

# CSC D70: Compiler Optimization

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# CSC D70: Compiler Optimization Introduction, Logistics

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Summary

- Syllabus
  - Course Introduction, Logistics, Grading
- Information Sheet
  - Getting to know each other
- Assignments
- Learning LLVM
- Compiler Basics

# **Syllabus: Who Are We?**

# Gennady (Gena) Pekhimenko

**Assistant Professor, Instructor**

[pekhimenko@cs.toronto.edu](mailto:pekhimenko@cs.toronto.edu)

<http://www.cs.toronto.edu/~pekhimenko/>

Office: BA 5232 / IC 454

PhD from Carnegie Mellon

Worked at Microsoft Research, NVIDIA, IBM

Research interests: computer architecture, systems, machine learning, compilers, hardware acceleration



**Computer Systems and Networking Group (CSNG)**

**EcoSystem Group**

# Bojian Zheng

MSe. PhD Student, TA

[bojian@cs.toronto.edu](mailto:bojian@cs.toronto.edu)

Office: BA 5214 D02

BSc. from UofT ECE

Research interests: computer architecture, GPUs, machine learning



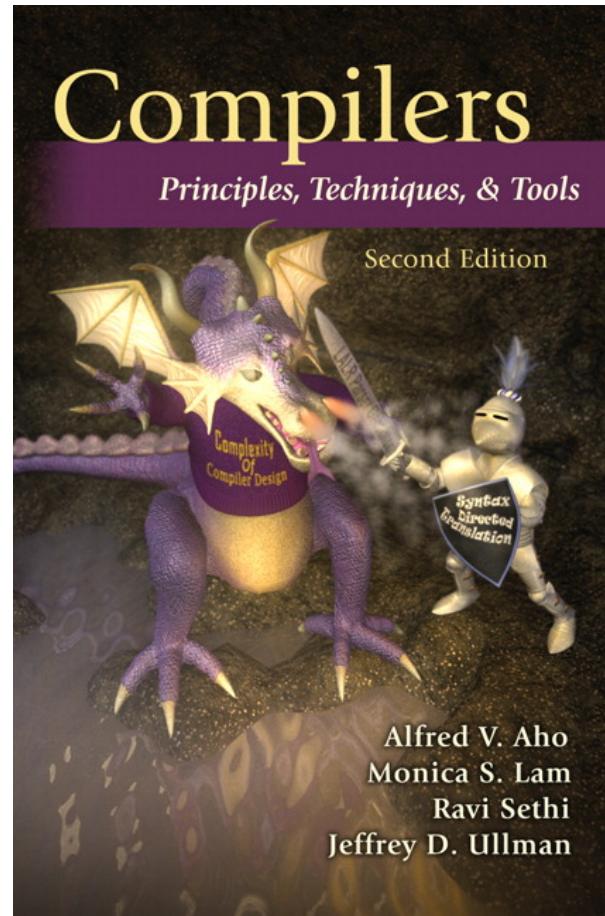
Computer Systems and Networking Group (CSNG)

EcoSystem Group

# Course Information: Where to Get?

- Course Website:  
<http://www.cs.toronto.edu/~pekhimenko/courses/cscd70-w19/>
  - Announcements, Syllabus, Course Info, Lecture Notes, Tutorial Notes, Assignments
- Piazza:  
<https://piazza.com/utoronto.ca/winter2019/cscd70/home>
  - Questions/Discussions, Syllabus, Announcements
- Quercus
  - Emails/announcements
- Your email

# Useful Textbook



# CSC D70: Compiler Optimization Compiler Introduction

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# **Why Computing Matters (So Much)?**

# **WHAT IS THE DIFFERENCE BETWEEN THE COMPUTING INDUSTRY AND THE PAPER TOWEL INDUSTRY?**



# Industry of replacement



1971

2017

# **CAN WE CONTINUE BEING AN INDUSTRY OF NEW POSSIBILITIES?**

Personalized  
healthcare

Virtual  
reality

Real-time  
translators

# Moore's Law

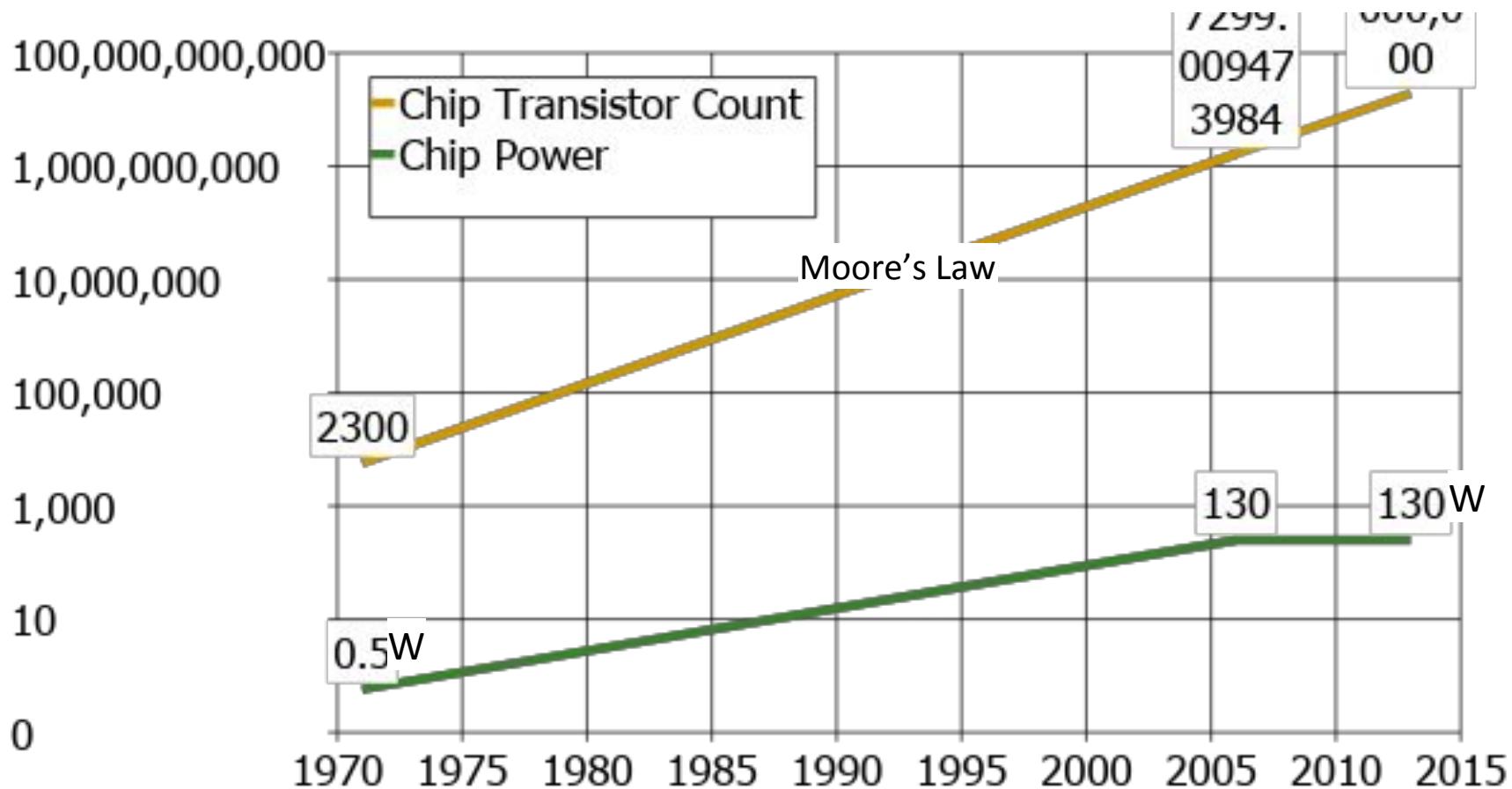
Or, how we became an industry of new possibilities

Every 2 Years

- Double the number of transistors
- Build higher performance general-purpose processors
  - Make the transistors available to masses
  - Increase performance ( $1.8\times\uparrow$ )
  - Lower the cost of computing ( $1.8\times\downarrow$ )

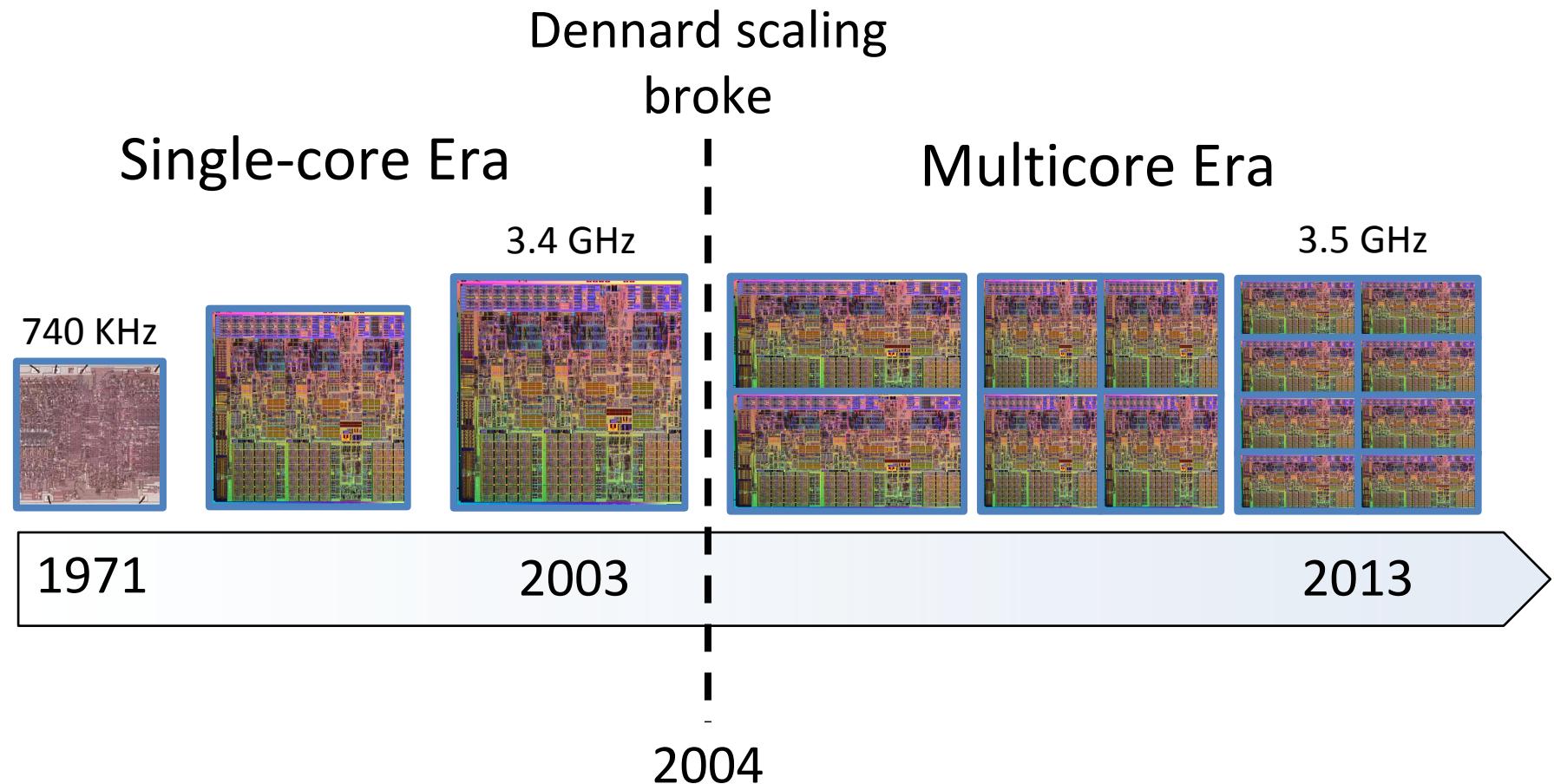
# What is the catch?

Powering the transistors without melting the chip



# Looking back

## Evolution of processors



# Any Solution Moving Forward?

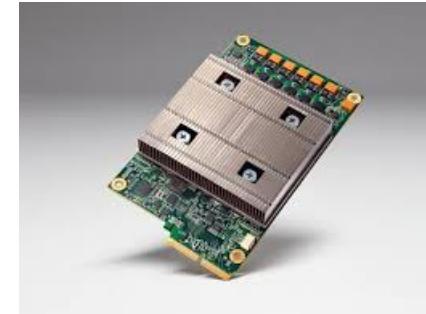
Hardware accelerators:



**GPUs**  
**(Graphics Processing Units)**

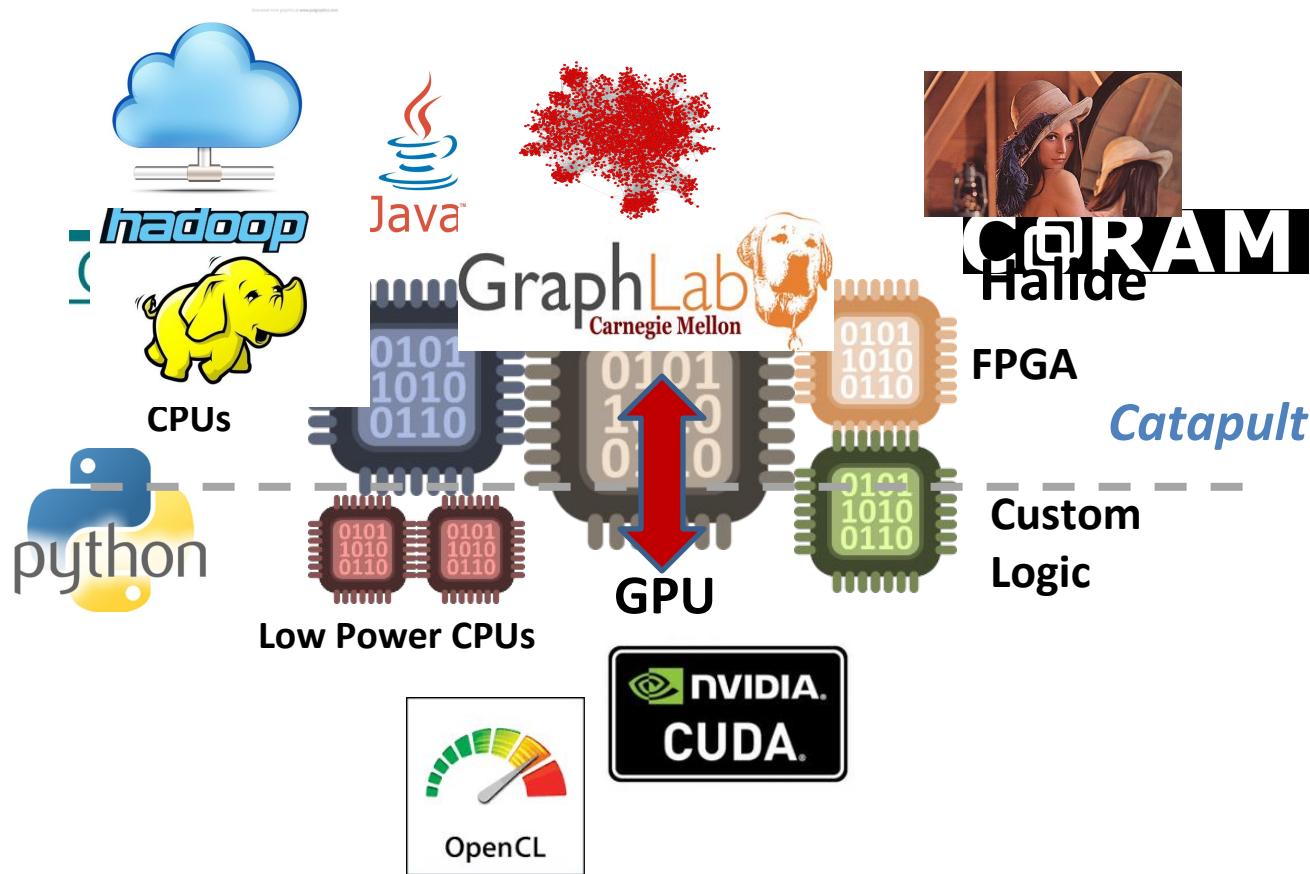


**FPGAs**  
**(Field Programmable Gate Arrays)**

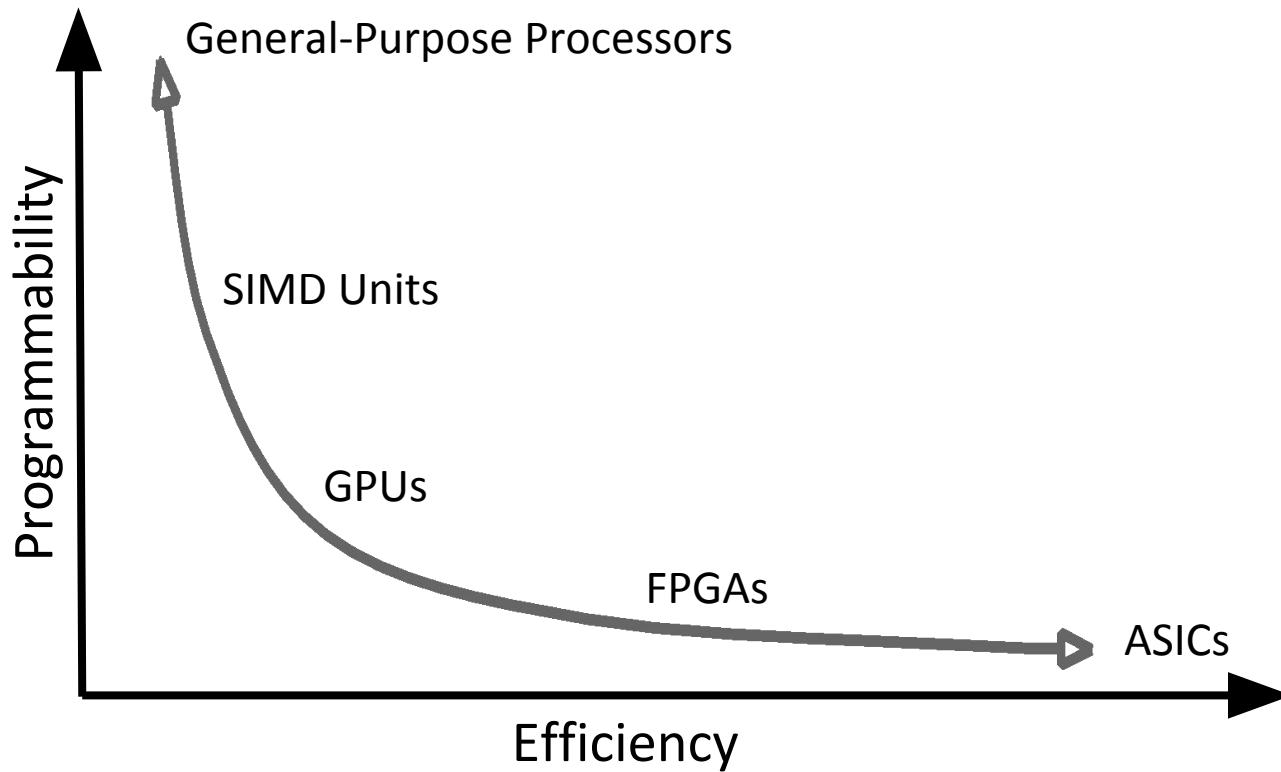


**TPUs**  
**(Tensor Processing Units)**

# Heterogeneity and Specialization



# Programmability versus Efficiency



We need compilers!

# Introduction to Compilers

- What would you get out of this course?
- Structure of a Compiler
- Optimization Example

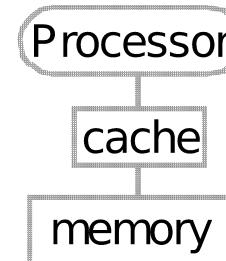
# What Do Compilers Do?

1. Translate one language into another
  - e.g., convert C++ into x86 object code
  - difficult for “natural” languages, but feasible for computer languages
2. Improve (i.e. “optimize”) the code
  - e.g., make the code run 3 times faster
    - or more energy efficient, more robust, etc.
  - driving force behind modern processor design

# How Can the Compiler Improve Performance?

Execution time = Operation count \* Machine cycles per operation

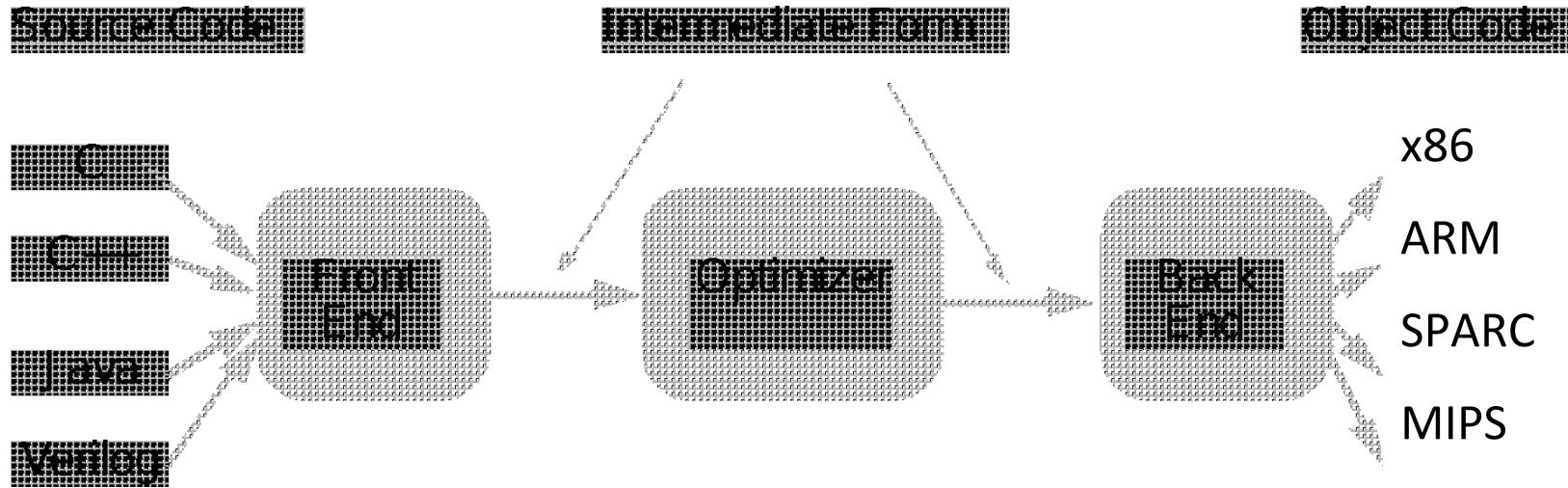
- **Minimize the number of operations**
  - arithmetic operations, memory accesses
- **Replace expensive operations with simpler ones**
  - e.g., replace 4-cycle multiplication with 1-cycle shift
- **Minimize cache misses**
  - both data and instruction accesses
- **Perform work in parallel**
  - instruction scheduling within a thread
  - parallel execution across multiple threads



# What Would You Get Out of This Course?

- Basic knowledge of existing compiler optimizations
- Hands-on experience in constructing optimizations within a fully functional research compiler
- Basic principles and theory for the development of new optimizations

# Structure of a Compiler

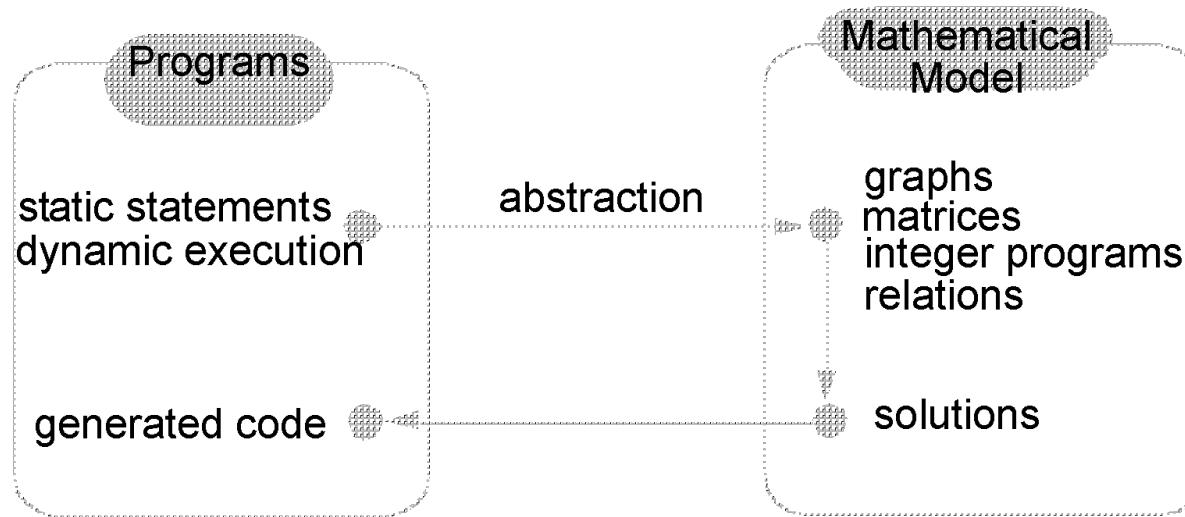


- Optimizations are performed on an “**intermediate form**”
  - similar to a generic RISC instruction set
- Allows easy **portability** to multiple source languages, target machines

# Ingredients in a Compiler Optimization

- **Formulate optimization problem**
  - Identify opportunities of optimization
    - applicable across many programs
    - affect key parts of the program (loops/recursions)
    - amenable to “efficient enough” algorithm
- **Representation**
  - Must **abstract essential details** relevant to optimization

# Ingredients in a Compiler Optimization



# Ingredients in a Compiler Optimization

- **Formulate optimization problem**
  - Identify opportunities of optimization
    - applicable across many programs
    - affect key parts of the program (loops/recursions)
    - amenable to “efficient enough” algorithm
- **Representation**
  - Must abstract essential details relevant to optimization
- **Analysis**
  - Detect when it is desirable and safe to apply transformation
- **Code Transformation**
- **Experimental Evaluation (and repeat process)**

# Representation: Instructions

- **Three-address code**

**A := B op C**

- LHS: name of variable e.g. **x**, **A[t]** (address of **A** + contents of **t**)
- RHS: value

- **Typical instructions**

**A := B op C**

**A := unaryop B**

**A := B**

**GOTO s**

**IF A relop B GOTO s**

**CALL f**

**RETURN**

# Optimization Example

- **Bubblesort program that sorts an array A that is allocated in static storage:**
  - an element of A requires **four bytes** of a byte-addressed machine
  - elements of A are numbered 1 through **n** (**n** is a variable)
  - **A[j]** is in location **&A+4\*(j-1)**

```
FOR i := n-1 DOWNTO 1 DO
    FOR j := 1 TO i DO
        IF A[j]> A[j+1] THEN BEGIN
            temp := A[j];
            A[j] := A[j+1];
            A[j+1] := temp
        END
```

# Translated Code

```
i := n-1
S5: if i<1 goto s1
    j := 1
s4: if j>i goto s2
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2] ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6] ;A[j+1]
    if t3<=t7 goto s3

FOR i := n-1 DOWNTTO 1 DO
    FOR j := 1 TO i DO
        IF A[j]> A[j+1] THEN BEGIN
            temp := A[j];
            A[j] := A[j+1];
            A[j+1] := temp
        END
```

```
t8 :=j-1
t9 := 4*t8
temp := A[t9] ;A[j]
t10 := j+1
t11:= t10-1
t12 := 4*t11
t13 := A[t12] ;A[j+1]
t14 := j-1
t15 := 4*t14
A[t15] := t13 ;A[j]:=A[j+1]
t16 := j+1
t17 := t16-1
t18 := 4*t17
A[t18]:=temp ;A[j+1]:=temp
s3: j := j+1
    goto S4
s2: i := i-1
    goto s5
s1:
```

# Representation: a Basic Block

- **Basic block** = a sequence of 3-address statements
  - only the first statement can be reached from outside the block (no branches into middle of block)
  - all the statements are executed consecutively if the first one is (no branches out or halts except perhaps at end of block)
- We require basic blocks to be *maximal*
  - they cannot be made larger without violating the conditions
- Optimizations within a basic block are *local* optimizations

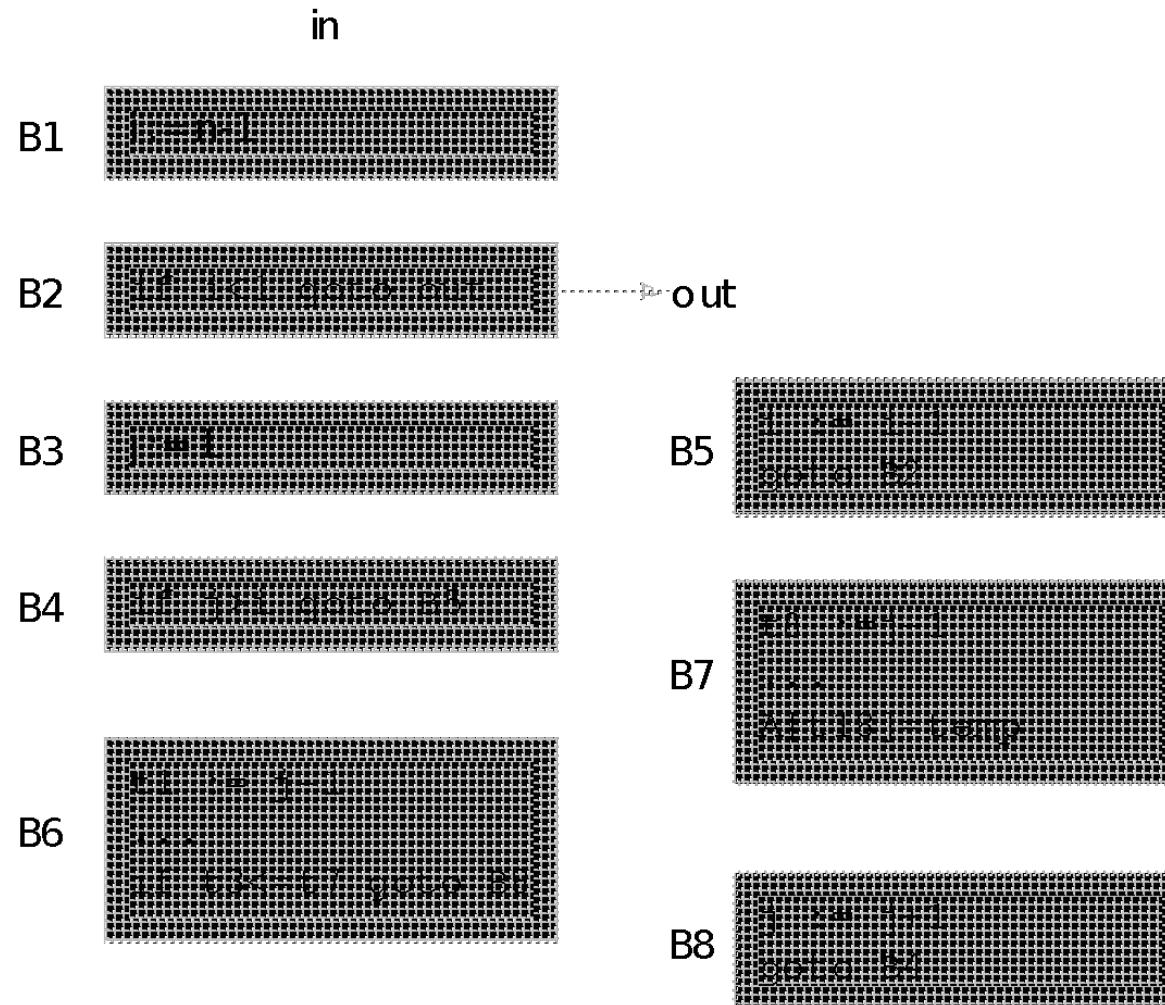
# Flow Graphs

- **Nodes:** basic blocks
- **Edges:**  $B_i \rightarrow B_j$ , iff  $B_j$  can follow  $B_i$  immediately in *some execution*
  - Either first instruction of  $B_j$  is target of a goto at end of  $B_i$
  - Or,  $B_j$  physically follows  $B_i$ , which does not end in an unconditional goto.
- The block led by first statement of the program is the ***start***, or ***entry*** node.

# Find the Basic Blocks

```
i := n-1  
s5: if i<1 goto s1  
     j := 1  
s4: if j>i goto s2  
     t1 := j-1  
     t2 := 4*t1  
     t3 := A[t2] ;A[j]  
     t4 := j+1  
     t5 := t4-1  
     t6 := 4*t5  
     t7 := A[t6] ;A[j+1]  
if t3<=t7 goto s3  
  
t8 :=j-1  
t9 := 4*t8  
temp := A[t9] ;A[j]  
t10 := j+1  
t11:= t10-1  
t12 := 4*t11  
t13 := A[t12] ;A[j+1]  
t14 := j-1  
t15 := 4*t14  
A[t15] := t13 ;A[j]:=A[j+1]  
t16 := j+1  
t17 := t16-1  
t18 := 4*t17  
A[t18]:=temp ;A[j+1]:=temp  
s3: j := j+1  
    goto S4  
s2: i := i-1  
    goto s5  
s1:
```

# Basic Blocks from Example



# Partitioning into Basic Blocks

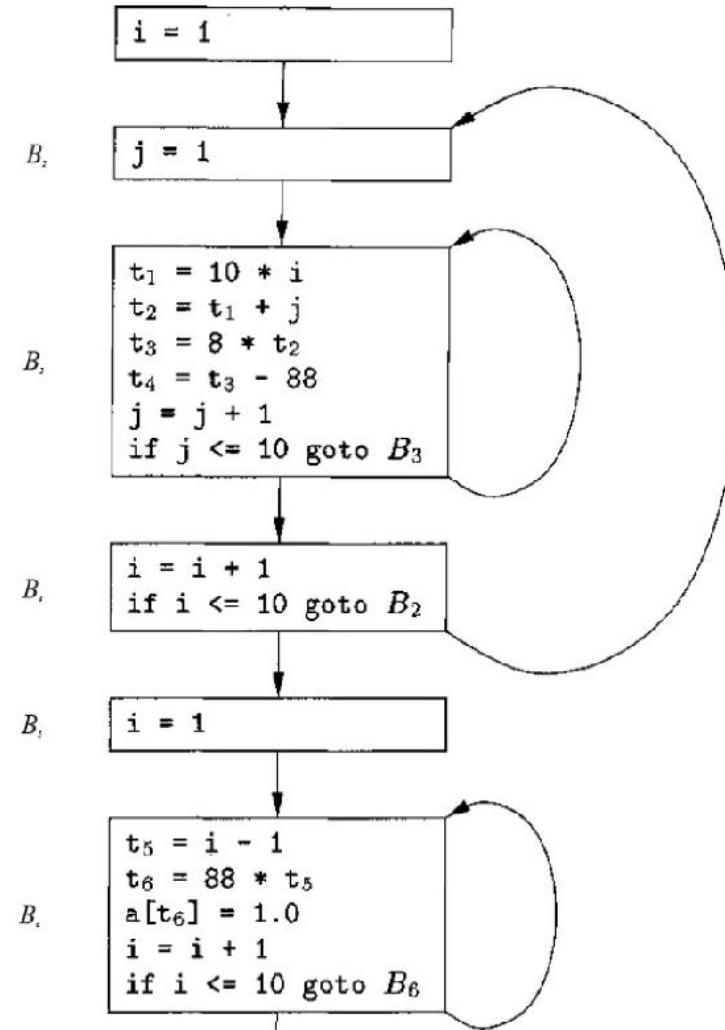
- Identify the leader of each basic block
  - First instruction
  - Any target of a jump
  - Any instruction immediately following a jump
- Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)

```

★ 1) i = 1
★ 2) j = 1
★ 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)
★ 10) i = i + 1
11) if i <= 10 goto (2)
★ 12) i = 1
★ 13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

```

★ = Leader



# Sources of Optimizations

- **Algorithm optimization**

- **Algebraic optimization**

$$A := B + 0 \quad \Rightarrow \quad A := B$$

- **Local optimizations**

- within a basic block -- across instructions

- **Global optimizations**

- within a flow graph -- across basic blocks

- **Interprocedural analysis**

- within a program -- across procedures (flow graphs)

# Local Optimizations

- Analysis & transformation performed **within a basic block**
- No control flow information is considered
- Examples of local optimizations:
  - local common subexpression elimination
    - analysis: same expression evaluated more than once in b.
    - transformation: replace with single calculation
  - local constant folding or elimination
    - analysis: expression can be evaluated at compile time
    - transformation: replace by constant, compile-time value
  - dead code elimination

# Example

```
i := n-1
S5: if i<1 goto s1
    j := 1
s4: if j>i goto s2
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2] ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6] ;A[j+1]
if t3<=t7 goto s3

t8 := j-1
t9 := 4*t8
temp := A[t9] ;A[j]
t10 := j+1
t11 := t10-1
t12 := 4*t11
t13 := A[t12] ;A[j+1]
t14 := j-1
t15 := 4*t14
A[t15] := t13 ;A[j]:=A[j+1]
t16 := j+1
t17 := t16-1
t18 := 4*t17
A[t18]:=temp ;A[j+1]:=temp

s3: j := j+1
    goto S4
S2: i := i-1
    goto s5
s1:
```

# Example

```
B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2]      ;A[j]
    t6 := 4*j
    t7 := A[t6]      ;A[j+1]
    if t3<=t7 goto B8
```

```
B7: t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;temp:=A[j]
    t12 := 4*j
    t13 := A[t12]   ;A[j+1]
    A[t9]:= t13   ;A[j]:=A[j+1]
    A[t12]:=temp  ;A[j+1]:=temp
B8: j := j+1
    goto B4
B5: i := i-1
    goto B2
out:
```

# (Intraprocedural) Global Optimizations

- **Global versions of local optimizations**
  - global common subexpression elimination
  - global constant propagation
  - dead code elimination
- **Loop optimizations**
  - reduce code to be executed in each iteration
  - code motion
  - induction variable elimination
- **Other control structures**
  - Code hoisting: eliminates copies of identical code on parallel paths in a flow graph to reduce code size.

# Example

```
B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2]      ;A[j]
    t6 := 4*j
    t7 := A[t6]      ;A[j+1]
    if t3<=t7 goto B8
```

```
B7: t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;temp:=A[j]
    t12 := 4*j
    t13 := A[t12]   ;A[j+1]
    A[t9]:= t13   ;A[j]:=A[j+1]
    A[t12]:=temp  ;A[j+1]:=temp
B8: j := j+1
    goto B4
B5: i := i-1
    goto B2
out:
```

# Example (After Global CSE)

B1: i := n-1	B7: A[t2] := t7
B2: if i<1 goto out	A[t6] := t3
B3: j := 1	B8: j := j+1
B4: if j>i goto B5	goto B4
B6: t1 := j-1	B5: i := i-1
t2 := 4*t1	goto B2
t3 := A[t2] ;A[j]	out:
t6 := 4*j	
t7 := A[t6] ;A[j+1]	
if t3<=t7 goto B8	

# Induction Variable Elimination

- **Intuitively**
  - Loop indices are induction variables (counting iterations)
  - Linear functions of the loop indices are also induction variables (for accessing arrays)
- **Analysis: detection of induction variable**
- **Optimizations**
  - strength reduction:
    - replace multiplication by additions
  - elimination of loop index:
    - replace termination by tests on other induction variables

# Example

```
B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2]      ;A[j]
    t6 := 4*j
    t7 := A[t6]      ;A[j+1]
    if t3<=t7 goto B8
```

```
B7: A[t2] := t7
     A[t6] := t3
B8: j := j+1
     goto B4
B5: i := i-1
     goto B2
out:
```

# Example (After IV Elimination)

```
B1: i := n-1
B2: if i<1 goto out
B3: t2 := 0
    t6 := 4
B4: t19 := 4*I
    if t6>t19 goto B5
B6: t3 := A[t2]
    t7 := A[t6] ;A[j+1]
    if t3<=t7 goto B8
```

```
B7: A[t2] := t7
    A[t6] := t3
B8: t2 := t2+4
    t6 := t6+4
    goto B4
B5: i := i-1
    goto B2
out:
```

# Loop Invariant Code Motion

- **Analysis**
  - a computation is done within a loop and
  - result of the computation is the same as long as we keep going around the loop
- **Transformation**
  - move the computation outside the loop

# Machine Dependent Optimizations

- Register allocation
- Instruction scheduling
- Memory hierarchy optimizations
- etc.

# Local Optimizations (More Details)

- **Common subexpression elimination**
  - array expressions
  - field access in records
  - access to parameters

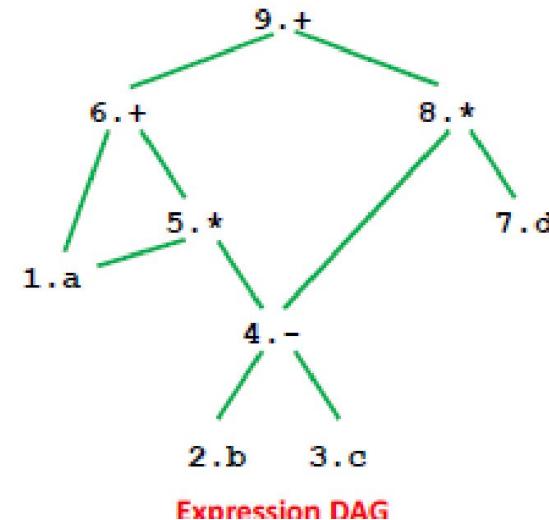
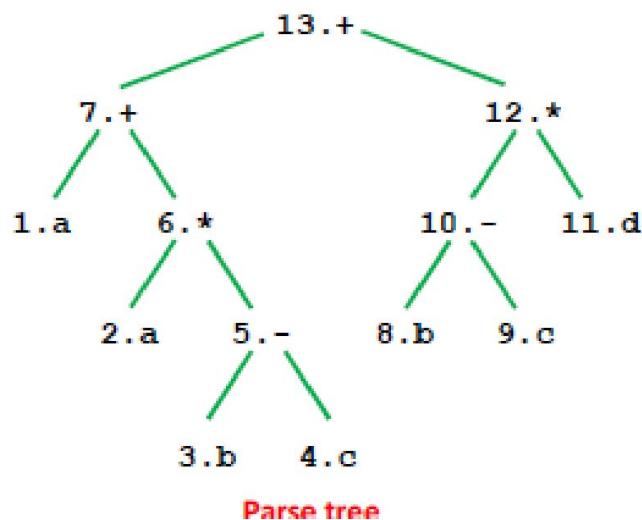
# Graph Abstractions

Example 1:

- grammar (for bottom-up parsing):

$E \rightarrow E + T \mid E - T \mid T, T \rightarrow T^* F \mid F, F \rightarrow ( E ) \mid id$

- expression:  $a + a^*(b - c) + (b - c)^*d$



# Graph Abstractions

Example 1: an expression

$$a + a * (b - c) + (b - c) * d$$

Optimized code:

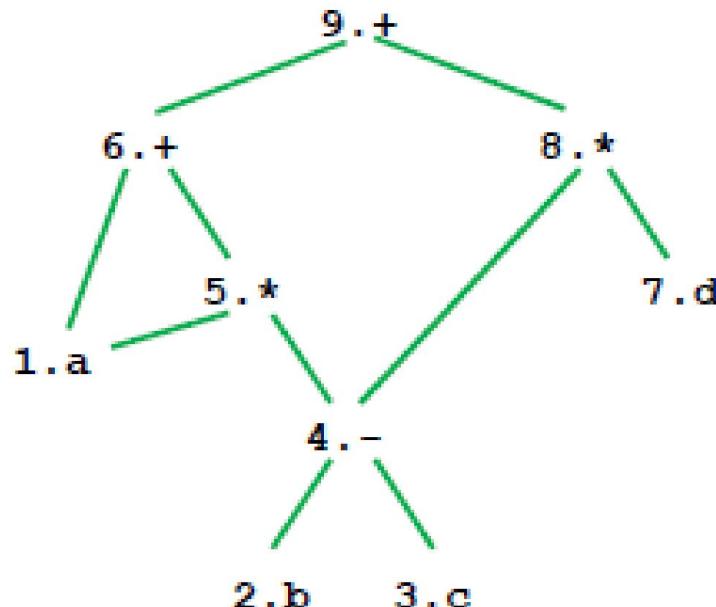
$t1 = b - c$

$t2 = a * t1$

$t3 = a + t2$

$t4 = t1 * d$

$t5 = t3 + t4$



# How well do DAGs hold up across statements?

DAG – directed acyclic graph

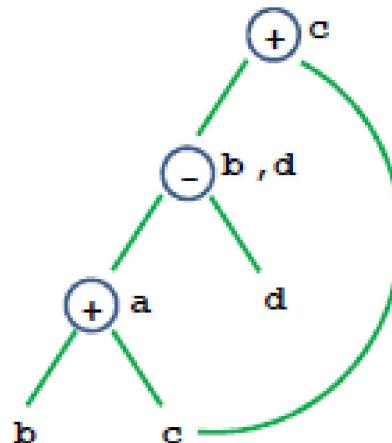
- Example 2

$a = b+c;$

$b = a-d;$

$c = b+c;$

$d = a-d;$



Is this optimized code correct?

$a = b+c;$

$d = a-d;$

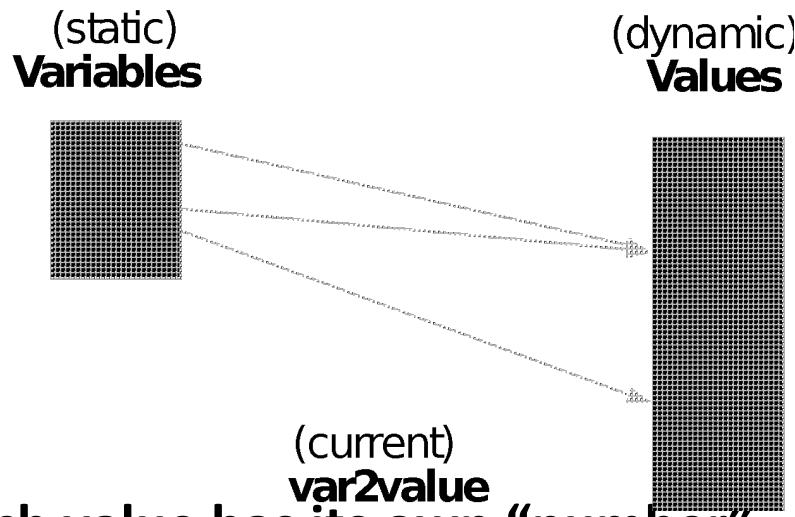
$c = d+c;$

# Critique of DAGs

- **Cause of problems**
  - Assignment statements
  - Value of variable depends on TIME
- **How to fix problem?**
  - build graph in order of execution
  - attach variable name to latest value
- **Final graph created is not very interesting**
  - Key: variable->value mapping across time
  - loses appeal of abstraction

# Value Number: Another Abstraction

- More explicit with respect to VALUES, and TIME



- each value has its own “number”
    - common subexpression means same value number
  - var2value: current map of variable to value
    - used to determine the value number of current expression
- $r1 + r2 \Rightarrow \text{var2value}(r1)+\text{var2value}(r2)$**

# Algorithm

Data structure:

```
VALUES = Table of
    expression      // [OP, valnum1, valnum2]
    var            // name of variable currently holding expression
```

For each instruction (`dst = src1 OP src2`) in execution order

```
valnum1 = var2value(src1); valnum2 = var2value(src2);
```

```
IF [OP, valnum1, valnum2] is in VALUES
```

```
    v = the index of expression
```

```
    Replace instruction with CPY dst = VALUES[v].var
```

```
ELSE
```

```
    Add
```

```
        expression = [OP, valnum1, valnum2]
```

```
        var      = dst
```

```
    to VALUES
```

```
    v = index of new entry; tv is new temporary for v
```

```
    Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                                dst = tv;
```

```
set_var2value (dst, v)
```

# More Details

- **What are the initial values of the variables?**
  - values at beginning of the basic block
- **Possible implementations:**
  - Initialization: create “initial values” for all variables
  - Or dynamically create them as they are used
- **Implementation of VALUES and var2value: hash tables**

# Example

Assign:  $a \rightarrow r1, b \rightarrow r2, c \rightarrow r3, d \rightarrow r4$

$a = b+c;$       ADD  $t1 = r2, r3$

                  CPY  $r1 = t1$

$b = a-d;$       SUB  $t2 = r1, r4$

                  CPY  $r2 = t2$

$c = b+c;$       ADD  $t3 = r2, r3$

                  CPY  $r3 = t3$

$d = a-d;$       SUB  $t4 = r1, r4$

                  CPY  $r4 = t4$

# Conclusions

- **Comparisons of two abstractions**
  - DAGs
  - Value numbering
- **Value numbering**
  - VALUE: distinguish between variables and VALUES
  - TIME
    - Interpretation of instructions in order of execution
    - Keep dynamic state information

# CSC D70: Compiler Optimization Introduction, Logistics

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# CSC D70: Compiler Optimization Dataflow Analysis

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Refreshing from Last Lecture

- Basic Block Formation
- Value Numbering

# Partitioning into Basic Blocks

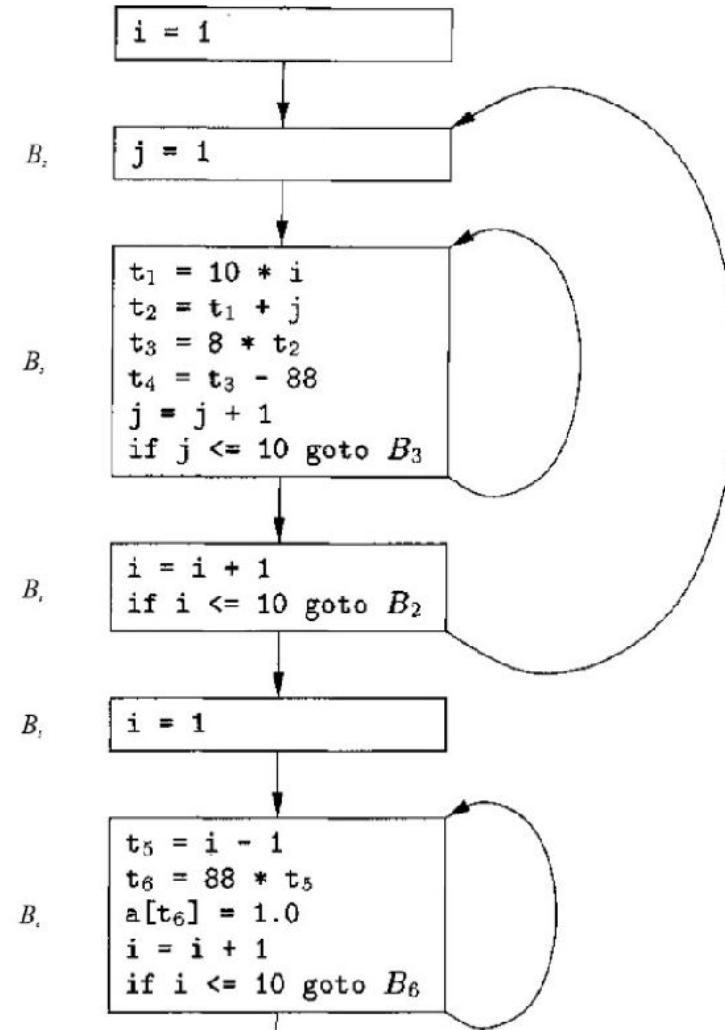
- Identify the leader of each basic block
  - First instruction
  - Any target of a jump
  - Any instruction immediately following a jump
- Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)

```

★ 1) i = 1
★ 2) j = 1
★ 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)
★ 10) i = i + 1
11) if i <= 10 goto (2)
★ 12) i = 1
★ 13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

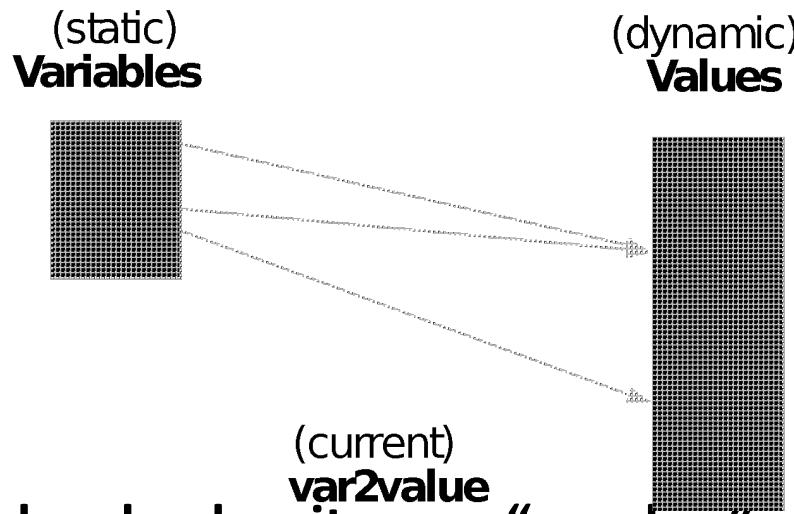
```

★ = Leader



# Value Numbering (VN)

- More explicit with respect to VALUES, and TIME



- each value has its own “number”
    - common subexpression means same value number
  - var2value: current map of variable to value
    - used to determine the value number of current expression
- $r1 + r2 \Rightarrow \text{var2value}(r1)+\text{var2value}(r2)$**

# Algorithm

Data structure:

```
VALUES = Table of
    expression      // [OP, valnum1, valnum2]
    var            // name of variable currently holding expression
```

For each instruction (`dst = src1 OP src2`) in execution order

```
valnum1 = var2value(src1); valnum2 = var2value(src2);
```

```
IF [OP, valnum1, valnum2] is in VALUES
```

```
    v = the index of expression
```

```
    Replace instruction with CPY dst = VALUES[v].var
```

```
ELSE
```

```
    Add
```

```
        expression = [OP, valnum1, valnum2]
```

```
        var      = dst
```

```
    to VALUES
```

```
    v = index of new entry; tv is new temporary for v
```

```
    Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                                CPY dst = tv;
```

```
set_var2value (dst, v)
```

# VN Example

Assign:  $a \rightarrow r1, b \rightarrow r2, c \rightarrow r3, d \rightarrow r4$

$a = b+c;$	ADD t1 = r2, r3
	CPY r1 = t1 // ( $a = t1$ )
$b = a-d;$	SUB t2 = r1, r4
	CPY r2 = t2 // ( $b = t2$ )
$c = b+c;$	ADD t3 = r2, r3
	CPY r3 = t3 // ( $c = t3$ )
$d = a-d;$	CPY r2 = t2

# Questions about Assignment #1

- Tutorial #1
- Tutorial #2 next week
  - More in-depth LLVM coverage

# Outline

1. Structure of data flow analysis
2. Example 1: Reaching definition analysis
3. Example 2: Liveness analysis
4. Generalization

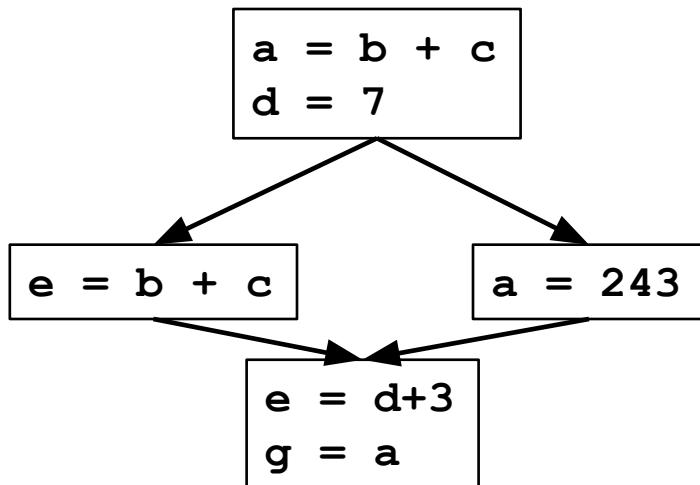
# What is Data Flow Analysis?

- **Local analysis (e.g., value numbering)**
  - analyze effect of each instruction
  - compose effects of instructions to derive information from beginning of basic block to each instruction
- **Data flow analysis**
  - analyze effect of each basic block
  - compose effects of basic blocks to derive information at basic block boundaries
  - from basic block boundaries, apply local technique to generate information on instructions

# What is Data Flow Analysis? (2)

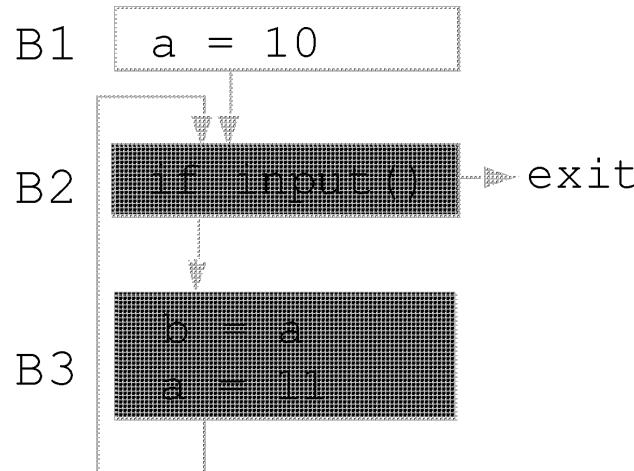
- **Data flow analysis:**
  - Flow-sensitive: sensitive to the control flow in a function
  - intraprocedural analysis
- **Examples of optimizations:**
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination

# What is Data Flow Analysis? (3)



- For each variable  $x$  determine:
- Value of  $x$ ?
  - Which “definition” defines  $x$ ?
  - Is the definition still meaningful (live)?

# Static Program vs. Dynamic Execution



- **Statically:** Finite program
- **Dynamically:** Can have infinitely many possible execution paths
- **Data flow analysis abstraction:**
  - For each point in the program:  
combines information of all the instances of the same program point.
- **Example of a data flow question:**
  - Which definition defines the value used in statement “ $b = a$ ”?

# Effects of a Basic Block

- Effect of a statement:  $a = b+c$ 
  - **Uses** variables (b, c)
  - **Kills** an old definition (old definition of a)
  - new **definition** (a)
- Compose effects of statements -> Effect of a basic block
  - A **locally exposed use** in a b.b. is a use of a data item which is not preceded in the b.b. by a definition of the data item
  - any definition of a data item in the basic block **kills** all definitions of the same data item reaching the basic block.
  - A **locally available definition** = last definition of data item in b.b.

# Effects of a Basic Block

A **locally available definition** = last definition of data item in b.b.

t1 = r1+r2

**Locally exposed uses?** r1

r2 = t1

t2 = r2+r1

**Kills any definitions?** Any other  
definition  
of t2

r1 = t2

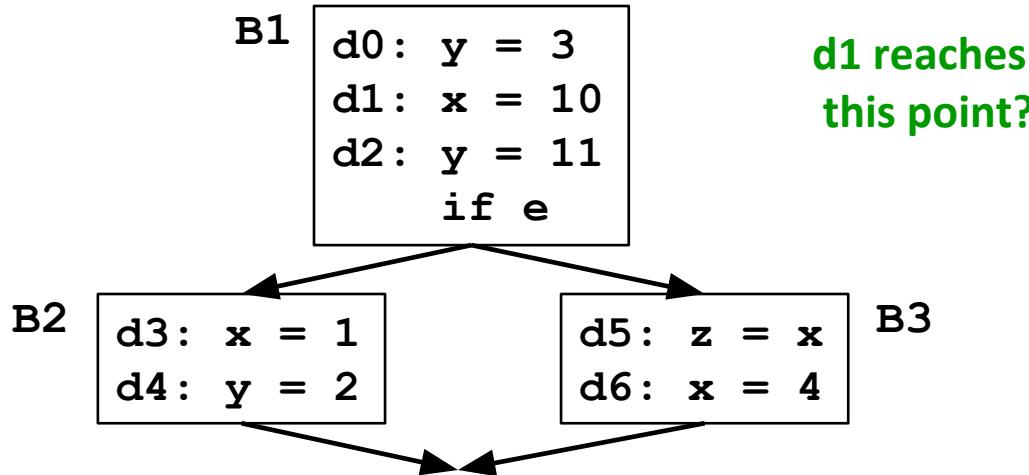
t3 = r1\*r1

r2 = t3

if r2>100 goto L1

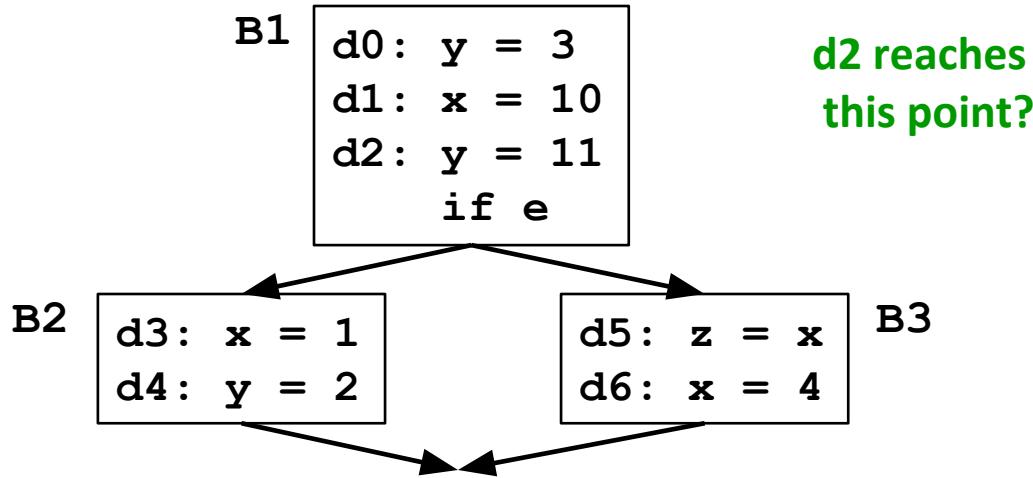
**Locally avail. definition?** t2

# Reaching Definitions



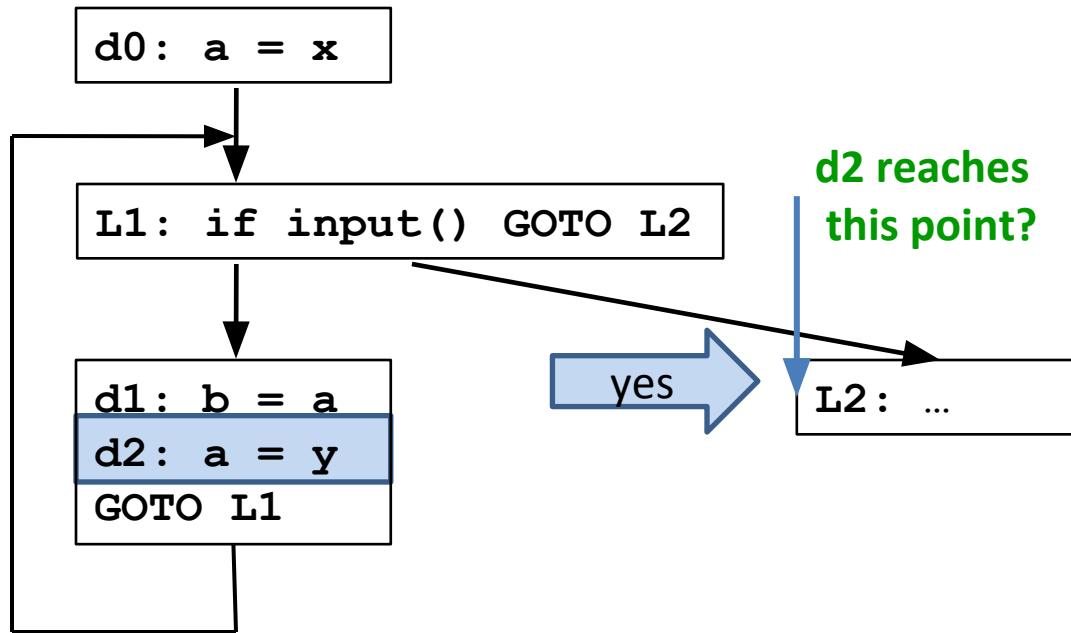
- Every assignment is a **definition**
- A **definition  $d$  reaches** a point  $p$  if **there exists** path from the point immediately following  $d$  to  $p$  such that  $d$  is **not killed** (overwritten) along that path.
- Problem statement
  - For each point in the program, determine if each definition in the program reaches the point
  - A bit vector per program point, vector-length = #defs

# Reaching Definitions (2)

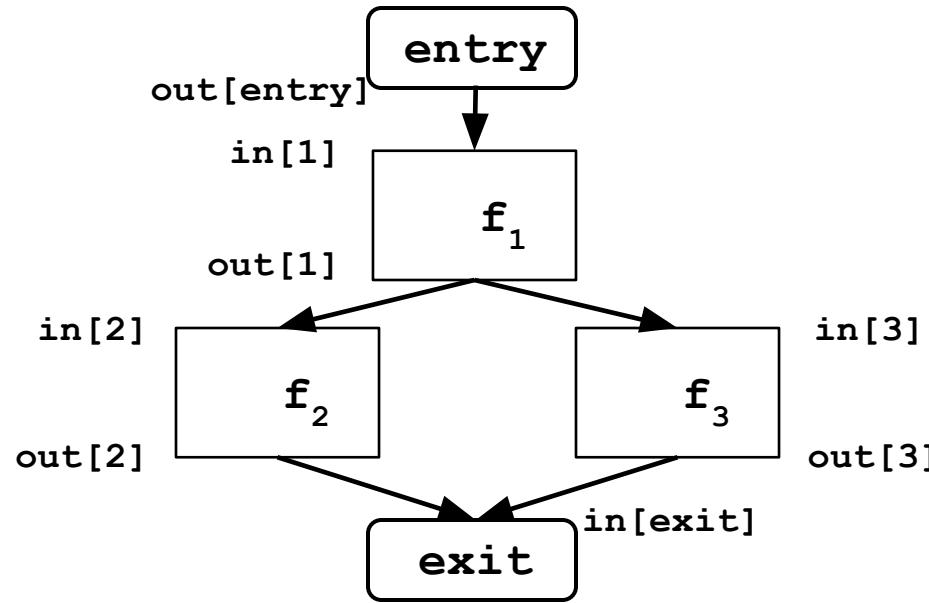


- Every assignment is a **definition**
- A **definition**  $d$  **reaches** a point  $p$  if **there exists** path from the point immediately following  $d$  to  $p$  such that  $d$  is **not killed** (overwritten) along that path.
- Problem statement
  - For each point in the program, determine if each definition in the program reaches the point
  - A bit vector per program point, vector-length = #defs

# Reaching Definitions (3)

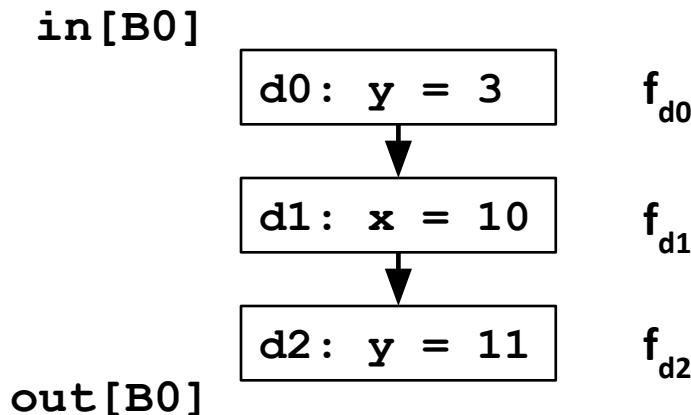


# Data Flow Analysis Schema



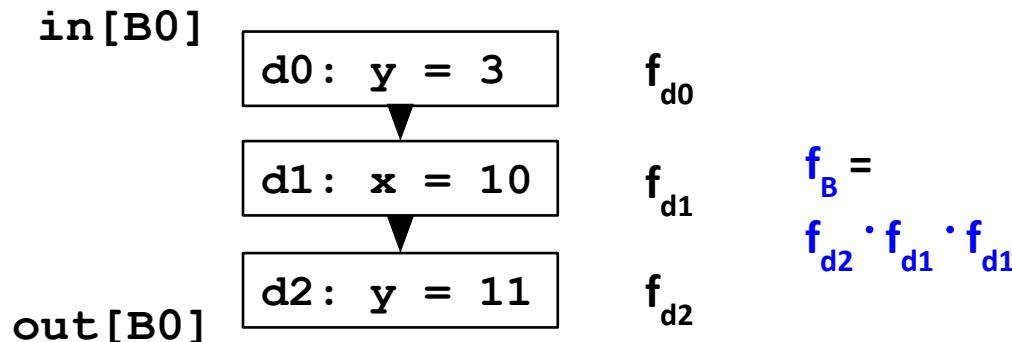
- Build a **flow graph** (nodes = basic blocks, edges = control flow)
- Set up a set of equations between  $\text{in}[b]$  and  $\text{out}[b]$  for all basic blocks  $b$ 
  - Effect of **code in basic block**:
    - Transfer function  $f_b$  relates  $\text{in}[b]$  and  $\text{out}[b]$ , for same  $b$
  - Effect of **flow of control**:
    - relates  $\text{out}[b_1], \text{in}[b_2]$  if  $b_1$  and  $b_2$  are adjacent
- Find a solution to the equations

# Effects of a Statement



- $f_s$ : A transfer function of a statement
  - abstracts the execution with respect to the problem of interest
- For a statement  $s$  ( $d: x = y + z$ )  
 $\text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s] - \text{Kill}[s])$ 
  - **Gen[s]**: definitions generated:  $\text{Gen}[s] = \{d\}$
  - **Propagated definitions**:  $\text{in}[s] - \text{Kill}[s]$ , where **Kill[s]**=set of all other defs to x in the rest of program

# Effects of a Basic Block



- Transfer function of a statement  $s$ :
  - $\text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s] - \text{Kill}[s])$
- Transfer function of a **basic block  $B$** :
  - Composition of transfer functions of statements in  $B$
- $\text{out}[B] = f_B(\text{in}[B]) = f_{d_2} f_{d_1} f_{d_0}(\text{in}[B])$ 

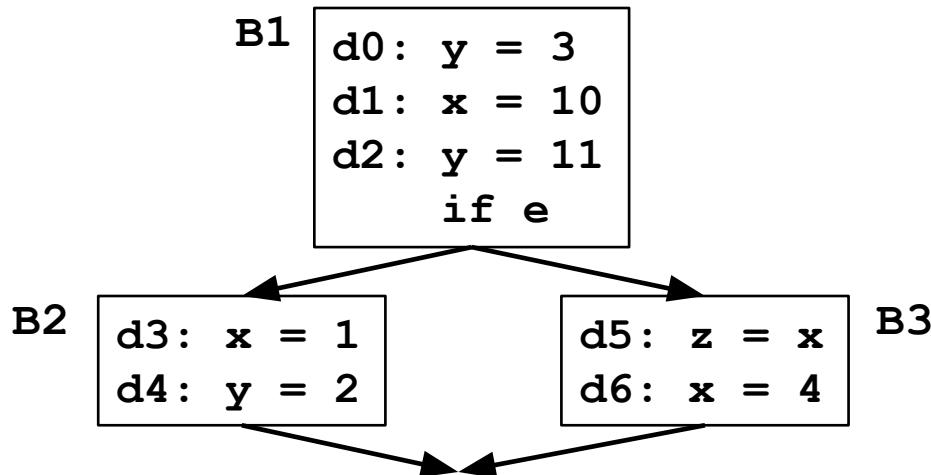
$$= \text{Gen}[d_2] \cup (\text{Gen}[d_1] \cup (\text{Gen}[d_0] \cup (\text{in}[B] - \text{Kill}[d_0])) - \text{Kill}[d_1])) - \text{Kill}[d_2]$$

$$= \text{Gen}[d_2] \cup (\text{Gen}[d_1] \cup (\text{Gen}[d_0] - \text{Kill}[d_1]) - \text{Kill}[d_2]) \cup$$

$$\quad \text{in}[B] - (\text{Kill}[d_0] \cup \text{Kill}[d_1] \cup \text{Kill}[d_2])$$

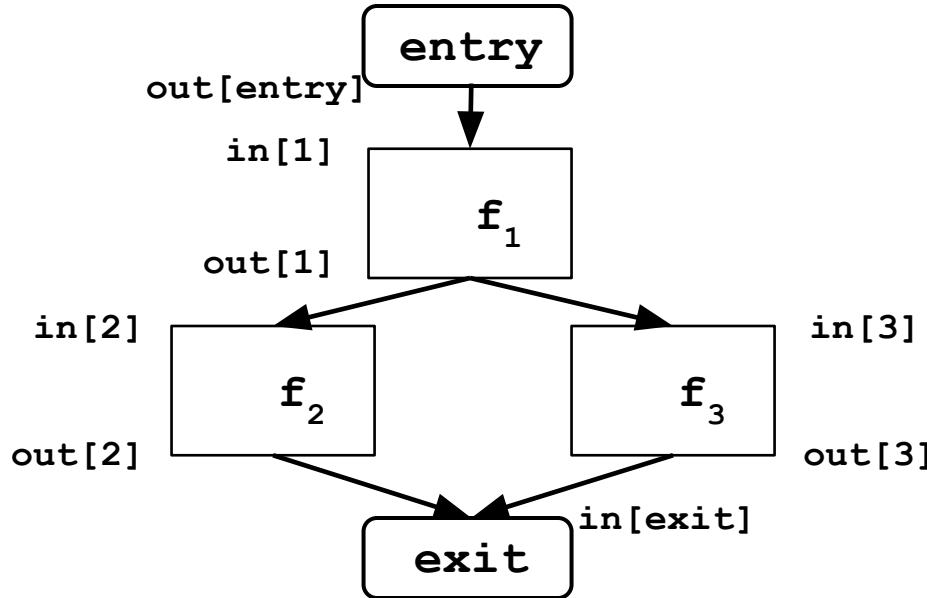
$$= \text{Gen}[B] \cup (\text{in}[B] - \text{Kill}[B])$$
  - $\text{Gen}[B]$ : locally exposed definitions (available at end of bb)
  - $\text{Kill}[B]$ : set of definitions killed by  $B$

# Example



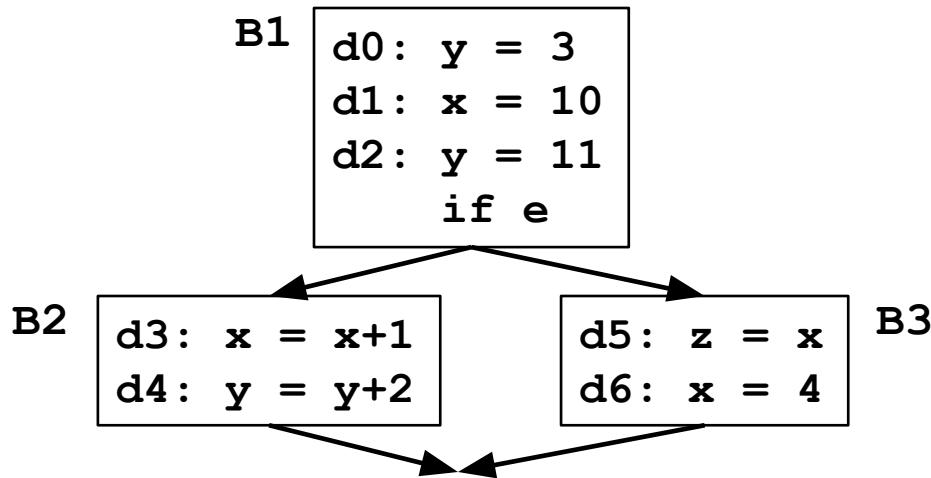
- a **transfer function**  $f_b$  of a basic block  $b$ :  
$$\text{OUT}[b] = f_b(\text{IN}[b])$$
incoming reaching definitions -> outgoing reaching definitions
- A basic block  $b$ 
  - **generates** definitions:  $\text{Gen}[b]$ ,
    - set of locally available definitions in  $b$
  - **kills** definitions:  $\text{in}[b] - \text{Kill}[b]$ ,  
where  $\text{Kill}[b] = \text{set of defs (in rest of program) killed by defs in } b$
- **out[b] = Gen[b] U (in(b)-Kill[b])**

# Effects of the Edges (acyclic)



- $\text{out}[b] = f_b(\text{in}[b])$
- Join node: a node with multiple predecessors
- **meet** operator:  
$$\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \dots \cup \text{out}[p_n]$$
, where  
 $p_1, \dots, p_n$  are all predecessors of b

# Example

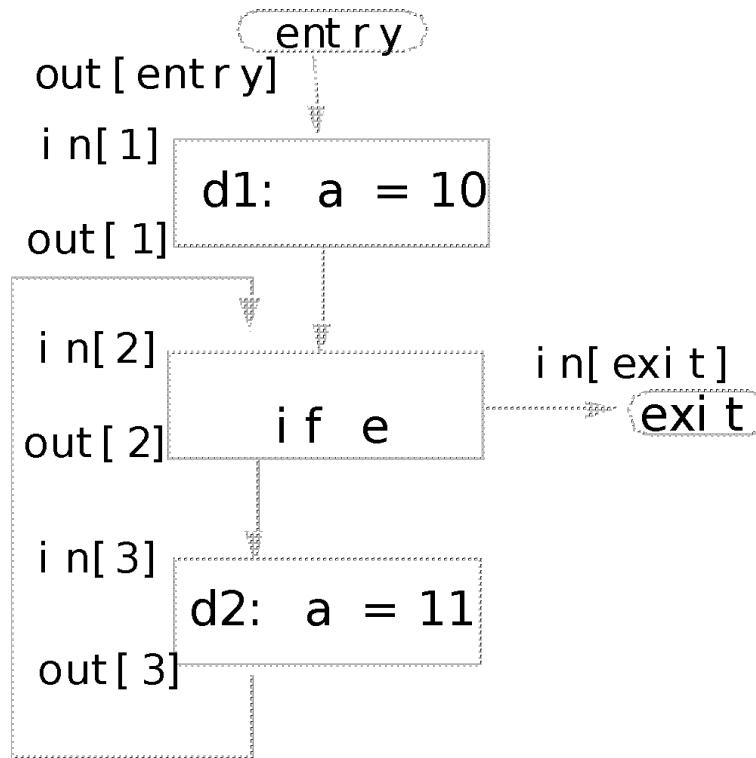


f	Gen	Kill
1	{1,2}	{0,2,3,4,6}
2	{3,4}	{0,1,2,6}
3	{5,6}	{1,3}

- $\text{out}[b] = f_b(\text{in}[b])$
- Join node: a node with multiple predecessors
- **meet** operator:

$\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \dots \cup \text{out}[p_n]$ , where  
 $p_1, \dots, p_n$  are all predecessors of b

# Cyclic Graphs



- Equations still hold
  - $\text{out}[b] = f_b(\text{in}[b])$
  - $\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \dots \cup \text{out}[p_n], p_1, \dots, p_n \text{ pred.}$
- Find: fixed point solution

# Reaching Definitions: Iterative Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
out[Entry] = ∅

// Initialization for iterative algorithm
For each basic block B other than Entry
    out[B] = ∅

// iterate
While (Changes to any out[] occur) {
    For each basic block B other than Entry {
        in[B] = U (out[p]), for all predecessors p of B
        out[B] = fB(in[B])      // out[B]=gen[B] U (in[B]-kill[B])
    }
}
```

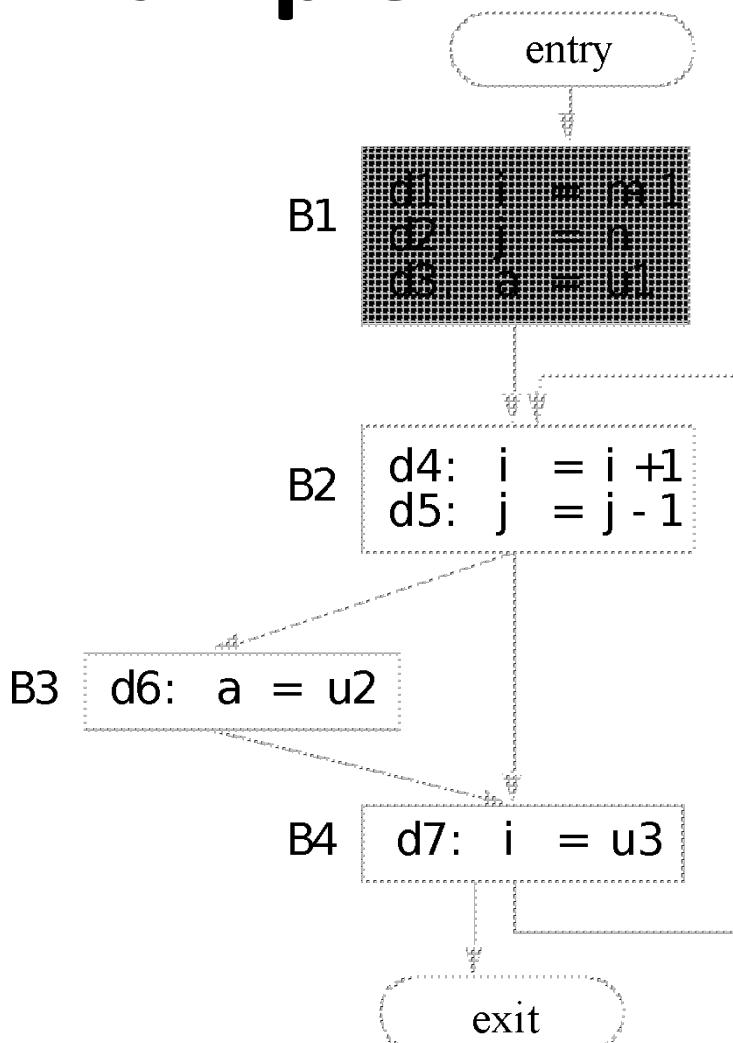
# Reaching Definitions: Worklist Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
    out[Entry] = ∅          // can set out[Entry] to special def
                            // if reaching then undefined use
    For all nodes i
        out[i] = ∅          // can optimize by out[i]=gen[i]
    ChangedNodes = N

// iterate
    While ChangedNodes ≠ ∅ {
        Remove i from ChangedNodes
        in[i] = U (out[p]), for all predecessors p of i
        oldout = out[i]
        out[i] = fi(in[i])    // out[i]=gen[i]U(in[i]-kill[i])
        if (oldout ≠ out[i]) {
            for all successors s of i
                add s to ChangedNodes
        }
    }
```

# Example



	First Pass	Second Pass
IN[B1]	000 00 0 0	000 00 0 0
OUT[B1]	111 00 0 0	111 00 0 0
IN[B2]	111 00 0 0	111 01 1 1
OUT[B2]	001 11 0 0	001 11 1 0
IN[B3]	001 11 0 0	001 11 1 0
OUT[B3]	000 11 1 0	000 11 1 0
IN[B4]	001 11 1 0	001 11 1 0
OUT[B4]	001 01 1 1	001 01 1 1
IN[exit]	001 01 1 1	001 01 1 1

# Live Variable Analysis

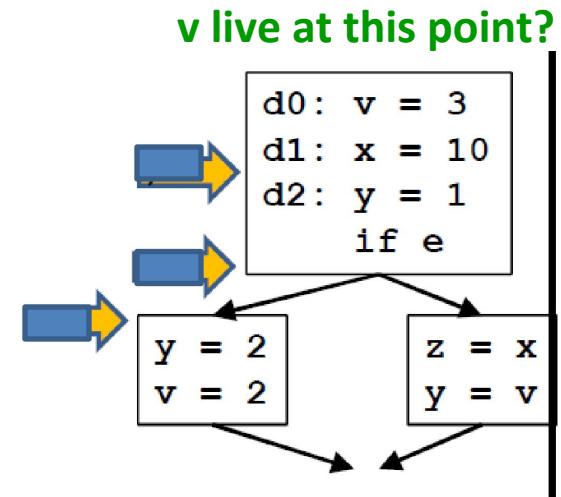
- **Definition**

- A variable **v** is **live** at point *p* if
  - the value of **v** is used along some path in the flow graph starting at *p*.
- Otherwise, the variable is **dead**.

- **Motivation**

- e.g. register allocation

```
for i = 0 to n  
  ... i ...  
...  
for i = 0 to n  
  ... i ...
```



- **Problem statement**

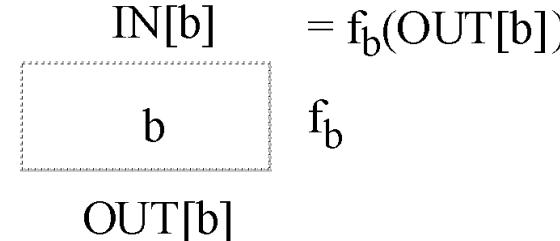
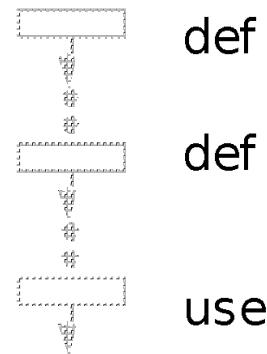
- For each basic block
  - determine if each variable is live in each basic block
- Size of bit vector: one bit for each variable

# Transfer Function

- Insight: Trace uses backwards to the definitions an execution path

control flow

example



d3:  $a = 1$   
d4:  $b = 1$

d5:  $c = a$   
d6:  $a = 4$

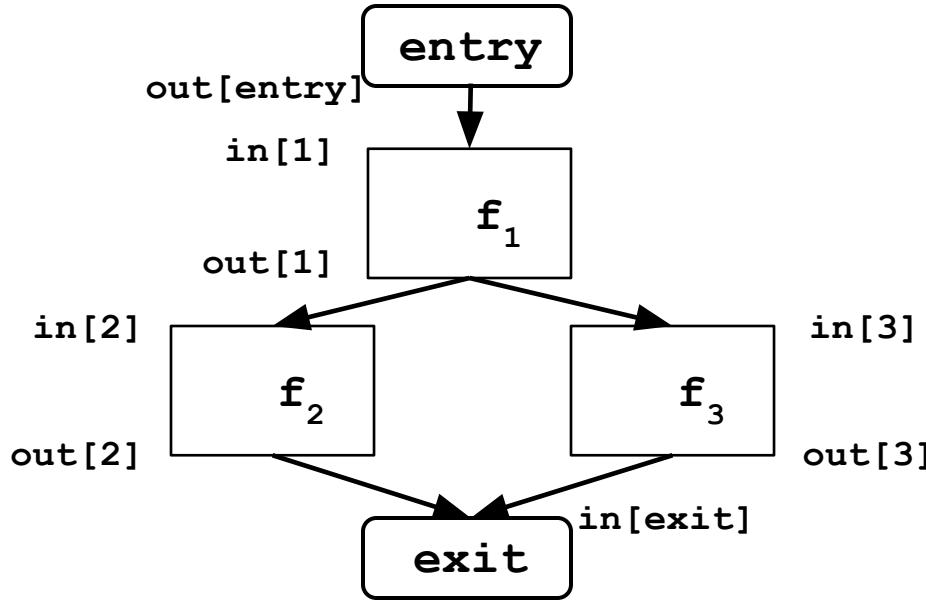
- A basic block b can

- generate live variables: **Use[b]**
  - set of locally exposed uses in b
- propagate incoming live variables: **OUT[b]** - **Def[b]**,
  - where **Def[b]**= set of variables defined in b.b.

- **transfer function** for block b:

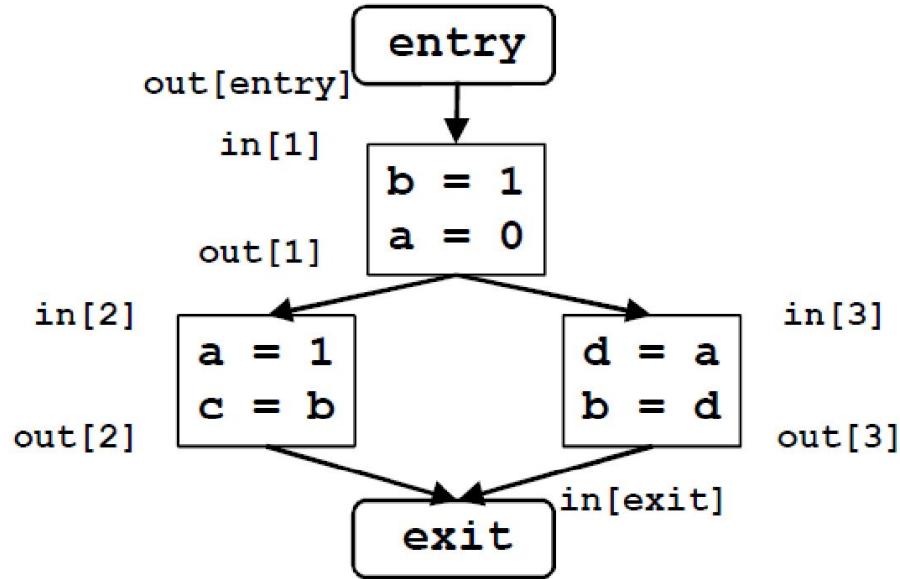
$$\text{in}[b] = \text{Use}[b] \cup (\text{out}(b) - \text{Def}[b])$$

# Flow Graph



- $\text{in}[b] = f_b(\text{out}[b])$
- **Join node:** a node with multiple **successors**
- **meet** operator:  
$$\text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \dots \cup \text{in}[s_n], \text{ where}$$
$$s_1, \dots, s_n \text{ are all successors of } b$$

# Flow Graph (2)



f	Use	Def
1	{ }	{a,b}
2	{b}	{a,c}
3	{a}	{b,d}

- $\text{in}[b] = f_b(\text{out}[b])$
- **Join node**: a node with multiple **successors**
- **meet** operator:  
$$\text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \dots \cup \text{in}[s_n],$$
 where  $s_1, \dots, s_n$  are all successors of  $b$

# Liveness: Iterative Algorithm

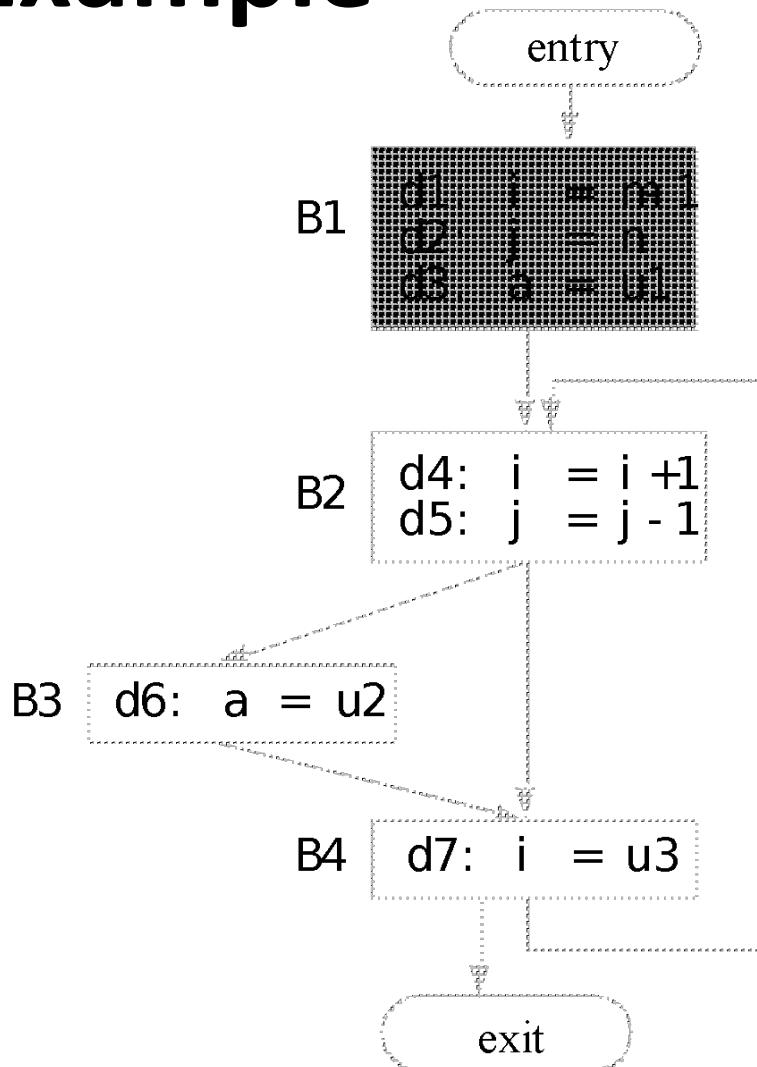
```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
in[Exit] = ∅

// Initialization for iterative algorithm
For each basic block B other than Exit
    in[B] = ∅

// iterate
While (Changes to any in[] occur) {
    For each basic block B other than Exit {
        out[B] = U (in[s]), for all successors s of B
        in[B] = fB(out[B])      // in[B]=Use[B] U (out[B]-Def[B])
    }
}
```

# Example



	First Pass	Second Pass
OUT[entry]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
IN[B1]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
OUT[B1]	{i,j,u2,u3}	{i,j,u2,u3}
IN[B2]	{i,j,u2,u3}	{i,j,u2,u3}
OUT[B2]	{u2,u3}	{j,u2,u3}
IN[B3]	{u2,u3}	{j,u2,u3}
OUT[B3]	{u3}	{j,u2,u3}
IN[B4]	{u3}	{j,u2,u3}
OUT[B4]	{}	{i,j,u2,u3}

# Framework

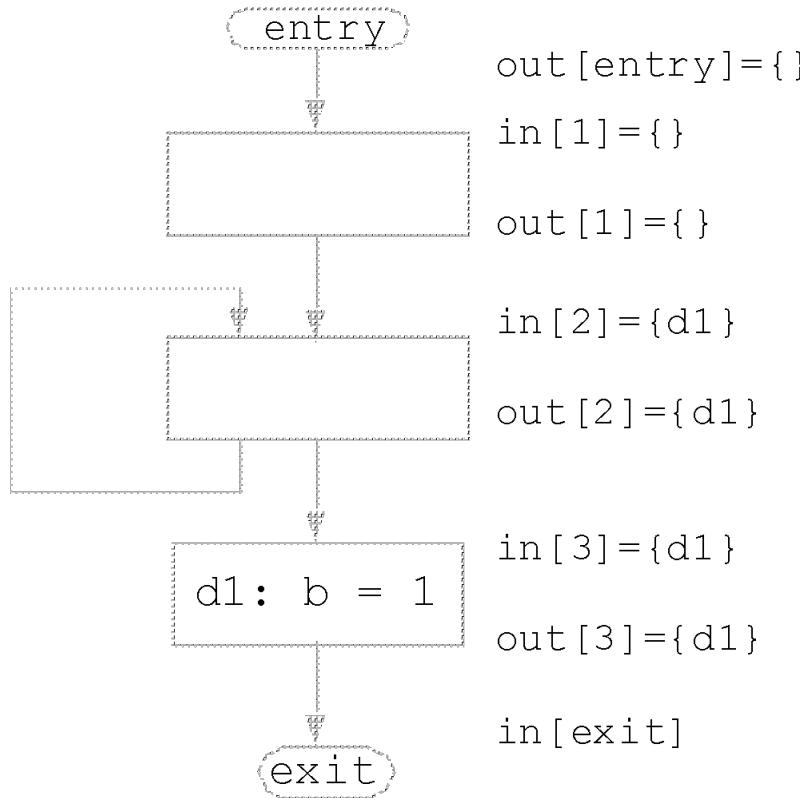
	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward: $\text{out}[b] = f_b(\text{in}[b])$ $\text{in}[b] = \bigwedge \text{out}[\text{pred}(b)]$	backward: $\text{in}[b] = f_b(\text{out}[b])$ $\text{out}[b] = \bigwedge \text{in}[\text{succ}(b)]$
Transfer function	$f_b(x) = \text{Gen}_b \cup (x - \text{Kill}_b)$	$f_b(x) = \text{Use}_b \cup (x - \text{Def}_b)$
Meet Operation ( $\wedge$ )	$\cup$	$\cup$
Boundary Condition	$\text{out}[\text{entry}] = \emptyset$	$\text{in}[\text{exit}] = \emptyset$
Initial interior points	$\text{out}[b] = \emptyset$	$\text{in}[b] = \emptyset$

Other examples (e.g., Available expressions), defined in ALSU 9.2.6

# Thought Problem 1. “Must-Reach” Definitions

- A definition D ( $a = b+c$ ) must reach point P iff
  - D appears at least once along all paths leading to P
  - a is not redefined along any path after last appearance of D and before P
- How do we formulate the data flow algorithm for this problem?

# Thought Problem 2: A legal solution to (May) Reaching Def?



- Will the worklist algorithm generate this answer?

# Questions

- **Correctness**
  - equations are satisfied, if the program terminates.
- **Precision: how good is the answer?**
  - is the answer ONLY a union of all possible executions?
- **Convergence: will the analysis terminate?**
  - or, will there always be some nodes that change?
- **Speed: how fast is the convergence?**
  - how many times will we visit each node?

# **Foundations of Data Flow Analysis**

- 1. Meet operator**
- 2. Transfer functions**
- 3. Correctness, Precision, Convergence**
- 4. Efficiency**

- Reference: ALSU pp. 613-631
- Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
- Marlowe & Ryder, Properties of data flow frameworks: a unified model. Rutgers tech report, Apr. 1988

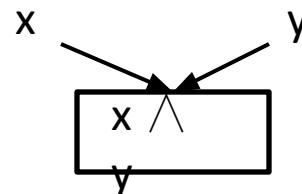
# A Unified Framework

- Data flow problems are defined by
  - Domain of values:  $V$
  - Meet operator ( $V \wedge V \rightarrow V$ ), initial value
  - A set of transfer functions ( $V \rightarrow V$ )
- Usefulness of unified framework
  - To answer questions such as correctness, precision, convergence, speed of convergence for a family of problems
    - If meet operators and transfer functions have properties X, then we know Y about the above.
  - Reuse code

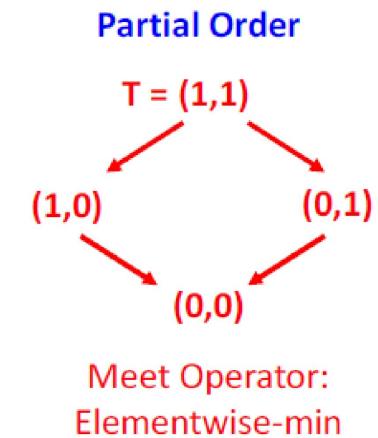
# Meet Operator

- Properties of the meet operator

- commutative:  $x \wedge y = y \wedge x$



- idempotent:  $x \wedge x = x$
- associative:  $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- there is a Top element  $T$  such that  $x \wedge T = x$

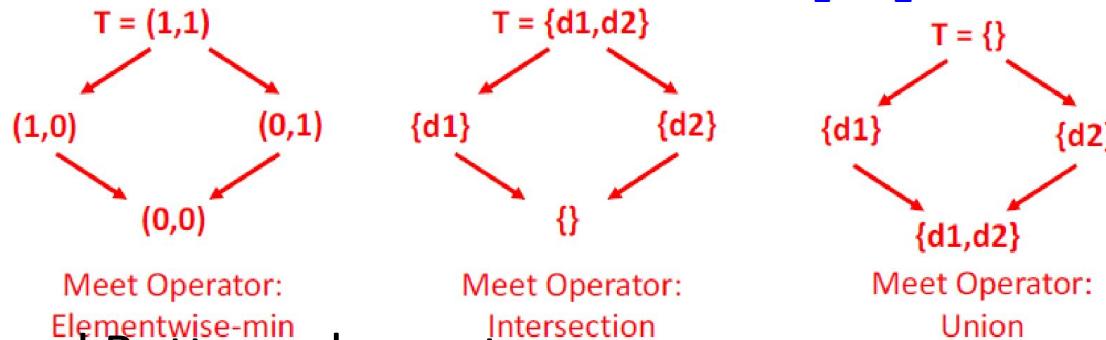


- Meet operator defines a partial ordering on values

- $x \leq y$  if and only if  $x \wedge y = x$  ( $y \rightarrow x$  in diagram)
  - Transitivity: if  $x \leq y$  and  $y \leq z$  then  $x \leq z$
  - Antisymmetry: if  $x \leq y$  and  $y \leq x$  then  $x = y$
  - Reflexitivity:  $x \leq x$

# Partial Order

- Example: let  $V = \{x \mid \text{such that } x \subseteq \{\mathbf{d}_1, \mathbf{d}_2\}\}$ ,  $\wedge = \cap$



- Top and Bottom elements

- Top  $T$  such that:  $x \wedge T = x$
- Bottom  $\perp$  such that:  $x \wedge \perp = \perp$

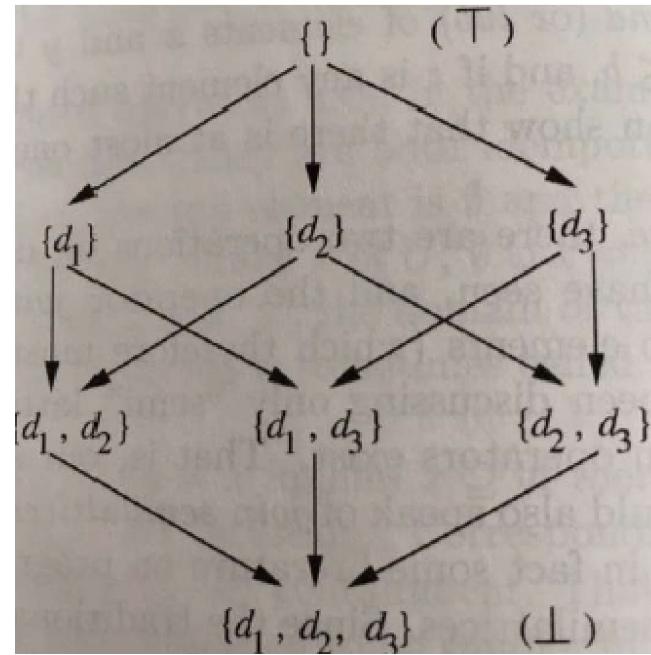
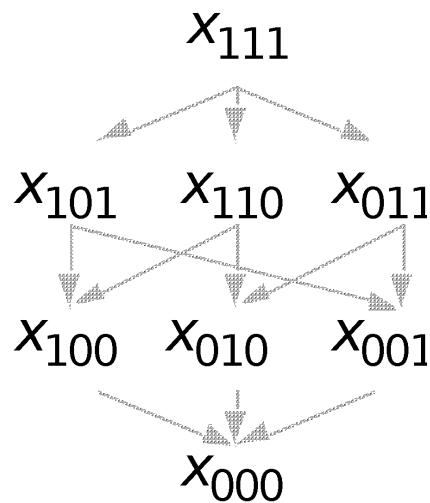
- Values and meet operator in a data flow problem define a semi-lattice:
  - there exists a  $T$ , but not necessarily a  $\perp$ .
- $x, y$  are ordered:  $x \leq y$  then  $x \wedge y = x$  ( $y \rightarrow x$  in diagram)
- what if  $x$  and  $y$  are not ordered?
  - $x \wedge y \leq x, x \wedge y \leq y$ , and if  $w \leq x, w \leq y$ , then  $w \leq x \wedge y$

# One vs. All Variables/Definitions

- Lattice for each variable: e.g. intersection



- Lattice for three variables:



# Descending Chain

- **Definition**
  - The **height** of a lattice is the largest number of **> relations** that will fit in a descending chain.
$$x_0 > x_1 > x_2 > \dots$$
- **Height of values in reaching definitions?**

Height n – number of definitions
- **Important property: finite descending chain**
- **Can an infinite lattice have a finite descending chain?**

yes
- **Example: Constant Propagation/Folding**
  - To determine if a variable is a constant
- **Data values**
  - undef, ... -1, 0, 1, 2, ..., not-a-constant

# Transfer Functions

- **Basic Properties  $f: V \rightarrow V$** 
  - Has an identity function
    - There exists an  $f$  such that  $f(x) = x$ , for all  $x$ .
  - Closed under composition
    - if  $f_1, f_2 \in F$ , then  $f_1 \cdot f_2 \in F$

# Monotonicity

- A framework  $(F, V, \wedge)$  is **monotone** if and only if
  - $x \leq y$  implies  $f(x) \leq f(y)$
  - i.e. a “smaller or equal” input to the same function will always give a “smaller or equal” output
- Equivalently, a framework  $(F, V, \wedge)$  is **monotone** if and only if
  - $f(x \wedge y) \leq f(x) \wedge f(y)$
  - i.e. merge input, then apply  $f$  is **small than or equal to** apply the transfer function individually and then merge the result

# Example

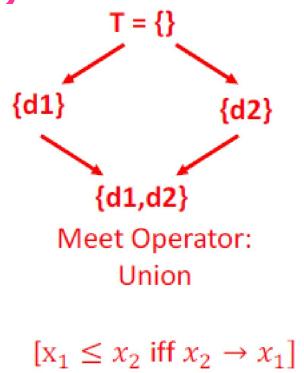
- Reaching definitions:  $f(x) = \text{Gen} \cup (x - \text{Kill})$ ,  $\wedge = \cup$

- Definition 1:

- $x_1 \leq x_2, \text{Gen} \cup (x_1 - \text{Kill}) \leq \text{Gen} \cup (x_2 - \text{Kill})$

- Definition 2:

- $$\begin{aligned} & (\text{Gen} \cup (x_1 - \text{Kill})) \cup (\text{Gen} \cup (x_2 - \text{Kill})) \\ &= (\text{Gen} \cup ((x_1 \cup x_2) - \text{Kill})) \end{aligned}$$



- Note: Monotone framework does not mean that  $f(x) \leq x$

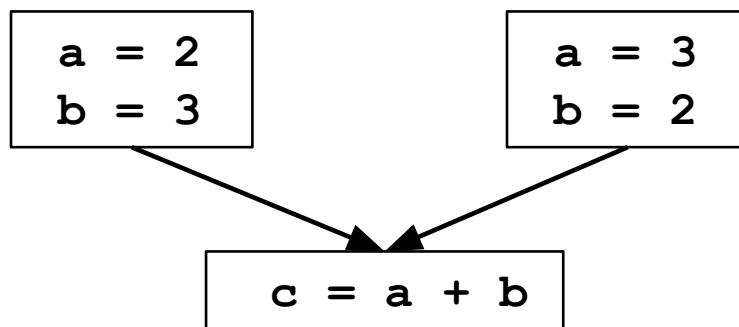
- e.g., reaching definition for two definitions in program
  - suppose:  $f_x : \text{Gen}_x = \{d_1, d_2\}; \text{Kill}_x = \{\}$

- If  $\text{input(second iteration)} \leq \text{input(first iteration)}$

- $\text{result(second iteration)} \leq \text{result(first iteration)}$

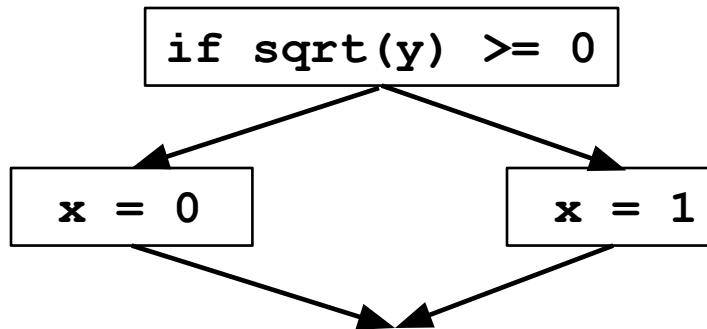
# Distributivity

- A framework  $(F, V, \wedge)$  is **distributive** if and only if
  - $f(x \wedge y) = f(x) \wedge f(y)$
  - i.e. merge input, then apply  $f$  is **equal to** apply the transfer function individually then merge result
- Example: Constant Propagation is NOT distributive



# Data Flow Analysis

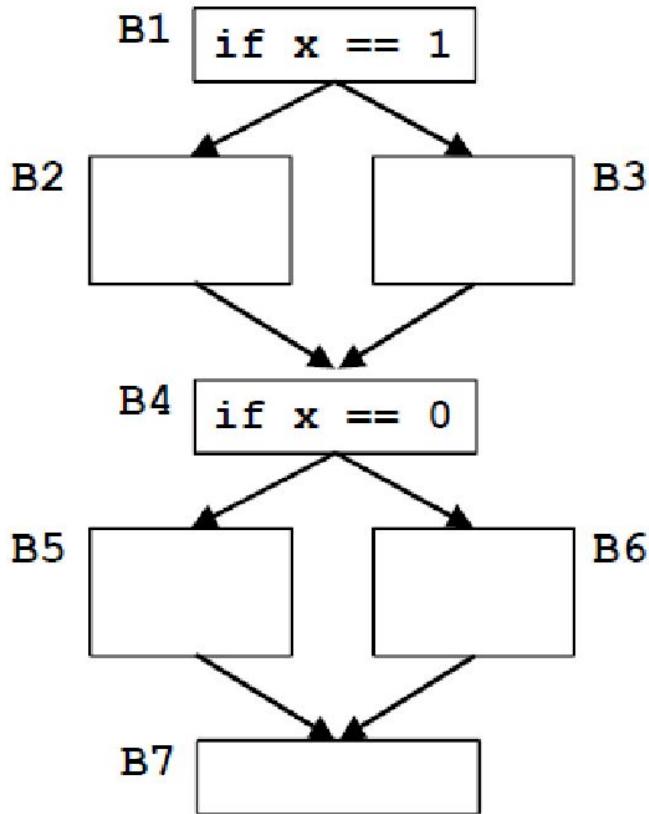
- **Definition**
  - Let  $f_1, \dots, f_m : \in F$ , where  $f_i$  is the transfer function for node  $i$ 
    - $f_p = f_{n_k} \cdot \dots \cdot f_{n_1}$ , where  $p$  is a path through nodes  $n_1, \dots, n_k$
    - $f_p$  = identify function, if  $p$  is an empty path
- **Ideal data flow answer:**
  - For each node  $n$ :  
 $\wedge f_{pi}(T)$ , for all possibly executed paths  $p_i$  reaching  $n$ .
- But determining all possibly executed paths is undecidable



# Meet-Over-Paths (MOP)

- Error in the conservative direction
- Meet-Over-Paths (MOP):
  - For each node  $n$ :
$$\text{MOP}(n) = \bigwedge f_{pi}(\text{T}), \text{ for all paths } p_i \text{ reaching } n$$
  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - MOP = Perfect-Solution  $\wedge$  Solution-to-Unexecuted-Paths
  - MOP  $\leq$  Perfect-Solution
  - Potentially more constrained, solution is small
    - hence *conservative*
  - It is not **safe** to be  $>$  Perfect-Solution!
- **Desirable solution: as close to MOP as possible**

# MOP Example



Ideal: Considers only 2 paths

B1-B2-B4-B6-B7 (i.e., x=1)

B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths

B1-B2-B4-B5-B7

B1-B3-B4-B6-B7

Assume: B2 & B3 do not update x

# Solving Data Flow Equations

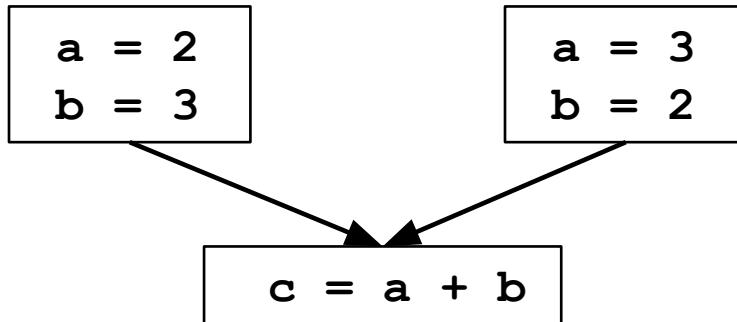
- Example: Reaching definitions
  - $\text{out}[\text{entry}] = \{\}$
  - Values = {subsets of definitions}
  - Meet operator:  $\cup$ 
    - $\text{in}[b] = \cup \text{out}[p]$ , for all predecessors  $p$  of  $b$
  - Transfer functions:  $\text{out}[b] = \text{gen}_b \cup (\text{in}[b] - \text{kill}_b)$
- Any solution satisfying equations = Fixed Point Solution (FP)
- Iterative algorithm
  - initializes  $\text{out}[b]$  to  $\{\}$
  - if converges, then it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations
- Properties:
  - $\text{FP} \leq \text{MFP} \leq \text{MOP} \leq \text{Perfect-solution}$
  - FP, MFP are safe
  - $\text{in}(b) \leq \text{MOP}(b)$

# Partial Correctness of Algorithm

- If data flow framework is **monotone**, then if the algorithm converges,  $\text{IN}[b] \leq \text{MOP}[b]$
- Proof: Induction on path lengths
  - Define  $\text{IN}[\text{entry}] = \text{OUT}[\text{entry}]$  and transfer function of entry = Identity function
  - Base case: path of length 0
    - Proper initialization of  $\text{IN}[\text{entry}]$
  - If true for path of length  $k$ ,  $p_k = (n_1, \dots, n_k)$ , then true for path of length  $k+1$ :  $p_{k+1} = (n_1, \dots, n_{k+1})$ 
    - Assume:  $\text{IN}[n_k] \leq f_{nk-1}(f_{nk-2}(\dots f_{n1}(\text{IN}[\text{entry}])))$
    - $\text{IN}[n_{k+1}] = \text{OUT}[n_k] \wedge \dots$   
 $\leq \text{OUT}[n_k]$   
 $\leq f_{nk}(\text{IN}[n_k])$   
 $\leq f_{nk-1}(f_{nk-2}(\dots f_{n1}(\text{IN}[\text{entry}])))$

# Precision

- If data flow framework is **distributive**, then if the algorithm converges, **IN[b] = MOP[b]**



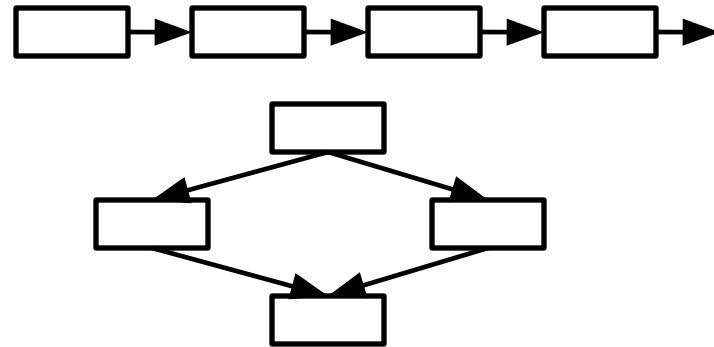
- Monotone but not distributive: behaves as if there are additional paths

# Additional Property to Guarantee Convergence

- Data flow framework (**monotone**) converges if there is a **finite descending chain**
- For each variable  $\text{IN}[b]$ ,  $\text{OUT}[b]$ , consider the sequence of values set to each variable **across iterations**:
  - if sequence for  $\text{in}[b]$  is monotonically decreasing
    - sequence for  $\text{out}[b]$  is monotonically decreasing
      - ( $\text{out}[b]$  initialized to  $T$ )
  - if sequence for  $\text{out}[b]$  is monotonically decreasing
    - sequence of  $\text{in}[b]$  is monotonically decreasing

# Speed of Convergence

- Speed of convergence depends on order of node visits



- Reverse “direction” for backward flow problems

# Reverse Postorder

- Step 1: depth-first post order

```
main() {
    count = 1;
Visit(root);
}
Visit(n) {
    for each successor s that has not been visited
        Visit(s);
PostOrder(n) = count;
count = count+1;
}
```

- Step 2: reverse order

```
For each node i
rPostOrder = NumNodes - PostOrder(i)
```

# Depth-First Iterative Algorithm (forward)

```
input: control flow graph CFG = (N, E, Entry, Exit)
/* Initialize */

    out[entry] = init_value
    For all nodes i
        out[i] = T
    Change = True
/* iterate */

    While Change {
        Change = False
        For each node i in rPostOrder {
            in[i] =  $\wedge$ (out[p]), for all predecessors p of i
            oldout = out[i]
            out[i] =  $f_i$ (in[i])
            if oldout  $\neq$  out[i]
                Change = True
        }
    }
```

# Speed of Convergence

- **If cycles do not add information**
  - information can flow in one pass down a series of nodes of increasing order number:
    - e.g., 1 -> 4 -> 5 -> 7 -> 2 -> 4 ...
  - passes determined by **number of back edges in the path**
    - essentially the nesting depth of the graph
  - **Number of iterations = number of back edges in any acyclic path + 2**
    - (2 are necessary even if there are no cycles)
- **What is the depth?**
  - corresponds to depth of intervals for “reducible” graphs
  - in real programs: average of 2.75

# A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values
  - meet operator
  - top, bottom
  - finite descending chain?
- **Transfer functions**
  - function of each basic block
  - monotone
  - distributive?
- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: rPostOrder
  - depth of the graph

# Conclusions

- Dataflow analysis examples
  - Reaching definitions
  - Live variables
- Dataflow formation definition
  - Meet operator
  - Transfer functions
  - Correctness, Precision, Convergence
  - Efficiency

# CSC D70: Compiler Optimization Dataflow Analysis

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# CSC D70: Compiler Optimization Dataflow Analysis-2

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Framework

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward: $\text{out}[b] = f_b(\text{in}[b])$ $\text{in}[b] = \bigwedge \text{out}[\text{pred}(b)]$	backward: $\text{in}[b] = f_b(\text{out}[b])$ $\text{out}[b] = \bigwedge \text{in}[\text{succ}(b)]$
Transfer function	$f_b(x) = \text{Gen}_b \cup (x - \text{Kill}_b)$	$f_b(x) = \text{Use}_b \cup (x - \text{Def}_b)$
Meet Operation ( $\wedge$ )	$\cup$	$\cup$
Boundary Condition	$\text{out}[\text{entry}] = \emptyset$	$\text{in}[\text{exit}] = \emptyset$
Initial interior points	$\text{out}[b] = \emptyset$	$\text{in}[b] = \emptyset$

Other examples (e.g., Available expressions), defined in ALSU 9.2.6

# **Foundations of Data Flow Analysis**

- 1. Meet operator**
- 2. Transfer functions**
- 3. Correctness, Precision, Convergence**
- 4. Efficiency**

- Reference: ALSU pp. 613-631
- Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
- Marlowe & Ryder, Properties of data flow frameworks: a unified model. Rutgers tech report, Apr. 1988

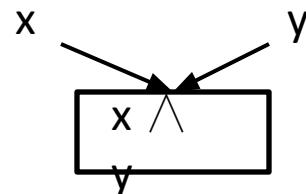
# A Unified Framework

- Data flow problems are defined by
  - Domain of values:  $V$
  - Meet operator ( $V \wedge V \rightarrow V$ ), initial value
  - A set of transfer functions ( $V \rightarrow V$ )
- Usefulness of unified framework
  - To answer questions such as correctness, precision, convergence, speed of convergence for a family of problems
    - If meet operators and transfer functions have properties X, then we know Y about the above.
  - Reuse code

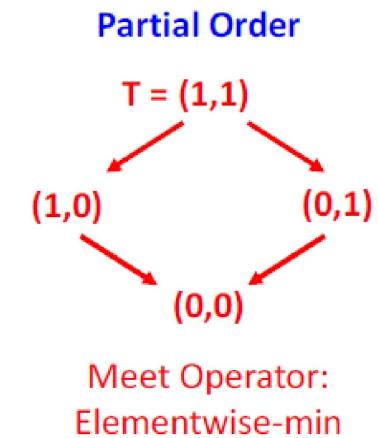
# Meet Operator

- Properties of the meet operator

- commutative:  $x \wedge y = y \wedge x$



- idempotent:  $x \wedge x = x$
- associative:  $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- there is a Top element  $T$  such that  $x \wedge T = x$

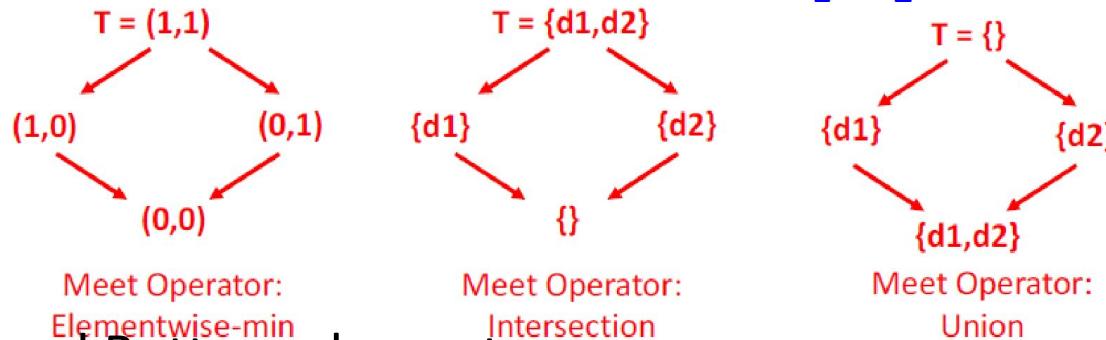


- Meet operator defines a partial ordering on values

- $x \leq y$  if and only if  $x \wedge y = x$  ( $y \rightarrow x$  in diagram)
  - Transitivity: if  $x \leq y$  and  $y \leq z$  then  $x \leq z$
  - Antisymmetry: if  $x \leq y$  and  $y \leq x$  then  $x = y$
  - Reflexitivity:  $x \leq x$

# Partial Order

- Example: let  $V = \{x \mid \text{such that } x \subseteq \{d_1, d_2\}\}$ ,  $\wedge = \cap$



- Top and Bottom elements

- Top  $T$  such that:  $x \wedge T = x$
- Bottom  $\perp$  such that:  $x \wedge \perp = \perp$

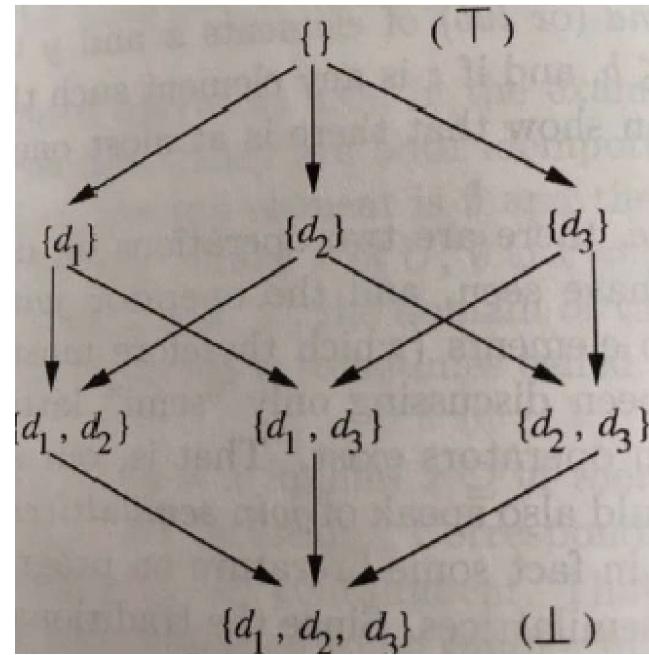
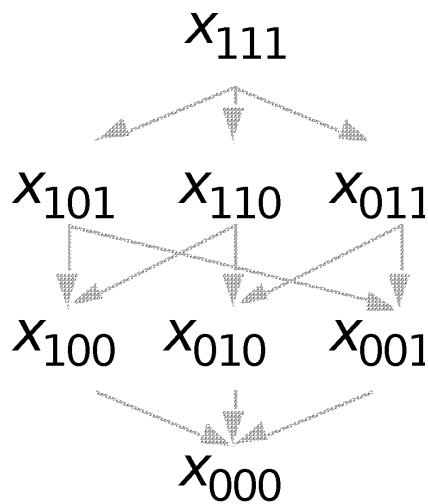
- Values and meet operator in a data flow problem define a semi-lattice:
  - there exists a  $T$ , but not necessarily a  $\perp$ .
- $x, y$  are ordered:  $x \leq y$  then  $x \wedge y = x$  ( $y \rightarrow x$  in diagram)
- what if  $x$  and  $y$  are not ordered?
  - $x \wedge y \leq x, x \wedge y \leq y$ , and if  $w \leq x, w \leq y$ , then  $w \leq x \wedge y$

# One vs. All Variables/Definitions

- Lattice for each variable: e.g. intersection



- Lattice for three variables:



# Descending Chain

- **Definition**
  - The **height** of a lattice is the largest number of **> relations** that will fit in a descending chain.
$$x_0 > x_1 > x_2 > \dots$$
- **Height of values in reaching definitions?**

Height n – number of definitions
- **Important property: finite descending chain**
- **Can an infinite lattice have a finite descending chain?**

yes
- **Example: Constant Propagation/Folding**
  - To determine if a variable is a constant
- **Data values**
  - undef, ... -1, 0, 1, 2, ..., not-a-constant

# Transfer Functions

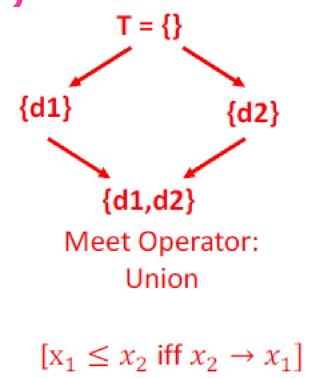
- **Basic Properties  $f: V \rightarrow V$** 
  - Has an identity function
    - There exists an  $f$  such that  $f(x) = x$ , for all  $x$ .
  - Closed under composition
    - if  $f_1, f_2 \in F$ , then  $f_1 \cdot f_2 \in F$

# Monotonicity

- A framework  $(F, V, \wedge)$  is **monotone** if and only if
  - $x \leq y$  implies  $f(x) \leq f(y)$
  - i.e. a “smaller or equal” input to the same function will always give a “smaller or equal” output
- Equivalently, a framework  $(F, V, \wedge)$  is **monotone** if and only if
  - $f(x \wedge y) \leq f(x) \wedge f(y)$
  - i.e. merge input, then apply  $f$  is **small than or equal to** apply the transfer function individually and then merge the result

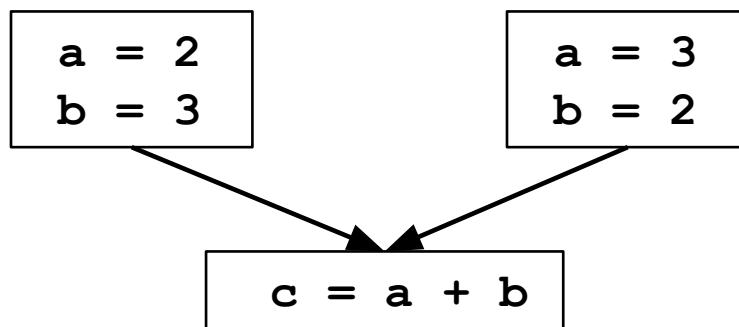
# Example

- **Reaching definitions:**  $f(x) = \text{Gen} \cup (x - \text{Kill})$ ,  $\wedge = \cup$ 
  - Definition 1:
    - $x_1 \leq x_2, \text{Gen} \cup (x_1 - \text{Kill}) \leq \text{Gen} \cup (x_2 - \text{Kill})$
  - Definition 2:
    - $(\text{Gen} \cup (x_1 - \text{Kill})) \cup (\text{Gen} \cup (x_2 - \text{Kill}))$   
 $= (\text{Gen} \cup ((x_1 \cup x_2) - \text{Kill}))$
- **Note: Monotone framework does not mean that  $f(x) \leq x$** 
  - e.g., reaching definition for two definitions in program
  - suppose:  $f_x : \text{Gen}_x = \{d_1, d_2\}; \text{Kill}_x = \{\}$
- **If  $\text{input(second iteration)} \leq \text{input(first iteration)}$** 
  - $\text{result(second iteration)} \leq \text{result(first iteration)}$



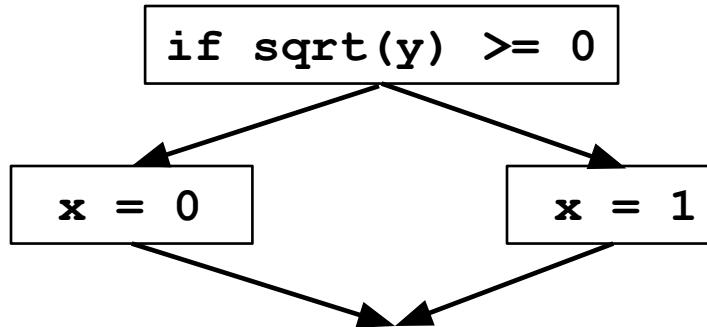
# Distributivity

- A framework  $(F, V, \wedge)$  is **distributive** if and only if
  - $f(x \wedge y) = f(x) \wedge f(y)$
  - i.e. merge input, then apply  $f$  is **equal to** apply the transfer function individually then merge result
- Example: Constant Propagation is NOT distributive



# Data Flow Analysis

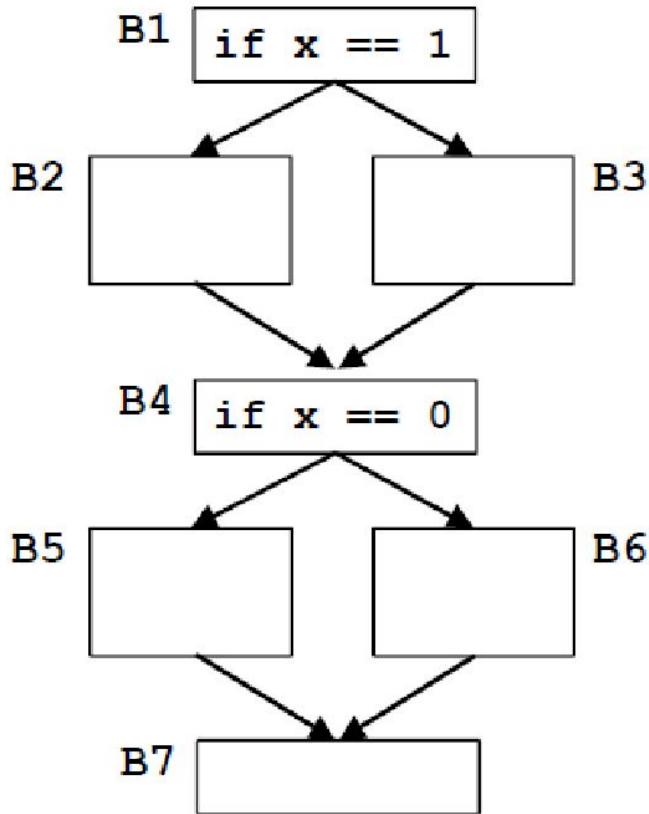
- **Definition**
  - Let  $f_1, \dots, f_m : \in F$ , where  $f_i$  is the transfer function for node  $i$ 
    - $f_p = f_{n_k} \cdot \dots \cdot f_{n_1}$ , where  $p$  is a path through nodes  $n_1, \dots, n_k$
    - $f_p$  = identify function, if  $p$  is an empty path
- **Ideal data flow answer:**
  - For each node  $n$ :  
 $\wedge f_{pi}(T)$ , for all possibly executed paths  $p_i$  reaching  $n$ .
- But determining all possibly executed paths is undecidable



# Meet-Over-Paths (MOP)

- Error in the conservative direction
- Meet-Over-Paths (MOP):
  - For each node  $n$ :
$$\text{MOP}(n) = \bigwedge f_{p_i}(\text{T}), \text{ for all paths } p_i \text{ reaching } n$$
  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - MOP = Perfect-Solution  $\wedge$  Solution-to-Unexecuted-Paths
  - MOP  $\leq$  Perfect-Solution
  - Potentially more constrained, solution is small
    - hence *conservative*
  - It is not **safe** to be  $>$  Perfect-Solution!
- **Desirable solution: as close to MOP as possible**

# MOP Example



Ideal: Considers only 2 paths

B1-B2-B4-B6-B7 (i.e., x=1)

B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths

B1-B2-B4-B5-B7

B1-B3-B4-B6-B7

Assume: B2 & B3 do not update x

# Solving Data Flow Equations

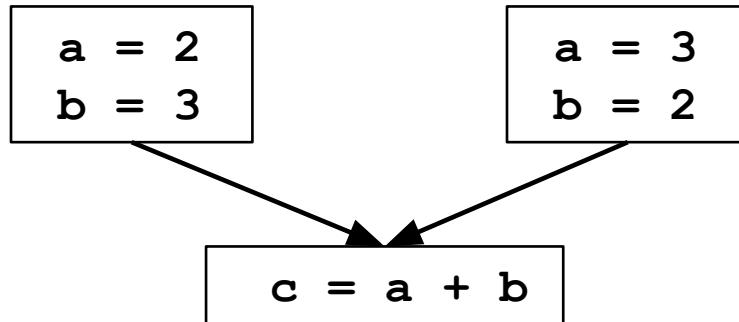
- Example: Reaching definitions
  - $\text{out}[\text{entry}] = \{\}$
  - Values = {subsets of definitions}
  - Meet operator:  $\cup$ 
    - $\text{in}[b] = \cup \text{out}[p]$ , for all predecessors  $p$  of  $b$
  - Transfer functions:  $\text{out}[b] = \text{gen}_b \cup (\text{in}[b] - \text{kill}_b)$
- Any solution satisfying equations = Fixed Point Solution (FP)
- Iterative algorithm
  - initializes  $\text{out}[b]$  to  $\{\}$
  - if converges, then it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations
- Properties:
  - $\text{FP} \leq \text{MFP} \leq \text{MOP} \leq \text{Perfect-solution}$
  - FP, MFP are safe
  - $\text{in}(b) \leq \text{MOP}(b)$

# Partial Correctness of Algorithm

- If data flow framework is **monotone**, then if the algorithm converges,  $\text{IN}[b] \leq \text{MOP}[b]$
- Proof: Induction on path lengths
  - Define  $\text{IN}[\text{entry}] = \text{OUT}[\text{entry}]$  and transfer function of entry = Identity function
  - Base case: path of length 0
    - Proper initialization of  $\text{IN}[\text{entry}]$
  - If true for path of length  $k$ ,  $p_k = (n_1, \dots, n_k)$ , then true for path of length  $k+1$ :  $p_{k+1} = (n_1, \dots, n_{k+1})$ 
    - Assume:  $\text{IN}[n_k] \leq f_{nk-1}(f_{nk-2}(\dots f_{n1}(\text{IN}[\text{entry}])))$
    - $\text{IN}[n_{k+1}] = \text{OUT}[n_k] \wedge \dots$   
 $\leq \text{OUT}[n_k]$   
 $\leq f_{nk}(\text{IN}[n_k])$   
 $\leq f_{nk-1}(f_{nk-2}(\dots f_{n1}(\text{IN}[\text{entry}])))$

# Precision

- If data flow framework is **distributive**, then if the algorithm converges, **IN[b] = MOP[b]**



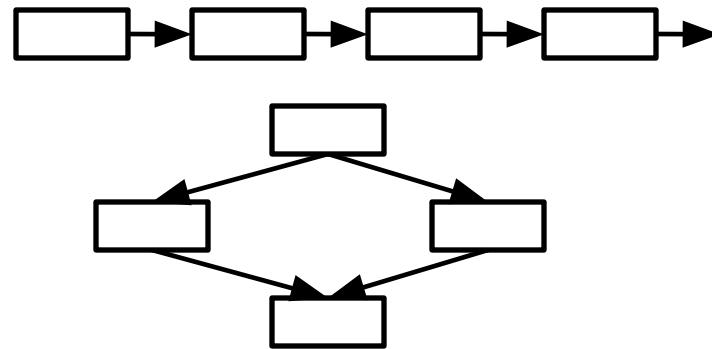
- Monotone but not distributive: behaves as if there are additional paths

# Additional Property to Guarantee Convergence

- Data flow framework (**monotone**) converges if there is a **finite descending chain**
- For each variable  $\text{IN}[b]$ ,  $\text{OUT}[b]$ , consider the sequence of values set to each variable **across iterations**:
  - if sequence for  $\text{in}[b]$  is monotonically decreasing
    - sequence for  $\text{out}[b]$  is monotonically decreasing
      - ( $\text{out}[b]$  initialized to  $T$ )
  - if sequence for  $\text{out}[b]$  is monotonically decreasing
    - sequence of  $\text{in}[b]$  is monotonically decreasing

# Speed of Convergence

- Speed of convergence depends on order of node visits



- Reverse “direction” for backward flow problems

# Reverse Postorder

- Step 1: depth-first post order

```
main() {  
    count = 1;  
Visit(root);  
}  
Visit(n) {  
    for each successor s that has not been visited  
        Visit(s);  
    PostOrder(n) = count;  
    count = count+1;  
}
```

- Step 2: reverse order

```
For each node i  
rPostOrder = NumNodes - PostOrder(i)
```

# Depth-First Iterative Algorithm (forward)

```
input: control flow graph CFG = (N, E, Entry, Exit)
/* Initialize */

    out[entry] = init_value
    For all nodes i
        out[i] = T
    Change = True
/* iterate */

    While Change {
        Change = False
        For each node i in rPostOrder {
            in[i] =  $\wedge$ (out[p]), for all predecessors p of i
            oldout = out[i]
            out[i] =  $f_i$ (in[i])
            if oldout  $\neq$  out[i]
                Change = True
        }
    }
```

# Speed of Convergence

- **If cycles do not add information**
  - information can flow in one pass down a series of nodes of increasing order number:
    - e.g., 1 -> 4 -> 5 -> 7 -> 2 -> 4 ...
  - passes determined by **number of back edges in the path**
    - essentially the nesting depth of the graph
  - **Number of iterations = number of back edges in any acyclic path + 2**
    - (2 are necessary even if there are no cycles)
- **What is the depth?**
  - corresponds to depth of intervals for “reducible” graphs
  - in real programs: average of 2.75

# A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values
  - meet operator
  - top, bottom
  - finite descending chain?
- **Transfer functions**
  - function of each basic block
  - monotone
  - distributive?
- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: rPostOrder
  - depth of the graph

# Conclusions

- Dataflow analysis examples
  - Reaching definitions
  - Live variables
- Dataflow formation definition
  - Meet operator
  - Transfer functions
  - Correctness, Precision, Convergence
  - Efficiency

# CSC D70: Compiler Optimization Loops

Prof. Gennady Pekhimenko

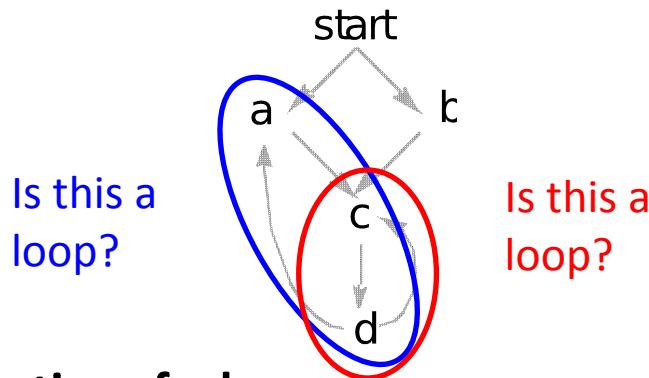
University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# What is a Loop?

- **Goals:**
  - Define a loop in graph-theoretic terms (control flow graph)
  - Not sensitive to input syntax
  - A uniform treatment for all loops: DO, while, goto's
- **Not every cycle is a “loop” from an optimization perspective**

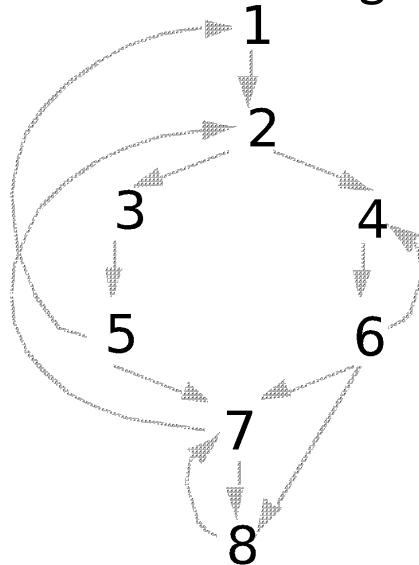


- **Intuitive properties of a loop**
  - single entry point
  - edges must form at least a cycle

# Formal Definitions

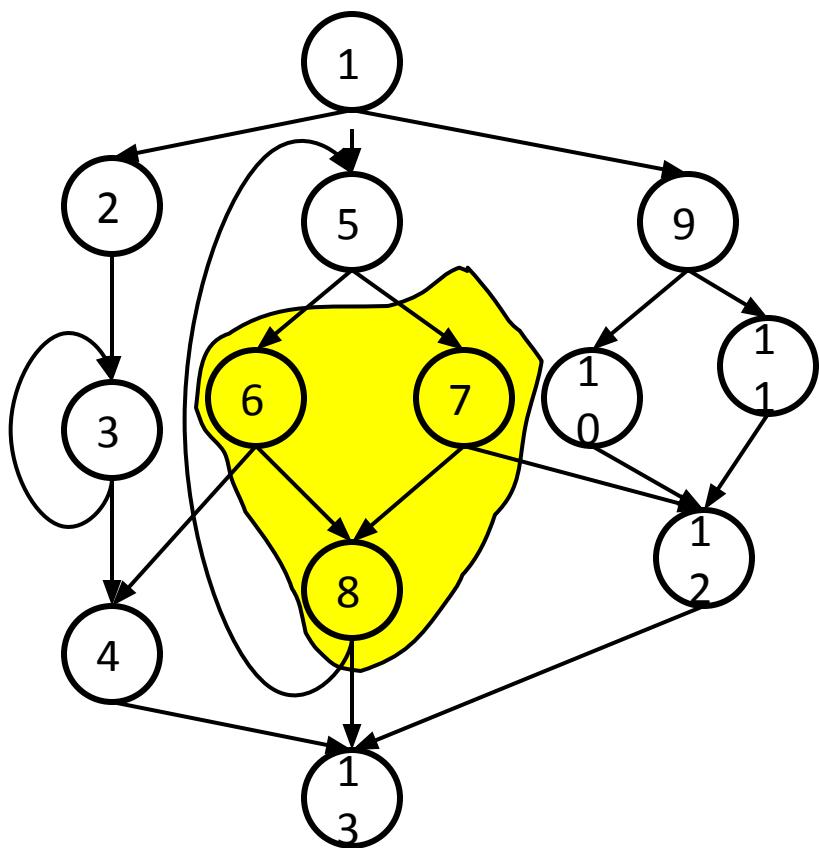
- **Dominators**

- Node  $d$  **dominates** node  $n$  in a graph ( $d \text{ dom } n$ ) if every path from the start node to  $n$  goes through  $d$

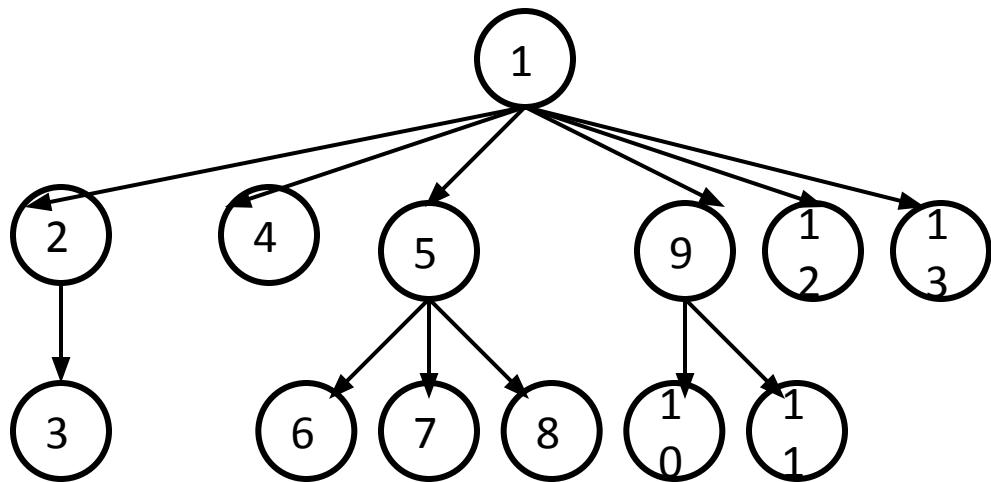


- Dominators can be organized as a **tree**
  - $a \rightarrow b$  in the **dominator tree** iff  $a$  immediately dominates  $b$

# Dominance



CFG



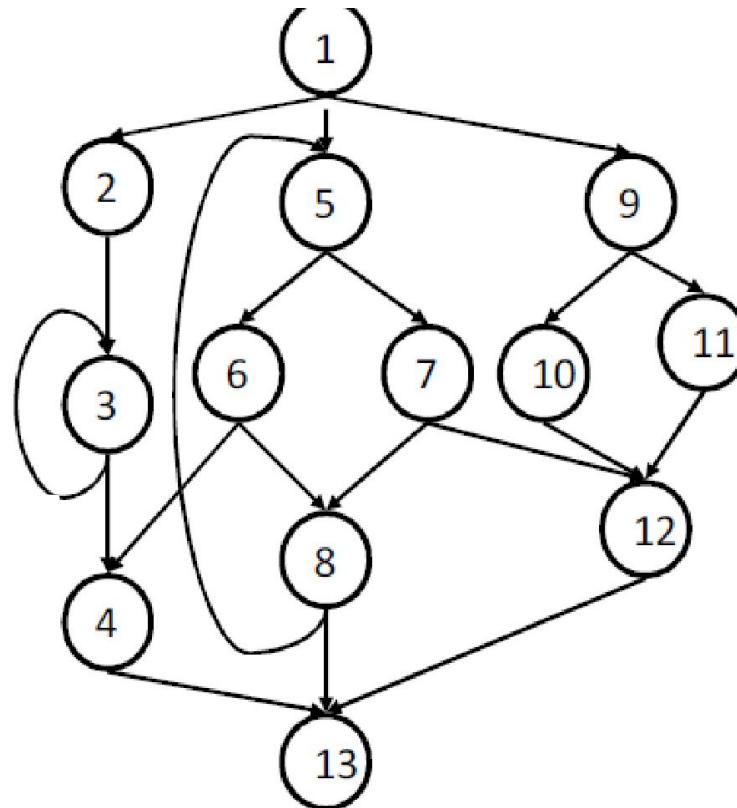
D-Tree

$x$  strictly dominates  $w$  ( $x$  sdom  $w$ ) iff  $x$  dom  $w$  AND  $x \neq w$

# Natural Loops

- **Definitions**
  - Single entry-point: *header*
    - a header **dominates** all nodes in the loop
  - A *back edge* is an arc whose **head** dominates its **tail** (tail  $\rightarrow$  head)
    - a back edge **must be a part of at least one loop**
  - The **natural loop of a back edge** is the **smallest set** of nodes that **includes the head and tail of the back edge**, and has **no predecessors outside the set**, except for the predecessors of the header.

# Natural Loops - Example



# Algorithm to Find Natural Loops

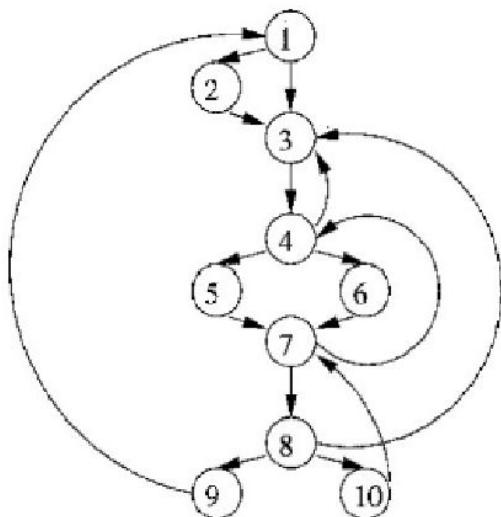
- Find the dominator relations in a flow graph
- Identify the back edges
- Find the natural loop associated with the back edge

# 1. Finding Dominators

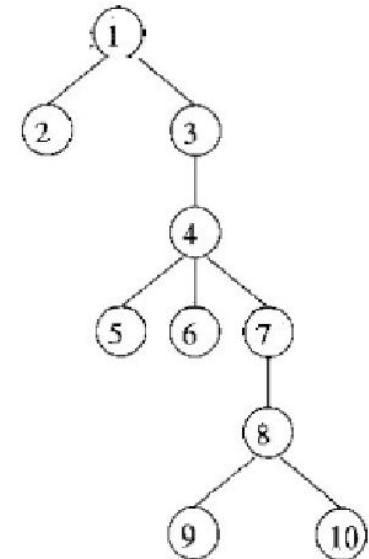
- **Definition**
  - Node  $d$  dominates node  $n$  in a graph ( $d \text{ dom } n$ )  
if every path from the start node to  $n$  goes through  $d$
- **Formulated as MOP problem:**
  - node  $d$  lies on all possible paths reaching node  $n \Rightarrow d \text{ dom } n$ 
    - Direction:
    - Values:
    - Meet operator:
    - Top:
    - Bottom:
    - Boundary condition: start/entry node =
    - Initialization for internal nodes
    - Finite descending chain?
    - Transfer function:
- **Speed:**
  - With reverse postorder, most flow graphs (reducible flow graphs) converge in 1 pass

# Example

$$\text{OUT}[b] = \{b\} \cup (\cap_{p=\text{pre } (b)} \text{OUT}[p])$$



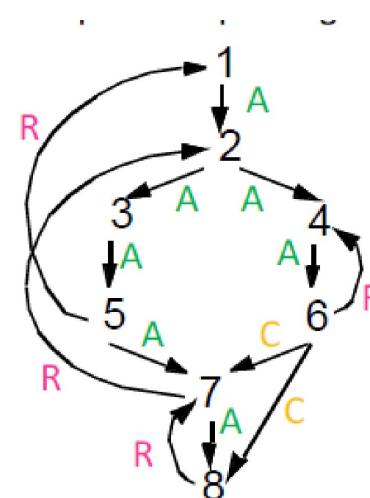
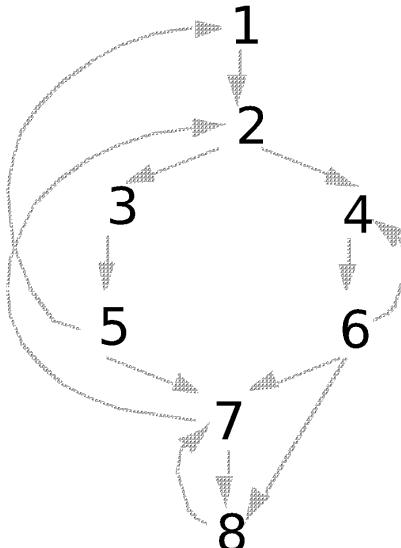
$\text{OUT}[1] = \{1\}$   
 $\text{OUT}[2] = \{1,2\}$   
 $\text{OUT}[3] = \{1,3\}$   
 $\text{OUT}[4] = \{1,3,4\}$   
 $\text{OUT}[5] = \{1,3,4,5\}$   
 $\text{OUT}[6] = \{1,3,4,6\}$   
 $\text{OUT}[7] = \{1,3,4,7\}$   
 $\text{OUT}[8] = \{1,3,4,7,8\}$   
 $\text{OUT}[9] = \{1,3,4,7,8,9\}$   
 $\text{OUT}[10] = \{1,3,4,7,8,10\}$   
(No change in second iteration)



# 2. Finding Back Edges

- Depth-first spanning tree

- Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree

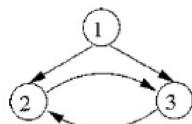


- Categorizing edges in graph

- Advancing (A) edges: from ancestor to proper descendant
- Cross (C) edges: from right to left
- Retreating (R) edges: from descendant to ancestor (not necessarily proper)

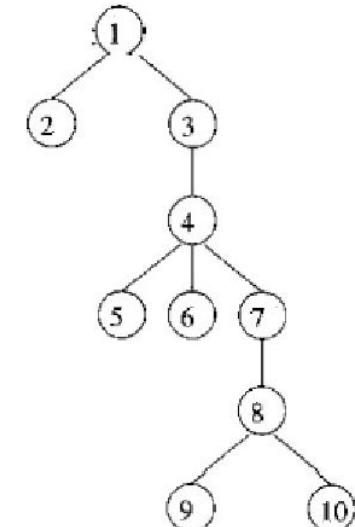
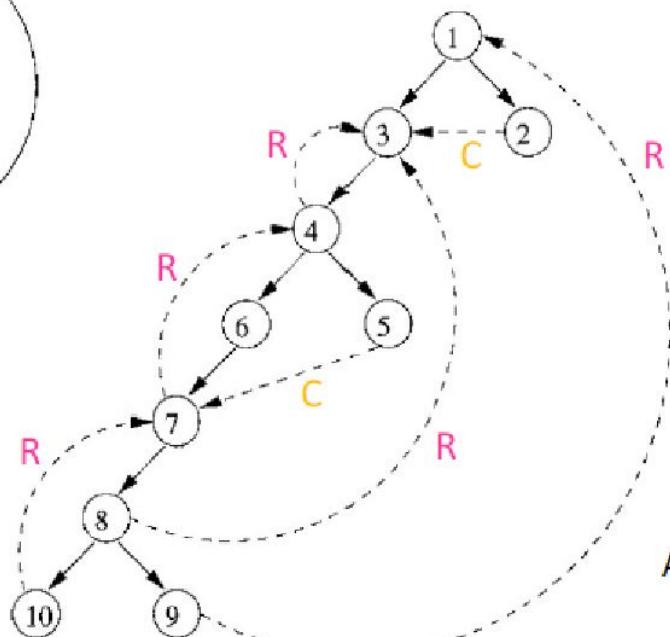
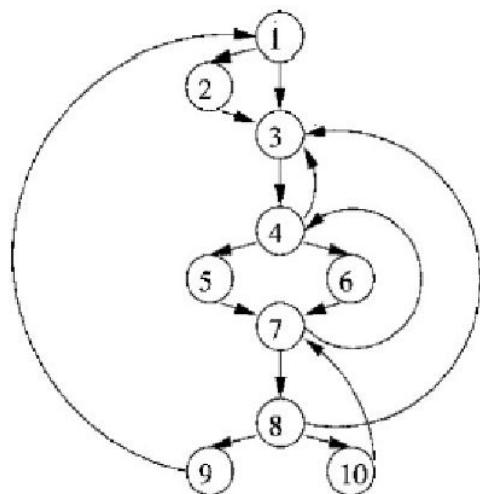
# Back Edges

- **Definition**
  - **Back edge**:  $t \rightarrow h$ ,  $h$  dominates  $t$
- **Relationships between graph edges and back edges**
- **Algorithm**
  - Perform a depth first search
  - For each retreating edge  $t \rightarrow h$ , check if  $h$  is in  $t$ 's dominator list
- **Most programs (all structured code, and most GOTO programs) have **reducible** flow graphs**
  - retreating edges = back edges



A **nonreducible** flow graph

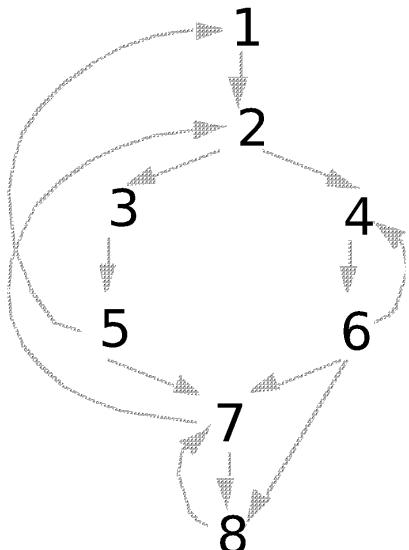
# Examples



All the retreating edges  
are back edges

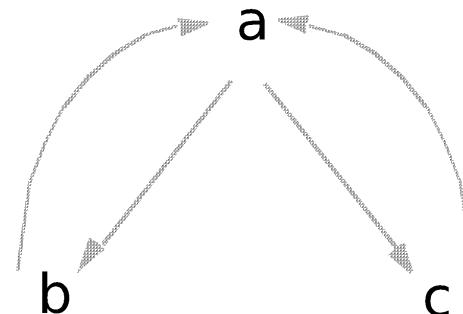
# 3. Constructing Natural Loops

- The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.
- **Algorithm**
  - delete  $h$  from the flow graph
  - find those nodes that can reach  $t$   
(those nodes plus  $h$  form the natural loop of  $t \rightarrow h$ )



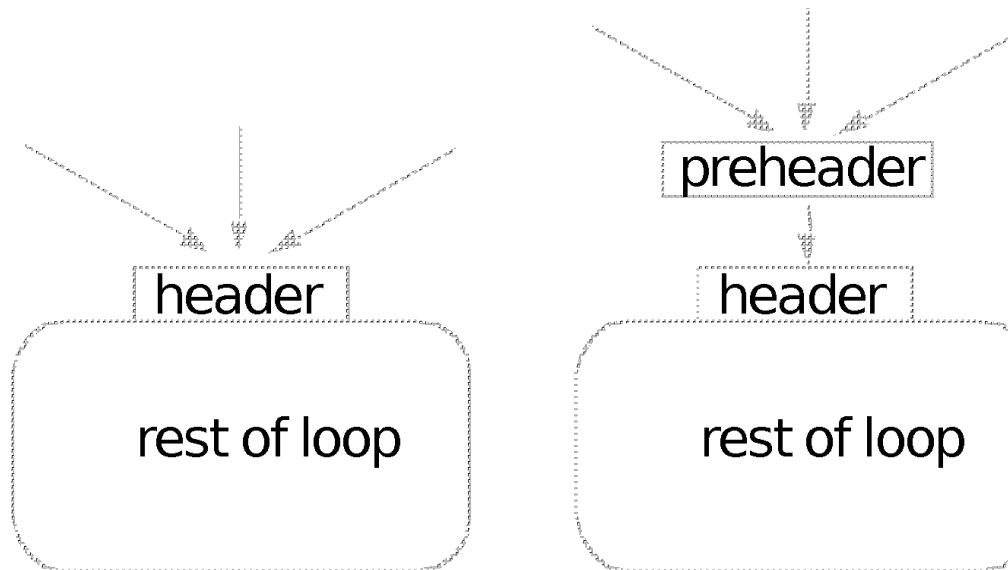
# Inner Loops

- **If two loops do not have the same header:**
  - they are either disjoint, or
  - one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.
- **If two loops share the same header:**
  - Hard to tell which is the inner loop
  - Combine as one



# Preheader

- Optimizations often require code to be executed once before the loop
- Create a preheader basic block for every loop



# Finding Loops: Summary

- Define loops in graph theoretic terms
- Definitions and algorithms for:
  - Dominators
  - Back edges
  - Natural loops

# CSC D70: Compiler Optimization Dataflow-2 and Loops

Prof. Gennady Pekhimenko

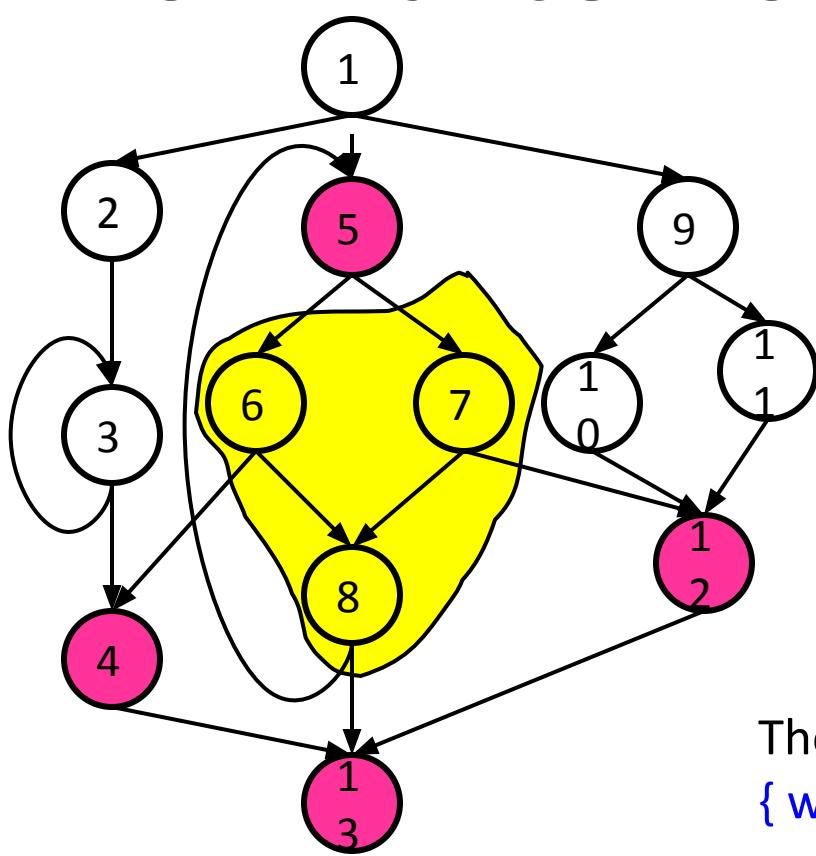
University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Backup Slides

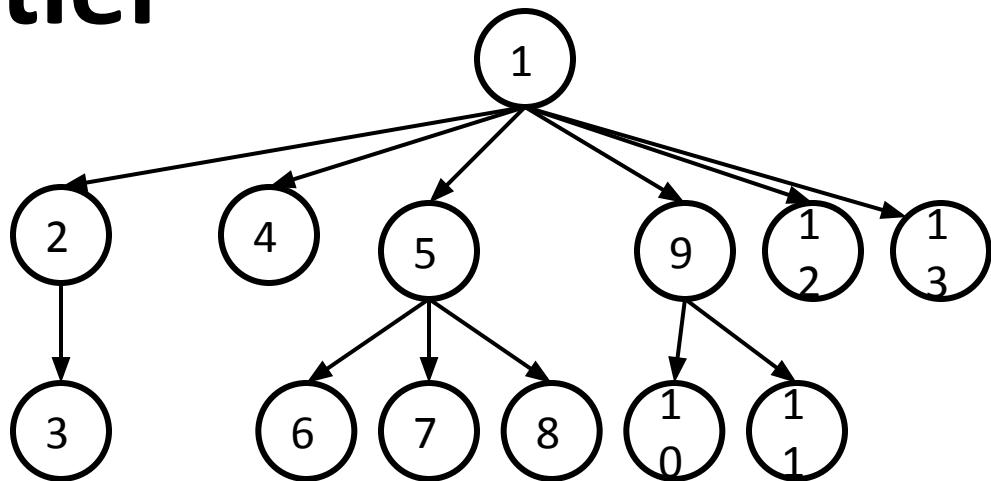
# Dominance Frontier



CFG

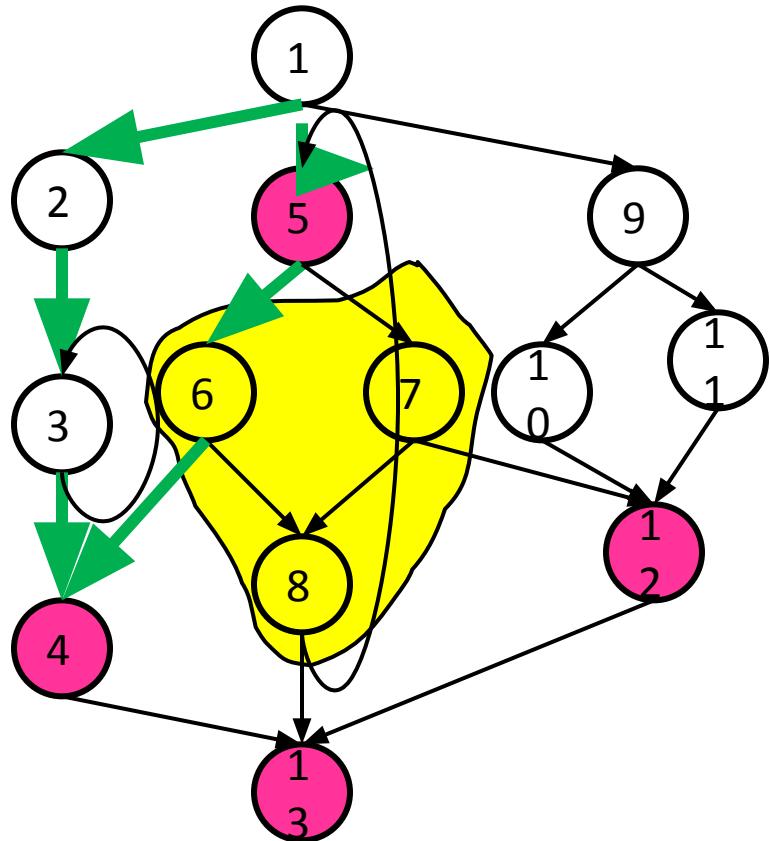
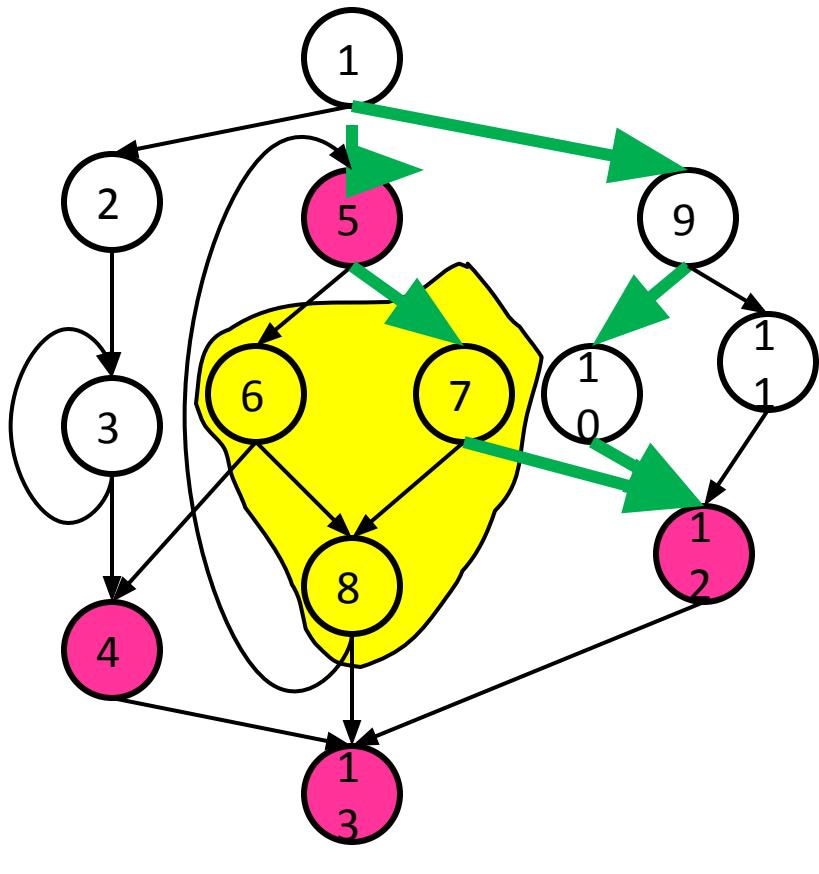
The Dominance Frontier of a node  $x$  =  
 $\{ w \mid x \text{ dom pred}(w) \text{ AND } !(x \text{ sdom } w)\}$

$x$  strictly dominates  $w$  ( $x$  sdom  $w$ ) iff  $x \text{ dom } w \text{ AND } x \neq w$



D-Tree

# Dominance Frontier and Path Convergence



# CSC D70: Compiler Optimization Static Single Assignment (SSA)

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# From Last Lecture

- What is a Loop?
- Dominator Tree
- Natural Loops
- Back Edges

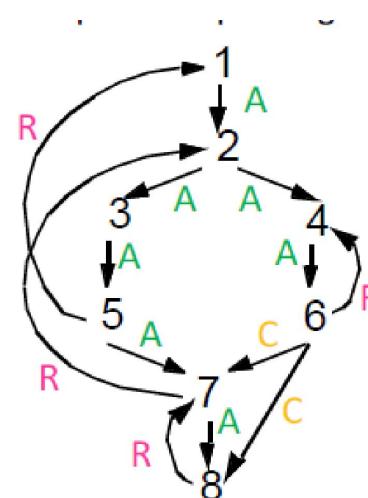
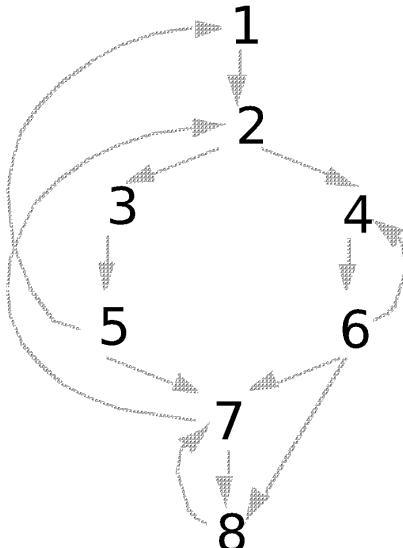
# Finding Loops: Summary

- Define loops in graph theoretic terms
- Definitions and algorithms for:
  - Dominators
  - Back edges
  - Natural loops

# Finding Back Edges

- Depth-first spanning tree

- Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree

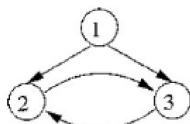


- Categorizing edges in graph

- Advancing (A) edges: from ancestor to proper descendant
- Cross (C) edges: from right to left
- Retreating (R) edges: from descendant to ancestor (not necessarily proper)

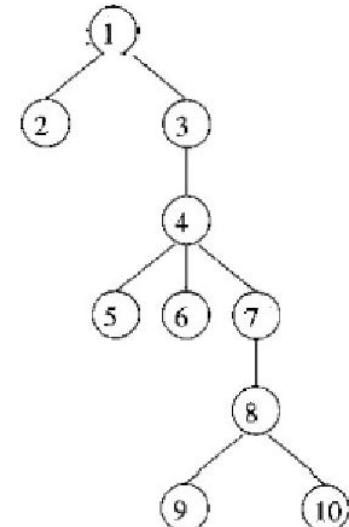
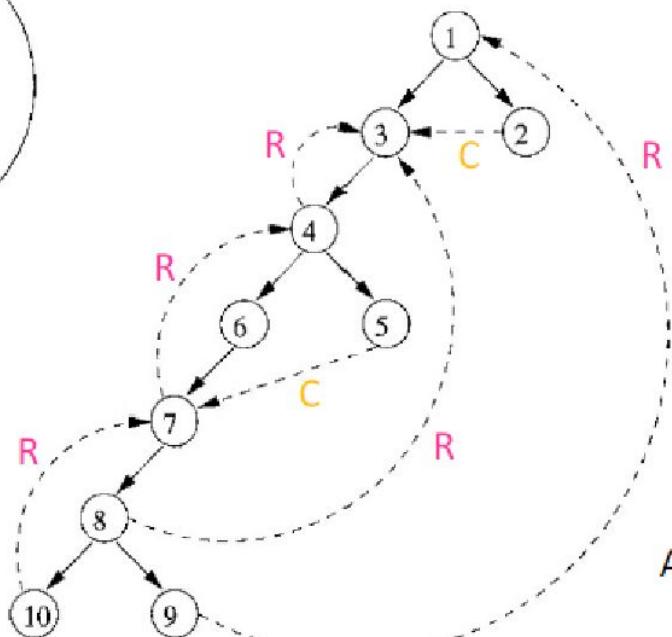
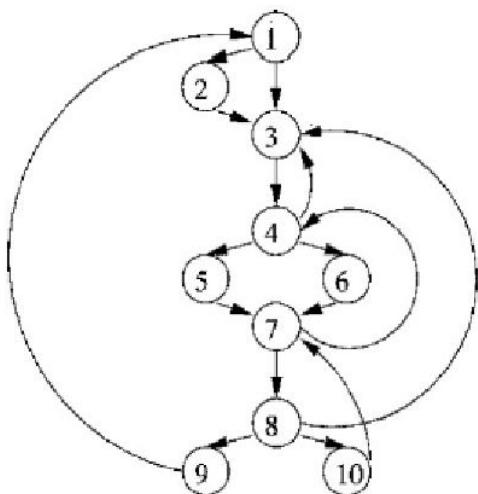
# Back Edges

- **Definition**
  - **Back edge**:  $t \rightarrow h$ ,  $h$  dominates  $t$
- **Relationships between graph edges and back edges**
- **Algorithm**
  - Perform a depth first search
  - For each retreating edge  $t \rightarrow h$ , check if  $h$  is in  $t$ 's dominator list
- **Most programs (all structured code, and most GOTO programs) have **reducible** flow graphs**
  - retreating edges = back edges



A **nonreducible** flow graph

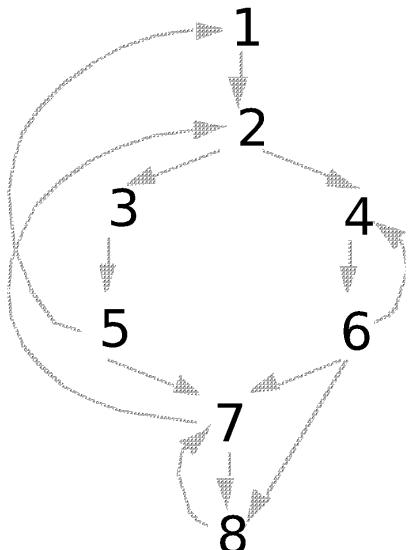
# Examples



All the retreating edges  
are back edges

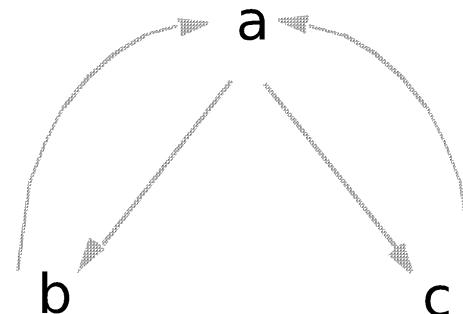
# Constructing Natural Loops

- The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.
- **Algorithm**
  - delete  $h$  from the flow graph
  - find those nodes that can reach  $t$   
(those nodes plus  $h$  form the natural loop of  $t \rightarrow h$ )



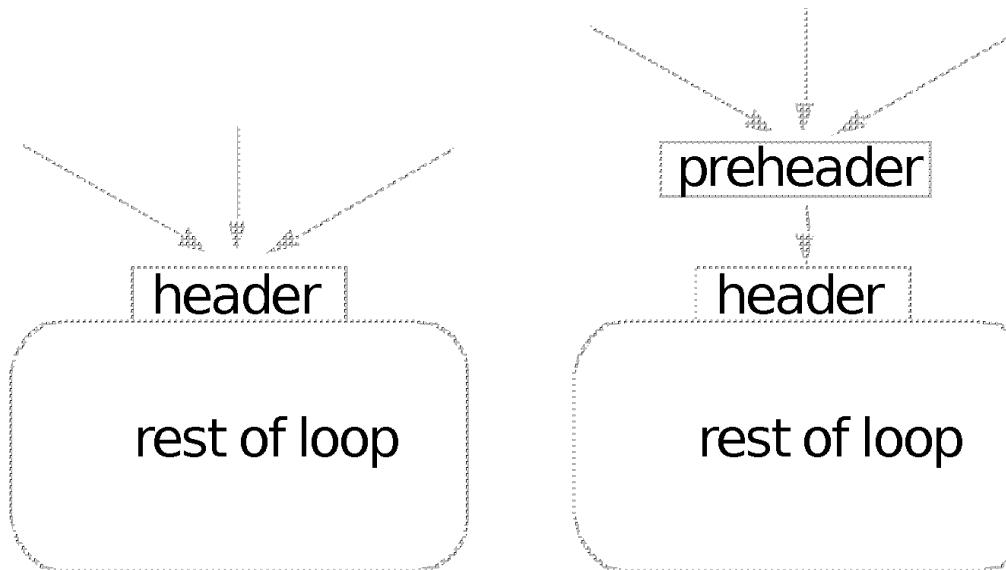
# Inner Loops

- **If two loops do not have the same header:**
  - they are either disjoint, or
  - one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.
- **If two loops share the same header:**
  - Hard to tell which is the inner loop
  - Combine as one



# Preheader

- Optimizations often require code to be executed once before the loop
- Create a preheader basic block for every loop



# CSC D70: Compiler Optimization Static Single Assignment (SSA)

Prof. Gennady Pekhimenko

University of Toronto

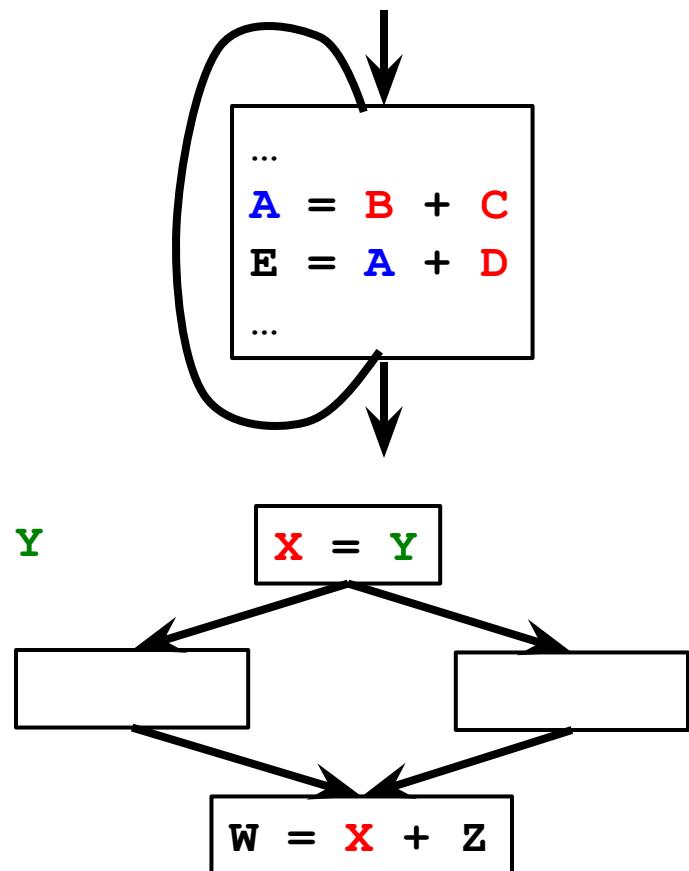
Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Where Is a Variable Defined or Used?

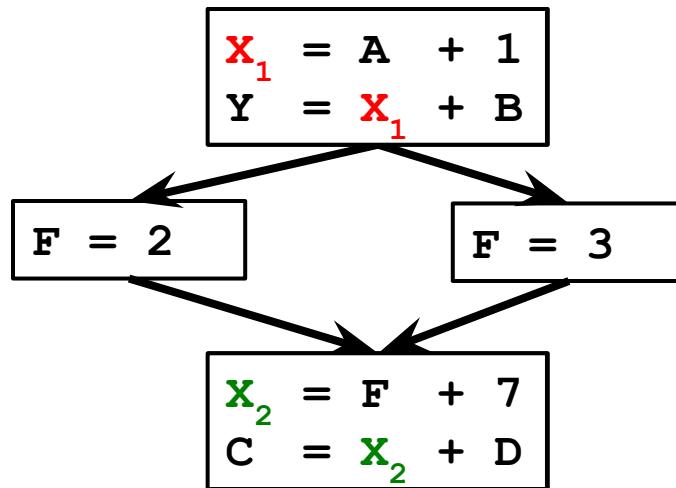
- Example: Loop-Invariant Code Motion
  - Are **B**, **C**, and **D** only defined outside the loop?
  - Other definitions of **A** inside the loop?
  - Uses of **A** inside the loop?

- Example: Copy Propagation
  - For a given use of **X**:
    - Are all reaching definitions of **X**:
      - copies from same variable: e.g., **X** = **Y**
      - Where **Y** is not redefined since that copy?
    - If so, substitute use of **X** with use of **Y**



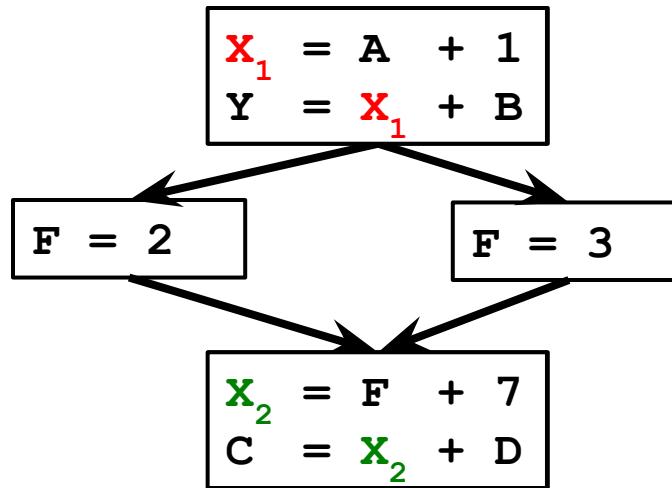
- It would be nice if we could *traverse directly* between related uses and def's
  - this would enable a form of *sparse* code analysis (skip over "don't care" cases)

# Appearances of Same Variable Name May Be Unrelated



- The values in reused storage locations may be provably independent
  - in which case the compiler can optimize them as separate values
- Compiler could use renaming to make these different versions more explicit

# Definition-Use and Use-Definition Chains



- Use-Definition (UD) Chains:
  - for a given definition of a variable X, what are all of its uses?
- Definition-Use (DU) Chains:
  - for a given use of a variable X, what are all of the reaching definitions of X?

# DU and UD Chains Can Be Expensive

```
foo(int i, int j) {  
    ...  
    switch (i) {  
        case 0: x=3; break;  
        case 1: x=1; break;  
        case 2: x=6; break;  
        case 3: x=7; break;  
        default: x = 11;  
    }  
    switch (j) {  
        case 0: y=x++; break;  
        case 1: y=x+4; break;  
        case 2: y=x-2; break;  
        case 3: y=x+1; break;  
        default: y=x+9;  
    }  
    ...
```

In general,

N defs

M uses

⇒ O(NM) space and time

One solution: limit each variable to ONE definition site

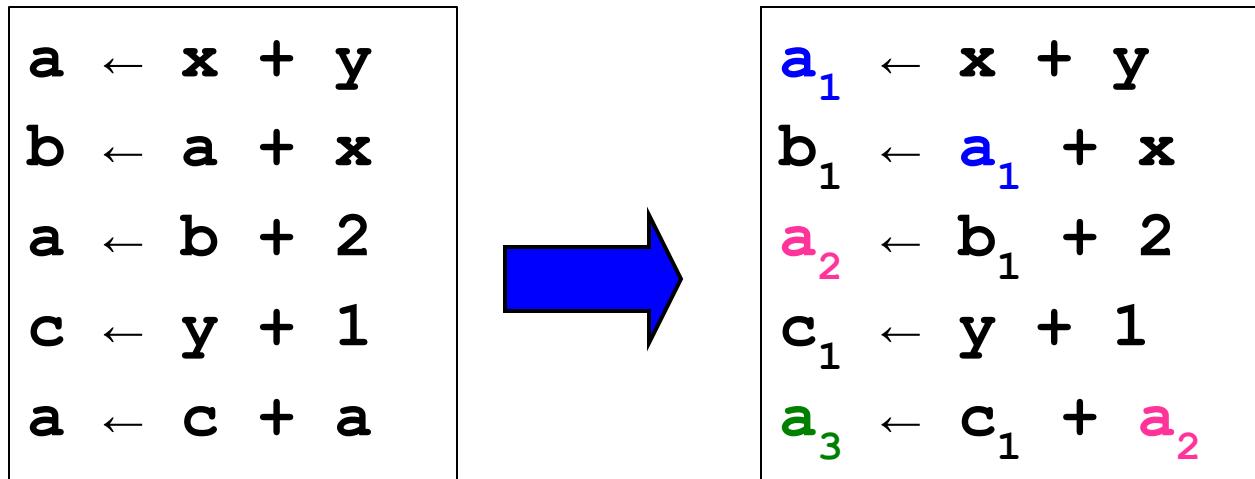
# DU and UD Chains Can Be Expensive (2)

```
foo(int i, int j) {  
    ...  
    switch (i) {  
        case 0: x=3; break;  
        case 1: x=1; break;  
        case 2: x=6;  
        case 3: x=7;  
        default: x = 11;  
    }  
x1 is one of the above x's  
    switch (j) {  
        case 0: y=x1+7;  
        case 1: y=x1+4;  
        case 2: y=x1-2;  
        case 3: y=x1+1;  
        default: y=x1+9;  
    }  
    ...
```

One solution: limit each variable to ONE definition site

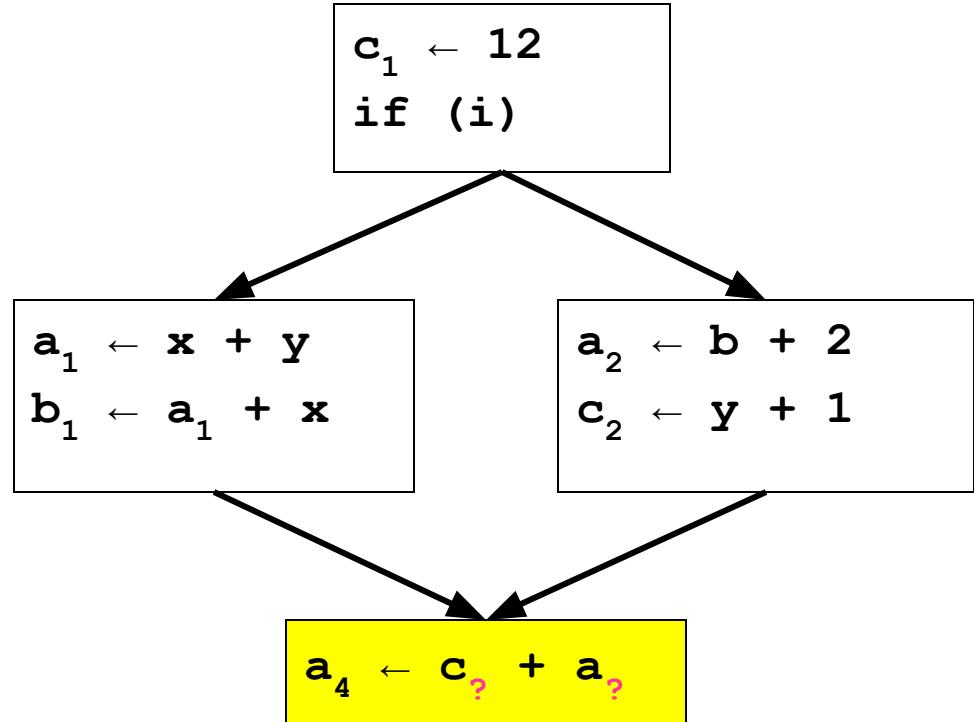
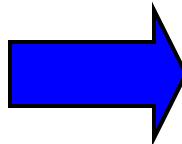
# Static Single Assignment (SSA)

- Static single assignment is an IR where **every variable** is assigned a value **at most once** in the program text
- Easy for a basic block (reminiscent of Value Numbering):
  - Visit each instruction in program order:
    - LHS: **assign** to a *fresh version* of the variable
    - RHS: **use the *most recent version*** of each variable



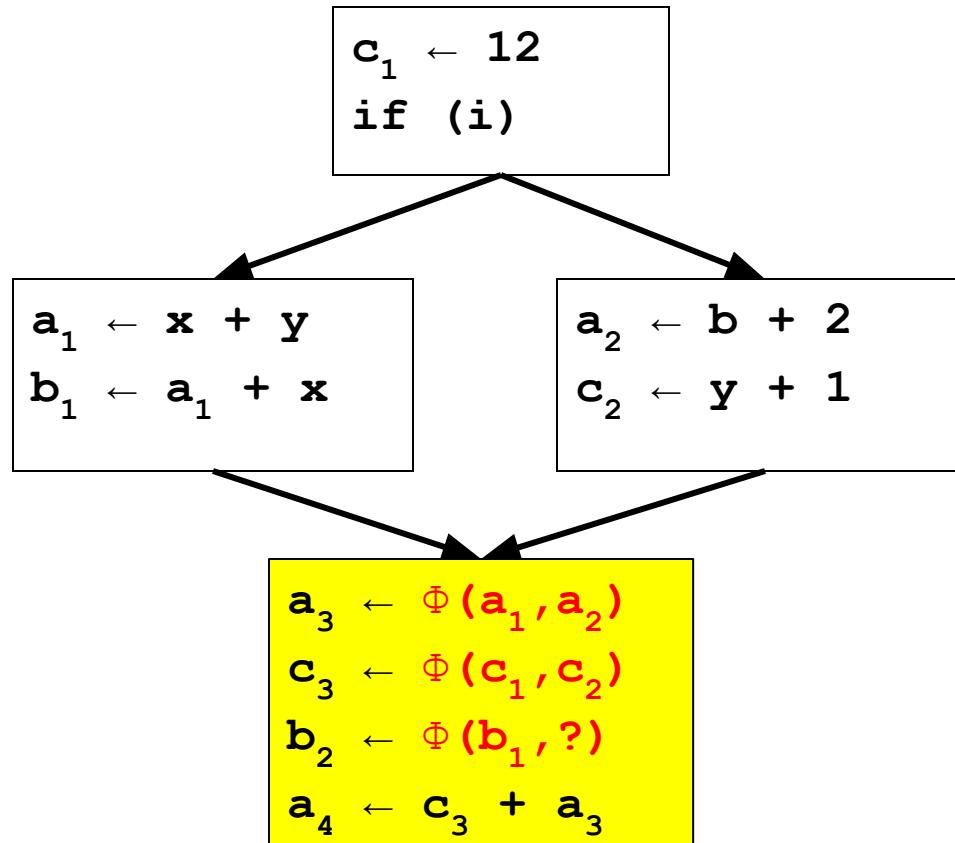
# What about Joins in the CFG?

```
c ← 12
if (i) {
    a ← x + y
    b ← a + x
} else {
    a ← b + 2
    c ← y + 1
}
a ← c + a
```



→ Use a notational fiction: a  $\Phi$  function

# Merging at Joins: the $\Phi$ function



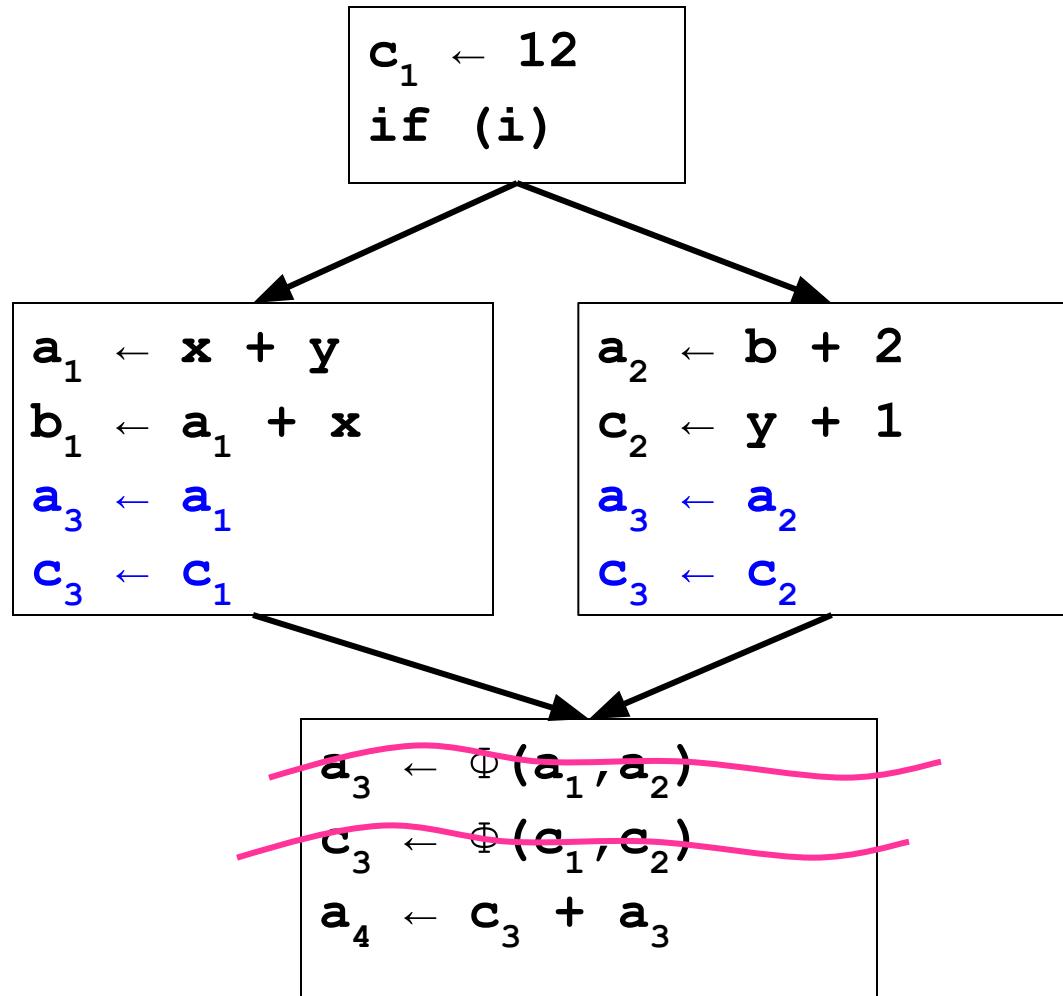
# The $\Phi$ function

- $\Phi$  merges multiple definitions along multiple control paths into a single definition.
- At a basic block with  $p$  predecessors, there are  $p$  arguments to the  $\Phi$  function.

$$x_{\text{new}} \leftarrow \Phi(x_1, x_1, x_1, \dots, x_p)$$

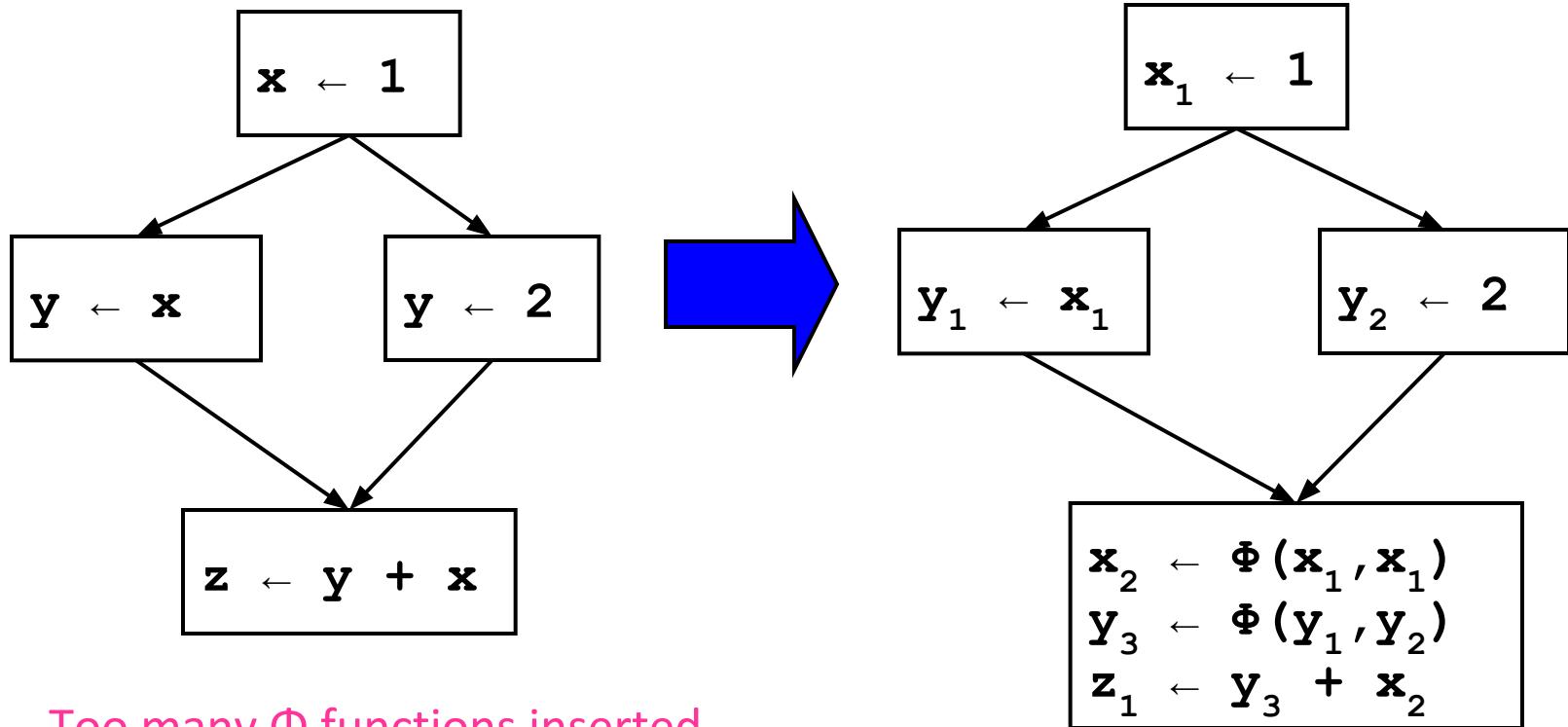
- How do we choose which  $x_i$  to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

# “Implementing” $\Phi$



# Trivial SSA

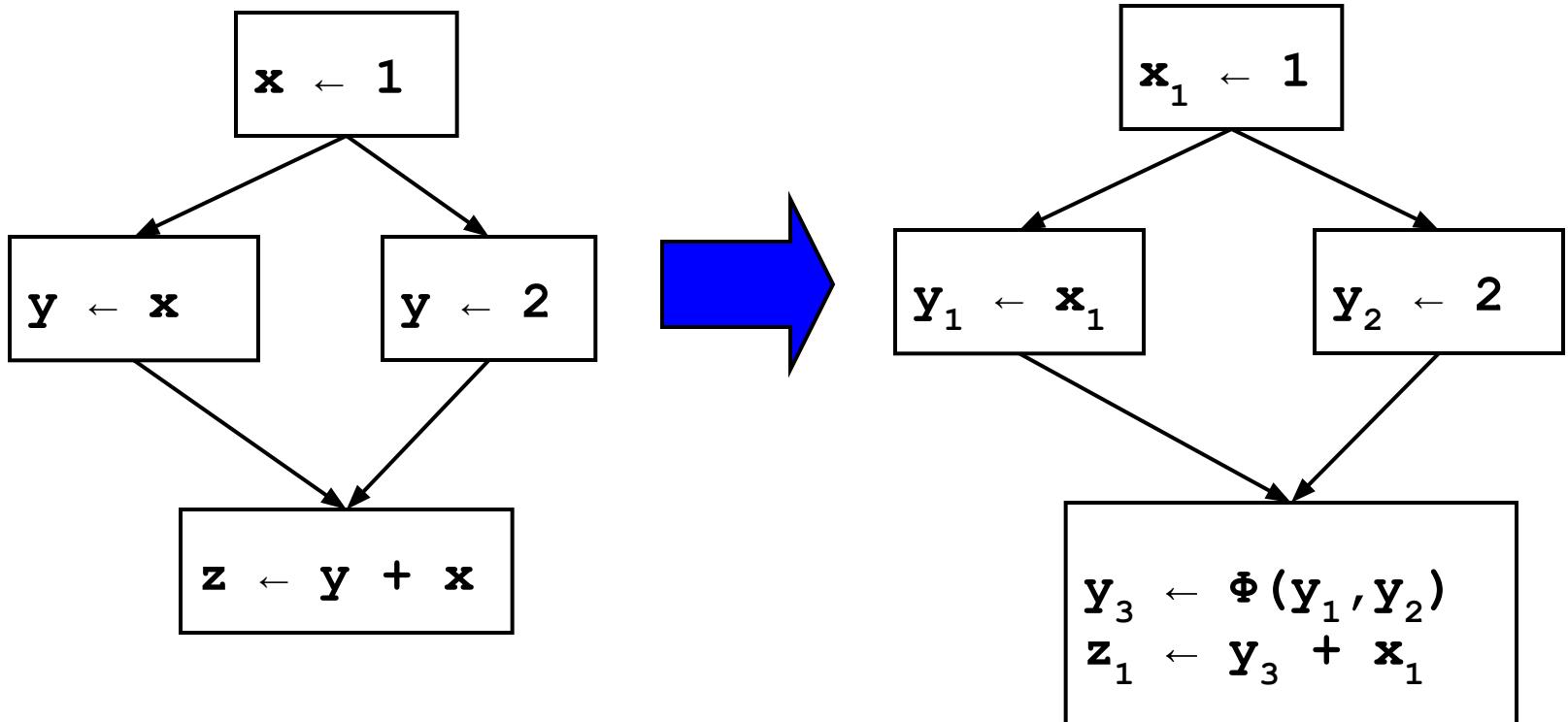
- Each assignment generates a fresh variable.
- At each join point insert  $\Phi$  functions for **all live variables**.



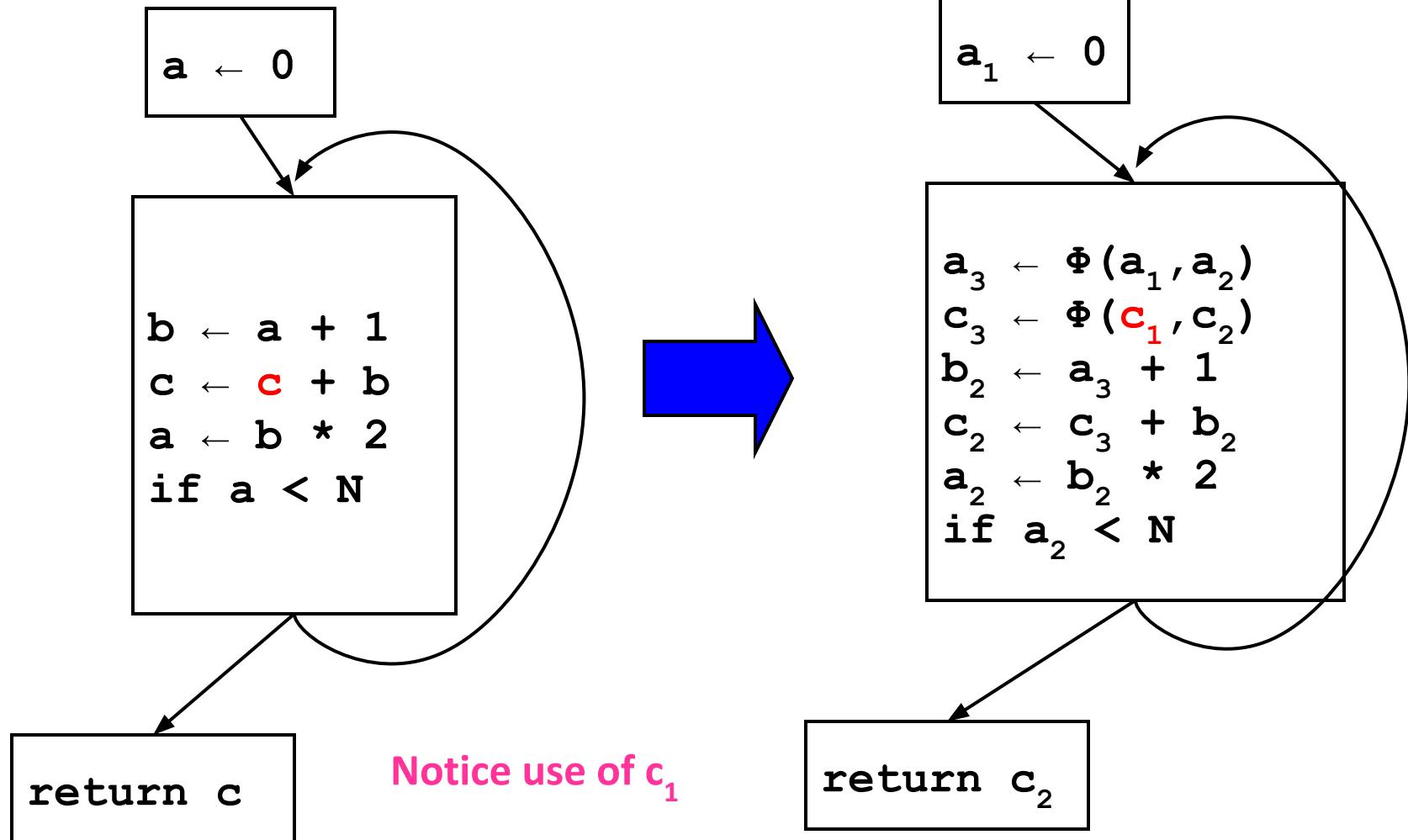
Too many  $\Phi$  functions inserted.

# Minimal SSA

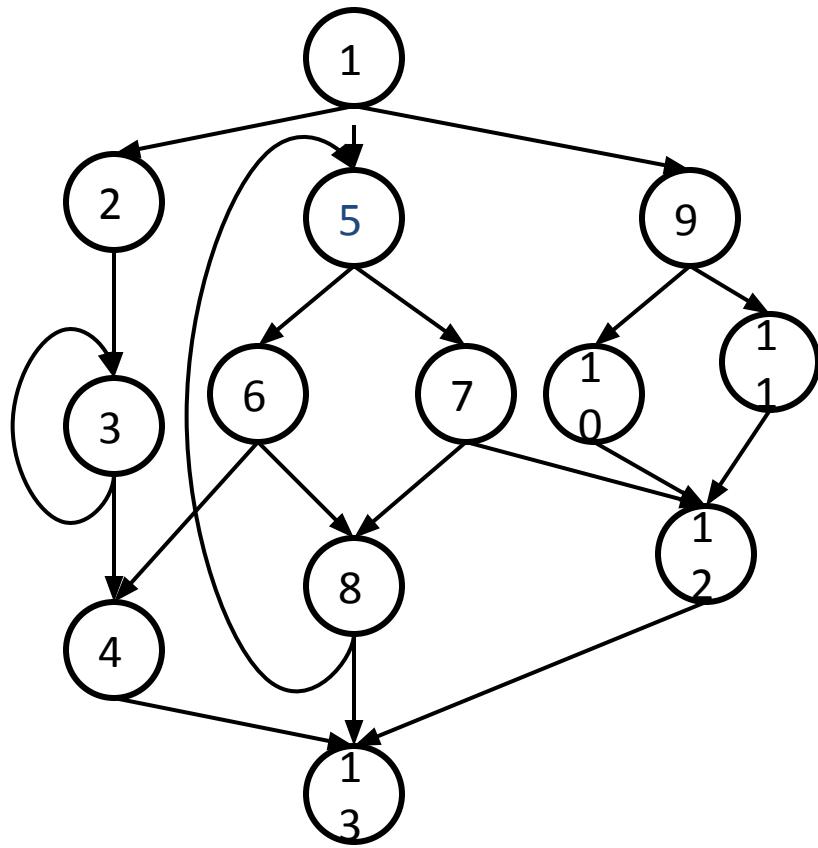
- Each assignment generates a fresh variable.
- At each join point insert  $\Phi$  functions for **all live variables** with **multiple outstanding defs**.



# Another Example



# When Do We Insert $\Phi$ ?

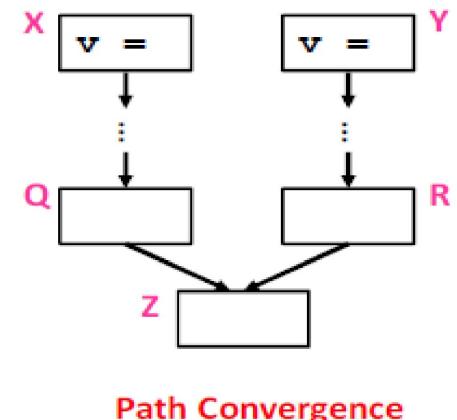


CFG

If there is a def of **a** in block 5,  
which nodes need a  $\Phi()$ ?

# When do we insert $\Phi$ ?

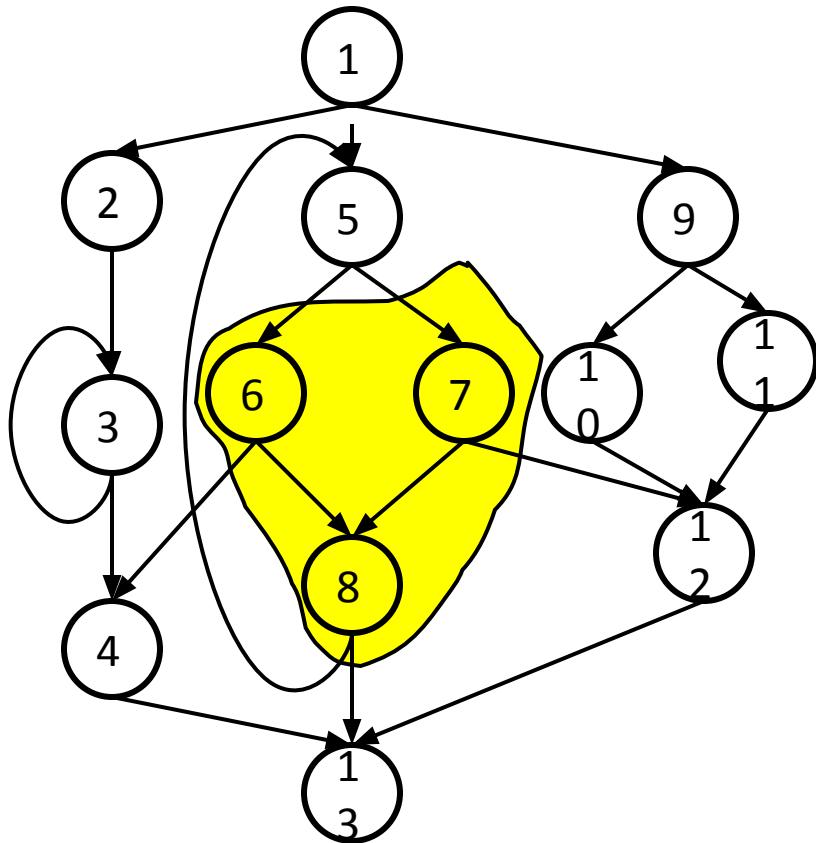
- We insert a  $\Phi$  function for variable  $A$  in block  $Z$  iff:
  - $A$  was defined more than once before
    - (i.e.,  $A$  defined in  $X$  and  $Y$  AND  $X \neq Y$ )
  - There exists a non-empty path from  $x$  to  $z$ ,  $P_{xz}$ ,
  - and a non-empty path from  $y$  to  $z$ ,  $P_{yz}$ , s.t.
    - $P_{xz} \cap P_{yz} = \{ z \}$   
*(Z is only common block along paths)*
    - $z \notin P_{xq}$  or  $z \notin P_{yr}$  where  $P_{xz} = P_{xq} \rightarrow z$  and  $P_{yz} = P_{yr} \rightarrow z$   
*(at least one path reaches Z for first time)*
- Entry block contains an implicit def of all vars
- Note:  $v = \Phi(\dots)$  is a def of  $v$



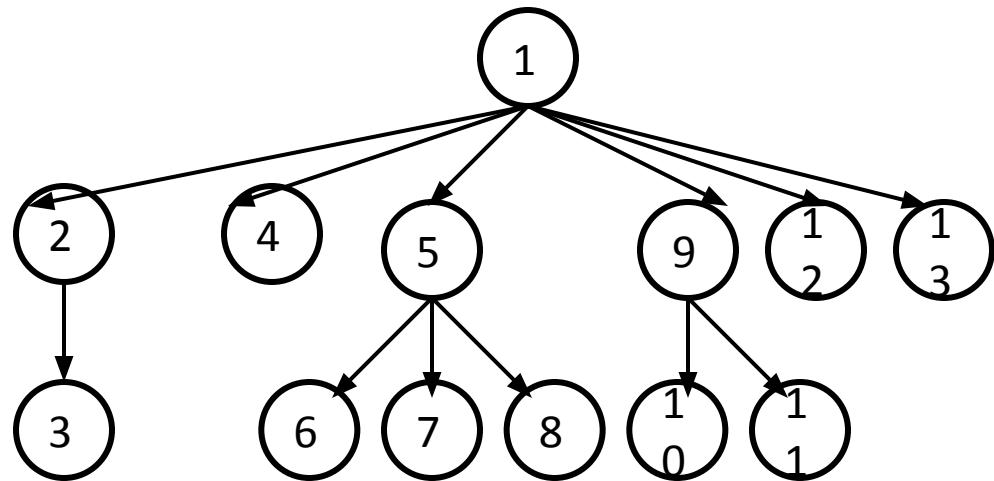
# Dominance Property of SSA

- In SSA, definitions dominate uses.
  - If  $x_i$  is used in  $x \leftarrow \Phi(\dots, x_i, \dots)$ , then  $\text{BB}(x_i)$  dominates  $i^{\text{th}}$  predecessor of  $\text{BB}(\text{PHI})$
  - If  $x$  is used in  $y \leftarrow \dots x \dots$ , then  $\text{BB}(x)$  dominates  $\text{BB}(y)$
- We can use this for an efficient algorithm to convert to SSA

# Dominance



CFG

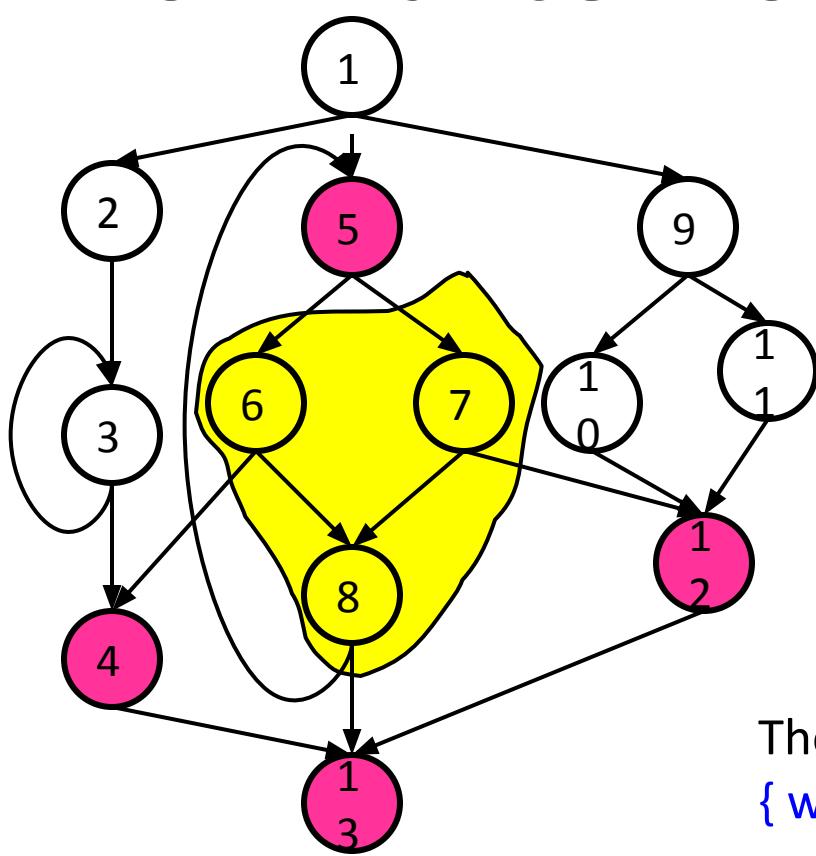


D-Tree

If there is a def of **a** in block 5,  
which nodes need a  $\Phi()$ ?

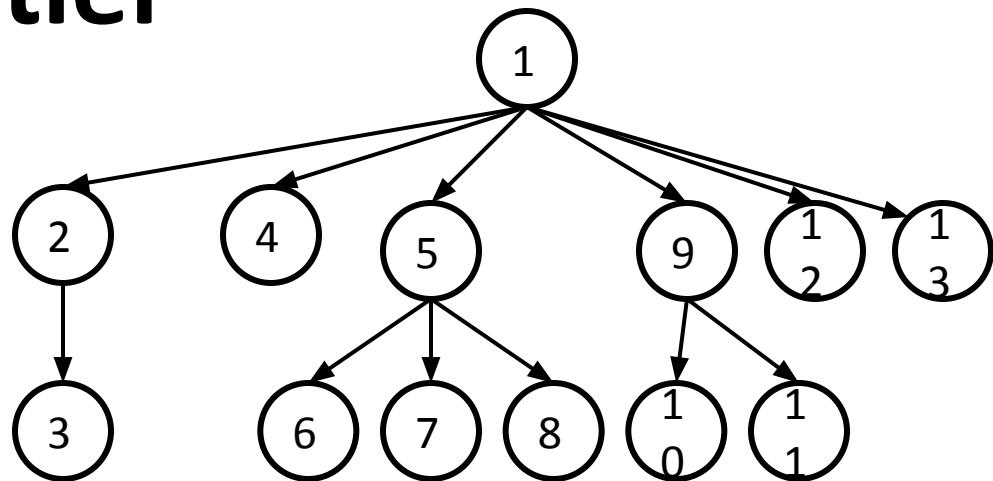
**x strictly dominates w** ( $x \text{ sdom } w$ ) iff  $x \text{ dom } w \text{ AND } x \neq w$

# Dominance Frontier



CFG

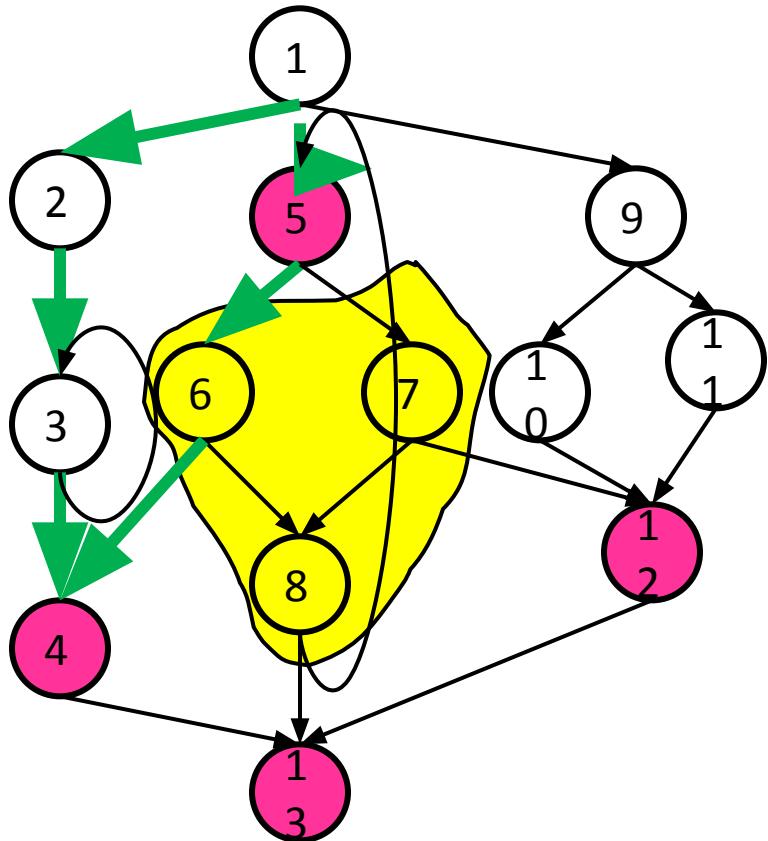
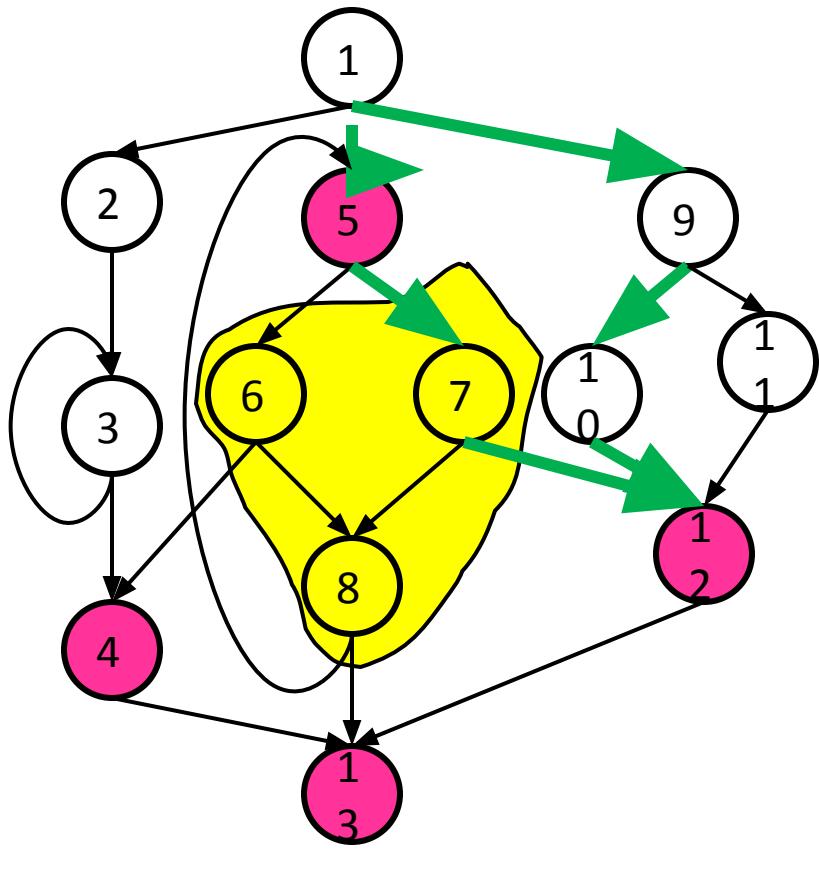
The Dominance Frontier of a node  $x$  =  
 $\{ w \mid x \text{ dom pred}(w) \text{ AND } !(x \text{ sdom } w)\}$



D-Tree

$x$  strictly dominates  $w$  ( $x$  sdom  $w$ ) iff  $x$  dom  $w$  AND  $x \neq w$

# Dominance Frontier and Path Convergence



If there is a def of **a** in block 5,  
nodes in DF(5) need a  $\Phi()$  for **a**

# Using Dominance Frontier to Compute SSA

- place all  $\Phi()$
- Rename all variables

# Using Dominance Frontier to Place $\Phi()$

- Gather all the defsites of every variable
- Then, for **every variable**
  - **foreach defsite**
    - **foreach node in DominanceFrontier(defsite)**
      - if we haven't put  $\Phi()$  in node, then **put one in**
      - if this node didn't define the variable before, then **add this node to the defsites**- This essentially computes the **Iterated Dominance Frontier** on the fly, inserting the minimal number of  $\Phi()$  necessary

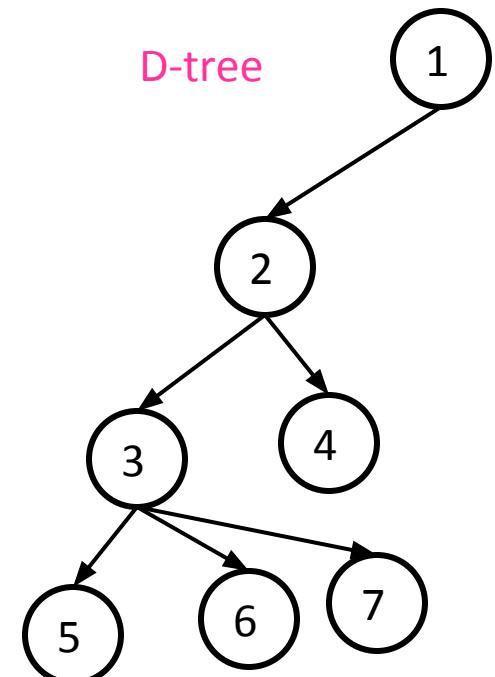
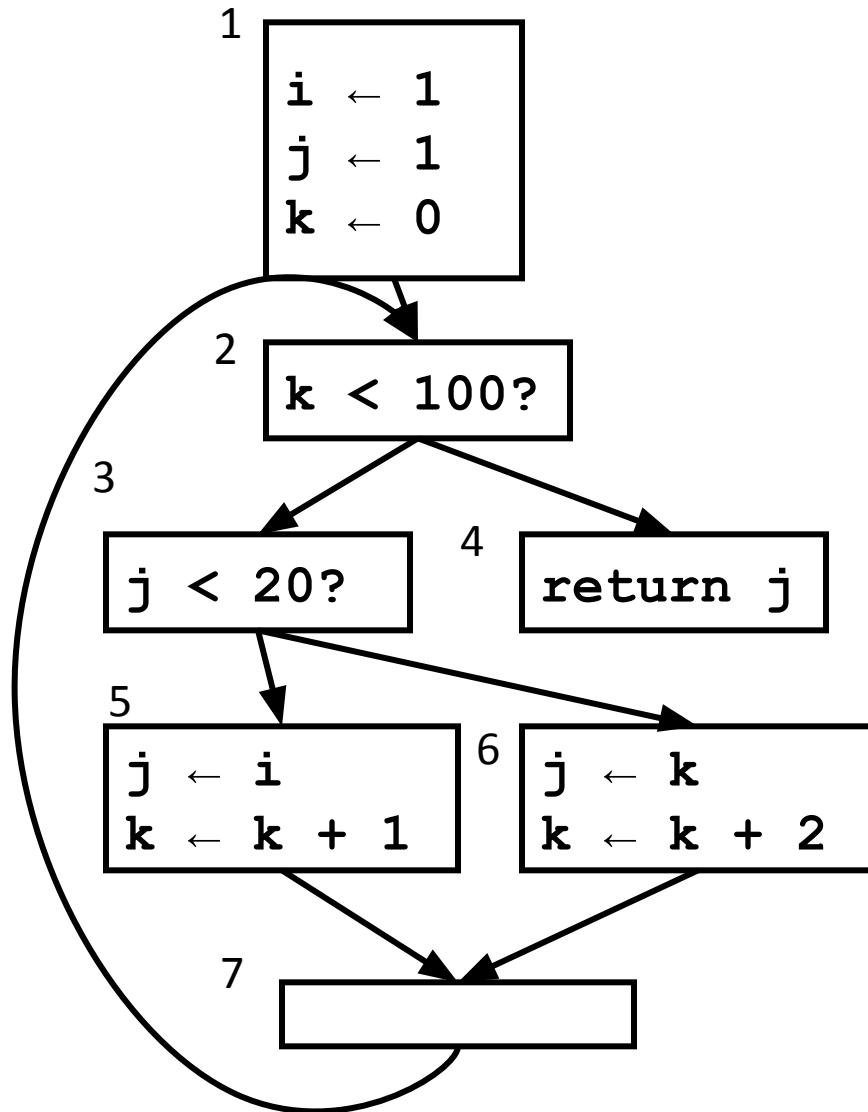
# Using Dominance Frontier to Place $\Phi()$

```
foreach node n {
    foreach variable v defined in n {
        orig[n] U= {v}
        defsites[v] U= {n}
    }
}
foreach variable v {
    W = defsites[v]
    while W not empty {
        n = remove node from W
        foreach y in DF[n]
            if ynotin PHI[v] {
                insert "v ← Φ(v,v,...)" at top of y
                PHI[v] = PHI[v] U {y}
                if vnotin orig[y]: W = W U {y}
            }
    }
}
```

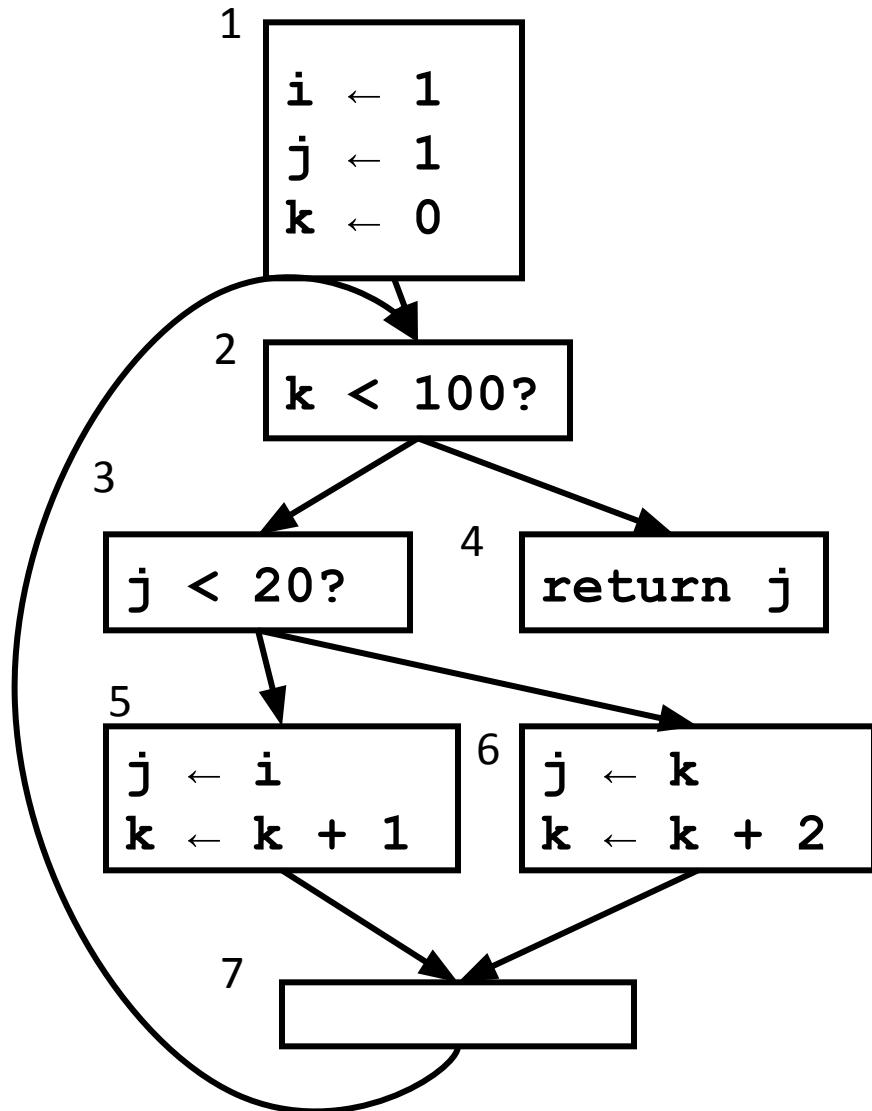
# Renaming Variables

- Algorithm:
  - Walk the D-tree, renaming variables as you go
  - Replace uses with more recent renamed def
- For straight-line code this is easy
- What if there are branches and joins?
  - use the closest def such that the def is above the use in the D-tree
- Easy implementation:
  - for each var: `rename (v)`
  - `rename(v):`    replace uses with top of stack  
              at def: push onto stack  
              call `rename(v)` on all children in D-tree  
              for each def in this block pop from stack

# Compute Dominance Tree

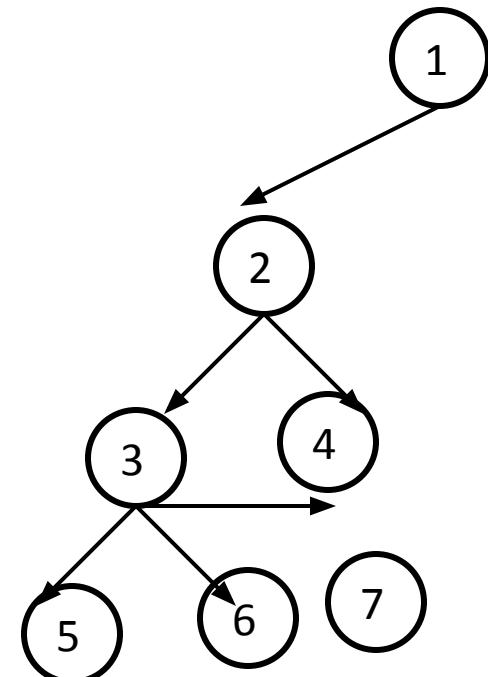


# Compute Dominance Frontiers

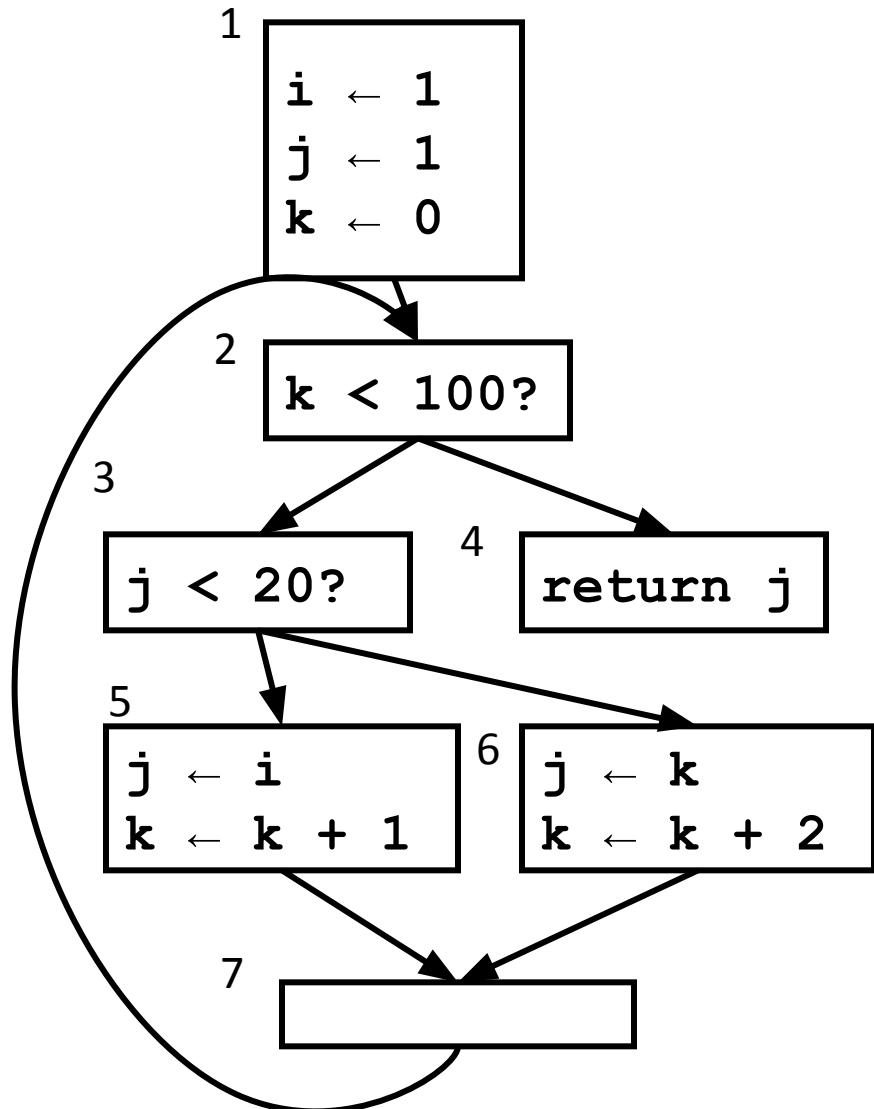


DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}



# Insert $\Phi()$



DFs

1 {}  
2 {2}  
3 {2}  
4 {}  
5 {7}  
6 {7}  
7 {2}

orig[n]

1 {i,j,k}  
2 {}  
3 {}  
4 {}  
5 {j,k}  
6 {j,k}  
7 {}

defsites[v]

i {1}  
j {1,5,6}  
k {1,5,6}

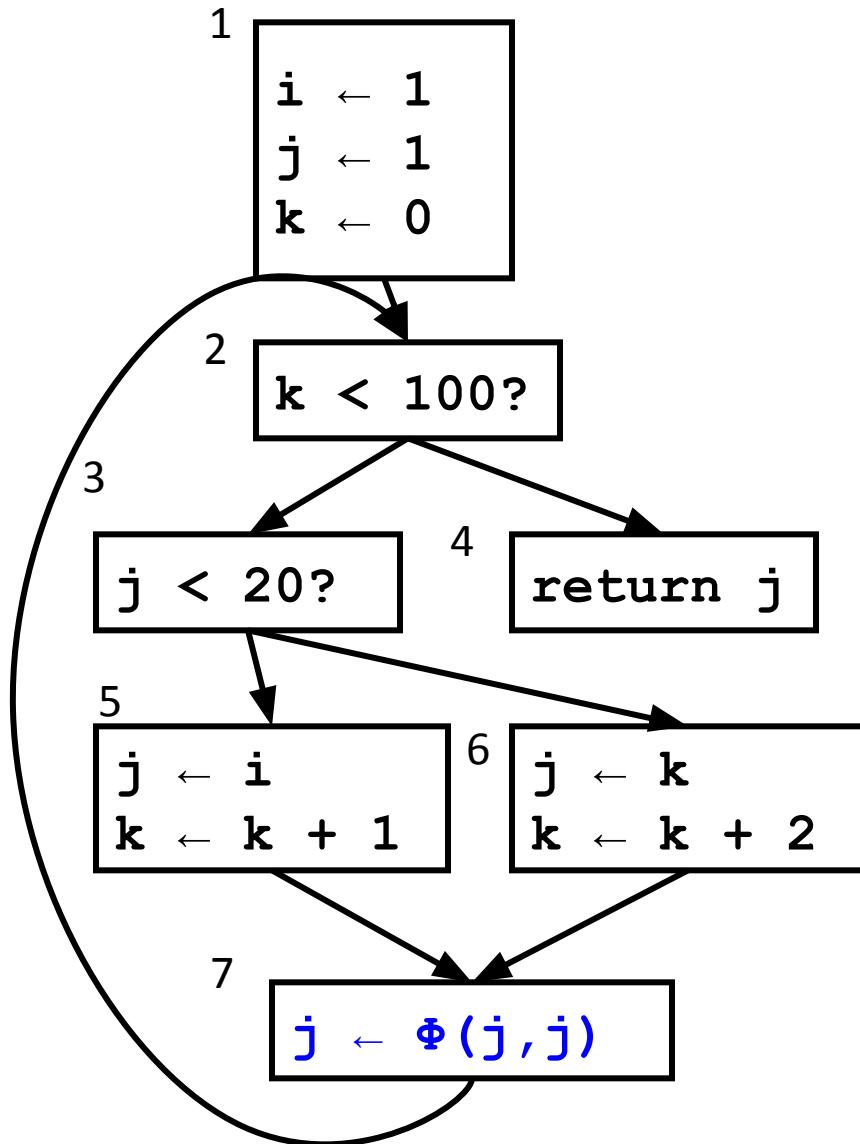
DFs

var i: W={1}

var j: W={1,5,6}

DF{1} DF{5}

# Insert $\Phi()$



DFs

1 {}  
2 {2}  
3 {2}  
4 {}  
5 {7}  
6 {7}  
7 {2}

orig[n]

1 {i,j,k}  
2 {}  
3 {}  
4 {}  
5 {j,k}  
6 {j,k}  
7 {}

defsites[v]

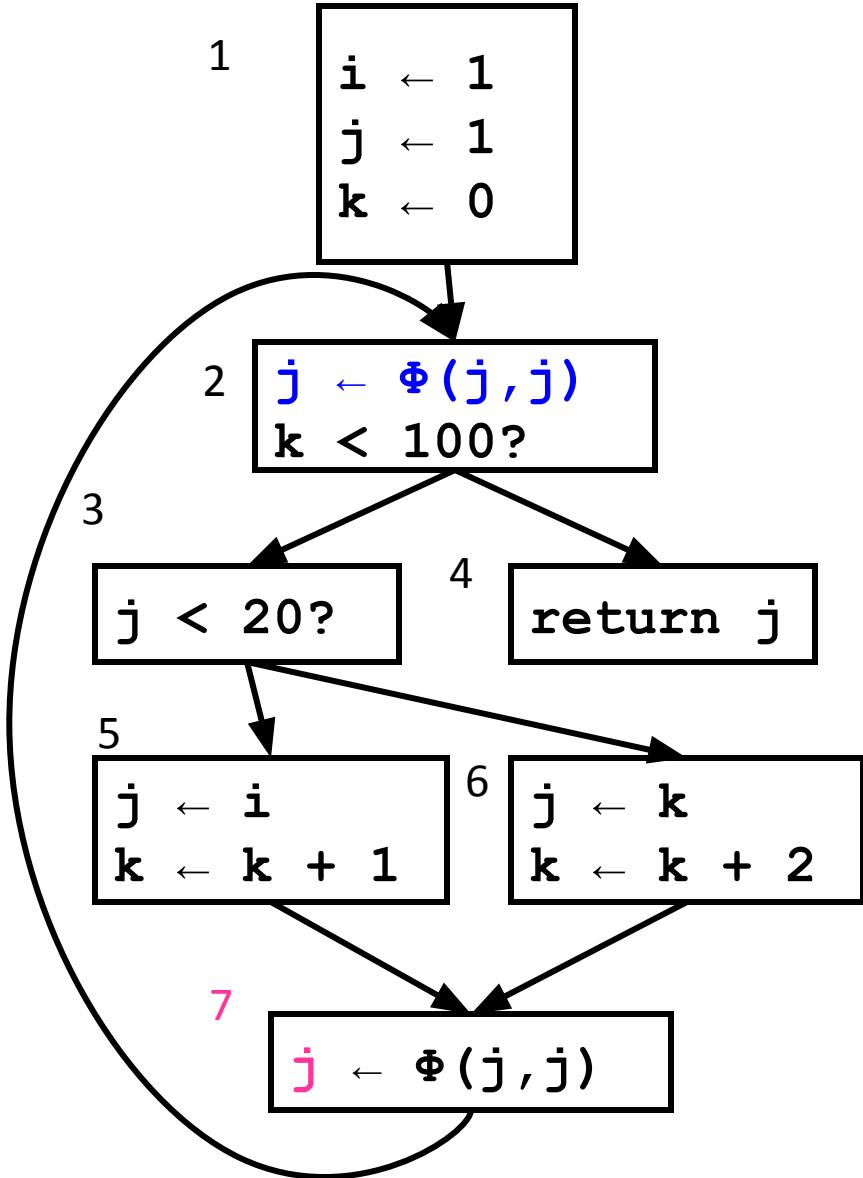
i {1}  
j {1,5,6}  
k {1,5,6}

DFs

var j: W={1,5,6}

DF{1}

DF{5}



DFs

1 {}  
2 {2}  
3 {2}  
4 {}  
5 {7}  
6 {7}  
7 {2}

orig[n]

1 {i,j,k}  
2 {}  
3 {}  
4 {}  
5 {j,k}  
6 {j,k}  
7 {}

defsites[v]

i {1}  
j {1,5,6,7}  
k {1,5,6}

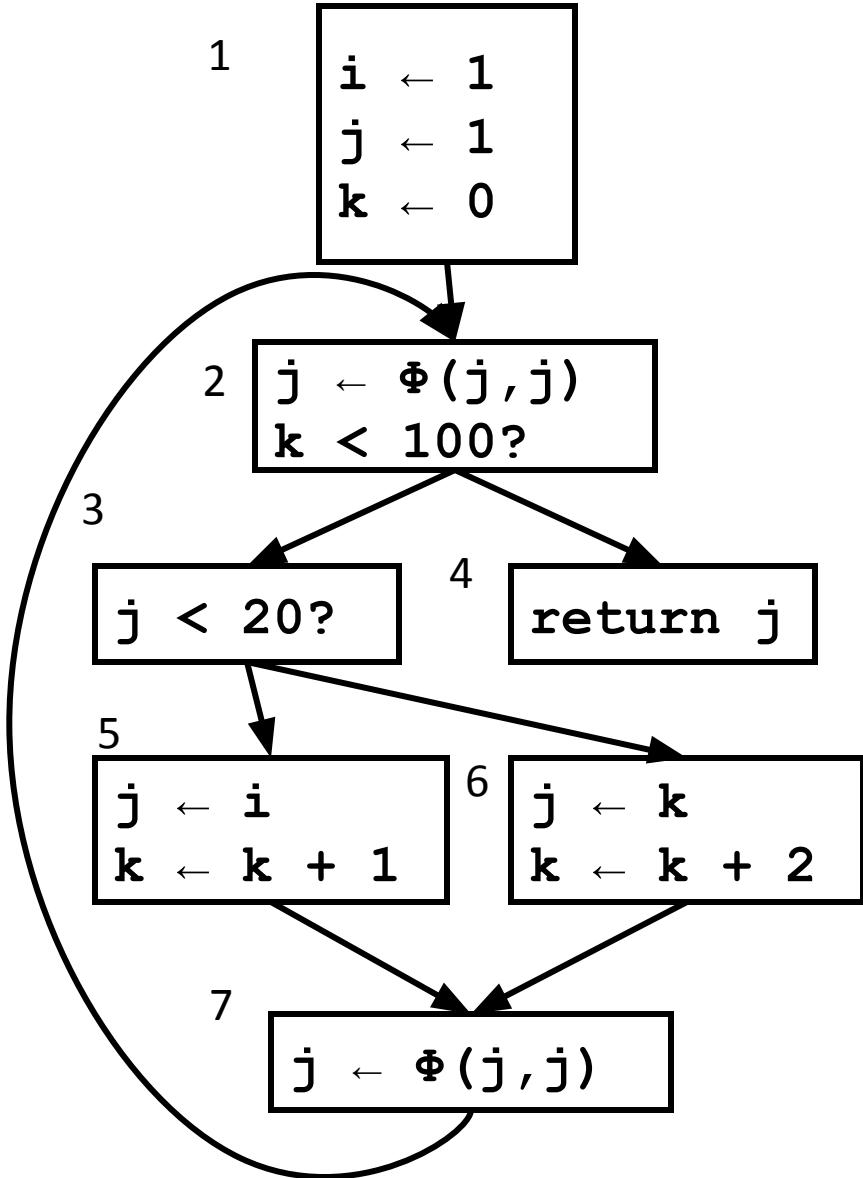
DFs

var j: W={1,5,6,7}

DF{1}

DF{5}

DF{7}



DFs

1 {}  
2 {2}  
3 {2}  
4 {}  
5 {7}  
6 {7}  
7 {2}

orig[n]

1 {i,j,k}  
2 {}  
3 {}  
4 {}  
5 {j,k}  
6 {j,k}  
7 {}

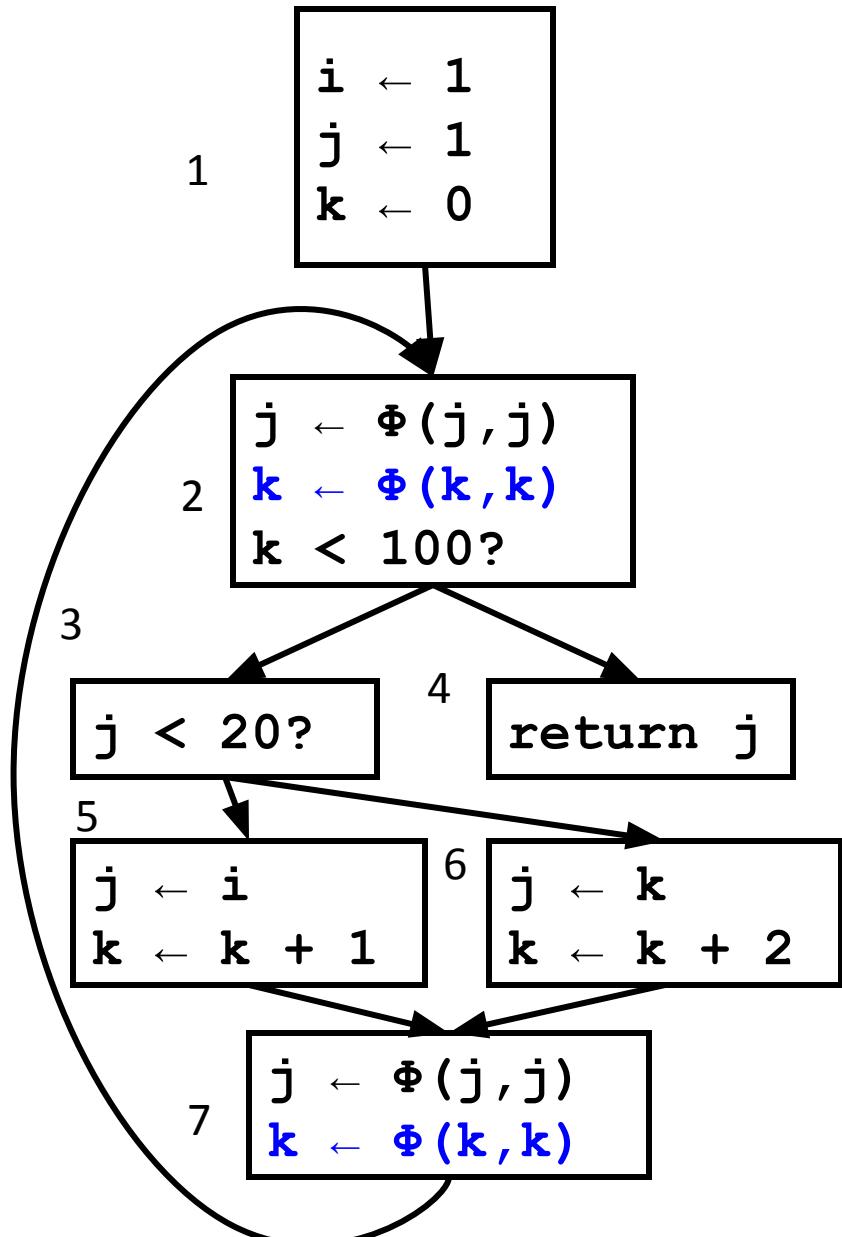
defsites[v]

i {1}  
j {1,5,6}  
k {1,5,6}

DFs

var j: W={1,5,6,7}

DF{1} DF{5} DF{7} DF{6}



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

orig[n]

1	{i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

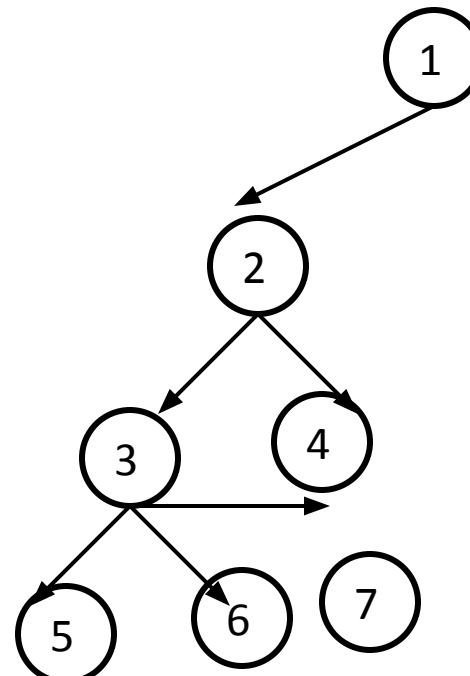
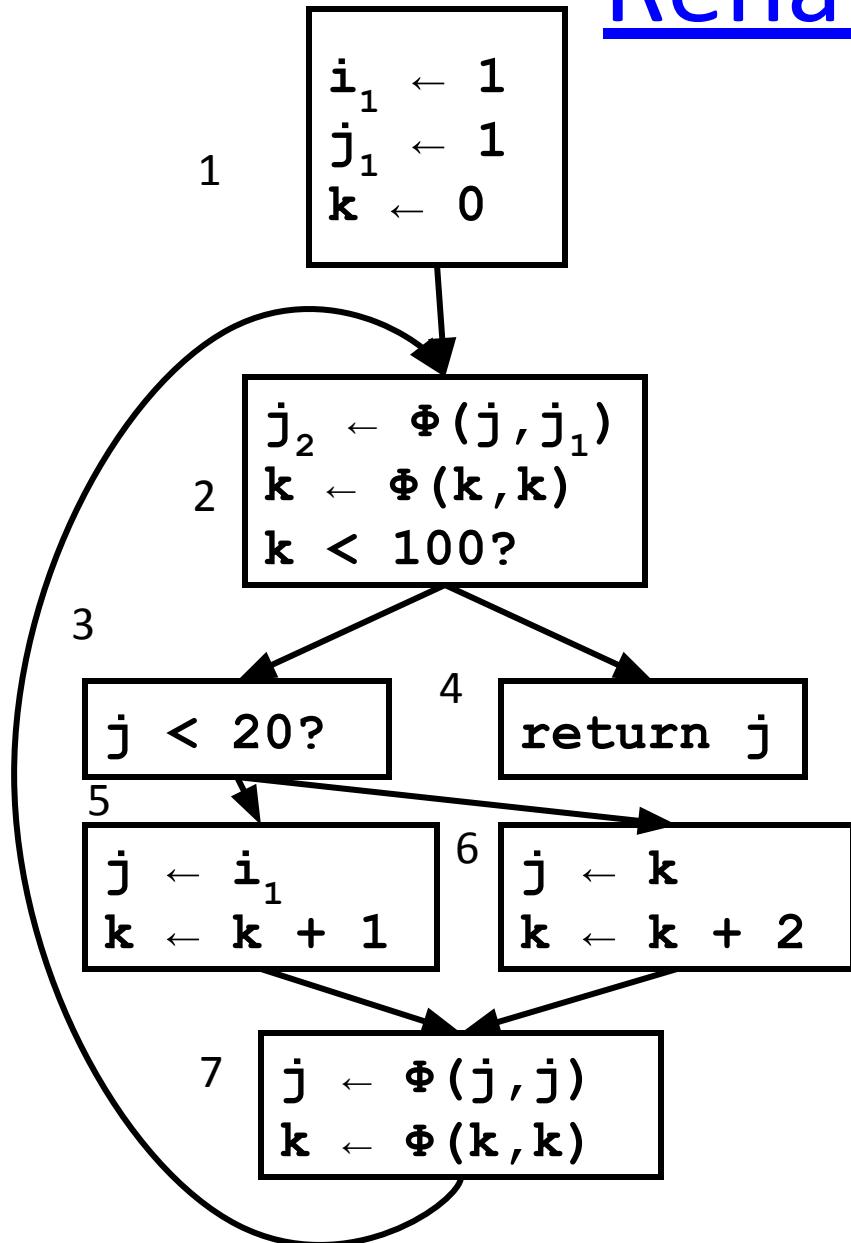
Def sites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

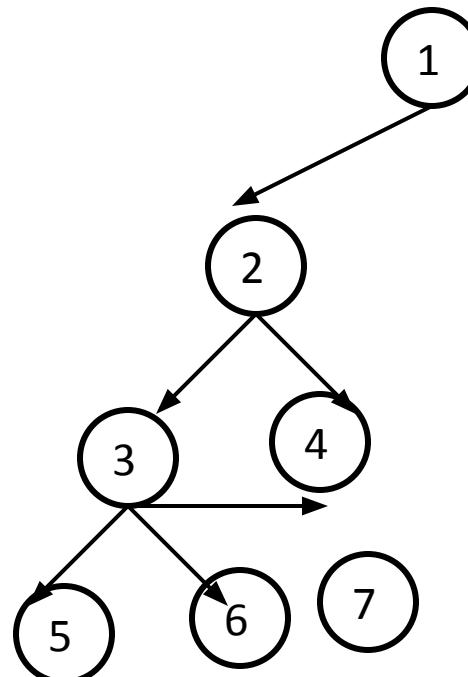
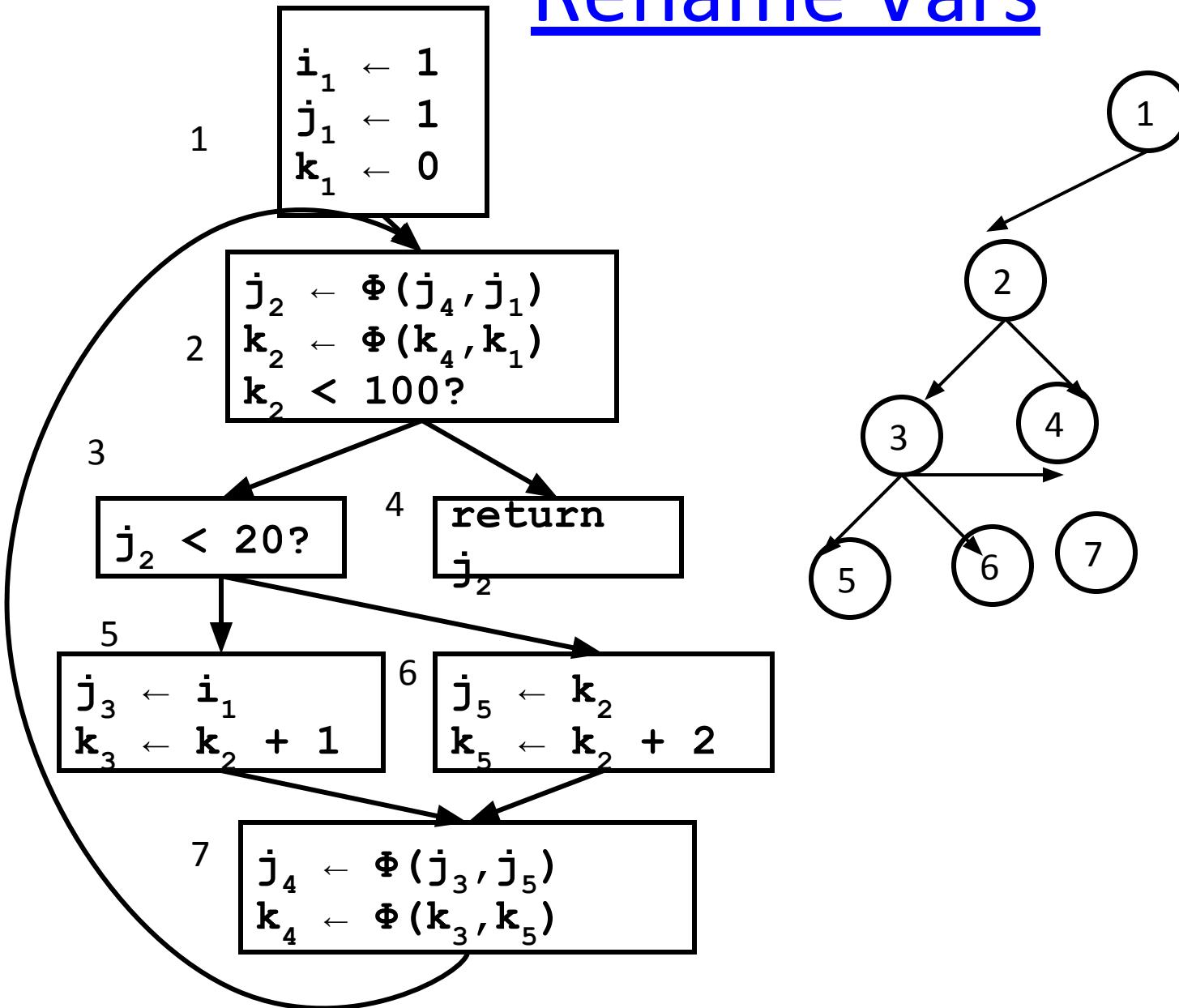
DFs

var **k**: W={1,5,6}

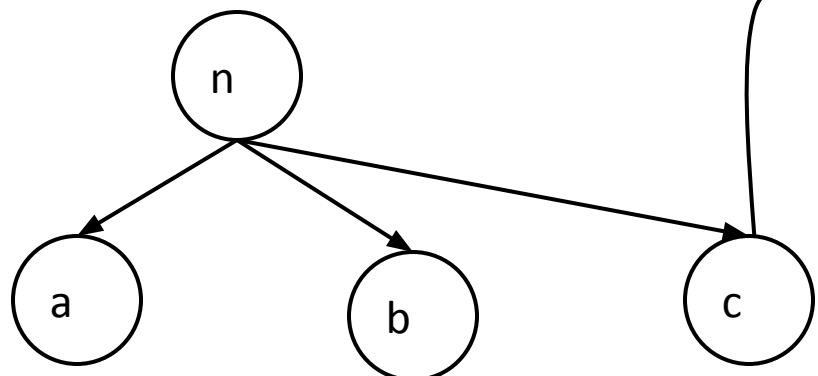
# Rename Vars



# Rename Vars

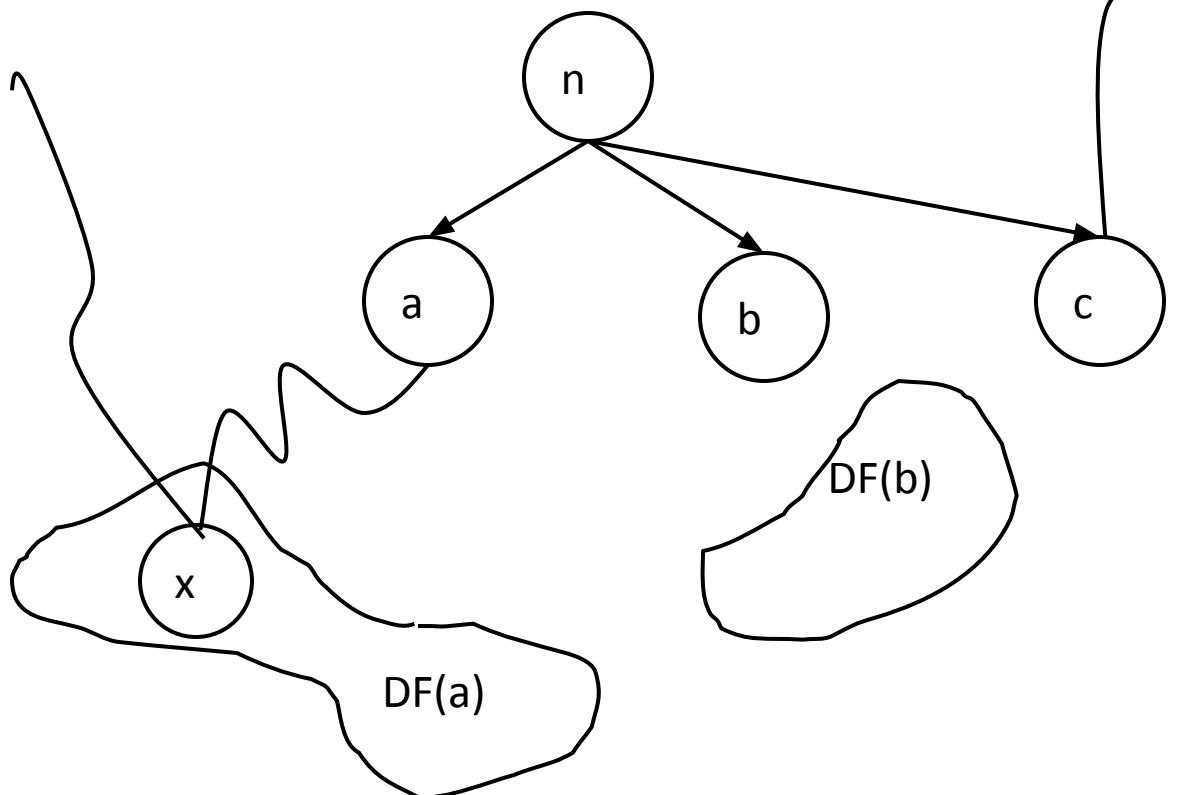


# Computing DF(n)



n dom a  
n dom b  
!n dom c

# Computing $DF(n)$



$n \text{ dom } a$   
 $n \text{ dom } b$   
 $\neg n \text{ dom } c$

# Computing the Dominance Frontier

```
compute-DF(n)
```

```
S = {}
```

```
foreach node y in succ[n]
```

```
    if idom(y) ≠ n
```

```
        S = S ∪ {y}
```

```
foreach child of n, c, in D-tree
```

```
    compute-DF(c)
```

```
    foreach w in DF[c]
```

```
        if !n dom w
```

```
            S = S ∪ {w}
```

```
DF[n] = S
```

The Dominance Frontier of a node  $x$  =  
 $\{ w \mid x \text{ dom } \text{pred}(w) \text{ AND } !(x \text{ sdom } w)\}$

# SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses

# Constant Propagation

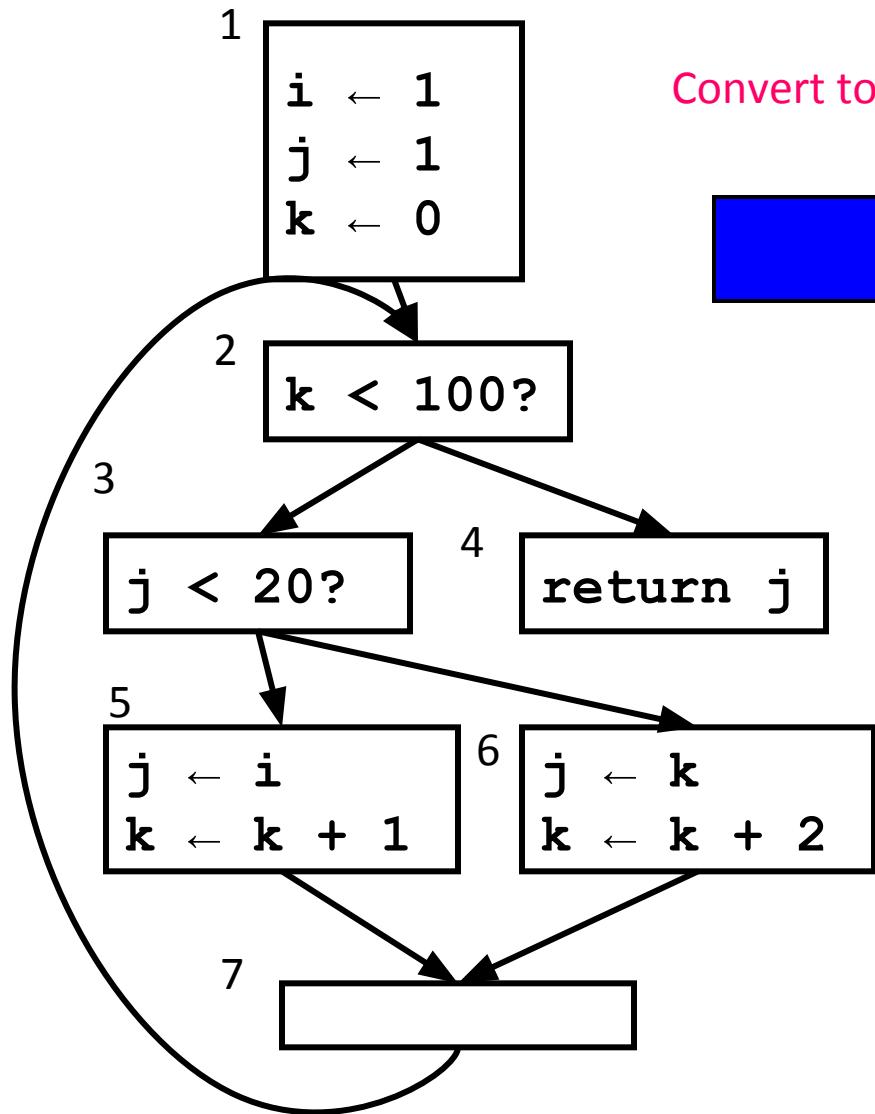
- If “ $v \leftarrow c$ ”, replace all uses of  $v$  with  $c$
- If “ $v \leftarrow \Phi(c,c,c)$ ” (each input is the same constant), replace all uses of  $v$  with  $c$

```
w ← list of all defs
while !W.isEmpty {
    Stmt S ← W.removeOne
    if ((S has form "v ← c") ||
        (S has form "v ← Φ(c,...,c)")) then {
        delete S
        foreach stmt U that uses v {
            replace v with c in U
            W.add(U)
        }
    }
}
```

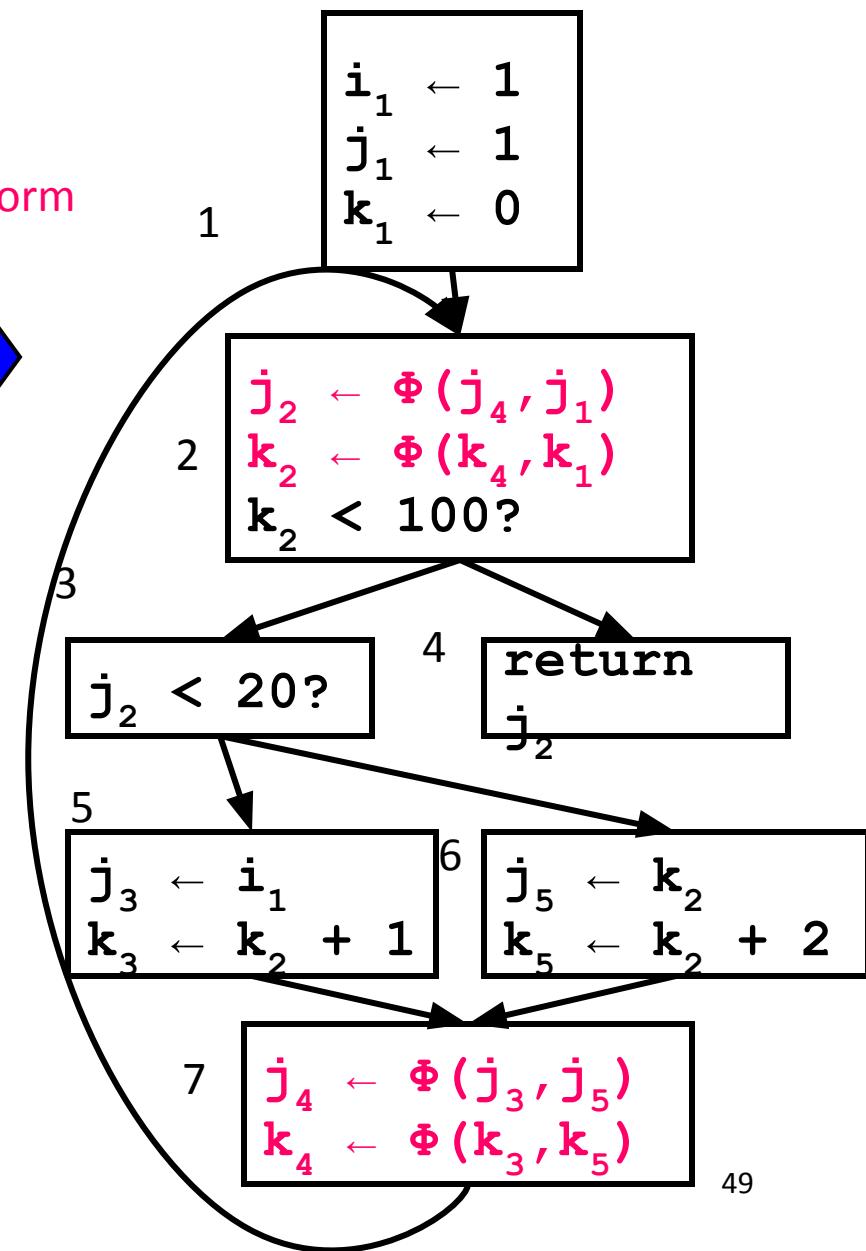
# Other Optimizations with SSA

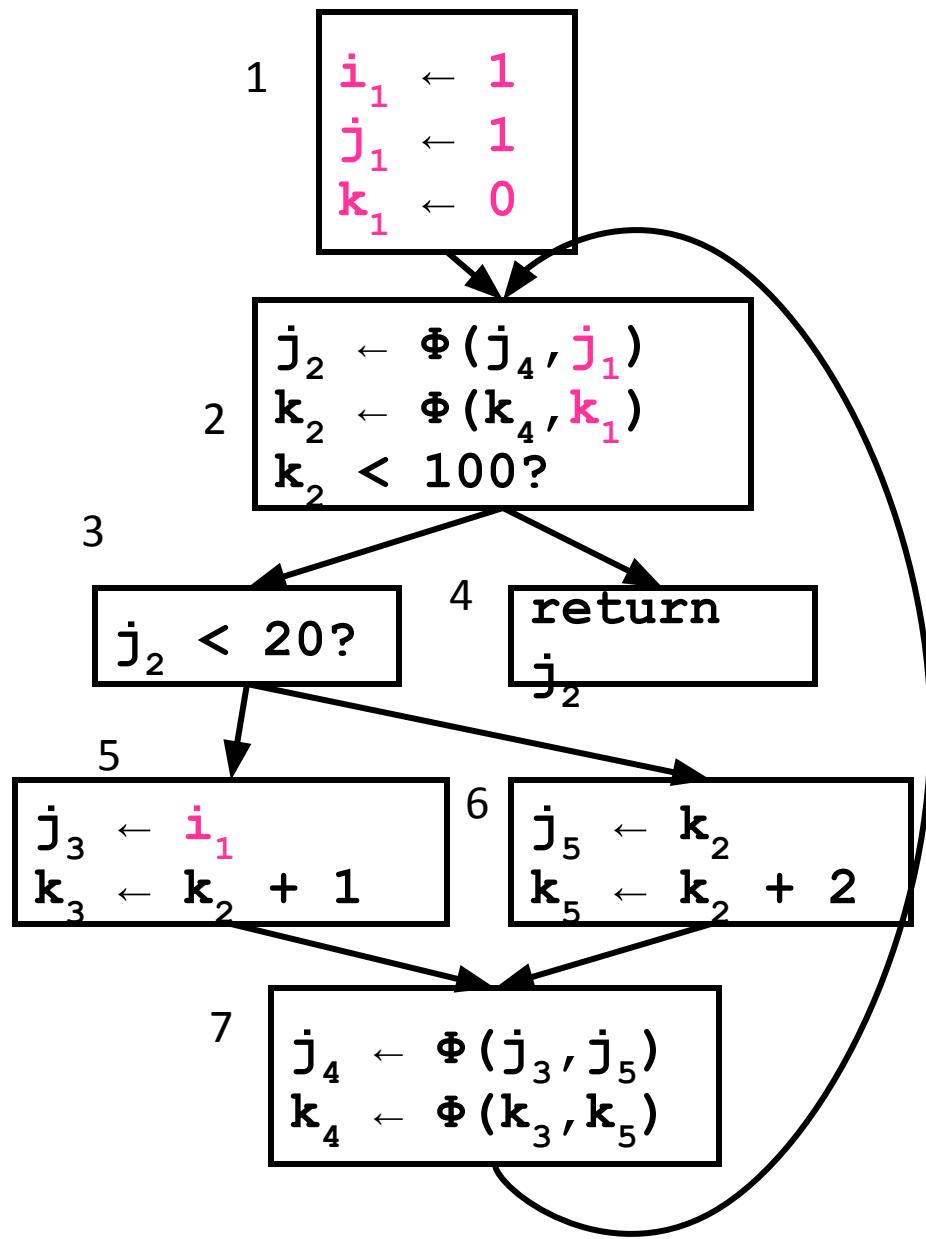
- Copy propagation
  - delete “ $x \leftarrow \Phi(y,y,y)$ ” and replace all x with y
  - delete “ $x \leftarrow y$ ” and replace all x with y
- Constant Folding
  - (Also, constant conditions too!)

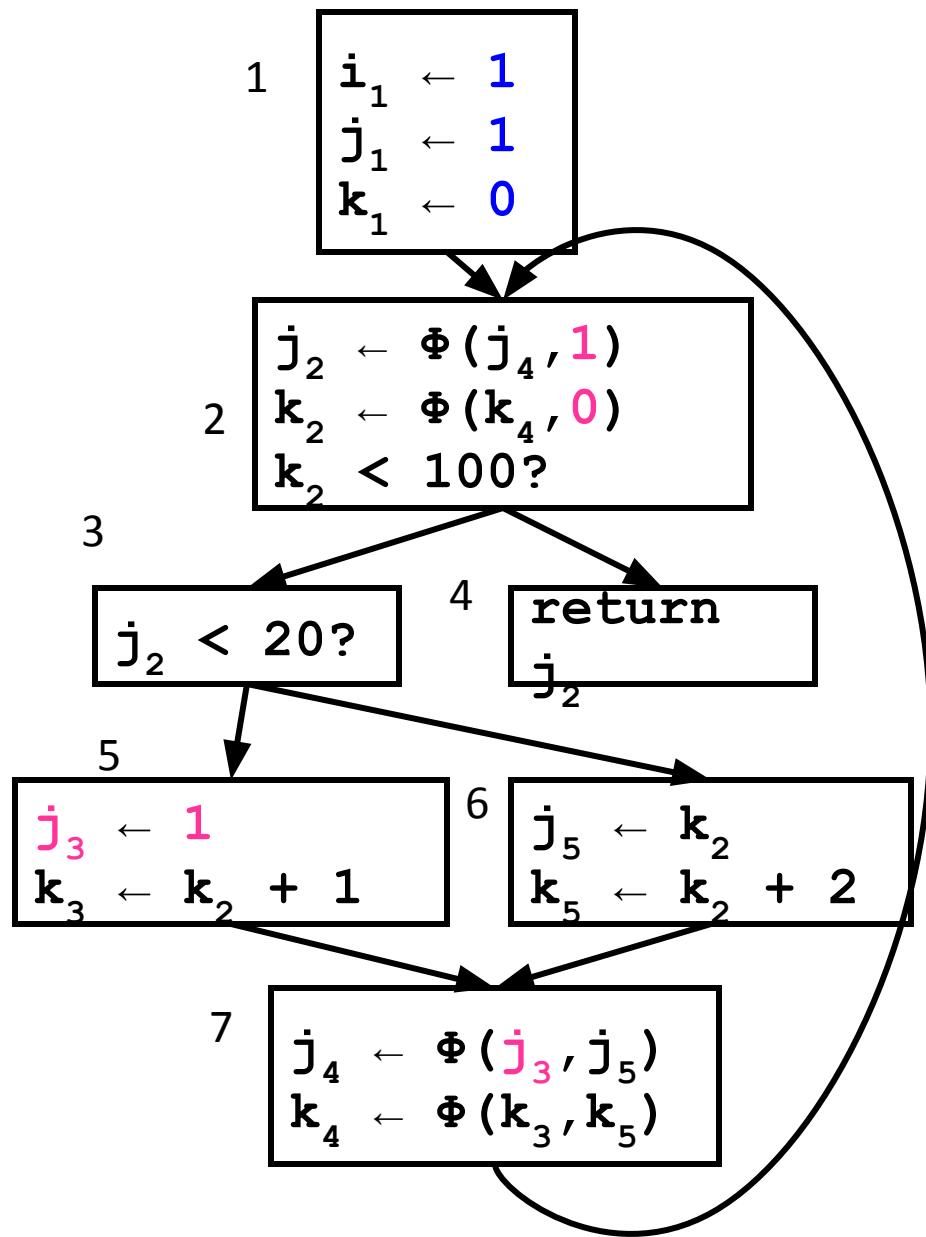
# Constant Propagation

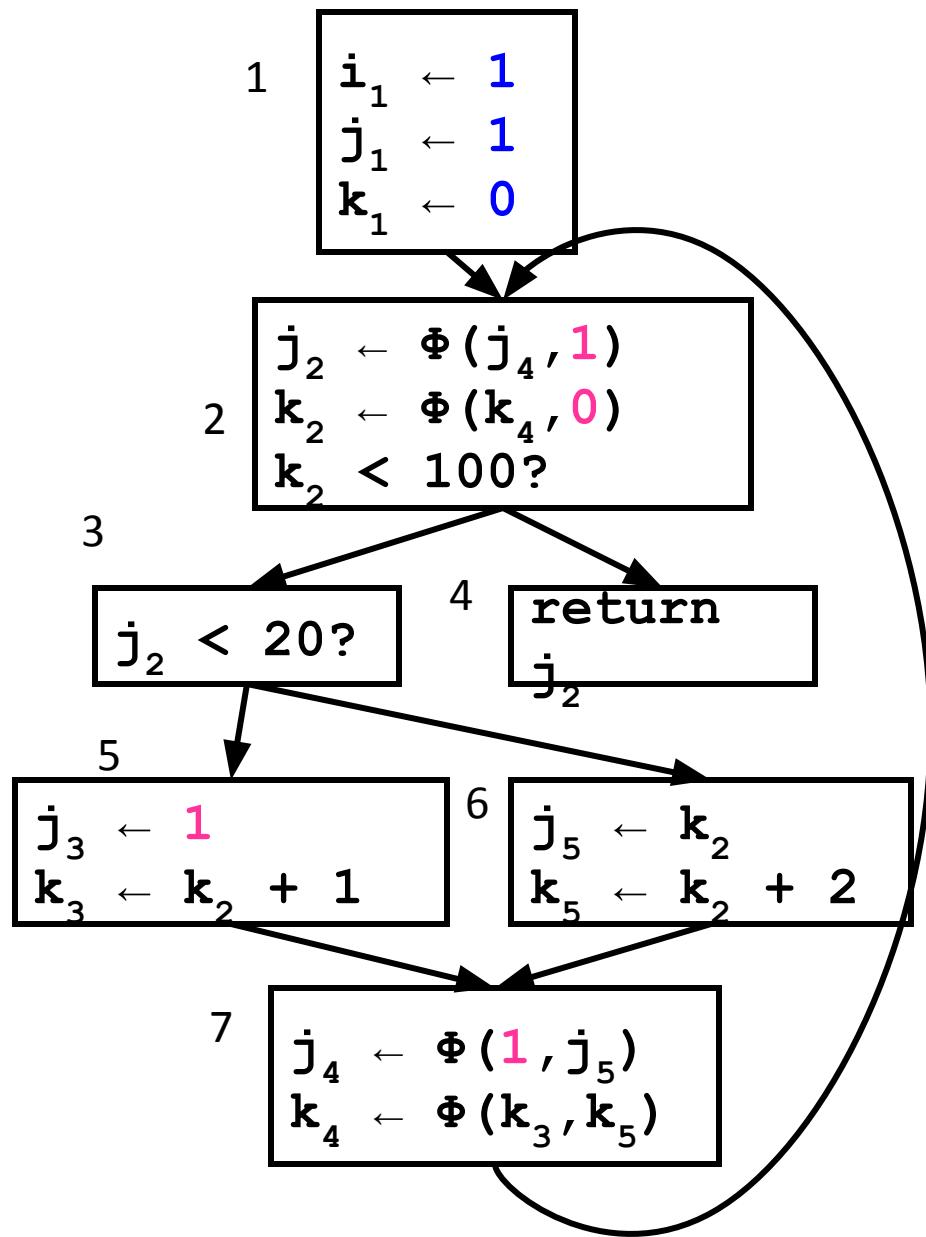


Convert to SSA Form



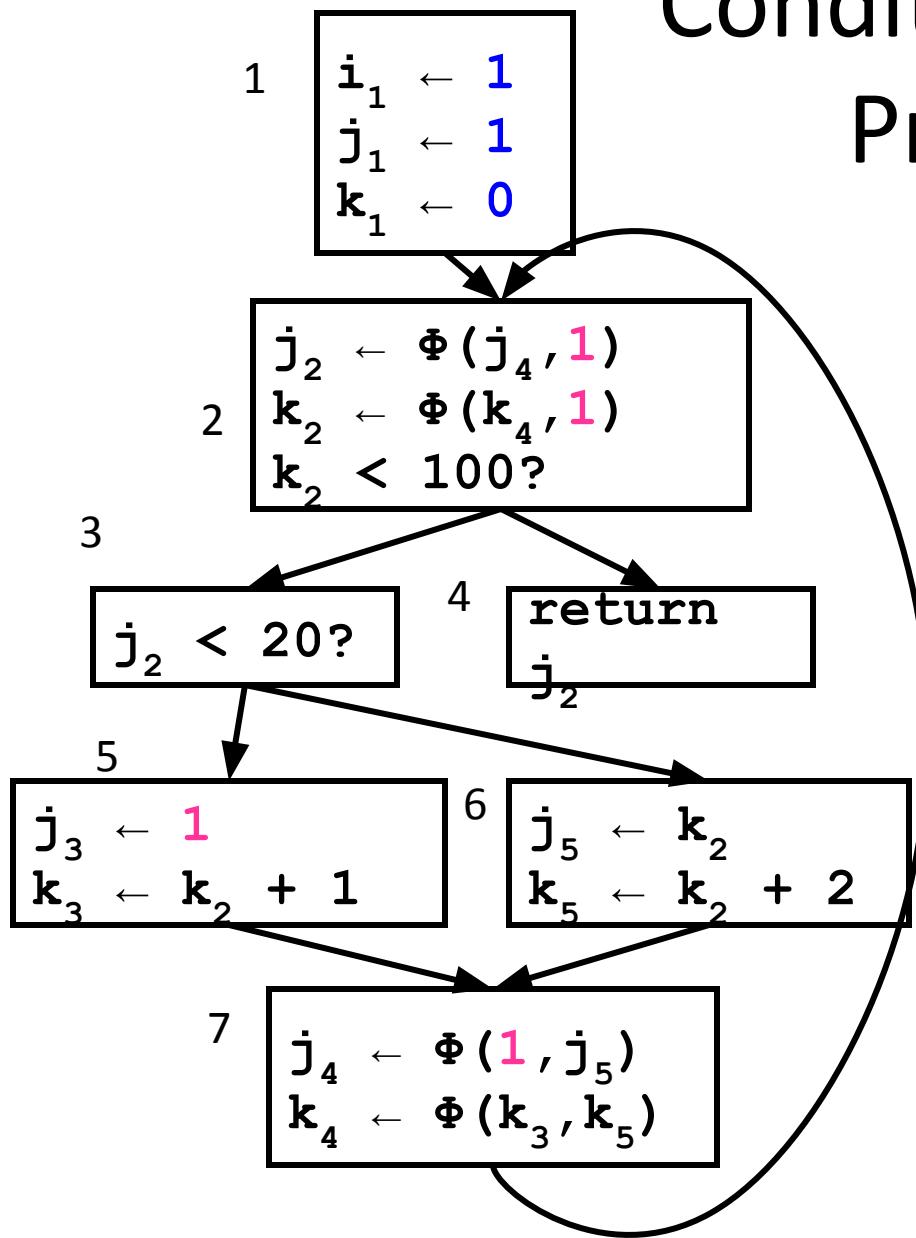






Not a very exciting result (yet)...

# Conditional Constant Propagation

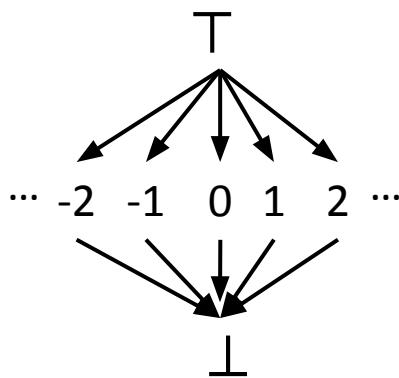


- Does block 6 ever execute?
- Simple Constant Propagation can't tell
- But “Conditional Const. Prop.” *can* tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes values are constants until proven otherwise

# Conditional Constant Propagation Algorithm

## Keeps track of:

- **Blocks**
  - assume unexecuted until proven otherwise
- **Variables**
  - assume not executed (only with proof of assignments of a non-constant value do we assume not constant)
  - Lattice for representing variables:

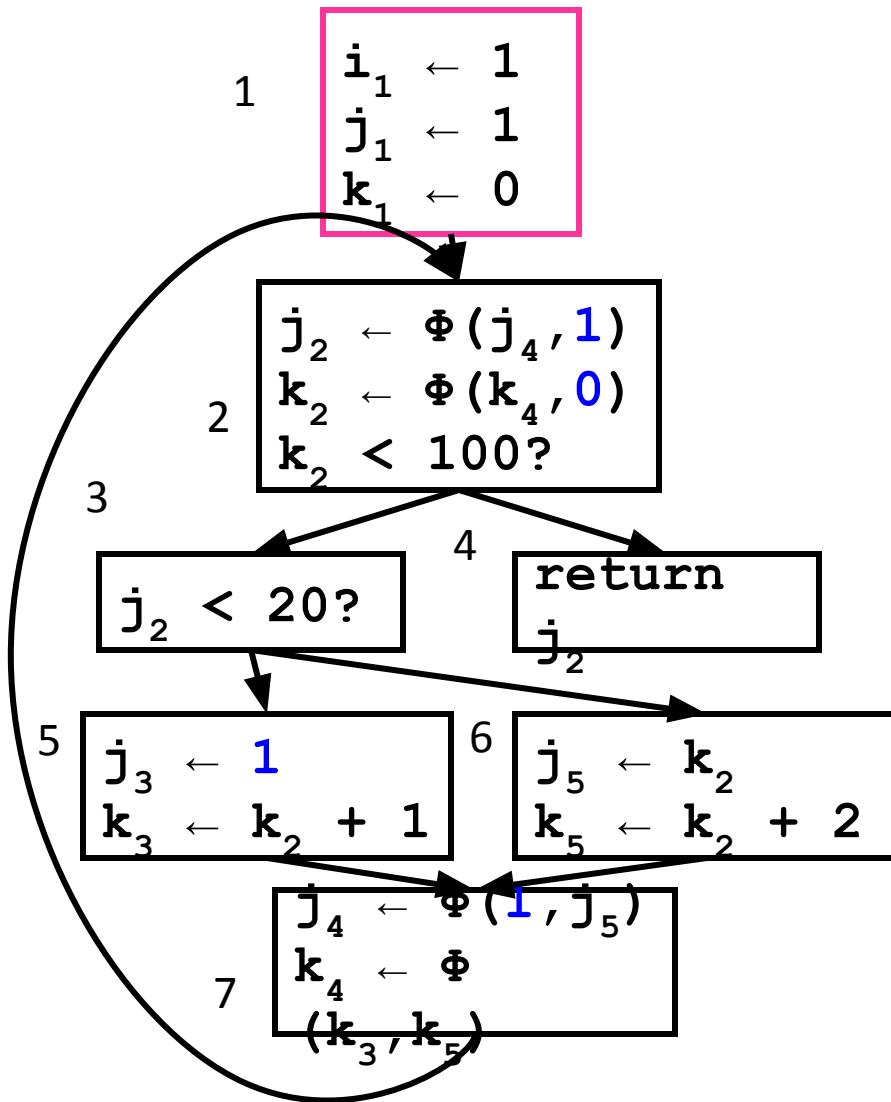


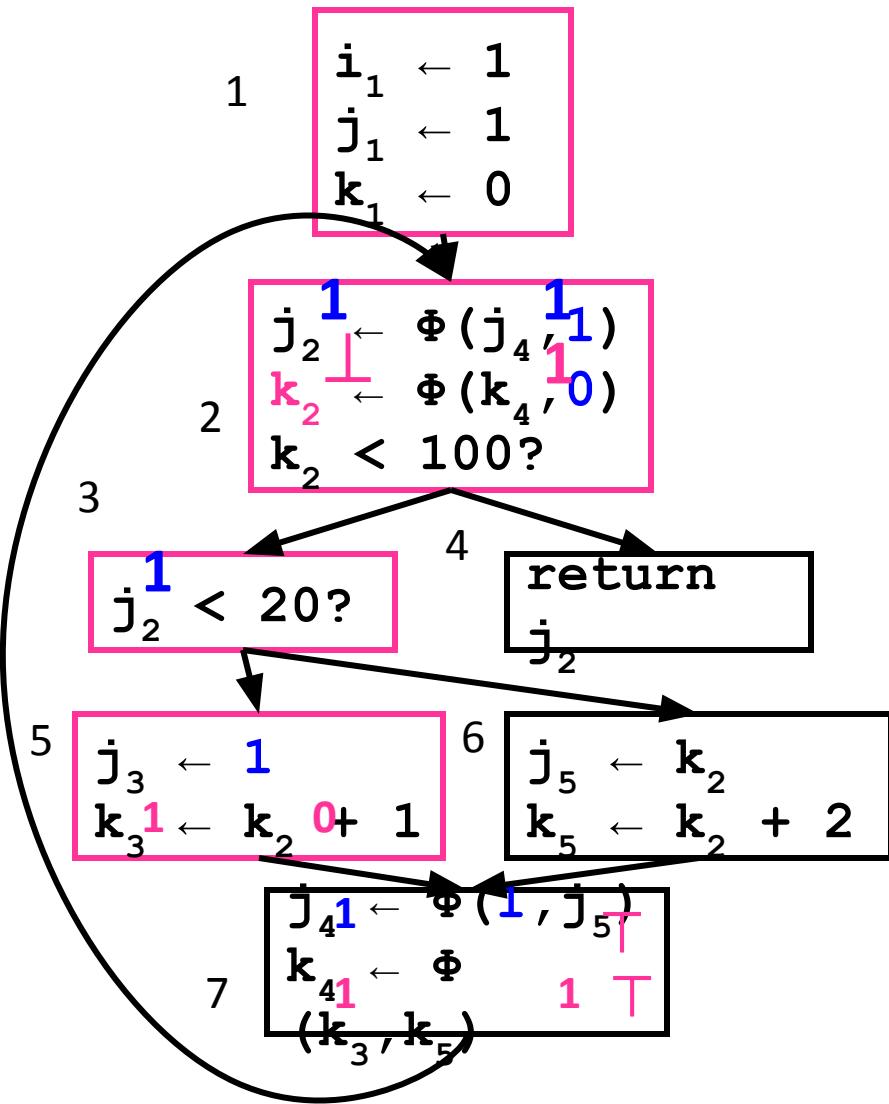
not executed

we have seen **evidence** that the variable has been **assigned a constant** with the value

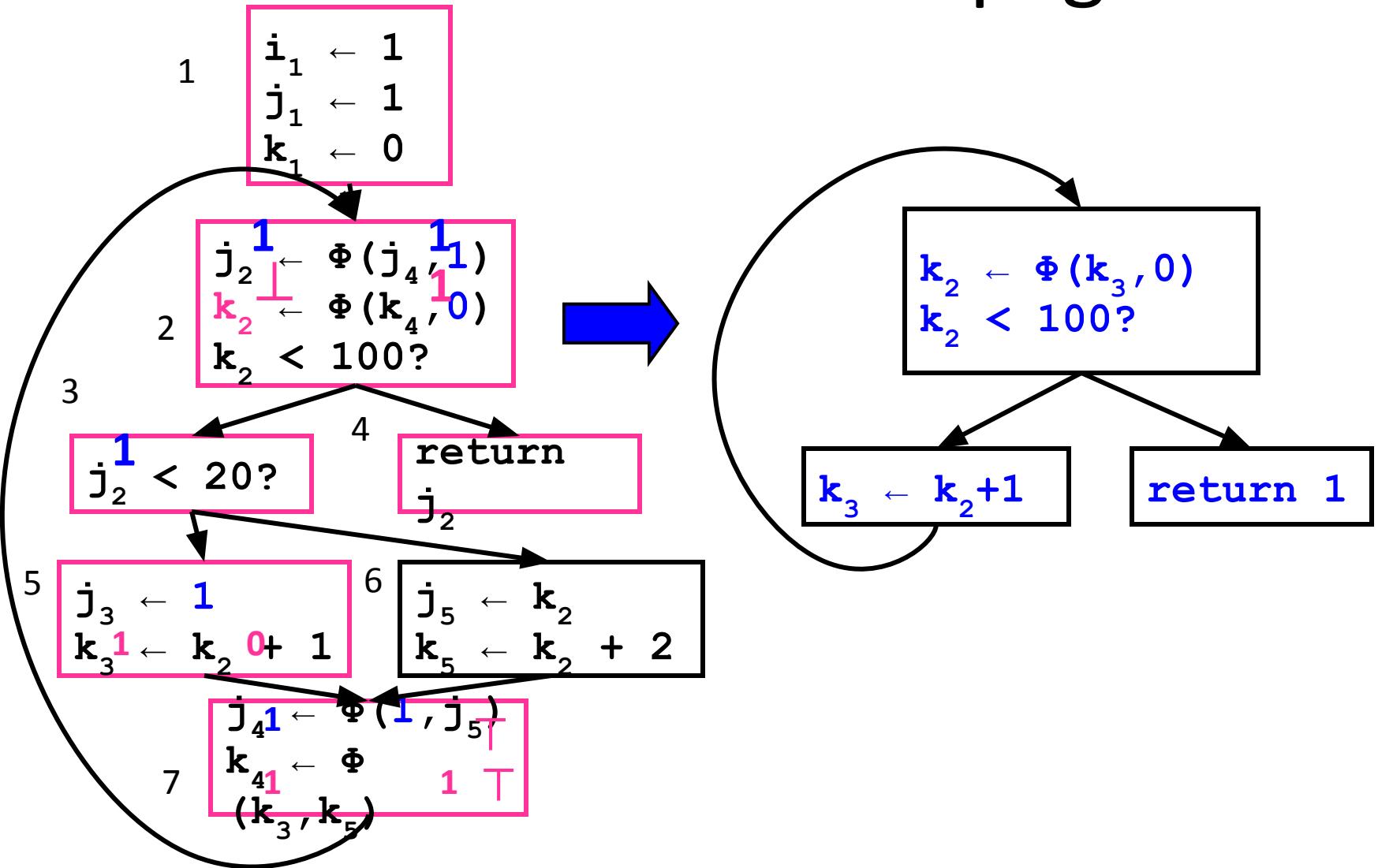
we have seen **evidence** that the variable **can hold different values** at different times

# Conditional Constant Propagation





# Conditional Constant Propagation



# CSC D70: Compiler Optimization Static Single Assignment (SSA)

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Backup Slides

# CSC D70: Compiler Optimization LICM: Loop Invariant Code Motion

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

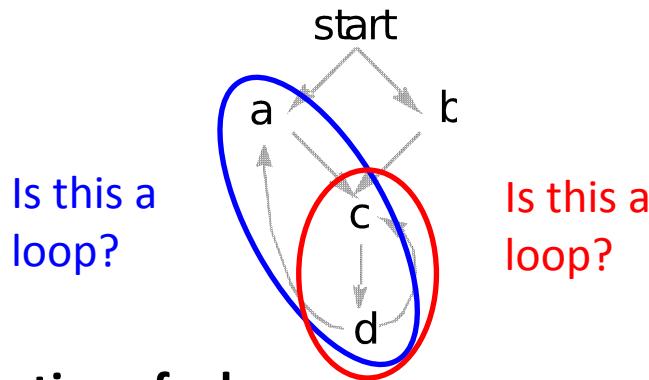
# Announcements

- Assignment 2 is out soon
- Midterm is Feb 28<sup>th</sup> (during the class)

# Refreshing: Finding Loops

# What is a Loop?

- **Goals:**
  - Define a loop in graph-theoretic terms (control flow graph)
  - Not sensitive to input syntax
  - A uniform treatment for all loops: DO, while, goto's
- **Not every cycle is a “loop” from an optimization perspective**

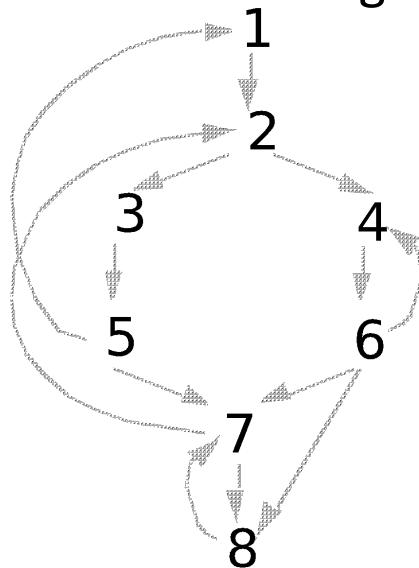


- **Intuitive properties of a loop**
  - single entry point
  - edges must form at least a cycle

# Formal Definitions

- **Dominators**

- Node  $d$  **dominates** node  $n$  in a graph ( $d \text{ dom } n$ ) if every path from the start node to  $n$  goes through  $d$



- Dominators can be organized as a **tree**
  - $a \rightarrow b$  in the **dominator tree** iff  $a$  immediately dominates  $b$

# Natural Loops

- Definitions
  - Single entry-point: *header*
    - a header dominates all nodes in the loop
  - A *back edge* is an arc whose head dominates its tail (tail  $\rightarrow$  head)
    - a back edge must be a part of at least one loop
  - The *natural loop of a back edge* is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.

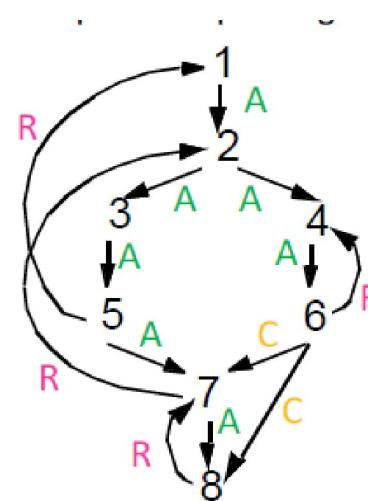
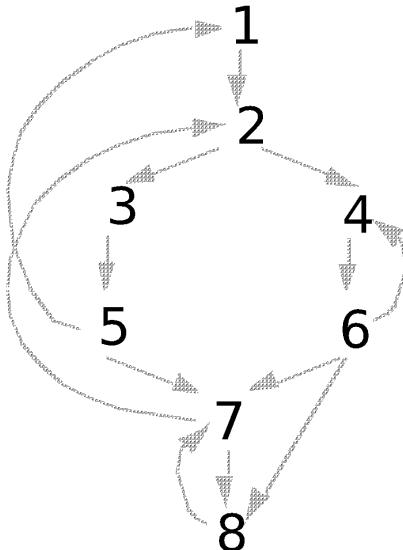
# Algorithm to Find Natural Loops

- Find the dominator relations in a flow graph
- Identify the back edges
- Find the natural loop associated with the back edge

# Finding Back Edges

- Depth-first spanning tree

- Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree

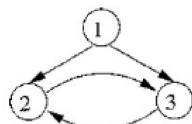


- Categorizing edges in graph

- Advancing (A) edges: from ancestor to proper descendant
- Cross (C) edges: from right to left
- Retreating (R) edges: from descendant to ancestor (not necessarily proper)

# Back Edges

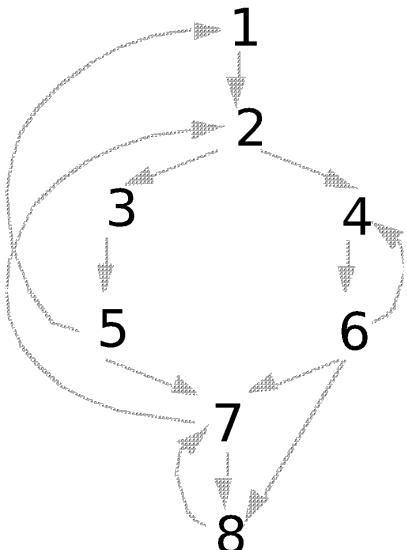
- **Definition**
  - **Back edge**:  $t \rightarrow h$ ,  $h$  dominates  $t$
- **Relationships between graph edges and back edges**
- **Algorithm**
  - Perform a depth first search
  - For each retreating edge  $t \rightarrow h$ , check if  $h$  is in  $t$ 's dominator list
- **Most programs (all structured code, and most GOTO programs) have **reducible** flow graphs**
  - retreating edges = back edges



A **nonreducible** flow graph

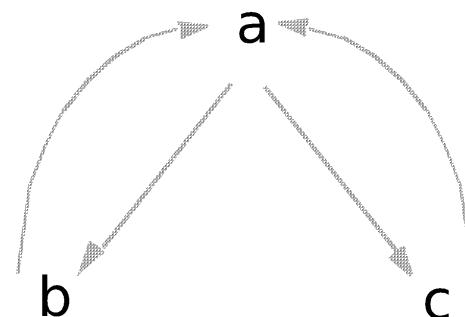
# Constructing Natural Loops

- The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.
- **Algorithm**
  - delete  $h$  from the flow graph
  - find those nodes that can reach  $t$   
(those nodes plus  $h$  form the natural loop of  $t \rightarrow h$ )



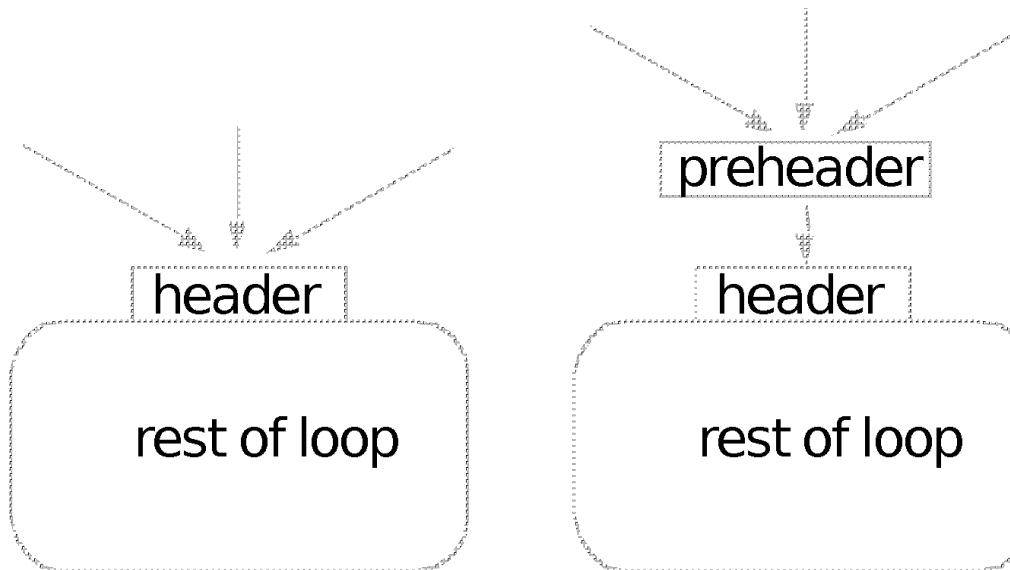
# Inner Loops

- **If two loops do not have the same header:**
  - they are either disjoint, or
  - one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.
- **If two loops share the same header:**
  - Hard to tell which is the inner loop
  - Combine as one



# Preheader

- Optimizations often require code to be executed once before the loop
- Create a preheader basic block for every loop

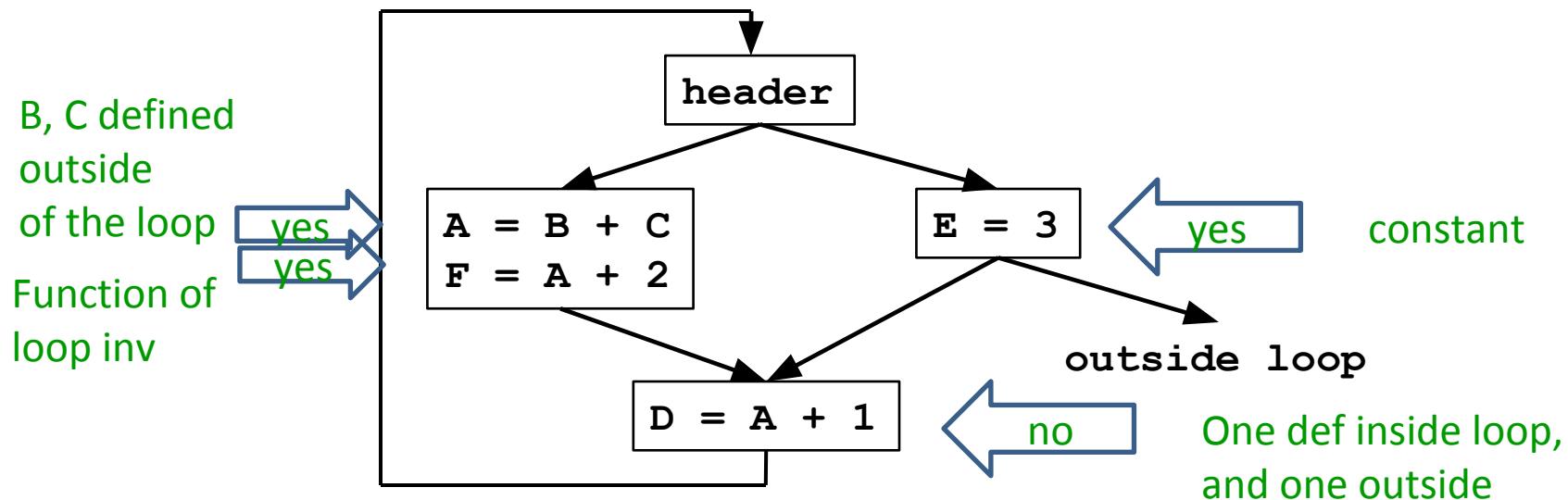


# Finding Loops: Summary

- Define loops in graph theoretic terms
- Definitions and algorithms for:
  - Dominators
  - Back edges
  - Natural loops

# Loop-Invariant Computation and Code Motion

- A **loop-invariant computation**:
  - a computation whose value does not change as long as control stays within the loop
- **Code motion**:
  - to move a statement within a loop to the preheader of the loop



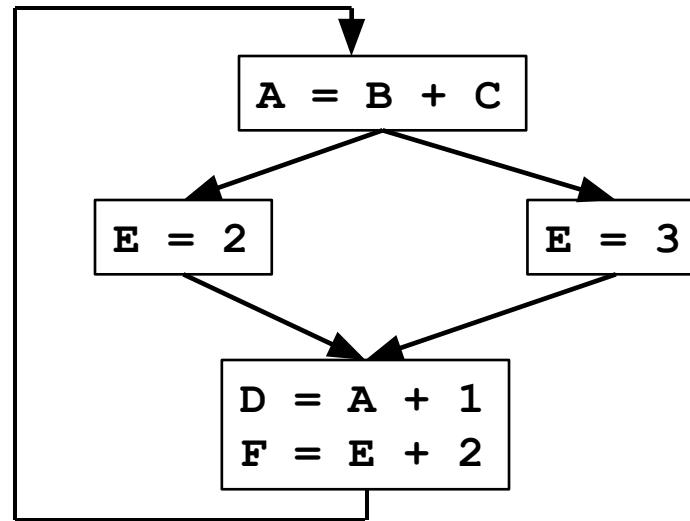
# Algorithm

- **Observations**
  - Loop invariant
    - operands are defined outside loop or invariant themselves
  - Code motion
    - not all loop invariant instructions can be moved to preheader
- **Algorithm**
  - Find invariant expressions
  - Conditions for code motion
  - Code transformation

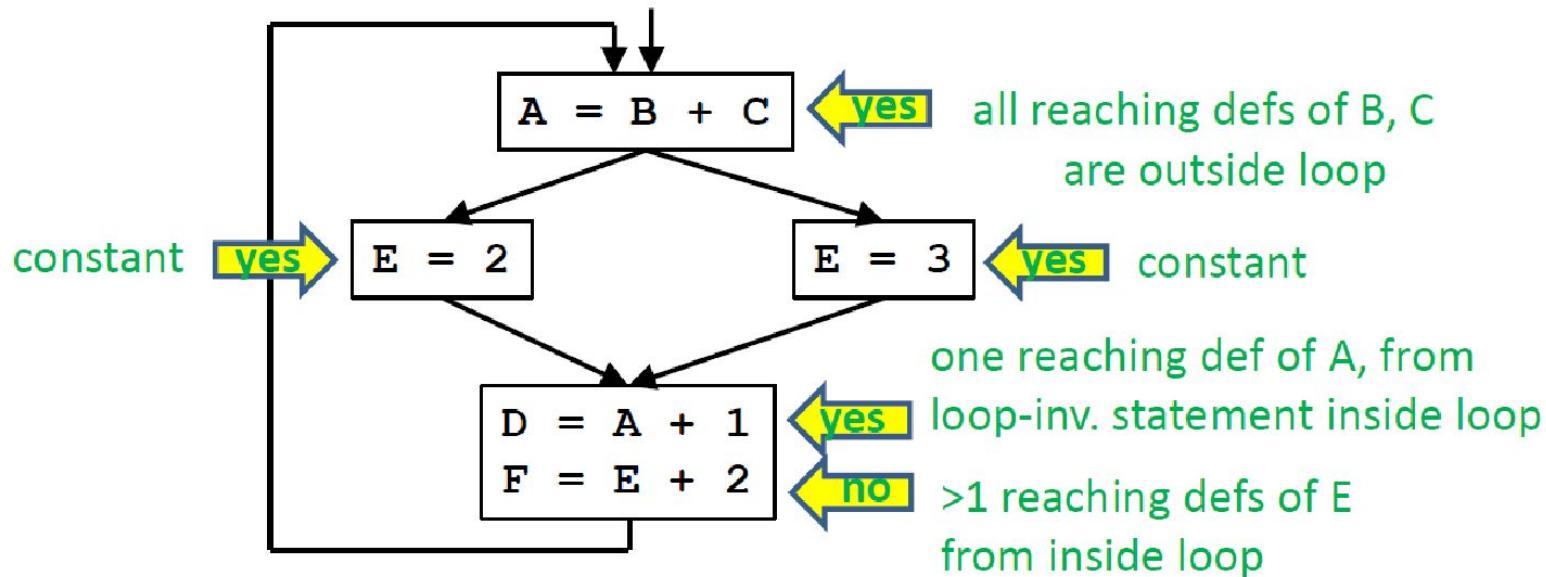
# Detecting Loop Invariant Computation

- Compute reaching definitions
- Mark INVARIANT if
  - all the definitions of B and C that reach a statement  $A=B+C$  are outside the loop
    - constant B, C?
- Repeat: Mark INVARIANT if
  - all reaching definitions of B are outside the loop, or
  - there is exactly one reaching definition for B, and it is from a loop-invariant statement inside the loop
  - similarly for Cuntil no changes to set of loop-invariant statements occur.

# Example

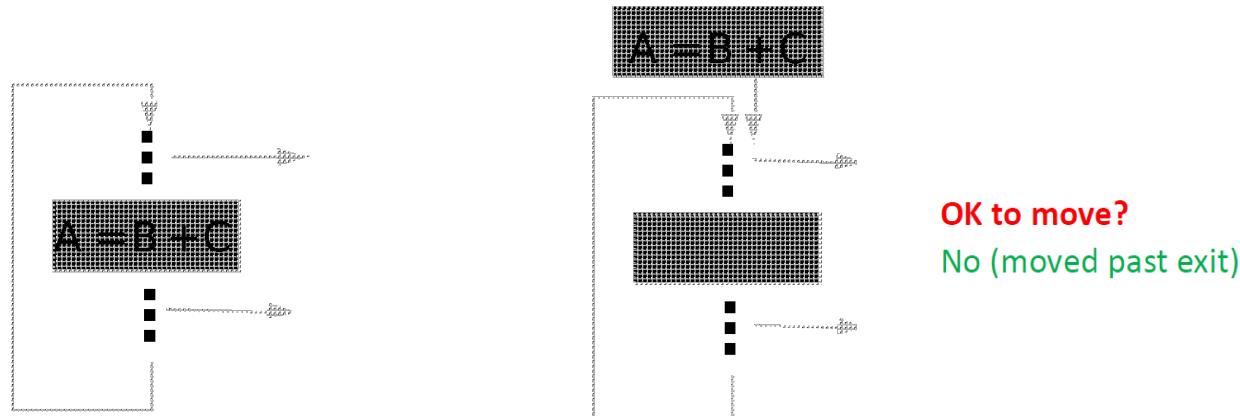


# Example



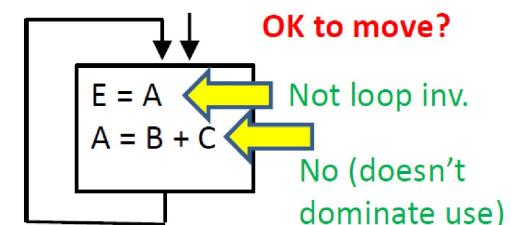
# Conditions for Code Motion

- **Correctness:** Movement does not change semantics of program
- **Performance:** Code is not slowed down



- **Basic idea:** defines once and for all

- control flow: once?  
Code dominates all exists
- other definitions: for all?  
No other definition
- other uses: for all?  
Dominates use or no other reaching defs to use

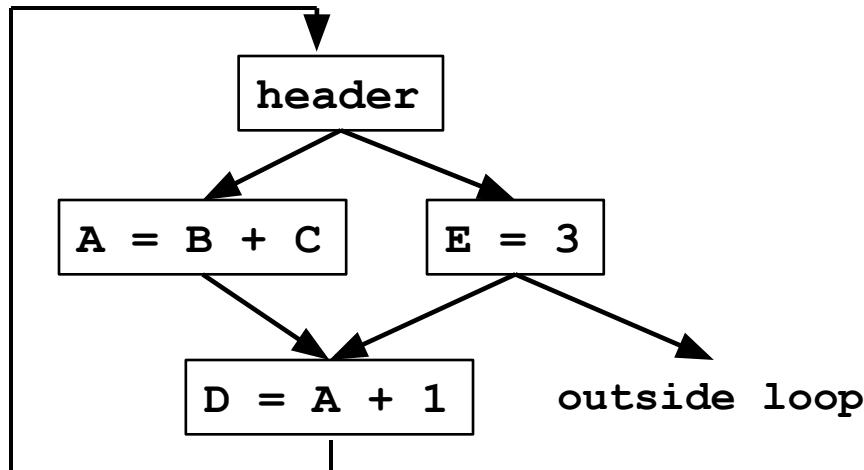


# Code Motion Algorithm

Given: a set of nodes in a loop

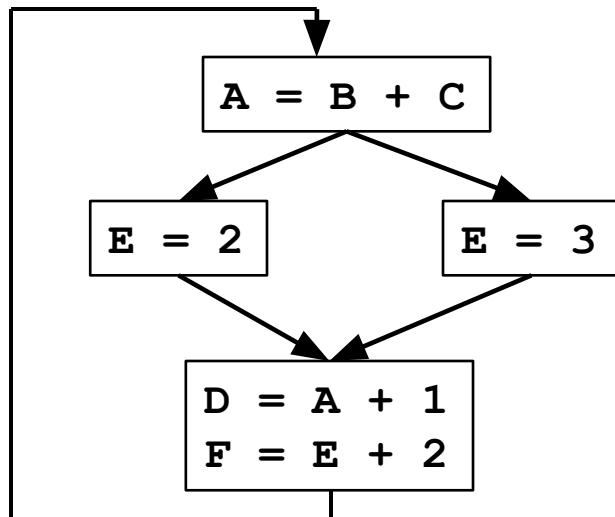
- **Compute reaching definitions**
- **Compute loop invariant computation**
- **Compute dominators**
- **Find the exits of the loop (i.e. nodes with successor outside loop)**
- **Candidate statement for code motion:**
  - loop invariant
  - in blocks that dominate all the exits of the loop
  - assign to variable not assigned to elsewhere in the loop
  - in blocks that dominate all blocks in the loop that use the variable assigned
- **Perform a depth-first search of the blocks**
  - Move candidate to preheader if all the invariant operations it depends upon have been moved

# Examples



Which statements can be moved to loop preheader?

Only  $E=3$ : only statement dominating all exits



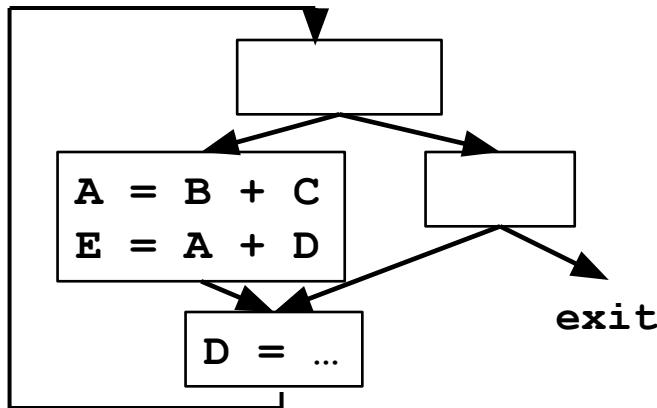
$A=B+C$   
 $D=A+1$

(Although  $E=2$ ,  $E=3$  are invariant, neither is only def of  $E$ )

defines once and for all

# More Aggressive Optimizations

- **Gamble on: most loops get executed**
  - Can we relax constraint of dominating all exits?



Can relax if destination not live after loop  
& can compute in preheader  
w/o causing an exception

- **Landing pads**

```
While p do s    →  if p {  
                      preheader  
                      repeat  
                      s  
                      until not p;  
                  }
```

Ensures preheader  
executes only  
if enter loop

# LICM Summary

- Precise definition and algorithm for loop invariant computation
- Precise algorithm for code motion
- Use of reaching definitions and dominators in optimizations

# **Induction Variables and Strength Reduction**

- I. Overview of optimization
- II. Algorithm to find induction variables

# Example

```
FOR i = 0 to 100  
    A[i] = 0;
```

```
i = 0  
L2: IF i>=100 GOTO L1  
    t1 = 4 * i  
    t2 = &A + t1  
    *t2 = 0  
    i = i+1  
    GOTO L2
```

L1:

```

    i = 0
L2: IF i>=100...
    t1 = 4 * i
    t2 = &A + t1
    *t2 = 0
    i = i+1
    GOTO L2

```

# Definitions

- A **basic induction variable** is
  - a variable  $X$  whose only definitions within the loop are assignments of the form:
 
$$X = X + c \text{ or } X = X - c,$$
 where  $c$  is either a **constant** or a **loop-invariant variable**.
- An **induction variable** is
  - a **basic induction variable**, or
  - a variable **defined once** within the loop, whose value is a **linear function of some basic induction variable** at the time of the definition:  

$$A = c_1 * B + c_2$$
- The **FAMILY of a basic induction variable  $B$**  is
  - the set of induction variables  $A$  such that each time  $A$  is assigned in the loop, the value of  $A$  is a linear function of  $B$ .

# Optimizations

## 1. Strength reduction:

- A is an induction variable in family of basic induction variable B ( $A = c_1 * B + c_2$ )

- Create new variable:  $A'$
- Initialization in preheader:  $A' = c_1 * B + c_2;$
- Track value of B: add after  $B=B+x$ :  $A' = A' + x * c_1;$
- Replace assignment to A: replace lone  $A =$  with  $A = A'$

```
i = 0
L2: IF i>=100 GOTO L1
t1 = 4 * i
t2 = &A + t1
*t2 = 0
i = i+1
GOTO L2
```

```
t1' = 0
t2' = &A
```

```
t1 = t1'
t2 = t2'
t1' = t1' + 4
t2' = t2' + 4
```

Induction variables:  
 $t1 = 4*i$   
 $t2 = 4*i + &A$

# Optimizations (continued)

## 2. Optimizing **non-basic** induction variables

- copy propagation
- dead code elimination

## 3. Optimizing **basic** induction variables

- Eliminate basic induction variables used only for
  - calculating other induction variables and loop tests
- Algorithm:
  - Select an **induction variable A in the family of B**, preferably with simple constants ( $A = c_1 * B + c_2$ ).
  - Replace a comparison such as

```
if B > x goto L1
```

with

```
if (A' > c1 * x + c2) goto L1
```

(assuming  $c_1$  is positive)
  - **if B is live at any exit from the loop, recompute it from A'**
    - After the exit,  $B = (A' - c_2) / c_1$

# Example (continued)

```
for(i=0; i<100; i++)
    A[i] = 0;
```

Induction variables:  
 $t1 = 4i$   
 $t2 = 4i + \&A$

```
i = 0
L2: IF i >= 100 GOTO L1
t1 = 4 * i
t2 = &A + t1
*t2 = 0
i = i + 1
GOTO L2
L1:
```

$t1' = 0$   
 $t2' = \&A$   
 $t1 = t1'$   
 $t2 = t2'$   
 $*t2' = 0$   
 $t1' = t1' + 4$   
 $t2' = t2' + 4$

```
IF  $t2' >= \&A + 400$ 
 $t2' = \&A$ 
 $t3' = \&A + 400$ 
L2: IF  $t2' >= t3'$  GOTO L1
 $*t2' = 0$ 
 $t2' = t2' + 4$ 
GOTO L2
L1:
```

$$B \geq X \Rightarrow A' \geq c_1 * X + c_2$$

# II. Basic Induction Variables

- A **BASIC induction variable in a loop L**
  - a variable  $X$  whose **only definitions within L** are assignments of the form:  
 $X = X+c$  or  $X = X-c$ , where  $c$  is either a constant or a loop-invariant variable.
- **Algorithm:** can be detected by scanning L
- **Example:**

$k = 0;$

```
for (i = 0; i < n; i++) {  
    k = k + 3;  
    ... = m;  
    if (x < y)  
        k = k + 4;  
    if (a < b)  
        m = 2 * k;  
    k = k - 2;  
    ... = m;
```

Basic induction variable(s)?  $i, k$

Additional induction variable(s)?

$m = 2k+0$  (in family of  $k$ )

*Each iteration may execute a different number of increments/decrements!!*

# Strength Reduction Algorithm

- **Key idea:**
  - For each induction variable A, ( $A = c_1 * B + c_2$  at time of definition)
    - variable A' holds expression  $c_1 * B + c_2$  at all times
    - replace definition of A with  $A=A'$  only when executed
- **Result:**
  - Program is correct
  - Definition of A does not need to refer to B

# Finding Induction Variable Families

- Let B be a basic induction variable
  - Find all induction variables A in family of B:
    - $A = c_1 * B + c_2$   
(where B refers to the value of B at time of definition)
- Conditions:
  - If A has a single assignment in the loop L, and assignment is one of:

```
A = B * c
A = c * B
A = B / c    (assuming A is real)
A = B + c
A = c + B
A = B - c
A = c - B
```
  - OR, ... (next page)

# Finding Induction Variable Families (continued)

Let D be an induction variable in the family of B ( $D = c_1 * B + c_2$ )

- If A has a single assignment in the loop L, and assignment is one of:

```
A = D * c
A = c * D
A = D / c    (assuming A is real)
A = D + c
A = c + D
A = D - c
A = c - D
```

- No definition of D outside L reaches the assignment to A
- Between the lone point of assignment to D in L and the assignment to A, there are no definitions of B

# Induction Variable Family - 1

```
L2: IF i>=100 GOTO L1  
      t2 = t1 + 10  
      t1 = 4 * i  
      t3 = t1 * 8  
      i = i + 1  
      goto L2
```

L1:

Is i a basic induction variable?	yes
Is t2 in family of i?	no (fails Rule 2)
Is t1 in family of i?	yes (by C1)
Is t3 in family of i?	yes (by C2 with A:t3, D:t1, B:i)

## Condition C1

A has a single assignment in the loop L of the form  $A = B*c$ ,  $c*B$ ,  $B+c$ , etc

## Condition C2

A is in family of B if  $D = c_1 * B + c_2$  for basic induction variable B and:

- Rule 1: A has a single assignment in the loop L of the form  $A = D*c$ ,  $D+c$ , etc
- Rule 2: No definition of D outside L reaches the assignment to A
- Rule 3: Every path between the lone point of assignment to D in L and the assignment to A has the same sequence (possibly empty) of definitions of B

# Induction Variable Family - 2

```
L3: IF i>=100 GOTO L1  
      t1 = 4 * i  
      IF t1 < 50 GOTO L2  
      i = i + 2  
L2: t2 = t1 + 10  
      i = i + 1  
      goto L3
```

Is **i** a basic induction variable? yes (all are  $i=i+c$ )  
Is **t1** in family of **i**? yes (by C1)  
Is **t2** in family of **i**? no (fails Rule 3)

L1:

## Condition C1

A has a single assignment in the loop L of the form  $A = B*c$ ,  $c*B$ ,  $B+c$ , etc

## Condition C2

A is in family of B if  $D = c_1 * B + c_2$  for basic induction variable B and:

- Rule 1: A has a single assignment in the loop L of the form  $A = D*c$ ,  $D+c$ , etc
- Rule 2: No definition of D outside L reaches the assignment to A
- Rule 3: Every path between the lone point of assignment to D in L and the assignment to A has the same sequence (possibly empty) of definitions of B

# Summary

- Precise definitions of induction variables
- Systematic identification of induction variables
- Strength reduction
- Clean up:
  - eliminating basic induction variables
    - used in other induction variable calculations
    - replacement of loop tests
  - eliminating other induction variables
    - standard optimizations

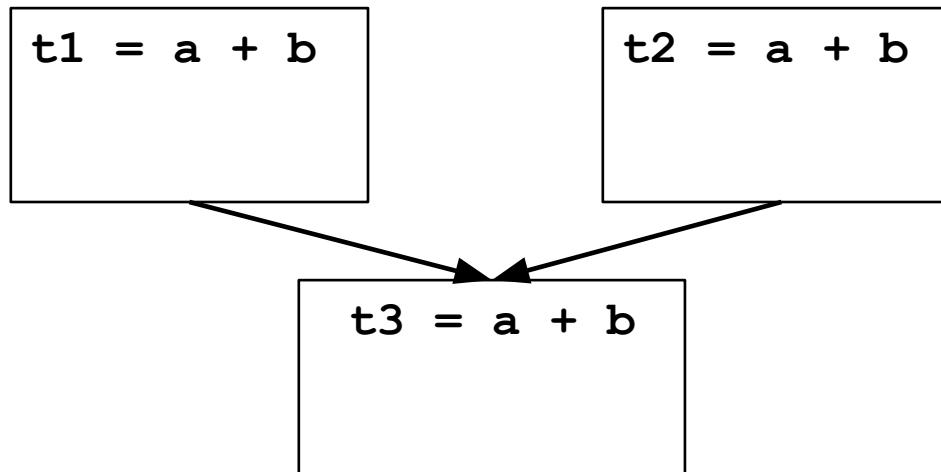
# Partial Redundancy Elimination

## Global code motion optimization

1. Remove partially redundant expressions
2. Loop invariant code motion
3. Can be extended to do Strength Reduction
  - No loop analysis needed
  - Bidirectional flow problem

# Redundancy

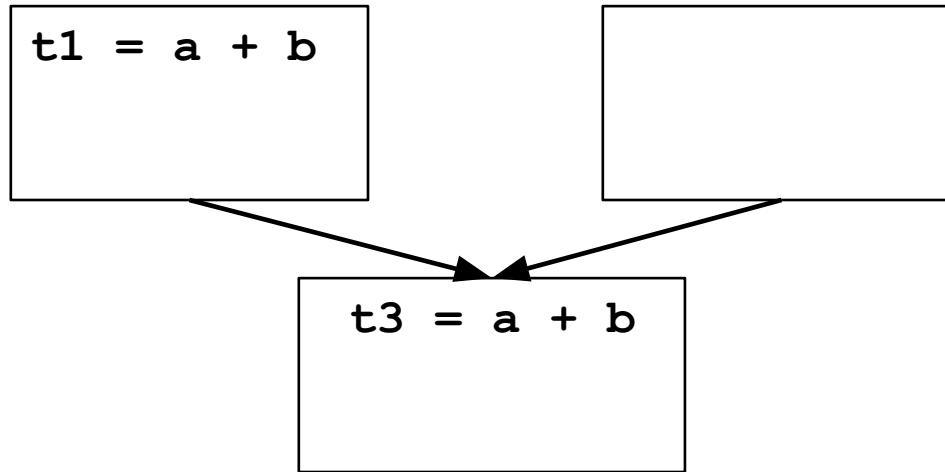
- A Common Subexpression is a Redundant Computation



- Occurrence of expression E at P is **redundant** if E is **available** there:
  - E is evaluated along every path to P, with no operands redefined since.
- Redundant expression can be eliminated

# Partial Redundancy

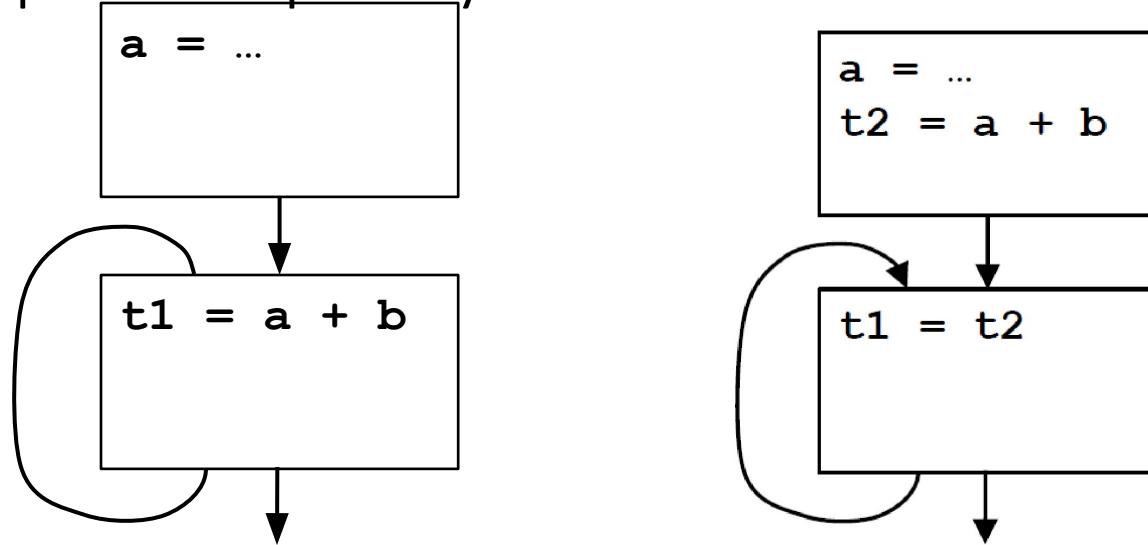
- Partially Redundant Computation



- Occurrence of expression E at P is **partially redundant** if E is **partially available** there:
  - E is evaluated along **at least one path** to P, with no operands redefined since.
- Partially redundant expression **can be eliminated** if we can **insert computations** to make it **fully redundant**.

# Loop Invariants are Partial Redundancies

- Loop invariant expression is partially redundant:



- As before, partially redundant computation can be eliminated if we insert computations to make it fully redundant.
- Remaining copies can be eliminated through copy propagation or more complex analysis of partially redundant assignments.

# Partial Redundancy Elimination (PRE)

- **The Method:**
  1. Insert Computations to make partially redundant expression(s) fully redundant.
  2. Eliminate redundant expression(s).
- **Issues [Outline of Lecture]:**
  1. What expression occurrences are candidates for elimination?
  2. Where can we safely insert computations?
  3. Where do we want to insert them?
- For this lecture, we assume one expression of interest,  $a+b$ .
  - In practice, with some restrictions, can do many expressions in parallel.

# Which Occurrences Might Be Eliminated?

- In CSE,
  - E is **available** at P if it is previously evaluated along **every** path to P, with no subsequent redefinitions of operands.
  - If so, we can eliminate computation at P.
- In PRE,
  - E is **partially available** at P if it is previously evaluated along **at least one** path to P, with no subsequent redefinitions of operands.
  - If so, we might be able to eliminate computation at P, if we can insert computations to make it fully redundant.
- Occurrences of E where E is **partially available** are candidates for elimination.

# Finding Partially Available Expressions

- Forward flow problem

- Lattice = { 0, 1 }, meet is union ( $\cup$ ), Top = 0 (not PAVAIL), entry = 0

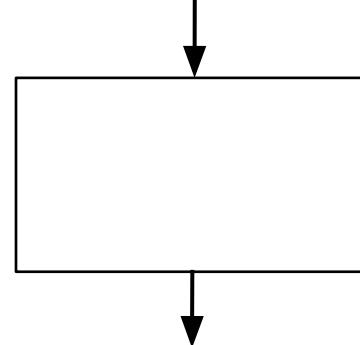
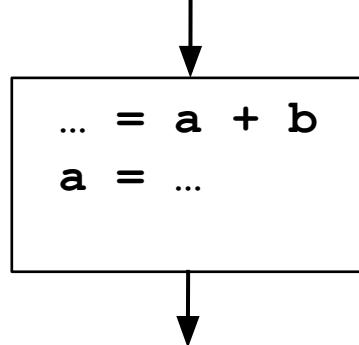
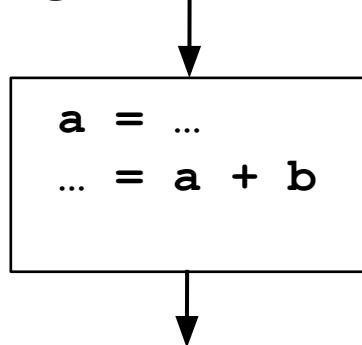
- $\text{PAVOUT}[i] = (\text{PAVIN}[i] - \text{KILL}[i]) \cup \text{AVLOC}[i]$

- $\text{PAVIN}[i] = \begin{cases} 0 & i = \text{entry} \\ \bigcup_{p \in \text{parent}(i)} \text{PAVOUT}[p] & \text{otherwise} \end{cases}$

- For a block: Expression is locally available (AVLOC)

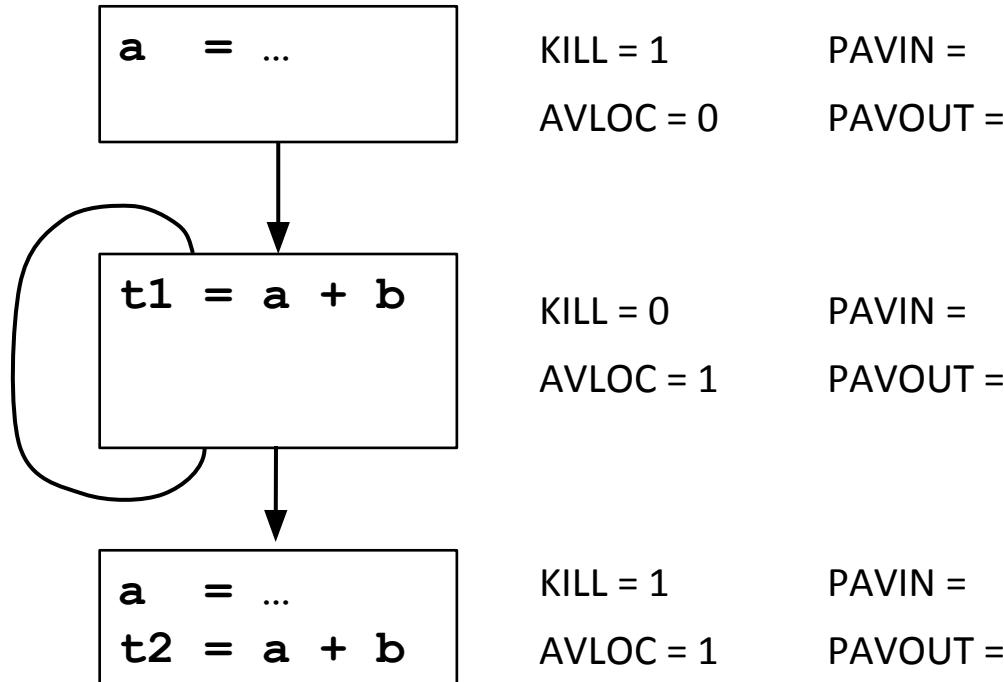
downwards exposed; Expression is killed (KILL) if any

assignments to operands.



# Partial Availability Example

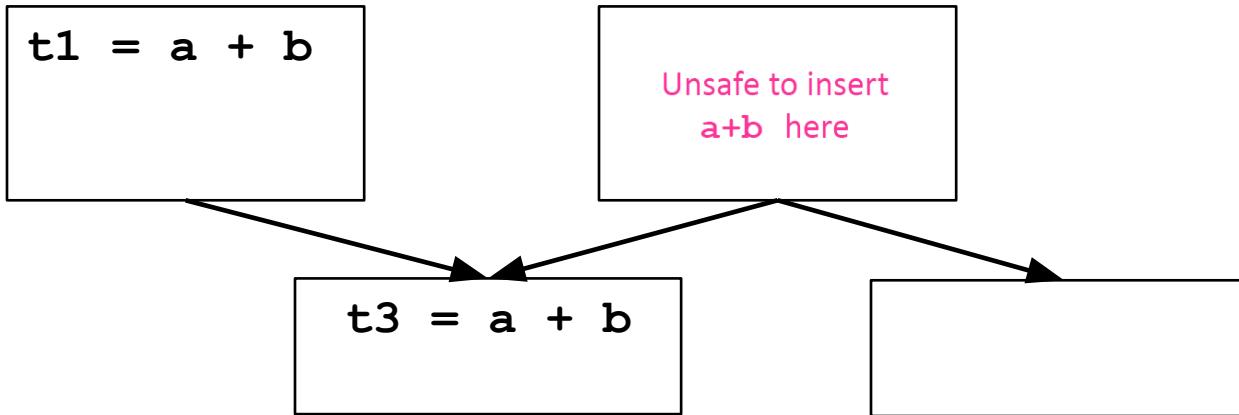
- For expression  $a+b$ .



- Occurrence in loop is partially redundant.

# Where Can We Insert Computations?

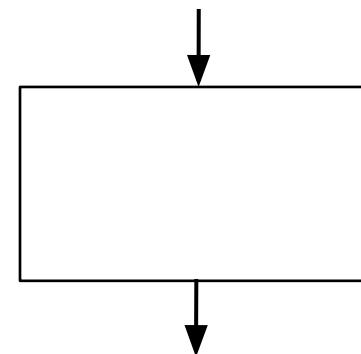
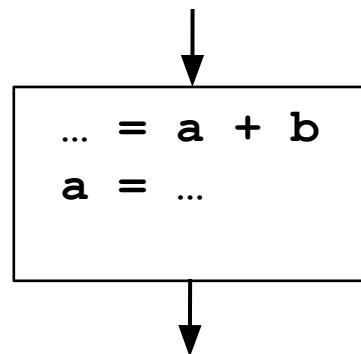
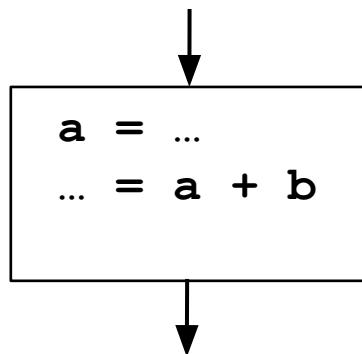
- **Safety:** never introduce a new expression along any path.



- Insertion could introduce exception, change program behavior.
- If we can add a new basic block, can insert safely in most cases.
- Solution: insert expression only where it is **anticipated**.
- **Performance:** never increase the # of computations on any path.
  - Under simple model, guarantees program won't get worse.
  - Reality: might increase register lifetimes, add copies, lose.

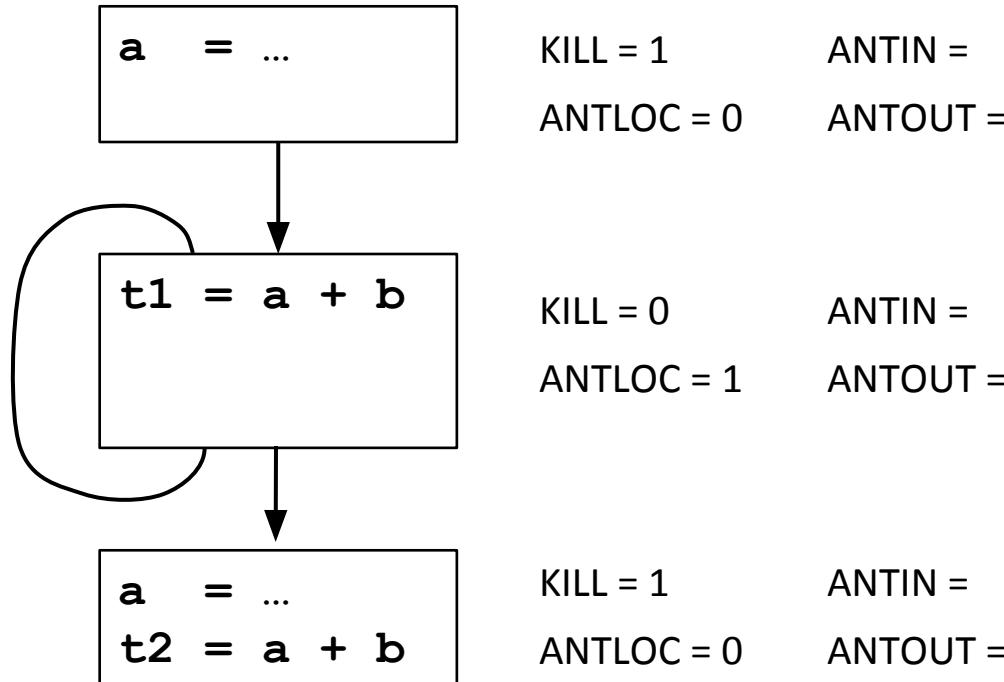
# Finding Anticipated Expressions

- Backward flow problem
  - Lattice = { 0, 1 }, meet is intersection ( $\cap$ ), top = 1 (ANT), exit = 0
    - $\text{ANTIN}[i] = \text{ANTLOC}[i] \cup (\text{ANTOUT}[i] - \text{KILL}[i])$
    - $\text{ANTOUT}[i] = \begin{cases} 0 & i = \text{exit} \\ \cap_{s \in \text{succ}(i)} \text{ANTIN}[s] & \text{otherwise} \end{cases}$
- For a block: Expression  $\text{locally anticipated (ANTLOC)}$  if upwards exposed.



# Anticipation Example

- For expression  $a+b$ .

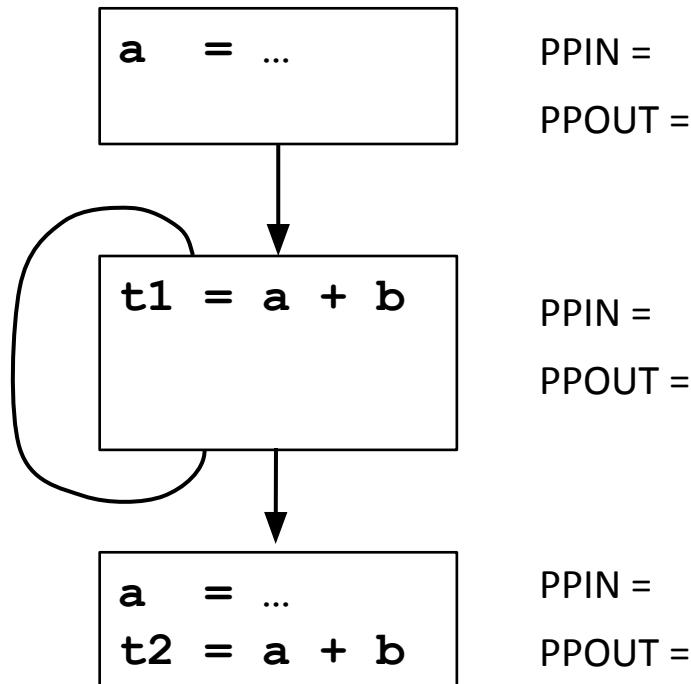


- Expression is anticipated at end of first block.
- Computation may be safely inserted there.

# Where Do We Want to Insert Computations?

- Morel-Renvoise and variants: “Placement Possible”
  - Dataflow analysis shows where to insert:
    - PPIN = “Placement possible at entry of block or before.”
    - PPOUT = “Placement possible at exit of block or before.”
  - Insert at earliest place where PP = 1.
  - Only place at end of blocks,
    - PPIN really means “Placement possible or not necessary in each predecessor block.”
  - Don’t need to insert where expression is already available.
    - $\text{INSERT}[i] = \text{PPOUT}[i] \cap (\neg \text{PPIN}[i] \cup \text{KILL}[i]) \cap \neg \text{AVOUT}[i]$
  - Remove (upwards-exposed) computations where PPIN=1.
    - $\text{DELETE}[i] = \text{PPIN}[i] \cap \text{ANTLOC}[i]$

# Where Do We Want to Insert?



PPIN =

PPOUT =

PPIN =

PPOUT =

PPIN =

PPOUT =

# Formulating the Problem

- **PPOUT:** we want to place at output of this block only if
  - we want to place at entry of all successors
- **PPIN:** we want to place at input of this block only if (all of):
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we can gain something by placing it here (**PAVIN**)
- **Forward or Backward?**
  - **BOTH!**
- **Problem is *bidirectional*, but lattice  $\{0, 1\}$  is finite, so**
  - as long as transfer functions are **monotone**, it converges.

# Computing “Placement Possible”

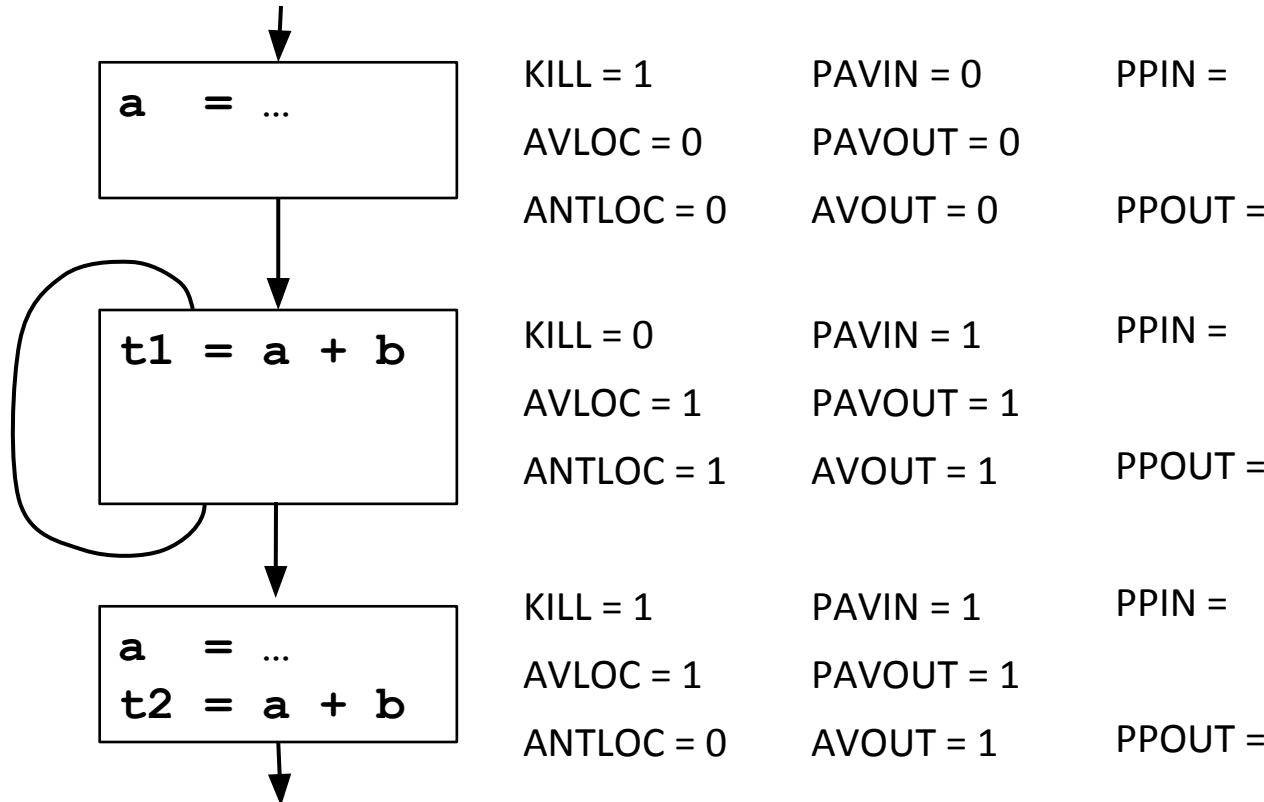
- **PPOUT:** we want to place at output of this block only if
  - we want to place at entry of all successors

$$\bullet \text{PPOUT}[i] = \begin{cases} 0 & i = \text{entry} \\ \cap_{\substack{s \in \\ \text{succ}(i)}} \text{PPIN}[s] & \text{otherwise} \end{cases}$$

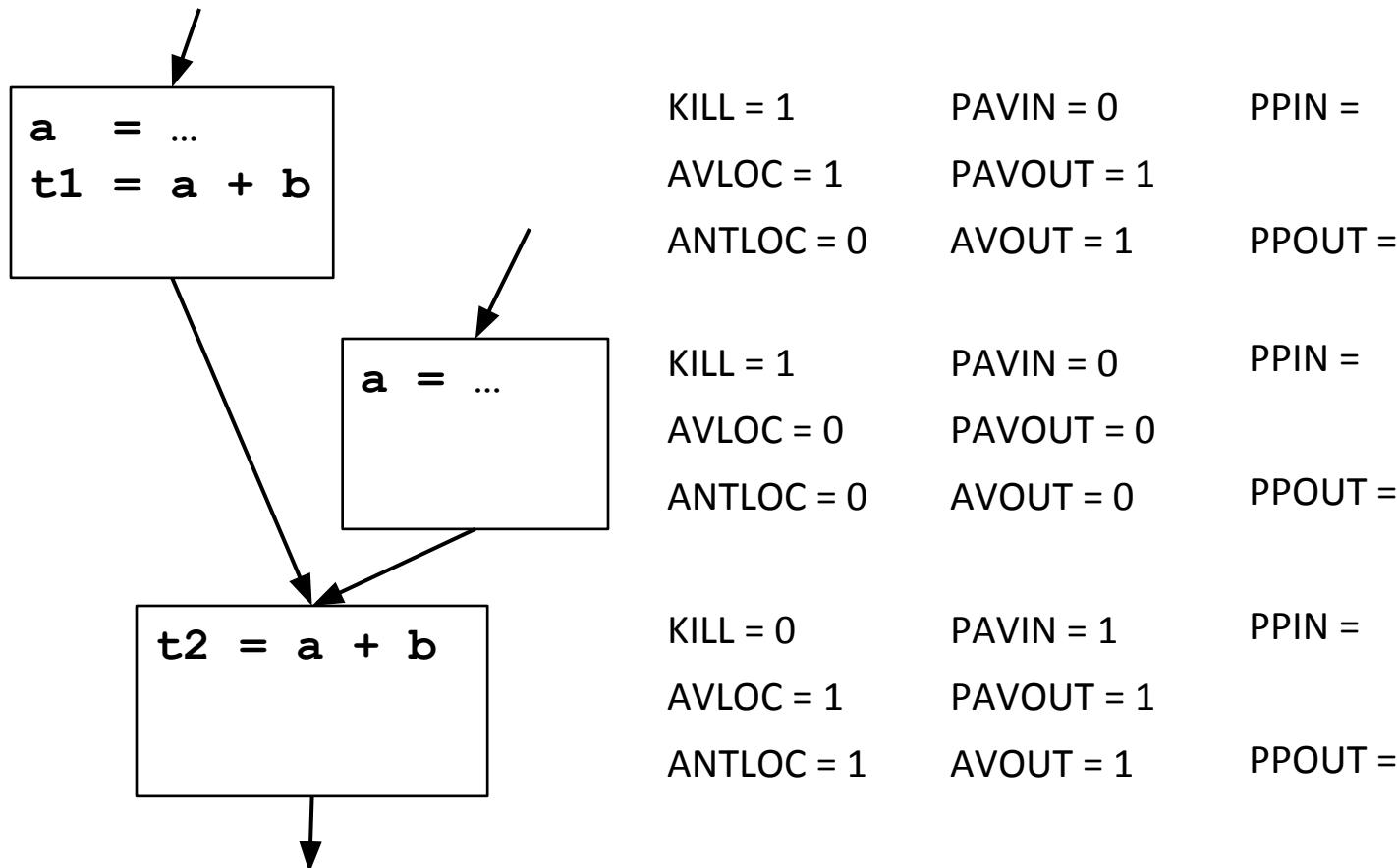
- **PPIN:** we want to place at start of this block only if (all of):
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don't insert at input)
  - we gain something by moving it up (PAVIN heuristic)

$$\bullet \text{PPIN}[i] = \begin{cases} 0 & i = \text{exit} \\ ([\text{ANTLOC}[i]] \cup (\text{PPOUT}[i] - \text{KILL}[i])) \\ \cap \bigcap_{\substack{p \in \\ \text{preds}(i)}} (\text{PPOUT}[p] \cup \text{AVOUT}[p]) & \text{otherwise} \\ \cap \text{PAVIN}[i] \end{cases}$$

# “Placement Possible” Example 1



# “Placement Possible” Example 2



# “Placement Possible” Correctness

- **Convergence** of analysis: transfer functions are monotone.
- **Safety**: Insert only if anticipated.

$$\text{PPIN}[i] \subseteq (\text{PPOUT}[i] - \text{KILL}[i]) \cup \text{ANTLOC}[i]$$

$$\text{PPOUT}[i] = \begin{cases} 0 & i = \text{exit} \\ \cap_{s \in \text{succ}(i)} \text{PPIN}[s] & \text{otherwise} \end{cases}$$

- **INSERT**  $\subseteq \text{PPOUT} \subseteq \text{ANTOUT}$ , so insertion is safe.
- **Performance**: never increase the # of computations on any path
  - **DELETE** =  $\text{PPIN} \cap \text{ANTLOC}$
  - On every path from an INSERT, there is a DELETE.
  - The number of computations on a path does not increase.

# CSC D70: Compiler Optimization LICM: Loop Invariant Code Motion

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Backup Slides

# CSC D70: Compiler Optimization Register Allocation

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Register Allocation and Coalescing

- Introduction
- Abstraction and the Problem
- Algorithm
- Spilling
- Coalescing

Reading: ALSU 8.8.4

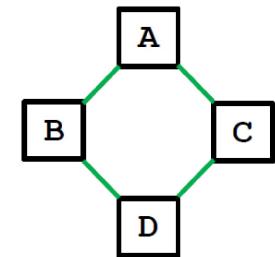
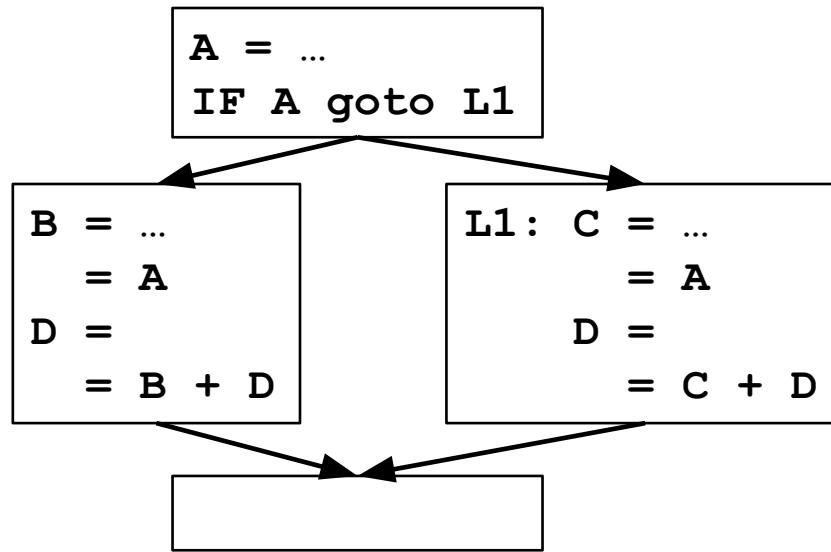
# Motivation

- **Problem**
  - Allocation of variables (pseudo-registers) to hardware registers in a procedure
- **A very important optimization!**
  - Directly reduces running time
    - (memory access → register access)
  - Useful for other optimizations
    - e.g. CSE assumes old values are kept in registers.

# Goals

- Find an allocation for all pseudo-registers, if possible.
- If there are not enough registers in the machine,  
choose registers to spill to memory

# Register Assignment Example



- Find an assignment (no spilling) with only 2 registers
  - A and D in one register, B and C in another one
- What assumptions?
  - After assignment, no use of A & (and only one of B and C used)

# An Abstraction for Allocation & Assignment

- **Intuitively**
  - Two pseudo-registers **interfere** if at some point in the program they cannot both occupy the same register.
- **Interference graph**: an **undirected graph**, where
  - **nodes** = pseudo-registers
  - there is an **edge** between two nodes if their corresponding pseudo-registers interfere
- **What is not represented**
  - Extent of the interference between uses of different variables
  - Where in the program is the interference

Interfere many times vs. once  
E.g., cold path vs. hot path

# Register Allocation and Coloring

- A graph is **n-colorable** if:
  - every node in the graph can be colored with one of the n colors such that two adjacent nodes do not have the same color.
- Assigning n register (without spilling) = Coloring with n colors
  - assign a node to a register (color) such that no two adjacent nodes are assigned same registers (colors)
- Is spilling necessary? = Is the graph n-colorable?
- To determine if a graph is n-colorable is **NP-complete**, for  $n > 2$ 
  - Too expensive
  - Heuristics

# Algorithm

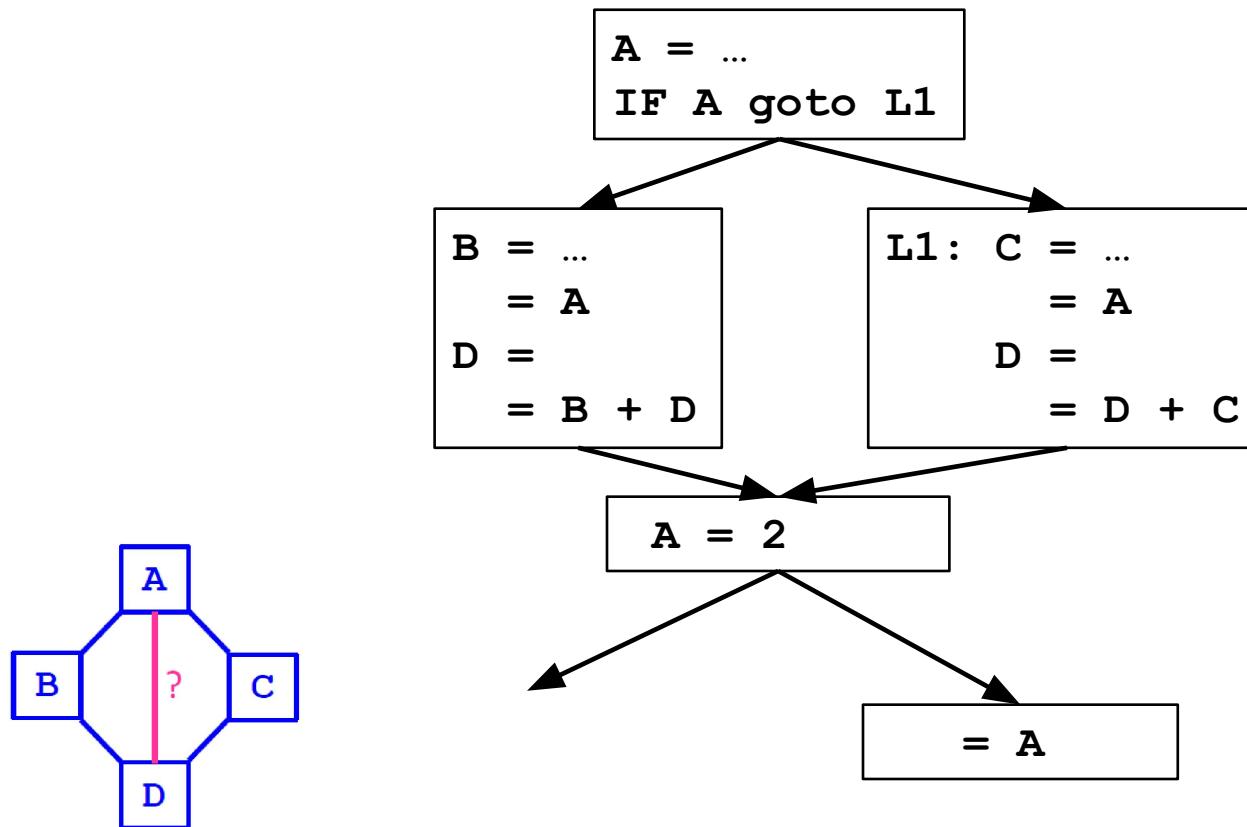
## Step 1. Build an interference graph

- a. refining notion of a node
- b. finding the edges

## Step 2. Coloring

- use heuristics to try to find an n-coloring
  - Success:
    - colorable and we have an assignment
  - Failure:
    - graph not colorable, or
    - graph is colorable, but it is too expensive to color

# Step 1a. Nodes in an Interference Graph



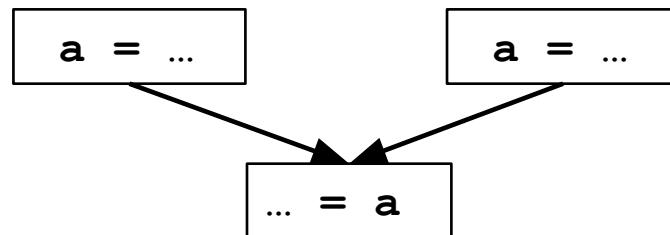
Interference Graph

Should we add A-D edge?

No, since new def of A

# Live Ranges and Merged Live Ranges

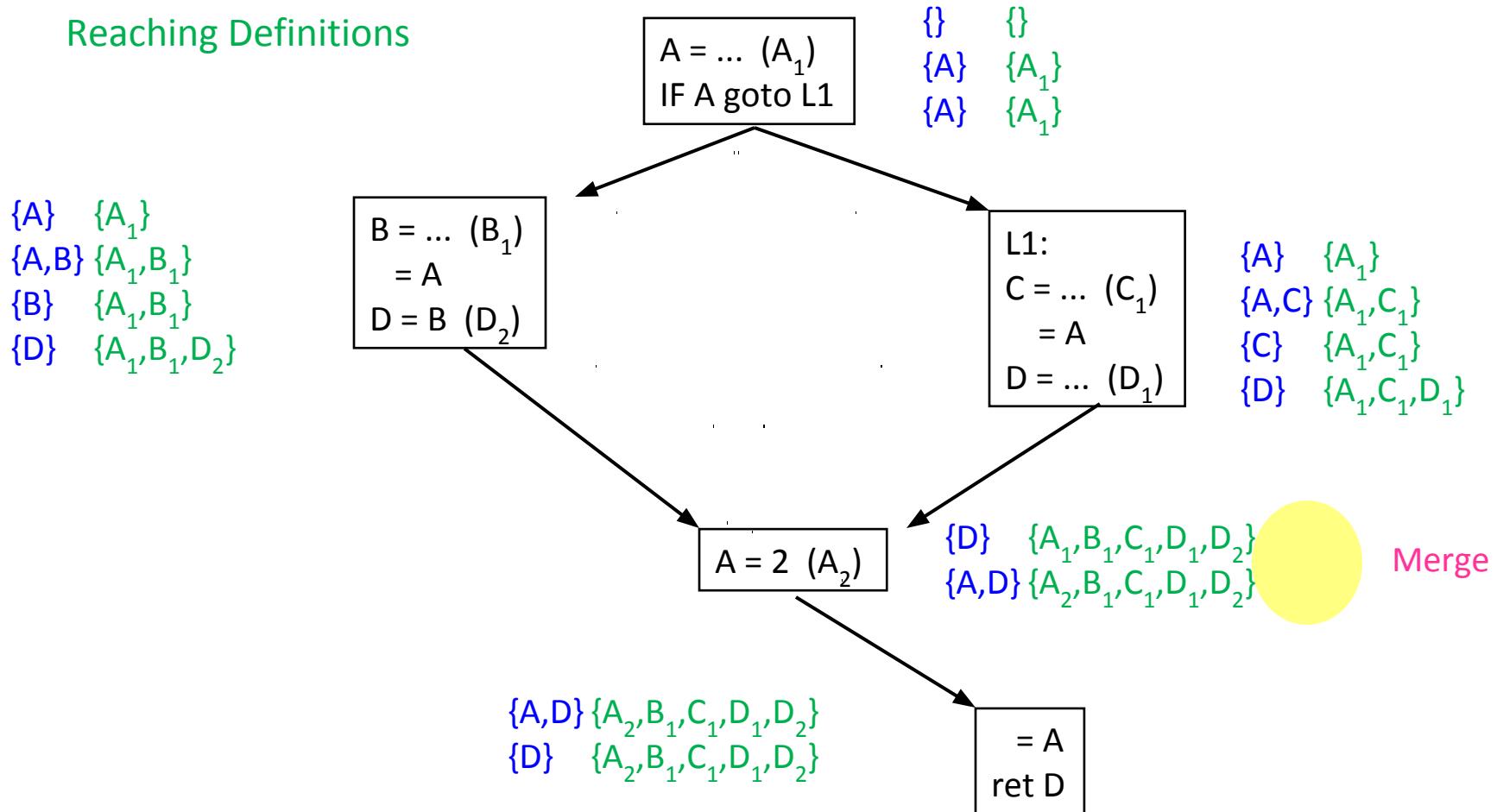
- Motivation: to create an interference graph that is easier to color
  - Eliminate interference in a variable's "dead" zones.
  - Increase flexibility in allocation:
    - can allocate same variable to different registers
- A **live range** consists of a definition and all the points in a program in which that definition is live.
  - How to compute a live range?
- Two overlapping live ranges for the **same** variable must be merged



# Example (Revisited)

Live Variables

Reaching Definitions



# Merging Live Ranges

- **Merging definitions into equivalence classes**
  - Start by putting each definition in a different equivalence class
  - Then, **for each point** in a program:
    - if (i) **variable is live**, and (ii) there are **multiple reaching definitions for the variable**, then:
      - merge the equivalence classes of all such definitions into one equivalence class
    - *(Sound familiar?)*
- **From now on, refer to merged live ranges simply as live ranges**
  - merged live ranges are also known as “**webs**”

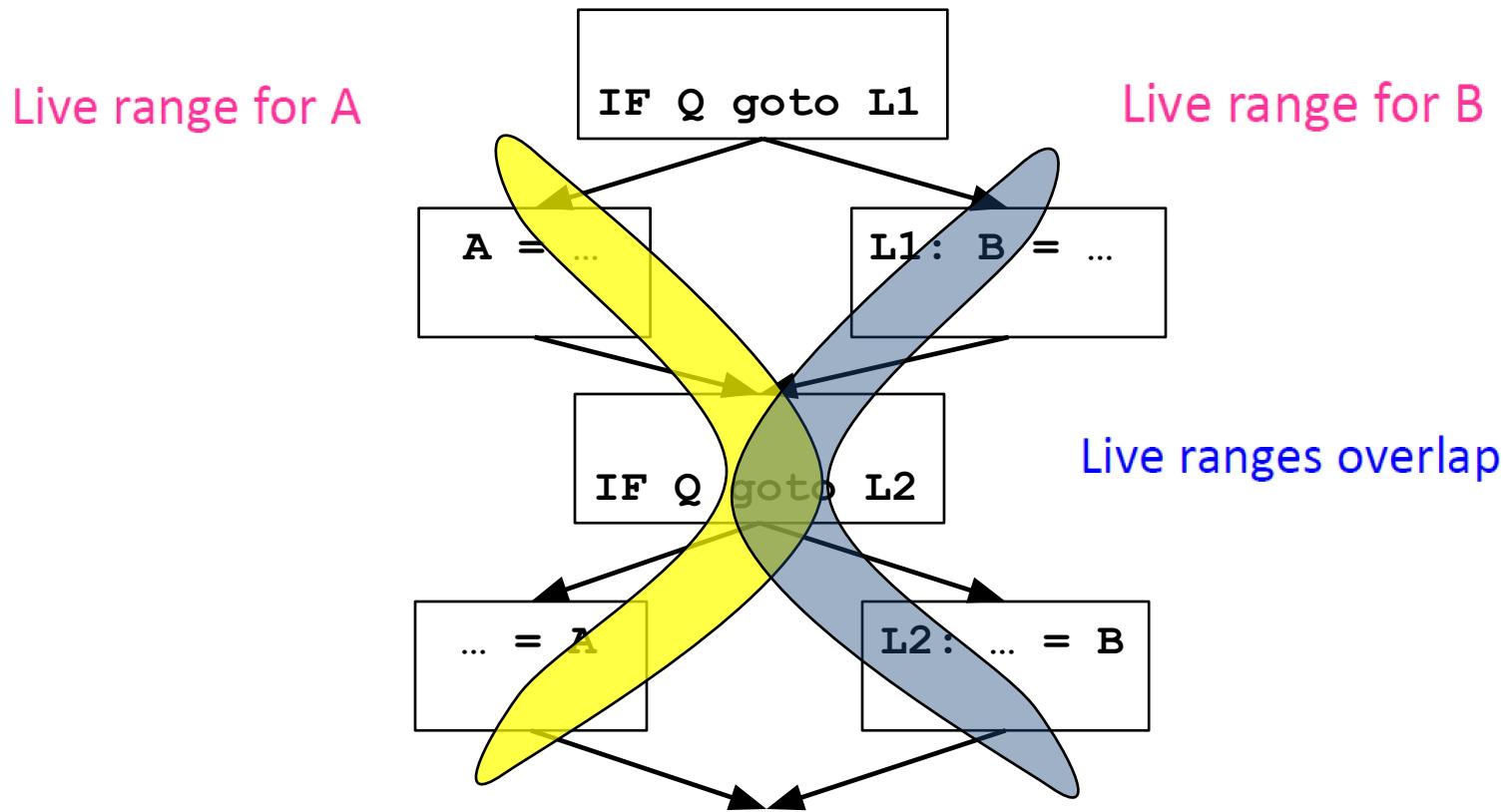
# SSA Revisited: What Happens to $\Phi$ Functions

- Now we see why it is unnecessary to “implement” a  $\Phi$  function
  - $\Phi$  functions and SSA variable renaming simply turn into merged live ranges
- When you encounter:  $\textcolor{blue}{x_4} = \Phi(x_1, x_2, x_3)$ 
  - merge  $x_1, x_2, x_3$ , and  $x_4$  into the same live range
  - delete the  $\Phi$  function
- Now you have effectively converted back out of SSA form

# Step 1b. Edges of Interference Graph

- **Intuitively:**
  - Two live ranges (necessarily of different variables) may **interfere if they overlap at some point in the program.**
  - Algorithm:
    - At each point in the program:
      - enter an **edge** for every pair of live ranges at that point.
- **An optimized definition & algorithm for edges:**
  - Algorithm:
    - check for interference only at the start of each live range
  - Faster
  - Better quality

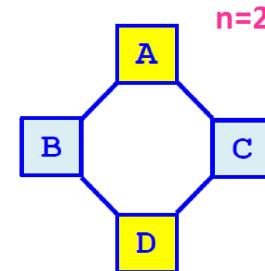
# Live Range Example 2



Because ranges overlap: Won't assign A and B to same register  
(even though would have been ok: path sensitive vs. path insensitive analysis)

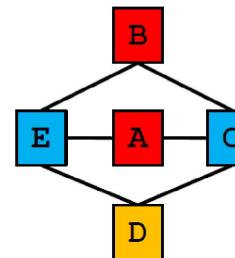
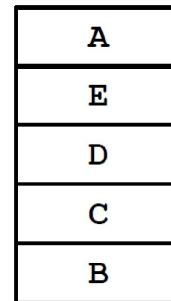
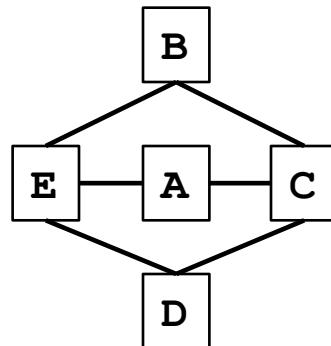
# Step 2. Coloring

- Reminder: **coloring for  $n > 2$  is NP-complete**
- Observations:
  - a node with **degree  $< n$**  ⇒
    - can always color it successfully, given its neighbors' colors
  - a node with **degree  $= n$**  ⇒
    - can only color if at least two neighbors share same color
  - a node with **degree  $> n$**  ⇒
    - maybe, not always



# Coloring Algorithm

- Algorithm:
  - Iterate until stuck or done
    - Pick any node with degree  $< n$
    - Remove the node and its edges from the graph
  - If done (no nodes left)
    - reverse process and add colors
- Example ( $n = 3$ ):



- Note: degree of a node may drop in iteration
- Avoids making arbitrary decisions that make coloring fail

# More details

- **Apply coloring heuristic**

Build interference graph

Iterate until there are no nodes left

If there exists a node  $v$  with less than  $n$  neighbor

push  $v$  on register allocation stack

else

return (coloring heuristics fail)

remove  $v$  and its edges from graph

- **Assign registers**

While stack is not empty

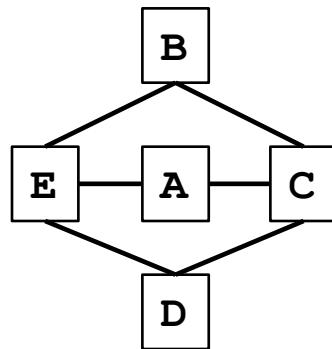
Pop  $v$  from stack

Reinsert  $v$  and its edges into the graph

Assign  $v$  a color that differs from all its neighbors

# What Does Coloring Accomplish?

- **Done:**
  - colorable, also obtained an assignment
- **Stuck:**
  - colorable or not?



# Extending Coloring: Design Principles

- A **pseudo-register** is
  - Colored successfully: allocated a hardware register
  - Not colored: left in memory
- **Objective function**
  - Cost of an uncolored node:
    - proportional to number of uses/definitions (dynamically)
    - estimate by its loop nesting
  - Objective: minimize sum of cost of uncolored nodes
- **Heuristics**
  - Benefit of spilling a pseudo-register:
    - increases colorability of pseudo-registers it interferes with
    - can approximate by its degree in interference graph
  - Greedy heuristic
    - spill the pseudo-register with lowest cost-to-benefit ratio, whenever spilling is necessary

# Spilling to Memory

- CISC architectures
  - can operate on data in memory directly
  - memory operations are slower than register operations
- RISC architectures
  - machine instructions can only apply to registers
  - Use
    - must first load data from memory to a register before use
  - Definition
    - must first compute RHS in a register
    - store to memory afterwards
  - Even if spilled to memory, needs a register at time of use/definition

# Chaitin: Coloring and Spilling

- **Identify spilling**

- Build interference graph

- Iterate until there are no nodes left

- If there exists a node  $v$  with less than  $n$  neighbor

- place  $v$  on stack to register allocate

- else

- $v = \text{node with highest degree-to-cost ratio}$

- mark  $v$  as spilled

- remove  $v$  and its edges from graph

- **Spilling may require use of registers; change interference graph**

- While there is spilling

- rebuild interference graph and perform step above

- **Assign registers**

- While stack is not empty

- Remove  $v$  from stack

- Reinsert  $v$  and its edges into the graph

- Assign  $v$  a color that differs from all its neighbors

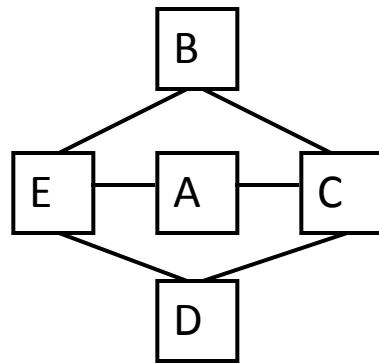
# Spilling

- What should we spill?
  - Something that will eliminate a lot of interference edges
  - Something that is used infrequently
  - Maybe something that is live across a lot of calls?
- One Heuristic:
  - spill cheapest live range (aka “web”)
  - Cost =  $[(\# \text{ defs} \& \text{ uses}) * 10^{\text{loop-nest-depth}}] / \text{degree}$

# Quality of Chaitin's Algorithm

- Giving up too quickly

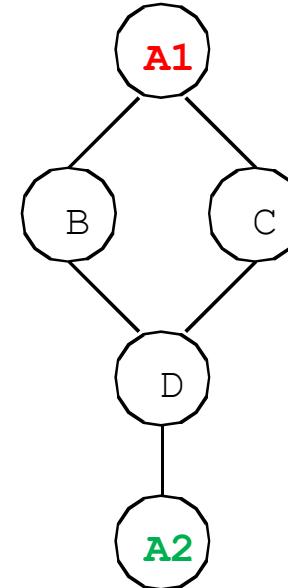
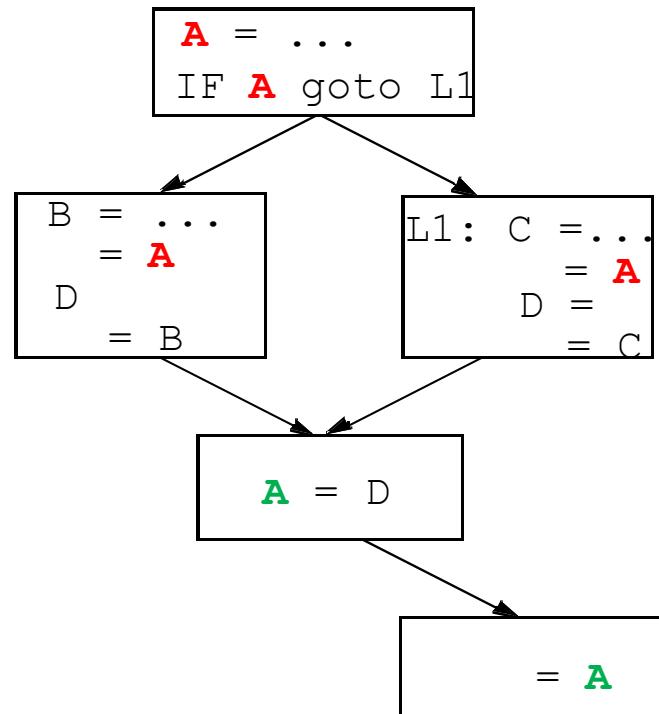
- $N=2$



- An optimization: “Prioritize the coloring”
  - Still eliminate a node and its edges from graph
  - Do not commit to “spilling” just yet
  - Try to color again in assignment phase.

# Splitting Live Ranges

- Recall: Split pseudo-registers into live ranges to create an interference graph that is easier to color
  - Eliminate interference in a variable's “dead” zones.
  - Increase flexibility in allocation:
    - can allocate same variable to different registers



# Insight

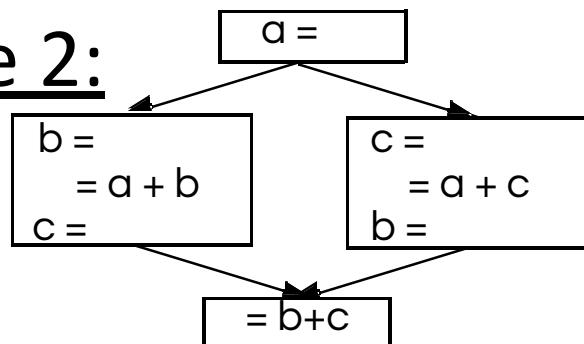
- Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color
  - Eliminate interference in a variable's “nearly dead” zones.
    - Cost: Memory loads and stores
      - Load and store at boundaries of regions with no activity
    - # active live ranges at a program point can be  $>$  # registers
  - Can allocate same variable to different registers
    - Cost: Register operations
      - a register copy between regions of different assignments
    - # active live ranges cannot be  $>$  # registers

# Examples

## Example 1:

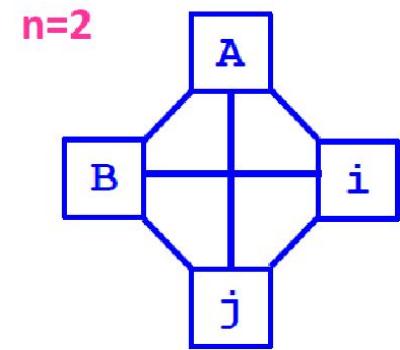
```
FOR i = 0 TO 10
FOR j = 0 TO 10000
    A = A + ...
    (does not use B)
FOR j = 0 TO 10000
    B = B + ...
    (does not use A)
```

## Example 2:

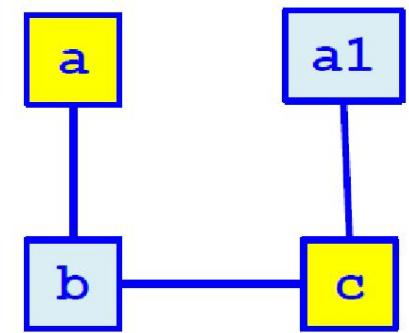
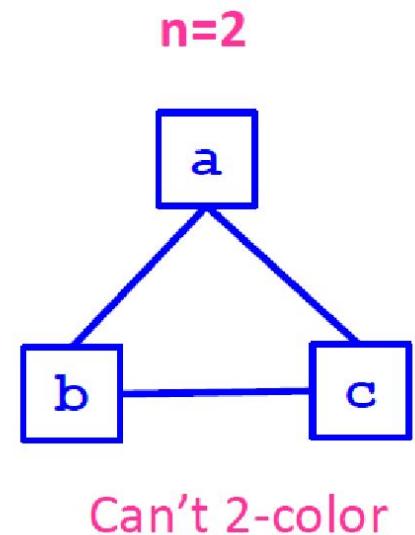
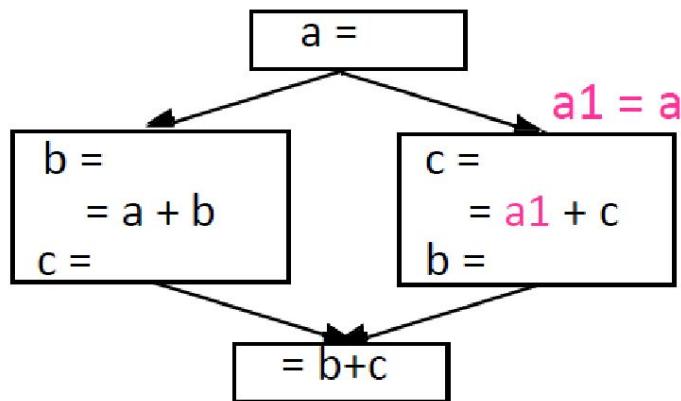
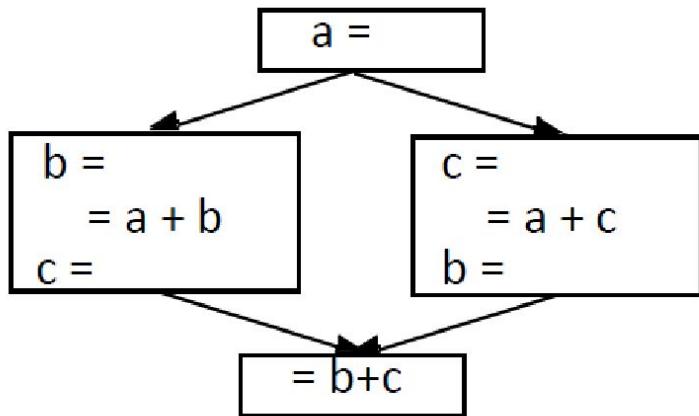


# Example 1

```
FOR i = 0 TO 10
spill i
    FOR j = 0 TO 10000
        spill B
            A = A + ...
            (does not use B)
        spill A
            B = B + ...
            (does not use A)
```



# Example 2



Can 2-color  
("a" gets 2 regs)

# Live Range Splitting

- When do we apply live range splitting?
- Which live range to split?
- Where should the live range be split?
- How to apply live-range splitting with coloring?
  - Advantage of coloring:
    - defers arbitrary assignment decisions until later
  - When coloring fails to proceed, may not need to split live range
    - degree of a node  $\geq n$  does not mean that the graph definitely is not colorable
  - Interference graph does not capture positions of a live range

# One Algorithm

- Observation: spilling is absolutely necessary if
  - number of live ranges active at a program point  $> n$
- Apply live-range splitting before coloring
  - Identify a point where number of live ranges  $> n$
  - For each live range active around that point:
    - find the outermost “block construct” that does not access the variable
  - Choose a live range with the largest inactive region
  - Split the inactive region from the live range

# Summary

- **Problems:**
  - Given  $n$  registers in a machine, is spilling avoided?
  - Find an assignment for all pseudo-registers, whenever possible.
- **Solution:**
  - Abstraction: an **interference graph**
    - nodes: **live ranges**
    - edges: presence of live range at time of definition
  - Register Allocation and Assignment problems
    - equivalent to  **$n$ -colorability** of interference graph  
→ NP-complete
  - Heuristics to find an assignment for  $n$  colors
    - **successful**: colorable, and finds **assignment**
    - **not successful**: colorability unknown & **no assignment**

# CSC D70: Compiler Optimization Register Coalescing

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Let's Focus on Copy Instructions

```
X = A + B;  
...  
Y = X;  
...  
Z = Y + 4;
```



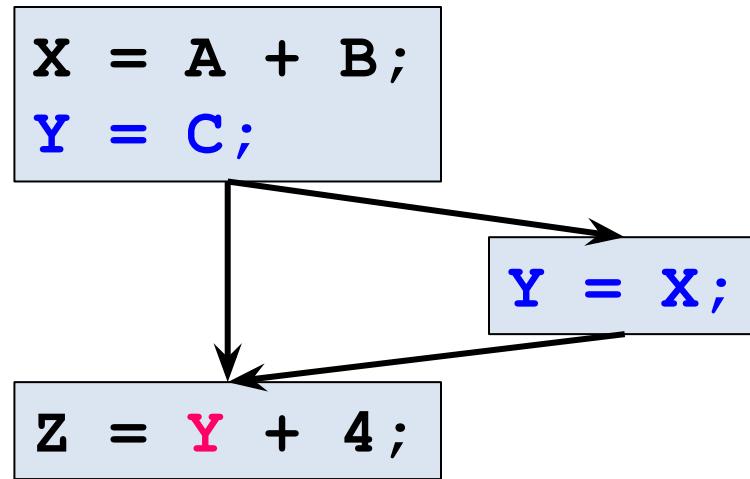
```
X = A + B;  
...  
// deleted  
...  
Z = X + 4;
```

2. Dead Code Elimination

1. Copy Propagation

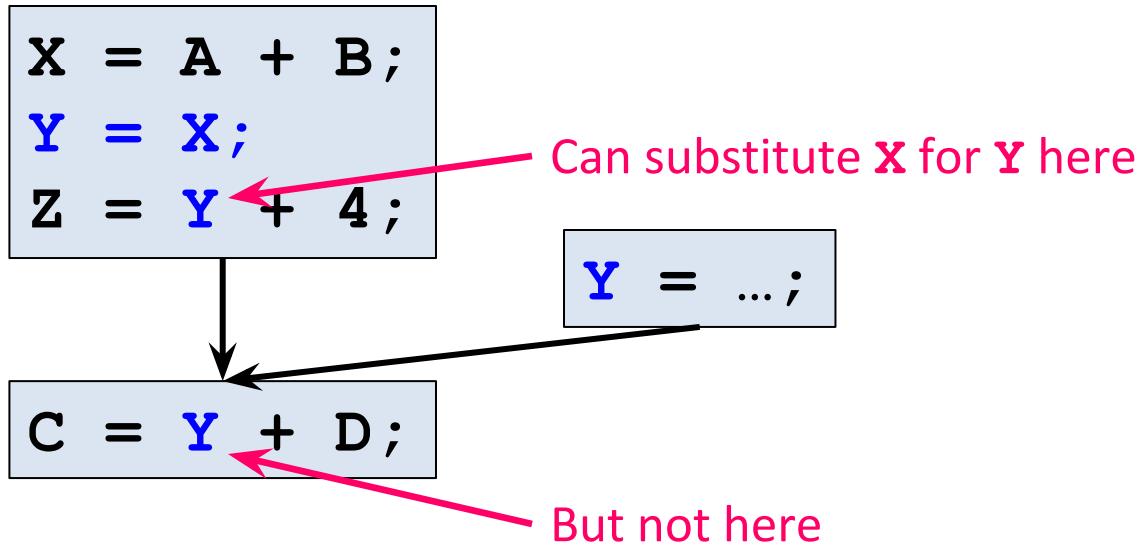
- Optimizations that help optimize away copy instructions:
  - Copy Propagation
  - Dead Code Elimination
- Can all copy instructions be eliminated using this pair of optimizations?

# Example Where Copy Propagation Fails



- Use of copy target has multiple (conflicting) reaching definitions

# Another Example Where the Copy Instruction Remains



- Copy target ( $Y$ ) still live even after some successful copy propagations
- Bottom line:
  - copy instructions may still exist when we perform register allocation

# Copy Instructions and Register Allocation

- What clever thing might the register allocator do for copy instructions?

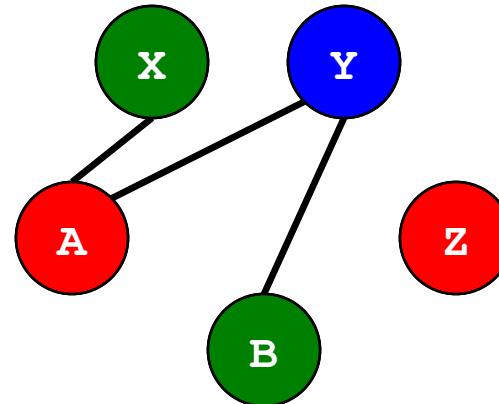
```
...  
Y = X;  
...
```

```
...  
r7 = r7;  
...
```

- If we can assign both the **source** and **target** of the copy to the **same register**:
  - then we don't need to perform the copy instruction at all!
  - **the copy instruction can be removed from the code**
    - even though the optimizer was unable to do this earlier
- One way to do this:
  - treat the copy **source** and **target** as the **same node in the interference graph**
    - then the coloring algorithm will naturally assign them to the same register
  - this is called "**coalescing**"

# Simple Example: Without Coalescing

```
X = ...;  
A = 5;  
Y = X;  
B = A + 2;  
Z = Y + B;  
return Z;
```

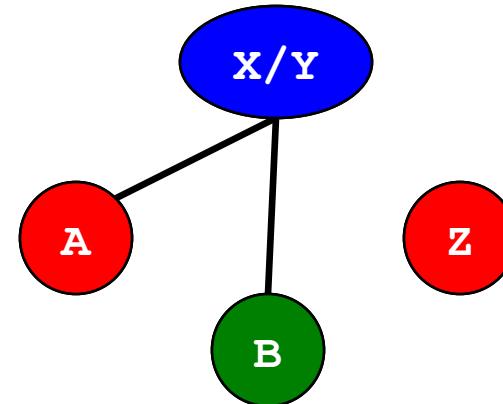


Valid coloring with 3 registers

- Without coalescing, X and Y can end up in different registers
  - cannot eliminate the copy instruction

# Example Revisited: With Coalescing

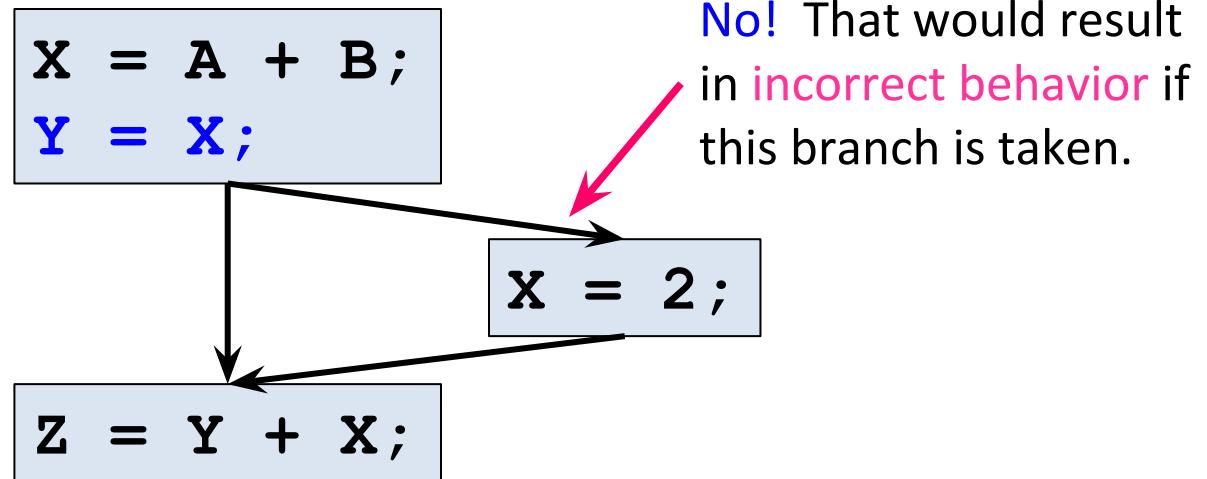
```
X = ...;  
A = 5;  
Y = X;  
B = A + 2;  
Z = Y + B;  
return Z;
```



Valid coloring with 3 registers

- With coalescing, X and Y are now guaranteed to end up in the same register
  - the copy instruction can now be eliminated
- Great! So should we go ahead and do this for every copy instruction?

# Should We Coalesce X and Y In This Case?



- It is **legal** to coalesce **X** and **Y** for a “**Y = X**” copy instruction iff:
  - initial definition of **Y**’s live range is this copy instruction, AND
  - the **live ranges of X and Y do not interfere otherwise**
- But just because it is legal doesn’t mean that it is a good idea...

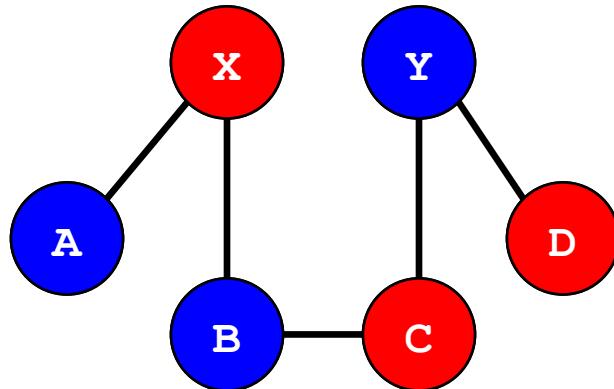
# Why Coalescing May Be Undesirable

```
X = A + B;  
... // 100 instructions  
Y = X;  
... // 100 instructions  
Z = Y + 4;
```

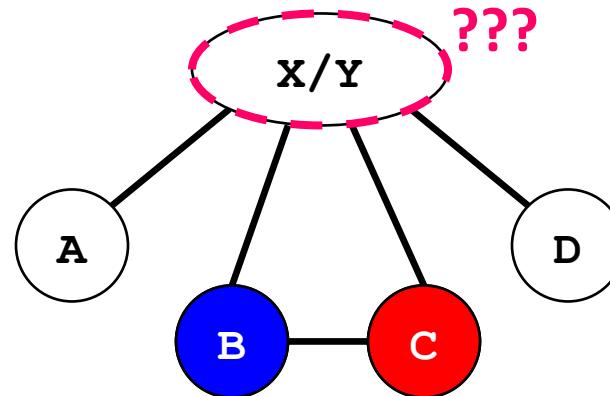
- What is the likely impact of coalescing **X** and **Y** on:
  - live range size(s)?
    - recall our discussion of live range splitting
  - colorability of the interference graph?
- Fundamentally, coalescing adds further constraints to the coloring problem
  - doesn't make coloring easier; may make it more difficult
- If we coalesce in this case, we may:
  - save a copy instruction, BUT
  - cause significant spilling overhead if we can no longer color the graph

# When to Coalesce

- Goal when coalescing is legal:
  - coalesce *unless* it would make a colorable graph **non-colorable**
- The bad news:
  - predicting colorability is tricky!
    - it depends on the shape of the graph
    - graph coloring is NP-hard
- Example: assuming **2 registers**, should we **coalesce X and Y**?



2-colorable

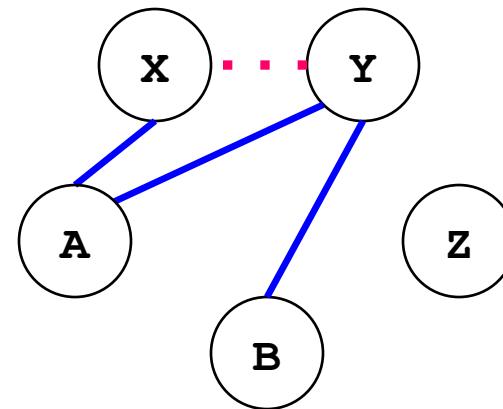


Not 2-colorable

# Representing Coalescing Candidates in the Interference Graph

- To decide whether to coalesce, we augment the interference graph
- Coalescing candidates are represented by a **new type of interference graph edge**:
  - **dotted lines**: coalescing candidates
    - try to assign vertices the **same color**
      - (unless that is problematic, in which case they can be given different colors)
  - **solid lines**: interference
    - vertices *must* be assigned **different colors**

```
X = ...;  
A = 5;  
Y = X;  
B = A + 2;  
Z = Y + B;  
return Z;
```

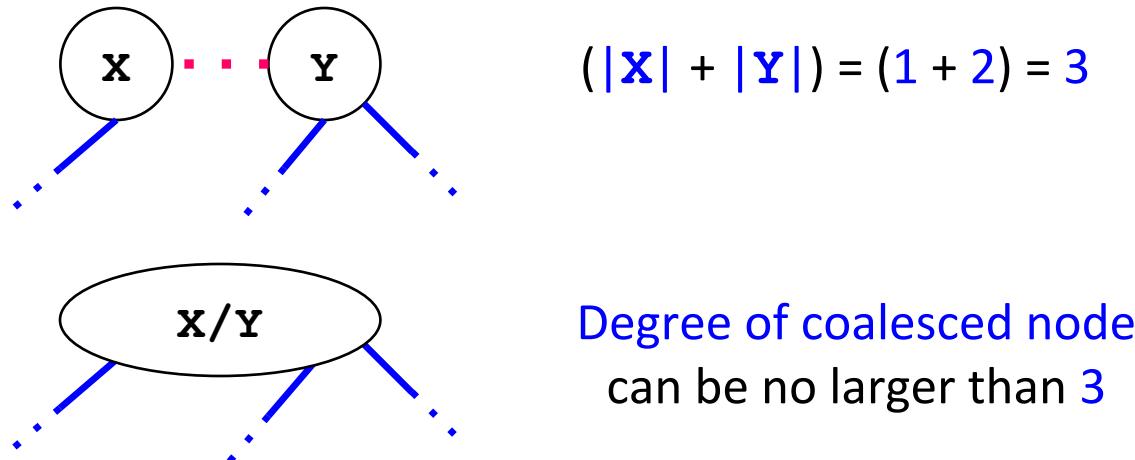


# How Do We Know When Coalescing Will Not Cause Spilling?

- Key insight:
  - Recall from the coloring algorithm:
    - we can always successfully  $N$ -color a node if its  $\text{degree} < N$
- To ensure that coalescing does not cause spilling:
  - check that the  $\text{degree} < N$  invariant is still locally preserved after coalescing
    - if so, then coalescing won't cause the graph to become non-colorable
  - no need to inspect the entire interference graph, or do trial-and-error
- Note:
  - We do NOT need to determine whether the full graph is colorable or not
  - Just need to check that coalescing does not cause a colorable graph to become non-colorable

# Simple and Safe Coalescing Algorithm

- We can safely coalesce nodes **X** and **Y** if  $(|X| + |Y|) < N$ 
  - Note:  $|x|$  = degree of node **X** counting interference (not coalescing) edges
- Example:

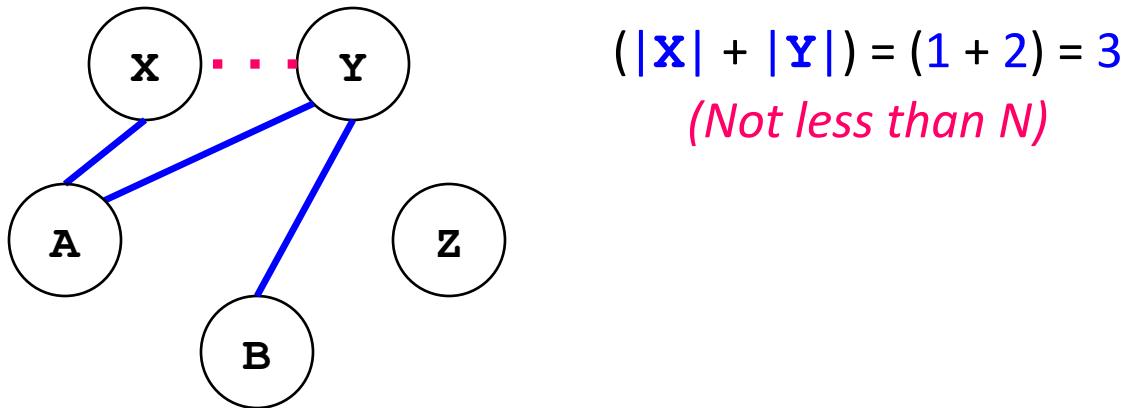


- if  $N \geq 4$ , it would always be safe to coalesce these two nodes
  - this cannot cause new spilling that would not have occurred with the original graph
- if  $N < 4$ , it is unclear

*How can we (safely) be more aggressive than this?*

# What About This Example?

- Assume  $N = 3$
- Is it safe to coalesce **X** and **Y**?

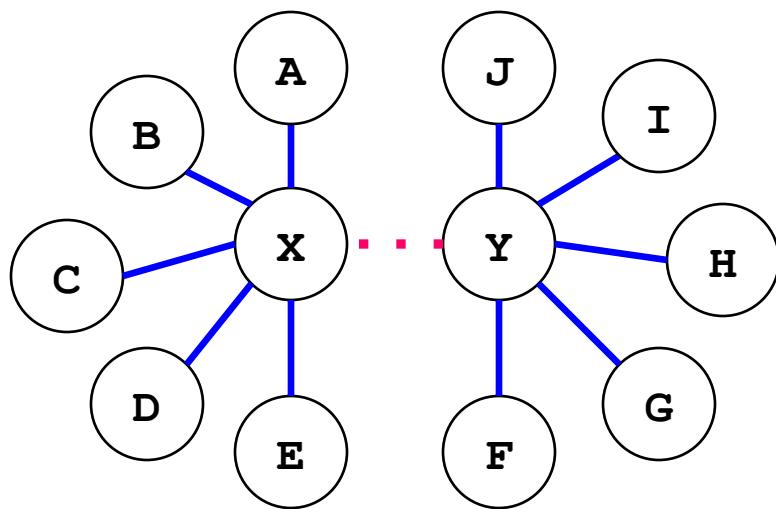


- Notice: **X** and **Y** share a common (interference) neighbor: node **A**
  - hence the degree of the coalesced **X/Y** node is actually 2 (not 3)
  - therefore coalescing **X** and **Y** is guaranteed to be safe when  $N = 3$
- How can we adjust the algorithm to capture this?

# Another Helpful Insight

- Colors are not assigned until nodes are popped off the stack
  - nodes with degree  $< N$  are pushed on the stack first
  - when a node is popped off the stack, we know that it can be colored
    - because the number of potentially conflicting neighbors must be  $< N$
- Spilling only occurs if there is no node with degree  $< N$  to push on the stack
- Example: ( $N=2$ )

# Another Helpful Insight



$$|X| = 5$$
$$|Y| = 5$$

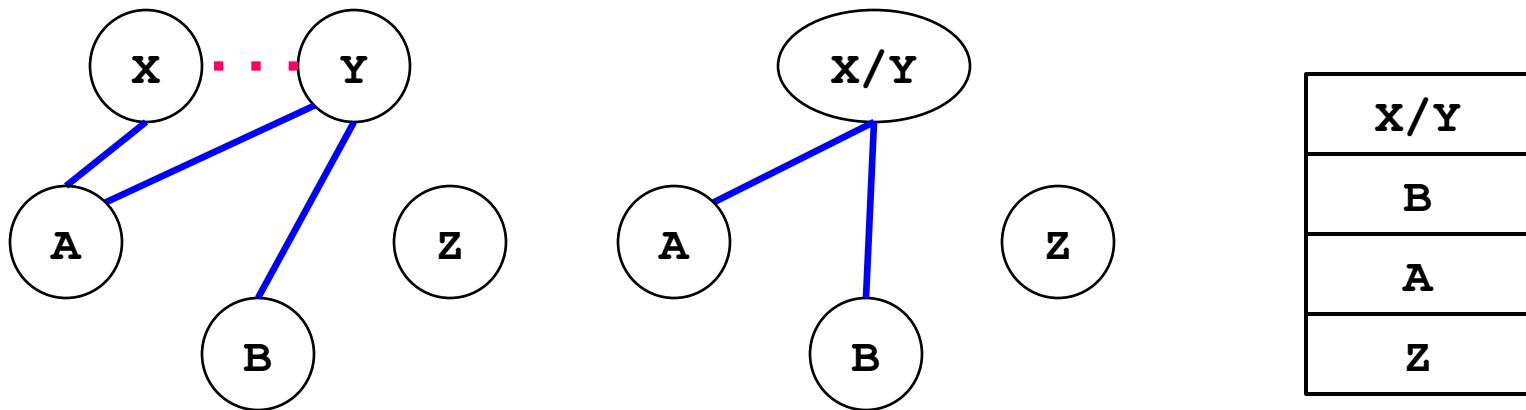
2-colorable after  
coalescing **X** and **Y**?

# Building on This Insight

- When would coalescing cause the stack pushing (aka “simplification”) to get stuck?
  1. coalesced node must have a degree  $\geq N$ 
    - otherwise, it can be pushed on the stack, and we are not stuck
  2. AND it must have at least  $N$  neighbors that each have a degree  $\geq N$ 
    - otherwise, all neighbors with degree  $< N$  can be pushed before this node
      - reducing this node’s degree below  $N$  (and therefore we aren’t stuck)
- To coalesce more aggressively (and safely), let’s exploit this second requirement
  - which involves looking at the degree of a coalescing candidate’s neighbors
    - not just the degree of the coalescing candidates themselves

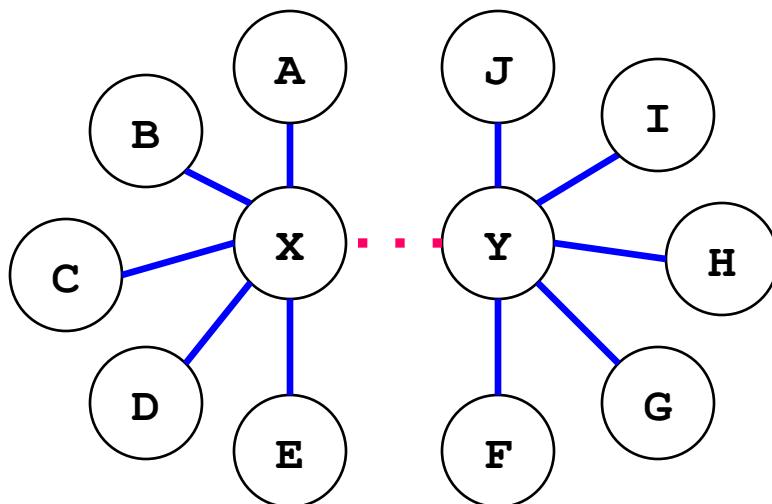
# Briggs's Algorithm

- Nodes **X** and **Y** can be coalesced if:
  - (number of neighbors of **X/Y** with degree  $\geq N$ )  $< N$
- Works because:
  - all other neighbors can be pushed on the stack before this node,
  - and then its degree is  $< N$ , so then it can be pushed
  - Example: ( $N = 2$ )



# Briggs's Algorithm

- Nodes **X** and **Y** can be coalesced if:
  - (number of neighbors of **X/Y** with degree  $\geq N$ )  $< N$
- More extreme example: (N = 2)

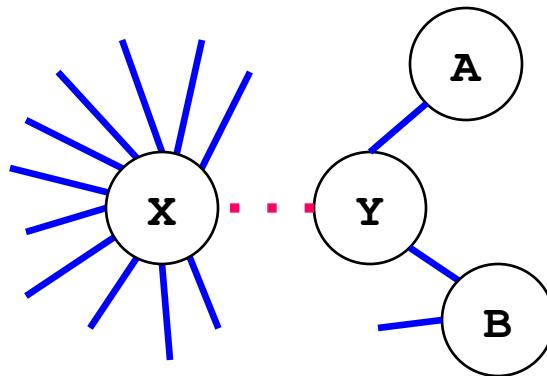


x/Y
J
I
H
G
F
E
D
C
B
A

# George's Algorithm

## Motivation:

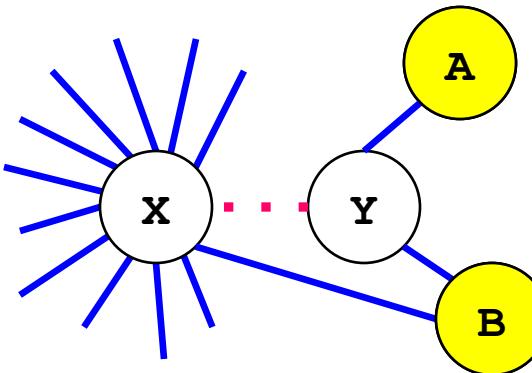
- imagine that **X** has a very high degree, but **Y** has a much smaller degree
  - (perhaps because **X** has a large live range)



- With Briggs's algorithm, we would inspect all neighbors both **X** and **Y**
  - but **X** has a lot of neighbors!
- Can we get away with just inspecting the neighbors of **Y**?
  - showing that coalescing makes coloring no worse than it was given **X**?

# George's Algorithm

- Coalescing **X** and **Y** does no harm if:
  - foreach neighbor **T** of **Y**, either:
    1. degree of **T** is  $< N$ , or       $\leftarrow$  similar to Briggs: **T** will be pushed before **X/Y**
    2. **T** interferes with **X**       $\leftarrow$  hence no change compared with coloring **X**
- Example: ( $N=2$ )



# Summary

- *Coalescing* can enable register allocation to **eliminate copy instructions**
  - if both source and target of copy can be allocated to the same register
- However, coalescing must be applied with care to **avoid causing register spilling**
- Augment the interference graph:
  - dotted lines for coalescing candidate edges
  - try to allocate to same register, unless this may cause spilling
- Coalescing Algorithms:
  - simply based upon **degree of coalescing candidate nodes ( $\mathbf{x}$  and  $\mathbf{y}$ )**
  - **Briggs's algorithm**
    - look at **degree of neighboring nodes of  $\mathbf{x}$  and  $\mathbf{y}$**
  - **George's algorithm**
    - asymmetrical: **look at neighbors of  $\mathbf{y}$  (degree and interference with  $\mathbf{x}$ )**

# CSC D70: Compiler Optimization Register Allocation & Coalescing

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# CSC D70: Compiler Optimization Pointer Analysis

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry, Greg Steffan, and Phillip Gibbons*

# Outline

- Basics
- Design Options
- Pointer Analysis Algorithms
- Pointer Analysis Using BDDs
- Probabilistic Pointer Analysis

# Pros and Cons of Pointers

- Many procedural languages have pointers
  - e.g., C or C++: `int *p = &x;`
- Pointers are powerful and convenient
  - can build arbitrary data structures
- Pointers can also hinder compiler optimization
  - hard to know where pointers are pointing
  - must be conservative in their presence
- Has inspired much research
  - analyses to decide where pointers are pointing
  - many options and trade-offs
  - open problem: a scalable accurate analysis

# Pointer Analysis Basics: Aliases

- Two variables are **aliases** if:
  - they **reference the same memory location**
- More useful:
  - prove variables reference different location

```
int x,y;  
int *p = &x;  
int *q = &y;  
int *r = p;  
int **s = &q;
```

**Alias Sets ?**  
{x, \*p, \*r}  
{y, \*q, \*\*s}  
{q, \*s}

**p and q point to different locs**

# The Pointer Alias Analysis Problem

- Decide for **every pair of pointers** at **every program point**:
  - do they point to the same memory location?
- A difficult problem
  - shown to be **undecidable** by Landi, 1992
- **Correctness:**
  - report all pairs of pointers which do/may alias
- **Ambiguous:**
  - two pointers which **may or may not** alias
- **Accuracy/Precision:**
  - **how few pairs of pointers** are reported while **remaining correct**
  - i.e., reduce ambiguity to improve accuracy

# Many Uses of Pointer Analysis

- Basic compiler optimizations
  - register allocation, CSE, dead code elimination, live variables, instruction scheduling, loop invariant code motion, redundant load/store elimination
- Parallelization
  - instruction-level parallelism
  - thread-level parallelism
- Behavioral synthesis
  - automatically converting C-code into gates
- Error detection and program understanding
  - memory leaks, wild pointers, security holes

# Challenges for Pointer Analysis

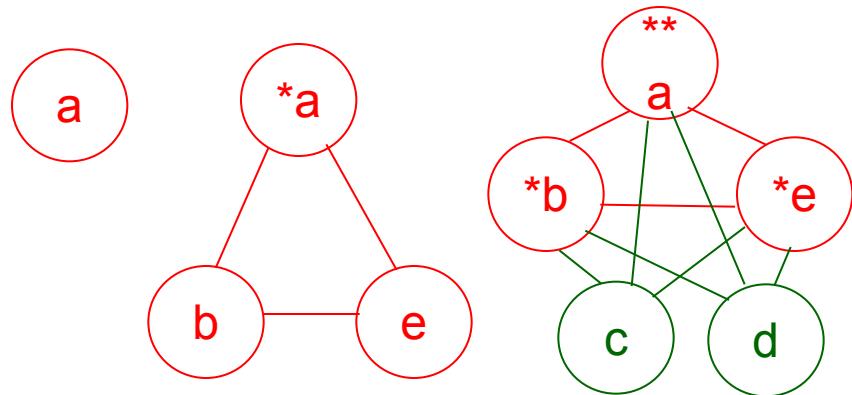
- **Complexity**: huge in **space** and **time**
  - compare every pointer with every other pointer
  - at every program point
  - potentially considering all program paths to that point
- **Scalability vs. accuracy trade-off**
  - different analyses motivated for different purposes
  - many useful algorithms (adds to confusion)
- **Coding corner cases**
  - pointer arithmetic (`*p++`), casting, function pointers, long-jumps
- **Whole program?**
  - most algorithms require the entire program
  - library code? optimizing at link-time only?

# Pointer Analysis: Design Options

- Representation
- Heap modeling
- Aggregate modeling
- Flow sensitivity
- Context sensitivity

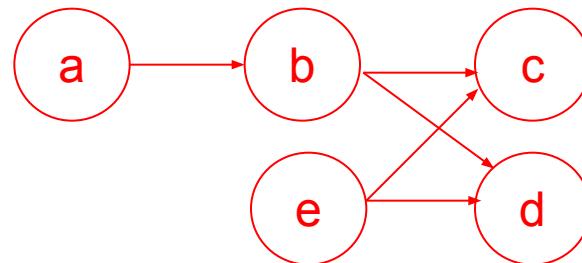
# Alias Representation

- Track **pointer** aliases
  - $\langle *a, b \rangle, \langle *a, e \rangle, \langle b, e \rangle$   
 $\langle **a, c \rangle, \langle **a, d \rangle, \dots$
  - More precise, less efficient



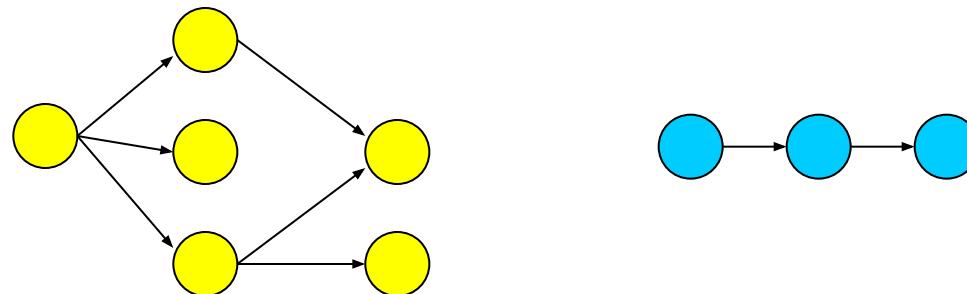
- Track **points-to** info
  - $\langle a, b \rangle, \langle b, c \rangle, \langle b, d \rangle,$   
 $\langle e, c \rangle, \langle e, d \rangle$
  - Less precise, more efficient
  - Why?

```
a = &b;  
b = &c;  
b = &d;  
e = b;
```



# Heap Modeling Options

- Heap merged
  - i.e. “no heap modeling”
- Allocation site (any call to malloc/calloc)
  - Consider each to be a unique location
  - Doesn’t differentiate between multiple objects allocated by the same allocation site
- Shape analysis
  - Recognize linked lists, trees, DAGs, etc.



# Aggregate Modeling Options

## Arrays



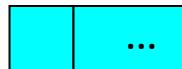
Elements are treated as **individual locations**

or



Treat entire array as a **single location**

or



Treat **first element** **separate** from others

## Structures



Elements are treated as **individual locations** (“field sensitive”)

or



Treat entire structure as a **single location**

What are the tradeoffs?

# Flow Sensitivity Options

- **Flow insensitive**
  - The order of statements doesn't matter
    - Result of analysis is the same regardless of statement order
  - Uses a single global state to store results as they are computed
  - Not very accurate
- **Flow sensitive**
  - The order of the statements matter
  - Need a control flow graph
  - Must store results for each program point
  - Improves accuracy
- **Path sensitive**
  - Each path in a control flow graph is considered

# Flow Sensitivity Example

(assuming allocation-site heap modeling)

```
S1: a = malloc(...);  
S2: b = malloc(...);  
S3: a = b;  
S4: a = malloc(...);  
S5: if(c)  
    a = b;  
S6: if(!c)  
    a = malloc(...);  
S7: ... = *a;
```

## Flow Insensitive

$a_{S7} \text{?}$  {heapS1, heapS2, heapS4, heapS6}

(order doesn't matter, union of all possibilities)

## Flow Sensitive

$a_{S7} \text{?}$  {heapS2, heapS4, heapS6}

(in-order, doesn't know s5 & s6 are exclusive)

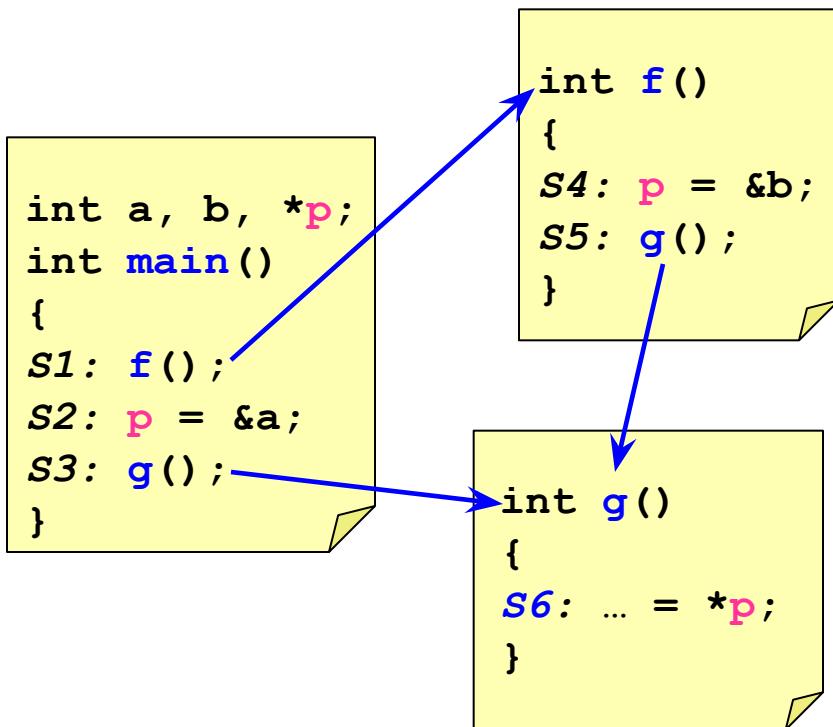
## Path Sensitive

$a_{S7} \text{?}$  {heapS2, heapS6}

(in-order, knows s5 & s6 are exclusive)

# Context Sensitivity Options

- Context insensitive/sensitive
  - whether to consider different calling contexts
  - e.g., what are the possibilities for **p** at **S6**?



Context Insensitive:

$p_{S6} \Rightarrow \{a, b\}$

Context Sensitive:

Called from S5: $p_{S6} \Rightarrow \{b\}$   
Called from S3: $p_{S6} \Rightarrow \{a\}$

# Pointer Alias Analysis Algorithms

## References:

- “*Points-to analysis in almost linear time*”, Steensgaard, POPL 1996
- “*Program Analysis and Specialization for the C Programming Language*”, Andersen, Technical Report, 1994
- “*Context-sensitive interprocedural points-to analysis in the presence of function pointers*”, Emami et al., PLDI 1994
- “*Pointer analysis: haven't we solved this problem yet?*”, Hind, PASTE 2001
- “*Which pointer analysis should I use?*”, Hind et al., ISSTA 2000
- ...
- “*Introspective analysis: context-sensitivity, across the board*”, Smaragdakiset al., PLDI 2014
- “*Sparse flow-sensitive pointer analysis for multithreaded programs*”, Sui et al., CGO 2016
- “*Symbolic range analysis of pointers*”, Paisanteet al., CGO 2016

# Address Taken

- Basic, fast, ultra-conservative algorithm
  - flow-insensitive, context-insensitive
  - often used in production compilers
- Algorithm:
  - Generate the set of all variables whose addresses are assigned to another variable.
  - Assume that any pointer can potentially point to any variable in that set.
- Complexity:  $O(n)$  - linear in size of program
- Accuracy: very imprecise

# Address Taken Example

```
T *p, *q, *r;

int main() {
S1: p = alloc(T);
    f();
    g(&p);
S4: p = alloc(T);
S5: ... = *p;
}
```

```
void f() {
S6: q = alloc(T);
    g(&q);
S8: r = alloc(T);
}
```

```
g(T **fp) {
    T local;
    if(...)
S9:     p = &local;
}
```

$$p_{S5} = \{ \text{heap\_S1, p, heap\_S4, heap\_S6, q, heap\_S8, local} \}$$

# Andersen's Algorithm

- Flow-insensitive, context-insensitive, iterative
- Representation:
  - one points-to graph for entire program
  - each node represents exactly one location
- For each statement, build the points-to graph:

$y = &x$	$y$ points-to $x$
$y = x$	if $x$ points-to $w$ then $y$ points-to $w$
$*y = x$	if $y$ points-to $z$ and $x$ points-to $w$ then $z$ points-to $w$
$y = *x$	if $x$ points-to $z$ and $z$ points-to $w$ then $y$ points-to $w$

- Iterate until graph no longer changes
- Worst case complexity:  $O(n^3)$ , where  $n$  = program size

# Andersen Example

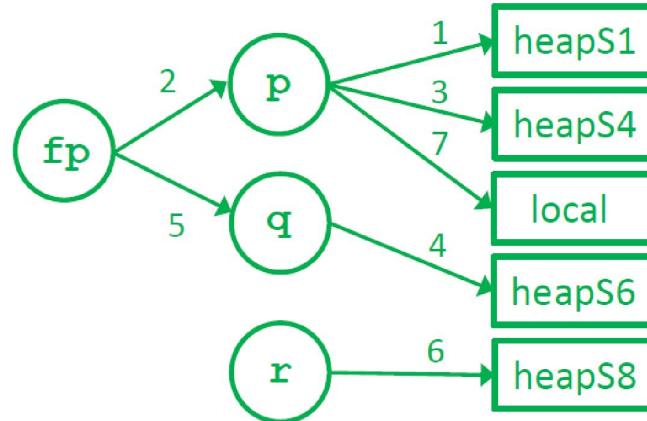
```
T *p, *q, *r;

int main() {
S1: p = alloc(T);
    f();
    g(&p);
S4: p = alloc(T);
S5: ... = *p;
}
```

```
void f() {
S6: q = alloc(T);
    g(&q);
S8: r = alloc(T);
}
```

```
g(T **fp) {
    T local;
    if(...)
S9:   p = &local;
}
```

$$p_{S5} = \{ \text{heap\_S1}, \text{heap\_S4}, \text{local} \}$$



# Steensgaard's Algorithm

- Flow-insensitive, context-insensitive
- Representation:
  - a **compact points-to** graph for entire program
    - each node can represent **multiple locations**
    - but **can only point to one other node**
      - i.e. every node has a **fan-out of 1 or 0**
- ***union-find*** data structure implements fan-out
  - “unioning” while finding **eliminates need to iterate**
- Worst case complexity:  **$O(n)$**
- Precision: less precise than Andersen's

# Steensgaard Example

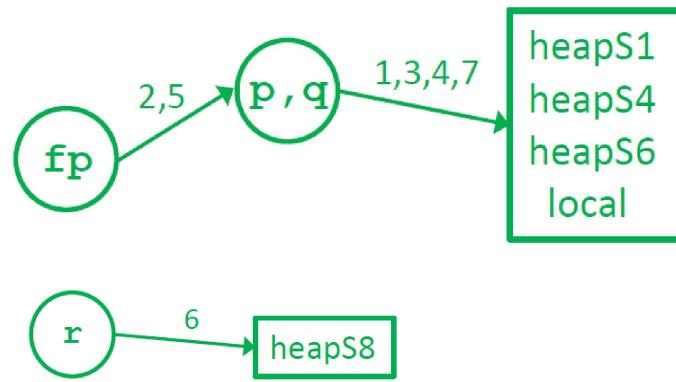
```
T *p, *q, *r;

int main() {
S1: p = alloc(T);
    f();
    g(&p);
S4: p = alloc(T);
S5: ... = *p;
}
```

```
void f() {
S6: q = alloc(T);
    g(&q);
S8: r = alloc(T);
}
```

```
g(T **fp) {
    T local;
    if(...)
S9:    p = &local;
}
```

$p_{S5} = \{ \text{heap\_S1}, \text{heap\_S4}, \text{heap\_S6}, \text{local} \}$



# Example with Flow Sensitivity

```
T *p, *q, *r;
```

```
int main() {  
    S1: p = alloc(T);  
        f();  
        g(&p);  
    S4: p = alloc(T);  
    S5: ... = *p;  
}
```

```
void f() {  
    S6: q = alloc(T);  
        g(&q);  
    S8: r = alloc(T);  
}
```

```
g(T **fp) {  
    T local;  
    if(...)  
    s9:     p = &local;  
}
```

$$p_{S5} = \{ \text{heap\_S4} \}$$

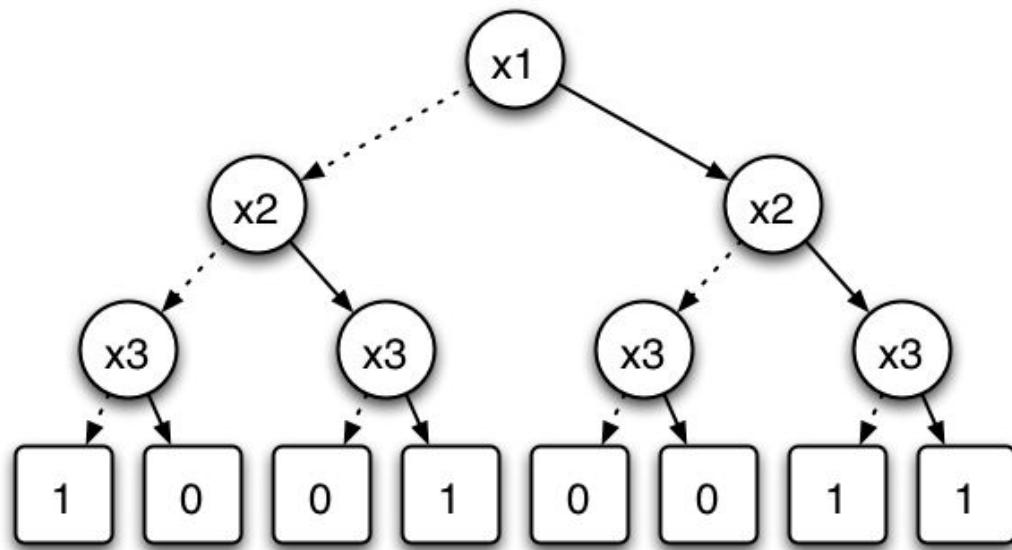
$$P_{S9} = \{ \text{local, heap\_s1} \}$$

# Pointer Analysis Using BDDs: Binary Decision Diagrams

## References:

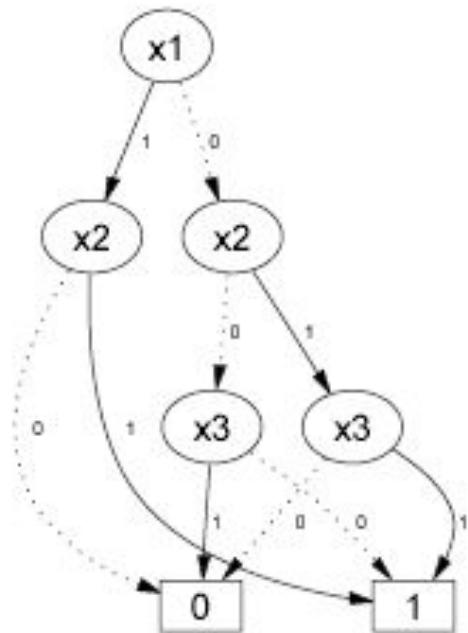
- “*Cloning-based context-sensitive pointer alias analysis using binary decision diagrams*”, Whaley and Lam, PLDI 2004
- “*Symbolic pointer analysis revisited*”, Zhu and Calman, PDLI 2004
- “*Points-to analysis using BDDs*”, Berndl et al, PDLI 2003

# Binary Decision Diagram (BDD)



$x_1$	$x_2$	$x_3$	$f$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Truth Table



# BDD-Based Pointer Analysis

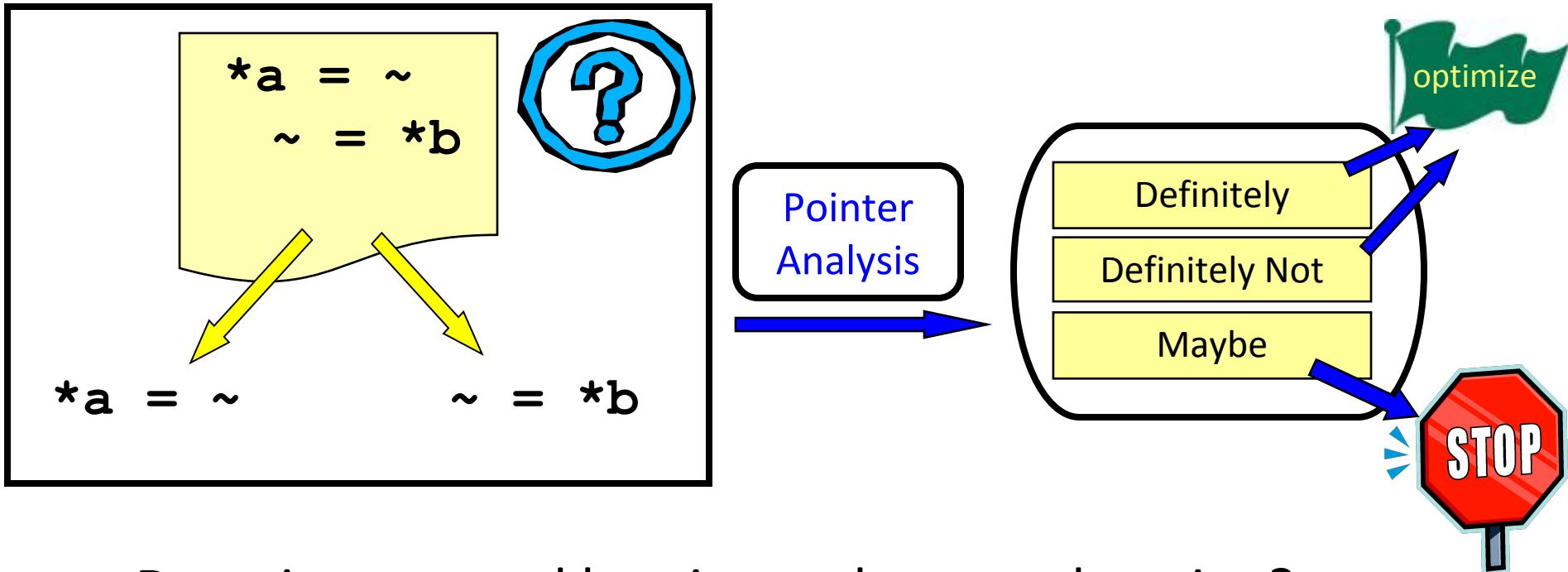
- Use a BDD to represent transfer functions
  - encode procedure as a function of its calling context
  - compact and efficient representation
- Perform context-sensitive, inter-procedural analysis
  - similar to dataflow analysis
  - but across the procedure call graph
- Gives accurate results
  - and scales up to large programs

# Probabilistic Pointer Analysis

## References:

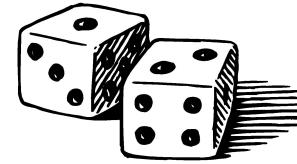
- “*A Probabilistic Pointer Analysis for Speculative Optimizations*”, DaSilva and Steffan, ASPLOS 2006
- “*Compiler support for speculative multithreading architecture with probabilistic points-to analysis*”, Shen et al., PPoPP 2003
- “*Speculative Alias Analysis for Executable Code*”, Fernandez and Espasa, PACT 2002
- “*A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion*”, Dai et al., CGO 2005
- “*Speculative register promotion using Advanced Load Address Table (ALAT)*”, Lin et al., CGO 2003

# Pointer Analysis: Yes, No, & Maybe



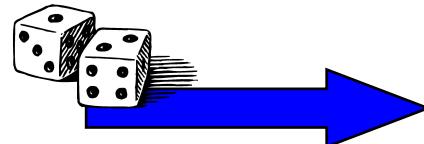
- Do pointers a and b point to the same location?
  - Repeat for every pair of pointers at every program point
- How can we optimize the “maybe” cases?

# Let's Speculate



- Implement a **potentially unsafe** optimization
  - Verify and Recover if necessary

```
int *a, x;  
...  
while (...) {  
    x = *a;  
    ...  
}
```



**a** is *probably*  
loop invariant

```
int *a, x, tmp;  
...  
tmp = *a;  
while (...) {  
    x = tmp;  
    ...  
}  
<verify, recover?>
```

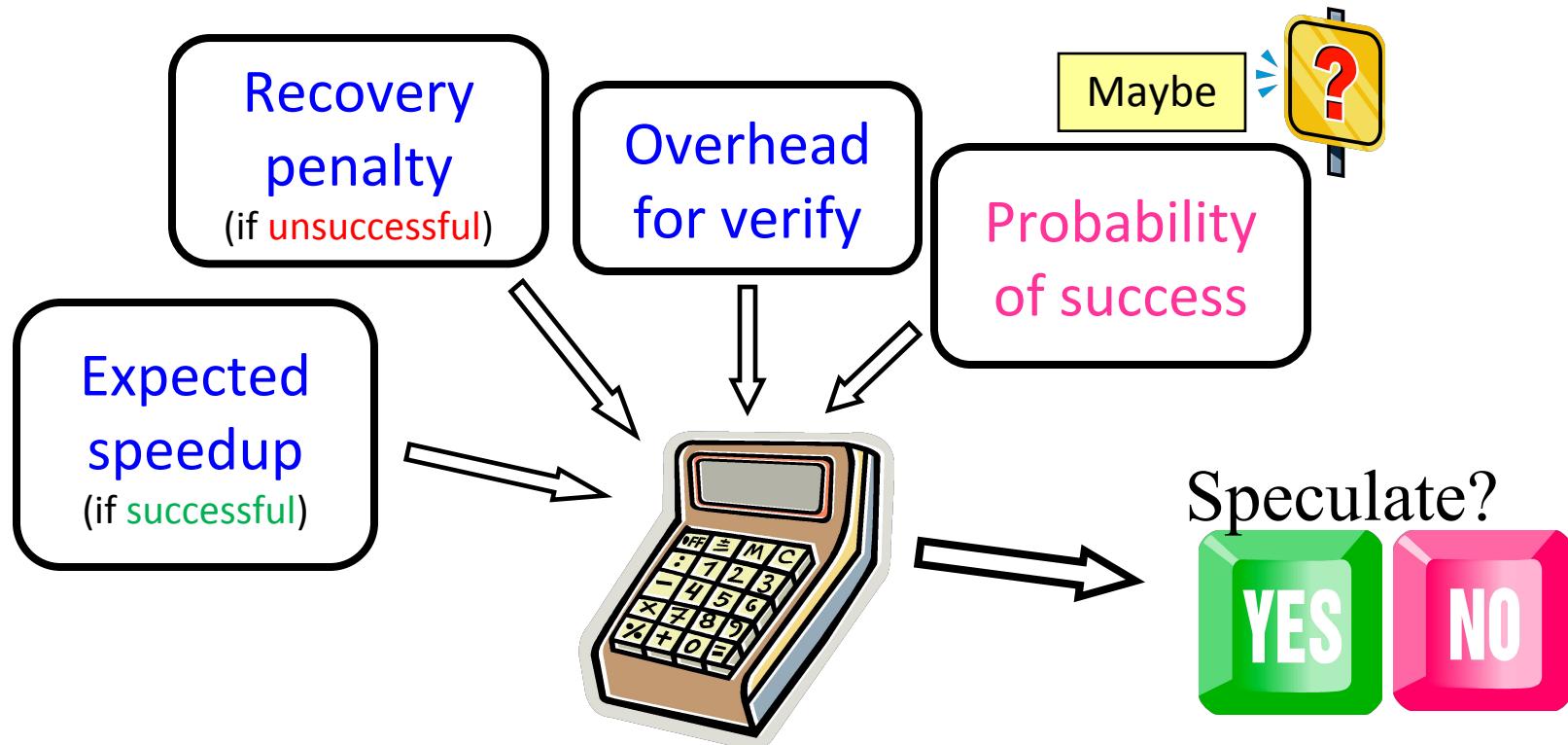
# Data Speculative Optimizations

- EPIC Instruction sets
  - Support for speculative load/store instructions (e.g., Itanium)
- Speculative compiler optimizations
  - Dead store elimination, redundancy elimination, copy propagation, strength reduction, register promotion
- Thread-level speculation (TLS)
  - Hardware and compiler support for speculative parallel threads
- Transactional programming
  - Hardware and software support for speculative parallel transactions

*Heavy reliance on detailed profile feedback*

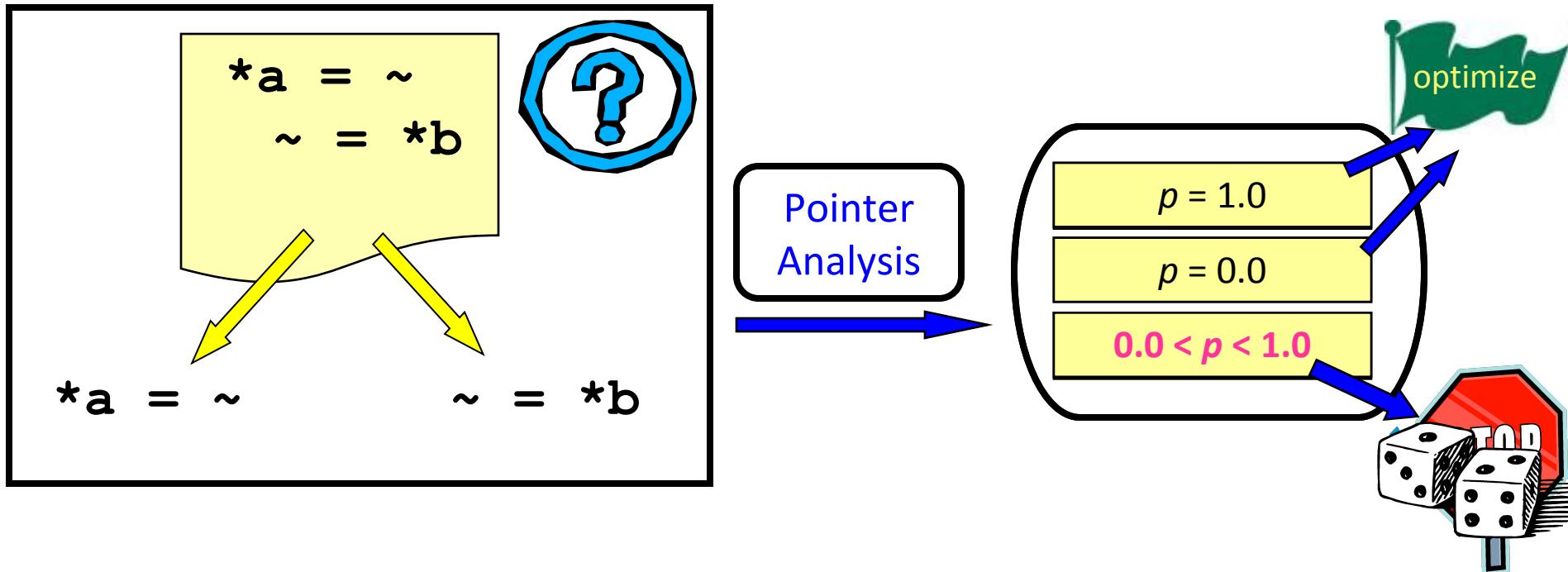
# Can We Quantify “Maybe”?

- Estimate the potential benefit for speculating:



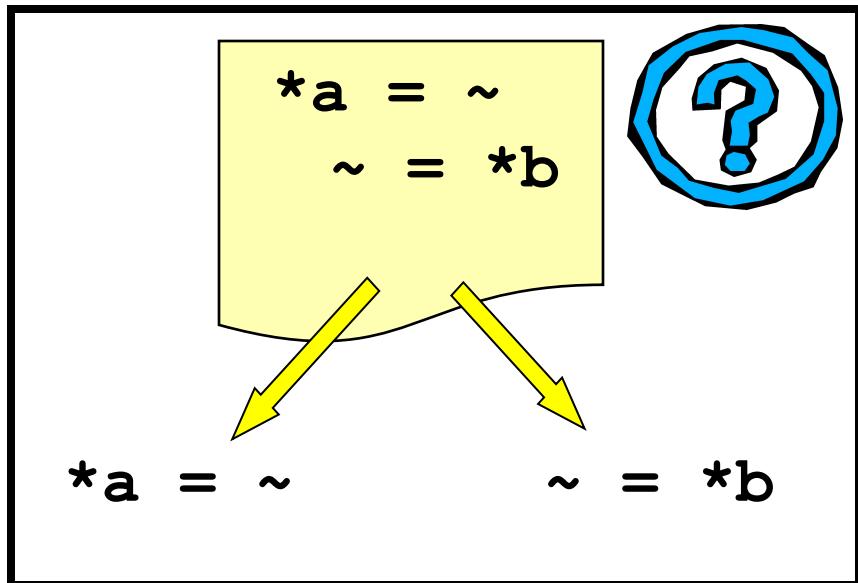
Ideally “maybe” should be a **probability**.

# Conventional Pointer Analysis

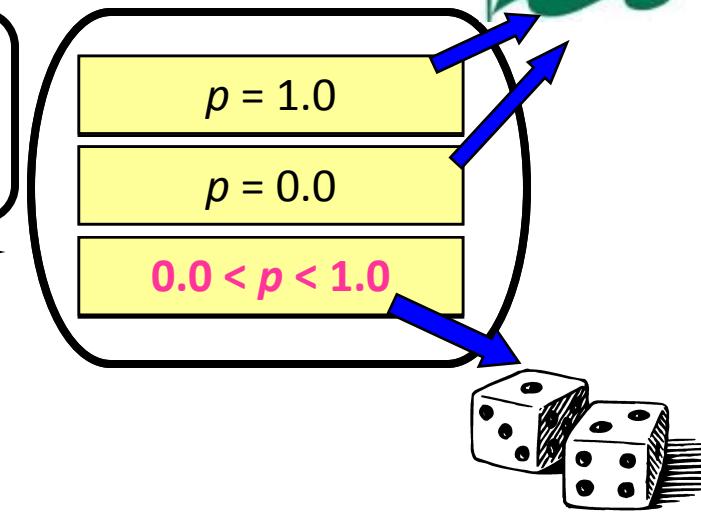


- Do pointers **a** and **b** point to the same location?
  - Repeat for every pair of pointers at every program point

# Probabilistic Pointer Analysis



Probabilistic  
Pointer  
Analysis



- Potential advantage of Probabilistic Pointer Analysis:
  - it doesn't need to be safe

# PPA Research Objectives

- Accurate points-to probability information
  - at every static pointer dereference
- Scalable analysis
  - Goal: entire SPEC integer benchmark suite
- Understand scalability/accuracy tradeoff
  - through flexible static memory model

*Improve our understanding of programs*

# Algorithm Design Choices

## Fixed:

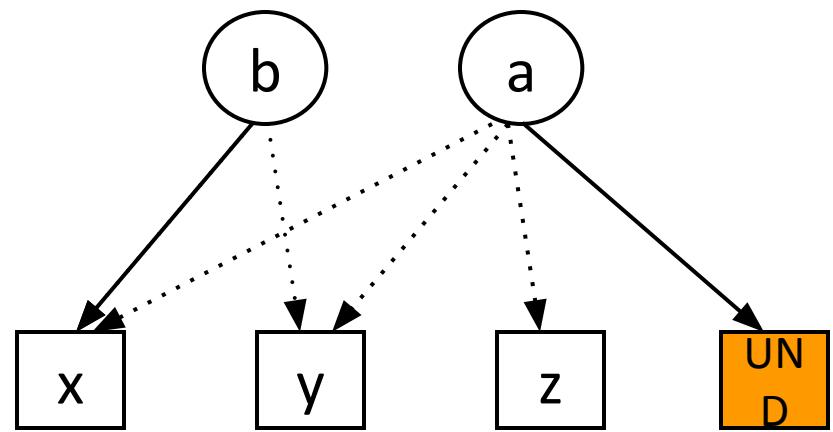
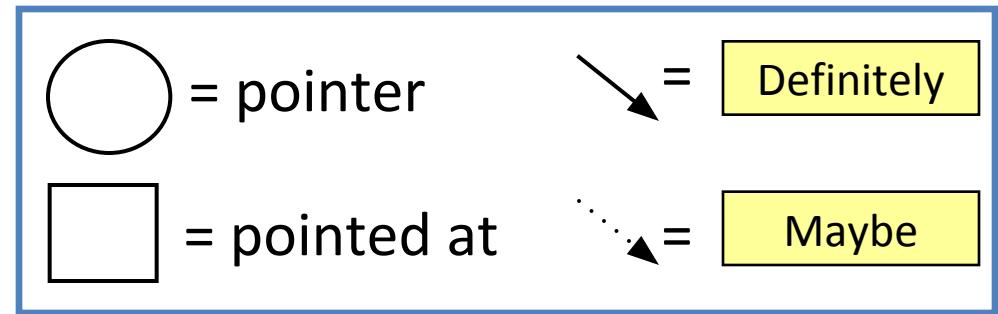
- Bottom Up / Top Down Approach
- Linear transfer functions (for scalability)
- One-level context and flow sensitive

## Flexible:

- Edge profiling (or static prediction)
- Safe (or unsafe)
- Field sensitive (or field insensitive)

# Traditional Points-To Graph

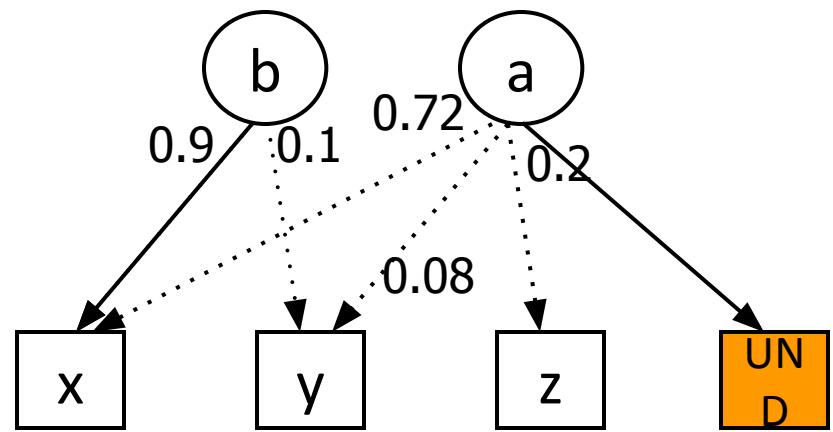
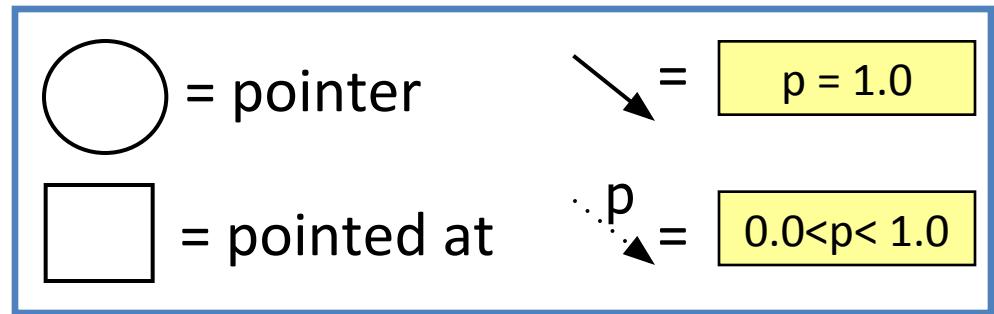
```
int x, y, z, *b = &x;  
void foo(int *a) {  
    if(...) b = &y;  
    if(...) a = &z;  
    else(...) a = b;  
    while(...) {  
        x = *a;  
        ...  
    }  
}
```



Results are inconclusive

# Probabilistic Points-To Graph

```
int x, y, z, *b = &x;  
void foo(int *a) {  
    if(...) □0.1 taken(edge profile)  
        b = &y;  
    if(...) □0.2 taken(edge profile)  
        a = &z;  
    else  
        a = b;  
    while(...) {  
        x = *a;  
        ...  
    }  
}
```



Results provide more information

# Probabilistic Pointer Analysis Results Summary

- Matrix-based, transfer function approach
  - SUIF/Matlab implementation
- Scales to the SPECint 95/2000 benchmarks
  - One-level context and flow sensitive
- As accurate as the most precise algorithms
- Interesting result:
  - ~90% of pointers tend to point to only one thing

# Pointer Analysis Summary

- Pointers are hard to understand at compile time!
  - accurate analyses are large and complex
- Many different options:
  - Representation, heap modeling, aggregate modeling, flow sensitivity, context sensitivity
- Many algorithms:
  - Address-taken, Steensgarde, Andersen, Emami
  - BDD-based, probabilistic
- Many trade-offs:
  - space, time, accuracy, safety
- Choose the right type of analysis given how the information will be used

# CSC D70: Compiler Optimization Memory Optimizations (Intro)

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# Caches: A Quick Review

- How do they work?
- Why do we care about them?
- What are typical configurations today?
- What are some important cache parameters that will affect performance?

# Optimizing Cache Performance

- Things to enhance:
  - temporal locality
  - spatial locality
- Things to minimize:
  - conflicts (i.e. bad replacement decisions)

What can the *compiler* do to help?

# Two Things We Can Manipulate

- Time:
  - When is an object accessed?
- Space:
  - Where does an object exist in the address space?

*How do we exploit these two levers?*

# Time: Reordering Computation

- What makes it difficult to know *when* an object is accessed?
- How can we predict a **better time** to access it?
  - What information is needed?
- How do we know that this would be **safe**?

# Space: Changing Data Layout

- What do we know about an object's **location**?
  - scalars, structures, pointer-based data structures, arrays, code, etc.
- How can we tell what a **better layout** would be?
  - how many can we create?
- To what extent can we **safely** alter the layout?

# Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays

# Scalars

- Locals
- Globals
- Procedure arguments
- Is cache performance a concern here?
- If so, what can be done?

```
int x;  
double y;  
foo(int a) {  
    int i;  
    ...  
    x = a*i;  
    ...  
}
```

# Structures and Pointers

- What can we do here?
  - within a node
  - across nodes

```
struct {
    int count;
    double velocity;
    double inertia;
    struct node *neighbors[N];
} node;
```

- What limits the compiler's ability to optimize here?

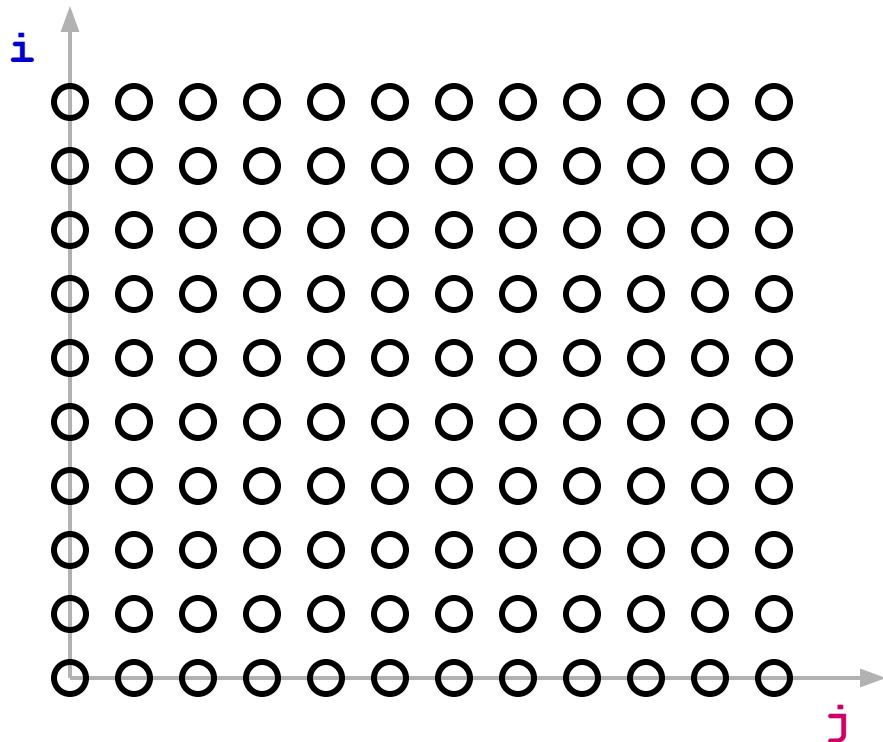
# Arrays

```
double A[N] [N] , B[N] [N] ;  
...  
for i = 0 to N-1  
    for j = 0 to N-1  
        A[i] [j] = B[j] [i] ;
```

- usually accessed within **loops nests**
  - makes it easy to understand “time”
- what we know about **array element addresses**:
  - start of array?
  - relative position within array

# Handy Representation: “Iteration Space”

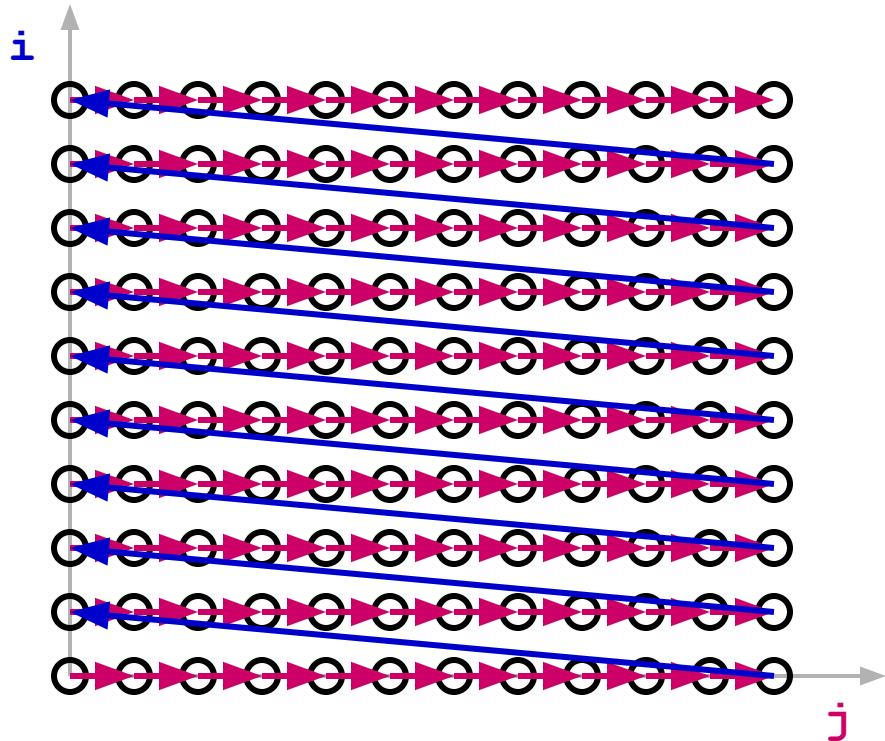
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```



- each position represents an iteration

# Visitation Order in Iteration Space

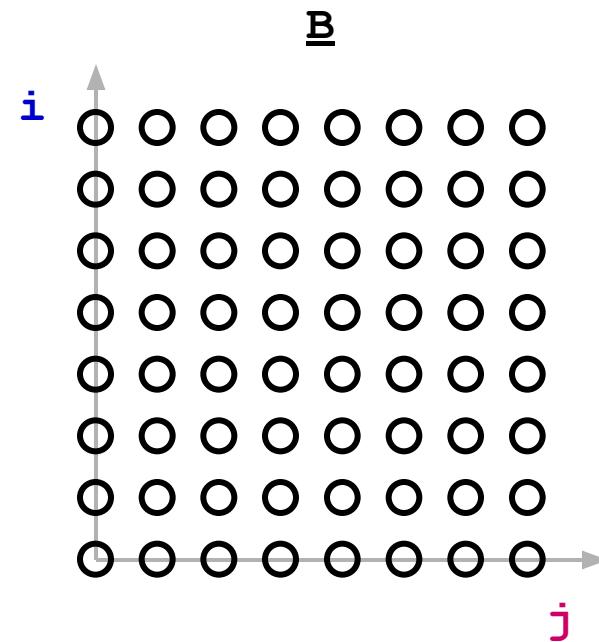
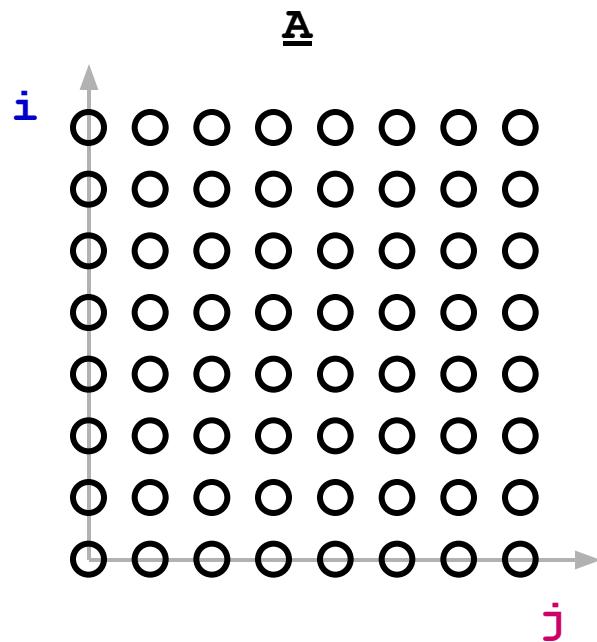
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```



- Note: iteration space  $\neq$  data space

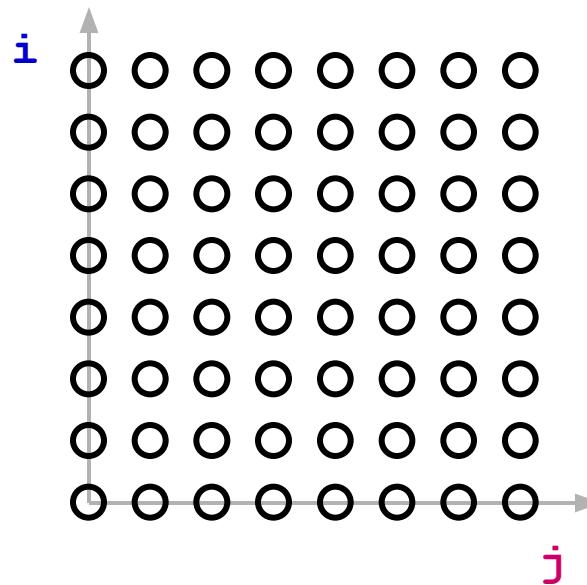
# When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```



# When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i+j][0] = i*j;
```



# Optimizing the Cache Behavior of Array Accesses

- We need to answer the following questions:
  - when do cache misses occur?
    - use “locality analysis”
  - can we change the order of the iterations (or possibly data layout) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use “dependence analysis”

# Examples of Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- ...

*(we will briefly discuss the first two next week)*

# CSC D70: Compiler Optimization Pointer Analysis & Memory Optimizations (Intro)

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# CSC D70: Compiler Optimization Memory Optimizations

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry, Greg Steffan, and Phillip Gibbons*

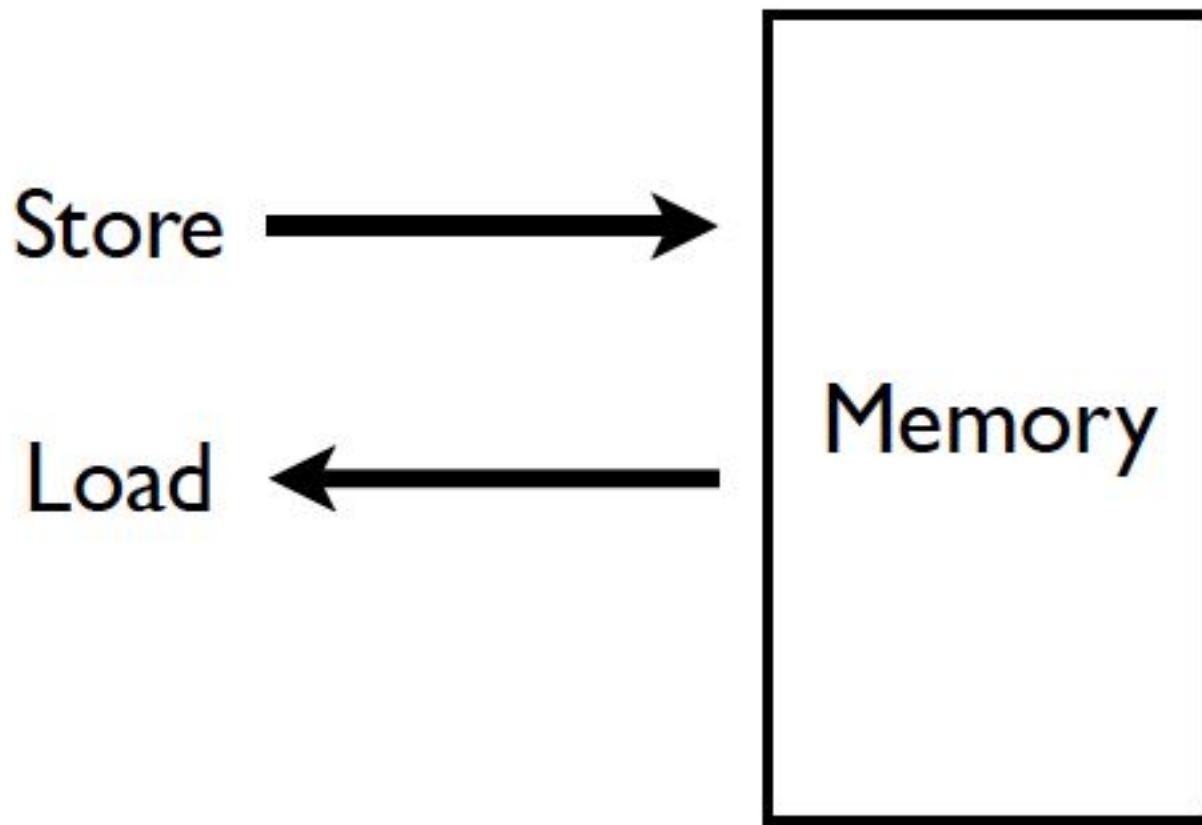
# Pointer Analysis (Summary)

- Pointers are hard to understand at compile time!
  - accurate analyses are large and complex
- Many different options:
  - Representation, heap modeling, aggregate modeling, flow sensitivity, context sensitivity
- Many algorithms:
  - Address-taken, Steensgarde, Andersen
  - BDD-based, probabilistic
- Many trade-offs:
  - space, time, accuracy, safety
- Choose the right type of analysis given how the information will be used

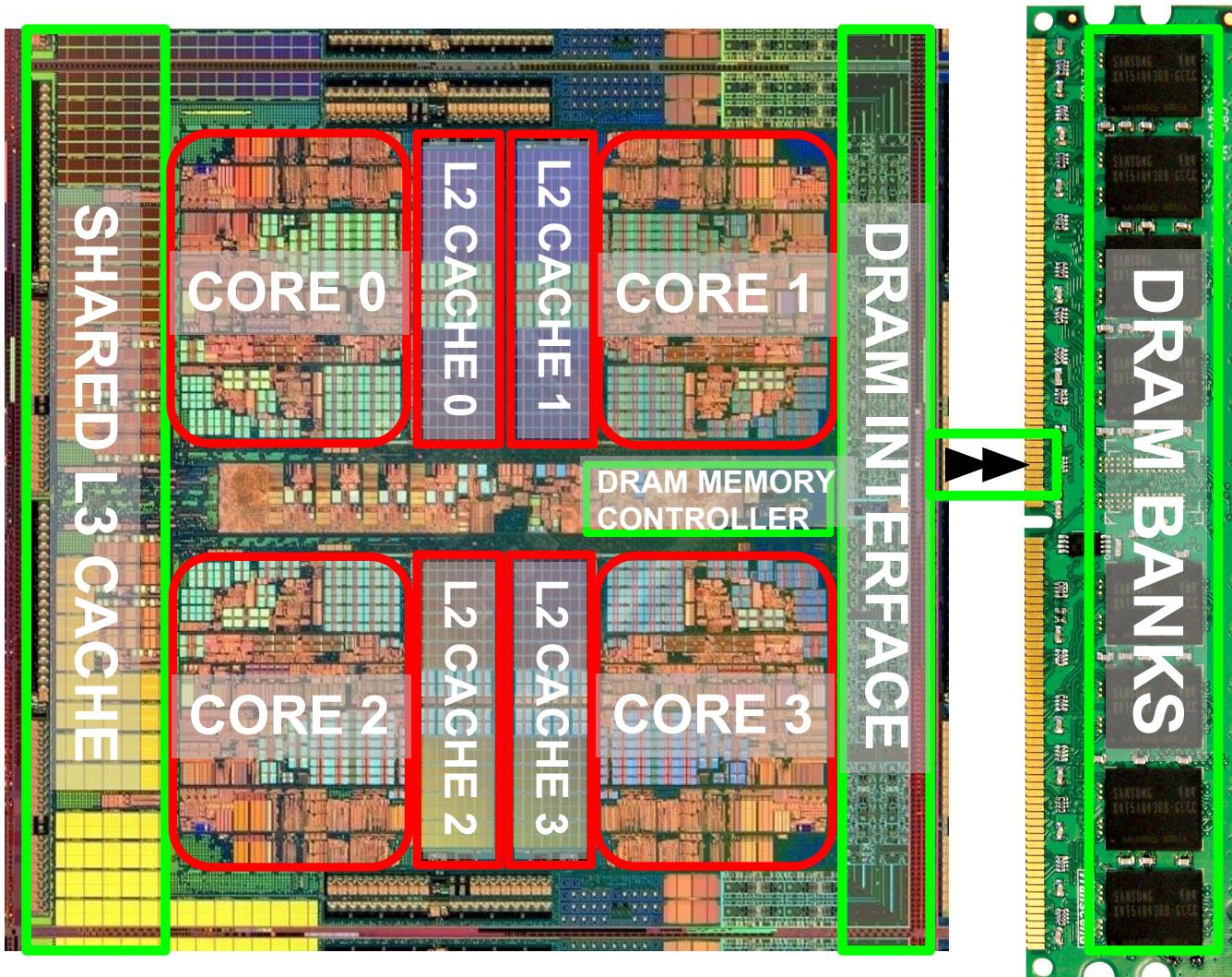
# Caches: A Quick Review

- How do they work?
- Why do we care about them?
- What are typical configurations today?
- What are some important cache parameters that will affect performance?

# Memory (Programmer's View)



# Memory in a Modern System



# Ideal Memory

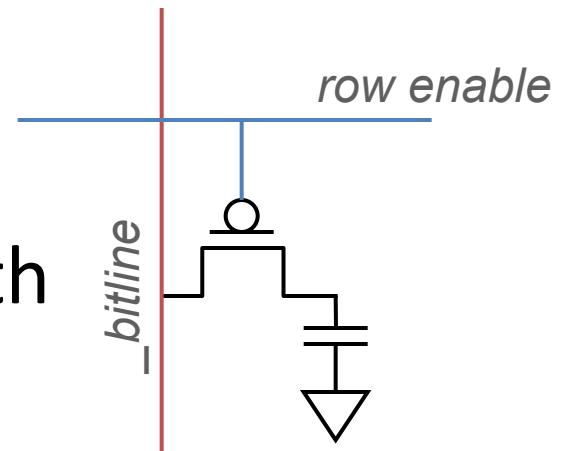
- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger ↗ Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Flash vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

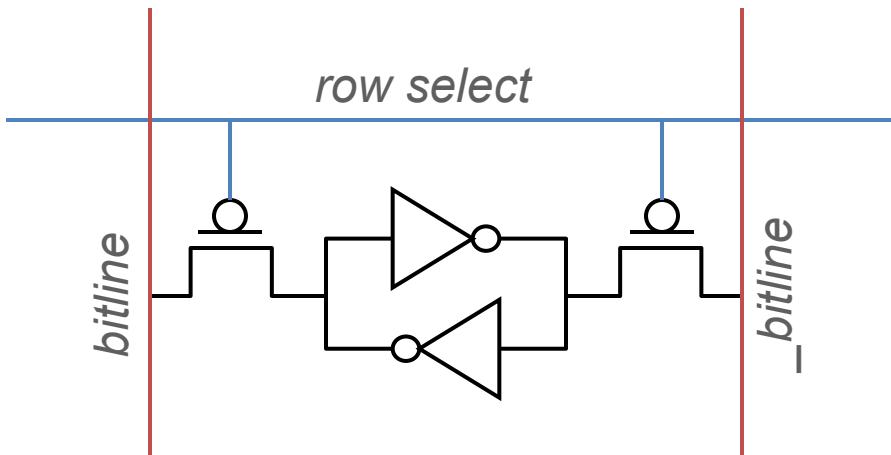
# Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor
- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed



# Memory Technology: SRAM

- Static random access memory
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the “cell”
  - 4 transistors for storage
  - 2 transistors for access



# Why Memory Hierarchy?

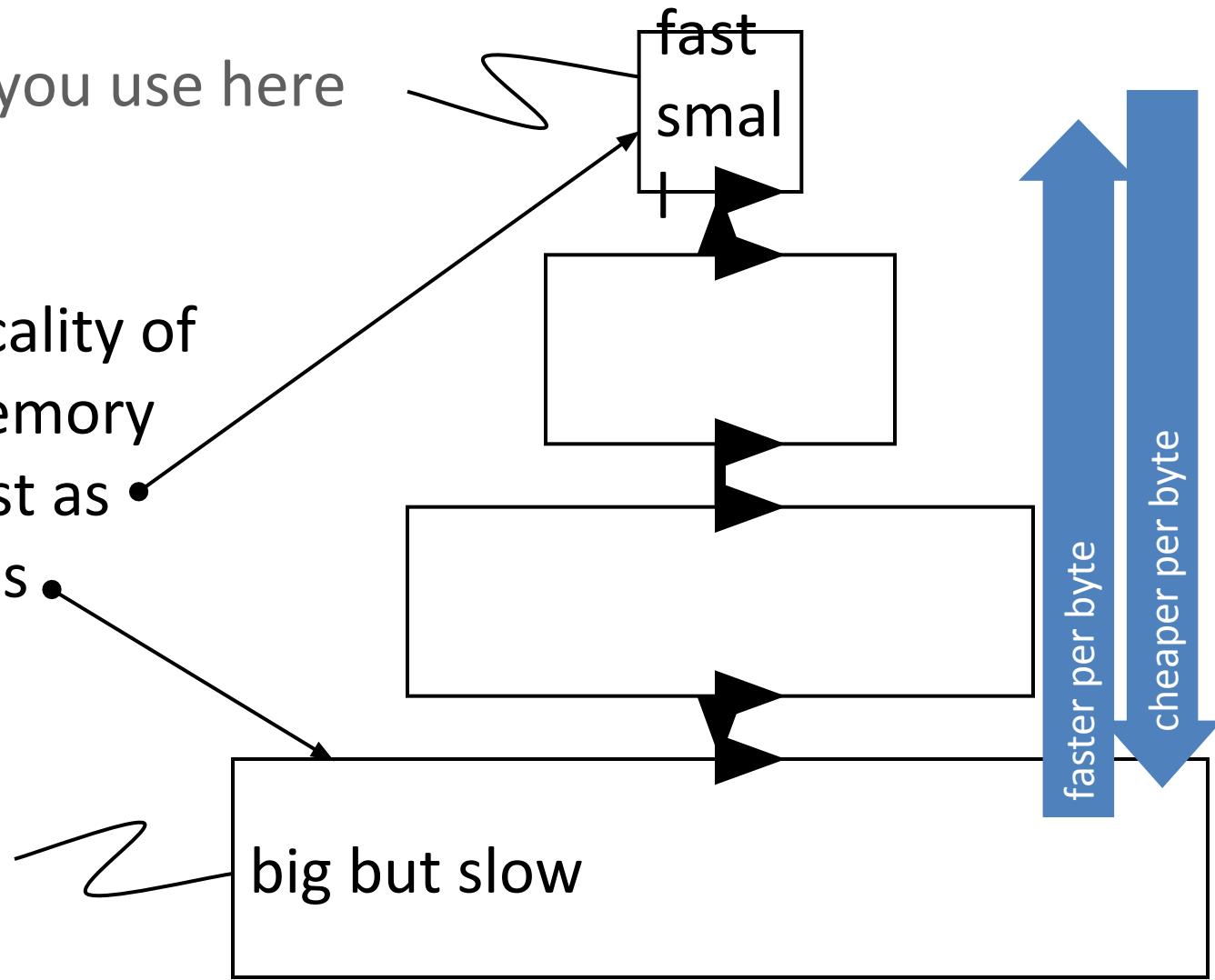
- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

# The Memory Hierarchy

move what you use here

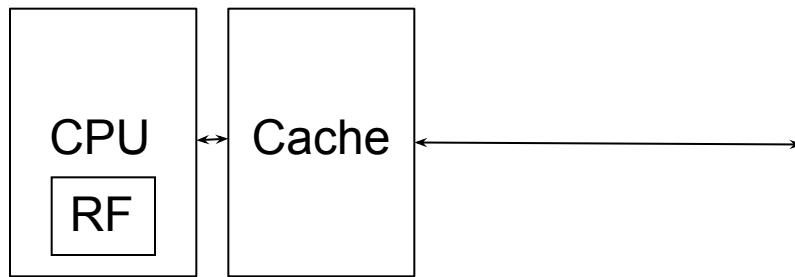
With good locality of reference, memory appears as fast as and as large as

backup everything here

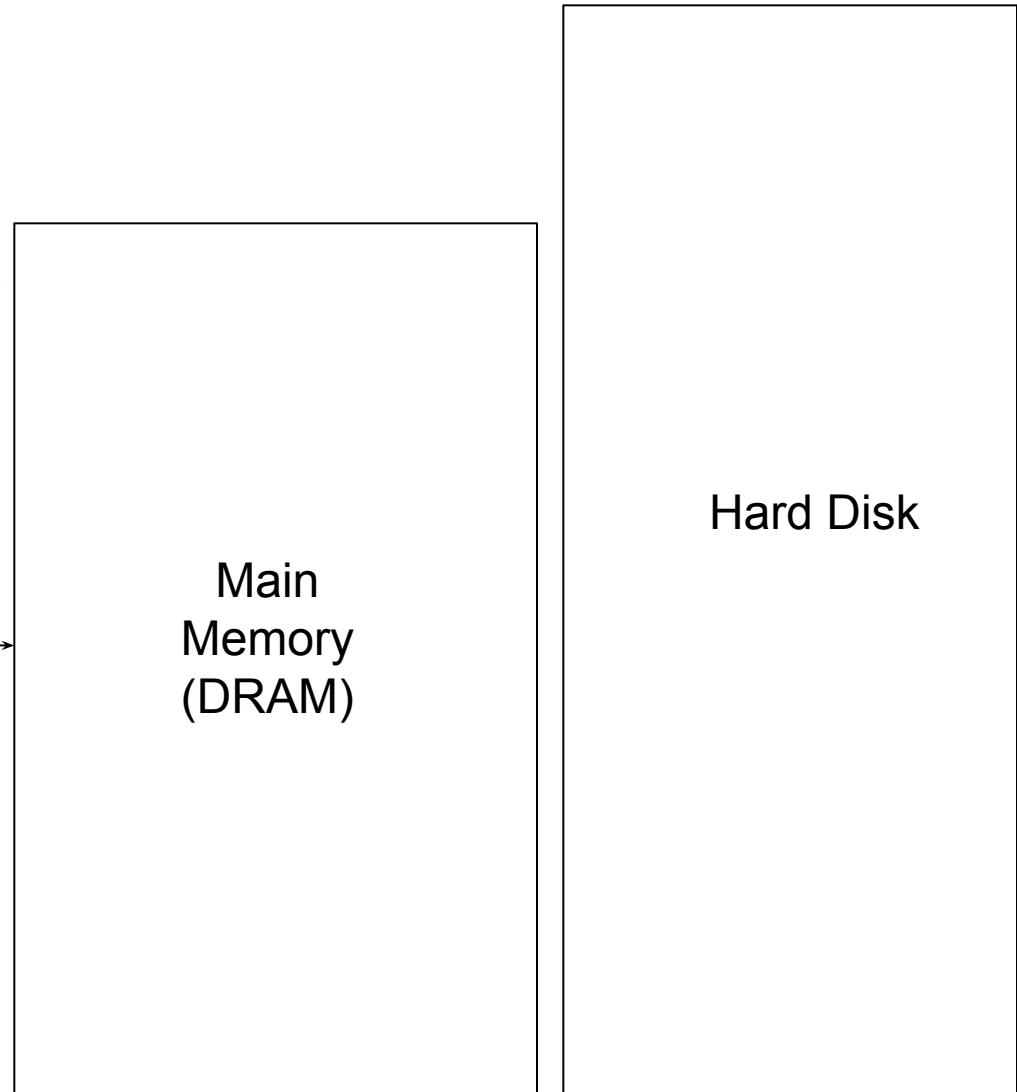


# Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: **Memory hierarchy**



- Latency, cost, size, bandwidth



# Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon
- Temporal locality principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - Wilkes, “Slave Memories and Dynamic Storage Allocation,” IEEE Trans. On Electronic Computers, 1965.
    - “The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.”

# Caching Basics: Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon
- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, “Structural aspects of the System/360 Model 85 II: the cache,” IBM Systems Journal, 1968.

# Optimizing Cache Performance

- Things to enhance:
  - temporal locality
  - spatial locality
- Things to minimize:
  - conflicts (i.e. bad replacement decisions)

What can the *compiler* do to help?

# Two Things We Can Manipulate

- Time:
  - When is an object accessed?
- Space:
  - Where does an object exist in the address space?

*How do we exploit these two levers?*

# Time: Reordering Computation

- What makes it difficult to know *when* an object is accessed?
- How can we predict a **better time** to access it?
  - What information is needed?
- How do we know that this would be **safe**?

# Space: Changing Data Layout

- What do we know about an object's **location**?
  - scalars, structures, pointer-based data structures, arrays, code, etc.
- How can we tell what a **better layout** would be?
  - how many can we create?
- To what extent can we **safely** alter the layout?

# Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays

# Scalars

- Locals
- Globals
- Procedure arguments
- Is cache performance a concern here?
- If so, what can be done?

```
int x;  
double y;  
foo(int a) {  
    int i;  
    ...  
    x = a*i;  
    ...  
}
```

# Structures and Pointers

- What can we do here?
  - within a node
  - across nodes

```
struct {
    int count;
    double velocity;
    double inertia;
    struct node *neighbors[N];
} node;
```

- What limits the compiler's ability to optimize here?

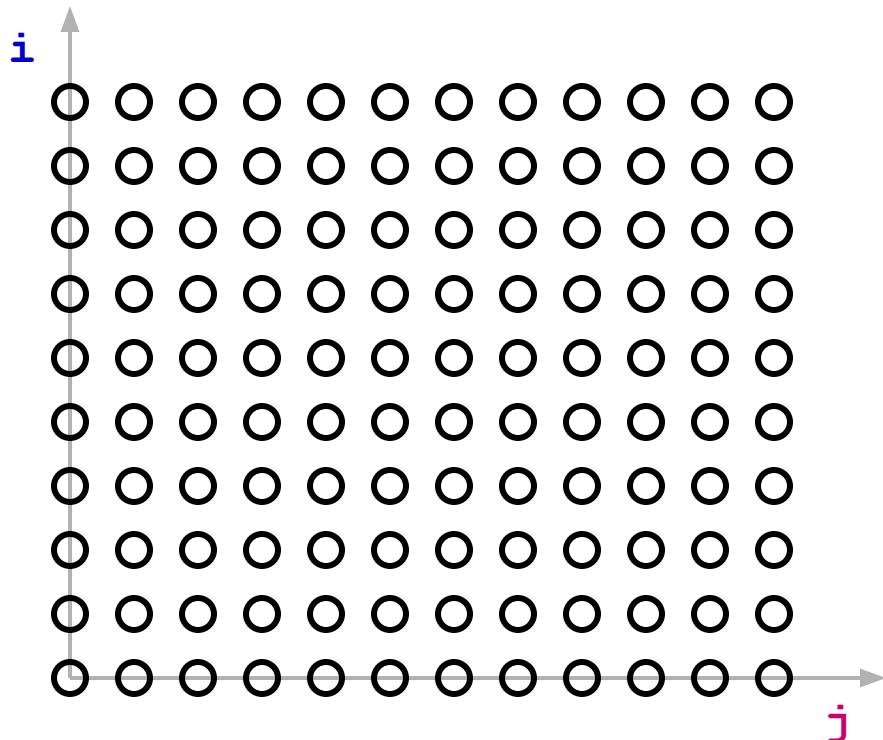
# Arrays

```
double A[N] [N] , B[N] [N] ;  
...  
for i = 0 to N-1  
    for j = 0 to N-1  
        A[i] [j] = B[j] [i] ;
```

- usually accessed within **loops nests**
  - makes it easy to understand “time”
- what we know about **array element addresses**:
  - start of array?
  - relative position within array

# Handy Representation: “Iteration Space”

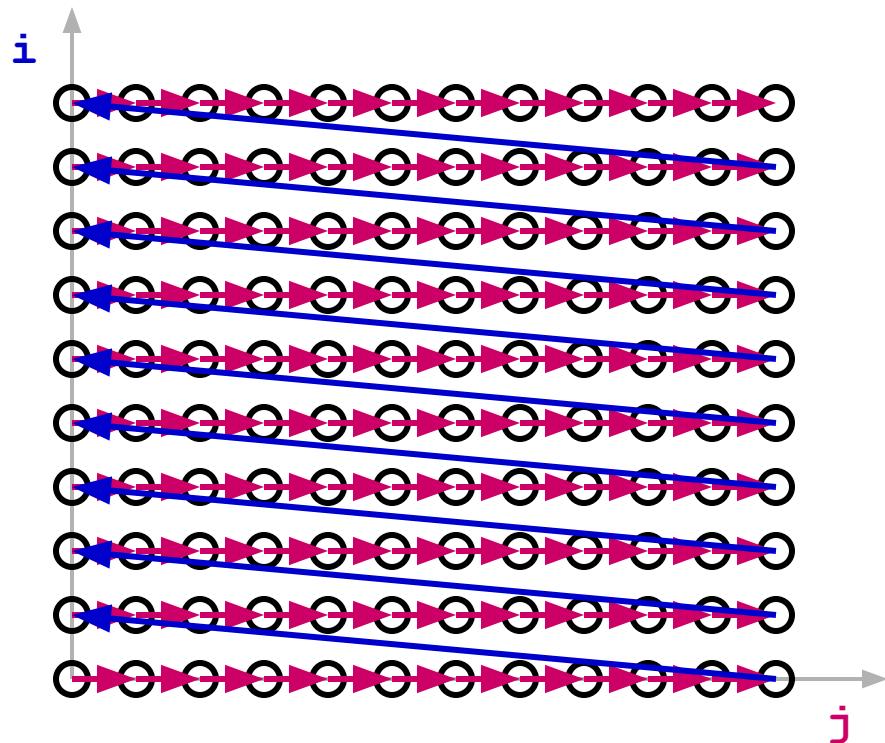
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```



- each position represents an iteration

# Visitation Order in Iteration Space

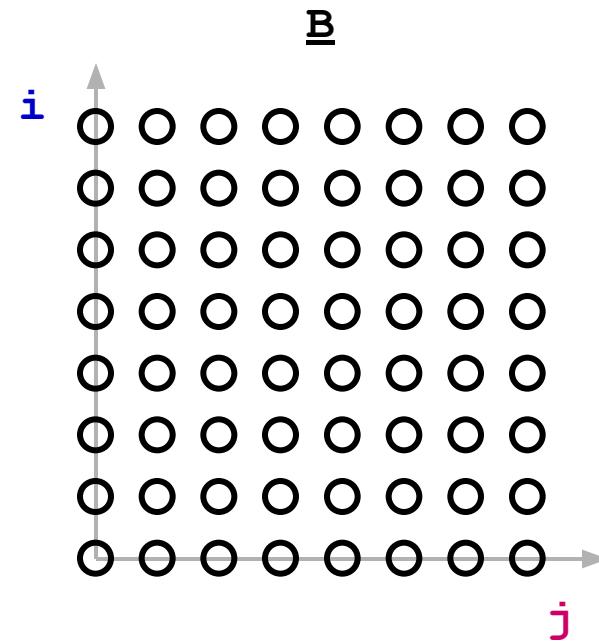
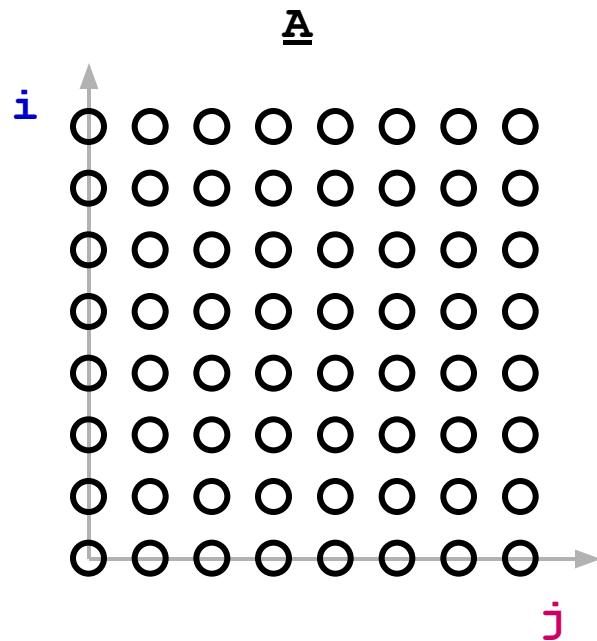
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```



- Note: iteration space  $\neq$  data space

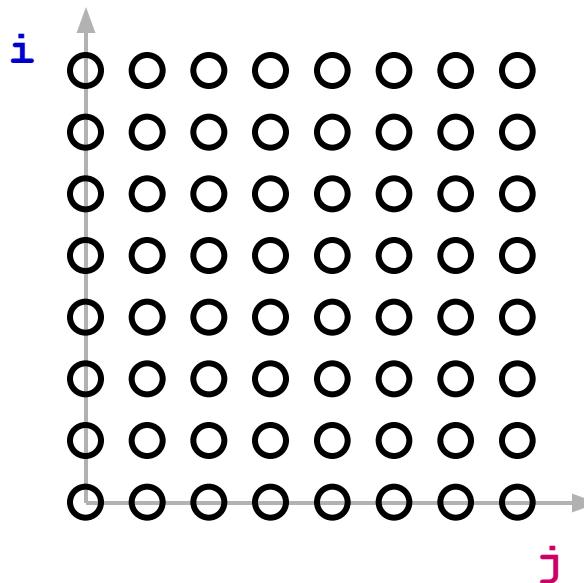
# When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```



# When Do Cache Misses Occur?

```
for i = 0 to N-1  
    for j = 0 to N-1  
        A[i+j][0] = i*j;
```



# Optimizing the Cache Behavior of Array Accesses

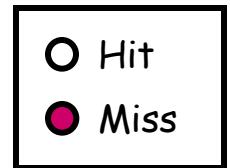
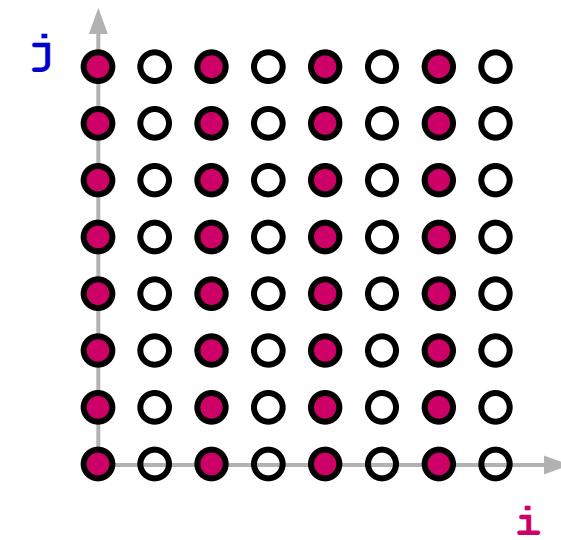
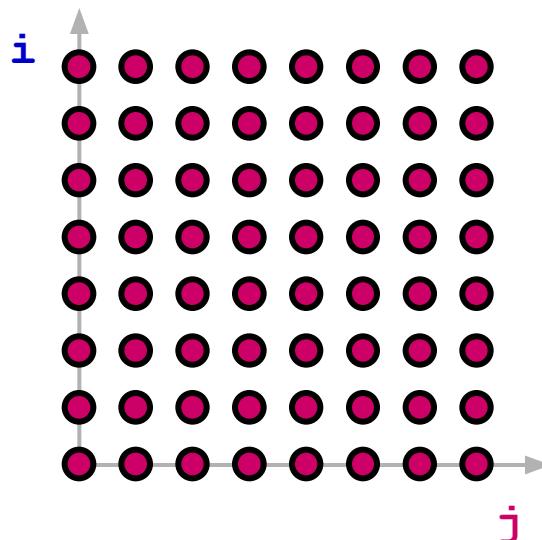
- We need to answer the following questions:
  - when do cache misses occur?
    - use “locality analysis”
  - can we change the order of the iterations (or possibly data layout) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use “dependence analysis”

# Examples of Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- ...

# Loop Interchange

```
for i = 0 to N-1           for j = 0 to N-1
    for j = 0 to N-1       ↖ ↘
        A[j][i] =          for i = 0 to N-1
                            A[j][i] =
                            i*j;
```

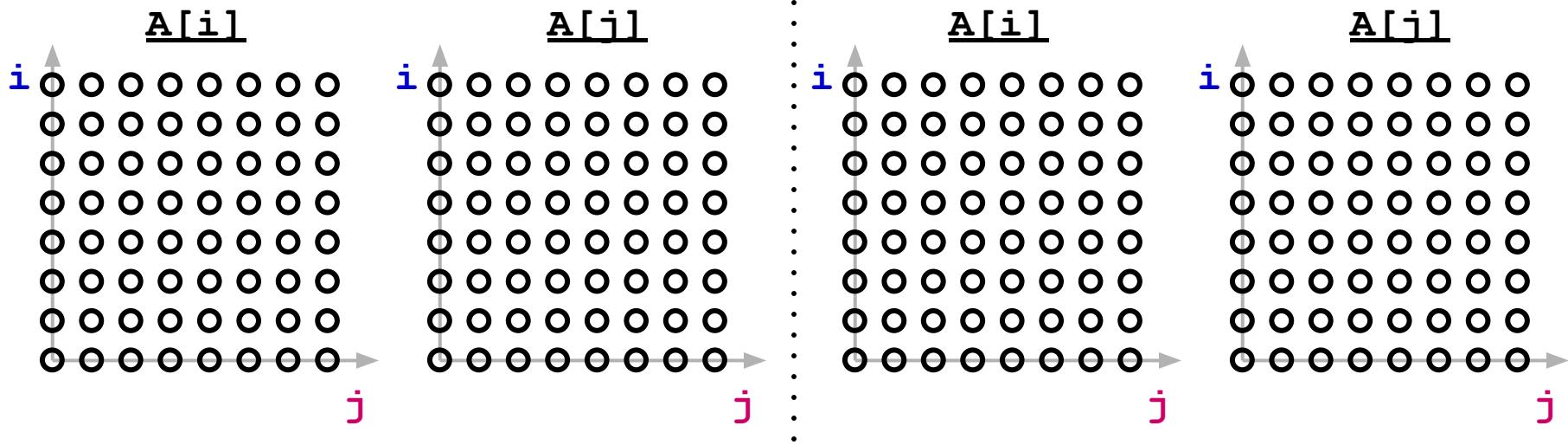


- (assuming  $N$  is large relative to cache size)

# Cache Blocking (aka “Tiling”)

```
for i = 0 to N-1  
    for j = 0 to N-1  
        f(A[i],A[j]);
```

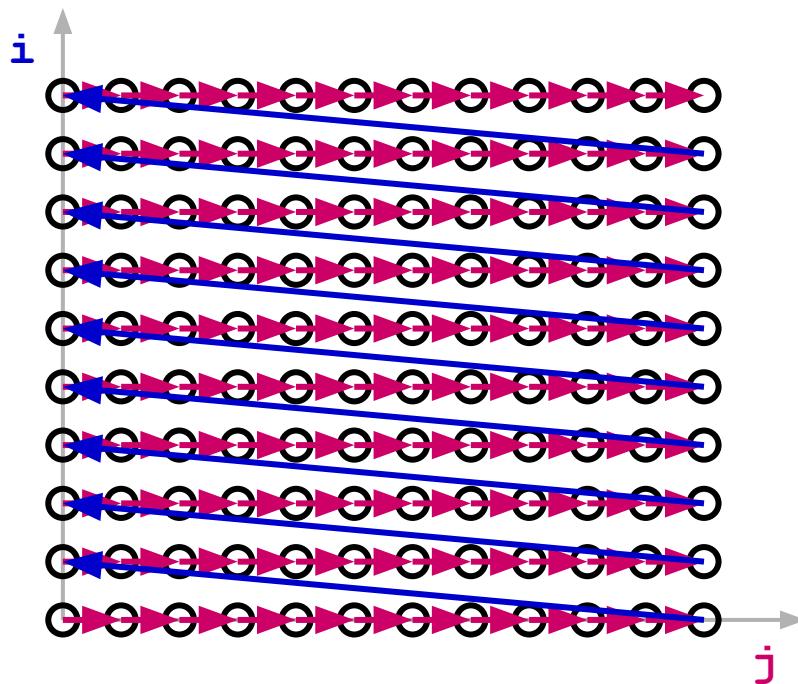
→ for JJ = 0 to N-1 by B  
 for i = 0 to N-1  
 for j = JJ to min(N-1, JJ+B-1)  
 f(A[i],A[j]);



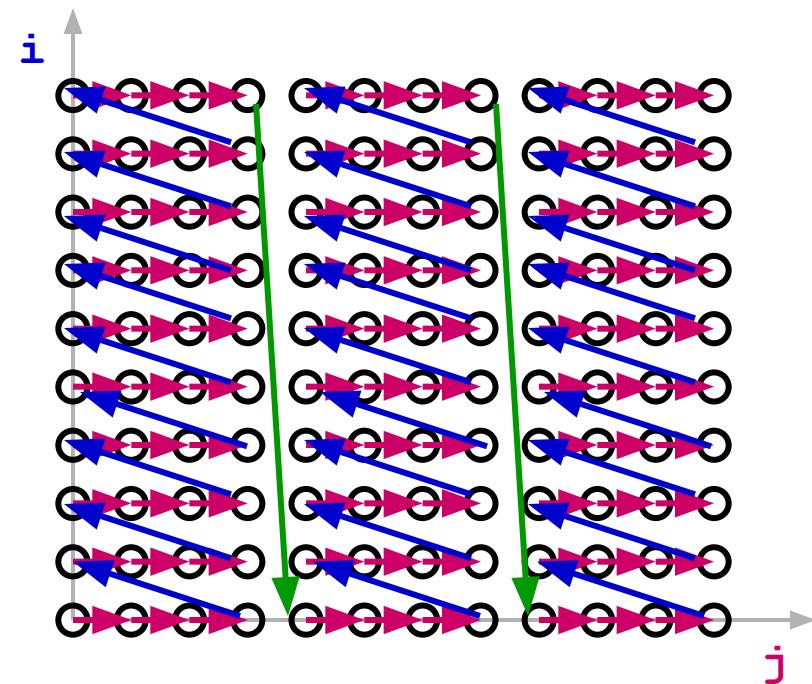
*now we can exploit temporal locality*

# Impact on Visitation Order in Iteration Space

```
for i = 0 to N-1  
  for j = 0 to N-1  
    f(A[i],A[j]);
```



→ for JJ = 0 to N-1 by B  
 for i = 0 to N-1  
 for j = JJ to min(N-1, JJ+B-1)  
 f(A[i],A[j]);



# Cache Blocking in Two Dimensions

```
for i = 0 to N-1
    for j = 0 to N-1
        for k = 0 to N-1
            c[i,k] +=
a[i,j]*b[j,k];
```

```
for JJ = 0 to N-1 by B
    for KK = 0 to N-1 by B
        for i = 0 to N-1
            for j = JJ to
min(N-1,JJ+B-1)
                for k = KK to
min(N-1,KK+B-1)
                    c[i,k] +=
a[i,j]*b[j,k];
```

- brings square sub-blocks of matrix “**b**” into the cache
- completely uses them up before moving on

# Predicting Cache Behavior through “Locality Analysis”

- Definitions:
  - Reuse:
    - accessing a location that has been accessed in the past
  - Locality:
    - accessing a location that is now found in the cache
- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
    - why not?

# Steps in Locality Analysis

## 1. Find data reuse

- if caches were infinitely large, we would be finished

## 2. Determine “localized iteration space”

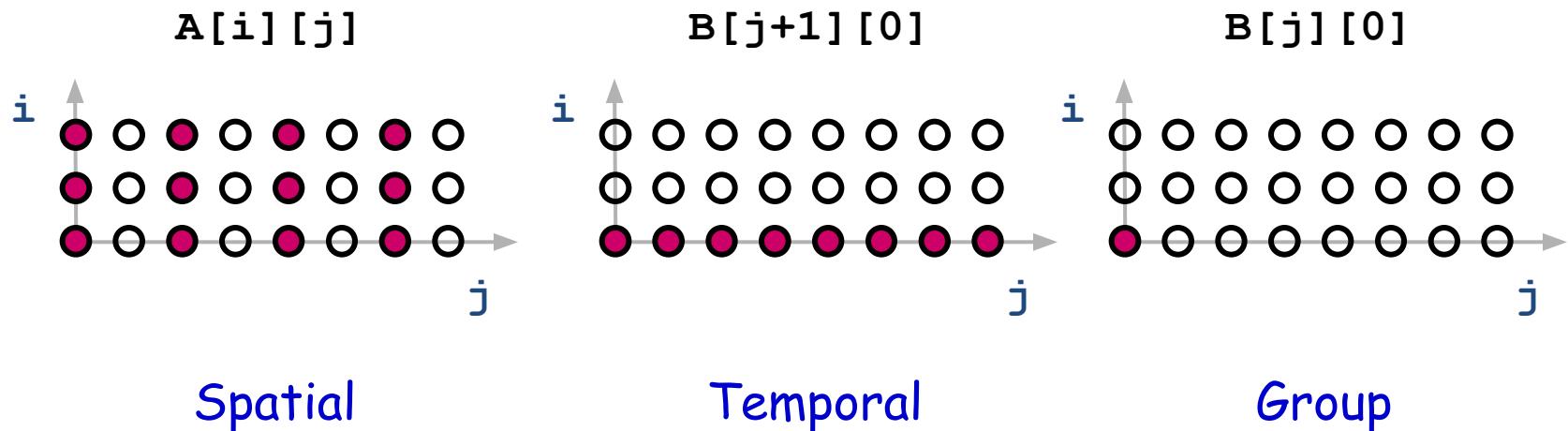
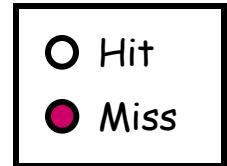
- set of inner loops where the data accessed by an iteration is expected to fit within the cache

## 3. Find data locality:

- reuse  $\cap$  localized iteration space  $\Rightarrow$  locality

# Types of Data Reuse/Locality

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```



# Reuse Analysis: Representation

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```

- Map *n* loop indices into *d* array indices via array indexing function:

$$\vec{f}(\vec{i}) = H\vec{i} + \vec{c}$$

$$A[i][j] = A \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

$$B[j][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

$$B[j+1][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)$$

# Finding Temporal Reuse

- Temporal reuse occurs between iterations  $\vec{i}_1$  and  $\vec{i}_2$  whenever:

$$H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c}$$

$$H(\vec{i}_1 - \vec{i}_2) = \vec{0}$$

- Rather than worrying about individual values  $\vec{i}_1$  or  $\vec{i}_2$  and, we say that reuse occurs along **direction**  $\vec{r}$  **vector** when:

$$H(\vec{r}) = \vec{0}$$

- **Solution:** compute the *nullspace* of  $H$

# Temporal Reuse Example

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```



- Reuse between iterations  $(i_1, j_1)$  and  $(i_2, j_2)$  whenever:

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

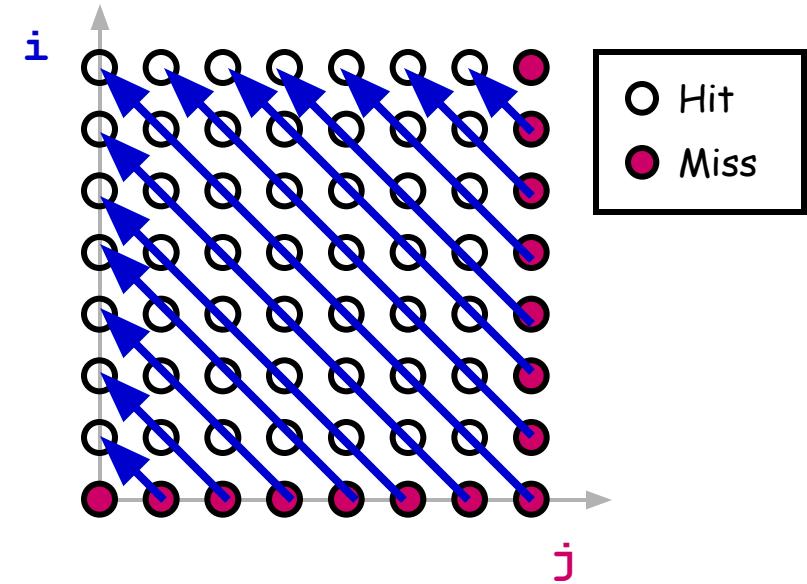
- True whenever  $j_1 = j_2$ , and regardless of the difference between  $i_1$  and  $i_2$ .

- i.e. whenever the difference lies along the nullspace of  $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
  - which is  $\text{span}\{(1,0)\}$  (i.e. the outer loop).

# More Complicated Example

```
for i = 0 to N-1  
  for j = 0 to N-1  
    A[i+j][0] = i*j;
```

$$A[i+j][0] = A \left( \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$



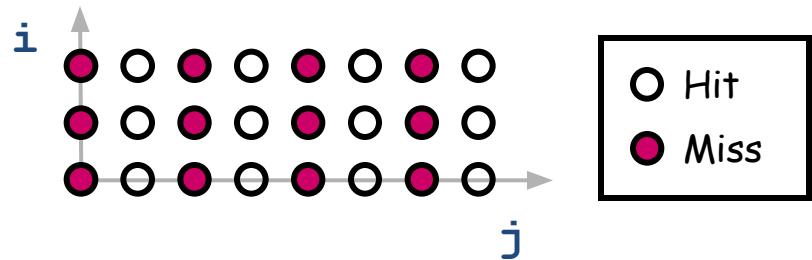
- Nullspace of  $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} = \text{span}\{(1, -1)\}$ .

# Computing Spatial Reuse

- Replace last row of  $H$  with zeros, creating  $H_s$
- Find the nullspace of  $H_s$
- Result: vector along which we access the same row

# Computing Spatial Reuse: Example

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```



$$A[i][j] = A \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

- $H_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$

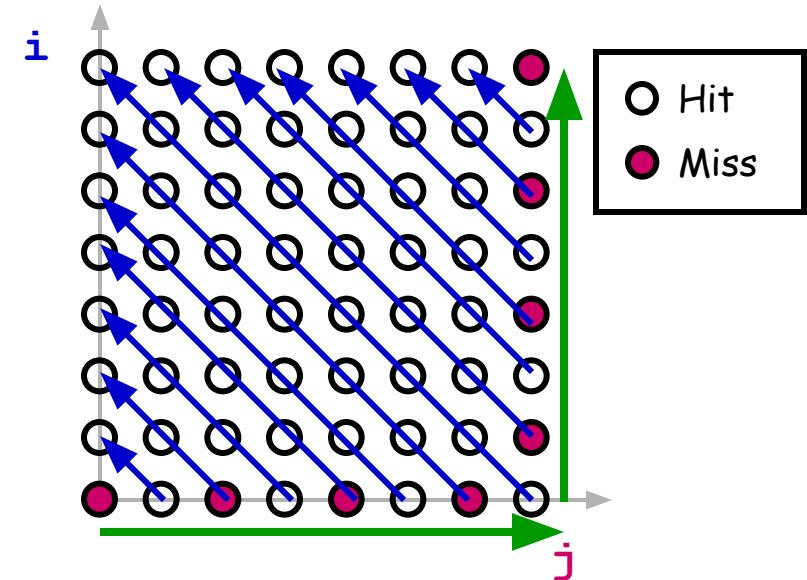
- Nullspace of  $H_s = \text{span}\{(0,1)\}$ 
  - i.e. access same row of  $A[i][j]$  along inner loop

# Computing Spatial Reuse: More Complicated Example

```
for i = 0 to N-1  
  for j = 0 to N-1  
    A[i+j] = i*j;
```

$$A[i+j] = A \left( \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} \right)$$

- $H_s = \begin{bmatrix} 0 & 0 \end{bmatrix}$
- Nullspace of  $H = \text{span}\{(1, -1)\}$
- Nullspace of  $H_s = \text{span}\{(1, 0), (0, 1)\}$



# Group Reuse

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```

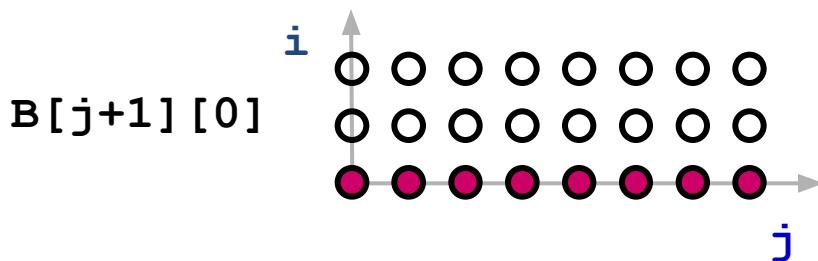

$$A[i][j] = B[j][0] + B[j+1][0]$$

- Only consider “uniformly generated sets”
  - index expressions differ only by constant terms
- Check whether they actually do access the same cache line
- Only the “leading reference” suffers the bulk of the cache misses

# Localized Iteration Space

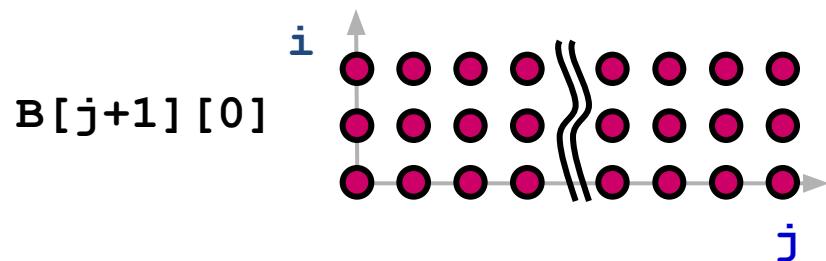
- Given finite cache, **when does reuse result in locality?**

```
for i = 0 to 2
    for j = 0 to 8
        A[i][j] = B[j][0] +
B[j+1][0];
```



Localized: both *i* and *j* loops  
(i.e.  $\text{span}\{(1,0), (0,1)\}$ )

```
for i = 0 to 2
    for j = 0 to 1000000
        A[i][j] = B[j][0] +
B[j+1][0];
```



Localized: *j* loop only  
(i.e.  $\text{span}\{(0,1)\}$ )

- Localized if accesses less data than *effective cache size*

# Computing Locality

- Reuse Vector Space  $\cap$  Localized Vector Space  $\Rightarrow$  Locality Vector Space
- Example:

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
        B[j+1][0];
```


- If both loops are localized:
  - $\text{span}\{(1,0)\} \cap \text{span}\{(1,0), (0,1)\} \Rightarrow \text{span}\{(1,0)\}$
  - i.e. temporal reuse *does* result in temporal locality
- If only the innermost loop is localized:
  - $\text{span}\{(1,0)\} \cap \text{span}\{(0,1)\} \Rightarrow \text{span}\{\}$
  - i.e. no temporal locality

# CSC D70: Compiler Optimization Memory Optimizations

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# CSC D70: Compiler Optimization Prefetching

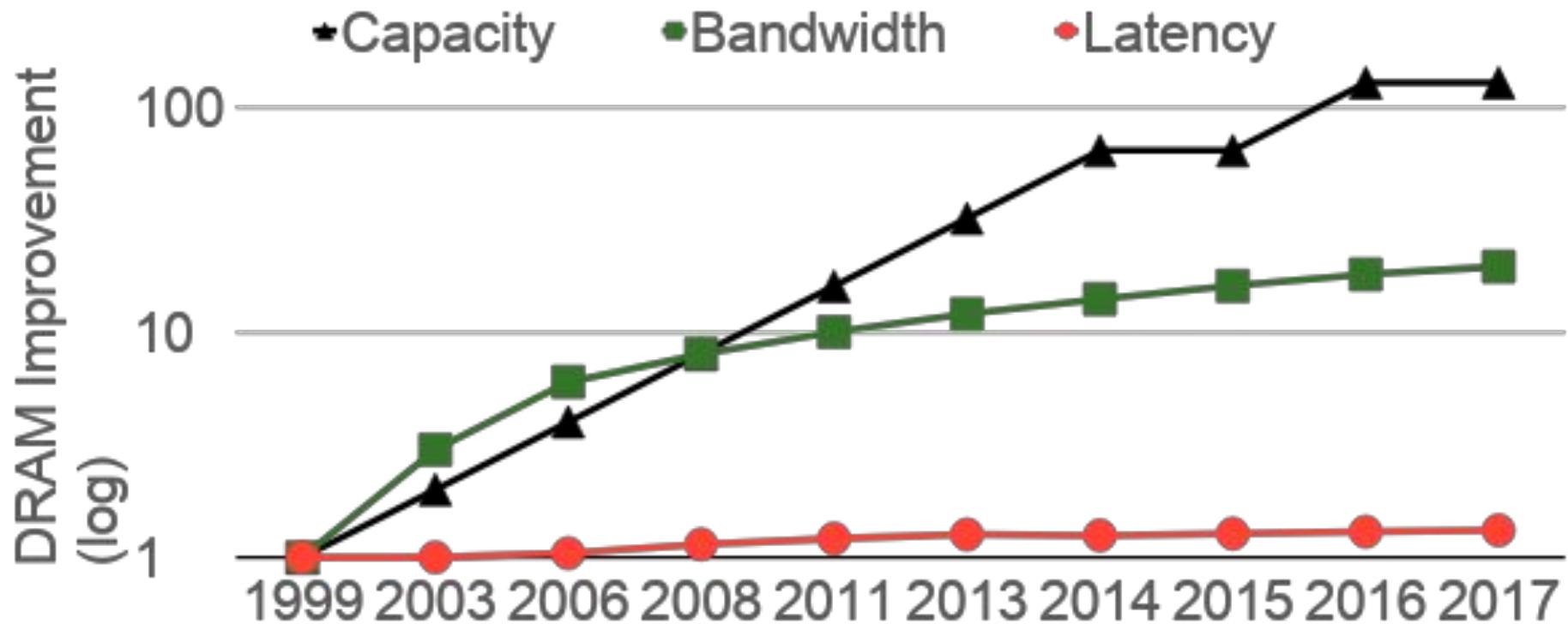
Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Phillip Gibbons*

# The Memory Latency Problem



- $\square$  processor speed >>  $\square$  memory speed
- caches are not a panacea

# Prefetching for Arrays: Overview

- Tolerating Memory Latency
- Prefetching Compiler Algorithm and Results
- Implications of These Results

# Coping with Memory Latency

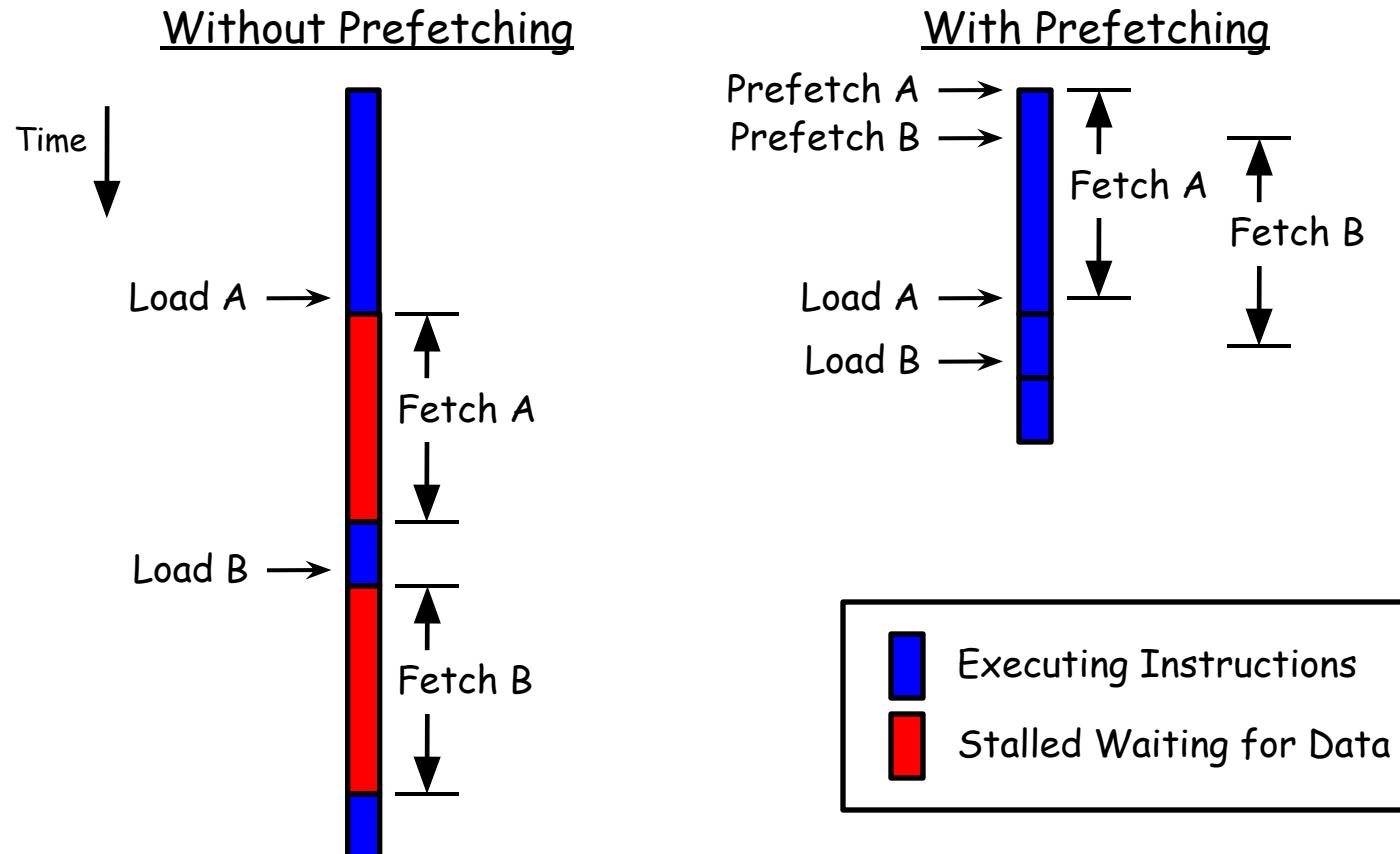
## Reduce Latency:

- Locality Optimizations
  - reorder iterations to improve cache reuse

## Tolerate Latency:

- Prefetching
  - move data close to the processor before it is needed

# Tolerating Latency Through Prefetching



- overlap memory accesses with computation and other accesses

# Types of Prefetching

## Cache Blocks:

- (-) limited to unit-stride accesses

## Nonblocking Loads:

- (-) limited ability to move back before use

## Hardware-Controlled Prefetching:

- (-) limited to constant-strides and by branch prediction
- (+) no instruction overhead

## Software-Controlled Prefetching:

- (-) software sophistication and overhead
- (+) minimal hardware support and broader coverage

# Prefetching Goals

- Domain of Applicability
- Performance Improvement
  - maximize benefit
  - minimize overhead

# Prefetching Concepts

*possible* only if addresses can be determined ahead of time

*coverage factor* = fraction of misses that are prefetched

*unnecessary* if data is already in the cache

*effective* if data is in the cache when later referenced

Analysis: what to prefetch

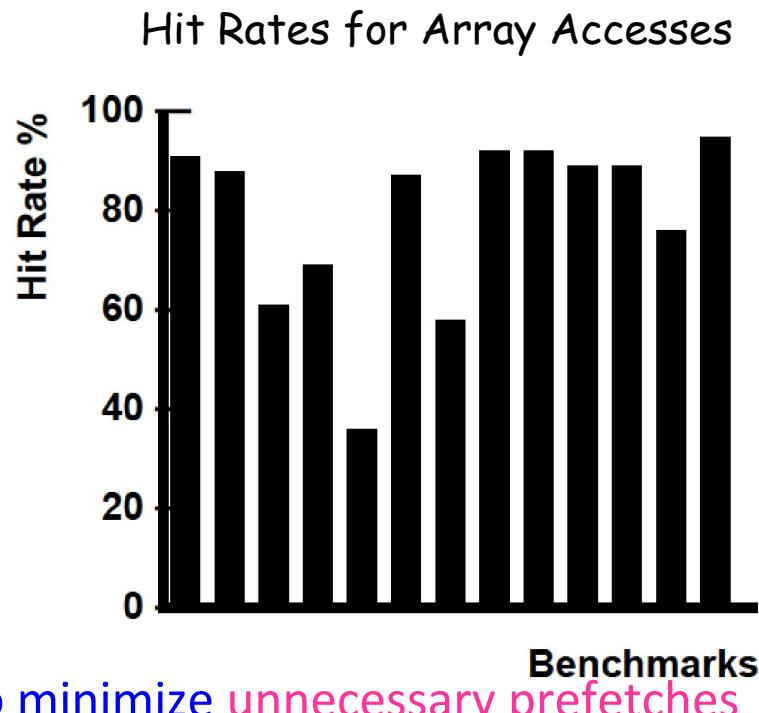
- maximize coverage factor
- minimize unnecessary prefetches

Scheduling: when/how to schedule prefetches

- maximize effectiveness
- minimize overhead per prefetch

# Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system



- important to minimize unnecessary prefetches

# Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

# Steps in Locality Analysis

## 1. Find data reuse

- if caches were infinitely large, we would be finished

## 2. Determine “localized iteration space”

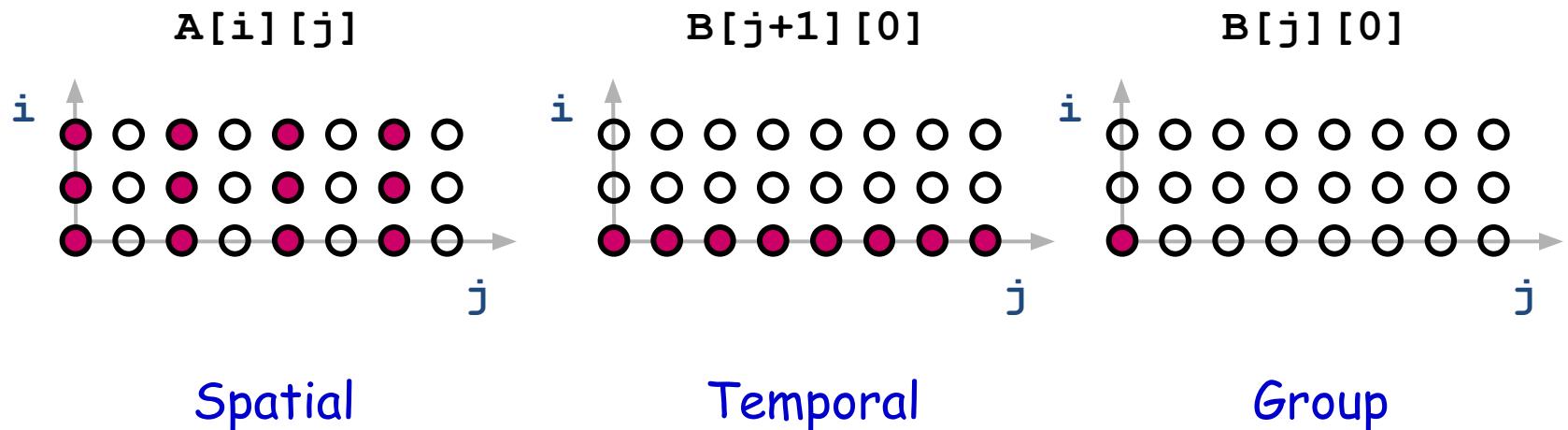
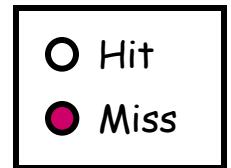
- set of inner loops where the data accessed by an iteration is expected to fit within the cache

## 3. Find data locality:

- reuse  $\cap$  localized iteration space  $\Rightarrow$  locality

# Data Locality Example

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```



# Reuse Analysis: Representation

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```

- Map *n* loop indices into *d* array indices via array indexing function:

$$\vec{f}(\vec{i}) = H\vec{i} + \vec{c}$$

$$A[i][j] = A \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

$$B[j][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right)$$

$$B[j+1][0] = B \left( \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right)$$

# Finding Temporal Reuse

- Temporal reuse occurs between iterations  $\vec{i}_1$  and  $\vec{i}_2$  whenever:

$$H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c}$$

$$H(\vec{i}_1 - \vec{i}_2) = \vec{0}$$

- Rather than worrying about individual values  $\vec{i}_1$  or  $\vec{i}_2$  and, we say that reuse occurs along **direction**  $\vec{r}$  **vector** when:

$$H(\vec{r}) = \vec{0}$$

- **Solution:** compute the *nullspace* of  $H$

# Temporal Reuse Example

```
for i = 0 to 2
    for j = 0 to 100
        A[i][j] = B[j][0] +
B[j+1][0];
```



- Reuse between iterations  $(i_1, j_1)$  and  $(i_2, j_2)$  whenever:

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_2 \\ j_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 - i_2 \\ j_1 - j_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- True whenever  $j_1 = j_2$ , and regardless of the difference between  $i_1$  and  $i_2$ .

- i.e. whenever the difference lies along the nullspace of  $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
  - which is  $\text{span}\{(1,0)\}$  (i.e. the outer loop).

# Prefetch Predicate

Locality Type	Miss Instance	Predicate
None	Every Iteration	True
Temporal	First Iteration	$i = 0$
Spatial	Every $l$ iterations ( $l$ = cache line size)	$(i \bmod l) = 0$

Example: `for i = 0 to 2  
 for j = 0 to 100  
 A[i][j] = B[j][0] +  
 B[j+1][0];`

Reference	Locality	Predicate
$A[i][j]$	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	$(j \bmod 2) = 0$
$B[j+1][0]$	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	$i = 0$

# Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

# Loop Splitting

- Decompose loops to isolate cache miss instances
  - cheaper than inserting IF statements

Locality Type	Predicate	Loop Transformation
None	True	None
Temporal	$i = 0$	Peel loop $i$
Spatial	$(i \bmod l) = 0$	Unroll loop $i$ by $l$

- Apply transformations recursively for nested loops
- Suppress transformations when loops become too large
  - avoid code explosion

# Software Pipelining

$$\text{Iterations Ahead} = \lceil \frac{l}{s} \rceil$$

where  $l$  = memory latency,  $s$  = shortest path through loop body

## Original Loop

```
for (i = 0; i<100; i++)
    a[i] = 0;
```

## Software Pipelined Loop (5 iterations ahead)

```
for (i = 0; i<5; i++)      /* Prolog */
    prefetch(&a[i]);

for (i = 0; i<95; i++) { /* Steady State */
    prefetch(&a[i+5]);
    a[i] = 0;
}

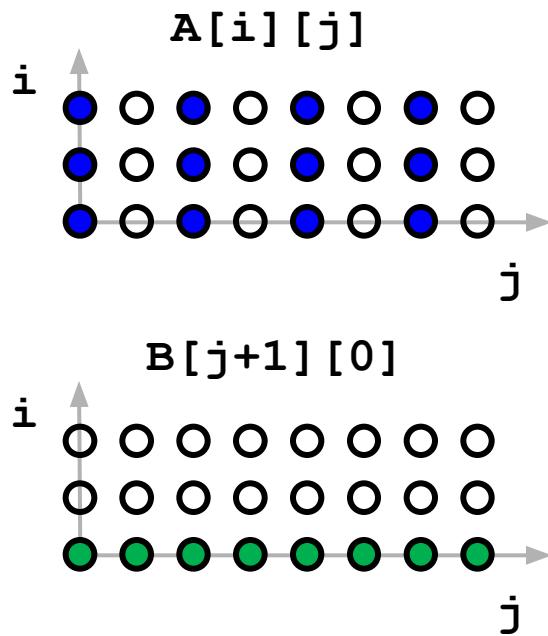
for (i = 95; i<100; i++) /* Epilog */
    a[i] = 0;
```

# Example Revisited

## Original Code

```
for (i = 0; i < 3; i++)
    for (j = 0; j < 100; j++)
        A[i][j] = B[j][0] + B[j+1][0];
```

- Cache Hit
- ● Cache Miss



## Code with Prefetching

```
prefetch(&A[0][0]);
for (j = 0; j < 6; j += 2) {
    prefetch(&B[j+1][0]);
    prefetch(&B[j+2][0]);
    prefetch(&A[0][j+1]);
}
for (j = 0; j < 94; j += 2) {
    prefetch(&B[j+7][0]);
    prefetch(&B[j+8][0]);
    prefetch(&A[0][j+7]);
    A[0][j] = B[j][0]+B[j+1][0];
    A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for (j = 94; j < 100; j += 2) {
    A[0][j] = B[j][0]+B[j+1][0];
    A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for (i = 1; i < 3; i++) {
    prefetch(&A[i][0]);
    for (j = 0; j < 6; j += 2)
        prefetch(&A[i][j+1]);
    for (j = 0; j < 94; j += 2) {
        prefetch(&A[i][j+7]);
        A[i][j] = B[j][0] + B[j+1][0];
        A[i][j+1] = B[j+1][0] + B[j+2][0];
    }
    for (j = 94; j < 100; j += 2) {
        A[i][j] = B[j][0] + B[j+1][0];
        A[i][j+1] = B[j+1][0] + B[j+2][0];
    }
}
```

# Prefetching Indirections

```
for (i = 0; i<100; i++)
    sum += A[index[i]];
```

## Analysis: what to prefetch

- both dense and **indirect** references
- difficult to predict whether indirections hit or miss

## Scheduling: when/how to issue prefetches

- modification of software pipelining algorithm

# Software Pipelining for Indirections

## Original Loop

```
for (i = 0; i<100; i++)
    sum += A[index[i]];
```

## Software Pipelined Loop (5 iterations ahead)

```
for (i = 0; i<5; i++)      /* Prolog 1 */
    prefetch(&index[i]);

for (i = 0; i<5; i++) {      /* Prolog 2 */
    prefetch(&index[i+5]);
    prefetch(&A[index[i]]);
}
for (i = 0; i<90; i++) { /* Steady State */
    prefetch(&index[i+10]);
    prefetch(&A[index[i+5]]);
    sum += A[index[i]];
}
for (i = 90; i<95; i++) { /* Epilog 1 */
    prefetch(&A[index[i+5]]);
    sum += A[index[i]];
}
for (i = 95; i<100; i++) /* Epilog 2 */
    sum += A[index[i]];
```

# Summary of Results

## Dense Matrix Code:

- eliminated 50% to 90% of memory stall time
- overheads remain low due to prefetching selectively
- significant improvements in overall performance (6 over 45%)

## Indirections, Sparse Matrix Code:

- expanded coverage to handle some important cases

# Prefetching for Arrays: Concluding Remarks

- Demonstrated that software prefetching is effective
  - selective prefetching to eliminate overhead
  - dense matrices and indirections / sparse matrices
  - uniprocessors and multiprocessors
- Hardware should focus on providing sufficient memory bandwidth

# Prefetching for Recursive Data Structures

# Recursive Data Structures

- Examples:
  - linked lists, trees, graphs, ...
- A common method of building large data structures
  - especially in non-numeric programs
- Cache miss behavior is a concern because:
  - large data set with respect to the cache size
  - temporal locality may be poor
  - little spatial locality among consecutively-accessed nodes

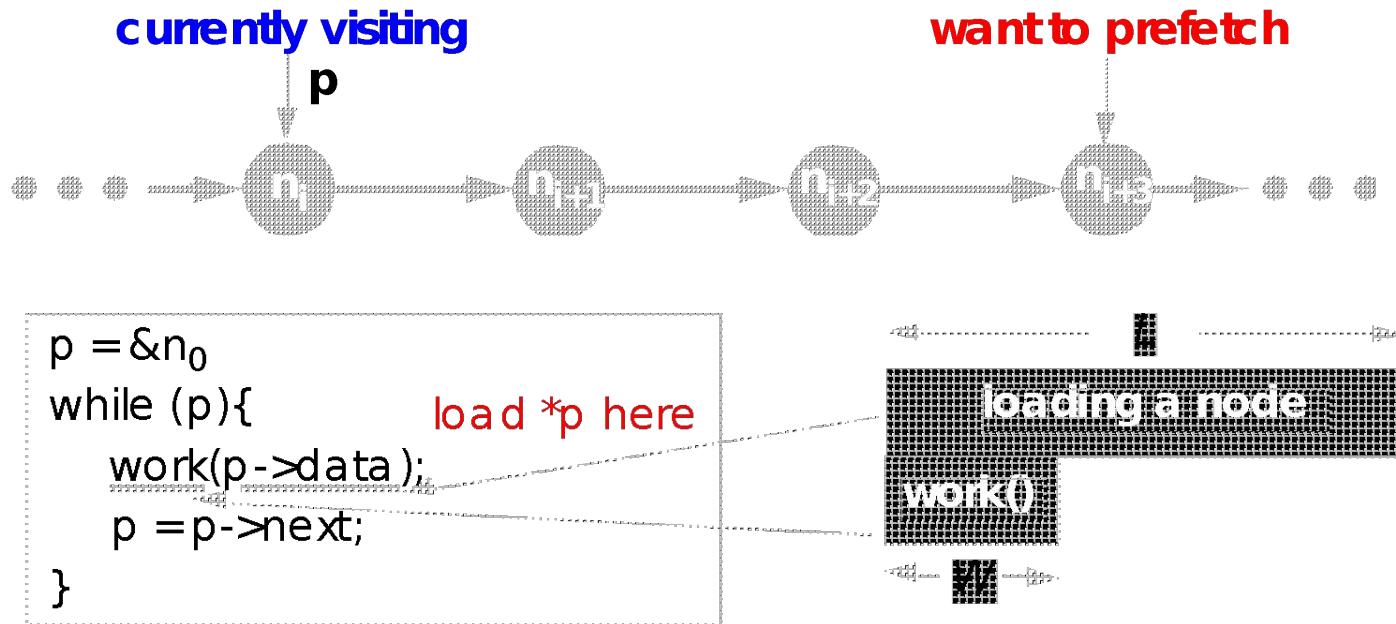
## Goal:

- Automatic Compiler-Based Prefetching for Recursive Data Structures

# Overview

- Challenges in Prefetching Recursive Data Structures
- Three Prefetching Algorithms
- Experimental Results
- Conclusions

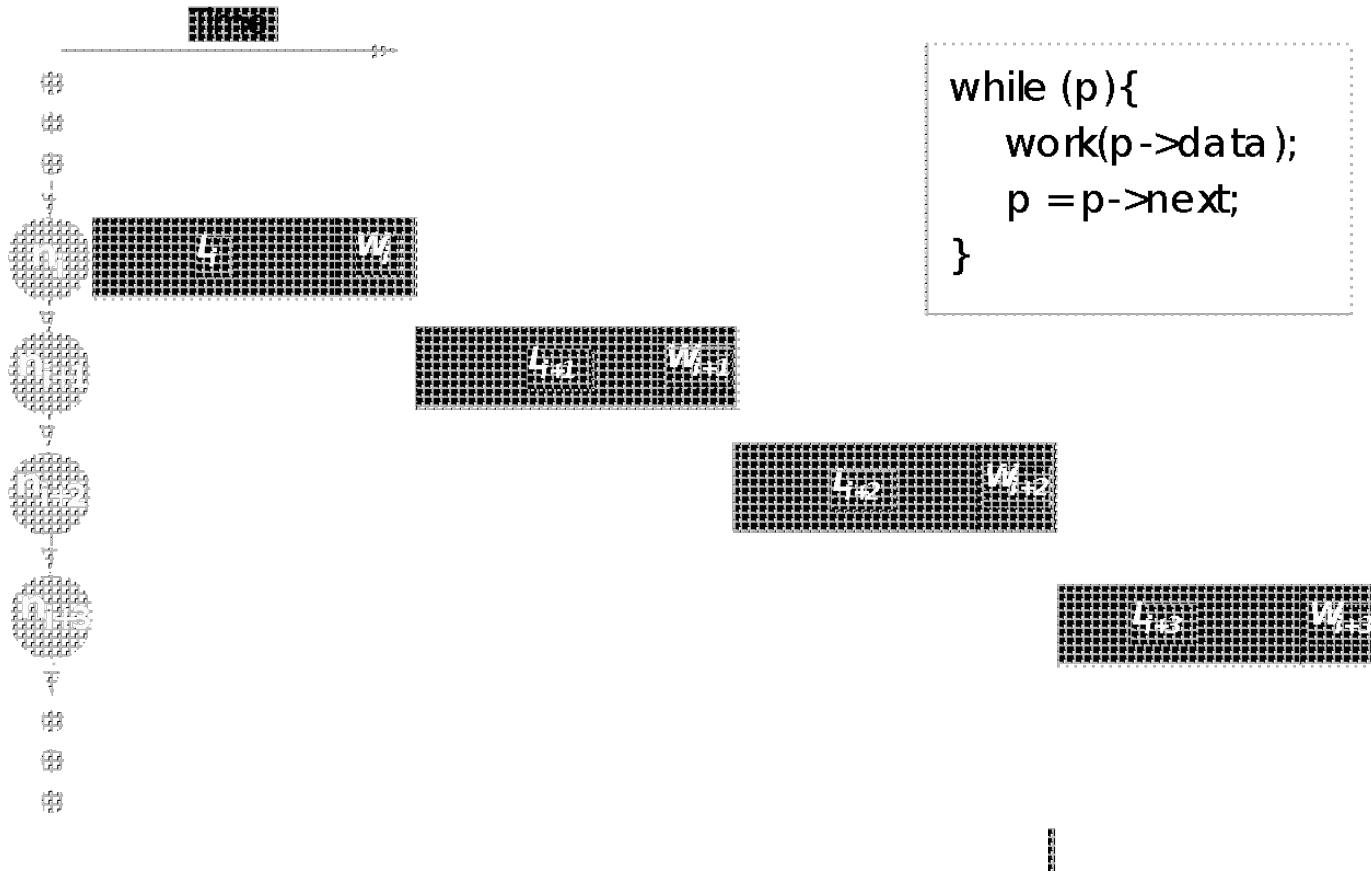
# Scheduling Prefetches for Recursive Data Structures



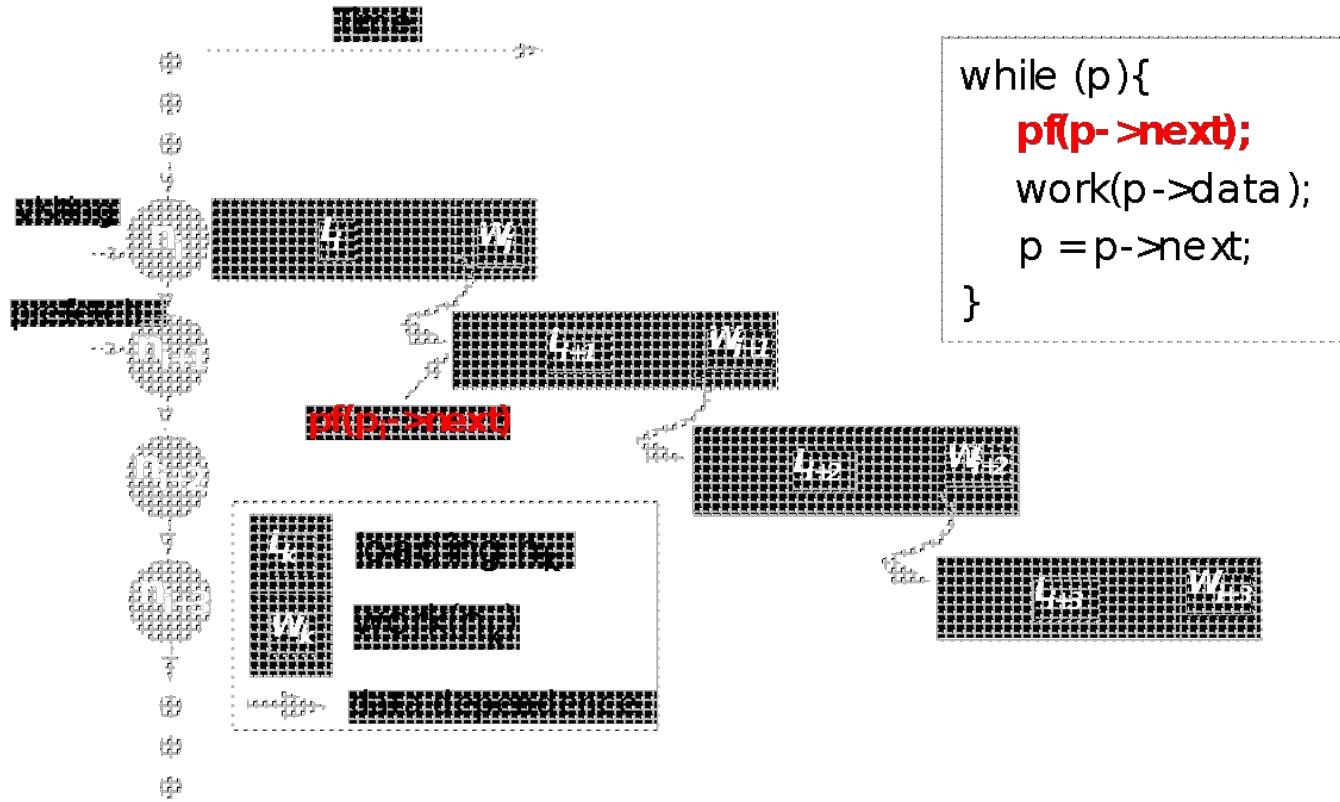
Our Goal: *fully hide latency*

- thus achieving fastest possible computation rate of  $1/W$
- e.g., if  $L = 3W$ , we must prefetch 3 nodes ahead to achieve this

# Performance without Prefetching

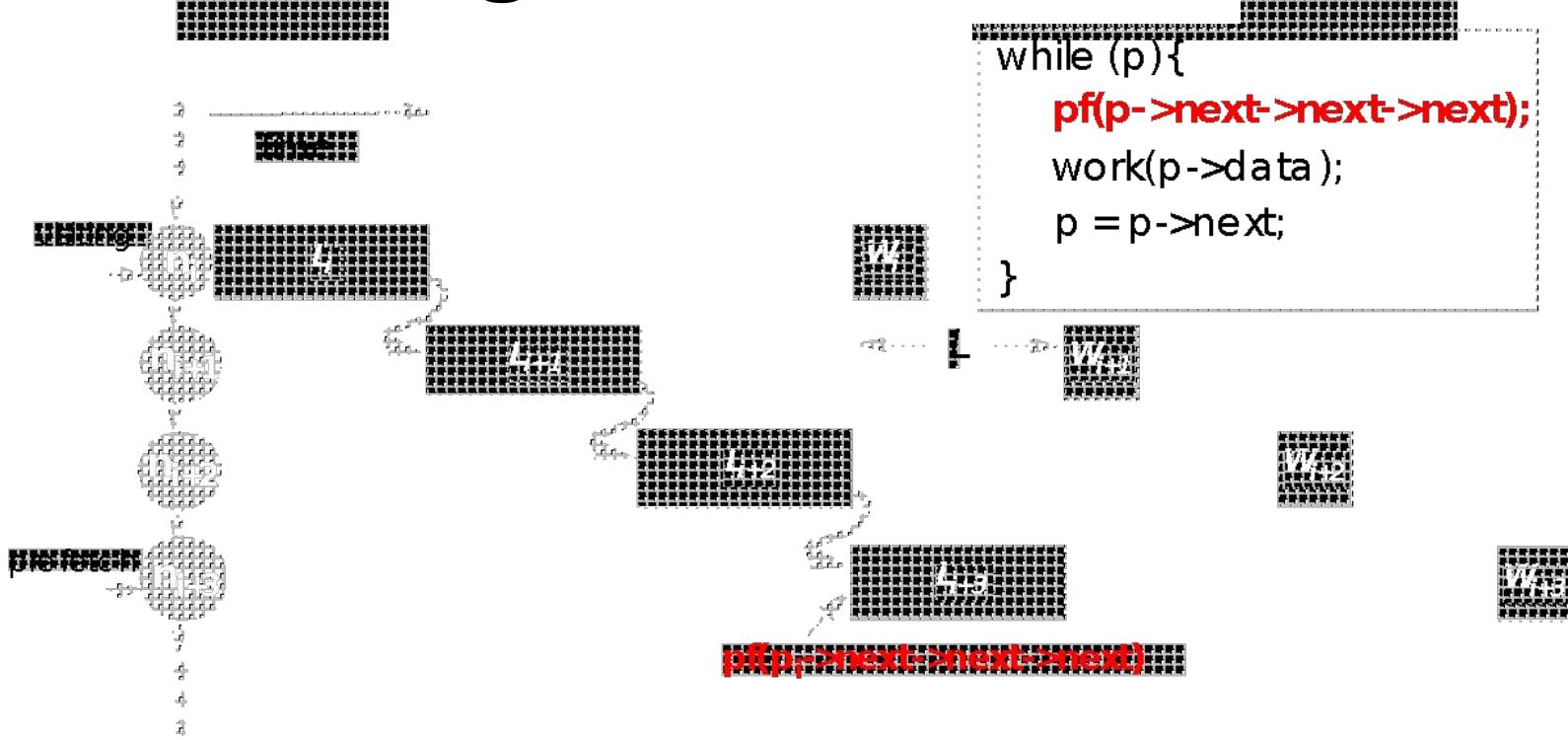


# Prefetching One Node Ahead



- Computation is overlapped with memory accesses  
 $\text{computation rate} = 1/L$

# Prefetching Three Nodes Ahead

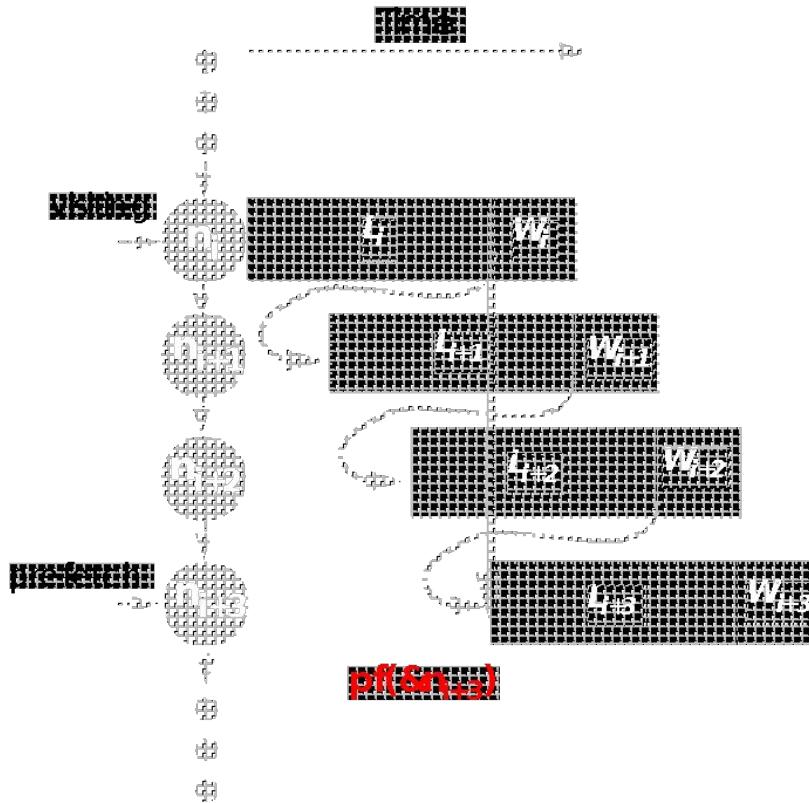


*computation rate does not improve (still = 1/L)!*

## Pointer-Chasing Problem:

- any scheme which follows the pointer chain is limited to a rate of  $1/L$

# Our Goal Fully Hide Latency



```
while (p){  
    pf(&ni+3);  
    work(p->data);  
    p = p->next;  
}
```

- achieves the fastest possible computation rate of  $1/W$

# Overview

- Challenges in Prefetching Recursive Data Structures
- Three Prefetching Algorithms
  - Greedy Prefetching
  - History-Pointer Prefetching
  - Data-Linearization Prefetching
- Experimental Results
- Conclusions

# Pointer-Chasing Problem

Key:

- $n_i$  needs to know  $\&n_{i+d}$  without referencing the  $d-1$  intermediate nodes

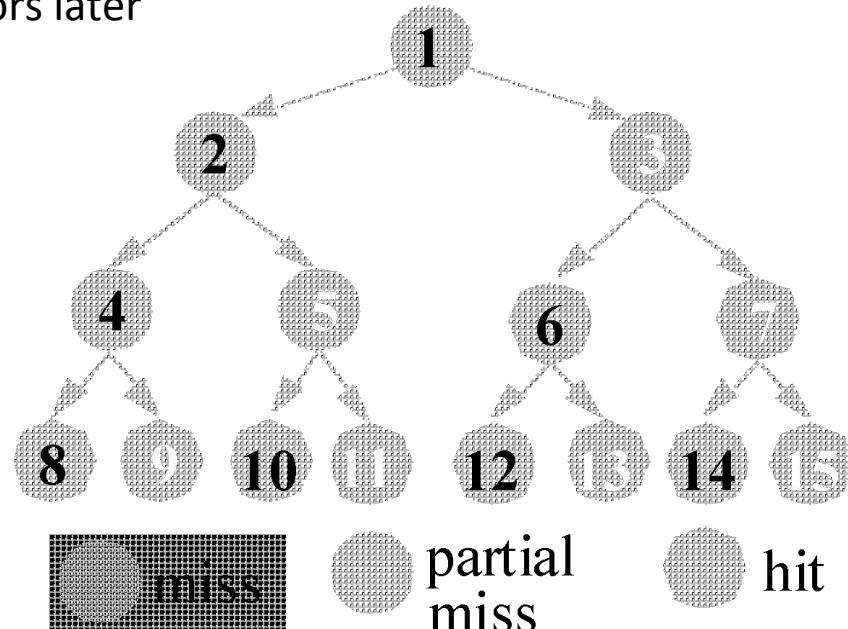
Our proposals:

- use *existing* pointer(s) in  $n_i$  to approximate  $\&n_{i+d}$ 
  - Greedy Prefetching
- add *new* pointer(s) to  $n_i$  to approximate  $\&n_{i+d}$ 
  - History-Pointer Prefetching
- compute  $\&n_{i+d}$  *directly* from  $\&n_i$  (no ptr deref)
  - History-Pointer Prefetching

# Greedy Prefetching

- Prefetch all neighboring nodes (simplified definition)
  - only one will be followed by the immediate control flow
  - hopefully, we will visit other neighbors later

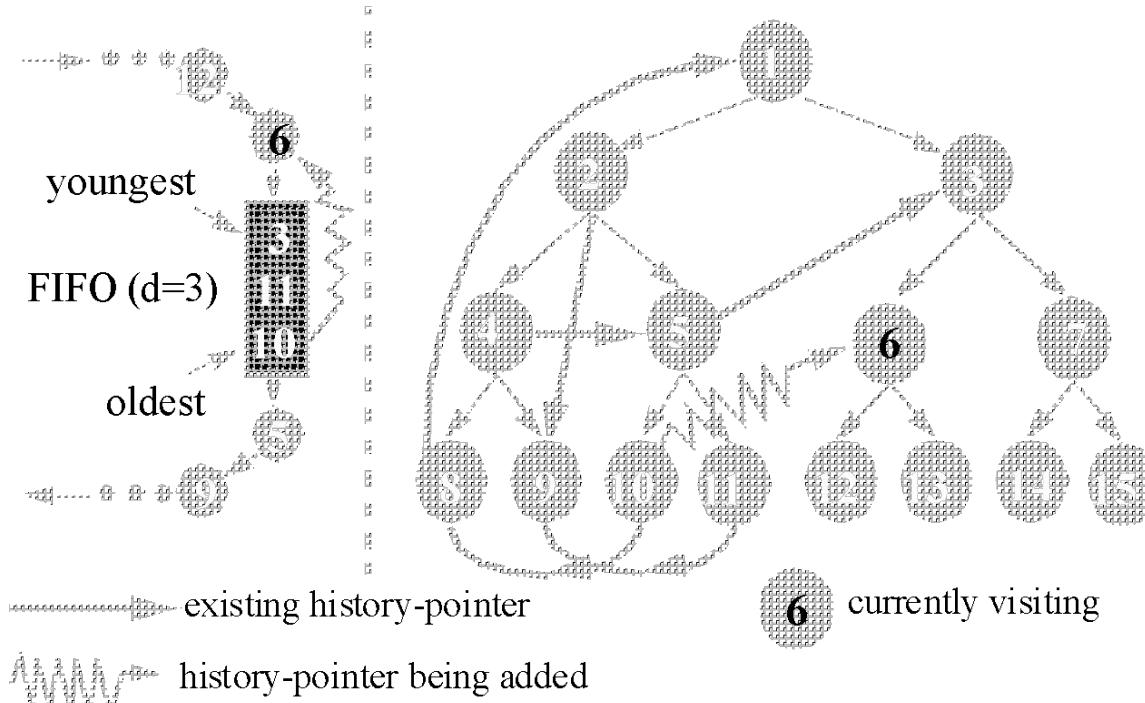
```
preorder(treeNode * t) {  
    if (t != NULL) {  
        pf(t->left);  
        pf(t->right);  
        process(t->data);  
        preorder(t->left);  
        preorder(t->right);  
    }  
}
```



- Reasonably effective in practice
- However, little control over the prefetching distance

# History-Pointer Prefetching

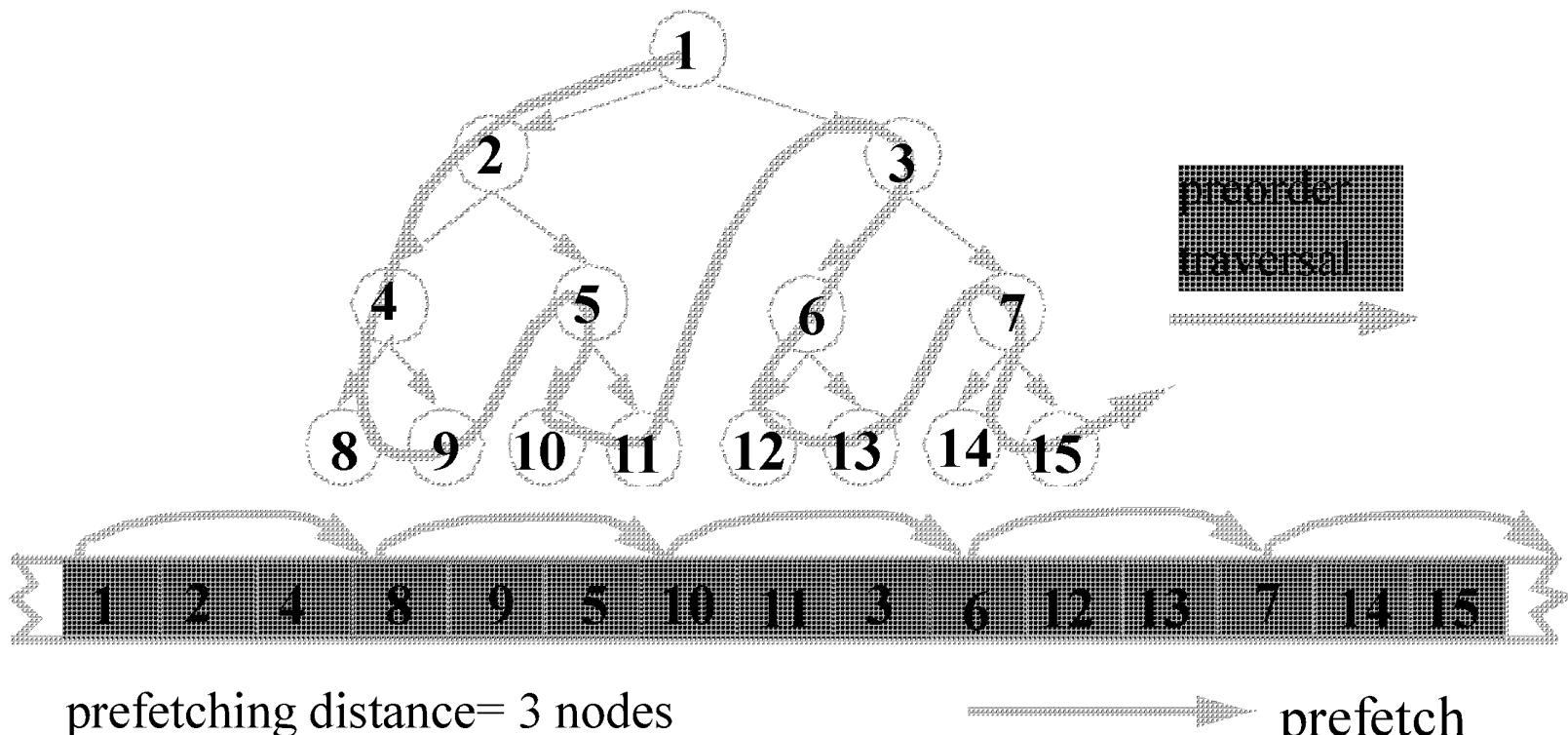
- Add new pointer(s) to each node
  - history-pointers are obtained from some recent traversal



- Trade space & time for better control over prefetching distances

# Data-Linearization Prefetching

- No pointer dereferences are required
- Map nodes close in the traversal to contiguous memory



# Summary of Prefetching Algorithms

	Greedy	History-Pointer	Data-Linearization
Control over Prefetching Distance	little	more precise	more precise
Applicability to Recursive Data Structures	any RDS	revisited; changes only slowly	must have a major traversal order; changes only slowly
Overhead in Preparing Prefetch Addresses	none	space + time	none in practice
Ease of Implementation	relatively straightforward	more difficult	more difficult

# Conclusions

- Propose 3 schemes to overcome the pointer-chasing problem:
  - Greedy Prefetching
  - History-Pointer Prefetching
  - Data-Linearization Prefetching
- Automated greedy prefetching in SUIF
  - improves performance significantly for half of Olden
  - memory feedback can further reduce prefetch overhead
- The other 2 schemes can outperform greedy in some situations

# CSC D70: Compiler Optimization Parallelization

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Tarek Abdelrahman*

# Announcements

- Final exam: **Monday, April 15,  
2:00-4:00pm; Room: IC200**
- Covers the whole semester
- Course evaluation (right now)

# Data Dependence

$$\begin{array}{ll} S_1 : & A = 1.0 \\ S_2 : & B = A + 2.0 \\ S_3 : & C = D \\ & \square \\ S_4 : & A = B/C \end{array}$$

We define four types of data dependence.

- **Flow (true) dependence**: a statement  $S_i$  precedes a statement  $S_j$  in execution and  $S_i$  computes a data value that  $S_j$  uses.
- Implies that  $S_i$  must execute before  $S_j$ .

$$S_i \delta^+ S_j \quad (S_1 \delta^+ S_2 \quad \text{and} \quad S_2 \delta^+ S_4)$$

# Data Dependence

$$\begin{array}{ll} S_1 : & A = 1.0 \\ S_2 : & B = A + 2.0 \\ S_3 : & C = D \\ & \square \\ S_4 : & A = B/C \end{array}$$

We define four types of data dependence.

- **Anti dependence**: a statement  $S_i$  precedes a statement  $S_j$  in execution and  $S_i$  uses a data value that  $S_j$  computes.
- It implies that  $S_i$  must be executed before  $S_j$ .

$$S_i \delta^a S_j \quad (S_2 \delta^a S_3)$$

# Data Dependence

$$\begin{array}{ll} S_1 : & A = 1.0 \\ S_2 : & B = A + 2.0 \\ S_3 : & C = D \\ & \vdots \\ S_4 : & A = B/C \end{array}$$

We define four types of data dependence.

- **Output dependence:** a statement  $S_i$  precedes a statement  $S_j$  in execution and  $S_i$  computes a data value that  $S_j$  also computes.
- It implies that  $S_i$  must be executed before  $S_j$ .

$$S_i \delta^o S_j \quad (S_1 \delta^o S_3 \quad \text{and} \quad S_3 \delta^o S_4)$$

# Data Dependence

$$\begin{array}{ll} S_1 : & A = 1.0 \\ S_2 : & B = A + 2.0 \\ S_3 : & C = D \\ & \vdots \\ S_4 : & A = B/C \end{array}$$

We define four types of data dependence.

- **Input dependence**: a statement  $S_i$  precedes a statement  $S_j$  in execution and  $S_i$  uses a data value that  $S_j$  also uses.
- Does this imply that  $S_i$  must execute before  $S_j$ ?

$$S_i \delta^I S_j \quad (S_3 \delta^I S_4)$$

# Data Dependence (continued)

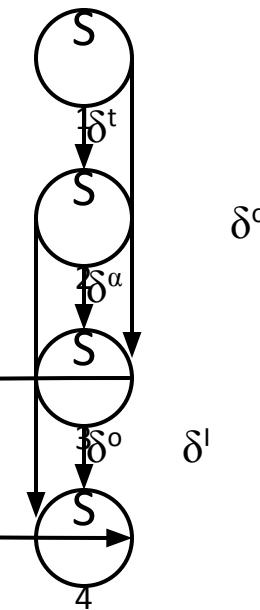
- The dependence is said to **flow** from  $S_i$  to  $S_j$  because  $S_i$  precedes  $S_j$  in execution.
- $S_i$  is said to be the **source** of the dependence.  $S_j$  is said to be the **sink** of the dependence.
- The only “true” dependence is flow dependence; it represents the flow of data in the program.
- The other types of dependence are caused by programming style; they may be eliminated by re-naming.

$$\begin{array}{ll} S_1 : & A = 1.0 \\ S_2 : & B = A + 2.0 \\ S_3 : & A1 = C - D \\ & \square \\ S_4 : & A2 = B/C \end{array}$$

# Data Dependence (continued)

- Data dependence in a program may be represented using a **dependence graph**  $G=(V,E)$ , where the nodes  $V$  represent statements in the program and the directed edges  $E$  represent dependence relations.

$S_1 : A = 1.0$   
 $S_2 : B = A + 2.0$   
 $S_3 : A = C - D$   
 $S_4 : A = B/C$



# Value or Location?

- There are two ways a dependence is defined:  
**value-oriented** or **location-oriented**.

$$S_1 : A = 1.0$$

$$S_2 : B = A + 2.0$$

$$S_3 : A = C - D$$

□

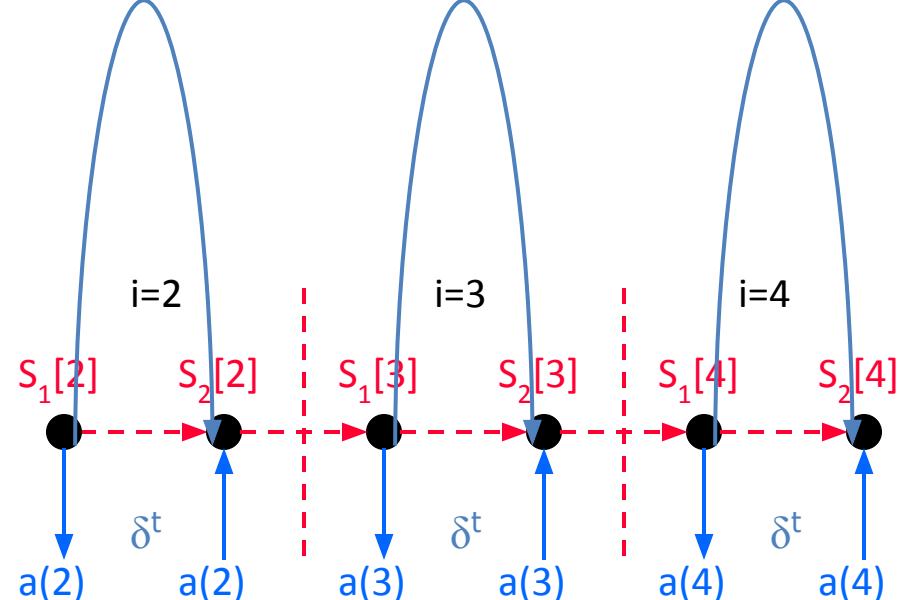
$$S_4 : A = B/C$$

# Example 1

```

do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i)
end do

```



- There is an instance of  $S_1$  that precedes an instance of  $S_2$  in execution and  $S_1$  produces data that  $S_2$  consumes.
- $S_1$  is the **source** of the dependence;  $S_2$  is the **sink** of the dependence.
- The dependence flows between instances of statements in the same iteration (**loop-independent** dependence).
- The number of iterations between source and sink (**dependence distance**) is 0. The **dependence direction** is =.

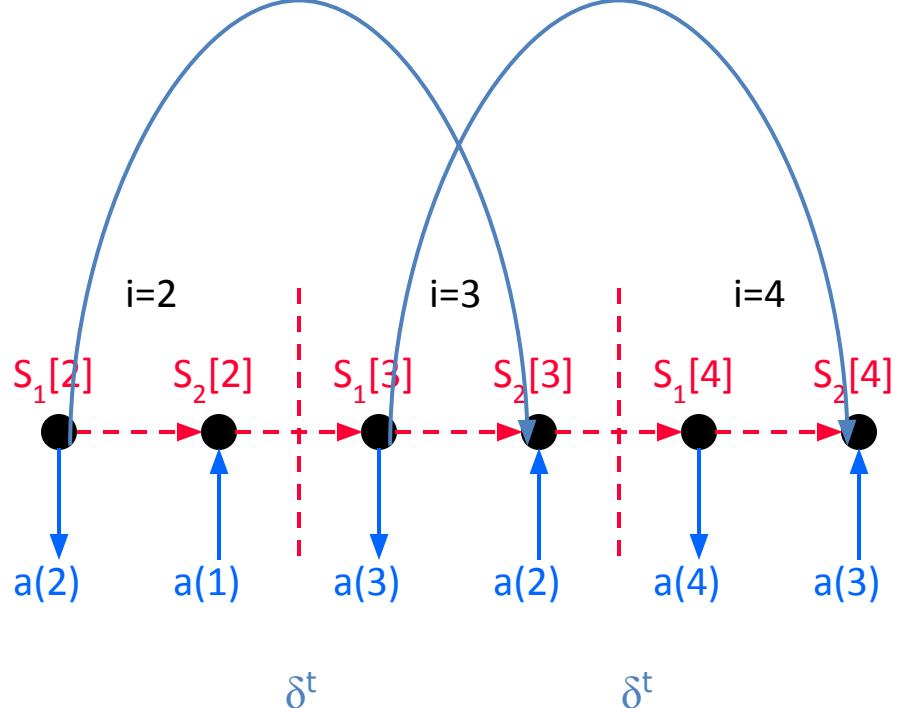
$$S_1 \delta_{=}^\dagger S_2 \quad \text{or} \quad S_1 \delta_0^\dagger S_2$$

# Example 2

```

do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i-1)
end do

```



- There is an instance of  $S_1$  that precedes an instance of  $S_2$  in execution and  $S_1$  produces data that  $S_2$  consumes.
- $S_1$  is the source of the dependence;  $S_2$  is the sink of the dependence.
- The dependence flows between instances of statements in different iterations (**loop-carried** dependence).
- The dependence distance is 1. The direction is positive ( $<$ ).

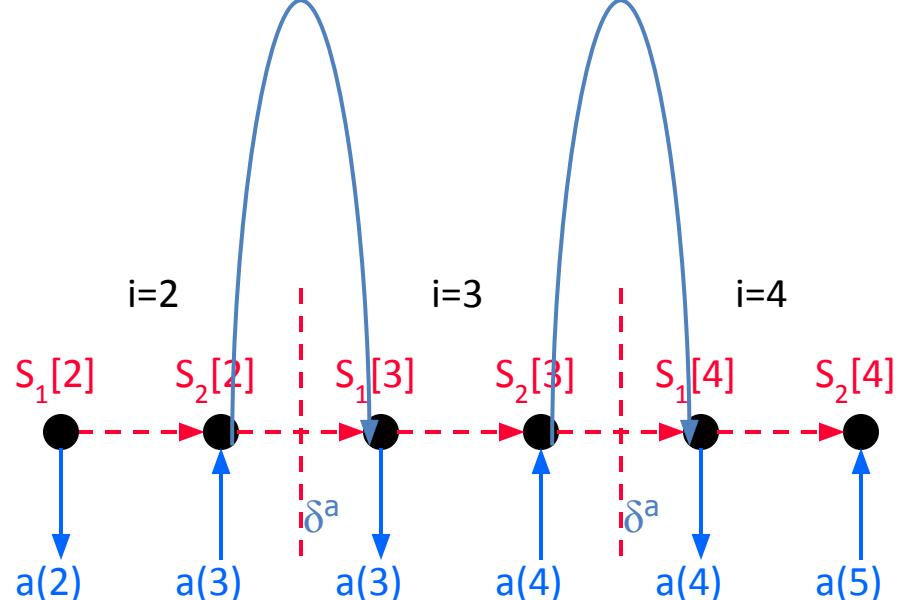
$$S_1 \delta_{<}^\dagger S_2 \quad \text{or} \quad S_1 \delta_1^\dagger S_2$$

# Example 3

```

do i = 2, 4
S1: a(i) = b(i) + c(i)
S2: d(i) = a(i+1)
end do

```



- There is an instance of  $S_2$  that precedes an instance of  $S_1$  in execution and  $S_2$  consumes data that  $S_1$  produces.
- $S_2$  is the source of the dependence;  $S_1$  is the sink of the dependence.
- The dependence is loop-carried.
- The dependence distance is 1.

$$S_2 \delta_{<}^a S_1 \quad \text{or} \quad S_2 \delta_1^a S_1$$

- Are you sure you know why it is  $S_2 \delta_{<}^a S_1$  even though  $S_1$  appears before  $S_2$  in the code?

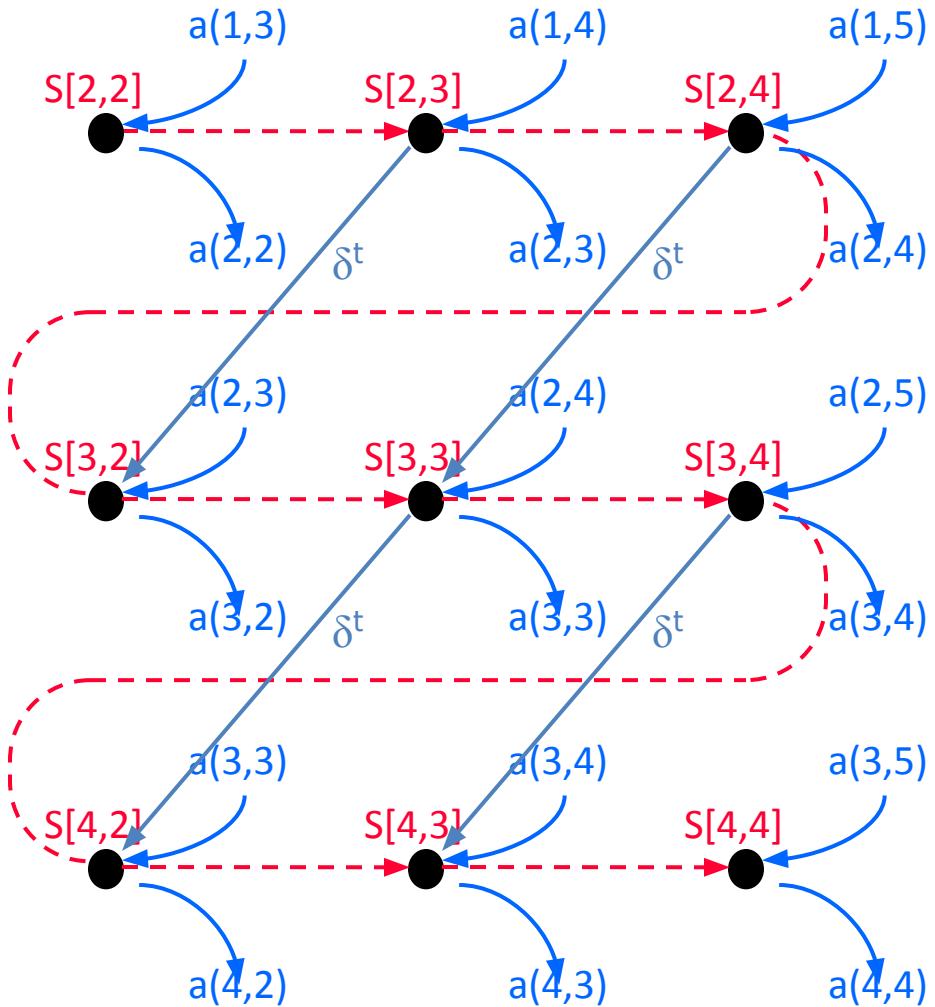
# Example 4

```

do i = 2, 4
    do j = 2, 4
        S:   a(i,j) = a(i-1,j+1)
            end do
        end do
    
```

- An instance of S precedes another instance of S and S produces data that S consumes.
- S is both source and sink.
- The dependence is loop-carried.
- The dependence distance is  $(1, -1)$ .

$$S\delta_{(<,>)}^+ S \quad \text{or} \quad S\delta_{(1,-1)}^+ S$$



# Problem Formulation

- Consider the following **perfect nest** of depth  $d$ :

```

do I1 = L1, U1
  do I2 = L2, U2
    ...
    do Id = Ld, Ud
      a(f1(I), f2(I), ..., fm(I)) = ...
      ...
      = a(g1(I), g2(I), ..., gm(I))
    enddo
    ...
  enddo
enddo

```

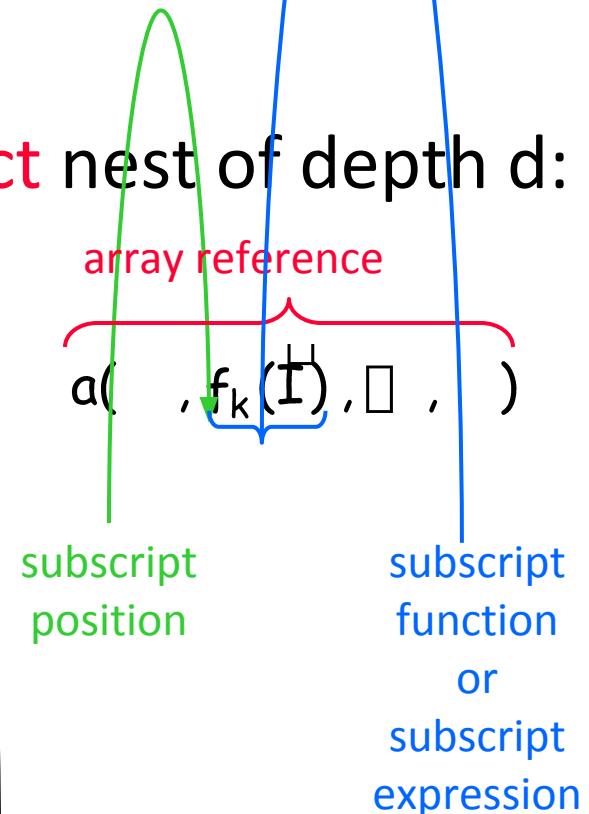
$$I = (I_1, I_2, \dots, I_d)$$

$$L = (L_1, L_2, \dots, L_d)$$

$$U = (U_1, U_2, \dots, U_d)$$

$$L \leq U$$

-14-



linear functions

$$b_0 + b_1 I_1 + b_2 I_2 + \dots + b_d I_d$$

# Problem Formulation

- Dependence will exist if there exists two iteration vectors  $\underline{k}$  and  $\underline{j}$  such that  $\underline{\ell} \leq \underline{k} \leq \underline{j} \leq \underline{U}$  and:

and  $f_1(\underline{k}) = g_1(\underline{j})$   
and  $f_2(\underline{k}) = g_2(\underline{j})$   
and  $f_m(\underline{k}) = g_m(\underline{j})$

- That is:

and  $f_1(\underline{k}) - g_1(\underline{j}) = 0$   
and  $f_2(\underline{k}) - g_2(\underline{j}) = 0$   
and  $f_m(\underline{k}) - g_m(\underline{j}) = 0$

# Problem Formulation - Example

```
do i = 2, 4  
S1: a(i) = b(i) + c(i)  
S2: d(i) = a(i-1)  
end do
```

- Does there exist two iteration vectors  $i_1$  and  $i_2$ , such that  $2 \leq i_1 \leq i_2 \leq 4$  and such that:

$$i_1 = i_2 - 1?$$

- Answer: yes;  $i_1=2$  &  $i_2=3$  and  $i_1=3$  &  $i_2=4$ .
- Hence, there is dependence!
- The dependence distance vector is  $i_2 - i_1 = 1$ .
- The dependence direction vector is  $\text{sign}(1) = <$ .

# Problem Formulation - Example

```
do i = 2, 4  
S1: a(i) = b(i) + c(i)  
S2: d(i) = a(i+1)  
end do
```

- Does there exist two iteration vectors  $i_1$  and  $i_2$ , such that

$2 \leq i_1 \leq i_2 \leq 4$  and such that:

$$i_1 = i_2 + 1?$$

- Answer: yes;  $i_1=3$  &  $i_2=2$  and  $i_1=4$  &  $i_2=3$ . (But, but!).
- Hence, there is dependence!
- The dependence distance vector is  $i_2 - i_1 = -1$ .
- The dependence direction vector is  $\text{sign}(-1) = >$ .
- Is this possible?

# Problem Formulation - Example

```
do i = 1, 10  
S1: a(2*i) = b(i) + c(i)  
S2: d(i) = a(2*i+1)  
end do
```

- Does there exist two iteration vectors  $i_1$  and  $i_2$ , such that  
 $1 \leq i_1 \leq i_2 \leq 10$  and such that:

$$2*i_1 = 2*i_2 + 1?$$

- Answer: no;  $2*i_1$  is even &  $2*i_2 + 1$  is odd.
- Hence, there is no dependence!

# Problem Formulation

- Dependence testing is equivalent to an **integer linear programming** (ILP) problem of 2d variables & m+d constraint!
- An algorithm that determines if there exists two iteration vectors  $\vec{j}$  and  $\vec{k}$  that satisfies these constraints is called a **dependence tester**.
- The dependence distance vector is given by  $\vec{j} - \vec{k}$
- The dependence direction vector is given by  $\text{sign}(\vec{j} - \vec{k})$ .
- Dependence testing is NP-complete!
- A dependence test that reports dependence only when there is dependence is said to be **exact**. Otherwise it is **in-exact**.
- A dependence test must be **conservative**; if the existence of dependence cannot be ascertained, dependence must be assumed.

# Dependence Testers

- Lamport's Test.
- GCD Test.
- Banerjee's Inequalities.
- Generalized GCD Test.
- Power Test.
- I-Test.
- Omega Test.
- Delta Test.
- Stanford Test.
- etc...

# Lamport's Test

- Lamport's Test is used when there is a single index variable in the subscript expressions, and when the coefficients of the index variable in both expressions are the same.

$$A(\square, b^*i + c_1, \square) = \square \\ \square = A(\square, b^*i + c_2, \square)$$

- The dependence problem: does there exist  $i_1$  and  $i_2$ , such that  $L_i \leq i_1 \leq i_2 \leq U_i$  and such that

$$b^*i_1 + c_1 = b^*i_2 + c_2 \quad \text{or} \quad i_2 - i_1 = \frac{c_1 - c_2}{b}?$$

- There is integer solution if and only if  $\frac{c_1 - c_2}{b}$  is integer.
- The dependence distance is  $d = \frac{c_1 - c_2}{b}$  if  $L_i \leq |d| \leq U_i$ .
- $d > 0 \Rightarrow$  true dependence.
- $d = 0 \Rightarrow$  loop independent dependence.
- $d < 0 \Rightarrow$  anti dependence.

# Lamport's Test - Example

```
do i = 1, n  
  do j = 1, n  
    S:   a(i,j) = a(i-1,j+1)  
    end do  
  end do
```

$$i_1 = i_2 - 1?$$

$$b = 1; c_1 = 0; c_2 = -1$$

$$\frac{c_1 - c_2}{b} = 1$$

There is dependence.  
Distance (i) is 1.

$$j_1 = j_2 + 1?$$

$$b = 1; c_1 = 0; c_2 = 1$$

$$\frac{c_1 - c_2}{b} = -1$$

There is dependence.  
Distance (j) is -1.

$$S \delta_{(1,-1)}^{\dagger} S \quad \text{or} \quad S \delta_{(<,>)}^{\dagger} S$$

# Lamport's Test - Example

```
do i = 1, n  
  do j = 1, n  
    S:   a(i,2*j) = a(i-1,2*j+1)  
    end do  
  end do
```

$$i_1 = i_2 - 1?$$

$$b = 1; c_1 = 0; c_2 = -1$$

$$\frac{c_1 - c_2}{b} = 1$$

There is dependence.  
Distance (i) is 1.

$$2*j_1 = 2*j_2 + 1?$$

$$b = 2; c_1 = 0; c_2 = 1$$

$$\frac{c_1 - c_2}{b} = -\frac{1}{2}$$

There is no dependence.

?

There is no dependence!

# GCD Test

- Given the following equation:

$$\sum_{i=1}^n a_i x_i = c \quad a_i's \text{ and } c \text{ are integers}$$

an integer solution exists if and only if:

$$\gcd(a_1, a_2, \dots, a_n) \text{ divides } c$$

- Problems:
  - ignores loop bounds.
  - gives no information on distance or direction of dependence.
  - often  $\gcd(\dots)$  is 1 which always divides c, resulting in false dependences.

# GCD Test - Example

```
do i = 1, 10  
S1: a(2*i) = b(i) + c(i)  
S2: d(i) = a(2*i-1)  
end do
```

- Does there exist two iteration vectors  $i_1$  and  $i_2$ , such that  $1 \leq i_1 \leq i_2 \leq 10$  and such that:

$$2*i_1 = 2*i_2 - 1?$$

or

$$2*i_2 - 2*i_1 = 1?$$

- There will be an integer solution if and only if  $\text{gcd}(2, -2)$  divides 1.
- This is not the case, and hence, there is no dependence!

# GCD Test Example

```
do i = 1, 10  
S1: a(i) = b(i) + c(i)  
S2: d(i) = a(i-100)  
end do
```

- Does there exist two iteration vectors  $i_1$  and  $i_2$ , such that  $1 \leq i_1 \leq i_2 \leq 10$  and such that:

$$i_1 = i_2 - 100?$$

or

$$i_2 - i_1 = 100?$$

- There will be an integer solution if and only if  $\gcd(1, -1)$  divides 100.
- This is the case, and hence, there is dependence! Or is there?

# Dependence Testing Complications

- Unknown loop bounds.

```
do i = 1, N  
S1: a(i) = a(i+10)  
      end do
```

What is the relationship between N and 10?

- Triangular loops.

```
do i = 1, N  
      do j = 1, i-1  
S:    a(i,j) = a(j,i)  
      end do  
      end do
```

Must impose  $j < i$  as an additional constraint.

# More Complications

- User variables

```
do i = 1, 10  
S1: a(i) = a(i+k)  
end do
```

```
do i = L, H  
S1: a(i) = a(i-1)  
end do
```

Same problem as unknown loop bounds, but occur due to some loop transformations (e.g., normalization).



```
do i = 1, H-L  
S1: a(i+L) = a(i+L-1)  
end do
```

# More Complications: Scalars

```
do i = 1, N  
S1: x = a(i)  
S2: b(i) = x  
end do
```

⇒

```
do i = 1, N  
S1: x(i) = a(i)  
S2: b(i) = x(i)  
end do
```

```
j = N-1  
do i = 1, N  
S1: a(i) = a(j)  
S2: j = j - 1  
end do
```

⇒

```
do i = 1, N  
S1: a(i) = a(N-i)  
end do
```

```
sum = 0  
do i = 1, N  
S1: sum = sum + a(i)  
end do
```

⇒

```
do i = 1, N  
S1: sum(i) = a(i)  
end do  
sum += sum(i) i = 1, N
```

# Serious Complications

- Aliases.
  - Equivalence Statements in Fortran:

```
real a(10,10), b(10)
```

makes b the same as the first column of a.

- Common blocks: Fortran's way of having shared/global variables.

```
common /shared/a,b,c  
      :  
      :
```

```
subroutine foo (...)
```

```
common /shared/a,b,c
```

```
common /shared/x,y,z
```

# Loop Parallelization

- A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1  
  do j = 2, m-1  
    a(i, j) = ...  
    ...           = a(i, j)
```

```
b(i, j) = ...  
...           = b(i, j-1)
```

```
c(i, j) = ...  
...           = c(i-1, j)  
end do  
end do
```

# Loop Parallelization

A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1
    do j = 2, m-1
         $\delta_{=,=}^\dagger$       a(i, j) = ...
        ...                  = a(i, j)

        b(i, j) = ...
        ...      = b(i, j-1)

        c(i, j) = ...
        ...      = c(i-1, j)
    end do
end do
```

# Loop Parallelization

A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1  
  do j = 2, m-1  
    a(i, j) = ...  
    ...           = a(i, j)
```

$$\delta_{=,<}^\dagger \quad \begin{aligned} b(i, j) &= \dots \\ \dots &= b(i, j-1) \end{aligned}$$

```
c(i, j) = ...  
...       = c(i-1, j)  
end do  
end do
```

# Loop Parallelization

A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1  
    do j = 2, m-1  
        a(i, j) = ...  
        ...           = a(i, j)
```

```
b(i, j) = ...  
...      = b(i, j-1)
```

$$\delta_{<=}^{\dagger} \quad \begin{array}{l} c(i, j) = ... \\ ... = c(i-1, j) \end{array}$$

```
end do  
end do
```

# Loop Parallelization

A dependence is said to be **carried** by a loop if the loop is the outmost loop whose removal eliminates the dependence. If a dependence is not carried by the loop, it is **loop-independent**.

```
do i = 2, n-1  
    do j = 2, m-1  
         $\delta_{=,=}^\dagger$       a(i, j) = ...  
                          ...        = a(i, j)
```

```
 $\delta_{=,<}^\dagger$       b(i, j) = ...  
                          ...        = b(i, j-1)
```

```
 $\delta_{<,=}^\dagger$       c(i, j) = ...  
                          ...        = c(i-1, j)
```

```
end do  
end do
```

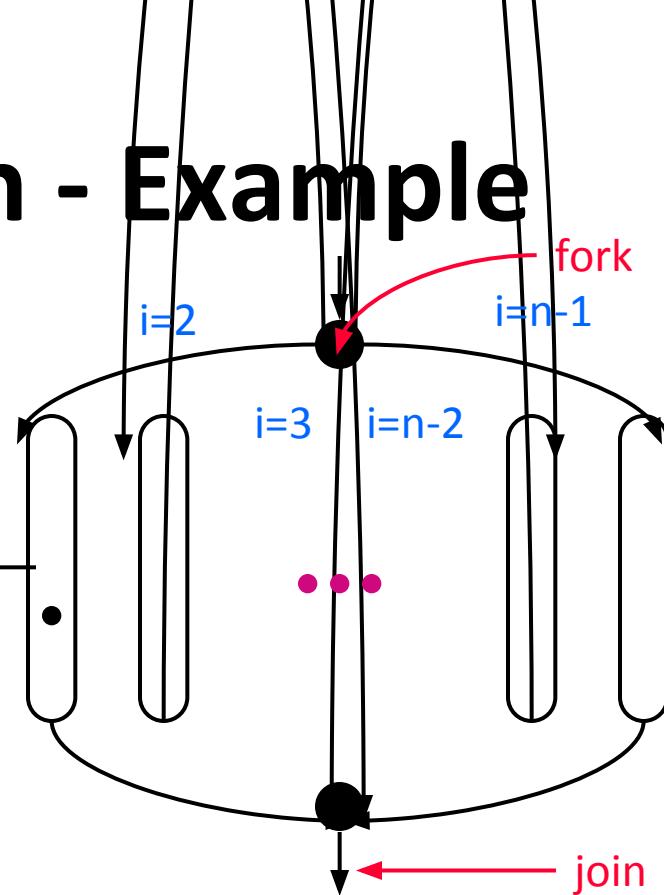
- Outermost loop with a non “=” direction carries dependence!

# Loop Parallelization

The iterations of a loop may be executed in parallel with one another if and only if no dependences are carried by the loop!

# Loop Parallelization - Example

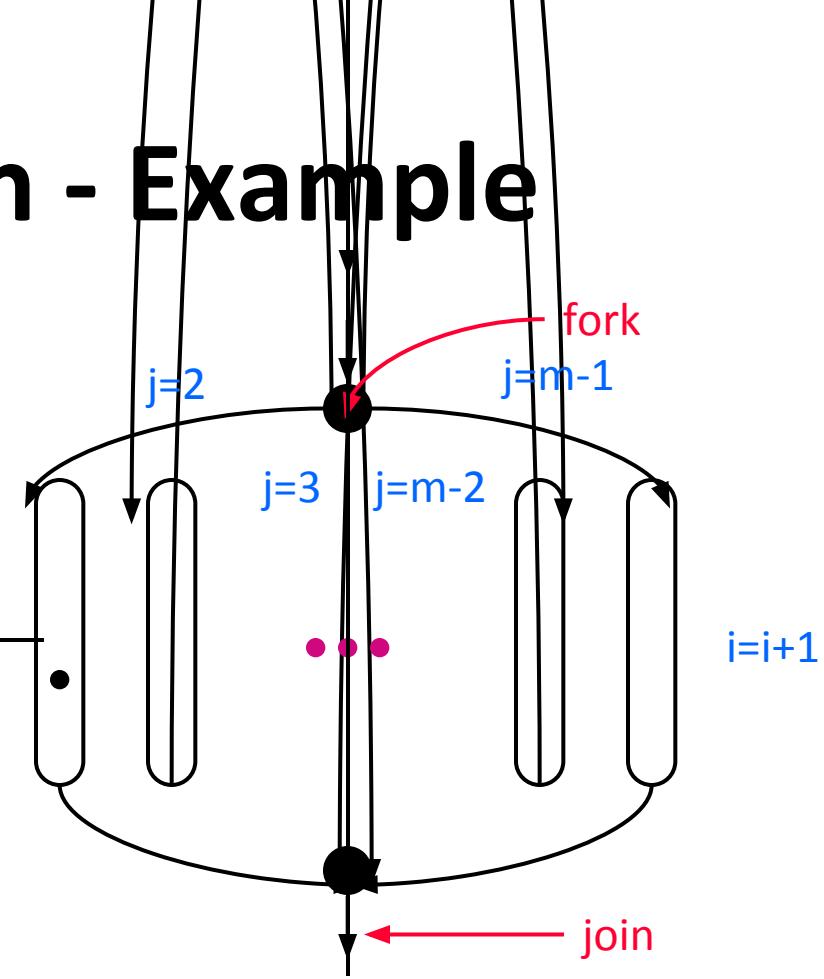
```
do i = 2, n-1
    do j = 2, m-1
        b(i, j) = ...
        ... = b(i, j-1)
    end do
end do
```



- Iterations of loop  $j$  must be executed sequentially, but the iterations of loop  $i$  may be executed in parallel.
- Outer loop parallelism.

# Loop Parallelization - Example

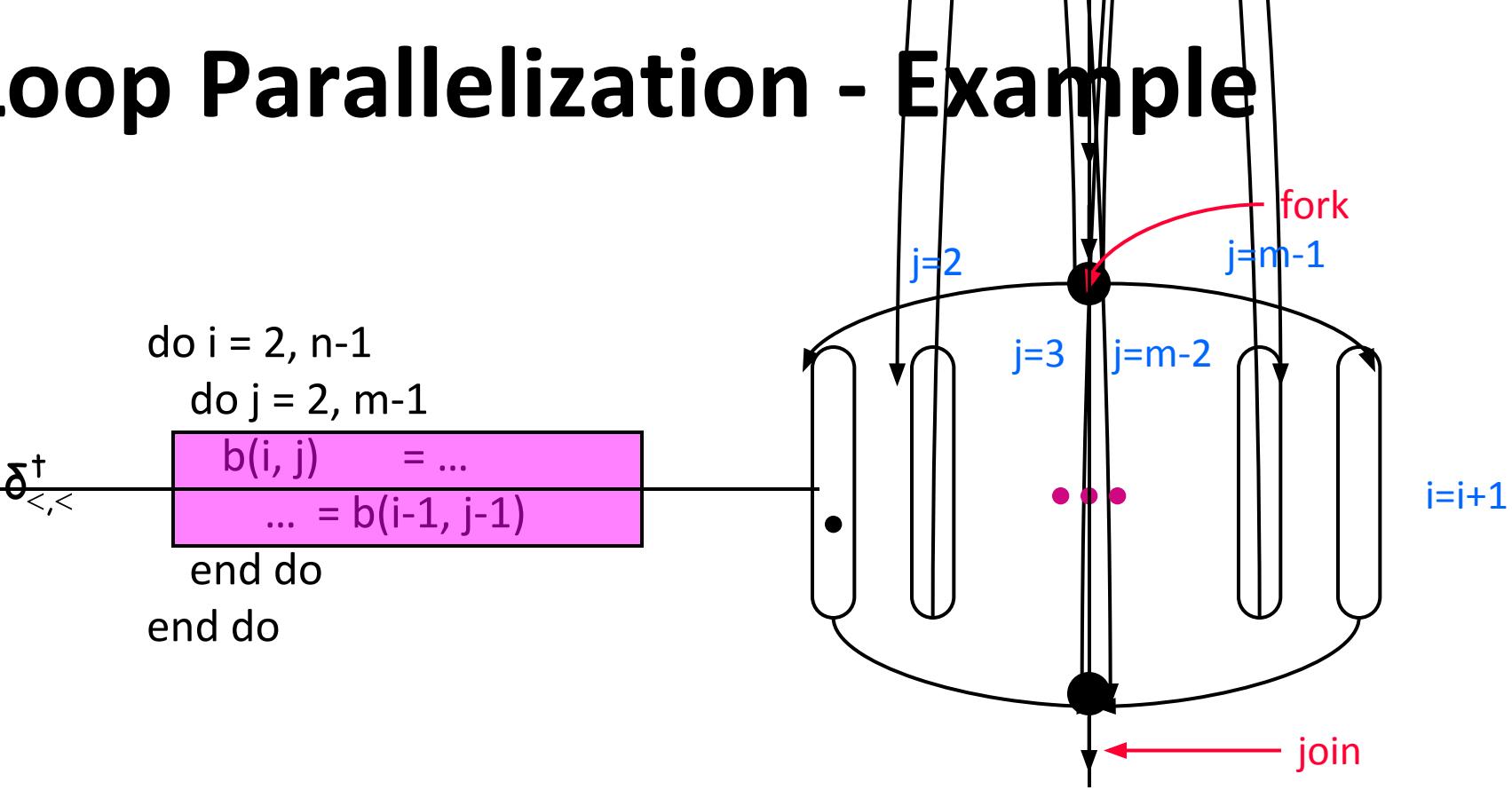
```
do i = 2, n-1  
    do j = 2, m-1  
        b(i, j) = ...  
        ... = b(i-1, j)  
    end do  
end do
```

 $\delta_{<,=}^t$ 

- Iterations of loop  $i$  must be executed sequentially, but the iterations of loop  $j$  may be executed in parallel.
- Inner loop parallelism.

# Loop Parallelization - Example

```
do i = 2, n-1
    do j = 2, m-1
        b(i, j) = ...
        ... = b(i-1, j-1)
    end do
end do
```

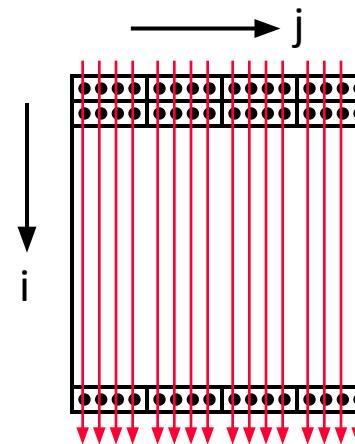
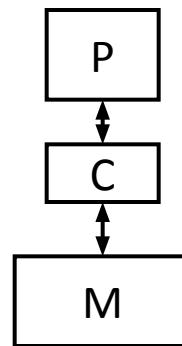


- Iterations of loop  $i$  must be executed sequentially, but the iterations of loop  $j$  may be executed in parallel.  
**Why?**
- Inner loop parallelism.

# Loop Interchange

Loop interchange changes the order of the loops to improve the spatial locality of a program.

```
do j = 1, n  
  do i = 1, n  
    ... a(i,j) ...  
  end do  
end do
```

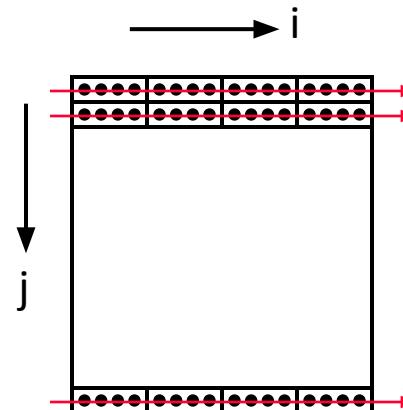
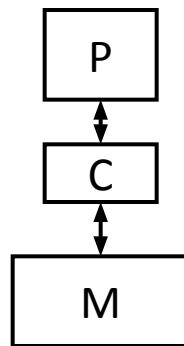


# Loop Interchange

Loop interchange changes the order of the loops to improve the spatial locality of a program.

```
do j = 1, n  
  do i = 1, n  
    ... a(i,j) ...  
  end do  
end do
```

```
do i = 1, n  
  do j = 1, n  
    ... a(i,j) ...  
  end do  
end do
```



# Loop Interchange

- Loop interchange can improve the granularity of parallelism!

```
do i = 1, n  
  do j = 1, n  
    a(i,j) = b(i,j)  
    c(i,j) = a(i-1,j)  
  end do  
end do
```

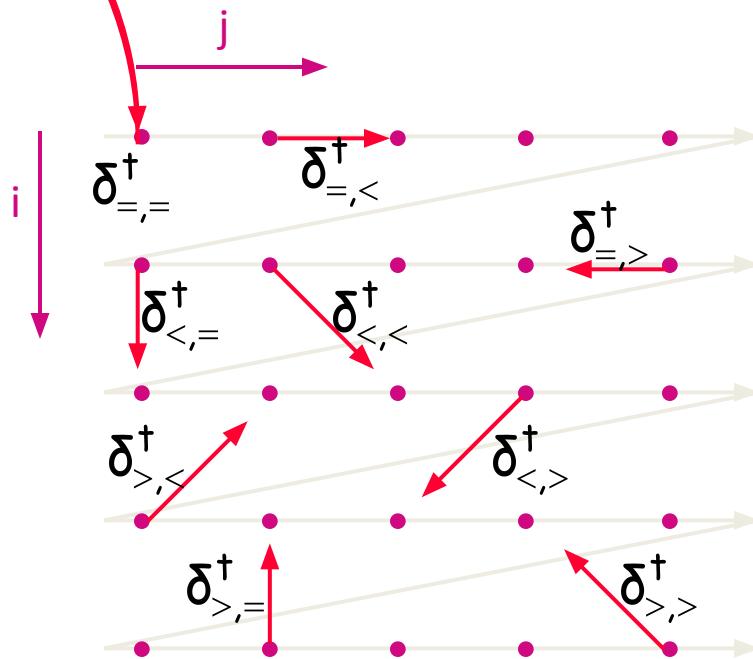
```
| do j = 1, n  
|   do i = 1, n  
|     a(i,j) = b(i,j)  
|     c(i,j) = a(i-1,j)  
|   end do  
| end do
```

$$\delta_{<,=}^\dagger$$

$$\delta_{=,<}^\dagger$$
  


# Loop Interchange

```
do i = 1,n  
  do j = 1,n  
    ... a(i,j) ...  
  end do  
end do
```

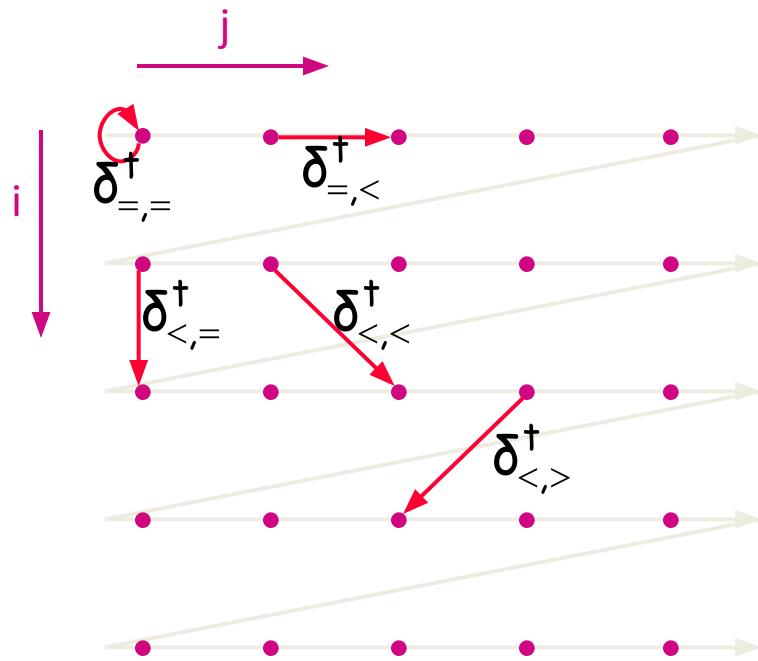


```
do j = 1,n  
  do i = 1,n  
    ... a(i,j) ...  
  end do  
end do
```

- When is loop interchange legal?

# Loop Interchange

```
do i = 1,n  
  do j = 1,n  
    ... a(i,j) ...  
  end do  
end do
```

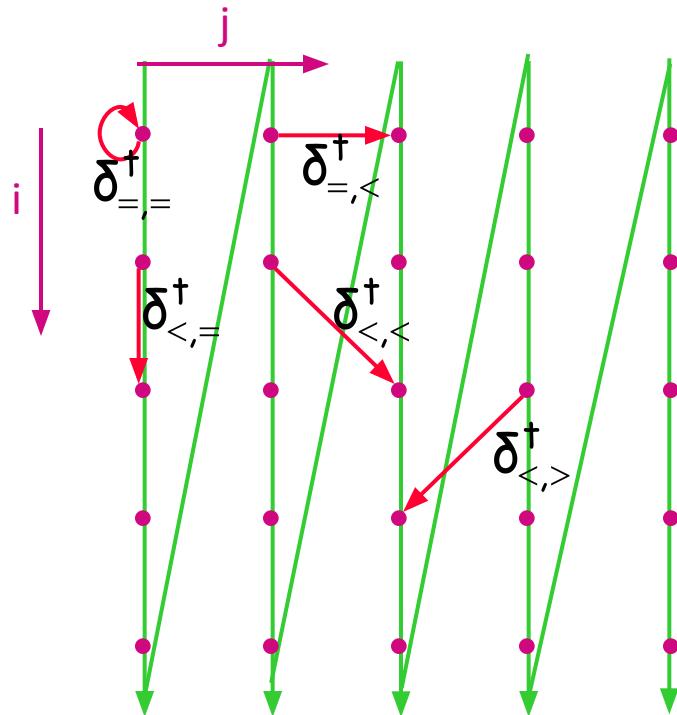


```
do j = 1,n  
  do i = 1,n  
    ... a(i,j) ...  
  end do  
end do
```

- When is loop interchange legal?

# Loop Interchange

```
do i = 1,n  
  do j = 1,n  
    ... a(i,j) ...  
  end do  
end do
```

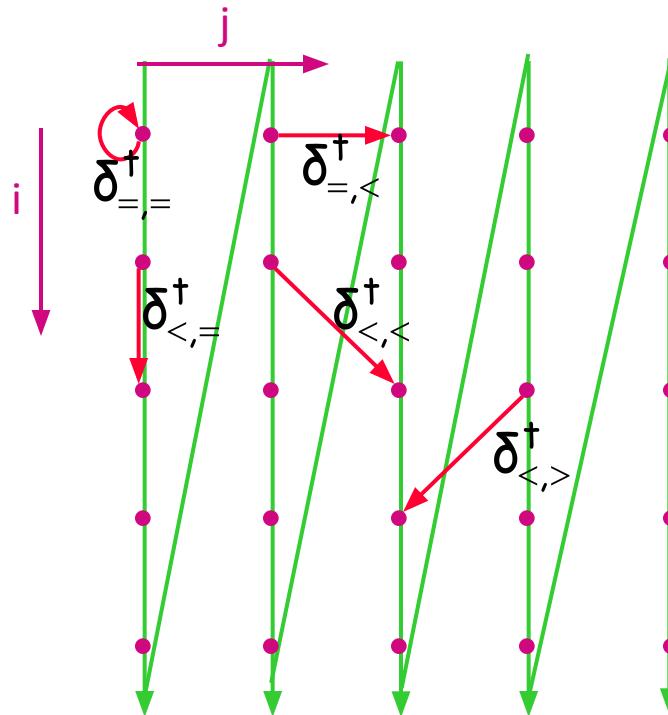


```
do j = 1,n  
  do i = 1,n  
    ... a(i,j) ...  
  end do  
end do
```

- When is loop interchange legal?

# Loop Interchange

```
do i = 1,n  
  do j = 1,n  
    ... a(i,j) ...  
  end do  
end do
```



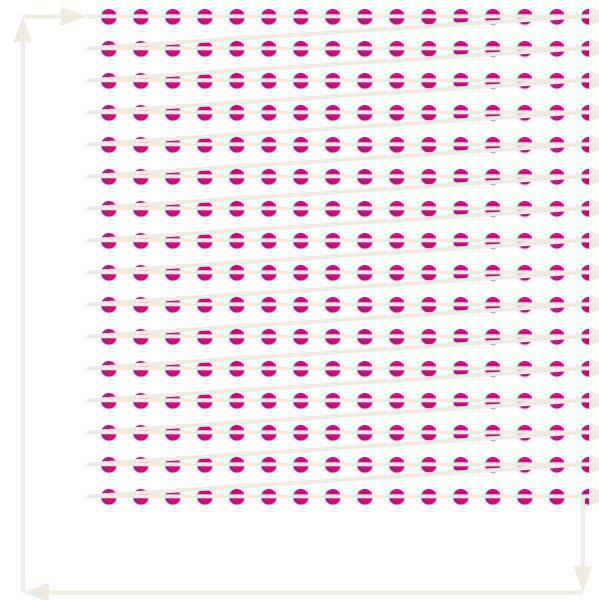
```
do j = 1,n  
  do i = 1,n  
    ... a(i,j) ...  
  end do  
end do
```

- When is loop interchange legal? when the “interchanged” dependences remain lexicographically positive!

# Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```
do t = 1,T  
  do i = 1,n  
    do j = 1,n  
      ... a(i,j) ...  
    end do  
  end do  
end do
```



# Loop Blocking (Loop Tiling)

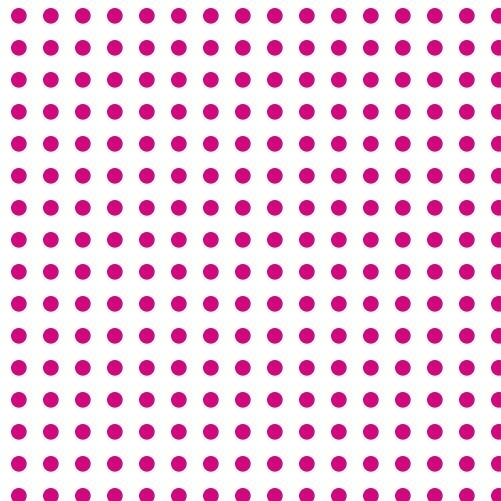
Exploits temporal locality in a loop nest.

```
do ic = 1, n, B
  do jc = 1, n , B
    do t = 1,T
      do i = 1,B
        do j = 1,B
          ... a(ic+i-1,jc+j-1) ...
        end do
      end do
    end do
  end do
end do
```



control loops

B: Block size



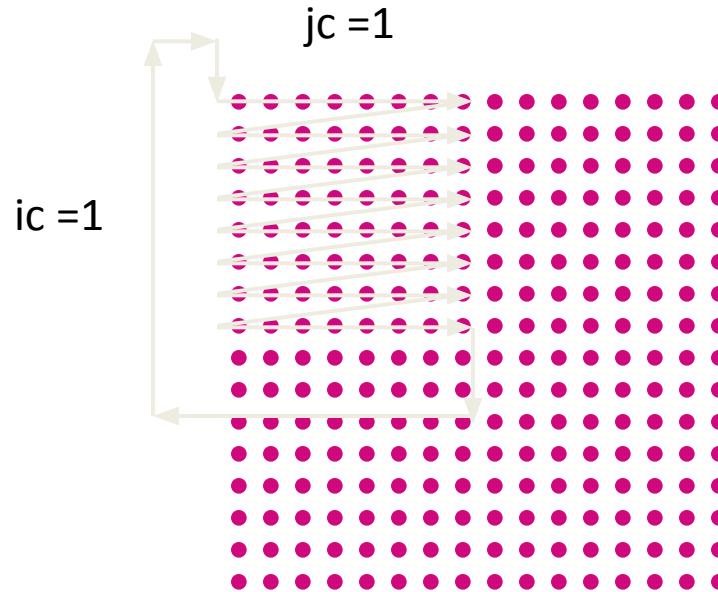
# Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```
do ic = 1, n, B
  do jc = 1, n , B
    do t = 1,T
      do i = 1,B
        do j = 1,B
          ... a(ic+i-1,jc+j-1) ...
        end do
      end do
    end do
  end do
end do
```

B: Block size

control loops



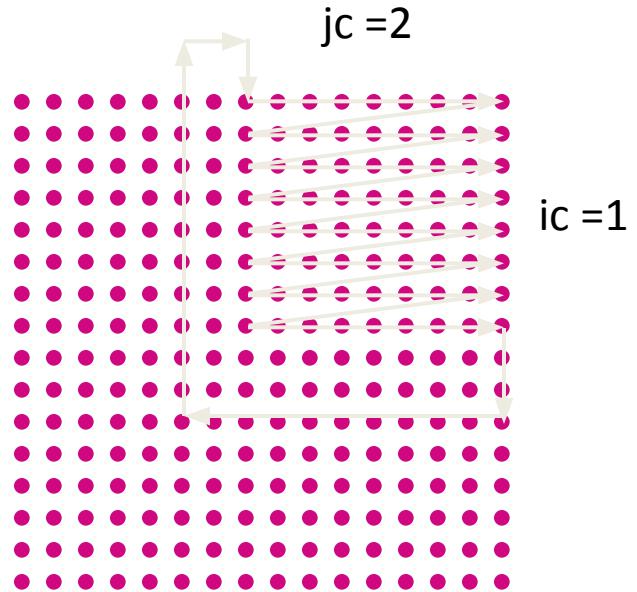
# Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```
do ic = 1, n, B  
  do jc = 1, n , B  
    do t = 1,T  
      do i = 1,B  
        do j = 1,B  
          ... a(ic+i-1,jc+j-1) ...  
        end do  
      end do  
    end do  
  end do  
end do
```

B: Block size

control loops



# Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

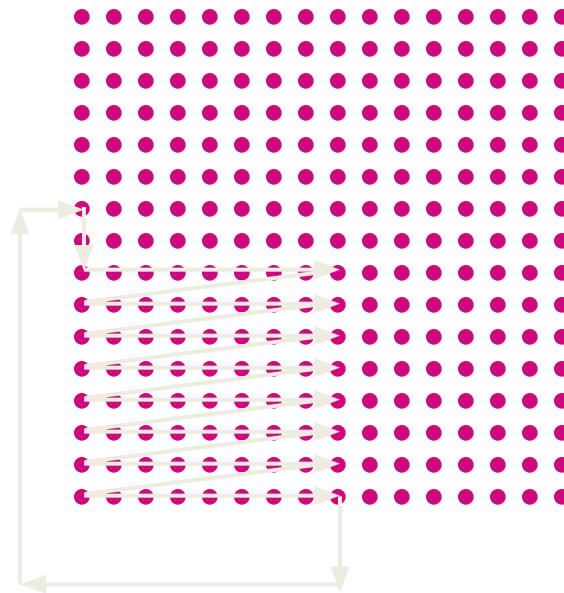
```
do ic = 1, n, B  
  do jc = 1, n , B  
    do t = 1,T  
      do i = 1,B  
        do j = 1,B  
          ... a(ic+i-1,jc+j-1) ...  
        end do  
      end do  
    end do  
  end do  
end do
```

B: Block size

control loops

ic = 2

jc = 1



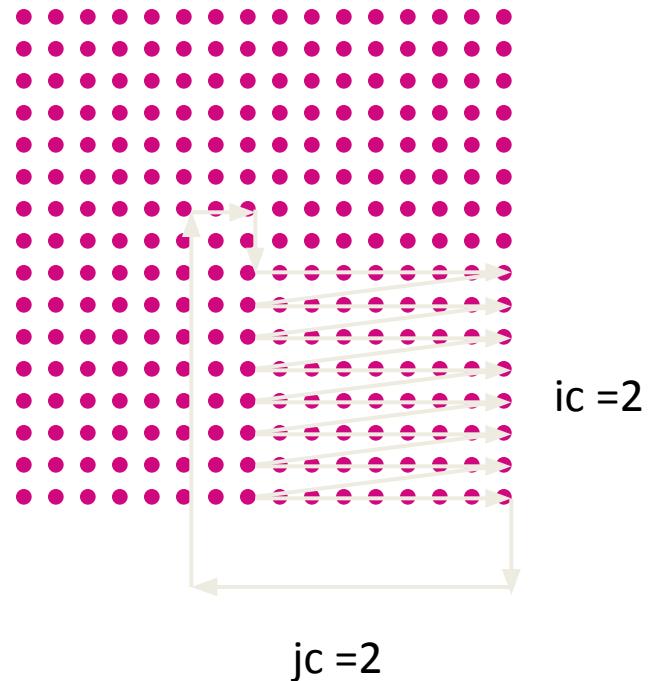
# Loop Blocking (Loop Tiling)

Exploits temporal locality in a loop nest.

```
do ic = 1, n, B  
  do jc = 1, n , B  
    do t = 1,T  
      do i = 1,B  
        do j = 1,B  
          ... a(ic+i-1,jc+j-1) ...  
        end do  
      end do  
    end do  
  end do  
end do
```

B: Block size

control loops



# Loop Blocking (Tiling)

```
do t = 1,T  
  do i = 1,n  
    do j = 1,n  
      ... a(i,j) ...  
    end do  
  end do  
end do
```

```
do t = 1,T  
  do ic = 1, n, B  
    do i = 1,B  
      do jc = 1, n, B  
        do j = 1,B  
          ... a(ic+i-1,jc+j-1) ...  
        end do  
      end do  
    end do  
  end do
```

```
do ic = 1, n, B  
  do jc = 1, n , B  
    do t = 1,T  
      do i = 1,B  
        do j = 1,B  
          ... a(ic+i-1,jc+j-1) ...  
        end do  
      end do  
    end do  
  end do  
end do
```

- When is loop blocking legal?

# CSC D70: Compiler Optimization Parallelization

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

*The content of this lecture is adapted from the lectures of  
Todd Mowry and Tarek Abdelrahman*