Super-optimizing LLVM IR

Duncan Sands

DeepBlueCapital / CNRS
Thanks to

Google

for sponsorship
Super optimization

- Optimization → Improve code
Super optimization

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

- Optimization → Improve code
- Super-optimization → Obtain perfect code

Super-optimization → automatically find code improvements
Super optimization

- Optimization → Improve code
- Super-optimization → Obtain perfect code

Super-optimization → 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

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

- non-negative number + power-of-two $\neq 0 \rightarrow$ 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)
$ 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:@07:28:20:@29
{
  ; In function: "bsPutUIInt32()", 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
```

07:00:09

```{
    ; In function: "mainGtU()", BB: "entry"
    %0 = zext i32 %i1 to i64
}

07:00:3c:12:03c:06:07:24:28:20:029

```{`
    ; 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 code sequences after running standard optimizers

$ opt -load=./harvest.so -std-compile-opts -harvest -details \\ -disable-output bzip2.bc

{;
  In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}

{;
  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

```plaintext
$ 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
$ opt -load=./harvest.so -std-compile-opts -harvest -details \  
disable-output bzip2.bc

{  
    ; In function: "mainGtU()", BB: "entry"
    %0 = zext i32 %i1 to i64
}

{  
    ; 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

{  ; In function: "mainGtU()", BB: "entry"
   %0 = zext i32 %i1 to i64
}

{  ; 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

{  ; In function: "mainGtU()", BB: "entry"
  %0 = zext i32 %i1 to i64
}

{  ; 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 \
   -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 $\rightarrow$ sext X  \hspace{1cm} sext = sign-extend
00:07:2c $\rightarrow$ X != 0
07:09 $\rightarrow$ zext X  \hspace{1cm} zext = zero-extend
05:07:0f $\rightarrow$ X +nsw -1  \hspace{1cm} +nsw = add with no-signed wrap
00:07:2b $\rightarrow$ X == 0
07:07:13 $\rightarrow$ X -nsw Y  \hspace{1cm} -nsw = sub with no-signed wrap
07:07:32 $\rightarrow$ X >=s Y  \hspace{1cm} >=s = signed greater than or equal
01:07:0f $\rightarrow$ X +nsw 1
06:07:0a:16 $\rightarrow$ (sext X) * power-of-2  \hspace{1cm} 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
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- Reduce to sub-expression

- Unused variables

- Rule reduction
Analysing expressions

Four modes:

- Constant folding
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- Reduce to sub-expression
  \[ ((x + z) *nsw y) /s y \rightarrow x + z \]

- Unused variables

- Rule reduction
Analysing expressions

Four modes:

- Constant folding
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- Reduce to sub-expression
  \[ (x + z) \text{nsw } y /s y \rightarrow x + z \]

- Unused variables

- Rule reduction
Analysing expressions

Four modes:

- Constant folding
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- Reduce to sub-expression
  \[ ((x + z) \ast nsw y) /s y \rightarrow x + z \]

- Unused variables
  \[ x - (x + y) \rightarrow 0 - y \]

- Rule reduction
Analysing expressions

Four modes:

- **Constant folding**
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- **Reduce to sub-expression**
  \[ ((x + z) \ast nsw y) /s y \rightarrow x + z \]

- **Unused variables**
  \[ \text{x - (x + y)} \rightarrow 0 - y \]

- **Rule reduction**

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

Four modes:

- Constant folding
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- Reduce to sub-expression
  \[ ((x + z) *\text{nsw y}) /s y \rightarrow x + z \]

- Unused variables
  \[ x - (x + y) \rightarrow 0 - y \]

- Rule reduction
  Repeatedly apply rules from a list.
  Search minimum of cost function.
Analysing expressions

Four modes:

• Constant folding
  \[ \text{zext } x <s 0 \rightarrow 0 \ (\text{i.e. false}) \]

• Reduce to sub-expression
  \[ ((x + z) *\text{nsw } y) /s y \rightarrow x + z \]

• 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
  \[ \text{zext x } \lt s 0 \rightarrow 0 \text{ (i.e. false)} \]

- 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.
Analysing expressions

Four modes:

- Constant folding
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- Reduce to sub-expression
  \[ ((x + z) \ast\text{nsw } y) /s y \rightarrow x + z \]

- Unused variables
  \[ x - (x + y) \rightarrow 0 - y \]

- Rule reduction
  
  Repeatedly apply rules from a list.
  Search minimum of cost function.
Analysing expressions

Four modes:

- Constant folding
  \[ \text{zext } x <s 0 \rightarrow 0 \text{ (i.e. false)} \]

- Reduce to sub-expression
  \[ (x + z) *\text{nsw } y /s y \rightarrow x + z \]

- Unused variables
  \[ x - (x + y) \rightarrow 0 - y \]

- Rule reduction

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

Four modes:

- **Constant folding**

  $$\text{zext } x \leq 0 \rightarrow 0 \text{ (i.e. false)}$$

- **Reduce to sub-expression**

  $$\((x + z) \; \text{nsw} \; y) \div 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.
Analysing expressions

Four modes:

- **Constant folding**
  
  \[
  \text{zext } x <s 0 \rightarrow 0 \quad (\text{i.e. false})
  \]

- **Reduce to sub-expression**
  
  \[
  ((x + z) \ast nsw y) /s y \rightarrow x + z
  \]

- **Unused variables**
  
  \[
  x - (x + y) \rightarrow 0 - y
  \]

- **Rule reduction**

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

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

• 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.

Repeat many times
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: \( A \mid (B + 1) \mid (C - 1) = 0 \)
False positives

Eg: $A \mid (B + 1) \mid (C - 1) == 0$

Mostly evaluates to “false”
False positives

Eg: \[ A \mid (B + 1) \mid (C - 1) = 0 \]

A, B and C have i8 type \(\rightarrow\) 1 / 2\(^{24}\) chance of seeing “true”
False positives

Eg: $A \mid (B + 1) \mid (C - 1) == 0$

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

Eg: $A \mid (B + 1) \mid (C - 1) == 0$

Mostly evaluates to “false”

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) >>\text{L power-of-two} ) & Z ) + \text{power-of-two} == 0 \rightarrow \text{false}
Examples

Constant folds found in “fully optimized” code:

• $( (X + Y) \gg L\text{ power-of-two } ) \& Z ) + \text{ power-of-two } == 0 \rightarrow \text{ false}

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

Constant folds found in “fully optimized” code:

• \(( (X + Y) \gg L \text{ power-of-two} ) \& Z ) + \text{power-of-two} == 0 \rightarrow \text{false}\)

• \(( (X >s Y) ? X : Y ) >s X \rightarrow \text{true}\)
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
  “max(X, Y) >= X”. Implemented several max/min folds.
Examples

Constant folds found in “fully optimized” code:

- $( ( (X + Y) \gg L \text{ power-of-two} ) \& Z ) + \text{power-of-two} == 0 \rightarrow \text{false}$

- $( (X >s Y) ? X : Y ) >=s X \rightarrow \text{true}$

- $X \text{ rem } ( Y ? X : 1 ) \rightarrow 0$

- $(Y /u X) >u Y \rightarrow \text{false}$
Examples

Constant folds found in “fully optimized” code:

- \(( (X + Y) \gg \text{L power-of-two} ) \& Z ) + \text{power-of-two} == 0 \rightarrow \text{false}\)

- \(( (X >\text{s} Y) ? X : Y ) >=\text{s} X \rightarrow \text{true}\)

- \(X \text{ rem ( Y ? X : 1 ) } \rightarrow 0\)

- \((Y /\text{u} X) >\text{u} Y \rightarrow \text{false}\)

Require reasoning about undefined behaviour
Undefined behaviour

\((X \div Y) > u X \rightarrow false\)
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

```
i8  42
```
```
i8  0
```
```
Register (X)  UDiv  undefined  ICMP_UGT
```
```
Register (Y)
```
Undefined behaviour

\[(X \div Y) > Y \rightarrow false\]

Any operation with an undef operand gets an undef result

- Avoids false negatives
- May result in subtle false positives
Reduce to subexpression

\[(X \, *\text{nsw} \, Y) \, /s \, Y \rightarrow X\]
Reduce to subexpression

\[(X *\text{nsw} Y) /s Y \rightarrow X\]

- Assign types to nodes

  Strategies: (1) Random choice; (2) All small types.
Reduce to subexpression

\[(X \times\text{ns}w\ Y) / s\ Y \rightarrow X\]

- 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

\[(X \times_{\text{nsw}} Y) /s Y \rightarrow X\]

- 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

\[(X \times \text{nsw} Y) \div s Y \rightarrow X\]

- 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

\[(X *\text{nsw} Y) /s Y \rightarrow X\]

- 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

Always same

Repeat many times
Register pressure

\[(X \times \text{nsw Y}) \div Y \rightarrow X\]  
Is this always a win?
(X *nsw Y) /s Y → X

Is this always a win?

Z = X *nsw Y

...
Register pressure

\[(X *\text{nsw } Y) /s Y \rightarrow X\]

Is this always a win?

\[Z = X *\text{nsw } Y\]

X not used again

\[\text{...}\]

\[W = Z /s Y\]

call @foo(W, Y, Z)
Register pressure

\[(X *\text{nsw } Y) /s Y \rightarrow X\]  
Is this always a win?

\[Z = X *\text{nsw } Y\]

Two registers needed (for \(Y, Z\))

\[X \text{ not used again}\]

\[W = Z /s Y\]

\[\text{call } @\text{foo}(W, Y, Z)\]
Register pressure

\[(X *\text{nsw} Y) /\text{s} Y \rightarrow X\]  
Is this always a win?

\[Z = X *\text{nsw} Y\]  
\[Z = X *\text{nsw} Y\]

Transform: \(W \rightarrow X\)

\[W = Z /\text{s} Y\]  
\[\ldots \rightarrow \ldots\]

\[\text{call @foo}(W, Y, Z)\]  
\[\ldots \text{W not computed} \ldots\]
\[\text{call @foo}(X, Y, Z)\]
Register pressure

\[(X * \text{nsw } Y) /s Y \rightarrow X\]  
Is this always a win?

Three registers needed (for X, Y, Z)  
\[Z = X * \text{nsw } Y\]

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

\[(X \, {\text{nsw}} \, Y) \, /s \, Y \rightarrow X\]

Is this always a win?

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

\[ X + nsw Z \geq s Z + nsw Y \]

Z is an “unused variable”
Unused variables

\[ X + \text{nsw} \ Z \geq \text{nsw} \ Y + \text{nsw} \ Z \]

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

\[ X + \text{nsw} \ Z \geq \text{s} \ Z + \text{nsw} \ Y \]

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:

\[ X + \text{nsw} \ Z \geq \text{s} \ Z + \text{nsw} \ Y \rightarrow X \geq \text{s} \ Y \]
Unused variables

\[ X +\text{nsw} \ Z \geq \text{s} \ Z +\text{nsw} \ Y \]

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:
\[ X +\text{nsw} \ Z \geq \text{s} \ Z +\text{nsw} \ Y \rightarrow X \geq \text{s} \ Y \]

Detect similarly to constant folding etc.
Examples

Unused variables found in “fully optimized” code:

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

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

E.g.: \((A + B) \times (C + D) \equiv B \times C + B \times D\)

\[B\] 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: \((A + B) \times (C + D) \equiv B \times C + B \times D\)

\[B \text{ is an unused variable}\]

Transforms to: \(A \times C + A \times D \equiv 0\)
Problems with unused variables

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

Eg: \((A + B) \times (C + D) == B \times C + B \times D\)

\(B\) is an unused variable

Transforms to: \(A \times C + A \times D == 0\)

Requires computing \(A \times C, A \times D\) 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  
(X & Y) | (Y & AllOnesValue)

Cost: 22  
Cost: 30
Rule reduction

Requires a list of rules, eg:

- rule \((0 \text{ And } 1) \Rightarrow (1 \text{ And } 0)\); // Commutativity
- rule \((0 \text{ And } \text{AllBitsSet}) \Leftrightarrow 0\); // AllBitsSet is And-identity
- rule \(((0 \text{ Or } 1) \text{ And } 2) \Leftrightarrow ((0 \text{ And } 2) \text{ Or } (1 \text{ And } 2))\); // Distributivity
- rule \((0 \text{ Or } \text{AllBitsSet}) \Rightarrow \text{AllBitsSet}\); // AllBitsSet is Or-annihilator.

\[(X \& Y) \mid Y \quad \text{Cost: 22}\]
\[(X \& Y) \mid (Y \& \text{AllOnesValue}) \quad \text{Cost: 30}\]
\[(X \& Y) \mid (\text{AllOnesValue} \& Y) \quad \text{Cost: 30}\]
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.

\[(X \& Y) \mid Y\]  Cost: 22
\[(X \& Y) \mid (Y \& \text{AllOnesValue})\]  Cost: 30
\[(X \& Y) \mid (\text{AllOnesValue} \& Y)\]  Cost: 30
\[(X \mid \text{AllOnesValue}) \& Y\]  Cost: 22
\[\text{AllOnesValue} \& Y\]  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.

(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
Rule reduction

Requires a list of rules, eg:

- rule $(0 \text{ And } 1) \Rightarrow (1 \text{ And } 0)$; // Commutativity
- rule $(0 \text{ And } \text{AllBitsSet}) \iff 0$; // AllBitsSet is And-identity
- rule $((0 \text{ Or } 1) \text{ And } 2) \iff ((0 \text{ And } 2) \text{ Or } (1 \text{ And } 2))$; // Distributivity
- rule $(0 \text{ Or } \text{AllBitsSet}) \Rightarrow \text{AllBitsSet}$; // AllBitsSet is Or-annihilator.

\begin{align*}
(X \& Y) \mid Y & \quad \text{Cost: 22} \\
(X \& Y) \mid (Y \& \text{AllOnesValue}) & \quad \text{Cost: 30} \\
(X \& Y) \mid (\text{AllOnesValue} \& Y) & \quad \text{Cost: 30} \\
(X \mid \text{AllOnesValue}) \& Y & \quad \text{Cost: 22} \\
\text{AllOnesValue} \& Y & \quad \text{Cost: 11} \\
Y & \quad \text{Cost: 3}
\end{align*}

\text{Time: 1 minute} \\
\text{SubExpr: 0.05 secs} \\
\text{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?

\[(\text{zext } X) + \text{power-of-two} \equiv 0 \rightarrow \text{false}\]
Rule reduction problems

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

\[(\text{zext } X) + \text{power-of-two } == 0 \rightarrow \text{false}\]
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

```c
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

```c
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

```c
((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