Comp 512
Rice University
Spring 2015

Operator Strength Reduction

— Generalities and the Cocke-Kennedy Algorithm —

Citation numbers refer to entries in the EaC2e bibliography.

Operator Strength Reduction

Consider the following simple loop

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

```
        loadI    0         ⇒ r_sum
        loadI    1         ⇒ r_i
        loadI    100       ⇒ r_100
loop:   subI     r_i,1     ⇒ r_1
        multI    r_1,4     ⇒ r_2
        addI     r_2,@a    ⇒ r_3
        load     r_3       ⇒ r_4
        add      r_4,r_sum ⇒ r_sum
        addI     r_i,1     ⇒ r_i
        cmp_LT   r_i,r_100 ⇒ r_5
        cbr      r_5       → loop, exit
exit: ...
```

address of a(i)

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

```
        loadI    0         ⇒ r_sum
        loadI    1         ⇒ r_i
        loadI    100       ⇒ r_100
loop:   subI     r_i,1     ⇒ r_1
        multI    r_1,4     ⇒ r_2
        addI     r_2,@a    ⇒ r_3
        load     r_3       ⇒ r_4
        add      r_4,r_sum ⇒ r_sum
        addI     r_i,1     ⇒ r_i
        cmp_LT   r_i,r_100 ⇒ r_5
        cbr      r_5       → loop,exit
exit: ...
```

$r_{sum} = \bot$

r_i $= \{ 1, 2, 3, 4, … \}$

$r_{100} = \{ 100 \}$

r_1 $= \{ 0, 1, 2, 3, … \}$

r_2 $= \{ 0, 4, 8, 12, … \}$

r_3 $= \{ @a, @a+4, @a+8, @a+12, … \}$

$r_4 = \bot$

$r_5 = \bot$

r_i, r_1, r_2, 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

```
        loadI   0        ⇒ r_sum
        loadI   1        ⇒ r_i
        loadI   100      ⇒ r_100
        loadI   @a       ⇒ r_3
loop:   load    r_3      ⇒ r_4
        addI    r_3, 4   ⇒ r_3
        add     r_4,r_sum ⇒ r_sum
        addI    r_i,1    ⇒ r_i
        cmp_LT  r_i,r_100 ⇒ r_5
        cbr     r_5      → loop,exit
exit: ...
```

address of a(i)

r_3 = { @a, @a+4, @a+8, @a+12, … }

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

```
         loadI    0          ⇒ r_sum
         loadI    @a         ⇒ r_3
         addI     r_3,396    ⇒ r_lim
loop:    load     r_3        ⇒ r_4
         addI     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
exit: ...
```

r_3 = { @a, @a+4, @a+8, @a+12, … }

- Address computation went from -,+,* to +
- Exit test went from +, <u>cmp</u> to <u>cmp</u>
- 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

```
        loadI    0         => r_sum
        loadI    @a        => r_3
        addI     r_3,396   => r_lim
loop:   load     r_3       => r_4       }
        addI     r_3, 4    => r_3       }  Copy # 1 of loop body
        add      r_4,r_sum => r_sum     }
        load     r_3       => r_4       }
        addI     r_3, 4    => r_3       }  Copy # 2 of loop body
        add      r_4,r_sum => r_sum     }
        cmp_LT   r_3,r_lim => r_5       }  Shared test & branch
        cbr      r_5       -> loop,exit }
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

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

Critical loop nest from
dmxpy in the Linpack library

- 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

```
      do 60 j = 1, n2
         nextra = mod(n1,4)
         if (nextra .ge. 1) then
            do 49 i, nextra, 1
               y(i) = y(i) + x(j) * m(i,j)
49          continue

         do 50 i = nextra+1, n1, 4
            y(i) = y(i) + x(j)     * m(i,j)
                        + x(j+1) * m(i,j+1)
                        + x(j+2) * m(i,j+2)
                        + x(j+3) * m(i,j+3)
50          continue
60    continue
```

One of several hand-optimized
versions of the loop

Opportunities

Operator Strength Reduction

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

```
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
```

33 distinct addresses (+ i & j)

Opportunities

Operator Strength Reduction

- A reference, such as V[i], translates into an address expression

  @V_0 + (i-low) * w

- A loop with references to V[i], V[i+1], & V[i-1] generates

  @V_0 + (i-low) * w

  @V_0 + (i-(low-1)) * w

  @V_0 + (i-(low+1) * w

Assumptions:
V is declared V[low:high].
Elements are w bytes wide.
Constants have been folded.

- 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

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

See, for example, Lefevre's paper on the class web site.

The Problem

- Its easy to <u>see</u> the transformation
- Its somewhat harder to automate the process

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" (**SCR**s)
- 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**.

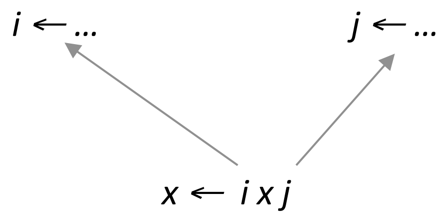
J. Cocke and K. Kennedy, "An Algorithm for Reduction of Operator Strength," *Communications f the ACM* 20(11), Nov. 1977, pages 850 – 856.

Operator Strength Reduction

The Cocke-Kennedy Algorithm

The Problem

- Easy to apply transformation by hand *
- Difficult to automate the process



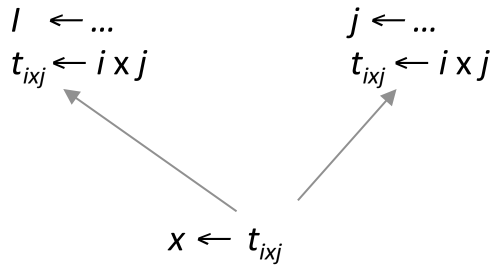
Before

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



$I \leftarrow \dots$

$t_{ixj} \leftarrow i \times j$

$j \leftarrow \dots$

$t_{ixj} \leftarrow i \times j$

$x \leftarrow t_{ixj}$

After

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

A large number of passes over the IR

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

"Linear function test replacement"

Operator Strength Reduction

The Cocke-Kennedy Algorithm

Step 1: Find *loops* in the **CFG** as **SCR**s

Apply Tarjan's strongly-connected region finder to the **CFG**

See R.E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal of Computing,* 1(2), 1972 *pages 146 - 160*

→ *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

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

The Cocke-Kennedy Algorithm

Step 3: Find Induction Variables

Assumes **SCR**s and **RC**s

```
IV ← Ø
for each op o (t ← $o_1$ op $o_2$)  in the SCR do
    if op ∈ { ADD, SUB, NEG, COPY}
      IV ← IV ∪ { t }
changed ← true
while (changed)
    changed ← false
    for each operation o where $t \in$ IV
      if $o_1 \notin$ (IV ∪ RC) or $o_2 \notin$ (IV ∪ RC)
        remove t from IV
        changed ← true
```

Simple fixed-point algorithm
Applied to each **SCR**

Operator Strength Reduction

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

The Cocke-Kennedy Algorithm

Step 4: Find operations that are candidates to be reduced

Assumes **SCR**s, **IV**, and **RC**

```
CANDIDATES ← Ø
for each op o (t ← $o_1$ op o2) do
    if op is a MULTIPLY then
      if ($o_1 \in$ IV and $o_2 \in$ RC) or ($o_1 \in$ RC and $o_2 \in$ IV)
       then CANDIDATES ← CANDIDATES ∪ {o}
```

Applied to each **SCR**

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 (*hash them*)
- Insert an initialization for each new name in the appropriate landing pad
- After each assignment to $i \in$ IV, insert an update to the affected new names

Reducing $a \leftarrow i \times c$	
Assignment	**Operation to Insert**
$i \leftarrow k$	$t_{ixc} \leftarrow t_{kxc}$
$i \leftarrow -k$	$t_{ixc} \leftarrow -t_{kxc}$
$i \leftarrow j + k$	$t_{ixc} \leftarrow t_{jxc} + t_{kxc}$
$i \leftarrow j - k$	$t_{ixc} \leftarrow t_{jxc} - t_{kxc}$

We tested for these four ops on admission to IV

To deal with all of these cases, we build, for $i \in$ **IV**, a set **AFFECT**(i) that contains every $j \in$ **IV** ∪ **RC** that can affect the value of i.

Operator Strength Reduction

The Cocke-Kennedy Algorithm

Step 5: Computing **AFFECT** Sets

Assumes **SCR**s and **IV** are already available

```
for each $i \in$ IV
    AFFECT($i$) $\leftarrow \{i\}$
for each op o ($t \leftarrow o_1$ op $o_2$) where $t \in$ IV do
    AFFECT($t$) $\leftarrow$ AFFECT($t$) $\cup \{o_1, o_2\}$
changed $\leftarrow$ true
while (changed)
    changed $\leftarrow$ false
    for each $i \in$ IV
        NEW $\leftarrow \bigcup_{o \in \text{AFFECT}(i) \cap \text{IV}}$ AFFECT($o$)
        if AFFECT($i$) $\cap$ NEW $\neq \emptyset$
            then changed = true
        AFFECT($i$) $\leftarrow$ AFFECT($i$) $\cup$ NEW
```

Transitive closure

Applied to each **SCR**

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.

CLIST(y) is the set of constant multipliers for y

```
/* build up a set of multipliers for each variable */
for each $x \in$ IV $\cup$ RC
    CLIST($x$) $\leftarrow \emptyset$
for each op $p \in$ CANDIDATES ($t \leftarrow i \times c,\ i \in$ IV, $c \in$ RC) do
    for each $y \in$ AFFECT($t$)
        CLIST($y$) $\leftarrow$ CLIST($y$) $\cup \{c\}$
for each $y \in$ (IV $\cup$ RC) with CLIST($y$) $\neq \emptyset$ do
    for each $c \in$ CLIST($y$)    /* initialize reduced IV */
        $T(y,c) \leftarrow$ a new temporary name
        insert "$T(y,c) \leftarrow y \times c$" in landing pad
/* insert updates for each reduced IV */
for each op $p$ ($t \leftarrow o_1$ op $o_2$) with $t \in$ IV and CLIST($t$) $\neq \emptyset$
    for each $c \in$ CLIST($t$)
        insert after $p$ "$T(t,c) \leftarrow T(o_1,c)$ op $T(o_2,c)$"
/* replace the candidate operations */
for each operation $p \in$ CANDIDATES do
    replace $p$ with the operation $t \leftarrow T(x,c)$
```

Applied to each **SCR**

Recall that each **CANDIDATE** has the form ($t \leftarrow i \times c,\ i \in$ **IV**, $c \in$ **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 $i \in$ IV & $k \in$ RC
      then
        select some $c \in$ CLIST($i$)  /* $t_{i \times c}$ 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 $t_{a \times b}$ 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
  - See K. Cooper, L.T. Simpson, and C. Vick, “Operator Strength Reduction,” *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23(5), Sept 2001, pages 603-625.
- 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