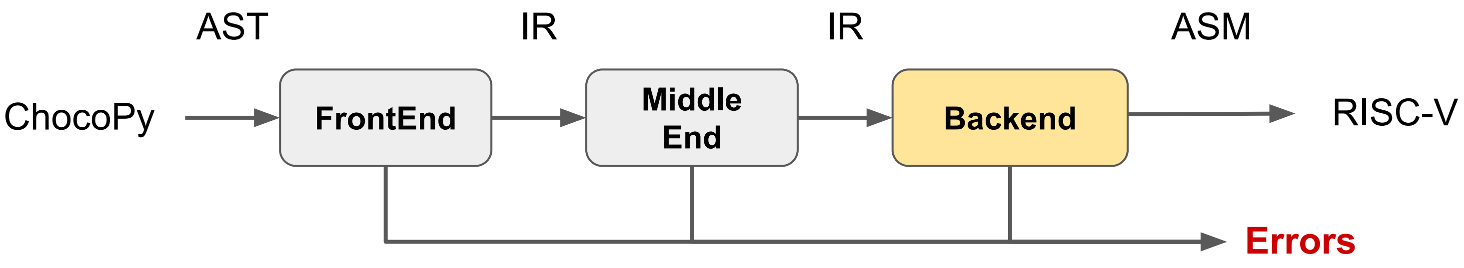
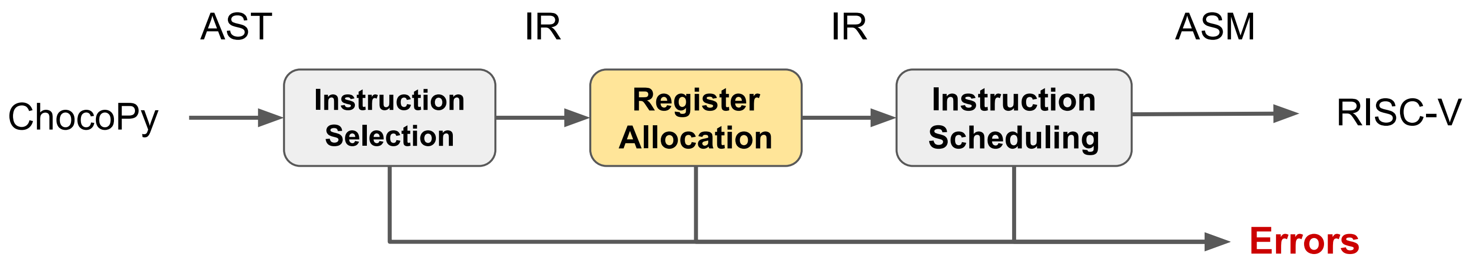
Compiling Techniques

Lecture 17: Register Allocation

Overview


AST
IR
IR
ASM
ChocoPy
FrontEnd
Middle End
Backend
RISC-V
Errors

Register Allocation



Introduction

This lecture:

- Local Allocation - spill code
- Global Allocation based on graph colouring
- Techniques to reduce spill code

Register Allocation

- Physical machines have limited number of registers
- Scheduling and selection typically assume infinite registers
- Register allocation and assignment from infinite to k registers

- Produce correct code that uses k (or fewer) registers
- Minimise added loads and stores
- Minimise space used to hold spilled values
- Operate efficiently:
  - $O(n)$, $O(n^2)$, but not $O(2^n)$

Register Allocation: Definitions

Allocation vs Assignment:

- *Allocation* is deciding which values to keep in registers
- *Assignment* is choosing specific registers for values

Liveness

A value is live from its definition to its last use.

Interference

Two values cannot be mapped to the same register wherever they are both live. Such values are said to **interfere**.

Live Range

The live range of a value is the set of statements at which it is live. A live range may be conservatively overestimated (e.g, just begin → end)

Register Allocation: Definitions

Spilling

Spilling saves a value from a register to memory.
That register is then free - Another value often loaded

Requires $\boldsymbol{F}$ registers to be reserved.

Clean and dirty values

A previously spilled value is **clean** if not changed since last spill.
Otherwise it is **dirty**.

A clean value can be spilled without a new store instruction.

Local Register Allocation

Register allocation only on basic block.

Let MAXLIVE be the maximum, over each instruction i in the block, of the number of values (pseudo-registers) live at i.

- If MAXLIVE ≤ k, allocation should be easy
- If MAXLIVE ≤ k, no need to reserve F registers for spilling
- If MAXLIVE > k, some values must be spilled to memory
- If MAXLIVE > k, need to reserve F registers for spilling

Two main forms:

- Top down
- Bottom up

Local Register Allocation: MAXLIVE

loadI	1028		$\Rightarrow r_a$	// $r_a \leftarrow 1028$
load	r_a		$\Rightarrow r_b$	// $r_b \leftarrow \text{MEM}(r_a)$
mult	r_a,	r_b	$\Rightarrow r_c$	// $r_c \leftarrow 1028 \cdot y$
load	x		$\Rightarrow r_d$	// $r_d \leftarrow x$
sub	r_d,	r_b	$\Rightarrow r_e$	// $r_e \leftarrow x-y$
load	z		$\Rightarrow r_f$	// $r_f \leftarrow z$
mult	r_e,	r_f	$\Rightarrow r_g$	// $r_g \leftarrow z \cdot (x-y)$
sub	r_g,	r_c	$\Rightarrow r_h$	// $r_h \leftarrow z \cdot (x-y) - 1028 \cdot y$
store	r_h		$\rightarrow r_a$	// $\text{MEM}(r_a) \leftarrow z \cdot (x-y) - 1028 \cdot y$

Local Register Allocation: MAXLIVE

Example MAXLIVE computation

Live registers

loadI	1028	$\Rightarrow r_a$	//	r_a
load	r_a	$\Rightarrow r_b$	//	r_a r_b
mult	r_a, r_b	$\Rightarrow r_c$	//	r_a r_b r_c
load	x	$\Rightarrow r_d$	//	r_a r_b r_c r_d
sub	r_d, r_b	$\Rightarrow r_e$	//	r_a r_c r_e
load	z	$\Rightarrow r_f$	//	r_a r_c r_e r_f
mult	r_e, r_f	$\Rightarrow r_g$	//	r_a r_c r_g
sub	r_g, r_c	$\Rightarrow r_h$	//	r_a r_h
store	r_h	$\rightarrow r_a$	//	

Local Register Allocation: MAXLIVE

Example MAXLIVE computation

MAXLIVE is 4

loadI	1028	$\Rightarrow$	r_a	//	r_a
load	r_a	$\Rightarrow$	r_b	//	r_a r_b
mult	r_a, r_b	$\Rightarrow$	r_c	//	r_a r_b r_c
load	x	$\Rightarrow$	r_d	//	r_a r_b r_c r_d
sub	r_d, r_b	$\Rightarrow$	r_e	//	r_a r_c r_e
load	z	$\Rightarrow$	r_f	//	r_a r_c r_e r_f
mult	r_e, r_f	$\Rightarrow$	r_g	//	r_a r_c r_g
sub	r_g, r_c	$\Rightarrow$	r_h	//	r_a r_h
store	r_h	$\rightarrow$	r_a	//	

Local register allocation: Top Down

Algorithm:

- If number of values > k
  - Rank values by occurrences
  - Allocate first k - *F* values to registers
  - Spill other values

Local register allocation: top down

Example top down

Usage counts

loadI	1028		$\Rightarrow r_a$	//	r_a								
load	r_a		$\Rightarrow r_b$	//	r_a	r_b							
mult	r_a,	r_b	$\Rightarrow r_c$	//	r_a	r_b	r_c						
load	x		$\Rightarrow r_d$	//	r_a	r_b	r_c	r_d					
sub	r_d,	r_b	$\Rightarrow r_e$	//	r_a		r_c		r_e				
load	z		$\Rightarrow r_f$	//	r_a		r_c		r_e	r_f			
mult	r_e,	r_f	$\Rightarrow r_g$	//	r_a		r_c				r_g		
sub	r_g,	r_c	$\Rightarrow r_h$	//	r_a							r_h	
store	r_h		$\rightarrow r_a$	//									

Counts
$r_a=4$
$r_b=3$
$r_c=2$
$r_d=2$
$r_e=2$
$r_f=2$
$r_g=2$
$r_h=2$

Local register allocation: Top Down

			Counts
loadI 1028 ⇒ r_a	// r_a	Must have r_d	$r_a=4$
load r_a ⇒ r_b	// r_a r_b	$r_c < r_a, r_b$	$r_b=3$
mult r_a, r_b ⇒ r_c	// r_a r_b r_c	Spill r_c	$r_c=2$
load x ⇒ r_d	// r_a r_b r_c r_d		$r_d=2$
sub r_d, r_b ⇒ r_e	// r_a r_c r_e		$r_e=2$
load z ⇒ r_f	// r_a r_c r_e r_f		$r_f=2$
mult r_e, r_f ⇒ r_g	// r_a r_c r_g	Restore r_c	$r_g=2$
sub r_g, r_c ⇒ r_h	// r_a r_h		$r_h=2$
store r_h → r_a	//		

Local Register Allocation: Top down

Example top down

Spill code inserted

```
loadI  1028              r_a
load   r_a               r_b
mult   r_a, r_b      =>  r_c
store  r_c           ->  r_arp, spill_c
load   x                 r_d
sub    r_d, r_b          r_e
load   z                 r_f
mult   r_e, r_f          r_g
load   r_arp, spill_c    r_c
sub    r_f, r_c          r_h
store  r_h           ->  r_a
```

Local register allocation: Top Down

Example top down

Register assignment straightforward

```
loadI  1028              r1
load   r1                r2
mult   r1, r2        =>  r3
store  r3            ->  rarp, spillc
load   x                 r3
sub    r3, r2            r2
load   z                 r3
mult   r2, r3            r2
load   rarp, spillc      r3
sub    r2, r3            r2
store  r2            ->  r1
```

Local register allocation: Bottom Up

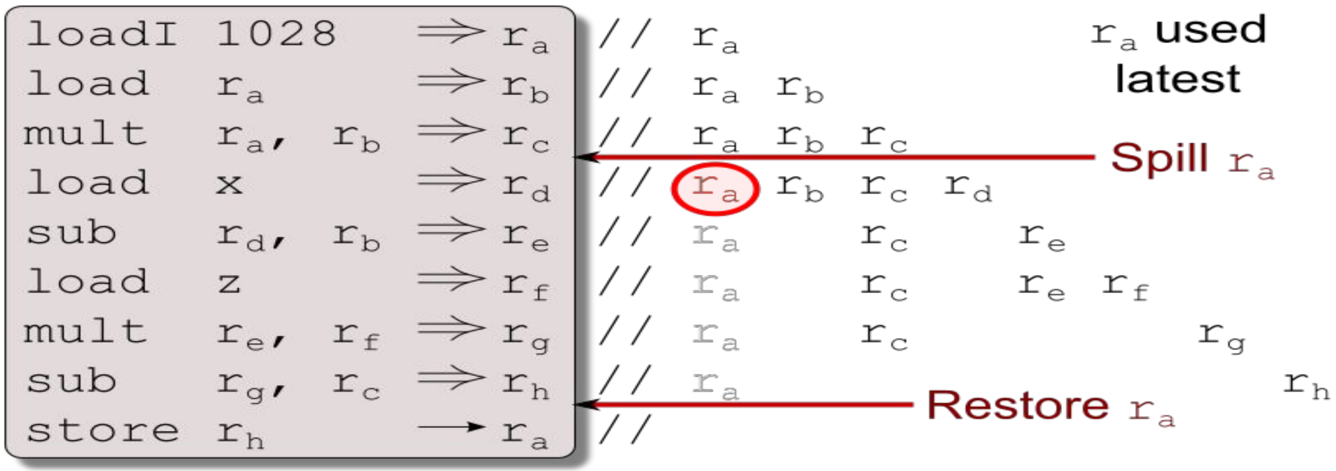
Algorithm:

- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement:

- Spill the value whose next use is farthest in the future
- Prefer clean value to dirty value

Local register allocation: Bottom Up



Local register allocation: Bottom Up

Example bottom up

Spill code inserted

loadI	1028		r_a
load	r_a		r_b
mult	r_a, r_b	$\Rightarrow$	r_c
store	**r_a**	$\rightarrow$	**r_{arp}, $spill_a$**
load	x		r_d
sub	r_d, r_b		r_e
load	z		r_f
mult	r_e, r_f		r_g
sub	r_f, r_c		r_h
load	**r_{arp}, $spill_a$**		**r_a**
store	r_h	$\rightarrow$	r_a

Global register allocation

Local allocation does not capture reuse of values across multiple blocks

Most modern, global allocators use a graph-colouring paradigm

- Build a "**conflict graph**" or "**interference graph**"
  - Data flow based liveness analysis for interference
- Find a **k-colouring** for the graph, or change the code to a nearby problem that it can k-colour
- NP-complete under nearly all assumptions[1]

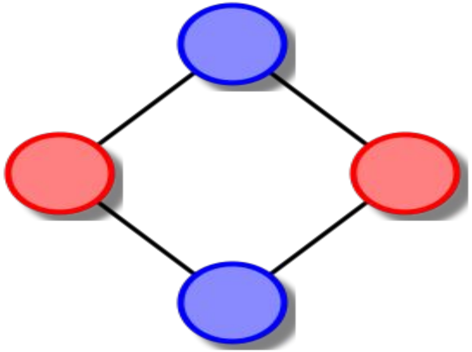
[1] Local allocation is NP-complete with dirty vs clean

Global register allocation: algorithm sketch

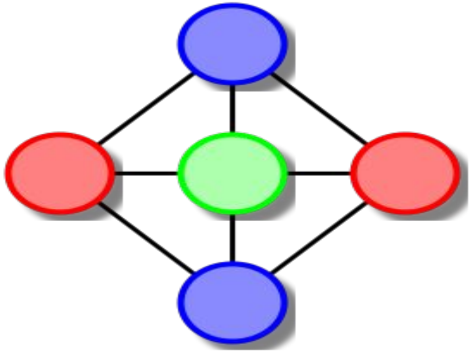
- From live ranges construct an interference graph
- Colour interference graph so that no two neighbouring nodes have same colour
- If graph needs more than k colours - transform code
  - Coalesce merge-able copies
  - Split live ranges
  - Spill
- Colouring is NP-complete so we will need heuristics
- Map colours onto physical registers

Global register allocation: Graph Coloring

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



2-colourable



3-colourable

Global register allocation: Interference Graph

The interference graph, $G = (N, E)$

- Nodes in G represent values, or live ranges
- Edges in G represent individual interferences
- $\forall x, y \in N, x \rightarrow y \in E$ *iff* x and y interfere

A *k-colouring* of G can be mapped into an allocation to k registers

Two values interfere wherever they are both live
Two live ranges interfere if their values interfere at any point

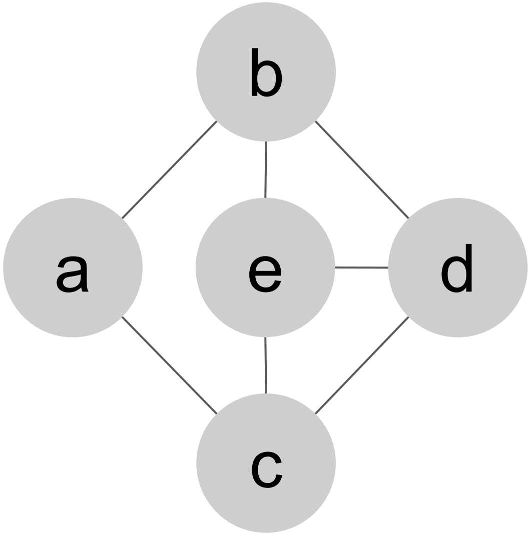
Global register allocation: Coloring the Register Graph

- Degree[3] of a node (n°) is a loose upper bound on colourability
- Any node, n, such that $n^\circ < k$ is always trivially k-colourable
  - Trivially colourable nodes cannot adversely affect the colourability of neighbours
  - Can remove them from graph
  - Reduces degree of neighbours - may be trivially colourable
- If left with any nodes such that $n^\circ \geq k$ spill one
  - Reduces degree of neighbours - may be trivially colourable

Global register allocation: Chaitin’s Algorithm

1. While ∃ vertices with < k neighbours in G
   - Pick any vertex n such that n° < k and put it on the stack
   - Remove n and all edges incident to it from G
2. If G is non-empty (n° >= k, ∃ n ∋ G) then:
   - Pick vertex n (heuristic), spill live range of n
   - Remove vertex n and edges from GI, put n on “spill list”
   - Goto step 1
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 of the stack and colour them in the lowest colour not used by some neighbour

Global Register Allocation: Chaitin’s Algorithm



Colour with k = 3 colours

r1

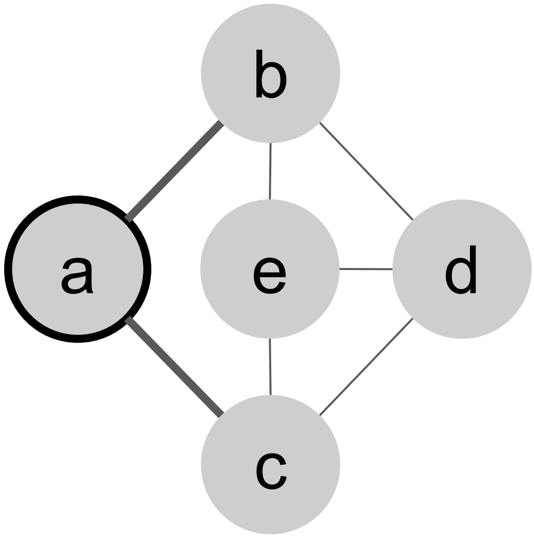
r2

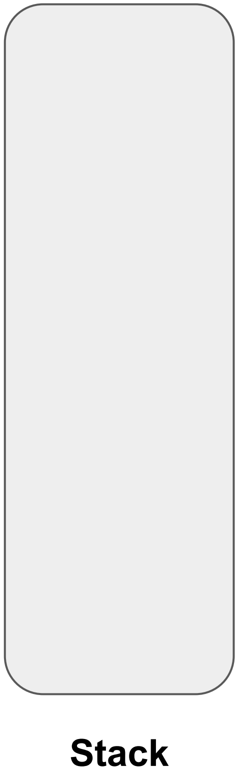
r3

Stack

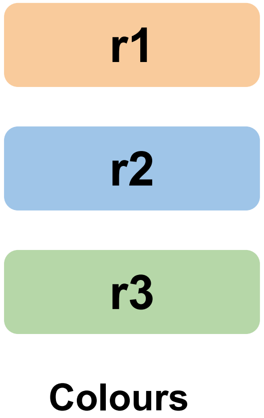
Colours

Global Register Allocation: Chaitin’s Algorithm

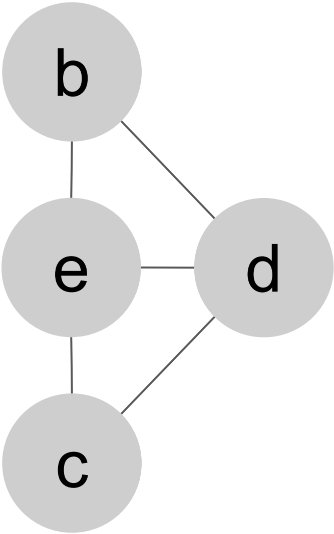


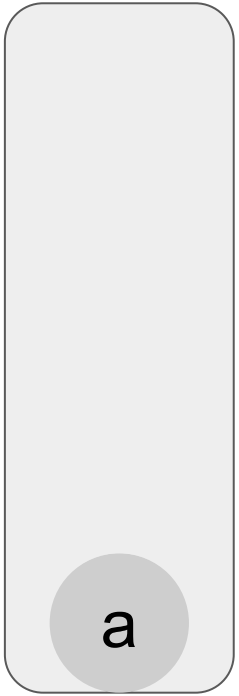


a° = 2 < k Choose a



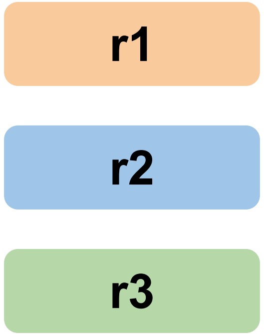
Global Register Allocation: Chaitin's Algorithm





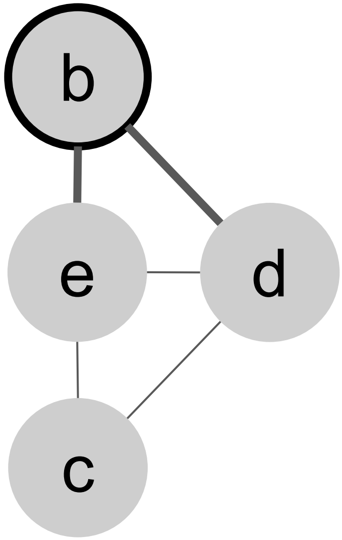
Stack

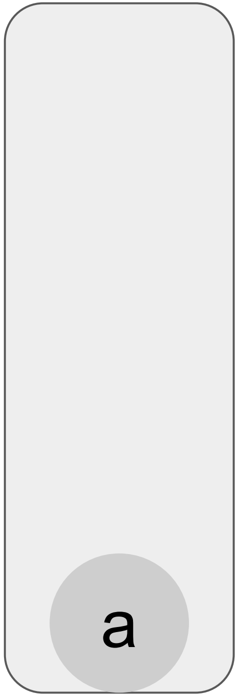
Push a and remove from graph



Colours

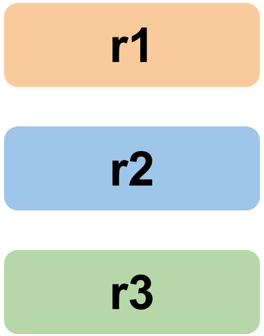
Global Register Allocation: Chaitin's Algorithm





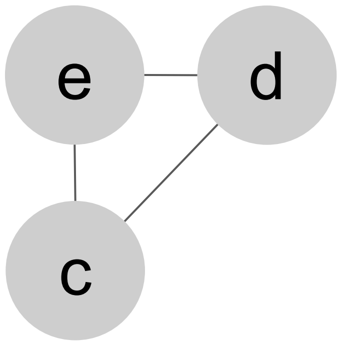
Stack

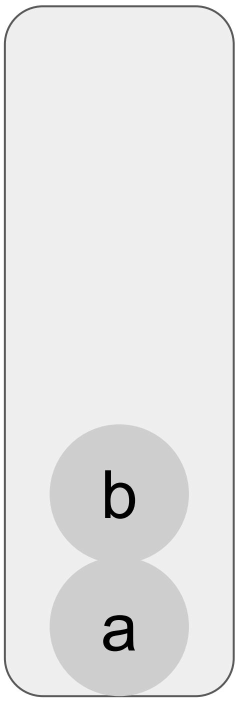
$b^\circ = 2 < k$ and
$c^\circ = 2 < k$
Choose b



Colours

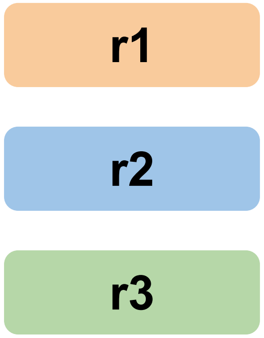
Global Register Allocation: Chaitin’s Algorithm





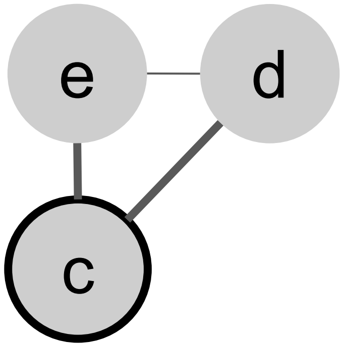
Stack

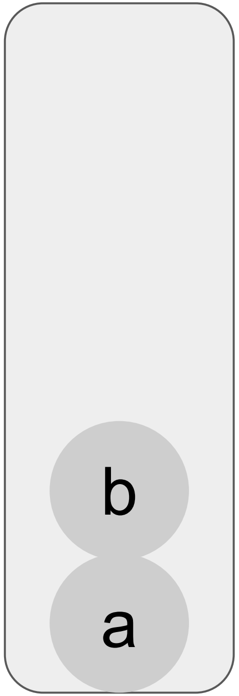
Push b and remove from graph



Colours

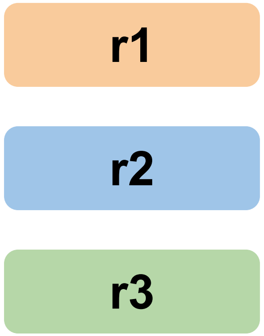
Global Register Allocation: Chaitin’s Algorithm





Stack

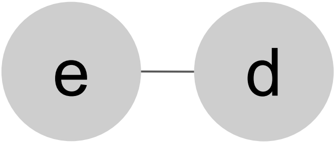
c° = 2 < k,
d° = 2 < k, and
e° = 2 < k
Choose c

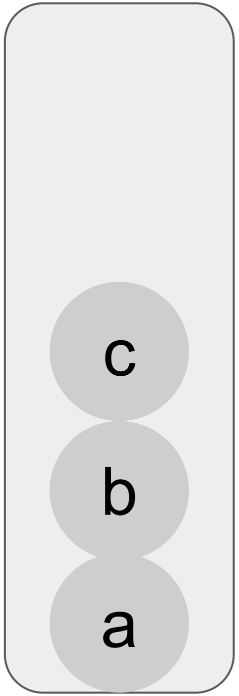


Colours

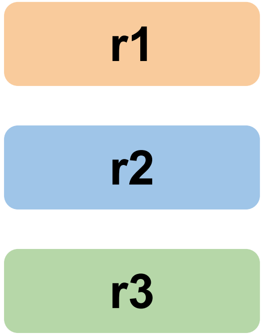
Global Register Allocation: Chaitin’s Algorithm

Push c and remove from graph



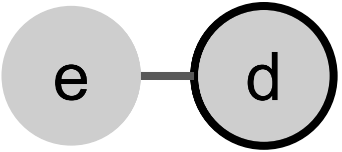


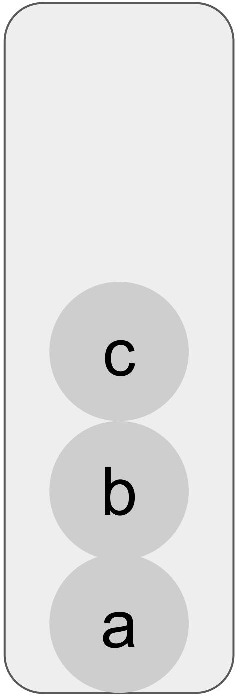
Stack



Colours

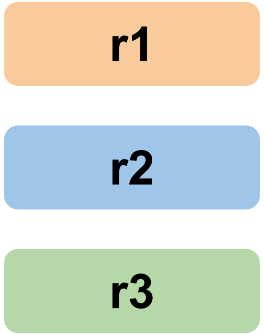
Global Register Allocation: Chaitin's Algorithm





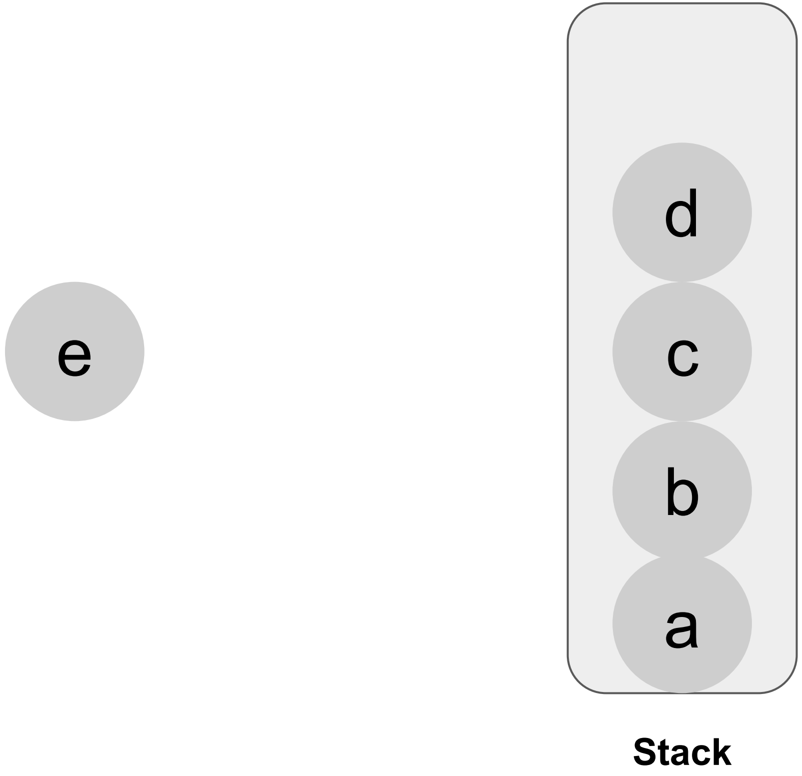
Stack

$d^\circ = 1 < k$ and
$e^\circ = 1 < k$
Choose d

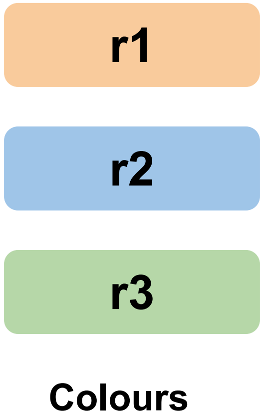


Colours

Global Register Allocation: Chaitin’s Algorithm

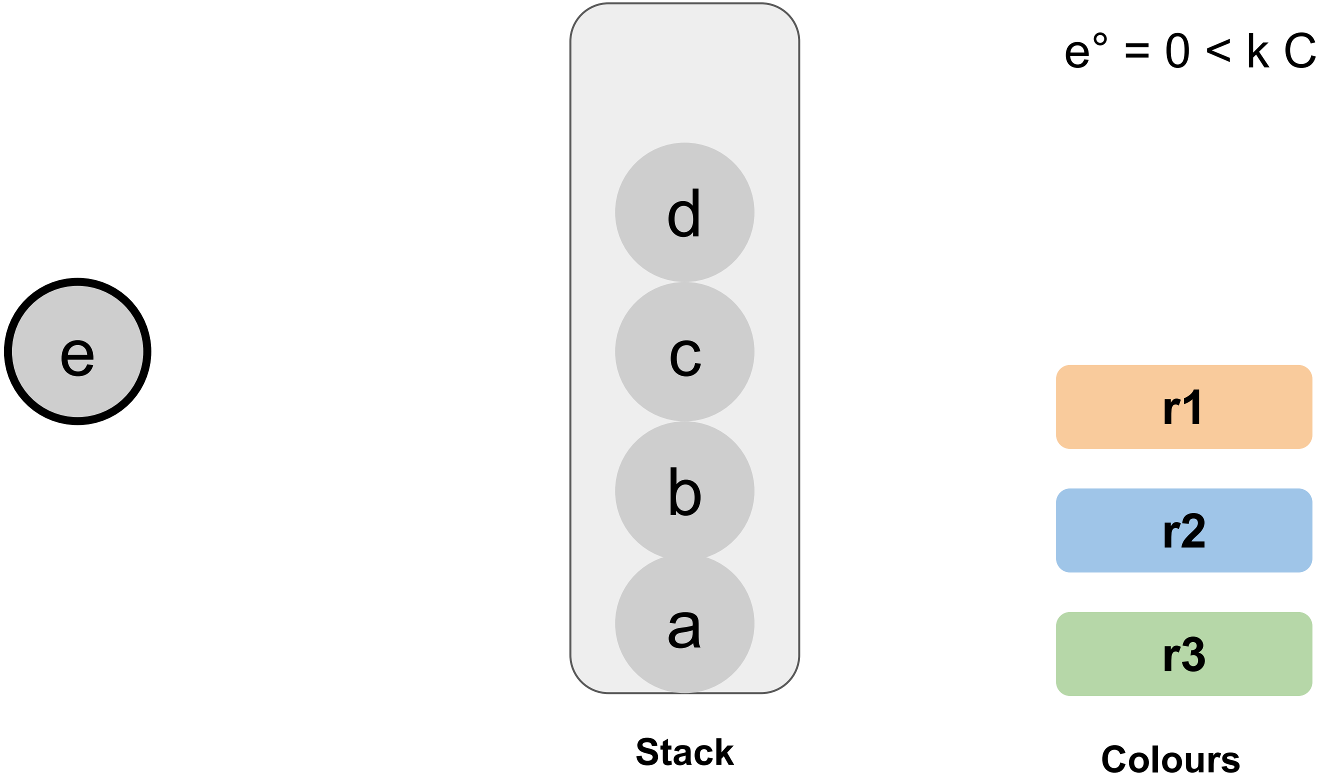


Push d and remove from graph

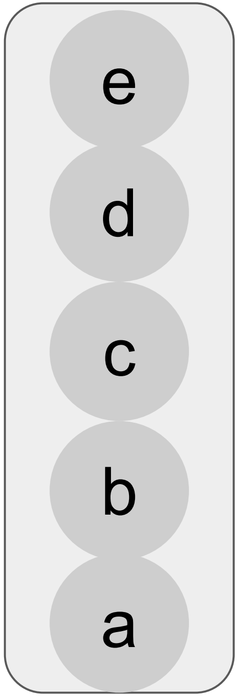


Global Register Allocation: Chaitin’s Algorithm

$e^{\circ} = 0 < k$ Choose e

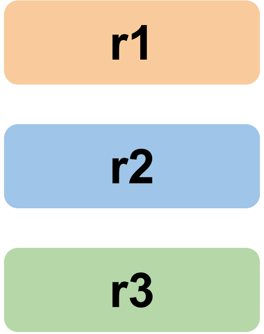


Global Register Allocation: Chaitin’s Algorithm



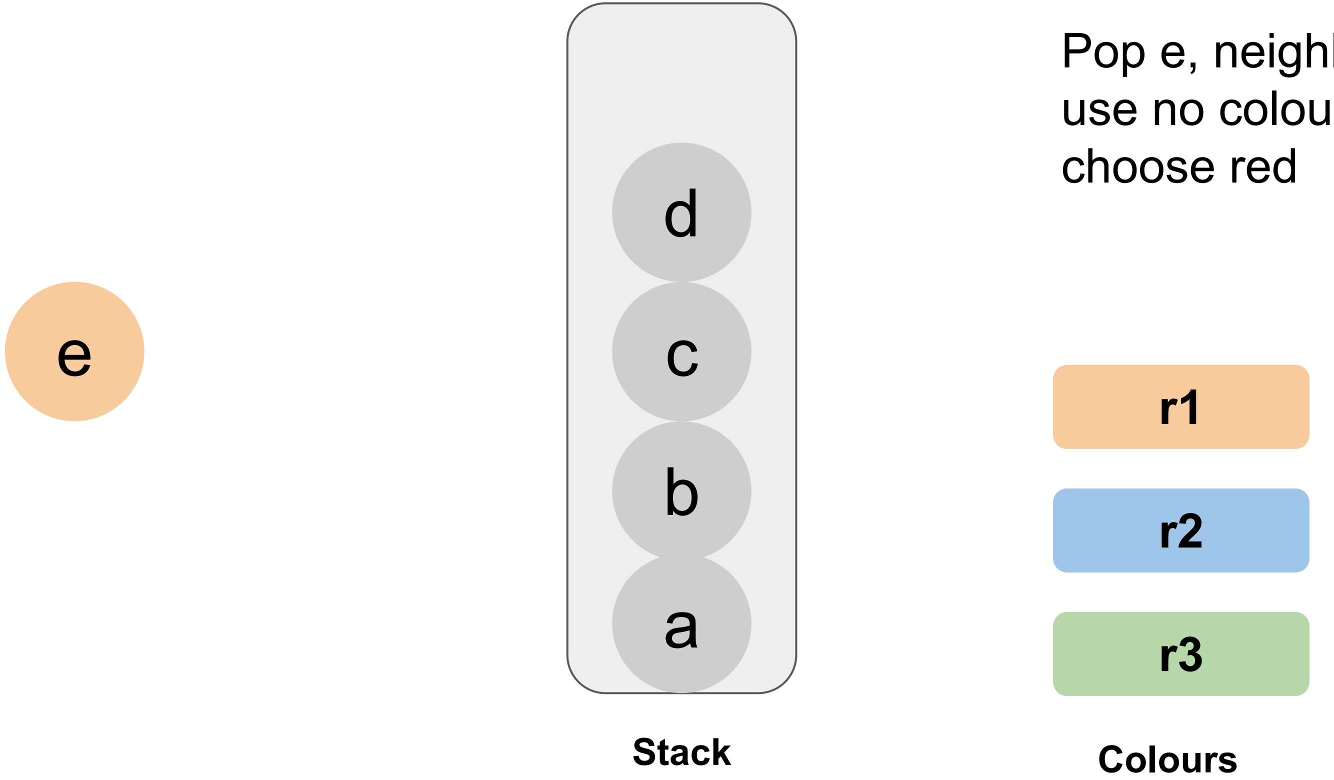
Stack

Push e and remove from graph



Colours

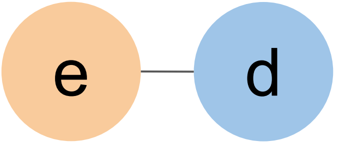
Global Register Allocation: Chaitin’s Algorithm

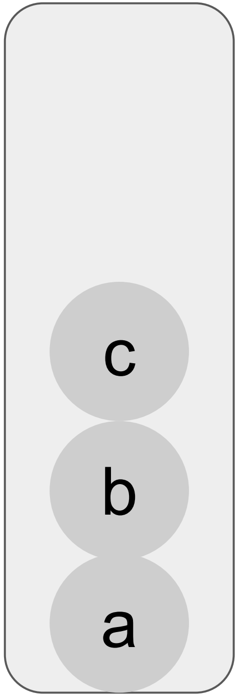


Pop e, neighbours use no colours, choose red

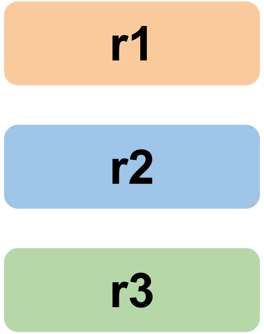
Global Register Allocation: Chaitin’s Algorithm

Pop d, neighbours use red, choose blue



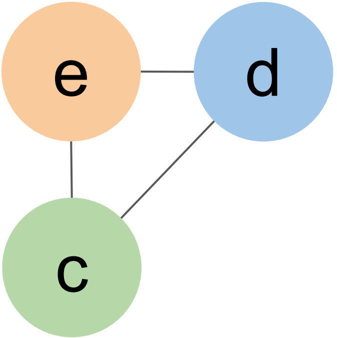


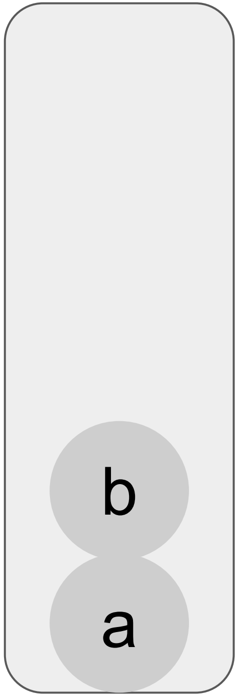
Stack



Colours

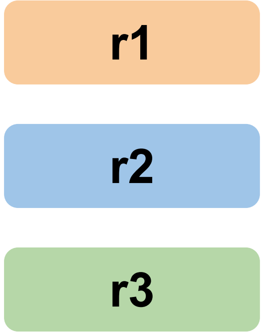
Global Register Allocation: Chaitin’s Algorithm





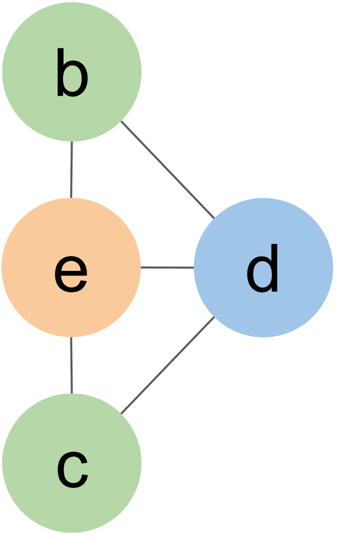
Stack

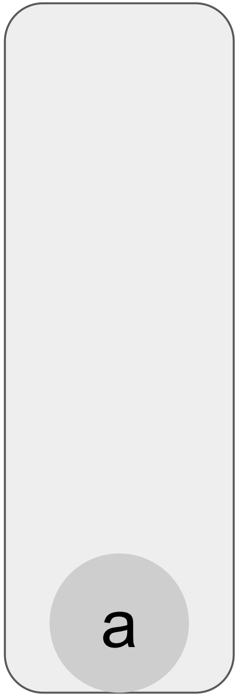
Pop c, neighbours use red and blue choose green



Colours

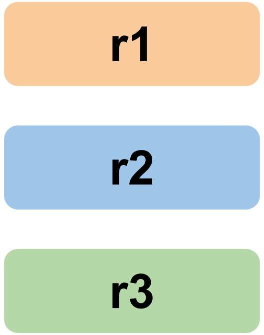
Global Register Allocation: Chaitin’s Algorithm





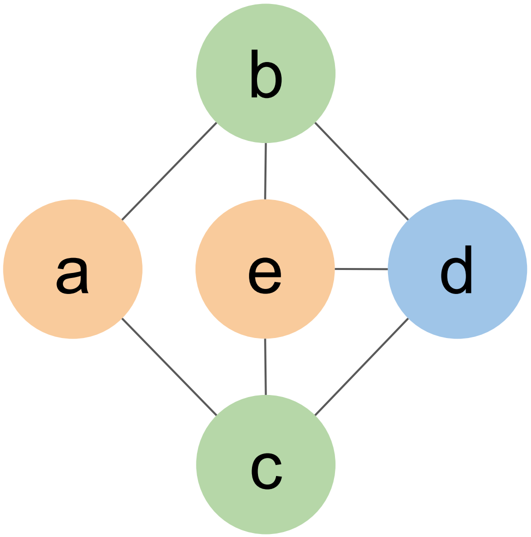
Stack

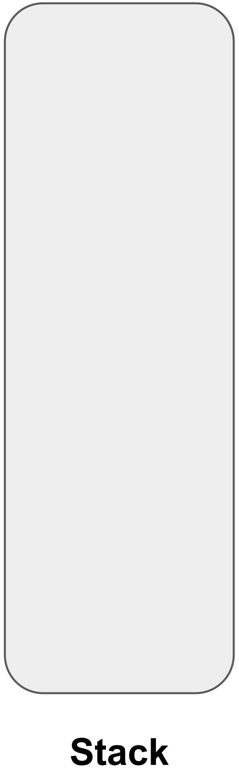
Pop c, neighbours use red and blue choose green



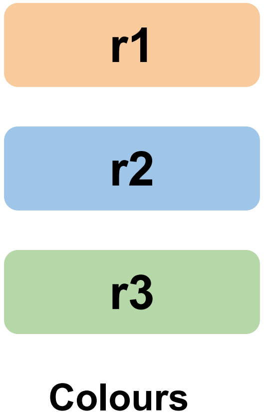
Colours

Global Register Allocation: Chaitin's Algorithm





Pop a, neighbours use blue choose red



Global Register Allocation: Optimistic Colouring

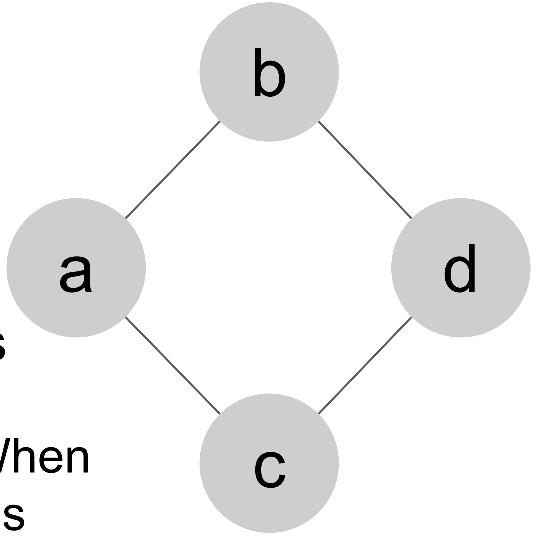
If Chaitin's algorithm reaches a state where every node has k or more neighbours, it chooses a node to spill.

Example of Chaitin overzealous spilling

k = 2

Graph is 2-colourable

Chaitin must immediately spill one of these nodes

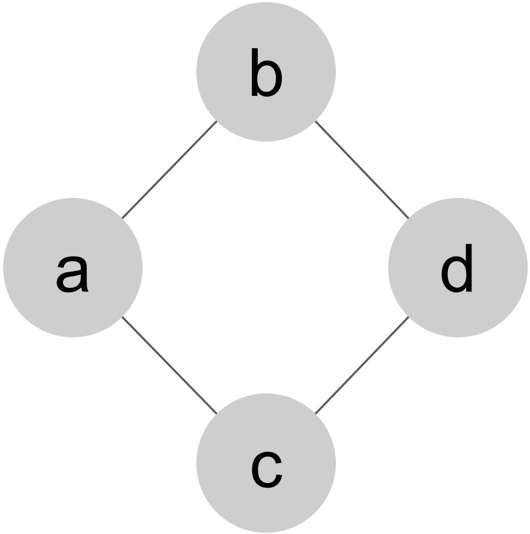


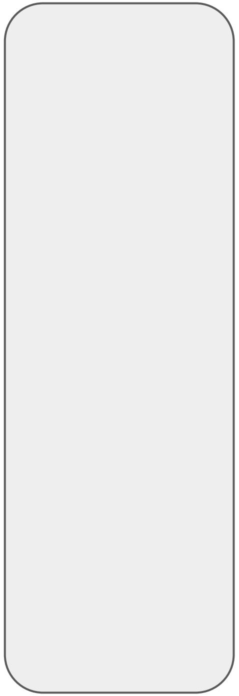
Briggs said, take that same node and push it on the stack! When you pop it off, a colour might be available for it! Chaitin-Briggs algorithm uses this to colour that graph

Global register allocation: Chaitin-Briggs algorithm

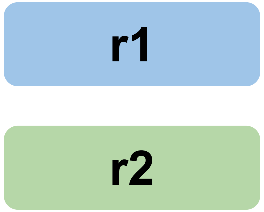
- While $\exists$ vertices with $< k$ neighbours in G
  - Pick any vertex n such that $n° < k$ and put it on the stack
  - Remove n and all edges incident to it from GI
- If G is non-empty ($n° >= k$, $\forall n \in G$) then:
  - Pick vertex n (heuristic) (Do not spill)
  - Remove vertex n from GI, put n on stack (Not spill list)
  - Goto step 1
- Otherwise, successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour
  - If some vertex cannot be coloured, then pick an uncoloured vertex to spill, spill it, and restart at step 1

Global Register Allocation: Chaitin-Briggs Algorithm



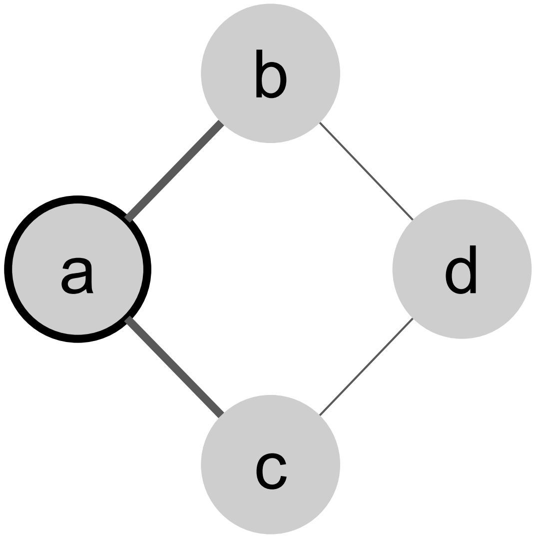


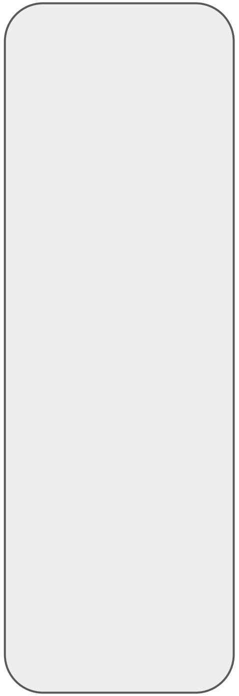
Stack



Colours

Global Register Allocation: Chaitin-Briggs Algorithm





Stack

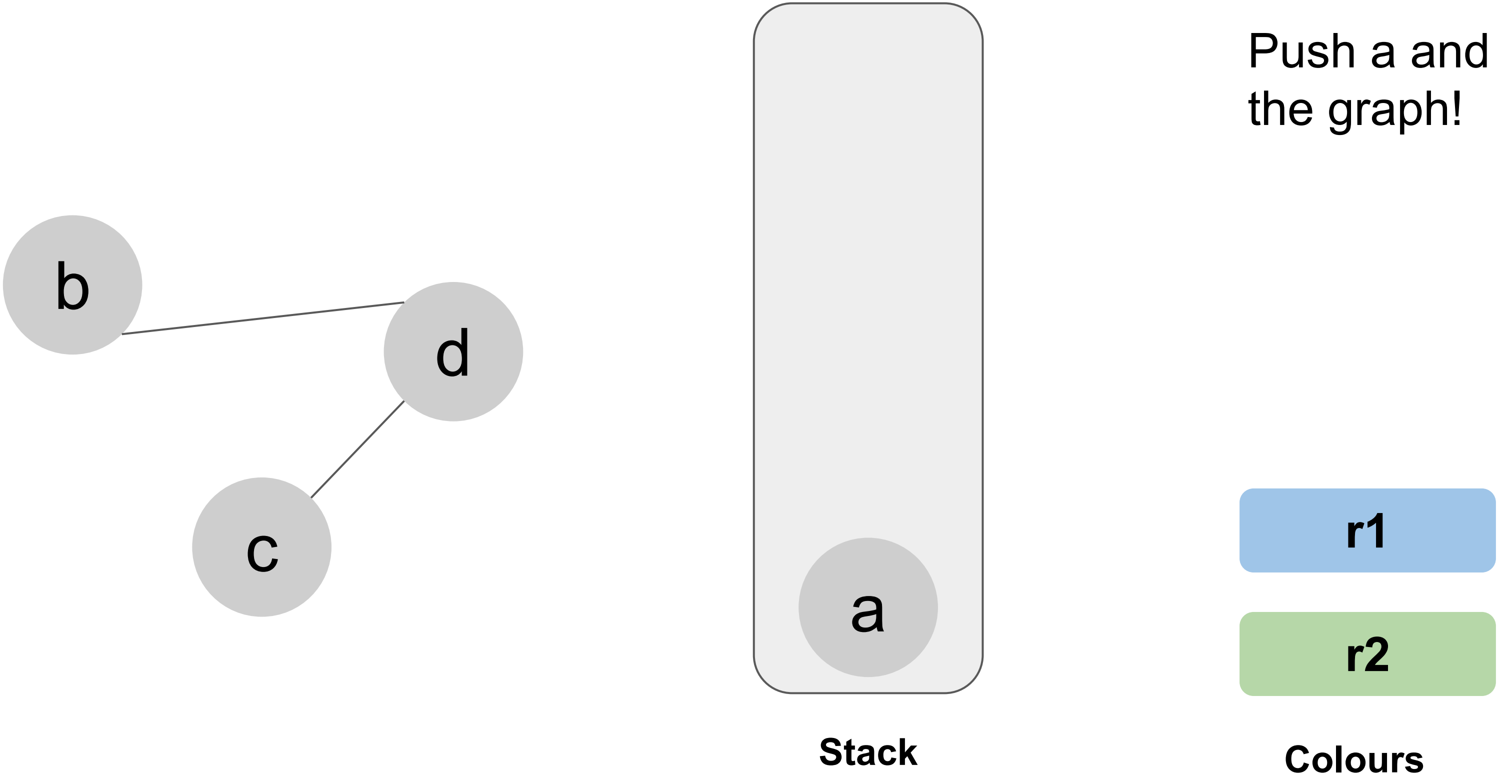
$a^\circ = 2 \geq k$
Don’t Spill, Choose a!

r1

r2

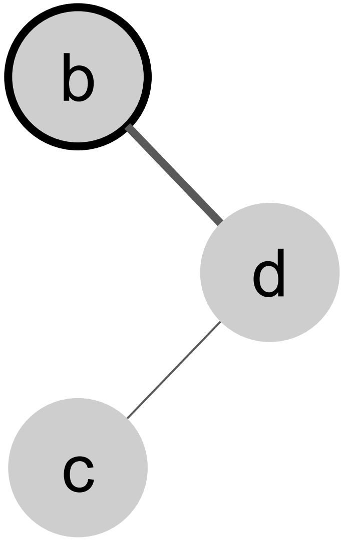
Colours

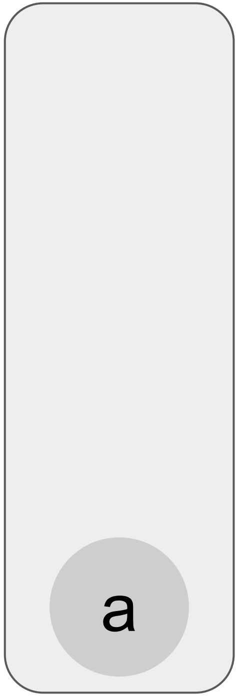
Global Register Allocation: Chaitin-Briggs Algorithm



Push a and remove the graph!

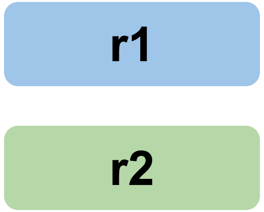
Global Register Allocation: Chaitin-Briggs Algorithm





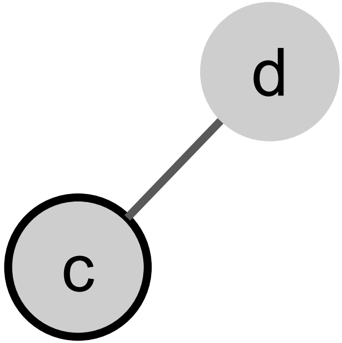
Stack

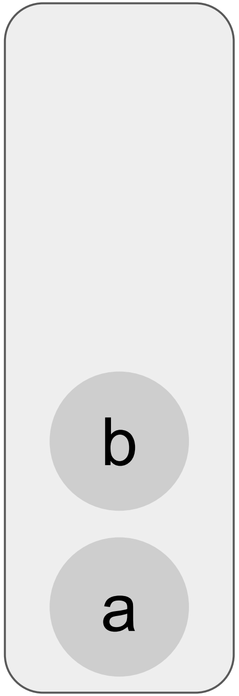
b° = 1 < k and
c° = 1 < k
Choose b



Colours

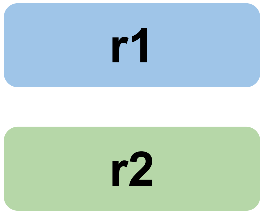
Global Register Allocation: Chaitin-Briggs Algorithm





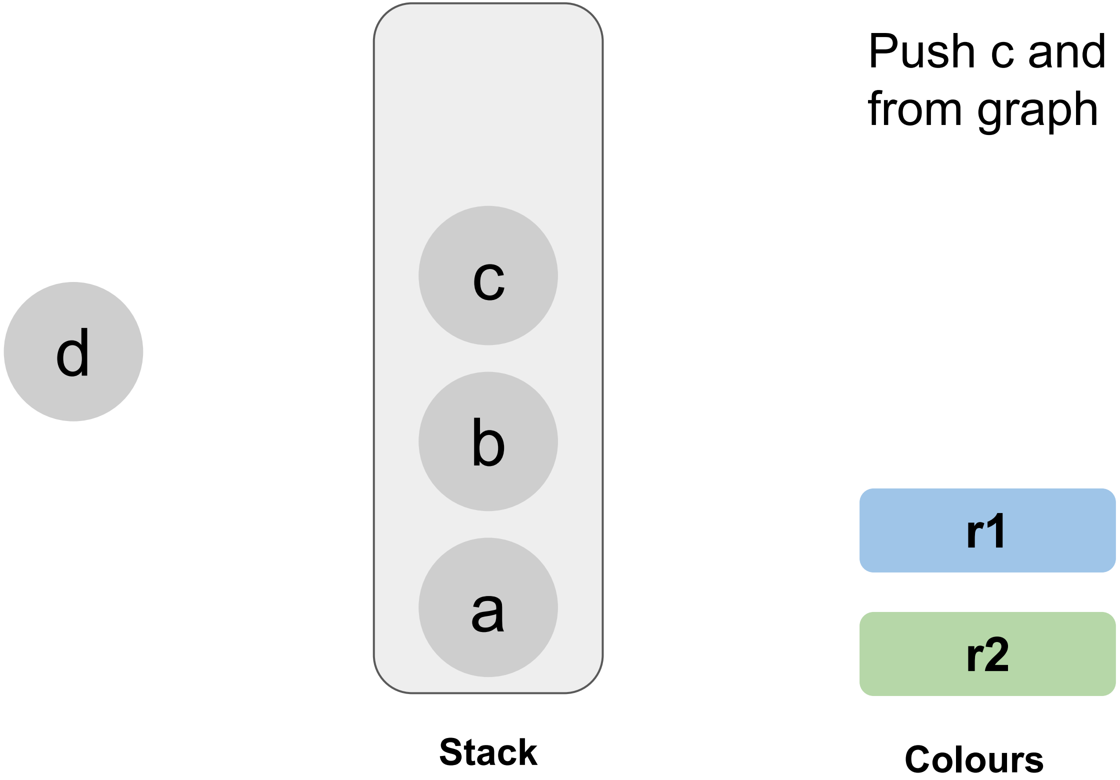
Stack

$c^\circ = 1 < k$,
and $d^\circ = 1 < k$
Choose c



Colours

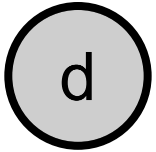
Global Register Allocation: Chaitin-Briggs Algorithm

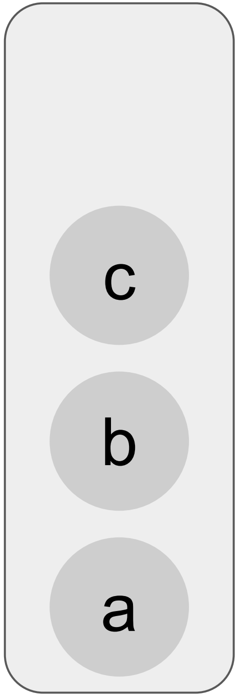


Push c and remove from graph

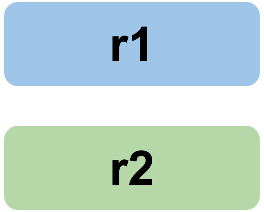
Global Register Allocation: Chaitin-Briggs Algorithm

d° = 1 < k Choose d



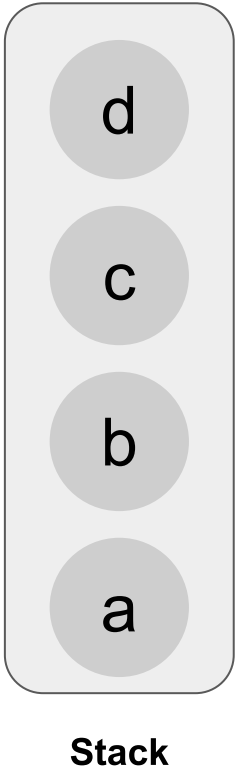


Stack

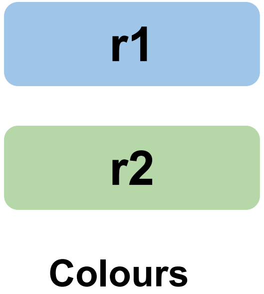


Colours

Global Register Allocation: Chaitin-Briggs Algorithm

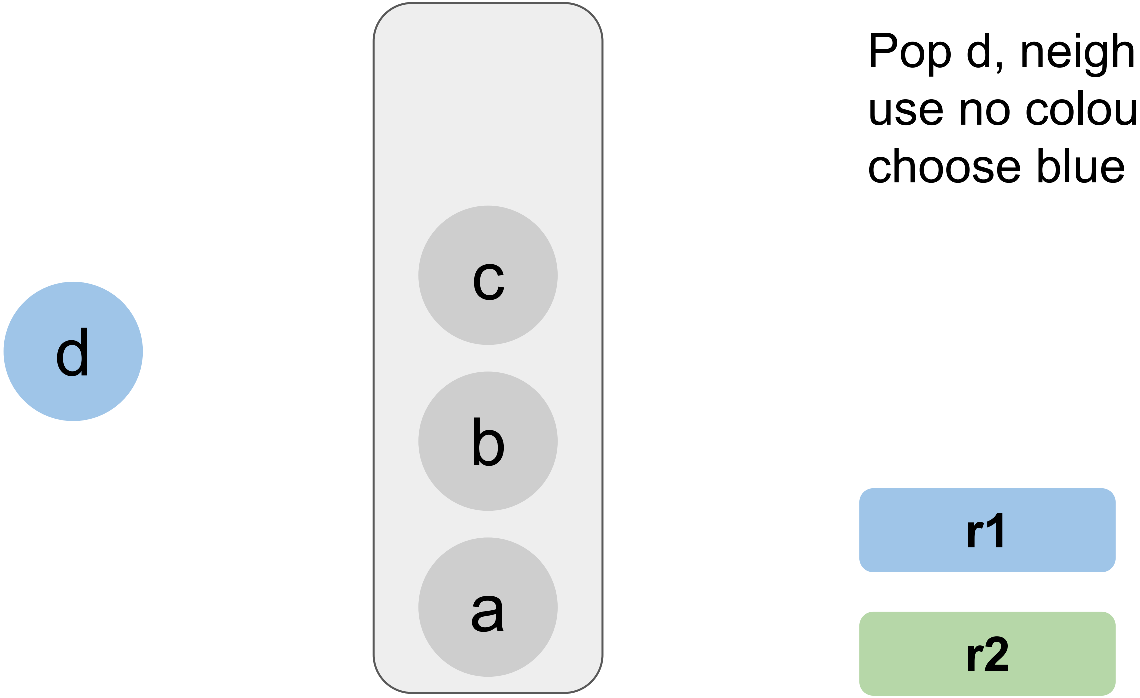


Push d and remove from graph



Global Register Allocation: Chaitin-Briggs Algorithm

Pop d, neighbours use no colours, choose blue

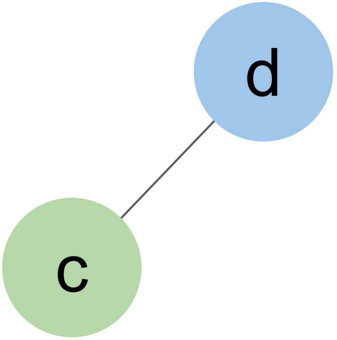


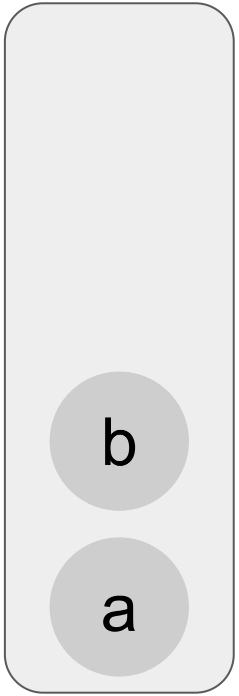
Stack

Colours

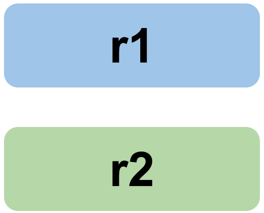
Global Register Allocation: Chaitin-Briggs Algorithm

Pop c, neighbours use blue choose green



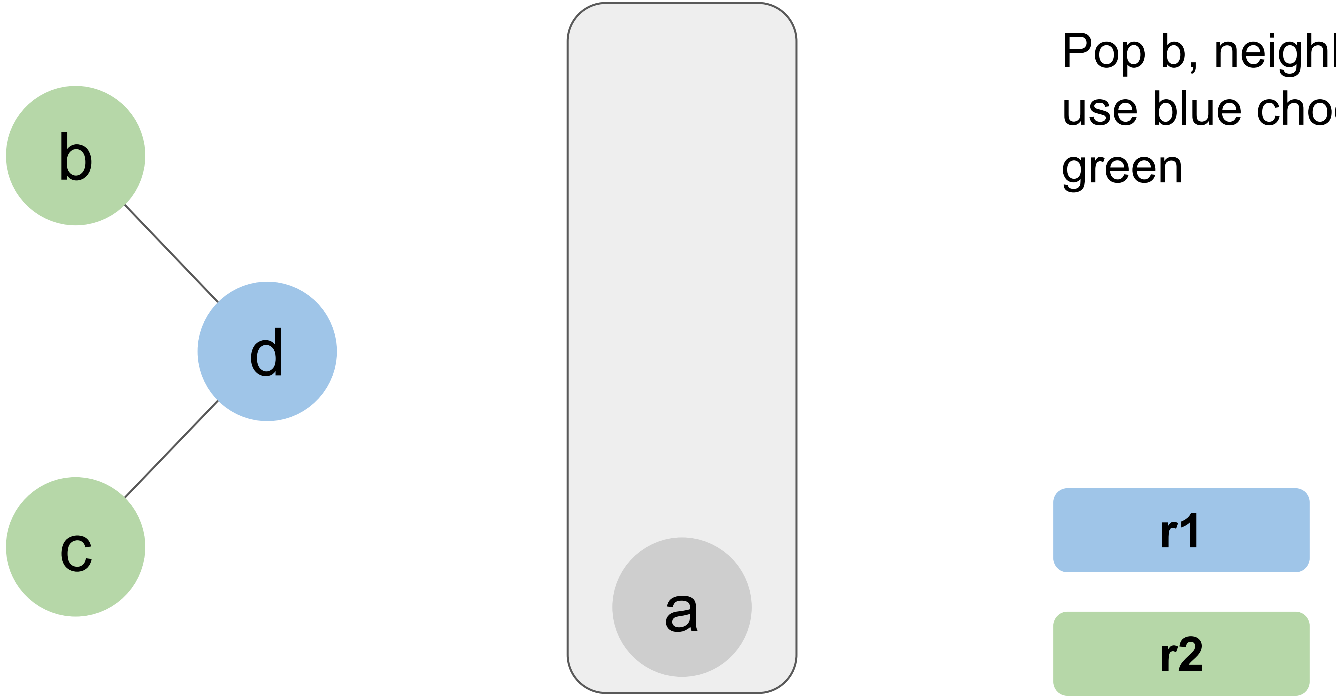


Stack



Colours

Global Register Allocation: Chaitin-Briggs Algorithm

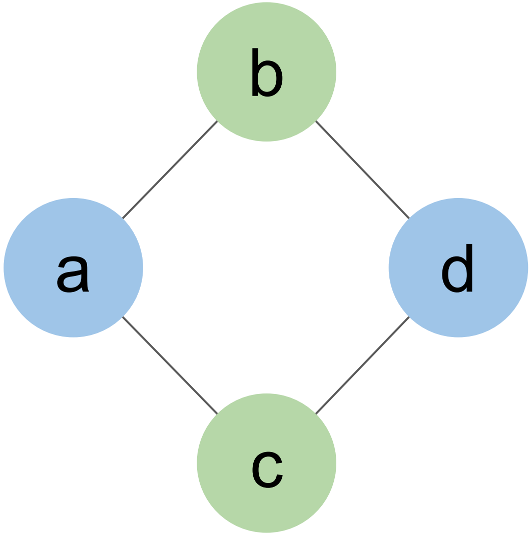


Stack

Colours

Pop b, neighbours use blue choose green

Global Register Allocation: Chaitin-Briggs Algorithm



Pop a, neighbours use green choose blue

Stack

r1

r2

Colours

Global register allocation: Spill Candidates

- Minimise spill cost/degree
- Spill cost is the loads and stores needed. Weighted by scope - i.e. avoid inner loops
- The higher the degree of a node to spill the greater the chance that it will help colouring
- Negative spill cost load and store to same memory location with no other uses
- Infinite cost - definition immediately followed by use. Spilling does not decrease live range

Global Register Allocation: Alternative Spilling

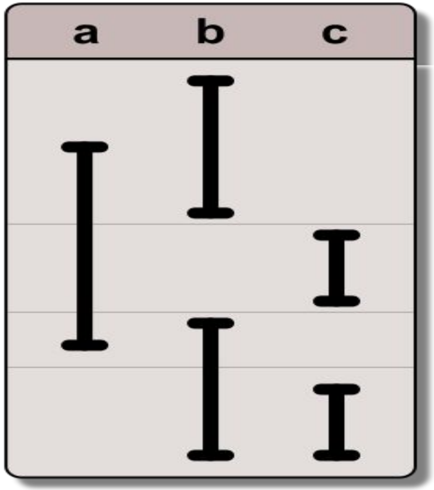
- Splitting live ranges
- Coalesce

Global Register Allocation: Live Range Splitting

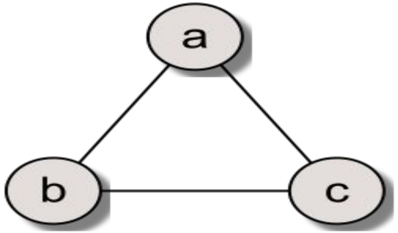
- A whole live range may have many interferences, but perhaps not all at the same time
- Split live range into two variables connected by copy
- Can reduce degree of interference graph
- Smart splitting allows spilling to occur in “cheap” regions

Global register allocation

Splitting example: Non contiguous live ranges - cannot be 2 coloured



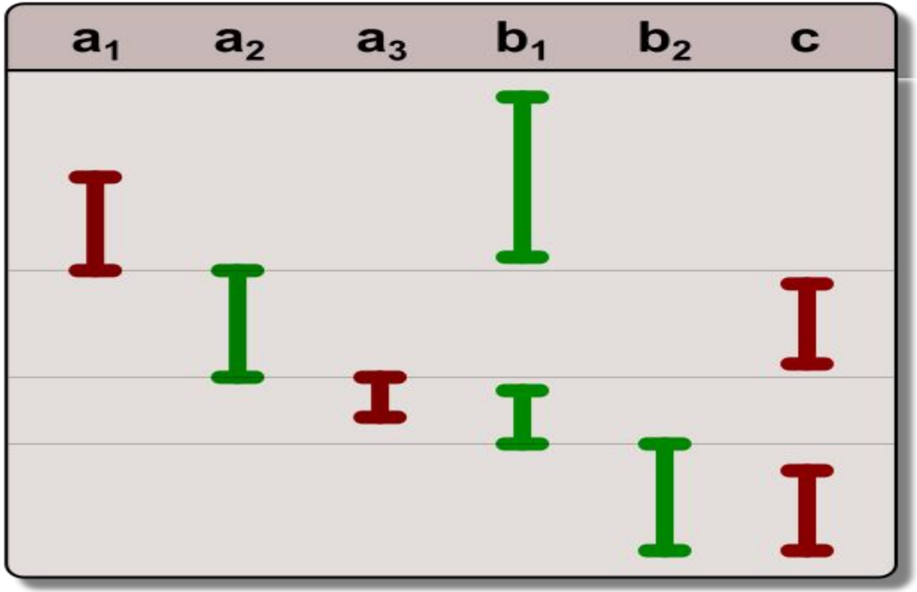
Live ranges



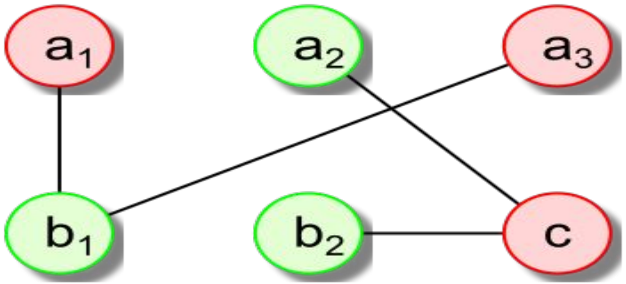
Interference Graph

Global Register Allocation: Live Range Splitting

Splitting example: Non contiguous live ranges - can be 2 coloured



Live ranges



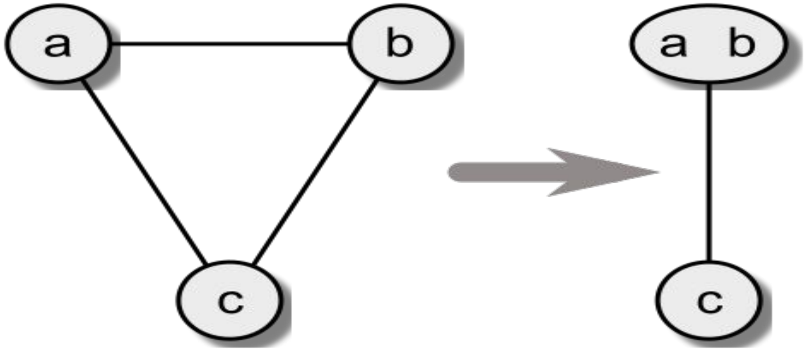
Interference Graph

Global register allocation: Coalescing

If two ranges don’t interfere and are connected by a copy coalesce into one – opposite of splitting. Reduces degree of nodes that interfered with both

If x := y and x → y ∈ G_I *then can combine* LR_x *and* LR_y

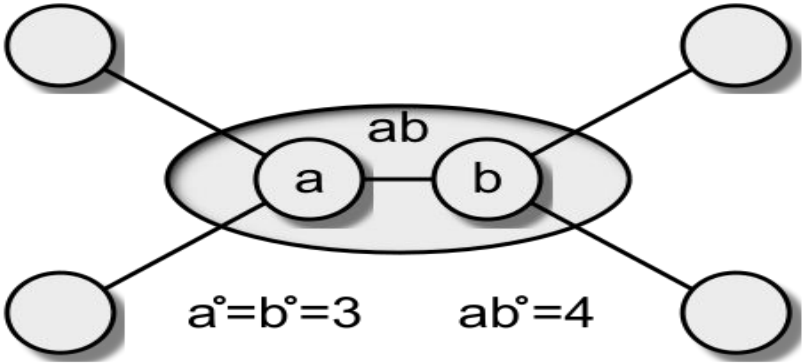
- Eliminates the copy operation
- Reduces degree of LRs that interfere with both *x* and *y*
- If a node interfered with both before, coalescing helps
- As it reduces degree, often applied before colouring takes place



Global register allocation: Coalescing

Coalescing can make the graph harder to color

- Typically, $LR_{xy}°$ > max ($LR_x°$, $LR_y°$)
- If max ($LRx°$, $LRy°$) < k and k < $LR_{xy}°$ then LR_{xy} might spill, while LR_x and LR_y would not spill



Global register allocation: Coalescing

Observation led to conservative coalescing

1. Conceptually, coalesce x and y iff $x \rightarrow y \in G_I$ and $LR_{xy}{}^{\circ} < k$
2. We can do better
   - Coalesce LR_x and LR_y iff LR_{xy} has < k neighbours with degree > k
   - Only neighbours of "significant degree" can force LR_{xy} to spill
3. Always safe to perform coalesce
   - Cannot introduce a node of non-trivial degree
   - Cannot introduce a new spill

Global register allocation: Other Approaches

- Top-down uses high level priorities to decide on colouring
- Hierarchical approaches - use control flow structure to guide allocation
- Exhaustive allocation - go through combinatorial options - very expensive but occasional improvement
- Re-materialisation - if easy to recreate a value do so rather than spill
- Passive splitting using a containment graph to make spills effective
- Linear scan - fast but weak; useful for JITs

Global register allocation: Ongoing work

- Eisenbeis et al examining optimality of combined reg alloc and scheduling. Difficulty with general control-flow
- Partitioned register sets complicate matters. Allocation can require insertion of code which in turn affects allocation.
- Leupers investigated use of genetic algs for TM series partitioned reg sets.
- New work by Fabrice Rastello and others. Chordal graphs reduce complexity
- As latency increases see work in combined code generation, instruction scheduling and register allocation

Summary

- Local Allocation - spill code
- Global Allocation based on graph colouring
- Techniques to reduce spill code