Register Allocation

We now are able to do register allocation on our interference graph.

We want to deal with two types of constraints:

1. Two values are alive at overlapping points (interference graph)
2. A value must or must not be in a particular architectural register

Register Allocation is comprised of two parts:

- **Register allocation** in which we find a set of virtual registers that could be placed in the same physical location
- **Register assignment** in which we actually decide which architectural register to put a set of non-interfering values

Register Allocation

Register Allocation reduces to a problem known as Graph Coloring, where we try to "color" a graph without using the same color on two neighboring territories.

If we consider the set of colors to be our registers and our territories the virtual registers, it's clearly the problem we wish to solve.

However, it is NP-complete, which means it is (likely) intractable without enumeration of all possible combinations. Or at least, we don't know how to do it otherwise.

The good news is that we can use **heuristics** (informed rules of guessing) to make the problem practically solvable in reasonable time with a good enough answer.

Principle Phases

Build

- Build the interference graph from the live ranges of our virtual register set

Simplify

- Use a heuristic to attempt to color the graph
- We'll use the heuristic that if our interference graph has a node *m* of degree (amount of neighbors) < *K* (our register set size), we can remove that node from the graph.
- If the resulting graph can be *K* colored, so can the original graph, since some color can be found to color node *m*.
- This results in a recursive algorithm. Each simplification step makes the problem smaller and easier to solve.

Principle Phases

Spill

- Unfortunately, all of the remaining nodes may have ≥ K neighbors even after simplification.
- These nodes are of *significant degree*
- The decision might need to be that we move an item to memory and keep it there. We call this a **spill**.
- Any spilled value does not interfere with any of the nodes in the graph, and so it may be removed and the procedure continues simplifying

Select

- Assign colors to nodes in the graph by starting with an empty graph and adding nodes from our simplify stack.
- There must always be a color to assign due to our heuristic.
- We now determine potential spills from actual spills. We might be lucky and get a node that has 2 or more neighbors that are the same color, and thus we can color the node. This is *optimistic coloring*.

Spills

If the select phase is unable to find a color for a potential spill, it becomes an actual spill.

Actual spills must be stored in memory.

This means that we are changing the program code to load the value from memory into a register before we use it.

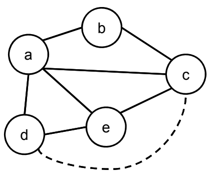
Since we need a register to load the memory value into, our live ranges have changed:

- We now have new registers with tiny live ranges which potentially interfere with our coloring.

We must start all over with the rewritten (spills incorporated) program code.

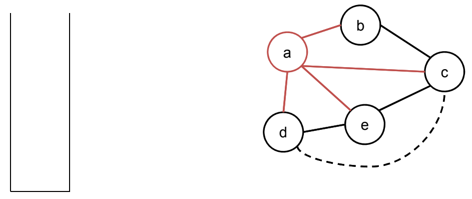
Example: Build

```
a := 0
b := mem[a]
c := b * 2
c := c + b
d := c
e := a * c
e := e + d
a := a + 1
```



The dashed line indicates a copy, not an interference.

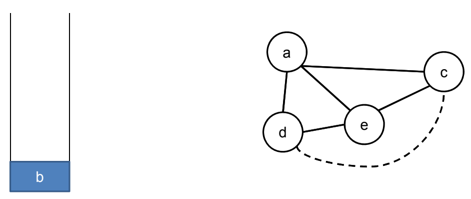
Example: Simplify



Start with our interference graph, an empty stack, and K=3 colors.

Variable ‘a’ seems worrying since it is of *significant degree* (i.e., it has a degree ≥ 3).

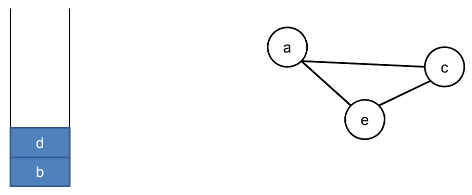
Example: Simplify



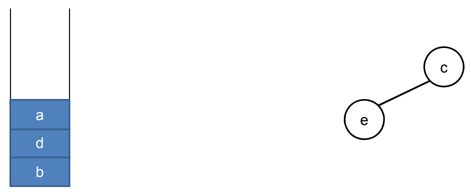
Remove a node with degree < K and push it on the stack.

The removed node’s edges will decrease the degree of the remaining nodes.

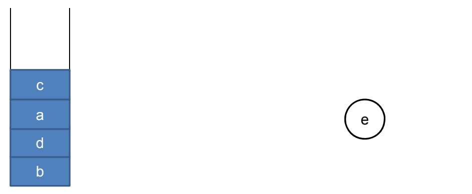
Example: Simplify



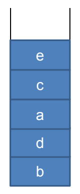
Example: Simplify



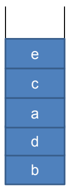
Example: Simplify



Example: Simplify

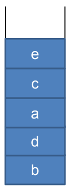


Example: Simplify



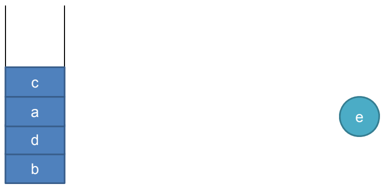
We found no potential spills, so we can skip that step and go straight to the select phase.

Example: Select

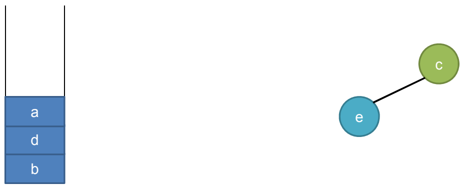


Starting from an empty graph, pop off each node and give it a color.

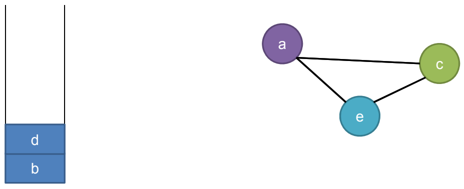
Example: Select



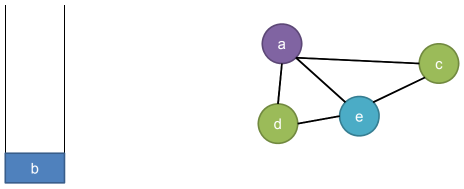
Example: Select



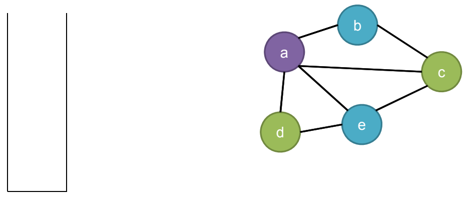
Example: Select



Example: Select



Example: Select



Done. We have 3-colored our interference graph.

Coalescing

We can remove redundant moves if the source and destinations have no edge between them in the interference graph.

We can then simply tell the register allocator to allocate both the source and destination into the same register by **coalescing** both nodes into a single node in the graph.

However, since this node is the union of the two nodes being coalesced, the possibility exists of preventing the resulting graph from being K-colorable when the previous graph was.

Conservatively, we can do a safe coalescing algorithm that can guarantee we will never produce an uncolorable graph from a colorable one.

Briggs Coalescing

Nodes *a* and *b* can be coalesced if the resulting node, *ab*, will have fewer than K nodes of significant degree.

This works because simplify removed all nodes of insignificant degree.

This means that *ab* will only be adjacent to the nodes of significant degree.

If there are less than K neighbors of significant degree, then simplify can remove this node and coloring can proceed.

George Coalescing

Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either:

- *t* already interferes with *b*, or
- *t* is of insignificant degree

Proof:
Let *S* be the set of neighbors of *a* with insignificant degree.

If coalescing is not done, Simplify can remove all of *S*, leaving graph G_1
If coalescing is done, Simplify can still remove all of *S*, leaving graph G_2

G_2 is a subgraph of G_1 since node *ab* corresponds to the node *b* in G_1.

Thus, G_2 must be as easy to color as G_1.

Coalescing

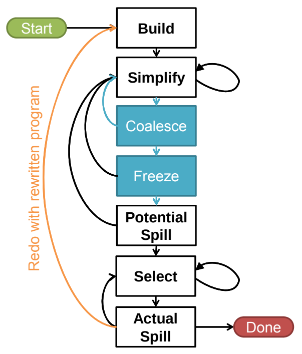
These are safe, but conservative coalescing schemes, so they may miss opportunities to coalesce moves.

However, an extra move is less costly than any additional spills.

A Coalesce phase will be added to our algorithm from before, and our Simply routine will be interleaved with it: we simplify and coalesce until our graph is empty.

This will eliminate most move instructions but still be colorable.

Register Coloring Algorithm



RegAlloc with Coalescing

Build

- Build the interference graph
- Classify each node as *move-related* or not
- Move-related means that it is the destination or source of a move

Simplify

- Remove each non-move related node of low degree (<K) one at a time

Coalesce

- Perform one of the conservative algorithms we have shown on the resulting simplified graph
- Eliminate the associated move instruction
- If the resulting node is no longer move-related, it can be simplified in the next round
- Simplify and Coalesce are repeated until only *significant-degree* or *move-related* nodes remain.

RegAlloc with Coalescing

Freeze

- If we cannot simplify or coalesce, we look for a move-related node of low degree
- We *freeze* such a node, that is, we give up trying to coalesce it
- Mark it as non-move-related and go back to doing Simplify and Coalesce

Spill

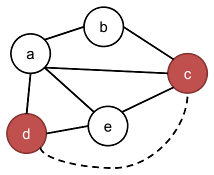
- If we can't simplify, coalesce, or freeze and only significant degree nodes remain, select a significant degree node to potentially spill
- Push it on the stack

Select

- Pop each element off the stack and assign it a color

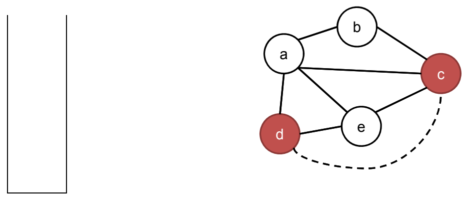
Example: Build

```
a := 0
b := mem[a]
c := b * 2
c := c + b
d := c
e := a * c
e := e + d
a := a + 1
```



There are two move-related nodes, c and d.

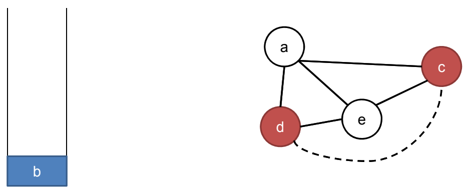
Example: Simplify



We will simplify only those nodes that are of non-significant degree and non-move-related.

For K=3, that's only b.

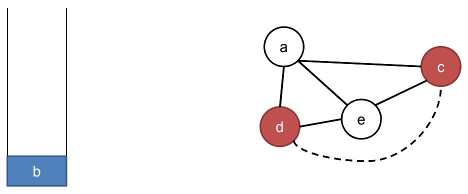
Example: Simplify



We will simplify only those nodes that are of non-significant degree and non-move-related.

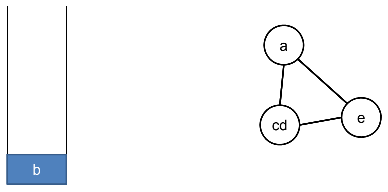
For K=3, that's only b.

Example: Coalesce



c and d only have two neighbors of significant degree and so they may be safely coalesced.

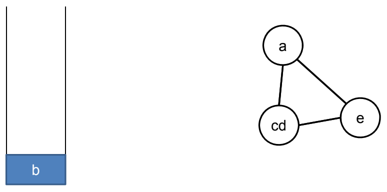
Example: Coalesce



c and d only have two neighbors of significant degree and so they may be safely coalesced.

We remove the move instruction and mark cd as non-move-related.

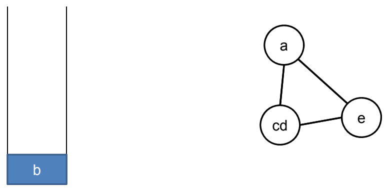
Example: Coalesce



c and d only have two neighbors of significant degree and so they may be safely coalesced.

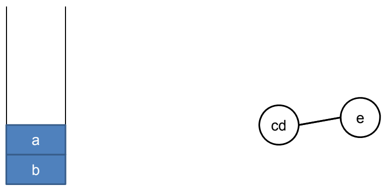
We remove the move instruction and mark cd as non-move-related.

Example: Simplify

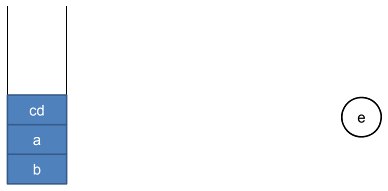


We now have nodes that have non-significant degree that we can simplify.

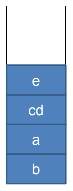
Example: Simplify



Example: Simplify



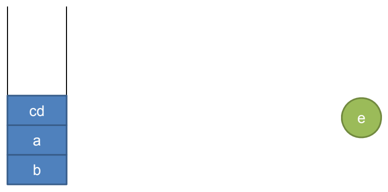
Example: Simplify



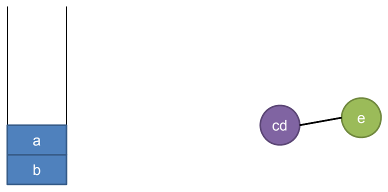
With an empty graph, we have no need for the Freeze or Spill steps.

We now rebuild the graph assigning colors as we go.

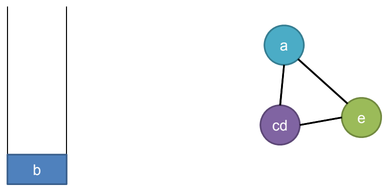
Example: Select



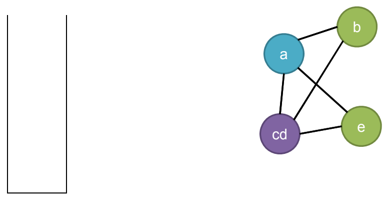
Example: Select



Example: Select

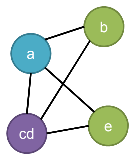


Example: Select



Example: Result

```
a := 0
b := mem[a]
c := b * 2
c := c + b
e := a * c
e := e + d
a := a + 1
```



With the coalescing of c and d, we have rewritten the code to remove the move and c and d will both be assigned to the same register.

Constrained Moves

Sometimes there may be multiple choices for coalescing moves.

For instance, we may have three variables, a, b, and c and two moves:

```
a := b
b := c
```

If (a, c) is an edge in our interference graph, coalescing a and b into ab results in:

```
ab := c
```

Which has an interference edge between them. This move is **constrained** and cannot be coalesced.

We treat it as non-move-related and is no longer considered for coalescing.

Spilling

When we have to spill, Build and Simplify must be rerun on the program.

With coalescing, we'd throw away work that doesn't increase the number of spills.

A better algorithm can keep all coalesces done before the first potential spill was discovered.

However, even more can be done. If we have a very constrained machine, such as x86 with its 6 general purpose registers, we may end up spilling many temporaries into the stack frame.

Moves of spilled values involve loading the source into a register, creating a new live range and further constraining the results.

Spill Coalescing

We can actually do graph coloring on the spilled locations because they may not be alive either.

It's actually easier to do because we are not constrained by the number of activation record locations, so we can coalesce aggressively.

The algorithm is:

1. Construct interference for spilled nodes
2. Coalesce all non-interfering spilled nodes connected via a move
3. Use Simplify and Select to color the graph. There is no limit on the number of colors, so Simplify can pick the lowest degree node and Select can pick the first available color
4. Use the colors to assign stack locations to the spills

Do this before generating spill instructions to avoid unnecessary load/stores for coalesced moves of spills.

Precoloring

Sometimes we have constraints on the machine registers that hold specific values. For instance, the stack pointer, the frame pointer, the argument registers, etc.

We can assign these registers a color before we run our algorithm so that we can bind a value to that register. This is called **precoloring** and there must be at most one precolored node of a given color.

Select and Coalesce can assign nodes the same color as precolored nodes as long as they do not interfere. For instance, $a3 in MIPS could be used to hold a temporary in a function that does not take 4 parameters.

When we have K registers, we can add K precolored nodes that all interfere with each other, and thus cannot be coalesced. They should not be spilled, since they are, by definition, registers. They also cannot be simplified, we must treat them as having infinite degree.

Precolored Nodes

The algorithm now calls Simplify, Coalesce, and Spill until only the precolored nodes remain. Then Select will add the other nodes back in while coloring them.

Precolored nodes do not spill and thus their live ranges should be kept short.

This can be done by generating moves for values into and out of precolored nodes.

Only if there is considerable **register pressure** – high demand for registers – will the move remain after coalescing.

Caller- and Callee-saved Registers

The constraints of caller-saved and callee-saved registers can be enforced by augmenting the call instructions of our IR.

A call instruction can be annotated to interfere with all of the caller-save registers. If a variable is not alive across the procedure call, it will most likely be allocated to a caller-save register.

If a variable is alive across a call, it must then interfere with all of the caller-saved registers and it will interfere with the temporaries created for callee-saved registers. This will result in a spill, saving the value to memory.

We can estimate spill cost to determine which value is best put in memory. A basic heuristic would spill a node with high degree in the interference graph but few uses.