Optimization

Last Time

- Loop Invariant Code Motion
- Induction Variable Recognition

Today

- More Loop Optimizations
  - Strength Reduction
  - Linear Test Replacement
  - Loop Unrolling
  - Scalar Replacement
Induction Variables Review

Definitions

1. A basic induction variable is a variable $J$
   - whose only definition within the loop is an assignment of the form $J := J \pm c$, where $c$ is loop invariant).

2. A mutual induction variable $I$ may be
   - defined once within the loop, and its value is a linear function of some other induction variable $I'$ such that
     $$ I = c_1 \times I' \pm c_2 $$
   or
     $$ I = I' / c_1 \pm c_2. $$
   where $c_1, c_2$ are loop invariant.

3. A family of induction variables includes a basic induction variable and any mutual induction variables.
Strength Reduction

Philosophy: *Replace an expensive instruction, multiply, with a cheaper one, addition.*

- Applied to uses of an induction variable (or uses of a family of induction variables)
- Opportunity: array indexing
- Why: slow or non-existent integer multiply

**Example**

\[
J = 0 \quad \text{for } (J = 0; J<100; J++)
\]
\[
A(J) = 0
\]

L2: if (J>=100) GOTO L1
   I := 4 * J + &A
   *I := 0
   J := J + 1
   GOTO L2

L1:

In *Linpackd*, on the IBM RT/PC, strength reduction led to an improvement of about 15 percent.

### Strength Reduction Algorithm

**Algorithm**

Let $I$ be an induction variable in the family of basic induction variable $J$, such that: $I = c_1 \cdot J + c_2$

- Create new variable, $I'$
- Initialize in preheader, $I' = c_1 \cdot J + c_2$
- Track value of $J$. After $J := J + c_3$, add
  $I' := I' + (c_1 \cdot c_3)$
- Replace definition of $I$ with $I := I'$

**Key point**

- $c_1$, $c_2$ and $c_3$ are constant or loop invariant, so the computation can be moved out of the loop or folded at compile time
- reduces number of multiplies executed at run time
**Strength Reduction Example**

**Original Code**

\[
\begin{align*}
J & := 0 \\
L2: & \text{ if } (J \geq 100) \text{ GOTO L1} \\
& I := 4 \times J + &A \\
& *I := 0 \\
& J := J + 1 \\
& \text{GOTO L2} \\
L1: & \\
\end{align*}
\]

**After Strength Reduction**

\[
\begin{align*}
J & := 0 \\
I' & := 4 \times J + &A \\
L2: & \text{ if } (J \geq 100) \text{ GOTO L1} \\
& I := I' \\
& *I := 0 \\
& J := J + 1 \\
& I' := I' + (4 \times 1) \\
& \text{GOTO L2} \\
L1: & \\
\end{align*}
\]
Candidates for Strength Reduction

• *IV* multiplied by an invariant

\[
i = 2 \quad i = 2 \\
i.50 = i \times 50 \quad i.50 = i.50 + 50
\]

\[\Rightarrow\]

\[
i = i + 1 \quad i = i + 1 \\
... i \times 50 \quad ... i.50
\]

\[
candidates = \emptyset \\
\text{for each statement } s \\
\quad \text{if (opcode = MUL and one operand in IV} \\
\quad \quad \text{and the other is invariant) \\
\quad \quad \text{add } s \text{ to } candidates
\]

end for

• Polynomials - *IV* multiplied by different *IV*
• *IV* multiplied by itself
• *IV* modulo a constant
• addition of induction variables
while candidates not empty
    remove s from candidates
    if s is “e = i * c + a” replace it with “e = i.c + a”
    else let i.c = s.c, i = IV in rhs
    for each reaching definition point to i
        if i.c assigned at Def point continue
        if Def is outside of loop
            insert “i.c = i * c” in landing pad
        else if Def is “i = j”
            insert “i.c = j * c”
            add “i.c = j * c” to candidates
        else if Def is “i = i + a”
            insert “i.c = i.c + a * c”
            add “i = i + a” with s.c = i.c to candidates
        else if Def is “i = j + a”
            insert “i.c = j * c + a * c”
            add “i.c = j * c + a * c” to candidates
        endif
    end for
end while
Examples

\[ i = 2 \]

\[ \text{for } i < k \implies \text{for } i < k \]
\[ i = i + 1 \]
\[ l = i \times 50 \]

\[ i = 2 \]
\[ i.50 = i \times 50 \]

\[ i = i + 1 \]
\[ i.50 = i.50 + 50 \]
\[ l = i.50 \]

\[ j = 2 \]

\[ \text{for } j < k \implies \text{for } i < k \]
\[ e = j \times 3 \]

\[ i = j + 1 \]

\[ l = i \times 50 \]

\[ j = j + 1 \]
Strength Reduction Details

• What happens if two induction variables I1 and I2 are in the family of the same basic induction variable J with the same constants c1 and c2?

• When might this happen in real code?

\[
i = 0
\]

\[
\text{l1: ...}
\]

\[
\text{do \ i = 1, n}
\]

\[
\text{A(i) = B(i) + B(i+1)}
\]

\[
i = i + 1
\]

\[
j = i + 1
\]

\[
t1 = 4*i + &A
\]

\[
t2 = 4*i + &B
\]

\[
t3 = 4*j + &B
\]

\[
\text{...}
\]
Linear Test Replacement

Eliminate the induction variable altogether

- the loop test often is the last use of a basic induction variable after strength reduction
- fewer instructions, fewer live ranges

Algorithm:

- If the only use of a IV is the loop test and its own increment
- and if the test is always computed (i.e., there is only one exit from the loop)
- Then replace the test with an equivalent one

say test is “i compare k”,

if ∃ IV named i.c,
    replace test with “i.c compare c*k”

- How does the sign of c affect the test?
Example

\[ i = 2 \]
\[ i.50 = i \times 50 \]

\textbf{for} \ i < k \quad \implies \quad \textbf{for} \ i.50 < k \times 50

\[ i = i + 1 \]
\[ i.50 = i.50 + 50 \]
\[ \ldots \ i.50 \]
\[ \ldots \ i.50 \]
### Reduction of operator strength

<table>
<thead>
<tr>
<th>Taxonomy — Reduction of Operator Strength</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Machine Independent</strong></td>
</tr>
<tr>
<td>remove redundancy</td>
</tr>
<tr>
<td>move evaluation</td>
</tr>
<tr>
<td>specialize</td>
</tr>
<tr>
<td>remove useless code</td>
</tr>
<tr>
<td>expose opportunities</td>
</tr>
<tr>
<td><strong>Machine Dependent</strong></td>
</tr>
<tr>
<td>costly op→cheap op</td>
</tr>
<tr>
<td>hide latency</td>
</tr>
<tr>
<td>use powerful op</td>
</tr>
</tbody>
</table>
Loop Unrolling

To reduce loop overhead, we can unroll loops.

```plaintext
do i = 1 to 100 by 1
   a(i) = a(i+1) + b(i)
end

⇒

do i = 1 to 100 by 4
   a(i) = a(i+1) + b(i)
   a(i+1) = a(i+2) + b(i+1)
   a(i+2) = a(i+3) + b(i+2)
   a(i+3) = a(i+4) + b(i+3)
end
```

Unrolled by a factor of four

Advantages

- execute fewer total instructions
- more fodder for cse, strength reduction, instruction scheduling, etc.
- move consecutive accesses closer together

Disadvantages

- code size increase
- may confuse register allocator and instruction scheduler
Scalar Replacement

**Problem:** register allocators never keep \( a(i) \) in a register

**Idea:** trick the allocator

1. locate patterns of consistent re-use
2. replace load with a copy into temporary
3. replace store with copy from temporary
4. may need copies at end of loop (re-use spans > 1 iteration)

**Benefits**

- decrease number of loads and stores
- keep re-used values in registers
- often see improvements by factors of \( 2 \times \) to \( 3 \times \)

Scalar Replacement

\[
\begin{align*}
\text{do } i &= 1, n \\
\text{do } j &= 1, n \\
a(i) &= a(i) + b(j) \\
\text{enddo} \\
\text{endo}
enddo
\end{align*}
\]

Scalar replacement exposes the reuse of \( a(i) \)

- traditional scalar analysis is inadequate
- use dependence analysis to understand array references

\[
\begin{align*}
\text{do } i &= 1, n \\
a(i) &= a(i) + b(j) \\
\text{enddo}
enddo
\end{align*}
\]

\[
\begin{align*}
\text{do } i &= 1, n \\
t &= a(i) \\
\text{do } j &= 1, n \\
t &= t + b(j) \\
\text{enddo} \\
a(i) &= t \\
\text{enddo}
enddo
\end{align*}
\]

\[
\begin{align*}
\text{do } i &= 1, n \\
\text{do } j &= 1, n \\
a(i) &= a(i) + b(j) \\
\text{enddo} \\
\text{endo}
enddo
\end{align*}
\]
Next Time

Data Flow Analysis