More Optimizations

Last Time

• Loop invariant code motion
• Loop induction variables

Today

• Global Common Subexpression Elimination
• Value Numbering

Common Subexpression Elimination (Example)

Given \( A[i][j] = A[i][j] + 1 \), and assuming

1. row-major order
2. each array element is 4 bytes, and
3. \( R \) rows in the array

will yield the following 3-address intermediate code:

\[
\begin{align*}
t0 &= A \\
t1 &= R * 4 \\
t2 &= t1 * i \\
t3 &= t0 + t2 \\
t4 &= j * 4 \\
t5 &= t3 + t4 \\
t6 &= [t5] \\
t7 &= t6 + 1 \\
t8 &= A \\
t9 &= R * 4 \\
t10 &= t8 * i \\
t11 &= t8 + t10 \\
t12 &= j * 4 \\
t13 &= t11 + t12 \\
[t13] &= t7
\end{align*}
\]

What are the common sub-expressions?
Global Common Subexpression Elimination

The Goal

to find common subexpressions whose range spans basic blocks, and eliminate unnecessary re-evaluations

Safety

- available expressions (AVAIL) proves that value is current
- transformation gives value the right name

Profitability

- don't add evaluations to any path
- add some copy instructions
  ⇒ copy is as cheap as any operator
  ⇒ may shrink, may stretch live ranges

Available expressions

For a block \( b \)

- \( \text{AVAIL}(b) \) the set of expressions available on entry to \( b \).
- \( \text{NKILL}(b) \) the set of expressions not killed in \( b \).
- \( \text{DEF}(b) \) the set of expressions defined in \( b \) and not subsequently killed in \( b \).

\[
\text{AVAIL}(b) = \bigcap_{x \in \text{pred}(b)} \left( \text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)) \right)
\]

\( \text{IN}(b) = \)

\( \text{OUT}(b) = \)
What expressions are available at the end of this basic block?

- \( t_0 = A \)
- \( t_1 = R \times 4 \)
- \( t_2 = t_1 \times i \)
- \( t_3 = t_0 + t_2 \)
- \( t_4 = j \times 4 \)
- \( t_5 = t_3 + t_4 \)
- \( t_6 = [t_5] \)

How do we compute AVAIL for a basic block?

Returning to our example. How can we detect that \( t_2 \) and \( r_10 \) compute the same value?

- \( t_0 = A \)
- \( t_1 = R \times 4 \)
- \( t_2 = t_1 \times i \)
- \( t_3 = t_0 + t_2 \)
- \( t_4 = j \times 4 \)
- \( t_5 = t_3 + t_4 \)
- \( t_6 = [t_5] \)
- \( t_7 = t_6 + 1 \)
- \( t_8 = A \)
- \( t_9 = R \times 4 \)
- \( t_{10} = t_8 \times i \)
- \( t_{11} = t_8 + t_{10} \)
- \( t_{12} = j \times 4 \)
- \( t_{13} = t_{11} + t_{12} \)
- \([t_{13}] = t_7\)
Value Numbering

Value numbering computes available expressions (AVAIL or DEF) for a basic block.

Input

- basic block of tuples (statements)
- symbol table

Output

- improved basic block (cse, constant folding)
- table of available expressions
- table of constant values

† An expression is available at point \( p \) if it is defined along each path leading to \( p \) and none of its constituent values has been subsequently redefined.

Key Notions

- each variable, each expression, and each constant is assigned a unique number, its value number
  - same number \( \Rightarrow \) same value
  - based solely on information from within the block
- if an expression’s value is available (already computed), we should not recompute it
- constants denoted in both SYMBOLS and tuples
Value numbering

Principle data structures

**CODE**
- array of tuples
- Fields: result, operator (op), operands (lhs, rhs)

**SYMBOLS**
- hash table keyed by variable name
- Fields: name, val, isConstant

**AVAIL**
- hash table keyed by (lhs, op, rhs)
- Fields: lhsVal, op, rhsVal, resultVal, tuple

**CONSTANTS**
- table to hold funky machine specific values
- important in cross-compilation
- Fields: val, bits

---

Value numbering

result = lhs op rhs

for $i \leftarrow 1$ to $n$
  $r \leftarrow$ value number for $rhs[i]$
  $l \leftarrow$ value number for $lhs[i]$
  if $op[i]$ is a store then
    SYMBOLS[$result[i]$].val $\leftarrow r$
  if $r$ is constant then
    SYMBOLS[$lhs[i]$].isConstant $\leftarrow$ true
  else /* an expression */
    if $l$ is constant then replace $lhs[i] with constant
    if $r$ is constant then replace $rhs[i]$ with constant
    if $l$ and $r$ are both constant then
      create CONSTANTS($l, op[i], r$)
      CONSTANTS($l, op[i], r$).bits $\leftarrow$ eval($l op[i] r$)
      CONSTANTS($l, op[i], r$).val $\leftarrow$ new value number
      replace ($lhs op[i] rhs$) with “constant ($l op[i] r$)”
    else if ($l, op[i], r$) $\in$ AVAIL then
      replace ($lhs op[i] rhs$) with “AVAIL($l, op[i], r$).result[j]”
    else create AVAIL($l, op[i], r$)
      AVAIL($l, op[i], r$).val $\leftarrow$ new value number
    endif
  SYMBOLS[$result[i]$].val $\leftarrow$ value number of expression
  endif
for all AVAIL($l, op[j], r$)
  if $result[j].val = l$ or $r$ or $result[j].val$, ($j < i$)
    remove ($l, op[j], r$) from AVAIL
endfor
endfor
### Example

<table>
<thead>
<tr>
<th>Tuples</th>
<th>Source</th>
<th>Avail</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ← C4</td>
<td>a ← 4</td>
<td></td>
</tr>
<tr>
<td>T2 ← i × j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3 ← T2 + C5</td>
<td>k ← i × j + 5</td>
<td></td>
</tr>
<tr>
<td>T5 ← C5 × a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6 ← T5 × k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l ← T6</td>
<td>l ← 5 × a × k</td>
<td></td>
</tr>
<tr>
<td>m ← i</td>
<td>m ← i</td>
<td></td>
</tr>
<tr>
<td>T9 ← m × j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 ← i × a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T11 ← T9 + T10</td>
<td>b ←</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m × j + i × a</td>
<td></td>
</tr>
</tbody>
</table>

### Example

\[
A \leftarrow X + Y \quad A_0 \leftarrow X_0 + Y_0 \\
B \leftarrow X + Y \quad B_0 \leftarrow X_0 + Y_0 \\
A \leftarrow 1 \quad A_1 \leftarrow 1 \\
C \leftarrow A + Y \quad C_0 \leftarrow A_1 + Y_0 \\
B \leftarrow A \quad B_1 \leftarrow A_1 \\
C \leftarrow 3 \quad C_1 \leftarrow 3 \\
D \leftarrow B + Y \quad D_0 \leftarrow B_1 + Y_0
\]

**Original**  
**SSA Form**
Global Common Subexpression Elimination

Algorithm:

1. \( \forall \) block \( b \), compute \( \text{DEF}(b) \) and \( \text{NKILL}(b) \)
2. \( \forall \) block \( b \), compute \( \text{AVAIL}(b) \)
3. \( \forall \) block \( b \), value number the block using \( \text{AVAIL} \)
4. replace expressions in \( \text{AVAIL}(b) \) with references

Computing \( \text{DEF}(b) \) and \( \text{NKILL}(b) \)

- use value numbering, or
- do it by inspection

Specifics

1. \( \forall \) block \( b \), compute \( \text{DEF}(b) \) and \( \text{KILL}(b) \)
2. assign each \( e \in \text{AVAIL}(b) \) a name \((\text{small integer})\)
3. \( \forall \) variables \( v \), initialize \( \text{MAP}(v) \) to empty
4. \( \forall \) expressions \( e \), \( \forall v \in e \), add name to \( \text{MAP}(v) \)
5. \( \forall \) block \( b \), \( \text{NKILL}(b) = \bigcup_{v \notin \text{KILL}(b)} \text{MAP}(v) \)

A bit-vector set implementation works fine for \( \text{AVAIL} \)
⇒ can use bit position as \( e \)'s name

How do we handle naming?

Would like to ensure that \( e \in \text{AVAIL}(b) \) has a unique name.

1. as replacements are done
   (a) generate unique temporary name
   (b) add a copy at each \( \text{DEF} \) for that expression
2. map textual expressions into unique names
   \((\text{hash, bv pos'ns})\)
3. equivalent value numbers get same temporary name

Strategies to be discussed

(1) the classical method - it works
(2) limits replacement to textually identical expressions
(3) requires more analysis but yields more cses

Are copies a concern? Ask the register allocator.
Global Common Subexpression Elimination

**Approach 1:**

generate a unique name for each replacement

In value numbering step

1. \( \forall e \in \text{AVAIL}(b) \), initialize AVAILTAB and mark it
2. if we use \( e \) and \( e \in \text{AVAIL}(b) \),
   (a) allocate a name \( n \),
   (b) search backwards from \( b \) along each path in the cfg to find last DEF of \( e \),
   (c) insert a copy to \( n \) after DEF
   (d) replace \( e \) with \( n \)

Problems:

- searching might take a fair amount of time
  - conceptually ugly notion
  - Scarborough suggests that this is not a major worry
- \(|\text{names}| \ll |\text{USES}|\) \( (> |\text{AVAIL}|)\)
- single DEF followed by many copies

---

Example

```
C = A + B
D = A + B
E = A + B
```

Approach 1

What's the best we can do?
Global Common Subexpression Elimination

Approach 2:

textually identical expressions get the same name

Before any value numbering

1. initialize an array USED to false
   (|USED| = |AVAIL|)
2. ∀ e ∈ AVAIL(b), initialize AVAILTAB and mark it
3. if we use e and e ∈ AVAIL(b)
   (a) if e is unused (i.e., it has not been assigned a name) allocate one, else use assigned name
   (b) set USED[name] to true
   (c) replace e with name

After all value numbering

4. ∀ block b, if e ∈ DEF(b) and USED[e]
   insert a copy to name after the DEF of e

Problems

• may insert extra copies
• adds one more pass over the code

Global Common Subexpression Elimination

Approach 3:

textually identical expressions get the same name

In value numbering step

1. ∀ e ∈ AVAIL(b), initialize AVAILTAB
2. at an evaluation of e
       (hash 'em)
       (a) insert a copy of e to name
3. at a use of e, if e ∈ AVAIL(b)
       replace it with a reference to name

Problems:

• inserts more extraneous copies than approach 2
• extra copies are dead
What about all those extra copies?

- dead code elimination
- subsumption or coalescing

⇒ rely on other optimizations to catch them!

Common strategy (PL.8 compiler)

- insert copies that might be useful
- let dead code eliminator discover the useless ones
- simplifies compiler (for a price)

Dead code elimination

- must be able to recognize global usefulness
- must be able to eliminate useless stores
- must have strong notion of “dead"
Global Common Subexpression Elimination

The iterative algorithm computes a maximum fixed point MFP solution to the set of equations. This need not be same as the MOP.

- if \( \mathcal{F} \) is distributive, \( \text{MOP} = \text{MFP} \)
- if \( \mathcal{F} \) is not distributive, \( \text{MOP} \geq \text{MFP} \)

Is \( \text{AVAIL}(b) \) distributive?

Is \( \text{AVAIL}(b) \) rapid, i.e. does \( \text{AVAIL}(b) \) converge after two iterations around a loop?

Next Time

Scheduling and Register in a single basic block