Robust Practical Binary Optimization at Run-time using LLVM

Alexis Engelke    Martin Schulz

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

LLVM-HPC 2020, virtual
Run-time Binary Optimization

- Performance is important for HPC
  ⇒ compilers provide many optimizations/transformations

- Compiler cannot exploit run-time information
  - E.g., dynamic data layout, configurations
  - Compile-time specialization generally not possible
  ⇒ generate optimized code at run-time

- *Simple* recompilation is no solution for HPC
  - Source/compiler inaccessible on compute node; high compilation times

  ⇒ *optimize machine code* further at run-time

Motivation & Prior Work

Approach

Lifting Machine Code

Optimization & Lowering

Benchmarks
Application-guided Optimization

- Hard to predict run-time constant data and performance impact
  ⇒ Optimization **guided explicitly by application developer**

- API approach previously explored by DBrew
  - Library with C API for optimizing functions
  - Parameters and memory regions marked as constant
  - Generates specialized function for these values
DBrew: Library for Rewriting Machine Code

- DBrew: simple optimization at assembly-level
  - Propagates constant values, aggressive loop unrolling and inlining
  - Very fast rewriting procedure
  - Very limited optimizations

- DBrew-LLVM: post-process DBrew output with LLVM
  - Assembly from DBrew lifted to LLVM-IR, using LLVM code generator
  - Better quality of produced code
  - Limited by DBrew and strict separation of lifting/optimization

⇒ Go directly from machine code to LLVM-IR, avoiding DBrew
Optimizing Machine Code with LLVM

Lifting

- CPU Semantics
- Function Semantics
- Opt.
- Code Gen. (MCJIT)

Optimization & Lowering

- Specialized Function
- Opt.

Motivation & Prior Work

Approach

Lifting Machine Code

Optimization & Lowering

Benchmarks
Example using Rewriting API

```c
int sub(int a, int b) { return a - b; }
int main(void) {
    BinoptHandle h = binopt_init();
    BinoptCfgRef bc = binopt_cfg_new(h, sub);
    binopt_cfg_type(bc, 2, BINOPT_TY_INT32,
                    BINOPT_TY_INT32, BINOPT_TY_INT32);

    binopt_cfg_set_parami(bc, 1, 42);

    int(*sub42)(int, int) = binopt_spec_create(bc);

    int res = sub42(48, 16);
}
```
Example using Rewriting API

```c
int sub(int a, int b) { return a - b; }
int main(void) {
    BinoptHandle h = binopt_init();
    BinoptCfgRef bc = binopt_cfg_new(h, sub);
    binopt_cfg_type(bc, 2, BINOPT_TY_INT32,
                    BINOPT_TY_INT32, BINOPT_TY_INT32);
    binopt_cfg_set_parami(bc, 1, 42);
    int(*sub42)(int, int) = binopt_spec_create(bc);
    int res = sub42(48, 16);
}
```

Create new library handle and configuration for sub.
Example using Rewriting API

```c
int sub(int a, int b) { return a - b; }
int main(void) {
    BinoptHandle h = binopt_init();
    BinoptCfgRef bc = binopt_cfg_new(h, sub);
    binopt_cfg_type(bc, 2, BINOPT_TY_INT32,
                    BINOPT_TY_INT32, BINOPT_TY_INT32);
    binopt_cfg_set_parami(bc, 1, 42);
    int(*sub42)(int, int) = binopt_spec_create(bc);
    int res = sub42(48, 16);
}
```

Set function signature

Motivation & Prior Work
Approach
Lifting Machine Code
Optimization & Lowering
Benchmarks
Example using Rewriting API

```c
int sub(int a, int b) { return a - b; }
int main(void) {
    BinoptHandle h = binopt_init();
    BinoptCfgRef bc = binopt_cfg_new(h, sub);
    binopt_cfg_type(bc, 2, BINOPT_TY_INT32,
                    BINOPT_TY_INT32, BINOPT_TY_INT32);

    binopt_cfg_set_parami(bc, 1, 42); // Set 2\textsuperscript{nd} param to const 42

    int (*sub42)(int, int) = binopt_spec_create(bc);

    int res = sub42(48, 16);
}
```
Example using Rewriting API

```c
int sub(int a, int b) { return a - b; }
int main(void) {
    BinoptHandle h = binopt_init();
    BinoptCfgRef bc = binopt_cfg_new(h, sub);
    binopt_cfg_type(bc, 2, BINOPT_TY_INT32, BINOPT_TY_INT32, BINOPT_TY_INT32);
    binopt_cfg_set_parami(bc, 1, 42);

    int(*sub42)(int, int) = binopt_spec_create(bc);

    int res = sub42(48, 16);
}
```

Create specialized function for constant parameter
Example using Rewriting API

```c
int sub(int a, int b) { return a - b; }
int main(void) {
    BinoptHandle h = binopt_init();
    BinoptCfgRef bc = binopt_cfg_new(h, sub);
    binopt_cfg_type(bc, 2, BINOPT_TY_INT32, BINOPT_TY_INT32, BINOPT_TY_INT32);
    binopt_cfg_set_parami(bc, 1, 42);
    int(*sub42)(int, int) = binopt_spec_create(bc);
    int res = sub42(48, 16);
}
```

Call optimized function with same signature as original
res will be 6 (instead of 32)
Lifting: Instruction Semantics

Raw machine code $\rightarrow$ LLVM-IR with same semantics

- First lifting step: lift machine code semantics to LLVM-IR
- Use *Rellume*; library designed for dynamic binary instrumentation
  - Focus on performance of lifted code and lifting process itself
- Reconstruct control flow by following direct jumps
  - Stops on: calls, return, indirect jumps, unsupported instrs.
- Create single LLVM-IR function with same semantics
- Virtual CPU state stored in CPU state struct, given as parameter
  - Contains rip, registers, flags, etc.
Lifting: Function Semantics

LLVM-IR code semantics → Original function interface

- Create LLVM-IR function with same signature as original
- Allocate a fixed-size virtual stack
  - Direct access to stack not possible in LLVM-IR
- Allocate and fill initial CPU state according to System V ABI
- Call & inline Rellume-lifted function
- Extract return values from CPU state
- CPU state removed during optimization by SROA
Lifting: Further Code Discovery

▸ Targets of indirect calls/jumps may become known after optimization
  ▸ E.g., callback functions, function dispatch, switch statements

▸ Problem: not predictable during initial lifting

▸ Solution: fixed-point code discovery
  1. Initially lift the function
  2. Optimize all lifted code
  3. Check for new known jump targets
     If so, lift them and go back to optimization
Shared Library Code

- Code in shared libraries can be lifted/optimized, too
- Practice experience: problematic

- High performance rewriting requires highly selective optimization
  - Complex functions with few optimization opportunities: `malloc`, etc.
  - Highly optimized target-specific functions: `memcpy`, etc.

- Lazy binding: target is unknown; address in writable memory
- Avoid lifting the dynamic linker itself

⇝ For now, detect and avoid PLT stubs
Machine Code Optimization with LLVM

- Generally use standard -03 pipeline
- Minor modifications necessary for new context
- Alias analysis plug-in for CPU state $\rightarrow$ less memory accesses
- New pass: constant propagation from memory
  - Context specific setting: loads from constant addresses
  - If target is configured as constant: load can be removed
- New pass: fold ptrtoint–add–inttoptr to getelementptr
  - Better constant propagation and alias analysis
  - May be integrated in upstream InstrCombine (?)
Lowering Indirect Jumps

- Sometimes, indirect jumps cannot be resolved
  ⇒ no further optimizations possible
- Goal: optimize until that jump, then continue with original code

- Challenges:
  - All registers must have values as in original code
  - Stack pointerrspmust point to top of virtual stack
  - Function return must be intercepted to restore to callee stack

- Solved (ab)using interrupt return (iretq)
Benchmarks

- Micro-benchmark: apply arbitrary sparse 2D stencil
  - Specialize generic kernel for 4-point stencil
  - Comparison: original, run-time binary opt., compile-time opt.

- Real-world code: widely-used image processing kernels (GEGL)
  - gblur-1d: specialize on variably-sized dense 1D stencil
  - bilateral-filter: specialize on variably-sized 2D dense stencil
  - Comparison: original, with run-time binary optimization

Software: GCC 9.2.1 with -Ofast -mtune=native, run-time opt. with LLVM 9.0, glibc 2.30. System: Intel Core i5-8250U @ 1.6 GHz (3.4 GHz Turbo); Fedora 31; Linux kernel 5.7.15; 64-bit mode. AVX disabled for all generated code.
Benchmark Results

Motivation & Prior Work
Approach
Lifting Machine Code
Optimization & Lowering

Rewriting time
31ms (2.1%)
23ms (0.9%)
39ms (1.1%)

Normalized Execution Time
Discussion

- Further performance increase with run-time optimizations
- Profitable even on optimized real-world applications
- Rewriting time negligible, depending on workload

- Optimization time strongly depends on code complexity
- Technique must be used very selectively
Conclusions

- Performance increase by run-time optimization of compiled code
- LLVM-IR reconstructed from raw machine code
- Apply standard LLVM optimizations and code generator with only minor modifications
- Robust integration: in error case, continue in original code
- First experience with real-world code: optimization cost comparably low

Future work:
- Compiler-support for estimating optimization potential
- System-wide infrastructure for caching optimized code