Global Register Allocation via Graph Coloring

Global Register Allocation

The Big Picture

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

At each point in the code

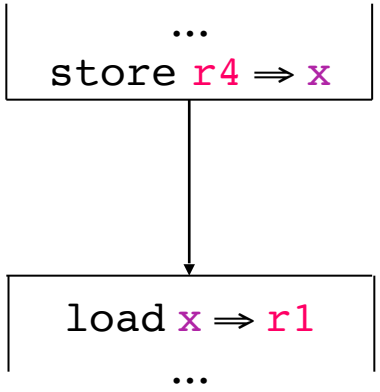
1. Determine which values will reside in registers
2. Select a register for each such value

For local allocation we saw

-the frequency-count allocator (top down)

-the allocator based on distance to the next use (bottom up)

What Makes Global Register Allocation Hard?

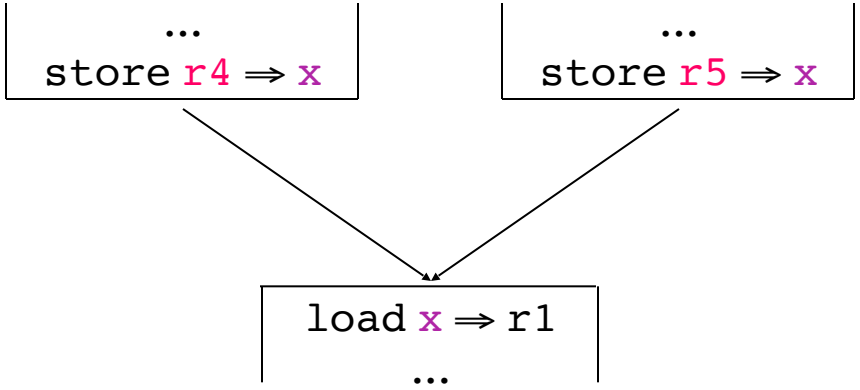


This is an assignment problem, not an allocation problem !
If x is kept in the right register we could just replace store-load with a move

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

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
- 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

Difference between two different allocations for the same code lies

- the number of loads and stores
- the placement operations (different blocks execute different times and this may vary at every run)

Global vs Local Register Allocation

- The structure of global live range can be more complex : a global live range is a web of definitions and uses

- In a local live rage all reference execute once per execution of the block . Thus the cost of spilling is uniform

- In a global allocator the cost of spilling depends on where the spilling occurs
  - Global allocators annotate each reference with an estimated execution frequency derived by static analysis or from profile data

Graph colouring paradigm

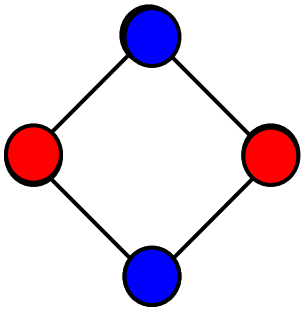
1. Build an interference graph G_I for the procedure
   - Computing LIVE is harder than in the local case
   - G_I is not an interval graph as in the local case
2. (try to) construct a k-coloring
   - Minimal coloring is NP-Complete
   - Spill placement becomes a critical issue
3. Map colors onto physical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

Graph Coloring (A Background Digression)

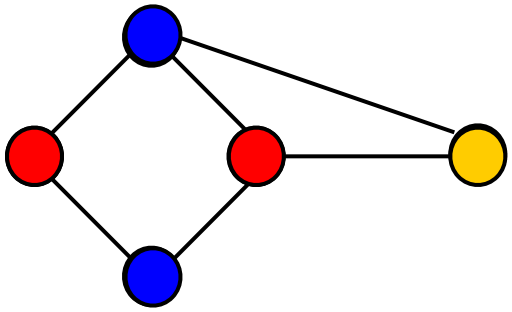
The problem

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

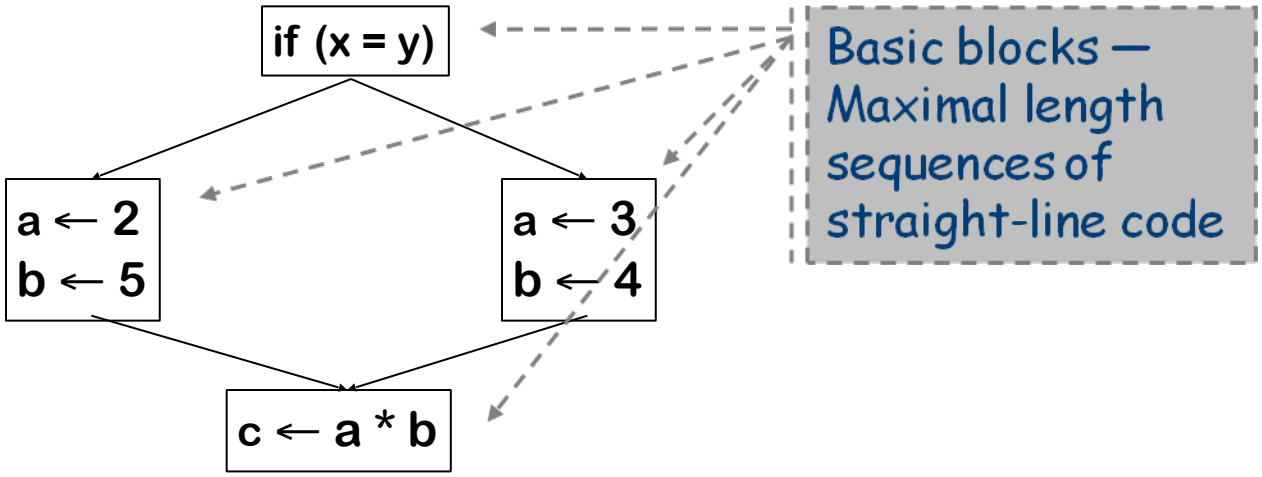
Each color can be mapped to a distinct physical register

Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



Static Single Assignment (SSA) form

- The main idea: each name is defined exactly once
- The name refers to the variable in some program point
- Encodes both control and value flow
- Introduce ϕ-functions to make it work

Original

```
x ← …
y ← …
while (x < k)
   x ← x + 1
   y ← y + x
```

SSA-form

```
      x0 ← …
      y0 ← …
      if (x0 >= k) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < k) goto loop
next:   …
```

Strengths of SSA-form

- each use refers to a single definition
- (sometimes) faster algorithms

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

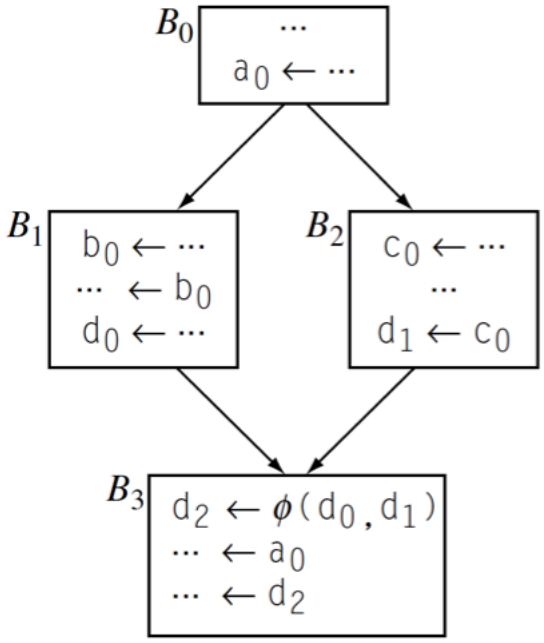
To build the interference graph

1. Discover live ranges
   - Construct the SSA form of the procedure
   - 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, bottom-up
   - Track the current LIVE set
   - At each operation, add appropriate edges & update LIVE

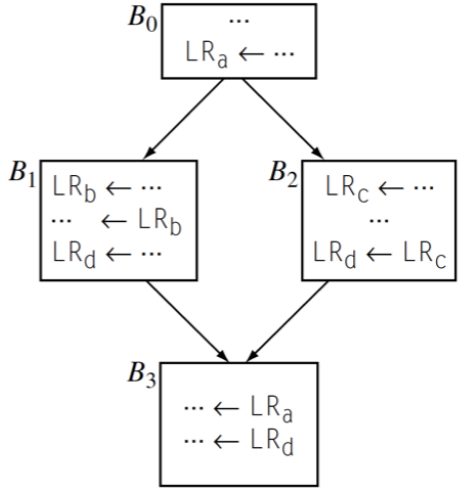
Point 1

Discover live ranges

- Construct the SSA form of the procedure
- At each ϕ-function, take the union of the arguments
- Rename to reflect these new "live ranges": arguments of the same phi functions has to be united together



(a) Code Fragment in SSA Form



(b) Rewritten in Terms of Live Ranges

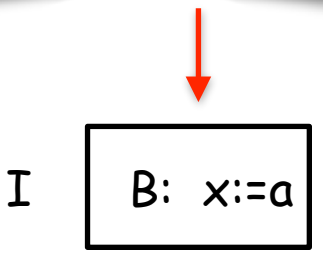
Live ranges {LRa=a0,LRb=b0,LRc=c0, LRd=d0 ⋃d1⋃d2 }

Point 2: Computing LIVE Sets

A value is live between its definition and its uses

- Find definitions ($x \leftarrow \ldots$) and uses ($y \leftarrow \ldots x \ldots$)
- From definition to last use is its live range
  - How does a second definition affect this?
- Can represent live range as an interval [i,j] (in block)

$LV_o(l)$ are the variables live right before the block I



$LV_\bullet(l)$ are the variables live at the exit of the block I

- $LV_\bullet(l) = \bigcup \{LV_o(l') \mid l' \text{ in } post(l)\}$

- $LV_o(l) = (LV_\bullet(l) \setminus def(B)) \cup use(B)$

$def([x:=a]^l) = \{x\}$ and $\emptyset$ elsewhere

$use([x:=a]^l) = FV(a)$

$use([b]^l) = FV(b)$ and $\emptyset$ elsewhere

Solve the equations using a fixed-point iterative scheme

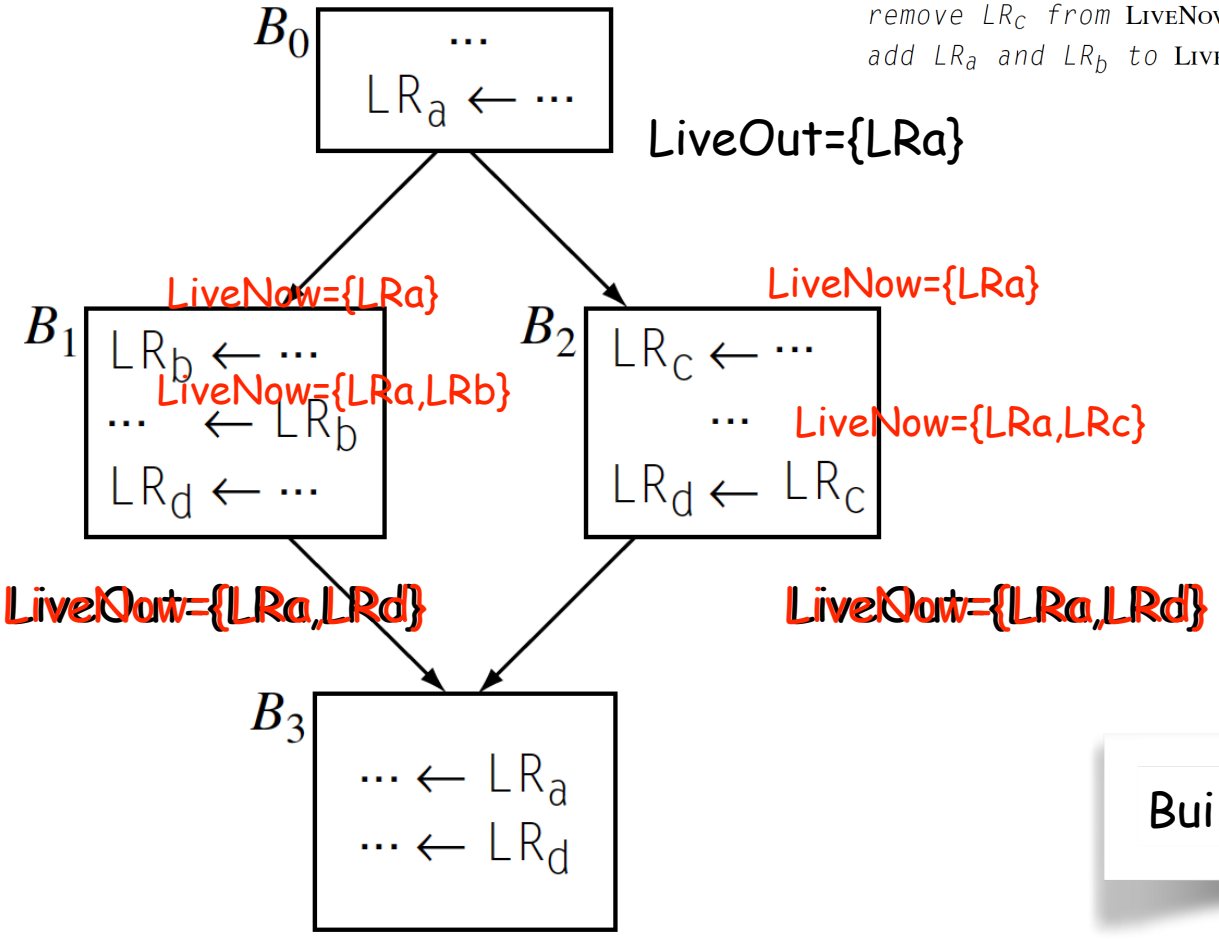
Point 3:constructing the Interference graph

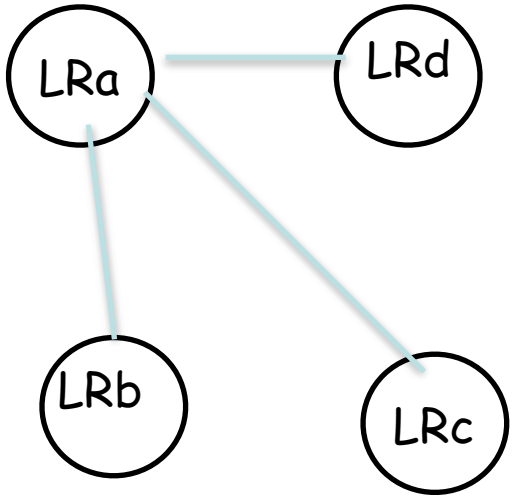
- LRj interferences with LRi if one is live at a definition of the other

- Once the allocator has built global variable ranges and annotate each basic block with its LiveOut set ($LV_\bullet(B)$), it can construct Interference graph with a linear pass over each block.

```
for each LR_i
   create a node n_i ∈ N
for each basic block b
   LIVENOW ← LIVEOUT(b)
   for each operation op_n, op_{n-1}, op_{n-2}, ...op_1 in b
      with form op_i LR_a , LR_b ⇒ LR_c
      for each LR_i ∈ LIVENOW
         add (LR_c , LR_i ) to E
      remove LR_c from LIVENOW
      add LR_a and LR_b to LIVENOW
```

```
for each LR_i
    create a node n_i ∈ N
for each basic block b
    LIVENOW ← LIVEOUT(b)
    for each operation op_n, op_n-1, op_n-2, ...op_1 in b
        with form op_i LR_a , LR_b ⇒ LR_c
        for each LR_i ∈ LIVENOW
            add (LR_c , LR_i ) to E
        remove LR_c from LIVENOW
        add LR_a and LR_b to LIVENOW
```





Building interference Graph!

Account for Execution Frequency

The compiler annotate each block with estimated execution counts

These informations can be derived from

- profile data or from heuristics
- fixed assumptions, for example, a loop executes 10 time,
  an unpredictable if-then-else divides by 2

To estimate the cost of spilling a single reference the allocator adds the cost of the address and memory operation and multiply by frequencies

For each live range it sum up the cost of individual references

Building an allocator

To build an allocator based on graph coloring on the interference graph, the compiler writer needs two additional mechanisms:

- the allocator needs an efficient technique to discover a k-coloring (remember finding is NP-complete)
  - Register allocator uses fast approximations that are not guaranteed to find a k-coloring
- The allocator needs a strategy that handles the case no color remain for a specific live range
  - The allocator chooses one or more live range to spill and reconsider the problem.

Now the interference graph may be colorable!

Observation on Coloring for Register Allocation

- Suppose you have k registers (not all the physical ones: some dedicated to keep, e.g. base addresses)—look for a k coloring
- Any vertex n that has fewer than k neighbours in the interference graph ($n^\circ < k$) can always be colored!
  - Pick any color not used by its neighbours — there must be one

Top-Down Coloring

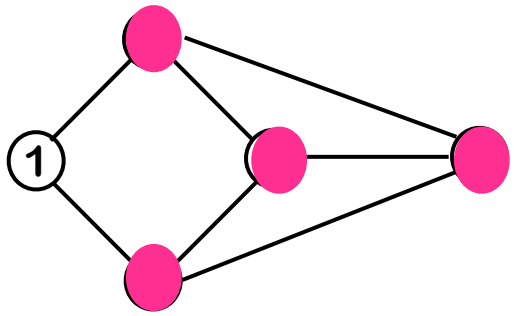
The Big Picture

- Use high-level priorities to rank live ranges
- Allocate registers for them in priority order
- Use coloring to assign specific registers to live ranges

Improving the Top-Down algorithm

1. Rank all live ranges with their estimated runtime saving, (analogous of the spilling cost)

2. Separate constrained from unconstrained live ranges
   - A live range is constrained if it has ≥ k neighbours in G_I



k=4

3. Constrained live ranges are coloured first in rank order

Handling Spills

- When the top down allocator encounters a live range that cannot be coloured it spills the live range to change the problem.
- Since the all previously coloured live ranges were ranked higher than the uncoloured one, it spills the uncoloured one.
- It could think of uncolor some of the previous one but it must exercise care to avoid the full cost of backtracking
- After the spilling the problem becomes easier and a new interference graph can be constructed.

Bottom-Up global allocator

- Ideas behind Chaitin's algorithm:
  - Pick any vertex n such that n°< 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 neighbours
  - At the end, if some vertex n still has k or more neighbours, 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 neighbour

Chaitin's Algorithm

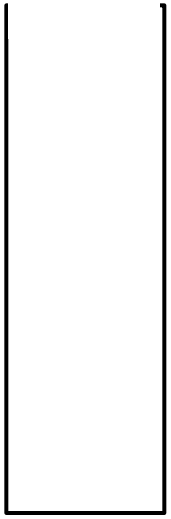
1. While ∃ vertices with < k neighbours in G_I
   - Pick any vertex n such that n°< k and put it on the stack
   - Remove that vertex and all edges incident to it from G_I

Lowers degree of n's neighbours

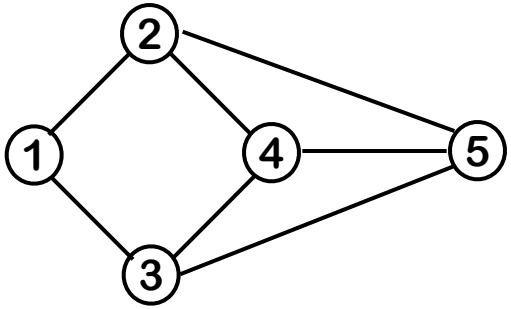
2. If G_I is non-empty (all vertices have k or more neighbours) then:
   - Pick a vertex n (using some heuristic) and spill the live range associated with n
   - Remove vertex n from G_I , along with all edges incident to it and put it on the "spill list"
   - If this causes some vertex in G_I to have fewer than k neighbours, 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 neighbour

Chaitin's Algorithm in Practice

3 Registers



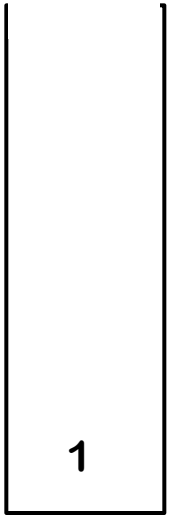
Stack



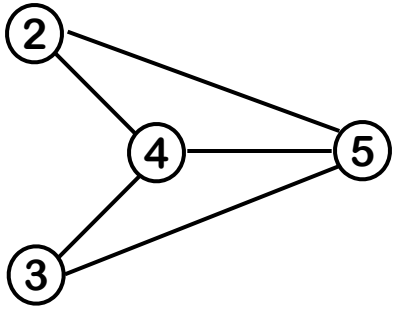
1 is the only node with degree < 3

Chaitin's Algorithm in Practice

3 Registers



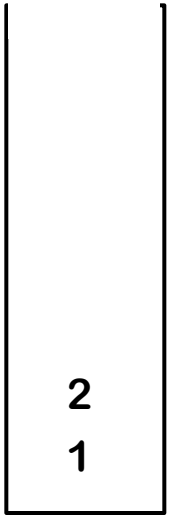
Stack



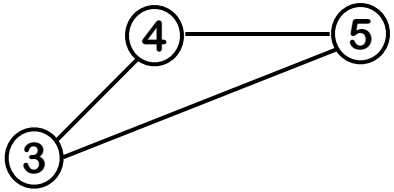
Now, 2 & 3 have degree < 3

Chaitin's Algorithm in Practice

3 Registers



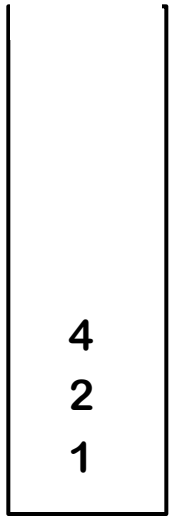
Stack



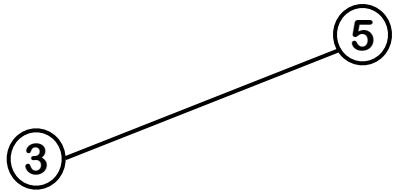
Now all nodes have degree < 3

Chaitin's Algorithm in Practice

3 Registers

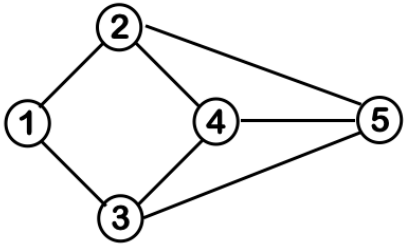


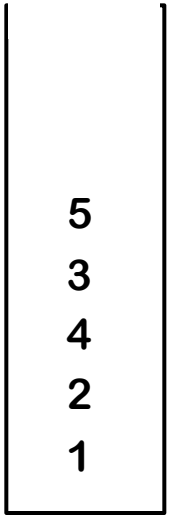
Stack



Chaitin's Algorithm in Practice

3 Registers





Stack

Colors:

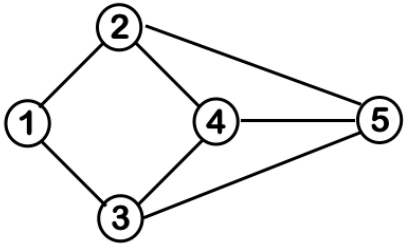
1:

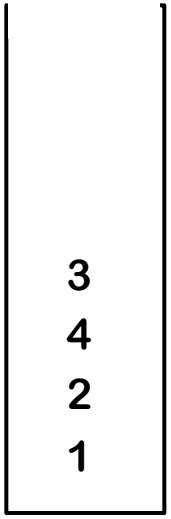
2:

3:

Chaitin's Algorithm in Practice

3 Registers





Stack

Colors:

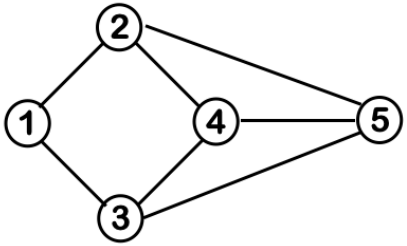
1:

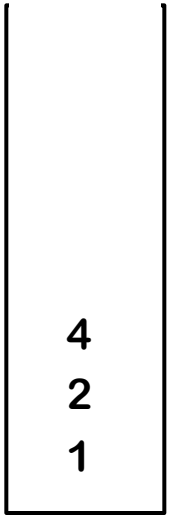
2:

3:

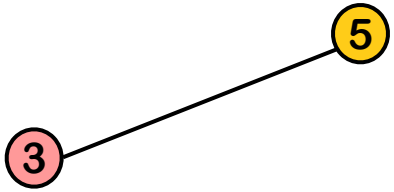
Chaitin's Algorithm in Practice

3 Registers





Stack



Colors:

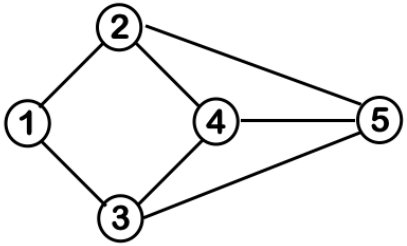
1:

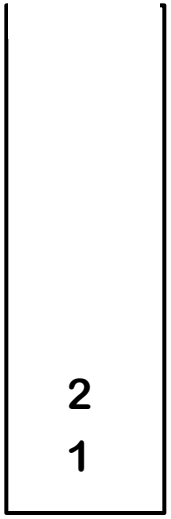
2:

3:

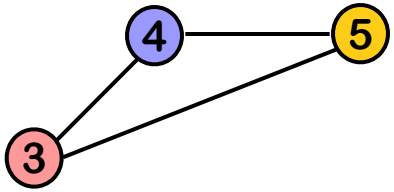
Chaitin's Algorithm in Practice

3 Registers





Stack



Colors:

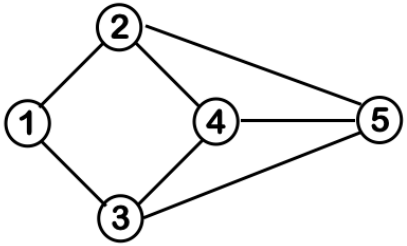
1:

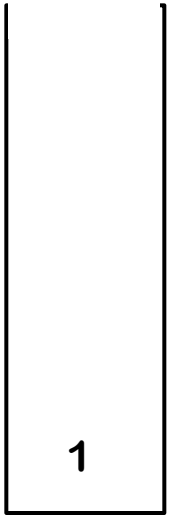
2:

3:

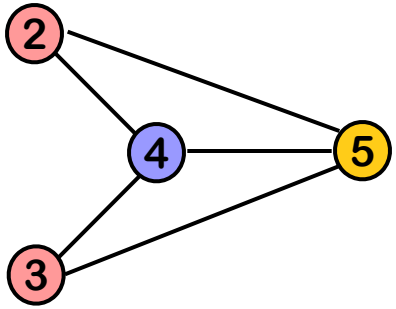
Chaitin's Algorithm in Practice

3 Registers





Stack



Colors:

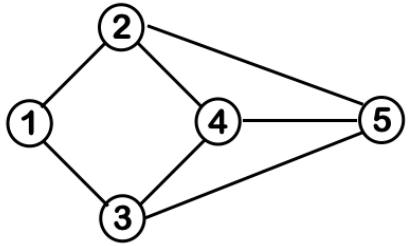
1:

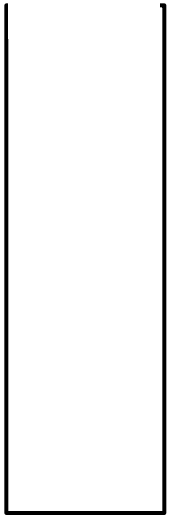
2:

3:

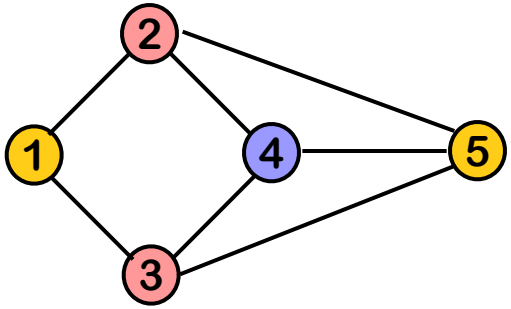
Chaitin's Algorithm in Practice

3 Registers





Stack



Colors:

1:

2:

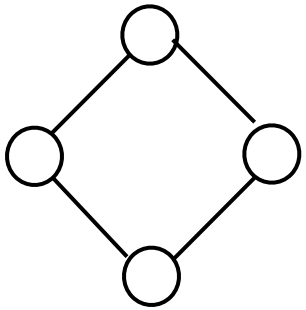
3:

Improvement in Coloring Scheme

Optimistic Coloring

- If Chaitin's algorithm reaches a state where every node has k or more neighbours, 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:

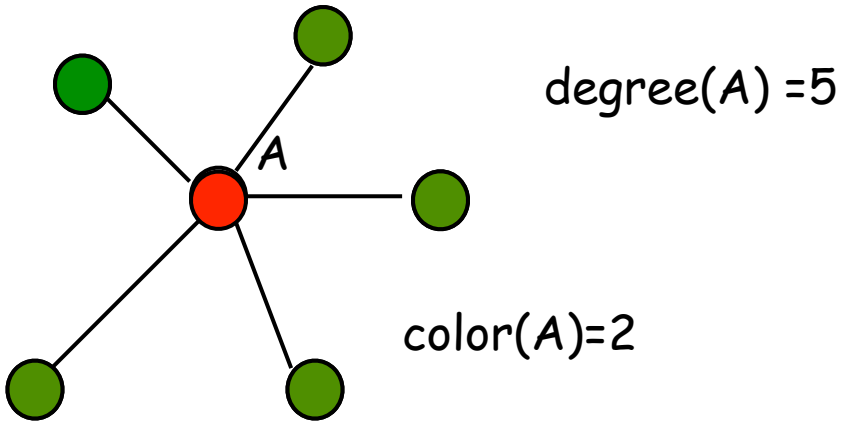


Chaitin's algorithm immediately spills one of these nodes

Improvement in Coloring Scheme

- A node n might have k+2 neighbours, but those neighbours might only use 3 (<k) colors

  → Degree is a <u>loose upper bound</u> on colorability



Briggs et al, PLDI 89 (Also, TOPLAS 1994)

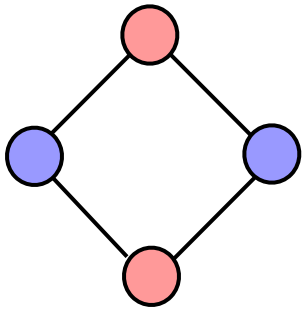
Improvement in Coloring Scheme

Optimistic Coloring

- If Chaitin's algorithm reaches a state where every node has k or more neighbours, 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

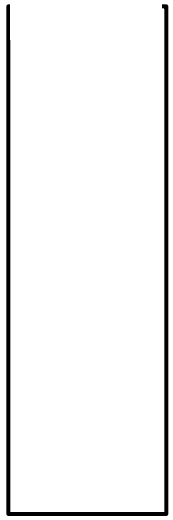
  - For example, a node n might have k+2 neighbours, but those neighbours might only use just one color (or any number < k)
    - → Degree is a loose upper bound on colorability

Chaitin-Briggs Algorithm

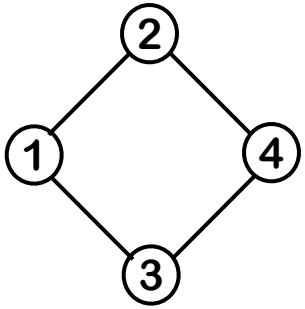
1. While ∃ vertices with < k neighbours in G_I
   - Pick any vertex n such that n°< k and put it on the stack
   - Remove that vertex and all edges incident to it from G_I
     - → This action often creates vertices with fewer than k neighbours
2. If G_I is non-empty (all vertices have k or more neighbours) then:
   - Pick a vertex n (using some heuristic condition, spill metric as cost/degree), 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 neighbours, 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 neighbour
   - 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

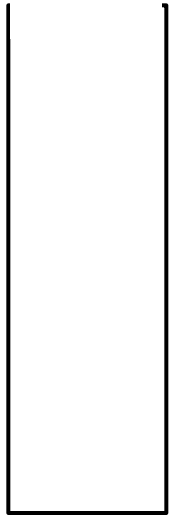


No node has degree < 2

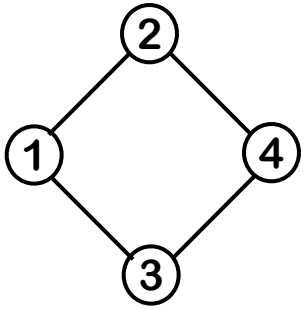
- Chaitin would spill a node
- Briggs picks the same node & stacks it

Chaitin-Briggs in Practice

2 Registers



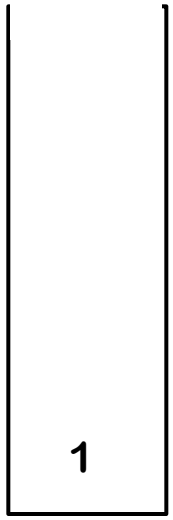
Stack



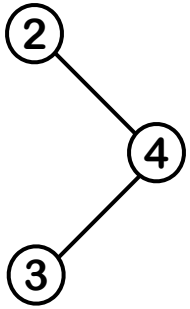
Pick a node, say 1

Chaitin-Briggs in Practice

2 Registers



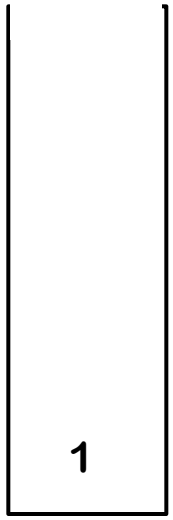
Stack



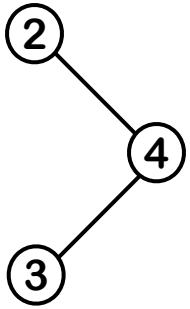
Pick a node, say 1

Chaitin-Briggs in Practice

2 Registers



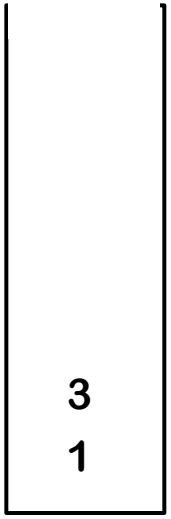
Stack



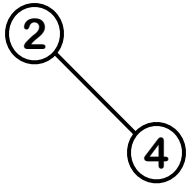
Now, both 2 & 3 have degree < 2
Pick one, say 3

Chaitin-Briggs in Practice

2 Registers



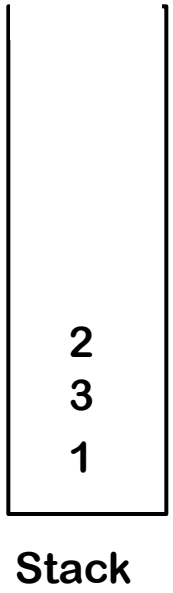
Stack



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

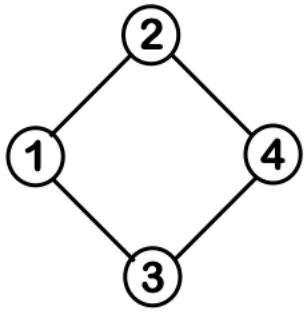
Chaitin-Briggs in Practice

2 Registers

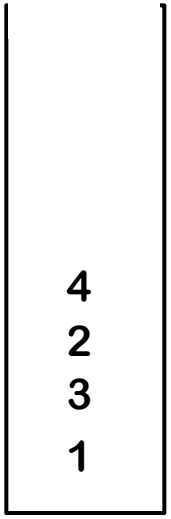


④

Chaitin-Briggs in Practice



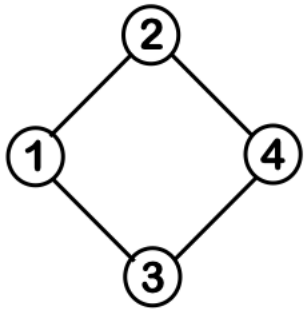
2 Registers



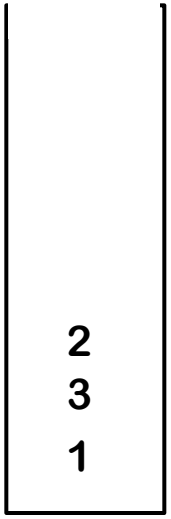
Stack

Now, rebuild the graph

Chaitin-Briggs in Practice



2 Registers



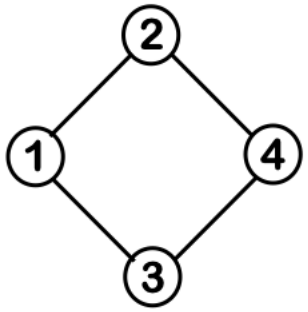
Stack

Colors:

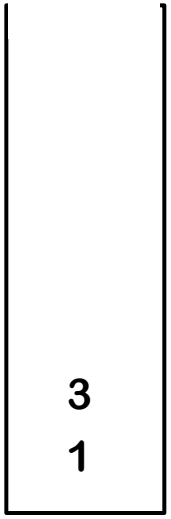
1:

2:

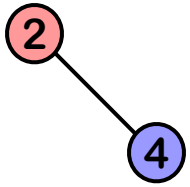
Chaitin-Briggs in Practice



2 Registers



Stack

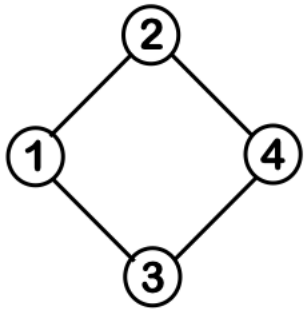


Colors:

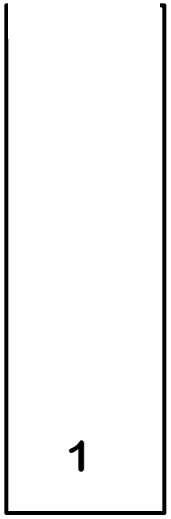
1:

2:

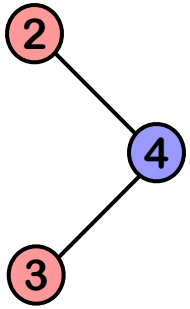
Chaitin-Briggs in Practice



2 Registers



Stack

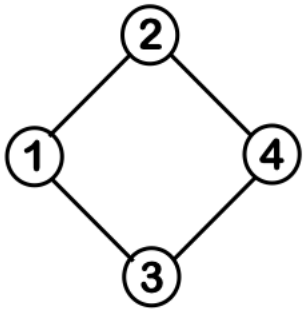


Colors:

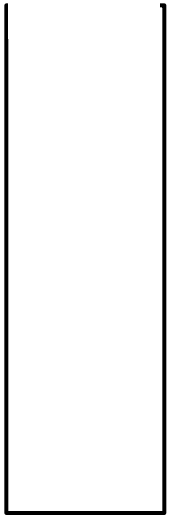
1:

2:

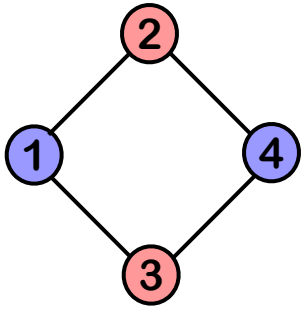
Chaitin-Briggs in Practice



2 Registers



Stack



Colors:

1:

2:

Comparing Top-Down and Bottom-Up approach

- Top-Down constrained nodes first
- Bottom-Up unconstrained nodes first and in this way some constrained becomes unconstrained
- No clear way to compare the results

Advanced Topics in Global Allocation

Coalescing Copies I

- For reduce the degree the compiler writer can use the interference graph to determine when two live ranges that are connected by a copy can be coalescing or combined.

`i2i` $LR_1 \Rightarrow LR_2$ Copy from register to register

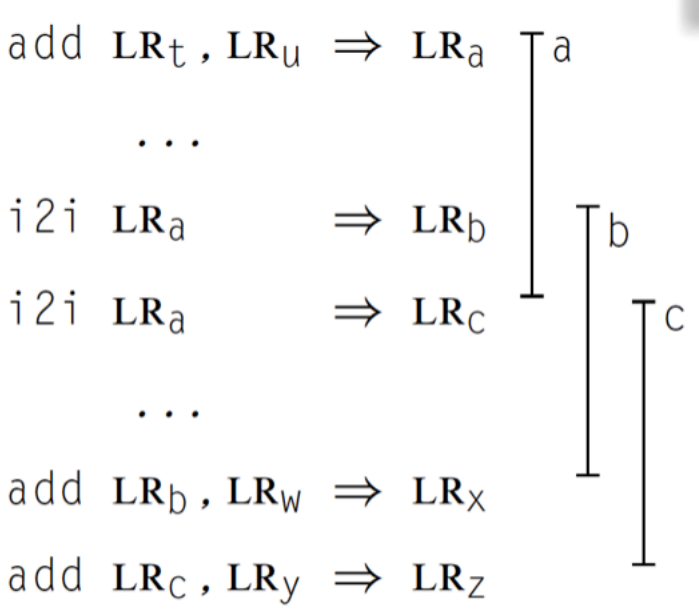
- If LR1 and LR2 do not otherwise interfere, the operation can be eliminated and all references to LR2 can be rewritten to use LR1

Several advantages:

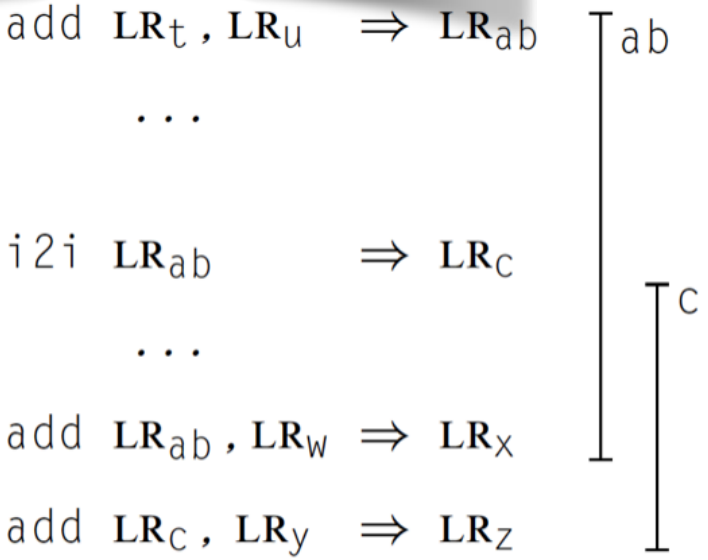
- It eliminates the copy operation
- It reduces the degree of any LR that interfered with both LR1 and LR2

Coalescing Copies II

Both copy operations are candidate for coalescing!



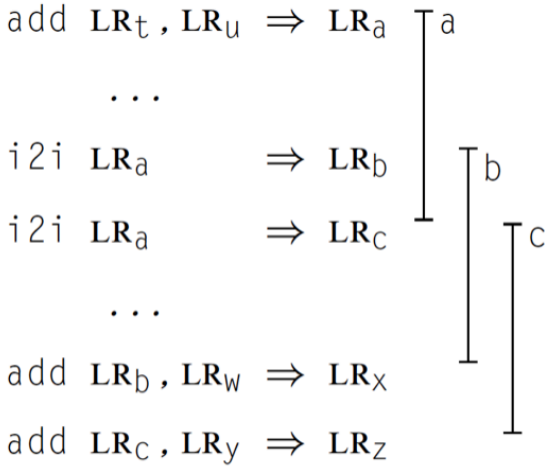
(a) Before Coalescing



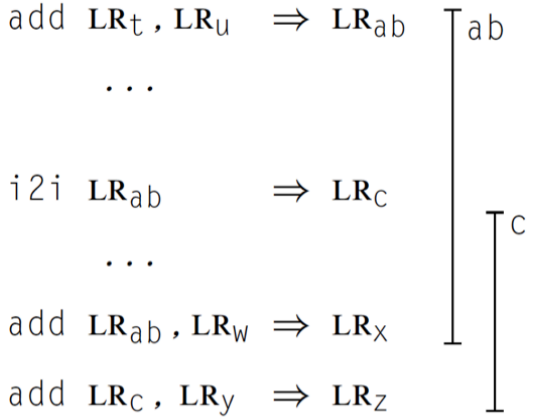
(b) After Coalescing LR_a and LR_b

- Even if LRa overlaps both LRb and LRc, it interferes with neither of them because the source and destination of a copy do not interfere

Coalescing Copies III

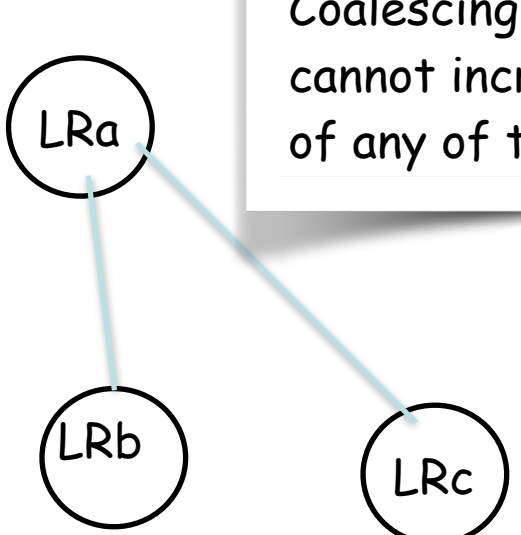


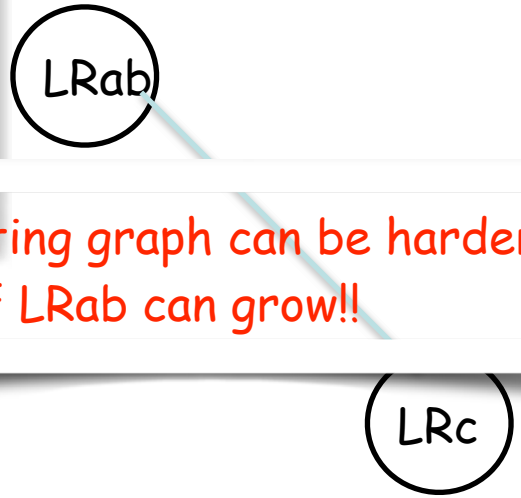
(a) Before Coalescing



(b) After Coalescing LR_a and LR_b

Coalescing two live ranges cannot increase the degrees of any of their neighbours





but the resulting graph can be harder to color, the degree of LRab can grow!!

Safe Coalescing

- To perform coalescing, the allocator walks each block and examines each copy operation in the block

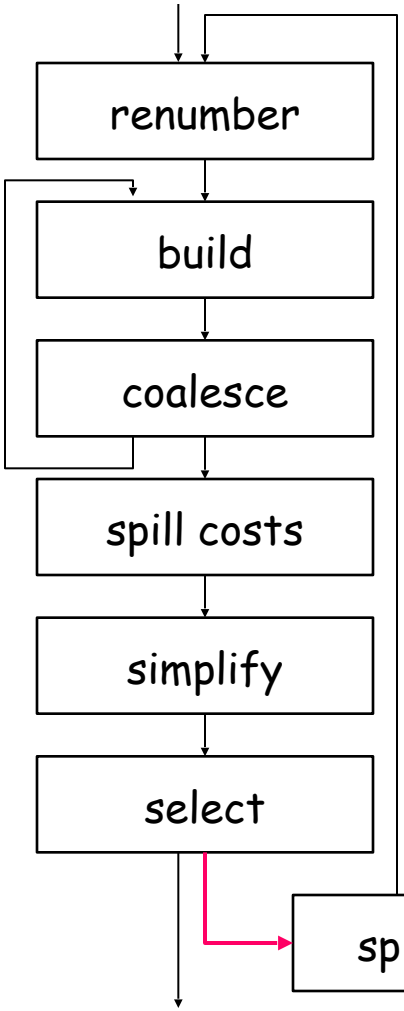
- When it finds `i2i` $LR_1 \Rightarrow LR_2$ with LR1 and LR2 that do not interfere the allocator combines them, eliminates the copy and update the Interference graph

- Coalescing two live range can prevent new coalescing: the order of coalescing matters

Several heuristics have been developed to decide when coalescing is safe, *i.e.* when it is guaranteed that it will not turn a *K*-colourable graph into one that is not *K*-colourable.

Using such heuristics, it is possible to interleave simplification steps with safe coalescing steps, thereby removing many useless move operations.

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 $\langle LR_x, LR_y \rangle \notin G_I \Rightarrow$ 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*°< k then
    push *n* onto stack
  else pick *n to (maybe) spill*
    push *n* onto stack
  remove *n* from $\mathbf{G_I}$

While stack is non-empty

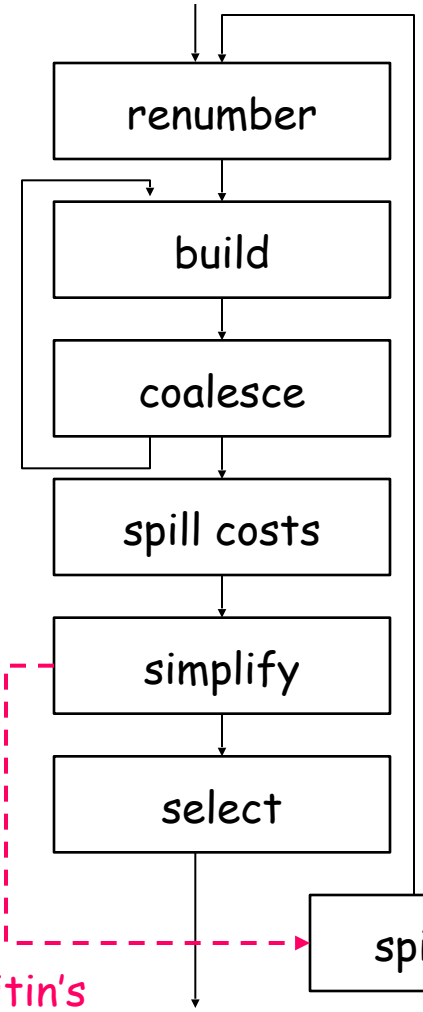
pop n, insert n into G_I, & try to color it

Spill uncolored definitions & uses

Briggs' algorithm (1989)

Quick Aside …

Chaitin's Allocator (Bottom-up Coloring)



Chaitin's algorithm

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*°< 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

For contrast, Chaitin's algorithm (1981)

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

Is improvement still possible ?

⇒ yes, but the returns are getting rather small

Linear Scan Allocation

Coloring allocators are often viewed as too expensive for use in JIT environments, where compile time occurs at runtime

Linear scan allocators use an approximate interference graph and a version of the bottom-up local algorithm

Sun's HotSpot server compiler uses a complete Chaitin-Briggs allocator.

Live Interval

- [*i,j*] : live interval for variable *v* if there is no instruction with number *j´* > *j* such that *v* is live at *j´*, and there is no instruction with number *i´* < *i* such that *v* is live at *i´*
- conservative approximation of live ranges
- there may be sub ranges [i,j] in which v is not live
- trivial live range for any variable – [1,N]

Linear Scan Allocation

Building the Interval Graph

- Consider the procedure as a linear list of operations
- A live range for some name is an interval (x,y)

- Intervals overestimates live ranges and therefore interference

The Algorithm

- Use bottom-up local algorithm
- Distance to next use is well defined
- Algorithm is fast & produces reasonable allocations

Variations have been proposed that build on this scheme

The linear scan algorithm

- compute the live intervals.
- live intervals are stored in a list sorted in order of increasing start point.
- At each step, the algorithm maintains a list, *active*, of live intervals that overlap the current point and have been placed in registers.
- *active* list is sorted in order of increasing end point.

The code

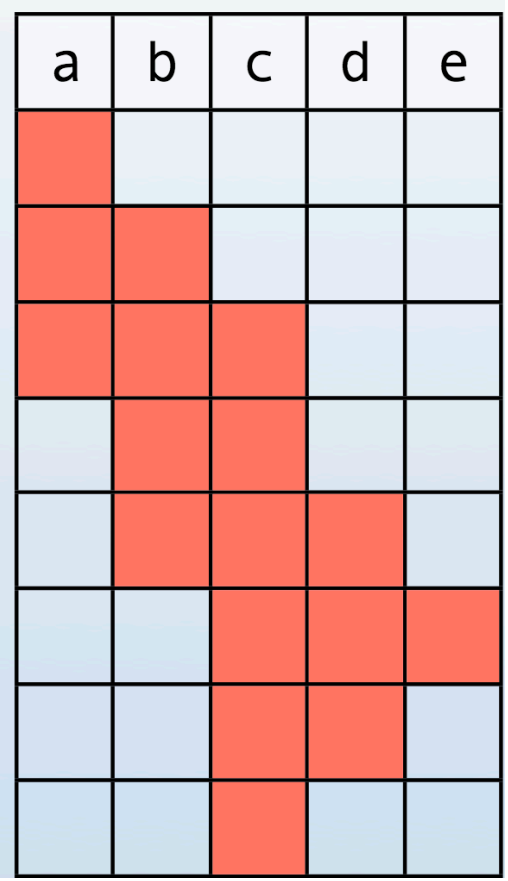
```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) =R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint [j] ≥ startpoint [i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint [spill] > endpoint [i] then
        register[i]← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i]← new stack location
```

Linear Scan example

Live ranges



Allocation

R1	R2
a	
a	b
a	b
	b
d	b
d	e
d	

c is spilled

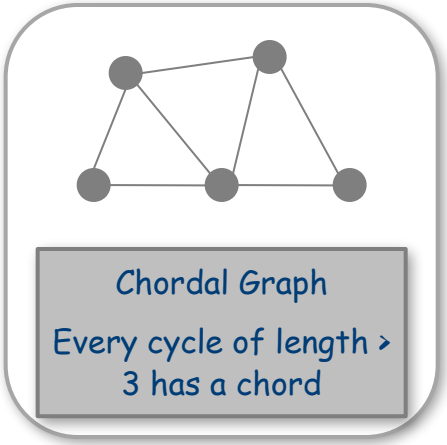
Global Coloring from SSA Form

- Chaitin-Briggs works from live ranges that are a coalesced version of SSA names

Observation: The interference graph of a program in SSA form is a chordal graph.

A chordal graph is a graph in which all cycles >3 has a cord
(an edge that is not part of the cycle
but connects two vertices of the cycle)

Observation: Chordal graphs can be colored in **O**(N) time.



Global Coloring from SSA Form

These two facts suggest allocation using an interference graph built from SSA Form

- SSA allocators use raw SSA names as live ranges

A-based allocation has created a lot of excitement in the last couple of years

Global Coloring from SSA Form

Coloring from SSA Names has its advantages

- If graph is k-colorable, it finds the coloring
  - (Opinion) An SSA-based allocator will find more k-colorable graphs than a live-range based allocator because SSA names are shorter and, thus, have fewer interferences.
- Allocator should be faster than a live-range allocator
  - Cost of live analysis folded into SSA construction, where it is amortised over other passes
  - Biggest expense in Chaitin-Briggs is the Build-Coalesce phase, which SSA allocator avoids, as it destroys the chordal graph

Global Coloring from SSA Form

Coloring from SSA Names has its disadvantages

- Coloring is rarely the problem
  - Most non-trivial codes spill; on trivial codes, both SSA allocator and classic Chaitin-Briggs are overkill. (Try linear scan?)
- SSA form provides no obvious help on spilling
- After allocation, code is still in SSA form
  - Need out-of-SSA translation
  - Introduce copies after allocation
  - Must run a post-allocation coalescing phase
    - → Algorithms exist that do not use an interference graph
    - → They are not as powerful as the Chaitin-Briggs coalescing phase

Hybrid Approach ?

How can the compiler attain both speed and precision?

Observation: lots of procedures are small & do not spill

Observation: some procedures are hard to allocate

Possible solution:

- Try different algorithms
- First, try linear scan
  - It is cheap and it may work
- If linear scan fails, try heavyweight allocator of choice
  - Might be Chaitin-Briggs, SSA, or some other algorithm
  - Use expensive allocator only when cheap one spills

This approach would not help with the speed of a complex compilation, but it might compensate on simple compilations