> Evaluating Souper: A Synthesizing Superoptimizer

e.b.demir
@student.tudelft.nl

d.g.sprokholt,
s.s.chakraborty
@tudelft.nl

1 Superoptimization

- The optimizations in the “middle-end” of the compiler are numerous, time-consuming to develop, hard to get right.
- Instead of proving upper or lower bounds for abstract algorithms, a superoptimizer finds the shortest program in the program space defined by a instruction set.

2 Souper

- A synthesizing superoptimizer that automatically derives novel middle-end optimizations.
- With an intermediate representation (IR) that resembles a purely functional, control-flow-free subset of LLVM IR

3

Are the original results reproducible?

What program classes does Souper work best in?

1. Output Souper IR
2. Binary Size
3. Compilation Speed

4

1-) LLVM-IR is generated via a front-end

2-) For each integer-type returning LLVM module Souper extracts a root for a LHS

3-) Souper recursively follows dataflow edges adding path conditions and blockpcs as necessary

4-) RHS are enumerated using CEGIS

5-) The equivalence of the RHS & LHS are checked using an SMT solver

6-) If this query is satisfiable and the RHS is smaller than the LHS an optimization is found.

```
int
f(bool cond, int z) {
  int x, y;
    if (cond) {
      x = 3 * z;
      y = z;
    } else {
      x = 2 * z;
      y = 2 * z;
    }
  return x + y;
}
```

```
define i32 @f(i1 %0, i32 %1) {
br i1 %0, label %3, label %5
label %3:
%4 = mul nsw i32 %1, 3
br label %8
label %5:
%6 = shl nsw i32 %1, 1
%7 = shl nsw i32 %1, 1
br label %8
label %8:
%.07 = phi i32 [ %4, %3 ], [ %6, %5 ]
%.0 = phi i32 [ %1, %3 ], [ %7, %5 ]
%9 = add nsw i32 %.07, %.0
ret i32 %9
}
```

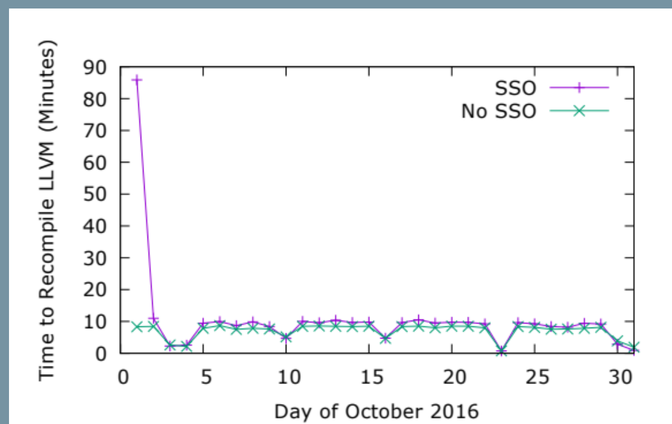
```
%0 = block 2
%1:i32 = var
%2:i32 = shlnsw %1, 1:i32
%3:i32 = phi %0, %1, %2
%4:i32 = mulnsw 3:i32, %1
%5:i32 = phi %0, %4, %2
%6:i32 = addnsw %3, %5
infer %6
⇒
%7:i32 = shl %1, 2:i32
result %7
```

7

> Original Results

Compiling clang

	Compile time(minutes)	Binary size (MiB):
Souper-optimized:	88 / 14	64.3
Regular build:	13	67.2



> Attempts at reproducing results

- Using the drop-in compiler to compile the latest version of clang + LLVM w/ & w/o an external cache
- Using the drop-in compile to compile clang 3.9.0 w/ & w/o an external cache
- linking Souper against LLVM 3.9.0
- Using flags from *slumps*.
- Extracting left-hand sides without inferring optimizations and using the inferring from the cache separately.

8

> Program Classes

> High cyclomatic complexity

- Exploiting correlated Φ nodes

> Programs Souper cannot fully extract from

- Memory manipulation
- I/O intensive
- Floating point operations

> Programs with UB

- LLVM vs. Souper’s UB representation
- How well can Souper support UB?

```
unsigned foo(unsigned a) {
  switch (a % 4) {
    case 0:
      a += 3;
      break;
    case 1:
      a += 2;
      break;
    case 2:
      a += 1;
      break;
  }
  return a & 3;
}
```

```
%0 = block 4
%1:i32 = var
%2:i32 = urem %1, 4:i32
%3:i1 = ne 0:i32, %2
%4:i1 = ne 1:i32, %2
%5:i1 = ne 2:i32, %2
blockpc %0 0 %3 1:i1
blockpc %0 0 %4 1:i1
blockpc %0 0 %5 1:i1
blockpc %0 1 %2 2:i32
blockpc %0 2 %2 1:i32
blockpc %0 3 %2 0:i32
%6:i32 = add 1:i32, %1
%7:i32 = add 2:i32, %1
%8:i32 = add 3:i32, %1
%9:i32 = phi %0, %1, %6, %7, %8
%10:i32 = and 3:i32, %9
infer %10
⇒
result 3:i32
```

5

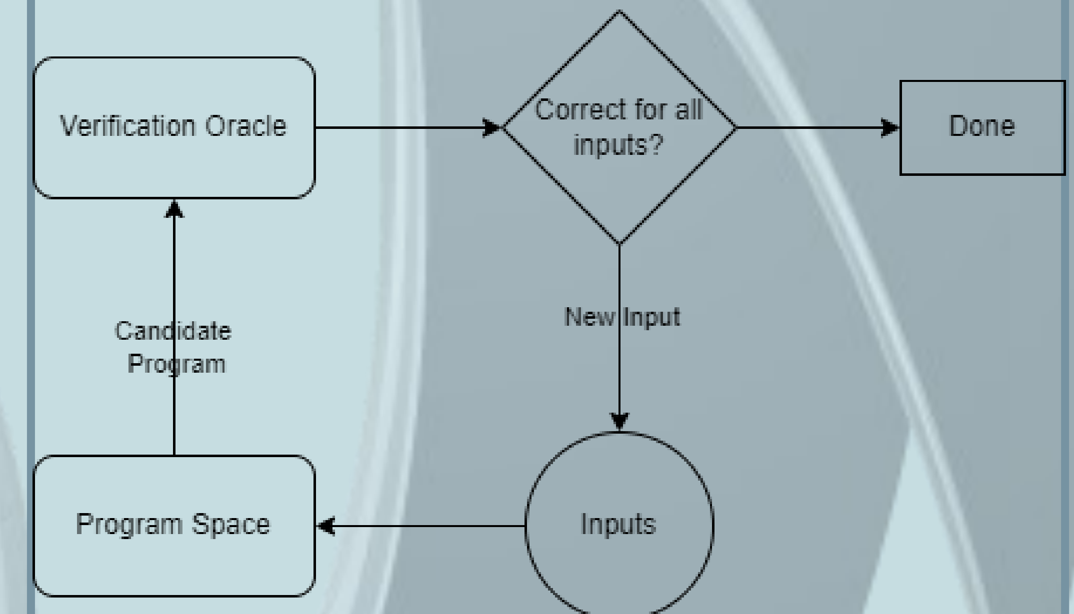
> CEGIS

1-) Program space is searched for a candidate P that satisfy

$$\exists \mathbf{P} \cdot \forall \mathbf{x} \in INPUTS \cdot \sigma(\mathbf{P}, \mathbf{x})$$

2-) Query the verification oracle with P

3-) If it works for all inputs return P else add



- Even though CEGIS does not attempt to naively enumerate the entire program space and performs well in most cases, Souper makes use of various strategies to shrink the search space.

6

> Pruning the search space

> w/o Dataflow Analysis

> An outer CEGIS loop

> Not synthesizing constants directly

> Cost model

> Ad hoc pruning Strategies

> w/ Dataflow Analysis

Souper tries to find conflicting dataflow facts between a specification and a potentially symbolic candidate to prune branches of the search tree

Consider the specification:

$f(x) = (x * x * x) \mid 1$

And the optimization candidate:

$g(x) = H(x) \ll C$

Based on the specification, last bit is always set.

Whereas the candidate has the last bit always cleared.

-> Conflict, this subtree of candidates can be pruned.

TU Delft