Lecture #31

Code Optimization

Dead Code Elimination

- Remove code which does not affect the program results.

- Benefits:
  - It shrinks program size
  - It allows the running program to avoid executing irrelevant operations and thus reduces running time.

- Dead code includes:
  - Code that is never be executed (unreachable code)
  - Code that only affects **dead variables**, that is, variables whose definition remains unused.

Dead Code Elimination

```
int foo(void) {
   int a = 24;
   int b = 25; /* Assignment to dead variable */
   int c;
   c = a << 2;
   return c;
   b = 24;       /* Unreachable code */
   return 0;     /* Unreachable code */
}
```

Common Subexpression Elimination (CSE)

- Two operations are *common* if they produce the same result.
- An expression is *alive* if the operands used to compute the expression have not been changed.
    - An expression not alive is *dead*.
- *CSE* searches for instances of expressions that evaluate to the same value and analyses if it may be replaced with a single variable holding the computed value.

```
a = b * c + g;
d = b * c * d;
```

→

```
tmp = b * c;
a = tmp + g;
d = tmp * d;
```

CSE done by using “tmp” in the definition of “d”

Local Optimization Example

a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f

Operator strength reduction

Local Optimization Example

a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f

Constant and copy propagation

Local Optimization Example

a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f

Constant folding

Local Optimization Example

a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f

Common Subexpression Elimination

Local Optimization Example

a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f

Constant and copy propagation

Local Optimization Example

a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f

Dead code elimination

Local Optimization Example

a := x * x
f := a + a
g := 6 * f

Optimized form

An Automated Local Optimization Algorithm

- Value Numbering – Central idea:
    - Assign numbers (called value numbers (vn)) to expressions in such a way that two expressions receive the same VN if the compiler can prove that they are equal for all possible program inputs

- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks

- Can take advantage of algebraic simplification and re-association

Value Numbering

- The algorithm uses three tables indexed by appropriate hash values:
  - HashTable: holds vn for expressions
    - Format: <expression, vn>
    - Indexed by expression hash value
  - ValnumTable: holds vn for named identifiers
    - Format: <name, vn>
    - Indexed by name hash value
  - NameTable: holds the values for names in case of constants
    - Format: <name list, constant value, constflag>
    - Indexed by vn
    - In the field Namelist, first name is the defining occurrence and replaces all other names with the same vn with itself (or its constant value)

Value Numbering Example

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$ $b = 4 * a$ $c = i * j + b$ $d = 15 * a * c$ $e = i$ $c = e * j + i * a$	1. $a = 10$ 2. $b = 4 * a$ 3. $t1 = i * j$ 4. $c = t1 + b$ 5. $t2 = 15 * a$ 6. $d = t2 * c$ 7. $e = i$ 8. $t3 = e * j$ 9. $t4 = i * a$ 10. $c = t3 + t4$	1. $a = 10$ 2. $b = 40$ 3. $t1 = i * j$ 4. $c = t1 + 40$ 5. $t2 = 150$ 6. $d = 150 * c$ 7. $e = i$ 8. $t3 = i * j$ 9. $t4 = i * 10$ 10. $c = t1 + t4$ (Instructions 5 and 8 can be deleted)

Value Numbering Example

HashTable

Expression	Value-Number
$i * j$	5
$t1 + 40$	6
$150 * c$	8
$i * 10$	9
$t1 + t4$	11

NameTable

Name	Constant Value	Constant Flag
a	10	T
b	40	T
i, e		
j		
$t1, t3$		
$t2$	150	T
d		
c		

ValNumTable

Name	Value-Number
a	1
b	2
i	3
j	4
$t1$	5
c	6,11
$t2$	7
d	8
e	3
$t3$	5
$t4$	10

Value Numbering Example

- a = 10
  - *a* is entered into ValnumTable (with a vn of 1, say) and into NameTable (with a constant value of 10)
- b = 4 * a
  - *a* is found in ValnumTable, its constant value of 10 in NameTable
    - We perform constant propagation and folding
    - 4 * a is evaluated to 40
    - *b* is entered into ValnumTable (with a vn of 2) and into NameTable (with a constant value of 40)
- t1 = i * j
  - *i* and *j* are entered into the two tables with new vn (as above), but with no constant value
  - *i * j* is entered into HashTable with a new vn
  - *t1* is entered into ValnumTable with the same vn

Value Numbering Example

- Similar actions continue till $e = i$
  - e gets the same vn as i
- t3 = e * j
  - e and i have the same vn
  - $e * j$ is detected to be the same as $i * j$
  - since $i + j$ is already in the HashTable, we have found a *common subexpression*
  - from now on, all uses of $t3$ can be replaced by $t1$
  - The instruction $t3 = e * j$ can be deleted
- c = t3 + t4
  - $t3$ and $t4$ already exist and have vn
  - $t3 + t4$ is entered into HashTable with a new vn
  - Reassignment to c; c gets a different vn, same as that of $t3 + t4$

Value Numbering Example

- When a search for an expression $i + j$ in HashTable fails, try for $j + i$
- If there is an instruction $x = i + 0$, replace it with $x = i$
- Any quad of the type, $y = j * 1$ can be replaced with $y = j$
- After the above two types of replacements, value numbers of x and y become the same as those of i and j, respectively
- Quads whose LHS variables are used later can be marked as *alive*
- All unmarked quads can be deleted at the end