

Code Generation for Data Processing

Lecture 12: Binary Translation

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2023/24

Motivation

- ▶ Run program on other architecture
- ▶ Use-case: application compatibility
 - ▶ Other architecture with incompatible instruction encoding
 - ▶ Applications using unavailable ISA extensions⁵⁷
- ▶ Use-case: architecture research
 - ▶ Development of new ISA extensions without existing hardware

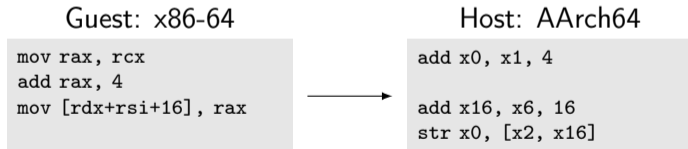
⁵⁷Exception-based implementation possible, but slow.

ISA Emulation

- ▶ Simplest approach: interpreting machine code
 - ▶ Simulate individual instructions, don't generate new code
- ▶ Frequently used approach before JIT-compilation became popular
- + Simple, works almost anywhere, high correctness
- Very inefficient

Binary Translation

- ▶ Idea: translate guest machine code to host machine code
- ▶ Replace interpretation overhead with translation overhead
- ▶ Difficult: very rigid semantics, but few code constraints imposed
 - ▶ Self-modifying code, overlapping instructions, indirect jumps
 - ▶ Exceptions with well-defined states, status flags



Warning for same-ISA translation: passing all instructions through as-is is a bad idea! Behavior might differ.

Static vs. Dynamic Binary Translation

Static BT

- ▶ Translate guest executable into host executable
- ▶ Do translation before execution

- + Low runtime overhead
- Binaries tend to be huge
- Cannot handle all cases
 - ▶ E.g., JIT-compiled code

Dynamic BT

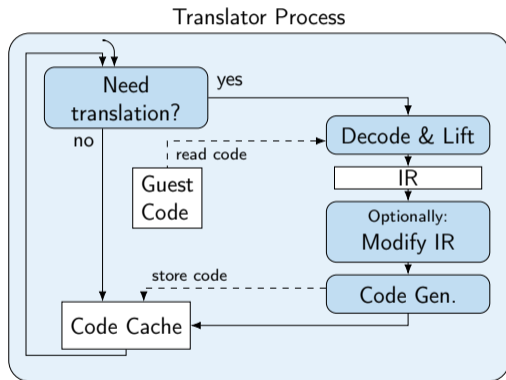
- ▶ Translate code on-the-fly during program execution
- ▶ Host code just lives in memory

- + Allows for high correctness
- ~ Can use JIT optimizations
- Translation overhead at run-time

Static Binary Translation

- ▶ Goal: create new binary for host with same functionality
- ▶ Program may access its own code/data in various ways
 - ▶ Guest binary must be retained as-is in-place
- ▶ Indirect jumps problematic
 - ▶ Need prediction of all possible targets
 - ▶ Keeping lots of dynamically possible entries prohibits optimizations
- ▶ JIT-compiled/self-modifying code impossible to handle
- ▶ Purely static translation impossible for the general case

Dynamic Binary Translation



- ▶ Iteratively translate code chunks on-demand
 - ▶ Typically basic blocks
- ▶ Store new code in-memory for execution and later re-use
- ▶ Code executed in same address space as original
 - ▶ Guest code/data must be accessible

Dynamic Binary Translation: Code Fragment

RISC-V Code

```
400560: slli a0, a0, 2
400564: jalr x0, ra, 0 // ret
```

Translation Engine

```
void emulate(uintptr_t pc) {
    uint64_t* regs = init();
    while (true)
        pc = translate(pc)(regs);
}
```

Semantical representation

```
uintptr_t trans_400560(uint64_t* regs) {
    regs[10] = regs[10] << 2;
    return regs[1];
}
```

```
// or with tail call:
_Noreturn void trans_400560(uint64_t* regs) {
    regs[10] = regs[10] << 2;
    translate(regs[1])(regs);
    // unreachable
}
```


Guest State

- ▶ Guest CPU state must be completely emulated
 - ▶ Registers: general-purpose, floating-point, vector, ...
 - ▶ Flags, control registers, system registers, segments, TLS base
- ▶ Memory – user-space emulation: use host address space
 - + no overhead through additional indirection
 - no isolation between emulator and guest
- ▶ Memory – system emulation: need software/hardware paging support
 - ▶ Software implementation: considerable performance overhead
 - ▶ Hardware implementation: guest and host need same page size

Guest Interface

- ▶ User-space emulation: OS interface needs to be emulated
 - ▶ Mainly system calls, but also vDSO, memory maps, ...
 - ▶ Host libraries are hard to use: ABI differences (e.g. struct padding)
 - ▶ Syscall emulation tedious: different flag numbers, arguments, orders
structs have different fields, alignments, padding bytes
- ▶ System-level emulation: CPU interface for operating systems
 - ▶ **Many** system/control registers
 - ▶ Different execution modes, memory configurations, etc.
 - ▶ Emulation of hardware components

Dynamic Binary Translation: Optimizations

- ▶ Fully correct emulation of CPU (and OS) is slow
 - ▶ Every memory access is a potential page fault
 - ▶ Signals can be delivered at any instruction boundary
 - ▶ *many* other traps...
- ▶ But: these “special” features are used extremely rarely
- ▶ Idea: optimize for common case
- ▶ Aggressively trade correctness for performance

Translation Granularity

- ▶ Larger translation granules allow for more optimization
 - ▶ E.g., omit status flag computation; fold immediate construction
- ▶ Instruction: great for debugging
- ▶ Basic block: allows for some important opt.
 - ▶ Easy to detect (up to next branch), easy to translate (no control flow)
- ▶ Superblock: up to next unconditional jump
 - ▶ Reduces transfers between blocks in fallthrough case
 - ▶ Translated code not necessarily executed
- ▶ Function: follow all conditional control flow
 - ▶ Allows most optimizations, e.g. for loop induction variables
 - ▶ Complex codegen, ind. jumps problematic, lot of code never executed

Chaining

- ▶ Observation: many basic blocks have constant successors
 - ▶ Often conditional branches with fallthrough and constant offset
- ▶ (Hash)map lookup and indirect jump after every block expensive
- ▶ Idea: after successor is translated, patch end to jump directly to that code
 - ▶ First execution is expensive, later executions are fast

```
// Initially generated code
// ...
mov rdi, 0x40068c
lea rsi, [rip+1f]
jmp translate_and_dispatch
1:.byte ... // store patch information
```

```
// After patching
// ...
jmp trans_40068c
// (garbage remains)
```

Chaining: Limitations

- ▶ First execution still slow, patching adds overhead
 - ▶ Can speculatively translate continuations
 - ▶ Translation of possibly unneeded code adds overhead
- ▶ Does not work for indirect jumps
 - ▶ Not necessarily predictable, esp. when considering a single basic block
 - ▶ Occur fairly often: function returns
- ▶ Removing translated functions from code cache becomes harder
 - ▶ Arbitrary other code may directly branch to translated chunk
 - ▶ Often solved by limiting chaining to same page or memory region

Return Address Prediction

- ▶ Observation: function calls very often return ordinarily
 - ▶ Return is an indirect jump, *but* highly predictable
 - ▶ But: even for “normal” code, this is not always the case: `setjmp/longjmp`, exceptions
- ▶ Hardware has return address stack keeping track of call stack
 - ▶ `call` pushes next address to stack, `ret` predicted to pop
 - ▶ Usually implemented as 16/32 entry ring buffer
- ▶ Idea: similarly optimize for common case of ordinary return

Return Address Prediction in DBT

- ▶ Option 1: keep separate shadow stack of guest/host target pairs
 - ▶ Can be implemented as ring buffer, too
 - ▶ Pop from stack needs verification of actual guest return address
 - Doesn't use host hardware return address prediction

- ▶ Option 2: use host stack as shadow stack
 - ▶ Allows using host `call/ret` instructions
 - ▶ Verification before/after return still required
 - Can degenerate, need to bound shadow stack
(guest might repeatedly call, discard return address, but never return)

Status Flags

- ▶ Observation: many status flags are rarely used
- ▶ But: eager computation can be expensive
 - ▶ E.g., x86 parity (PF) or auxiliary carry (AF)
- ▶ Idea: compute flags only when needed
- ▶ On flag computation, store operands needed for flag computation
- ▶ Flag usage in same block allows for optimizations
 - ▶ E.g., use idiomatic branches (`jle`, ...)
- ▶ Flag usage in different block: compute flags from operands
 - ▶ More expensive, but happens seldomly

Correct Binary Translation

- ▶ Goal 1: precise emulation – application works properly
- ▶ Goal 2: stealthness/isolation – application can't compromise DBT

- ▶ Problem: CPU and OS have huge and very-well-specified interfaces
 - ▶ ...and even if unspecified, software often depends on it
- ▶ Increased difficulty: different guest/host architectures
 - ▶ E.g., different page size or memory semantics
- ▶ Increased difficulty for user-space: different guest/host OS
 - ▶ Depending on syscall interface, nearly impossible (see WSL1)

POSIX Signals

- ▶ POSIX specifies signals, which can interrupt program at any point
- ▶ Kernel pushes signal frame to stack with user context and calls signal handler
- ▶ Signal handler can read/modify user context and continue execution

- ▶ Synchronous signals: e.g., SIGSEGV, SIGBUS, SIGFPE, SIGILL
 - ▶ For example, due to page fault or FP exception
 - ▶ Delivered in response to “error” in current thread

- ▶ Asynchronous signals: e.g., SIGINT, SIGTERM, SIGCHILD
 - ▶ Delivered externally, e.g. using `kill`
 - ▶ Can be delivered to any thread at any time
 - ▶ (usually a bad idea to use them)

Correct DBT: Signals

- ▶ DBT must register signal handler and propagate signals
- ▶ Synchronous signals
 - ▶ Delivered at “constrainable” points in program
 - ▶ *Must* recover fully consistent guest architectural state
 - ▶ JIT-compiled code must be sufficiently annotated for this
- ▶ Asynchronous signals
 - ▶ Can really be delivered at any time
 - ▶ Must not be immediately delivered to guest
 - ↪ Usually delivered when convenient
 - ▶ But: real-time signals have special semantics

Correct DBT: Memory Accesses

- ▶ Option: emulating paging in software (slow, but works)
 - ▶ Every memory accesses becomes a hash table lookup
 - ▶ Shared memory still problematic: host OS might have larger pages
- ▶ Using host paging is much faster, but problematic for correctness
- ▶ Host OS might have larger pages
- ▶ Every memory access can cause a page fault (see signal handling)
- ▶ Guest can access/modify arbitrary addresses in its address space... including the DBT and its code cache
- ▶ Tracking read/write/execute permissions, e.g. check X before translation

Correct DBT: Memory Ordering

- ▶ CPUs (aggressively) reorder memory operations
 - ▶ x86: total store ordering – stores can be reordered after loads
 - ▶ Most others: weak ordering – everything can be reordered
- ▶ Relevant for multi-core systems: other thread can observe ordering
- ▶ Atomic operations and fences limit reordering (e.g., acq/rel/seqcst)

- ▶ Emulating weak memory on TSO: easy
- ▶ Emulating TSO on weak memory: hard
 - ▶ Can try to make all operations atomic
 - ▶ Atomic operations often need alignment guarantees (not on x86)
 - ▶ Only viable solution so far: insert fences everywhere

Correct DBT: Self-modifying Code

- ▶ Writable code regions (or with `MAP_SHARED`) can change at any time
- ▶ Idea: before translation, remap as read-only
- ▶ On page fault (`SIGSEGV`), remove relevant parts from code cache
 - ▶ Requires code cache segmentation and mapping of code to original page
- ▶ When executing possibly modifiable code: every store can change code!
- ▶ Doesn't easily work for shared memory, need to track this, too
 - ▶ Might be impossible when shared with other process

Correct DBT: Floating-point

- ▶ Floating-point arithmetic is standardized in IEE-754
- ▶ ...except for some details and non-standard operations
- ▶ x86 `maxsd`: if one operand is NaN, result is second operand
- ▶ RISC-V `fmax.d`: if one operand is NaN, result is non-NaN operand
- ▶ AArch64 `fmax`: if one operand is NaN, result is NaN operand
 - ▶ Unless configured differently in `fpcr`
- ▶ Correctness typically requires software emulation (e.g., QEMU does this)

Correct DBT: OS and CPU Specifics

- ▶ Emulating all syscalls correctly is hard
 - ▶ Version-specifics, structure layouts, feature support
 - ▶ Huge interface
- ▶ `/proc/self/*` – how to emulate?
 - ▶ Catch all file system accesses? Follow all possible symlinks?
 - ▶ What if `procfs` is mounted somewhere else?
- ▶ `cpuid` – how to emulate?
 - ▶ Cache sizes, processor model, ...
 - ▶ Application can do timing experiment to detect DBT

Binary Translation – Summary

- ▶ ISA emulation often used for cross-ISA program execution
- ▶ Binary Translation allows for more performance than interpretation
- ▶ Static Binary Translation handles whole program ahead-of-time
- ▶ Dynamic Binary Translation translates code on-demand
- ▶ ISA often highly restricts optimization possibilities
- ▶ Optimizations typically very low-level
- ▶ Correct emulation of CPU/OS challenging due to large interface

Binary Translation – Questions

- ▶ What are use cases of binary translation?
- ▶ What is the difference between static and dynamic binary translation?
- ▶ Why is static BT strictly less powerful than dynamic BT?
- ▶ What are typical translation granularities for DBT?
- ▶ How to optimize control flow handling in DBT?
- ▶ Why is correct binary translation hard to optimize?
- ▶ What problem can occur when not emulating paging for user-space emulation?