Super-optimizing LLVM IR

Duncan Sands

DeepBlueCapital / CNRS

Thanks to

Google

for sponsorship

Super optimization

- Optimization $\rightarrow$ Improve code

Super optimization

- Optimization $\rightarrow$ Improve code
- Super-optimization $\rightarrow$ Obtain perfect code

Super optimization

- ~~Optimization $\rightarrow$ Improve code~~
- ~~Super-optimization $\rightarrow$ Obtain perfect code~~

Super-optimization $\rightarrow$ automatically find code improvements

Super optimization

- ~~Optimization $\rightarrow$ Improve code~~
- ~~Super-optimization $\rightarrow$ Obtain perfect code~~

Super-optimization $\rightarrow$ automatically find code improvements

Idea from LLVM OpenProjects web-page
(suggested by John Regehr)

Goal

Automatically find simplifications missed by the LLVM optimizers

- And have a human implement them in LLVM

Goal

Automatically find simplifications missed by the LLVM optimizers

- And have a human implement them in LLVM

Non goal

Directly optimize programs

Goal

Automatically find simplifications missed by the LLVM optimizers

- And have a human implement them in LLVM

Non goal

~~Directly optimize programs~~

→ It doesn't matter if the simplifications found are sometimes wrong

Examples

Missed simplifications found in “fully optimized” code:

- $X - (X - Y) \rightarrow Y$

Examples

Missed simplifications found in “fully optimized” code:

- X - (X - Y) $\rightarrow$ Y   Not done because of operand uses

Examples

Missed simplifications found in “fully optimized” code:

- X - (X - Y) $\rightarrow$ Y   Not done because of operand uses
- (X<<1) - X $\rightarrow$ X

Examples

Missed simplifications found in “fully optimized” code:

- X - (X - Y) $\rightarrow$ Y — Not done because of operand uses
- (X<<1) - X $\rightarrow$ X — Not done because of operand uses

Examples

Missed simplifications found in “fully optimized” code:

- X - (X - Y) $\rightarrow$ Y — Not done because of operand uses
- (X<<1) - X $\rightarrow$ X — Not done because of operand uses
- non-negative number + power-of-two != 0 $\rightarrow$ true

Examples

Missed simplifications found in “fully optimized” code:

- $X - (X - Y) \rightarrow Y$ Not done because of operand uses
- $(X<<1) - X \rightarrow X$ Not done because of operand uses
- non-negative number + power-of-two != 0 $\rightarrow$ true New!

Process

- Compile program to bitcode

Process

- Compile program to bitcode
- Run optimizers on bitcode

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications
- Implement the simplifications in LLVM

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications
- Implement the simplifications in LLVM

Repeat

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications
- Implement the simplifications in LLVM
- Profit!

Repeat

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications
- Implement the simplifications in LLVM
- Profit!

Repeat

Inspired by “Automatic Generation of Peephole Superoptimizers” by Bansal & Aiken (Computer Systems Lab, Stanford)

Harvesting

```
$ opt -load=./harvest.so -std-compile-opts -harvest -details \
    -disable-output bzip2.bc
@07:@09
{
  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
{
  ; In function: "bsPutUInt32()", BB: "bsW.exit"
  %28 = lshr i32 %u, 16
  %29 = and i32 %28, 255
  %49 = sub i32 24, %48 ; From BB: "bsW.exit24"
  %50 = shl i32 %29, %49        ; From BB: "bsW.exit24"
  %51 = or i32 %50, %47 ; From BB: "bsW.exit24"
}
...
```

Harvesting

Plugin pass that harvests code sequences

```
$ opt -load=./harvest.so -std-compile-opts -harvest -details \
    -disable-output bzip2.bc
@07:@09
{
  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
{
  ; In function: "bsPutUInt32()", BB: "bsW.exit"
  %28 = lshr i32 %u, 16
  %29 = and i32 %28, 255
  %49 = sub i32 24, %48 ; From BB: "bsW.exit24"
  %50 = shl i32 %29, %49        ; From BB: "bsW.exit24"
  %51 = or i32 %50, %47 ; From BB: "bsW.exit24"
}
...
```

Harvesting

Harvest code sequences after running standard optimizers

```
$ opt -load=./harvest.so -std-compile-opts -harvest -details \
    -disable-output bzip2.bc
@07:@09
{
  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
{
  ; In function: "bsPutUInt32()", BB: "bsW.exit"
  %28 = lshr i32 %u, 16
  %29 = and i32 %28, 255
  %49 = sub i32 24, %48 ; From BB: "bsW.exit24"
  %50 = shl i32 %29, %49        ; From BB: "bsW.exit24"
  %51 = or i32 %50, %47 ; From BB: "bsW.exit24"
}
...
```

Harvesting

```
$ opt -load=./harvest.so -std-compile-opts -harvest -details \
    -disable-output bzip2.bc
@07:@09
{
  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
{
  ; In function: "bsPutUInt32()", BB: "bsW.exit"
  %28 = lshr i32 %u, 16
  %29 = and i32 %28, 255
  %49 = sub i32 24, %48 ; From BB: "bsW.exit24"
  %50 = shl i32 %29, %49        ; From BB: "bsW.exit24"
  %51 = or i32 %50, %47 ; From BB: "bsW.exit24"
}
...
```

Code sequences

Harvesting

```
$ opt -load=./harvest.so -std-compile-opts -harvest -details \
    -disable-output bzip2.bc
@07:@09
{
  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
{
  ; In function: "bsPutUInt32()", BB: "bsW.exit"
  %28 = lshr i32 %u, 16
  %29 = and i32 %28, 255
  %49 = sub i32 24, %48 ; From BB: "bsW.exit24"
  %50 = shl i32 %29, %49          ; From BB: "bsW.exit24"
  %51 = or i32 %50, %47 ; From BB: "bsW.exit24"
}
...
```

Code sequences

Code sequence = maximal connected subgraph of the LLVM IR containing only supported operations

Harvesting

```
$ opt -load=./harvest.so -std-compile-opts -harvest -details \
    -disable-output bzip2.bc
@07:@09
{
  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
{
  ; In function: "bsPutUInt32()", BB: "bsW.exit"
  %28 = lshr i32 %u, 16
  %29 = and i32 %28, 255
  %49 = sub i32 24, %48 ; From BB: "bsW.exit24"
  %50 = shl i32 %29, %49        ; From BB: "bsW.exit24"
  %51 = or i32 %50, %47 ; From BB: "bsW.exit24"
}
...
```

Normalized expressions

Harvesting

```
$ opt -load=./harvest.so -std-compile-opts -harvest -details \
    -disable-output bzip2.bc
@07:@09
{
  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
{
  ; In function: "bsPutUInt32()", BB: "bsW.exit"
  %28 = lshr i32 %u, 16
  %29 = and i32 %28, 255
  %49 = sub i32 24, %48 ; From BB: "bsW.exit24"
  %50 = shl i32 %29, %49        ; From BB: "bsW.exit24"
  %51 = or i32 %50, %47 ; From BB: "bsW.exit24"
}
...
```

Explanatory annotations (ignored)

Harvesting

```
$ opt -load=./harvest.so -std-compile-opts -harvest \
    -disable-output bzip2.bc
@07:@09
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
...
```

Normalized & encoded form allows textual comparisons:

```
$ opt -load=./harvest.so -std-compile-opts -harvest \
  -disable-output bzip2.bc | sort | uniq -c | sort -r -n
    265 @00:07:@2b
    178 @01:07:@0f
    120 @00:@07:@2b
    ...
```

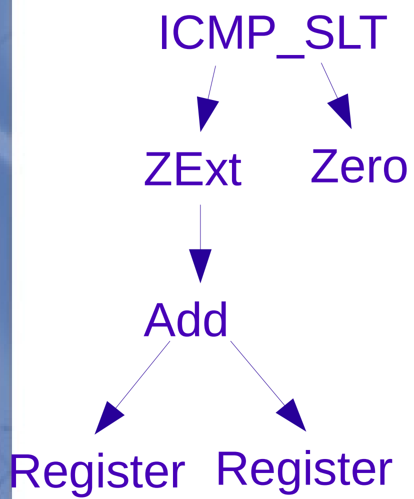
} Ordered by frequency of occurrence

Harvesting

Most common expressions in unoptimized bitcode from the LLVM testsuite:

07:0a	→ sext X	sext = sign-extend
00:07:2c	→ X != 0	
07:09	→ zext X	zext = zero-extend
05:07:0f	→ X +nsw -1	+nsw = add with no-signed wrap
00:07:2b	→ X == 0	
07:07:13	→ X -nsw Y	-nsw = sub with no-signed wrap
07:07:32	→ X >=s Y	>=s = signed greater than or equal
01:07:0f	→ X +nsw 1	
06:07:0a:16	→ (sext X) * power-of-2	power-of-2 = constant that is a power of two

Expressions



- Directed acyclic graph - no loops!
- Integer operations only - no floating point!
- No memory operations (load/store)!
- No types!
- Limited set of constants (eg: Zero, One, SignBit)

Most integer operations supported (eg: ctlz, overflow intrinsics).
Doesn't support byteswap (because of lack of types).

Analysing expressions

Four modes:

- Constant folding
- Reduce to sub-expression
- Unused variables
- Rule reduction

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 $\rightarrow$ 0 (i.e. false)

- Reduce to sub-expression
- Unused variables
- Rule reduction

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 $\rightarrow$ 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y $\rightarrow$ x + z

- Unused variables

- Rule reduction

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

- Rule reduction

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

- Rule reduction

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

  Result does not depend on x
  Can replace x with (eg) 0

- Rule reduction

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 $\rightarrow$ 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y $\rightarrow$ x + z

- Unused variables

  x - (x + y) $\rightarrow$ 0 - y

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Rafael Auler's GSOC project

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Fast!

Always a win!

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Fast!

Always a win!

Fast!

Often a win!

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Fast!
Always a win!
Fast!
Often a win!
Fast!
Sometimes a win!

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Fast! Always a win!

Fast! Often a win!

Fast! Sometimes a win!

Slow! Work in progress!

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x + z) *nsw y) /s y → x + z

- Unused variables

  x - (x + y) → 0 - y

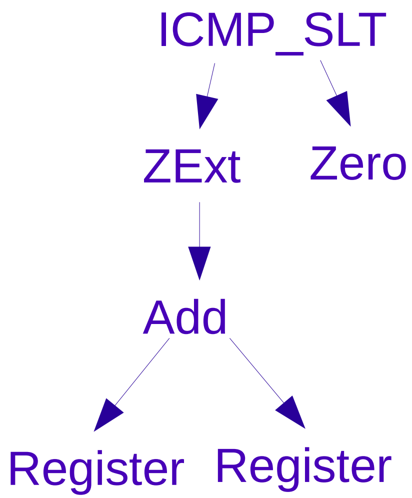
- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Implement in LLVM's InstructionSimplify analysis

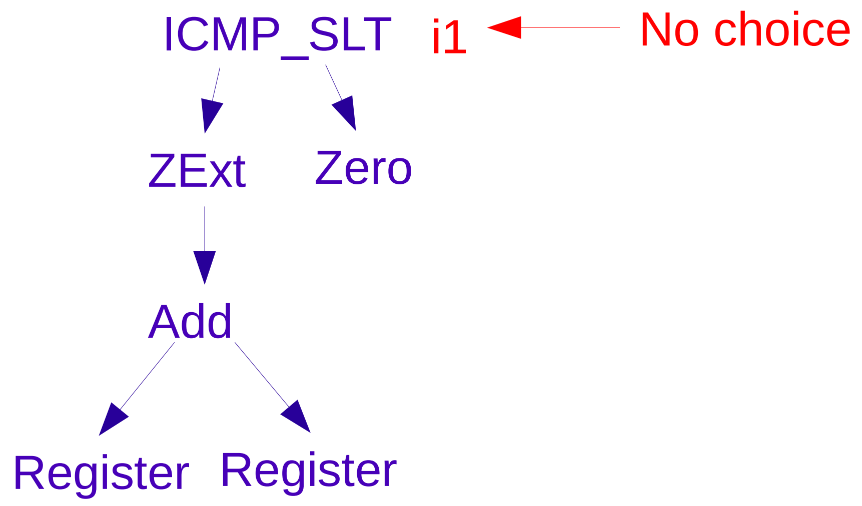
Implement in LLVM's InstCombine transform

Constant folding



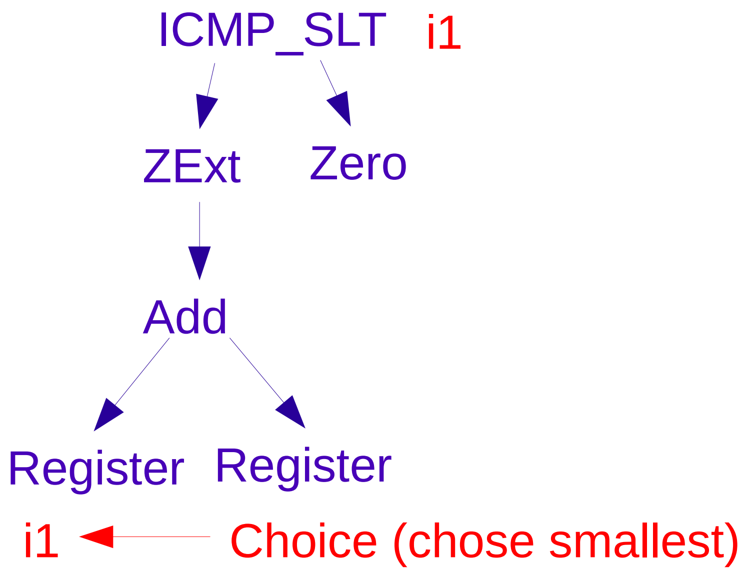
- Assign types to nodes

Constant folding



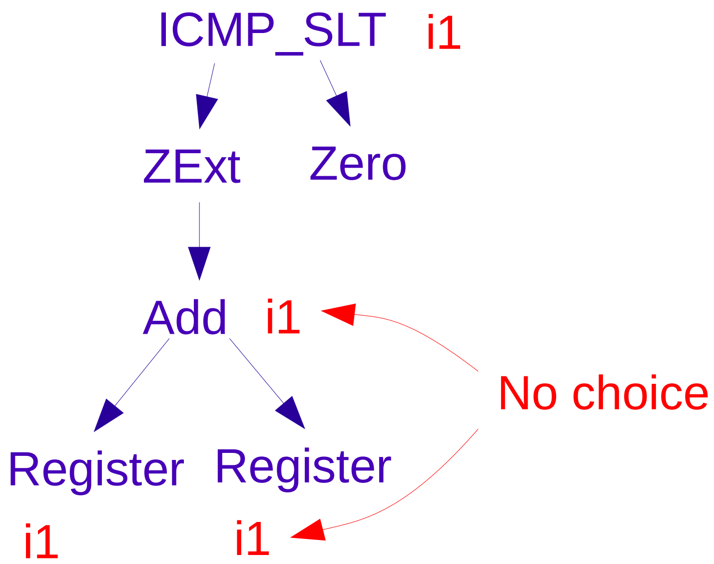
- Assign types to nodes

Constant folding



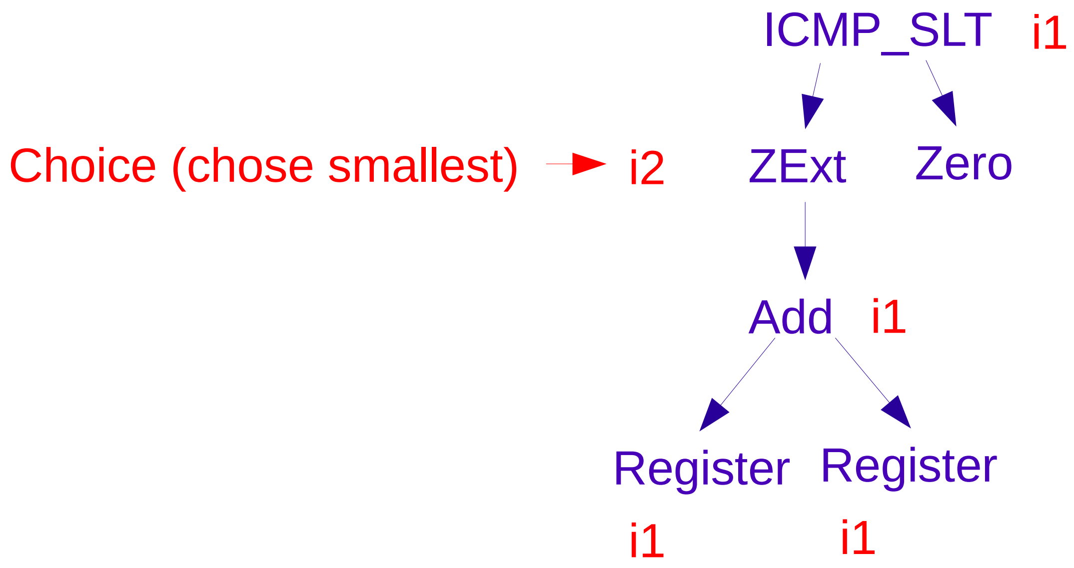
- Assign types to nodes

Constant folding



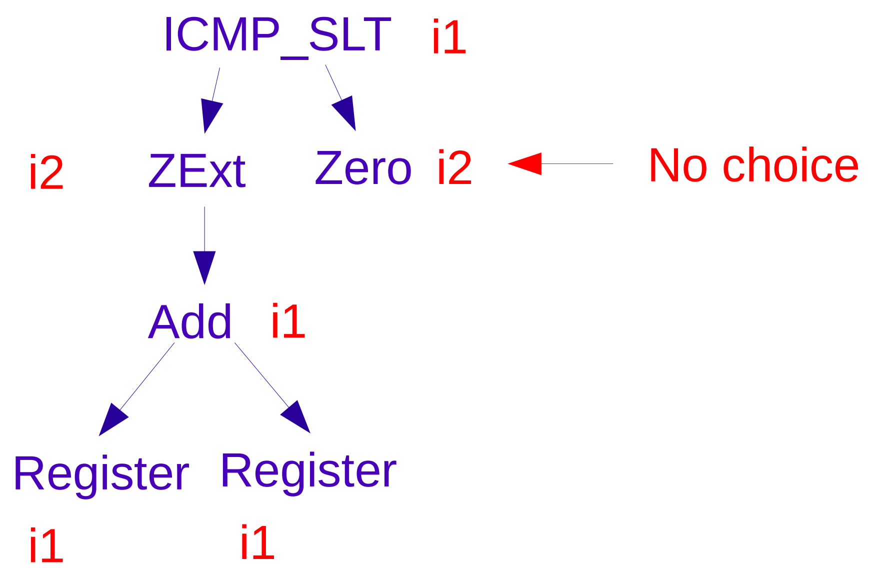
- Assign types to nodes

Constant folding



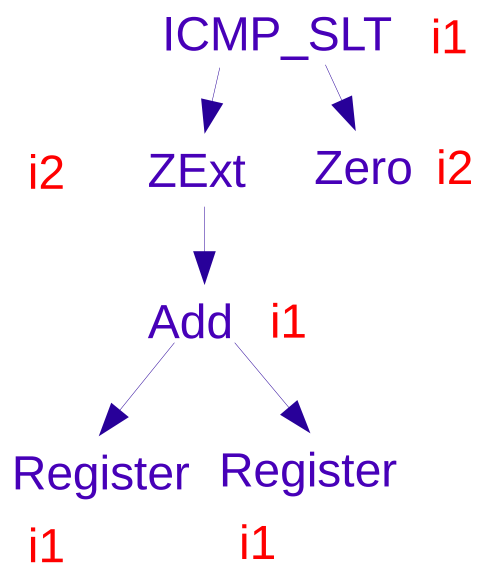
- Assign types to nodes

Constant folding



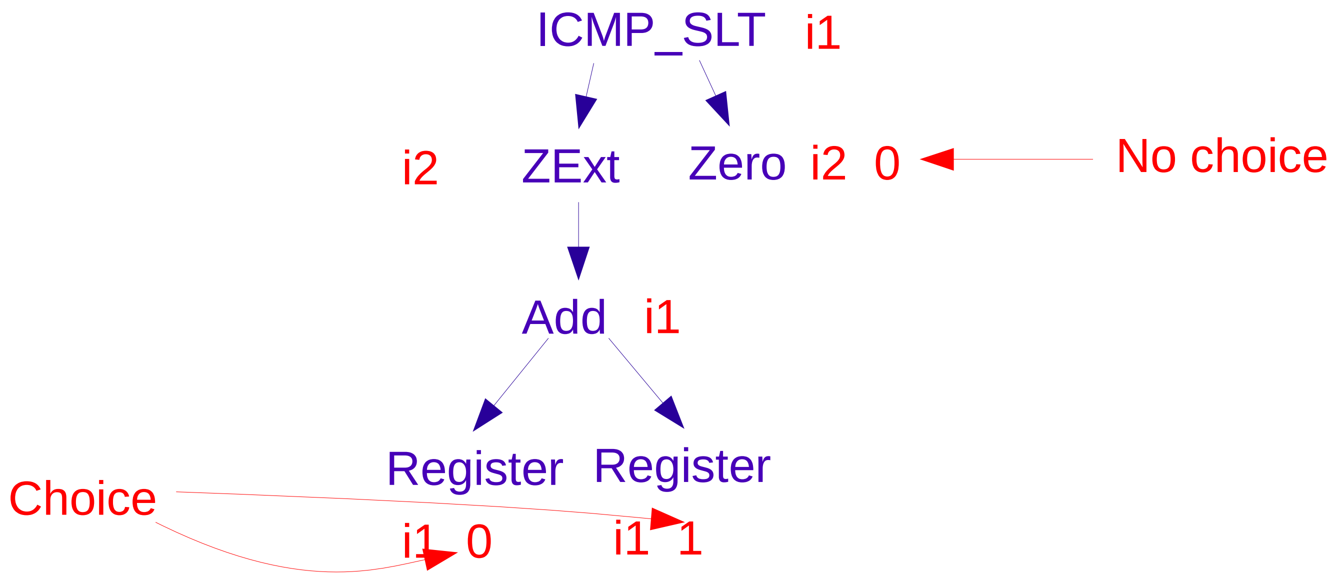
- Assign types to nodes

Constant folding



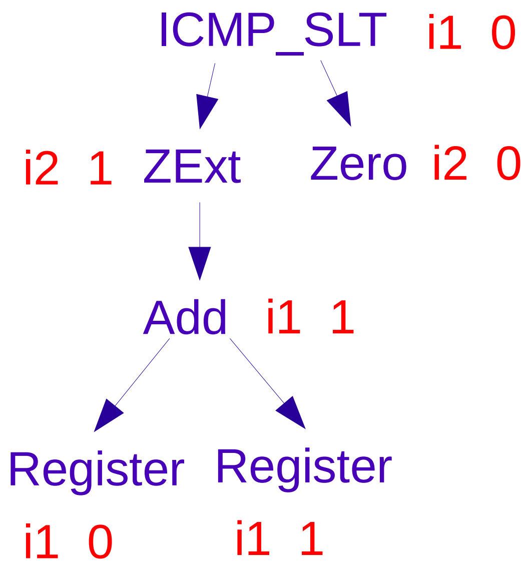
- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.

Constant folding



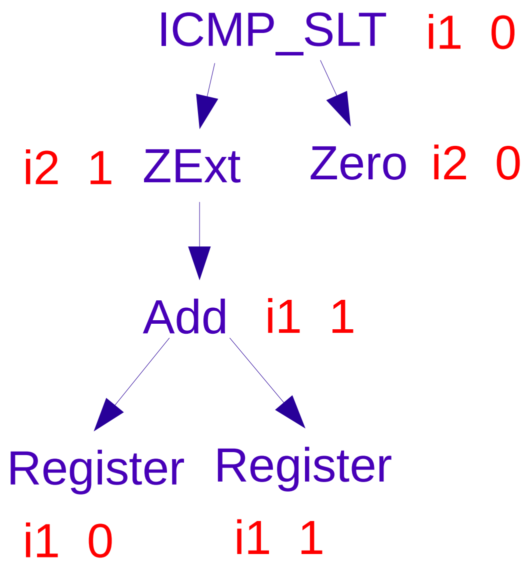
- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up

Constant folding



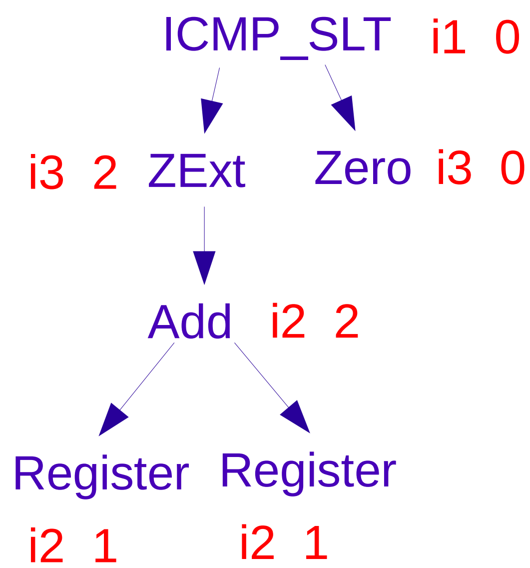
- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up

Constant folding



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.

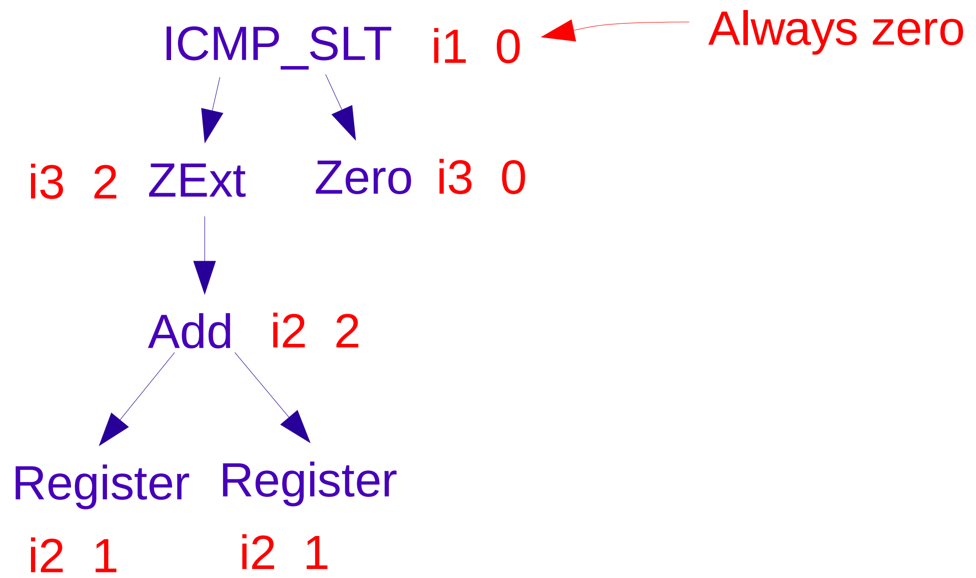
Constant folding



Repeat many times

- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.

Constant folding



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.
- Result at the root always the same
  → found a constant fold

False positives

Eg: A | (B + 1) | (C - 1) == 0

False positives

Mostly evaluates to “false”

Eg: A | (B + 1) | (C - 1) == 0

False positives

Mostly evaluates to "false"

Eg: A | (B + 1) | (C - 1) == 0

A, B and C have i8 type → 1 / 2^24 chance of seeing "true"

False positives

Mostly evaluates to “false”

Eg: A | (B + 1) | (C - 1) == 0

A, B and C have i8 type $\rightarrow$ 1 / 2^24 chance of seeing “true”

A, B and C have i1 type $\rightarrow$ 1 / 8 chance of seeing “true”

False positives

Mostly evaluates to "false"

Eg: A | (B + 1) | (C - 1) == 0

A, B and C have i8 type → 1 / 2^24 chance of seeing "true"

A, B and C have i1 type → 1 / 8 chance of seeing "true"

Use of small types hugely reduces the number of false positives

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false

  Implemented as: “non-negative-number + power-of-two != 0”

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 → false
- ((X >s Y) ? X : Y) >=s X → true

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false
- ((X >s Y) ? X : Y) >=s X $\rightarrow$ true

  “max(X, Y) >= X”. Implemented several max/min folds.

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false
- ((X >s Y) ? X : Y) >=s X $\rightarrow$ true
- X rem (Y ? X : 1) $\rightarrow$ 0
- (Y /u X) >u Y $\rightarrow$ false

Examples

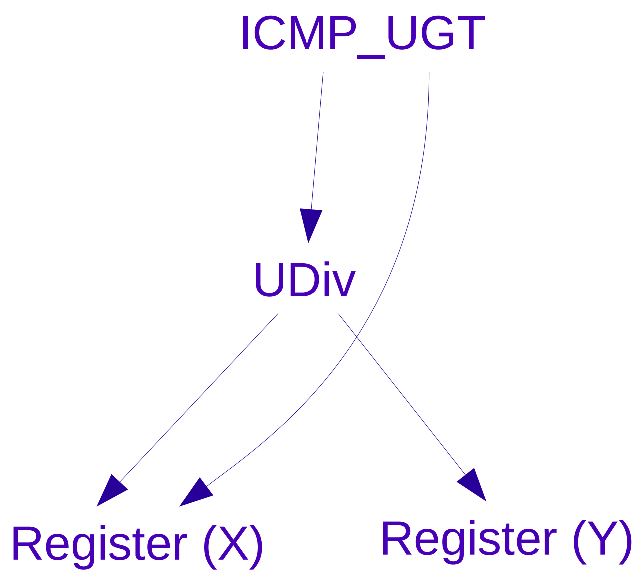
Constant folds found in "fully optimized" code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false
- ((X >s Y) ? X : Y) >=s X $\rightarrow$ true
- X rem (Y ? X : 1) $\rightarrow$ 0
- (Y /u X) >u Y $\rightarrow$ false

Require reasoning about undefined behaviour

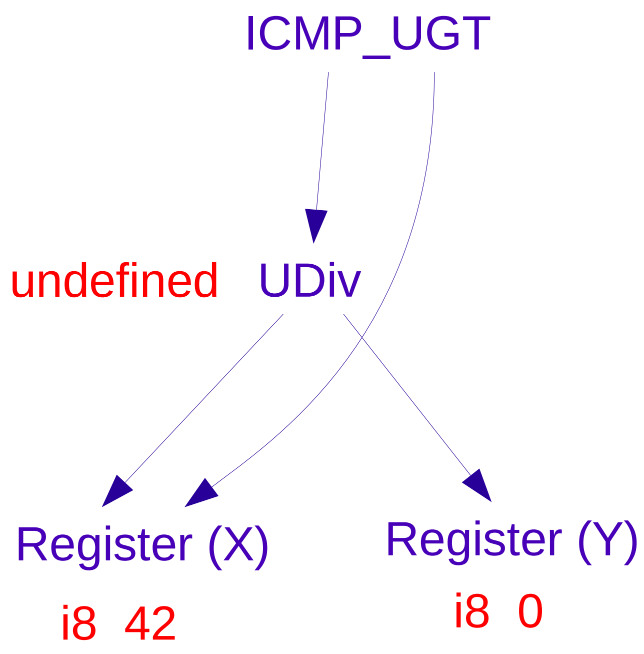
Undefined behaviour

(X /u Y) >u X $\rightarrow$ false



Undefined behaviour

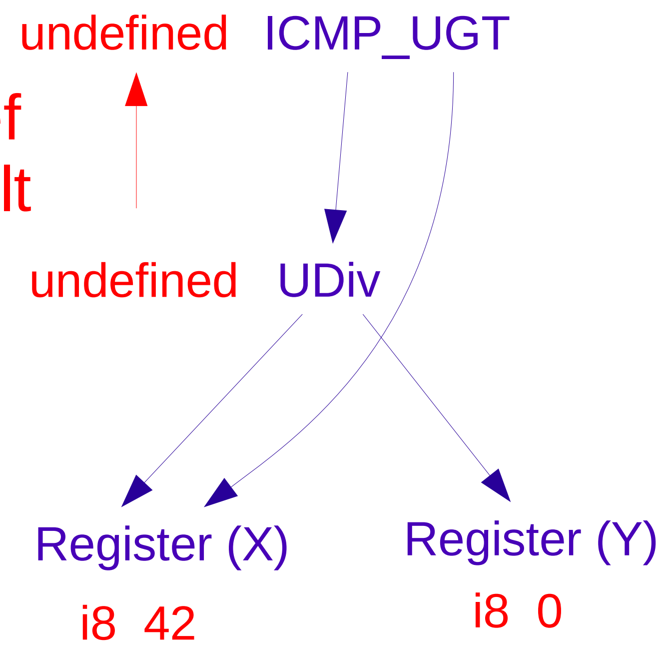
(X /u Y) >u X $\rightarrow$ false



Undefined behaviour

(X /u Y) >u X $\rightarrow$ false

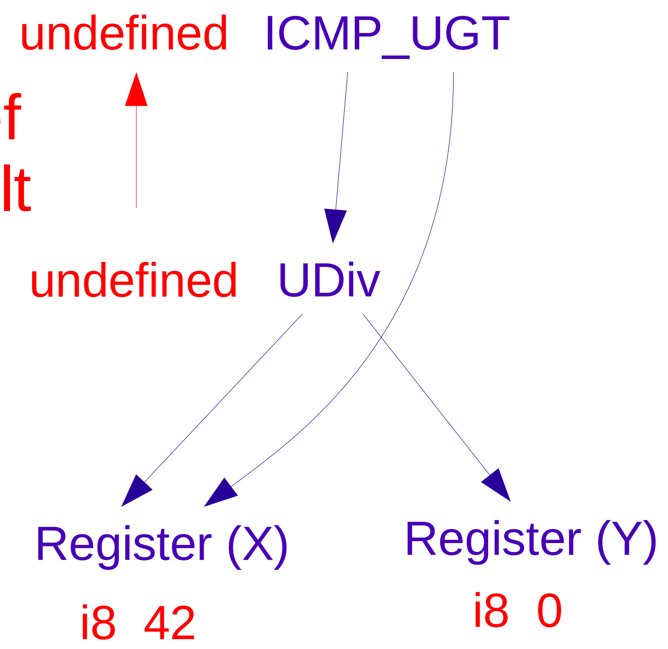
Any operation with an undef operand gets an undef result



Undefined behaviour

(X /u Y) >u X $\rightarrow$ false

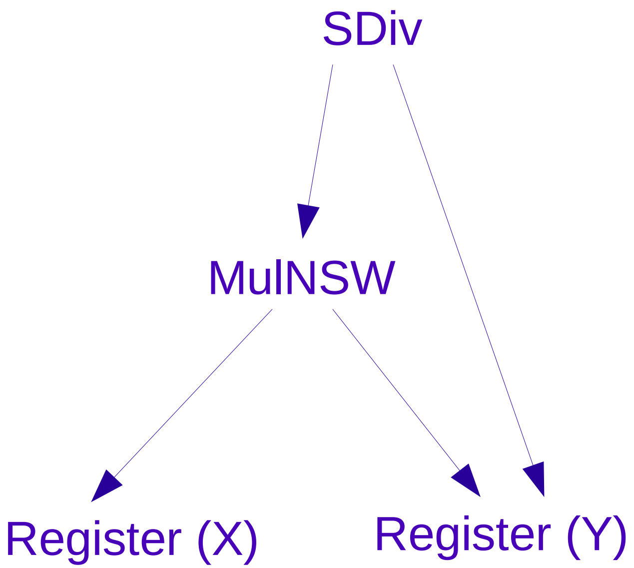
Any operation with an undef operand gets an undef result



- Avoids false negatives
- May result in subtle false positives

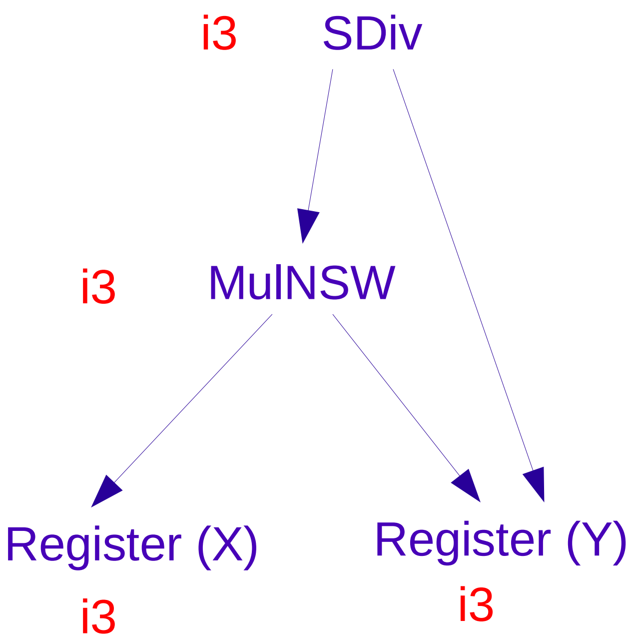
Reduce to subexpression

(X *nsw Y) /s Y $\rightarrow$ X



Reduce to subexpression

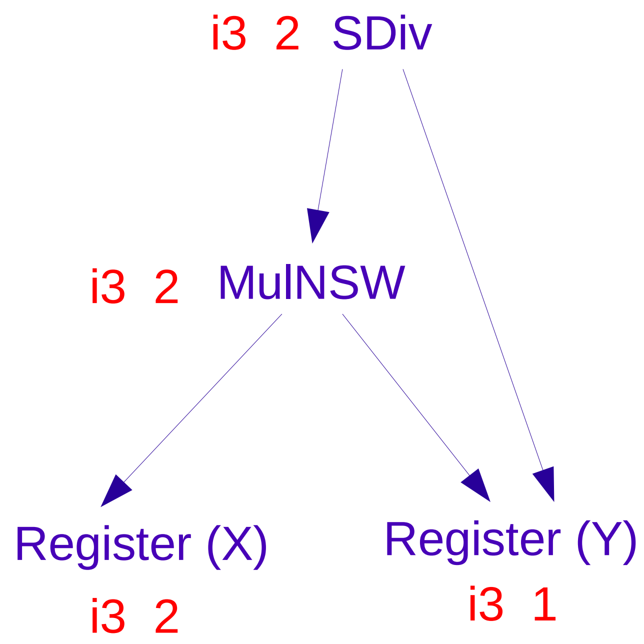
(X *nsw Y) /s Y $\rightarrow$ X



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.

Reduce to subexpression

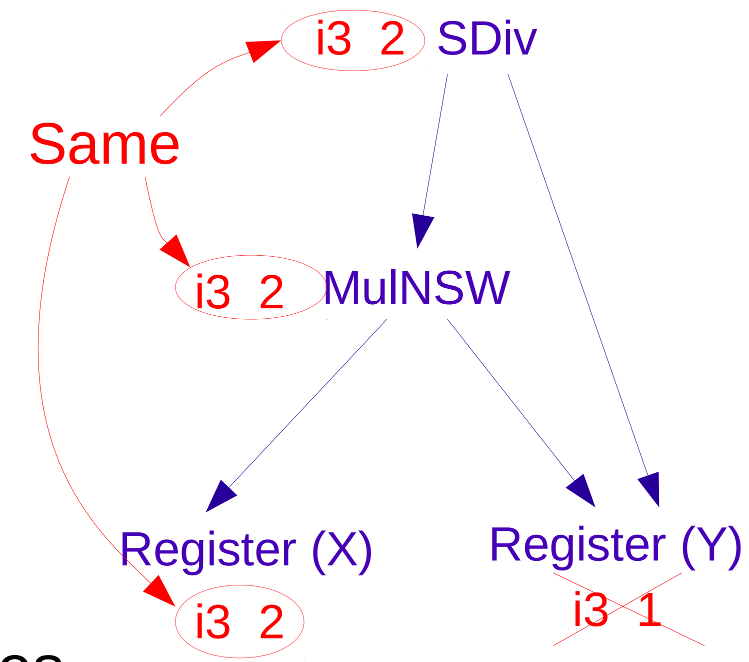
(X *nsw Y) /s Y $\rightarrow$ X



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.

Reduce to subexpression

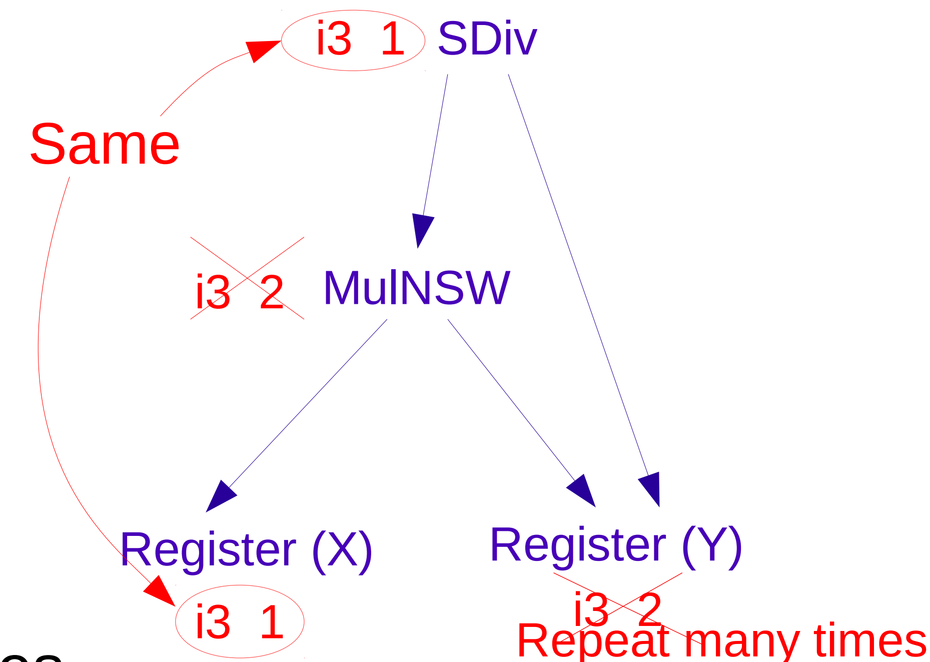
(X *nsw Y) /s Y $\rightarrow$ X



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.
- See if some node always has same value as root (or undef)

Reduce to subexpression

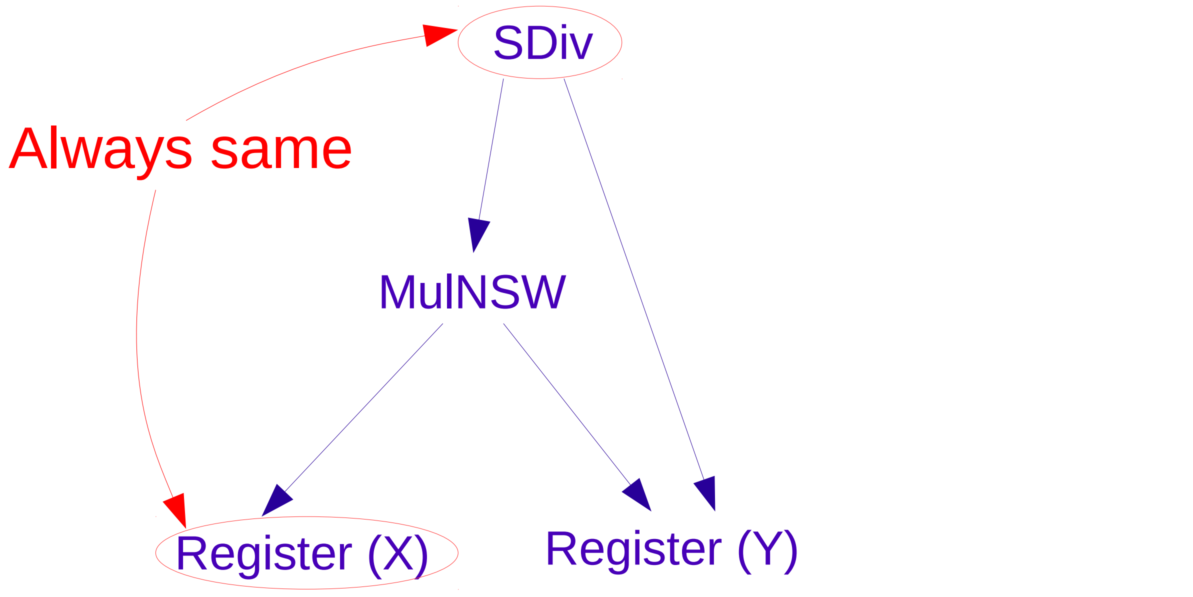
(X *nsw Y) /s Y $\rightarrow$ X



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.
- See if some node always has same value as root (or undef)

Reduce to subexpression

(X *nsw Y) /s Y $\rightarrow$ X



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.
- See if some node always has same value as root (or undef)

$\rightarrow$ found a subexpression reduction

Register pressure

(X *nsw Y) /s Y $\rightarrow$ X ← Is this always a win?

Register pressure

(X *nsw Y) /s Y $\rightarrow$ X ← Is this always a win?

Z = X *nsw Y

...

W = Z /s Y
call @foo(W, Y, Z)

Register pressure

(X *nsw Y) /s Y $\rightarrow$ X ← Is this always a win?

```
Z = X *nsw Y
```

X not used again

```
...

W = Z /s Y
call @foo(W, Y, Z)
```

Register pressure

(X *nsw Y) /s Y $\rightarrow$ X ← Is this always a win?

Z = X *nsw Y

Two registers needed (for Y, Z)

X not used again

...

W = Z /s Y
call @foo(W, Y, Z)

Register pressure

(X *nsw Y) /s Y $\rightarrow$ X ◄—— Is this always a win?

Z = X *nsw Y		Z = X *nsw Y
...	Transform: W $\rightarrow$ X ——►	...
W = Z /s Y call @foo(W, Y, Z)		... W not computed ... call @foo(X, Y, Z)

Register pressure

(X *nsw Y) /s Y → X ← Is this always a win?

Three registers needed (for X, Y, Z)

Z = X *nsw Y

...

... W not computed ...
call @foo(X, Y, Z)

Register pressure

(X *nsw Y) /s Y $\rightarrow$ X ← Is this always a win?

Transform increases the number of long lived registers by one.
May require spilling to the stack.

Unused variables

X +nsw Z >=s Z +nsw Y

Z is an “unused variable”

Unused variables

X +nsw Z >=s Z +nsw Y

Z is an “unused variable”

For every choice of the other variables (X, Y) the result of the expression does not depend on the value of Z (or is undefined)

Unused variables

X +nsw Z >=s Z +nsw Y

Z is an “unused variable”

For every choice of the other variables (X, Y) the result of the expression does not depend on the value of Z (or is undefined)

Replaced Z with 0

Transform:

X +nsw Z >=s Z +nsw Y → X >=s Y

Unused variables

X +nsw Z >=s Z +nsw Y

Z is an “unused variable”

For every choice of the other variables (X, Y) the result of the expression does not depend on the value of Z (or is undefined)

Replaced Z with 0

Transform:

X +nsw Z >=s Z +nsw Y → X >=s Y

Detect similarly to constant folding etc.

Examples

Unused variables found in “fully optimized” code:

- X >=s X +nsw Y
- ((X + Y) + -1) == X
- Y >>exact X == 0
- Y <<nsw X == 0

X is unused

Problems with unused variables

- More false positives than other modes

Problems with unused variables

- More false positives than other modes
- May increase register pressure

Problems with unused variables

- More false positives than other modes
- May increase register pressure
- May increase the amount of computation

Problems with unused variables

- More false positives than other modes
- May increase register pressure
- May increase the amount of computation

Eg: (A + B) * (C + D) == B * C + B * D

B is an unused variable

Problems with unused variables

- More false positives than other modes
- May increase register pressure
- May increase the amount of computation

Eg: (A + B) * (C + D) == B * C + B * D

B is an unused variable

Transforms to: A * C + A * D == 0

Problems with unused variables

- More false positives than other modes
- May increase register pressure
- May increase the amount of computation

Eg: (A + B) * (C + D) == B * C + B * D

B is an unused variable

Transforms to: A * C + A * D == 0

Requires computing A*C, A*D etc.

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);             // Commutativity
rule (0 And AllBitsSet) <=> 0;           // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);              // Commutativity
rule (0 And AllBitsSet) <=> 0;            // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y

Cost: 22

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y   Cost: 22

(X & Y) | (Y & AllOnesValue)   Cost: 30

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);             // Commutativity
rule (0 And AllBitsSet) <=> 0;           // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y — Cost: 22

(X & Y) | (Y & AllOnesValue) — Cost: 30

(X & Y) | (AllOnesValue & Y) — Cost: 30

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11
Y	Cost: 3

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);              // Commutativity
rule (0 And AllBitsSet) <=> 0;            // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11
Y	Cost: 3

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);             // Commutativity
rule (0 And AllBitsSet) <=> 0;           // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Time: 1 minute

(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11
Y	Cost: 3

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);              // Commutativity
rule (0 And AllBitsSet) <=> 0;            // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11
Y	Cost: 3

Time: 1 minute

SubExpr: 0.05 secs

UnusedVar: 0.08 secs

Rule reduction problems

- Slow

Rule reduction problems

- Slow
- Needs more rules

Rule reduction problems

- Slow
- Needs more rules
- Can this approach find unexpected simplifications?

(zext X) + power-of-two == 0 $\rightarrow$ false

Rule reduction problems

- Slow

Needs more work!

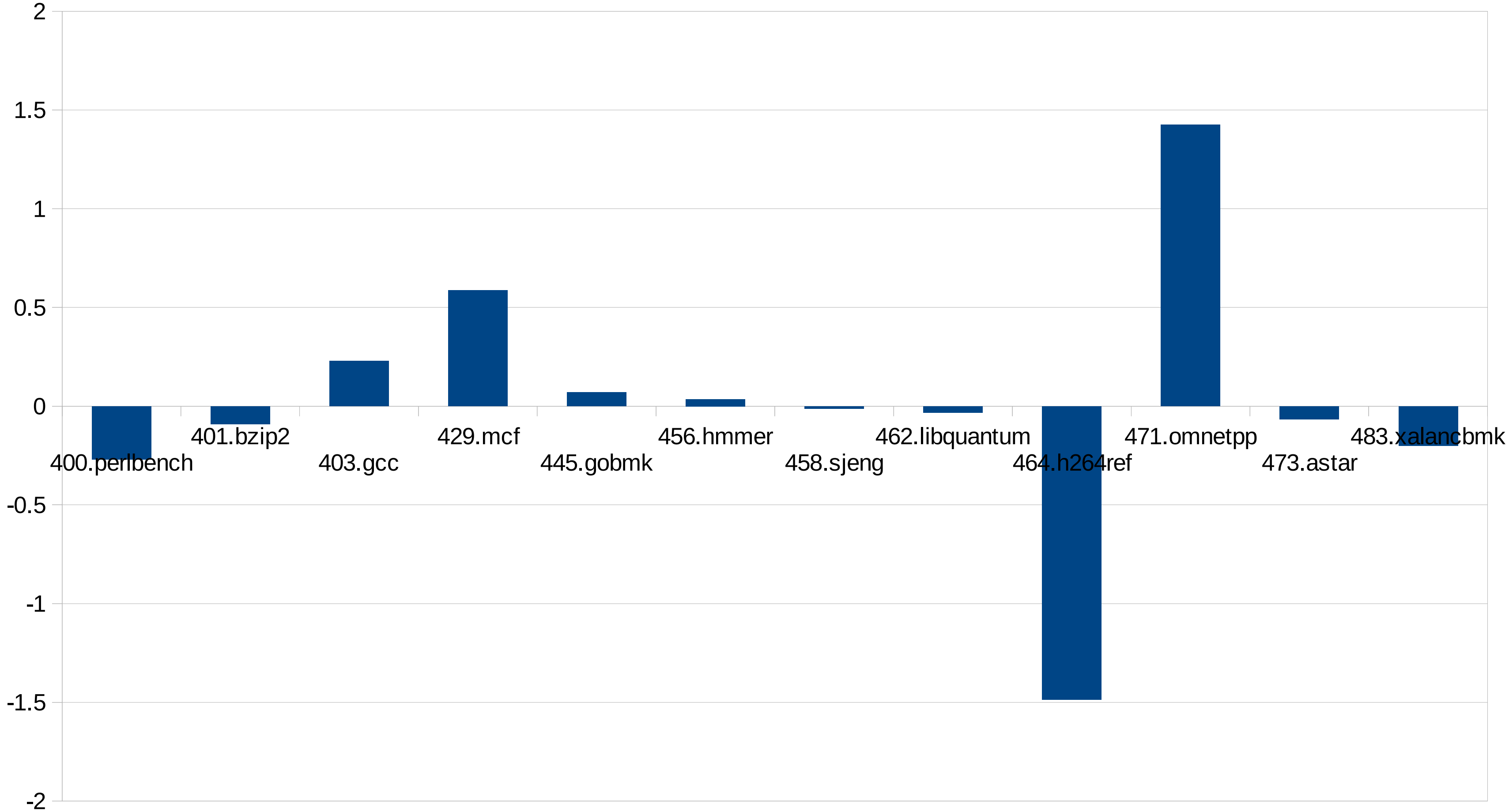
- Needs more rules
- Can this approach find unexpected simplifications?

(zext X) + power-of-two == 0 $\rightarrow$ false

Profit!

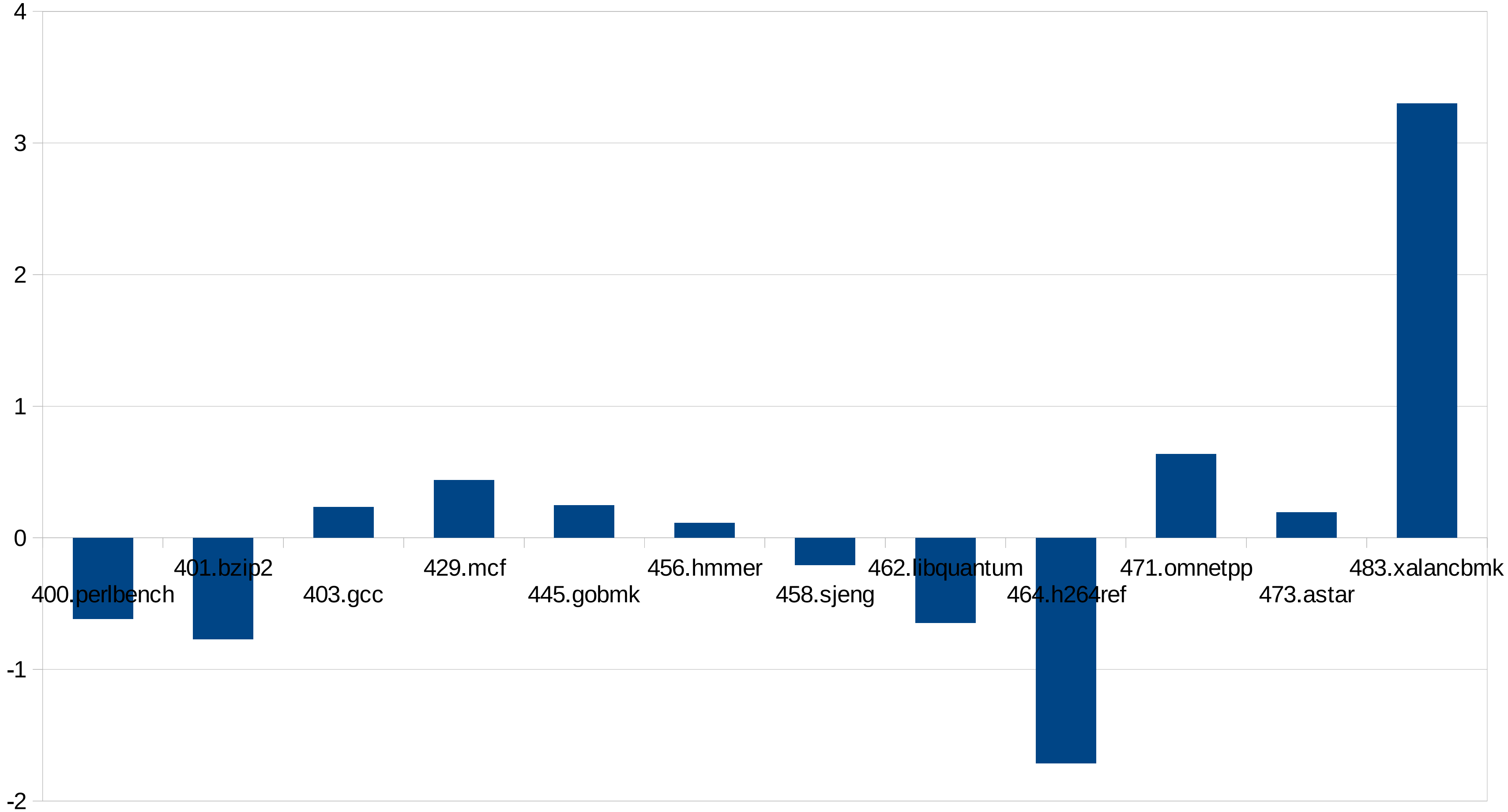
Profit?

Approximate % speed-up: constant folds



Profit?!

Approximate % speed-up: constant folds & reduce to sub-expr:



Improvements

- Work directly with LLVM IR

Improvements

- Work directly with LLVM IR

```
define i64 @combine(i64 %x) {
  %xl = trunc i64 %x to i32
  %h = lshr i64 %x, 32
  %xh = trunc i64 %h to i32
  %eh = zext i32 %xh to i64
  %el = zext i32 %xl to i64
  %h2 = shl i64 %eh, 32
  %r = or i64 %h2, %el
  ret i64 %r
}
```

← Simplifies to: ret %x

Improvements

- Work directly with LLVM IR

```
define i64 @combine(i64 %x) {
  %xl = trunc i64 %x to i32
  %h = lshr i64 %x, 32
  %xh = trunc i64 %h to i32
  %eh = zext i32 %xh to i64
  %el = zext i32 %xl to i64
  %h2 = shl i64 %eh, 32
  %r = or i64 %h2, %el
  ret i64 %r
}
```

← Simplifies to: ret %x

Impossible to find, due to

- Type-free expressions
- Limited number of constants

((zext (trunc (X >>l pow-2))) << pow-2) | (zext (trunc X))

Improvements

- Work directly with LLVM IR

  (Constant folding, subexpression reduction, unused variables)

  How to avoid many false positives?

Improvements

- Work directly with LLVM IR

  (Constant folding, subexpression reduction, unused variables)

  How to avoid many false positives?

- Sort expressions by execution frequency rather than textual frequency

Improvements

- Work directly with LLVM IR

  (Constant folding, subexpression reduction, unused variables)

  How to avoid many false positives?

- Sort expressions by execution frequency rather than textual frequency

  Eg: generate fake debug info using the encoded expression for the “function”.

  Hottest “functions” reported by profiling tools are the hottest expressions!

Getting it

svn://topo.math.u-psud.fr/harvest