Machine-Independent Optimizations - Part 1

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

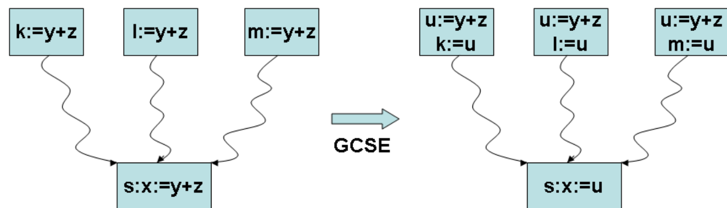
Outline of the Lecture

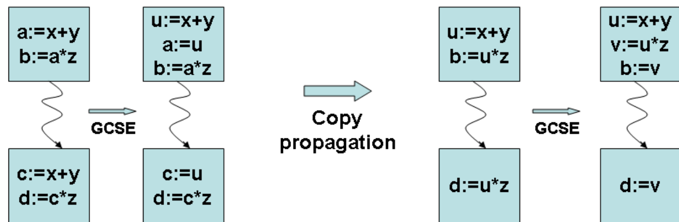
- Global common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Region based data-flow analysis

Elimination of Global Common Sub-expressions

- Needs available expression information
- For every $s: \; x := y + z$, such that $y + z$ is available at the beginning of s's block, and neither y nor z is defined prior to s in that block, do the following
  1. Search backwards from s's block in the flow graph, and find first block in which $y + z$ is evaluated. We need not go through any block that evaluates $y + z$
  2. Create a new variable u and replace each statement $w := y + z$ found in the above step by the code segment $\{u := y + z; w := u\}$, and replace s by $x := u$
- Repeated application of GCSE may be needed to catch "deep" CSE

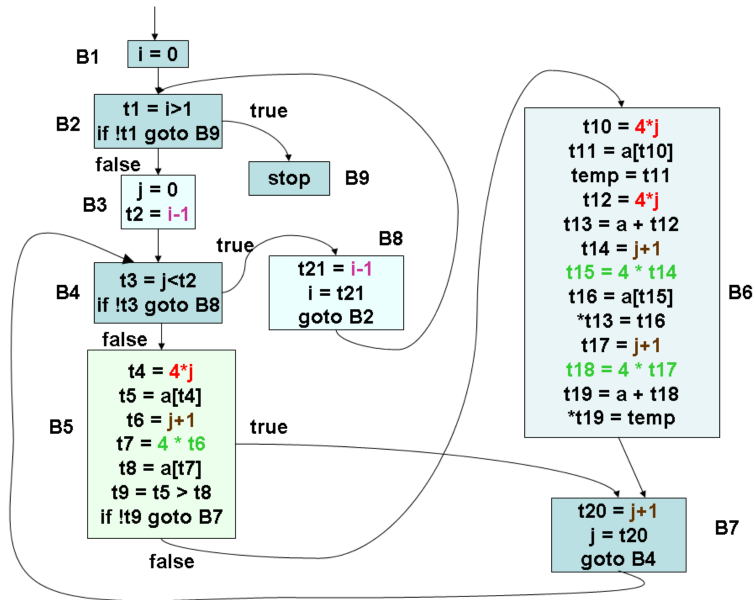
GCSE Conceptual Example



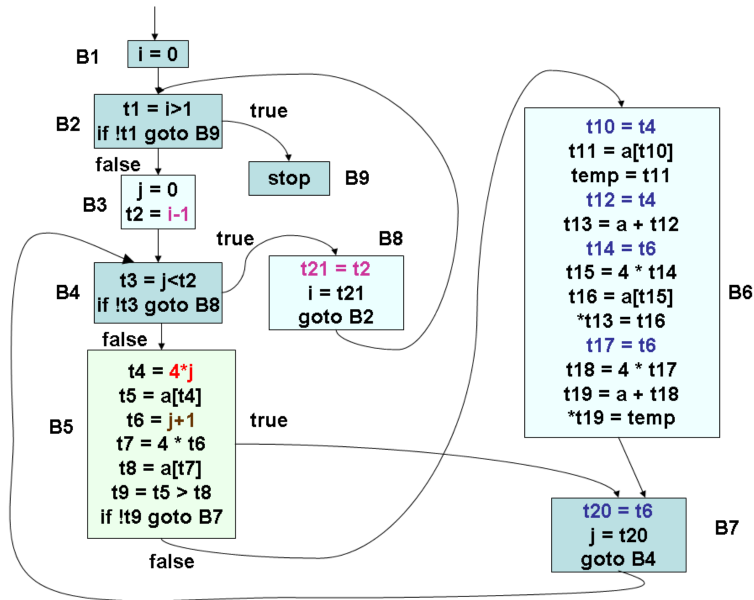


Demonstrating the need for repeated application of GCSE

GCSE on Running Example - 1



GCSE on Running Example - 2



Copy Propagation

- Eliminate copy statements of the form $s: \; x := y$, by substituting y for x in all uses of x reached by this copy
- Conditions to be checked
  1. u-d chain of use u of x must consist of s only. Then, s is the only definition of x reaching u
  2. On every path from s to u, including paths that go through u several times (but do not go through s a second time), there are no assignments to y. This ensures that the copy is valid
- The second condition above is checked by using information obtained by a new data-flow analysis problem
  - $c_gen[B]$ is the set of all copy statements, $s: x := y$ in B, such that there are no subsequent assignments to either x or y within B, after s
  - $c_kill[B]$ is the set of all copy statements, $s: x := y$, s not in B, such that either x or y is assigned a value in B
  - Let U be the universal set of all copy statements in the program

Copy Propagation - The Data-flow Equations

- $c_in[B]$ is the set of all copy statements, $x := y$ reaching the beginning of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path
- $c_out[B]$ is the set of all copy statements, $x := y$ reaching the end of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path

$$
\begin{aligned}
c_in[B] &= \bigcap_{P \text{ is a predecessor of } B} c_out[P], \; B \text{ not initial} \\
c_out[B] &= c_gen[B] \bigcup (c_in[B] - c_kill[B]) \\
c_in[B1] &= \phi, \text{ where } B1 \text{ is the initial block} \\
c_out[B] &= U - c_kill[B], \text{ for all } B \neq B1 \text{ (initialization only)}
\end{aligned}
$$

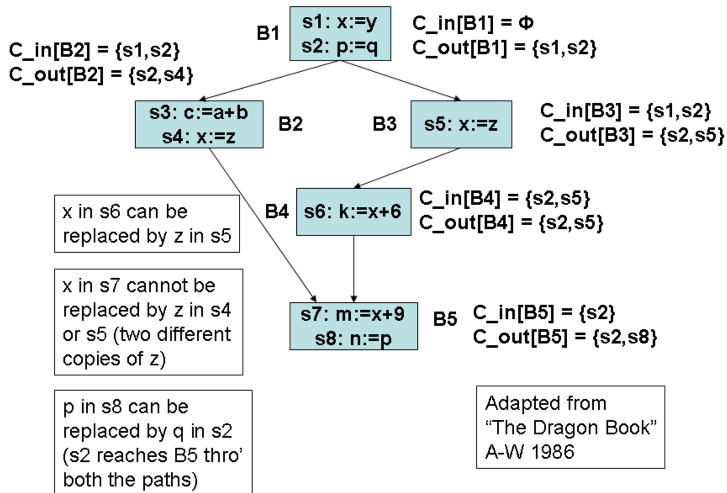
Algorithm for Copy Propagation

For each copy, $s : x := y$, do the following

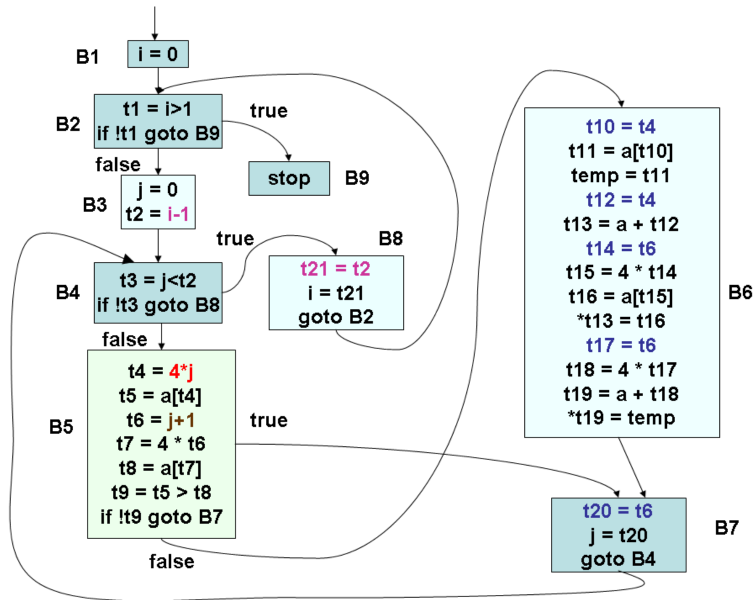
1. Using the $du - chain$, determine those uses of x that are reached by s
2. For each use of x found in (1) above, check that
   (i) s is in $c_in[B]$, where B is the block to which the use of x belongs. This ensures that
      - s is the only definition of x that reaches this block
      - No definitions of x or y appear on this path from s to B

   (ii) no definitions x or y occur within B prior to this use of x found in (1) above
3. If s meets the conditions above, then remove s and replace all uses of x found in (1) above by y

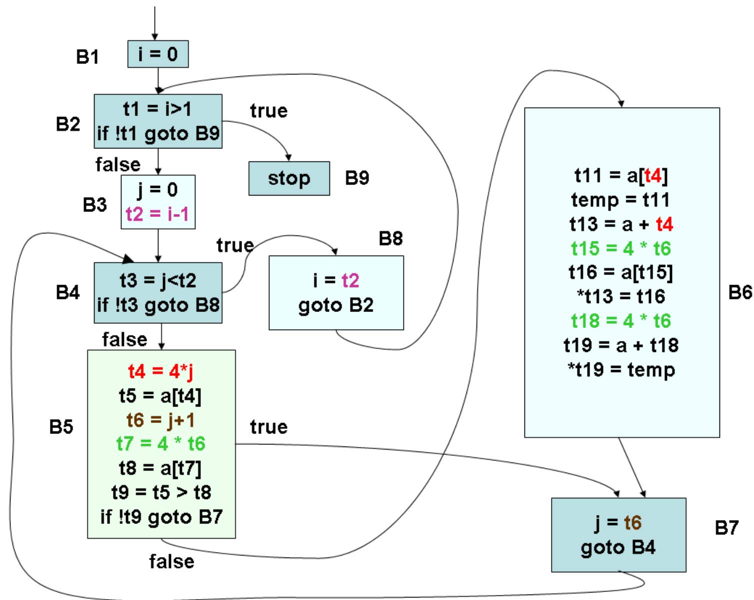
Copy Propagation Example 1



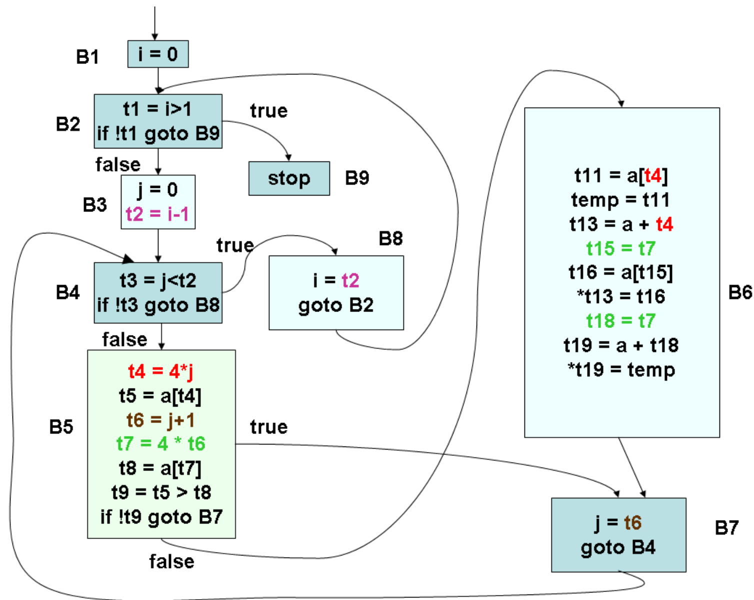
Copy Propagation on Running Example 1.1



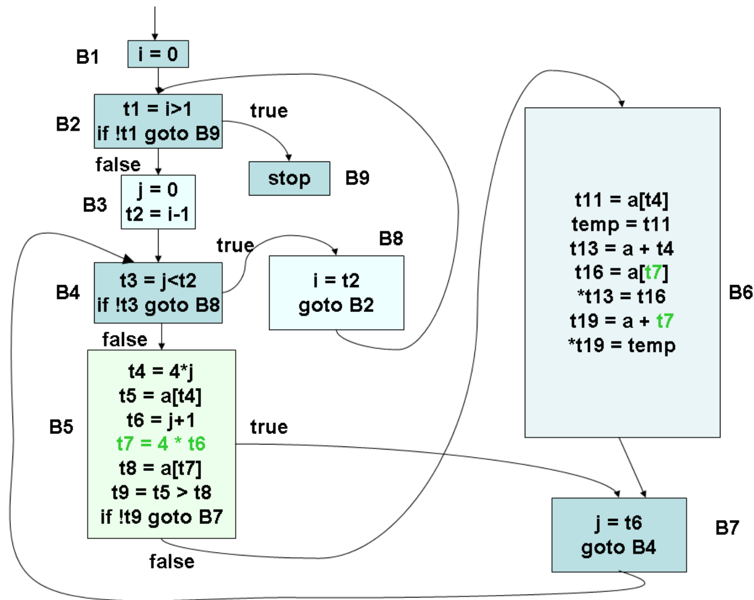
Copy Propagation on Running Example 1.2



GCSE and Copy Propagation on Running Example 1.1



GCSE and Copy Propagation on Running Example 1.2



Detection of Loop-invariant Computations

Given a loop L, and the $u - d$ and $d - u$ chains

Mark as "invariant", those statements whose operands are all either constant or have all their reaching definitions outside L

Repeat {
  Mark as "invariant" all those statements not previously so marked all of whose operands are constants, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked "invariant"
} until no new statements are marked "invariant"

$u - d$ chains are useful in marking statements as "invariant"
$d - u$ chains are useful in examining all uses of a definition marked "invariant"

Loop Invariant Code motion Example

```
      t1 = 202
      i = 1
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t3 = addr(a)
      t4 = t3 - 4
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before LIV code motion

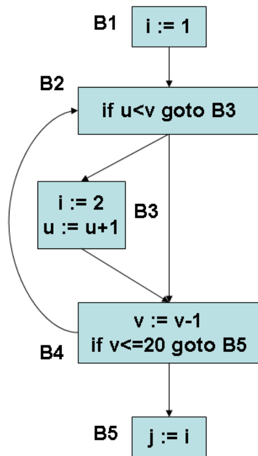
```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

After LIV code motion

Loop-Invariant Code Motion Algorithm

1. Find loop-invariant statements
2. For each statement s defining x found in step (1), check that
   (a) it is in a block that dominates all exits of L
   (b) x is not defined elsewhere in L
   (c) all uses in L of x can only be reached by the definition of x in s
3. Move each statement s found in step (1) and satisfying conditions of step (2) to a newly created preheader
   - provided any operands of s that are defined in loop L have previously had their definition statements moved to the preheader
4. Update all the $u - d$ and $d - u$ chains appropriately

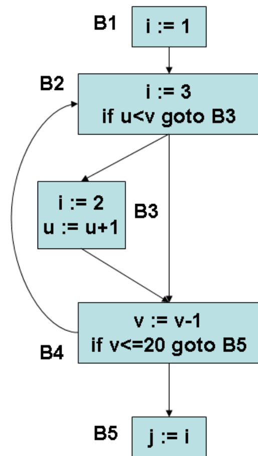
Code Motion - Violation of condition 2(a)



The statement i:=2 from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4)
The computation gets altered due to code movement
i always gets value 2, and never 1, and hence j always gets value 2

Condition 2(a):
s dominates all exits of *L*

Code Motion - Violation of condition 2(b)



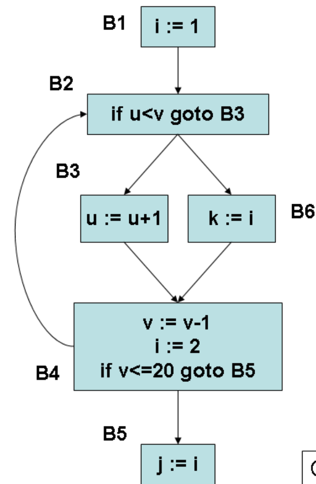
B2 dominates B4 and hence condition 2(a) is satisfied for i:=3 in B2. However statement i:=3 from B2 cannot be moved to a preheader since condition 2(b) is violated (i is defined in B3)

The computation gets altered due to code movement
If the loop is executed twice, i may pass its value of 3 from B2 to j in the original loop.
In the revised loop, i gets the value 2 in the second iteration and retains it forever

Condition 2(a):
s dominates all exits of *L*

Condition 2(b):
x is not defined elsewhere in *L*

Code Motion - Violation of condition 2(c)



Condition 2(a):
s dominates all exits of *L*

Conditions 2(a) and 2(b) are satisfied. However statement i:=2 from B4 cannot be moved to a preheader since condition 2(c) is violated (use of i in B6 is reached by defs of i in B1 and B4)

The computation gets altered due to code movement
In the revised loop, i gets the value 2 from the def in the preheader and k becomes 2.
However, k could have received the value of either 1 (from B1) or 2 (from B4) in the original loop

Condition 2(b): *x* is not defined elsewhere in *L*
Condition 2(c): All uses of *x* in *L* can only be reached by the definition of *x* in *s*

Induction Variables

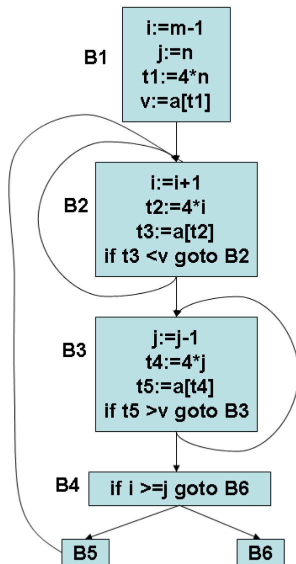
- An **induction variable** x of a loop L changes its value only through an increment or decrement operation by a constant amount
- **Basic induction variables**: variables i whose only assignments within a loop L are of the form $i := i \pm n$, where n is a constant
- Another variable j which is *defined only once* within L, and whose value is $c * i + d$ (linear function of i) is an $i.v.$ in the **family** of i
- We associate a triple (i, c, d) with j (c and d are constants), and i belongs to its own family with a triple $(i, 1, 0)$

Induction Variables - Example 1

```
        t1 = 202
        i = 1
        t3 = addr(a)
        t4 = t3 - 4
L1:     t2 = i>100
        if t2 goto L2
        t1 = t1-2
        t5 = 4*i
        t6 = t4+t5
        *t6 = t1
        i = i+1
        goto L1
L2:
```

i is a basic i.v. and t5 is a derived i.v. in the family of i

Induction Variables - Example 2



i and j are both basic i.v. in both inner and outer loops

t2 (in the family of i) and t4 (in the family of j) are both derived i.v. in both inner and outer loops

Detection of Induction Variables

We need a loop L, reaching definitions, and loop-invariant computation information

1. Find all the basic $i.v.$, by scanning the statements of L
2. Search for variables k, with a single assignment to k within L, having one of the following forms:
   $k := j * b,\ k := b * j,\ k := j/b,\ k := j \pm b,\ k := b \pm j,$ $k := j * b \pm a,\ k := a \pm j * b$, where b is a constant and j is an i.v., basic or otherwise
   (a) If j is basic, then for $k := j * b$, the triple for k is $(j, b, 0)$ (similarly for other forms)
   (b) If j is not basic, then let its triple be (i, c, d). We need to check two more conditions
      (i) there is no assignment to i between the lone point of assignment to j in L and the assignment to k
      (ii) no definition of j outside L reaches k