Advanced Compilers

CMPSCI 710

Spring 2003

Basic Loop Optimizations

Emery Berger

University of Massachusetts, Amherst

Topics

- Last time
  - Optimizations using SSA form
    - Constant propagation & dead code elimination
    - Loop invariant code motion
- This time
  - Loop optimizations
    - Induction variable
    - Linear test replacement
    - Loop unrolling
    - Scalar replacement

Easy Detection of Loop Induction Variables

- Pattern match & check:
  - Search for "i = i + b" in loop
  - i is induction variable if no other assignment to i in loop
- Pros & Cons:
  - \+ Easy!
  - \- Does not catch all loop induction variables e.g., "i = a * c + 2"

Taxonomy of Induction Variables

- *basic* induction variable:
  - only definition in loop is assignment $j = j \pm c$, where c is loop invariant
- *mutual* induction variable:
  - definition is linear function of other induction variable i':
    - $i = c1 * i' \pm c2$
    - $i = i' / c1 \pm c2$
- *family* of basic induction variable j:
  - set of induction variables i such that i is always assigned linear function of j

Strength Reduction

- Replace "expensive" op by "cheaper" one
  - E.g., replace multiply by addition
- Apply to induction variable families
  - Especially: array indexing

```
    j = 0;
L2: if (j >= 100) goto l1;
    i = &a + 4 * j;
    *i = 0;
    j = j + 1;
    goto L2;
L1:
```

→

```
for (j = 0; j < 100; j++)
  a[j] = 0;
```


Strength Reduction Algorithm

- Let i be induction variable in the family of basic induction variable j:
  - $i = c1 * j + c2$
- Create new variable i'
- Initialize in pre-header: $i' = c1*j + c2$
- Track value of j: after $j = j + c3$, add $i' = i' + (c1 * c3)$

Strength Reduction Example

```
     j = 0;
L2:  if (j >= 100) goto l1;
     i = &a + 4 * j;
     *i = 0;
     j = j + 1;
     goto L2;
L1:
```

→

```
     j = 0;
     i' = &a + 4 * j;
L2:  if (j >= 100) goto l1;
     i = i';
     *i = 0;
     j = j + 1;
     i' = i' + (4 * 1);
     goto L2;
L1:
```


Candidates for Strength Reduction

- Induction variable IV multiplied by invariant

```
i = 2;

i = i + 1;
a[i] = i * 50;
```

→

```
i = 2;
i_50 = i * 50;

i = i + 1;
i_50 = i_50 + 50;
a[i] = i_50;
```

- Recursively:
  - IV * IV, IV mod constant, IV + IV

Strength Reduction Algorithm

```
insert all basic induction variables
    into candidates list
while candidates not empty
    remove s from candidates
    if s is "t = i*c + a"
      replace with "t = i.c + a"
    else
      let i.c = s.c, i = IV in rhs
    for each reaching def pointing to i
      if i.c assigned at def point, continue
      elseif def is "i = j"
          insert "i.c = j*c"
          add "i.c = j*c" to candidates
      elseif def is "i = i + a"
          insert "i.c = i.c + a*c"
          add "i = i + a" with s.c = i.c to candidates
      elseif def is "i = j + a"
          insert "i.c = j*c + a*c"
          add "i.c = j*c + a*c" to candidates
```


Strength Reduction Examples

```
i = 2;

while i < k
    i = i + 1;
    l = i * 50;
```

→

```
i = 2;
i_50 = i * 50;
while i < k
    i = i + 1;
    i_50 = i_50 + 50;
    l = i_50;
```

```
j = 2;

while j < k

    e = j * 3;

    i = j + 1;

    l = i * 50;
```

→

```
j = 2;

while j < k

      i = j + 1;
```

- *basic* induction variable:
  - only definition in loop is assignment $j = j \pm c$, where c is loop invariant
- *mutual* induction variable:
  - definition is linear function of other induction variable i':
    - $i = c1 * i' \pm c2$
    - $i = i' / c1 \pm c2$
- *family* of basic induction variable j:
  - set of induction variables i such that i is always assigned linear function of j

Linear Test Replacement

- Eliminates induction variable!
  - After strength reduction, loop test is often last use of induction variable
- Algorithm:
  - If only use of IV is loop test and its own increment, and test is always computed
    - i.e., only one exit from loop
  - Replace test with equivalent one:
    - E.g., "i comp k" ⇒ "i_50 comp k*50"

Linear Test Replacement Example

```
i = 2;
i_50 = i * 50;
while i < k
    i = i + 1;
    i_50 = i_50 + 50;
    ... i_50
```

→

```
i = 2;
i_50 = i * 50;
while i_50 < k * 50

    i_50 = i_50 + 50;
    ... i_50
```

Loop Unrolling

- To reduce loop overhead, we can *unroll* loops

```
for (i = 1; i < 100; i++) {
  a[i] = a[i+1] + b[i];
}
```

→

```
for (i = 1; i < 100; i += 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];
}
```

- Advantages:
  - Execute fewer total instructions
  - More fodder for common subexpression elimination, strength reduction, etc.
  - Move consecutive access closer together
- Disadvantages:
  - Code bloat
  - Still updating through memory

Scalar Replacement

- Problem: register allocators never keep a[i] in register
- Idea: trick allocator
  - Locate patterns of consistent reuse
  - Replace load with copy into temporary
  - Replace store with copy from temporary
  - May need copies at end of loop
    - E.g., when reuse spans > 1 iteration
- Advantages:
  - Decreases number of loads and stores
  - Keeps reused values in registers
  - Big performance impact (2x, 3x!)

Scalar Replacement Example

```
for (i = 1; i < n; i++) {
  for (j = 1; j < n; j++) {
    a[i] = a[i] + b[j];
  }
}
```

→

```
for (i = 1; i < 100; i++) {
  t = a[i];
  for (j = 1; j < 100; j++) {
    t = t + b[j];
  }
  a[i] = t;
}
```

- Scalar replacement exposes the reuse of a[i]
  - Traditional scalar analysis – inadequate
  - Use *dependence analysis* to understand array references (later)

Next Time

- Common Subexpression Elimination
- Read ACDI:
  - Ch. 12, pp. 343-355
  - Ch. 13, pp. 378-396