 ldr r21, [r27, #32] // Load x4 into r21
19 mul r22, r20, r21 // Perform mul
20 str r22, [r27, #40] // Store x5 into struct

```

Code 1. Example of instruction-local RA.

performed per instruction even if subsequent instructions use the same operands. In the example code, x1 is refetched for mul even though the corresponding values would be available in r22.

Since this creates multiple loads and stores per translated instruction, the overall performance may be hampered. Nevertheless, due to its simplicity, early versions of QEMU (0.9.0 and below) used a similar approach.

B. Basic-Block-Local Register Allocation

A more sophisticated RA in the scope of basic blocks is used by QEMU and SIM-V. SIM-V’s method involves linearly scanning the basic block and assigning registers in a greedy fashion. An example using the same target code as in Subsection III-A is shown in Code 2. Similarly to the previous method, operands that are used for the first time need to be loaded from the target register file. However, while scanning the code, the register allocator keeps a record of which target registers are currently loaded. Hence, the second instruction mul x5, x1, x4, does not need to reload register x1. Only x4 is loaded and assigned the next free register of the allocator’s available registers. At the end of the basic block, all dirty target registers are written back.

A problem with SIM-V’s greedy assignment is a lack of information. The register allocator does not know which registers are reused in future instructions and which are not. For instance, after performing the addition, the values of r20/x2 and r21/x3 are not used in future instructions and thus the registers can be reassigned. Hence, r20 and r21 are unnecessarily blocked.

This flaw is remedied by QEMU’s RA. Again, a linear scan is performed, and registers are allocated in a greedy fashion. However, a preceding live-variable analysis provides information about which target registers are “live” (i.e. they are used in future instructions). Hence, target registers do not unnecessarily block host registers, and the risk of memory spills is reduced. This comes at the cost of requiring more time for code translation due to the additional overhead of the live-variable analysis. Also, with this method, instruction selection and RA need to be two separate phases.

```

1 // Target code: RISC-V
2 add x1, x2, x3 // x1 = x2 + x3
3 mul x5, x1, x4 // x5 = x1 * x4
4
5 // Host code: ARM64
6 ldr r20, [r27, #16] // Load x2 into r20
7 ldr r21, [r27, #24] // Load x3 into r21
8 add r22, r20, r21 // Perform add
9 ldr r23, [r27, #32] // Load x4 into r23
10 mul r24, r20, r23 // Perform mul
11 str r22, [r27, #8] // Store x1 into struct
12 str r24, [r27, #40] // Store x5 into struct

```

Code 2. Example of basic-block-local RA.

C. Global Register Allocation for DBT

In this section we present our innovative method: a global RA for DBT. The problem of QEMU’s and SIM-V’s method is their limited scope, requiring loads and stores per basic

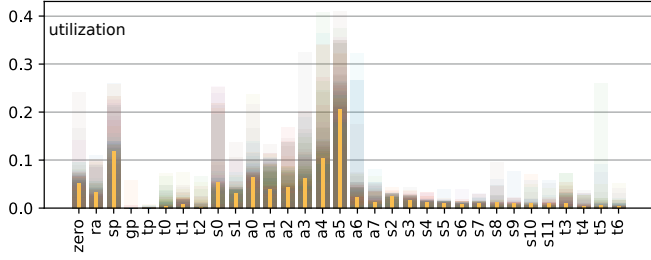


Fig. 2. Utilization of RISC-V general purpose registers in 114 benchmarks. Shaded bars represent individual benchmarks. The average is given by the orange bar.

block. More specifically, the number of used target registers per basic block represents a lower limit for the number of additional loads and store. For example, the target code in Listing 2 uses 5 different target registers, requiring a total of 5 loads and stores. In the worst case, memory spills increase this number further. Since basic blocks tend to be short, we assume that the additional loads and stores of basic-block-local RA significantly impact DBT performance.

Tackling this inherent flaw, *global RA* regards the problem on an inter-basic-block level. An established and popular method is graph coloring allocation, which was initially proposed by Chaitin et al. [14] in 1981. This method often produces quality code, but the quadratic runtime complexity renders it impractical for JIT compilers and DBTs. A global RA with a linear runtime is *linear scan* which was proposed by Poletto et al. [23] in 1999. Motivated by the linear runtime, linear scan, or variants of it, often find application in JIT compilers, such as the v8 Javascript engine [16].

While all established global RA methods may be applicable to DBT per se, the high CFG uncertainty poses a major challenge. That means, whenever the DBT engine discovers a new basic block, the CFG changes and global RA has to be rerun. For that reason, we introduce a static global RA, which does not require reruns even if the CFG changes.

The key idea is to keep frequently used target registers in static host registers, while less often used registers are allocated in a local way. This idea was motivated by a statement of A. Waterman, who said that “instructions express great spatial locality of register reference” [30]. His statement was based on a dynamic and static analysis of the SPEC CPU 2006 benchmarks. To validate his assertion and even extend on that, we ran our own dynamic analysis using more than 114 different benchmarks compiled for RISC-V (see Fig. 3). These benchmarks cover scenarios from embedded software to high performance computing, and include multiple different programming languages. The result of the register utilization analysis is depicted in Fig. 2. As can be seen, there is in fact a great average spatial locality, which is also relatively consistent throughout all benchmarks. For example, the 8 argument registers (a0-a7) already cover 60.1% of all register accesses on average.

The drawback of our approach is an obligatory initialization of the static registers in the prologue. Even if the registers are not used in the executed target code, the corresponding

| | | | |
|--|--|---|--|
| SPEC CPU 2017 [25] (1) 500.perlbench (2) 503.bwaves (3) 505.mcf (4) 507.cactuBSSN (5) 508.namd (6) 510.parest (7) 511.povray (8) 519.lbm (9) 520.omnettp (10) 523.xalanbm (11) 525.x264 (12) 527.cam4 (13) 531.deepsjeng (14) 538.imagick (15) 541.leela (16) 544.nab (17) 548.exchange2 (18) 549.fotonik3d (19) 554.roms (20) 557.xz | glmark2 [3] (30) buffer (31) build (32) bump (33) clear (34) conditionals (35) desktop (36) effect2d (37) function (38) ideas (39) jellyfish (40) loop (41) pulsar (42) refract (43) shading (45) shadow (46) terrain (47) texture | NPB [6] NPB.bt.A NPB.cg.A NPB.ep.A NPB.ft.A NPB.mg.A NPB.sp.A Other alexnet-train aobench [17] buildroot-boot cray [26] cryptsetup dhrystone-clang dhrystone-gcc fbench [27] fbench [28] glxgears himenio [4] lenet-infer linpack32 [5] linpack64 [5] SciMark 2.0 [10] stream [11] whetstone [15] | OpenNN [8] iris_plant breast_cancer simple_approx simple_class logical_operations airfoil mnist outlier_detection NumPy [19] linalg scalar mbench [18] adcpm basicmath bitcount blowfish crc32 dijkstra fft gsm jpeg lane patricia qsort rsynth sha stringsearch susan-edges typeset |
| FinanceBench [2] (21) Black Scholes (22) Bonds (23) Monte Carlo | Octane 2.0 [7] (48) all (49) box2d (50) codelead (51) crypto (52) deltablue (53) earleyboyer (54) mandreel (55) navierstoke (56) raytrace (57) regexp (58) splay | CoreMark-PRO [1] loops-all-mid-10k linear_alg-mid-100x100 net_test radix2-big-64k | |

Fig. 3. The 114 benchmarks performed 169,111,077,904,832 register accesses.

values have to be fetched. Furthermore, we load statically mapped registers in callee-saved registers to avoid frequent reloading due to function calls from the generated code (see “function calls” in Fig. 1). Since function calls from the code can be frequent, DBTs, like QEMU and RISE-SIM, prefer to keep target registers in callee-saved host registers. With our approach, parts of the 10 callee-saved ARM64 registers are already blocked. Ultimately, finding a good compromise between statically and dynamically mapped registers is key. In our experiments, the best results were obtained by mapping 8 RISC-V registers statically: a5, a4, a3, a0, ra, sp, x0, and a2. According to our data, this covers 69.5% of all register accesses.

IV. RELATED WORK

While RA in classical compiler engineering is an extensively explored field of research, publications for DBT are sparse. One of the first works in that domain was published by Hu et al. [20] in 2009. They describe how the instruction-local RA of QEMU v0.9.0 can be improved by a basic-block-local method. The reported speedups are up to $1.75\times$ for a Loongson-on-x86 translation. In the same year, Cai et al. compared several allocation methods for the their MIPS-on-x86 Crossbit DBT [13]. This also covers a similar global allocation method based on dynamic usage frequencies. However, their algorithm only maps one variable statically due to IA-32’s limited register file leading to no significant improvement. The best performance is attained with a simplified graph-coloring RA. Yet, performance differences between individual methods mostly differ only by a single-digit percentage. Similar observations were also made by Wang et al. [29] in 2018 using a x86-on-Sunway DBT.

V. RESULTS AND DISCUSSION

A. Benchmark Results

To assess the performance of our static RA approach, we incorporated the method in our RISC-V-on-ARM64 user-mode

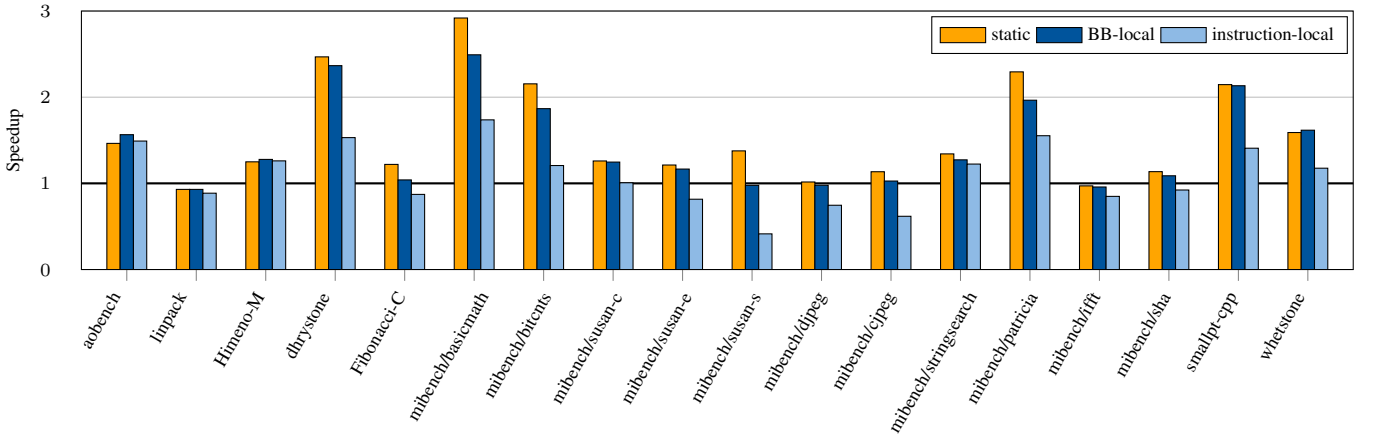


Fig. 4. Simulation performance of different configurations of RISE-SIM. The performance of user-mode QEMU v7.2.0 was used as a reference.

simulator *RISE-SIM*. Subsequently, an exhaustive benchmark study was conducted, whereby each benchmark was run with 4 different setups:

- 1) RISE-SIM with instruction-local RA: This method should yield a worst-case performance (see Subsection III-A).
- 2) RISE-SIM with basic-block-local RA: Here, our simulator was configured to use the BB-local RA method without live variable analysis (see Subsection III-B).
- 3) RISE-SIM with static RA: Our approach as explained in Subsection III-C. A setup with 8 statically mapped registers was used. The number is motivated by the results in Fig. 5.
- 4) QEMU v7.2.0: The popular open-source user-mode simulator uses the BB-local RA with live variable analysis.

The results of the benchmark study are depicted in Fig. 4. As can be seen in the Fig., static RA performs best in the majority of the cases, followed by the BB-local and instruction-local approaches. The performance difference between static RA and instruction-local RA can exceed 3x (see “mibench/susan-s”). This highlights why efficient register allocation is key for the performance of user-mode DBTs. The performance improvement of our static RA compared to basic-block-local RA is 1.1 \times on average with some benchmarks, such as

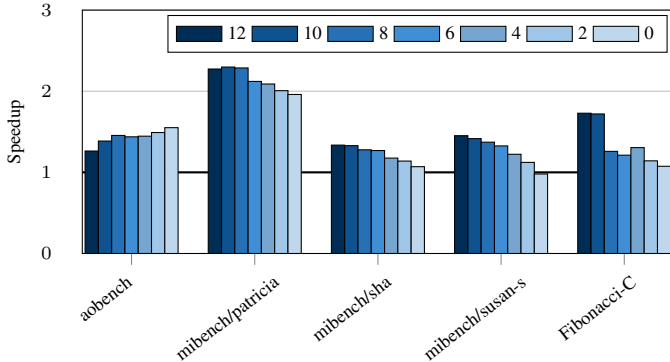


Fig. 5. Speedup of the static RA approach using different numbers of statically mapped registers. Each setup maps the $N \in \{0, 2, 4, 6, 8, 10, 12\}$ most frequently used target register. QEMU v7.2.0 was used as a reference.

“mibench/susan-s”, attaining speedups of up to 1.4 \times . Yet, there are some cases where BB-local RA is slightly ahead. In general, our simulator also outperforms QEMU in nearly all cases. However, from floating point simulation to basic block caching; RISE-SIM and QEMU differ in many regards. Thus, also other factors may play into the observed performance difference.

B. Result Analysis

In this subsection, we analyze the DBT-generated code to understand the characteristics of the proposed approaches.

Fig. 6 shows the main loop of a Fibonacci benchmark and the corresponding code by generated QEMU and RISE-SIM using the static allocator. It is the same benchmark, which was also used for the performance analysis in Fig. 4. Despite being a microbenchmark, the application showcases the improvements of static RA in an understandable manner. Starting from C code, the main loop of the Fibonacci benchmark is compiled to the RISC-V code in Fig. 6 b) using gcc 12.2.2 with O3 optimizations. When executing the application, QEMU and RISE-SIM generate the host code, as depicted in Fig. 6 c).

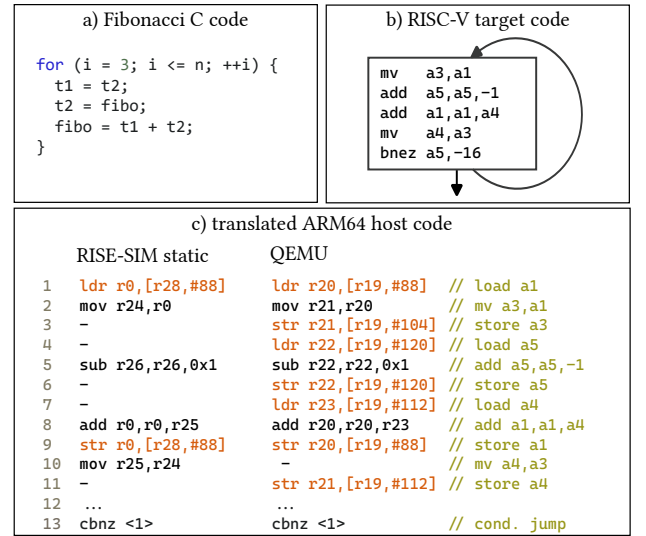


Fig. 6. Fibonacci example from C code to DBT-generated ARM64 host code.

Both QEMU and RISE-SIM start by loading target register `a1` from the memory for the subsequent `mv a3, a1`. The translated target instruction is then executed in Line 2. At this point, the value of register `a3` does not change anymore in the given basic block, leading to a write-back by QEMU in Line 3. Since RISE-SIM keeps the target register `a5`, `a4`, `a3`, and `a0` in static host registers, no write-back needs to be performed. Such a situation also arises at other points (Line 4, 6, 7). Opposed to RISE-SIM, QEMU also performs optimization on the generated code, which can be seen in Line 10 and 11. Here, QEMU skips `mv a4, a3` and directly writes `a3` into the memory location of `a4`. We believe that the time needed for instruction optimization outweighs the benefits of improved code, which is why RISE-SIM does not include such features. Ultimately, the end of the basic block is reached, and depending on which Fibonacci number should be generated, the control flow either branches back to the first instruction or continues. Since the branch address can be determined during translation time, direct block chaining avoids leaving the execution of the target code. The direct block chaining mechanism is simplified by a simple conditional jump `cbnz`.

Overall, the Fibonacci example highlights why RA is of major importance for DBTs. While the target’s RISC-V machine code only comprises 5 instructions, QEMU inflates this to 11 instructions due to a high number of additional loads and stores. The static RA of RISE-SIM significantly reduces this number and achieves the same functionality with only 7 instructions.

VI. CONCLUSION & OUTLOOK

In this work, we showed how a static global Register Allocation (RA) can significantly improve the performance of RISC-on-RISC DBT-based simulators. Using our RISC-V-on-ARM64 simulator RISE-SIM, we were able to achieve an average $1.1\times$ speedup over commonly used basic-block local methods. The attained speedup is motivated by the static mapping of our RA approach. Even though RISC-V provides an ample set of registers, most applications only accesses a limited set of them. If these target registers are statically mapped to host registers, loading and storing of target registers can be moved from basic block entry/exit to the simulation entry/exit. However, with our global approach, target registers are loaded/stored regardless of their later usage. Also, due to the static mapping, register pressure may increase. This highlights why only mapping a certain subset is key, and why the host architecture needs to provide a sufficient number of registers. In our case, the best results were obtained by statically mapping 8 RISC-V registers (`a5`, `a4`, `a3`, `a0`, `ra`, `sp`), which cover 69.5% of all accesses. Opposed to established RA methods from classical compiler engineering, our approach does not require reevaluations if the Control Flow Graph (CFG) changes. These two characteristics are important for Dynamic Binary Translators (DBTs), where code generation and execution are coupled phases, and where CFG visibility is limited.

REFERENCES

- [1] “CoreMark®-PRO,” <https://github.com/eembc/coremark-pro>, accessed: 2024-10-25.
- [2] “FinanceBench,” <https://github.com/cavazos-lab/FinanceBench>, accessed: 2024-10-25.
- [3] “glmark2,” <https://github.com/glmark2/glmark2>, accessed: 2024-10-25.
- [4] “Himeno Benchmark,” <https://github.com/kowsalyaChidambaram/Himeno-Benchmark>, accessed: 2024-10-25.
- [5] “linpack,” <https://www.netlib.org/linpack/>, accessed: 2024-10-25.
- [6] “NAS Parallel Benchmarks,” <https://www.nas.nasa.gov/software/npb.html>, accessed: 2024-10-25.
- [7] “Octane 2.0,” <https://github.com/chromium/octane>, accessed: 2024-10-25.
- [8] “OpenNN Examples,” <https://github.com/Artelnics/opennn/tree/master/examples>, accessed: 2024-10-25.
- [9] “Procedure Call Standard for the Arm® 64-bit Architecture (AArch64),” <https://github.com/ARM-software/abi-aa>, accessed: 2024-10-25.
- [10] “SciMark 2.0,” <https://math.nist.gov/scimark2/>, accessed: 2024-10-25.
- [11] “STREAM benchmark,” <https://www.cs.virginia.edu/stream/>, accessed: 2024-10-25.
- [12] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05. USA: USENIX Association, 2005, p. 41.
- [13] Z. Cai *et al.*, “Performance comparison of register allocation algorithms in dynamic binary translation,” in *2009 International Conference on Knowledge and Systems Engineering*, 2009, pp. 113–119.
- [14] G. J. Chaitin *et al.*, “Register allocation via coloring,” *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0096055181900485>
- [15] H. Curnow *et al.*, “whetstone,” <https://netlib.org/benchmark/whetstone.c>, accessed: 2024-10-25.
- [16] J. Eisl *et al.*, “Trace-based register allocation in a jit compiler,” in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2016, pp. 1–11.
- [17] S. Fujita, “aobench,” <https://github.com/syoyo/aobench>, accessed: 2024-10-25.
- [18] M. Guthaus *et al.*, “Mibench: A free, commercially representative embedded benchmark suite,” in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.
- [19] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: <https://doi.org/10.1038/s41586-020-2649-2>
- [20] Y. Hu *et al.*, “An Optimization Approach for QEMU,” in *2009 First International Conference on Information Science and Engineering*, 2009, pp. 129–132.
- [21] L. Jünger *et al.*, “SIM-V: Fast, Parallel RISC-V Simulation for Rapid Software Verification,” *DVCON Europe 2022*, 2022.
- [22] M. Moulin, “smallpt,” <https://github.com/matt77hias/smallpt>, accessed: 2024-10-25.
- [23] M. Poletto *et al.*, “Linear scan register allocation,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 895–913, 1999.
- [24] *RISC-V ABIs Specification*, RISC-V Foundation, 2023.
- [25] “SPEC CPU 2017,” <https://spec.org/cpu2017/>, Standard Performance Evaluation Corporation, accessed: 2024-10-25.
- [26] J. Tsiombikas, “c-ray,” <https://github.com/jtsiomb/c-ray>, accessed: 2024-10-25.
- [27] J. Walker, “fbench,” <https://www.fourmilab.ch/fbench/fbench.html>, accessed: 2024-10-25.
- [28] —, “ffbench,” <https://www.fourmilab.ch/fbench/ffbench.html>, accessed: 2024-10-25.
- [29] J. Wang *et al.*, “A binary translation backend registers allocation algorithm based on priority,” in *Geo-Spatial Knowledge and Intelligence: 5th International Conference, GSKI 2017, Chiang Mai, Thailand, December 8-10, 2017, Revised Selected Papers, Part II 5*. Springer, 2018, pp. 414–425.
- [30] A. Waterman, “Design of the RISC-V Instruction Set Architecture,” Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2016. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html>