supervectorizer

work in progress

Greta Yorsh
Queen Mary University of London

superoptimization

+

auto**vectorizer**

=

supervectorizer

unbounded
superoptimization

+

auto**vectorizer**

=

supervectorizer

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

How to generate efficient code for **new** cpu?

- Changes to code generation and libraries
- 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

- 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
- intrinsics

library routines: handcrafted assembly for memcpy, memset, memchr strcmp, strchr, strlen, integer dividision, directed rounding

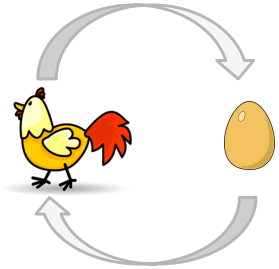
How to generate efficient code for **new** cpu?

- **Goal: best performance out of hardware**
- Practice: "just make it go faster"
- Compilers not changing fast enough to keep up with
  - variety of hardware designs
  - fast-paced design cycles
- 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

Aims

- Take advantage of SIMD capabilities of existing and emerging microprocessor designs **without modifying the compiler**

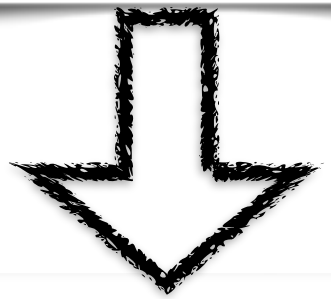
- Generate efficient code
- Guarantee correctness per compilation
- Support accurate performance models
- Aid hardware design process

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') | s' 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 := ∅
  for n:=0,1,2,… do
    for each s ∈ I^n do
      if check(s,Tests) then
        φ := encode(p,s)
        if not satisfiable(φ) then return s
        cex ← getModel(φ)
        Tests := Tests U getTests(p,cex)
```

⟦φ⟧ observed behaviour of p and s differ

Unbounded Superoptimization

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

⟦χ⟧ instruction sequences that correctly implement p in a

Unbounded Superoptimization

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

⟦χ⟧ instruction sequences that correctly implement p in a

Encoding Correctness

$[\![\chi]\!]$ = instruction sequences **s** that correctly implement **p** in **a**

$$\chi = \forall\, x, x', y, y' .\ \hat{p} \wedge \hat{a} \wedge \hat{o}$$

IR constraints
semantics of code p

ISA constraints
semantics of arbitrary instruction sequence in target architecture

observational equivalence constraints

ISA Constraints

length of instruction sequence

instruction at location **j** is **i**

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

$\hat{a}$

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

semantics of instruction i

ISA Constraints Example

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

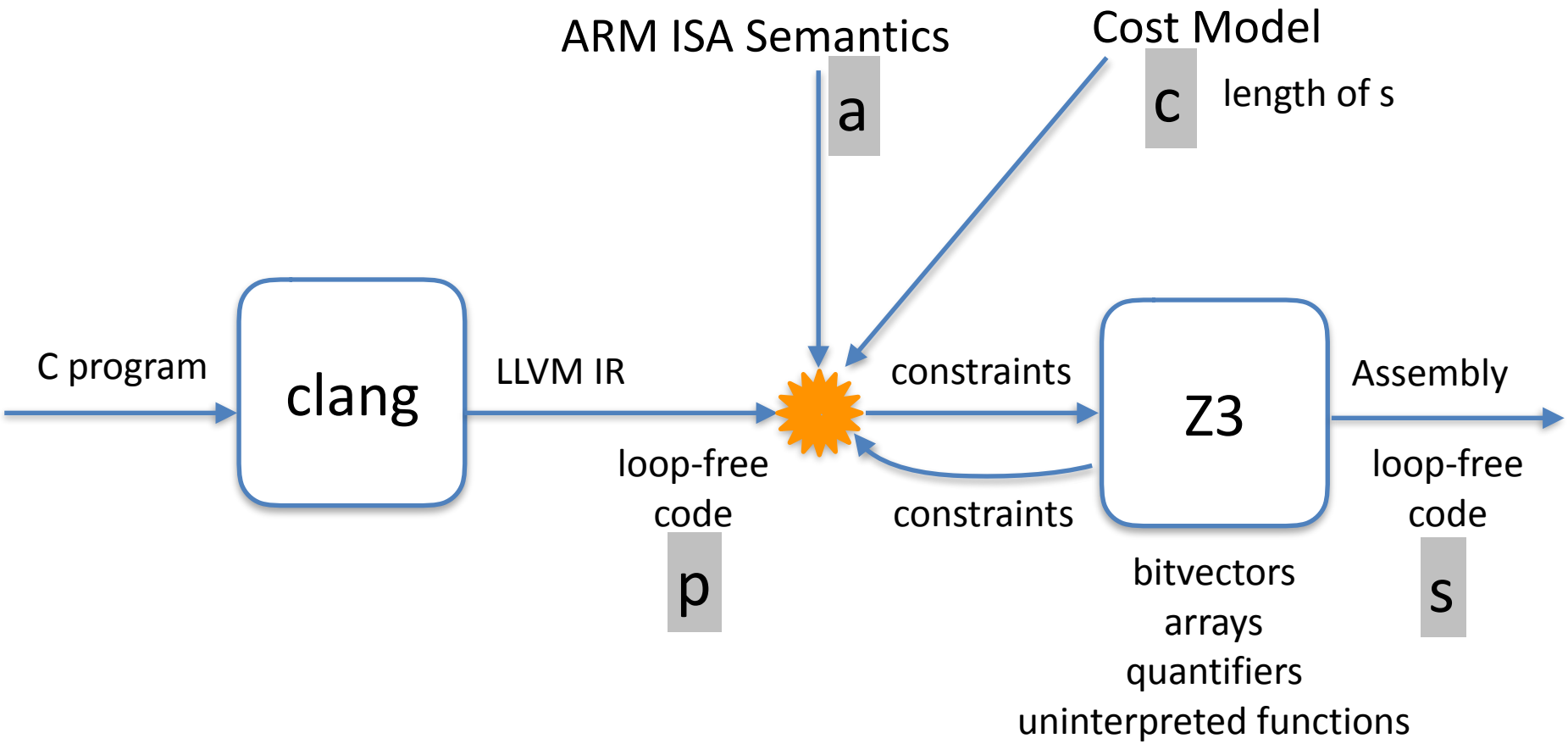
$\wedge_a$

$\forall j\ .\ 0 \le j < n.$
instr(j)="SUB R0, R1" → state(j+1)[R0]=state(j))[R0]−state(j)[R1]∧PRES
instr(j)="MOV R0, R1 → state(j+1)[R0]=state(j)[R1]∧PRES
instr(j)="MOVGT R0, R1" → ite(GT,
state(j+1))[R0]=state(j)[R1]∧PRES,
state(j+1)=state(j))
…..

Shifting the search into the solver

- opportunity to **reuse reasoning** within 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 ISA
- but larger and more complex formulas to solve

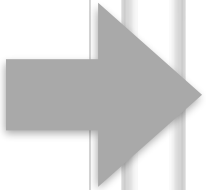
Preliminary Prototype



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



$L0 \leftrightarrow \rho1=(\rho x<0)\wedge$

$(((\rho1=true)\wedge L1)\vee((\rho1 =false)\wedge L2))$

$L2 \leftrightarrow \rho2=(\rho x>0)\wedge$

$(((\rho2 =true)\wedge L3)\vee((\rho2 =false)\wedge L4))$

$L1 \leftrightarrow L5\wedge(\rho3 =-1)$

$L3 \leftrightarrow L5\wedge(\rho3 =1)$

$L4 \leftrightarrow L5\wedge(\rho3 =0)$

$L5 \leftrightarrow true$

$\hat{p}$

Unbounded Superoptimization

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

⟦χ⟧ instruction sequences that correctly implement p in a

cost of instruction sequences is less than c(getCode(m))

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

Incremental

- fine-grained control over
  compilation time vs quality of generated code

- stop at any time with a correct possibly suboptimal solution that can be improved upon later

Challenges

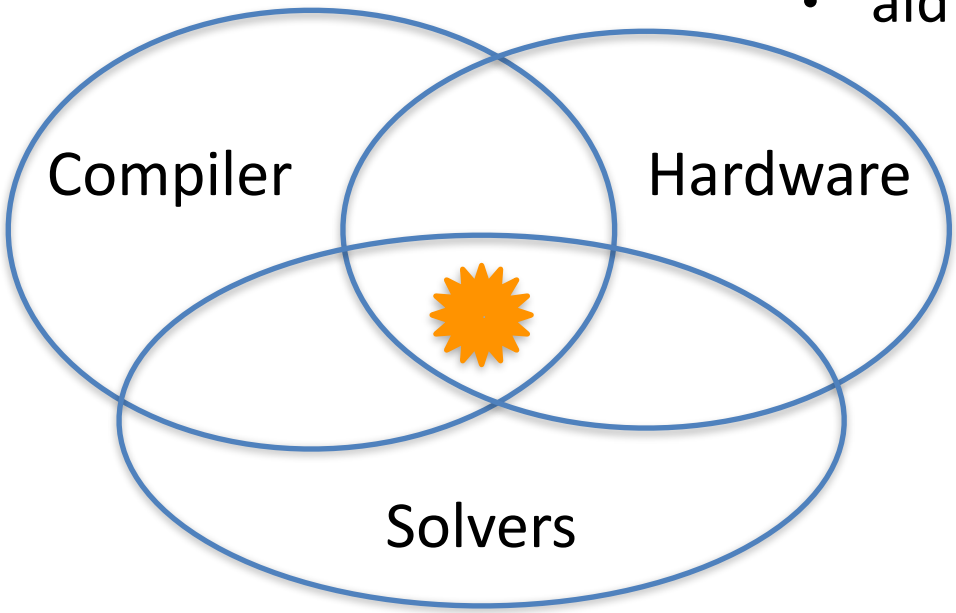
- Solver termination and completeness
- Compilation time
- Correctness of constraints
- Hardware specification availability
- Cost models availability
- Loops

Why SIMD targets?

- Unbounded superoptimization handles loop-free code, not just straight line code
- Automatically identify and exploit data level parallelism
- SIMD designs increase in complexity
- Aid vectorizer development in traditional compilers

- **memory operations**
- **loops**
- integrate in other toolchains

- **cost models**
- target other ISA
- target GPU
- aid cpu design



- **quantifiers**
- **MaxSMT**
- **different combination of theories**
- partial evaluation
- search heuristics