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
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 appears 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 ...
    ... b+c+d ...
    ... a+b+c+d...
```

• 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

```
a ← ... ; b ← ...
t_1 ← a+b
do i ...
c ← ... ; d ← ...;
t_2 ← c+d
t_3 ← b+t_2
t_4 ← t_1 + t_2
do j ...
    ... t_1 ...
    ... t_3 ...
    ... t_3 ...
```

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

Operator Strength Reduction

- Transformed code has lots of address arithmetic
- With wrong shape, it has 9 or 10 induction variables, each needing a register
- Another version of this loop has 33 or more potential induction variables

Critical loop nest from dmxpy in the Linpack library

```
do 60 j = 1, n2
  do 50 i = 1 to n1
    y(i) = y(i) + x(j) * m(i,j)
  50 continue
60 continue

One of several hand-optimized versions of the loop
```
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)
Opportunities

Operator Strength Reduction

- A reference, such as $V[i]$, translates into an address expression
  $$@V_0 + (i\text{-}\text{low}) \times w$$
- A loop with references to $V[i]$, $V[i+1]$, & $V[i-1]$ generates
  $$@V_0 + (i\text{-}\text{low}) \times w$$
  $$@V_0 + (i-(\text{low}-1)) \times w$$
  $$@V_0 + (i-(\text{low}+1)) \times w$$
- 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]
  > @A_0 + (i * \text{len}_A + j) * w
  > @B_0 + ((i+1) * \text{len}_B + j) * w
  > @C_0 + ((3*i) * \text{len}_A + j) * w

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

• 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.

**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*
  - Reassociation has more opportunity with more operands
- May want to transform code to build larger expressions

Complexity grows with arity of operators

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

Any algorithm must cope with these challenges
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

Need a guiding principle

♦ Order subscripts to improve code motion (& constant propagation)
Computing Ranks

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

\[
\begin{align*}
  & + \\
  & + \\
  & + \\
  & + \\
  + & \\
  + & \\
  + & \\
  + & \\
  & \\
\end{align*}
\]

inner loop
middle loop
outer loop

c_1, c_2
constants 1st
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 $\max(\text{rank}(x), \text{rank}(y))$

This numbering produces the “right” intuitive properties
Forward Propagation

The intuition

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

The implementation

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

Notes

• Forward propagation is not an improvement
• Addresses limitation in PRE (expr live across > 1 block)
• Eliminates some partially-dead expressions

Split them here, not during ranking!
Reordering Expressions

The intuition
- Rank shows how far LCM can move an expression
- Sort subexpressions into ascending rank order
- Lets LCM move 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
Making It Work with Lazy Code Motion

What have we done?

• 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
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 ...
Other Issues

Code Size

- Forward propagation is potentially exponential in space
- 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
Next Class

This approach works well for code motion, but …

- May not extend well to other problems, such as CSE
  > Simple rank order is not enough; need consistent orders

- Not clear how to extend it for strength reduction
  > Want a complex reorganization
  > May need to solve an offline problem to choose best shape

- Next class, we will try to generalize this work, based on Eckhardt’s paper and comments in Markstein, Markstein, & Zadeck.