SSA, Loop Dependence and Parallelism

Announcements

- Midterm is Monday March 6th, in class, two weeks from today
- Assignment 2 is due the week after that. Start now!

Today

- Discuss assignment 1
- Finish SSA
- Data dependencies between loop iterations
- Determining when a loop is parallelizable

Assignment 1: Kaleidoscope compiler

Grading

- Done anonymously if possible
- Attempted to compile and run all programs using provided examples.
- Do cite other submissions if your submission built on some of the ideas you saw while doing peer reviews.

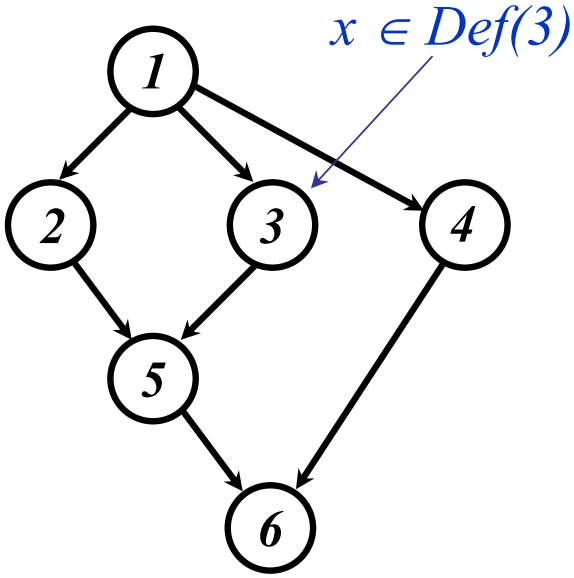
Issues people ran into

- Left and right associativity was not discussed enough.
- Recursive descent parsing most often done with lookahead.
- AST descriptions missed “abstract” piece. Executable?
- Beautiful abstractions for lexing?
- LLVM library functions
- Can we bound how well program optimization can do?
- 260564 had dumpast and dumpir features
- For loops in Kaleidoscope were do while loops?

Dominance Frontiers Revisited

Suppose that node 3 defines variable x

DF(3) = {5}



Do we need to insert a ϕ- function for x anywhere else?

Yes. At node 6. Why?

Dominance Frontiers and SSA

Let

- $DF_1(S) = DF(S)$
- $DF_{i+1}(S) = DF(S \cup DF_i(S))$

Iterated Dominance Frontier

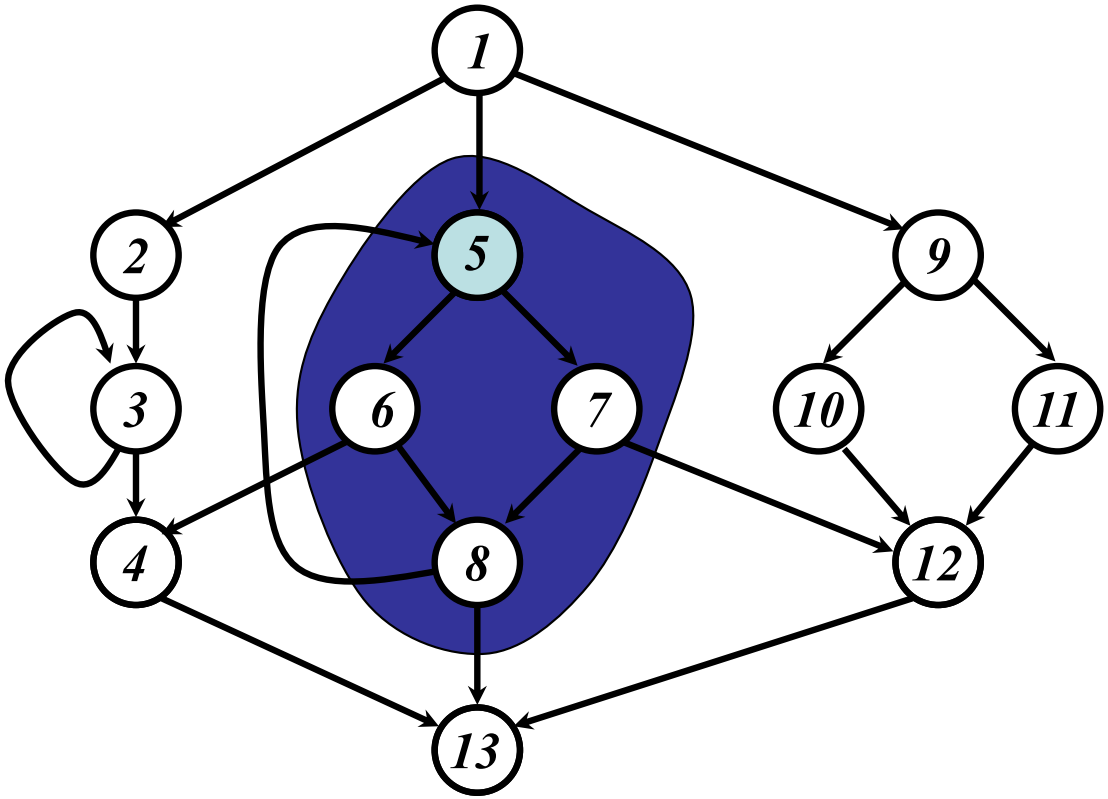
- $DF_\infty(S)$

Theorem

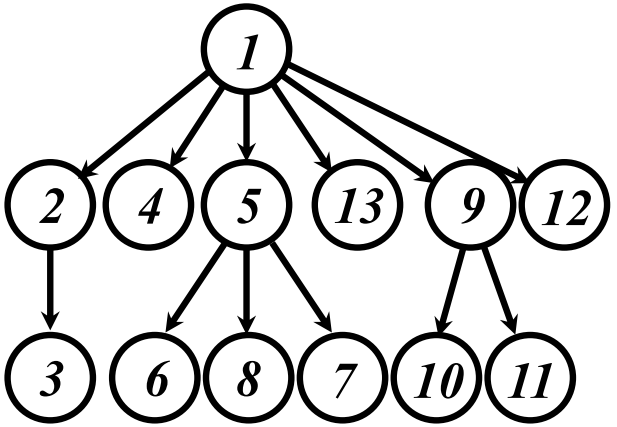
- If S is the set of CFG nodes that define variable v, then $DF_\infty(S)$ is the set of nodes that require ϕ-functions for v

Dominance Tree Example

The dominance tree shows the immediate dominance relation



CFG



Dominance Tree

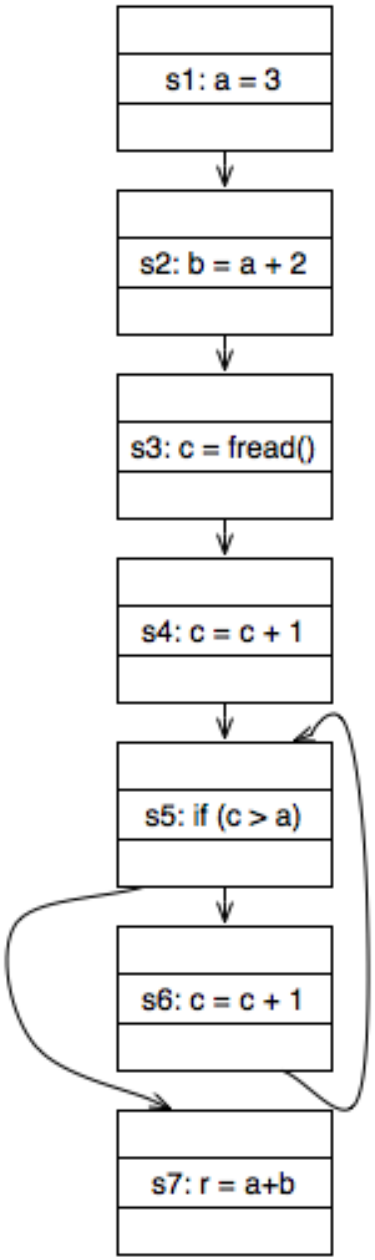
Inserting Phi Nodes

Calculate the dominator tree

- a lot of research has gone into calculating this quickly

Computing dominance frontier from dominator tree

- $DF_{local}[n]$= successors of n (in CFG) that are not strictly dominated by n
- $DF_{up}[n]$= nodes in the dominance frontier of n that are not strictly dominated by n's immediate dominator
- $DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[c]$



Algorithm for Inserting ϕ-Functions

```
for each variable v
    WorkList ← ∅
    EverOnWorkList ← ∅
    AlreadyHasPhiFunc ← ∅
    for each node n containing an assignment to v      Put all defs of v on the worklist
        WorkList ← WorkList ∪ {n}
    EverOnWorkList ← WorkList
    while WorkList ≠ ∅
        Remove some node n for WorkList
        for each d ∈ DF(n)
            if d ∉ AlreadyHasPhiFunc                   Insert at most one ϕ function per node
                Insert a ϕ-function for v at d
                AlreadyHasPhiFunc ← AlreadyHasPhiFunc ∪ {d}
                if d ∉ EverOnWorkList                  Process each node at most once
                    WorkList ← WorkList ∪ {d}
                    EverOnWorkList ← EverOnWorkList ∪ {d}
```

Transformation to SSA Form

Two steps

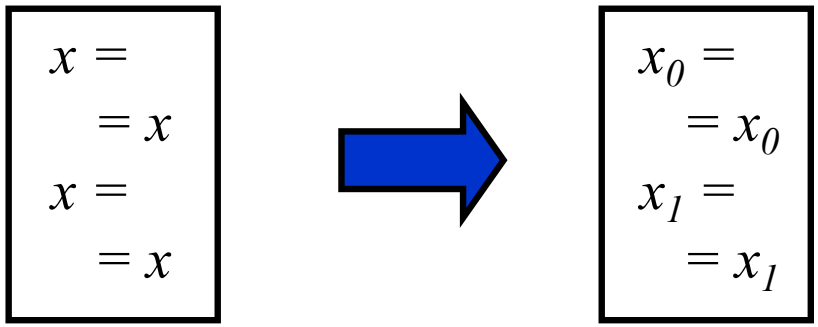
- Insert ϕ-functions
- Rename variables

Variable Renaming

Basic idea

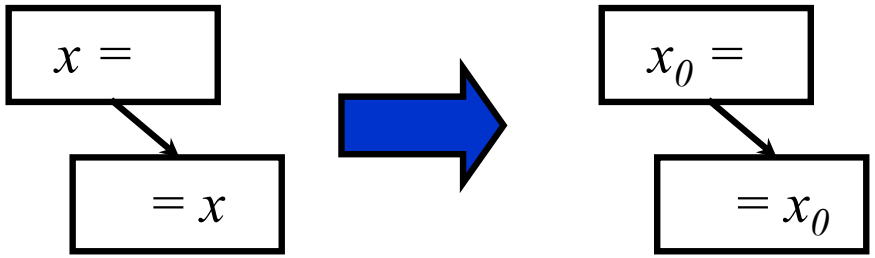
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

Easy for straightline code



Use a stack when there's control flow

- *For each use of x, find the definition of x that dominates it*



Traverse the dominance tree

Variable Renaming (cont)

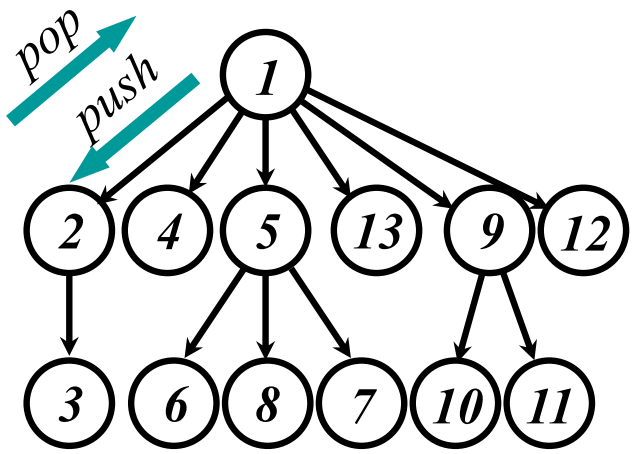
Data Structures

- Stacks[v] ∀v
  Holds the subscript of most recent definition of variable v, initially empty
- Counters[v] ∀v
  Holds the current number of assignments to variable v; initially 0

Auxiliary Routine

procedure GenName(variable v)

```
i := Counters[v]
push i onto Stacks[v]
Counters[v] := i + 1
```



Use the Dominance Tree to remember the most recent definition of each variable

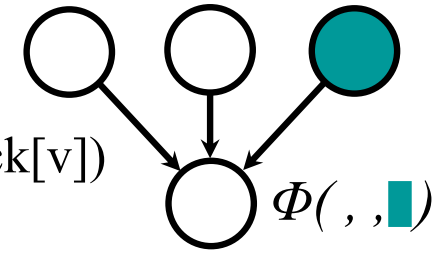
Variable Renaming Algorithm

```
procedure Rename(block b)
    if b previously visited return

    for each statement s in b (in order)
        for each variable v ∈ RHS(s) (except for ϕ-functions)
            replace v by v_i, where i = Top(Stacks[v])
        for each variable v ∈ LHS(s)
            GenName(v) and replace v with v_i, where i=Top(Stack[v])
    for each s ∈ succ(b)    (in CFG)
        j ← position in s's ϕ-function corresponding to block b
        for each ϕ-function p in s
            replace the j^th operand of RHS(p) by v_i, where i = Top(Stack[v])

    for each s ∈ child(b)  (in DT)
        Rename(s)
    for each ϕ-function or statement t in b
        for each v_i ∈ LHS(t)
            Pop(Stack[v])
```

Call Rename(entry-node)



Recurse using Depth First Search

Unwind stack when done with this node

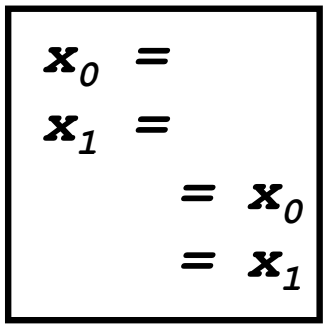
Transformation from SSA Form

Proposal

- Restore original variable names (*i.e.*, drop subscripts)
- Delete all ϕ-functions

Complications (the proposal doesn't work!)

- *What if versions get out of order? (simultaneously live ranges)*



Alternative

- *Perform dead code elimination (to prune ϕ-functions)*
- *Replace ϕ-functions with copies in predecessors*
- *Rely on register allocation coalescing to remove unnecessary copies*

Loop Data Dependence Analysis

Loop transformations

- Parallelization
- Rescheduling to improve data locality

Data dependence analysis

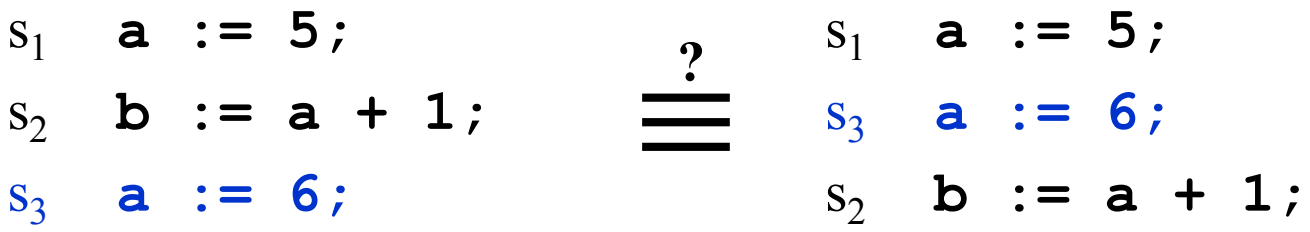
- What is the partial ordering between iterations based on data dependences?
- That partial ordering must be maintained.

Data Dependences

Recall

- A data dependence defines ordering relationship two between statements
- In executing statements, data dependences must be respected to preserve correctness

Example



Dependences and Loops

Loop-independent dependences

```
do i = 1,100
      A(i) = B(i)+1
      C(i) = A(i)*2
enddo
```

Dependences within the same loop iteration

Loop-carried dependences

```
do i = 1,100
   A(i) = B(i)+1
  C(i) = A(i-1)*2
enddo
```

Dependences that cross loop iterations

Data Dependence Terminology

We say statement s_2 depends on s_1

- **True (flow) dependence**: s_1 writes memory that s_2 later reads
- **Anti-dependence**: s_1 reads memory that s_2 later writes
- **Output dependences**: s_1 writes memory that s_2 later writes
- **Input dependences**: s_1 reads memory that s_2 later reads

Notation: $s_1 \ \delta \ s_2$

- s_1 is called the **source** of the dependence
- s_2 is called the **sink** or **target**
- s_1 must be executed before s_2

Data Dependences and Loops

How do we identify dependences in loops?

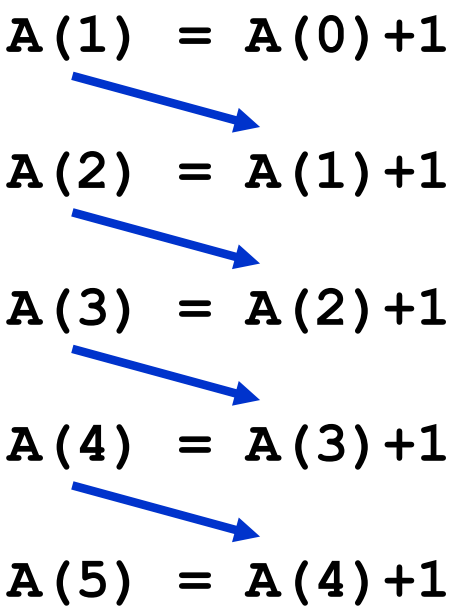
```
do i = 1,5
   A(i) = A(i-1)+1
enddo
```

Simple view

- Imagine that all loops are fully unrolled
- Examine data dependences as before

Problems

- Impractical and often impossible
- Lose loop structure



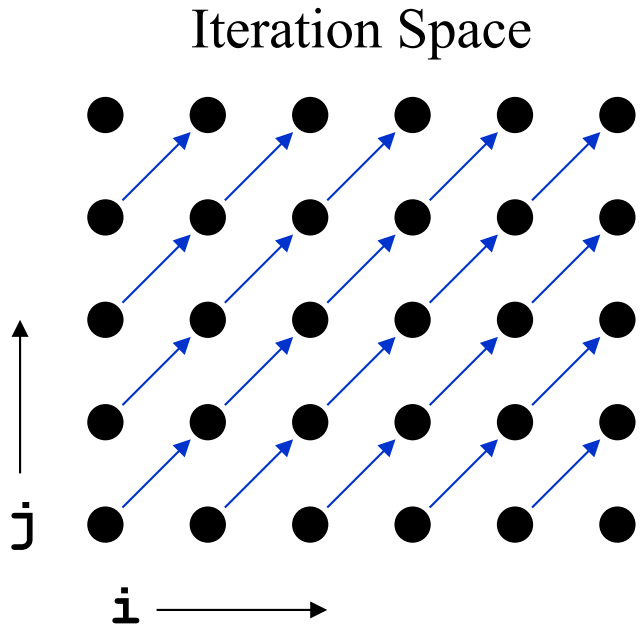
Iteration Spaces

Idea

– Explicitly represent the iterations of a loop nest

Example

```
do i = 1,6
     do j = 1,5
        A(i,j) = A(i-1,j-1)+1
     enddo
enddo
```



Iteration Space

– A set of tuples that represents the iterations of a loop

– Can visualize the dependences in an iteration space

Distance Vectors

Idea

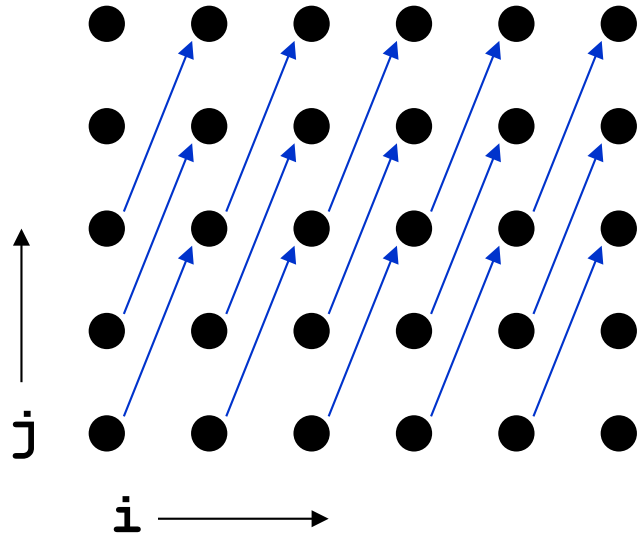
- Concisely describe dependence relationships between iterations of an iteration space
- For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location

Definition

- $\mathbf{v} = \mathbf{i}^T - \mathbf{i}^S$

Example

```
do i = 1,6
   do j = 1,5
         A(i,j) = A(i-1,j-2)+1
     enddo
enddo
```



outer loop

Distance Vector: (1,2)

inner loop

Distance Vectors: Legality

Definition

- A dependence vector, *v*, is **lexicographically nonnegative** when the left-most entry in *v* is positive or all elements of *v* are zero
  Yes: (0,0,0), (0,1), (0,2,-2)
  No: (-1), (0,-2), (0,-1,1)

- A dependence vector is **legal** when it is lexicographically nonnegative (assuming that indices increase as we iterate)

Why are lexicographically negative distance vectors illegal?

What are legal direction vectors?

Loop-Carried Dependences

Definition

- A dependence $D=(d_1,...d_n)$ is **carried** at loop level i if d_i is the first nonzero element of D

Example

```
do i = 1,6
   do j = 1,6
      A(i,j) = B(i-1,j)+1
      B(i,j) = A(i,j-1)*2
      enddo
enddo
```

Distance vectors: (0,1) for accesses to **A**
(1,0) for accesses to **B**

Loop-carried dependences

- The j loop carries dependence due to **A**
- The i loop carries dependence due to **B**

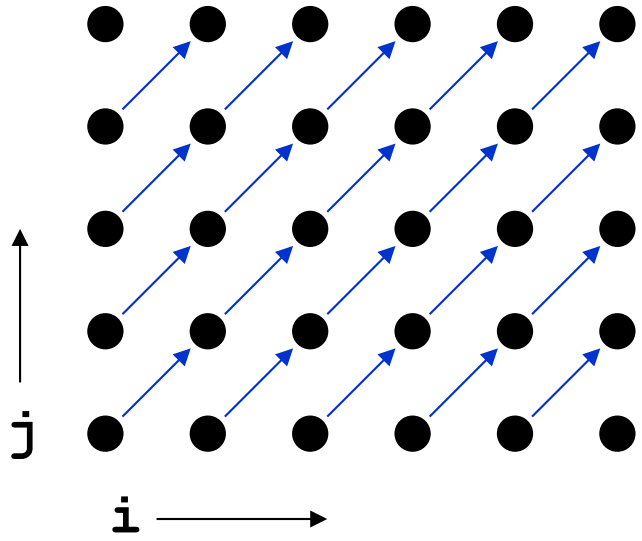
Direction Vector

Definition

- A direction vector serves the same purpose as a distance vector when less precision is required or available
- Element *i* of a direction vector is <, >, or = based on whether the source of the dependence precedes, follows or is in the same iteration as the target in loop *i*

Example

```
do i = 1,6
     do j = 1,5
        A(i,j) = A(i-1,j-1)+1
     enddo
enddo
```



Direction vector: (<,<)

Distance vector: (1,1)

Data dependence analysis examples

Do some examples in class.

Concepts

SSA

- Dominance frontiers and dominance trees
- Algorithm for inserting phi nodes
- Algorithm for renaming variables
- How to convert SSA back to executable 3-address code

Loops

- Data dependences including distance vectors,
- loop carried dependences, and
- direction vectors.
- How to determine a loop can be parallelized.

Next Time

Reading

- Advanced Compiler Optimizations for Supercomputers by Padua and Wolfe

Lecture

- Parallelization including vectorization