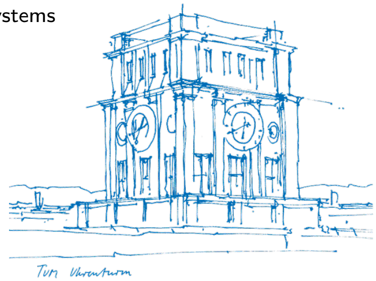
Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation

Alexis Engelke   Martin Schulz

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

VEE 2020, virtual



Program Instrumentation

- Enhance program with additional code
- Use-cases: analysis, debugging, optimization, portability
- Dynamic Binary Instrumentation (DBI)
  - Binary code instrumented/modified at run-time
  - Works without recompiling program and libraries
  - Very popular approach $\implies$ many frameworks available

DBI Frameworks

- Most popular framework: Valgrind
  - Program behavior can be extended and modified
  - Allows for extensive code transformations
- Usual focus: low rewriting time, not overall performance
  - Few optimizations, instrumented code has low quality

Solution: use standard compiler back-end

LLVM for DBI

- LLVM features high quality optimizer/code generator
  - Built-in JIT-compiler allows use at run-time
- DBILL uses LLVM JIT-compiler for code generation
  - Machine code → TCG IR → LLVM-IR

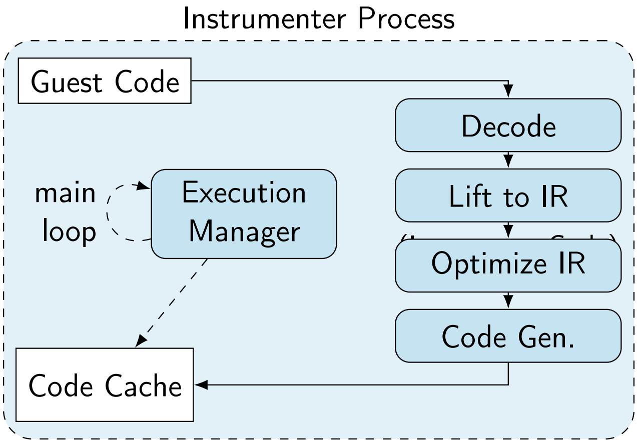
  + Easy to support several architectures

  – No (efficient) floating-point/SIMD support

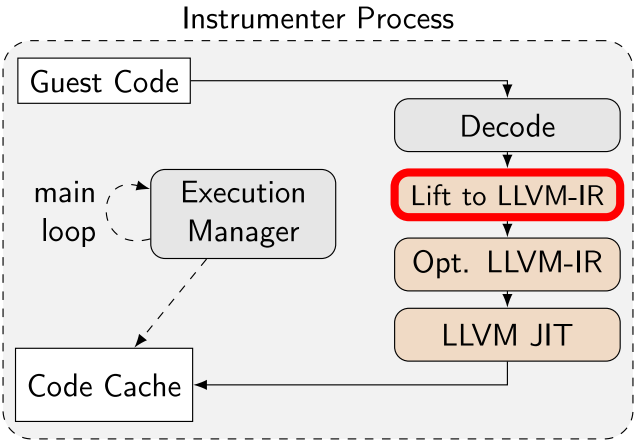
  – Optimizations limited to basic blocks

Solution: lift machine code directly to LLVM-IR

Classical DBI Architecture



Architecture Using LLVM-IR



Lifting x86-64 Code to LLVM-IR

- Focus on most common x86-64 architecture
- Requirements:
  1. LLVM-IR must be handled well by optimizer/code gen.
     $\rightsquigarrow$ *run-time performance*
  2. Avoid unnecessary transformations
     $\rightsquigarrow$ *reduced rewriting time*
  3. Only use architecture-independent LLVM-IR constructs
     $\rightsquigarrow$ *retargetability* (assuming same pointer size)

Implemented in our lifting library: *Rellume*

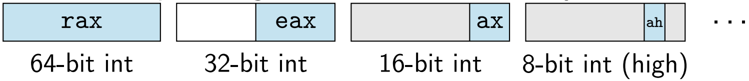
Lifting Stages

1. Decode & Recover Control Flow
   - Decode machine code, following jump targets
   - Stops on indirect branches, calls, returns
   - Split into basic blocks
2. Lift Instructions Individually
   - Create skeleton LLVM-IR function
   - Generate LLVM-IR for each instruction
3. Create Epilogue & Fixup Branches
   - Add branches between basic blocks, map data flow

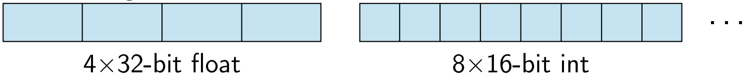
Register Facets

- **Facet**: typed view on a register (part)
- Store and propagate multiple facets for registers
  - Relevant for partial access and different data types
  - Avoids many insert/extract/cast ops $\rightsquigarrow$ better code
- Benefit: better optimizations across basic blocks

- General Purpose registers: scalar facets only



- Vector registers: scalar and vector facets



Example

```
define void @func_40061e(i8* %cpu) {
prologue:
  ; ...


bb_40061e:
  ; ...


epilogue:
  ; ...



}
```

Single parameter:
CPU struct

- Instruction Ptr.
- Registers
- Status Flags
- ...

Example

```
define void @func_40061e(i8* %cpu) {
prologue:
  %rip_p_i8 = gep i8, i8* %cpu, i64 0
  %rip_p = bitcast i8* %rip_p_i8 to i64*
  %rsp_p_i8 = gep i8, i8* %cpu, i64 40
  %rsp_p = bitcast i8* %rsp_p_i8 to i64*
  %rsp = load i64, i64* %rsp_p
  ; ... load other registers ...
  br label %bb_40061e

bb_40061e:
  ; ...
epilogue:
  ; ...
}
```

Construct ptrs. into CPU struct

Load registers into SSA variables

Example

```
define void @func_40061e(i8* %cpu) {
prologue:
  ; ...



bb_40061e:
  %rsp_2 = phi i64 [%rsp, %prologue]
  ; sub rsp, 176
  %rsp_3 = sub i64 %rsp_2, 176
  ; ... compute flags ...
  br label %epilogue

epilogue:
  ; ...
}
```

Lift instruction semantics

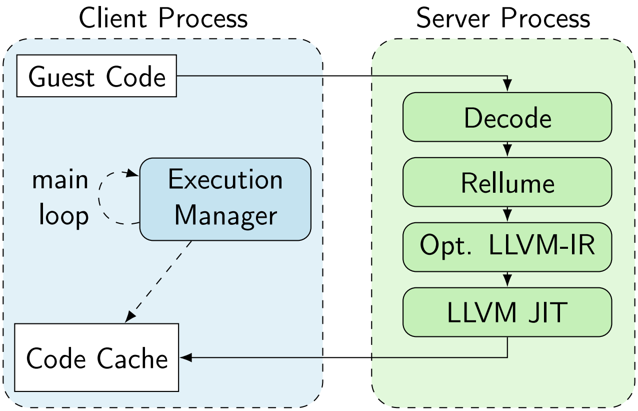
Example

```
define void @func_40061e(i8* %cpu) {
prologue:
  ; ...


bb_40061e:
  ; ...

epilogue:
  %rsp_4 = phi i64 [%rsp_3, %bb_40061e]
  store i64 %rsp_4, i64* %rsp_p
  ; ... store flags ...
  store i64 0x400625, i64* %rip_p
  ret void
}
```

Store new values

Store new RIP

Instrew Architecture



Client-Server Architecture

- Instrew Server
  - Rewrites code chunks on client request
  - Returns an ELF object file containing rewritten code
- Instrew Client
  - Manages execution and local code cache
  - Sends request with program code to server process
  - Relocates and links ELF files
- Communication: custom IPC protocol

Translation Details

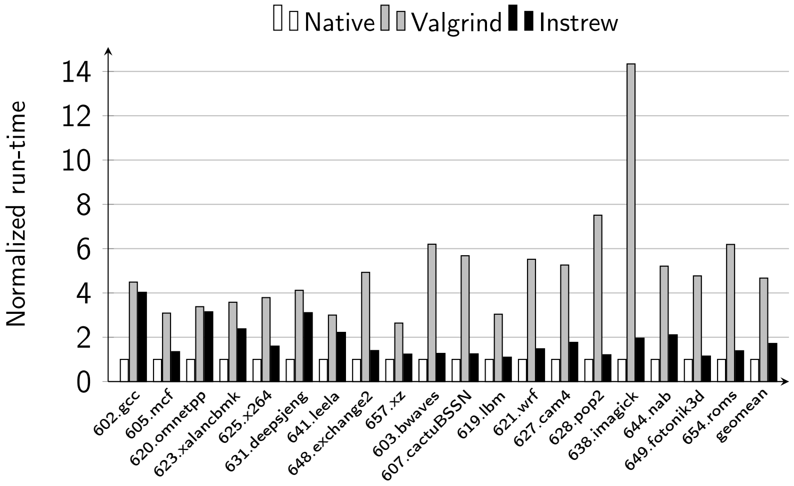
- Translate code chunks with **function granularity**
  - Decode until call/ret/indirect jump
  - Enables power of LLVM's whole-function optimizations
  - Reduces number of rewrite requests
- Use special calling convention
  - Reduces number of memory accesses to CPU structure
- Don't compute flags before `call`/`ret`
  - Flags extremely rarely used to pass args/return vals

Evaluation

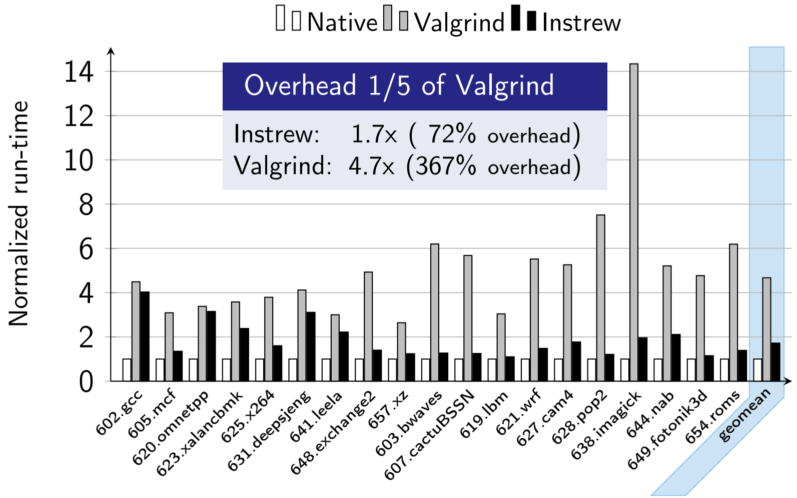
- Run on SPEC CPU2017 benchmarks
- Comparison with Valgrind
  - Most popular tool with similar set of use-cases
- No comparison with DBILL (no sources) and Pin (different scope of code modifications)

System: 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 12; 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 2.22; LLVM 9.0. SPEC CPU2017 intspeed+fpspeed benchmarks, ref workload, single thread. Comparison: Valgrind 3.15.0.

SPEC CPU2017 Results



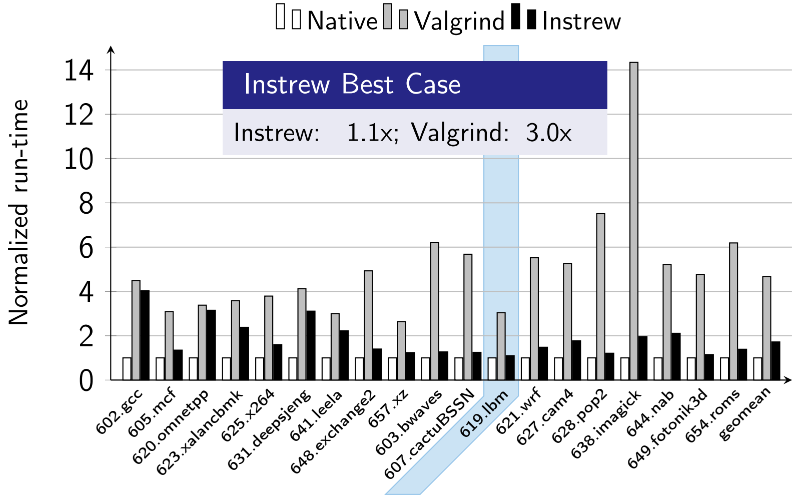
SPEC CPU2017 Results



Overhead 1/5 of Valgrind

Instrew:	1.7x	(72% overhead)
Valgrind:	4.7x	(367% overhead)

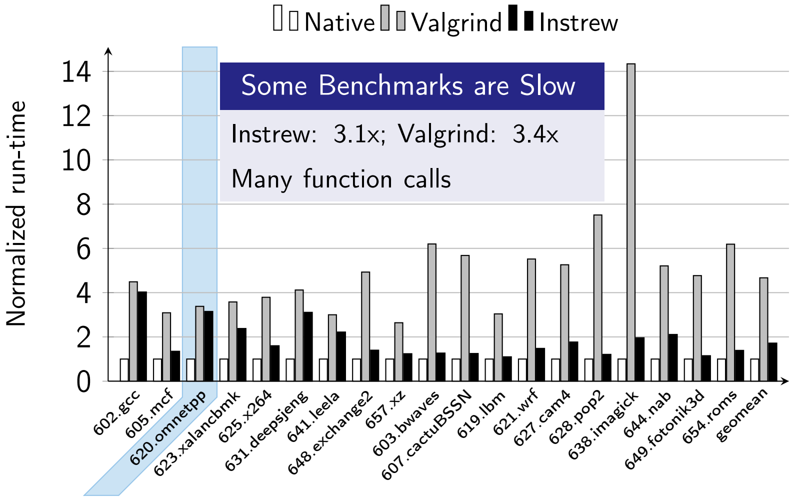
SPEC CPU2017 Results



Instrew Best Case

Instrew: 1.1x; Valgrind: 3.0x

SPEC CPU2017 Results

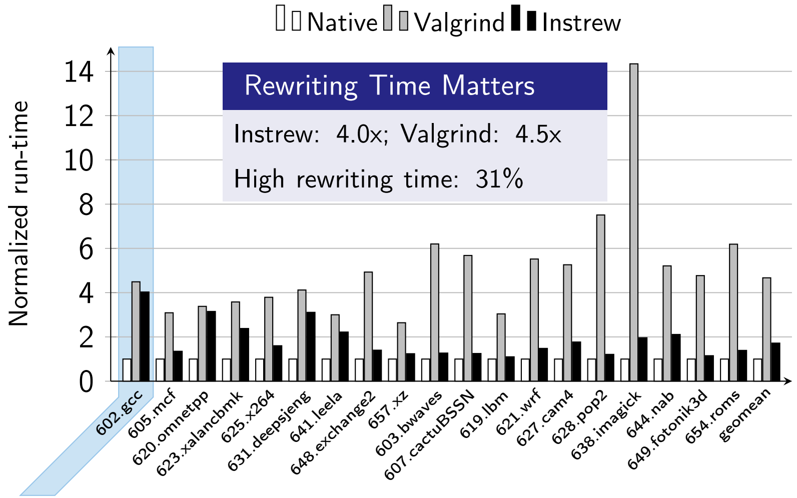


Some Benchmarks are Slow

Instrew: 3.1x; Valgrind: 3.4x

Many function calls

SPEC CPU2017 Results



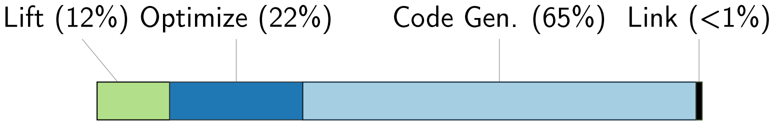
Rewriting Time Matters

Instrew: 4.0x; Valgrind: 4.5x

High rewriting time: 31%

Rewriting Overhead

- Mean Rewriting Time: 0.94%
  - Notable exception: 602.gcc with 31%
- Mean Rewriting Time Breakdown:
  - Most time spent for machine code generation
  - SelectionDAG instruction selector known to be slow
  - Replacement GlobalISel not yet ready



Discussion

- Clear performance improvement over Valgrind
  - More and better optimizations
  - High-quality code generator
- Expected to be faster than DBILL
  - Instrew: 109% overhead on SPEC CPU2017 INT
  - DBILL: 240% overhead on SPEC CINT2006
- Biggest drawback: rewriting time
  - Needs to amortize over run of program
  - Ongoing developments in LLVM will reduce this issue

Instrew: LLVM-based DBI

- Dynamic Binary Instrumentation based on LLVM
- First to lift whole functions directly to LLVM-IR, use LLVM's high-quality optimizer/JIT code generator
- Client-server approach enabling further optimizations
- Reduction of overhead by 80% compared to Valgrind

Instrew is **Free Software**!

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