Compilers (CS30003)

Lecture 29-30

Pralay Mitra

Autumn 2023-24

Control Flow Graph and Local Optimization

- What is code optimization and why is it needed?
- Types of optimizations
- Basic blocks and control flow graphs
- Local optimizations
- Building a control flow graph
- Directed acyclic graphs and value numbering

Machine Independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)
- Optimizations may be classified as *local* and *global*

Example: Vector Product

```
int a[5], b[5], c[5];
int i, n = 5;

for(i = 0; i < n; i++) {
    if (a[i] < b[i])
        c[i] = a[i] * b[i];
    else
        c[i] = 0;
}
return;
```

```
// int i, n = 5;
100: t1 = 5
101: n = t1
// for(i = 0; i < n; i++) {
102: t2 = 0
103: i = t2
104: if i < n goto 109 // T
105: goto 129 // F
106: t3 = i
107: i = i + 1
108: goto 104
// if (a[i] < b[i])
109: t4 = 4 * i
110: t5 = a[t4]
111: t6 = 4 * i
112: t7 = b[t6]
113: if t5 < t7 goto 115 // T
114: goto 124 // F
```

```
// c[i] = a[i] * b[i];
115: t8 = 4 * i
116: t9 = c + t8
117: t10 = 4 * i
118: t11 = a[t10]
119: t12 = 4 * i
120: t13 = b[t12]
121: t14 = t11 * t13
122: *t9 = t14
123: goto 106 // next
// c[i] = 0;
124: t15 = 4 * i
125: t16 = c + t15
126: t17 = 0
127: *t16 = t17
// }
128: goto 106 // for
// return;
129: return
```

Local Optimization

Local optimization: within basic blocks

- Local Common Sub-Expression (LCSE) elimination
- Constant propagation and constant folding
- Eliminating local def-use of temporary
- Dead-code elimination
- Reordering computations using algebraic laws
- Eliminating redundant instructions

Global Optimization

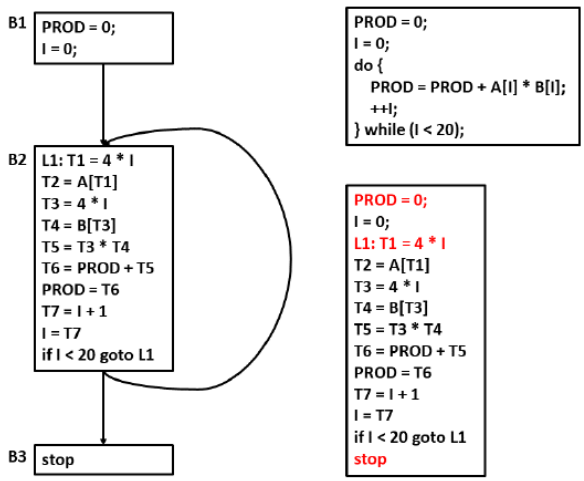
Global optimization: on whole procedures/programs

- Global Common Sub-Expression (GCSE) elimination
- Constant propagation and constant folding
- Eliminating unreachable code
- Eliminating jumps over jumps
- Eliminating jumps to jumps (chain of jumps)
- Eliminating def-use of temporary
- Eliminating redundant instructions
- Loop invariant code motion
- Partial redundancy elimination
- Loop unrolling and function inlining
- Vectorization and Concurrentization

Basic Blocks and Control-Flow Graphs

- **Basic Blocks** (BB) are sequences of intermediate code with a *single entry* and a *single exit*
- **Control Flow Graphs** (CFG) show control flow among basic blocks
- Basic blocks are represented as **Directed Acyclic Graphs** (DAGs), which are in turn represented using the value-numbering method applied on quadruples
- Optimizations on basic blocks

Example of Basic Blocks and Control Flow Graph



Basic Blocks

Algorithm: Partition IR (three-address instructions) into basic blocks

- **Input:** A sequence of IR (three-address instructions)
- **Output:** A list of basic blocks in which each instruction is assigned to exactly one basic block

- **Method:**

Identify the leaders using following rules

1. First three address code in the IR is a leader.
2. Any instruction that is the target of the condition/unconditional jump is a leader.
3. Any instruction that immediately follows a condition/unconditional jump is a leader.

For each leader, its basic block consists of itself and all the instructions up to but not including the next leader or the end of IR.

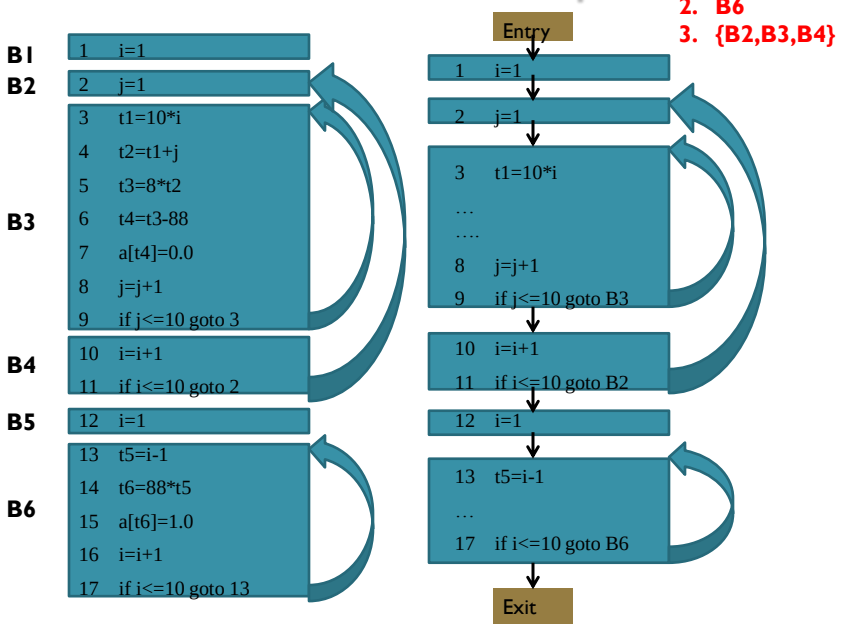
Basic Blocks

B1
```
1  i=1
```
B2
```
2  j=1
```
B3
```
3  t1=10*i
4  t2=t1+j
5  t3=8*t2
6  t4=t3-88
7  a[t4]=0.0
8  j=j+1
9  if j<=10 goto 3
```
B4
```
10 i=i+1
11 if i<=10 goto 2
```
B5
```
12 i=1
```
B6
```
13 t5=i-1
14 t6=88*t5
15 a[t6]=1.0
16 i=i+1
17 if i<=10 goto 13
```

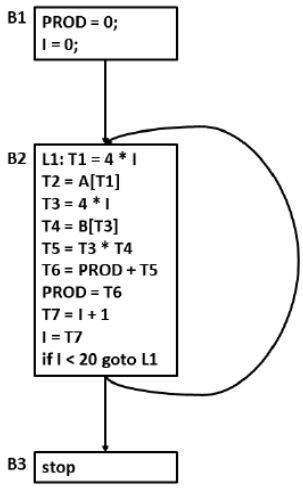
Next use information

Basic Blocks to Flow Graphs

Loops???
1. **B3**
2. **B6**
3. **{B2,B3,B4}**



Example of Basic Blocks and CFG




```
PROD = 0;
I = 0;
do {
   PROD = PROD + A[I] * B[I];
   ++I;
} while (I < 20);
```

```
PROD = 0;
I = 0;
L1: T1 = 4 * I
T2 = A[T1]
T3 = 4 * I
T4 = B[T3]
T5 = T3 * T4
T6 = PROD + T5
PROD = T6
T7 = I + 1
I = T7
if I < 20 goto L1
stop
```

Control Flow Graph

- The nodes of the CFG are basic blocks
- One node is distinguished as the initial node
- There is a directed edge $B1 \longrightarrow B2$, if B2 can immediately follow B1 in some execution sequence:
  - There is a conditional or unconditional jump from the last statement of B1 to the first statement of B2, or
  - B2 immediately follows B1 in the order of the program, and B1 does not end in an unconditional jump
- A basic block is represented as a record consisting of
  [1] a count of the number of quads in the block
  [2] a pointer to the leader of the block
  [3] pointers to the predecessors of the block
  [4] pointers to the successors of the block

Note: *Jump statements point to basic blocks and not* quads *so as to make code movement easy*

Example: Vector Product: Control Flow Graph

[1] First quad of the program

[2] quad's as target of some goto

[3] quad's following a conditional goto

```
100: n = 5                  [1]
101: i = 0
102: if i < n goto 106      [2]
103: goto 124               [3]
104: i = i + 1              [2]
105: goto 102
106: t4 = 4 * i             [2]
107: t5 = a[t4]
108: t6 = 4 * i
109: t7 = b[t6]
110: if t5 >= t7 goto 120
```

```
111: t8 = 4 * i             [3]
112: t9 = c + t8
113: t10 = 4 * i
114: t11 = a[t10]
115: t12 = 4 * i
116: t13 = b[t12]
117: t14 = t11 * t13
118: *t9 = t14
119: goto 104
120: t15 = 4 * i            [2]
121: t16 = c + t15
122: *t16 = 0
123: goto 104
124: return                 [2]
```

Example: Vector Product: Control Flow Graph

Control Flow Graph is shown below:

```
// Block B1
0: 100: n = 5
1: 101: i = 0
 :     : goto B2 [Fall through]

// Block B2
0: 102: if i < n goto B4 [106]
 : 103: goto B7 [124]

// Block B3
0: 104: i = i + 1
 : 105: goto B2 [102]

// Block B4
0: 106: t4 = 4 * i
1: 107: t5 = a[t4]
2: 108: t6 = 4 * i
3: 109: t7 = b[t6]
4: 110: if t5 >= t7 goto B6 [120]
 :     : goto B5 [Fall through]
```

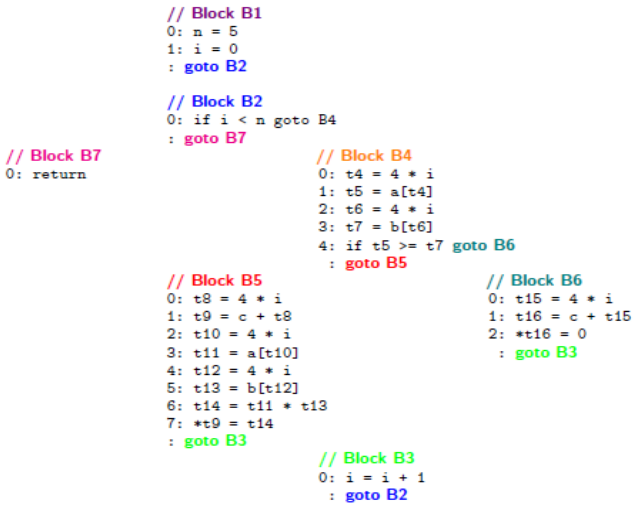
```
// Block B5
0: 111: t8 = 4 * i
1: 112: t9 = c + t8
2: 113: t10 = 4 * i
3: 114: t11 = a[t10]
4: 115: t12 = 4 * i
5: 116: t13 = b[t12]
6: 117: t14 = t11 * t13
7: 118: *t9 = t14
 : 119: goto B3 [104]

// Block B6
0: 120: t15 = 4 * i
1: 121: t16 = c + t15
2: 122: *t16 = 0
 : 123: goto B3 [104]

// Block B7
0: 124: return
```

There is no unreachable quad to remove.

Example: Vector Product: Control Flow Graph: Graphical Depiction

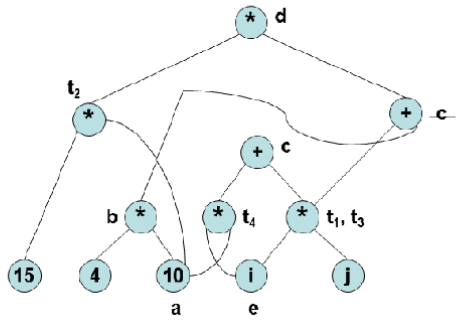


Optimization of Basic Blocks Directed Acyclic Graph (DAG) Representation

```
1: a = 10
2: b = 4 * a
3: t1 = i * j
4: c = t1 + b
5: t2 = 15 * a
6: d = t2 * c
7: e = i
8: t3 = e * j
9: t4 = i * a
10: c = t3 + t4
```

Optimization of Basic Blocks
Directed Acyclic Graph (DAG) Representation

```
1: a = 10
2: b = 4 * a
3: t1 = i * j
4: c = t1 + b
5: t2 = 15 * a
6: d = t2 * c
7: e = i
8: t3 = e * j
9: t4 = i * a
10: c = t3 + t4
```



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numb* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such that two expressions receive the same number if the compiler can prove that th equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values: *HashTable*, *ValnumTable*, and *NameTable*

Value Numbering in Basic Blocks

- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one

Data Structures for Value Numbering

In the field *Namelist*, first name is the defining occurrence and replaces all other names with the same value number with itself (or its constant value)

ValueNumber Table (VNT) Entry
Indexed by Name Hash Value

Name	Value Number

Name Table (NT) Entry
Indexed by Value Number

Name List	Constant Value	Constant Flag

Hash Table (HT) Entry
Indexed by Expression Hash Value

Expression	Value Number

Example of Value Numbering

HLL Program	Quad's before value-numbering	Quad's after value-numbering
a = 10 b = 4 * a c = i * j + b d = 15 * a * c e = i c = e * j + i * a	01. a = 10 02. b = 4 * a 03. t1 = i * j 04. c = t1 + b 05. t2 = 15 * a 06. d = t2 * c 07. e = i 08. t3 = e * j 09. t4 = i * a 10. c = t3 + t4	a = 10 b = 40 t1 = i * j c = t1 + 40 t2 = 150 d = 150 * c e = i t3 = i * j t4 = i * 10 c = t1 + t4 Quad's 5 & 8 can be deleted

Example of Value Numbering

VN Table

Name	VN
a	1
b	2
i	3
j	4
t1	5
c	6, 10
t2	7
d	8
e	3
t3	5
t4	9

Name Table

Index	Name	Val
1	a	10
2	b	40
3	i, e	
4	j	
5	t1, t3	
6	c	
7	t2	150
8	d	
9	t4	
10	c	

Hash Table

Expr	VN
i * j	5
t1 + 40	6
150 * c	8
i * 10	9
t1 + t4	10

01. a = 10	a = 10
02. b = 4 * a	b = 40
03. t1 = i * j	t1 = i * j
04. c = t1 + b	c = t1 + 40
05. t2 = 15 * a	t2 = 150
06. d = t2 * c	d = 150 * c
07. e = i	e = i
08. t3 = e * j	t3 = i * j
09. t4 = i * a	t4 = i * 10
10. c = t3 + t4	c = t1 + t4

Running the algorithm through the example (1)

[1] $a = 10$:
- a is entered into *ValnumTable* (with a *vn* of 1, say) and into *NameTable* (with a constant value of 10)

[2] $b = 4 * a$:
- a is found in *ValnumTable*, its constant value is 10 in *NameTable*
  - We have performed *constant propagation*
  - $4 * a$ is evaluated to 40, and the quad is rewritten
  - We have now performed *constant folding*
  - b is entered into *ValnumTable* (with a *vn* of 2) and into *NameTable* (with a constant value of 40)

[3] $t1 = i * j$:
- i and j are entered into the two tables with new *vn* (as above), but with no constant value
- $i * j$ is entered into *HashTable* with a new *vn*
- $t1$ is entered into *ValnumTable* with the same *vn* as $i * j$

Running the algorithm through the example (2)

[4] Similar actions continue till $e = i$
- e gets the same *vn* as i

[5] $t3 = e * j$:
- e and i have the same *vn*
- hence, $e * j$ is detected to be the same as $i * j$
- since $i * j$ is already in the HashTable, we have found a *common subexpression*
- from now on, all uses of $t3$ can be replaced by $t1$
- quad $t3 = e * j$ can be deleted

[6] $c = t3 + t4$:
- $t3$ and $t4$ already exist and have *vn*
- $t3 + t4$ is entered into *HashTable* with a new *vn*
- this is a reassignment to c
- c gets a different *vn*, same as that of $t3 + t4$

[7] Quads are renumbered after deletions

Example of Value Numbering

If the same code snippet is translated by our automated scheme, we shall get a more verbose 3 address code. Here we show that this auto-translated code too gets optimized to the same as before

HLL Program	Quad's before value-numbering	Quad's after value-numbering
a = 10 b = 4 * a c = i * j + b d = 15 * a * c e = i c = e * j + i * a	01. a = 10 02. t1 = 4 * a 03. b = t1 04. t2 = i * j 05. t3 = t2 + b 06. c = t3 07. t4 = 15 * a 08. t5 = t4 * c 09. d = t5 10. e = i 11. t6 = e * j 12. t7 = i * a 13. t8 = t6 + t7 14. c = t8	01. a = 10 02. t1 = 40 03. b = 40 04. t2 = i * j 05. t3 = t2 + 40 06. c = t3 07. t4 = 150 08. t5 = 150 * t3 09. d = t5 10. e = i 11. t6 = i * j 12. t7 = i * 10 13. t8 = t2 + t7 14. c = t8 • Quad's 2, 6, 7 & 11 can be deleted • Copy can be propagated (in reverse) to eliminate t5 (between 8 & 9) and t8 (between 13 & 14) • Note that e in 10 cannot be removed as it may be used outside the block

Example of Value Numbering

VN Table

Name	VN
a	1
t1	2
b	2
i	3
j	4
t2	5
t3	6
c	6
t4	7
t5	8
d	8
e	3
t6	5
t7	9
t8	10
c	10

Hash Table

Expr	VN
i * j	5
t2 + 40	6
150 * t3	8
i * 10	9
t6 + t7	10

Name Table

Index	Name	Val
1	a	10
2	t1, b	40
3	i, e	
4	j	
5	t2, t6	
6	t3, c	
7	t4	150
8	t5, d	
9	t7	
10	t8, c	

Example: Vector Product: LCSE

We need to perform LCSE step for blocks:

```
B4:
// if (a[i] < b[i]) {
// Block B4
0: t4 = 4 * i
1: t5 = a[t4]
2: t6 = 4 * i
3: t7 = b[t6]
4: if t5 >= t7 goto B6
 : goto B5
```

and

```
B5:
// c[i] = a[i] * b[i];
// Block B5
0: t8 = 4 * i
1: t9 = c + t8
2: t10 = 4 * i
3: t11 = a[t10]
4: t12 = 4 * i
5: t13 = b[t12]
6: t14 = t11 * t13
7: *t9 = t14
 : goto B3
```

Homework: Perform LCSE on B4 and B5

Handling Commutativity etc.

- When a search for an expression $i + j$ in *HashTable* fails, try for $j + i$
- If there is a quad $x = i + 0$, replace it with $x = i$
- Any quad of the type, $y = j * 1$ can be replaced with $y = j$
- After the above two types of replacements, value numbers of x and y become the same as those of i and j, respectively
- Quads whose LHS variables are used later can be marked as *useful*
- All unmarked quads can be deleted at the end

Handling Array References

Consider the sequence of quads:

[1] $X = A[i]$

[2] $A[j] = Y$: i and j could be the same

[3] $Z = A[i]$: in which case, $A[i]$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = A[i]$; $A[j] = Y$; $Z = X$
- When $A[j] = Y$ is processed during value numbering, ALL references to array A so far are searched in the tables and are marked KILLED - this kills quad 1 above
- When processing $Z = A[i]$, killed quads not used for CSE
- Fresh table entries are made for $Z = A[i]$
- However, if we know apriori that $i \neq j$, then $A[i]$ can be used for CSE

Handling Pointer References

Consider the sequence of quads:

[1] $X = *p$

[2] $*q = Y$: p and q could be pointing to the same object

[3] $Z = *p$: in which case, $*p$ is not a common sub-expression here

- The above sequence cannot be replaced by: $X = *p$; $*q = Y$; $Z = X$
- Suppose no pointer analysis has been carried out
  - p and q can point to *any* object in the basic block
  - Hence, When $*q = Y$ is processed during value numbering, ALL table entries created so far are marked KILLED - this kills quad 1 above as well
  - When processing $Z = *p$, killed quads not used for CSE
  - Fresh table entries are made for $Z = *p$

Handling Pointer References and Procedure Calls

- However, if we know apriori which objects p and q point to, then table entries corresponding to only those objects need to killed
- Procedure calls are similar
- With no dataflow analysis, we need to assume that a procedure call can modify any object in the basic block
  - changing call-by-reference parameters and global variables within procedures will affect other variables of the basic block as well
- Hence, while processing a procedure call, ALL table entries created so far are marked KILLED
- Sometimes, this problem is avoided by making a procedure call a separate basic block

Peephole Optimizations

- Simple but effective local optimization
- Usually carried out on machine code, but intermediate code can also benefit from it
- Examines a sliding window of code (peephole), and replaces it by a shorter or faster sequence, if possible
- Each improvement provides opportunities for additional improvements
- Therefore, repeated passes over code are needed

Peephole Optimizations

- Some well known peephole optimizations
  - eliminating redundant instructions
  - eliminating unreachable code
  - eliminating jumps over jumps
  - algebraic simplifications
  - strength reduction
  - use of machine idioms

Peephole Optimization

- Redundant instruction elimination
  - LD R0, a
  - ST a, R0

Is it always redundant?

- Algebraic simplification
- Machine Idioms

Peephole Optimization

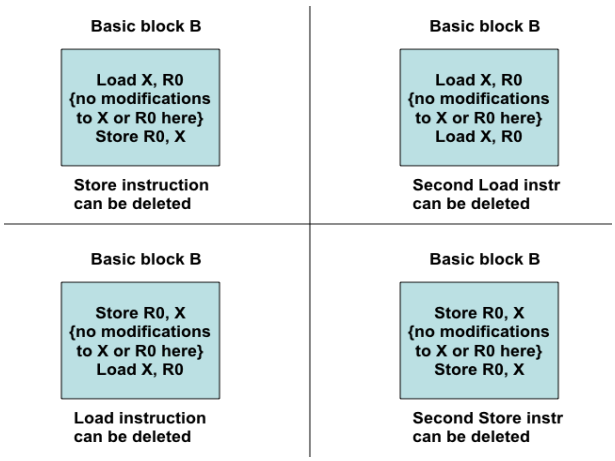
- Flow of control

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

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

```
        goto L1
        ……
L1:     goto L2
```

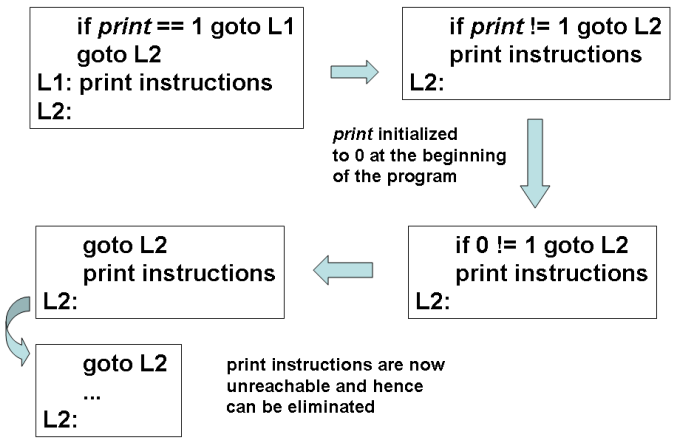
Elimination of Redundant Loads and Stores



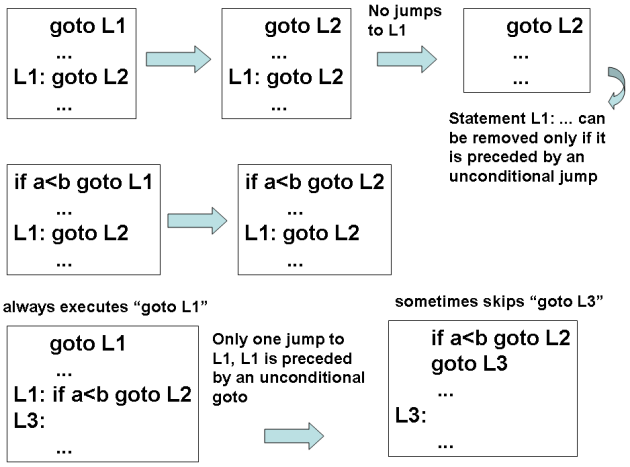
Eliminating Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed
  - May be produced due to debugging code introduced during development
  - Or due to updates to programs (changes for fixing bugs) without considering the whole program segment

Eliminating Unreachable Code



Flow-of-Control Optimizations



Compilers (CS30003)

Lecture 31

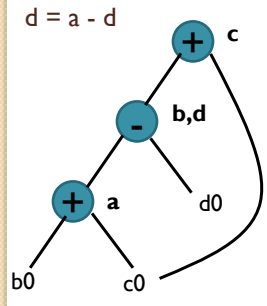
Pralay Mitra

Autumn 2023-24

Optimization of Basic Blocks

- Local Common Subexpression

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

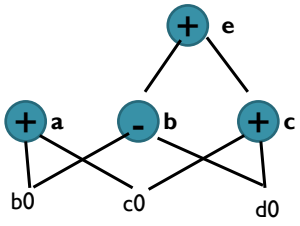


Is *b* live on exit?

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

e=b+c=(b-d)+(c+d)

Can you see?

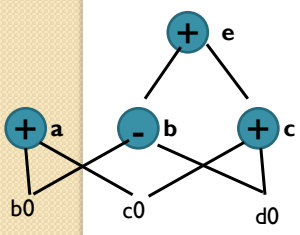


Optimization of Basic Blocks

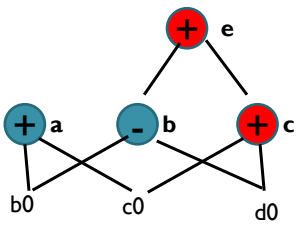
- Dead code elimination:

  Repeatedly delete root from a DAG that has no live variable attached.

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



e and c are not live on exit



Optimization of Basic Blocks

- Algebraic Identity and expression

Identity

X + 0 = 0 + X = X
X - 0 = X

X × 1 = 1 × X = X
X / 1 = X

Strength Reduction

$X^2 = X \times X$
$2 \times X = X + X$

Constant Folding

2 * 3.14 / 1.21

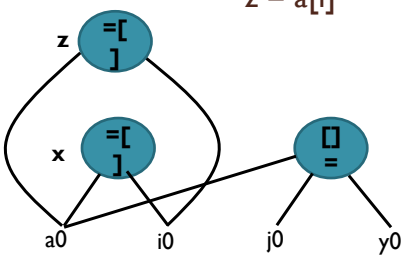
Generate IR

d = 2 * 3.14
a = b + c
e = c + d + b
f = 2 * 3.14 * a + e

Optimization of Basic Blocks

- Representation of array references

x = a[i]
a[j] = y
z = a[i]



b = 12 + a
x = b[i]
b[j] = y

DAG?
Can you kill any node?

Can I kill?

Reduction in Strength and Use of Machine Idioms

- x^2 is cheaper to implement as x*x, than as a call to an exponentiation routine
- For integers, $x*2^3$ is cheaper to implement as x << 3 (x left-shifted by 3 bits)
- For integers, $x/2^2$ is cheaper to implement as x >> 2 (x right-shifted by 2 bits)
- Floating point division by a constant can be approximated as multiplication by a constant
- Auto-increment and auto-decrement addressing modes can be used wherever possible
  - Subsume INCREMENT and DECREMENT operations (respectively)
- Multiply and add is a more complicated pattern to detect

Data Flow Analysis (DFA)

Data Flow Analysis

- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, ..., p_n$ such that for each $i = 1, 2, ..., n-1$, either
  1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  2. p_i is the end of some block and p_{i+1} is the beginning of a successor block
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path
- Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts
- No analysis is necessarily a perfect representation of the state

Path Examples

```
    p0
100: n = 5
    p1
101: i = 0
    p2
102: if i < n goto 106
    p3
103: goto 124
    p4
104: i = i + 1
    p5
105: goto 102
    p6
106: t4 = i << 2
    p7
107: t5 = a[t4]
    p8
```

```
    p8
108: t6 = i << 2
    p9
109: t7 = b[t6]
    p10
110: if t5 >= t7 goto 120
    p11
111: t8 = i << 2
    p12
112: t9 = c + t8
    p13
113: t10 = i << 2
    p14
114: t11 = a[t10]
    p15
115: t12 = i << 2
    p16
```

```
    p16
116: t13 = b[t12]
    p17
117: t14 = t11 * t13
    p18
118: *t9 = t14
    p19
119: goto 104
    p20
120: t15 = i << 2
    p21
121: t16 = c + t15
    p22
122: *t16 = 0
    p23
123: goto 104
    p24
124: return
```

Path-1: OUT[S] p0-p1-p2-p3-p24
Path-2: p0-p1-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2
Path-3: p0-p1-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2
Path-4: p0-p1-p2-p6-p7-p8-p9-p10-p20-p21-p22-p23-p4-p5-p2-p3-p24

Path Examples: Basic Block

```
                    p0: // Block B1
                      0: n = 5
                      1: i = 0
                    p1: // goto B2

                    p2: // Block B2
                      0: if i < n goto B4
                    p3: // goto B7
p12: // Block B7                        p4: // Block B4
  0: return                               0: t4 = 4 * i
p13:                                      1: t5 = a[t4]
                                          2: t6 = 4 * i
                                          3: t7 = b[t6]
                                          4: if t5 >= t7 goto B6
                                        p5: // goto B5
                    p6: // Block B5                          p8: // Block B6
                      0: t8 = 4 * i                           0: t15 = 4 * i
                      1: t9 = c + t8                          1: t16 = c + t15
                      2: t10 = 4 * i                          2: *t16 = 0
                      3: t11 = a[t10]                       p9: // goto B3
                      4: t12 = 4 * i
                      5: t13 = b[t12]
                      6: t14 = t11 * t13
                      7: *t9 = t14
                    p7: // goto B3
                                        p10: // Block B3
                                          0: i = i + 1
                                        p11: // goto B2

Path: p0-p1-p2-p4-p5-p8-p9-p10-p11-p2-p3-p12-p13
```

DFA Motivation

- Program debugging
  - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*
- Program optimizations
  - Constant folding
  - Copy propagation
  - Common sub-expression elimination etc.

DFA Steps

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point
- The set of all possible data-flow values is the *domain* for the application under consideration
  - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of definitions in the program
  - A particular data-flow value is a set of definitions
- $IN[s]$ and $OUT[s]$: data-flow values *before* and *after* each statement s
- The *data-flow problem* is to find a solution to a set of constraints on $IN[s]$ and $OUT[s]$, for all statements s

DFA Steps

- Two kinds of constraints
  - Those based on the semantics of statements (*transfer functions*)
  - Those based on flow of control
- A DFA schema consists of
  - A control-flow graph
  - A direction of data-flow (forward or backward)
  - A set of data-flow values
  - A confluence operator (usually set union or intersection)
  - Transfer functions for each block
- We always compute *safe* estimates of data-flow values
- A decision or estimate is *safe* or *conservative*, if it never leads to a change in what the program computes (after the change)
- These safe values may be either subsets or supersets of actual values, based on the application

Reaching Definitions

Reaching Definitions (RD) Problem

- We *kill* a definition of a variable *a*, if between two points along the path, there is an assignment to *a*
- A definition *d* reaches a point *p*, if there is a path from the point immediately following *d* to *p*, such that *d* is not *killed* along that path
- Unambiguous and ambiguous definitions of a variable

```
        a := b+c
(unambiguous definition of 'a')
        ...
        *p := d
(ambiguous definition of 'a', if 'p' may point to variables
other than 'a' as well; hence does not kill the above
definition of 'a')
        ...
        a := k-m
(unambiguous definition of 'a'; kills the above definition of 'a')
```

RD Problem

- We compute supersets of definitions as *safe* values
- It is safe to assume that a definition reaches a point, even if it does not.
- In the following example, we assume that both `a=2` and `a=4` reach the point after the complete if-then-else statement, even though the statement `a=4` is not reached by control flow

```
if (a==b) a=2; else if (a==b) a=4;
```

Reaching Definitions: How to use them?

- Build *use / def Chains*
- *Constant Propagation*: For a use like

```
n: x = ... v ...
```

if all definitions that reach *n* are of the form

```
d: v = c // c is a constant
```

we can replace *v* in *n* by *c*

- *Un-initialized Variables*: How to detect?
- *Loop-invariant Code Motion*: For

```
d1: a = . . .;
d2: b = . . .;
for (. . .) {
    . . .
    n: x = a + b;
    . . .
}
```

if all definitions of variables on RHS of *n* and that reach *n* are outside the loop like *d1* and *d2*, *n* can also be moved outside the loop

RD Analysis: GEN and KILL

In other blocks:

d5: b = a+4
d6: f = e+c
d7: e = b+d
d8: d = a+b
d9: a = c+f
d10: c = e+a

d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

B

Set of all definitions = {d1,d2,d3,d4,d5,d6,d7,d8,d9,10}

GEN[B] = {d2,d3,d4}
KILL[B] = {d4,d9,d5,d10,d1}

Compilers (CS30003)

Lecture 32-34

Pralay Mitra

Autumn 2023-24

RD Problem

- The data-flow equations (constraints)

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

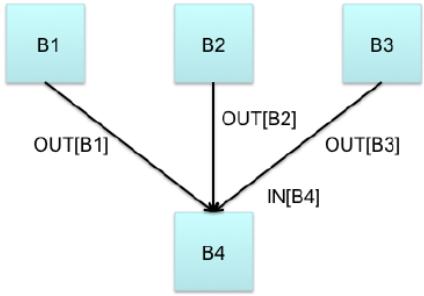
$$IN[B] = \phi, \text{for all } B \text{ (initialization only)}$$

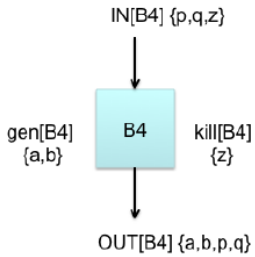
- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
  - Direction of flow does not imply traversing the basic blocks in a particular order
  - The final result does not depend on the order of traversal of the basic blocks

RD Problem

- $GEN[B]$ = set of all definitions inside B that are "visible" immediately after the block - *downwards exposed* definitions
  - If a variable x has two or more defintions in a basic block, then only the last definition of x is downwards exposed; all others are not visible outside the block
- $KILL[B]$ = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in B
  - If a variable x has a definition d_i in a basic block, then d_i kills all the definitions of the variable x in the program, except d_i

RD Analysis: DF Equations





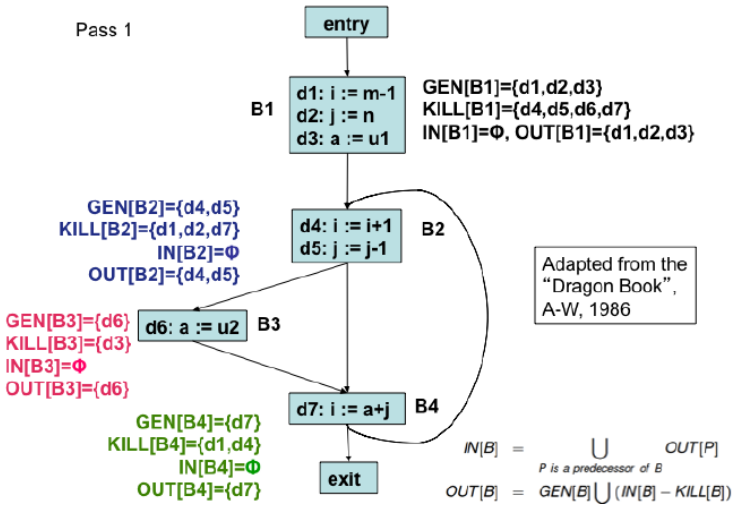
IN[B4] = OUT[B1] ∪ OUT[B2] ∪ OUT[B3]

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

OUT[B4] = gen[B4] ∪ (IN[B4] – kill[B4])

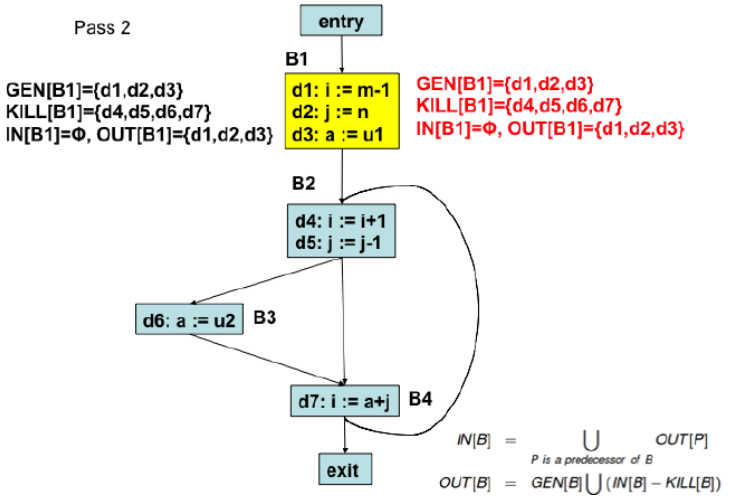
RD Analysis: An example



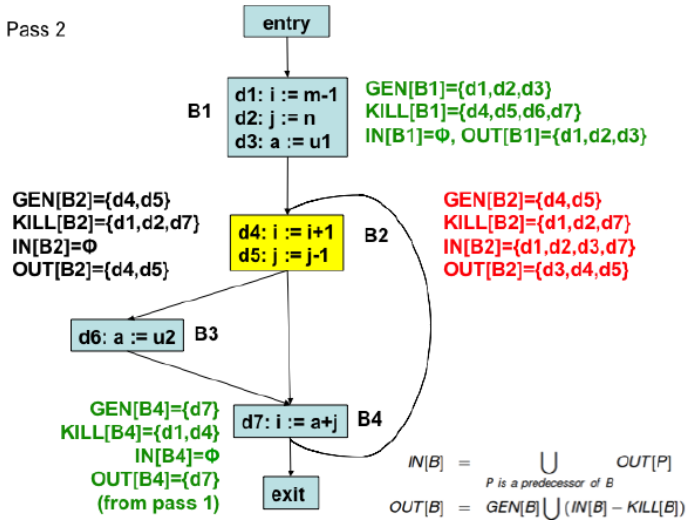
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

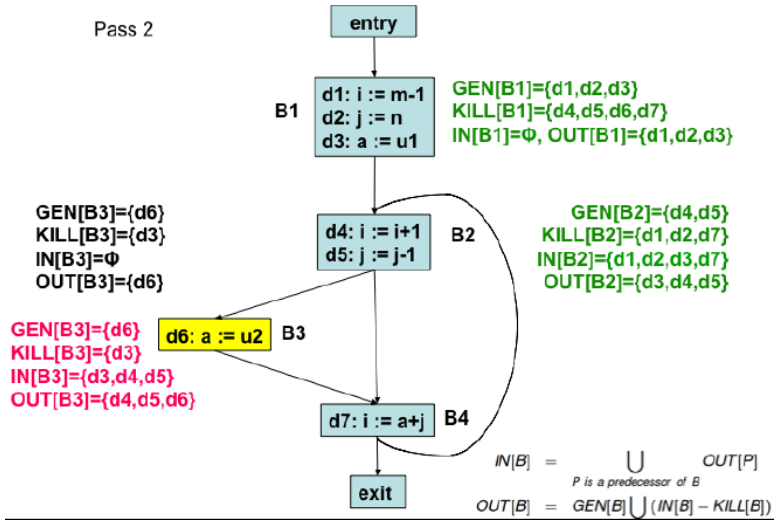
RD Analysis: An example



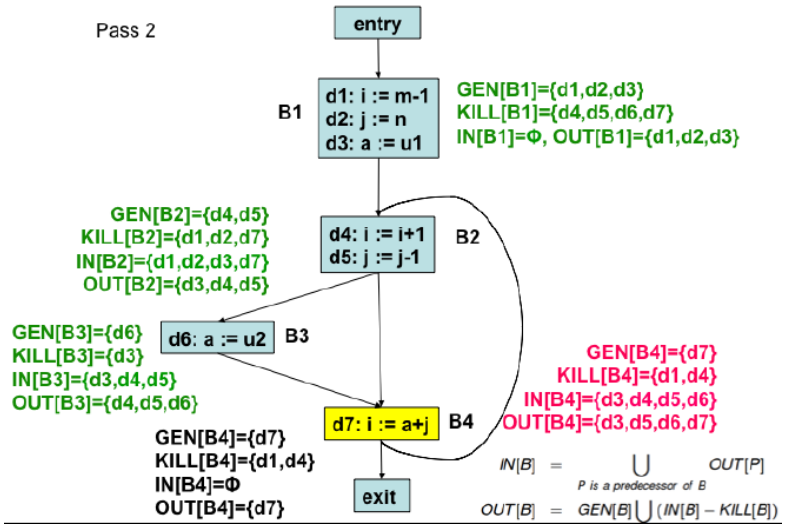
RD Analysis: An example



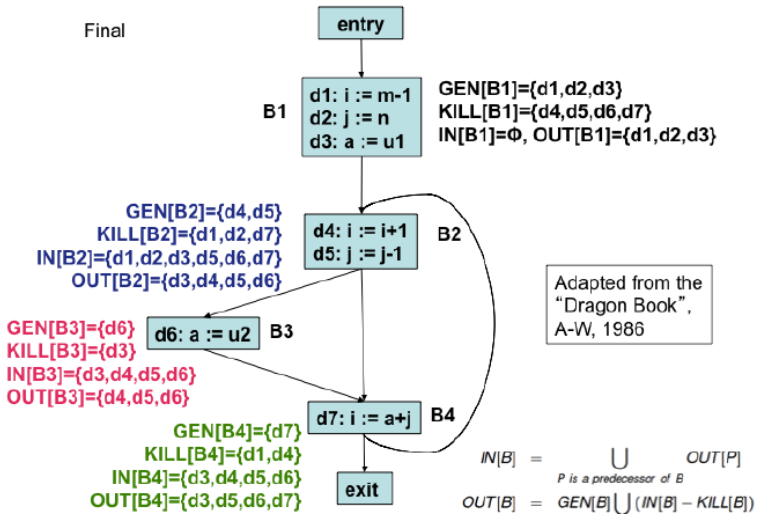
RD Analysis: An example



RD Analysis: An example



RD Analysis: An example



Computing RD

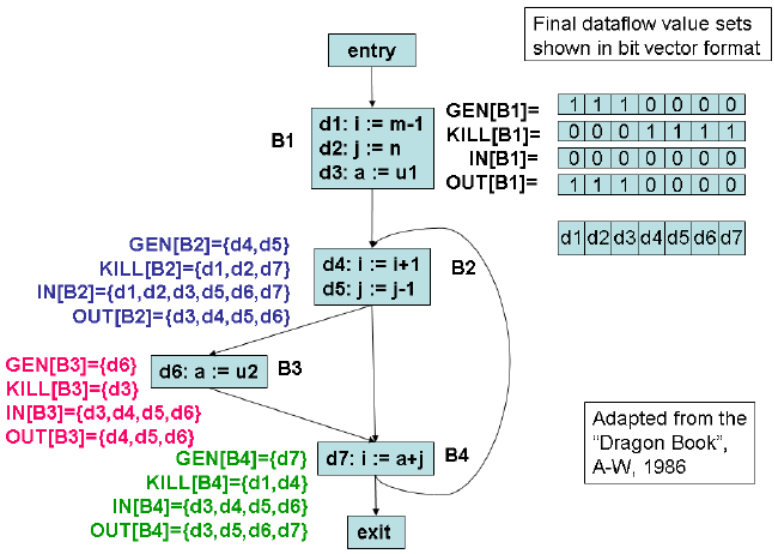
for each block B do { $IN[B] = \phi$; $OUT[B] = GEN[B]$; }
$change = true$;
while $change$ do { $change = false$;
  for each block B do {

$$IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P];$$
$$oldout = OUT[B];$$
$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B]);$$

    if ($OUT[B] \neq oldout$) $change = true$;
  }
}

- GEN, $KILL$, IN, and OUT are all represented as bit vectors with one bit for each definition in the flow graph

RD: Bit vector representation



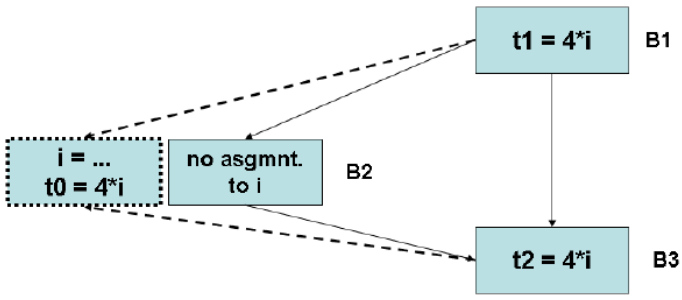
Available Expressions

DFA: Available expression

- Sets of expressions constitute the domain of data-flow values
- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not necessarily cycle-free) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it assigns (or may assign) to x or y and does not subsequently recompute $x + y$.
- A block *generates* $x + y$, if it definitely evaluates $x + y$, and does not subsequently redefine x or y

DFA: Available expression

- Useful for global common sub-expression elimination
- $4 * i$ is a CSE in $B3$, if it is available at the entry point of $B3$ *i.e.*, if i is not assigned a new value in $B2$ or $4 * i$ is



DFA: Available expression
(e_gen and e_kill)

- For statements of the form $x = a$, step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

e_gen[q] = A

q ▪
x = y + z
p ▪

Computing e_gen[p]

1. A = A **U** {y+z}
2. A = A – {all expressions involving x}
3. e_gen[p] = A

e_kill[q] = A

q ▪
x = y + z
p ▪

Computing e_kill[p]

1. A = A – {y+z}
2. A = A **U** {all expressions involving x}
3. e_kill[p] = A

DFA: Available expression
(e_gen and e_kill)

In other blocks:

d5: b = a+4
d6: f = e+c
d7: e = b+d
d8: d = a+b
d9: a = c+f
d10: c = e+a

d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

B

Set of all expressions = {f+1,a+7,b+d,d+c,a+4,e+c,a+b,c+f,e+a}

EGEN[B] = {f+1,b+d,d+c}
EKILL[B] = {a+4,a+b,e+a,e+c,c+f,a+7}

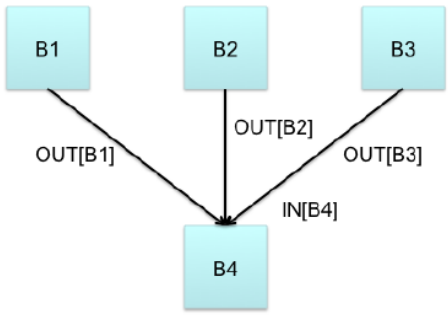
DFA: Available expression (DF equations)

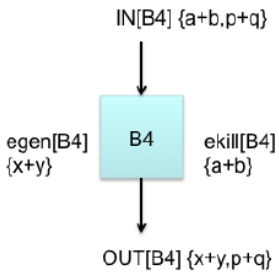
- The data-flow equations

$$
\begin{aligned}
IN[B] &= \bigcap_{P \text{ is a predecessor of } B} OUT[P], \ B \text{ not initial} \\
OUT[B] &= e\_gen[B] \bigcup (IN[B] - e\_kill[B]) \\
IN[B1] &= \phi \\
IN[B] &= U, \text{ for all } B \neq B1 \text{ (initialization only)}
\end{aligned}
$$

- $B1$ is the intial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$ is always ϕ
- U is the universal set of all expressions
- Initializing $IN[B]$ to ϕ for all $B \neq B1$, is restrictive

DFA: Available expression (DF equations)



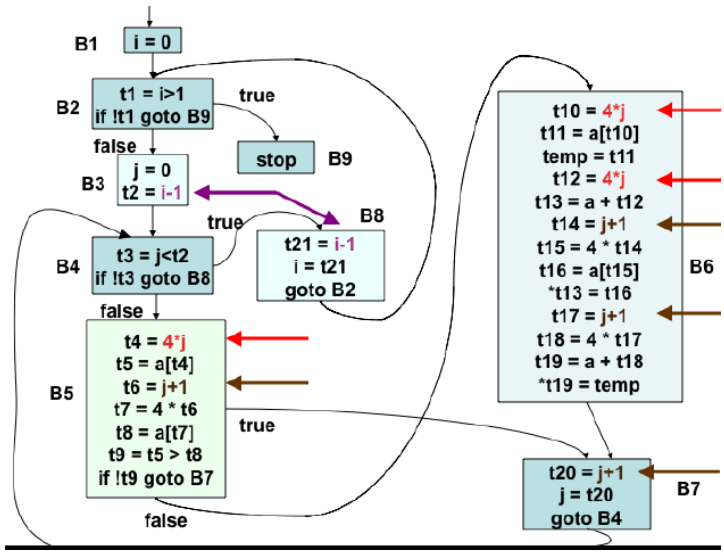


IN[B4] = OUT[B1] ⋂ OUT[B2] ⋂ OUT[B3]

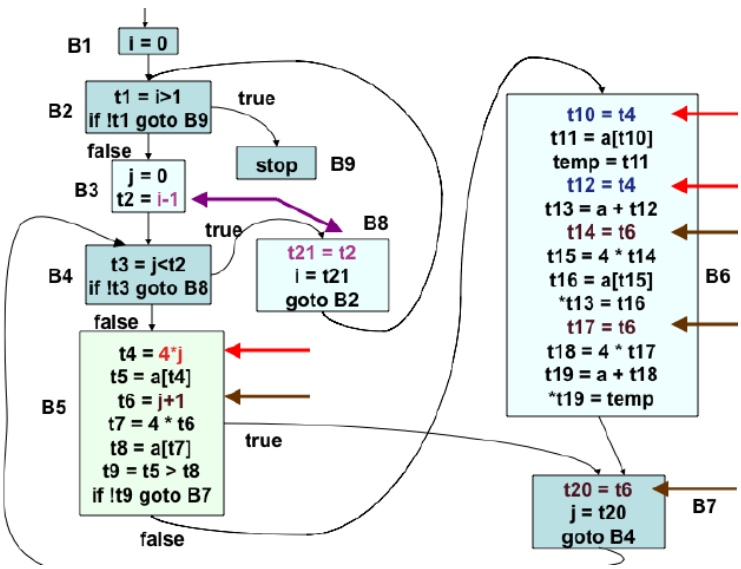
OUT[B4] = egen[B4] ⋃ (IN[B4] – ekill[B4])

$$
\begin{aligned}
IN[B] &= \bigcap_{P \text{ is a predecessor of } B} OUT[P], \ B \text{ not initial} \\
OUT[B] &= e\_gen[B] \bigcup (IN[B] - e\_kill[B])
\end{aligned}
$$

DFA: Available expression (Example)



DFA: Available expression (Example)



Computing Available Expression

for each block $B \neq B1$ do $\{OUT[B] = U - e\_kill[B];\ \}$
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
$change = true$;
while $change$ do $\{\ change = false$;
  for each block $B \neq B1$ do $\{$

$$IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P];$$

$$oldout = OUT[B];$$

$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B]);$$

    if $(OUT[B] \neq oldout)\ change = true$;
  $\}$
$\}$

Live Variables

DFA: Live Variables

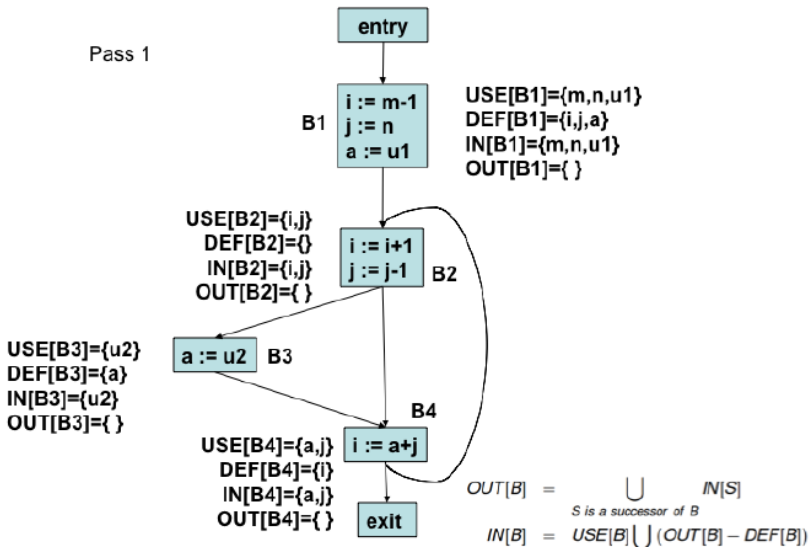
- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p
- Sets of variables constitute the domain of data-flow values
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of variables live at the beginning of B
- $OUT[B]$ is the set of variables live just after B
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

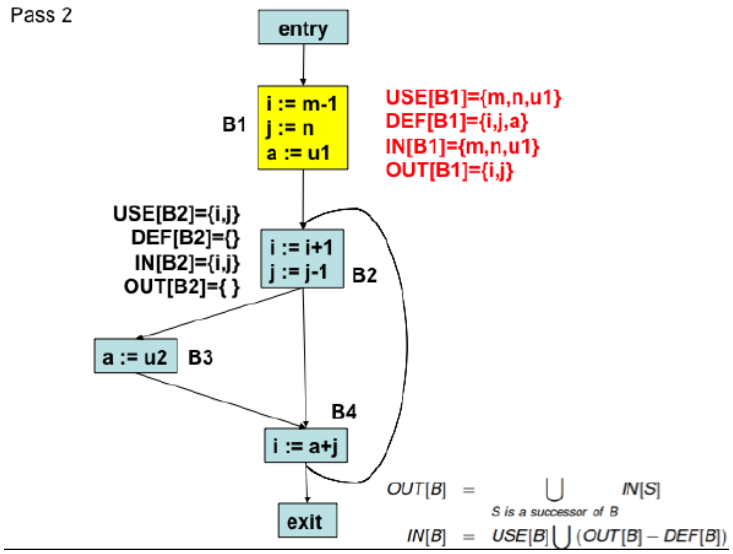
$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

$$IN[B] = \phi, \text{for all } B \text{ (initialization only)}$$

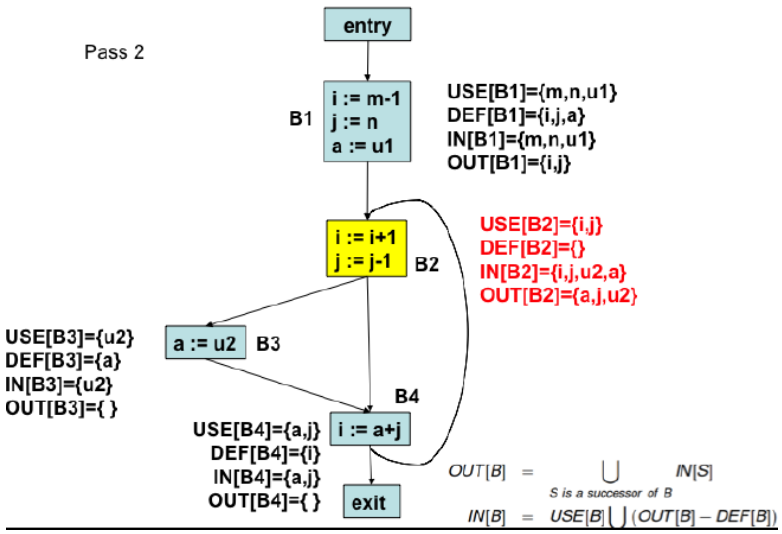
Live Variable Example



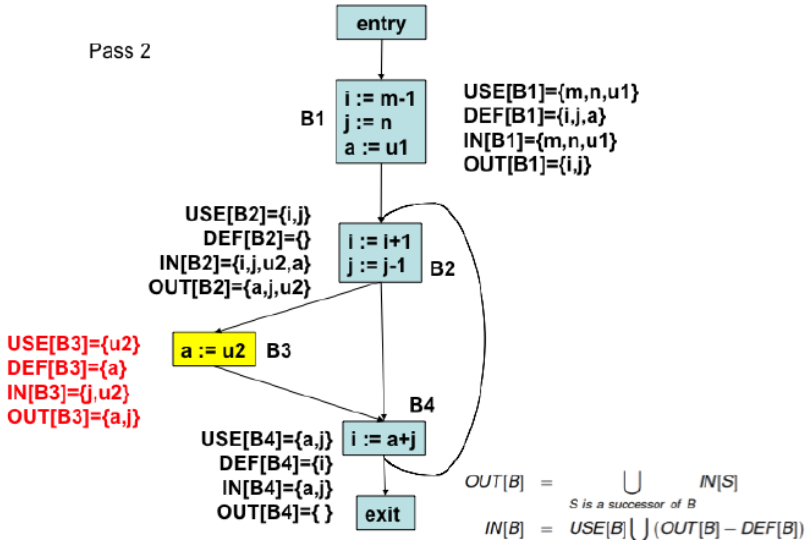
Live Variable Example



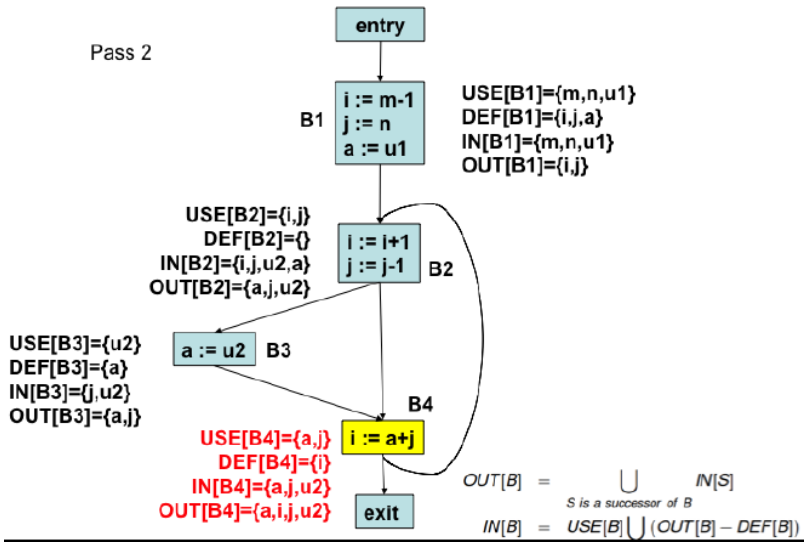
Live Variable Example



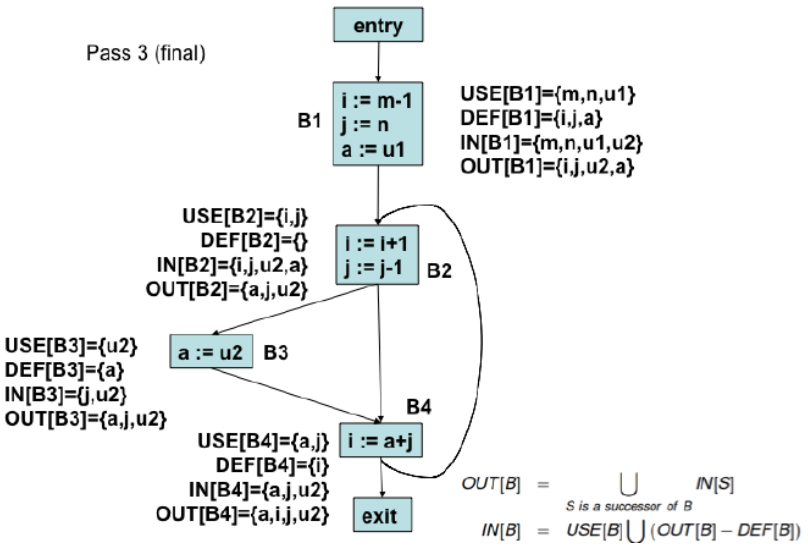
Live Variable Example



Live Variable Example



Live Variable Example



Copy Propagation

DFA: 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

DFA: Copy Propagation (DF 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

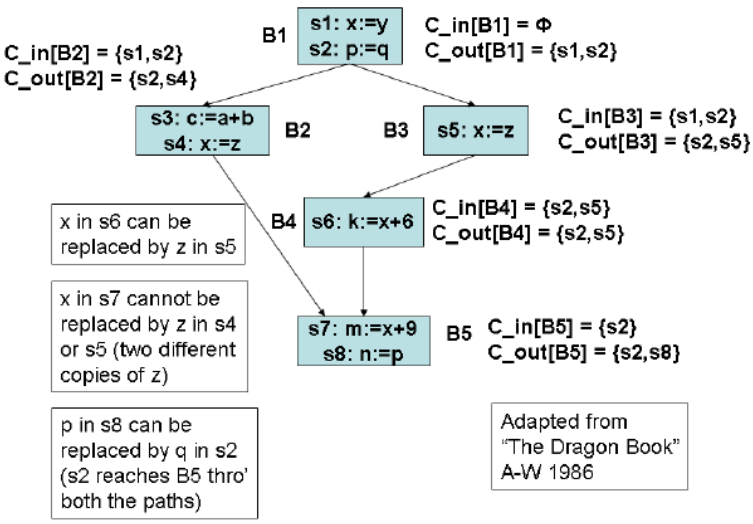
$$
\begin{aligned}
c\_in[B] &= \bigcap_{P \text{ is a predecessor of } B} c\_out[P], \ B \text{ not initial} \\
c\_out[B] &= c\_gen[B] \bigcup (c\_in[B] - c\_kill[B]) \\
c\_in[B1] &= \phi, \text{ where } B1 \text{ is the initial block} \\
c\_out[B] &= U - c\_kill[B], \text{ for all } B \neq B1 \text{ (initialization only)}
\end{aligned}
$$

Copy Propagation Computation

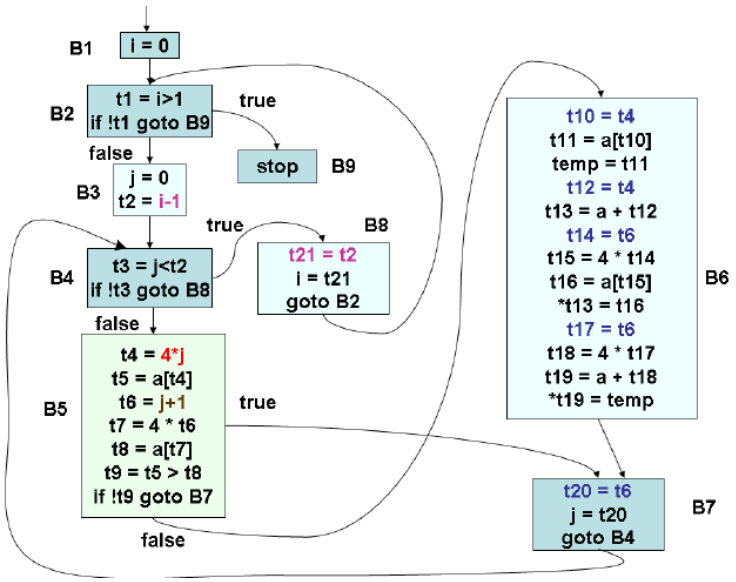
For each copy, $s : x := y$, do the following

1. Using the $du - chain$, determine those uses of x that are reached by s
2. For each use u of x found in (1) above, check that
   - (i) u-d chain of u consists of s only
   - (ii) s is in $c\_in[B]$, where B is the block to which u 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
   - (iii) no definitions x or y occur within B prior to u 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

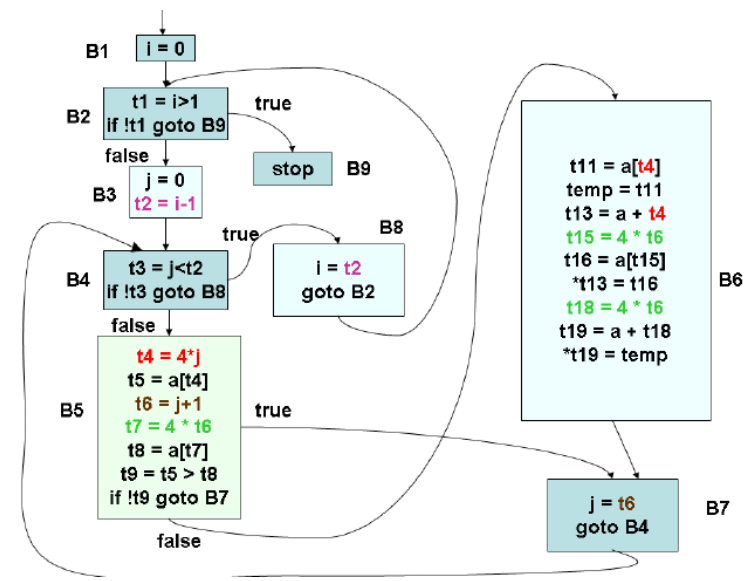
Copy Propagation Example



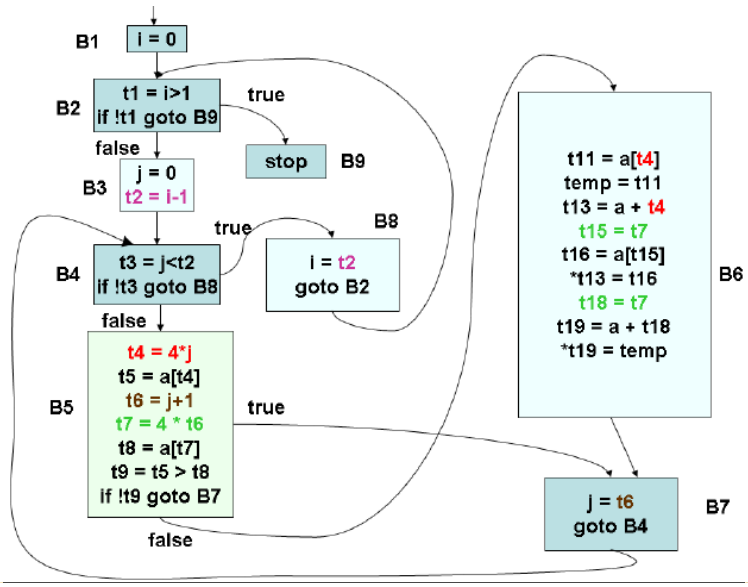
Copy Propagation Example



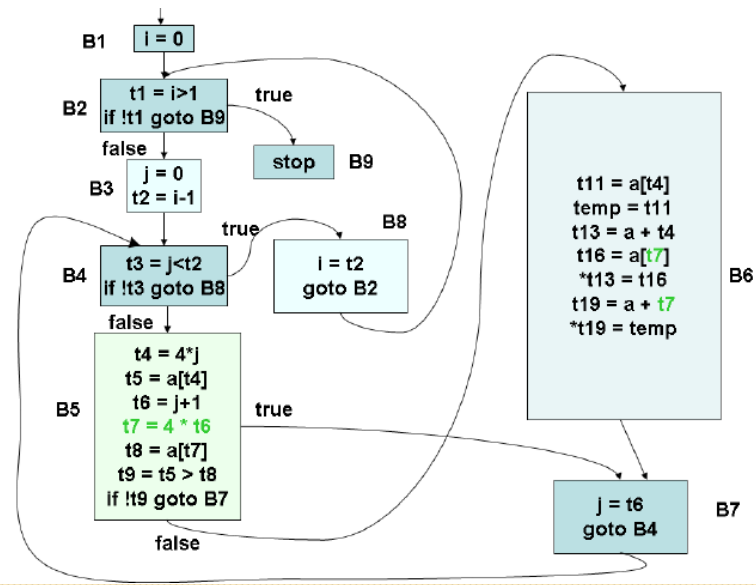
Copy Propagation Example



Copy Propagation Example



Copy Propagation Example



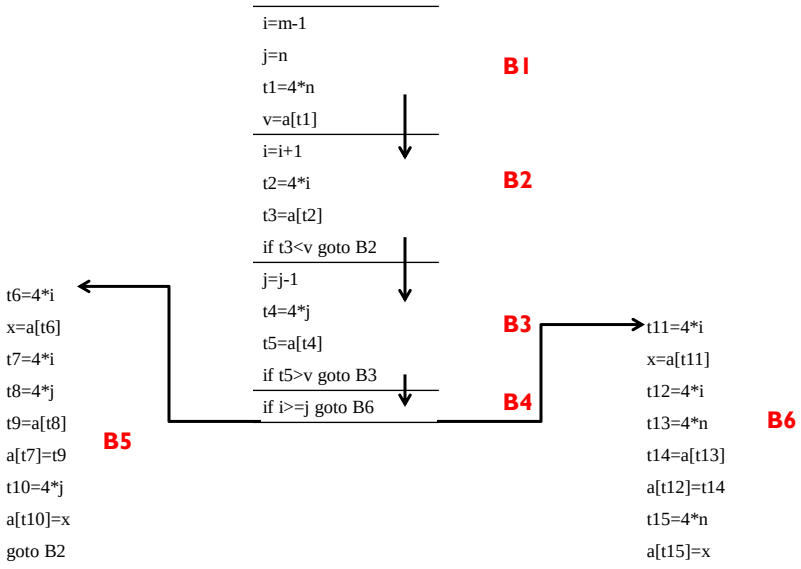
A Sorting Example

```
void sort(int m, int n)
{
        int i,j;
        int v,x;
        if(n<=m) return;
        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;
        sort(m,j); sort(i+1,n);
}
```

A Sorting Example

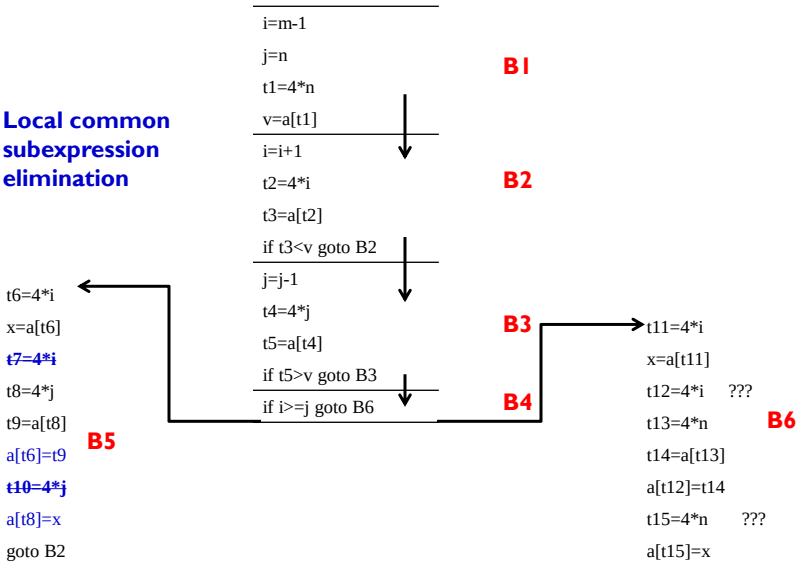
1	i=m-1	14	t6=4*i
2	j=n	15	x=a[t6]
3	t1=4*n	16	t7=4*i
4	v=a[t1]	17	t8=4*j
5	i=i+1	18	t9=a[t8]
6	t2=4*i	19	a[t7]=t9
7	t3=a[t2]	20	t10=4*j
8	if t3<v goto 5	21	a[t10]=x
9	j=j-1	22	goto 5
10	t4=4*j	23	t11=4*i
11	t5=a[t4]	24	x=a[t11]
12	if t5>v goto 9	25	t12=4*i
13	if i>=j goto 23	26	t13=4*n
		27	t14=a[t13]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

A Sorting Example



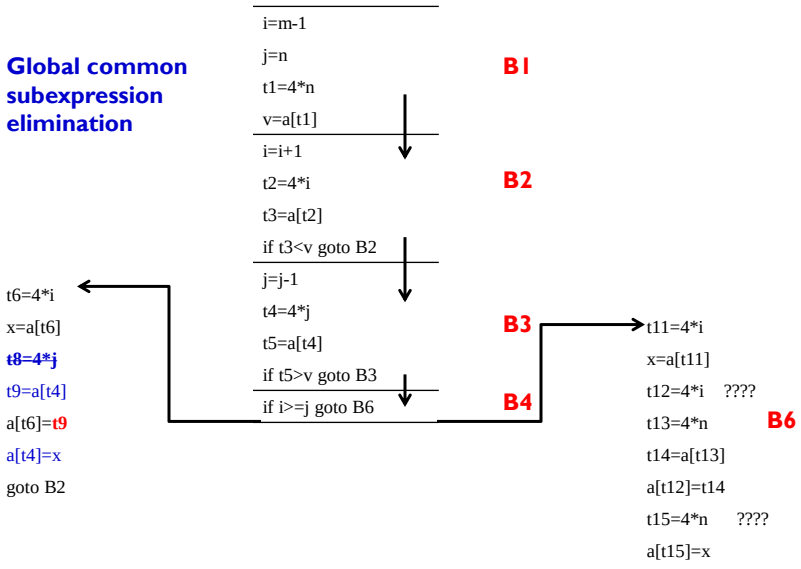
A Sorting Example

Local common subexpression elimination



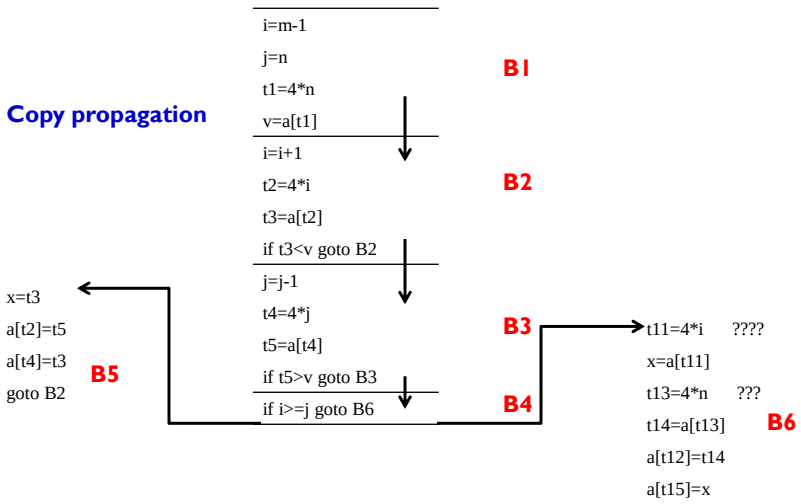
A Sorting Example

Global common subexpression elimination

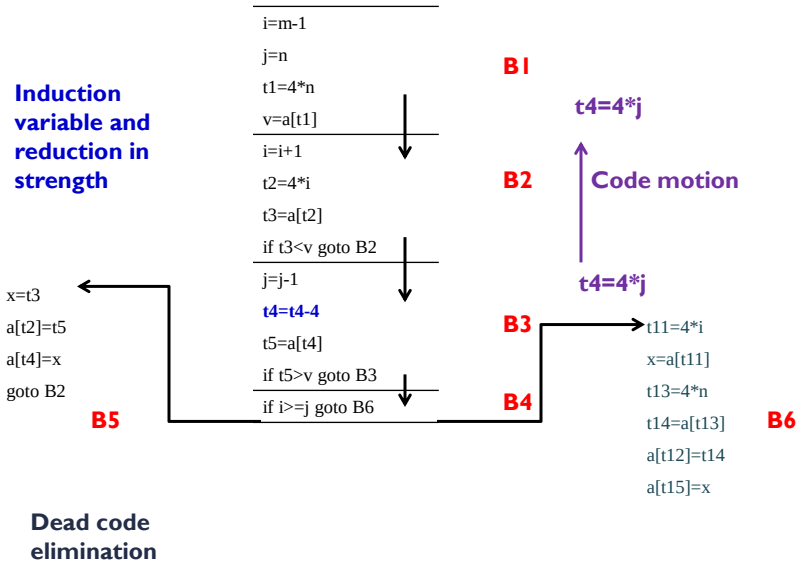


A Sorting Example

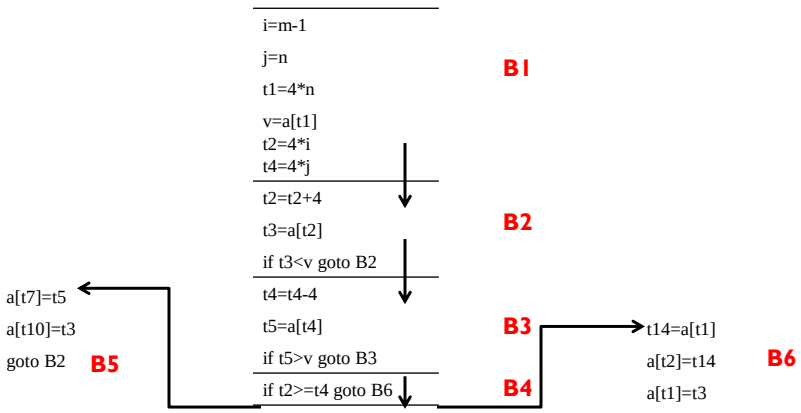
Copy propagation



A Sorting Example



A Sorting Example



Work Out

```
1   dp=0
2   i=0
3   t1=i*8
4   t2=a[t1]
5   t3=i*8
6   t4=b[t3]
7   t5=t2*t4
8   dp=dp+t5
9   i=i+1
10  if i<n goto 3
```