Code Optimization
5TH Phase of Compiler Construction

CODE OPTIMIZATION TECHNIQUES

- Constant propagation
- Constant folding
- Algebraic simplification, strength reduction
- Copy propagation
- Common subexpression elimination
- Unreachable code elimination
- Dead code elimination
- Loop Optimization
- Function related
  - Function inlining,
  - Function cloning

CONSTANT PROPAGATION

If the value of a variable is a constant, then replace the variable by the constant

- It is not the constant definition, but a variable is assigned to a constant
- The variable may not always be a constant

Example:

```
pi := 3.14286
area = pi * r ** 2
```

area = 3.14286 * r ** 2

Example:

```
N := 10; C := 2;
for (i:=0; i<N; i++) { s := s + i*C; }
```

for (i:=0; i<10; i++) { s := s + i*2; }

Requirement:

- After a constant assignment to the variable
- Until next assignment of the variable
- Perform data flow analysis to determine the propagation

CONSTANT FOLDING

Evaluation of an expression with constant operands to replace the expression with single value

Example:

area := (22.0/7.0) * r ** 2 **area := 3.14286 * r ** 2**

Example:

#define M 10

x := 2 * M; **x := 20**

If (M < 0) goto L **can be eliminated**

y := 10 * 5

y := 50

Difference between Constant Propagation and Folding

Propagation: only substitute a variable by its assigned constant

Folding: Consider variables whose values can be computed at compilation time and controls whose decision can be determined at compilation time

Algebraic simplification and Strength Reduction

More general form of **constant folding**, e.g.,

x + 0 = x x – 0 = x

x / 1 = x x * 1 = x

x * 0 = 0

Repeatedly apply the rules

(y * 1 + 0) / 1 = y

Strength reduction

Replace expensive operations

A ^ 2 = A * A

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.

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

Example:

```
x[i] = a;                    x[i] = a;

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

Example

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

May not appear to be code improvement, but opens up scope for other optimizations

Common Subexpression Elimination

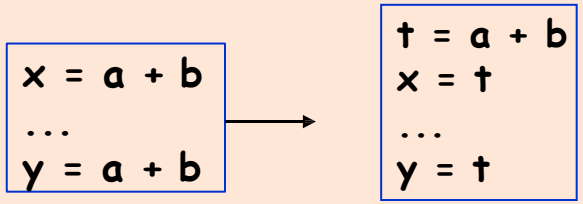
Identify common sub-expression present in different expression, compute once, and use the result in all the places.

The *definition* of the variables involved should not change

a := b * c	**temp := b * c**
…	**a := temp**
…	…
x := b * c + 5	**x := temp + 5**

Common Subexpression Elimination

- Local common subexpression elimination
- Performed within basic blocks
- Algorithm sketch:
  - Traverse Block from top to bottom
  - Maintain table of expressions evaluated so far
    - if any operand of the expression is redefined, remove it from the table
  - Modify applicable instructions as you go
    - generate temporary variable, store the expression in it and use the variable next time the expression is encountered



Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```

```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /a /* commutative */ = j +
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

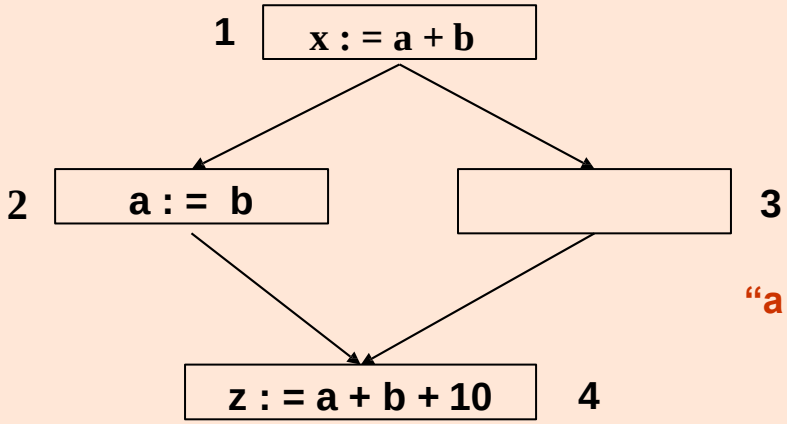
Common Subexpression Elimination

Global common subexpression elimination

- Performed on flow graph
- Requires available expression information

In addition to finding what expressions are available at the endpoints of basic blocks, we need to know where each of those expressions was most recently evaluated (which block and which position within that block).

COMMON SUBEXPRESSION ELIMINATION



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

None of the variable involved should be modified in any path

Unreacheable code elimination

- Construct the control flow graph
- Unreachable code block will not have an incoming edge
- After constant propagation/folding, unreachable branches can be eliminated

Dead code elimination

- 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

Ineffective statements

x := y + 1 (immediately redefined, eliminate!)

y := 5 → **y := 5**

x := 2 * z **x := 2 * z**

LOOP OPTIMIZATION

Consumes 90% of the execution time

$\Rightarrow$ a larger payoff to optimize the code within a loop

Techniques

- Loop invariant detection and code motion
- Induction variable elimination
- Strength reduction in loops
- Loop unrolling
- Loop fusion

LOOP INVARIANT DETECTION AND CODE MOTION

- If the result of a statement or expression does not change within a loop, and it has no external side- effect, Computation can be moved to outside of the loop
- Example

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

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

Code Motion

- Identify invariant expression:

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

- Hoist the expression out of the loop:

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

STRENGTH REDUCTION IN LOOPS

- **Example**

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

⇒ **s := 0;**

for (i=0; i<n; i++) { v := v + 4; s := s + v;)

Induction variable elimination

The code fragment below has three induction variables (i_1, i_2, and i_3) that can be replaced with one induction variable, thus eliminating two induction variables.

```
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;
}
```

Induction variable elimination

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;
}
```

```
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;
}
```

LOOP UNROLLING

- Execute loop body multiple times at each iteration

```
for (i = 0; i< n; i++) { S }
```

- Unroll loop four times:

```
for (i = 0; i < n-3; i+=4) { S; S; S; S; }
for ( ; i < n; i++) S;
```

- Get rid of the conditional branches, if possible
- Allow optimization to cross multiple iterations of the loop
  Especially for parallel instruction execution
- Space time tradeoff

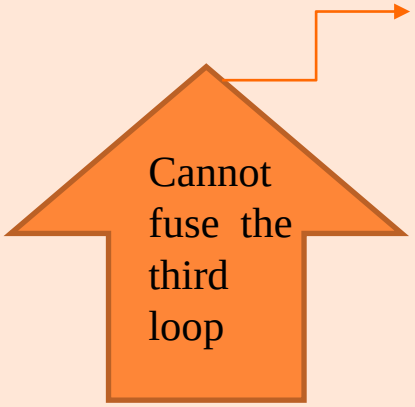
  Increase in code size, reduce some instructions

LOOP FUSION

Before Loop Fusion

Example

```
for i=1 to N do
  A[i] =  B[i] + 1
Endfor
for i=1 to N do
  C[i] = A[i] / 2
endfor
for i=1 to N do
  D[i] = 1 / C[i+1]
endfor
```



After Loop Fusion

```
for i=1 to N do
A[i] = B[i] + 1
C[i] = A[i] / 2
D[i] = 1 / C[i+1]
endfor
```

FUNCTION INLINING

Replace a function call with the body of the function

Save a lot of copying of the parameters, return address, etc.

In the code fragment below, the function add() can be expanded inline at the call site in the function sub().

```
int add (int x, int y)
    {
        return x + y;
    }
int sub (int x, int y)
  {
        return add (x, -y);
  }
```

Expanding add() at the call site in sub() yields

```
int add (int x, int y)
        {
return x + -y
}
```

Can be further optimized

```
int add (int x, int y)
{

return x-y
}
```

Function inlining usually increases code space, which is affected by the size of the inlined function, the number of call sites that are inlined, and the opportunities for additional optimizations after inlining.

Function cloning

- Create specialized code for a function for different calling parameters

void f(int x[], int n, int m) {for(int i=0; i

For a call f(a, 10, 1), create a specialized version of f:

void f_1(int x[])

{ for(int i=0; i<n;i++) { x[i] = x[i] + i*m; } }

• For another call f(b, p, 0), create another version f_2(…)

Can be useful for parallel processing