supervectorizer

work in progress

Greta Yorsh
Queen Mary University of London
superoptimization + autovectorizer = 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 division, 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 \( c \)
  • \( c(s) = \min\{c(s') \mid s' \text{ implements } p\} \)
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
```
Superoptimization

\textbf{Superoptimizer}(p,a) \quad // p is straightline code

Tests := \emptyset

\textbf{for} n:=0,1,2,... \textbf{do}

\textbf{for} each s \in I^n \textbf{do}

\textbf{if} check(s,Tests) \textbf{then}

\varphi := encode(p,s)

\textbf{if} not satisfiable(\varphi) \textbf{then} return s

\textbf{cex} \leftarrow \text{getModel}(\varphi)

Tests := Tests \cup \text{getTests}(p,\text{cex})

\lbrack \varphi \rbrack \text{ observed behaviour of } p \text{ and } s \text{ differ}
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 \(s\)
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

\[ [\chi] = \text{instruction sequences } s \text{ that correctly implement } p \text{ in } a \]

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

IR constraints
semantics of code \( p \)

ISA constraints
semantics of arbitrary instruction sequence in target architecture

observational equivalence constraints
ISA Constraints

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

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

Example

∀j. 0 ≤ j < n. \( \bigwedge \) instr(j)=i → \( \tau_i \)(state(j), state(j+1))

\( i \in I \)

∀j . 0 ≤ 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 $\phi$ depends on $n$
  - size of $\chi$ depends on ISA
- but larger and more complex formulas to solve
Preliminary Prototype

C program → clang → LLVM IR → constraints → Z3

ARM ISA Semantics

Cost Model

P	constraints

a

C

length of s

p

Z3

bitvectors
arrays
quantifiers
uninterpreted functions

loop-free code

Assembly

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 = (ρx < 0) ∧
    (((ρ1 = true) ∧ L1) ∨ ((ρ1 = false) ∧ L2))
L2 ← ρ2 = (ρx > 0) ∧
    (((ρ2 = true) ∧ L3) ∨ ((ρ2 = false) ∧ L4))
L1 ← L5 ∧ (ρ3 = −1)
L3 ← L5 ∧ (ρ3 = 1)
L4 ← L5 ∧ (ρ3 = 0)
L5 ← true
Unbounded Superoptimizer \((p, a, c)\)
\[
\chi := \text{encodeCorrectness}(p, a)
\]
\[
\text{if not satisfiable}(\chi) \quad \text{then return FAIL}
\]
\[
\text{repeat}
\]
\[
m := \text{getModel}(\chi)
\]
\[
\chi := \chi \land \text{encodeBound}(m, c)
\]
\[
\text{until not satisfiable}(\chi)
\]
\[
s := \text{getCode}(m)
\]
\[
\text{return } s
\]
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