Algebraic Reassociation of Expressions
— With Application To Lazy Code Motion —


Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved

Citation numbers refer to entries in the EaC2e bibliography.
The Problem

Compiler front end generates expressions in arbitrary order
- Some orders (or shapes) may cost less to evaluate
- Sometimes “better” is a local property
- Sometimes “better” is a non-local property

Compiler should reorder expressions to fit their context

Old Problem
- Recognized in 1961 by Floyd
- Scarborough & Kolsky did it manually in Fortran H Enhanced
- PL/8 and HP compilers claimed to solve it, without publishing

Need an efficient & effective way to rearrange expressions

COMP 512, Rice University

Floyd, 1961
Opportunities

Common Subexpression Elimination & Constant Propagation

• Best shape for CP (probably) moves constants together
  ♦ Which operands are constant? $x \& y$, $x \& z$, or $x \& y$

• Best shape for CSE is context dependent
  ♦ Which expressions appear elsewhere? $x + y$, $x + z$, or $x + y$?

Assume that $x$, $y$, & $z$ are integers & that addition is commutative.
Opportunities

**Code Motion**

- In a loop nest, want to move loop-invariant code into the outermost loop where it does not vary

  ```
  a ← ... ; b ← ...
  do i ...
    c ← ... ; d ← ...;
  do j ...
    ... a+b ...
    ... d+b+c ...
    ... a+c+b+d...
  ```

  ```
  a ← ... ; b ← ...
  t₁ ← a+b
  do i ...
    c ← ... ; d ← ...;
    t₂ ← c+d
    t₃ ← b+t₂
    t₄ ← t₁+t₂
    do j ...
      ... t₁ ...
      ... t₃ ...
      ... t₃ ...
  ```

- In `a + b + c`, the operands may vary in different loops
- Need two or more operations in a subexpression to make distribution over two levels of loops profitable

*Briggs & Cooper proposed a ranking to address this problem*

“Best” ranking might assign different ranks to “x” in different loop nests. (⇒ SSA names?)
Opportunities

Operator Strength Reduction

subroutine dmxpy (n1, y, n2, ldm, x, m)
double precision y(*), x(*), m(ldm,*)

... 
jmin = j+16
do 60 j = jmin, n2, 16
  do 50 i = 1, n1
    y(i) = ((((((((((((((( (y(i))
      + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14)) + x(j-13)*m(i,j-13))
      + x(j-12)*m(i,j-12)) + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
      + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8)) + x(j- 7)*m(i,j- 7))
      + x(j- 6)*m(i,j- 6)) + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
      + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2)) + x(j- 1)*m(i,j- 1))
      + x(j) *m(i,j)
  50 continue
 60 continue
...
end

The largest version of the hand-optimized loop in dmxpy.

33 distinct addresses (+ i & j)

Done poorly, this loop can easily generate 33 or more distinct induction variables.
With some care (and reassociation of the address expressions),
the compiler might get that down to two or three.
Opportunities

Operator Strength Reduction

• A reference, such as \( V[i] \), translates into an address expression
  \[ @V_0 + (i\text{-}low) \times w \]

• A loop with references to \( V[i] \), \( V[i+1] \), & \( V[i-1] \) generates
  \[ @V_0 + (i\text{-}low) \times w \]
  \[ @V_0 + (i-(low-1)) \times w \]
  \[ @V_0 + (i-(low+1)) \times w \]

• **OSR** may create distinct induction variables for these expressions, or it may
  create one common induction variable

  ♦ *Matter of code shape in the expression*
  ♦ *Difference between 33 induction variables in the dmxpy loop and one or two*

• Situation gets more complex with multi-dimensional arrays
Opportunities

**Operator Strength Reduction**

- Consider references to \(A[i,j]\), \(B[i+1,j]\), and \(C[3*i,j-1]\)
  - \(\diamond @A_0 + (i \cdot \text{len}_A + j) \cdot w\)
  - \(\diamond @B_0 + ((i+1) \cdot \text{len}_B + j) \cdot w\)
  - \(\diamond @C_0 + ((3^i) \cdot \text{len}_C + j) \cdot w\)

- The diversity of address expressions may increase likelihood of generating too many induction variables in OSR

- Want to canonicalize their shape in a way that minimizes the number of induction variables.

- Problem has been known for a long time. See, for example, Markstein, Markstein & Zadeck.

Assume \(A, B, C\) may have different bounds but all have element width \(w\). Row major order.

Challenges

Expressions are small (in real code)
- In IR from human-written code, many expressions are small
  - Frequent assignment to variables breaks up computation
    - May be cognitive reasons for this style of code
  - More operations and operands means more opportunity for reassociation
- May want to transform code to build larger expressions

Complexity grows with number of operands
- Pairwise commutativity is easy to handle (think LVN)
- With 5, 6, ... operands, the number of orders is large
- Suggests a “rank & sort” methodology (Briggs)
  - Need to derive a rank scheme that achieves desired result

Any algorithmic approach to reassociation must cope with these challenges
The Running Example (from [BC 94])

**Fortran 90 Source Code**

```fortran
FUNCTION foo(y, z)
  s = 0
  x = y + z
  DO i = x, 100
    s = 1 + s + x
  ENDDO
  RETURN s
END foo
```

**Intermediate Code**

```
enter(r_y, r_z)
r_s ← 0
r_x ← r_y + r_z
r_i ← r_x
if r_i > 100 branch
    ↓
    r_1 ← r_s + 1
    r_s ← r_1 + r_x
    r_i ← r_i + 1
    if r_i ≤ 100 branch
    return r_s
```
Briggs-Cooper Approach

To improve results out of LCM

1. Reassociation
   ♦ Discover facts about global code shape
   ♦ Reorder subexpressions based on that knowledge

2. Renaming
   ♦ Use redundancy elimination to find equivalences
   ♦ Rename virtual registers to reflect equivalences, and to conform to the code shape constraints for LCM
   ♦ Encode value equality into the name space

3. LCM
   ♦ Run LCM unchanged on the result
   ♦ Performs code placement, partial redundancy elimination
   ♦ Run it anywhere, anytime, on any code

This lecture focuses on reassociation & renaming
Reassociation

Simple Idea

• Use algebraic properties to rearrange expressions
• Hard part is to choose one shape quickly

The Approach

1. Compute a rank for each expression
2. Propagate expressions forward to their uses
3. Reorder by sorting operands into rank order

The algorithm needs a guiding principle

• Order subscripts to improve code motion & constant propagation
The Intuitions

- Each expression & subexpression assigned a rank
- Loop-invariant’s rank < loop-variant’s rank
- Deeper nesting ⇒ higher rank
- Invariant in 2 loops < invariant in 1 loop
- All constants assigned the same rank
- Constants should sort together
1. **Compute Ranks**

**The Algorithm**

1. Build **pruned SSA** form & fold copies into $\phi$-functions
2. Traverse **CFG** in reverse postorder (**RPO**)
   a. Assign each block a rank number as visited
   b. Each expression in block is ranked
      i. $x$ is constant $\Rightarrow$ rank$(x)$ is 0
      ii. result of $\phi$-function has block’s RPO number
      iii. $x <op> y$ has rank max(rank$(x)$, rank$(y)$)

This numbering produces the “right” intuitive properties

*Recall that pruned SSA form only inserts phi-functions that are **LIVE** — that is, whose results are actually used.*
enter\( (r_0, r_1) \)
\[
\begin{align*}
  r_2^0 & \leftarrow 0 \\
  r_3^1 & \leftarrow r_0^1 + r_1^1 \\
  \text{if } r_3 > 100 \text{ branch} \\
  & \quad \downarrow \\
  r_4^2 & \leftarrow \phi (r_3, r_8) \\
  r_5^2 & \leftarrow \phi (r_2, r_7) \\
  r_6^2 & \leftarrow r_5 + 1 \\
  r_7^2 & \leftarrow r_6 + r_3 \\
  r_8^2 & \leftarrow r_4 + 1 \\
  \text{if } r_8 \leq 100 \text{ branch} \\
  & \quad \downarrow \\
  r_9^3 & \leftarrow \phi (r_7, r_2) \\
  & \quad \text{return } r_9
\end{align*}
\]

**Example**

The example is shown in pruned SSA form

- Use \( \phi \) functions to compute ranks
- Name space of SSA form is important

Rank computation:
- \( \phi \)'s rank & parameter rank is RPO number of its block
- Constant’s rank is 0
- \( \text{Rank}(x \ op \ y) \) is \( \text{max}(\text{rank}(x), \text{rank}(y)) \)

**Pruned SSA Form, With Computed Ranks**
2. Propagate Expressions Forward to Their Uses

The Intuition
• Copy expressions forward to their uses
• Build up large expression trees from small ones

The Implementation
• Split critical edges to create appropriate predecessors
• Replace $\phi$-functions with copies in predecessor blocks†
• Trace back from copy to build expression tree

Notes
• Forward propagation does not improve the code
• Addresses a subtle limitation in PRE and LCM (expr live across $> 1$ block)
• Eliminates some partially-dead expressions

† Based on Briggs et al.’s algorithm for out-of-SSA translation [50].
Translate Out of SSA Form

Example

PRE/LCM operate on the code in conventional (non-SSA) form

• Split critical edges
• Use any out-of-SSA translation technique
• Chain of copies preserves name space for forward propagation

\[
\begin{align*}
\text{enter}(r_0, r_1) \\
r_2 &\leftarrow 0 \\
r_3 &\leftarrow r_0 + r_1 \\
\text{if } r_3 > 100 \text{ branch} \\
r_4 &\leftarrow r_3 \\
r_5 &\leftarrow r_2 \\
r_6 &\leftarrow r_5 + 1 \\
r_7 &\leftarrow r_6 + r_3 \\
r_8 &\leftarrow r_4 + 1 \\
\text{if } r_8 \leq 100 \text{ branch} \\
r_9 &\leftarrow r_7 \\
\text{return } r_9
\end{align*}
\]
After Forward Propagation

```
enter(r_0, r_1)
\( r_3 \leftarrow r_0 + r_1 \)
if \( r_3 > 100 \) branch

\( r_2 \leftarrow 0 \)
\( r_3 \leftarrow r_0 + r_1 \)
\( r_4 \leftarrow r_3 \)
\( r_5 \leftarrow r_2 \)
\( r_7 \leftarrow 1 + r_0 + r_1 + r_5 \)
```

```
\( r_8 \leftarrow r_4 + 1 \)
\( r_9 \leftarrow r_7 \)
```

```
\( r_8 \leftarrow r_4 + 1 \)
if \( r_8 \leq 100 \) branch

after
```

Example

Replace uses with the defining expressions
- Move immediate values
- Builds up larger expressions
- Removes partially dead expressions
- Technical issue with \textbf{PRE} — \textit{expr live across >1 block}

```
\( r_2 \leftarrow 0 \)
\( r_9 \leftarrow r_2 \)

return \( r_9 \)
3. Reorder Operands

The Intuition

• Rank shows how far LCM can move an expression
• Sort subexpressions into ascending rank order
• Allows LCM to move subexpression each as far as possible

The Implementation

• Rewrite \( x - y + z \) as \( x + (-y) + z \) [Frailey 1970]
• Sort operands of associative ops by rank
• Distribute operations where both legal & profitable

Distribution

• Sometimes pays off, sometimes does not
• We explored one strategy: low rank \( x \) over high-rank +

Room for more work on this issue
Rewrite the code

- Rank expressions
- Sort operands of associative operations by their rank
- Convert back to binary operators

In example, $r_7$ was already in sorted order.

### After Reordering

```plaintext
enter(r_0, r_1)

r_3 ← r_0 + r_1
if r_3 > 100 branch

r_2 ← 0
r_3 ← r_0 + r_1
r_4 ← r_3
r_5 ← r_2

r_a ← r_0 + 1
r_b ← r_a + r_1
r_7 ← r_b + r_5
r_8 ← r_4 + 1
r_4 ← r_8
r_5 ← r_7

r_8 ← r_4 + 1
if r_8 ≤ 100 branch

r_c ← r_0 + 1
r_d ← r_c + r_1
r_7 ← r_d + r_5
r_9 ← r_7

r_2 ← 0
r_9 ← r_2

return r_9
```

### Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>r_0</td>
<td>1</td>
</tr>
<tr>
<td>r_1</td>
<td>1</td>
</tr>
<tr>
<td>r_2</td>
<td>0</td>
</tr>
<tr>
<td>r_3</td>
<td>1</td>
</tr>
<tr>
<td>r_4</td>
<td>2</td>
</tr>
<tr>
<td>r_5</td>
<td>2</td>
</tr>
<tr>
<td>r_6</td>
<td>2</td>
</tr>
<tr>
<td>r_7</td>
<td>2</td>
</tr>
<tr>
<td>r_8</td>
<td>2</td>
</tr>
<tr>
<td>r_9</td>
<td>3</td>
</tr>
</tbody>
</table>
Making It Work with Lazy Code Motion

What have we done to the code, so far?

• Rewritten every expression based on global ranks
  ♦ and local concerns of constant propagation ...
• Tailored order of evaluation for LCM
• Broken the name space that LCM needs
  ♦ so, we cannot possibly run LCM

Undoing the damage

• Must systematically rename values to create LCM name space
• Can improve on the original name space, if we try
  ♦ Choose names in a way that encodes values
• Need a global renaming phase
Renaming

The intuition
• Use Alpern et al.’s partitioning method
• Rename every value to expose congruences found by AWZ

The implementation
• $x, y \in$ same congruence class $\Rightarrow$ use same name
• Use hash table to regenerate consistent names
• Reserve variable names & insert copies

Notes
• Clever implementation might eliminate some stores
• Variables become obvious from conflicting definitions

Any renaming scheme that builds the right name space will work. We will see AWZ in a couple of lectures.
Now, reconstruct the PRE name space

- Use some global value numbering technique (AWZ, Simpson)
- Encode value identity in lexical identity

After renaming, compiler can run PRE/LCM

Example

After Renaming

\[
\begin{align*}
\text{enter}(r_0, r_1) \\
r_3 &\leftarrow r_0 + r_1 \\
\text{if } r_3 > 100 \text{ branch} \\
r_2 &\leftarrow 0 \\
r_3 &\leftarrow r_0 + r_1 \\
r_4 &\leftarrow r_3 \\
r_5 &\leftarrow r_2 \\
r_6 &\leftarrow r_0 + 1 \\
r_7 &\leftarrow r_6 + r_1 \\
r_8 &\leftarrow r_7 + r_5 \\
r_9 &\leftarrow r_4 + 1 \\
r_4 &\leftarrow r_9 \\
r_5 &\leftarrow r_8 \\
r_9 &\leftarrow r_4 + 1 \\
\text{if } r_9 \leq 100 \text{ branch} \\
r_6 &\leftarrow r_0 + 1 \\
r_7 &\leftarrow r_6 + r_1 \\
r_8 &\leftarrow r_7 + r_5 \\
r_{10} &\leftarrow r_8 \\
r_2 &\leftarrow 0 \\
r_{10} &\leftarrow r_2 \\
\text{return } r_{10}
\end{align*}
\]
Results

What do we gain from all this manipulation?

• Can run LCM (or PRE) at any point in the optimizer
  ♦ Can reconstruct the name space
  ♦ Makes results independent of choices made in front end

• More effective redundancy elimination
  ♦ Measured with PRE (not LCM)
  ♦ Reductions of up to 40% in total operations (over PRE)

• Sometimes, code runs more slowly
  ♦ Forward propagation moves code into loop
  ♦ PRE cannot move it back out of the loop

Stronger methods can remove them, but this is a minor effect and ...
PRE/LCM move code out of the loop

- Landing pad grows
- Loop body shrinks
  → *In this case, the split block in the back edge*

Role of PRE is placement

Name space trick makes redundancy aspect more effective, too.

- Example does not highlight that effect

After PRE

\[
\begin{align*}
  \text{enter}(r_0, r_1) \\
  r_3 &\leftarrow r_0 + r_1 \\
  \text{if } r_3 > 100 \text{ branch} \\
  r_2 &\leftarrow 0 \\
  r_4 &\leftarrow r_3 \\
  r_5 &\leftarrow r_2 \\
  r_6 &\leftarrow r_0 + 1 \\
  r_7 &\leftarrow r_6 + r_1 \\
  r_8 &\leftarrow r_7 + r_5 \\
  r_9 &\leftarrow r_4 + 1 \\
  \text{if } r_9 \leq 100 \text{ branch} \\
  r_8 &\leftarrow r_7 + r_5 \\
  r_{10} &\leftarrow r_8 \\
  r_2 &\leftarrow 0 \\
  r_{10} &\leftarrow r_2 \\
  \text{return } r_{10}
\end{align*}
\]
After coalescing

Chaitin-Briggs coalescing cleans up the copies
• Note the clean, small loop body
• Of course, Briggs & Cooper recommend aggressive coalescing
→ Despite what other authors say
• Result is code that you might write yourself
Other Issues

**Code Size**
- Forward propagation has the potential for exponential growth in size
- Measured results
  - Average was 1.27x; maximum was 2.488; 1 of 50 was ≥ 2
- Stronger LCM methods avoid this problem by cloning, so ...

**Distribution**
- Can destroy common subexpressions
- Has choice of shapes & can pick less profitable one

**Interaction with other transformations**
- Shouldn’t turn multiplies into shifts until later
- Reassociation should let OSR find fewer induction variables
Issues Related to Lazy Code Motion

Lazy code motion makes significant improvements

• Sometimes, it misses opportunities
• Can only find textual subexpressions
• Array subscripts are a particular concern

LCM has its limitations

• Requires strict naming scheme
  ♦ Can only run it once, early in optimization
  ♦ Other optimizations will destroy name space
• Relies on lexical identity (not value identity)

Would like version of LCM that fixes these problems
Should be fast, easy to implement, & simple to teach ...

⇒ And, as long as I am wishing, it should operate directly on SSA
What is Left in Reassociation?

This approach works well for code motion, but ...

• The Briggs scheme may not extend well to other problems
  ♦ For example, it maximizes code motion but may eliminate some redundancies
  ♦ Simple rank order is not enough; need consistent orders

• Not clear how to extend it for strength reduction
  ♦ Want to reorganize in a way that minimizes the number of induction variables
    (demand for registers) and updates (arithmetic operations)
  ♦ May need to solve an offline problem to choose best shape

• Eckhardt took a more general approach
  ♦ Reassociation to help scalar replacement & cross-iteration redundancies
  ♦ Much more involved approach
  ♦ We will see this algorithm in the next lecture