Where we are

Course so far

- Program representation
- Data flow analysis
- SSA
- Optimizations
- Scheduling (just breezed the surface)

On the Horizon

- Register Allocation
- Compilers for Object-oriented languages: JIT, GC, locality
- Experimental Computer Science in a modern context
- Aliases (brief)
- Interprocedural Analysis (brief)
- Dependence, Parallelism, & Locality analysis (intro)
- Next generation architectures

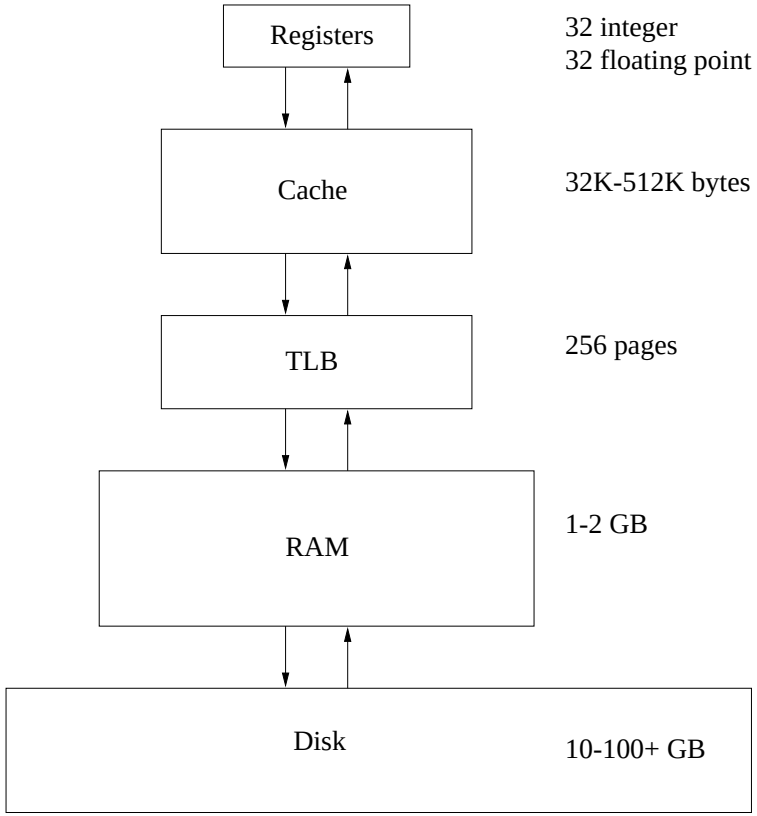
Today

- Register Allocation
- Chaitin et al. Graph Coloring Register Allocation

Register Allocation via Graph Coloring

- The abstraction
- G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," *Computer Languages*, 6(1), January 1981.
- P. Briggs, K. Cooper, and L. Torczon, "Improvements to Graph Coloring Register Allocation," *Transactions on Programming Languages and Systems*, 16(3), May 1994.
- F. Chow and J. Hennessy, "The Priority-Based Coloring Approach to Register Allocation," *Transactions on Programming Languages and Systems,* 12(4), October 1990.
- Traub, Holloway, and Smith, "Quality and Speed in Linear-scan Register Allocation," *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* June 1998, pp. 142-151.

Registers



Register Allocation

Backus et al. suggested a simplifying separation of concerns:

> *During optimization, assume an infinite set of registers; treat register allocation as a separate problem.*

Register allocation may be performed at many levels:

- Expression
- Local (basic block)
- Loop
- Global (method)
- Interprocedural

$\Longrightarrow$ Global optimization requires global register allocation.

Assumptions

- low-level intermediate or assembly code
- Unlimited number of *virtual* registers
- global optimization has been performed

What's Hard?

- Control flow, especially loops
- Machine dependent
- 2- and 3-address instructions
- Calling conventions (caller saves vs. callee saves)
- Restricted instructions
- Register pairs

Finally, there are practical considerations – space and time.

Graph coloring offers a simplifying abstraction.

Allocation via Coloring

- A *value* corresponds to a definition,
- A *live range* is composed of one or more values, connected by common uses. *based on stmts*
- A single virtual register name may comprise several live ranges

An *interference graph* is constructed, where

- Vertices represent live ranges.
- Edges represent *interferences* between two live ranges, *i.e.*, both ranges are live at some point and cannot occupy the same register.
- A coloring represents a register assignment.

Note that the abstraction subtly changes our goals; however, the separation of optimization and allocation justifies the goal of minimal coloring.

Interference

Two live ranges interfere if, at some point in the routine,

- Both live ranges have been defined,
- Both live ranges will be used, and
- The live ranges have different values.

Since these conditions are undecidable, we use a conservative approximation.

At each definition in the routine, we make the defined live range interfere with all live ranges that

- Are *available*
- Are *live*

However, if the definition is a copy instruction, we don't add an interference between the source and destination live ranges.

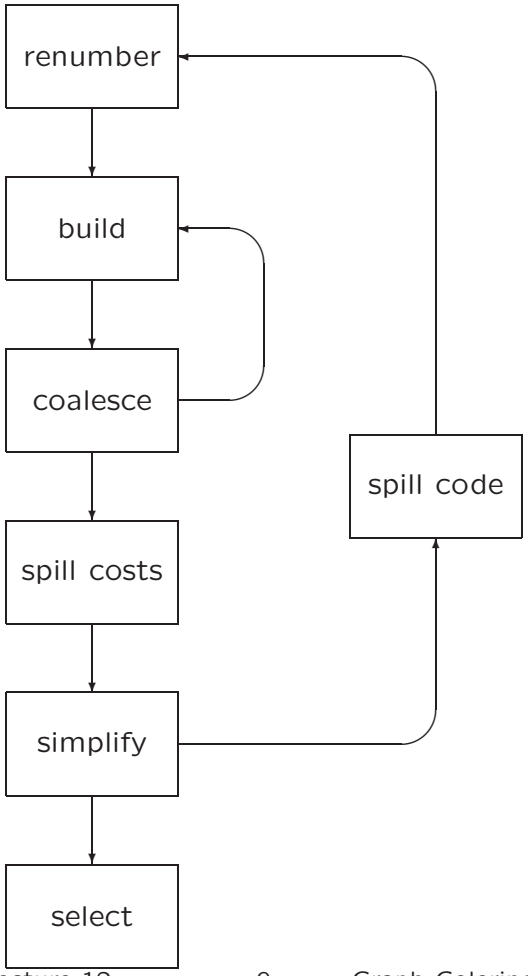
Example

```
        1. read a
        2. read b

        3. if (a > b) goto 4

4. c = a/2          5. c = b/2

        6. d = c
```

Interference Graph:

variable	live range
a	
b	
c	
d	

Chaitin's Structure



Chaitin's phases

renumber: Find all distinct live ranges and number them uniquely

build: Construct the interference graph

coalesce: For each copy where the source and destination live ranges don't interfere, union the two live ranges and remove the copy

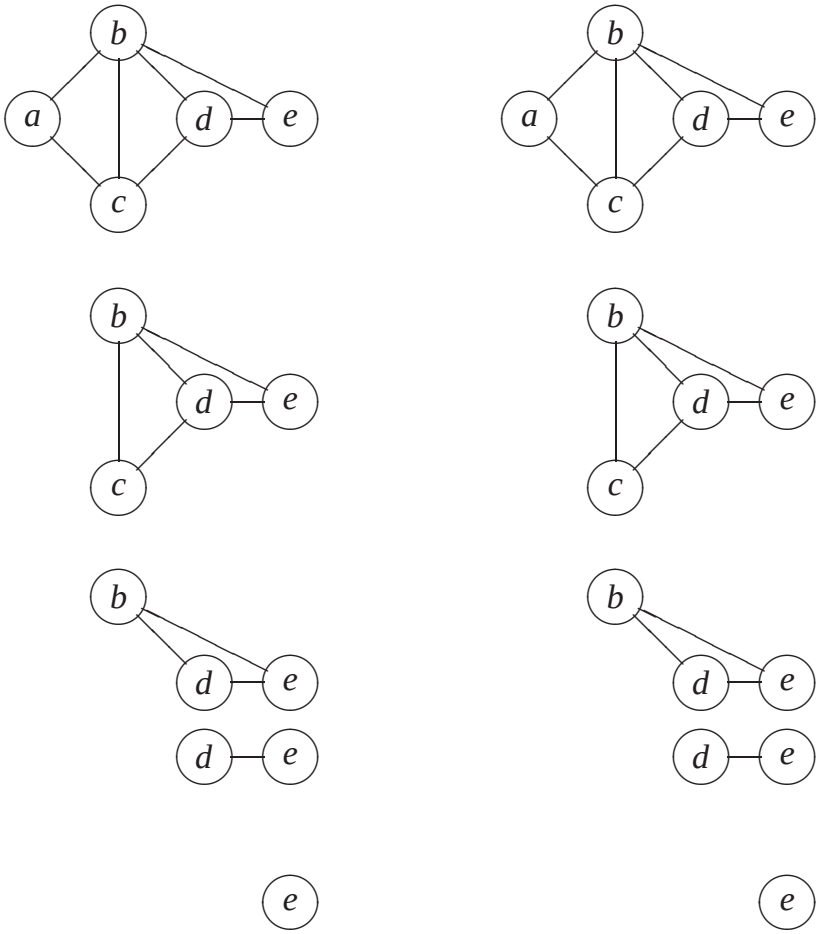
spill costs: Estimate the dynamic cost for spilling each live range (assumes only register-to-register operations)

simplify: Repeatedly remove nodes with degree $< k$ from the graph and push them on a stack. If every remaining node is degree $\geq k$, select a node, mark it for spilling and remove it from the graph.

spill code: For spilled nodes, insert a load or store at each use or definition and iterate.

select: Reassemble the graph with nodes popped from the stack. As each node is added to the graph, choose a color differing from neighbors in the graph.

Example



The Interference Graph

The representation of the interference graph is the major factor driving time and space requirements of the allocator (and the compiler itself, in many cases).

Required operations are:

new(n) Return a new graph with n vertices, but no edges.

add(g,x,y) Return a graph including g and an edge between the vertices x and y.

interfere(g,x,y) Return *true* if there's an edge between x and y in the graph g.

degree(g,x) Return the number of neighbors of x in g.

Finally, we need an efficient way to visit each of the neighbors of a given node.

Some large routines have O(5,000) vertices and O(1,000,000) edges.

The Interference Graph

Chaitin and Briggs use a dual representation:

- A triangular bit matrix, supporting efficient random access, and
- A vector of adjacency vectors, supporting efficient access to the neighbors of a node.

The structures are initialized in two passes over the code.

- Initially, allocate and clear the bit matrix. During the first pass, fill in the matrix and accumulate the number of neighbors for each vertex.
- Before the second pass, allocate the adjacency vectors and clear the bit matrix. During the second pass, rebuild the bit matrix, adding entries to the adjacency vectors.
- Since coalescing corrupts the graph, the second pass must be repeated until all coalescing is complete.
- After coalescing, the space for the bit matrix may be reclaimed.

Chaitin's Contribution

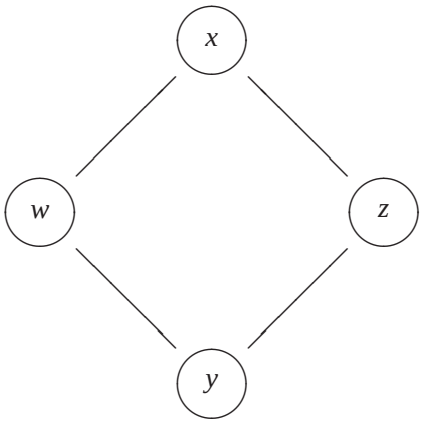
The first complete allocator based on graph coloring was described by Chaitin *et al.* in 1981. They developed

- A precise notion of *interference,*
- *Subsumption,*
- An efficient coloring heuristic, and
- Efficient data structures and algorithms for representing and manipulating the interference graph (a bit-matrix and a collection of adjacency lists).

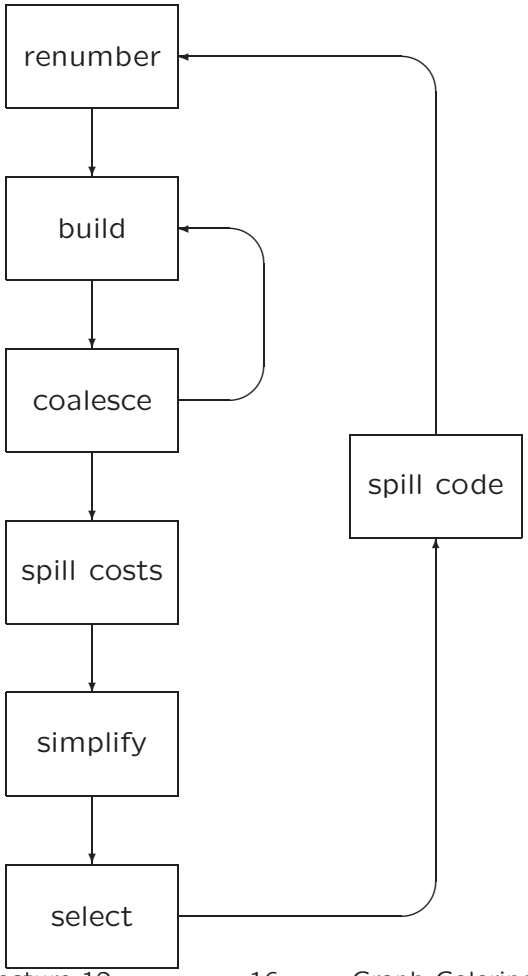
Additionally, they showed how to manipulate the interference graph in a systematic fashion to account for common machine "features."

Assumes each operand must be in a register, *i.e.,* only register-to-register operations.

A Problem



Briggs et al.'s Optimistic Allocator



Briggs' phases

renumber Find all distinct live ranges and number them uniquely

build Construct the interference graph

coalesce For each copy where the source and destination live ranges don't interfere, union the two live ranges and remove the copy

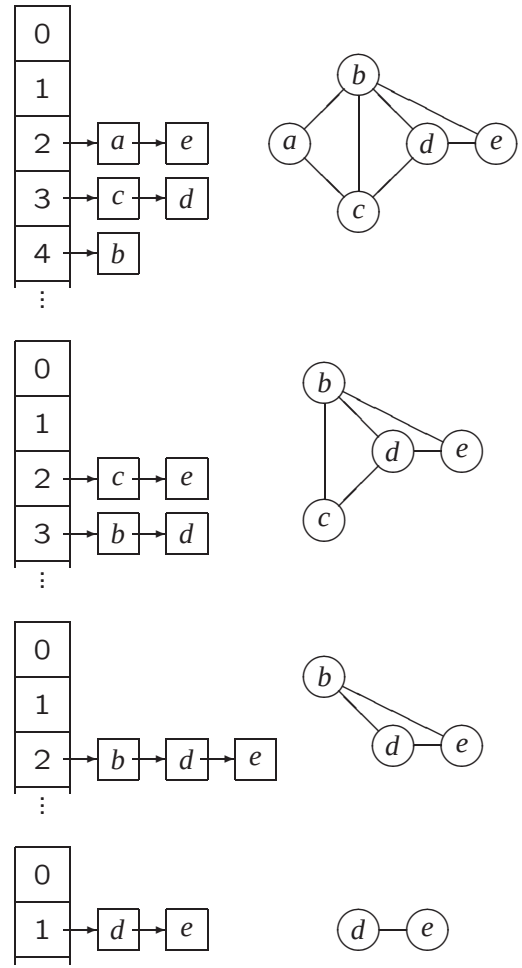
spill costs Estimate the dynamic cost for spilling each live range

simplify Repeatedly remove nodes with degree $< k$ from the graph and push them on a stack. When necessary, choose spill candidates and push them on the stack

select Reassemble the graph with nodes popped from the graph. As each node is added to the graph, choose a color differing from neighbors in the graph. If no color is available, the node is left uncolored

spill code Spill uncolored nodes by inserting a load or store at each use or definition

Optimistic Coloring — 2 colors



Example

```
Subroutine svd(nm,m,n,a,w,mu,u,mv,v,e,r)

    DO i = 1, m
        DO j = 1, n
            u(i,j) = a(i,j)
    DO
        9 doubly-nested loops
        3 triply-nested loops
    DO
        3 doubly-nested loops
        2 triply-nested loops
    DO
        2 doubly-nested loops
        1 triply-nested loop
    DO
        4 doubly-nested loops
        2 triply-nested loops
```


Spilling

Generally, the spill cost of a live range is a weighted sum of the number of uses and definitions, where each use and definition is weighted proportionally to its loop nesting depth.

However, this neglects two important refinements mentioned by Chaitin:

1. There's no benefit in spilling a live range between two uses or between a definition and use if no other live ranges dies in the interval.
2. Some live ranges should be *rematerialized* instead of being spilled to memory.

Useful Spilling

By spilling during simplification, Chaitin is assured that there will always be a color available during selection.

Instead of spilling, Briggs pushes the least expensive spill candidate on the stack. *All nodes go on the stack*.

Instead of asking
"is x of degree $< k$?"

Briggs asks
"does x get a color?"

By deferring spill decisions, Briggs wins twice:

- when neighbors of a node are the same color, and
- when a neighboring node has already been spilled.

Quicksort

	Regs	16	14	12	10	8
Spilled	Chaitin	3	5	8	13	17
	Briggs	3	4	6	8	11
	%		20	25	38	35
Spill Cost	Chaitin	1,303	5,105	11,809	37,000	125,000
	Briggs	1,303	2,568	7,750	15,875	71,675
	%		50	34	57	43
Object Size	Chaitin	360	384	400	440	464
	Briggs	360	368	392	416	432
	%		4	2	5	7
Run Time	Chaitin	8.2	8.3	8.7	10.0	13.2
	Briggs	8.2	8.2	8.4	8.9	11.2
	%		1	3	11	15

Briggs in perspective

Comparing Chaitin and Briggs:

- Chaitin's method colors a subset of the graphs Briggs colors.
- Any live-range Briggs spills, Chaitin spills.
- Chaitin often spills more registers.
- Improvements can be significant ($> 10\%$).
- Briggs achieves the same time bounds as Chaitin.

Next Time

Read: More of Brigg's thesis.