Compiler Optimization

Compiler optimizations transform code

- Code optimization transforms code to equivalent code
  - ... but with better performance

- The code transformation can involve either
  - **Replacing** code with more efficient code
  - **Deleting** redundant code
  - **Moving** code to a position where it is more efficient
  - **Inserting** new code to improve performance

The four categories of code transformations

- Replacing code (e.g. **strength reduction**)

  A=2*a; $\equiv$ A=a«1;

- Deleting code (e.g. **dead code elimination**)

  A=2; A=y; $\equiv$ A=y;

- Moving code (e.g. **loop invariant code motion**)

  for (i = 0; i < 100; i++) { sum += i + x * y; }

  $\equiv$

  t = x * y;

  for (i = 0; i < 100; i++) { sum += i + t; }

- Inserting code (e.g. **data prefetching**)

  for (p = head; p != NULL; p = p->next)

  { /* do work on node p */ }

  $\equiv$

  for (p = head; p != NULL; p = p->next)

  { prefetch(p->next); /* do work on node p */ }

Compiler optimization categories according to range

- How much code does the compiler view while optimizing?
  - The wider the view, the more powerful the optimization

- Axis 1: optimize across control flow?
  - **Local optimization**: optimizes only within straight line code
  - **Global optimization**: optimizes across control flow (if,for,...)

- Axis 2: optimize across function calls?
  - **Intra-procedural optimization**: only within function
  - **Inter-procedural optimization**: across function calls

- The two axes are orthogonal (any combination is possible)

Local vs. Global Constant Propagation

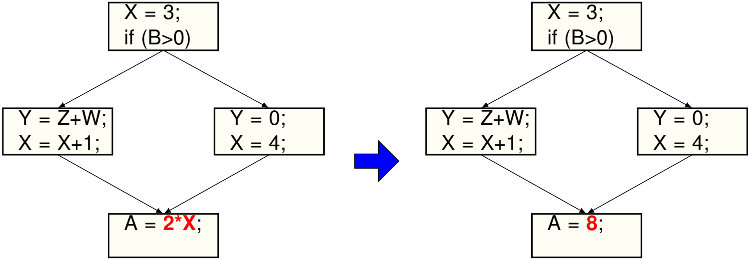
- **Constant propagation**
  - Optimization: if **x= y op z** and y and z are constants then compute at compile time and replace
- Local Constant Propagation

```
X = 3;
X = X+1;
A = X*2;
```

→

```
X = 3;
X = X+1;
A = 8;
```

- Global Constant Propagation



Intra- vs. Inter-procedural Constant Propagation

- Intra-procedural Constant Propagation

```
X = 3;
X = X+1;
A = X*2;
```

→

```
X = 3;
X = X+1;
A = 8;
```

- Inter-procedural Constant Propagation

```
X = 3;
foo(X);

void foo(int arg) {
  arg = arg+1;
  A = arg*2;
}
```

→

```
X = 3;
foo(X);

void foo(int arg) {
  arg = arg+1;
  A = 8;
}
```

  - Assuming all other calls to foo always pass in constant 3

Control Flow Analysis

Basic Block

- A function body is composed of one or more **basic blocks.**
- **Basic block**: a maximal sequence of instructions that
  - Has no jumps into the block other than the first instruction
  - Has no jumps out of the block other than the last instruction

- That means:
  - No instruction other than the first is a jump target
  - No instruction other than the last is a jump or branch

- Either all instructions in basic block execute or none
  - Smallest unit of execution in control flow analysis
  - Hence the descriptor "basic" in the name

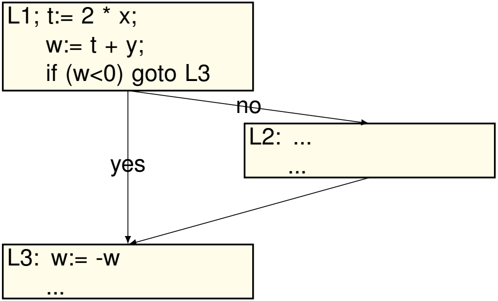
Control Flow Graph

- A **Control Flow Graph (CFG)** is a directed graph in which
  - Nodes are basic blocks
  - Edges represent flows of execution between basic blocks

- CFGs are widely used to represent a program for analysis
- CFGs are especially essential for global optimizations

Control Flow Graph Example

```
L1; t:= 2 * x;
    w:= t + y;
    if (w<0) goto L3
L2: ...
    ...
L3: w:= -w
    ...
```



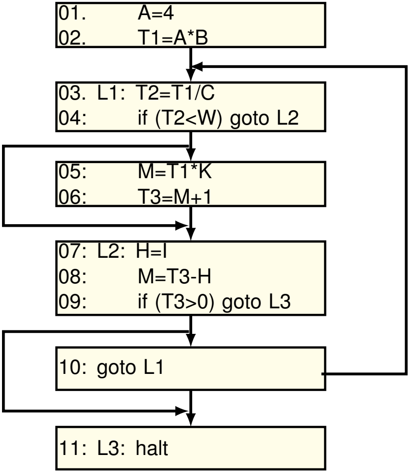
Construction of CFG

- Step 1: partition code into basic blocks
  - Identify **leader** instructions, where a leader is either:
    - the first instruction of a program, or
    - the target of any jump/branch, or
    - an instruction immediately following a jump/branch
  - Create a basic block out of each leader instruction
  - Expand basic block by adding subsequent instructions (Stopping when the next leader instruction is encountered)

- Step 2: add edge between two basic blocks B1 and B2 if
  - there exist a jump/branch from B1 to B2, or
  - B2 follows B1, and B1 does not end with jump/branch

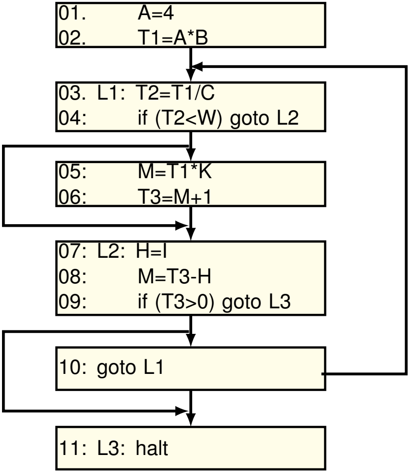
Example

```
01.       A=4
02.       T1=A*B
03.  L1:  T2=T1/C
04:       if (T2<W) goto L2
05:       M=T1*K
06:       T3=M+1
07:  L2:  H=I
08:       M=T3-H
09:       if (T3>0) goto L3
10:  goto L1
11:  L3:  halt
```



Example

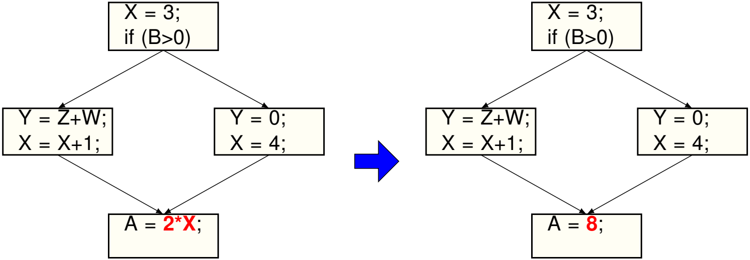
```
01.         A=4
02.         T1=A*B
03.  L1:  T2=T1/C
04:         if (T2<W) goto L2
05:         M=T1*K
06:         T3=M+1
07:  L2:  H=I
08:         M=T3-H
09:         if (T3>0) goto L3
10:  goto L1
11:  L3:  halt
```



Data Flow Analysis

Global Optimizations

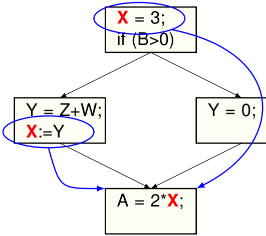
- Extend optimizations across control flows, i.e. CFG
- Like in this example of Global Constant Propagation (GCP):



- How do we know it is OK to globally propagate constants?

Correctness criteria for GCP

- There are situations that prohibit GCP:



- To replace X by a constant C **correctly**, we must know
  - **Along all paths**, the last assignment to X is "X = C"

- Paths may go through loops and/or branches
  - When two paths **meet**, need to make a **conservative** choice

Global Optimizations need to be Conservative

- Many compiler optimizations depend on knowing some property X at a particular point in program execution
  - Need to prove at that point property X holds along all paths

- To ensure correctness, optimization must be **conservative**
  - An optimization is enabled only when X is definitely true
  - If not sure, be conservative and say **don't know**
  - **Don't know** usually disables the optimization

Dataflow Analysis Framework

- **Dataflow analysis**: discovering properties about values
  - ... at certain points in the CFG to enable optimizations
  - E.g. discovering a value is constant at a statement
  - Done by observing the flow of data through the CFG

- **Dataflow analysis framework**:
  - A framework for describing different dataflow analyses
  - Can be defined using the following 4 components:

    $\{ \mathbf{D}, \mathbf{V}, \wedge: (V, V) \to V, \mathbf{F}: V \to V \}$

    - **D**: direction of dataflow (forward or backward)
    - **V**: domain of values denoting property
    - $\wedge$: **meet operator** that merges values when paths meet
    - **F**: **flow propagation function** that propagates values through a basic block

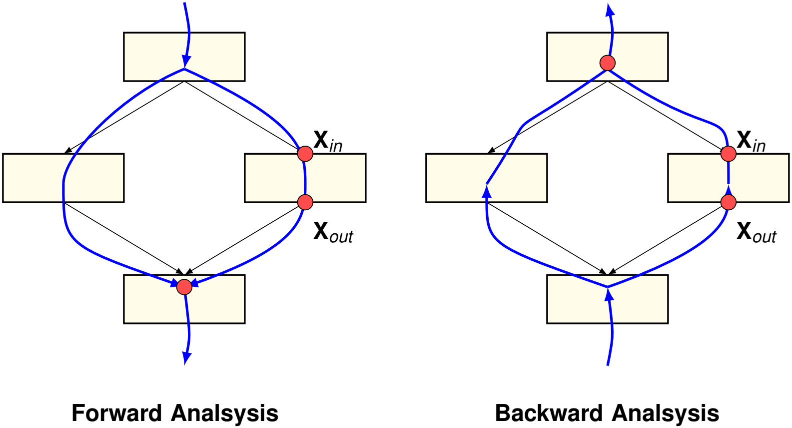
Global Constant Propagation (GCP)

- Let's use **GCP** to study dataflow analysis framework

- We will define each component one by one for GCP
  - **D**: direction of dataflow for constant property
  - **V**: domain of values denoting constant property
  - $\wedge$: **meet operator** that merges values when paths meet
  - **F**: **flow propagation function** for GCP

Direction D for GCP

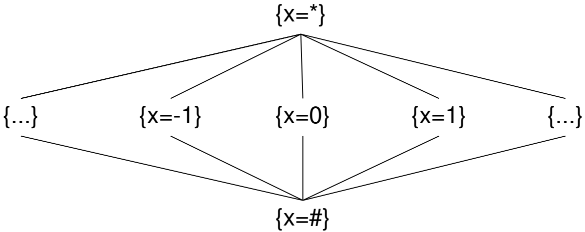
- Is GCP a forward or backward analysis?



- Forward, since "constantness" of a variable flows forward to subsequent instructions starting from assignment

V and meet operator ∧ for GCP

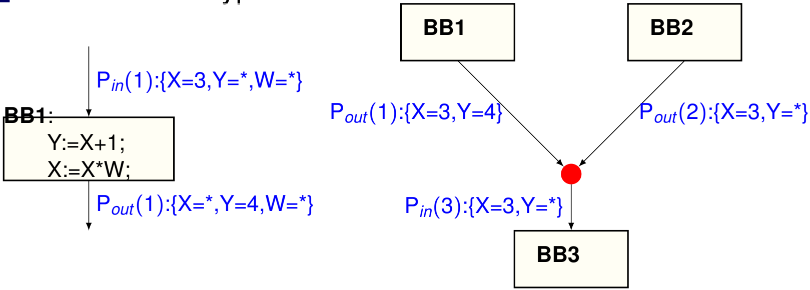
- Given an integer variable x, domain V is the set:
  - ..., {x=-1}, {x=0}, {x=1}, ... /* a constant */
  - {x=*} /* not a constant */
  - {x=#} /* x is not assigned on any path*/
- Meet operator ∧ is given by this **semi-lattice**:
  - a ∧ b = least upper bound in the below graph



  - {x=0} ∧ {x=1} = {x=*}: Different values on each path
  - {x=#} ∧ {x=1} = {x=1}: Constant definition on one path

Dataflow Equations for GCP

- There are two types of flow functions



- ➢ Flow transfer function F: V $\rightarrow$ V
  - Computes data flow within basic blocks
  - Remove those that become variables, add new constants
- ➢ Meet operator $\wedge$: (V, V) $\rightarrow$ V
  - Computes data flow across basic blocks
  - Merge values from two paths using the previous semi-lattice

Flow Transfer Function F for GCP

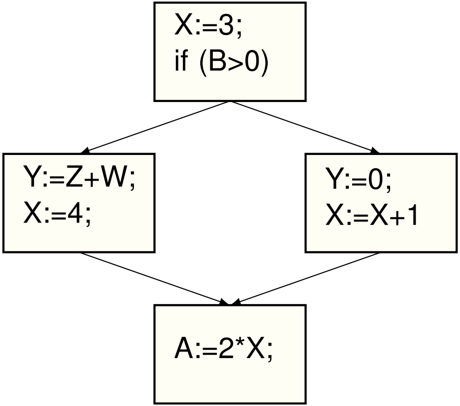
- **X(i)**: dataflow property X of basic block **i**
  - $\mathbf{X}_{in}\mathbf{(i)}$: at the entry of basic block **i**
  - $\mathbf{X}_{out}\mathbf{(i)}$: at the exit of basic block **i**

- F for Global constant propagation (GCP)

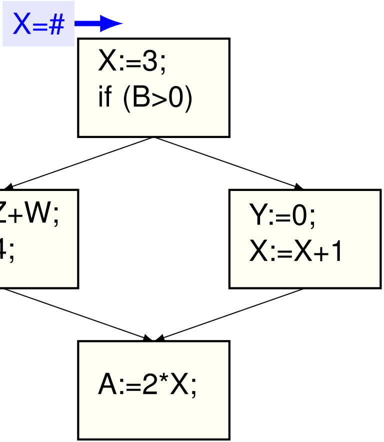
$$\mathbf{GCP}_{out}\mathbf{(i) = (GCP}_{in}\mathbf{(i) - DEF}_{v}\mathbf{(i))} \cup \mathbf{DEF}_{c}\mathbf{(i)}$$

where DEF_v(i) contains variable definitions in basic block i
DEF_c(i) contains constant definitions in basic block i

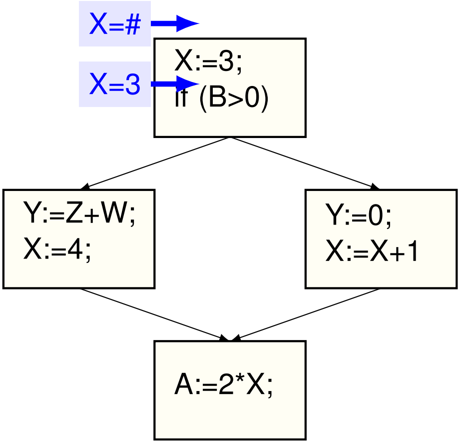
GCP Propagation without loops



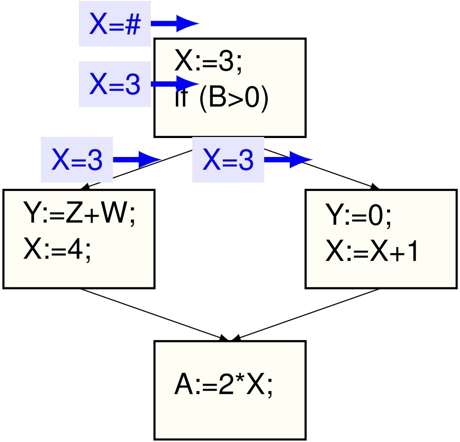
GCP Propagation without loops



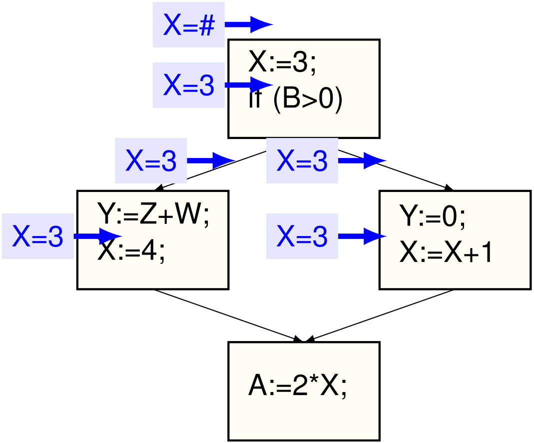
GCP Propagation without loops



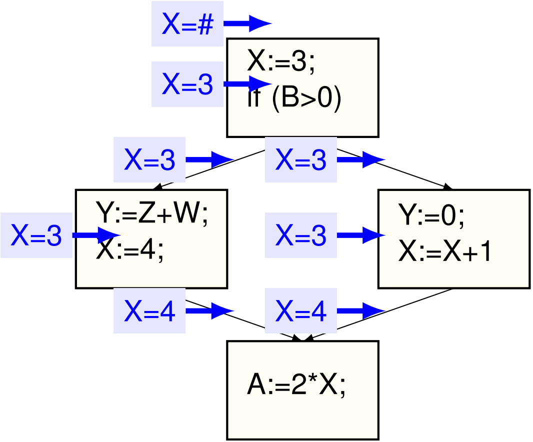
GCP Propagation without loops



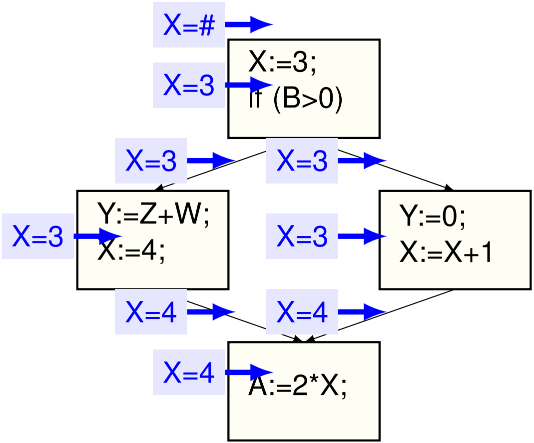
GCP Propagation without loops



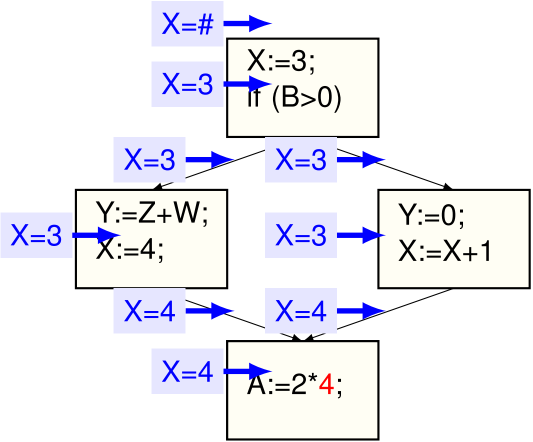
GCP Propagation without loops



GCP Propagation without loops

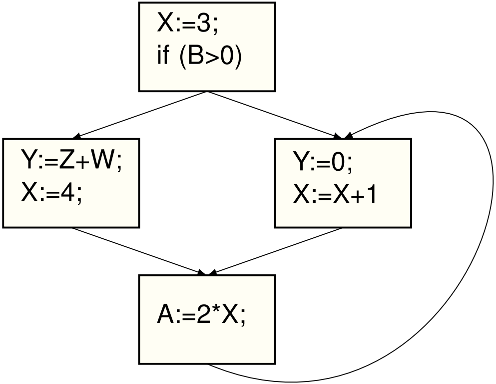


GCP Propagation without loops



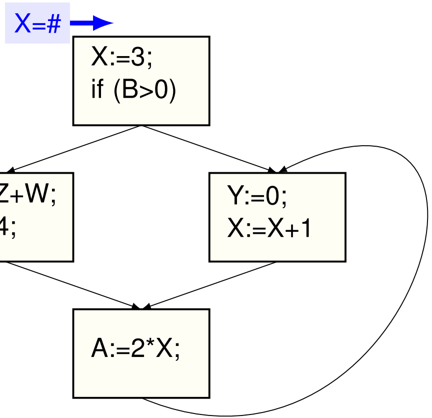
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



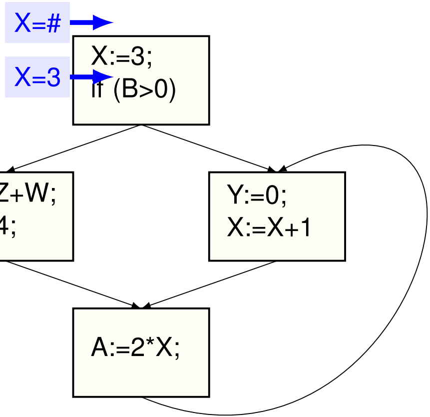
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



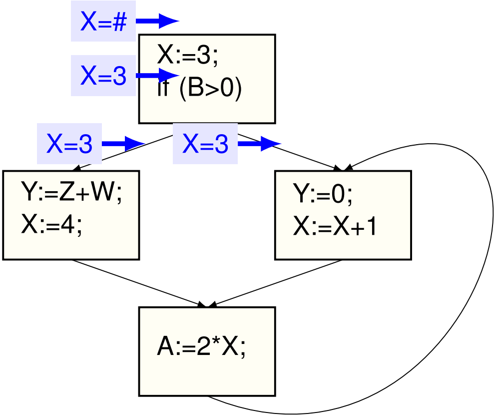
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



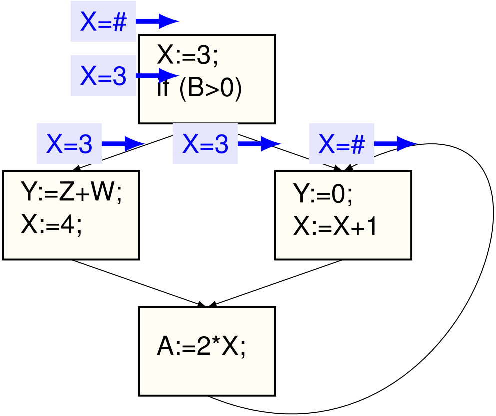
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



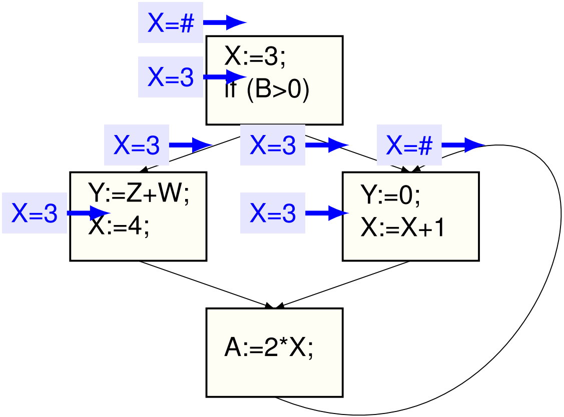
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



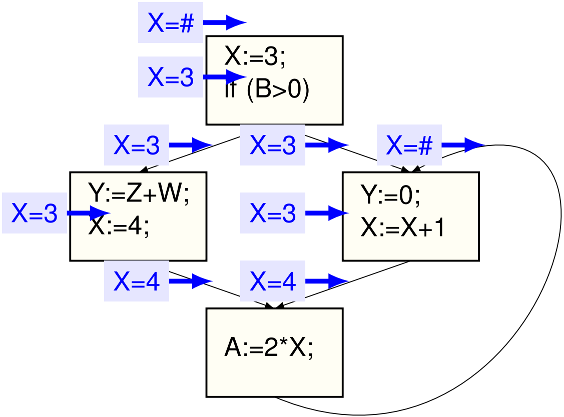
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



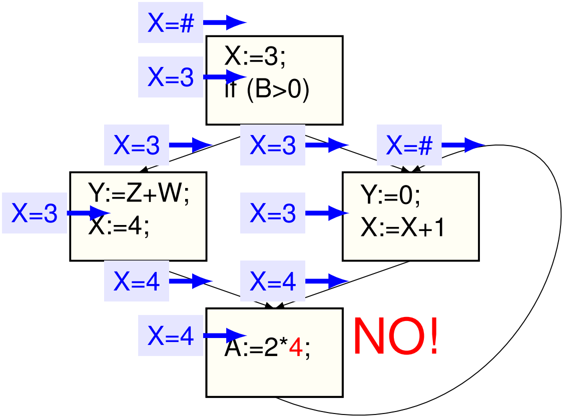
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



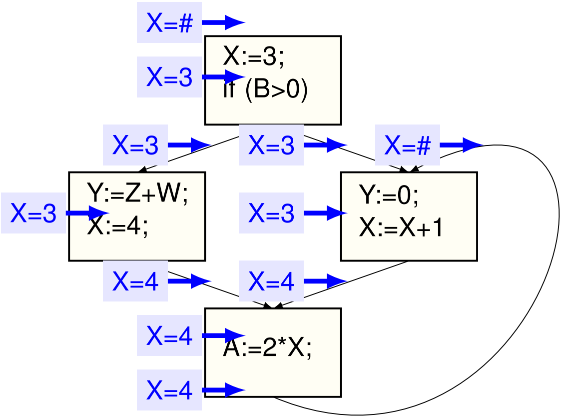
GCP Propagation with loops

- Iterate until there are no changes to values
  - ➢ This is called the **maximum fixed point** solution



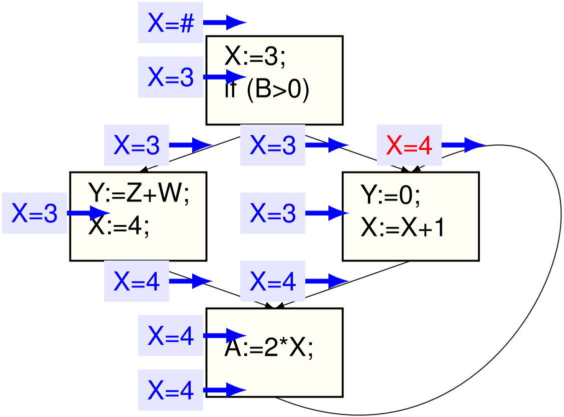
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



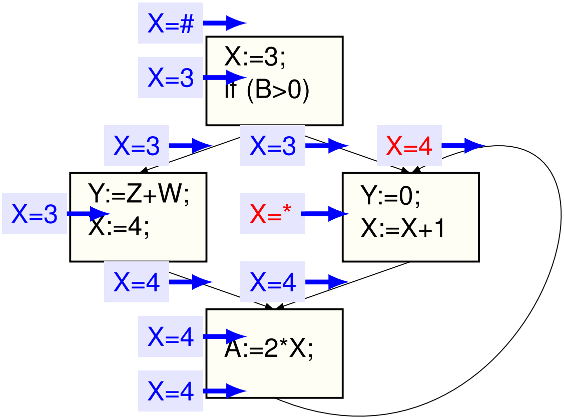
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



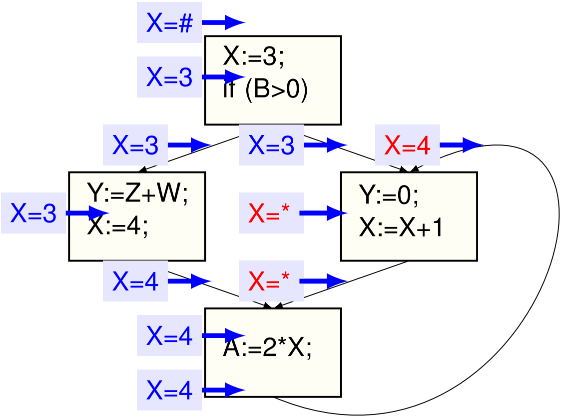
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



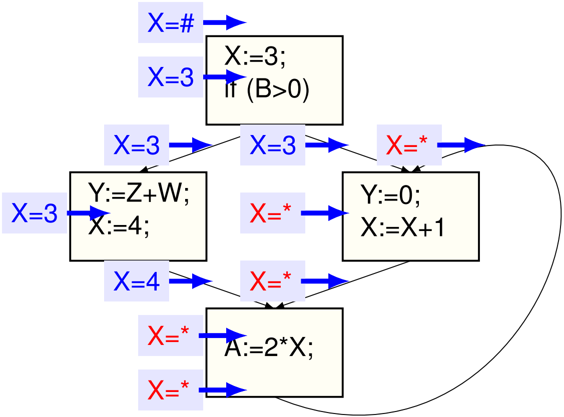
GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



GCP Propagation with loops

- Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



Analysis Algorithm for GCP

- GCP Algorithm
  (1). Set {x=#} at all the points in the procedure
  (2). Propagate the dataflow property along the control flow
  (3). Repeat step (2) until there are no changes

- Will GCP eventually stop?
  - If there are loops, we may propagate the loop many times
  - Is there a possibility to run into an endless loop?

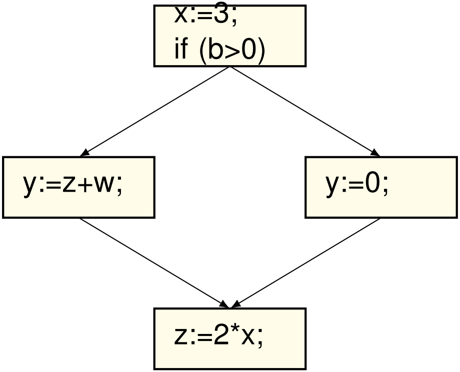
Termination Problem

- **Least upper bound** ensures the termination
  - Values starts from #
  - Values can only increase in the hierarchy
  - Any values can change at most twice in our example
    ... from # to C, and from C to *

- The maximal number of steps is O(program_size)

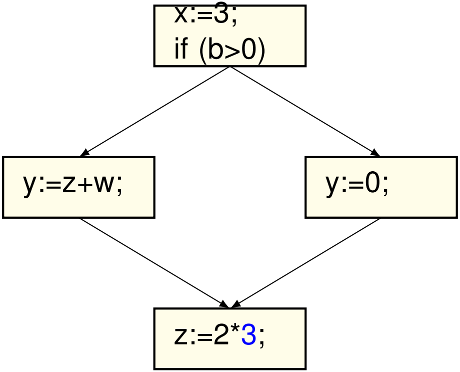
Another Analysis: Liveness Analysis

- Once constants have been globally propagated, we would like to eliminate the dead code



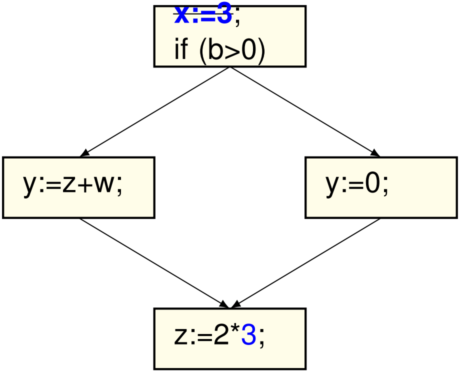
Another Analysis: Liveness Analysis

- Once constants have been globally propagated, we would like to eliminate the dead code



Another Analysis: Liveness Analysis

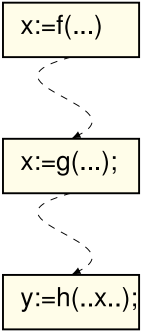
- Once constants have been globally propagated, we would like to eliminate the dead code



Live/Dead Statment

- A **dead statement** calculates a value that is not used later, or output to file
- Otherwise, it is a **live statement**

In the example,
the 1st statement is dead,
the 2nd statement is live



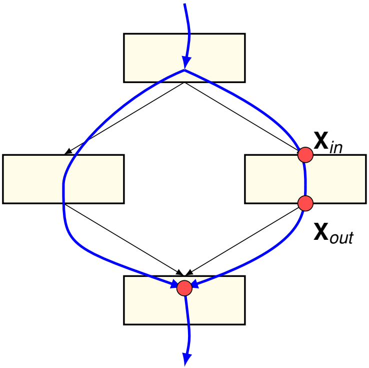
Liveness Analysis

- We can form liveness analysis as a dataflow analysis
  - ➢ Propagate the information along control flow
    - "x is dead", "x is live", "y is dead", "y is live"
  - ➢ Liveness is simpler than constant propagation
    - It is a boolean property
  - ➢ Liveness is different from constant propagation
    - Liveness analysis is a union problem
      ☞ x is alive if x is alive along one path
    - Constant propagation is an intersection problem
      ☞ x is NOT a constant if x is NOT a constant along one path
      ...

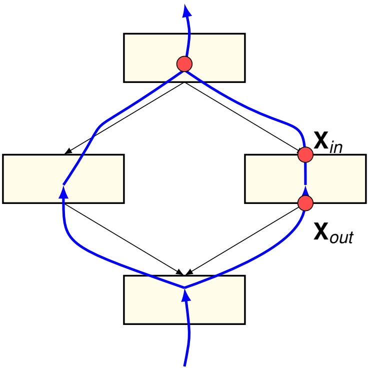
Forward and Backward Analysis

- The most significant difference is
  - Liveness analysis wants to know if it is used some time later
    - Use information after this statement to decide
    - **Backward analysis**
  - Constant analysis wants to know if it is constant for all possible executions at this point
    - Use information before this statement to decide
    - **Forward analysis**

Graphic View of Forward and Backward Analysis



Forward Analsysis



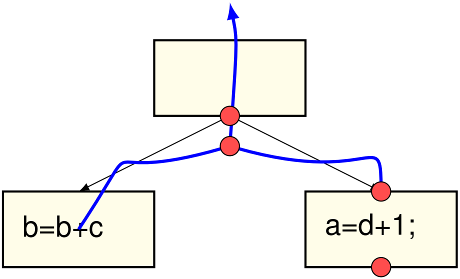
Backward Analsysis

Global Liveness Analysis

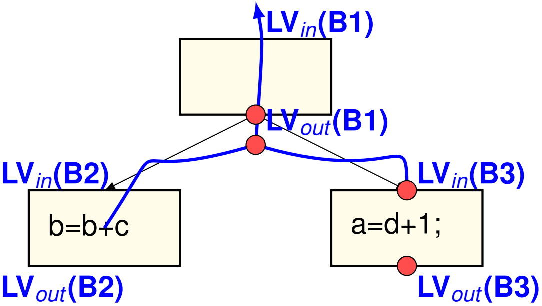
- Global liveness analysis
  - A variable x is live at statement s if
    - There exists a statement ss after s that use x
    - There is a path from s to ss
    - That path has no intervening assignment to x

- A backward dataflow analysis
  {**L**, ($\top$, $\bot$), ($\sqcap$, $\sqcup$), **f: L** $\rightarrow$ **L** }
  - Lattice, top and bottowm items
  - Operators
  - Dataflow functions

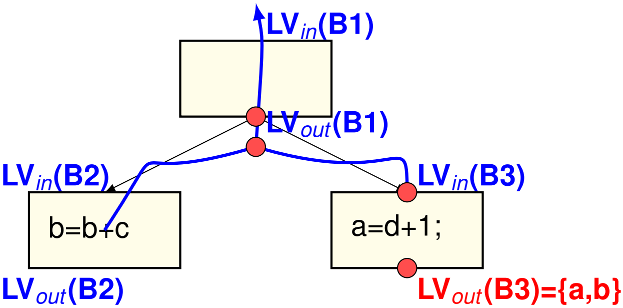
Liveness Example



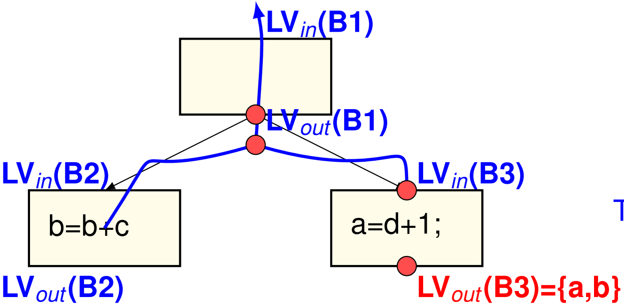
Liveness Example



Liveness Example

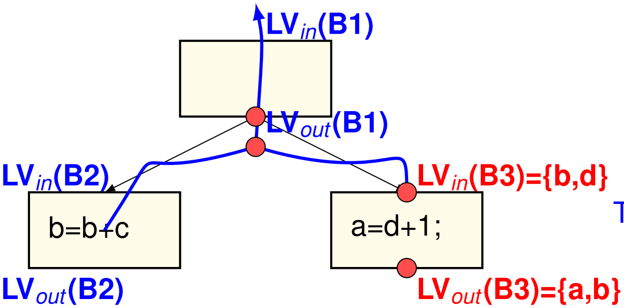


Liveness Example



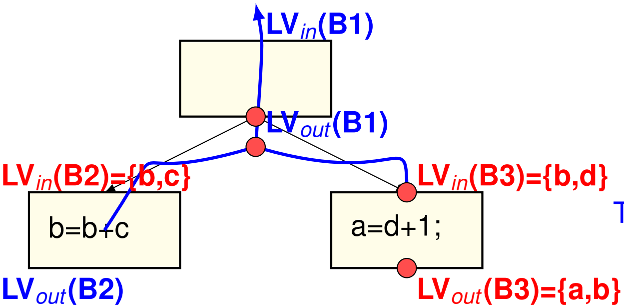
Two sets:
DEF={a}
USE={d}

Liveness Example



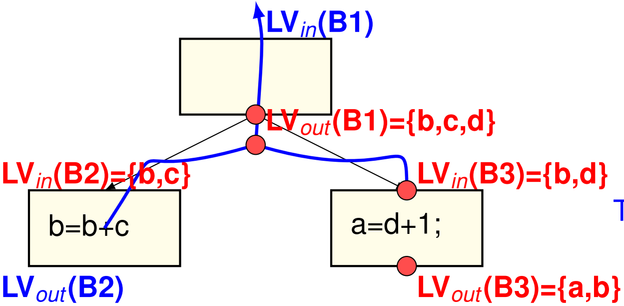
Two sets:
DEF={a}
USE={d}

Liveness Example



Two sets:
DEF={a}
USE={d}

Liveness Example



Two sets:
DEF={a}
USE={d}

Dataflow Equations for Liveness Analysis

- **X(i)** — dataflow property of basic block **i**
  - $\mathbf{X_{in}(i)}$ — at the entry of basic block **i**
  - $\mathbf{X_{out}(i)}$ — at the exit of basic block **i**
- Flow equations
  - flow transfer function
    $\mathbf{LV_{in}(i) = (\ LV_{out}(i) - DEF(i)\ ) \cup USE(i)}$
  - flow propagation function
    $\mathbf{LV_{out}(i) = \cup\ LV_{in}(k)}$ where **k** is successor of **i**

Comparison with Dataflow Equations for GCP

- **X(i)** — dataflow property of basic block **i**
  - $\mathbf{X_{in}(i)}$ — at the entry of basic block **i**
  - $\mathbf{X_{out}(i)}$ — at the exit of basic block **i**
- Global constant propagation (GCP)
  - flow transfer function
    $\mathbf{GCP_{out}(i) = (GCP_{in}(i) - DEF_{v}(i)) \cup DEF_{c}(i)}$

    where $DEF_v(i)$ contains variable definitions in basic block i
    $DEF_c(i)$ contains constant definitions in basic block i
  - flow propagation function
    $\mathbf{GCP_{in}(i) = \cap\ GCP_{out}(k)}$ where **k** is predecessor of **i**

Application of Liveness Analysis

- Global dead code elimination is based on global liveness analysis (GLA)
  - Dead code detection
    - A statement x:= ... is dead code if x is dead after this statement
    - Dead statement can be deleted from the program (Didn't consider side-effect)
- Global register allocation is also based on GLA
  - Live variables should be placed in registers
  - Registers holding dead variables can be reused

Summary of Dataflow Analysis

- There are many other global dataflow analysis
- Classification
  - Forward/Backward analysis
  - Union analysis — some property is true if it is true along at least one path Intersection analysis — some property is true if it is true along all paths
- Very useful in
  - compiler optimization
  - software engineering
  - debugging

The END !