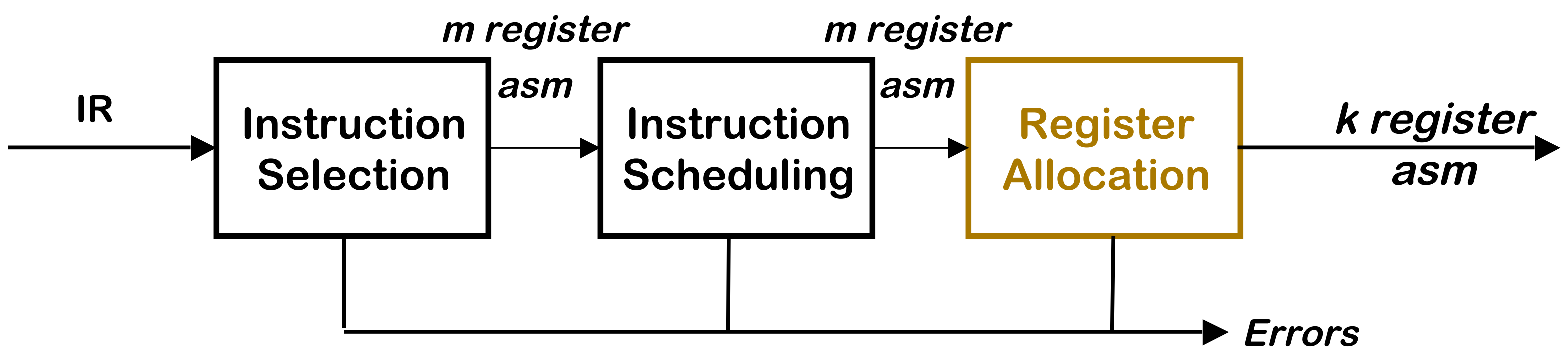
Global Register Allocation via Graph Coloring

COMP 412
Fall 2005

Register Allocation

Part of the compiler's back end



Critical properties

- Produce <u>correct</u> code that uses *k* (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold *spilled values*
- Operate efficiently

  $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

Global Register Allocation

The big picture

Optimal global allocation is NP-Complete, under almost any assumptions.

At each point in the code

1 Determine which values will reside in registers

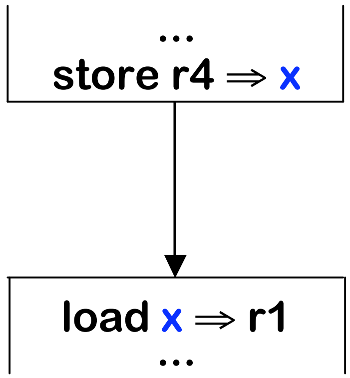
2 Select a register for each such value

The goal is an allocation that “minimizes” running time

Most modern, global allocators use a graph-coloring paradigm

- Build a “conflict graph” or “interference graph”
- Find a *k*-coloring for the graph, or change the code to a nearby problem that it can *k*-color

Global Register Allocation

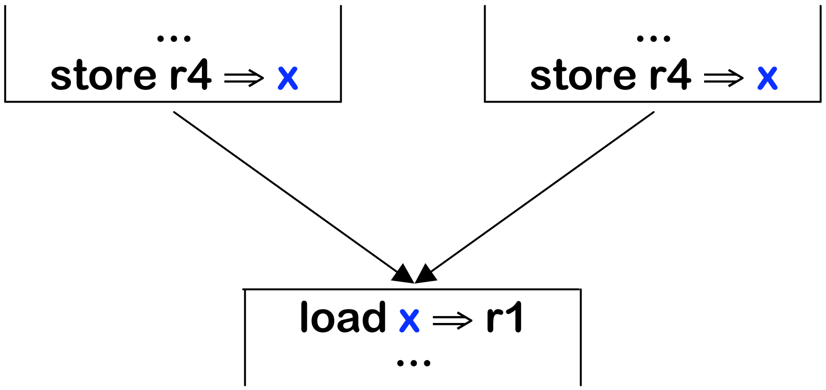


This is an assignment problem, not an allocation problem !

What's harder across multiple blocks?

- Could replace a load with a move
- Good assignment would obviate the move
- Must build a control-flow graph to understand inter-block flow
- Can spend an inordinate amount of time adjusting the allocation

Global Register Allocation



What if one block has x in a register, but the other does not?

A more complex scenario

- Block with multiple predecessors in the control-flow graph
- Must get the “right” values in the “right” registers in each predecessor
- In a loop, a block can be its own predecessor

This adds tremendous complications

Global Register Allocation

Taking a global approach

- Abandon the distinction between local & global
- Make systematic use of registers or memory
- Adopt a general scheme to approximate a good allocation

Graph coloring paradigm (*Lavrov & (later) Chaitin*)

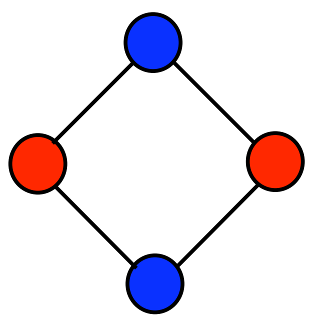
1. Build an interference graph G_I for the procedure
   - Computing LIVE is harder than in the local case
   - G_I is not an interval graph
2. (try to) construct a *k*-coloring
   - Minimal coloring is NP-Complete
   - Spill placement becomes a critical issue
3. Map colors onto physical registers

Graph Coloring (A Background Digression)

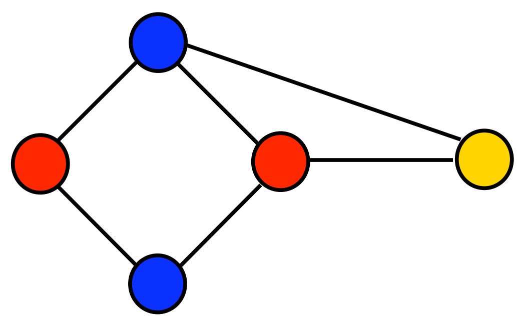
The problem

A graph *G* is said to be *k-colorable* iff the nodes can be labeled with integers 1… k so that no edge in G connects two nodes with the same label

Examples



2-colorable



3-colorable

Each color can be mapped to a distinct physical register

Building the Interference Graph

What is an "interference" ? (or conflict)

- Two values *interfere* if there exists an operation where both are simultaneously live
- If x and y interfere, they cannot occupy the same register

To compute interferences, we must know where values are "live"

The interference graph, $G_I = (N_I, E_I)$

- Nodes in G_I represent values, or live ranges
- Edges in G_I represent individual interferences
  - For $x, y \in N_I$, $\langle x,y \rangle \in E_I$ iff x and y interfere
- *A k*-coloring of G_I can be mapped into an allocation to *k* registers

Building the Interference Graph

To build the interference graph

1. Discover live ranges
   - Build SSA form
   - At each ϕ-function, take the union of the arguments
   - Rename to reflect these new "live ranges"
2. Compute LIVE sets over live ranges for each block
   - Use an iterative data-flow solver
   - Solve equations for LIVE over domain of live range names
3. Iterate over each block, from bottom to top
   - Track the current LIVE set
   - At each operation, add appropriate edges & update LIVE
     - Add an edge from result to each value in LIVE
     - Remove result from LIVE
     - Add each operand to LIVE

Update LIVE

Computing LIVE Sets

A value v is live at p if ∃ a path from p to some use of v along which v is not re-defined

Data-flow problems are expressed as simultaneous equations

$$\text{LIVEOUT}(b) = \cup_{s \in succ(b)} \text{LIVEIN}(s)$$

$$\text{LIVEIN}(b) = \text{UEVAR}(b) \cup (\text{LIVEOUT}(b) \cap \overline{\text{VARKILL}(b)})$$

$$\text{LIVEOUT}(n_f) = \varnothing$$

§ 9.2.1 in EaC

where

UEVAR(b) is the set of names used in block b before being defined in b

VARKILL(b) is the set of names defined in b

Solve the equations using a fixed-point iterative scheme

Computing LIVE Sets

The compiler can solve these equations with an iterative algorithm

```
WorkList ← { all blocks }
while ( WorkList ≠ Ø)
    remove a block b from WorkList
    Compute LIVEOUT(b)
    Compute LIVEIN(b)
    if LIVEIN(b) changed
        then add pred (b) to WorkList
```

The Worklist Iterative Algorithm

Why does this work?

- LIVEOUT, LIVEIN $\subseteq 2^{Names}$
- UEVAR, VARKILL are constants for b
- Equations are monotone
- Finite # of additions to sets

⇒ will reach a fixed point !

Speed of convergence depends on the order in which blocks are "removed" & their sets recomputed

This is the world's quickest introduction to data-flow analysis !

Observation on Coloring for Register Allocation

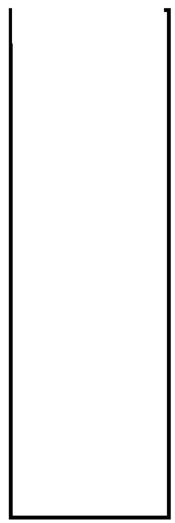
- Suppose you have *k* registers—look for a *k* coloring
- Any vertex *n* that has fewer than *k* neighbors in the interference graph ($n^\circ < k$) can always be colored!
  - Pick any color not used by its neighbors — there must be one
- Ideas behind Chaitin's algorithm:
  - Pick any vertex *n* such that $n^\circ < k$ and put it on the stack
  - Remove that vertex and all edges incident from the interference graph
    - This may make additional nodes have fewer than k neighbors
  - At the end, if some vertex *n* still has k or more neighbors, then spill the live range associated with *n*
  - Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbor

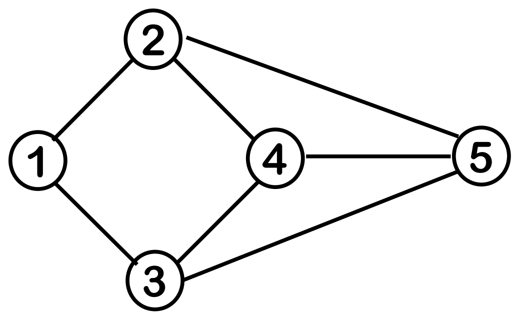
Chaitin's Algorithm

1. While ∃ vertices with < k neighbors in G_I
   > Pick any vertex n such that $n^\circ < k$ and put it on the stack
   > Remove that vertex and all edges incident to it from G_I
     - This will lower the degree of n's neighbors
2. If G_I *is non-empty* (all vertices have k or more neighbors) then:
   > Pick a vertex n (using some heuristic) and spill the live range associated with n
   > Remove vertex n from G_I, along with all edges incident to it and put it on the stack
   > If this causes some vertex in G_I to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2
3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor

Chaitin's Algorithm in Practice

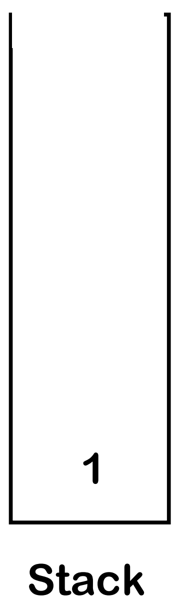
3 Registers

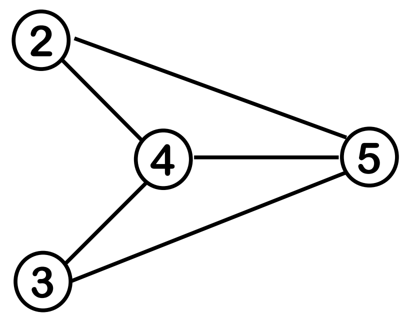




Chaitin's Algorithm in Practice

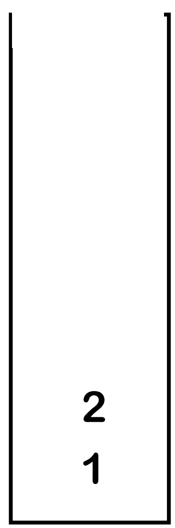
3 Registers



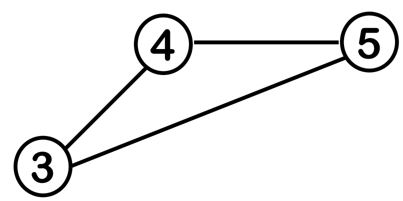


Chaitin's Algorithm in Practice

3 Registers

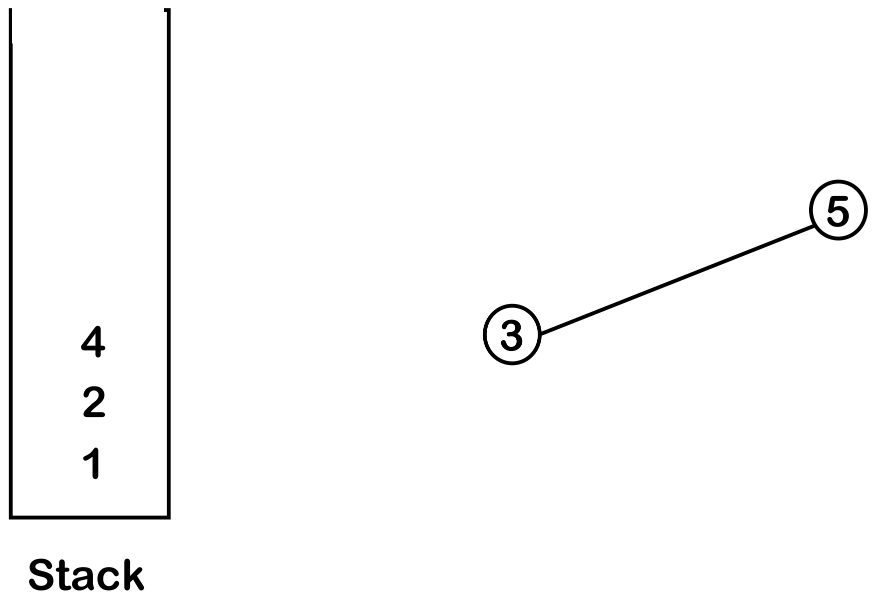


Stack



Chaitin's Algorithm in Practice

3 Registers



Chaitin's Algorithm in Practice

3 Registers

5
3
4
2
1

Stack

Colors:

1:

2:

3:

Chaitin's Algorithm in Practice

3 Registers

3
4
2
1

Stack

5

Colors:

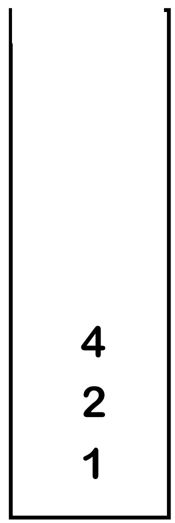
1:

2:

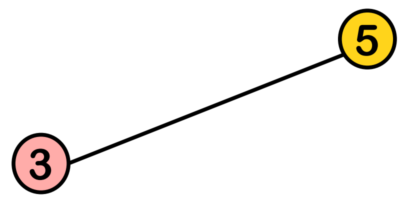
3:

Chaitin's Algorithm in Practice

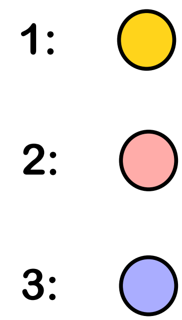
3 Registers



Stack

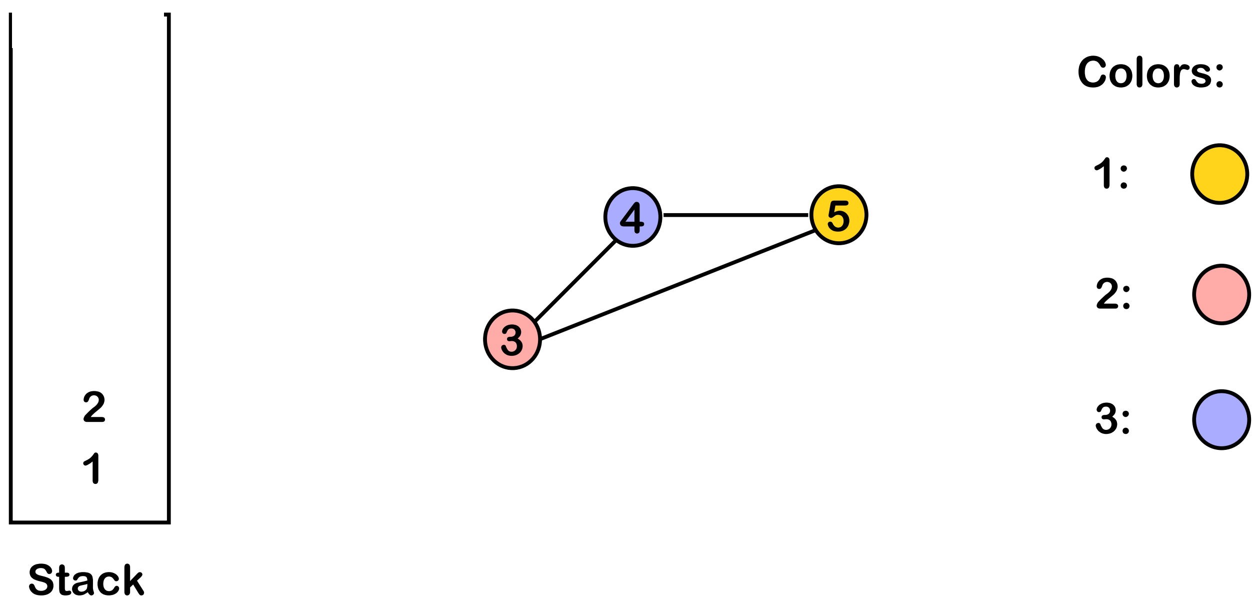


Colors:



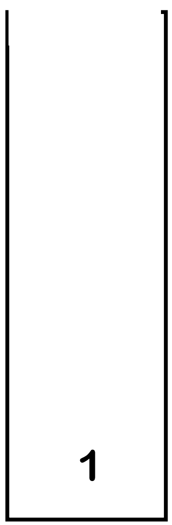
Chaitin's Algorithm in Practice

3 Registers

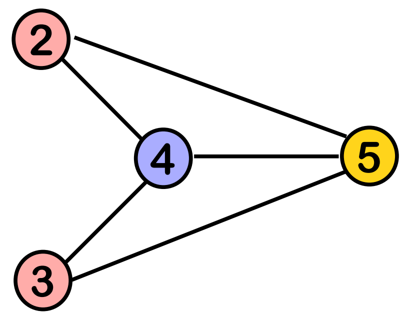


Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

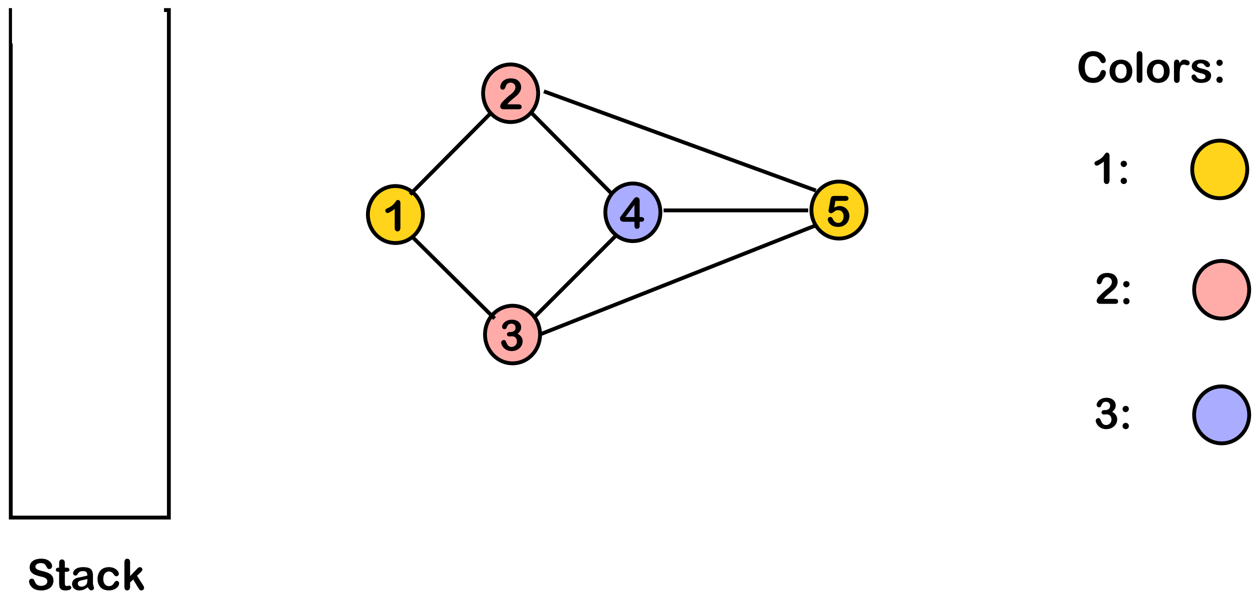
1:

2:

3:

Chaitin's Algorithm in Practice

3 Registers

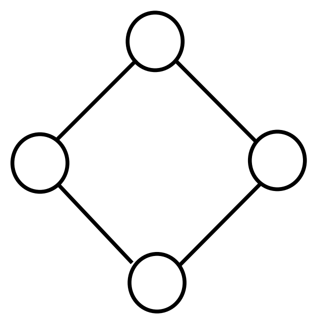


Improvement in Coloring Scheme

Optimistic Coloring (*Briggs, Cooper, Kennedy, and Torczon*)

- If Chaitin's algorithm reaches a state where every node has *k* or more neighbors, it chooses a node to spill.
- Briggs said, take that same node and push it on the stack
  - When you pop it off, a color might be available for it!

2 Registers:



Chaitin's algorithm immediately spills one of these nodes

  - For example, a node *n* might have *k+2* neighbors, but those neighbors might only use 3 (<*k*) colors
    - Degree is a *loose upper bound* on colorability

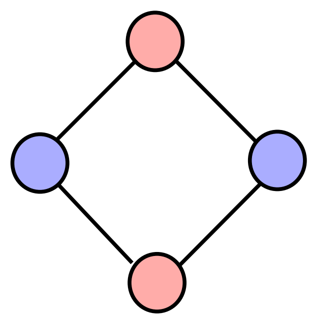
Improvement in Coloring Scheme

Optimistic Coloring (*Briggs, Cooper, Kennedy, and Torczon*)

- If Chaitin's algorithm reaches a state where every node has *k* or more neighbors, it chooses a node to spill.
- Briggs said, take that same node and push it on the stack
  - When you pop it off, a color might be available for it!

2 Registers:

2-colorable



Briggs algorithm finds an available color

  - For example, a node *n* might have *k+2* neighbors, but those neighbors might only use 3 (<*k*) colors
  - Degree is a *loose upper bound* on colorability

Chaitin-Briggs Algorithm

1. While ∃ vertices with < *k* neighbors in G_I
   > Pick any vertex *n* such that $n^\circ < k$ and put it on the stack

   > Remove that vertex and all edges incident to it from G_I

   - This may create vertices with fewer than *k* neighbors

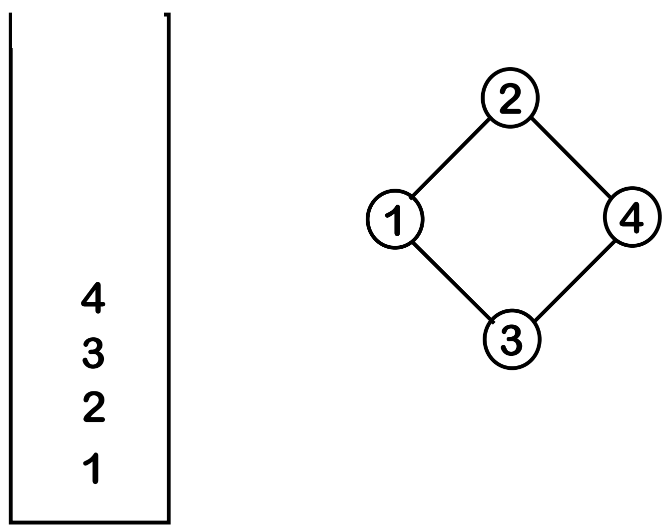
2. If G_I is non-empty (all vertices have *k* or more neighbors) then:
   > Pick a vertex *n* (using some heuristic condition), push *n* on the stack and remove *n* from G_I, along with all edges incident to it

   > If this causes some vertex in G_I to have fewer than *k* neighbors, then go to step 1; otherwise, repeat step 2

3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor
   > If some vertex cannot be colored, then pick an uncolored vertex to spill, spill it, and restart at step 1

Chaitin-Briggs in Practice

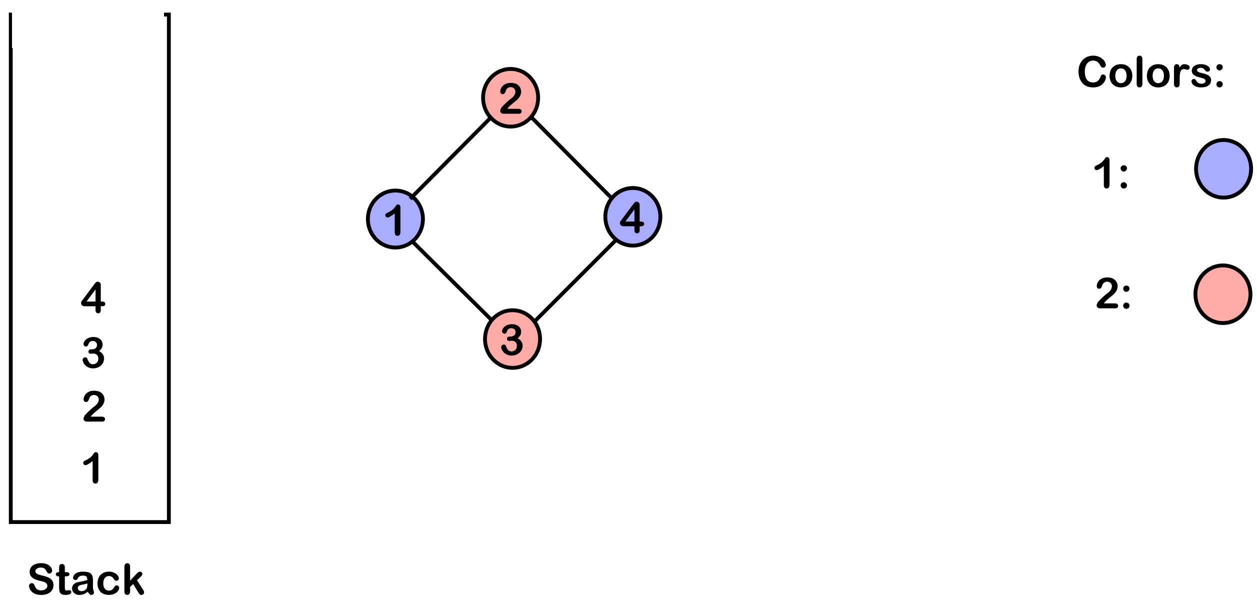
2 Registers



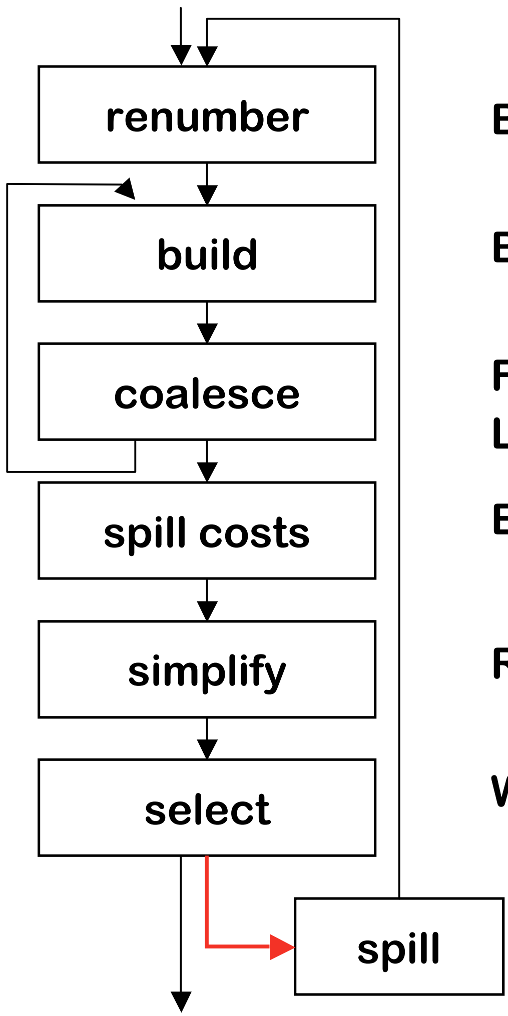
Stack

Chaitin-Briggs in Practice

2 Registers



Chaitin-Briggs Allocator (Bottom-up Coloring)



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies
$LR_x \rightarrow LR_y$, and $< LR_x, LR_y > \notin G_I \Rightarrow$ combine LR_x & LR_y

Estimate cost for spilling each live range

Remove nodes from the graph

while *N* is non-empty
if $\exists$ *n* with $n^\circ < k$ then
push *n* onto stack
else pick *n* to spill
push *n* onto stack
remove *n* from G_I

While stack is non-empty
pop *n*, insert *n* into G_I, & try to color it

Spill uncolored definitions & uses

Briggs' algorithm (1989)

Picking a Spill Candidate

When $\forall\ n \in G_I,\ n^\circ \geq k$, simplify must pick a spill candidate

Chaitin's heuristic

- Minimize spill cost ÷ current degree
- If LR_x has a negative spill cost, spill it pre-emptively
  - Cheaper to spill it than to keep it in a register
- If LR_x has an infinite spill cost, it cannot be spilled
  - No value dies between its definition & its use
  - No more than *k* definitions since last value died (*safety valve*)

Spill cost is weighted cost of loads & stores needed to spill x

Bernstein *et al.* suggest repeating simplify, select, & spill with several different spill choice heuristics & keeping the best

Other Improvements to Chaitin-Briggs

Spilling partial live ranges

- Bergner introduced <u>interference region spilling</u>
- Limits spilling to regions of high demand for registers

Splitting live ranges

- Simple idea — break up one or more live ranges
- Allocator can use different registers for distinct subranges
- Allocator can spill subranges independently (*use 1 spill location*)

Conservative coalescing & Iterative coalescing

- Combining $LR_x \rightarrow LR_y$ to form LR_{xy} may increase register pressure
- Limit coalescing to case where $LR_{xy}^{\circ} < k$
- Iterative form tries to coalesce before spilling

Chaitin-Briggs Allocator (Bottom-up Global)

Strengths & weaknesses

- ↑ Precise interference graph
- ↑ Strong coalescing mechanism
- ↑ Handles register assignment well
- ↑ Runs fairly quickly

- ↓ Known to overspill in tight cases
- ↓ Interference graph has no geography
- ↓ Spills a live range everywhere
- ↓ Long blocks devolve into spilling by use counts

Is improvement still possible ?

- Rising spill costs, aggressive transformations, & long blocks
  ⇒ yes, it is