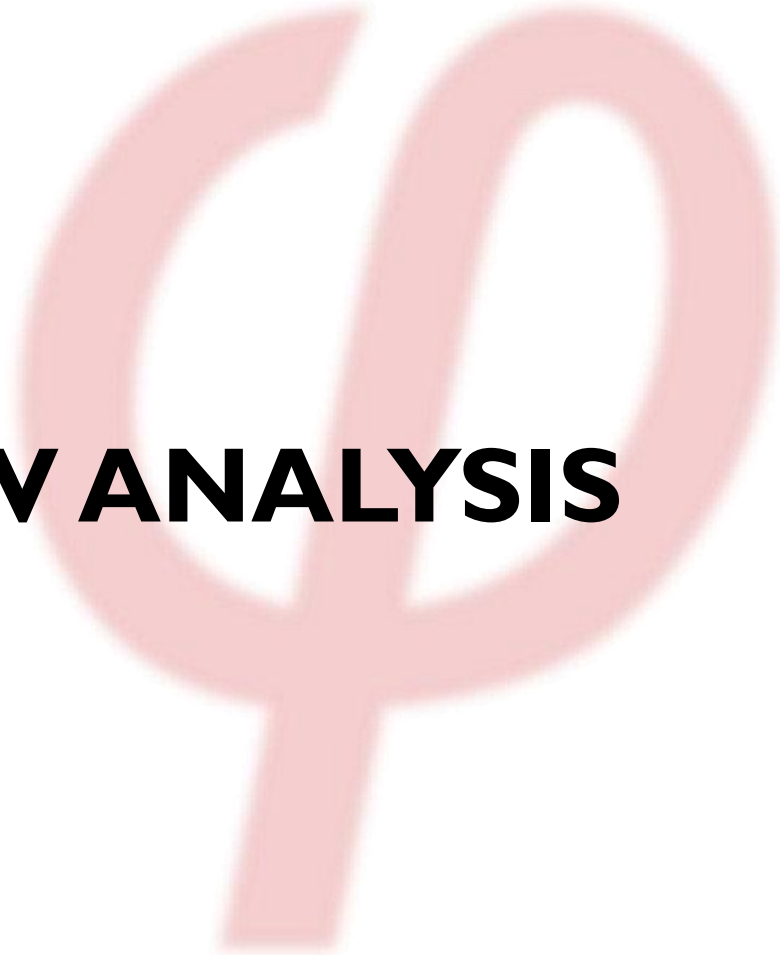
# CS 526

# Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

# CONTROL FLOW ANALYSIS

The slides adapted from Vikram Adve



# Flow Graphs

**Flow Graph:** A triple G=(N,A,s), where (N,A) is a (finite) directed graph, s ∈ N is a designated "initial" node, and there is a path from node s to every node n ∈ N.

- An *entry node* in a flow graph has no predecessors.
- An *exit node* in a flow graph has no successors.
- There is exactly one entry node, s. We can modify a general DAG to ensure this. *How?*

# Control Flow Graph (CFG)

**Flow Graph:** A triple G=(N,A,s), where (N,A) is a (finite) directed graph, $s \in N$ is a designated "initial" node, and there is a path from node s to every node $n \in N$.

**Control Flow Graph (CFG)** is a flow graph that represents all ***paths*** (sequences of statements) that might be traversed during program execution.

- Nodes in CFG are program statements, and edge $(S_1,S_2)$ denotes that statement $S_1$ can be followed by $S_2$ in execution.
- In CFG, a node unreachable from s can be safely deleted. *Why?*
- Control flow graphs are usually *sparse*. I.e., $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$.

# Control Flow Graph (CFG)

**Basic Block** is a sequence of statements $S_1$ ... $S_n$ such that execution control must reach $S_1$ before $S_2$, and, if $S_1$ is executed, then $S_2$ … $S_n$ are all executed in that order

- Unless a statement causes the program to halt

**Leader** is the first statement of a basic block

**Maximal Basic Block** is a basic block with a maximum number of statements (n)

# Control Flow Graph (CFG)

***Let us refine our previous definition***

**CFG** is a directed graph in which:

- Each node is a single basic block
- There is an edge b1 → b2 if block b2 *may* be executed after block b1 in *some* execution

We typically define it for a single procedure

A CFG is a conservative approximation of the control flow! Why?

# Example

**Source Code**

```
unsigned fib(unsigned n) {
   int i;
   int f0 = 0, f1 = 1, f2;

   if (n <= 1) return n;

   for (i = 2; i <= n; i++) {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
   }

   return f2;
}
```

**LLVM bitcode (ver 3.9.1)**

```
define i32 @fib(i32) {
  %2 = icmp ult i32 %0, 2
  br i1 %2, label %12, label %3

; <label>:3:
  br label %4

; <label>:4:
  %5 = phi i32 [ %8, %4 ], [ 1, %3 ]
  %6 = phi i32 [ %5, %4 ], [ 0, %3 ]
  %7 = phi i32 [ %9, %4 ], [ 2, %3 ]
  %8 = add i32 %5, %6
  %9 = add i32 %7, 1
  %10 = icmp ugt i32 %9, %0
  br i1 %10, label %11, label %4

; <label>:11:
  br label %12

; <label>:12:
  %13 = phi i32 [ %0, %1 ], [ %8, %11 ]
  ret i32 %13
}
```

# Dominance in Flow Graphs

Let d, d1, d2, d3, n be nodes in G.

d **dominates** n ("d dom n") *iff* every path from s to n contains d

d **properly dominates** n if d dominates n and d ≠ n

d **is the immediate dominator of** n ("d idom n")
if d is the last proper dominator on any path from initial node to n,

**DOM(**x**)** denotes the set of dominators of x,

**Dominator tree:** the children of each node d are the nodes n such that "d idom n" (immediately dominates)

# Dominator Properties

**Lemma 1:** DOM(s) = { s }.

**Lemma 2:** s dom d, for all nodes d in G.

**Lemma 3:** The dominance relation on nodes in a flow graph is a *partial ordering*

- *Reflexive— n dom n* is true for all n.
- *Antisymmetric —* If *d dom n*, then cannot be *n dom d*
- *Transitive — d1 dom d2* $\wedge$ *d2 dom d3* $\Rightarrow$ *d1 dom d3*

**Lemma 4:** The dominators of a node form a list.

**Lemma 5:** Every node except s has a unique immediate dominator.

# Finding Dominators in a Flow Graph

*Input* : A flow graph G = (N,A,s).

*Output* : The sets DOM(node) for each node ∈ N.

DOM(s) := { s }

forall $n \in N - \{s\}$ do
  DOM($n$) := $N$
od

while changes to any DOM($n$) occur do
  forall $n$ in $N - \{s\}$ do
    DOM($n$) := $\{n\} \bigcup \bigcap_{p \to n}$ DOM($p$)
  od
od

# Finding Dominators in a Flow Graph

*Input* : A flow graph G = (N,A,s).

*Output* : The sets DOM(node) for each node ∈ N.

DOM(s) := { s }

forall $n \in N - \{s\}$ do
  DOM($n$) := $N$
od

**Initialize**

while changes to any DOM($n$) occur do
  forall $n$ in $N - \{s\}$ do
    DOM($n$) := $\{n\} \bigcup \bigcap_{p \to n}$ DOM($p$)
  od
od

**Iterate**

# Loops

```
while (b) { … }  ⇒  ?
```

# Loops

***The right definition of "loop" is not obvious.***

**Obviously bad definitions**

- **Cycle:** Not necessarily properly nested or disjoint
- **Strongly Connected Components:**
  Too coarse; no nesting information

**What properties of the loops do we want to extract from CFG?**

# Natural Loops

**Def. Back Edge:** An edge n → d where d dom n

**Def. Natural Loop:** Given a back edge, n → d, the natural loop corresponding to n → d is the set of nodes ***{d + all nodes that can reach n without going through d}***

**Def. Loop Header:** A node d that dominates all nodes in the loop

- Header is unique for each natural loop *Why?*
- Implies d is the unique entry point into the loop
- Uniqueness is very useful for many optimizations

# Natural Loops

**Pros:**

+ Intuitive, and similar to SCC.
+ Single entry point: "loop header".
+ Identifies nested loops (if different headers)

**Cons:**

- Nested loops are not disjoint.
- Some nodes are not part of any natural loop.
- Does not include some cycles in "irreducible" flow graphs.

# Alternatives

**Natural loop**

- Defined using dominators

**Intervals**

- Defined in terms of reachability in flow graph (e.g. Muchnick, Sections 7.6 and 7.7)
- Main idea: split the flow graph in smaller regions (abstract nodes) that contain other nodes

# Reducibility of Flow Graphs

**Def. Reducible* flow graph**: a flow graph G is called reducible iff we can partition the edges into 2 disjoint sets:

- **forward edges**: should form a DAG in which every node is reachable from initial node s (or also header)
- **remaining edges must be back edges**: i.e., only those edges n → d such that *d dom n*

**Idea:**

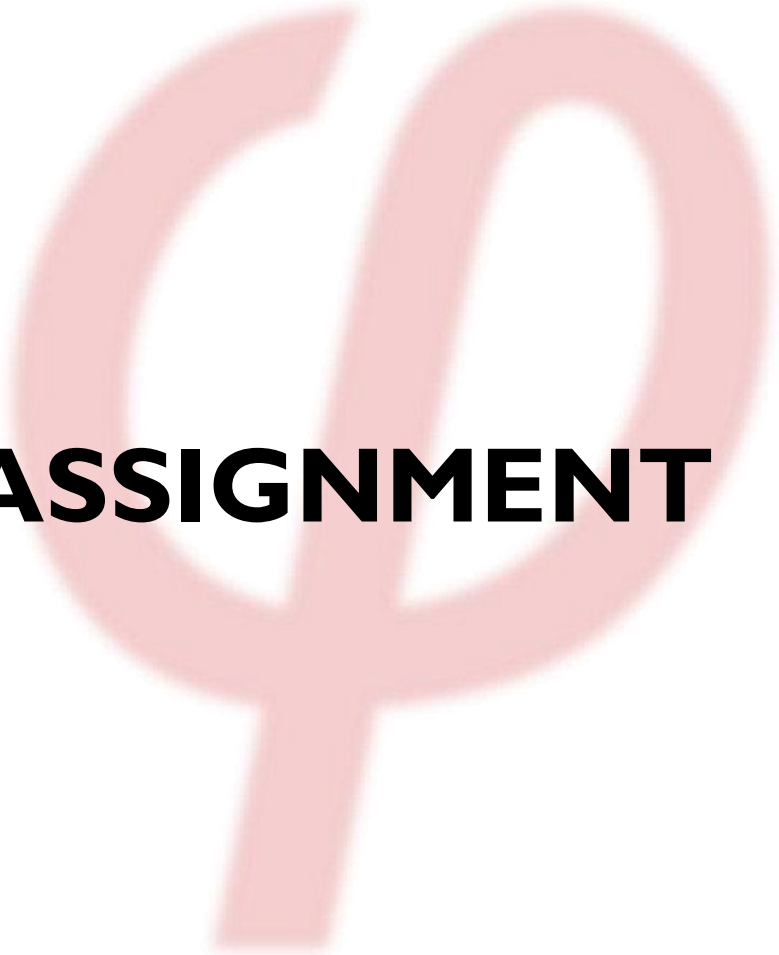
Every "cycle" has at least one back edge

⇒ All "cycles" are natural loops

Otherwise graph is called irreducible.

* Well-structured

# STATIC SINGLE ASSIGNMENT



The slides adapted from Vikram Adve

# References

Cytron, Ferrante, Rosen, Wegman, and Zadeck,
"Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. on Programming Languages and Systems*, 13(4), Oct. 1991, pp. 451–490.

Muchnick, Section 8.11 (*partially covered*).

Engineering a Compiler, Section 5.4.2 (*partially covered*).

***Beta Book***: SSA-Based Compiler Design
http://ssabook.gforge.inria.fr/latest/book.pdf

# What is SSA Form?

What is intermediate language?

**Design tradeoffs:**

- Expressive enough to represent source code information unambiguously
- Efficient for (numerous) optimizations and analyses
- Can easily generate backend code from it

Why do we study SSA ?

# What is SSA Form?

(Informally) A program can be converted into ***SSA form*** as follows:

- Each assignment to a variable is given a unique name
- All of the uses reached by that assignment are renamed.

Easy for straight-line code:

$$V \leftarrow 4$$
$$\ldots \leftarrow V + 5$$
$$V \leftarrow 6$$
$$\ldots \leftarrow V + 7$$

# What is SSA Form?

(Informally) A program can be converted into ***SSA form*** as follows:

- Each assignment to a variable is given a unique name
- All of the uses reached by that assignment are renamed.

Easy for straight-line code:

$$\begin{aligned} \mathbf{v_0} &\leftarrow 4 \\ \ldots &\leftarrow \mathbf{v_0} + 5 \\ \mathbf{v_1} &\leftarrow 6 \\ \ldots &\leftarrow \mathbf{v_1} + 7 \end{aligned}$$

# SSA Straight-line Code

```
X = 1;
X = X + 1;
Y = X;
```

# SSA Straight-line Code

```
X0 = 1;
X1 = X0 + 1;
Y = X1;
```

# SSA Straight-line Code

```
X = 1;
Y = f(X);
X = Y + X;
```

# SSA Straight-line Code

```
X0 = 1;
Y0 = f(X0);
X1 = Y0 + X0;
```

# SSA and Branches

```
if (...)
   X = 42;
else
   X = 7*7;

Y = X;
```

# SSA and Branches

```
if (...)
    X1 = 42;
else
    X2 = 7*7;
X3 = φ(X1, X2)
Y = X;
```

# SSA and Branches

```
if (...)
    X1 = 42;
else
    X2 = 7*7;
X3 = φ(X1, X2)
Y = X3;
```

# SSA and Branches

```
X = 0;
if (...)
    X = 42;

Y = X;
```

# SSA and Branches

```
X1 = 0;
if (...)
    X2 = 42;
X3 = φ(X1, X2)
Y = X;
```

# SSA and Branches

```
X1 = 0;
if (...)
    X2 = 42;
X3 = φ(X1, X2)
Y = X3;
```

# SSA and Branches

```
X = 0;
if (...) {
   if (...)
      X = 42;
   else
      X = 7*7;

}

Y = X;
```

# SSA and Branches

```
X0 = 0;
if (...) {
    if (...)
        X1 = 42;
    else
        X2 = 7*7;
    X3 = φ(X1, X2)
}
X4 = φ(X0, X3)
Y = X;
```

# SSA and Branches

```
X0 = 0;
if (...) {
    if (...)
        X1 = 42;
    else
        X2 = 7*7;
    X3 = φ(X1, X2)
}
X4 = φ(X0, X3)
Y = X4;
```

# SSA and Loops

```
j=1;

while (j < X)
    ++j;

N = j;
```

```
j = 1;

if (j >= X) goto E;

S:
j = j+1;
if (j < X) goto S;

E:
N = j;
```

# SSA and Loops

```
j=1;

while (j < X)
    ++j;

N = j;
```

```
j0 = 1;

if (j0 >= X) goto E;

S:
j = j+1;
if (j < X) goto S;

E:
N = j;
```

# SSA and Loops

```
j=1;

while (j < X)
    ++j;

N = j;
```

```
j0 = 1;

if (j0 >= X) goto E;

S: j1 = φ(j0, j2)
j2 = j1+1;
if (j2 < X) goto S;

E:
N = j;
```

# SSA and Loops

```
j=1;

while (j < X)
    ++j;

N = j;
```

```
j0 = 1;

if (j0 >= X) goto E;

S: j1 = φ(j0, j2)
j2 = j1+1;
if (j2 < X) goto S;

E: j4 = φ(j0, j2)
N = j4;
```

# SSA and Switches

```
X = 0;
switch (...) of
  a: X = 1;
  b: X = 2;
  c: X = 3;

Y = X;
```

# SSA and Switches

```
X0 = 0;
switch (...) of
  a: X1 = 1;
  b: X2 = 2;
  c: X3 = 3;

Y = X;
```

# SSA and Switches

```
X0 = 0;
switch (...) of
   a: X1 = 1;
   b: X2 = 2;
   c: X3 = 3;
X4 = φ(X0, X1, X2, X3)
Y = X4;
```

# Definition of $\varphi$ Function

In a basic block B with N predecessors, P1, P2, . . . , PN,

$$\mathbf{X = \varphi(V1, V2, \ldots, VN)}$$

assigns X = Vi if control enters block B from Pi , $1 \leq i \leq N$

**Properties of φ-functions:**

- φ is not an executable operation.
- φ has exactly as many arguments as the number of incoming basic block edges
- Think about the argument Vi as being evaluated on CFG edge from predecessor Pi to B

# More Definitions

**Value**: expression that cannot be evaluated further (numbers, words, memory addresses, …)

**Storage location:** register or memory address

- Machine and virtual registers
- Stack and heap

**Variable**: named storage location (map from name to address)

**Pointer**: variable whose value is another memory location

**Alias**: an alternative name of an entity (a variable, a location, …)

# More Definitions

**Use of a variable**: A use of variable X is a reference that may read the value stored in the location named X.

**Definition of a variable**: A definition (def) of a variable X is a reference that may store a value into the location named X. *Examples*: Assignment; input I/O

**Ambiguity of definitions**:

Unambiguous definition **(must)**: guaranteed to store to X

Ambiguous definition **(may)**: may store to X

*Q. Where does ambiguity come from?*

We define ambiguous/unambiguous use similarly.

# Def-Use Chains

- **Def-use chain**: The set of uses reached by a particular definition.

- **Use-def chain**: The set of definitions reaching a particular use

# Definition of SSA Form

A program is in SSA form if:

- each variable is assigned a value in **exactly one** statement
- each **use** of a variable is **dominated** by the **definition**

# Which Variables to Convert?

Convert all variables to SSA form, ***except*** . . .

**Arrays:** Array elements do not have an explicit name (although note ArraySSA)

**Variables that may have aliases:** do not have a unique name

**Volatile variables:** can be modified "unexpectedly"

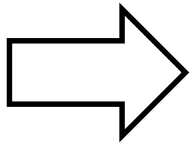
E.g., In LLVM, only variables in virtual registers are in SSA form.

# LLVM: Mem2reg

**-mem2reg: Promote Memory to Register**

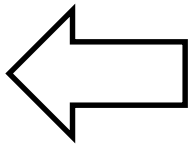
"This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. An alloca is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct "pruned" SSA form."

```
int f(int x) {
  int y = x + 1;
  return y
}
```



**No optimizations**

```
...
%6 = alloca i32, align 4
%7 = load i32, i32* %0, align 4
%8 = add nsw i32 1, %7
store i32 %8, i32* %6, align 4
%9 = load i32, i32* %6, align 4
ret i32 %9
```



**After mem2reg**

```
%3 = add nsw i32 %0, 1
ret i32 %3
```

# Advantages of SSA Form

**Makes def-use and use-def chains explicit:**

These chains are foundation of many dataflow optimizations

- We will see some soon!

**Compact, flow-sensitive* def-use information**

- fewer def-use edges per variable: one per CFG edge

* Takes the order of statements into account

# Advantages of SSA Form (cont.)

**No anti- and output dependences** on SSA variables

- Direct dependence: **A=1; B=A+1**
- Antidependence: A=1; **B=A+1; A=2**
- Output dependence: **A=1; A=2;** B=A+1

} **Cannot reoder**

**Explicit merging of values (φ):** key additional information

Can serve as **IR for code transformations** (see LLVM)

# Disadvantages of SSA Form

**Size** of SSA program is $O(N^2)$ for an ordinary program with N variables.

**Often Not used** for structures and arrays

**May not be used** for scalar variables with aliases

**If used as IR**, must be converted back to code (Not bad)

**Otherwise,** must be recomputed frequently (Often bad)