COMP 512
Rice University
Spring 2015

Global Register Allocation via Graph Coloring

The Chaitin-Briggs Approach

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.

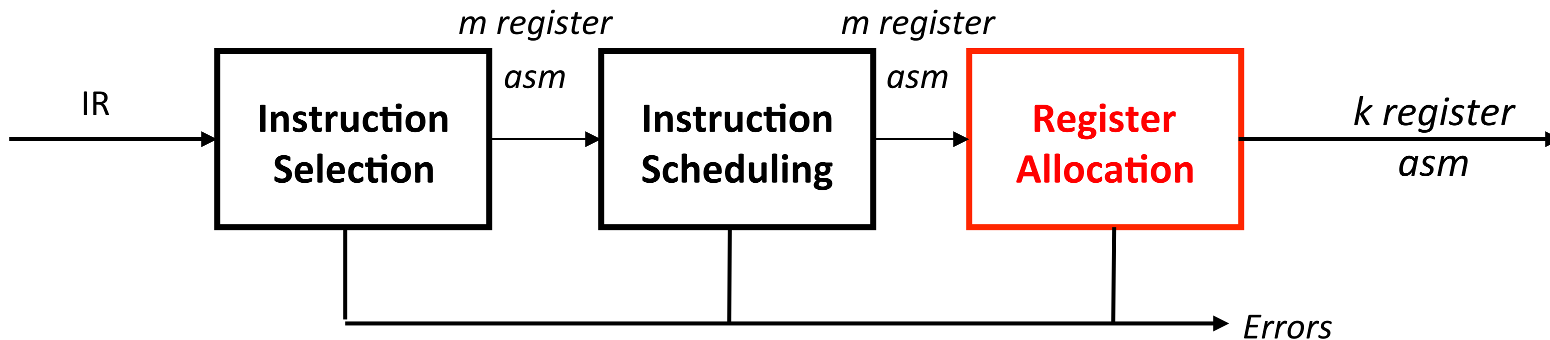
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved

Citation numbers refer to entries in the EaC2e bibliography.

Register Allocation

Part of the compiler's back end



Critical properties

- Produce correct 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 $O(2^n)$

Register Allocation

Early approaches to register allocation

- Frequency counts
  - ♦ FORTRAN I built a CFG and used a Markov Model to assign frequency counts so that it could determine the tradeoffs between different uses of scarce registers.
  - ♦ Freiburghouse formalized the use of frequency counts in a 1974 CACM paper.
- Local allocation
  - ♦ Best devised a near-optimal algorithm for basic blocks in the 1950s (FORTRAN I).
  - ♦ It has been reinvented many times (the *decade* algorithm).
- Loop-based
  - ♦ Several compilers tried local (or regional) allocation in inner loops, moving to outer loops (FORTRAN H, Knobe & Zadeck, Koblenz & Callahan …).
  - ♦ They tried different ideas for that regional allocation

Lavrov (& Ershov) described (& built) coloring storage allocators in the 60's.
Chaitin built the first coloring register allocator in the late 70s/early 80s.

Local Allocation: Best's Algorithm

Best's algorithm takes a simple approach: allocate registers until you run out; when you need more, spill the value used farthest in the future

for i ⇽ 0 to n
    if (OP[i].OP1.PR is invalid)
        get a PR, say x, load OP[i].OP1.VR into x, and set OP[i].OP1.PR ⇽ x
    if (OP[i].OP2.PR is invalid)
        get a PR, say y, load OP[i].OP2.VR into y, and set OP[i].OP2.PR ⇽ y
    if either OP1 or OP2 is a last use
        free the corresponding PR
    Get a PR, say z, and set OP[i].OP3.PR ⇽z

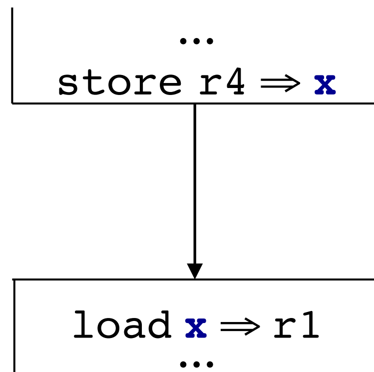
The action "get a **PR**" is the heart of the algorithm.

PR ≅ physical register

- If a **PR** is free, "get a PR" is easy
- If no **PR** is free, choose the occupied **PR** used farthest in the future, spill its contents to memory*, and return that register

* Necessitates a reload on the next use of the spilled value.

What Makes Global Register Allocation Hard?

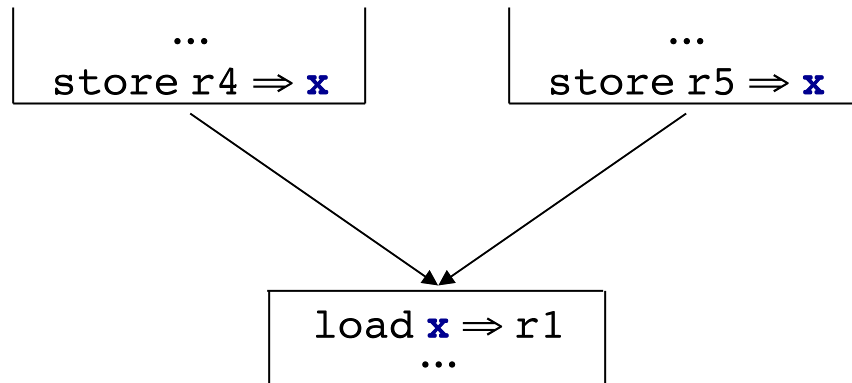


This is an assignment problem, not an allocation problem !

What's harder across multiple blocks?

- Could replace the load with a move (`r4 ⇒ r1`)
- 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

What Makes Global Register Allocation Hard?



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

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
  - ♦ Can break edge to create a place for the copy
  - ♦ Gets into the issue of "critical edges", as in out-of-SSA translation
- In a loop, a block can be its own predecessor

This issue complicates matters

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, where *k* is the number of available registers, or change the code to a nearby problem that it can *k*-color
  → *"change the code" means move some values from registers to memory*

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

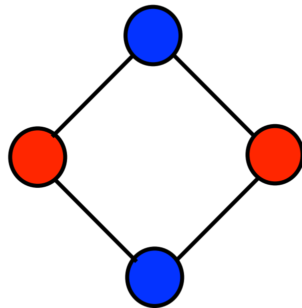
3 Map colors onto physical registers

Graph Coloring (A Background Digression)

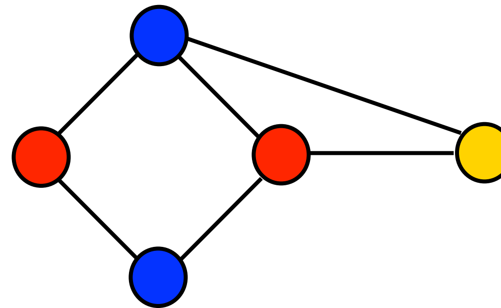
The problem

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

Global Register Allocation

High Level View of Chaitin's Algorithm

1 Build an interference graph G_I for the procedure

- Computing **LIVEOUT** sets, then walk blocks backwards for local **LIVE** information
- Two values that are live at some point p cannot share a register — they *interfere*

2 (try to) construct a *k*-coloring

- For general graphs, finding a minimal coloring is **NP**-Complete
  - → Use an approximate coloring algorithm
- Most interesting programs have a chromatic number > *k*
  - → Allocator must "spill" some values to memory
  - → Spill placement becomes a critical issue

3 Map colors onto physical registers

This description is the roadmap for today's lecture.

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

Interference graph construction relies on **LIVE** information

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

- Construct the **SSA form** of the procedure
- Assign each **SSA** name its own set
- At each ø-function, take the union of the arguments
- Rename to reflect these new "live ranges"

2 Compute **LIVE** sets over live ranges for each block

- Solve equations for **LIVE** over domain of live range names
- Use a simple iterative data-flow solver (*of course*)

3 Iterate over each block, from bottom to top

- Construct LIVENOW at each point in the block, in a backward traversal
- At each operation, add appropriate edges to the graph & update **LIVENOW**
  - → Add an edge from result to each value in **LIVE**
  - → Remove result from **LIVE**
  - → Add each operand to **LIVE**

} Update the **LIVENOW** set

Building the Interference Graph

To build the interference graph

1 Discover live ranges
 > Construct the **SSA form** of the procedure
 > Assign each **SSA** name its own set
 > At each ø-function, take the union of the arguments
 > Rename to reflect these new "live ranges"

2 Compute **LIVE** sets over live ranges for each block
 > Solve equations for **LIVE** over domain of live range names
 > Use a simple iterative data-flow solver (*of course*)

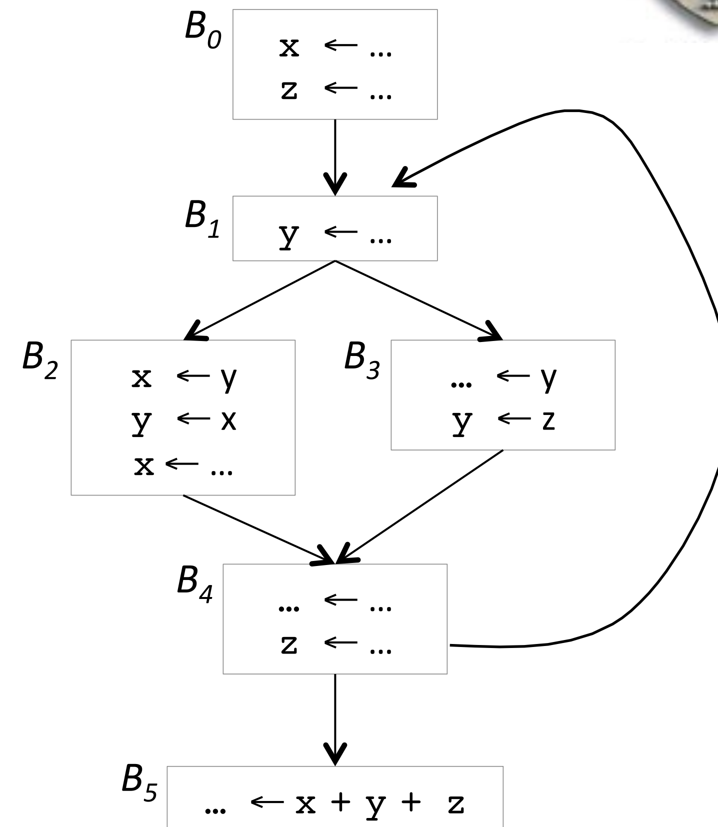
3 Iterate over each block, from bottom to top
 > Construct LIVENOW at each point in the block, in a backward traversal
 > At each operation, add appropriate edges to the graph & update LIVENOW
  → Add an edge from result to each value in **LIVE**
  → Remove result from **LIVE**
  → Add each operand to **LIVE**

} Update the **LIVENOW** set

Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

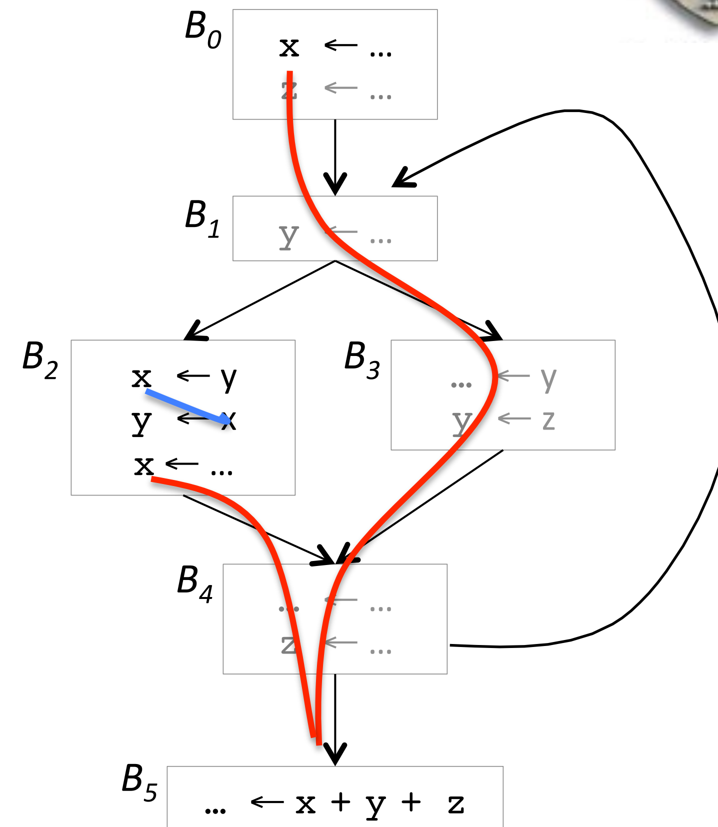
- Consider x, y, & z in the code to the right



Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

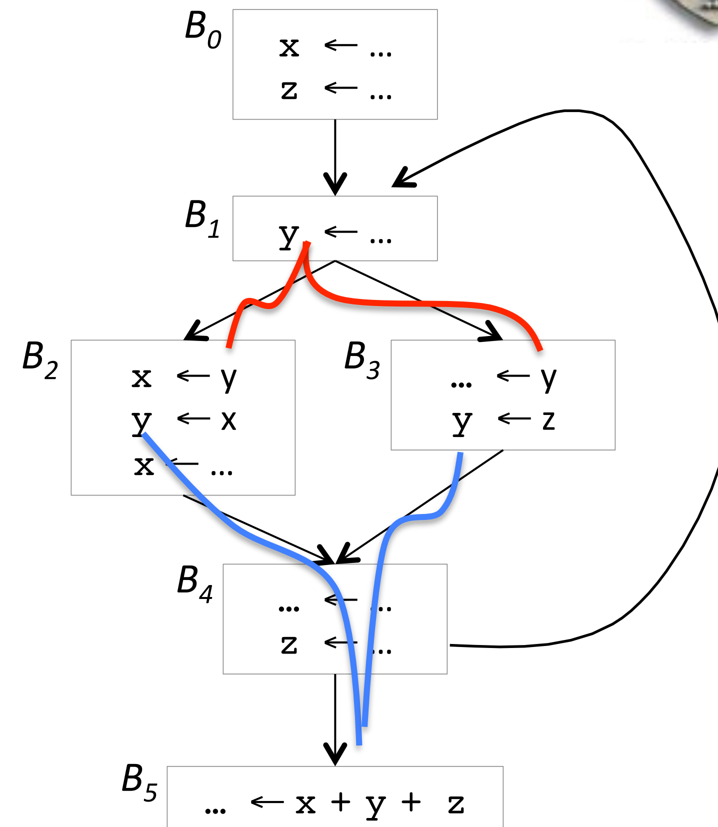
- Consider x, y, & z in the code to the right
  - ♦ x has 2 distinct live ranges



Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

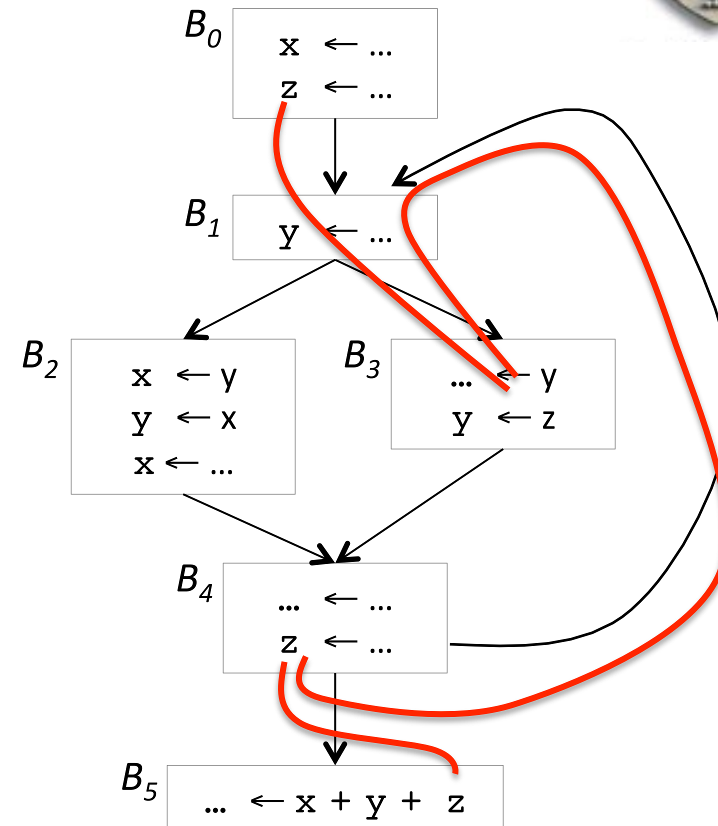
- Consider x, y, & z in the code to the right
  - ♦ x has 2 distinct live ranges
  - ♦ y has 2 distinct live ranges



Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

- Consider x, y, & z in the code to the right
  - ♦ x has 2 distinct live ranges
  - ♦ y has 2 distinct live ranges
  - ♦ z has just 1 live range
    - → z is never live in B_2
- Finding live ranges takes some work



Finding Live Ranges

We can use SSA form to find live ranges in a simple way

1. Build static single assignment form (**SSA** form)
2. Consider each **SSA** name a set
3. At each phi-function, union together the sets of the phi-function arguments
4. Each remaining set is a live range
5. Rename into live ranges

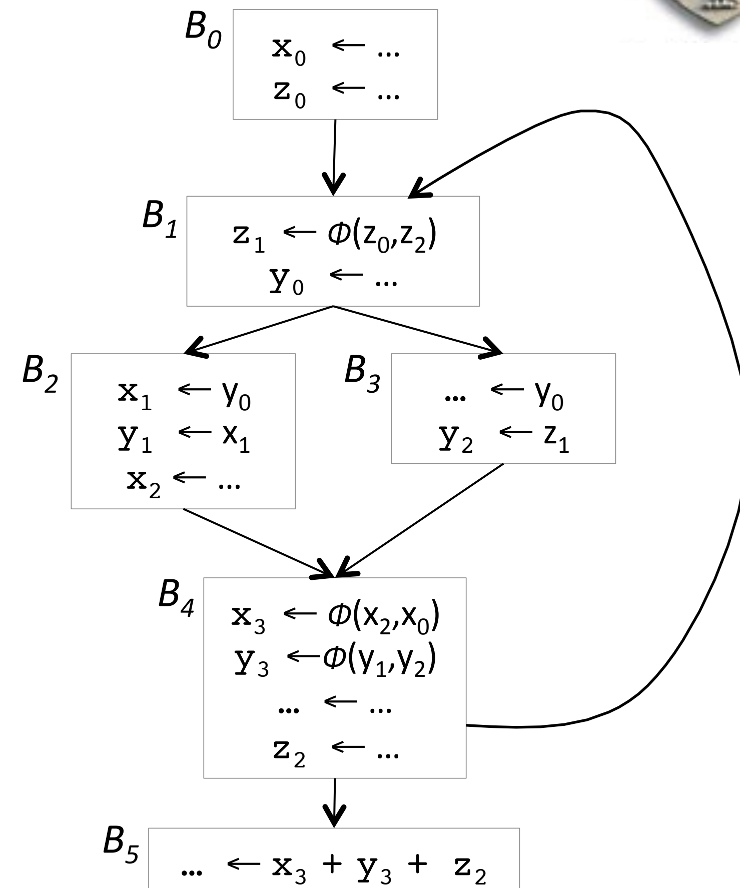
This idea was undoubtedly discovered by multiple different groups of people. I first saw this idea in a meeting with Briggs, Hopkins, Chaitin, Torczon, and a couple of others from IBM and Rice.

Live Ranges

Example in (Pruned) SSA Form

- Each name is defined in exactly one operation
- Each use refers to one name
- Live ranges are
  - ♦ (x_0,x_2,x3) and (x_1)
  - ♦ (y_0) and (y_1,y_2,y_3)
  - ♦ (z_0,z_1,z_2)

  as predicted several slides ago

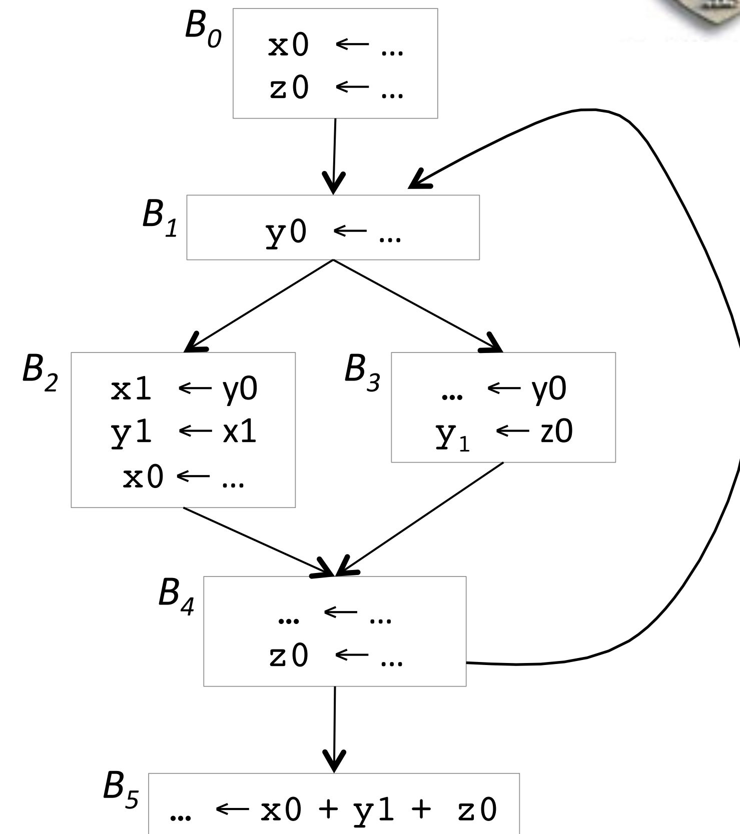


Live Ranges

Rename Into Live Ranges

- Go back to original (non-**SSA** code) & name each live range
- Live ranges are
  - (x_0,x_2,x3) and (x_1)
  - (y_0) and (y_1,y_2,y_3)
  - (z_0,z_1,z_2)

  as predicted several slides ago



Note: a copy operation, such as x ← y does not create an interference between x and y because they can, from that operation's perspective, occupy the same register.

Building the Interference Graph

To build the interference graph

1 Discover live ranges
  - Construct the **SSA form** of the procedure
  - Assign each **SSA** name its own set
  - At each ø-function, take the union of the arguments
  - Rename to reflect these new "live ranges"

2 Compute **LIVE** sets over live ranges for each block
  - Solve equations for **LIVE** over domain of live range names
  - Use a simple iterative data-flow solver (*of course*)

3 Iterate over each block, from bottom to top
  - Construct LIVENOW at each point in the block, in a backward traversal
  - At each operation, add appropriate edges to the graph & update LIVENOW
    - → Add an edge from result to each value in **LIVE**
    - → Remove result from **LIVE**
    - → Add each operand to **LIVE**

(The last two items are bracketed together: Update the **LIVENOW** set)

In 512, you have seen enough DFA that I will skip the details.

Building the Interference Graph

To build the interference graph

1 Discover live ranges
   > Construct the **SSA form** of the procedure
   > Assign each **SSA** name its own set
   > At each ø-function, take the union of the arguments
   > Rename to reflect these new "live ranges"

2 Compute **LIVE** sets over live ranges for each block
   > Solve equations for **LIVE** over domain of live range names
   > Use a simple iterative data-flow solver (*of course*)

3 Iterate over each block, from bottom to top
   > Construct LIVENOW at each point in the block, in a backward traversal
   > At each operation, add appropriate edges to the graph & update LIVENOW
      → Add an edge from result to each value in **LIVE**
      → Remove result from **LIVE**
      → Add each operand to **LIVE**

} Update the **LIVENOW** set

In 512, you have seen enough DFA that I will skip the details.

How do we color a graph: Chaitin's approximation

- Suppose you have *k* registers—the allocator should look for a *k* coloring
- Any vertex *n* that has fewer than *k* neighbors in the interference graph can ***always*** be colored ! We denote this as $n^\circ < k$.
  - ♦ Pick any color not used by its neighbors — there ***must*** be one
- Chaitin's algorithm computes an order in which the graph can be colored, then uses that order to assign colors to individual node
- 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 (*Smallest-last coloring*)

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

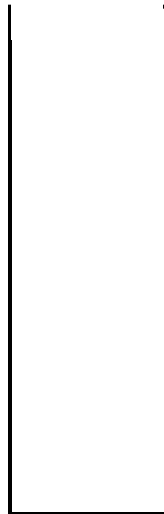
Lowers degree of n's neighbors

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 "spill list"
   > If this causes some vertex in G_I to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2
3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate, again
4. Otherwise, successively pop vertices off the stack and color them in the lowest color not used by some neighbor

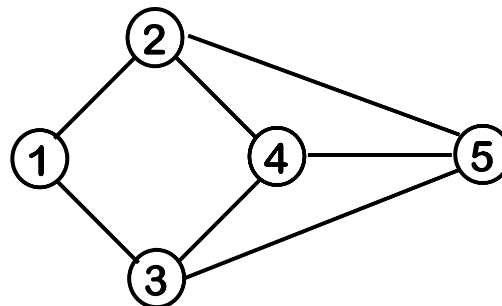
While this algorithm is colloquially referred to as Chaitin's algorithm, the first paper lists Chaitin, Auslander, Chandra, Cocke, Hopkins, and P. Markstein as authors. The second paper, which contains the current spill algorithm, is a single author paper by Chaitin.

Chaitin's Algorithm in Practice

3 Registers



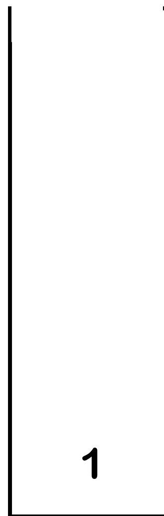
Stack



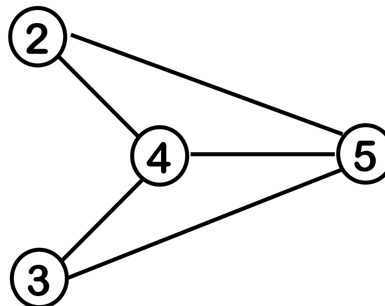
1 is the only node with degree < 3

Chaitin's Algorithm in Practice

3 Registers



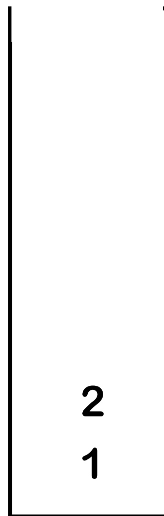
Stack

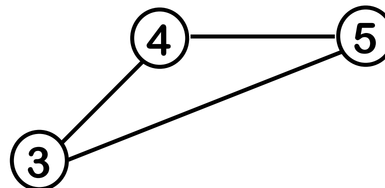


Now, 2 & 3 have degree < 3

Chaitin's Algorithm in Practice

3 Registers

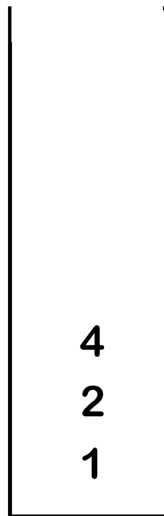


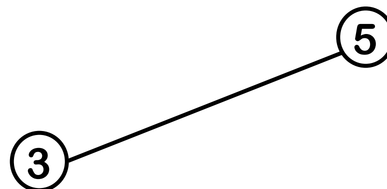


Now all nodes have degree < 3

Chaitin's Algorithm in Practice

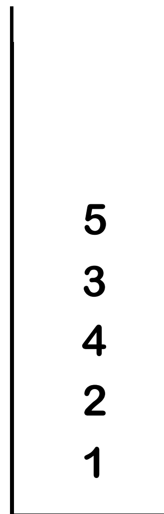
3 Registers





Chaitin's Algorithm in Practice

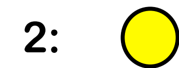
3 Registers



Stack

Colors:

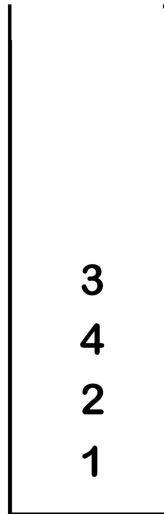
1:

2: 

3:

Chaitin’s Algorithm in Practice

3 Registers



Stack

5

Colors:

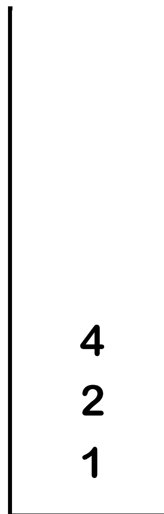
1:

2:

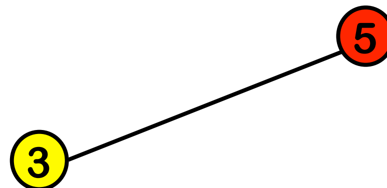
3:

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

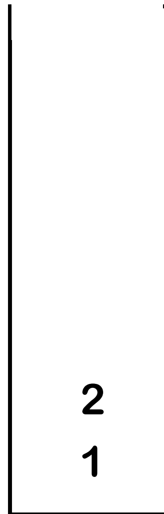
1:

2:

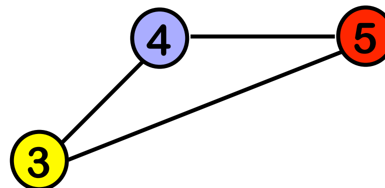
3:

Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

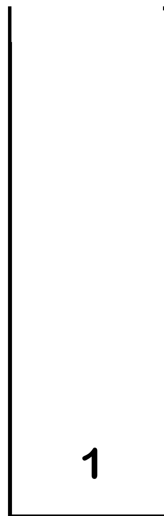
1:

2:

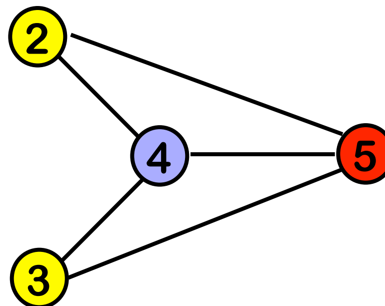
3:

Chaitin's Algorithm in Practice

3 Registers



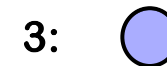
Stack



Colors:

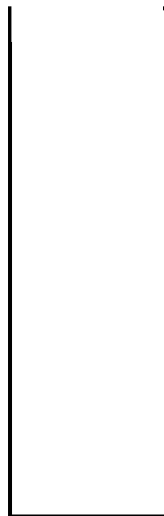
1:

2:

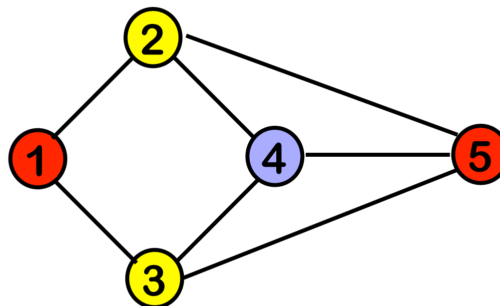
3: 

Chaitin's Algorithm in Practice

3 Registers

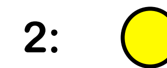


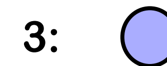
Stack



Colors:

1:

2: 

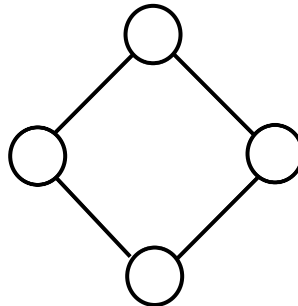
3: 

Improvement in Coloring Scheme

Optimistic Coloring

- 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

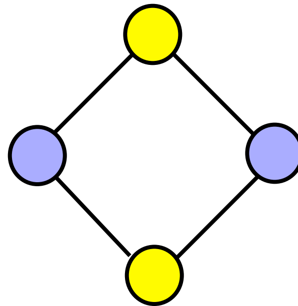
Improvement in Coloring Scheme

Optimistic Coloring

- 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

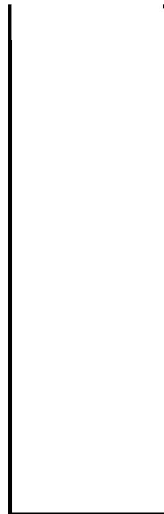
- ♦ For example, a node *n* might have *k+2* neighbors, but those neighbors might only use just one color (*or any number < k*)
  - → ***Degree is a <u>loose upper bound</u> on colorabilit***y

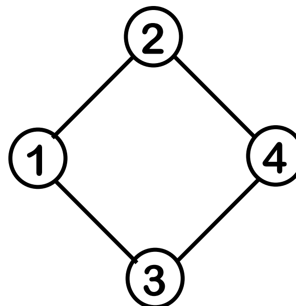
Chaitin-Briggs Algorithm

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 action often creates 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



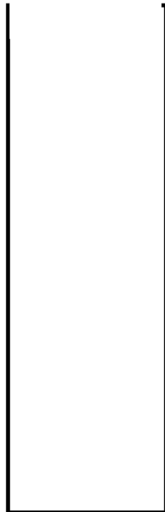


Stack

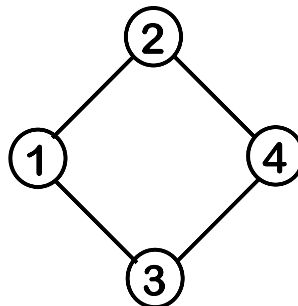
No node has degree < 2
- Chaitin would spill a node
- Briggs picks the same node & stacks it

Chaitin-Briggs in Practice

2 Registers



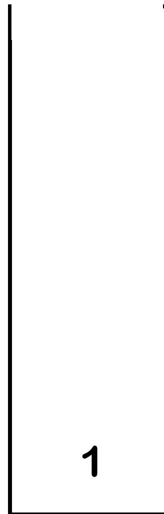
Stack

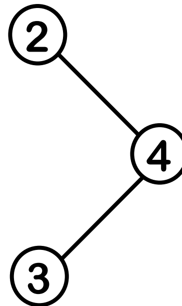


Pick a node, say 1

Chaitin-Briggs in Practice

2 Registers

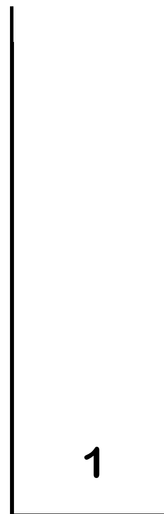


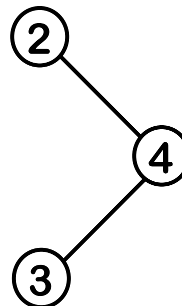


Pick a node, say 1

Chaitin-Briggs in Practice

2 Registers

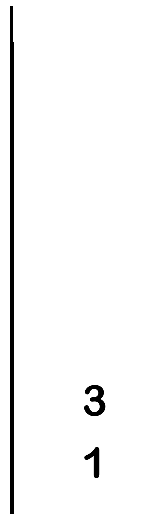


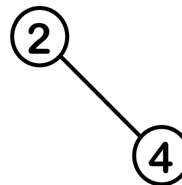


Now, both 2 & 3 have degree < 2
Pick one, say 3

Chaitin-Briggs in Practice

2 Registers

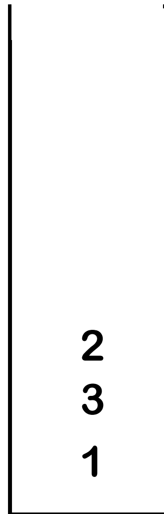




Both 2 & 4 have degree < 2.
Take them in order 2, then 4.

Chaitin-Briggs in Practice

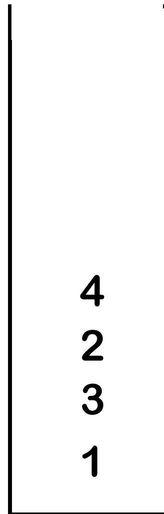
2 Registers



④

Chaitin-Briggs in Practice

2 Registers

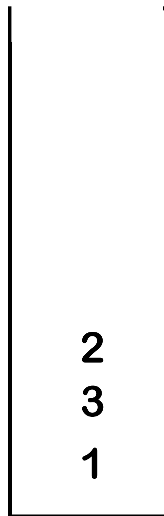


Stack

Now, rebuild the graph

Chaitin-Briggs in Practice

2 Registers

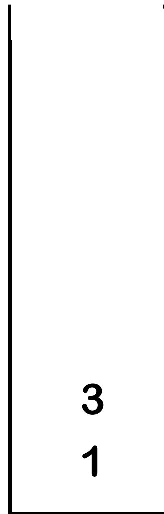


Stack

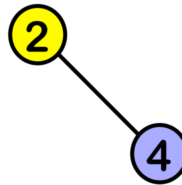
Colors:

Chaitin-Briggs in Practice

2 Registers

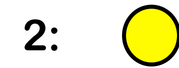


Stack



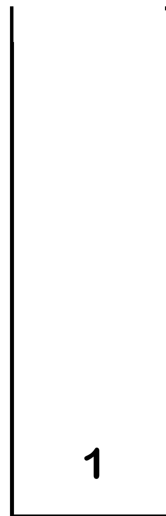
Colors:

1:

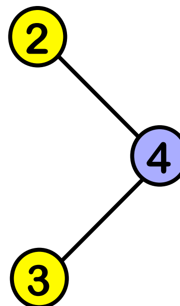
2: 

Chaitin-Briggs in Practice

2 Registers

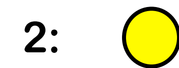


Stack



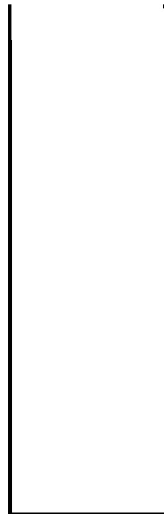
Colors:

1:

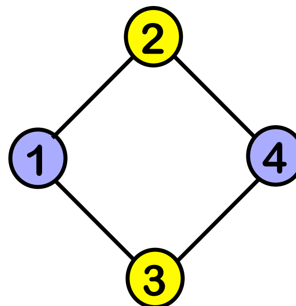
2: 

Chaitin-Briggs in Practice

2 Registers



Stack

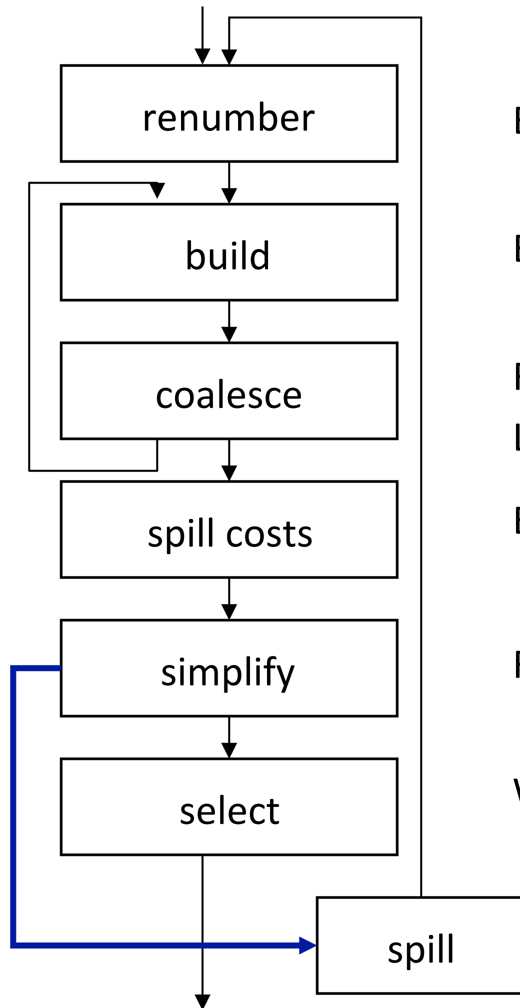


Colors:

1:

2:

Chaitin's Allocator (Bottom-up Coloring, '82 Spill method)



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

$LR_x \rightarrow LR_y$, and $\langle LR_x, LR_y \rangle \notin G_I \Rightarrow$ combine LR_x & LR_y

Estimate cost for spilling
each live range

Remove nodes from the graph

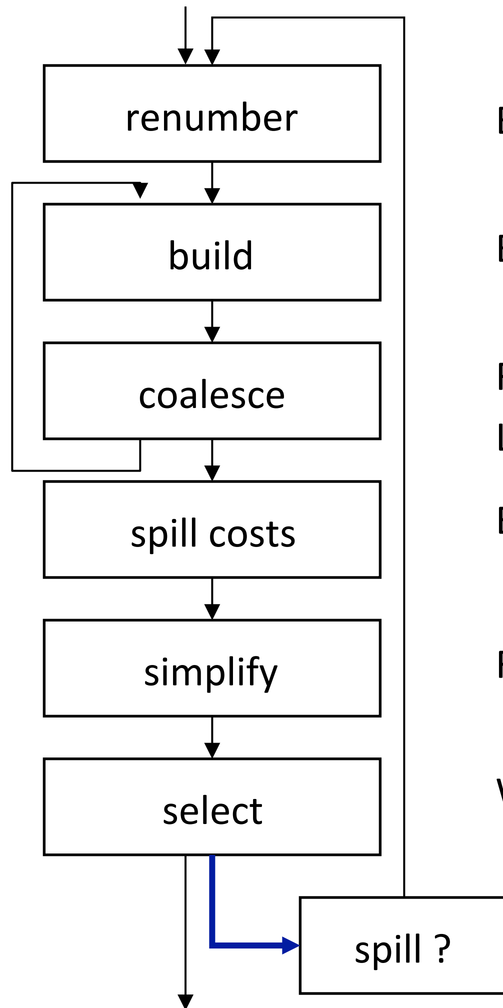
while N is non-empty
if $\exists$ n with $n^\circ < k$ then
push n onto stack
else pick n to spill
mark n for spill pass
remove n from G_I

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

Spill uncolored definitions & uses

Chaitin's algorithm

Chaitin-Briggs Allocator (Optimistic Coloring)



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

$LR_x \rightarrow LR_y$, and $\langle LR_x, LR_y \rangle \notin G_I \Rightarrow$ combine LR_x & LR_y

Estimate cost for spilling
each live range

Remove nodes from the graph

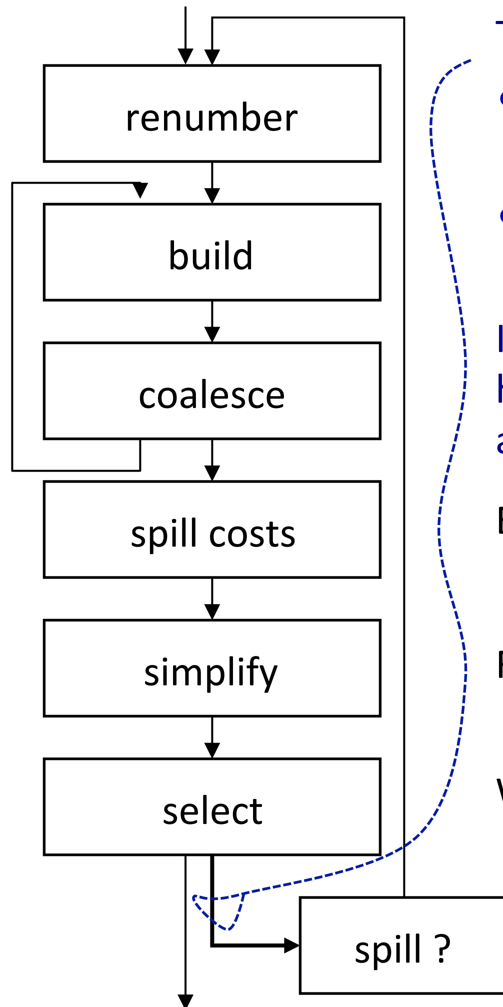
while *N* is non-empty
if ∃ *n* with $n^\circ < k$ then
push *n* onto stack
else pick *n* to spill
push n onto stack
remove *n* from G_I

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

Spill uncolored definitions & uses

Briggs' algorithm

Chaitin-Briggs Allocator (Optimistic Coloring)



This simple change improves the allocation

- x has high º, but x's neighbors use few colors (⇒ colors are available for x)
- Makes the allocation optimistic – assume it will work out and fix things if it does not

It was, in truth, a little hard to explain to a patent examiner how moving one end of the arrow changed the algorithm and the results!

Estimate cost for spilling
each live range

Remove nodes from the graph

while *N* is non-empty
  if ∃ *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

How do these allocators do?

Results are "pretty good"

- Simple procedures allocate without spills
- There is some room for improvement
  - ♦ Long blocks, regions of high pressure
  - ♦ Many implementation issues
- Many people have looked at improving Chaitin-Briggs

Better allocations

- Better coloring
- Softer coalescing
- Better spilling
- Spilling partial live ranges

Better implementations

- Faster graph construction
- Faster coalescing

Different approximate graphs

- Linear Scan allocation
- **SSA**-based allocation