Register allocation

Michel Schinz

Advanced Compiler Construction – 2008-05-16

Register allocation

The problem of **register allocation** consists in rewriting a program that makes use of an unbounded number of local variables – also called **virtual** or **pseudo-registers** – into one that only makes use of machine registers.

If there are not enough machine registers to store all variables, one or several variables must be **spilled**, *i.e.* stored in memory instead of in a register.

Register allocation is generally one of the very last phases of the compilation process – only instruction scheduling can come later. It is performed on an intermediate language that is extremely close to machine code.

Setting the scene

We will illustrate register allocation using programs written in a slight extension of minivm's assembly code:

- apart from *n* machine registers R_0, …, R_n, an unbounded number of virtual registers v_0, v_1, … are available *before* register allocation,
- machine registers that play a special role, like the frame pointer, are identified with a non-numerical index, *e.g.* R_{FP}; they are real registers nevertheless,
- a `MOVE` R_a R_b instruction is available, to copy the contents of R_b into R_a,
- `LOAD` and `STOR` instructions also accept integer values as their third operand, as in `LOAD R1 R2 5`.

Example function

To illustrate register allocation techniques, we will use a function computing the greatest common denominator of two numbers using Euclid's algorithm.

In minischeme

```
(define gcd
  (lambda (a b)
    (if (= 0 b)
        a
        (gcd b (% a b)))))
```

In (hand-coded) assembly

```
gcd:  LINT R3 done
      JMPZ R3 R2
      ADD  R3 R2 R0
      MOD  R2 R1 R2
      ADD  R1 R3 R0
      LINT R3 gcd
      JMPZ R3 R0
done: JMPZ R29 R0
```

Register allocation example

Before register allocation

```
gcd:   MOVE v0 RLK
       MOVE v1 R1
       MOVE v2 R2
loop:  LINT v3 done
       JMPZ v3 v2
       MOVE v4 v2
       MOD  v2 v1 v2
       MOVE v1 v4
       LINT v5 loop
       JMPZ v5 R0
done:  MOVE R1 v1
       JMPZ v0 R0
```

R0: zero

R1, R2: parameters

R_{LK}: return address

allocable registers: R1, R2, R3, R_{LK}

After register allocation

```
gcd:
loop:  LINT R3 done
       JMPZ R3 R2
       MOVE R3 R2
       MOD  R2 R1 R2
       MOVE R1 R3
       LINT R3 loop
       JMPZ R3 R0
done:  JMPZ RLK R0
```

Allocation:

v0 → R_{LK}

v1 → R1

v2 → R2

v3, v4, v5 → R3

Register allocation techniques

We will study the two most commonly used techniques:

1. register allocation by **graph colouring**, which is relatively slow but produces very good results,
2. **linear scan** register allocation, which is fast but produces slightly worse results – at least in its standard form.

Because it is slow, graph colouring tends to be used in batch compilers, while linear scan tends to be used in JIT compilers.

Both techniques are **global**, *i.e.* they allocate registers for a whole function at a time.

Technique #1
Register allocation by graph colouring

Allocation by graph colouring

The problem of register allocation can be reduced to the well-known problem of graph colouring, as follows:

1. The **interference graph** is built. It has one node per register (real or virtual), and two nodes are connected by an edge iff their registers are simultaneously live.
2. The interference graph is coloured with at most K colours – K = number of available registers – so that all nodes have a different colour than all their neighbours.

Problems:

1. for an arbitrary graph, the colouring problem is NP-complete,
2. a K-colouring might not even exist.

Interference graph example

Program

```
gcd:
  MOVE v0 R_LK
  MOVE v1 R1
  MOVE v2 R2
loop:
  LINT v3 done
  JMPZ v3 v2
  MOVE v4 v2
  MOD  v2 v1 v2
  MOVE v1 v4
  LINT v5 loop
  JMPZ v5 R0
done:
  MOVE R1 v1
  JMPZ v0 R0
```

Liveness

{in}{out}

$\{R_1,R_2,R_{LK}\}\{R_1,R_2,v_0\}$

$\{R_1,R_2,v_0\}\{R_2,v_0,v_1\}$

$\{R_2,v_0,v_1\}\{v_0\text{-}v_2\}$

$\{v_0\text{-}v_2\}\{v_0\text{-}v_3\}$

$\{v_0\text{-}v_3\}\{v_0\text{-}v_2\}$

$\{v_0\text{-}v_2\}\ \{v_0\text{-}v_2,v_4\}$

$\{v_0\text{-}v_2,v_4\}\{v_0\text{-}v_2,v_4\}$

$\{v_0\text{-}v_2,v_4\}\{v_0\text{-}v_2\}$

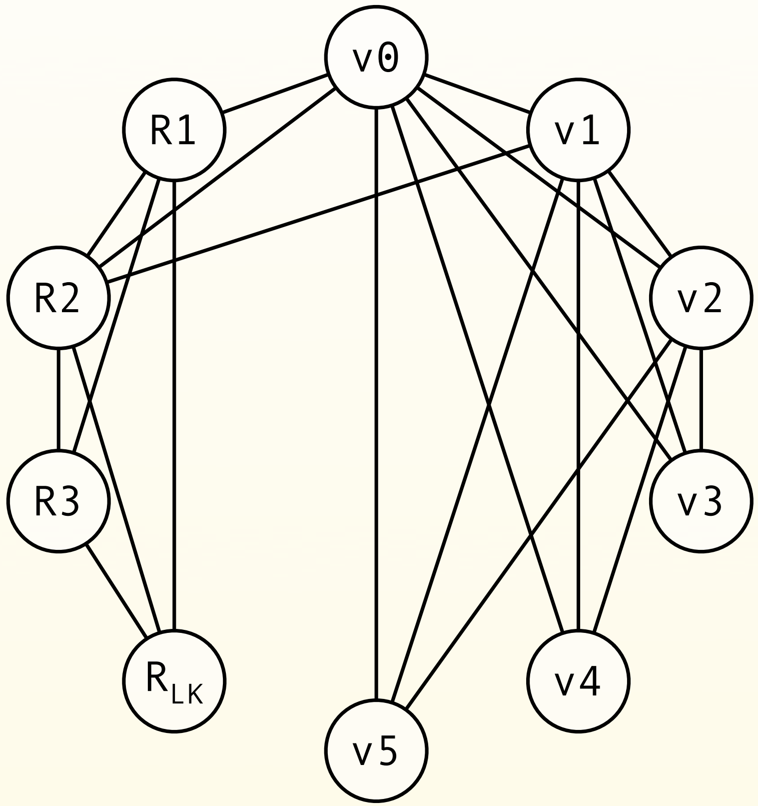
$\{v_0\text{-}v_2\}\{v_0\text{-}v_2,v_5\}$

$\{v_0\text{-}v_2,v_5\}\{v_0\text{-}v_2\}$

$\{v_0,v_1\}\{R_1,v_0\}$

$\{R_1,v_0\}\{R_1\}$

Interference graph

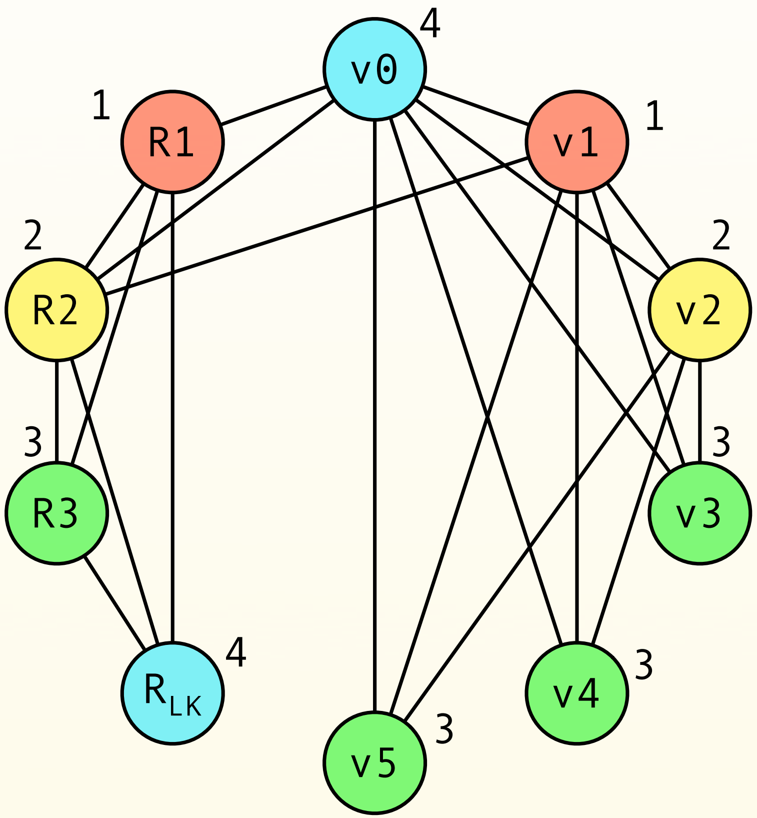


Colouring example

Original program

```
gcd:
  MOVE v0 RLK
  MOVE v1 R1
  MOVE v2 R2
loop:
  LINT v3 done
  JMPZ v3 v2
  MOVE v4 v2
  MOD  v2 v1 v2
  MOVE v1 v4
  LINT v5 loop
  JMPZ v5 R0
done:
  MOVE R1 v1
  JMPZ v0 R0
```

Coloured interference graph



Rewritten program

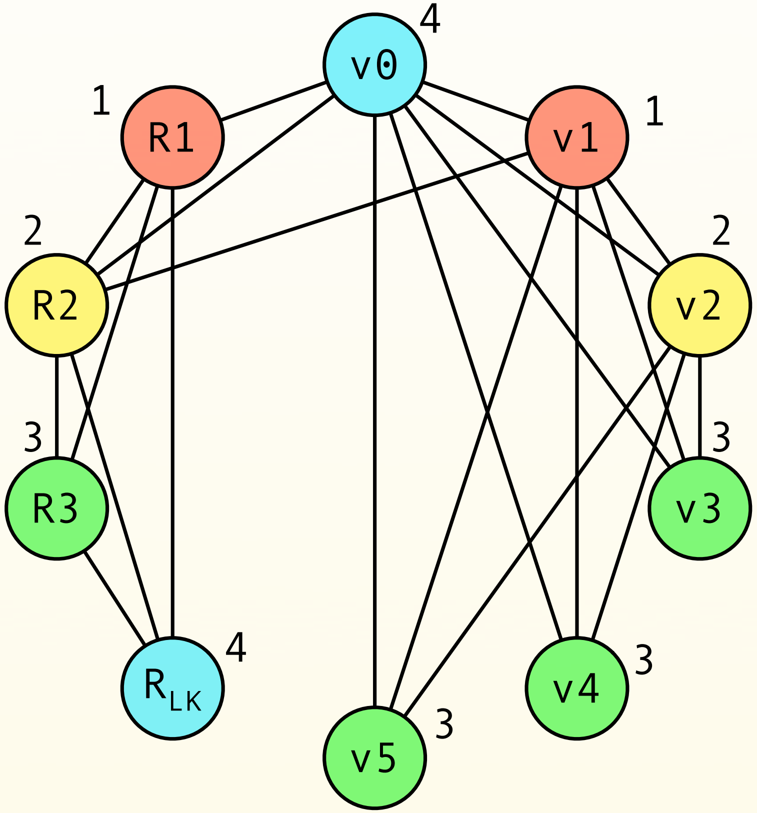
```
gcd:
  MOVE RLK RLK
  MOVE R1 R1
  MOVE R2 R2
loop:
  LINT R3 done
  JMPZ R3 R2
  MOVE R3 R2
  MOD  R2 R1 R2
  MOVE R1 R3
  LINT R3 loop
  JMPZ R3 R0
done:
  MOVE R1 R1
  JMPZ RLK R0
```

Colouring example

Original program

gcd:
  MOVE v0 R_{LK}
  MOVE v1 R1
  MOVE v2 R2
loop:
  LINT v3 done
  JMPZ v3 v2
  MOVE v4 v2
  MOD v2 v1 v2
  MOVE v1 v4
  LINT v5 loop
  JMPZ v5 R0
done:
  MOVE R1 v1
  JMPZ v0 R0

Coloured interference graph



Rewritten program

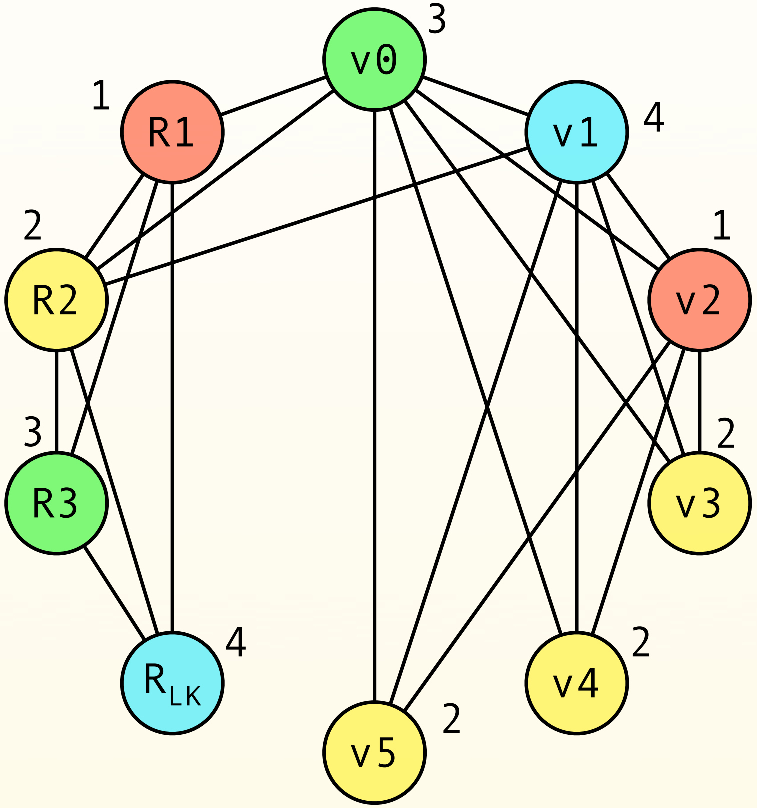
gcd:
  ~~MOVE R_{LK} R_{LK}~~
  ~~MOVE R1 R1~~
  ~~MOVE R2 R2~~
loop:
  LINT R3 done
  JMPZ R3 R2
  MOVE R3 R2
  MOD R2 R1 R2
  MOVE R1 R3
  LINT R3 loop
  JMPZ R3 R0
done:
  ~~MOVE R1 R1~~
  JMPZ R_{LK} R0

Colouring example (2)

Original program

```
gcd:
  MOVE v0 RLK
  MOVE v1 R1
  MOVE v2 R2
loop:
  LINT v3 done
  JMPZ v3 v2
  MOVE v4 v2
  MOD  v2 v1 v2
  MOVE v1 v4
  LINT v5 loop
  JMPZ v5 R0
done:
  MOVE R1 v1
  JMPZ v0 R0
```

Coloured interference graph



Rewritten program

```
gcd:
  MOVE R3 RLK
  MOVE RLK R1
  MOVE R1 R2
loop:
  LINT R2 done
  JMPZ R2 R1
  MOVE R2 R1
  MOD  R1 RLK R1
  MOVE RLK R2
  LINT R2 loop
  JMPZ R2 R0
done:
  MOVE R1 RLK
  JMPZ R3 R0
```

This second colouring is also correct, but implies worse code!

Colouring by simplification

Colouring by simplification is a heuristic technique to (try to) colour a graph with *K* colours.

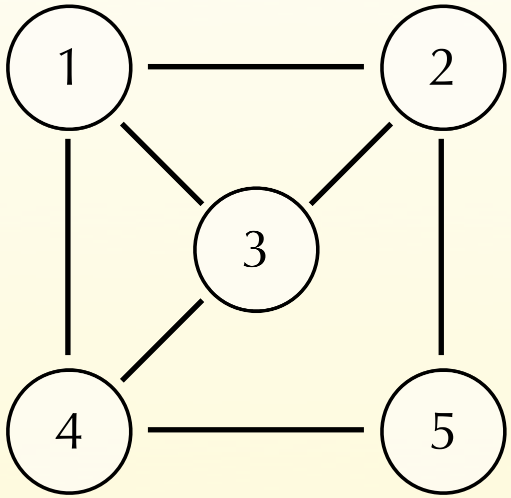
It works as follows: if the graph *G* has at least one node *n* with less than *K* neighbours, *n* is removed from *G*, and that simplified graph is recursively coloured. Once this is done, *n* is coloured with any colour not used by its neighbours.

There is always at least one colour available for *n*, because its neighbours use at most *K-1* colours.

If the graph does not contain a node with less than *K* neighbours, *K*-colouring might not be feasible, but will be attempted nevertheless, as we will see.

Colouring by simplification

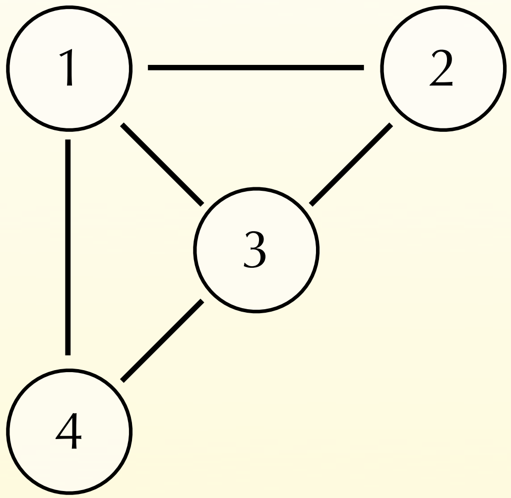
To illustrate colouring by simplification, we can colour the following graph with *K*=3 colours.



Stack of removed nodes:

Colouring by simplification

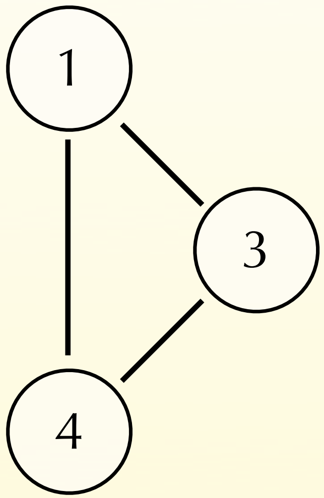
To illustrate colouring by simplification, we can colour the following graph with *K*=3 colours.



Stack of removed nodes: 5

Colouring by simplification

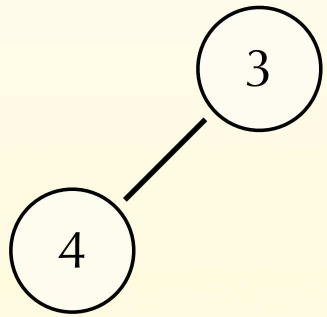
To illustrate colouring by simplification, we can colour the following graph with *K*=3 colours.



Stack of removed nodes: 5 2

Colouring by simplification

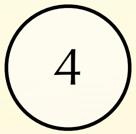
To illustrate colouring by simplification, we can colour the following graph with K=3 colours.



Stack of removed nodes: 5 2 1

Colouring by simplification

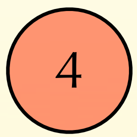
To illustrate colouring by simplification, we can colour the following graph with K=3 colours.



Stack of removed nodes: 5 2 1 3

Colouring by simplification

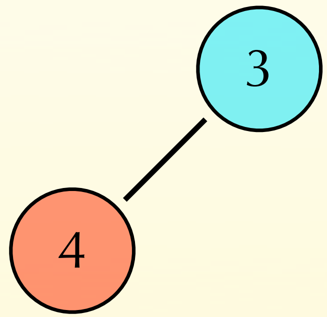
To illustrate colouring by simplification, we can colour the following graph with K=3 colours.



Stack of removed nodes: 5 2 1 3

Colouring by simplification

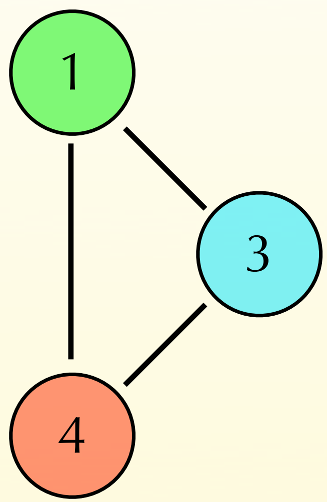
To illustrate colouring by simplification, we can colour the following graph with K=3 colours.



Stack of removed nodes: 5 2 1

Colouring by simplification

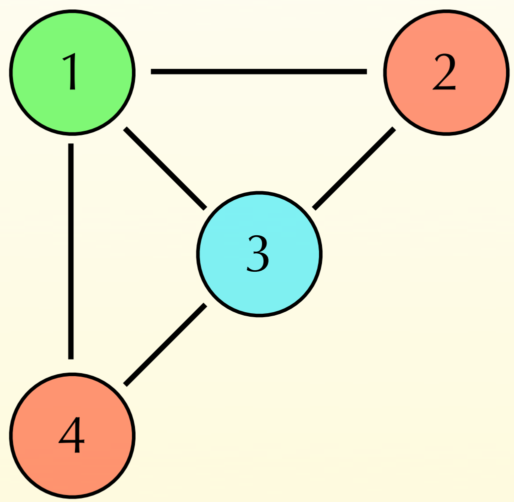
To illustrate colouring by simplification, we can colour the following graph with K=3 colours.



Stack of removed nodes: 5 2

Colouring by simplification

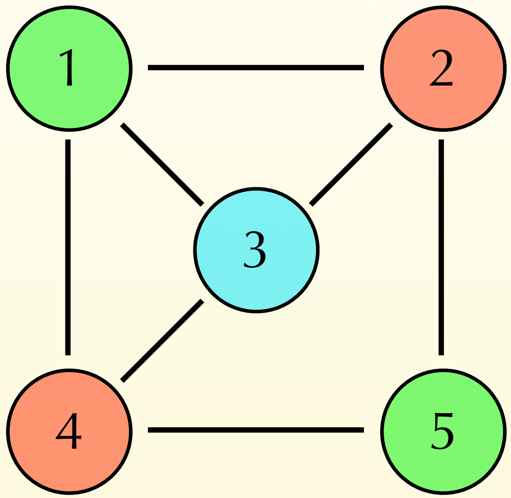
To illustrate colouring by simplification, we can colour the following graph with K=3 colours.



Stack of removed nodes: 5

Colouring by simplification

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



Stack of removed nodes:

Spilling
(in colouring-based allocators)

(Optimistic) spilling

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

When this occurs, a node *n* must be chosen to be **spilled**, *i.e.* have its value stored in memory instead of in a register.

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

After the simplified graph has been coloured, it is actually possible that the neighbours of *n* do not use all the possible colours! In this case, *n* is not spilled. Otherwise it must really be spilled.

Spill costs

The node to spill could be chosen at random, but it is clearly better to favour values that are not frequently used, or values that interfere with many others.

The following formula is often used as a measure of the spill cost for a node *n*. The node with the lowest cost should be spilled first.

$$\text{cost}(n) = [rw_0 + 10\ rw_1 + \ldots + 10^k\ rw_k] \ / \ \text{degree}(n)$$

where rw_i is the number of times the value of *n* is read or written in a loop of depth *i*, and degree(*n*) is the number of edges adjacent to *n* in the interference graph.

Spilling of pre-coloured nodes

As we have seen, the interference graph contains nodes corresponding to the registers of the machine.

These nodes are said to be **pre-coloured**, because the colour of each of them is given by the machine register it represents.

Pre-coloured nodes must never be simplified during the colouring process, as by definition they cannot be spilled.

Spilling example

To illustrate spilling, let's try to colour the same interference graph as before, but with only three colours.

The graph does not contain a node with degree less than three, so the one with the lowest cost must be spilled.

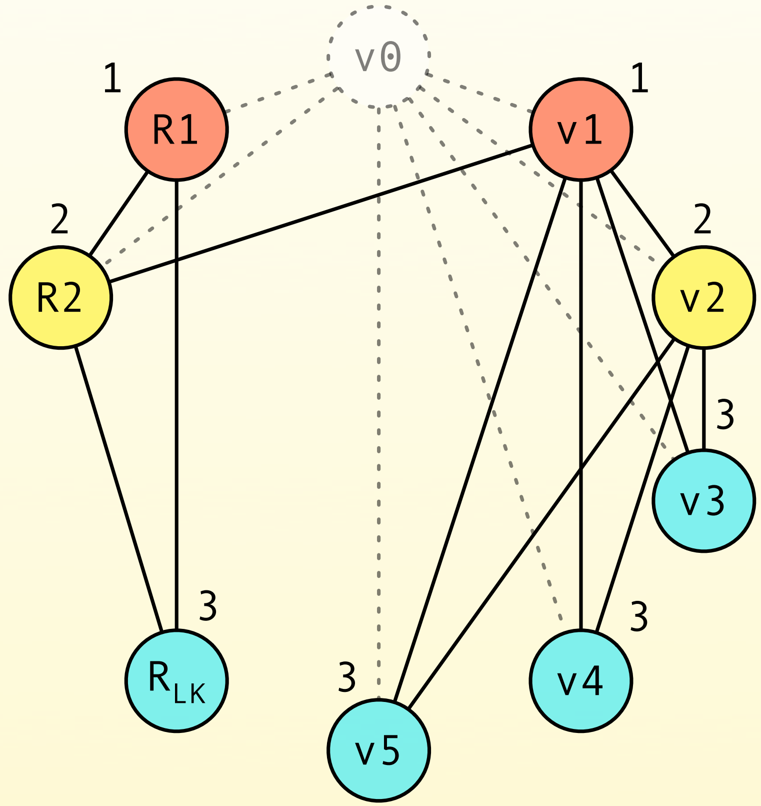
```
gcd:
  MOVE v0 RLK
  MOVE v1 R1
  MOVE v2 R2
loop:
  LINT v3 done
  JMPZ v3 v2
  MOVE v4 v2
  MOD  v2 v1 v2
  MOVE v1 v4
  LINT v5 loop
  JMPZ v5 R0
done:
  MOVE R1 v1
  JMPZ v0 R0
```

node	rw_0	rw_1	degree	cost
v0	2	0	7	0.29
v1	2	2	6	3.67
v2	1	4	6	6.83
v3	0	2	3	6.67
v4	0	2	3	6.67
v5	0	2	3	6.67

$$\text{cost} = (rw_0 + 10\, rw_1) / \text{degree}$$

Spilling example

Once `v0`, which has the lowest spill cost, is removed from the graph, the simplified graph is 3-colourable.



Consequences of spilling

Once a node has been spilled, the original program must be rewritten to take that spilling into account, as follows:

- just before the spilled value is read, code must be inserted to fetch it from memory,
- just after the spilled value is written, code must be inserted to write it back to memory.

Since that spilling code introduces new virtual registers, the whole register allocation process must be restarted from the beginning.

In practice, one or two iterations are enough in almost all cases.

Spilling code integration

Original program

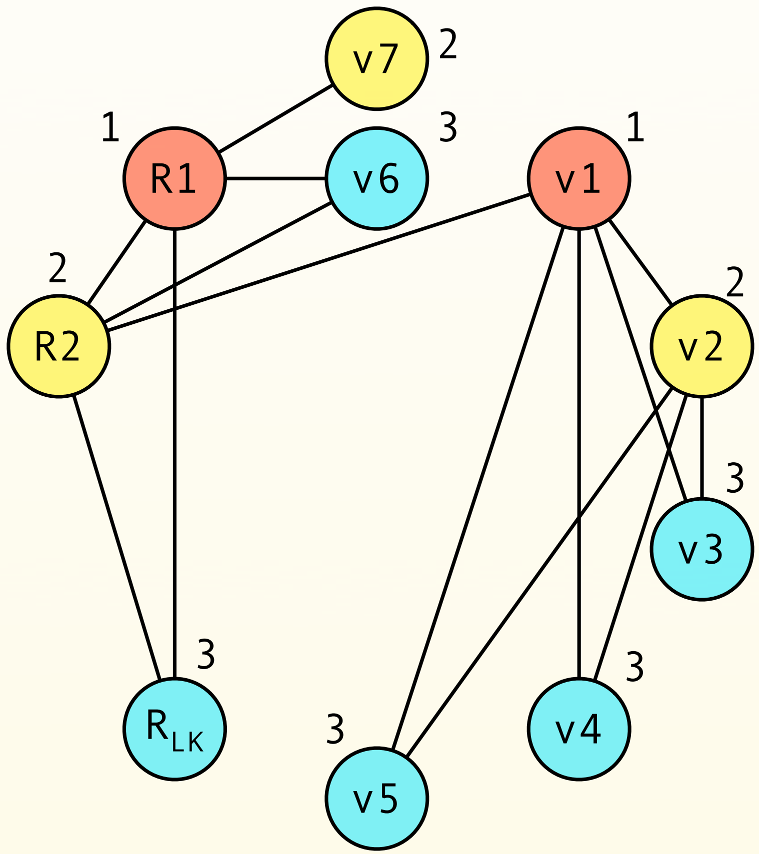
```
gcd:
   MOVE v0 R_LK
   MOVE v1 R1
   MOVE v2 R2
loop:
   LINT v3 done
   JMPZ v3 v2
   MOVE v4 v2
   MOD  v2 v1 v2
   MOVE v1 v4
   LINT v5 loop
   JMPZ v5 R0
done:
   MOVE R1 v1
   JMPZ v0 R0
```

Rewritten program

```
gcd:   ; allocate+link
       ; stack frame
       MOVE v6 R_LK
       STOR v6 R_FP 1
       MOVE v1 R1
       MOVE v2 R2
loop:  LINT v3 done
       JMPZ v3 v2
       MOVE v4 v2
       MOD  v2 v1 v2
       MOVE v1 v4
       LINT v5 loop
       JMPZ v5 R0
done:  MOVE R1 v1
       LOAD v7 R_FP 1
       ; unlink
       ; stack frame
       JMPZ v7 R0
```

New interference graph

Interference graph w/ spilling

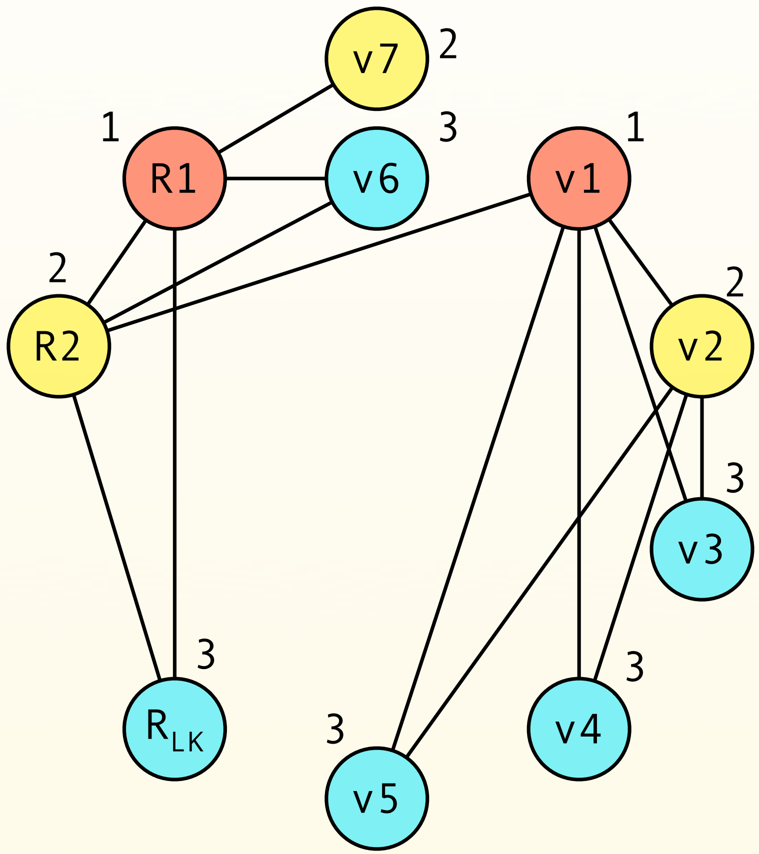


Final program

```
gcd:   ; allocate+link
       ; stack frame
       MOVE RLK RLK
       STOR RLK RFP 1
       MOVE R1 R1
       MOVE R2 R2
loop:  LINT RLK done
       JMPZ RLK R2
       MOVE RLK R2
       MOD  R2 R1 R2
       MOVE R1 RLK
       LINT RLK loop
       JMPZ RLK R0
done:  MOVE R1 R1
       LOAD R2 RFP 1
       ; unlink
       ; stack frame
       JMPZ R2 R0
```

New interference graph

Interference graph w/ spilling



Final program

```
gcd:   ; allocate+link
       ; stack frame
       MOVE RLK RLK
       STOR RLK RFP 1
       MOVE R1 R1
       MOVE R2 R2
loop:  LINT RLK done
       JMPZ RLK R2
       MOVE RLK R2
       MOD  R2 R1 R2
       MOVE R1 RLK
       LINT RLK loop
       JMPZ RLK R0
done:  MOVE R1 R1
       LOAD R2 RFP 1
       ; unlink
       ; stack frame
       JMPZ R2 R0
```

Coalescing (in colouring-based allocators)

Colouring quality

As we have seen in our first example, two valid *K*-colourings of the same interference graph are not necessary equal: one can lead to a much shorter program than the other.

This is due to the fact that a move instruction of the form

```
MOVE v1 v2
```

can be removed after register allocation if `v1` and `v2` end up being allocated to the same register. (Of course, this also holds when `v1` or `v2` is a real register before allocation).

A good register allocator must therefore try to make sure that this happens as often as possible.

Coalescing

Given a `MOVE` instruction of the form

```
MOVE v1 v2
```

and provided that v_1 and v_2 do not interfere, it is always possible to replace all instances of v_1 and v_2 by instances of a new virtual register $v_{1\&2}$. Once this has been done, the `MOVE` instruction becomes useless and can be removed.

This technique is known as **coalescing**, as the nodes of v_1 and v_2 in the interference graph coalesce into a single node.

Coalescing is not always a good idea, though: the coalesced node can have a higher degree than the two original nodes, which might make the graph impossible to colour with K colours and require spilling! Some conservatism is required.

Coalescing heuristics

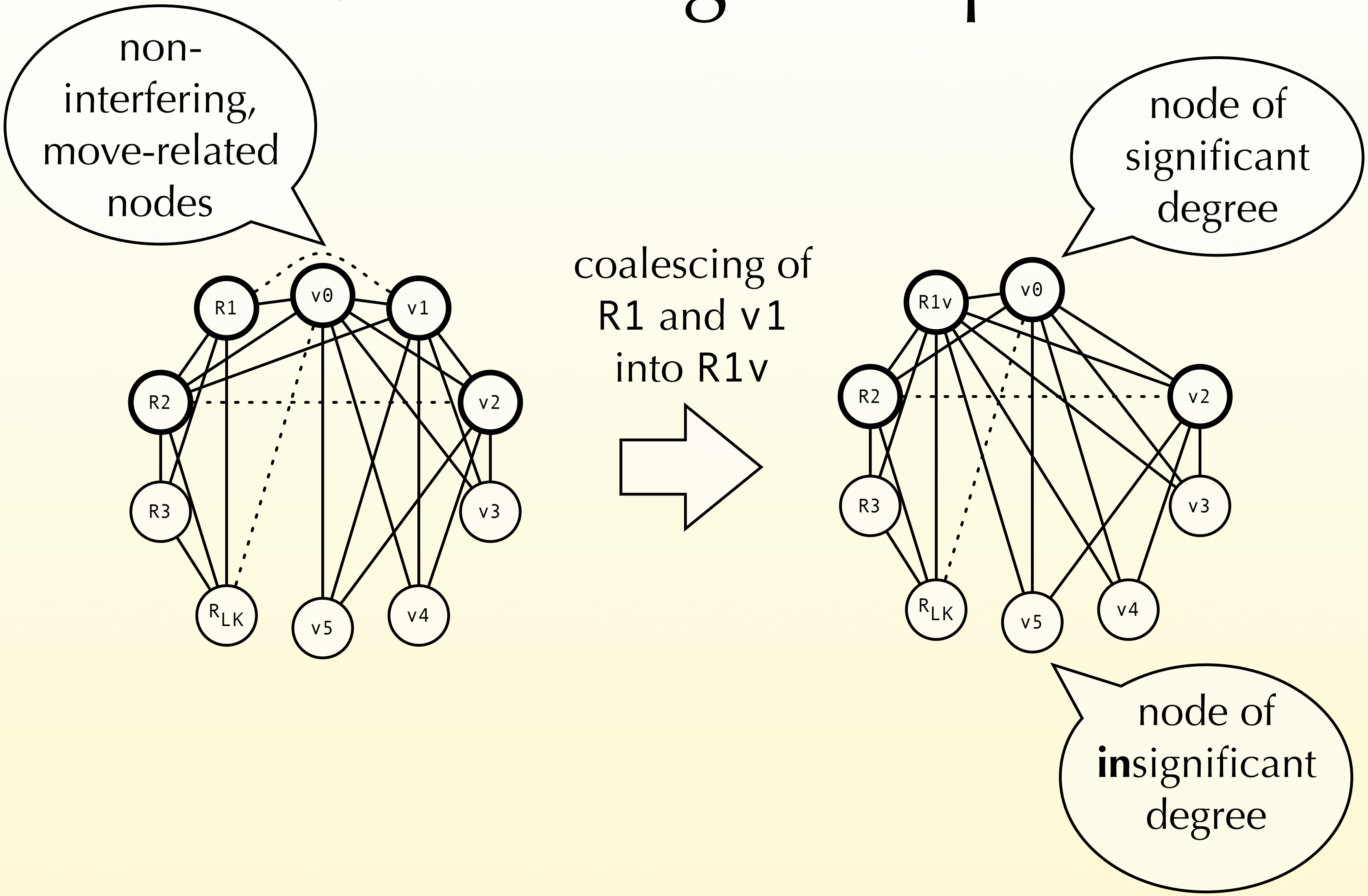
Two coalescing heuristics are commonly used:

Briggs: coalesce nodes n_1 and n_2 iff the resulting node has less than K neighbours of significant degree (*i.e.* of a degree greater or equal to K),

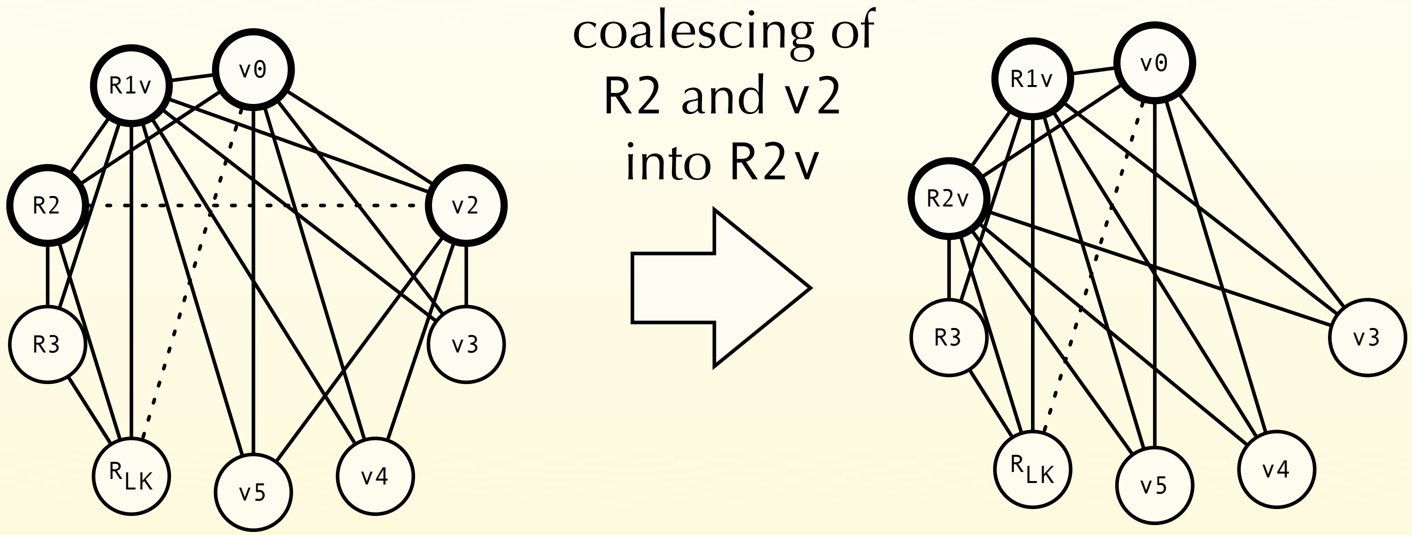
George: coalesce nodes n_1 and n_2 iff all neighbours of n_1 either already interfere with n_2 or are of insignificant degree.

Both heuristics are safe, in that they will not turn a K-colourable graph into a non-K-colourable one. But they are also conservative, in that they might prevent a coalescing that would be safe.

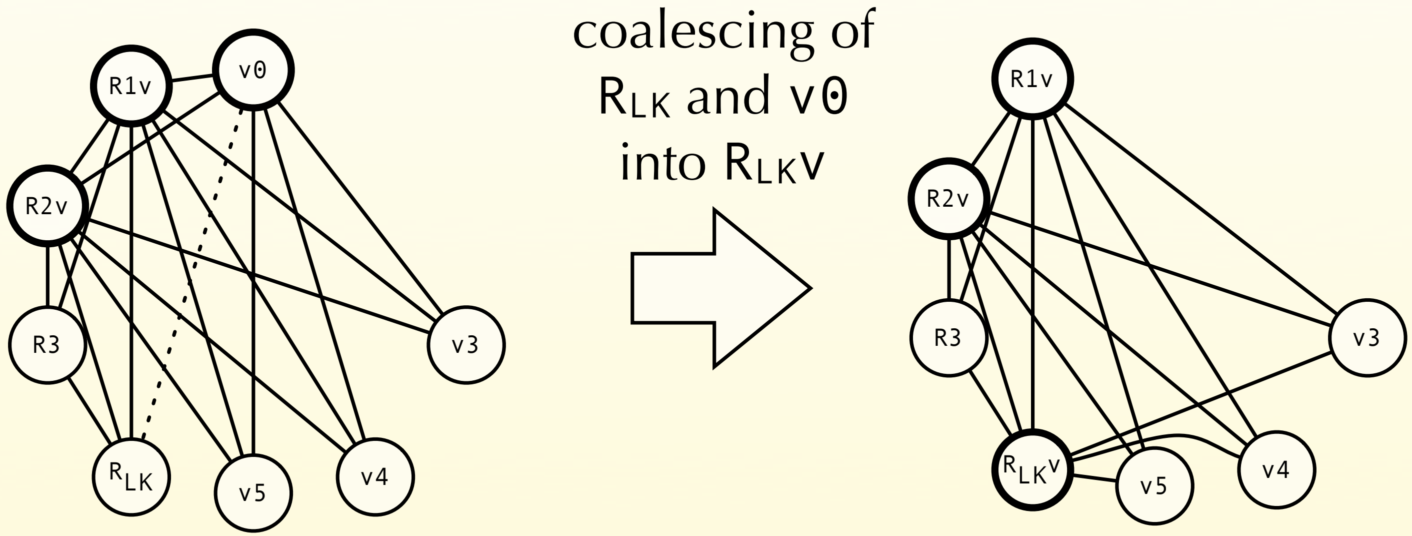
Coalescing example



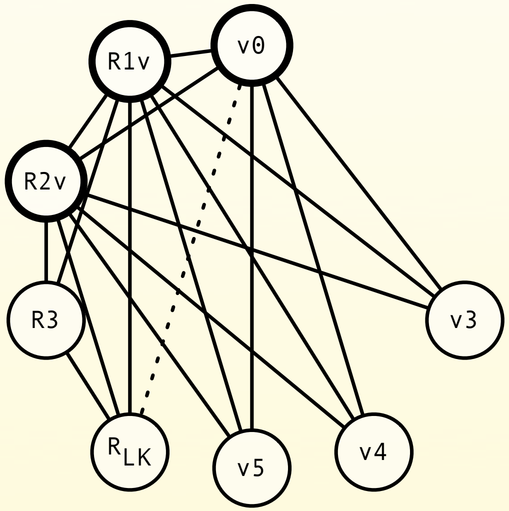
Coalescing example (2)



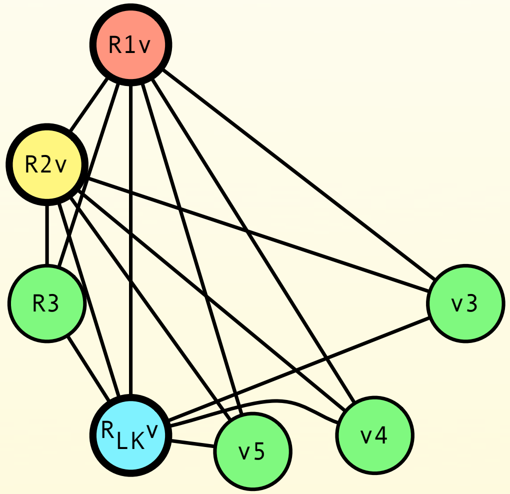
Coalescing example (3)



Coalescing example (3)



coalescing of
R_{LK} and v0
into R_{LK}v



4-colourable

Register classes

Most architectures separate the registers in several classes. Even in modern RISC architectures, there is typically one class for floating-point values and another one for integers and pointers.

Register classes can easily be taken into account in a colouring-based allocator: if a variable must be put in a register of some class A, then its node can be made to interfere with all pre-coloured nodes corresponding to registers of other classes.

Technique #2
Linear scan register allocation

Linear scan

The basic linear scan technique is very simple:

1. the program is linearised – *i.e.* represented as a linear sequence of instructions, not as a graph,
2. a *unique* live range is computed for every variable, going from the first to the last instruction during which it is live,
3. registers are allocated by iterating over the intervals sorted by increasing starting point: each time an interval starts, the next free register is allocated to it, and each time an interval ends, its register is freed,
4. if no register is available, the active range ending *last* is chosen to have its variable spilled.

Linear scan example

Let's try to allocate registers for our gcd procedure using linear scan, first with four allocable registers, then with three.

Program

```
1  gcd:  MOVE v0 RLK
2        MOVE v1 R1
3        MOVE v2 R2
4  loop: LINT v3 done
5        JMPZ v3 v2
6        MOVE v4 v2
7        MOD  v2 v1 v2
8        MOVE v1 v4
9        LINT v5 loop
10       JMPZ v5 R0
11 done: MOVE R1 v1
12       JMPZ v0 R0
```

Live ranges

v0: $[1^+,12^-]$

v1: $[2^+,11^-]$

v2: $[3^+,10^+]$

v3: $[4^+,5^-]$

v4: $[6^+,8^-]$

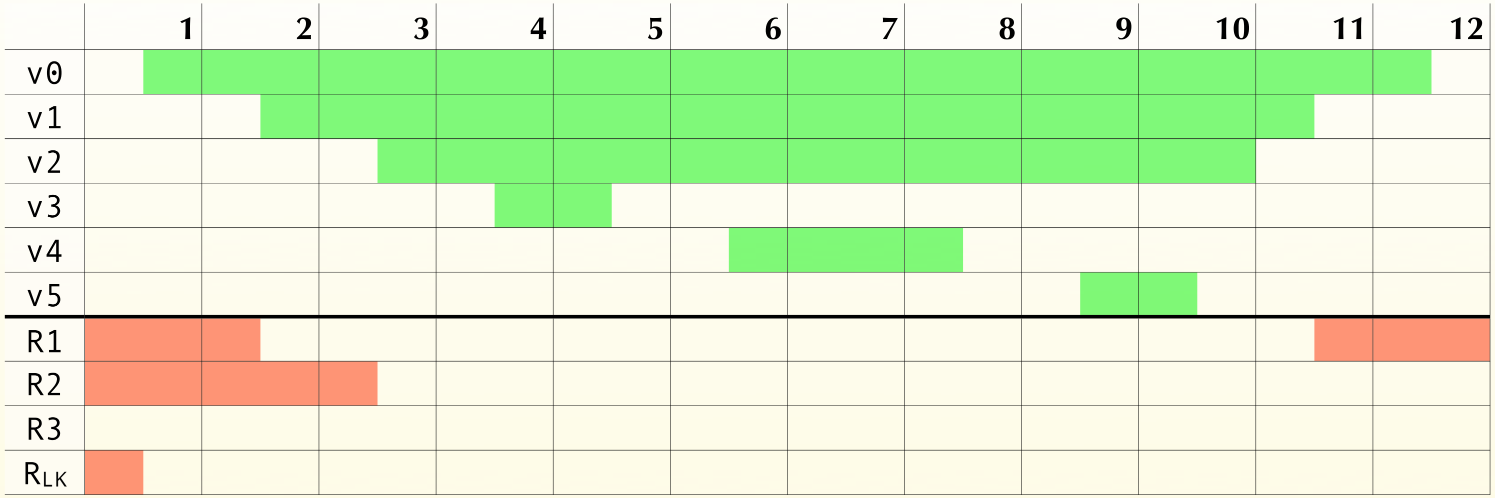
v5: $[9^+,10^-]$

Notation:

i^+ entry of instr. i

i^- exit of instr. i

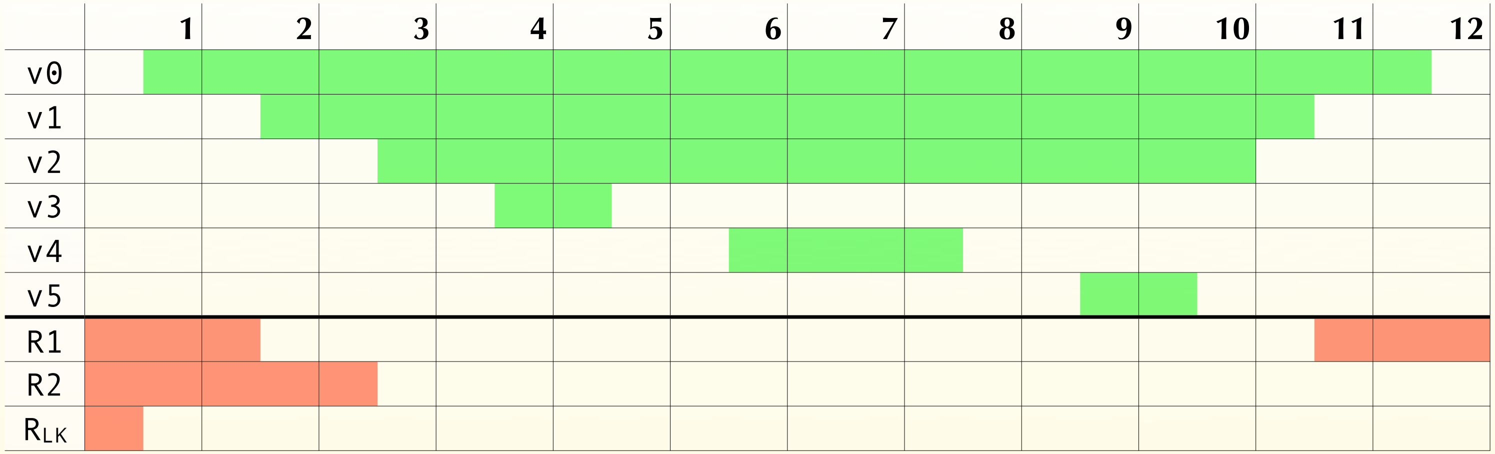
Linear scan example (4 regs)



time	active intervals	allocation
1^+	$[1^+,12^-]$	v0→R3
2^+	$[2^+,11^-],[1^+,12^-]$	v0→R3,v1→R1
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	v0→R3,v1→R1,v2→R2
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	v0→R3,v1→R1,v2→R2,v3→R_{LK}
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	v0→R3,v1→R1,v2→R2,v4→R_{LK}
9^+	$[9^+,10^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	v0→R3,v1→R1,v2→R2,v5→R_{LK}

Result: no spilling

Linear scan example (3 regs)



time	active intervals	allocation
1^+	$[1^+,12^-]$	v0→R_{LK}
2^+	$[2^+,11^-],[1^+,12^-]$	v0→R_{LK},v1→R1
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	v0→R_{LK},v1→R1,v2→R2
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-]$	v0→S,v1→R1,v2→R2,v3→R_{LK}
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-]$	v0→S,v1→R1,v2→R2,v4→R_{LK}
9^+	$[9^+,10^-],[3^+,10^+],[2^+,11^-]$	v0→S,v1→R1,v2→R2,v5→R_{LK}

Result: `v0` is spilled *during its whole life time*!

Linear scan improvements

The basic linear scan algorithm is very simple but still produces reasonably good code. It can be (and has been) improved in many ways:

- the liveness information about virtual registers can be described using a sequence of disjoint intervals instead of a single one,
- virtual registers can be spilled for only a part of their whole life time,
- more sophisticated heuristics can be used to select the virtual register to spill,
- etc.

Summary

Register allocation is probably the most important compiler optimisation.

Most current compilers allocate registers using one of the following two techniques:

1. by transforming the register allocation problem into a graph colouring problem, solved using heuristics,
2. by scanning the live ranges of variables and allocating registers sequentially.

Graph colouring produces the best results but is more complex and slower than the second one. For that reason, graph colouring is usually used in compilers where code quality is more important than compilation speed, and linear scan in the other case.