Compiler Design

Samit Biswas

samit@cs.iiests.ac.in

Department of Computer Science and Technology,
Indian Institute of Engineering Science and Technology, Shibpur

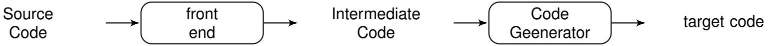
November 12, 2017

Code Optimization

Code Optimization

- A transformation to a program to make it run faster and/or take up less space.
- Optimization should be safe, preserve the meaning of a program.
- Code optimization is an important component in a compiler.

Getting Better Performances



User can
profile program,
change algorithm,
transform loops

Compiler can
improve loops,
procedure calls,
address cal-
culations

Compiler can
use registers,
select instructions,
do peephole
transformation

Levels:

- Window - Peephole Optimization
- Procedural - Global (Control flow graph)

Peephole Optimization

- examining a short sequence of target instructions and replacing these by faster sequences.
- Peephole is a small moving window on the target program.

Characteristics of Peephole Optimization

- redundant-instruction Elimination
- flow-of-control optimiztions
- algebraic simplifications
- use of machine idioms

Copy folding

`x = 32;` `x = x+32;`	Becomes	`x =64;`

Unreachable Code:

```
goto L2;
x = x+1;        unneeded
```

flow - of - Control Optimization

goto L1 ; . . . L1 : goto L2	Becomes	goto L2 . . . L1 : goto L2

- **algebraic simplifications**
  $x = x + 0 \quad \leftarrow$ Unneeded

- **Dead Code**
  $x = 32 \qquad \leftarrow$ where x not used after statement
  $x = x + y \quad \rightarrow$ x = y + 32

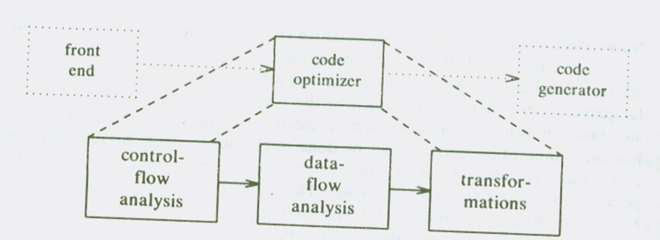
- **Reduction in Strength** - replace an expensive operation by a cheaper one

  $x = x * 2 \quad \rightarrow$ x = x + x

Peephole Optimization – Limitations

- Local in Nature.
- Pattern Driven.
- Limited by the size of the window.

Optimizing Compiler (Code Optimizer)



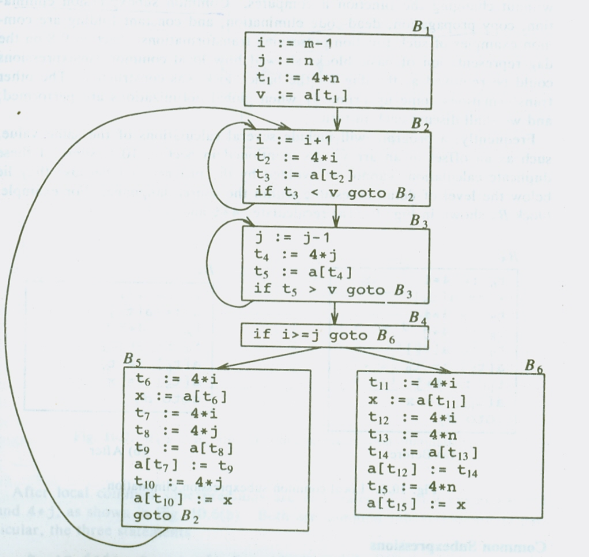
C Code for quicksort

```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while(1) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if (i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
```

Optimizing Compiler (Code Optimizer)

```
(1)   i := m-1
(2)   j := n
(3)   t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4*i
(17)  t8 := 4*j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4*j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4*i
(24)  x := a[t11]
(25)  t12 := 4*i
(26)  t13 := 4*n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4*n
(30)  a[t15] := x
```

Flow Graph



B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
B2
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2
B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
B5
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
B6
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x

Common Subexpressions

B_5

```
t6  := 4*i
x  := a[t6]
t7  := 4*i
t8  := 4*j
t9  := a[t8]
a[t7]  := t9
t10  := 4*j
a[t10]  := x
goto B2
```

(a) Before

B_5

```
t6  := 4*i
x  := a[t6]
t8  := 4*j
t9  := a[t8]
a[t6]  := t9
a[t8]  := x
goto B2
```

(b) After

- t_7 and t_{10} have common subexpressions $4 * i$ and $4 * j$ respectively in B_5.
- Eliminated by using t_6 instead of t_7 and t_8 instead of $t_1 0$.

- After common subexpressions are eliminated B_5 still evaluates $4 * i$ and $4 * j$. Both are common subexpressions ; in particular, the three statements $t_8 = 4 * j; t_9 = a[t_8]; a[t_8] = x$ in B_5 can be replaced by $t_9 = a[t_4]; a[t_4] = x$

Copy Propagations

Loop Optimizations

Mostly used loop optimization techniques are :

- **Code Motion** - which moves code outside the loop.
- **Induction variable elimination** - apply to eliminate i and j from the inner loops B_2 and B_3
- **Reduction in Strength** - which replaces an expensive operation by a cheaper one, such as multiplication by an addition.

Code Motion

```
while (i <= limit-2)    /* statement does not limit
```

Code motion will result in the equivalent of

```
t = limit - 2;
while (i <= t)   /* statement does not limit or t
```

Induction variables and Reduction in Strength