Interference Graph

Nodes: ***Webs, machine registers***

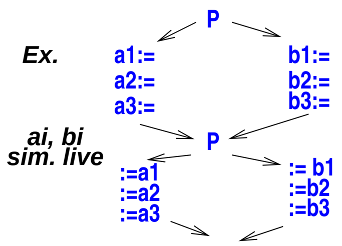
Undirected Edges*:*

X* ——— *Y

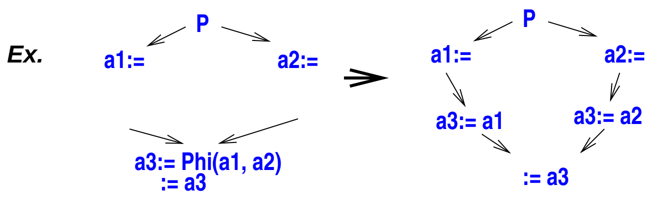
X and Y simultaneously live , or

Y live at a def of X

Which is better?

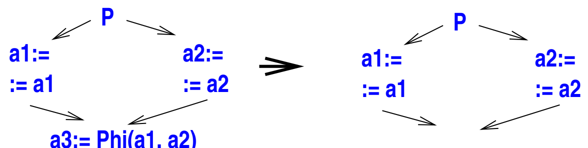


Getting Out of SSA



May need to split critical edges

May want to do Dead Code Elimination before



no further uses a3 ***No copies needed!***

Finding Dead Code

Initially, all statements are ***dead***

A statement is ***live*** if

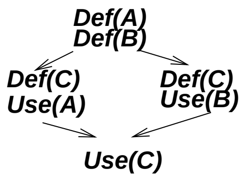
1. It affects program output (e.g. I/O)
2. It is an assignment to a variable, and there are live statements that use the variable's value.
3. The statement is a conditional branch, and there are statements marked live that are control dependent on the branch.

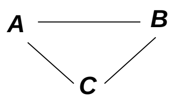
All statements not marked live (except for unconditional branches) are deleted.

May wind up with empty basic blocks, which can be deleted.

Coloring and SSA

Ex.

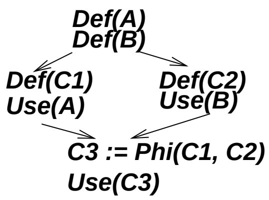


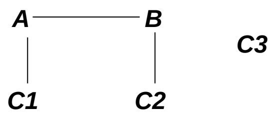


Maxlive = 2, need 3 colors

Maxlive = Maximal number of live variables at any point p

SSA:





2 colors sufficient!

Coloring and SSA (2)

Can color the IG for SSA with Maxlive colors in polynomial time

Idea: Can use join point to start new color, if needed

Vs: Coloring IG with minimum # colors is NP-complete

Getting out of SSA:

Useful to do coloring after copies inserted

Code may have changed due to optimization

Ex.

```
while (P) do
    read(V)
    W:= V+W
    V:= 6
    W:= V+W
endwhile
```

→

```
while (P) do
    W3:=Phi(W0,W2)
    V3:=Phi(V0,V2)
    read(V1)
    W1:= V1+W3
    V2:= 6
    W2:= V2+W1
endwhile
```

→

```
V2:= 6
while (P) do
  W3:=Phi(W0,W2)
  read(V1)
  W1:= V1+W3
  W2:= V2+W1
endwhile
```


Basic Block Dependency DAG

Nodes: ***Machine or LIL instruction***

Directed Edges : ***Dependences between instructions***

Useful for BB scheduling

Gives necessary dependences that must be satisfied by any schedule

	true	anti	output	move	resource
I1 ↓	*write*	*use*	*write*	*Ex. load ()*	*l+1 cycles of I1 before I2*
I2	*use*	*write*	*write*	*store()* may be same mem. location	*l is the latency*

Building the Dependence DAG

Need # cycles each resource needed for each instruction

Latency(I1,I2,n) = # latency cycles incurred by beginning I2 at nth cycle of I1

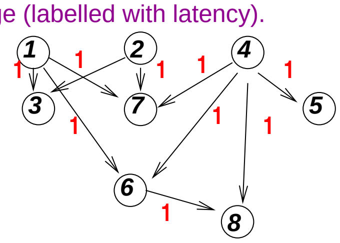
Linear pass over instructions I1 in BB. ***$O(N^2)$***

Determine if any I2 conflicts with I1.

If I1 has no conflicts, then it is a root of the DAG.

If conflict with I2, need new edge (labelled with latency).

Ex.(1) r3:= [r15] (4)
(2) r4:= [r15+4] (4)
(3) r2:= r3-r4
(4) r5:= [r12] (4)
(5) r12:= r12 + 4
(6) r6:= r3*r5
(7) [r15+4](4) := r3
(8) r5:= r6+2



Instruction Scheduling on Dep. DAG

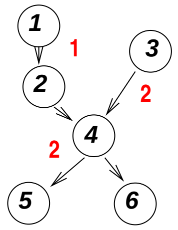
Any topological order will work

Can only execute node when all preds are already executed

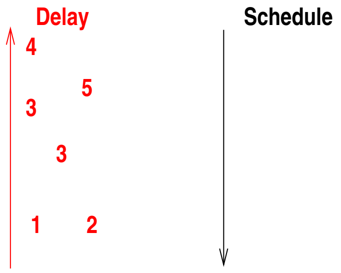
Want to minimize execution time

NP-hard: Need heuristic

Ex.



Exectime(6) = 2, all others 1



When choice, choose largest delay