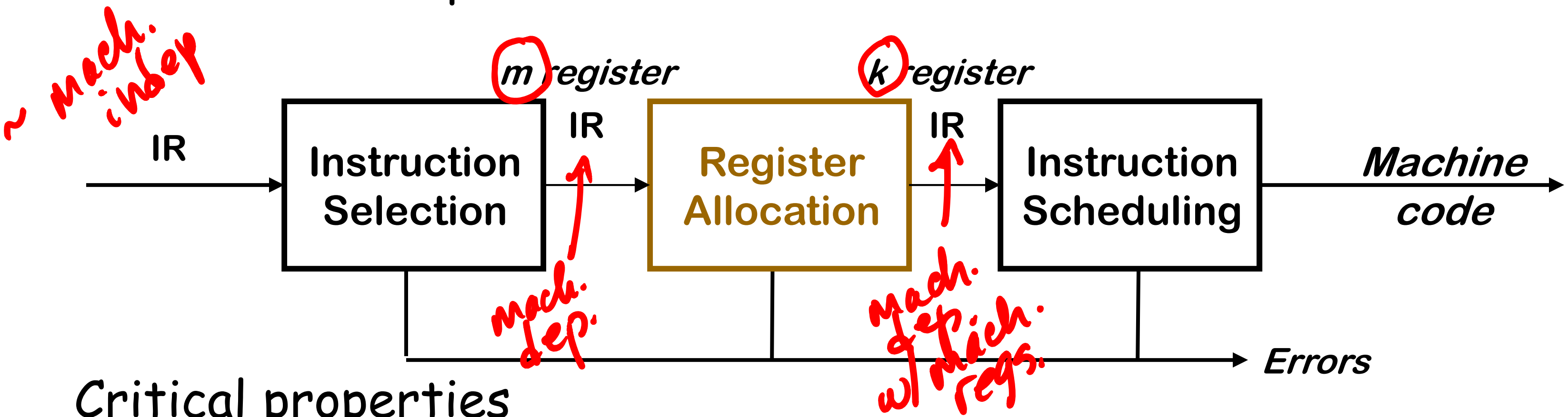
Intraprocedural ~~Global~~ Register Allocation via Graph Coloring

Register Allocation

Part of the compiler's back end



Critical properties

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

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

Global Register Allocation

The big picture

big *m* *register code* → **Register Allocator** → *k* *register code*

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

At each point in the code

1 Determine which values will reside in registers Allocation

2 Select a register for each such value Assignment

The goal is an allocation that "minimizes" running time

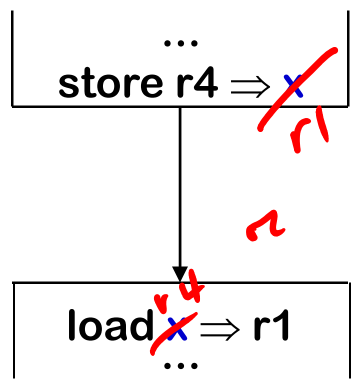
Most modern, global allocators use a graph-coloring paradigm

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

K-coloring

v_1, v_2 both live at same time anywhere, then must not receive same register → $v_1 + v_2$ conflict

Global Register Allocation

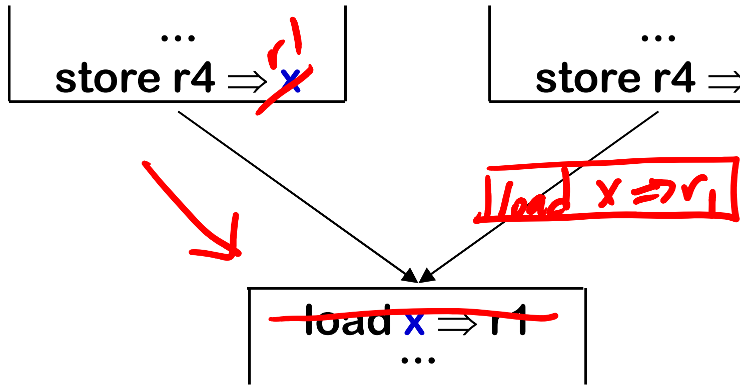


This is an assignment problem, not an allocation problem!

What's harder across multiple blocks?

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

Global Register Allocation



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

A more complex scenario

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

This adds tremendous complications

Global Register Allocation

Taking a global approach

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

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

1 Build an interference graph G_I for the procedure
   - → Computing LIVE is harder than in the local case (yes, but we've seen this)
   - → G_I is not an interval graph

2 (Try to) construct a k-coloring
   - → Minimal coloring is NP-Complete
   - → Spill placement becomes a critical issue

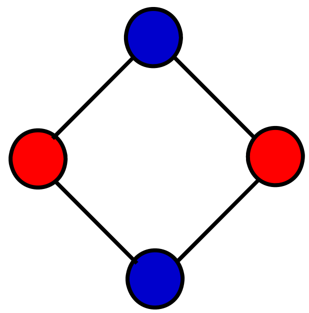
3 Map colors onto physical registers

Graph Coloring (A Background Digression)

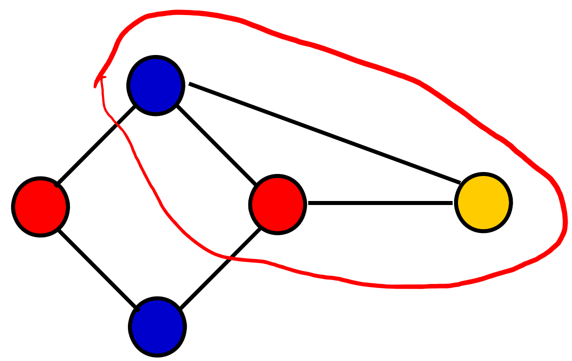
The problem

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

Examples



2-colorable



3-colorable

Each color can be mapped to a distinct physical register

Building the Interference Graph

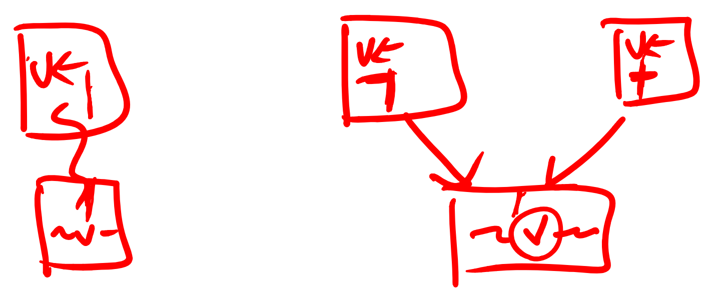
What is an "interference" ? (or conflict)

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

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

The interference graph, G_I

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



Building the Interference Graph

To build the interference graph

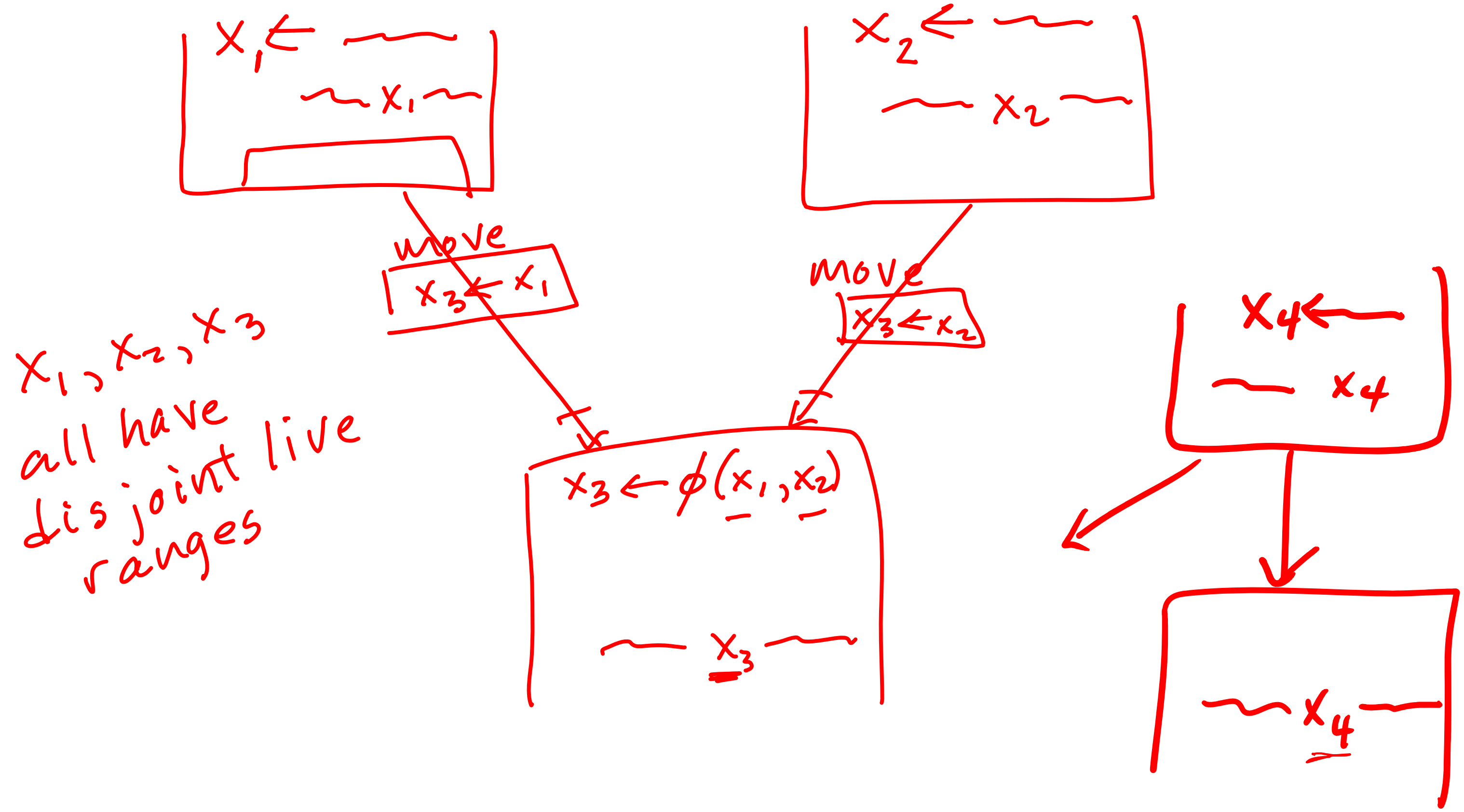
1 Discover live ranges
  › Build **SSA** form [Eliot's digression to explain …]
  › At each ϕ-function, take the union of the arguments

2 Compute **LIVE** sets for each block
  › Use an iterative data-flow solver
  › Solve equations for **LIVE** over domain of live range names

3 Iterate over each block (note: *backwards* flow problem)
  › Track the current **LIVE** set
  › At each operation, add appropriate edges & update **LIVE**
    ♦ Edge from result to each value in **LIVE**
    ♦ Remove result from **LIVE**
    ♦ Edge from each operand to each value in **LIVE**

Eliot's Digression about SSA

- SSA = Static Single Assignment form



What is a Live Range?

- A set LR of definitions $\{d_1, d_2, \ldots, d_n\}$ such that for any two definitions d_i and d_j in LR, there exists some use u that is reached by both d_i and d_j .
- How can we compute live ranges?
  - → For each basic block b in the program, compute REACHESOUT(b) — the set of definitions that reach the exit of basic block b
    - ♦ $d \in$ REACHESOUT(b) if there is no other definition on some path from d to the end of block b
  - → For each basic block b, compute LIVEIN(b)—the set of variables that are live on entry to b
    - ♦ $v \in$ LIVEIN(b) if there is a path from the entry of b to a use of v that contains no definition of v
  - → At each join point b in the **CFG**, for each live variable v (i.e., $v \in$ LIVEIN(b)), merge the live ranges associated with definitions in REACHESOUT(p), for all predecessors p of b, that assign a value to v.

Computing LIVE Sets

A value v is live at p iff

$\exists$ a path from *p* to some use of *v* along which *v* is not re-defined

Data-flow problems are expressed as simultaneous equations

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

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

where

UEVAR(b) is the set of upward-exposed variables in b
(names used before redefinition in block b)
VARKILL(b) is the set of variable names redefined in b

As output,

LIVEOUT(x) is the set of names live on exit from block x
LIVEIN(x) is the set of names live on entry to block x

solve it with the iterative algorithm

Observation on Coloring for Register Allocation

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

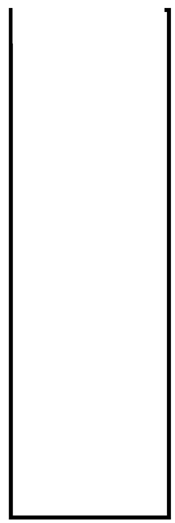
Chaitin's Algorithm

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

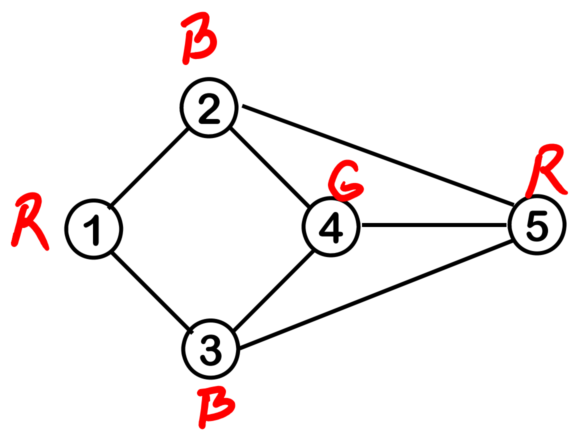
Chaitin's Algorithm in Practice

3 Registers

RGB

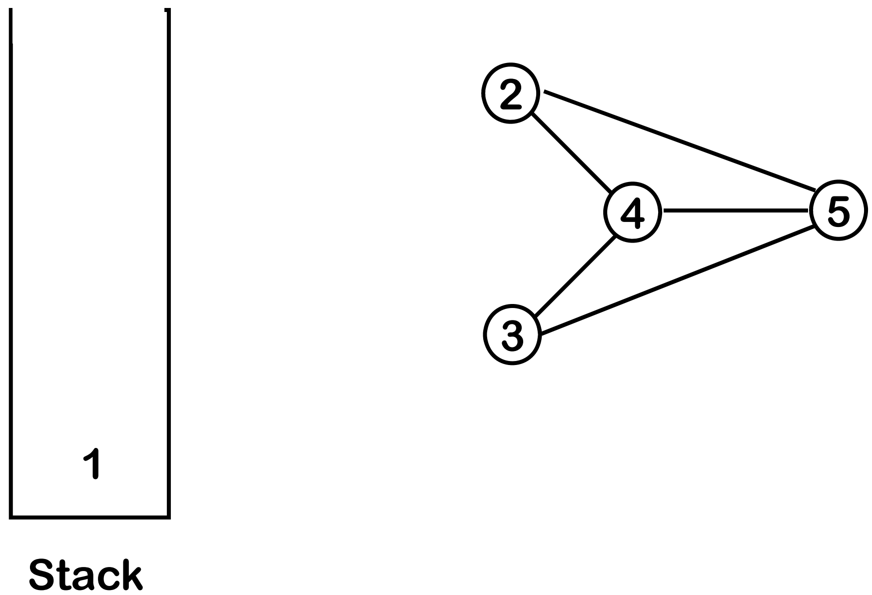


Stack



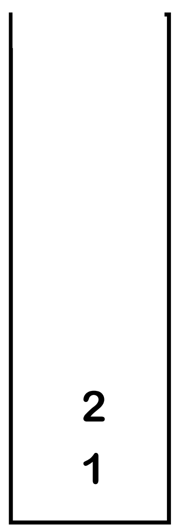
Chaitin's Algorithm in Practice

3 Registers

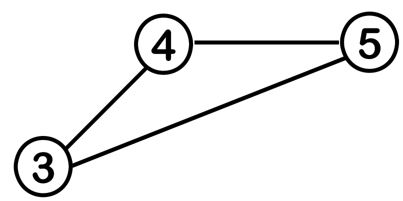


Chaitin's Algorithm in Practice

3 Registers

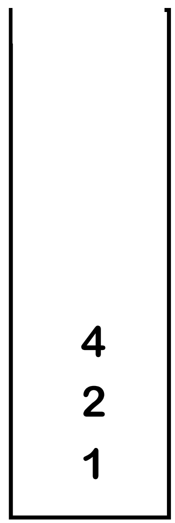


Stack

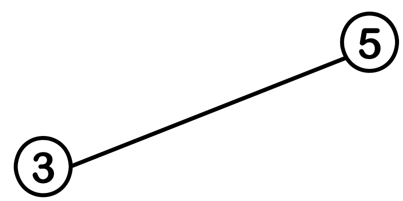


Chaitin's Algorithm in Practice

3 Registers

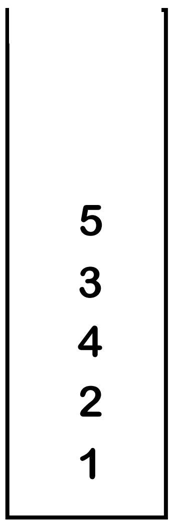


Stack



Chaitin's Algorithm in Practice

3 Registers



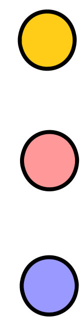
Stack

Colors:

1:

2:

3:

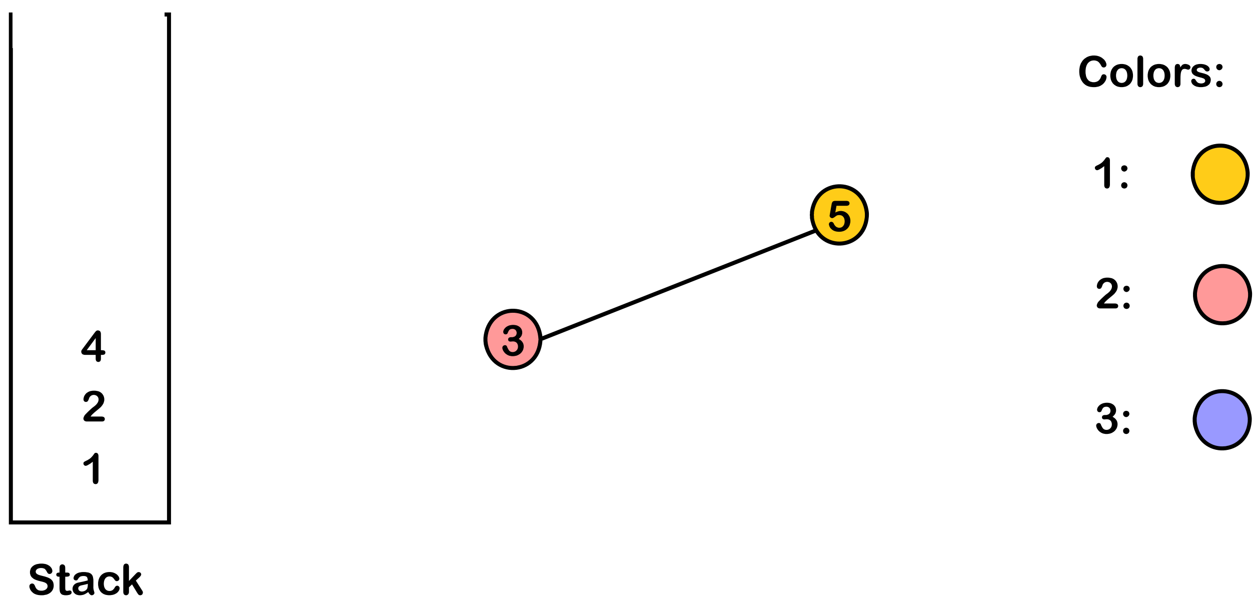


Chaitin's Algorithm in Practice

3 Registers

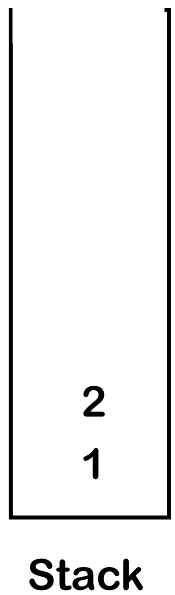
Chaitin's Algorithm in Practice

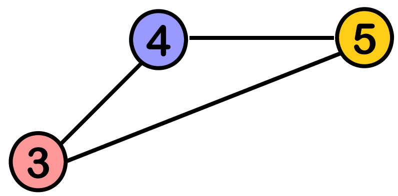
3 Registers

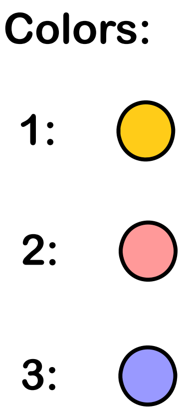


Chaitin's Algorithm in Practice

3 Registers

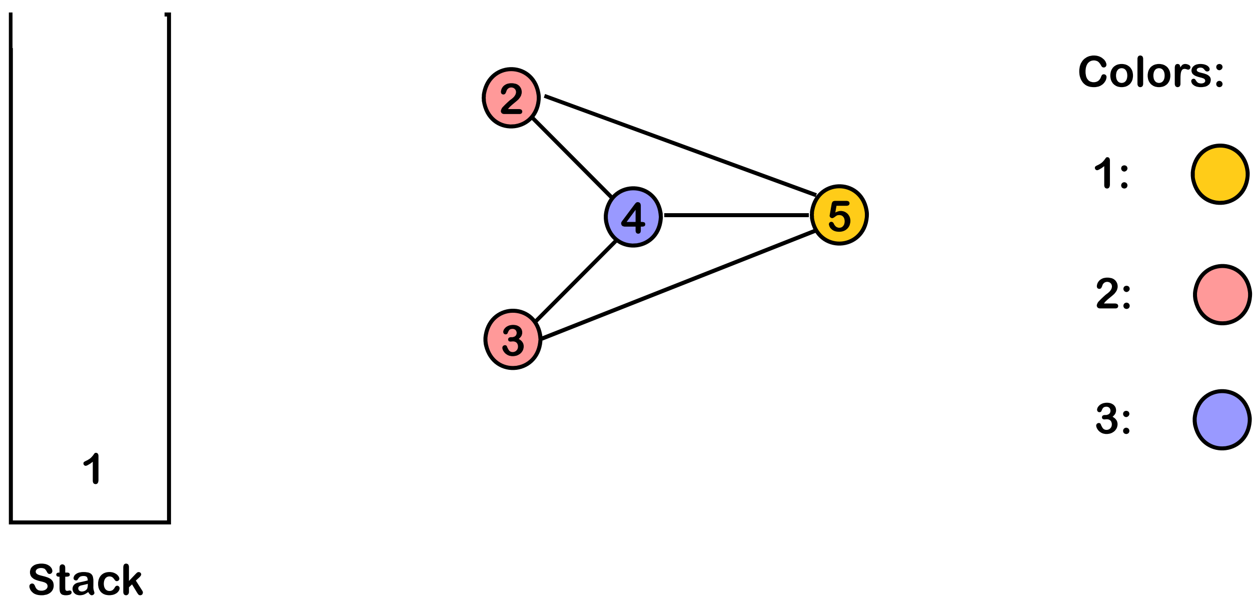






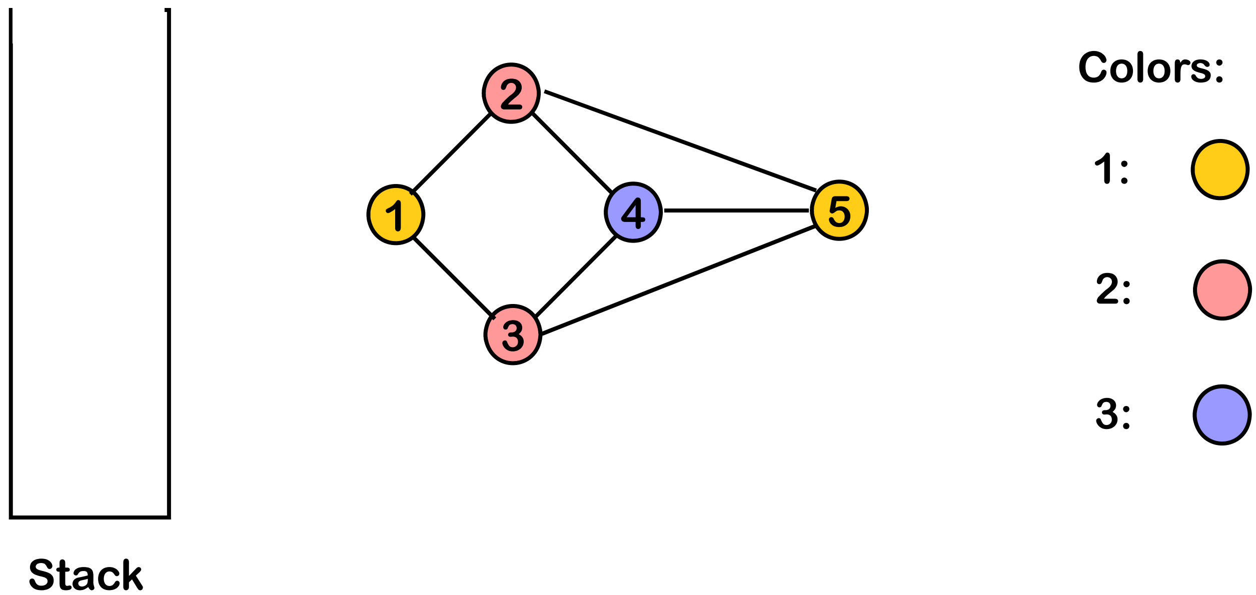
Chaitin's Algorithm in Practice

3 Registers



Chaitin's Algorithm in Practice

3 Registers

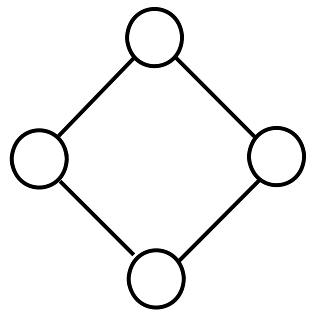


Improvement in Coloring Scheme

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

- Instead of stopping at the end when all vertices have at least k neighbors, put each on the stack according to some priority
  - → When you pop them off they may still color!

2 Registers:

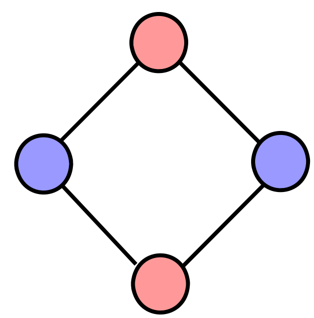


Improvement in Coloring Scheme

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

- Instead of stopping at the end when all vertices have at least k neighbors, put each on the stack according to some priority
  - → When you pop them off they may still color!

2 Registers:

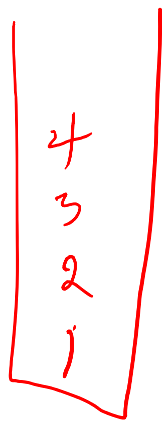


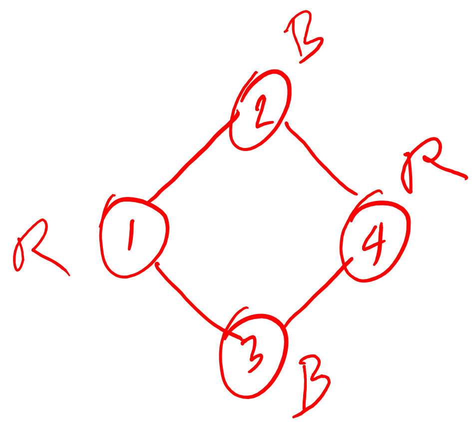
2-colorable

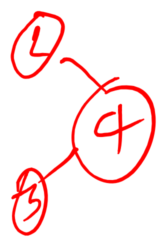
Chaitin-Briggs Algorithm

1. While ∃ vertices with < *k* neighbors in G_I
   > Pick any vertex *n* such that $n^\circ < k$ and put it on the stack
   > Remove that vertex and all edges incident to it from G_I
   - This may create vertices with fewer than *k* neighbors
2. If G_I is non-empty (all vertices have *k* or more neighbors) then:
   > Pick a vertex *n* (using some heuristic condition), push *n* on the stack and remove *n* from G_I, along with all edges incident to it
   > If this causes some vertex in G_I to have fewer than *k* neighbors, then go to step 1; otherwise, repeat step 2
3. Successively pop vertices off the stack and color them in the lowest color not used by some neighbor
   > If some vertex cannot be colored, then pick an uncolored vertex to spill, spill it, and restart at step 1

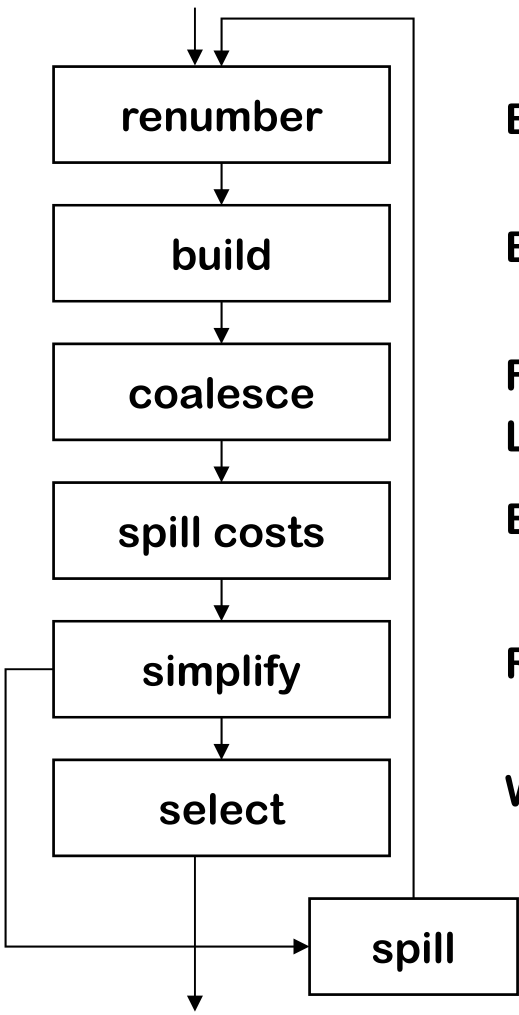
Working the 4-node example







Chaitin Allocator (Bottom-up Coloring)



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

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

Estimate cost for spilling
each live range

Remove nodes from the graph

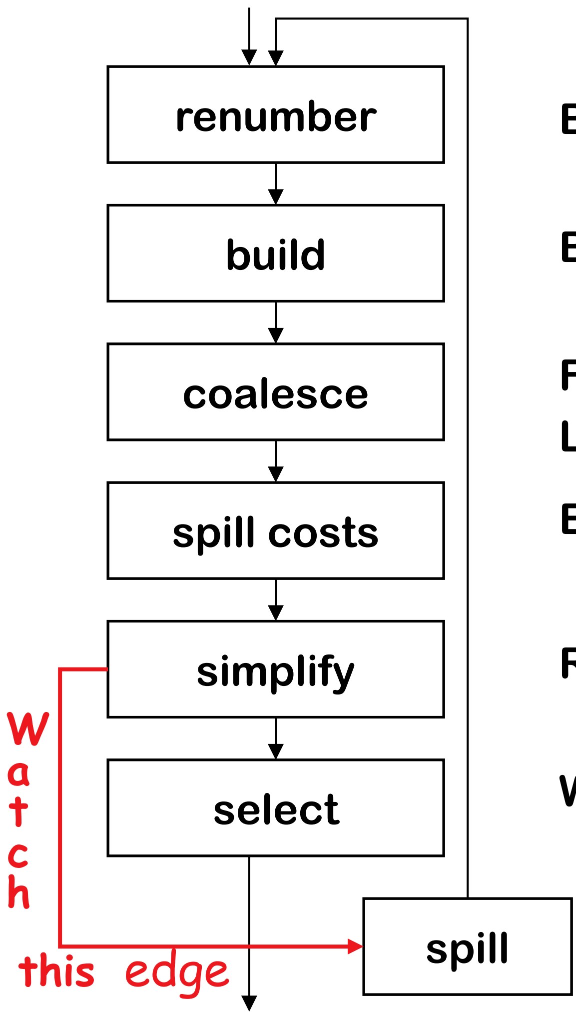
while *N* is non-empty
  if $\exists$ *n* with $n^{\circ} < k$ then
    push *n* onto stack
  else pick *n* to spill
    push *n* onto stack
  remove *n* from G_I

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

Spill uncolored definitions & uses

Chaitin's algorithm

Chaitin Allocator (Bottom-up Coloring)



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

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

Estimate cost for spilling
each live range

Remove nodes from the graph

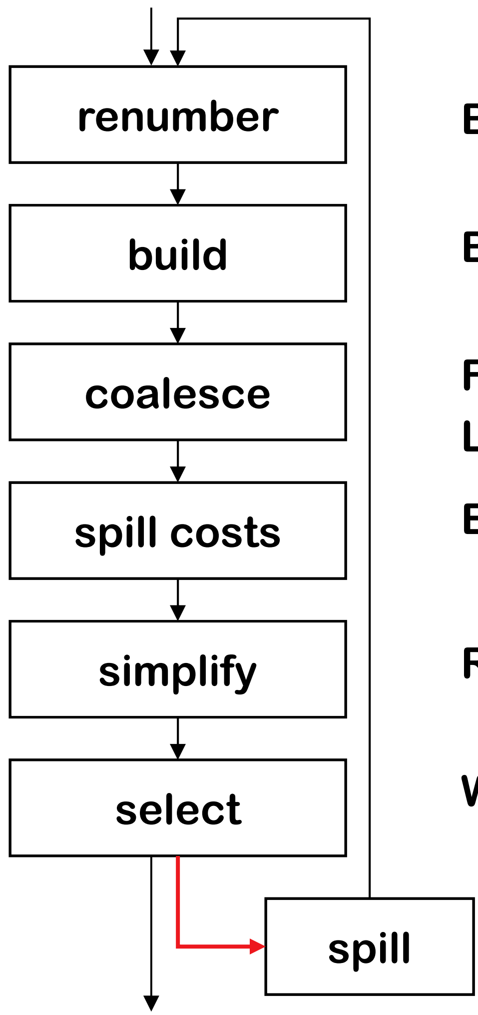
> while *N* is non-empty
> if $\exists$ *n* with $n^\circ < k$ then
> push *n* onto stack
> else pick *n* to spill
> push *n* onto stack
> remove *n* from G_I

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

Spill uncolored definitions & uses

Chaitin's algorithm

Chaitin-Briggs Allocator (Bottom-up Coloring)



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

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

Estimate cost for spilling
each live range

Remove nodes from the graph

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

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

Spill uncolored definitions & uses

Briggs' algorithm (1989)

Picking a Spill Candidate

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

Chaitin's heuristic

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

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

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

Other Improvements to Chaitin-Briggs

Spilling partial live ranges

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

Splitting live ranges

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

Conservative coalescing

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

Chaitin-Briggs Allocator (Bottom-up Global)

Strengths & weaknesses

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

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

Is improvement still possible ?

- Rising spill costs, aggressive transformations, & long blocks
  $\Rightarrow$ yes, it is

What about Top-down Coloring?

Use spill costs as priority function !

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

Unconstrained must receive a color !

The Details

- Separate constrained from unconstrained live ranges
  - A live range is constrained if it has $\geq k$ neighbors in G_I
- Color constrained live ranges first
- Reserve pool of local registers for spilling (or spill & iterate)
- Chow split live ranges before spilling them
  - Split into block-sized pieces
  - Recombine as long as $^\circ < k$

What about 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

More Details

- Chow used an imprecise interference graph
  - → $\langle x,y \rangle \in G_I \Leftrightarrow x,y \in$ LiveIn(b) for some block b
  - → Cannot coalesce live ranges since $x \rightarrow y \Rightarrow \langle x,y \rangle \in G_I$
- Quicker to build imprecise graph
  - → Chow's allocator runs faster on small codes, where demand for registers is also likely to be lower (*rationalization*)

Tradeoffs in Global Allocator Design

Top-down versus bottom-up

- Top-down uses high-level information
- Bottom-up uses low-level structural information

Spilling

- Reserve registers versus iterative coloring

Precise versus imprecise graph

- Precision allows coalescing
- Imprecision speeds up graph construction

Even JITs use this stuff …

Regional Approaches to Allocation

Hierarchical Register Allocation (Koblenz & Callahan)

- Analyze control-flow graph to find hierarchy of tiles
- Perform allocation on individual tiles, innermost to outermost
- Use summary of tile to allocate surrounding tile
- Insert compensation code at tile boundaries ($LR_x \rightarrow LR_y$)

Strengths

→ Decisions are largely local

→ Use specialized methods on individual tiles

→ Allocator runs in parallel

Weaknesses

→ Decisions are made on local information

→ May insert too many copies

Still, a promising idea

- Anecdotes suggest it is fairly effective
- Target machine is multi-threaded multiprocessor (Tera MTA)

Regional Approaches to Allocation

Probabilistic Register Allocation (Proebsting & Fischer)

- Attempt to generalize from Best's algorithm *(bottom-up, local)*
- Generalizes "furthest next use" to a probability
- Perform an initial local allocation using estimated probabilities
- Follow this with a global phase
  - → Compute a merit score for each **LR** as
    (benefit from x in a register = probability it stays in a register)
  - → Allocate registers to **LR**s in priority order, by merit score, working from inner loops to outer loops
  - → Use coloring to perform assignment among allocated **LR**s

- Little direct experience (either anecdotal or experimental)
- Combines top-down global with bottom-up local

Regional Approaches to Allocation

Register Allocation via Fusion (Lueh, Adl-Tabatabi, Gross)

- Use regional information to drive global allocation
- Partition CFGs into regions & build interference graphs
- Ensure that each region is *k*-colorable
- Merge regions by fusing them along CFG edges
  - → Maintain *k*-colorability by splitting along fused edge
  - → Fuse in priority order computed during the graph partition
- Assign registers using int. graphs

***i.e.*, execution frequency**

Strengths
- **Flexibility**
- **Fusion operator splits on low-frequency edges**

Weaknesses
- **Choice of regions is critical**
- **Breaks down if region connections have many live values**