Register allocation

Michel Schinz (based on Erik Stenman's slides)

Advanced Compiler Construction / 2006-06-09

Register allocation

Storage of values

Programs manipulate values, which are first defined – i.e. computed – and later used for further computation, possibly several times.

Between the time it is defined and used, a value must be stored somewhere. There are two options: in memory, or in a machine register.

Registers are the best location to store values, as they are faster than memory, and often the only location where computation is possible.

Register allocation

Since registers are a better location to store values than memory, *all* values should be stored in them, ideally.

Unfortunately, registers are a very scarce resource compared to memory. They must therefore be used as sparingly as possible.

The aim of register allocation is to decide how to use registers, *i.e.* which values to put in them, and when.

Register allocation techniques

There are several kinds of register allocation techniques:

- local techniques, which work on basic blocks or single expressions,
- global techniques, which work on whole functions,
- inter-procedural techniques, which work on several procedures at a time.

Register allocation by graph colouring

Register allocation by graph colouring

Register allocation by graph colouring is a global register allocation technique which performs well. It is probably the most commonly used technique in optimising compilers.

Its idea is to express the register allocation problem as a graph colouring problem, which is then solved using heuristics.

Interference graph

The interference graph represents the interference among program values.

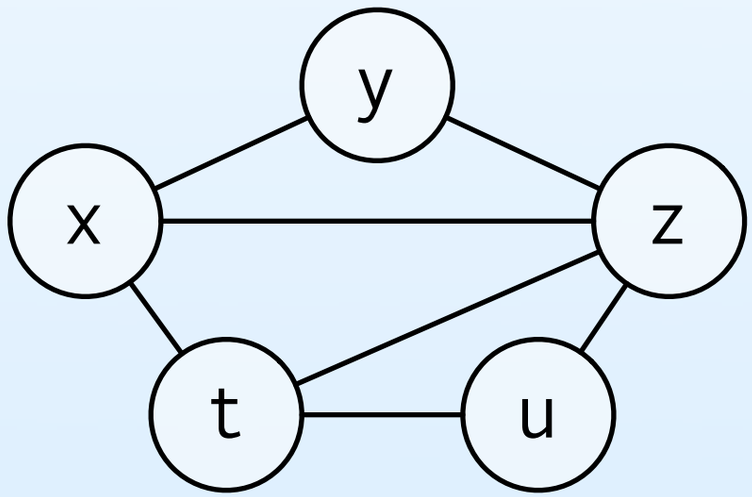
The nodes of the interference graph represent program values, and there is an edge from n_1 to n_2 if the values corresponding to n_1 and n_2 are simultaneously live.

Interference graph example

Live ranges

	x	y	z	t	u
x←1					
y←2	■				
z←x+y	■	■			
t←y	■	■	■		
u←x+t	■		■	■	
print z			■	■	■
print t				■	■
print u					■

Interference graph



Graph colouring

The goal of **graph colouring** is to find a way to assign *K* colours to the nodes of a graph so that no two nodes connected by an edge have the same colour.

Graph colouring can be used to allocate registers to values, by trying to colour the interference graph with as many colours as there are registers in the target machine. This is not always possible, in which case some values must be spilled – *i.e.* stored in memory.

Graph colouring complexity

Graph colouring is an NP-complete problem.

Heuristics therefore have to be used to perform register allocation by graph colouring. In practice, they give good results.

We will examine one such heuristic, colouring by simplification.

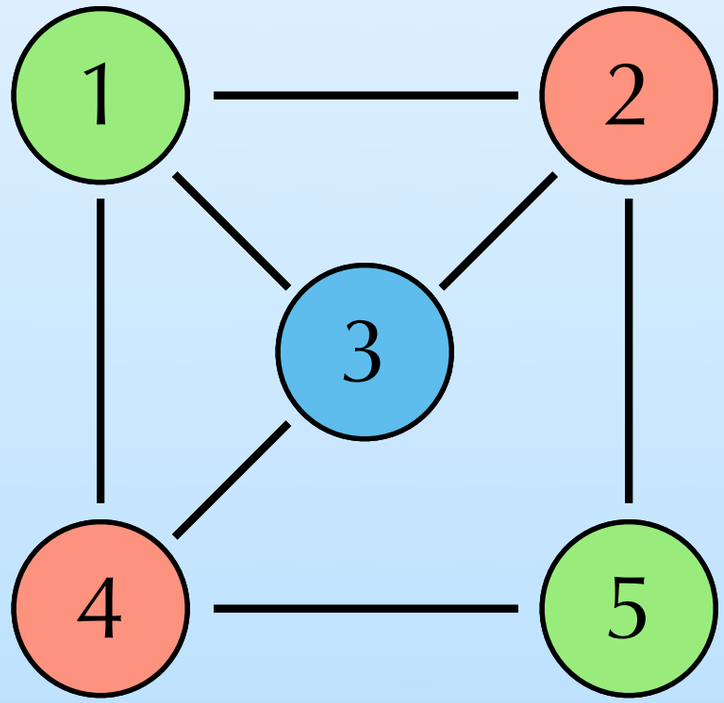
Colouring by simplification

Colouring by simplification works as follows: as long as the graph G has at least a node n with fewer than K neighbours – K being the numbers of available colours – n is removed from G, and colouring proceeds with that simplified graph.

Clearly, if the simplified graph is K-colourable, then so is G: since n has less than K neighbours, those use at most K-1 colours, and there is therefore at least one colour available for n.

Colouring by simplification: example

To illustrate colouring by simplification, we can colour the following graph with K=3 colours.



Stack of removed nodes: 5 2 1 3

Spilling

During simplification, it is perfectly possible to reach a point where all nodes have at least K neighbours.

When this occurs, a node must be chosen and its value must be stored in memory instead of in a register. This is called **spilling**.

As a first approximation, we can assume that the spilled value does not interfere with any other value, and remove its node from the graph.

Potential and actual spills

When colours are assigned to nodes, it can happen that a node initially designated as spilled can be coloured because its neighbours do not use all available colours.

When this happens, the potential spill is not turned into an actual spill.

This technique is known as **optimistic colouring**.

Consequences of spilling

When a node is really spilled, the program has to be rewritten to take this into account: each time the spilled value is used, it must be fetched from memory, and each time it is defined, the new value must be written back to memory.

This rewriting changes the interference graph, and therefore the allocation process must be restarted completely. In practice, it converges in one or two iterations in most cases.

Coalescing

When two nodes n_1 and n_2 in the interference graph do not share an edge, it is possible to **coalesce** them by replacing them by their union.

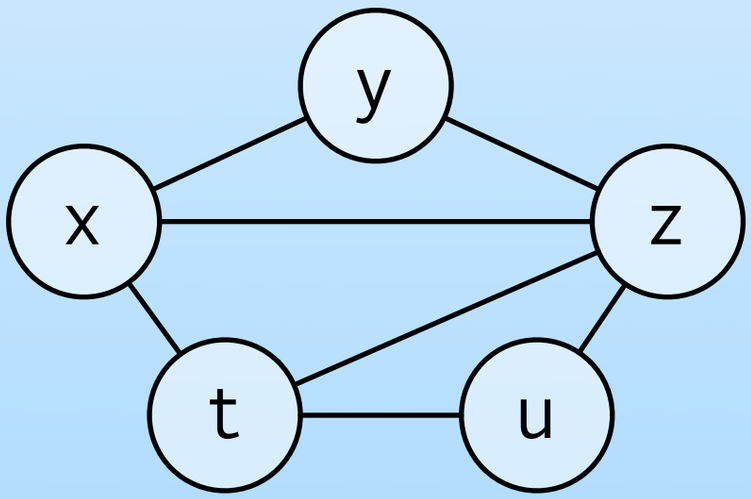
This has two consequences: the positive one is that all instructions which copy the value of n_1 into the value of n_2 – or the other way around – can be removed from the program; the negative one is that the resulting graph can be harder to colour.

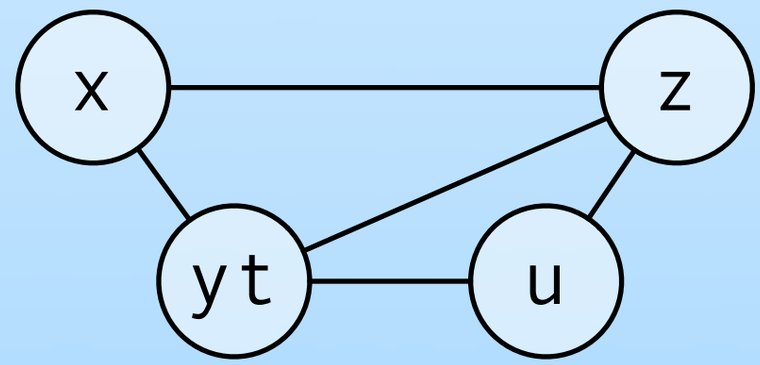
Coalescing example

```
x←1
y←2
z←x+y
t←y
u←x+t
print z
print t
print u
```

coalescing of y and t

```
x←1
yt←2
z←x+yt
u←x+yt
print z
print yt
print u
```





Coalescing heuristics

Several heuristics have been developed to decide when coalescing is safe, *i.e.* when it is guaranteed that it will not turn a K-colourable graph into one which is not K-colourable.

Using such heuristics, it is possible to interleave simplification steps with safe coalescing steps, thereby removing many useless move operations.

Live-range splitting

It can sometimes be beneficial to split a long live range in two or more parts, by saving the value to memory at one point, and re-fetching it later. This technique is called **live-range splitting**.

However, it is hard to find good heuristics to decide which live-ranges should be split, and where.

Live-range splitting example

	x	y	z
x←1			
y←x+2	■		
z←y*2	■	■	
y←y+z	■	■	■
print z	■	■	■
print y	■	■	
print x	■		

not 2-colourable

splitting x's range

	x	y	z
x←1			
y←x+2	■		
store x	■	■	
z←y*2		■	
y←y+z		■	■
print z		■	■
print y		■	
x←fetch			
print x	■		

2-colourable

Pre-coloured nodes

It is often necessary to put some values in specific registers, *e.g.* to adhere to calling conventions.

This can be handled as follows: if the machine has K registers, then K values will be created to represent them. In the interference graph, the nodes corresponding to those values will be pre-coloured, to ensure that they get "allocated" to their corresponding register.

Linear scan register allocation

Linear scan

Linear scan is a global register allocation technique which is substantially simpler – and faster – than graph colouring. It still gives very good results.

It is especially interesting for applications where compilation time must be kept as low as possible, for example in JIT compilers.

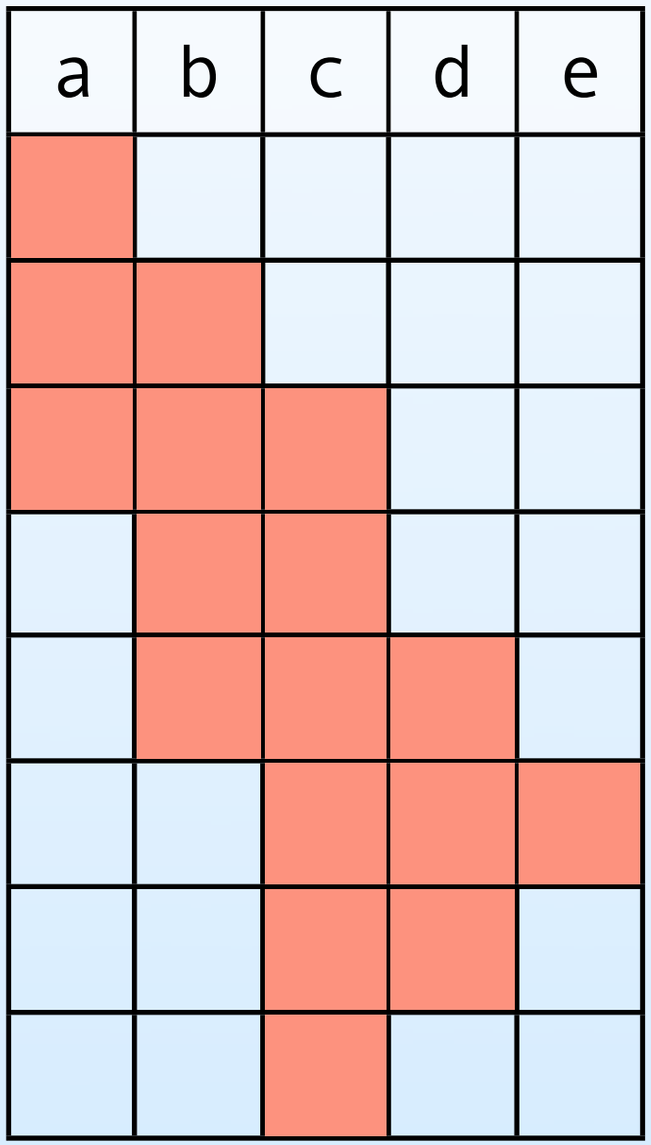
Linear scan algorithm

Linear scan works on a linear representation of the program. Live ranges must be known for all values.

The algorithm scans live ranges from first to last. Whenever there are less than K values live at the same time, they are all put in registers. When all registers are allocated and a new value becomes live, one of them must be spilled. The one whose live range ends last is systematically chosen.

Linear scan example

Live ranges



Allocation

R1	R2
a	
a	b
a	b
	b
d	b
d	e
d	

c is spilled

Summary

Register allocation is an optimisation which tries to make efficient use of registers, by storing as many values in them as possible.

Most techniques used in practice are global, i.e. they work on complete procedures.

We have examined two of them: graph colouring, which gives very good results but is relatively complicated; and linear scan, which is not as good, but faster and a lot simpler.