

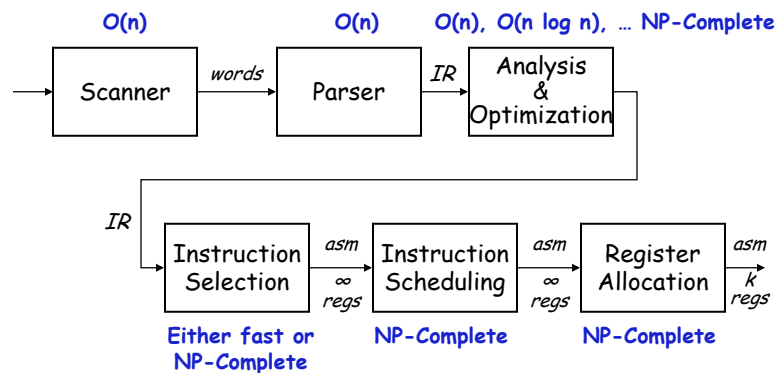
Compiler Optimizations

optimization

- ❑ tries to improve quality of code (may fail in some cases)
- ❑ optimizer typically consists of multiple passes
- ❑ different optimization (code improvement) objectives:
 - execution time reduction
 - reduction in resource requirements (memory, registers)
 - (peak) power and energy reduction
- ❑ criteria for effectiveness of optimizations
 - **safety** - program semantics **must** be preserved
 - opportunity - how often can it be applied?
 - profitability - how much improvement?

cs671, spring'08

Optimizing Compilers



cs671, spring'08

What I would like to talk about

- What is optimization? - an exploration through examples (machine independent optimization)
- Construction of the control flow graph
- Expression DAGs in basic blocks, local optimizations
- Loops, dominators
- Data flow analysis (data flow problems)
 - Reaching definitions (RD)
 - Live variables (LV)
 - Available expressions (AVAIL)
 - Very busy expressions (VBE)
- Iterative algorithm
- DU/UD chains

cs671, spring'08

Example Code

Fortran Source Code:

```
.  
.   
.   
sum = 0  
do 10 i = 1, n  
10 sum = sum + a(i) * a(i)  
.   
.   
.
```

cs671, spring'08

3-address code

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

cs671, spring'08

3-address code

1. sum = 0	sum = 0
2. i = 1	init for loop and check limit
3. if i > n goto 15	
4. t1 = addr(a) - 4	a[i]
5. t2 = i * 4	
6. t3 = t1[t2]	
7. t4 = addr(a) - 4	a[i]
8. t5 = i * 4	
9. t6 = t4[t5]	
10. t7 = t3 * t6	a[i] * a[i]
11. t8 = sum + t7	increment sum
12. sum = t8	
13. i = i + 1	
14. goto 3	Incr. loop counter back to loop check
15.	

cs671, spring'08

RISC
or
CISC ?

3-address code

1.	sum = 0	sum = 0
2.	i = 1	init for loop and check limit
3.	if i > n goto 15	
4.	t1 = addr(a) - 4	a[i]
5.	t2 = i * 4	
6.	t3 = t1[t2]	
7.	t4 = addr(a) - 4	a[i]
8.	t5 = i * 4	
9.	t6 = t4[t5]	
10.	t7 = t3 * t6	a[i] * a[i]
11.	t8 = sum + t7	increment sum
12.	sum = t8	
13.	i = i + 1	
14.	goto 3	Incr. loop counter back to loop check
15.		

cs671, spring'08

RISC
or
CISC ?

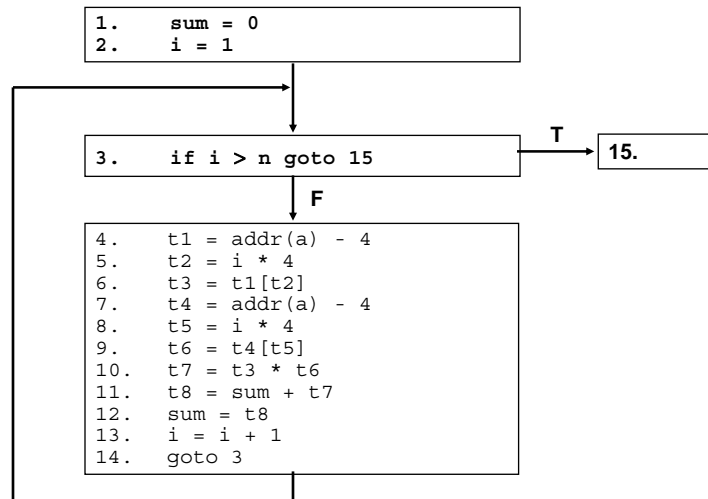
3-address code

1.	sum = 0	sum = 0
2.	i = 1	init for loop and check limit
3.	if i > n goto 15	
4.	t1 = addr(a) - 4	a[i]
5.	t2 = i * 4	
6.	t3 = t1[t2]	
7.	t4 = addr(a) - 4	a[i]
8.	t5 = i * 4	
9.	t6 = t4[t5]	
10.	t7 = t3 * t6	a[i] * a[i]
11.	t8 = sum + t7	increment sum
12.	sum = t8	
13.	i = i + 1	
14.	goto 3	Incr. loop counter back to loop check
15.		

Machine-independent
optimization

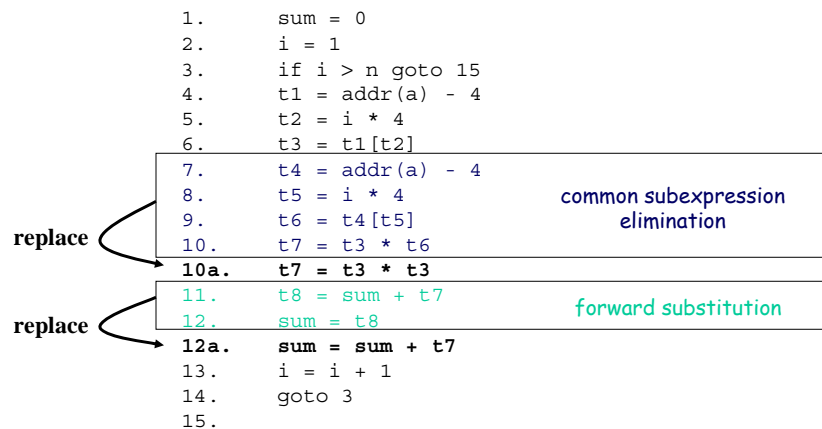
cs671, spring'08

Control Flow Graph (CFG)



cs671, spring'08

Local Common Subexpression Elimination & Forward Substitution



cs671, spring'08

Invariant Code Motion

```
1.    sum = 0
2.    i = 1
2a.   t1 = addr(a) - 4
3.    if i > n goto 15
4.    t1 = addr(a) - 4      invariant code motion
5.    t2 = i * 4
6.    t3 = t1[t2]
10a.  t7 = t3 * t3
12a.  sum = sum + t7
13.   i = i + 1
14.   goto 3
15.
```

cs671, spring'08

Strength Reduction

```
1.    sum = 0
2.    i = 1
2a.   t1 = addr(a) - 4
2b.   t2 = i * 4
3.    if i > n goto 15
5.    t2 = i * 4      strength reduction
6.    t3 = t1[t2]
10a.  t7 = t3 * t3
12a.  sum = sum + t7
12b.  t2 = t2 + 4
13.   i = i + 1
14.   goto 3
15.
```

What other strength reduction could we have done here?

cs671, spring'08

Strength Reduction

```
1.    sum = 0
2.    i = 1
2a.   t1 = addr(a) - 4
2b.  t2 = i * 4
3.    if i > n goto 15
5.    t2 = i * 4 strength reduction
6.    t3 = t1[t2]
10a.  t7 = t3 * t3
12a.  sum = sum + t7
12b. t2 = t2 + 4
13.   i = i + 1
14.   goto 3
15.
```

What other strength reduction could we have done here?
machine dependent shift operation

cs671, spring'08

Induction Variable Elimination and Loop Test Adjustment

```
1.    sum = 0
2.    i = 1
2a.   t1 = addr(a) - 4
2b.   t2 = i * 4
2c.  t9 = n * 4
3.    if i > n goto 15 loop test adjustment
3a.  if t2 > t9 goto 15
6.    t3 = t1[t2]
10a.  t7 = t3 * t3
12a.  sum = sum + t7
12b.  t2 = t2 + 4
13.   i = i + 1 induction variable elimination
14.   goto 3a
15.
```

cs671, spring'08

Constant Propagation

```
1.    sum = 0
2.    i = 1
2a.   t1 = addr(a) - 4
2b.   t2 = i * 4
2d.   t2 = 4
2c.   t9 = n * 4
3a.   if t2 > t9 goto 15
6.    t3 = t1[t2]
10a.  t7 = t3 * t3
12a.  sum = sum + t7
12b.  t2 = t2 + 4
14.   goto 3a
15.
```

constant propagation

cs671, spring'08

Dead Code Elimination

```
1.    sum = 0
2.    i = 1
2a.   t1 = addr(a) - 4
2d.   t2 = 4
2c.   t9 = n * 4
3a.   if t2 > t9 goto 15
6.    t3 = t1[t2]
10a.  t7 = t3 * t3
12a.  sum = sum + t7
12b.  t2 = t2 + 4
14.   goto 3a
15.
```

dead code elimination

cs671, spring'08

Final Optimized Code

```
1.    sum = 0
2.    t1 = addr(a) - 4
3.    t2 = 4
4.    t4 = n * 4
5.    if t2 > t4 goto 11
6.    t3 = t1[t2]
7.    t5 = t3 * t3
8.    sum = sum + t5
9.    t2 = t2 + 4
10.   goto 5
11.
```

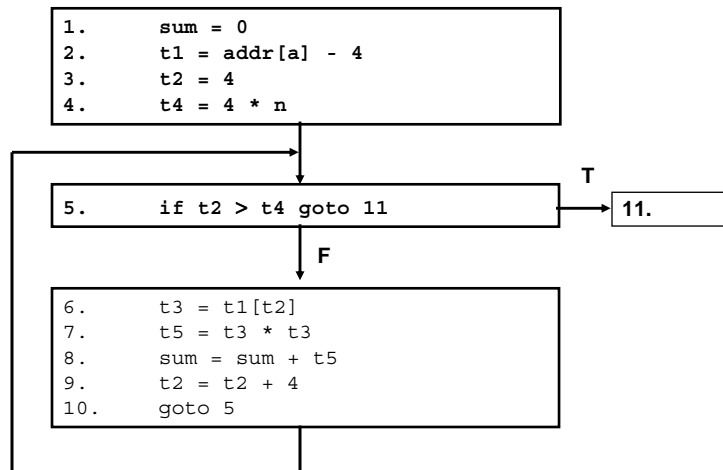
unoptimized: 8 temps, 11 stmts in innermost loop
optimized: 5 temps, 5 stmts in innermost loop

1 index addressing
1 multiplication
2 additions
1 jump
1 test

2 index addressing
3 multiplications
2 additions & 2 subtractions
1 jump
1 test
1 copy

cs671, spring'08

CFG of Final Optimized Code



cs671, spring'08

Basic Block Construction

- Find **leader** statements
 - First program statement
 - Targets of conditional or unconditional gotos
 - Any statement following a conditional goto
- ($\forall x \in \text{leaders}$) construct B_x , basic block headed by x : include all statements following x until next **leader** or end of program is reached.
- At end of algorithm, any statements not in some basic block are unreachable from program entry and are therefore, dead code.

cs671, spring'08

3-address code example

leader

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

cs671, spring'08

3-address code example

leader

```
1.  sum = 0
2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15.
```

cs671, spring'08

3-address code example

leader

basic block

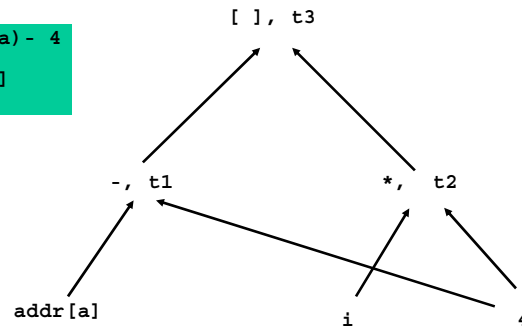
```
B1 1.  sum = 0
    2.  i = 1
B3 3.  if i > n goto 15
B4 4.  t1 = addr(a) - 4
    5.  t2 = i * 4
    6.  t3 = t1[t2]
    7.  t4 = addr(a) - 4
    8.  t5 = i * 4
    9.  t6 = t4[t5]
    10. t7 = t3 * t6
    11. t8 = sum + t7
    12. sum = t8
    13. i = i + 1
    14. goto 3
B15 15.
```

cs671, spring'08

Basic Block DAG Construction

B4

```
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
...
```



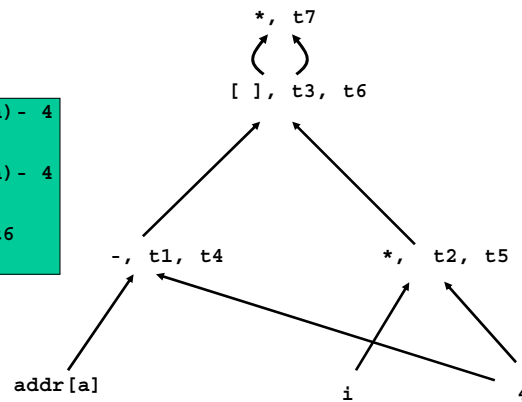
Reference: ASU, p.548

cs671, spring'08

Basic Block DAG Construction

B4

```
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
...
```



Reference: ASU, p.548

cs671, spring'08

Basic Block DAG Construction

B4

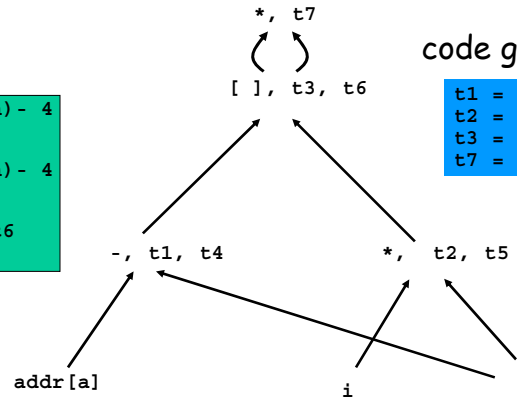
```

4.  t1 = addr(a) - 4
5.  t2 = i * 4
6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
8.  t5 = i * 4
9.  t6 = t4[t5]
10. t7 = t3 * t6
    . . .
    
```

code generated:

```

t1 = addr[a] - 4
t2 = i * 4
t3 = t1[t2]
t7 = t3 * t3
    
```



Reference: ASU, p.548

cs671, spring'08

DAG Construction Algorithm

How to add a subexpression into a partially constructed DAG:

$A = B + C$

Is there a node already for $B + C$?

- If so, add A to its list of labels.

- If not:

• is there a node labelled B already?

 If not, create a leaf labeled B .

• Is there a node labeled C already?

 If not, create a leaf labeled C .

• Create a node labeled A , for $+$, with left child B and right child C .

How to do this? **HASHING** $\langle \text{op}, \text{node}(\text{opd1}), \text{node}(\text{opd2}) \rangle$

Reference: ASU, p.548

cs671, spring'08

DAG Construction Algorithm

How to add a subexpression into a partially constructed DAG:

$A = B + C$

Is there a node already for $B + C$? $\langle +, \text{node}(B), \text{node}(C) \rangle$ defined?

- If so, add A to its list of labels.

- If not:

• is there a node labeled B already? $\text{node}(B)$ defined?

If not, create a leaf labeled B .

• Is there a node labeled C already? $\text{node}(C)$ defined?

If not, create a leaf labeled C .

• Create a node labeled A , for $+$, with left child B and right child C . Create $\text{node}(+)$ with children $\text{node}(B)$, $\text{node}(C)$

How to do this? HASHING $\langle \text{op}, \text{node}(\text{op1}), \text{node}(\text{op2}) \rangle$

Reference: ASU, p.548

cs671, spring'08

Dominators and Loops

Dominator:

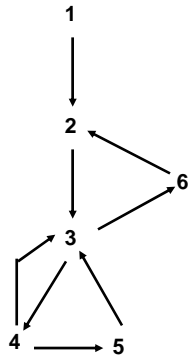
A node X dominates a node Y if and only if all paths from the control flow graph (CFG) entry node to Y pass through X . Note: every node dominates itself.

Loops:

Let (Y, X) be a CFG edge such that X dominates Y . Then all nodes on paths from X to Y are in the loop defined by (Y, X) .

cs671, spring'08

Dominators and Loops



CFG

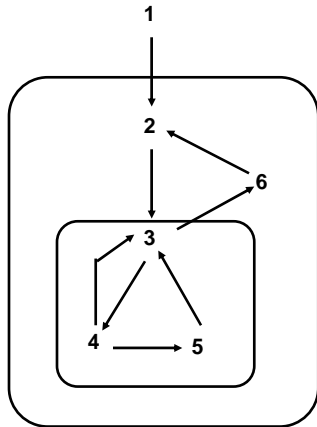
What is the dominance relation?

$\text{dom}(1) = \{ \dots \}$
 $\text{dom}(2) = \{ \dots \}$
 \dots

Where are the loops?

cs671, spring'08

Dominators and Loops



CFG

What is the dominance relation?

$\text{dom}(1) = \{ \dots \}$
 $\text{dom}(2) = \{ \dots \}$
 \dots

Loop (5,3) = {3, 4, 5}

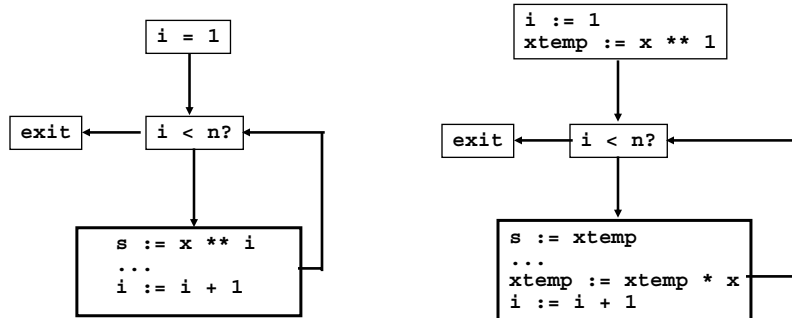
Loop (4,3) = {3, 4}

Coalesced because same loop entry node (3).

Loop (6,2) = {2, 3, 4, 5, 6}

cs671, spring'08

Strength Reduction



cs671, spring'08

General Code Motion

```
n := 1; k := 0; m := 3; read x;
while n < 10 do
  if 2 + x ≥ 5 then k := 5;
  if 3 + k = 3 then m := m + 2;
  n := n + k + m;
endwhile;
```

cs671, spring'08

General Code Motion

```
1. n := 1; 2. k := 0; 3. m := 3;
```

```
4. read x;
```

```
5. while n < 10 do
```

```
6.   if  $2 * x \geq 5$  then 7.  $k := 5$ ;
```

```
8.   if  $3 + k = 3$  then 9.  $m := m + 2$ ;
```

```
10.   $n := n + k + m$ ;
```

```
11. endwhile;
```



Invariant within loop and therefore moveable.



Unaffected by definitions in loop and guarded by invariant condition



Moveable after we move statements 6 and 7.



Not moveable because may use def of m from statement 9 on previous iteration.

cs671, spring'08

General Code Motion

```
n := 1; k := 0; m := 3; read x;
while n < 10 do
  if  $2 * x \geq 5$  then k := 5;
  if  $3 + k = 3$  then m := m + 2;
  n := n + k + m;
endwhile;
```



```
n := 1; k := 0; m := 3; read x;
if  $2 * x \geq 5$  then k := 5;
t1 := (3 + k = 3);
while n < 10 do
  if t1 then m := m + 2;
  n := n + k + m;
endwhile;
```

cs671, spring'08

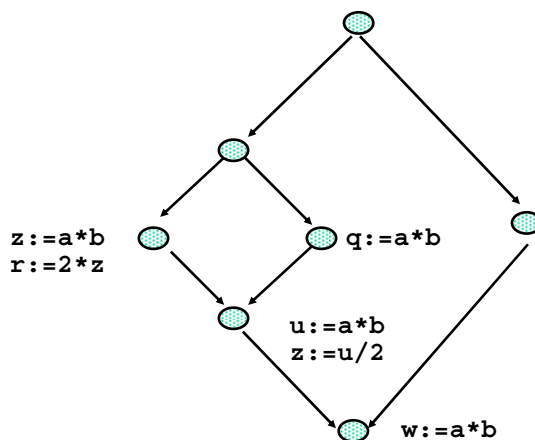
Additional Optimization: Code Specialization

```
n := 1; k := 0; m := 3; read x;
if 2 * x ≥ 5 then k := 5;
t1 := (3 + k = 3);
if t1 then
  while n < 10 do
    m := m + 2;
    n := n + k + m;
  endwhile;
else
  while n < 10 do
    n := n + k + m;
  endwhile;
endif
```

Specialization of while loop
depending on value of t1

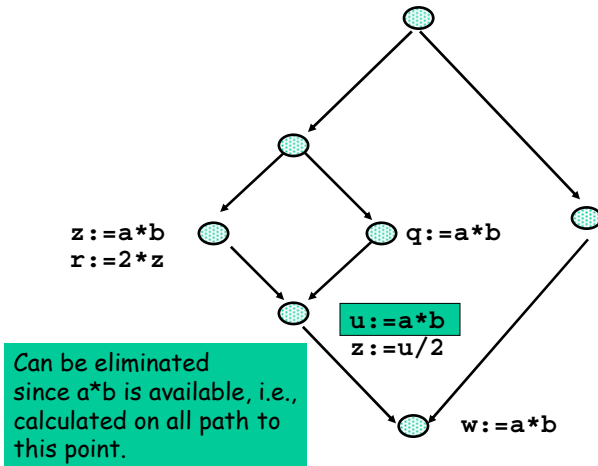
cs671, spring'08

Global Common Subexpression Elimination



cs671, spring'08

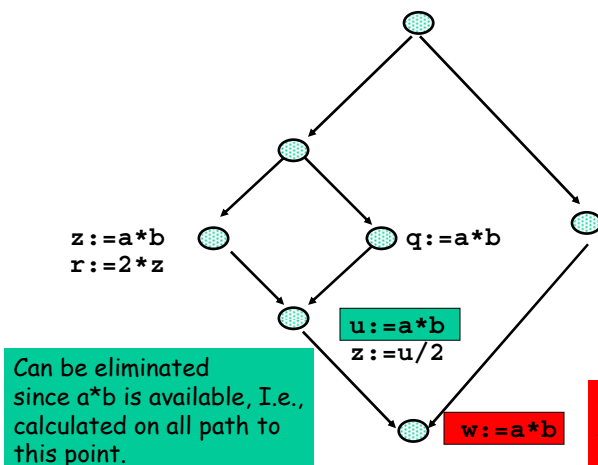
Global Common Subexpression Elimination



Can be eliminated since $a * b$ is available, i.e., calculated on all path to this point.

cs671, spring'08

Global Common Subexpression Elimination

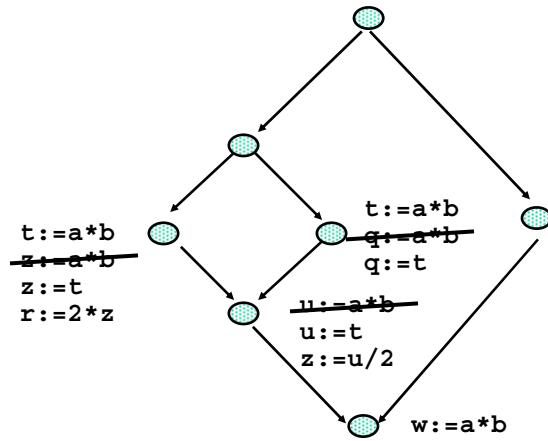


Can be eliminated since $a * b$ is available, i.e., calculated on all path to this point.

Can not be eliminated since $a * b$ is not available on all path reaching this point.

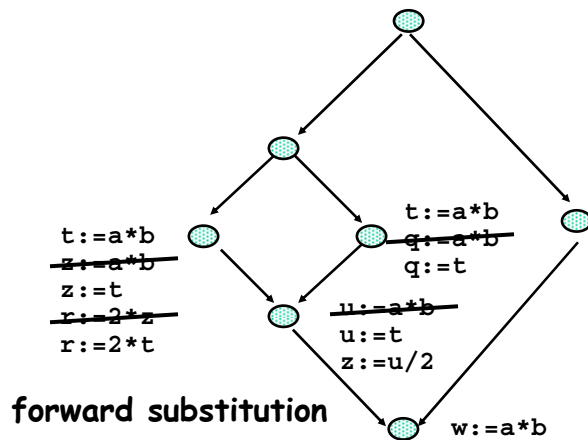
cs671, spring'08

Global Common Subexpression Elimination



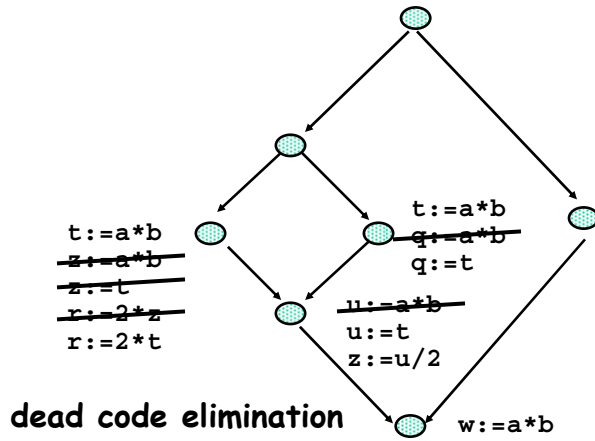
cs671, spring'08

Global Common Subexpression Elimination



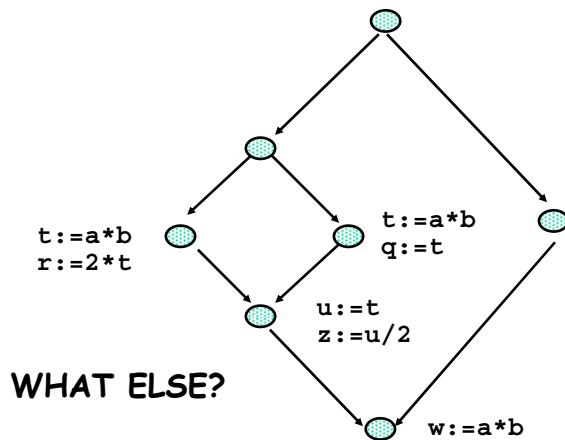
cs671, spring'08

Global Common Subexpression Elimination



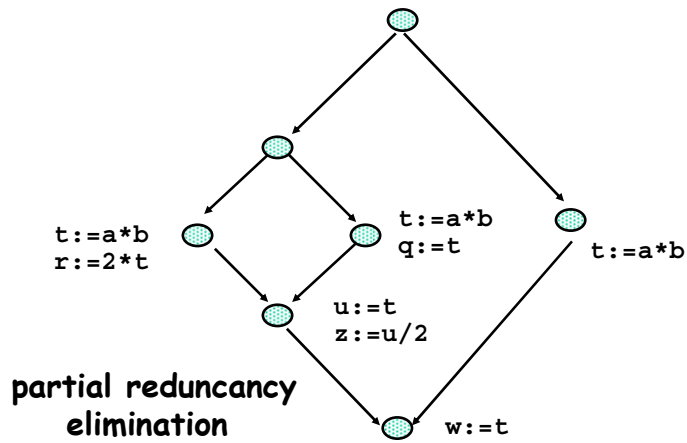
cs671, spring'08

Global Common Subexpression Elimination



cs671, spring'08

Global Common Subexpression Elimination



cs671, spring'08

Summary

Representations:

- 3 address code
- Basic blocks
- Control flow graph
- DAG constructors for basic blocks
- Loops
- Dominators

Optimizations:

- Code motion
- Strength reduction
- Test elimination
- Constant propagation
- Common subexpression elimination
- Forward substitution (Copy propagation)
- Dead code elimination
- Partial redundancy elimination
- Specialization

cs671, spring'08

Definitions

Flow analysis:

Fact finding about a program before its execution;

Control flow analysis:

Discerning possible execution paths.

Data flow analysis:

Determining information about modification, preservation, and use of data entities in a program;

compile-time reasoning about the **run-time** behavior of a program.

cs671, spring'08

(Global) Data Flow Problem

- Uses CFG as basic program representation
- Represents specific **facts** about run-time behavior that we interested in (universe of facts).
- Describes effect of "executing" each basic block on set of facts.
- Describes information flow between basic blocks.
- Has limitations:
 - **precision**: up to symbolic execution
 - no knowledge about control flow decisions (assume all paths are taken)
 - tradeoff between precision and cost of computing a solution
 - **solution**: cannot afford MOP solution
 - interesting class where MOP = MFP
 - not all problems fit that category
 - **aliasing**: disambiguate data accesses

cs671, spring'08

(Global) Data Flow Problem

- Represents specific **facts** about run-time behavior:
Set of facts form a lattice L .
- Describes effect of "executing" each basic block on set of facts:
Propagation functions $f_{\text{Block}}: L \rightarrow L$
- Describes information flow between basic blocks:
 - Formulated as set of simultaneous equations
 - Solve equations using
 - iterative framework over CFG
 - use methods that are based on program structure

cs671, spring'08

Classical Data Flow Problems

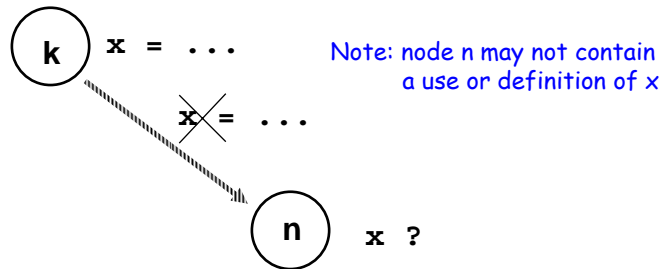
- **Reaching definitions (RD)**, **Live uses of variables (LV)**, **Available expressions (AVAIL)**, **Very Busy Expressions (VBE)**
- Def-Use and Use-Def chains, built from RD
- AVAIL enables global common subexpression elimination
- VBE can be used for code motion

cs671, spring'08

Reaching Definitions (RD)

A definition of a variable x is a statement that may modify the value of variable x .

A definition of a variable x at node k reaches node n if there is a definition-free path from k to n .



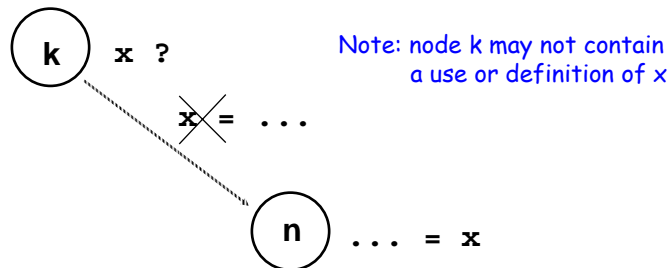
cs671, spring'08

Live Variables (LV)

Use:

An appearance of a variable x as a RHS operand that results in reading its value.

A use of a variable x is live on exit from node k if there is a definition-clear path for x from k to a node n that uses x .



cs671, spring'08

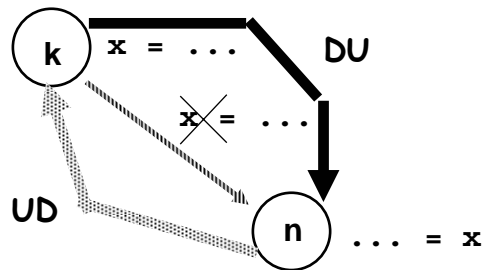
DU and UD Chains

UD-Chain:

Links each use of variable x to definitions which reach that use.

DU-Chain:

Links each definition of variable x to those uses which that definition can reach.



cs671, spring'08

Optimizations that can use DU and UD Chains

- Dead code elimination (DU)
- Code motion (UD)
- Strength reduction (UD)
- Constant propagation (UD/DU)
- Forward substitution (Copy propagation) (DU)

cs671, spring'08