Provably Correct Peephole Optimizations with Alive

Nuno Lopes
MSR

David Menendez
Santosh Nagarakatte
Rutgers Univ.

John Regehr
Univ. of Utah

Compilers are Buggy

- Csmith [PLDI'11]:
  79 bugs in GCC (25 P1)
  202 bugs in LLVM
  2 wrong-code bugs in CompCert

- Orion [PLDI'14]:
  40 wrong-code bugs in GCC
  42 wrong-code bugs in LLVM

- Last Week:
  439 open wrong-code bug reports in GCC (out of 9,691)
  24 open wrong-code bug reports in LLVM (out of 6,761)

Buggy Compilers = Security Bugs

- CVE-2006-1902
- GCC 4.1/4.2 (fold-const.c) had a bug that could remove valid pointer comparisons
- Result: Removed some bounds checks from programs

Peephole Optimizers are Particularly Buggy

- LLVM's InstCombine (a peephole optimizer) had the most bugs reported by fuzzing tools
- InstCombine has 20,000 lines of C++
- Semantics of LLVM IR are tricky; InstCombine exploits the corner cases to improve performance:
  - Undefined behavior, poison values, undefined values, overflows, …

Optimizations are Easy to Get Wrong

```
int a = x << c;
int b = a / d;
```

→

```
int t = d / (1 << c);
int b = x / t;
```

$x * 2^c / d$

$$
\begin{aligned}
x / (d / 2^c) &= x / d * 2^c \\
&= x * 2^c / d
\end{aligned}
$$

(c and d are constants)

Optimizations are Easy to Get Wrong

```
int a = x << c;
int b = a / d;
```

➡

```
int t = d / (1 << c);
int b = x / t;
```

```
ERROR: Domain of definedness of Target is smaller
than Source's for i4 %b

Example:
%X i4 = 0x0 (0)
c  i4 = 0x3 (3)
d  i4 = 0x7 (7)
%a i4 = 0x0 (0)
(1 << c) i4 = 0x8 (8, -8)
%t i4 = 0x0 (0)
Source value: 0x0 (0)
Target value: undef
```

LLVM bug #21245

Implementing Peephole Optimizers

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

Alive

- New language and tool for:
  Specifying peephole optimizations
  Proving them correct (or generate a counterexample)
  Generating C++ code for a compiler
- Design point: both practical and formal

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
int f(int x, int y) {
  return (x / y) * y;
}
```

Compile to LLVM IR

```
define i32 @f(i32 %x, i32 %y) {
  %1 = sdiv i32 %x, %y
  %2 = mul i32 %1, %y
  ret i32 %2
}
```

Optimize

```
define i32 @f(i32 %x, i32 %y) {
  %1 = srem i32 %x, %y
  %2 = sub i32 %x, %1
  ret i32 %2
}
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
define i32 @f(i32 %x, i32 %y) {
  %1 = sdiv i32 %x, %y
  %2 = mul i32 %1, %y
  ret i32 %2
}
```

=>

Optimize

```
define i32 @f(i32 %x, i32 %y) {
  %1 = srem i32 %x, %y
  %2 = sub i32 %x, %1
  ret i32 %2
}
```


```
define i32 @f(i32 %x, i32 %y) {
  %1 = srem i32 %x, %y
  %2 = sub i32 %x, %1
  ret i32 %2
}
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
%1 = sdiv i32 %x, %y
%2 = mul i32 %1, %y

=>

%t = srem i32 %x, %y
%2 = sub i32 %x, %t
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
%1 = sdiv i32 %x, %y
%2 = mul i32 %1, %y
  =>
%t = srem i32 %x, %y
%2 = sub i32 %x, %t
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
%1 = sdiv %x, %y
%2 = mul %1, %y
  =>
%t = srem %x, %y
%2 = sub %x, %t
```

A Simple Peephole Optimization

```
{
  Value *Op1C = Op1;
  BinaryOperator *BO = dyn_cast<BinaryOperator>(Op0);
  if (!BO ||
      (BO->getOpcode() != Instruction::UDiv &&
       BO->getOpcode() != Instruction::SDiv)) {
    Op1C = Op0;
    BO = dyn_cast<BinaryOperator>(Op1);
  }
  Value *Neg = dyn_castNegVal(Op1C);
  if (BO && BO->hasOneUse() &&
      (BO->getOperand(1) == Op1C || BO->getOperand(1) == Neg) &&
      (BO->getOpcode() == Instruction::UDiv ||
       BO->getOpcode() == Instruction::SDiv)) {
    Value *Op0BO = BO->getOperand(0), *Op1BO = BO->getOperand(1);

    // If the division is exact, X % Y is zero, so we end up with X or -X.
    if (PossiblyExactOperator *SDiv = dyn_cast<PossiblyExactOperator>(BO))
      if (SDiv->isExact()) {
        if (Op1BO == Op1C)
          return ReplaceInstUsesWith(I, Op0BO);
        return BinaryOperator::CreateNeg(Op0BO);
      }

    Value *Rem;
    if (BO->getOpcode() == Instruction::UDiv)
      Rem = Builder->CreateURem(Op0BO, Op1BO);
    else
      Rem = Builder->CreateSRem(Op0BO, Op1BO);
    Rem->takeName(BO);

    if (Op1BO == Op1C)
      return BinaryOperator::CreateSub(Op0BO, Rem);
    return BinaryOperator::CreateSub(Rem, Op0BO);
  }
}
```

```
Name: sdiv general
%1 = sdiv %x, %y
%2 = mul %1, %y
  =>
%t = srem %x, %y
%2 = sub %x, %t


Name: sdiv exact
%1 = sdiv exact %x, %y
%2 = mul %1, %y
  =>
%2 = %x
```

Alive Language

Precondition

```
Pre: C2 % (1<<C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
  =>
%r = sdiv %X, C2/(1<<C1)
```

Source template

Target template

Predicates in preconditions may be the result of a dataflow analysis.

Alive Language

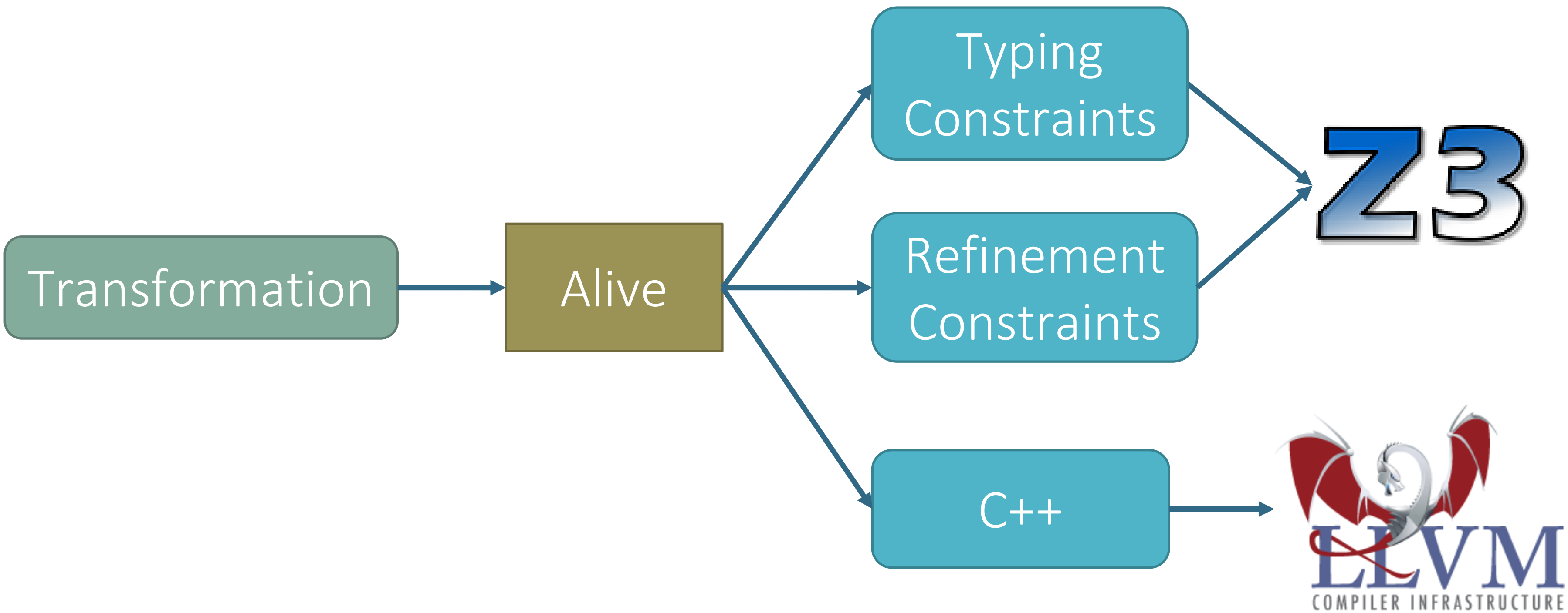
```
Pre: C2 % (1<<C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
  =>
%r = sdiv %X, C2/(1<<C1)
```

Constants

Generalized from LLVM IR:

- Symbolic constants
- Implicit types

Alive



Transformation
Alive
Typing Constraints
Refinement Constraints
C++
Z3
LLVM
COMPILER INFRASTRUCTURE

Correctness Criteria

1. Target invokes undefined behavior only when the source does

2. Result of target = result of source w source does not invoke undefined behavior

3. Final memory states are equiva

LLVM has 3 types of UB:

- Poison values
- Undef values
- True UB

See paper for more details

The story of a new optimization

A developer wrote a new optimization that improves benchmarks:

- 3.8% perlbmk (SPEC CPU 2000)
- 1% perlbench (SPEC CPU 2006)
- 1.2% perlbench (SPEC CPU 2006) w/ LTO+PGO

40 lines of code

August 2014

The story of a new optimization

- The first patch was wrong

```
Pre: isPowerOf2(C1 ^ C2)
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
  =>
%and = and %A, ~(C1 ^ C2)
%lhs = add %and, umax(C1, C2)
%r = icmp ult %lhs, C3
```

```
ERROR: Mismatch in values of %r

Example:
%A i4 = 0x0 (0)
C1 i4 = 0xA (10, -6)
C3 i4 = 0x5 (5)
C2 i4 = 0x2 (2)
%x i4 = 0xA (10, -6)
%i i1 = 0x0 (0)
%y i4 = 0x2 (2)
%j i1 = 0x1 (1, -1)
%and i4 = 0x0 (0)
%lhs i4 = 0xA (10, -6)
Source value: 0x1 (1, -1)
Target value: 0x0 (0)
```

The story of a new optimization

- The second patch was wrong
- The third patch was correct!
- Still fires on the benchmarks!

```
Pre: C1 u> C3 &&
     C2 u> C3 &&
     isPowerOf2(C1 ^ C2) &&
     isPowerOf2(-C1 ^ -C2) &&
     (-C1 ^ -C2) == ((C3-C1) ^ (C3-C2)) &&
     abs(C1-C2) u> C3
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
  =>
%and = and %A, ~(C1^C2)
%lhs = add %and, umax(C1,C2)
%r = icmp ult %lhs, C3
```

Experiments

1. Translated > 300 optimizations from LLVM’s InstCombine to Alive. Found 8 bugs; remaining proved correct.
2. Automatic optimal post-condition strengthening
   Significantly better than developers
3. Replaced InstCombine with automatically generated code

InstCombine: Stats per File

File	# opts.	# translated	# bugs
AddSub	67	49	2
AndOrXor	165	131	0
Calls	80	-	-
Casts	77	-	-
Combining	63	-	-
Compares	245	-	-
LoadStoreAlloca	28	17	0
MulDivRem	65	44	6
PHI	12	-	-
Select	74	52	0
Shifts	43	41	0
SimplifyDemanded	75	-	-
VectorOps	34	-	-
Total	1,028	334	8

14% wrong! (MulDivRem)

Optimal Attribute Inference

```
Pre: C1 % C2 == 0
%m = mul nsw %X, C1
%r = sdiv %m, C2
  =>
%r = mul nsw %X, C1/C2
```

States that the operation will not result in a signed overflow

Optimal Attribute Inference

- Weakened 1 precondition
- Strengthened the postcondition for 70 (21%) optimizations
  40% for AddSub, MulDivRem, Shifts

- Postconditions state, e.g., when an operation will not overflow

Alive is Useful!

- Released as open-source in Fall 2014
- In use by developers across 6 companies
- Already caught dozens of bugs in new patches
- Talks about replacing InstCombine

Conclusion

- (Peephole) optimizers are huge and buggy
- Presented Alive, a DSL+tool to specify peephole optimizations
  - Usable by compiler developers (easy to learn; friendly interface)
  - Automatic verification
  - Generates C++ implementation automatically
- Available from https://github.com/nunoplopes/alive/


ALIVE

Instruction Attributes

Instructions may become poison:

- NSW: no signed wrap
- NUW: no unsigned wrap
- Exact: lossless operation

Essential for optimization, but extremely hard to reason by hand

Valid Associativity

```
%t = add %A, %B
%r = add %t, %C
```

→

```
%t = add %B, %C
%r = add %A, %t
```

$$(A + B) + C = A + (B + C)$$

Associativity w/ NSW

%A = 50
%B = -50
%C = -100

```
%t = add nsw i8 %A, %B
%r = add nsw i8 %t, %C
```

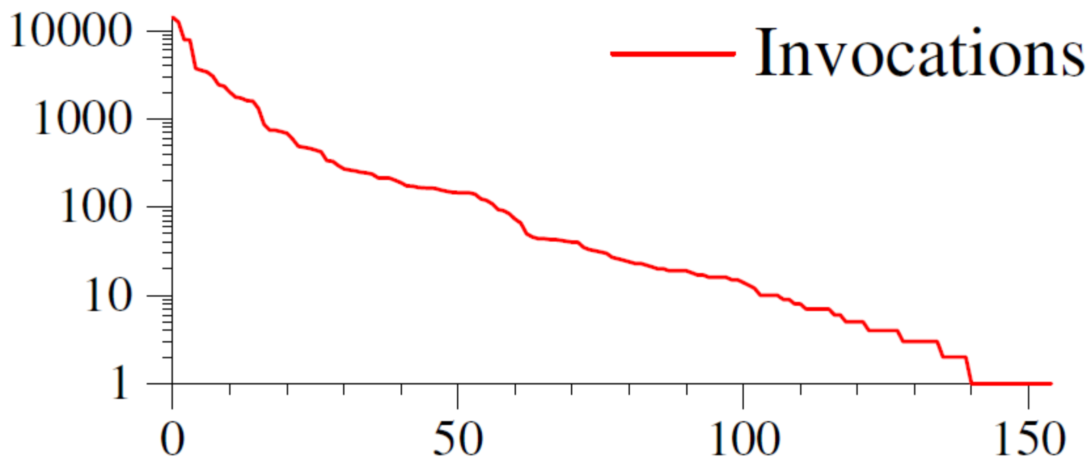
➡

```
%t = add nsw i8 %B, %C
%r = add nsw i8 %A, %t
```

%t = 0
%r = -100

%t = poison (-150)
%r = poison

Long Tail of Optimizations



SPEC gives poor code coverage

Definedness Constraints

Instruction	Definedness Constraint
sdiv a, b	$b \neq 0 \wedge (a \neq \mathit{INT\_MIN} \vee b \neq -1)$
udiv a, b	$b \neq 0$
srem a, b	$b \neq 0 \wedge (a \neq \mathit{INT\_MIN} \vee b \neq -1)$
urem a, b	$b \neq 0$
shl a, b	$b <_u B$
lshr a, b	$b <_u B$
ashr a, b	$b <_u B$

Poison-free Constraints

Instruction	Constraints for Poison-free execution
add nsw a, b	$SExt(a, 1) + SExt(b, 1) = SExt(a + b, 1)$
add nuw a, b	$ZExt(a, 1) + ZExt(b, 1) = ZExt(a + b, 1)$
sub nsw a, b	$SExt(a, 1) - SExt(b, 1) = SExt(a - b, 1)$
sub nuw a, b	$ZExt(a, 1) - ZExt(b, 1) = ZExt(a - b, 1)$
mul nsw a, b	$SExt(a, B) \times SExt(b, B) = SExt(a \times b, B)$
mul nuw a, b	$ZExt(a, B) \times ZExt(b, B) = ZExt(a \times b, B)$
sdiv exact a, b	$(a \div b) \times b = a$
udiv exact a, b	$(a \div_u b) \times b = a$
shl nsw a, b	$(a << b) >> b = a$
shl nuw a, b	$(a << b) >>_u b = a$
ashr exact a, b	$(a >> b) << b = a$
lshr exact a, b	$(a >>_u b) << b = a$

PR20186: wrong value

```
%div = sdiv %x, C
%r   = sub 0, %div
  =>
%r   = sdiv %x, -C
```

PR20189: introduce poison value

```
%B = sub 0, %A
%C = sub nsw %X, %B
  =>
%C = add nsw %X, %A
```

PR21242: introduce poison value

```
Pre: isPowerOf2(C1)
%r = mul nsw %x, C1
  =>
%r = shl nsw %x, log2(C1)
```

PR21243: wrong value

```
Pre: !WillNotOverflowSignedMul(C1, C2)
%Op0 = sdiv %X, C1
%r = sdiv %Op0, C2
  =>
%r = 0
```

PR21245: wrong value

```
Pre: C2 % (1<<C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
  =>
%r = sdiv %X, (C2 / (1 << C1))
```

PR21255: introduce undef behavior

```
%Op0 = lshr %X, C1
%r = udiv %Op0, C2
 =>
%r = udiv %X, (C2 << C1)
```

PR21256: introduce undef behavior

```
%Op1 = sub 0, %X
%r = srem %Op0, %Op1
 =>
%r = srem %Op0, %X
```

PR21274: introduce undef behavior

```
Pre: isPowerOf2(%Power)
%shl = shl %Power, %A
%Y = lshr %shl, %B
%r = udiv %X, %Y
 =>
%sub = sub %A, %B
%Y = shl %Power, %sub
%r = udiv %X, %Y
```

Precondition Predicates

equivalentAddressValues

isPowerOf2

isPowerOf2OrZero

isShiftedMask

isSignBit

MaskedValueIsZero

WillNotOverflowSignedAdd

WillNotOverflowUnsignedAdd

WillNotOverflowSignedSub

WillNotOverflowUnsignedSub

WillNotOverflowSignedMul

WillNotOverflowUnsignedMul

WillNotOverflowUnsignedShl