Register allocation

Advanced Compiler Construction
Michel Schinz – 2016–04–14

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 of them must be **spilled**, i.e. stored in memory instead of in a register.

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

Setting the scene

Register allocation will be performed on an RTL language with the following characteristics:

- apart from n machine registers R_0, …, R_{n-1}, an unbounded number of virtual registers v_0, v_1, … are available before register allocation,
- machine registers that play a special role, like the link register (containing the return address), are identified with a non-numerical index, e.g. R_{LK}; they are real registers nevertheless.

Running example

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

In L_3

```
(defrec gcd
  (fun (a b)
    (if (= 0 b)
        a
        (gcd b (% a b)))))
```

In RTL

```
gcd:  R3 ← done
      if R2 = 0 goto R3
      R3 ← R2
      R2 ← R1 % R2
      R1 ← R3
      R3 ← gcd
      goto R3
done: goto RLK
```

Calling conventions: arguments are passed in R_1, R_2, … and the result is put in R_1.

Register allocation example

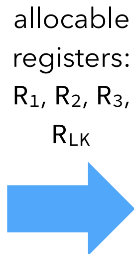
Before register allocation

```
gcd:  v₀ ← R_LK
      v₁ ← R₁
      v₂ ← R₂
loop: v₃ ← done
      if v₂ = 0 goto v₃
      v₄ ← v₂
      v₂ ← v₁ % v₂
      v₁ ← v₄
      v₅ ← loop
      goto v₅
done: R₁ ← v₁
      goto v₀
```

R_1, R_2: parameters

R_{LK}: return address

allocable registers: R_1, R_2, R_3, R_{LK}



After register allocation

```
gcd:
loop: R₃ ← done
      if R₂ = 0 goto R₃
      R₃ ← R₂
      R₂ ← R₁ % R₂
      R₁ ← R₃
      R₃ ← loop
      goto R₃
done: goto R_LK
```

Allocation:

$v_0 \to R_{LK}$

$v_1 \to R_1$

$v_2 \to R_2$

$v_3, v_4, v_5 \to R_3$

Techniques

We will study two commonly used techniques:

- register allocation by **graph coloring**, which is relatively slow but produces very good results,
- **linear scan** register allocation, which is fast but produces worse results.

Because it is slow, graph coloring 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: graph coloring

Allocation by graph coloring

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

- 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.
- The interference graph is colored with at most K colors (K being the number of available registers) so that all nodes have a different color than all their neighbors.

Problems:

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

Interference graph example

Program

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

Liveness
{in}{out}

```
{R1,R2,RLK}{R1,R2,v0}
{R1,R2,v0}{R2,v0,v1}
{R2,v0,v1}{v0-v2}
{v0-v2}{v0-v3}
{v0-v3}{v0-v2}
{v0-v2}{v0-v2,v4}
{v0-v2,v4}{v0-v2,v4}
{v0-v2,v4}{v0-v2}
{v0-v2}{v0-v2,v5}
{v0-v2,v5}{v0-v2}

{v0,v1}{R1,v0}
{R1,v0}{R1}
```

Interference graph

Coloring example

Original prog.

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

Colored interference graph

Rewritten prog.

```
gcd:
  RLK ← RLK   (struck out)
  R1 ← R1     (struck out)
  R2 ← R2     (struck out)
loop:
  R3 ← done
  if R2=0 goto R3
  R3 ← R2
  R2 ← R1 % R2
  R1 ← R3
  R3 ← loop
  goto R3
done:
  R1 ← R1     (struck out)
  goto RLK
```


Coloring example (2)

Original prog.

```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

Colored interference graph

Rewritten prog.

```
gcd:
  R3 ← RLK
  RLK ← R1
  R1 ← R2
loop:
  R2 ← done
  if R1=0 goto R2
  R2 ← R1
  R1 ← RLK % R1
  RLK ← R2
  R2 ← loop
  goto R2
done:
  R1 ← RLK
  goto R3
```

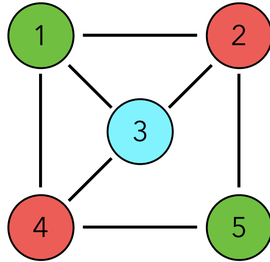
This second coloring is also correct, but produces worse code!

Coloring by simplification

Coloring by simplification is a heuristic technique to (try to) color a graph G with K colors.
It works as follows: if the graph G has at least one node n with less than K neighbors, n is removed from G, and that simplified graph is recursively colored. Once this is done, n is colored with any color not used by its neighbors.
There is always at least one color available for n, because its neighbors use at most K–1 colors.
If the graph does not contain a node with less than K neighbors, K-coloring might not be feasible, but will be attempted nevertheless, as we will see.

Coloring by simplification

To illustrate coloring by simplification, we can color the following graph with K=3 colors.



Stack of removed nodes: 5 2 1 3

Spilling

(Optimistic) spilling

During simplification, it is perfectly possible to reach a point where all nodes have at least K neighbors.
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 color the simplified graph as usual.
After the simplified graph has been colored, it is actually possible that the neighbors of n do not use all the possible colors! 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 favor 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.

$$cost(n) = (rw_0(n) + 10\ rw_1(n) + \ldots + 10^k\ rw_k(n)) / degree(n)$$

where $rw_i(n)$ 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-colored nodes

As we have seen, the interference graph contains nodes corresponding to the registers of the machine.
These nodes are said to be **pre-colored**, because the color of each of them is given by the machine register it represents.
Pre-colored nodes must never be simplified during the coloring process, as by definition they cannot be spilled.

Spilling example

Let's try to color the same interference graph as before, but with only three colors. There is no node with degree less than three, so the one with the lowest cost must be spilled.

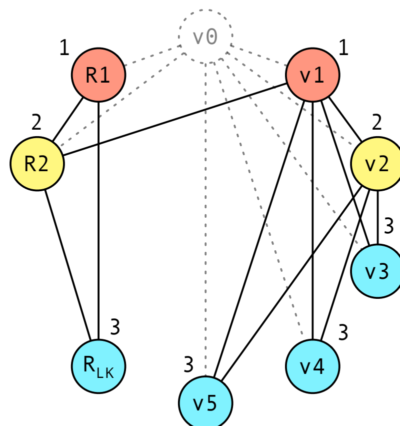
```
gcd:
  v₀ ← R_LK
  v₁ ← R₁
  v₂ ← R₂
loop:
  v₃ ← done
  if v₂=0 goto v₃
  v₄ ← v₂
  v₂ ← v₁ % v₂
  v₁ ← v₄
  v₅ ← loop
  goto v₅
done:
  R₁ ← v₁
  goto v₀
```

node	rw_0	rw_1	deg.	cost
v_0	2	0	7	0.29
v_1	2	2	6	3.67
v_2	1	4	6	6.83
v_3	0	2	3	6.67
v_4	0	2	3	6.67
v_5	0	2	3	6.67

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

Spilling example

Once v_0, which has the lowest spill cost, is removed from the graph, the simplified graph is 3-colorable.



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:
  v₀ ← R_LK
  v₁ ← R₁
  v₂ ← R₂
loop:
  v₃ ← done
  if v₂ = 0 goto v₃
  v₄ ← v₂
  v₂ ← v₁ % v₂
  v₁ ← v₄
  v₅ ← loop
  goto v₅
done:
  R₁ ← v₁
  goto v₀
```

spilling of v_0 →

Rewritten program

```
gcd:
  v₆ ← R_LK
  push v₆
  v₁ ← R₁
  v₂ ← R₂
loop:
  v₃ ← done
  if v₂ = 0 goto v₃
  v₄ ← v₂
  v₂ ← v₁ % v₂
  v₁ ← v₄
  v₅ ← loop
  goto v₅
done:
  R₁ ← v₁
  pop v₇
  goto v₇
```


New interference graph

Interference graph w/ spilling

Final program

```
gcd:
  R_LK ← R_LK   (struck out)
  push R_LK
  R₁ ← R₁       (struck out)
  R₂ ← R₂       (struck out)
loop:
  R_LK ← done
  if R₂ = 0 goto R_LK
  R_LK ← R₂
  R₂ ← R₁ % R₂
  R₁ ← R_LK
  R_LK ← loop
  goto R_LK
done:
  R₁ ← R₁
  pop R₂
  goto R₂
```


Coalescing

Coloring quality

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

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

```
v₁ ← v₂
```

can be removed after register allocation if v_1 and v_2 end up being allocated to the same register. (Of course, this also holds when v_1 or v_2 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

$v_1 \leftarrow v_2$

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 issue

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 color with K colors and require spilling!

Conservative coalescing heuristics have to be used.

Coalescing heuristics

Two coalescing heuristics are commonly used:

- **Briggs**: coalesce nodes n_1 and n_2 to $n_{1\&2}$ iff $n_{1\&2}$ has less than K neighbors of significant degree (i.e. of a degree greater or equal to K),
- **George**: coalesce nodes n_1 and n_2 to $n_{1\&2}$ iff all neighbors 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-colorable graph into a non-K-colorable one. But they are also conservative, in that they might prevent a coalescing that would be safe.

Heuristic #1: Briggs

Briggs' heuristic: coalesce nodes n_1 and n_2 to $n_{1\&2}$ iff $n_{1\&2}$ has less than K neighbors of significant degree (i.e. of degree $\geq$ K).

Rationale: during simplification, all the neighbors of $n_{1\&2}$ that are of insignificant degree will be simplified; at this point, $n_{1\&2}$ will have less than K neighbors and will therefore be simplifiable too.

This heuristic is safe, in that it will not turn a K-colorable graph into a non-K-colorable one. But it is also conservative, in that it might prevent a coalescing that would be safe.

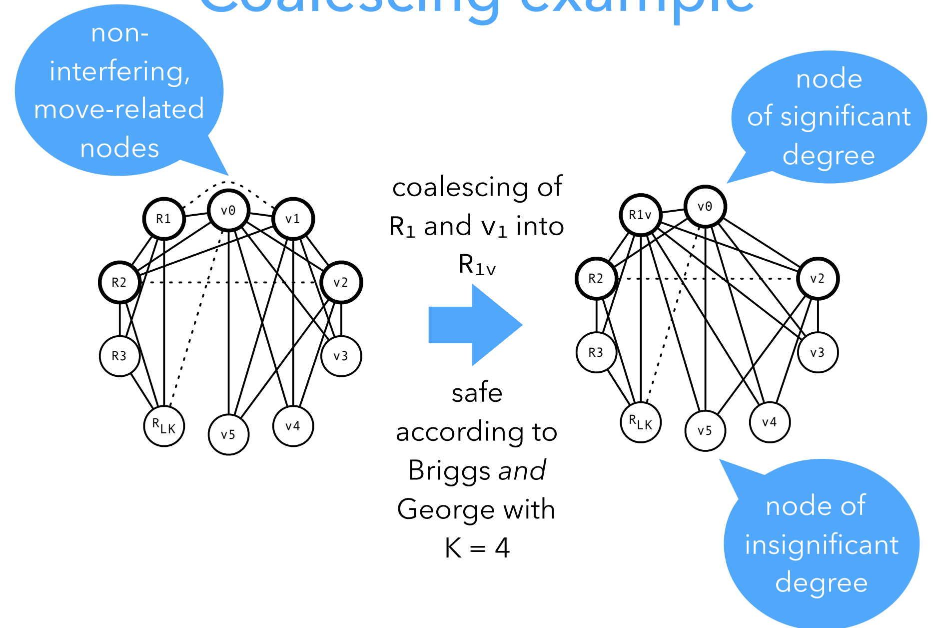
Heuristic #2: George

George's heuristic: coalesce nodes n_1 and n_2 to $n_{1\&2}$ iff all neighbors of n_1 either already interfere with n_2 or are of insignificant degree.

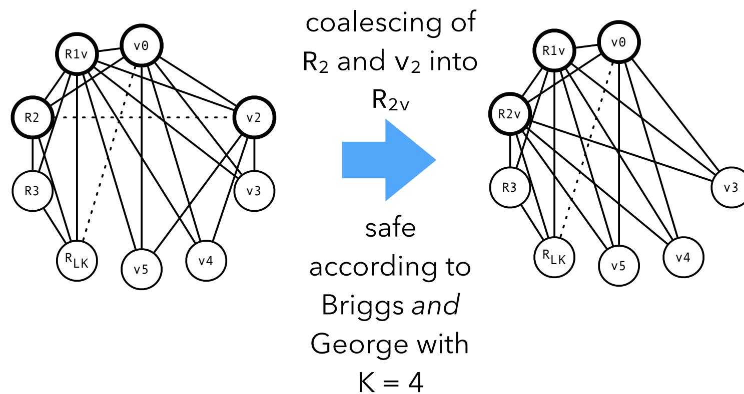
Rationale: the neighbors of $n_{1\&2}$ will be the same as the neighbors of n_2, plus all neighbors of n_1 that are of insignificant degree. The latter ones will all be simplified, at which point the graph will be a sub-graph of the original one.

Like Briggs', George's heuristic is safe but conservative.

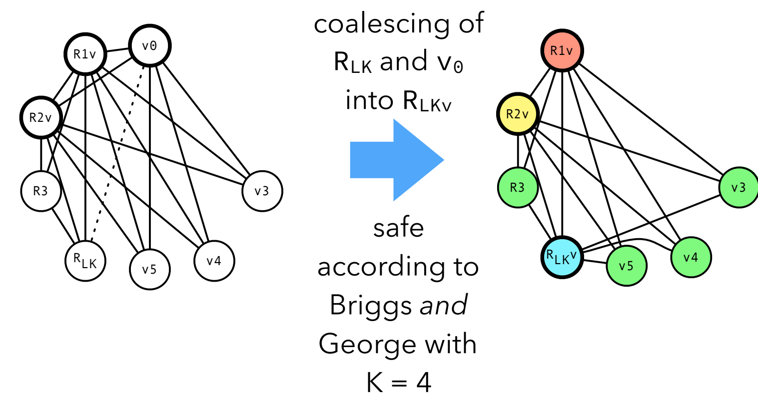
Coalescing example



Coalescing example (2)



Coalescing example (3)



Putting it all together

Iterated register coalescing

To get the best results, the phases of simplification and coalescing should be interleaved. This technique is known as **iterated register coalescing** (**IRC**):

1. Interference graph nodes are partitioned in two classes: move-related or not.
2. Simplification is done on *not* move-related nodes (as move-related ones could be coalesced).
3. Conservative coalescing is performed.
4. When neither simplification nor coalescing can proceed further, some move-related nodes are **frozen** (marked as non-move-related).
5. The process is restarted at 2.

Iterated register coalescing

build
simplify
coalesce
freeze
potential spill
select
actual spill
in case of actual spill

Assignment constraints

Assignment constraints

Until now, we have assumed that a virtual register can be assigned to any physical register, as long as it is free.

In practice, this is often not the case, as various architectural characteristics impose **assignment constraints**, e.g.:

- some architectures divide the registers in several classes, with different capabilities (e.g. address vs. data registers, integer vs. floating-point registers, etc.),
- some instructions require some of their arguments – or their result – to be in specific registers,
- calling conventions require function arguments and results to be in specific registers.

A realistic register allocator has to be able to satisfy these constraints.

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 coloring-based allocator: if a variable must be put in a register of some class, then its node can be made to interfere with all pre-colored nodes corresponding to registers of other classes.

Calling conventions

Many calling conventions pass arguments in registers.

At the beginning of all functions, move instructions have to be inserted to copy the arguments to new virtual registers:

```
fact:
  v₁ ← R₁   ; save first argument in v₁
```

Similarly, before any function call, move instructions have to be inserted to load the arguments in the appropriate registers:

```
R₁ ← v₂     ; load first argument from v₂
CALL fact
```

Whenever possible, theses move instructions will be removed by coalescing.

Caller/callee-saved registers

Calling conventions distinguish two kinds of registers:

- **caller-saved** registers are saved by the caller before a call and restored after it,
- **callee-saved** registers are saved by the callee at function entry and restored before function exit.

Ideally, all virtual registers that have to survive at least one call should be assigned to callee-saved registers, while other virtual registers should be assigned to caller-saved registers.

How can this be obtained in a coloring-based allocator?

Caller/callee-saved registers

The contents of caller-saved registers do not survive a function call. To model this, edges are added to the interference graph between all virtual registers that are live across at least one call and (physical) caller-saved registers. These edges ensure that virtual registers that are live across at least one call will not be assigned to caller-saved registers, and will therefore either be spilled or allocated to callee-saved registers!

Saving callee-saved registers

Callee-saved registers must be preserved by all functions. This can be achieved by copying them to fresh temporary registers at function entry and restoring them before exit.

Saving callee-saved registers

For example, if R_8 is a callee-saved register, a function could look like:

```
entry:
  v₁ ← R₈   ; save callee-saved R₈ in v₁
  …         ; function body
  R₈ ← v₁   ; restore callee-saved R₈
  goto R_LK
```

If the register pressure is low, then R_8 and v_1 will be coalesced, and the two move instructions removed. If register pressure is high, v_1 will be spilled, thereby making R_8 available in the function body, e.g. to store a virtual register live across a call.

Technique #2: linear scan

Linear scan

The basic linear scan technique is very simple:

- the program is linearized – i.e. represented as a linear sequence of instructions, not as a graph,
- a unique live range is computed for every variable, going from the first to the last instruction during which it is live,
- 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,
- 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

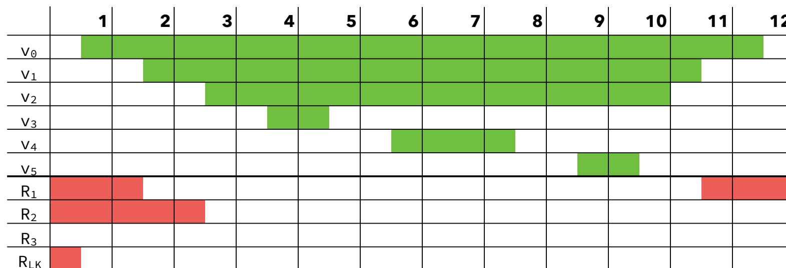
```
1  gcd:  v0 ← RLK
2        v1 ← R1
3        v2 ← R2
4  loop: v3 ← done
5        if v2=0 goto v3
6        v4 ← v2
7        v2 ← v1 % v2
8        v1 ← v4
9        v5 ← loop
10       goto v5
11 done: R1 ← v1
12       goto v0
```

Live ranges

v_0: $[1^+,12^-]$
v_1: $[2^+,11^-]$
v_2: $[3^+,10^+]$
v_3: $[4^+,5^-]$
v_4: $[6^+,8^-]$
v_5: $[9^+,10^-]$

Notation:
i^+ entry of instr. i
i^- exit of instr. i

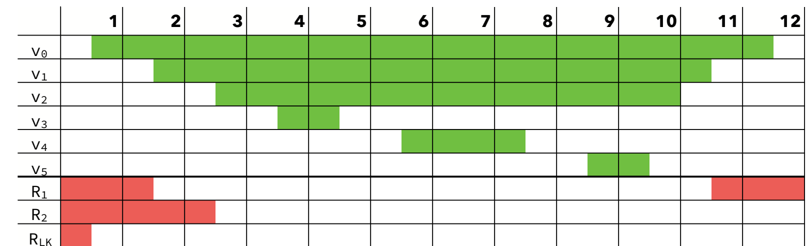
Linear scan example (4 r.)



time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \to R_3$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \to R_3, v_1 \to R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \to R_3, v_1 \to R_1, v_2 \to R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \to R_3, v_1 \to R_1, v_2 \to R_2, v_3 \to R_{LK}$
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \to R_3, v_1 \to R_1, v_2 \to R_2, v_4 \to R_{LK}$
9^+	$[9^+,10^-],[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \to R_3, v_1 \to R_1, v_2 \to R_2, v_5 \to R_{LK}$

Result: no spilling

Linear scan example (3 r.)



time	active intervals	allocation
1^+	$[1^+,12^-]$	$v_0 \to R_{LK}$
2^+	$[2^+,11^-],[1^+,12^-]$	$v_0 \to R_{LK}, v_1 \to R_1$
3^+	$[3^+,10^+],[2^+,11^-],[1^+,12^-]$	$v_0 \to R_{LK}, v_1 \to R_1, v_2 \to R_2$
4^+	$[4^+,5^-],[3^+,10^+],[2^+,11^-]$	$v_0 \to S, v_1 \to R_1, v_2 \to R_2, v_3 \to R_{LK}$
6^+	$[6^+,8^-],[3^+,10^+],[2^+,11^-]$	$v_0 \to S, v_1 \to R_1, v_2 \to R_2, v_4 \to R_{LK}$
9^+	$[9^+,10^-],[3^+,10^+],[2^+,11^-]$	$v_0 \to S, v_1 \to R_1, v_2 \to R_2, v_5 \to R_{LK}$

Result: v_0 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.