CA4003 - Compiler Construction

Register Allocation

David Sinclair

Registers and Temporaries

While generating the Intermediate Representation we have assumed that we can have an infinite number of temporaries.

During the Code Optimisation phase we have used an number of techniques, such as common subexpressions, dead code elimination, etc., to reduce the number of temporaries.

The remaining variables and temporaries are then assigned to registers and memory. Accessing registers is faster than accessing memory. Many architectures limit operations to registers, and even when operations can be preformed on memory, they are significantly slower that operations involving registers.

If temporaries t_i and t_j are not live at the same time, then we could use the same register for both t_i and t_j.

Colouring by Simplification

In general *graph colouring*, and hence *register allocation*, are NP-complete. However *Colouring by Simplification* is a linear-time approximation which iterates over 4 phases:

1. **Build** an *interference graph* where each node represents a variable or temporary and an edge exists between 2 nodes if the are both *live* at the same time.
2. **Simplify** removes a node with $< K$ neighbours (where K is the number of registers) and places it on a stack.
3. **Spill** nodes with $\geq K$ neighbours. We'll use *optimistic colouring* and assume that *potential spill* nodes do not interfere with other graph nodes. If later we are wrong they will become *actual spill nodes* and stored in memory.
4. **Select** a colour (register) for each popped node so that no neighbouring node has the same colour. If we can't select a colour it is an *actual spill* node. Rewrite the code to use memory for that variable/temporary and re-run the algorithm.

Colouring by Simplification - Build

Consider the following IR code fragment

1	$t_1 = j + 12$
2	$g = a[t_1]$
3	$h = k - 1$
4	$f = g * h$
5	$t_2 = j + 8$
6	$e = a[t_2]$
7	$t_3 = j + 16$
8	$m = a[t_3]$
9	$b = a[f]$
10	$c = e + 8$
11	$d = c$
12	$k = m + 4$
13	$j = b$

Assuming d, k and j are *live out*, then a *liveness* analysis yields:

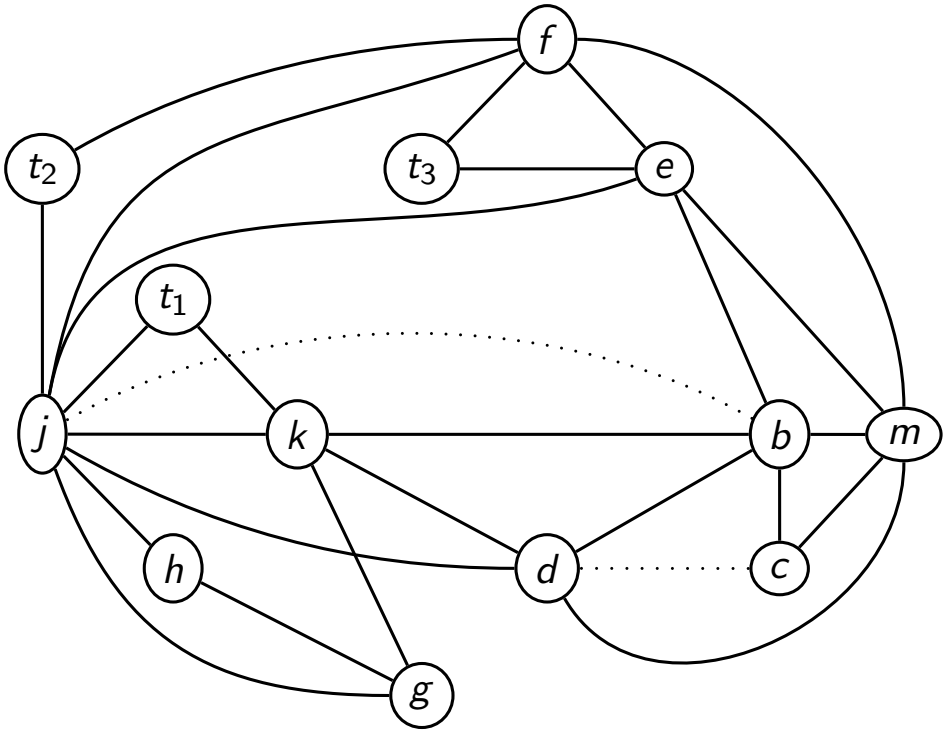
$out = \bigcup_{S \in succ(B)} in$

$in = use \cup (out - def)$

	use	def	out	in
13	b	j	d, j, k	b, d, k
12	m	k	b, d, k	m, b, d
11	c	d	m, b, d	c, m, b
10	e	c	c, m, b	e, m, b
9	f	b	e, m, b	f, e, m
8	t_3	m	f, e, m	t_3, f, e
7	j	t_3	t_3, f, e	j, f, e
6	t_2	e	j, f, e	t_2, j, f
5	j	t_2	t_2, j, f	j, f
4	g, h	f	j, f	g, h, j
3	k	h	g, h, j	k, g, j
2	t_1	g	k, g, j	t_1, k, j
1	j	t_1	t_1, k, j	k, j

Colouring by Simplification - Build (2)

The corresponding *interference graph* is:

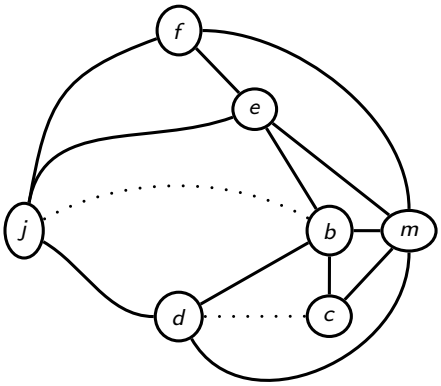


Dotted line represents move instructions.

Colouring by Simplification - Simplify

During the *simplify* phase we select any node in the *interference graph* with $< K$ edges, where K is the number of registers. We remove this node and its edges from the graph, and push it onto a stack. Removing a node and its edges may result in a node that previously had $\geq K$ edges, now having $< K$ edges. Repeat until there are no more node with $< K$ edges.

In our example, assuming $K = 4$, we can remove initially remove t_1, t_2, t_3, h, g and k, leaving the following graph and stack.



k
g
h
t_3
t_2
t_1

Colouring by Simplification - Spill

After the *simplify* phase has dealt with all the nodes with $< K$ edges, there are only *significant nodes*, $\geq K$ edges left. These may be a problem and may need to be *spilled*, i.e. stored in memory, if all the neighbouring nodes use K different colours.

We will adopt an *optimistic* approach and assume that any *potential spill* node can be coloured and push it onto the stack. If it can't, we will deal with it in the *select* phase.

Colouring by Simplification - Select

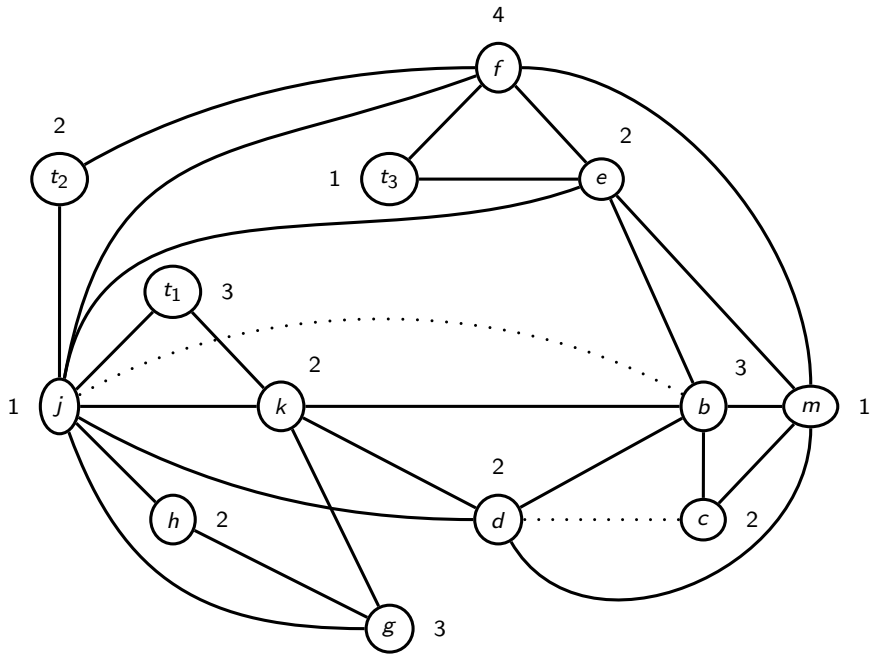
In the *select* phase, a node x is popped from the stack and the graph is rebuilt after assigning a colour (register) to the node. The colour is assigned arbitrarily but only limitation is that you cannot use a colour of a neighbouring node, a node connected to x by an edge, while rebuilding the graph.

Sometimes, when processing a *potential spill* node it may not be possible to assign it a colour. In this case a *potential spill* node becomes an *actual spill* node and the code must be rewritten to retrieve the variable from memory before using it and storing back to memory immediately after its use. Then the whole *Build-Simplify-Spill-Select* process is repeated.

Colouring by Simplification - Select (2)

In our example we could generate the following stack, which could result in the following colouring assuming 4 registers.

m
c
b
f
e
j
d
k
g
h
t_3
t_2
t_1



Colouring by Coalescing

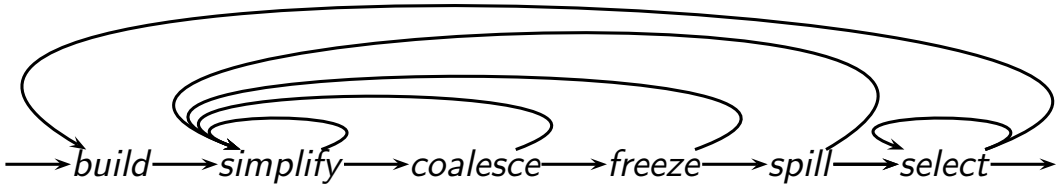
If two nodes are *move-related* but do not interfere with each other (i.e. no edge between them), then they can be *coalesced* into one node whose edges are the union of the edges of the individual nodes. This eliminates the move instruction.

Any two non-interfering nodes can be coalesced, but the resulting node has more edges and may become an *actual spill* node. We need a *conservative strategy* to *coalesce* nodes that will not transform a K-colourable interference graph into a K-noncolourable interference graph.

Once such strategy in the *Briggs* strategy where nodes x and y can be coalesced if the resulting node xy will have less that K neighbours of *significant degree* ($\geq K$ edges).

Colouring by Coalescing (2)

The *colouring by coalescing* algorithm is outlined below.

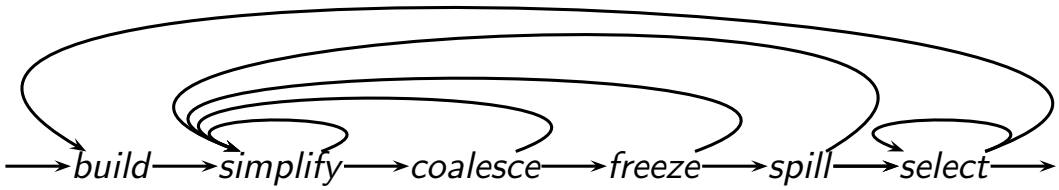


Build Build an interference graph as before but this time mark each node as *non-move-related* or *move-related* if the node was the source or destination of a move (copy) instruction.

Simplify Simplify *non-move-related* nodes with $< K$ edges.

Coalesce Conservatively calesce *move-related* nodes. The resulting nodes are *non-move-related* and are available for further *simplify* phases.

Colouring by Coalescing (3)



Freeze If neither *simplify* nor *coalesce* can be applied, select a low degree *move-related* node and *freeze* it. This causes the node to be considered as a *non-move-related* node. Essentially we have give up any hope of coalescing it. The node is now available for further *simplify* and *coalesce* phases.

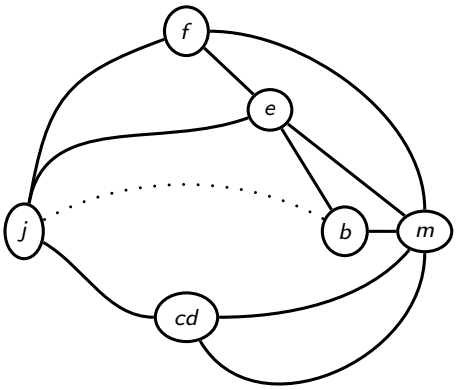
Spill If there are no low degree nodes, we select a significant degree node as a *potential spill* node and push it onto the stack.

Select As before, pop nodes from the stack, assign colour or make an *actual spill* node.

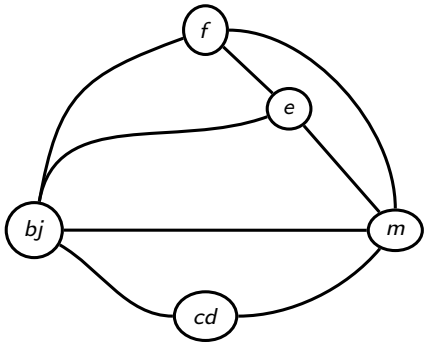
Colouring by Coalescing (4)

If we apply *colouring by coalescing* to the previous example, we can simplify nodes t_1, t_2, t_3, h, g and k as these are *non-move-related* nodes of low degree.

We can *coalesce* nodes c and d as the resulting node cd will only have 2 neighbours of significant degree (nodes b and m).



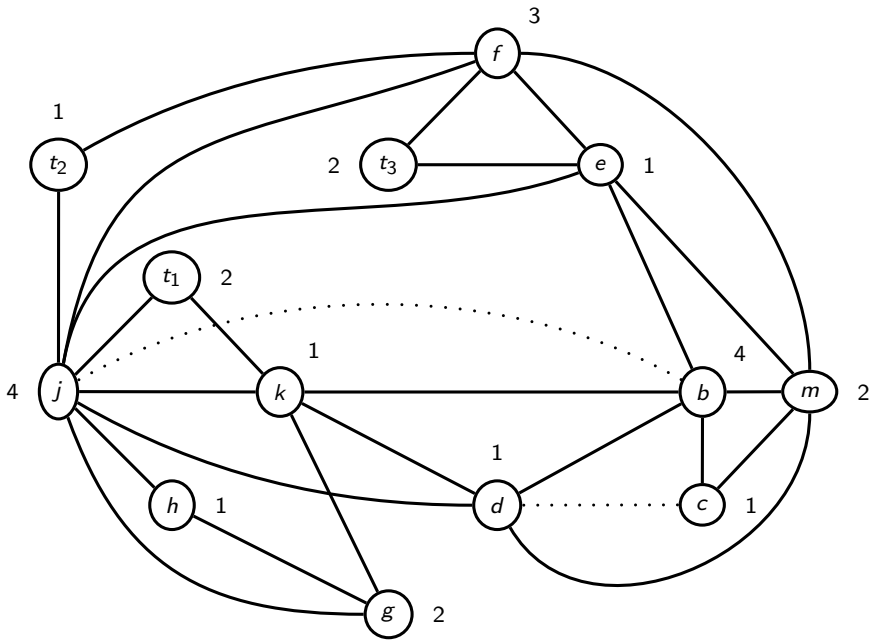
Coalescing nodes b and j yields:



Colouring by Coalescing (5)

This could generate the following stack, which could result in the following colouring assuming 4 registers.

e
m
f
bj
cd
k
h
g
t_3
t_2
t_1



Precoloured Nodes

Machine registers (e.g. stack pointer, frame pointer, arguments, return address, return value) are assigned to precoloured nodes.

- The *select* and *coalesce* phases can give nodes the same colour as a precoloured register as long as they do not interfere.
- The *simplify*, *freeze* and *spill* phases cannot be performed on precoloured nodes.
- In the *build* phase precoloured nodes interfere with all other precoloured nodes.
- Since precoloured nodes do not spill, their live ranges should be kept short. Move callee-save registers to fresh temporaries on entry and back on exit, spilling if necessary.

Spilling based on Priorities

When there is a choice of more than one variable to spill, we want to spill the variable which is used least and interferes most with other variables. A common approach is:

$$((uses + defs)_{outside\ loop} + \\ loops\ iterations * (uses + defs)_{inside\ loop})/node\ degree$$

Suppose we have a code fragment which loops 10 times, then for the following values the priorities are:

var	uses+defs outside loop	uses+defs inside loop	degree	priority
a	2	0	4	0.5
b	1	1	4	2.75
c	2	0	6	0.33
d	2	3	4	5.5
e	1	3	3	10.3

In this case we would *spill* variable *c* as it has the lowest priority.