Operator Strength Reduction

— the Vick-Simpson algorithm —
Operator Strength Reduction

The Algorithmic Plan

• Capitalize on the properties of SSA form
• Find SCCs in the SSA graph
  ♦ Each non-trivial SCC might be an IV
    → Test the SCC as it is discovered, so we need a cheap test
    → Discover RCs relative to the SCC with a cheap test
  ♦ Reduce operations on the fly
    → Recognize candidates for reduction with a cheap test
    → Use structural information (e.g., DOM) to place new computations
  ♦ Accumulate information for linear function test replacement
• Use results of prior transformations
  ♦ Assume constant propagation and code motion
  ♦ Use DOM information from SSA construction

Cheap means \(O(1)\) if possible
Operator Strength Reduction

Consider the following simple loop

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

COMP 512, Spring 2015
Operator Strength Reduction

Consider the value sequences taken on by the various registers

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

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

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

- \(r_1\) 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

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

Still, we can do better ...

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

Review from last lecture

Shifting the loop’s exit test from \( r_i \) to \( r_3 \) yields

\[
\begin{align*}
\text{loadl} & \quad 0 & \Rightarrow & \quad r_{\text{sum}} \\
\text{loadl} & \quad @a & \Rightarrow & \quad r_3 \\
\text{addl} & \quad r_3,396 & \Rightarrow & \quad r_{\text{lim}} \\
\text{loop:} & \quad \text{load} & \quad r_3 & \Rightarrow & \quad r_4 \\
& \quad \text{addl} & \quad r_3,4 & \Rightarrow & \quad r_3 \\
& \quad \text{add} & \quad r_4,r_{\text{sum}} & \Rightarrow & \quad r_{\text{sum}} \\
& \quad \text{cmp}_{\text{LT}} & \quad r_3,r_{\text{lim}} & \Rightarrow & \quad r_5 \\
& \quad \text{cbr} & \quad r_5 & \Rightarrow & \quad \text{loop,exit}
\end{align*}
\]

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

- 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

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

COMP 512, Spring 2015
Operator Strength Reduction

Assumptions for the OSR Algorithm

- Low-level IR, such as ILOC, converted into SSA form
- Constant propagation and loop-invariant code motion have been applied

Terminology

- A *strongly connected component* (SCC) of a directed graph is a region where a path exists from each node to every other node
- A *region constant* (RC) of an SCC is an SCC-invariant value
- An *induction variable* (IV) of an SCC is one whose value only changes in the SCC when operations increment it by an RC or an IV, or when it is the destination of a COPY from another IV
- A *candidate* for reduction is an operation “$x \leftarrow y \ast z$” where $y, z \in IV \cup RC$ and either $y \in IV$ or $z \in IV$

Intuitively, we are interested in induction variables that are updated in a cyclic fashion. The self-dependence creates the pattern of repetition from which the *strong form* of strength reduction derives its benefits.

The classic papers, e.g., Cocke-Kennedy, and Allen-Cocke-Kennedy, define IVs this way. The OSR algorithm only finds IVs that form a cycle in the SSA graph. The practical results are equivalent.
### Operator Strength Reduction

#### Our example in semi-pruned SSA Form

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadl</td>
<td>0</td>
<td>rs0</td>
</tr>
<tr>
<td>loadl</td>
<td>1</td>
<td>ri0</td>
</tr>
<tr>
<td>loadl</td>
<td>100</td>
<td>r100</td>
</tr>
<tr>
<td>loop: phi</td>
<td>rs0, rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>phi</td>
<td>ri0, ri2</td>
<td>ri1</td>
</tr>
<tr>
<td>subl</td>
<td>ri1, 1</td>
<td>r1</td>
</tr>
<tr>
<td>multl</td>
<td>r1, 4</td>
<td>r2</td>
</tr>
<tr>
<td>addl</td>
<td>r2, @a</td>
<td>r3</td>
</tr>
<tr>
<td>load</td>
<td>r3</td>
<td>r4</td>
</tr>
<tr>
<td>add</td>
<td>r4, rs1</td>
<td>rs2</td>
</tr>
<tr>
<td>addl</td>
<td>ri1, 1</td>
<td>ri2</td>
</tr>
<tr>
<td>cmp_LT</td>
<td>ri2, r100</td>
<td>r5</td>
</tr>
<tr>
<td>cbr</td>
<td>r5</td>
<td>loop, exit</td>
</tr>
</tbody>
</table>

#### SSA Form as a Graph

![ SSA Form as a Graph Diagram ]

**Short-lived temporary values**

**exit:** ...
Operator Strength Reduction

**SSA form as a graph**
- Each IV is an SCC
- Not every SCC is an IV
- $x \in \text{RC}$ if $x$ is a constant or its definition is in a block that dominates the entry of the SCC
- Compute DOM & RPO numbers for the SSA graph

**Using SSA as a graph simplifies OSR**
- Find IVs with SCC finder
- Test operations in SCC
- Constant time test for RC
  - $\geq$ Constant or test with DOM

Prior algorithms used multiple passes over the IR, inner loop to outer loop..
Operator Strength Reduction

Finding SCCs
• Use Tarjan’s algorithm
• Well-understood method
• Takes $O(N+E)$ time

Useful property
• SCC popped only after all its external operands have been popped
• Reduce the SCCs as popped
  ♦ $|\text{SCC}| > 1 \Rightarrow$ if its an IV, mark it
  ♦ $|\text{SCC}| = 1 \Rightarrow$ try to reduce it
• We only need to add one line

DFS(n)
  \[
  \begin{align*}
  &n.\text{DFSnum} \leftarrow \text{nextDFSnum++} \\
  &n.\text{visited} \leftarrow \text{true} \\
  &n.\text{low} \leftarrow n.\text{DFSnum} \\
  &\text{push}(n) \\
  &\text{for each } o \in \{ \text{operands of } n \} \\
  &\quad \text{if } o.\text{visited} = \text{false} \text{ then} \\
  &\quad\quad \text{DFS}(o) \\
  &\quad &n.\text{low} \leftarrow \min(n.\text{low}, o.\text{low}) \\
  &\quad \text{if } o.\text{DFSnum} < n.\text{DFSnum} \text{ and} \\
  &\quad\quad o \in \text{stack} \text{ then} \\
  &\quad\quad &n.\text{low} \leftarrow \min(n.\text{low}, o.\text{DFSnum}) \\
  &\quad \text{if } n.\text{low} = n.\text{DFSnum} \text{ then} \\
  &\quad &\text{sc}c \leftarrow \{ \} \\
  &\quad \text{until } x = n \text{ do} \\
  &\quad &x \leftarrow \text{pop}() \\
  &\quad &\text{sc}c \leftarrow \text{sc}c \cup \{ x \} \\
  &\quad \text{Process}(\text{sc}c) \\
\end{align*}
\]
Operator Strength Reduction

What should Process($r$) do?

- If $r$ is one node, try to reduce it
- If $r$ is a collection of nodes
  - Check to see if it is an IV
  - If so, reduce it & any ops that use it
  - If not, try to reduce the ops in $r$

```
Process($r$)
if $r$ has only one member, $n$ then
    if $n$ has the form $x \leftarrow \text{IV} \times \text{RC}$, $x \leftarrow \text{RC} \times \text{IV}$,
    $x \leftarrow \text{IV} \pm \text{RC}$, or $x \leftarrow \text{RC} + \text{IV}$ then
        Replace($n$,IV,RC)
    else $n$.header $\leftarrow$ NULL
else ClassifyIV($r$)
```

Let’s tackle the easier problem first – ClassifyIV()
Operator Strength Reduction

ClassifyIV(r)
header ← first(r)
for each node n ∈ r
  if header → RPOnum > n.block → RPOnum then
    header ← n.block
  for each node n ∈ r
    if n.op is not one of { Ø, +, -, COPY } then
      r is not an induction variable
    else
      for each o ∈ { operands of n }
        if o ∉ r and not RCon(o, header) then
          r is not an induction variable
      if r is an induction variable then
        for each node n ∈ r
          n.header ← header
        else
          for each node n ∈ r
            if n has the form x ← IV x RC, x ← RC x IV, x ← IV ± RC, or x ← RC + IV then
              Replace(n, IV, RC)
            else
              n.header ← NULL

Rcon(o, header)
if o.op is loadI /* constant */
  then return true
else if o.block >> header
  then return true
else return false

>> means “strictly dominates”
Operator Strength Reduction

/* replace n with a COPY */
Replace(n,iv,rc)
  result ← Reduce(n.op,iv,rc)
  Replace n with COPY from result
  n.header ← iv.header

/* create new IV & return its name */
Reduce(op,iv,rc)
  result ← search(op,iv,rc)
  if result is not found then
    result ← a new name
    add(op,iv,rc,result)
  newDef ← copyDef(iv,result)
  for each operand o of newDef
    if o.header = iv.header then
      replace o with Reduce(op,o,rc)
    else if (opcode = x or newDef.op = Ø) then
      replace o with Apply(op,o,rc)
  return result

Replace rewrites op n with a COPY operation from its reduced counterpart. It calls Reduce to create that counterpart, if necessary.

search and add deal with the hash table

Returns name of op applied to iv and rc

Clones the definition

Args defined outside SCC ⇒ initial value or the increment
/* replace n with a COPY */
Replace(n,iv,rc)  
result ← Reduce(n.op,iv,rc)  
Replace n with COPY from result
n.header ← iv.header

/* create new IV & return its name */
Reduce(op,iv,rc)  
result ← search(op,iv,rc)  
if result is not found then
result ← a new name
add(op,iv,rc,result)
newDef ← copyDef(iv,result)
for each operand o of newDef
if o.header = iv.header then
replace o with Reduce(op,o,rc)
else if (opcode = x or newDef.op = ∅) then
replace o with Apply(op,o,rc)
return result

The Big Picture

• Reduce() creates a new IV, with appropriate range & increment
• In the example, \( r_3 \) would range from @a to @a+396, with an increment of 4
• Replace takes a candidate operation and rewrites it with a COPY from the new IV. It uses Reduce to create the IV.

Net effect: replace \((i-1)*4+@a\) with a COPY from some new IV that runs from @a to @a+396 & increments by 4 on each iteration
Operator Strength Reduction

/* insert a new operation */
Apply(op, arg1, arg2)
  result ← search(op, arg1, arg2)
  if result is not found then
    if (arg1.header ≠ NULL /* ∈ IV */) & RCon(arg2, arg1.header) then
      result ← Reduce(op, arg1, arg2)
    else if arg2.header ≠ NULL /* ∈ IV */ & RCon(arg1, arg2.header) then
      result ← Reduce(op, arg2, arg1)
    else
      result ← a new name
      add(op, arg1, arg2, result)
  Choose a location to insert op
  Try constant folding
  Create newOp at the location
  newOp.header ← NULL
return result

The Big Picture

• Apply takes an op & 2 args and inserts the corresponding operation into the code (if it isn’t already there).
• Uses >> on arg1 & arg2 to find a location
  — does not use landing pad
  — may insert farther away
• Tries to reduce the operation
• Tries to simplify the operation
Example

And, most of this is dead ...
Example

The transformation to perform this simplification is called *linear function test replacement*.

This would be dead, except for the comparison & branch. Need to reformulate them on $r_{a8}$.
Linear Function Test Replacement

Each time a new, reduced IV is created

• Add an LFTR edge from old IV to new IV
• Label edge with the opcode and RC of the reduction
• Walk the LFTR edges to accumulate the transformation
• Use transformation to rewrite the test
Example

Follow the edges to find the right IV and to accumulate the transformation

\[(100 - 1) \times 4 + @a = 396 + @a\]
Example

Now, this code is dead.

Not dead!
Example

```
loadl 0 ⇒ r_sum
loadl @a ⇒ r3
addl r3,396 ⇒ r_lim
addl r3,4 ⇒ r3
add r4,r_sum ⇒ r_sum
cmp_LT r3,r_lim ⇒ r5
cbr r5 ⇒ loop, exit
```

exit: ...

And, we’re done ..
Complexity

What does OSR + LFTR cost?

• LFTR takes time proportional to the length of the LFTR edge chain that it follows

• What about OSR?
  ✦ Each cycle it creates clones every node in the cycle
  ✦ How bad can that get?
Worst-case Example

\[ \text{i} \leftarrow 0; \quad t_1 \leftarrow 0; \quad t_2 \leftarrow 0; \ldots; \quad t_n \leftarrow 0 \]

while(\( P_0 \))

if (\( P_1 \)) then

\[ i \leftarrow i + 1 \]
\[ k \leftarrow i \times c_1 \]

if (\( P_2 \)) then

\[ i \leftarrow i + 2 \]
\[ k \leftarrow i \times c_2 \]

\ldots

if (\( P_n \)) then

\[ i \leftarrow i + n \]
\[ k \leftarrow i \times c_n \]

end

This code requires a quadratic number of updates
Complexity

**What does OSR + LFTR cost?**

- **LFTR** takes time proportional to the length of the **LFTR** edge chain that it follows.
- What about **OSR**?
  - Each cycle it creates clones every node in the cycle.
  - How bad can that get?
  - In the worst case, **OSR** must insert a number of updates that is quadratic in the size of the original code.
  - Any strength reduction algorithm must insert the same set of updates, if it is to reduce the computation.
    - If it doesn’t, it misses the opportunity.
  - Complexity is **part of the problem**, not part of the solution.
- **OSR** is as fast (asymptotically) as others.
  - Constant factor faster than Cocke-Kennedy or Allen-Cocke-Kennedy.