CSc 553

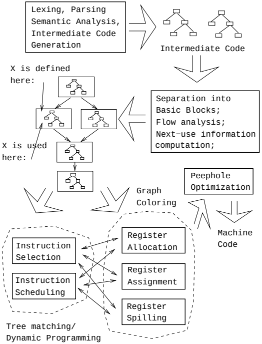
Principles of Compilation

23 : Register Allocation

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction



Register Allocation by Graph Coloring

Register Allocation

- Register allocation is difficult:
  1. Machines have weird instruction sets, register pairs (two consecutive registers that are the source or destination in an instruction), register classes (address, integer, index, floating),...
  2. Optimal solutions to the register allocation problem is NP-complete.
- Most compilers use complicated ad hoc heuristic register allocation algorithms. It would be helpful if we had a good model for register allocation the way we have finite automata for lexical analysis, attribute grammars for semantic analysis, etc.
- We can model register allocation using undirected graphs.

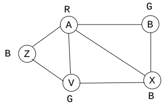
Graph Coloring

- Model register allocation as a graph coloring problem. Each color represents an available register.
- Create a graph node for each variable. If variables a and b are *active* (live) at the same point, they cannot be assigned to the same register. Add an edge (a, b) to the graph.
- Look for a k-coloring ($k = \#$ registers) of the graph. Assign colors so that neighboring nodes have different colors.
- If we cannot k-color our graph, we:
  1. Select a node (variable) n whose value we're willing to spill,
  2. Insert spill code,
  3. Delete node n and its edges,
  4. Look for a k-coloring.

The Interference Graph I

- The interference graph (an undirected graph where the nodes are the variables of the program) models which variables cannot be allocated to the same register.
- Connect a and b if a is live at a point where b is defined.

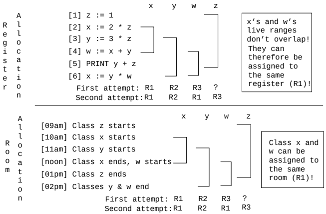
(1)	a	:=	5		
(2)	d	:=	9	+	a
(3)	e	:=	a	+	d
(4)	b	:=	d	+	a
(5)	f	:=	e	+	6
(6)	c	:=	b	+	f



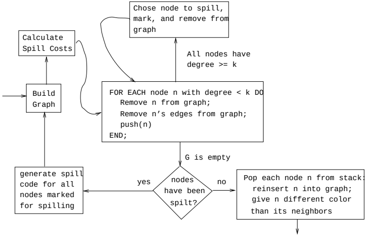
The Interference Graph II

- Register allocation is a bit like room scheduling.
- Room scheduling:
  1. We have a set of rooms (registers).
  2. We have a set of classes (variables) to fit into the rooms.
  3. Two classes that meet at the same time cannot be allocated to the same room.
- The difference is that in room scheduling there can be no **spilling**; no-one gets to have their lecture in the park!
- A variable's **live range**
  1. starts at the point in the code where the variable receives a value, and
  2. ends where that value is used for the last time.

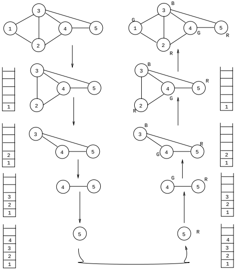
The Interference Graph III



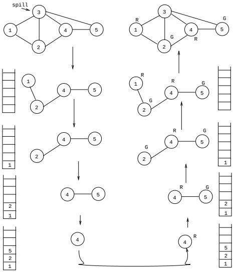
Chaitin's Coloring Algorithm



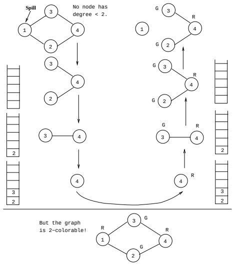
Coloring Example I (a) – $k = 3$



Coloring Example I (b) – $k = 2$



Coloring Example II – $k = 2$



Precoloring

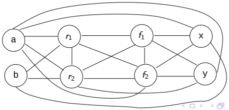
Precolored Nodes I

- Sometimes we will want to express that a particular variable **must** reside in a particular register. For example, if variable a is being passed as argument 1 to procedure P on the SPARC, we'd want to express that a must reside in register %o0, and nowhere else.
- Similarly, sometimes we want to express that a particular variable **must not** reside in a particular register. For example, a floating point variable should not be in an integer register.
- Such variables are *precolored*.
- We augment the interference graph with nodes for each available register, and an edge between variable a and register r if a cannot be allocated to r.

Precolored Nodes II

```
VAR x, y :  INTEGER;
VAR a, b :  REAL;
x := 100;
a := 1.0;
b := a + 5.2;
y := x + 50;
P(y,a);
```

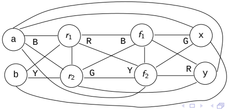
- We have two integer registers r_1 and r_2, and two FP registers f_1 and f_2.
- Procedure actuals are passed in registers: y in r_1 and a in f_1.



Precolored Nodes III

```
VAR x, y :  INTEGER;
VAR a, b :  REAL;
x := 100;
a := 1.0;
b := a + 5.2;
y := x + 50;
P(y,a);
```

- We color y and r_1 red (R), x and r_2 green (G).
- We color a and f_1 blue (B), b and f_2 yellow (Y).



Register Coalescing

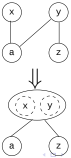
Register Coalescing

- Register coalescing is a kind of copy propagation that removes register copies.
- Search the intermediate code for copies $S_j \leftarrow S_i$ such that S_j and S_i don't interfere with each other.
- Modify any instruction $S_i \leftarrow \cdots$ to $S_j \leftarrow \cdots$ and merge the interference graph nodes for S_j and S_i.

```
x := 100;
a := x * 2;
y := x;
z := y + a;
PRINT y,z;
```

$\Downarrow$

```
x := 100;
a := x * 2;
z := x + a;
PRINT x,z;
```



Splitting Live Ranges

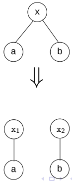
Splitting Live Ranges

- If we use the same variable for several unique tasks (e.g. `i` for all for-loops) the interference graph is overly constrained.
- Instead we let each graph node represent a unique use of a variable.

```
x := 100;
a := x * 2;
PRINT a;
x := 200;
b := x + 5;
PRINT b;
```

$\Downarrow$

```
x1 := 100;
a := x1 * 2;
PRINT a;
x2 := 200;
b := x2 + 5;
PRINT b;
```



Building the Interference Graph

Building the Interference Graph I

- We start by performing a liveness analysis.
- in [B] — Variables live on entrance to B.
- out [B] — Variables live on exit from B.
- def [B] — Variables assigned values in B before the variable is used.
- use [B] — Variables whose values are used before being assigned to.

Data-flow Equations:

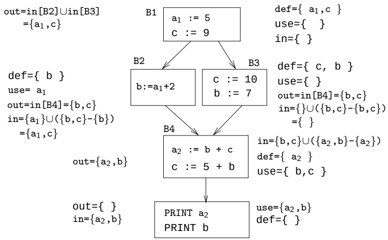
$$
\begin{aligned}
\texttt{in}[\texttt{B}] &= \texttt{use}[\texttt{B}] \cup (\texttt{out}[\texttt{B}] - \texttt{def}[\texttt{B}]) \\
\texttt{out}[\texttt{B}] &= \bigcup_{\texttt{succs}\ S\ \texttt{of}\ B} \texttt{in}[\texttt{S}]
\end{aligned}
$$

Building the Interference Graph II

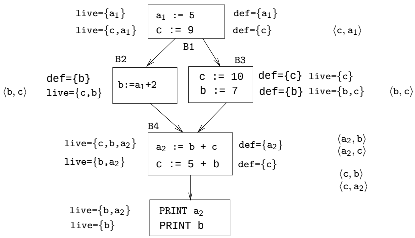
- Then we build the graph. For efficiency, we store it both as an adjacency matrix, and as adjacency lists.

```
FOR all basic blocks b in the program DO
   live := out[b];
   FOR all instructions I ∈ b, in reverse order DO
      FOR all d ∈ def(I) DO
         FOR all l ∈ live ∪ def(I) DO
            add the interference graph edge ⟨l, d⟩;
      live := use(I) ∪ (live − def(I));
```

Building the Interference Graph III



Building the Interference Graph IV



Building the Interference Graph V

- Here's the finished interference graph:

	a_1	a_2	b	c
a_1				√
a_2			√	√
b		√		√
c	√	√	√	

Summary

Readings and References

- Read the Tiger Book, Chapter 11, Register Allocation.
- The Dragon book: 513–521, 528–546, 554–559.
- Preston Briggs' thesis: *Register Allocation via Coloring*, `http://cs-tr.cs.rice.edu:80 /Dienst/Repository/2.0/Body/ ncstrl.rice_cs/TR92-183/postscript`.
- Steven Muchnick, *Advanced Compiler Design and Implementation*, Chapter 16, pp. 481–525.

Summary

- Graph coloring can be used to model register allocation. Each variable becomes a node in the graph. If two variables can't reside in the same register, we add en edge between them.
- The coloring algorithm assigns colors so that no neighboring nodes receive the same color.
- Optimal coloring is NP-complete (at least for **global** register allocation), so we need a heuristic algorithm that produces a good approximation.

Homework

Register Allocation by Graph Coloring

Homework III – Graph Coloring

- Construct the interference graph for the basic block below, and show the coloring produced by Chaitin's algorithm when two and three registers are available. Spill costs are X=3,Y=1,Z=2,V=2.

```
X := 5;
Y := X + 3;
Z := X + 5;
V := Y + 6;
X := X + Y;
X := V + Z;
```

Homework IV – Graph Coloring

- Construct the flow-graph and the interference graph for the procedure body below, and show the global coloring produced by Chaitin's algorithm when two and three registers are available. Spill costs are X=1,Y=2,Z=3,W=1,V=2.

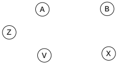
BEGIN
  X := ⋯; Z := ⋯;
  IF e_1 **THEN** Z := ⋯;
  ELSE Y := ⋯;
  ENDIF;
  ⋯ := X; ⋯ := Y;
  W := ⋯; V := ⋯;
  IF e_2 **THEN** ⋯ := W; ⋯ := Z;
  ELSE ⋯ := V;
  ENDIF;
  ⋯ := V + W;
END

Exam Problem I(a) (415.430 '95)

- Consider the following basic block:

```
X := 5;
A := X + 5;
B := X + 3;
V := A + B;
A := X + 5;
Z := V + A;
PRINT Z, V, A;
```

1. Construct the register interference graph for the block.

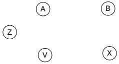


2. How many colors are necessary to color the graph optimally without register spills?

Exam Problem I(b) (415.430 '95)

```
X := 5;
A := X + 5;
B := X + 3;
V := A + B;
A := X + 5;
Z := V + A;
PRINT Z, V, A;
```

3. Show the graph after it has been colored with Chaitin's algorithm using 2 colors (Red and Blue). The spill-costs are: A=1, Z=2, B=3, V=2, X=4.

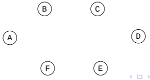


Exam Problem II/a (415.730 '96)

Consider the following basic block:

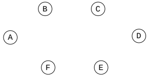
```
A := 5;
F := A + 1;
E := F + 5;
B := F * A;
PRINT B + E + A;
D := E + 5;
PRINT E;
C := D + B;
PRINT E + C;
```

1. Construct the register interference graph for the block.



Exam Problem II/b (415.730 '96)

2. How many colors are necessary to color the graph optimally without register spills?
3. Show such an optimal coloring!



4. Show the graph after it has been colored with Chaitin's algorithm using 2 colors (Red and Blue). The spill-costs are: C=1, D=2, E=3, B=A=4, F=5.