Operator Strength Reduction

— Generalities and the Cocke-Kennedy Algorithm —

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.
Operator Strength Reduction

Consider the following simple loop

```plaintext
sum = 0
do i = 1 to 100
    sum = sum + a(i)
end do
```

What’s wrong with this picture?

- Takes 3 operations to compute the address of `a(i)`
- On some machines, integer multiply is slow
Operator Strength Reduction

Consider the value sequences taken on by the various registers

\[
\begin{align*}
\text{loadl} & \quad 0 & \Rightarrow r_{\text{sum}} \\
\text{loadl} & \quad 1 & \Rightarrow r_i \\
\text{loadl} & \quad 100 & \Rightarrow r_{100} \\
\text{loop:} & \quad \text{subl} \quad r_i, 1 & \Rightarrow r_1 \\
& \quad \text{mul} \quad r_1, 4 & \Rightarrow r_2 \\
& \quad \text{addl} \quad r_2, @a & \Rightarrow r_3 \\
& \quad \text{load} \quad r_3 & \Rightarrow r_4 \\
& \quad \text{add} \quad r_4, r_{\text{sum}} & \Rightarrow r_{\text{sum}} \\
& \quad \text{addl} \quad r_i, 1 & \Rightarrow r_i \\
& \quad \text{cmp} \_{\text{LT}} \quad r_i, r_{100} & \Rightarrow r_5 \\
& \quad \text{cbr} \quad r_5 & \rightarrow \text{loop, exit}
\end{align*}
\]

\[r_{\text{sum}} = \bot\]
\[r_i = \{1, 2, 3, 4, \ldots\}\]
\[r_{100} = \{100\}\]
\[r_1 = \{0, 1, 2, 3, \ldots\}\]
\[r_2 = \{0, 4, 8, 12, \ldots\}\]
\[r_3 = \{@a, @a+4, @a+8, @a+12, \ldots\}\]
\[r_4 = \bot\]
\[r_5 = \bot\]

\(r_i, r_1, r_2, \text{ and } r_3\) take on predictable sequences of values

- \(r_1\) and \(r_2\) are intermediate values, while \(r_3\) and \(r_i\) play important roles
- We can compute them cheaply & directly
Operator Strength Reduction

Computing \( r_3 \) directly yields the following code

\[
\begin{align*}
\text{loadl} & \ 0 \quad \Rightarrow \ r_{\text{sum}} \\
\text{loadl} & \ 1 \quad \Rightarrow \ r_i \\
\text{loadl} & \ 100 \quad \Rightarrow \ r_{100} \\
\text{loadl} & \ @a \quad \Rightarrow \ r_3 \\
\text{load} & \ r_3 \quad \Rightarrow \ r_4 \\
\text{addl} & \ r_3, \ 4 \quad \Rightarrow \ r_3 \\
\text{add} & \ r_4, r_{\text{sum}} \quad \Rightarrow \ r_{\text{sum}} \\
\text{addl} & \ r_i, 1 \quad \Rightarrow \ r_i \\
\text{cmp\_LT} & \ r_i, r_{100} \quad \Rightarrow \ r_5 \\
\text{cbr} & \ r_5 \quad \rightarrow \ \text{loop,exit}
\end{align*}
\]

\( r_3 = \{ @a, @a+4, @a+8, @a+12, \ldots \} \)

Still, we can do better ...

- From 8 operations in the loop to 6 operations
- No expensive multiply, just cheap adds
Operator Strength Reduction

Shifting the loop’s exit test from $r_i$ to $r_3$ yields

```
loadl 0    \rightarrow r_{\text{sum}}
loadl @a   \rightarrow r_3
addl r_3,396 \rightarrow r_{\text{lim}}

\text{loop:}
load r_3    \rightarrow r_4
addl r_3,4  \rightarrow r_3
add r_4,r_{\text{sum}} \rightarrow r_{\text{sum}}
cmp_{LT} r_3,r_{\text{lim}} \rightarrow r_5
cbr r_5 \rightarrow \text{loop,exit}
```

- Address computation went from -,+,* to +
- Exit test went from +, cmp to cmp
- Loop body went from 8 operations to 5 operations
  - Got rid of that expensive multiply, too

Pretty good speedup on most machines
37.5% of ops in the loop, even if mult takes one cycle
Not redundant or invariant
Operator Strength Reduction

And, as an aside, unrolling also helps

```
loadl 0 ⇒ r_{sum}
loadl @a ⇒ r_3
addl r_3,396 ⇒ r_{lim}
```

loop:
```
load r_3 ⇒ r_4
addl r_3, 4 ⇒ r_3
add r_4, r_{sum} ⇒ r_{sum}
load r_3 ⇒ r_4
addl r_3, 4 ⇒ r_3
add r_4, r_{sum} ⇒ r_{sum}
```

```
exit: ...
```

```
load r_3 ⇒ r_4
addl r_3, 4 ⇒ r_3
add r_4, r_{sum} ⇒ r_{sum}
cmp_LT r_3, r_{lim} ⇒ r_5
cbr r_5 ⇒ loop,exit
```

Now, 8 operations for 2 iterations, or 50% of the operations and a smaller percentage of the cycles (due to elimination of multiplies)
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

```
50 continue
60 continue

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

49 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
  \[ @V_0 + (i\text{-low}) \times w \]
- A loop with references to $V[i]$, $V[i+1]$, & $V[i-1]$ generates
  \[
  @V_0 + (i\text{-low}) \times w \\
  @V_0 + (i\text{-low-1}) \times w \\
  @V_0 + (i\text{-low+1}) \times w
  \]
- 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

**Assumptions:**
- $V$ is declared $V[\text{low:high}]$.
- Elements are $w$ bytes wide.
- Constants have been folded.
Operator Strength Reduction

Definition

*Operator Strength Reduction* is a transformation that replaces a strong (expensive) operator with a weaker (cheaper) operator.

**Strong form**

- Replace series of multiplies with adds

**Weak form**

- Replace single multiply with shifts and/or adds

**The Problem**

- It's easy to see the transformation
- It's somewhat harder to automate the process

See, for example, Lefevre’s paper on the class web site.
Operator Strength Reduction

The Cocke-Kennedy Algorithm

To explain strength reduction, we will begin with the multi-pass Cocke-Kennedy algorithm.

Assumptions

• Intermediate representation is low-level, ILOC-like code
• Have already built a control-flow graph (CFG)
• Have found either “natural loops” or “strongly connected regions” (SCRs)
• Have added a landing pad to each region

Definitions

• A region constant (RC) is a variable whose value is unchanged in the SCR
• An induction variable (IV) is a variable whose value changes in the SCR only by operations that increment or decrement it by an RC or an IV.

Operator Strength Reduction

The Cocke-Kennedy Algorithm

The Problem
• Easy to apply transformation by hand *
• Difficult to automate the process

The Big Picture
• Find induction variables and their uses
• Introduce a new induction variable tailored to each use
  ♦ Requires both an initialization & appropriate updates
• Shift remaining uses from original induction variables to new ones
• Eliminate the original induction variables from the code
Operator Strength Reduction

The Cocke-Kennedy Algorithm

The Problem
• Easy to apply transformation by hand
• Difficult to automate the process

The Big Picture
• Find induction variables and their uses
• Introduce a new induction variable tailored to each use
  ♦ Requires both an initialization & appropriate updates
• Shift remaining uses from original induction variables to new ones
• Eliminate the original induction variables from the code
Operator Strength Reduction

The Cocke-Kennedy Algorithm

The Problem
• Easy to apply transformation by hand
• Difficult to automate the process

The Algorithmic Plan
1. Find loops in the control-flow graph
2. Find region constants for those loops
3. Find induction variables
4. Find operations that are candidates to be reduced
5. Find all the values that affect the uses in candidate operations
6. Perform the actual replacement
7. Rewrite end-of-loop tests onto newly introduced induction variables
8. Dead-code elimination

A large number of passes over the IR

“Linear function test replacement”
Operator Strength Reduction

The Cocke-Kennedy Algorithm

Step 1: Find loops in the CFG as SCR

Apply Tarjan’s strongly-connected region finder to the CFG


→ This algorithm is also the basis for next lecture, on the Vick-Simpson OSR algorithm, so you should read the paper if you haven’t already done so

Step 2: Find region constants in the loops

Assume that we have performed loop-invariant code motion first.¹

Any value that is used in the SCR and not defined in the SCR is in RC

For each SCR, build a set of names that are defined (DEF) and a set of names that are used (USE). (linear pass over blocks in the SCR)

Then, RC is just (USE - DEF) or (USE ∩ NOT(DEF))

¹ If not, the test for region constant must also consider a variable that is assigned the same value along different paths through the SCR. In practice, it is easier to perform something like LCM first.
Operator Strength Reduction

The Cocke-Kennedy Algorithm

Step 3: Find Induction Variables
Assumes SCR and RCs

\[ IV \leftarrow \emptyset \]

for each op \( o \) (\( t \leftarrow o_1 \text{ op } o_2 \)) in the SCR do
  if \( op \in \{ \text{ADD, SUB, NEG, COPY} \} \)
    \[ IV \leftarrow IV \cup \{ t \} \]
  \[ \text{changed} \leftarrow \text{true} \]
while (\( \text{changed} \))
  \[ \text{changed} \leftarrow \text{false} \]
  for each operation \( o \) where \( t \in IV \)
    if \( o_1 \notin (IV \cup \text{RC}) \) or \( o_2 \notin (IV \cup \text{RC}) \)
      remove \( t \) from \( IV \)
      \[ \text{changed} \leftarrow \text{true} \]

Simple fixed-point algorithm Applied to each SCR

An induction variable is only updated by an add, subtract, copy, or negation involving induction variables and region constants
Operator Strength Reduction

The Cocke-Kennedy Algorithm

**Step 4:** Find operations that are candidates to be reduced

Assumes **SCRs**, **IV**, and **RC**

```
CANDIDATES ← ∅
for each op o (t ← o₁ op o₂) do
  if op is a MULTIPLY then
    if (o₁ ∈ IV and o₂ ∈ RC) or (o₁ ∈ RC and o₂ ∈ IV)
    then CANDIDATES ← CANDIDATES ∪ {o}
```

For exposition, a candidate is a multiply than can be reduced.

**CANDIDATES** contains all multiplies that involve exactly one **RC** and one **IV**

To expand the algorithm to other reductions, expand the test for candidates
Operator Strength Reduction

The Cocke-Kennedy Algorithm

Naming

• Create a new name for each unique candidate expression \((\text{hash them})\)
• Insert an initialization for each new name in the appropriate landing pad
• After each assignment to \(i \in \text{IV}\), insert an update to the affected new names

<table>
<thead>
<tr>
<th>Reducing (a \leftarrow i \times c)</th>
<th>Operation to Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td></td>
</tr>
<tr>
<td>(i \leftarrow k)</td>
<td>(t_{ixc} \leftarrow t_{kxc})</td>
</tr>
<tr>
<td>(i \leftarrow -k)</td>
<td>(t_{ixc} \leftarrow -t_{kxc})</td>
</tr>
<tr>
<td>(i \leftarrow j + k)</td>
<td>(t_{ixc} \leftarrow t_{jxc} + t_{kxc})</td>
</tr>
<tr>
<td>(i \leftarrow j - k)</td>
<td>(t_{ixc} \leftarrow t_{jxc} - t_{kxc})</td>
</tr>
</tbody>
</table>

We tested for these four ops on admission to IV

To deal with all of these cases, we build, for \(i \in \text{IV}\), a set \(\text{AFFECT}(i)\) that contains every \(j \in \text{IV} \cup \text{RC}\) that can affect the value of \(i\).
Operator Strength Reduction

The Cocke-Kennedy Algorithm

**Step 5:** Computing AFFECT Sets

Assumes SCR and IV are already available

```latex
for each \( i \in IV \)
    \[ \text{AFFECT}(i) \leftarrow \{i\} \]

for each op \( o \) (\( t \leftarrow o_1 \text{ op } o_2 \)) where \( t \in IV \) do
    \[ \text{AFFECT}(t) \leftarrow \text{AFFECT}(t) \cup \{o_1,o_2\} \]

\( \text{changed} \leftarrow \text{true} \)

while (\( \text{changed} \))
    \( \text{changed} \leftarrow \text{false} \)

for each \( i \in IV \)
    \[ \text{NEW} \leftarrow \bigcup_{o \in \text{AFFECT}(i) \cap IV} \text{AFFECT}(o) \]

if \( \text{AFFECT}(i) \cap \text{NEW} \neq \emptyset \)
    then \( \text{changed} = \text{true} \)

\[ \text{AFFECT}(i) \leftarrow \text{AFFECT}(i) \cup \text{NEW} \]
```

Transitive closure

Applied to each SCR

Comp 512, Spring 2015
Operator Strength Reduction

The Cocke-Kennedy Algorithm

Step 6: Replacement
Assumes all the sets from steps 1 through 5

This step is the heart of the transformation.

\texttt{CLIST}(y) \text{ is the set of constant multipliers for } y

---

Recall that each \texttt{CANDIDATE} has the form $(t \leftarrow i \times c, i \in \text{IV}, c \in \text{RC})$
Operator Strength Reduction

The Cocke-Kennedy Algorithm

Step 7: Linear function-test replacement

for each operation $o$ in an SCR
  if $o$ is a conditional branch $(i \ op \ k \ \Rightarrow \ label)$ with $i \in \text{IV}$ & $k \in \text{RC}$
    then
      select some $c \in \text{CLIST}(i)$ /* $t_{ixc}$ already exists, from Step 6 */
      if neither $t_{kxc}$ or $t_{cxk}$ exist then
        insert $t_{cxk}$ into the hash table of names
        insert $t_{cxk} \leftarrow c \times k$ in the landing pad
        replace the conditional branch with
        $t_{ixc} \ op \ t_{cxk} \ \Rightarrow \ label$

  

Applied to each SCR
Operator Strength Reduction

The Cocke-Kennedy Algorithm

Step 8: Dead Code Elimination

• This algorithm leaves behind a mess
  ◆ Original induction variables and their updates are still in the code
  ◆ Shotgun approach to creating reduced induction variables leaves more behind
    → Not all of the $t_{axb}$ are actually used

• For the result to be an improvement, it needs some clean up

• Apply a standard dead-code elimination technique
  ◆ **DEAD** followed by **CLEAN** will do the job
  ◆ Other algorithms work, too
Operator Strength Reduction

The Cocke-Kennedy Algorithm

The Problem

• Easy to apply transformation by hand
• Difficult to automate the process

The Algorithmic Plan

1. Find loops in the control-flow graph  
   Entire CFG
2. Find region constants for those loops  
   # ops
3. Find induction variables  
   # ops
4. Find operations that are candidates to be reduced  
   # ops
5. Find all the values that affect the uses in candidate operations  
   # values\(^3\)
6. Perform the actual replacement  
   # candidates
7. Rewrite end-of-loop tests onto newly introduced induction variables  
   # ops
8. Dead-code elimination
Operator Strength Reduction

Next class

• Vick-Simpson OSR algorithm

• Operates over static single assignment form rather than the CFG and individual ops

• Properties of SSA let us simplify the algorithm and reduce its costs