Introduction to Machine-Independent Optimizations - 6

Machine-Independent Optimization Algorithms

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2,3, and 4)
- Fundamentals of control-flow analysis (in parts 4 and 5)
- Algorithms for machine-independent optimizations
- SSA form and optimizations

Detection of Loop-invariant Computations

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, or 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"

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

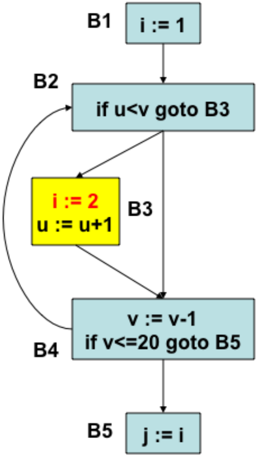
```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

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

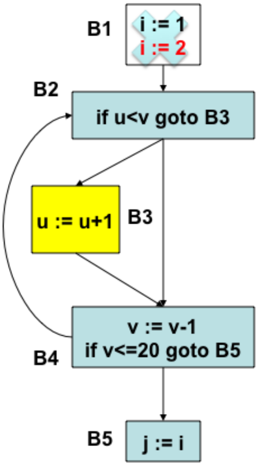
Code Motion - Violation of condition 2(a)-1



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*

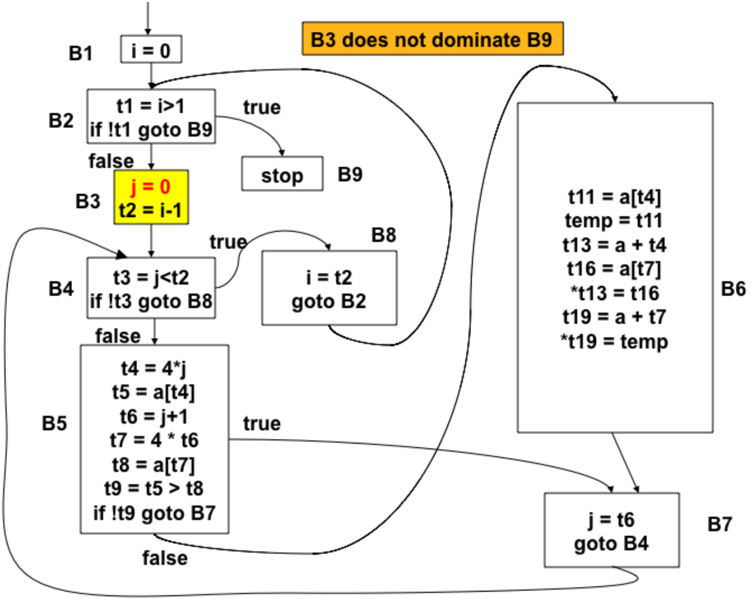
Code Motion - Violation of condition 2(a)-2



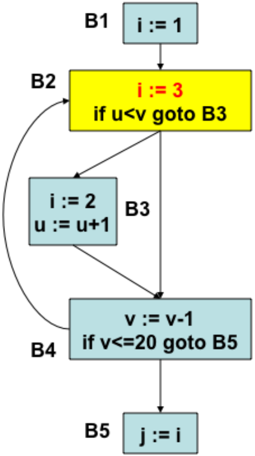
The statement i:=2 from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4)
The computation gets altered due to code movement
i always gets value 2, and never 1, and hence j always gets value 2

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

Violation of condition 2(a) - Running Example



Code Motion - Violation of condition 2(b)



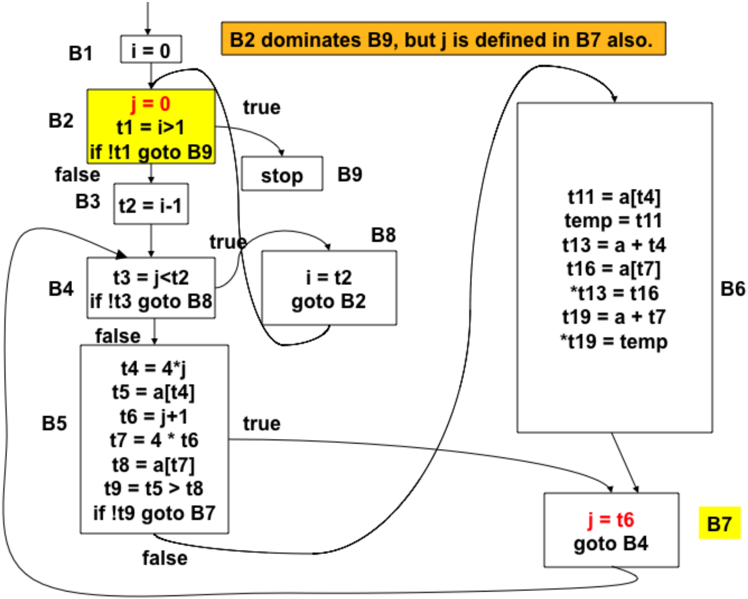
B2 dominates B4 and hence condition 2(a) is satisfied for i:=3 in B2. However statement i:=3 from B2 cannot be moved to a preheader since condition 2(b) is violated (i is defined in B3)

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

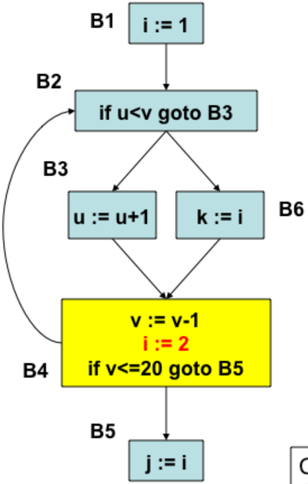
Condition 2(a):
s dominates all exits of *L*

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

Violation of condition 2(b) - Running Example



Code Motion - Violation of condition 2(c)

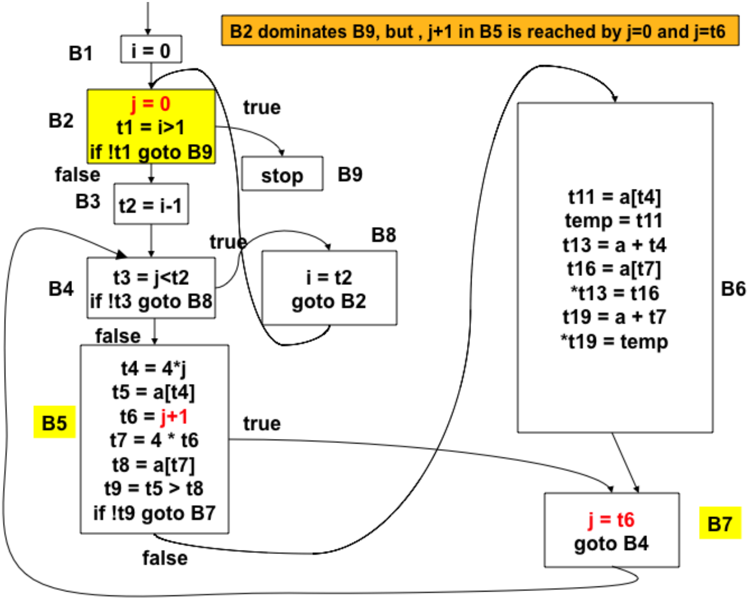


Conditions 2(a) and 2(b) are satisfied. However statement i:=2 from B4 cannot be moved to a preheader since condition 2(c) is violated (use of i in B6 is reached by defs of i in B1 and B4)

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

Violation of condition 2(c) - Running Example



The Static Single Assignment Form:

Application to Program Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

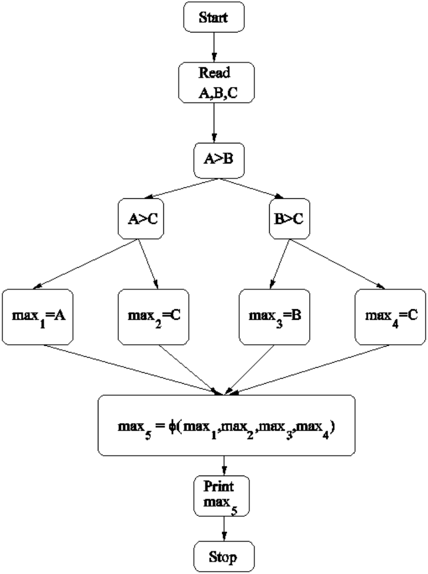
- SSA form - definition and examples
- Optimizations with SSA forms
  - Dead-code elimination
  - Simple constant propagation
  - Copy propagation
  - Conditional constant propagation and constant folding

The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a *static* single assignment form, and not a *dynamic* single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A *sparse* intermediate representation
  - If a variable has N uses and M definitions, then *def-use chains* need space and time proportional to $N.M$
  - But, the corresponding instructions of uses and definitions are only $N + M$ in number
  - SSA form, for most realistic programs, is linear in the size of the original program

A Program in non-SSA Form and its SSA Form

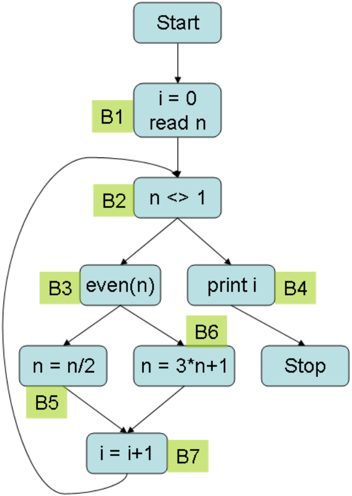
read A, B, C
if $(A > B)$
  if $(A > C)$ $max = A$
  else $max = C$
else if $(B > C)$ $max = B$
    else $max = C$
printf (max)

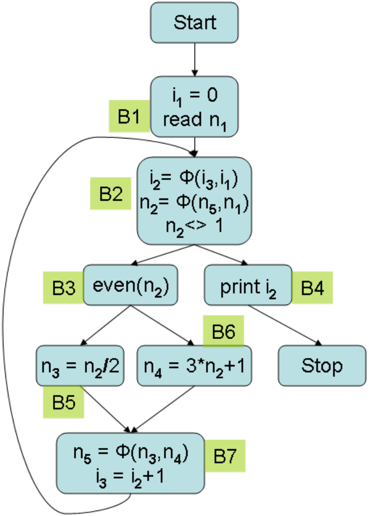


SSA Form: A Definition

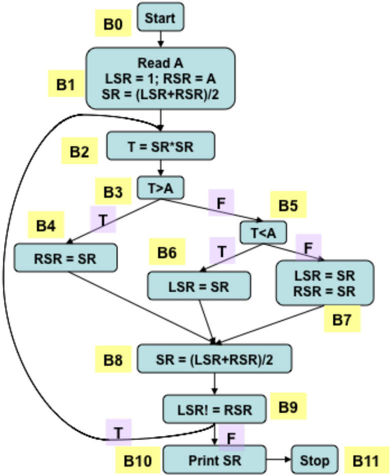
- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow of control remains the same as in the non-SSA form
- A special merge operator, ϕ, is used for selection of values in join nodes
- Not every join node needs a ϕ operator for every variable
- No need for a ϕ operator, if the same definition of the variable reaches the join node along all incoming edges
- Often, an SSA form is augmented with *u-d* and *d-u* chains to facilitate design of faster algorithms
- Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency

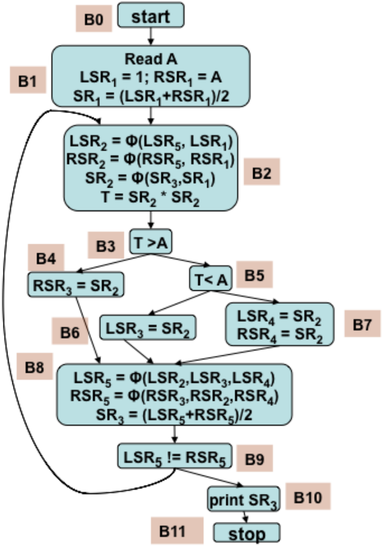
Program 2 in non-SSA and SSA Form





Program 3 in non-SSA and SSA Form





Optimization Algorithms with SSA Forms

- Dead-code elimination
  - Very simple, since there is exactly one definition reaching each use
  - Examine the *du-chain* of each variable to see if its use list is empty
  - Remove such variables and their definition statements
  - If a statement such as $x = y + z$ (or $x = \phi(y_1, y_2)$) is deleted, care must be taken to remove the deleted statement from the *du-chains* of y and z (or y_1 and y_2)
- Simple constant propagation
- Copy propagation
- Conditional constant propagation and constant folding
- Global value numbering

Simple Constant Propagation

```
{   Stmtpile = {S|S is a statement in the program}
    while Stmtpile is not empty {
        S = remove(Stmtpile);
        if S is of the form x = φ(c, c, ..., c) for some constant c
                replace S by x = c
        if S is of the form x = c for some constant c
                delete S from the program
                for all statements T in the du-chain of x do
                        substitute c for x in T; simplify T
                        Stmtpile = Stmtpile ∪ {T}
}
```

Copy propagation is similar to constant propagation

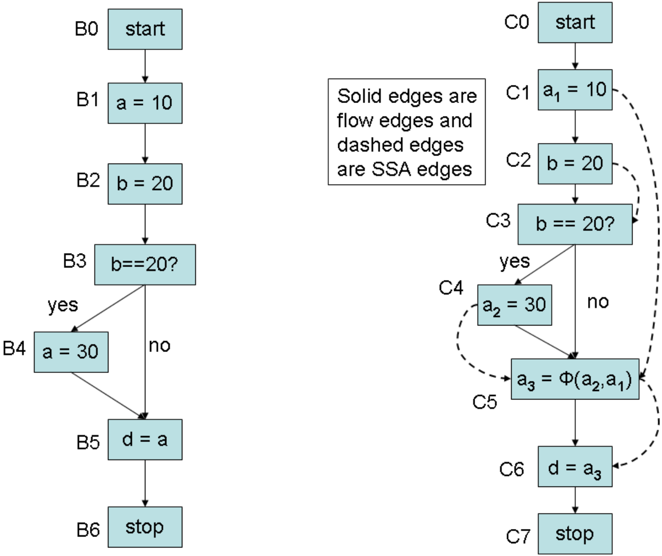
- A single-argument ϕ-function, $x = \phi(y)$, or a copy statement, $x = y$ can be deleted and y substituted for every use of x

Conditional Constant Propagation - 1

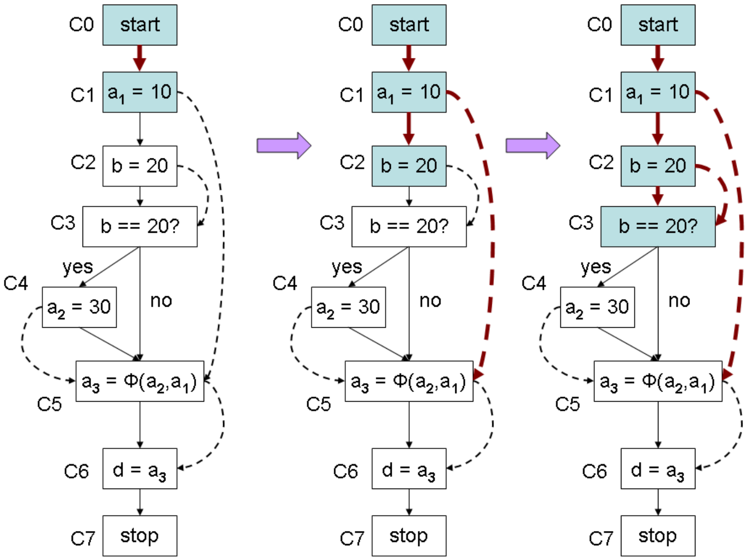
- SSA forms along with extra edges corresponding to *d-u* information are used here
  - Edge from every definition to each of its uses in the SSA form (called henceforth as *SSA edges*)
- Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile* , resp.)
- Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values
- Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor

Conditional Constant Propagation - 2

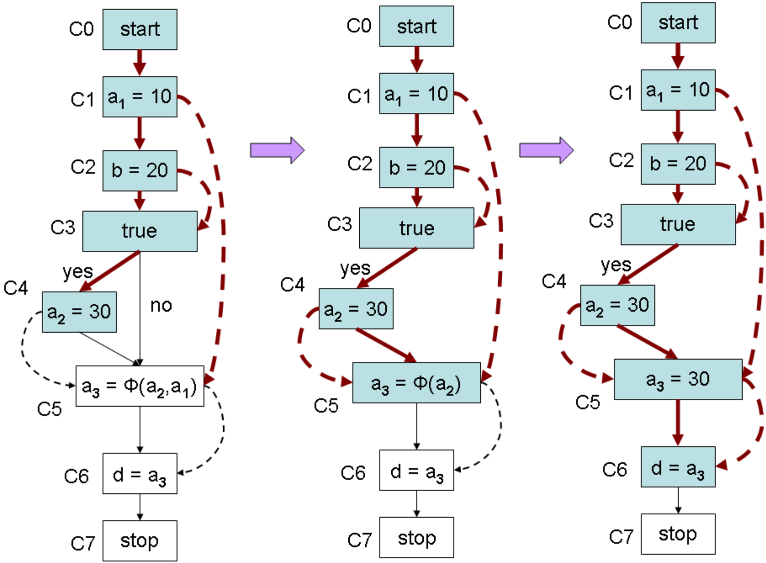
- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs much lesser storage compared to its non-SSA counterpart
- Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to *true* or *false*) or both edges (corresponding to $\perp$) are added to the worklist
- However, at any join node, the *meet* operation considers only those predecessors which are marked *executable*.


B0
start
B1
a = 10
B2
b = 20
B3
b==20?
yes
no
B4
a = 30
B5
d = a
B6
stop
Solid edges are flow edges and dashed edges are SSA edges
C0
start
C1
a_1 = 10
C2
b = 20
C3
b == 20?
yes
no
C4
a_2 = 30
C5
$a_3 = \Phi(a_2, a_1)$
C6
d = a_3
C7
stop

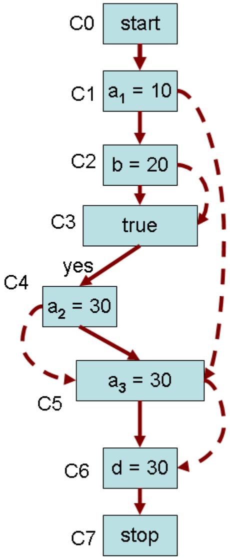
CCP Algorithm - Example 1 - Trace 1



CCP Algorithm - Example 1 - Trace 2



CCP Algorithm - Example 1 - Trace 3



CCP Algorithm - Example 2