Efficient LLVM-Based Dynamic Binary Translation

Alexis Engelke
Technical University of Munich
Garching near Munich, Germany
engelke@in.tum.de

Dominik Okwieka
Technical University of Munich
Garching near Munich, Germany
okwieka@in.tum.de

Martin Schulz
Technical University of Munich
Garching near Munich, Germany
schulzm@in.tum.de

Abstract

Emulation of other or newer processor architectures is necessary for a wide variety of use cases, from ensuring compatibility to offering a vehicle for computer architecture research. This problem is usually approached using dynamic binary translation, where machine code is translated, on the fly, to the host architecture during program execution. Existing systems, like QEMU, usually focus on translation performance rather than the overall program execution, and extensions, like HQEMU, are limited by their underlying implementation. Conversely, performance-focused systems are typically used for binary instrumentation. E.g., DynamoRIO reuses original instructions where possible, while Instrew utilizes the LLVM compiler infrastructure, but only supports same-architecture code generation.

In this short paper, we generalize Instrew to support different guest and host architectures by refactoring the lifter and by implementing target-independent optimizations to re-use host hardware features for emulated code. We demonstrate this flexibility by adding support for RISC-V as guest architecture and AArch64 as host architecture. Our performance results on SPEC CPU2017 show significant improvements compared to QEMU, HQEMU as well as the original Instrew.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; **Virtual machines**.

Keywords: Dynamic Binary Translation, LLVM, Optimization, Architecture Simulation, RISC-V

1 Introduction

Dynamic Binary Translation (DBT) is a common technique to allow the execution of code compiled for one architecture on systems with a different architecture. Such a translation layer can be used to run existing applications on other processor architectures, for example, newer architectures with new instruction sets or older ones with missing specialized instructions, even if source code is unavailable or porting would require too much effort.

Other use cases are low-level software development, e.g. for embedded systems, and research in computer architecture, where studying the impact of new instructions or other user-facing modifications requires software emulation, as few or no hardware is available.

QEMU [4] is a widely used CPU emulator supporting binary translation for many guest and host architectures. The original machine code is transformed to the architecture-independent intermediate code representation TCG, from which new machine code for the host architecture is generated. The code is lifted with basic block granularity and only few optimizations are performed, resulting in fast translation times, but high overhead for the translated code. The latter is especially problematic for longer-running applications.

Many approaches to improve the quality of TCG's generated code utilize the LLVM [16] compiler infrastructure [6, 13, 14]. However, they usually suffer either from performance problems of the LLVM code generator or from inherent TCG limitations such as the fine granularity of code translation preventing optimizations, or both.

HQEMU [12] circumvents the performance problem of LLVM by using a hybrid approach in which the entire code is first translated using TCG and only performance-sensitive sections are then optimized using LLVM. Further, the overhead caused by basic block granularity is addressed by analyzing and combining traces of basic blocks. Nonetheless, HQEMU still suffers from the known architectural limitations of TCG: it is not possible to further increase translation granularity to allow more complex control flows. Finally, HQEMU requires a modified version of LLVM to integrate with the QEMU architecture, decreasing maintainability considering the quickly evolving state of LLVM.

To avoid these limitations imposed by QEMU/TCG and to enable a more performance-focused solution, we take a different approach and rely on directly lifting machine code to LLVM-IR. This enables translating entire functions at once rather than at basic block granularity, matching the

native granularity of LLVM. Consequently, the translated functions retain more semantic information and can therefore take more advantage of LLVM's flexible, function-wide register allocator by exploiting the respective calling convention. This allows mapping frequently used guest registers directly to host registers, avoiding memory accesses caused by spilling.

We implement our approach by adapting the Instrew/Rellume [10] framework, which was originally designed for LLVM-based run-time binary instrumentation of x86-64 binaries. In particular, we generalize the lifter to be easily extensible for other guest architectures and modify the execution environment for more flexible support of different host architectures and calling conventions. To demonstrate the flexibility, we added a lifter for the 64-bit RISC-V architecture and support for AArch64 as host architecture. As an additional benefit, building on Instrew allows us to *instrument* the guest program during translation. This is an important use case for architecture development, allowing us to get performance metrics of the underlying machine code while still benefitting from high emulation speed resulting from JIT-compilation.

The main contributions of this paper are the following:

- A generalized performance-oriented library for lifting machine code to LLVM-IR that can be easily extended for other architectures.
- An efficient and flexible dynamic binary translation framework based solely on LLVM, currently supporting x86-64 and RISC-V rv64 as guest architectures.
- An approach to exploit the hardware return address stack for the emulated program by re-using host instructions for function calls and returns.

Our results show that this LLVM-based approach yields significant performance improvements compared to the state-of-the-art dynamic binary translators QEMU and HQEMU, having an emulation overhead of just 53% compared to QEMU with 648% and HQEMU with 124%.

This paper is structured as follows: in Section 2 we review the general structure of a dynamic binary translation system and illustrate the Instrew/Rellume architecture in more detail. In Section 3 we describe our changes to Instrew/Rellume and in Section 4 we show our results in comparison with state-of-the-art systems. In Section 5 we cover related work and in Section 6 we summarize our findings.

2 Binary Translation 101

In general, a dynamic binary translation system for userspace programs like QEMU [4] works as follows: during initialization, the guest code (the code to be emulated) is mapped into the address space of the emulator and the state of the emulated CPU is initialized. For translation of machine code from one architecture to another, small chunks of guest code are decoded and usually lifted into a low-level

intermediate code representation. In that representation, optimizations can be applied and the code is compiled to the host architecture (the architecture on which the program is intended to be executed). Compiled code fragments are stored in a *code cache* for future re-use. In the main execution loop, the dispatcher checks whether it already has translated a code block starting at the current program counter, potentially triggers the translation process, and then executes the translated code fragment, resulting in a modified CPU state with a new program counter value. This process repeats until the program exits. To reduce the overhead of dispatching the translated code fragments, two code blocks can be *chained* by inserting a direct jump at the end of the first code block to the following code block.

A common choice for the translation granularity are *basic blocks*, where a code chunk is ended by an instruction that can modify the control flow. This not only simplifies translation and code generation, but also ensures only actually executed code is translated.

A slightly different approach can be found in Instrew [10], a framework for retargetable dynamic binary instrumentation of x86-64 programs. Instead of a low-level internal code representation, it directly and exclusively uses the widely used LLVM [16] framework. It relies on the lifter Rellume to transform binary code to LLVM and implements several optimizations allowing for near-function granularity, only stopping at function calls, returns and indirect jumps.

3 Approach

Instrew's approach of directly leveraging LLVM forms a strong basis for an efficient binary translation system and has the ability to overcome many of the shortcomings in current state-of-the-art approaches. However, it also requires two major contributions: the generalization of the Rellume lifter, which initially only supported the x86-64 architecture, and the integration of new architecture-independent performance optimizations.

3.1 Multi-Architecture Support in Rellume

Generalizing Rellume requires challenging all assumptions specific to x86-64 by contrasting it with other, modern 64-bit architectures, such as AArch64 [3] and RISC-V [19]. This has a direct impact on the design of the machine code lifter, in our case the library Rellume.

In particular the register file, which is currently tailored to x86-64, must be modified to allow architecture-specific layouts. Architectures not only vary in the number of general-purpose or floating-point registers, but also in their size. For example, x86-64 provides 16 general-purpose registers and 16 SSE vector registers, each 128 bits wide, whereas 64-bit RISC-V has twice as many registers, but the floating-point registers only have a width of 64 bits. Whether condition

flags exist at all and, if so, which ones exist, also depends on the architecture.

Architectures also differ in their interpretation of the program counter. The instruction pointer rip of x86-64 points to the *end* of the instruction, whereas the program counter in RISC-V or AArch64 points to the address of the *beginning* of the current instruction, and on 32-bit ARM to the *second next* instruction. This requires careful handling during decoding and whenever the emulated program counter is inspected.

Some architectures, like AArch64 or RISC-V, support floating-point rounding modes to be encoded statically in the instruction itself or to be selected dynamically from a control register. LLVM, on the other side, has currently only very basic support for rounding modes other than the default *round-to-nearest* [2]. While for most operations minor deviations can be tolerated, this is not the case for conversions from floating-point values to integers—which are employed for widely used functions like floor or ceil. Further, architectures like x86-64 offer no way of encoding the rounding mode in an instruction, while changing the rounding mode in the control register is an expensive operation. Thus, Rellume implements some rounding operations explicitly using software emulation.

There are other architecture-specific peculiarities that must represented for proper translation; for example, the meaning of the carry flag after subtraction differs between x86-64 and AArch64, and RISC-V has no carry flag at all. Therefore, we moved the handling of such peculiarities to the architecture-specific part.

Despite these differences affecting Rellume's data structures and instruction handling, the generalized Rellume retains its function-granularity decoding and lifting strategy, enabling powerful optimizations. Apart from a configuration function to set the architecture, the Rellume API needed no further changes to enable multi-architecture use. Adapting Instrew to a new architecture therefore requires only very few modifications to extract the architecture given in the ELF header and to adjust the lifter's configuration accordingly.

This flexible structure and generic API also makes it possible to easily add further architectures by following these steps: (a) the number and sizes of registers have to be specified; (b) the mapping of instruction semantics to LLVM-IR semantics has to be specified; (c) control flow information for the generic decoder must be attached to decoded instructions, in particular the instruction length, kind (branch, call, return, etc.), and potential jump targets; and finally (d) the list of supported architectures has to be extended.

Case Study: Adding RISC-V Support. RISC-V [19] is a comparably new open architecture with a focus on simplicity and extensibility. Only few hardware is currently available, and several possible extensions, including vector extensions, still need to be finalized. This makes RISC-V an interesting target for our approach: dynamic binary translation aids

in evaluating the quality of compilers or design choices of ISA extensions, and the existing instrumentation capability allows injecting code to implement efficient profiling, e.g., to capture the effectiveness of vectorization.

We apply the previously listed steps to add RISC-V support for the 64-bit RISC-V base instruction set (RV64I) with commonly used extensions (*rv64imafdc*) to Rellume. Due to the generalizations described above, this is a straightforward process, except for one problematic case concerning atomic memory operations: while atomic read/modify/write operations are supported, *load-reserve/store-conditional* pairs are currently lifted as non-atomic stores, as these operations are very hard to represent in LLVM-IR code. Handling these is left to future work.

3.2 Performance Optimizations Covering Translations

Transformation to register-based calling convention. In the default lifting mode, Rellume lifts code into a function with a single parameter, the pointer to the *CPU state* structure. When the function is called, the pointer is passed in a register and the entire state of emulated registers is stored in memory. This works for every architecture. To improve performance, Engelke et al. [10] implemented a separate mode where the HHVM calling convention [18] is used, which supports passing 12 values in registers as arguments and return values on x86-64 hosts. This is used to map a fixed set of guest registers to host registers. Lifting to this calling convention, however, was implemented directly in the lifter, limiting this approach to x86-64 guests.

To make the general idea of using different calling conventions available for other guest–host combinations, we implement a mechanism in Instrew that transforms the basic CPU struct pointer variant to a more efficient one using the HHVM calling convention. Loads/stores with mapped registers in the CPU state are replaced by arguments and return values where possible. This way, other guest architectures can benefit from register-based argument passing—in particular, the adaption to RISC-V guests only requires the specification of the register mapping—and the code can be easily extended to other host architectures. Furthermore, it removes the need to implement more complex calling conventions in Rellume, simplifying the lifter.

Call-return optimization. While Rellume already supports some kind of whole-function lifting, it still stops lifting when it encounters function calls. However, there is a significant drawback of that approach: it does not exploit the fact that *most* functions return and continue after the corresponding function call instruction[1].

We therefore modify Rellume to support a callret mode, in which function calls in the original code are lifted to an LLVM-IR function *call* to the dispatcher. After that LLVM-IR

---

[1]This is not the case for exception or setjmp/longjmp.

call, the new program counter has to be verified to have the expected value so that the code after the call can be safely executed, otherwise the dispatcher needs to be called again. Function returns are lifted to LLVM-IR return instructions; indirect jumps are realized as tail calls to the dispatcher.

With these modifications, our binary translator essentially implements a shadow stack by translating entire functions including their call-relations at once. The difference to other approaches utilizing host call/return instructions [11, 15] lies in the increased translation granularity by continuing decoding after call instructions. However, the benefit of reduced overhead and usage of the return address stack of the CPU comes with a cost: translated code fragments may become significantly larger, directly impacting the translation time, which grows super-linearly with the code size.

4 Results

We evaluate our dynamic binary translation system using the SPEC CPU2017 benchmarks, which consists of a diverse set of compute- and memory-intensive applications and are, therefore, also used for the evaluation of similar systems [5, 12, 17]. As guest architectures, we use x86-64 and 64-bit RISC-V (*rv64imafdc*) and as host architectures we use x86-64 and AArch64. We compare the performance against a native compilation for the host architecture as well as against QEMU 5.2.0 [4]. For x86-64 guest code, we additionally compare against HQEMU 2.5.2 [12] in its hybrid mode[2].

4.1 Setup

For all benchmarks, we use the reference input size and run them single-threaded. Native code is compiled with GCC 9.2.0 and −03, cross-compiled code using GCC 10.2.0 and −03; everything is linked against glibc 2.32. For Instrew, we use LLVM 9.0, which is its latest supported version; for HQEMU we use LLVM 6.0, which is its latest supported version, and patch it according to their user documentation [1].

Our x86-64 target platform is based on Intel Xeon CPUs (E5-2697 v3, Haswell), 17 MiB L3 cache and 64 GiB main memory, running SUSE Linux 15 SP1 with Linux kernel 4.12.14-197.40 in 64-bit mode. Our AArch64 target platform is based on Cavium ThunderX2 99xx CPUs, 32 MiB L3 cache and 512 GiB main memory, running CentOS Linux 8 with Linux kernel 4.18.0-193.6.3 in AArch64 mode.

We exclude the benchmark 600.perlbench, as it is currently unsupported by Instrew [10]. QEMU did not finish all floating-point benchmarks in a reasonable time frame as a consequence of software floating-point emulation, which affects 638.imagick on x86-64 and the floating-point benchmarks targeted for AArch64.
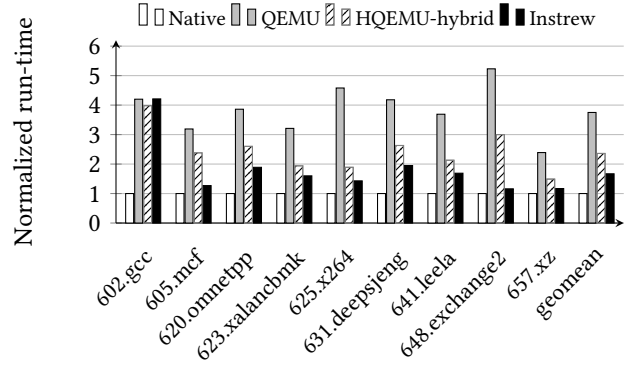
(a) Integer benchmarks.
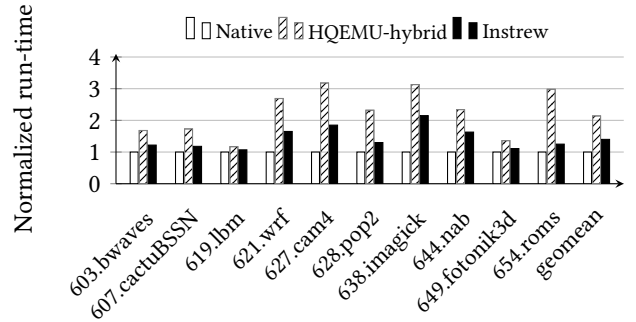
(b) Floating-point benchmarks. QEMU's slowdown (not shown here) ranges from 7.7x to 38x (geomean 16.4x).

Figure 1. Performance results on the SPEC CPU2017 benchmarks translating x86-64 to x86-64 with the *ref* workload.

4.2 Results

Translation x86-64→x86-64 (Fig. 1). The mean overhead across all benchmarks with our modified Instrew is 53% (67% INT, 40% FP) and therefore much lower than HQEMU with 124% (136% INT, 114% FP) and QEMU with 716% (275% INT, 1541% FP). For benchmark 619.lbm the overhead is as low as 7%, and the slowdown compared to the native execution is generally below a factor of 2. There is one exception: for the 602.gcc benchmark almost no performance improvement could be observed. As this benchmark executes a lot of code, around 55% of its time is spent on translating.

To better understand the impact of our described optimizations for modifying the calling convention (*hhvm*) and increasing the translation granularity (*callret*), we analyze their impact both individually and combined compared to the baseline with both optimizations turned off. The results are shown in Table 1. The use of a specialized calling convention generally improves performance, reducing the mean overhead by 0.27 at the cost of a slight increase in compilation time. The *callret* optimization on its own also leads to better performance, but the compilation time grows as well. An exception is 602.gcc, where run-time improvements are shadowed by the compilation time increase, resulting in

Table 1. Comparison of different optimizations compared for x86-64 guest code on an x86-64 host. Run-times are normalized to the native execution.

| Benchmark	base	hhvm	callret	both
602.gcc	3.41	3.15	3.66	4.21 (+0.80)
605.mcf	1.84	1.36	1.63	1.27 (−0.57)
620.omnetpp	2.33	1.96	2.25	1.89 (−0.44)
623.xalancbmk	1.75	1.72	1.62	1.60 (−0.15)
625.x264	1.57	1.46	1.49	1.43 (−0.14)
631.deepsjeng	2.57	1.96	2.38	1.95 (−0.62)
641.leela	2.38	1.71	2.12	1.69 (−0.69)
648.exchange2	1.33	1.33	1.16	1.16 (−0.17)
657.xz	1.28	1.17	1.32	1.17 (−0.11)
geomean	1.96	1.69	1.85	1.53 (−0.43)
avg. trans. time	782s	875s	1245s	1881s

an overall slowdown. In combination, the two optimizations have a much larger impact on overall performance than individually, reducing overhead by 0.43. However, the translation time increases on average by 140%. The main driver for this increase is the 602.gcc benchmark, which contains and calls a lot of functions. This implies that the limiting factor is the LLVM code generator, which appears to have performance problems when many registers are fixed at function call boundaries.

Translation RISC-V→x86-64 (Fig. 2). When translating 64-bit RISC-V code to x86-64, Instrew is generally more performant on both integer (76% Instrew vs. 202% QEMU) and floating-point benchmarks (40% Instrew vs. 1267% QEMU). The only exception is again the 602.gcc benchmark due to high translation times.

Another interesting observation is that the overhead of QEMU is lower when translating from RISC-V than from x86-64, whereas this is not the case for Instrew. This indicates that lifting to LLVM-IR abstracts most architecture-specific peculiarities. The remaining differences are likely to be caused by the current lack of vector instructions in RISC-V.

Translations to AArch64 (Figs. 3, 4). For translations from x86-64 and RISC-V to AArch64, our approach leads to overall significant performance improvements compared to state-of-the-art tools. However, for some benchmarks, only slight or no improvements can be observed.

Especially in comparison with the x86-64 host, the overall overhead of the programs emulated by Instrew is larger. This has several reasons: first, our calling convention optimization is not supported by upstream LLVM, which causes the entire register state to be written back to memory; second, for x86-64 guest programs, the additional registers provided by the architecture are rarely used; and, third, the CPU of our

(a) Integer benchmarks.

(b) Floating-point benchmarks.
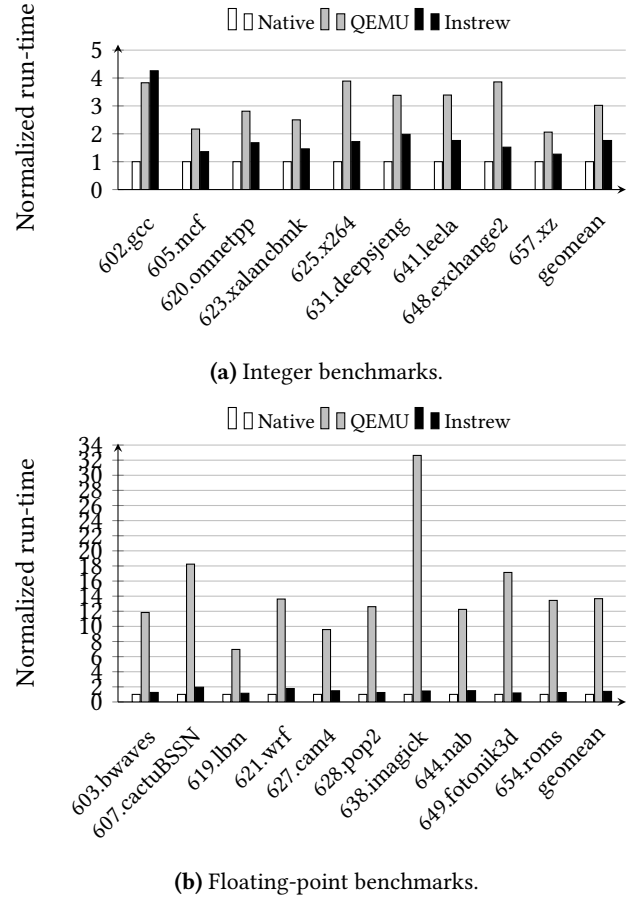
Figure 2. Performance results on the SPEC CPU2017 benchmarks translating RISC-V64 to x86-64 with the *ref* workload, compared against native x86-64 execution.

AArch64 host is performing fewer internal optimizations, resulting in less instruction-level parallelism.

4.3 Discussion

Our results show that increasing the translation granularity and avoiding TCG by lifting code directly to LLVM-IR leads to significant performance improvements for all measured configurations. The emulation overhead on x86-64 with 53% is much lower than HQEMU with 124%.

The approach of using a calling convention that allows flexible use of registers across translated code chunks for x86-64 hosts results in significant performance improvements. This motivates adding support for a more general *all-registers* calling convention in LLVM. When increasing translation granularity by lifting whole functions even beyond calls, we observe significant performance improvements for the actually translated code, as the return address prediction of the CPU can be exploited. However, the translation time also increases with bigger code chunks, negating performance gains on a single benchmark.
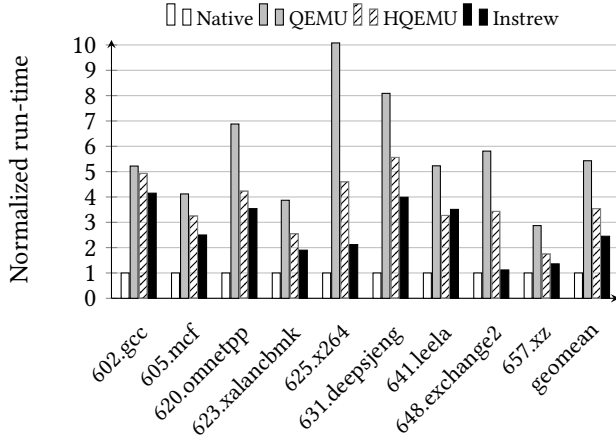
Figure 3. Performance results on the SPEC CPU2017 benchmarks translating x86-64 to AArch64 with the *ref* workload, compared against native AArch64 execution.
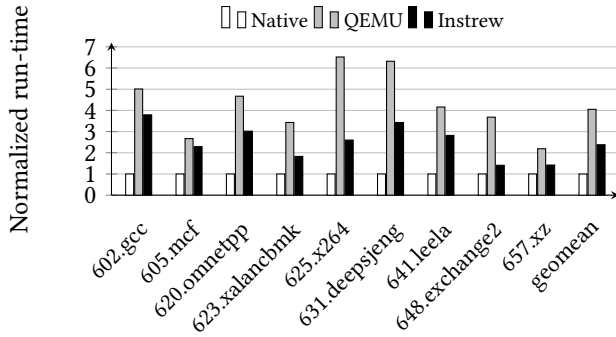

Figure 4. Performance results on the SPEC CPU2017 benchmarks translating RISC-V64 to AArch64 with the *ref* workload, compared against native AArch64 execution.

5 Related Work

QEMU [4] is the state-of-the-art tool for software emulation and virtualization. It supports several guest and host architectures and allows not only the emulation of Linux user space programs, but also full-system emulation.

Several approaches to address the performance problem of QEMU have been presented so far. As shown by Chipounov et al. [6], simply using LLVM to generate code for basic blocks leads to high translation overhead. Jeffery [14] explores the parallel use of TCG and LLVM, where the LLVM-IR is generated directly from ARM machine code and the full optimization/code generation routine runs in a separate thread, overwriting translated code fragments with more optimized versions. This approach did not lead to performance improvements, as the optimized fragments were found to be executed very rarely. LnQ [13] achieves performance improvements over QEMU by lifting guest code to LLVM-IR and applying the chaining optimization as well as caches for predicting

indirect jump targets and function returns. HQEMU [12] achieves further improvements by keeping the TCG code generator as fast path and merging TCG code fragments into traces to increase the size of LLVM-translated code blocks. In our work, we go further and increase the translation granularity to full functions, implicitly implementing a shadow stack for function returns using the native instruction for function calls and returns of the host architecture.

Pico [8] is a modification of QEMU enhanced with different ways to correctly handle atomic memory operations by leveraging hardware transactional memory.

Rv8 [7] is a dynamic binary translator which translates RISC-V machine code directly to x86-64 instructions and uses a fixed register mapping to reduce memory accesses. While they observe similar performance improvements compared to QEMU, their approach is not easily portable to other guest or host architectures.

McSema [20] translates entire binaries to LLVM-IR code, attempting to reconstruct functions, global variables and other common language constructs. However, as it is focused on software analysis and reverse engineering, performance is not a primary focus. RevGen [9] is a static binary translator based on McSema that translates binaries to LLVM-IR code using TCG as intermediate lifting step, but is not optimized for run-time performance.

Valgrind [17] enables same-architecture dynamic binary instrumentation, using VEX as architecture-independent code representation. It should be possible to retarget VEX code to other architectures, although its code generator usually incurs high overheads. DynamoRIO [5] is a performance-oriented system for same-architecture binary instrumentation, which internally uses the original machine code instructions where possible. Consequentially, retargeting program execution is not easily possible.

6 Summary & Outlook

In this paper, we described how we generalized the Instrew dynamic binary instrumentation framework to work efficiently with guest and host architectures other than x86-64. In particular, we described our approach to generalizing the lifting library Rellume for other architectures and demonstrated the flexibility by adding support for RISC-V. Further, we described two optimizations to use registers and the return address stack of the host processor more effectively. Our results show that this yields significant improvements over state-of-the-art dynamic binary translators.

Acknowledgments

References

[1] 2018. HQEMU v2.5.2 Technical Report, installation guide. http://csl.iis.sinica.edu.tw/hqemu/download/quickstart-2.5.2.pdf, accessed 2020-02-17.

[2] 2019. LLVM 9 Documentation: LLVM Language Reference Manual. http://releases.llvm.org/9.0.0/docs/LangRef.html, accessed 2020-02-17.

[3] Arm Limited. 2020. *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile.*

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO)*. 265–275.

[6] Vitaly Chipounov and George Candea. 2010. Dynamically Translating x86 to LLVM using QEMU. (2010).

[7] Michael Clark and Bruce Hoult. 2017. Rv8: a high performance RISC-V to x86 binary translator. In *1st Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[8] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA machine emulation for multicores. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO'17)*. 210–220.

[9] Cyberhaven. 2018. Translating binaries to LLVM with Revgen. http://s2e.systems/docs/Tutorials/Revgen/Revgen.html, accessed 2020-02-17.

[10] Alexis Engelke and Martin Schulz. 2020. Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. 172–184. https://doi.org/10.1145/3381052.3381319

[11] Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin. 2019. Transkernel: bridging monolithic kernels to peripheral cores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 675–692.

[12] Ding Yong Hong, Chun Chen Hsu, Pen Chung Yew, Jan Jan Wu, Wei Chung Hsu, Pangfeng Liu, Chien Min Wang, and Yeh Ching Chung. 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *International Symposium on Code Generation and Optimization (CGO)*. 104–113. https://doi.org/10.1145/2259016.2259030

[13] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, and Wei-Chung Hsu. 2011. LnQ: Building high performance dynamic binary translators with existing compiler backends. In *2011 International Conference on Parallel Processing*. IEEE, 226–234. https://doi.org/10.1109/ICPP.2011.57

[14] Andrew Jeffery. 2009. *Using the LLVM compiler infrastructure for optimised asynchronous dynamic translation in QEMU.* Master's thesis. University of Adelaide, Australia.

[15] Piyus Kedia and Sorav Bansal. 2013. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 101–115. https://doi.org/10.1145/2517349.2522718

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*.

[17] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN notices*, Vol. 42. 89–100. https://doi.org/10.1145/1250734.1250746

[18] Philip Reames. 2015. LLVM Phabricator: Calling convention for HHVM (D12681). https://reviews.llvm.org/D12681, accessed 2020-02-17.

[19] RISC-V Foundation. 2019. *The RISC-V Instruction Set Manual, Volume 1: User-Level ISA, Document Version 20190608-Base-Ratified.*

[20] Trail of Bits, Inc. [n. d.]. McSema. https://www.trailofbits.com/research-and-development/mcsema/, accessed 2020-02-17.