Module-4-Code Optimization

Code Optimization techniques

- Constant Folding
- Constant Propagation
- Algebraic Simplification
- Operator Strength Reduction
- Copy Propagation
- Dead Code Elimination

Constant Folding

- Evaluate constant expressions at compile time.

```
c:= 1 + 3   →   c:= 4

!true       →   false
```

Example:

In the code fragment below, the expression (3 + 5) can be evaluated at compile time and replaced with the constant 8.

```
int f ()
{
 return 3 + 5;
}
```

Below is the code fragment **after constant folding**.

```
int f ()
{
 return 8;
}
```

Expressions with constant operands can be evaluated at compile time, thus improving run-time performance and reducing code size by avoiding evaluation at run-time.

Constant Propagation

Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

- Variables that have constant value, **e.g.** **b** := 3
  - Later uses of **b** can be replaced by the constant, if no change of **b** in between.

```
b := 3
c := 1 + b
d := b + c
```

```
b := 3
c := 1 + 3
d := 3 + c
```

Example:

- In the code fragment below, the value of **x** can be propagated to the use of **x**.

```
x = 3;
y = x + 4;
```

- Below is the code fragment **after constant propagation** and constant folding.

```
x = 3;
y = 7;
```

Algebraic Simplification

- Use algebraic properties to simplify expressions `-(-i)` → `i`
- Some expressions can be simplified by replacing them with an equivalent expression that is more efficient.

Example:

The code fragment below contains expressions that can be simplified.

```
void f (int i)
{
 a[0] = i + 0;
 a[1] = i * 0;
 a[2] = i - i;
 a[3] = 1 + i + 1;
}
```

Below is the code fragment **after** **expression simplification**.

```
void f (int i)
{
 a[0] = i;
 a[1] = 0;
 a[2] = 0;
 a[3] = 2 + i;
}
```

Common Sub expression elimination

- Common Sub expression elimination is a optimization that searches for instances of identical expressions (i.e. they all evaluate the same value), and
- Analyses whether it is worthwhile replacing with a single variable holding the computed value.

Example:

```
a := b * c                temp := b * c
...                       a := temp
...                       ...
x := b * c + 5            x := temp + 5
```

Identify common sub-expression present in different expression, compute once, and use the result in all the places.

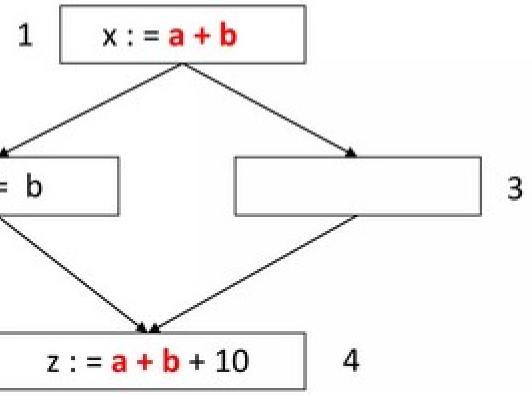
Common Sub-expression elimination

- Common sub-expression elimination
  - **Example 1:**

```
a := b + c                a := b + c
c := b + c                c := a
d := b + c                d := a
```

  - **Example 2:** in array index calculations
    - c[i+1] := a[i+1] + b[i+1]
    - During address computation, i+1 should be reused
    - Not visible in high level code, but in intermediate code

Common Sub-expression evaluation



"**a + b**" is not a common sub-expression in 1 and 4

Strength Reduction

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

```
16*x    -->     x << 4
```

  - Utility is machine-dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Strength reduction

- Example:
  - Replace X^2 computations by X*X
  - Replace multiplication by left shift
  - Replace division by right shift

Copy Propagation

- Given an **assignment** x = y, replace later uses of **x** with uses of **y**, provided there are no intervening assignments to **x** or **y**.
- **Example**

```
x[i] = a;                    x[i] = a;
sum = x[i] + a;      →       sum = a + a;
```

- **Example**

```
x := y;                      s := y * f(y)
s := x * f(x)        →
```

After y is assigned to x, use y to replace x till x is assigned again reduce the copying.

If y is reassigned in between, then this action cannot be performed.

Operator Strength Reduction

```
y := x * 2     →     y := x + x
```

- Replace expensive operations with simpler ones
- Typical cases of strength reduction occurs in address calculation of array references.
- **Example:** Multiplications replaced by additions ,

```
for i=1 to 10                temp = 5;
   {                         for i=1 to 10
   ...                       {
   x = i * 5                    ...
   ...                          x = temp
                                ...
   }                         temp = temp + 5
                             }
```

Replacement of an **operator** with a less costly one.

Dead code Optimization:

- Dead Code elimination removes code that does not affect a program.
- Removing such code has two benefits.
  - It shrinks program size.
  - It avoids the executing irrelevant operations, which reduces its running time.
- Two types of Dead Code elimination
  - Unreachable Code
  - Redundant statement

Unreachable Code - Dead code Optimization

- In Computer Programming, Unreachable Code or dead code is code that exists in the source code of a program but can never be executed.

Program Code

```
if (a>b)
        m=a
elseif (a<b)
        m=b
elseif (a==b)
        m=0
else
        m=-1
```

Optimized Code

```
if (a>b)  m=a
elseif (a<b)  m=b
else
m=0
```

Dead Code Elimination

- Dead Code are portion of the program which will not be executed in any path of the program. It can be removed
- Examples:
  - No control flows into a basic block
  - A variable is dead at a point (i.e) its value is not used anywhere in the program
  - An assignment is dead (i.e) assignment assigns a value to a dead variable
  - Ineffective statements:

x := y + 1		**(x is immediately redefined in 3rd line without use, therefore eliminate)**
y := 5		y := 5
x := 2 * z	→	x := 2 * z

- A variable is dead if it is never used after last definition
- Eliminate assignments to dead variables
- Need to do data flow analysis to find dead variables

Redundant Code - Dead code Optimization

- Redundant Code is code that is executed but has no effect on the output from a program

```
main()
{
        int a, b, c, r;
        a=5;
        b=6;
        c=a + b;
        r=2;  r++;          <-----    Adding time & space complexity
        printf(" %d" ,c);
}
```

Dead Code Elimination

- Remove code never reached

```
if (false)
{a := 5}
```

→

```
if (false)
{}
```

Loop optimization

- Loop optimization plays an important role in improving the performance of the source code by reducing overheads associated with executing loops.
- The inner loops where programs tend to spend the bulk of their time.
- The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- **Loop Optimization techniques**:
  - Code Motion - moves code outside a loop
  - Induction variables elimination - replace variables from inner loop
  - Reduction in strength - replaces expensive operation by a cheaper one, such as a multiplication by an addition

Code Motion - Loop Optimization

- Example - Computation can be moved to outside of the loop

```
i = 1
s= 0
do{
    s= s + i
    a =5
    i = i + 1
  }
  while (i <=n)
```

→

```
i = 1
s= 0
a =5
do{
    s= s + i
    i = i + 1
  }
  while (i <=n)
```

Bringing a=5 outside the do while loop, is called <u>code motion</u>.

Code Motion - Loop Optimization

- Example

```
for (i=0; i<n; i++)
        a[i] = a[i] + x/y;
```

- **Three address code**

```
for (i=0; i<n; i++)
    {
    c = x/y;
    a[i] = a[i] + c;
    }
```

↓

```
c = x/y;
for (i=0; i<n; i++)
    a[i] = a[i] + c;
```

Code Motion

- **Reduce frequency with which computation performed**
  - **If it will always produce same result**
  - **Especially moving code out of loop**

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Code hoisting - **Loop Optimization**

- *Code Space reduction*: Similar to common sub-expression elimination but with the objective to reduce code size.

Example: Code hoisting

```
if (a< b) then
   z := x ** 2
else
   y := x ** 2 + 10
```

```
temp : = x ** 2
if (a< b) then
  z := temp
else
   y := temp + 10
```

“x ** 2“ is computed once in both cases, but the code size in the second case reduces.

Induction variable elimination

- If there are multiple **induction variables** in a loop, can eliminate the ones which are used only in the test condition

The code fragment below has three **induction variables (i1, i2, and i3)** that can be replaced with one induction variable

```
int a[SIZE];
int b[SIZE];
void f (void)
{
 int i1, i2, i3;
for (i1 = 0, i2 = 0, i3 = 0; i1 < SIZE; i1++)
   a[i2++] = b[i3++];
   return;
}
```

The code fragment below shows the loop **after induction variable elimination**.

```
int a[SIZE];
int b[SIZE];

void f (void)
{
 int i1;

 for (i1 = 0; i1 < SIZE; i1++)
   a[i1] = b[i1];
 return;
}
```

Induction variable elimination

- **Example**

```
s := 0;
for (i=0; i<n; i++)
{
   s := 4 * i;
   …
}
```

```
s := 0;
e := 4*n;
while (s < e)
{
        s := s + 4;
}
```

// **i** is not referenced in loop

Loop Fusion - Loop Optimization

Before Loop Fusion

- **Example**

```
for (i=0; i<n; i++) {
   A[i] = B[i] + 1
   }
for (i=0; i<n; i++) {
   C[i] = A[i] / 2
}
for (i=0; i<n; i++) {
   D[i] = 1 / C[i+1]
}
```

```
for (i=0; i<n; i++) {
  A[i] = B[i] + 1
  C[i] = A[i] / 2
  D[i] = 1 / C[i+1]
}
```

Is this correct?

Actually, cannot fuse the third loop

```
for (i=0; i<n; i++) {
          A[i] = B[i] + 1
          C[i] = A[i] / 2
}
for (i=0; i<n; i++) {
          D[i] = 1 / C[i+1]
}
```

Loop unrolling or Loop collapsing - **Loop Optimization**

- Execute loop body multiple times at each iteration
- Try to get rid of the conditional branches, if possible
- Allow optimization to cross multiple iterations of the loop
  - Especially for parallel instruction execution

Loop unrolling or Loop collapsing - **Loop Optimization**

Example:

In the code fragment below, the double-nested loop on **i** and **j** can be collapsed into a single-nested loop.

```
int a[100][300];

for (i = 0; i < 300; i++)
  for (j = 0; j < 100; j++)
    a[j][i] = 0;
```

Here is the code fragment **after the loop has been collapsed**.

```
int a[100][300];
int *p = &a[0][0];

for (i = 0; i < 30000; i++)
  *p++ = 0;
```

Loop Unrolling: -

- For Example: -

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

- Can be written as

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

<table>
<tr><th>Normal loop</th><th>After loop unrolling</th></tr>
<tr><td><pre>
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
</pre></td><td><pre>
int x;
for (x = 0; x < 100; x+=5)
{
    delete(x);
    delete(x+1);
    delete(x+2);
    delete(x+3);
    delete(x+4);
}
</pre></td></tr>
</table>