

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

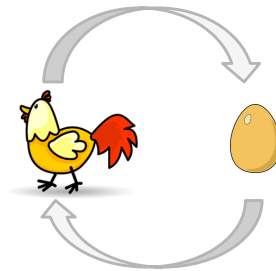
- Compilers not changing fast enough to keep up with
 - variety of hardware designs
 - fast-paced design cycles

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



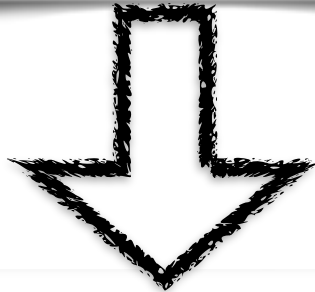
- Compilers: don't emit instructions that are slow

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

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



```
CMP    R0, #0    ; input value in R0
MOVGT  R0, #1
MOVLT  R0, #-1   ; return value in R0
```


Superoptimization

```
Superoptimizer(p, a)           // p is straightline code
Tests :=  $\emptyset$ 
for n:=0,1,2,... do
  for each s  $\in$   $I^n$  do
    if check(s, Tests) then
       $\varphi$  := encode(p, s)
      if not satisfiable( $\varphi$ ) then return s
      cex  $\leftarrow$  getModel( $\varphi$ )
      Tests := Tests U getTests(p, cex)
```

$\llbracket \varphi \rrbracket$ observed behaviour
of p and s differ

Unbounded Superoptimization

```
UnboundedSuperoptimizer (p, a, c)  
   $\chi$  := encodeCorrectness(p, a)  
  if not satisfiable( $\chi$ ) then return FAIL  
  repeat  
    m := getModel( $\chi$ )  
     $\chi$  :=  $\chi \wedge$  encodeBound(m, c)  
  until not satisfiable( $\chi$ )  
  s := getCode(m)  
  return s
```

$\llbracket \chi \rrbracket$ instruction sequences
that correctly implement p in a

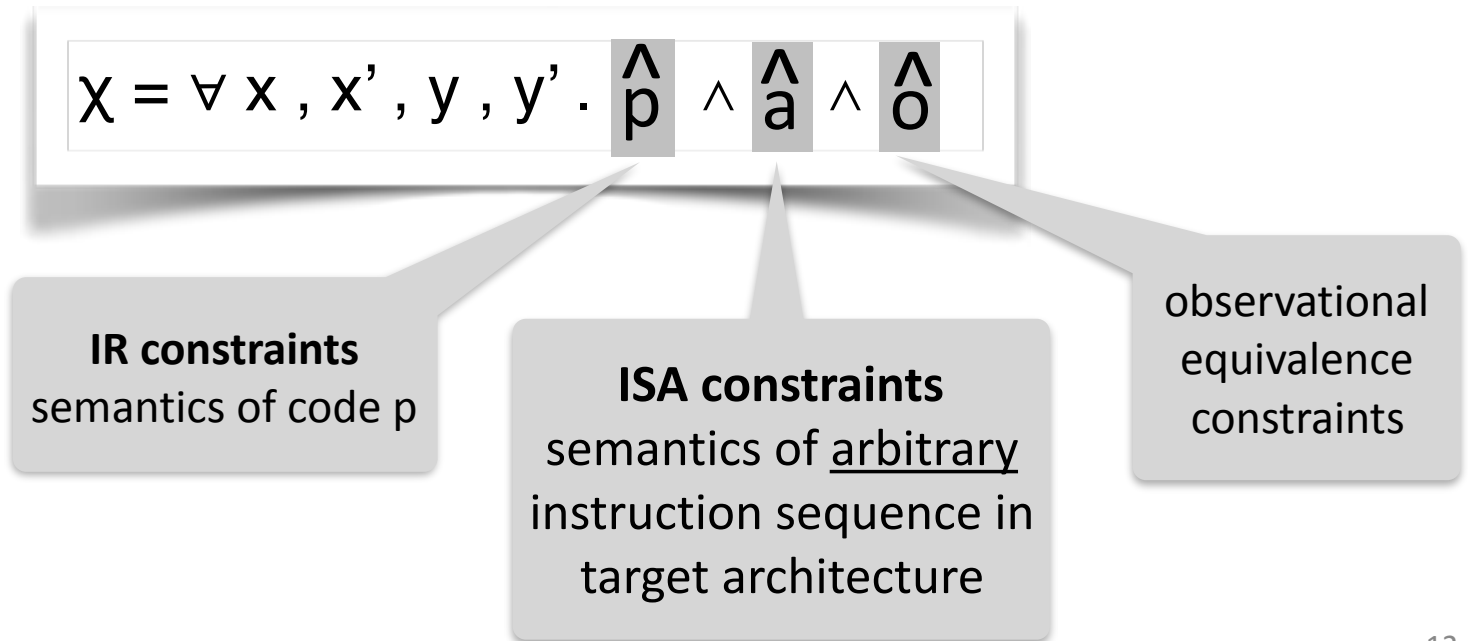
Unbounded Superoptimization

```
UnboundedSuperoptimizer(p, a, c)
   $\chi$  := encodeCorrectness(p, a)
  if not satisfiable( $\chi$ ) then return FAIL
  repeat
    m := getModel( $\chi$ )
     $\chi$  :=  $\chi$   $\wedge$  encodeBound(m, c)
  until not satisfiable( $\chi$ )
  s := getCode(m)
  return s
```

$\llbracket \chi \rrbracket$ instruction sequences
that correctly implement p in a

Encoding Correctness

$\llbracket \chi \rrbracket$ = instruction sequences s that correctly implement p in a



ISA Constraints

length of instruction sequence

instruction at location j is i

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

\wedge
a

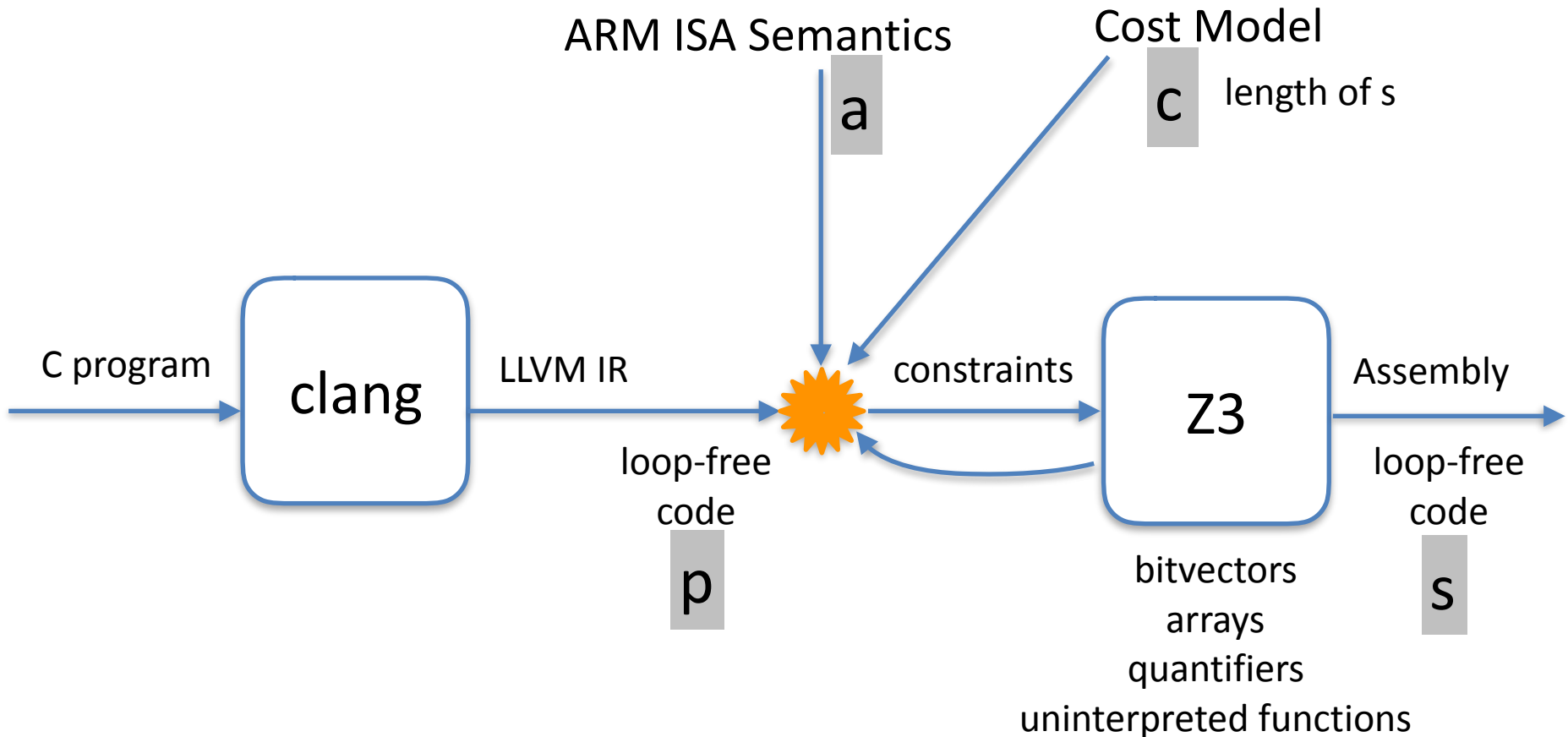
$I = \{$
 ADD r0, r1
 ADD r1, r2
 MUL r0, r2
 LDR r0, [r1] ... $\}$

semantics of instruction i

Shifting the search into the solver

- size of constraints does not depend on the candidate sequence of instructions
 - size of ϕ depends on n , size of χ 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 ϕ are counterexamples
 - models of χ are correct instruction sequences
- fine-grained control: compilation time vs quality of generated code
- loop-free code not just straight line code

Preliminary Prototype



LLVM IR Constraints

```
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
```

p

$L0 \leftrightarrow p1 = (px < 0) \wedge$
 $((p1 = \text{true}) \wedge L1) \vee ((p1 = \text{false}) \wedge L2)$
 $L2 \leftrightarrow p2 = (px > 0) \wedge$
 $((p2 = \text{true}) \wedge L3) \vee ((p2 = \text{false}) \wedge L4)$
 $L1 \leftrightarrow L5 \wedge (p3 = -1)$
 $L3 \leftrightarrow L5 \wedge (p3 = 1)$
 $L4 \leftrightarrow L5 \wedge (p3 = 0)$
 $L5 \leftrightarrow \text{true}$

\wedge
p

ARM ISA Constraints

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

\wedge
a

$\forall i. 0 \leq j < n.$

$\text{instr}(j)=\text{"SUB R0, R1"} \rightarrow \text{state}(j+1)[R0]=\text{state}(j)[R0]-\text{state}(j)[R1] \wedge \text{PRES}$

$\text{instr}(j)=\text{"MOV R0, R1"} \rightarrow \text{state}(j+1)[R0]=\text{state}(j)[R1] \wedge \text{PRES}$

$\text{instr}(j)=\text{"MOVGT R0, R1"} \rightarrow \text{ite}(\text{GT},$
 $\text{state}(j+1)[R0]=\text{state}(j)[R1] \wedge \text{PRES},$
 $\text{state}(j+1)=\text{state}(j))$

.....

Unbounded Superoptimization

```
UnboundedSuperoptimizer (p, a, c)  
   $\chi := \text{encodeCorrectness}(p, a)$   
  if not satisfiable( $\chi$ ) then return FAIL  
  repeat  
    m := getModel( $\chi$ )  
     $\chi := \chi \wedge \text{encodeBound}(m, c)$   
  until not satisfiable( $\chi$ )  
  s := getCode(m)  
  return s
```

$\llbracket \chi \rrbracket$ instruction sequences
that correctly implement p in a

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

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

Initial Feasibility Study

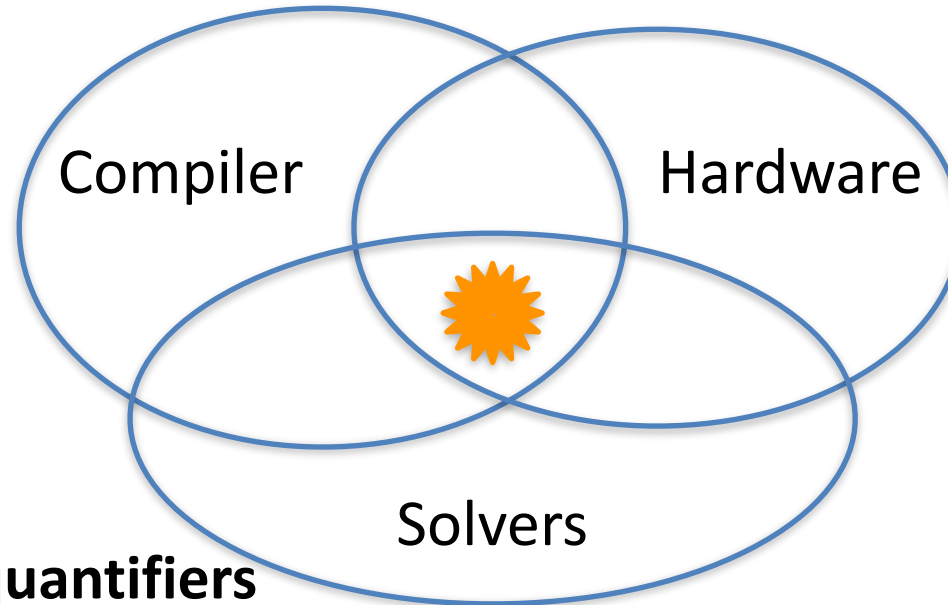
- Goal: can a solver handle complex constraints that arise from our encoding?
- Success on 17 out of 25 microbenchmarks but **slow**
 - under 30 min and up to 4 iterations per benchmark
 - unoptimized encoding
 - no solver heuristics
 - generated better code than gcc on 3 out of 17

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