Interference Graph

Summary

The interference graph is the basis of the main technique for performing register allocation

Register allocation consists in choosing the processor registers where each value will reside during program execution

As this depends on the target processor, the decision is made after code generation

However, as the foundation of the technique is independent from the target architecture, it is presented here applied to the intermediate representation

Live ranges (1)

Example (12, repeated)

Liveness analysis for block B_3 resulted in

	UEVar	VarKill	LiveOut	LiveIn
$t_4 \leftarrow$ i_lload $@r$	$@r$	t_4	$t_4, @n$	$@r, @n$
$t_5 \leftarrow$ i_aload $@n$	$@n$	t_5	$t_4, t_5, @n$	$t_4, @n$
$t_6 \leftarrow$ i_mul t_4, t_5	t_4, t_5	t_6	$t_6, @n$	$t_4, t_5, @n$
$@r \leftarrow$ i_lstore t_6	t_6	$@r$	$@n$	$t_6, @n$
$t_7 \leftarrow$ i_aload $@n$	$@n$	t_7	t_7	$@n$
$t_8 \leftarrow$ i_value 1	—	t_8	t_7, t_8	t_7
$t_9 \leftarrow$ i_sub t_7, t_8	t_7, t_8	t_9	t_9	t_7, t_8
$@n \leftarrow$ i_astore t_9	t_9	$@n$	—	t_9
jump B_2	—	—	—	—

Live ranges (2)

Example (12, cont.)

From the results of liveness analysis for block B_3, we see that

@r is only live on entry to instruction 1, and is never used again in the block

@n is live on entry to instructions 1, 2, 3, 4 e 5

t_4 is live from after instruction 1 until the entry to instruction 3

t_5 is live from after instruction 2 until the entry to instruction 3

...

The control flow graph edges where a value is live on exit from the source node and live on entry to the destination node constitute the live range of the value

The live range of a value corresponds to its lifetime

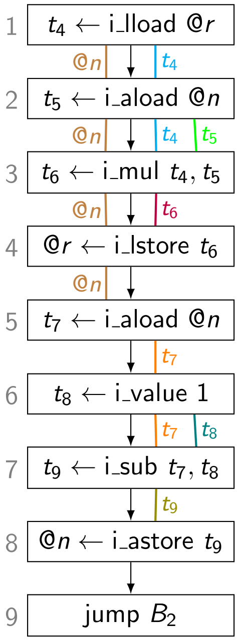
Live ranges (3)

Example (12, cont.)

The live ranges of the several values of block B_3 are shown on the right

- $@n$ is live on edges $1 \to 2 \to 3 \to 4 \to 5$, its live range is 1–5
- t_4 is live on edges $1 \to 2 \to 3$, its live range is 1–3
- t_5 is live on edge $2 \to 3$, its live range is 2–3
- ...

In this block, the live range of $@r$ is empty



Interference graph (1)

Interference

Two values interfere if their live ranges overlap

Two values **do not** interfere if one is a copy of the other everywhere their live ranges overlap

Example (12, cont.)

In B_3, values t_4 and t_5 interfere along edge $2 \rightarrow 3$

Values $@n$ and t_4 interfere along edges $1 \rightarrow 2 \rightarrow 3$

Value t_9 does not interfere with any other value

The live ranges of values $@n$ and t_5 overlap on edge $2 \rightarrow 3$, but since one is a copy of the other along this edge, they **do not** interfere

Interference graph (2)

The lifetimes of non-interfering values are disjoint and they may reside in the same location

Example (12, cont.)

Since t_4 and t_6 do not interfere, we could replace

$t_4 \leftarrow \text{i\_lload } @r$ $t_5 \leftarrow \text{i\_aload } @n$ $t_6 \leftarrow \text{i\_mul } t_4, t_5$ $@r \leftarrow \text{i\_lstore } t_6$	by	$t_4 \leftarrow \text{i\_lload } @r$ $t_5 \leftarrow \text{i\_aload } @n$ $t_4 \leftarrow \text{i\_mul } t_4, t_5$ $@r \leftarrow \text{i\_lstore } t_4$

without changing the meaning of the code

Interference graph (3)

Interference graph

- The interference graph is an undirected graph
- The nodes of the graph are the names (variables and temporaries) appearing in the code
- The edges of the graph correspond to the interferences found in the code

  Nodes a and b are connected by an edge if and only if a and b interfere

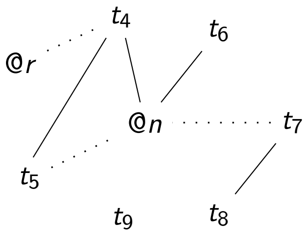
Remarks:

1. Two nodes of an undirected graph are said to be adjacent or neighbours when they are connected by an edge
2. The degree of a node is the number of edges connecting it to other nodes

Interference graph (4)

Example (12, cont.)

Interference graph for block B_3



Values which are at some point a copy of one another are said to be move related

Dotted edges in the graph connect move related nodes

Register allocation (1)

Graph colouring

Register allocation may be made based on the interference graph

If it is possible to colour the graph

- using k colours
- such that adjacent nodes have different colours

then k registers are enough to hold all the values used

Register allocation (2)

Graph colouring

Algorithm

Let k be the number of registers available

1. Simplify While there are nodes with degree less than k in the graph, remove them from the graph (along with their edges)
2. Spill If there is still some node remaining in the graph
   i. Remove one from the graph (it is a candidate for spilling)
   ii. Go back to 1.
3. Select Rebuild the graph, inserting nodes in the reverse order from which they were removed
   If possible, colour each inserted node with a colour different from those of its neighbours
4. Restart If it was not possible to colour some node in step 3., change the code so that the corresponding value(s) are stored to and loaded from memory, and restart from liveness analysis

Register allocation (3)

Graph colouring

Remarks

(a) A node removed from the graph in step 1. has less than k neighbours, hence it and its neighbours can all have a different colour

When it is reinserted into the graph (in step 3.), it is always possible to find a colour for it different from those of its neighbours

(b) A node removed from the graph in step 2. has at least k neighbours, and it may not be possible to find a colour for it which is different from those of its neighbours

However, if its neighbours do not use all k colours, it will be possible to colour it in step 3.

(c) Removing nodes in steps 1. and 2., may cause other nodes to be left with less than k neighbours

Register allocation (4)

Graph colouring

Remarks

(d) If, in step 3., it is not possible to colour a node with a colour different from those of its neighbours, there are, at some point in the code, more than k live values

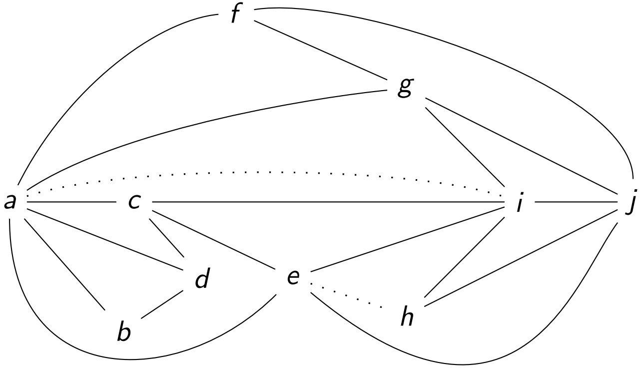
Since there are only k available registers, at least one of those values will have to be spilled to memory

(e) Spilling a value consists in introducing, into the code, instructions to store it to memory when it is created, and to load it from memory just before it is needed

(f) The knowledge that two values are move related may be used in step 3. as a hint for colouring both nodes with the same colour, which will allow deleting the instruction which copies one value to the other

Example (16)

Interference graph with 10 nodes, to be coloured with 4 colours



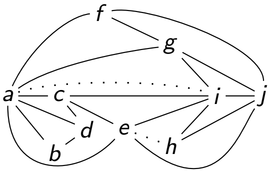
Node *a* has degree 6, nodes *i* and *j* have degree 5, nodes *c*, *e* and *g* have degree 4, *d* and *f* have degree 3, and *b* and *h* have degree 2

Register allocation (6)

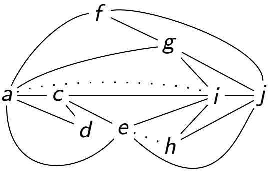
Graph colouring

Example (16, cont.)

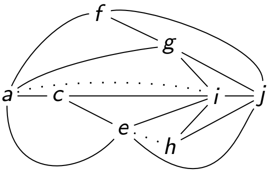
Simplify



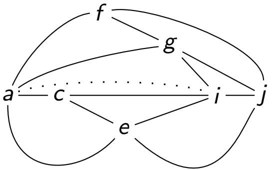
Initial graph



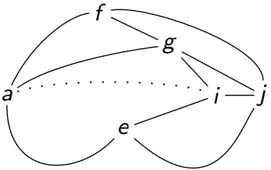
Removed *b*



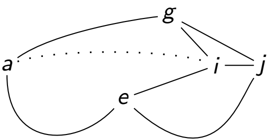
Removed *d*



Removed *h*



Removed *c*



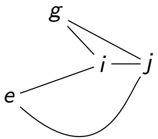
Removed *f*

Register allocation (7)

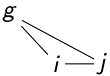
Graph colouring

Example (16, cont.)

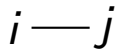
Simplify (cont.)



Removed *a*



Removed *e*



Removed *g*

Removed *i*

Removed *j*

After removing node *j*, the graph becomes empty

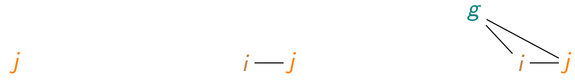
The selection phase initiates at this point

Register allocation (8)

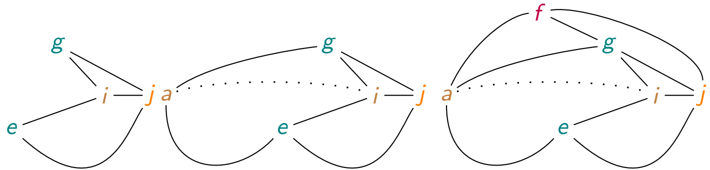
Graph colouring

Example (16, cont.)

Select, using the 4 colours ● ● ● ●



Inserted *j* Inserted *i* Inserted *g*



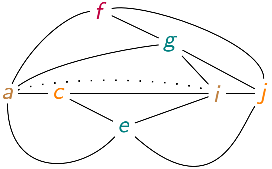
Inserted *e* Inserted *a* Inserted *f*

Register allocation (9)

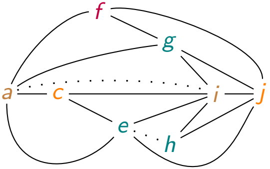
Graph colouring

Example (16, cont.)

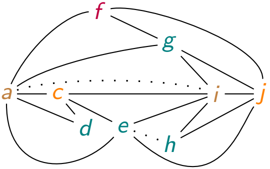
Select (cont.), using the 4 colours ● ● ● ●



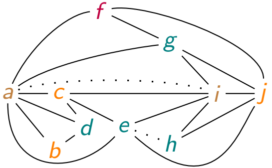
Inserted *c*



Inserted *h*



Inserted *d*



Inserted *b*

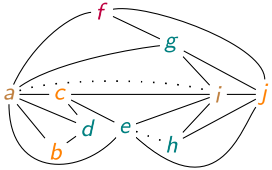
Every node has now been reinserted into the graph and each one was coloured with one of 4 colours, and all neighbours have different colours

This was possible even though there are nodes in the graph with more than 4 neighbours

Register allocation (10)

Graph colouring

Example (16, cont.)



1. This graph is 4-colourable, even though there are nodes with 4 or more neighbours
2. If the 4 registers are r_1, r_2, r_3 and r_4

   Values b, c and j may use register r_1

   Values a and i may use register r_2

   Values d, e, g and h may use register r_3

   Value f will be the only one to use register r_4
3. Since values e and h occupy the same register, the instruction that copies one to the other may be eliminated

   The same is true for values a and i
4. This is not the only possible 4-colouring (not considering colour renaming)

Register allocation (11)

Graph colouring

Example (17)

Spilling a value

Let t_i stand for the value which could not be coloured and let the instructions on the right be those where it occurs

$t_i \leftarrow$ i_add t_3, t_2
$\ldots$
$t_5 \leftarrow$ i_lt t_i, t_1

To spill t_i to memory, those instructions are replaced by the ones below, where

- $@s_1$ will correspond to a temporary memory location within the function activation record
- t_{i1} and t_{i2} are two new temporaries which replace t_i and whose live ranges are shorter than that of t_i

$t_{i1} \leftarrow$ i_add t_3, t_2
$@s1 \leftarrow$ i_lstore t_{i1}
$\ldots$
$t_{i2} \leftarrow$ i_lload $@s1$
$t_5 \leftarrow$ i_lt t_{i2}, t_1

Exercise (1)

For the IR on the next slide:

1. Compute the live ranges of all the temporaries
   (*Do not peek at slide 22 before finishing*)
2. Draw the interference graph
   (*Do not peek at slide 23 before finishing*)
3. Apply the graph colouring algorithm with $k = 4$
4. Modify the IR to include the spilling of `t0`
5. Apply the graph colouring algorithm with $k = 3$

Exercise (2)

IR

```
function @facx
            t0 <- i_aload @n
            t1 <- i_value 1
            t2 <- i_value 0
            t2 <- i_lt t0, t2
            cjump t2, l1, l0
l0:         t3 <- i_copy t1
l3:         t4 <- i_copy t0
            t5 <- i_value 0
            t6 <- i_lt t5, t4
            cjump t6, l4, l5
l4:         t3 <- i_mul t3, t0
            t0 <- i_sub t0, t1
            jump l3
l5:         jump l2
l1:         t3 <- i_value 1
            t3 <- i_inv t3
l2:         i_return t3
```

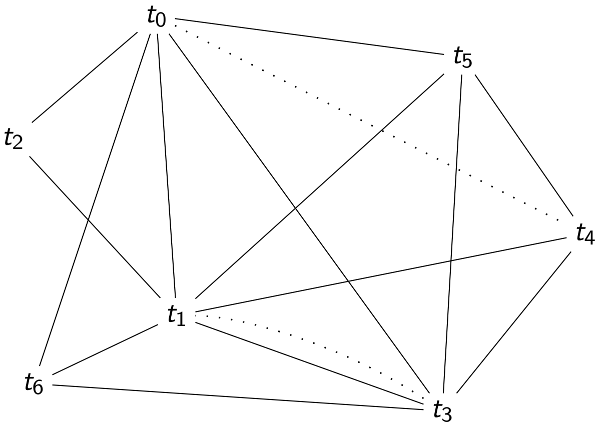
Exercise (3)

Live ranges

Temporary	Live range
t_0	1–13
t_1	2–13
t_2	3–5
t_3	6–17
t_4	7–9
t_5	8–9
t_6	9–10

Exercise (4)

Interference graph



The live ranges of t_0 and t_4 overlap on egdes $7 \to 8$ and $8 \to 9$, but since one is a copy the other along those edges, they **do not** interfere

Register allocation — Take 2 (1)

With coalescing and pre-coloured nodes

Node coalescing

Coalescing two move related nodes is fusing them together into a single node, which will inherit the edges of both nodes

When two nodes are coalesced, the degree of the new node will be less than or equal to the sum of the degrees of the coalesced nodes and the degree of other nodes may decrease

Two interference graph nodes are move related if, at some point, one value is the copy of the other

If move related nodes are coalesced, both values will be assigned to the same register and the copy instruction may be eliminated

Coalescing is only done when safe, i.e., if it will not turn a k-colourable graph into a non k-colourable one

Register allocation — Take 2 (2)

With coalescing and pre-coloured nodes

Criteria for safe node coalescing

Briggs

Two nodes may be coalesced if the resulting node will have less than k neighbours of degree greater than or equal to k

George

Nodes a and b may be coalesced if every neighbour of a

- already interferes with b, or
- has degree less than k

Usually, a is a pre-coloured node

These are conservative criteria

Register allocation — Take 2 (3)

With coalescing and pre-coloured nodes

Pre-coloured nodes

Pre-coloured nodes in an interference graph correspond to registers with predefined roles, such as holding function arguments, function results or the return address, or callee-saved registers

All pre-coloured nodes interfere with each other

Register allocation — Take 2 (4)

With coalescing and pre-coloured nodes

Algorithm

Let k be the number of registers available

1. Simplify While there are non move related and not pre-coloured nodes with degree less than k in the graph, remove them from the graph (along with their edges)
2. Coalesce If there are nodes that can be safely coalesced
   i. Coalesce every possible pair
   ii. Go back to 1
3. Freeze If there is a move-related node of degree less than k
   i. Since it cannot be coalesced, forget about the copy and simplify it

      This correponds to giving up the hope of eliminating the copy instruction, since the nodes may end up with different colours
   ii. Go back to 1

Register allocation — Take 2 (5)

With coalescing and pre-coloured nodes

Algorithm (cont.)

4. Spill If there is still some node remaining in the graph, other than pre-coloured nodes
   i. Remove one node from the graph (it is a candidate for spilling)
   ii. Go back to 1.
5. Select Rebuild the graph, inserting nodes in the reverse order from which they were removed
   If possible, colour each inserted node with a colour different from those of its neighbours
6. Restart If it was not possible to colour some node(s) in step 5., change the code so that the corresponding value(s) are stored to and loaded from memory, and restart from the liveness analysis

Register allocation — Take 2 (6)

With coalescing and pre-coloured nodes

Choosing a candidate for spilling

Spilling has a cost, since it involves storing a value into and loading it from memory

- The more times a spilled value is used or defined, the greater the cost
- Spilling a value that is accessed inside a loop entails a greater cost than a value only accessed outside loops

When a candidate for spilling is removed from the graph, the degree of its neighbours decreases

- The more neighbours the node has, the greater the chances of one node becoming simplifiable

Preference should be given to the higher degree values least accessed when choosing a spilling candidate

Register allocation — Take 2 (7)

With coalescing and pre-coloured nodes

Example (18)

Code for a function, including registers with predefined roles

Registers r_1 and r_2 contain the actual arguments of the function

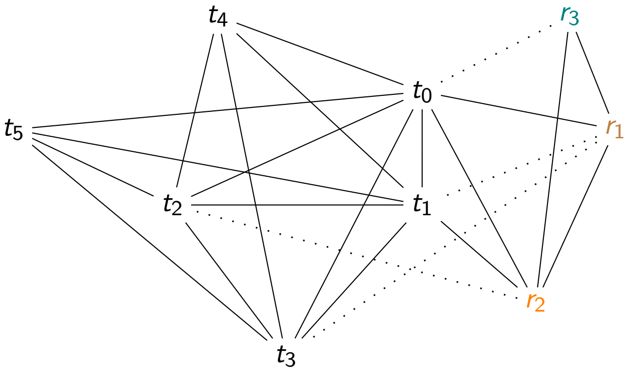
Register r_1 will hold the return value

Register r_3 is a callee-saved register

$$
\begin{array}{ll}
 & t_0 \leftarrow \text{i\_copy } r_3 \\
 & t_1 \leftarrow \text{i\_copy } r_1 \\
 & t_2 \leftarrow \text{i\_copy } r_2 \\
 & t_3 \leftarrow \text{i\_value } 0 \\
l_1 : & t_3 \leftarrow \text{i\_add } t_3, t_2 \\
 & t_4 \leftarrow \text{i\_value } 1 \\
 & t_1 \leftarrow \text{i\_sub } t_1, t_4 \\
 & t_5 \leftarrow \text{i\_value } 0 \\
 & t_5 \leftarrow \text{i\_lt } t_5, t_1 \\
 & \text{cjump } t_5, l_1, l_2 \\
l_2 : & r_1 \leftarrow \text{i\_copy } t_3 \\
 & r_3 \leftarrow \text{i\_copy } t_0 \\
 & \text{i\_return } r_1
\end{array}
$$

Example (18, cont.)

Interference graph

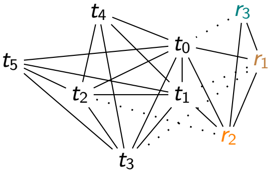


Register allocation — Take 2 (9)

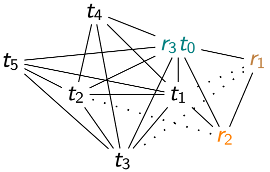
With coalescing and pre-coloured nodes

Example (18, cont.)

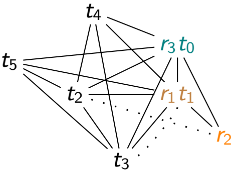
Colouring the graph with $k = 4$ colours



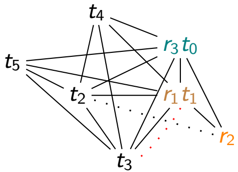
Initial graph



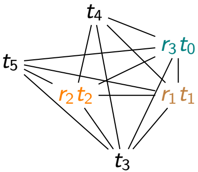
Coalesce r_3 and t_0



Coalesce r_1 and t_1



Constrained move



Coalesce r_2 and t_2

No simplifying, coalescing or freezing possible

Must choose a candidate for spilling

Register allocation — Take 2 (10)

With coalescing and pre-coloured nodes

Example (18, cont.)

Choosing a spilling candidate

Uses and definitions of the candidates

	Outside the loop		Inside the loop		Degree
	Uses	Defs	Uses	Defs	
$r_1\,t_1$	2	2	2	1	5
$r_2\,t_2$	1	1	1	0	5
$r_3\,t_0$	2	2	0	0	5
t_3	1	1	1	1	5
t_4	0	0	1	1	4
t_5	0	0	2	2	4

(Counts for coalesced nodes are the sum of the counts for the individual nodes)

Register allocation — Take 2 (11)

With coalescing and pre-coloured nodes

Example (18, cont.)

Choosing a spilling candidate (cont.)

Estimating spilling costs

$$
\begin{array}{lll}
r_1 t_1 & : (\ 2 + 2 + 10 \times (\ 2 + 1\ ))/\ 5 = \frac{34}{5} & \\
r_2 t_2 & : (\ 1 + 1 + 10 \times (\ 1 + 0\ ))/\ 5 = \frac{12}{5} & \\
r_3 t_0 & : (\ 2 + 2 + 10 \times (\ 0 + 0\ ))/\ 5 = \frac{4}{5} & \leftarrow \\
t_3 & : (\ 1 + 1 + 10 \times (\ 1 + 1\ ))/\ 5 = \frac{22}{5} & \\
t_4 & : (\ 0 + 0 + 10 \times (\ 1 + 1\ ))/\ 4 = \frac{20}{4} & \\
t_5 & : (\ 0 + 0 + 10 \times (\ 2 + 2\ ))/\ 4 = \frac{40}{4} &
\end{array}
$$

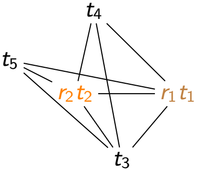
Uses and definitions outside any loop have cost 1, each loop level increases the cost by a factor of 10, and an `if` would halve the cost

Register allocation — Take 2 (12)

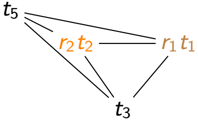
With coalescing and pre-coloured nodes

Example (18, cont.)

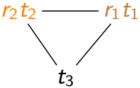
Colouring the graph (cont.)



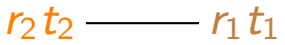
Spill (potential) $r_3\,t_0$



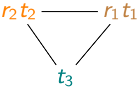
Simplify (t_4)



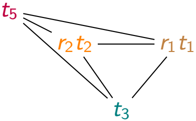
Simplify (t_5)



Simplify (t_3)



Select (t_3)



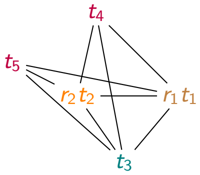
Select (t_5)

Register allocation — Take 2 (13)

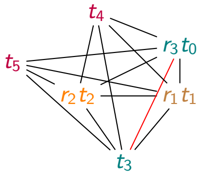
With coalescing and pre-coloured nodes

Example (18, cont.)

Colouring the graph (cont.)



Select (t_4)



Actually spill $r_3\,t_0$

The stack is empty, no more selecting is possible

It is time to actually spill a candidate

In this case, there is only one candidate

Register allocation — Take 2 (14)

With coalescing and pre-coloured nodes

Example (18, cont.)

Spilling $r_3 t_0$

t_0 is replaced by a series of new temporaries which hold its value in transit to and from memory

The new temporaries must not be spilled (they should have a spilling cost of ∞)

$@s_1$ represents the location in the current activation record where the value will be stored

(This location must be independently allocated)

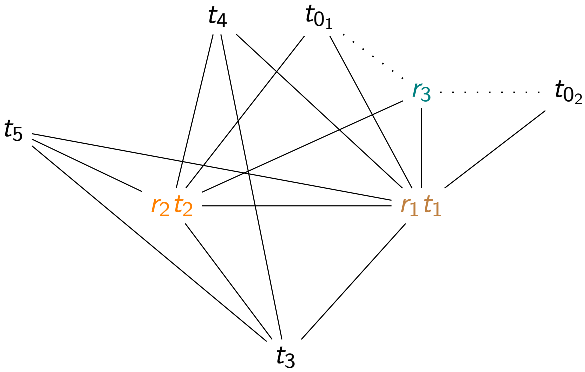
$$
\begin{array}{rl}
 & t_{0_1} \leftarrow \text{i\_copy } r_3 \\
 & @s_1 \leftarrow \text{i\_lstore } t_{0_1} \\
 & t_1 \leftarrow \text{i\_copy } r_1 \\
 & t_2 \leftarrow \text{i\_copy } r_2 \\
 & t_3 \leftarrow \text{i\_value } 0 \\
l_1: & t_3 \leftarrow \text{i\_add } t_3, t_2 \\
 & t_4 \leftarrow \text{i\_value } 1 \\
 & t_1 \leftarrow \text{i\_sub } t_1, t_4 \\
 & t_5 \leftarrow \text{i\_value } 0 \\
 & t_5 \leftarrow \text{i\_lt } t_5, t_1 \\
 & \text{cjump } t_5, l_1, l_2 \\
l_2: & r_1 \leftarrow \text{i\_copy } t_3 \\
 & t_{0_2} \leftarrow \text{i\_lload } @s_1 \\
 & r_3 \leftarrow \text{i\_copy } t_{0_2} \\
 & \text{i\_return } r_1
\end{array}
$$

Register allocation — Take 2 (15)

With coalescing and pre-coloured nodes

Example (18, cont.)

Interference graph for rewritten code



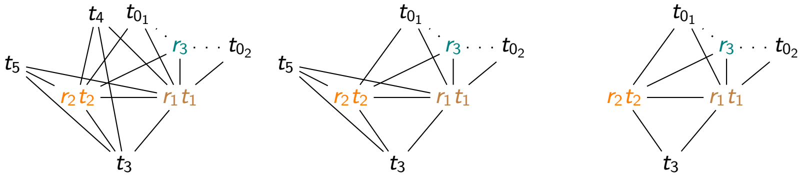
Nodes coalesced prior to the first potential spill may be kept

Register allocation — Take 2 (16)

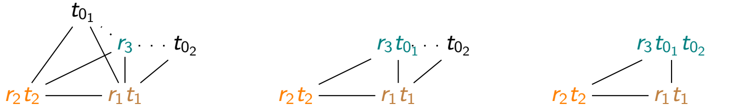
With coalescing and pre-coloured nodes

Example (18, cont.)

Colouring the new graph



Initial graph | Simplify (t_4) | Simplify (t_5)



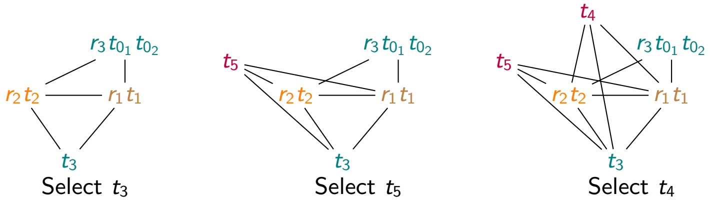
Simplify (t_3) | Coalesce r_3 and t_{0_1} | Coalesce $r_3\,t_{0_1}$ and t_{0_2}

Register allocation — Take 2 (17)

With coalescing and pre-coloured nodes

Example (18, cont.)

Colouring the new graph (cont.)



It's done!

According to the resulting colouring, t_1 will use register r_1, t_2 will use r_2, t_3, t_{0_1} and t_{0_2} will use r_3, and both t_4 and t_5 will use r_4

The temporaries will now be replaced by the registers they will use

Register allocation — Take 2 (18)

With coalescing and pre-coloured nodes

Example (18, cont.)

$$
\begin{array}{ll}
 & r_3 \leftarrow \text{i\_copy } r_3 \\
 & @s_1 \leftarrow \text{i\_lstore } r_3 \\
 & r_1 \leftarrow \text{i\_copy } r_1 \\
 & r_2 \leftarrow \text{i\_copy } r_2 \\
 & r_3 \leftarrow \text{i\_value } 0 \\
l_1: & r_3 \leftarrow \text{i\_add } r_3, r_2 \\
 & r_4 \leftarrow \text{i\_value } 1 \\
 & r_1 \leftarrow \text{i\_sub } r_1, r_4 \\
 & r_4 \leftarrow \text{i\_value } 0 \\
 & r_4 \leftarrow \text{i\_lt } r_4, r_1 \\
 & \text{cjump } r_4, l_1, l_2 \\
l_2: & r_1 \leftarrow \text{i\_copy } r_3 \\
 & r_3 \leftarrow \text{i\_lload } @s_1 \\
 & r_3 \leftarrow \text{i\_copy } r_3 \\
 & \text{i\_return } r_1
\end{array}
$$

Removing useless copy instructions

$$
\begin{array}{ll}
 & @s_1 \leftarrow \text{i\_lstore } r_3 \\
 & r_3 \leftarrow \text{i\_value } 0 \\
l_1: & r_3 \leftarrow \text{i\_add } r_3, r_2 \\
 & r_4 \leftarrow \text{i\_value } 1 \\
 & r_1 \leftarrow \text{i\_sub } r_1, r_4 \\
 & r_4 \leftarrow \text{i\_value } 0 \\
 & r_4 \leftarrow \text{i\_lt } r_4, r_1 \\
 & \text{cjump } r_4, l_1, l_2 \\
l_2: & r_1 \leftarrow \text{i\_copy } r_3 \\
 & r_3 \leftarrow \text{i\_lload } @s_1 \\
 & \text{i\_return } r_1
\end{array}
$$

Register allocation — Take 2 (19)

With coalescing and pre-coloured nodes

Example (19)

With one less register...

Using only 3 registers should result in something like the code on the right

$$
\begin{array}{rl}
 & @s_1 \leftarrow \text{i\_lstore } r_3 \\
 & @s_2 \leftarrow \text{i\_lstore } r_2 \\
 & r_3 \leftarrow \text{i\_value } 0 \\
l_1: & r_2 \leftarrow \text{i\_lload } @s_2 \\
 & r_3 \leftarrow \text{i\_add } r_3, r_2 \\
 & r_2 \leftarrow \text{i\_value } 1 \\
 & r_1 \leftarrow \text{i\_sub } r_1, r_2 \\
 & r_2 \leftarrow \text{i\_value } 0 \\
 & r_2 \leftarrow \text{i\_lt } r_2, r_1 \\
 & \text{cjump } r_2, l_1, l_2 \\
l_2: & r_1 \leftarrow \text{i\_copy } r_3 \\
 & r_3 \leftarrow \text{i\_lload } @s_1 \\
 & \text{i\_return } r_1
\end{array}
$$