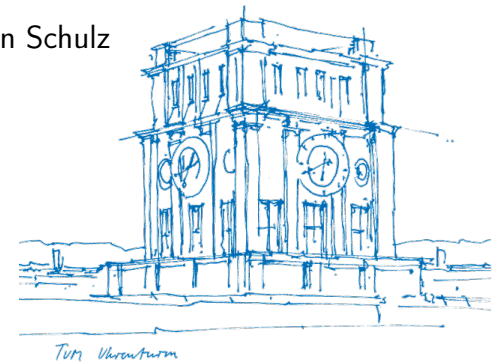
Efficient LLVM-based Dynamic Binary Translation

Alexis Engelke Dominik Okwieka Martin Schulz

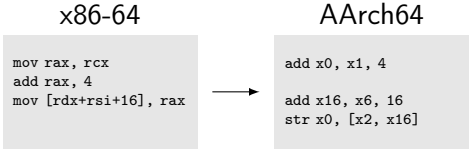
Chair of Computer Architecture and Parallel Systems
Department of Informatics
Technical University of Munich

VEE '21, Virtual, 2021-04-16



Dynamic Binary Translation

- Run program on other architecture, translate code for host CPU



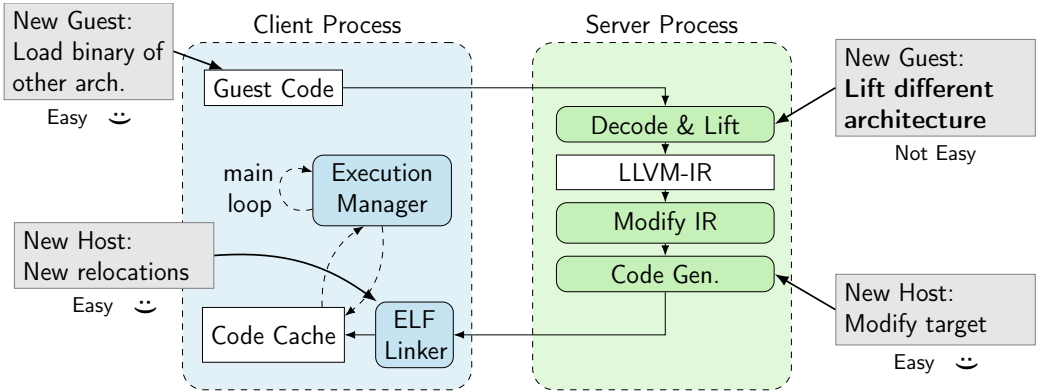
- Use-cases: compatibility, architecture research
- Example: QEMU-user, Rosetta 2

LLVM for Dynamic Binary Translation

- Problem: many DBT systems focus on **translation** performance
  - Example: QEMU
- Instead, use LLVM code generation for **run-time** performance
- **Instrew:** a fast LLVM-based dynamic rewriting framework
- But: tailored to x86-64

$\Rightarrow$ **Generalize Instrew** for other guest/host architectures

Instrew Architecture



Existing Lifting Approach

- Generate target-independent LLVM-IR
  - Use LLVM constructs where possible, e.g. vectors, comparisons
  - Helper functions for syscalls and `cpuid`
- Lifting stages:
  1. Decode instructions, recover control flow with basic blocks
     Use different decoder, annotate branches — Easy ☺
  2. Lift instructions/basic blocks
     Needs to be done... — Looks Hard 😕
  3. Fixup branches and PHI nodes

Arch. Differences: Registers

- Generally three types of register
  - General-Purpose/integer registers
  - Floating-point/vector registers
  - Status flags
- Architectures differ in **number** and **size**
  - x86-64: 16 GP registers; ARM: 31 GP registers
  - ARM: 4 status flags; RISC-V: none

$\Rightarrow$ Generalize register file and CPU state

Arch. Differences: Floating-Point Arith.

- Rounding mode: may be encoded in instruction
  - x86-64: never
  - AArch64: only conversions
  - RISC-V: always
- Generally difficult to represent in LLVM-IR $\rightsquigarrow$ intrinsics
- Relevant LLVM intrinsics may lower to library call

$\Rightarrow$ Provide software implementation of `floor/ceil/...`

Arch. Differences: Miscellaneous

- Program Counter: may point somewhere else...
  - Generally: points to current instruction
  - x86-64: points to next instruction
  - AArch32: points to second-next instruction
- Status flags may have same name, but different meaning

$\Rightarrow$ Move handling to architecture-specific part

Steps: How to Add New Architecture

1. Specify registers sizes and counts
2. Provide mapping of instruction semantics to LLVM-IR
3. Enhance decoded instructions with control flow information
4. Enhance list of supported architectures
5. **Profit!**

Case Study: Lifting RISC-V

- RISC-V: new, open architecture, ongoing standardization
- Few hardware available
  $\rightsquigarrow$ DBT aids ISA and compiler development
- Adapting Instrew: straight-forward process – follow described steps
- Only problem: LR/SC atomic loops – LLVM cannot represent this
  For now, treat them as non-atomic

Optimizations

- Keep guest registers in host registers
  - LLVM supports HHVM calling convention on x86-64
    allows to keep 12 guest regs. in host regs.
  - Previous approach: lifter generates appropriate code directly
  - New approach: transform lifted code separately
- Lift calls as calls and returns as returns
  - Use CPU return address stack
  - Continue decoding after function call

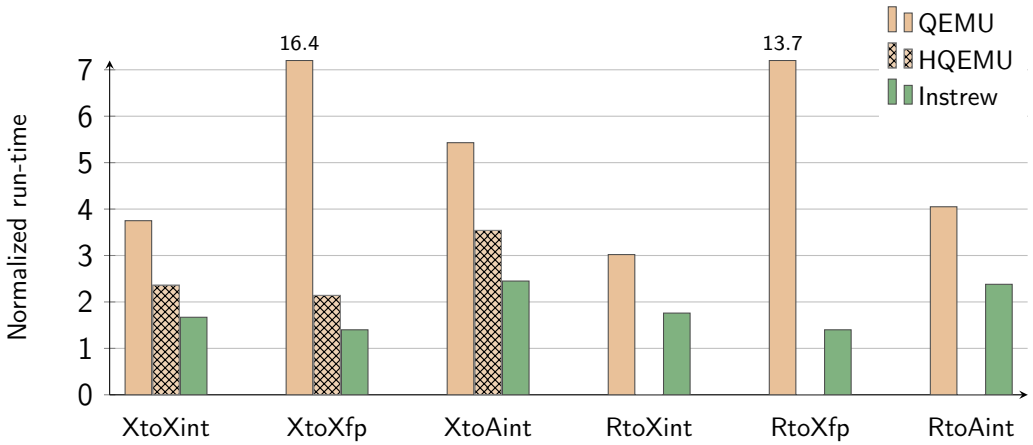
Evaluation

- Run on SPEC CPU2017 benchmarks
- Source architectures: x86-64, RISC-V64
- Target architectures: x86-64, AArch64
- Comparison with QEMU and HQEMU
- Baseline: natively optimized execution on host

System x86-64: 2×Intel Xeon CPU E5-2697 v3 (Haswell) @ 2.6 GHz (3.6 GHz Turbo), 17 MiB L3 cache; 64 GiB main memory; SUSE Linux 15.1 SP1; Linux kernel 4.12.14-95.32; 64-bit mode. Compiler: GCC 9.2.0 with `-O3 -march=x86-64`, implies SSE/SSE2 but no SSE3+/AVX. Libraries: glibc 2.32; LLVM 9.0.
System AArch64: 2×Cavium ThunderX2 99xx @ 2.5 GHz, 32 MiB L3 cache; 512 GiB main memory; CentOS 8; Linux kernel 4.18.0-193.14.2; 64-bit mode. Compiler: GCC 10 with `-O3`. Libraries: glibc 2.32; LLVM 9.0.
SPEC CPU2017 intspeed+fpspeed benchmarks, ref workload, single thread.

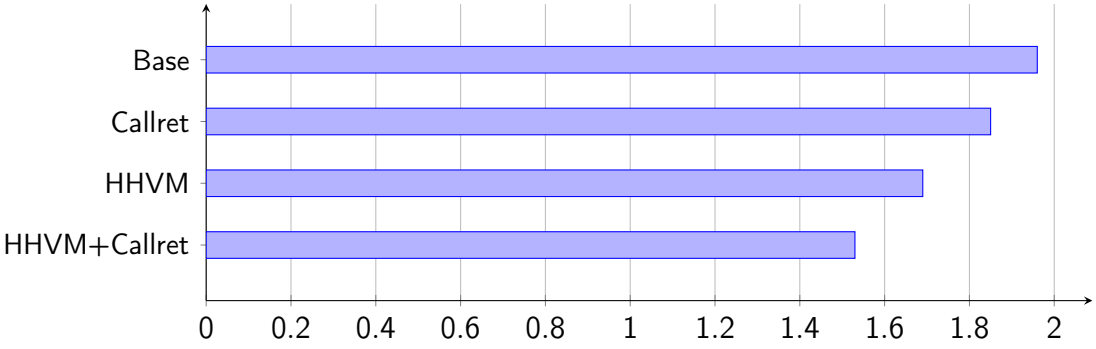
SPEC CPU2017 Results

Results normalized to native execution on host



Effect of Optimizations

Translating x86-64→x86-64, SPEC CPU2017



- Overall performance improvement: 22%

Instrew: Efficient LLVM-based Dynamic Binary Translation

- Fast Dynamic Binary Instrumentation/Translation based on LLVM
- Generalized to efficiently support other architectures
- Use host processor more effectively:
  - Speculate that function calls will properly return
  - Generic re-use host registers for guests
- Up to 50% less overhead compared to current LLVM-based DBT

Instrew is **Free Software**!

`https://github.com/aengelke/instrew`