Code Transformation

CS 544

Baris Aktemur

Contents taken from Vikram Adve's lecture notes.

1

Loop Invariant Code Motion (LICM)

- Given a statement

  `S: X = A + B;`

  inside a **natural** loop L;

  What's this?

- Goal:
  - move `A+B` out of L, if legal.
  - move assignment to `X` out of L, if legal.

2

Natural Loop

(def. from the Dragon Book)

- A natural loop is defined by two essential properties
  - It must have a **single-entry** node, called the **header**. This entry node dominates all nodes in the loop, or it would not be the sole entry to the loop.
  - There must be a **back edge** that enters the loop header. Otherwise, it is not possible for the flow of control to return to the header directly from the "loop" ; i.e., there really is no loop.

3

LICM

- *Analysis*: Find reaching defs of each variable in RHS and check if they are all outside the loop, or only one def reaches the variable and it is loop-invariant
- Consequences
  - Fewer computations (often, *much* fewer)
  - Adds some copy instructions ⇒ cheaper than any operation
  - May stretch some live ranges

4

LICM

- Opportunities
  - Array indexing expressions
  - Structure indexing expressions
  - Effect of previous transformations

5

LICM Examples

Example 1: Invariant def overwritten by later def

```
for (i=0; i < N; ++i) {
      X = a * b;    // hoist a*b but not def of X
      Y = X * i;
      X = Y + 1; }
```

Example 2: Def does not dominate exit

```
for (i=0; i < N; ++i) {
      if (...)
            X = a * b; // hoist a*b but not def of X }
```

Example 3: Multiple defs reach a use

```
for (i=0; i < N; ++i) {
      X = a * b;       // hoist a*b but not def of X
      if (...)
            X = X * i;
      Y = X; }
```

6

LICM Legality

- Moving expression evaluation out of L
  - (E1) *Strict*: S must dominate all exit nodes from loop L
  - (E1′) *Relaxed*: S must dominate all exit nodes from loop L ***or*** A + B must not cause any exceptions

7

LICM Legality

- Moving def of X out of L:
  - (D1) S must dominate all exit nodes from L except exit nodes where X is dead
  - (D2) No other statement in the loop must store to X
  - (D3) No use of X in L must be reached by any other def of X.
- Note: With SSA form, we only need D1!

8

LICM Algorithm (1/2)

Inputs

Procedure in 3-address form
Natural loop L, with preheader block P
Def-use and Use-def chains for the procedure

LICM()

```
repeat (until no new statements are marked)
   for (each statement S: X = expr in L)
      IsInvariant = true;
      for (all operands u ∈ S)
         if (any defs reaching u are within L)
            if (more than one def reaches u
                || (the single def d reaching u is
                      not constant and not invariant))
               { IsInvariant = false; break }
      if (IsInvariant)    // expr is loop-invariant
         Mark s invariant
```

9

LICM Algorithm (2/2)

```
for (each statement S: X = expr in L) do
   if (S is marked invariant)
      if (BB containing S dominates all loop exits
              || expr causes no exceptions)
         insert tmp = expr just before loop L
      if (conditions (D1)...(D3) are satisfied) {
         insert X = tmp just before loop L;
         delete S
      } else
         replace S with X = tmp
```

10

Global Common Subexpression Elimination (GCSE)

- Goal:
  - Eliminate redundant evaluation of an expression if it is available on all incoming paths
- Analysis: AVAIL proves that the value is current
- Transformation:
  - Introduce new temporary for each CSE discovered
  - don't add evaluations to any path

11

GCSE

- Consequences
  - same or fewer evaluations on every path
  - add some copy instructions
    ⇒ many copies coalesce away during allocation
  - major cost: can stretch live ranges
    ⇒ may need forward substitution to undo some CSE results

12

GCSE

- Opportunities
  - Array indexing expressions
  - Structure indexing expressions
  - *Clean* user-written code

13

GCSE Algorithm (1/2)

Inputs

(1) 3-address code + CFG for a procedure

(2) Numbered set of expressions $\mathcal{U} = \{e_1, \ldots e_N\}$
Use lexically identical expressions; apply reassociation first

(3) Available expressions, AVAIL$_{in}(B)$, for each block B

GCSE()

```
EverRedundant[i] = false,  ∀1 ≤ i ≤ N;
for each block B
    for each statement S : X = Y op Z in B
        if (e_j = "Y op Z" ∈ AVAIL_in(B)
              and e_j is not killed before S in B)
        {
            EverRedundant[j] = true
            Create new temporary tmp_j
            Replace S with X = tmp_j }
        }
```

GCSE Algorithm (2/2)

```
for each block B
    for each original statement T : X = Y op Z in B
        if (EverRedundant[k])      // where e_k = "Y op Z"
        {
            replace T with the pair:
                tmp_j = Y op Z
                W     = tmp_j
        }
```

15