How to generate efficient code for new hardware?

Greta Yorsh
Queen Mary University of London
Software tuning for micro-architecture

• **Goal:** best performance out of hardware

• Production compilers
  • open source: gcc, llvm,...
  • proprietary: icc, armcc, visualc...

• Highly optimized
  • generate efficient code for existing CPUs
  • keeping compilation times low enough to be used productively in large software projects
Software tuning for micro-architecture

• Compiler optimizations - register allocation, instruction selection, instruction scheduling
  • pipeline description: integer, floating point, vector
  • cost of branch and conditional execution
  • cost of addressing modes
  • preload strategy
  • load/store multiple and double
  • vector operations
  • compiler intrinsics

• Library routines - handcrafted assembly
  • memory: memcpy, memset, memchr, ...
  • strings: strcmp, strchr, strlen, ...
  • math: integer division, arithmetic, directed rounding, ...

• Operating system and applications
Fine-tuning production compiler for new CPU

• Changes to code generation and libraries
• Development process: implement, validate, and benchmark
• Even simple changes may take many months of expert compiler engineer's time
• Varying impact on performance of different benchmarks
• Hard to justify such changes in a production compiler

• Compilers not changing fast enough to keep up with
  • variety of hardware designs
  • fast-paced design cycles
Software tuning for micro-architecture

• Goal: best performance out of hardware

• Practice: “just make it go faster”

• Unrealized potential performance gains
Can it be fixed for the next version?

• Hardware designers have many constraints when working on new microarchitectures and revising existing ones

• Hardware: optimize common instruction sequences

• Compilers: don’t emit instructions that are slow
Code generation and optimization

• Given code $p$, generate code $s$ such that

• **Correctness**: $s$ correctly implements $p$
  • the observable behaviors of $s$ are a subset of the observable behaviors of $p$

• **Optimality**: the cost of $s$ is minimal with respect to cost function $c$
  • $c(s) = \min\{c(s') \mid s' \text{ implements } p\}$
Example

```c
int sign (int x)
{
    if (x < 0) return -1;
    if (x > 0) return 1;
    return 0;
}
```

```asm
CMP   R0, #0   ; input value in R0
MOVGT R0, #1
MOVLT R0, #-1  ; return value in R0
```
Superoptimizer \((p, a)\)  // \(p\) is straightline code
Tests := ∅
for \(n := 0, 1, 2, \ldots\) do
  for each \(s \in I^n\) do
    if check\((s, \text{Tests})\) then
      \(\phi := \text{encode}(p, s)\)
      if not satisfiable\((\phi)\) then return \(s\)
      cex ← getModel\((\phi)\)
      Tests := Tests \(\cup\) getTests\((p, cex)\)
Unbounded Superoptimizer

\[ \chi := \text{encodeCorrectness}(p, a) \]

if not satisfiable(\( \chi \)) then return FAIL
repeat
  \( m := \text{getModel}(\chi) \)
  \( \chi := \chi \land \text{encodeBound}(m, c) \)
until not satisfiable(\( \chi \))
s := \text{getCode}(m)
return \( s \)
Unbounded Superoptimizer $(p,a,c)$

$\chi := \text{encodeCorrectness}(p,a)$

if not satisfiable$(\chi)$ then return FAIL

repeat
    $m := \text{getModel}(\chi)$
    $\chi := \chi \land \text{encodeBound}(m,c)$
until not satisfiable$(\chi)$

$s := \text{getCode}(m)$
return $s$
Encoding Correctness

\[ \chi \] = instruction sequences \( s \) that correctly implement \( p \) in \( a \)

\[ \chi = \forall x, x', y, y'. p \land a \land o \]

- **IR constraints**
  semantics of code \( p \)

- **ISA constraints**
  semantics of \textit{arbitrary} instruction sequence in target architecture

- **observational equivalence constraints**
ISA Constraints

∀j. 0 ≤ j < n . \( \bigwedge_{i \in I} \text{instr}(j) = i \rightarrow \tau_i(\text{state}(j), \text{state}(j+1)) \)

I = \{ \text{ADD } \text{r0, r1} \}
\{ \text{ADD } \text{r1, r2} \}
\{ \text{MUL } \text{r0, r2} \}
\{ \text{LDR } \text{r0, [r1]} \} \ldots \}

length of instruction sequence

instruction at location j is i

semantics of instruction i
Shifting the search into the solver

- size of constraints does not depend on the candidate sequence of instructions
  - size of $\phi$ depends on $n$, size of $\chi$ depends on $a$
- more complex formula may take longer to solve
- opportunity to reuse reasoning within the solver
- stop at any time with a correct possibly suboptimal solution that can be improved upon later
  - models of $\phi$ are counterexamples
  - models of $\chi$ are correct instruction sequences
- fine-grained control: compilation time vs quality of generated code
- loop-free code not just straight line code
Preliminary Prototype

C program \( \rightarrow \) clang \( \rightarrow \) LLVM IR \( \rightarrow \) constraints \( \rightarrow \) Z3

ARM ISA Semantics

Cost Model

\( a \)

\( c \)

length of \( s \)

\( p \)

loop-free code

\( s \)

bitvectors
arrays
quantifiers
uninterpreted functions

loop-free code
def i32 sign (i32 x):
    ; <label>:L0
    v1 = icmp slt i32 x, 0
    br i1 v1, label L1, label L2
    ; <label>:L1
    br label L5
    ; <label>:L2
    v2 = icmp sgt i32 x, 0
    br i1 v2, label L3, label L4
    ; <label>:L3
    br label L5
    ; <label>:L4
    br label L5
    ; <label>:L5
    v3 = phi([L1,-1],[L3,1],[L4,0])
    ret i32 v3

L0 ← ρ1=(px<0)∧
    (((ρ1=true)∧L1)∨((ρ1=false)∧L2))
L2 ← ρ2=(px>0)∧
    (((ρ2=true)∧L3)∨((ρ2=false)∧L4))
L1 ← L5∧(ρ3 =−1)
L3 ← L5∧(ρ3 =1)
L4 ← L5∧(ρ3 =0)
L5 ← true

LLVM IR Constraints
ARM ISA Constraints

\[ \forall j. \ 0 \leq j < n . \ \bigwedge_{i \in I} \text{instr}(j)=i \rightarrow \tau_i(\text{state}(j), \text{state}(j+1)) \]

\[ \forall i. \ 0 \leq j < n. \]
\text{instr}(j)=“SUB R0, R1“ \rightarrow \text{state}(j+1)[R0]=\text{state}(j)][R0]=\text{state}(j)[R1] \land \text{PRES}
\text{instr}(j)=“MOV R0, R1“ \rightarrow \text{state}(j+1)[R0]=\text{state}(j)[R1] \land \text{PRES}
\text{instr}(j)=“MOVGT R0, R1“ \rightarrow \text{ite}(\text{GT,}
\text{state}(j+1))[R0]=\text{state}(j)[R1] \land \text{PRES,}
\text{state}(j+1)=\text{state}(j))

…..
Unbounded Superoptimizer \( (p, a, c) \)

\[ \chi := \text{encodeCorrectness}(p, a) \]

if not satisfiable(\( \chi \)) then return FAIL

repeat

\[ m := \text{getModel}(\chi) \]

\[ \chi := \chi \land \text{encodeBound}(m, c) \]

until not satisfiable(\( \chi \))

\[ s := \text{getCode}(m) \]

return \( s \)

\[ \lceil \chi \rceil \] instruction sequences that correctly implement \( p \) in a

if cost is length of generated instruction sequence and \( m \) represents a sequence of length \( K \), then \( \text{encodeBound}(m, c) \) returns \( n < K \)

cost of instruction sequences is less than \( c(\text{getCode}(m)) \)
Initial Feasibility Study

• Goal: can a solver handle complex constraints that arise from our encoding?

• Success on 17 out of 25 microbenchmarks but slow
  • under 30 min and up to 4 iterations per benchmark
  • unoptimized encoding
  • no solver heuristics
  • generated better code than gcc on 3 out of 17
Challenges

• Solver termination and completeness
• Compilation time
• Correctness of constraints
• Hardware specification availability
• Cost models availability
• Loops
• memory
• loops
• vectorizer
• integrate in other toolchains

• cost models
• target other ISA
• target GPU (shuffles)
• aid hardware design

• quantifiers
• maxSMT
• different combination of theories
• different solver (Lean?)
• partial evaluation
• search heuristics