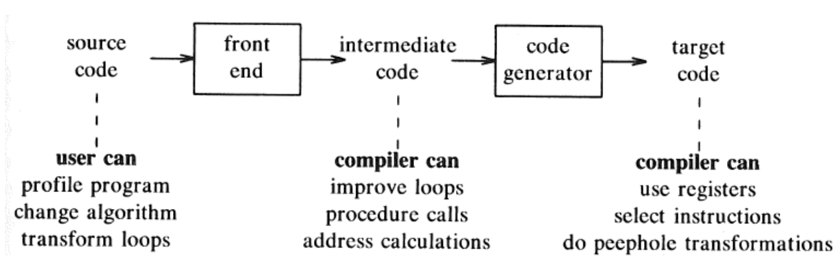
Various Optimization Techniques

Places for Optimization



Use of Algebraic Identities

```
x + 0 = 0 + x = x
x - 0 = x
x * 1 = 1 * x = x
x / 1 = x

x ** 2 = x * x
2.0 * x = x + x
x / 2 = x * 0.5
```

Use of Associative Laws

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

↓

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

↓

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

Control Flow Optimization

We'll focus on jumps to jumps, including conditional jumps.

Replace

```
    goto L1
    . . .
L1: goto L2
```

With

```
    goto L2
    . . .
L1: goto L2
```

If there are now no jumps to L1, then we may further simplify to:

```
    goto L2
    . . .
```

Control Flow Optimization

Suppose there is only one jump to **L1** and it is preceded by an unconditional **goto** in:

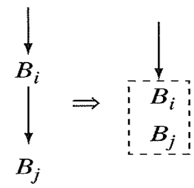
```
    goto L1
    . . .
L1: if a < b goto L2
L3:
```

We can simplify to:

```
    if a < b goto L2
    goto L3
    . . .
L3:
```

Eliminating Useless Control Flow

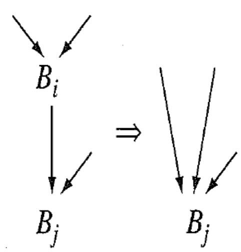
- If a block *Bi* ends in a jump to *Bj* and *Bj* has only one predecessor
- Combine the two blocks



Combine Blocks

Eliminating Useless Control Flow

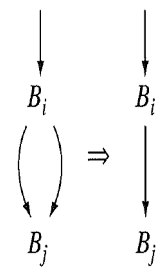
- If a block contains only a jump
- Merge the block into its successor



Remove an Empty Block

Eliminating Useless Control Flow

- If a block ends in a branch, and both sides of the branch target the same block
- Replace the branch with a jump to the target block



Fold a Redundant Branch

Unreachable Code

Consider the following C code:

```
#define debug 0
...
if ( debug ) {
    print debugging information
}
```

In the intermediate representation, the if-statement may be translated as:

```
    if debug = 1 goto L1
    goto L2
L1: print debugging information
L2:
```

Eliminate jumps over jumps:

```
    if debug ≠ 1 goto L2
    print debugging information
L2:
```

Unreachable Code (Cont'd)

Prior code:

```
     if debug ≠ 1 goto L2
     print debugging information
L2:
```

Replace constant **debug** with its value:

```
     if 0 ≠ 1 goto L2
     print debugging information
L2:
```

Since conditional evaluates to true, replace **if** with **goto**

```
     goto L2
     print debugging information
L2:
```

Finally, eliminate unreachable code:

```
L2:
```

Redundant Loads and Stores

The second instruction is redundant, eliminate it:

```
(1) MOV   R0, a
(2) MOV   a, R0
```

Loop Optimizations: Code Motion

Evaluation of `limit - 2` below is a loop-invariant.

```
while (i <= limit - 2)
```

Change to:

```
t = limit - 2;
while (i <= t)
```

Loop Unswitching

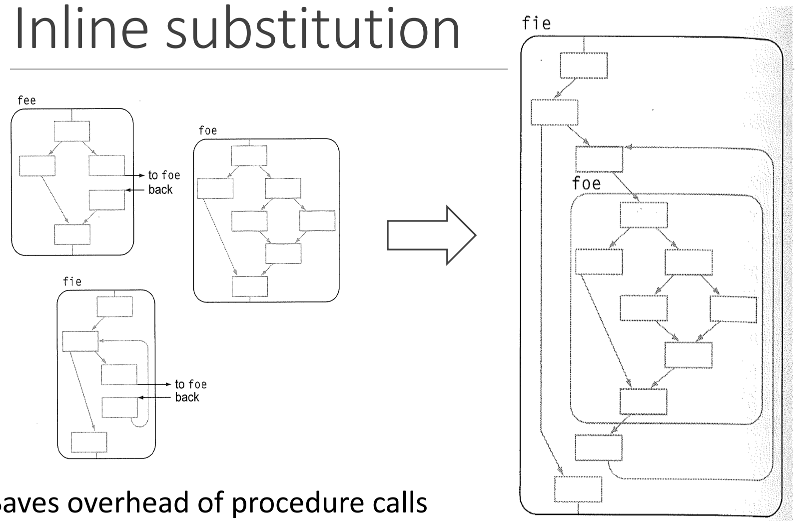
Loop unswitching hoists loop-invariant control-flow operations out of a loop.

```
do i = 1 to n
   if (x > y)
      then a(i) = b(i) * x
      else a(i) = b(i) * y
```

⇨

```
if (x > y) then
   do i = 1 to n
      a(i) = b(i) * x
else
   do i = 1 to n
      a(i) = b(i) * y
```

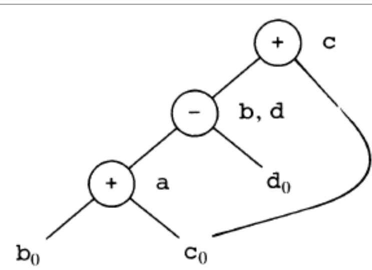
Inline substitution



- Saves overhead of procedure calls
- Consider number of times procedure gets called

Optimization of Basic Blocks

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



If b is not live after this block,
then we can eliminate the assignment:

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

User initiated changes

Bentley [1982] relates the following improvement:

- Sorting N elements: replace insertion sort with quicksort
- Improvement from 2.02 N^2 to 12 N $\log_2$ N
- N = 100: speedup by 2.5
- N = 100,000: speedup by more than a 1,000

Nice, but this is a compiler course, so we will focus on what the compiler can do.

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);
}
```

What the Compiler Cannot See

It cannot optimize.

Certain things cannot be seen at the source, but they may be visible in intermediate languages such as three-address code.

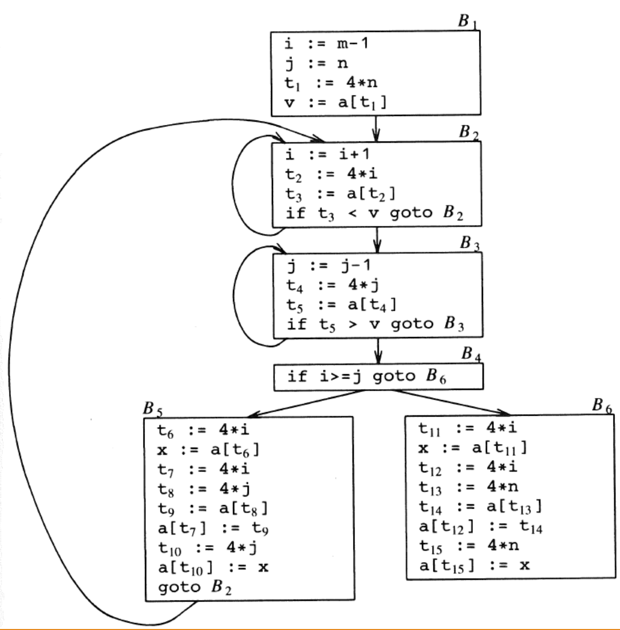
Quicksort in three-address code

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

Flow Graphs

In addition to three-address code, let us also look at the flow graph of the code.

Flow Graph of Three-Address Code



Function-Preserving Transformations

- In block B_5, eliminate local common subexpressions.
- They often occur when calculating offsets in an array.

Function-Preserving Transformations

- In block B_5, eliminate local common subexpressions.
- They often occur when calculating offsets in an array.

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

Function-Preserving Transformations

- Studying the code in B_5 in more detail.
- In particular, look at when **`i`** and **`j`** are calculated in the entire program fragment.
- Further simplify B_5.

Function-Preserving Transformations

- Studying the code in B_5 in more detail.
- In particular, look at when **`i`** and **`j`** are calculated in the entire program fragment.
- Further simplify B_5.

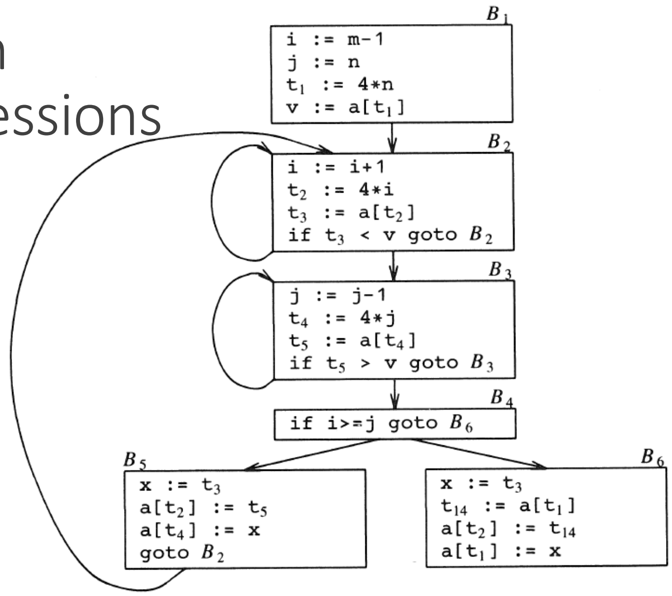
B_5

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

→

```
x := t3
a[t2] := t5
a[t4] := x
goto B2
```

Common Subexpressions



Copy Propagation

Use the rhs location for the lhs location, whenever possible.

x := t_3
a[t_2] := t_5
a[t_4] := x
goto B_2

→

x := t_3
a[t_2] := t_5
a[t_4] := t_3
goto B_2

Dead-Code Element Elimination

Very descriptive name.

```
x := t₃
a[t₂] := t₅
a[t₄] := t₃
goto B₂
```

→

```
a[t₂] := t₅
a[t₄] := t₃
goto B₂
```

Loop Optimizations: Induction Variables and Reduction in Strength

Consider loop around B_3 in next slide.

In the slide, we only show the relevant information.

Notice that the values of **j** and t_4 remain in lock-step.

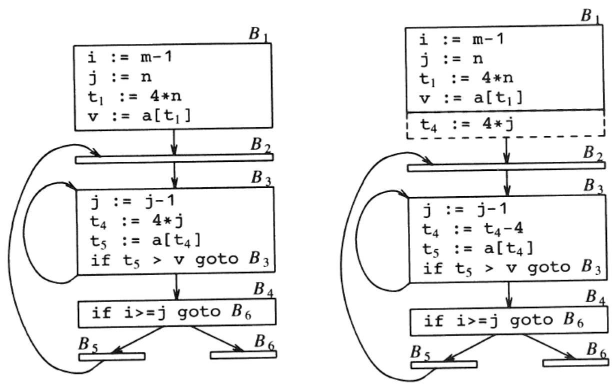
Every time **j** decreases by 1, t_4 decreases by 4

Such identifiers are called *induction variables*

Change t_4 = 4 * j to t_4 = t_4 - 4

Initialize t_4 to **4*j** in block above.

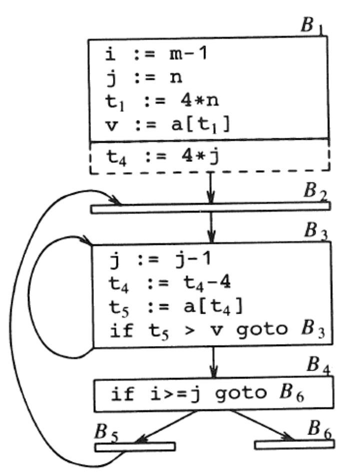
Loop Optimizations: Induction Variables and Reduction in Strength



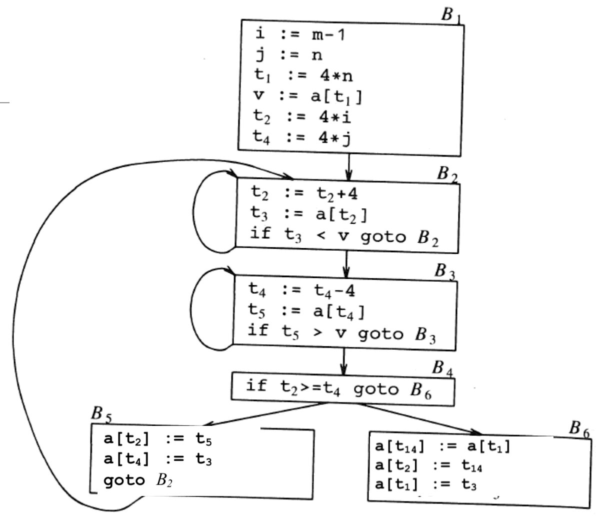
(a) Before (b) After

More dead code to be cleaned-up

`i` and `j` are now 



Result


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
t2 := 4*i
t4 := 4*j
B2
t2 := t2+4
t3 := a[t2]
if t3 < v goto B2
B3
t4 := t4-4
t5 := a[t4]
if t5 > v goto B3
B4
if t2>=t4 goto B6
B5
a[t2] := t5
a[t4] := t3
goto B2
B6
a[t14] := a[t1]
a[t2] := t14
a[t1] := t3