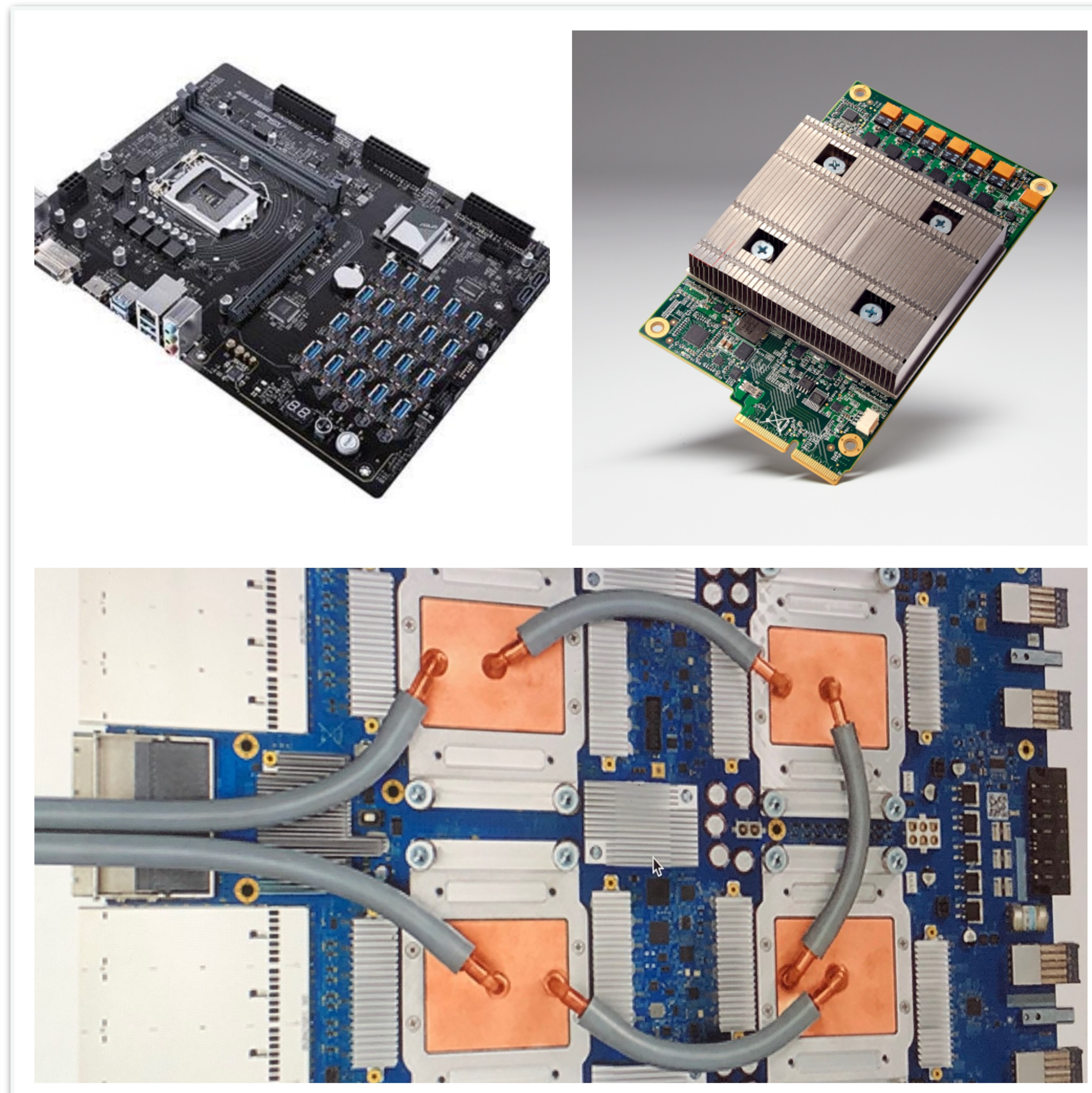
Improving Compiler Construction Using Formal Methods

Jubi Taneja
PhD Student

Advisor: John Regehr

University of Utah

The Trend



Improving Compiler Construction Using Formal Methods

Compiler

- **Correct**
- **Generate high quality code**
- **Fast**

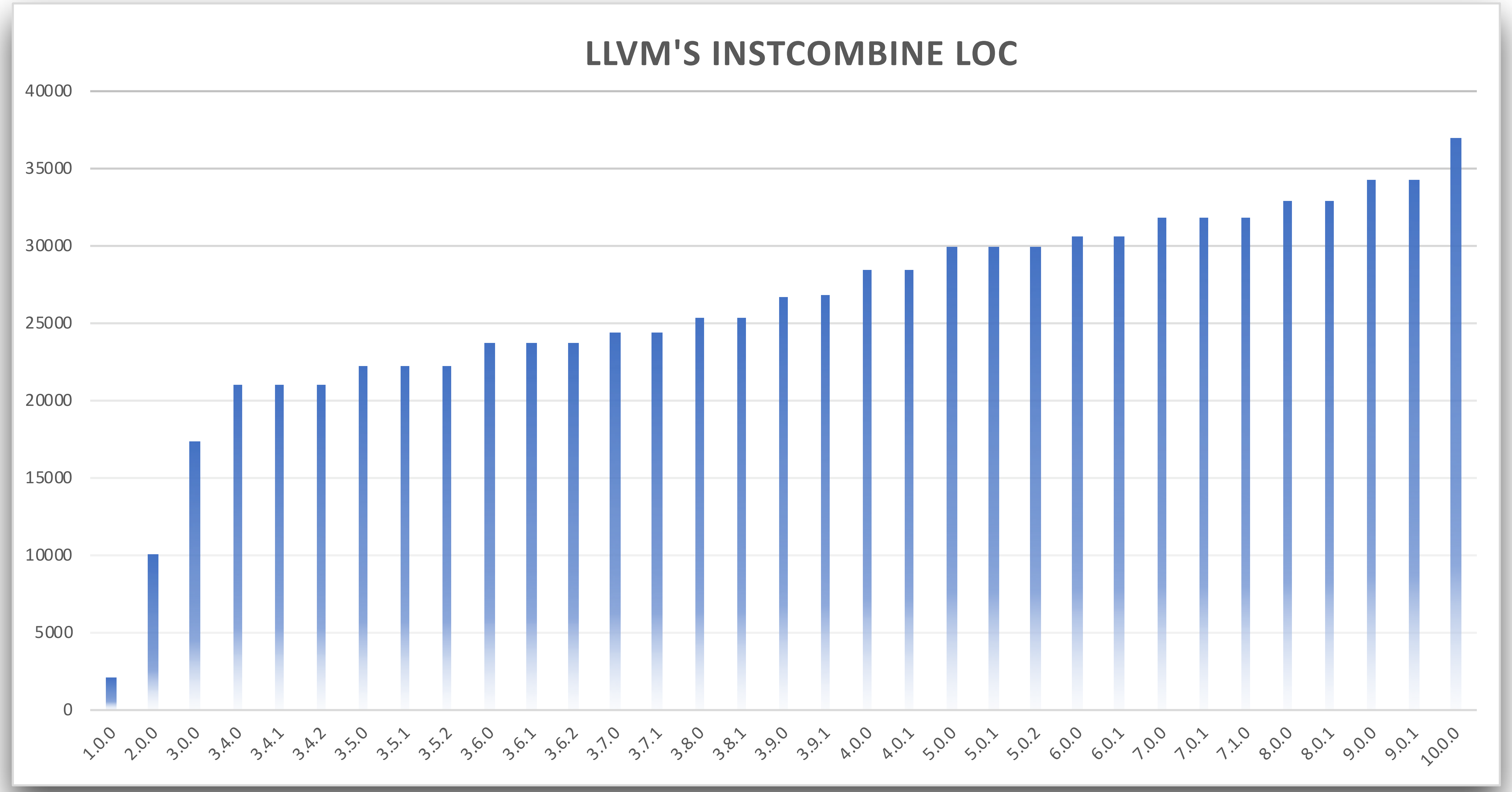
Why Compilers are Wrong?

```
int foo () {
  int n = 2;
  string s;
  for (int i = 0; i < n % 3; i++ {
    s += "ab";
  }
  printf("\n%s\n", s.c_str());
}
```

```
$ clang-3.7 input.cc -o exe
$ ./exe

abab
```

```
$ clang-3.7 -O2 input.cc -o exe
$ ./exe

ab
```

Manual Implementation



Problems

- **Misses important optimizations**
- **Time consuming to develop**
- **Prone to bugs**
  - **Bug in transformation, or**

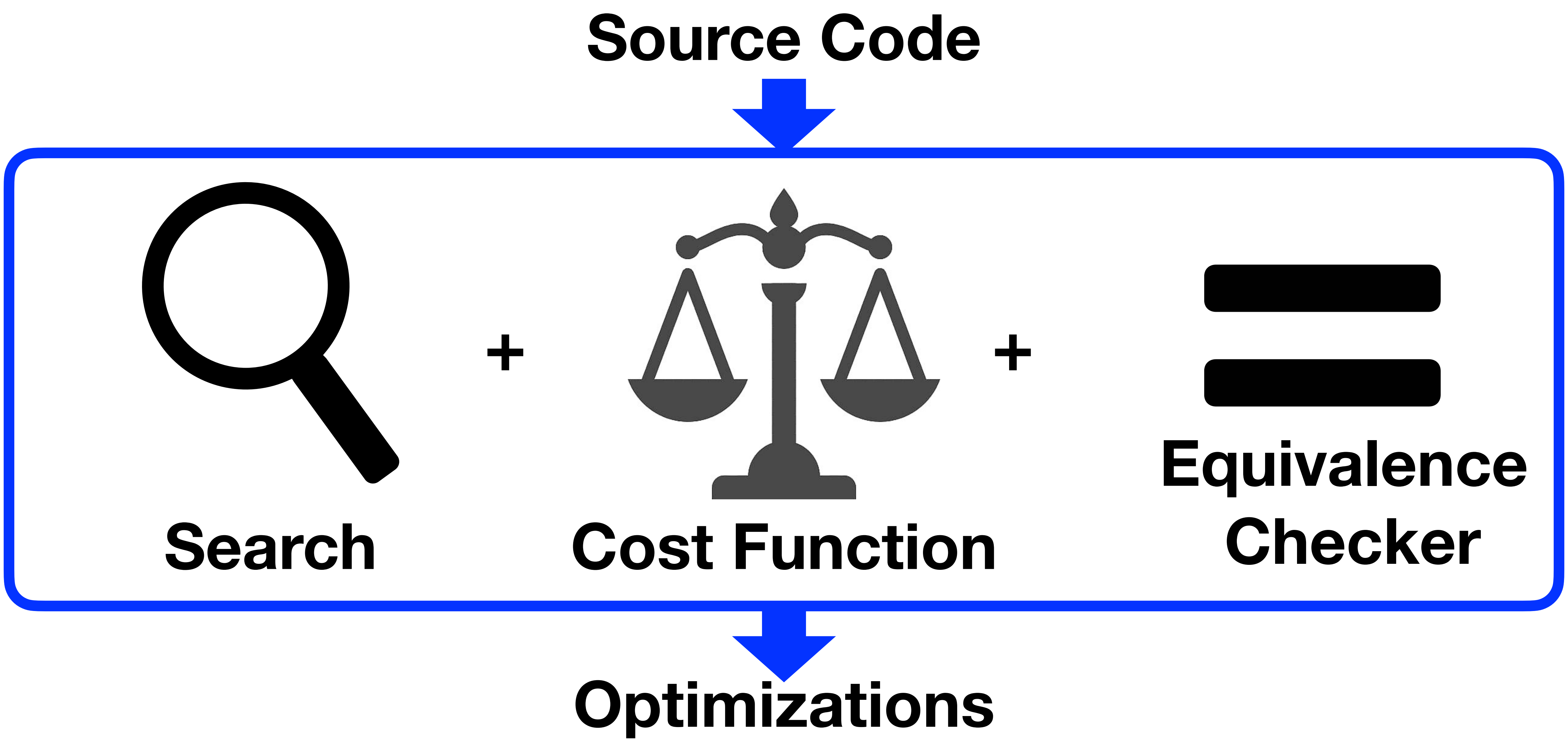
    Nuno Lopes et al.'s Alive/Alive2 verifies LLVM's transformations
  - **Bug in driver of the transformation, i.e. static analyses**

Goals

- **Automatically discover missing optimizations**
- **Automatically implement the optimizer**
- **Test Static Analyses using formal-methods-based techniques**

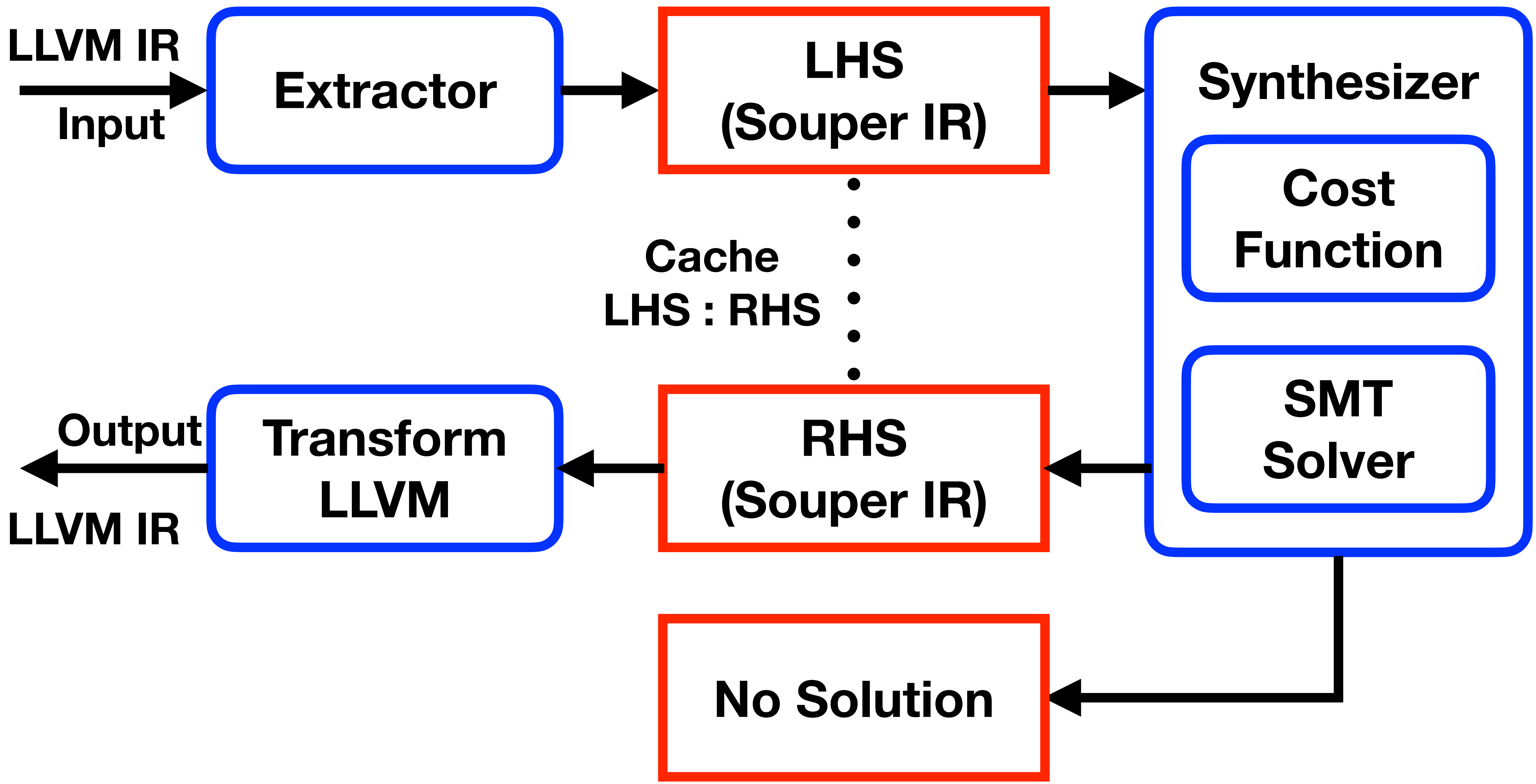
Automatically discover missing optimizations

Souper


Source Code
Search
+
Cost Function
+
Equivalence
Checker
Optimizations

```
define i32 @foo(i32 %x1) {
  %0 = and 0x55555555, %x1
  %1 = lshr i32 %x1, 1
  %2 = and 0x55555555, %1
  %3 = add i32 %0, %2
  %4 = and 0x33333333, %3
  %5 = lshr i32 %3, 2
  %6 = and 0x33333333, %5
  %7 = add i32 %4, %6
  %8 = and 0x0F0F0F0F, %7
  %9 = lshr i32 %7, 4
  %10 = and 0x0F0F0F0F, %9
  %11 = add i32 %8, %10
  %12 = and 0x00FF00FF, %11
  %13 = lshr i32 %11, 8
  %14 = and 0x00FF00FF, %13
  %15 = add i32 %12, %14
  %16 = and 0x0000FFFF, %15
  %17 = lshr i32 %15, 16
  %18 = and 0x0000FFFF, %17
  %19 = add i32 %16, %18
  ret i32 %19
}
```

```
define i32 @foo(i32 %x1) {
foo0:
  %0 = call i32
        @llvm.ctpop.i32
        (i32 %x1)
  ret i32 %0
}
```


LLVM IR
Input
Extractor
LHS
(Souper IR)
Synthesizer
Cost
Function
Cache
LHS : RHS
SMT
Solver
Output
LLVM IR
Transform
LLVM
RHS
(Souper IR)
No Solution

Impact of Souper

- **Souper makes clang-5.0 binary 1.6 MB smaller**
- **Production compiles like, MSVC, Mono, Binaryen have used Souper to find missing optimizations**

Testing Static Analyses for Precision and Soundness

Static Analysis

- **Sound**
- **Precise**
- **Fast**

Motivation: Precision

```
define i1 @foo(i8 %x) {
entry:
  %0 = icmp eq i8 %x, 42
  %1 = icmp eq i8 %x, 43
  %2 = or i1 %0, %1
  %3 = select i1 %2, i8 1, i8 %x
  %4 = icmp eq i8 %3, 42
  ret i1 %4
}
```

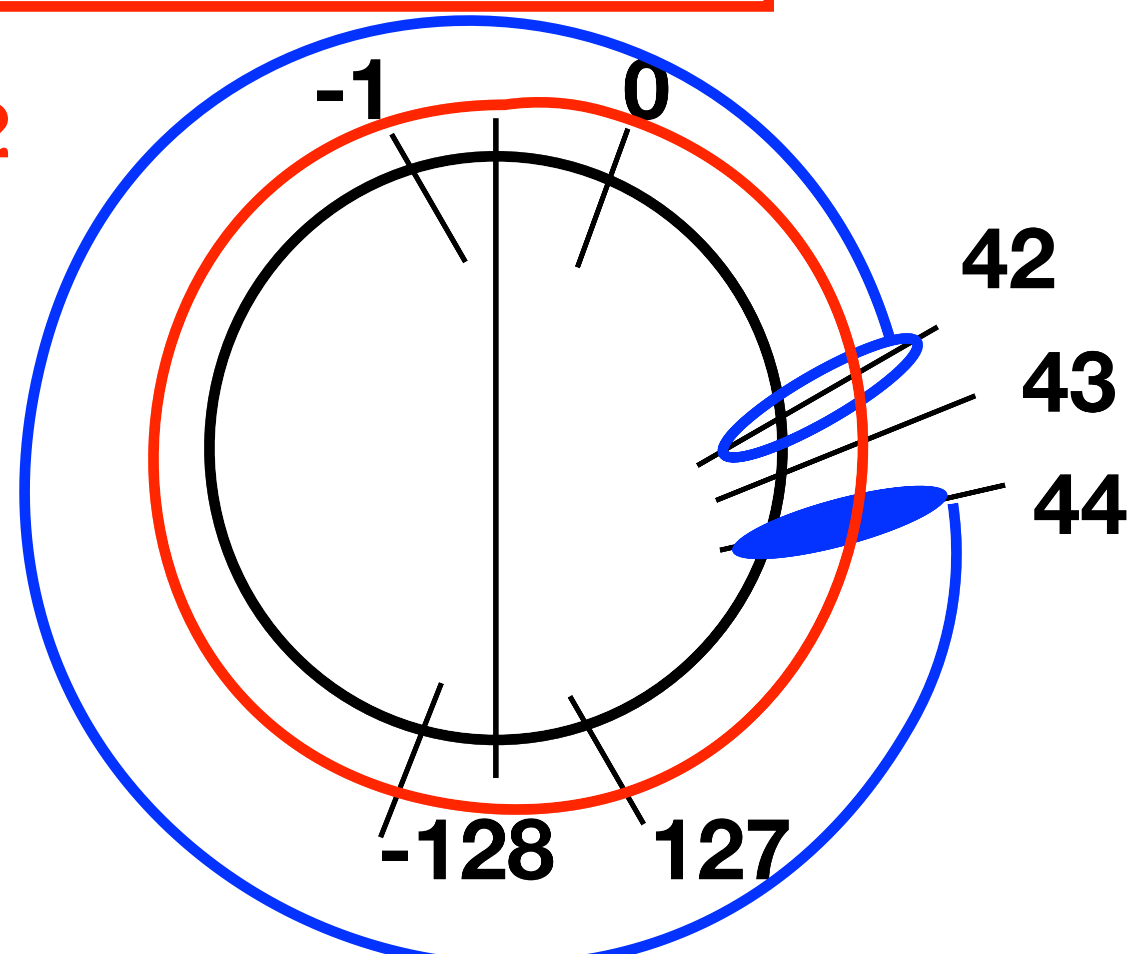
```
define i1 @foo(i8 %x) {
entry:
  %0 = icmp eq i8 %x, 42
  %1 = icmp eq i8 %x, 43
  %2 = or i1 %0, %1
```

```
  %3 = select i1 %2, i8 1, i8 %x
```

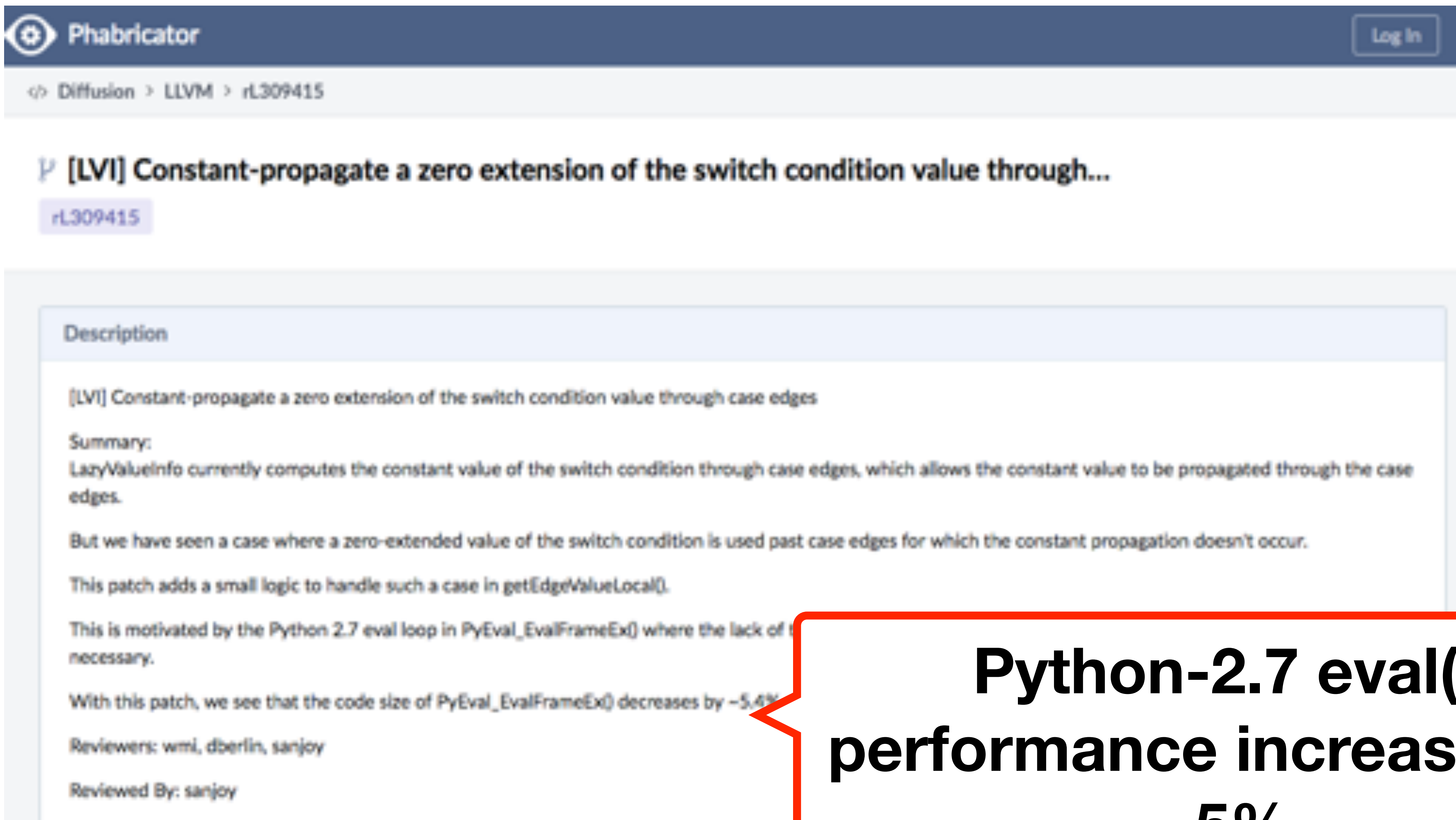
```
  %4 = icmp eq i8 %3, 42
  ret i1 %4
}
```

LLVM Result : No info

Best Result: [44, 42)



Imprecision in LLVM's Integer Range Analysis


Phabricator
Log In
Diffusion › LLVM › rL309415

[LVI] Constant-propagate a zero extension of the switch condition value through...

rL309415

Description

[LVI] Constant-propagate a zero extension of the switch condition value through case edges

Summary:
LazyValueInfo currently computes the constant value of the switch condition through case edges, which allows the constant value to be propagated through the case edges.

But we have seen a case where a zero-extended value of the switch condition is used past case edges for which the constant propagation doesn't occur.

This patch adds a small logic to handle such a case in getEdgeValueLocal().

This is motivated by the Python 2.7 eval loop in PyEval_EvalFrameEx() where the lack of t
necessary.

With this patch, we see that the code size of PyEval_EvalFrameEx() decreases by ~5.4%

Reviewers: wmi, dberlin, sanjoy

Reviewed By: sanjoy

Python-2.7 eval() performance increased by ~5%

Miscompilation Bug in LLVM

```
int foo () {
  int n = 2;
  string s;
  for (int i = 0; i < n % 3; i++ {
    s += "ab";
  }
  printf(“\n%s\n”, s.c_str());
}
```

Motivation: Soundness

```
define i32 @foo(i32 %x) {
entry:
  %0 = srem i32 %x, 3
  ret i32 %0
}
```

Q: Number of sign bits for %0?

LLVM-3.7 Result : 31

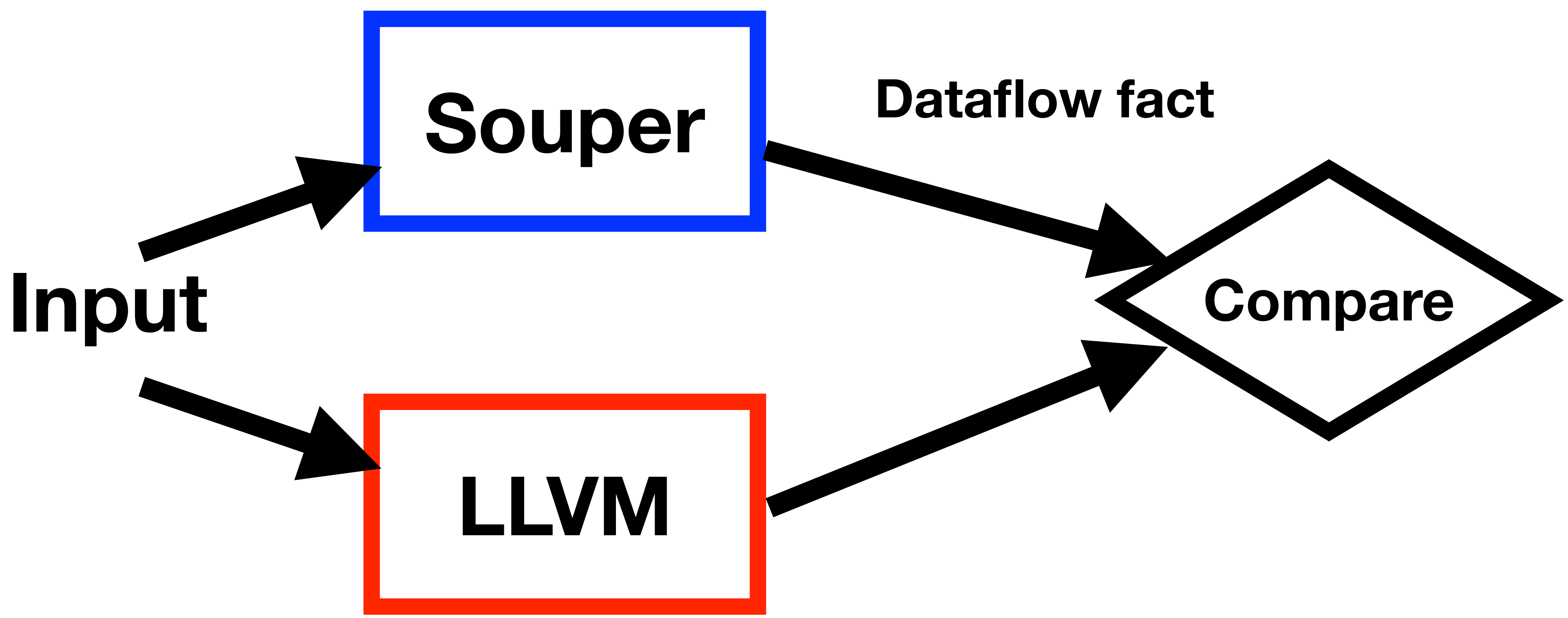
Best Result : 30

Problems

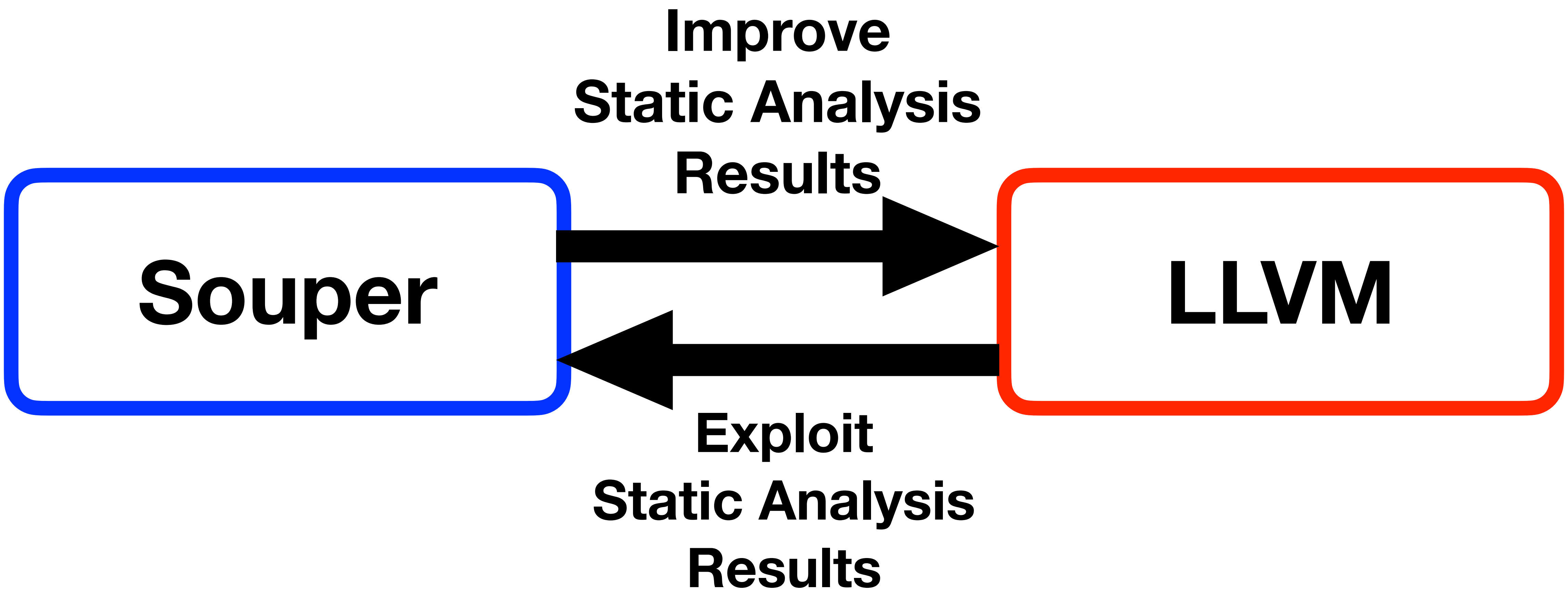
- Imprecision
- Unsoundness
- Developers are manually improving the static analysis without any help of formal methods

Goal

Automatic testing of LLVM's static analyses using formal methods



Static Analysis and Souper



Using Dataflow Facts

```
define i32 @foo() {
  ...
  // isKnownToBeAPowerOfTwo(%x) == true
  %2 = call i32 @llvm.ctpop.i32(i32 %x)
  ret i32 %2
}
```

```
ret i32 1
```

- **Known Bits**
- **Integer Range**
- **Number of Sign Bits**
- **Non-zero**
- **Non-negative**
- **Negative**
- **Power of Two**

Forward

- **Demanded Bits**

Backward

Known Bits

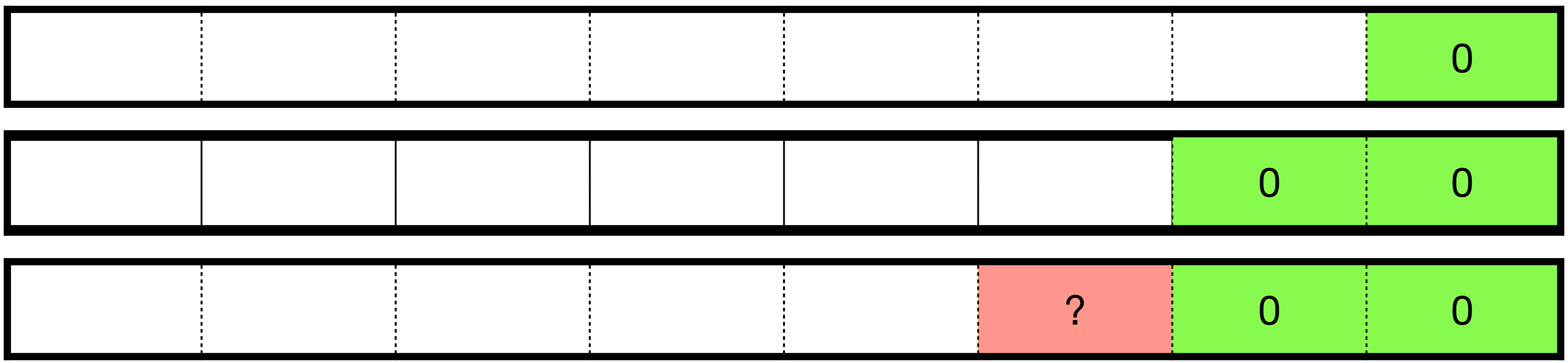
```
%0 = shl i8 %x, 4
```

?	?	?	?	0	0	0	0

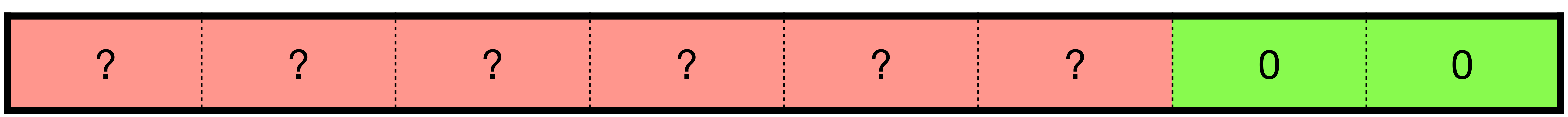
```
%0 = shl i8 4, %x
```

?	?	?	?	?	?	?	?

Solver-based Algorithm to Compute Known Bits for 4 << x



more failing guesses …



```
%0 = shl i8 4, %x
```

LLVM:

?	?	?	?	?	?	?	?

Souper:

?	?	?	?	?	?	0	0

- **Our algorithm uses at most $\boxed{2 * \text{BitWidth}}$ solver calls**
    - **Brute force algorithm uses 3^{BitWidth} calls, which is infeasible**
- **Computes maximally precise known bits as the lattice is separable at bit level**
    - **Details in our CGO 2020 paper**

Integer Range

```
define i4 @foo(i4 %y) {
entry:
   %0 = mul i4 %y, %y
   ret i4 %0
}
```

Range of %0?

LLVM: No information

Range [X, X+M)

- **Algorithm for M - Binary Search to find the smallest M**
- **Algorithm for X - Solver-based Constant Synthesis**

Find X such that (y * y) ∈ [X, X+M)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

X = 0, Range = [0, 0+12)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

X = 0, Range = [0, 0+10)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

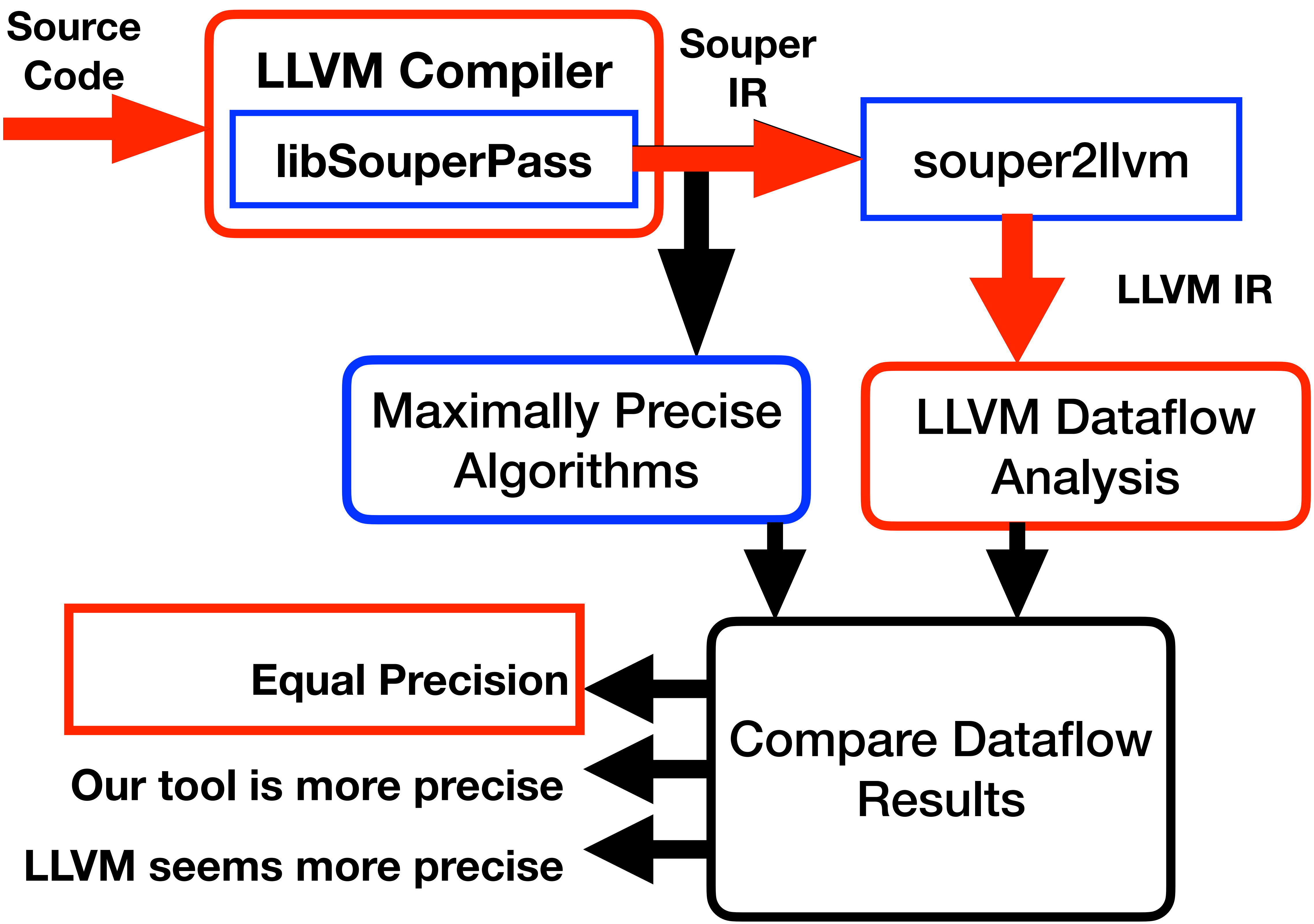
```
%0 = mul i4 %y, %y
```

LLVM: No Information

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Souper: [0, 10)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15


Source
Code
LLVM Compiler
libSouperPass
Souper
IR
souper2llvm
LLVM IR
Maximally Precise
Algorithms
LLVM Dataflow
Analysis
Equal Precision
Our tool is more precise
LLVM seems more precise
Compare Dataflow
Results

Impact on LLVM

- **Five concrete precision improvements made in LLVM's static analyses (< version 8) already discussed in the paper.**
- **All integer range imprecisions discussed in the paper have already been fixed in LLVM 10.**
- **More known bits imprecisions have also been fixed in code generation phase.**

What happens if LLVM calls our analyses instead of its own?

Too Slow!

Is LLVM Unsound?

- **No new soundness bugs were found in LLVM+Clang-8.0**
- **Introduced three old soundness bugs from LLVM-2.9+ and our tool detected all of them**

Takeaway

- **Encourage compiler developers to use formal methods based techniques to test static analyses**

Automatic Generation of a Peephole Optimizer

Motivation

$$\boxed{(a \mathbin{\&} {\sim}b) \mid ({\sim}a \mathbin{\&} b) \Rightarrow (a \wedge b)}$$

$$(a \mathbin{\&} {\sim}b) \wedge ({\sim}a \mathbin{\&} b) \Rightarrow (a \wedge b)$$

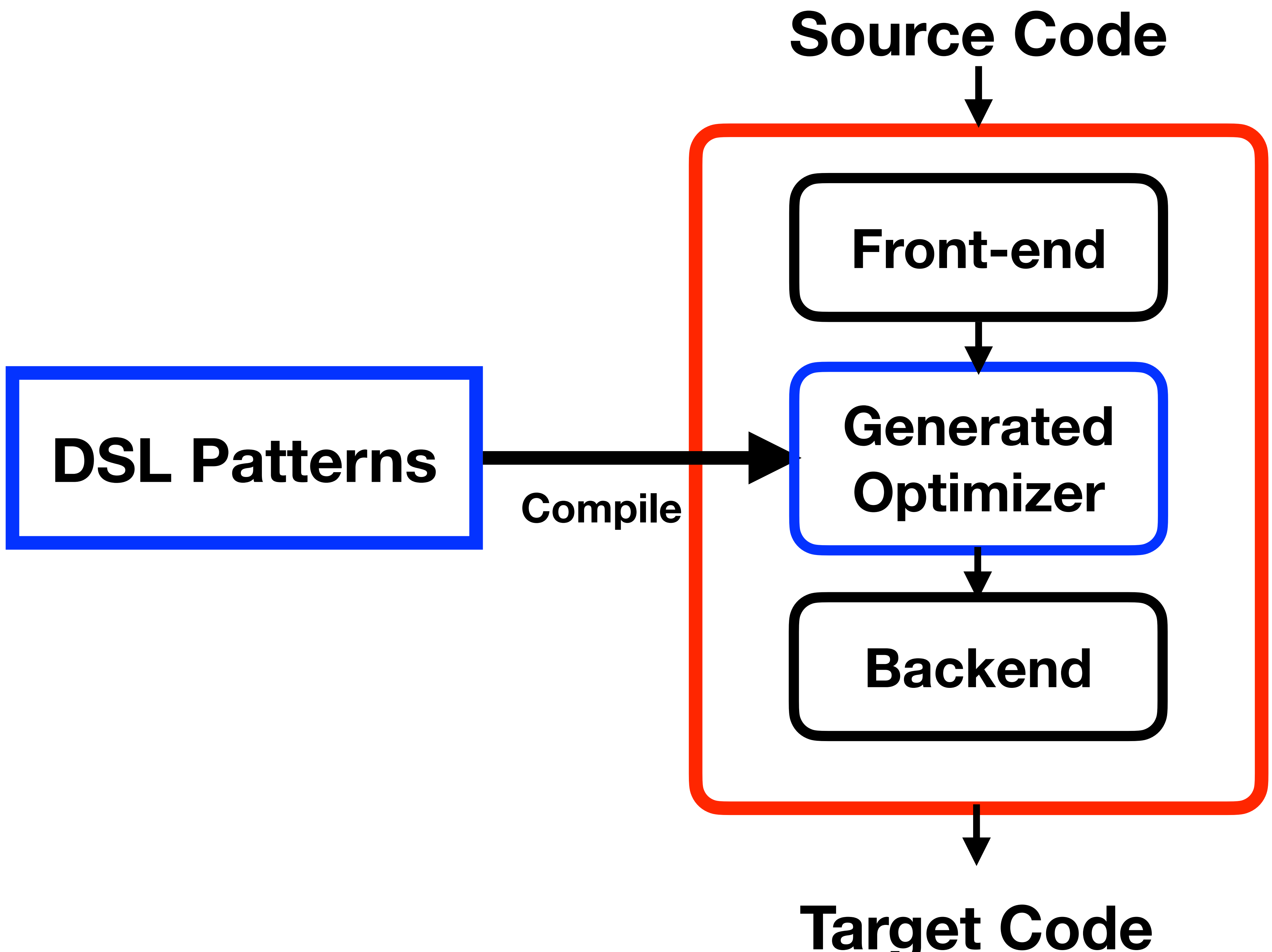
$$(a \mathbin{\&} {\sim}b) + ({\sim}a \mathbin{\&} b) \Rightarrow (a \wedge b)$$

```
stati
                                                                 lder)
{
  ass
  Val
  Value *Op1 = I.getOperand(1);
  Value *A, *B;

  // (A & ~B) | (~A & B) --> A ^ B
  if (match(Op0, m_c_And(m_Value(A), m_Not(m_Value(B)))) &&
      match(Op1, m_c_And(m_Not(m_Specific(A)), m_Specific(B))))
    return BinaryOperator::CreateXor(A, B);

  return nullptr;
}
```

Goals

- **Finding the patterns and expressing them in a simpler way**
- **Implementing the patterns in a fast way**

DSL-based Approach



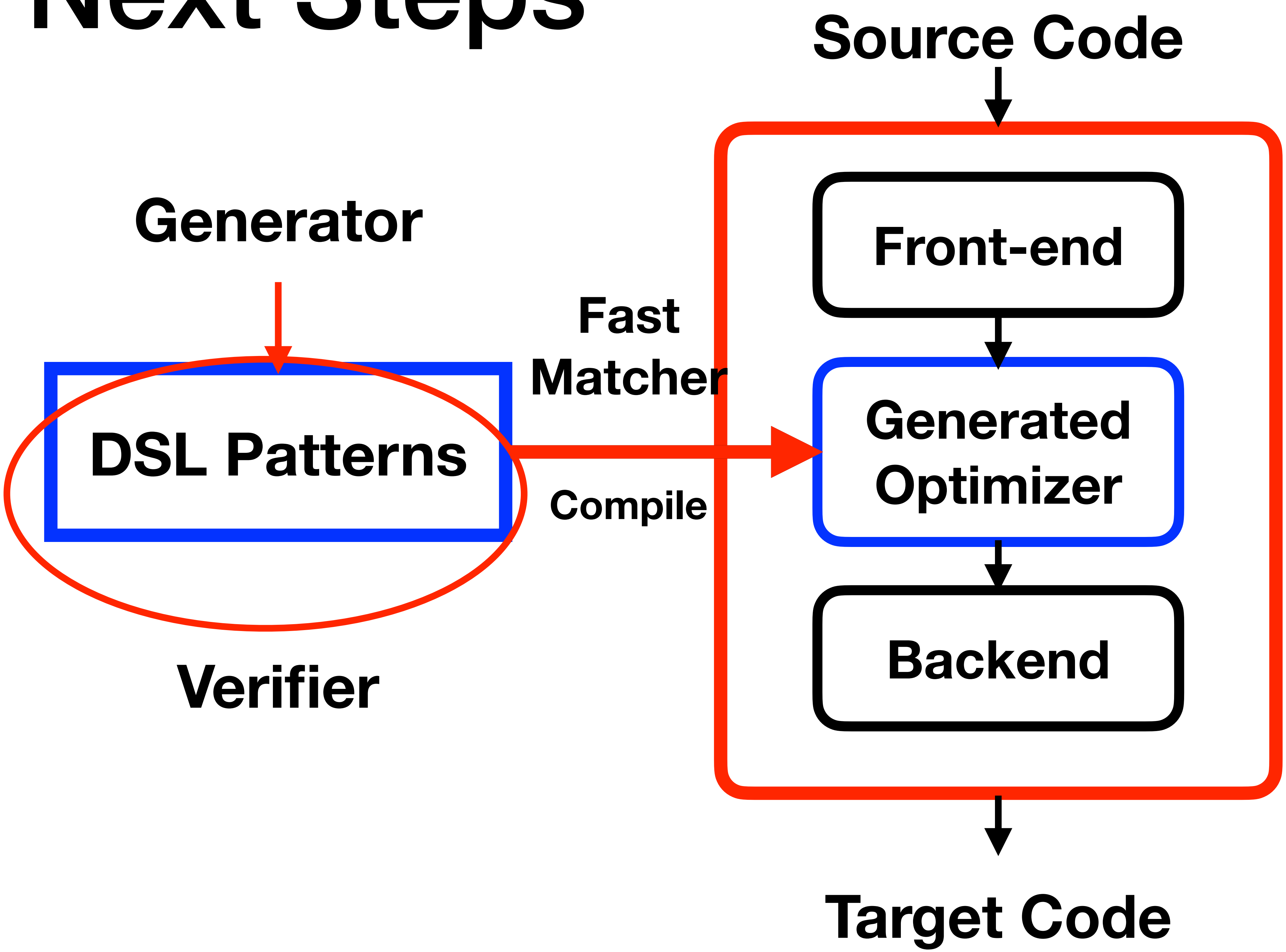
DSL Patterns

```
GCC: /* Simplify (A & ~B) |^+ (~A & B) -> A ^ B.  */

(for op (bit_ior bit_xor plus)
 (simplify
  (op (bit_and:c @0 (bit_not @1)) (bit_and:c (bit_not @0)
@1))
   (bit_xor @0 @1)))
```

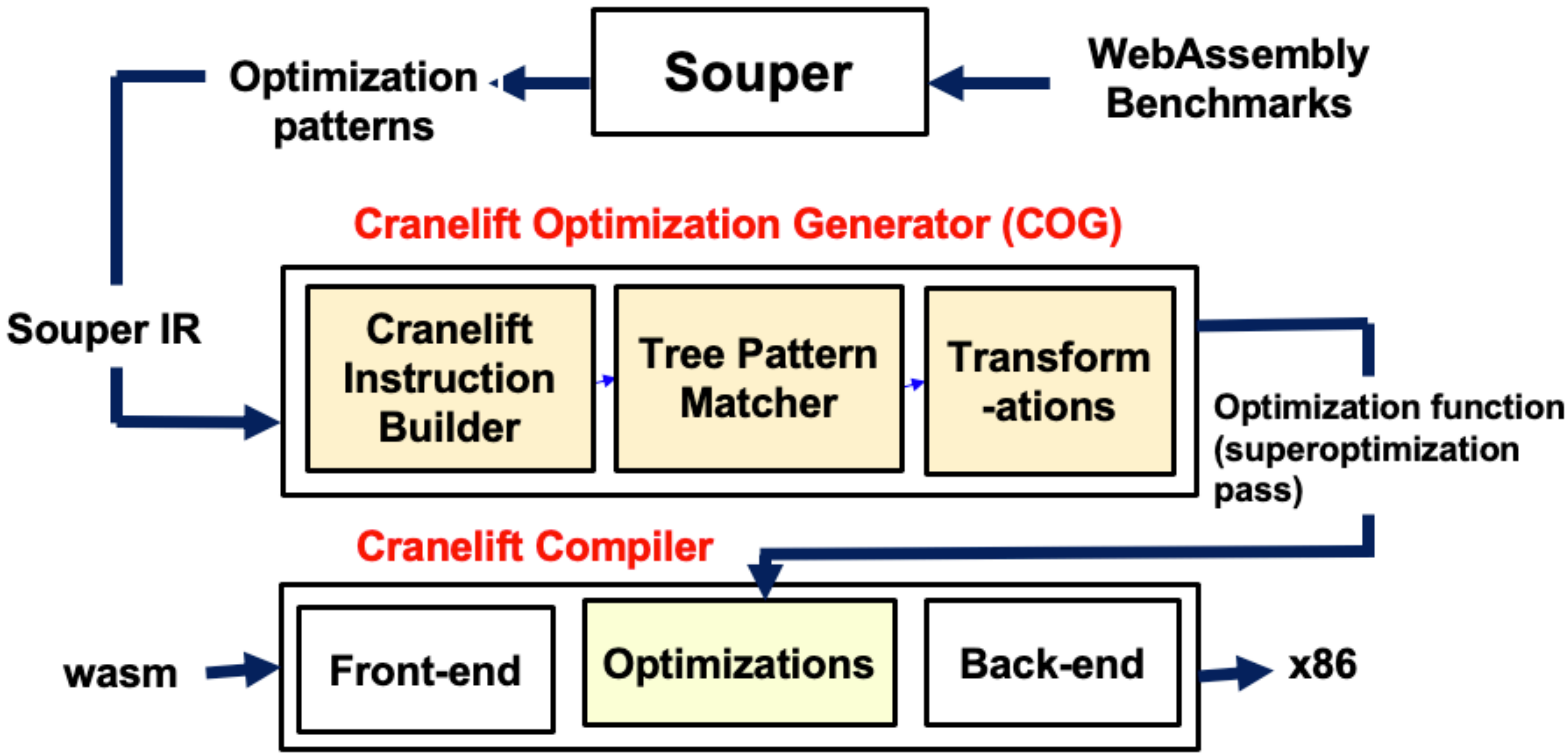
```
Go

(Xor64 (And64 a (Not b)) (And64 (Not a) b)) -> (Xor64 a b)
(Or64 (And64 a (Not b)) (And64 (Not a) b)) -> (Xor64 a b)
```

```
Cranelift

(=> (bxor (band_not $a $b) (band_not $b $a)) (bxor $a $b))
(=> (bor (band_not $a $b) (band_not $b $a))  (bxor $a $b))
```

Next Steps


Source Code
Generator
Front-end
Fast
Matcher
DSL Patterns
Generated
Optimizer
Compile
Backend
Verifier
Target Code

Superoptimizer-based Approach



DSL with pre-condition

```
(=> (when (imul $x $C)
          (is-power-of-two $C))
    (ishl $x $(log2 $C)))
```

Peepmatic: A peephole optimizer compiler for Cranelift
Author: Nick Fitzgerald
Github: https://github.com/bytecodealliance/wasmtime/tree/master/cranelift/peepmatic

Future Work

Derive weakest preconditions and automatic generalization of optimizations

- For this, use dataflow facts and synthesis algorithms

Future Work

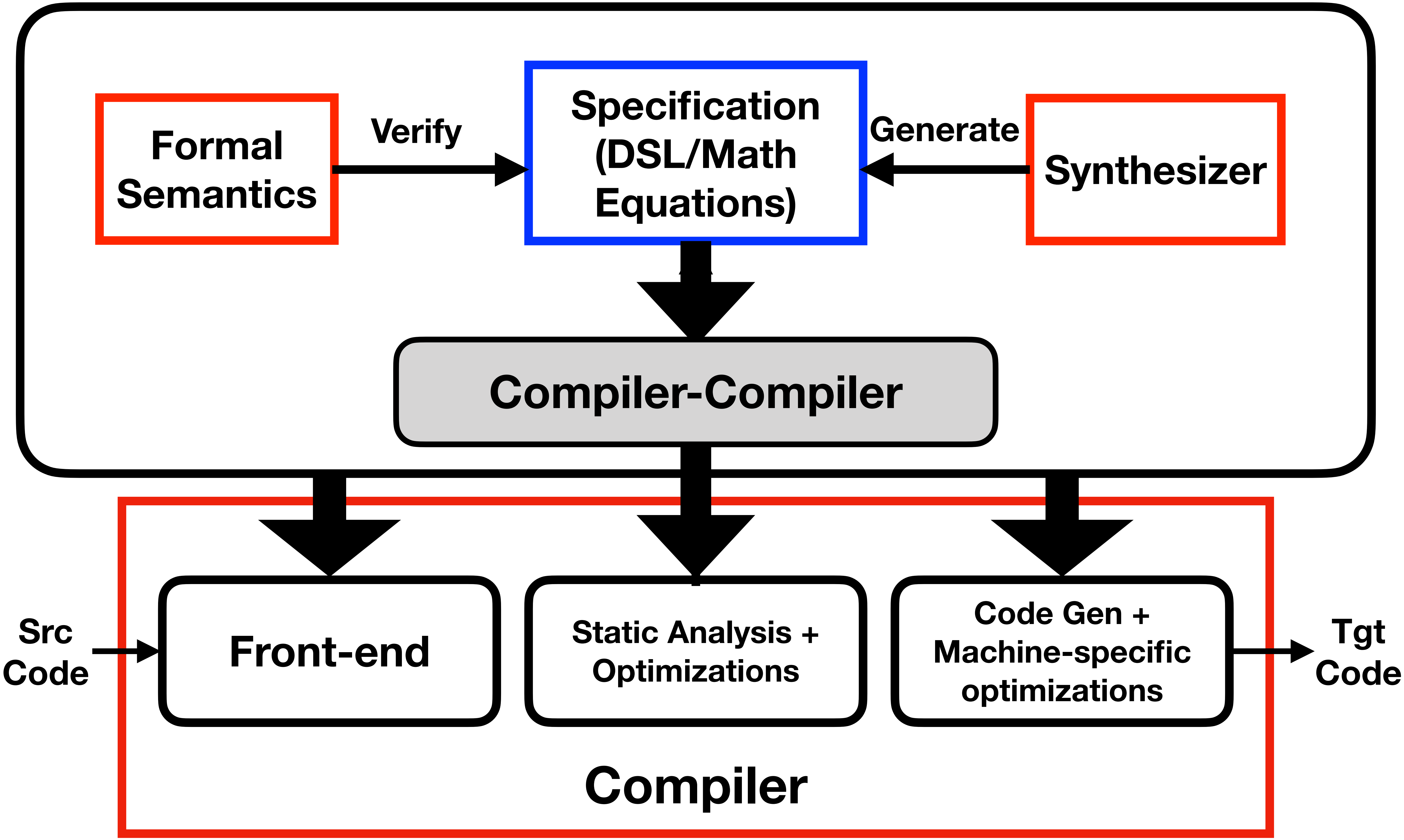
Generalizing testing of static analyses in different abstract domains, and across different compilers

Future Work

Superoptimizing source programs

Future Work

Superoptimizing loop transformations


Formal Semantics
Verify
Specification (DSL/Math Equations)
Generate
Synthesizer
Compiler-Compiler
Src Code
Front-end
Static Analysis + Optimizations
Code Gen + Machine-specific optimizations
Tgt Code
Compiler

Questions?

@jubitaneja