

Register allocation

Michel Schinz
Advanced Compiler Construction – 2009-05-22

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.

2

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, \dots, R_n , an unbounded number of virtual registers V_0, V_1, \dots 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** $R_1 R_2 5$.

3

Example function

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

<pre>In minischeme (define gcd (lambda (a b) (if (= 0 b) a (gcd b (% a b))))))</pre>	<pre>In (hand-coded) assembly gcd: LINT R3 done JEQ R3 R2 R0 ADD R3 R2 R0 MOD R2 R1 R2 ADD R1 R3 R0 LINT R3 gcd JEQ R3 R0 R0 done: JEQ R1K R0 R0</pre>
--	--

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

4

Register allocation example

Before register allocation

```
gcd: MOVE V0 R1K
     MOVE V1 R1
     MOVE V2 R2
loop: LINT V3 done
     JEQ V3 V2 R0
     MOVE V4 V2
     MOD V2 V1 V2
     MOVE V1 V4
     LINT V5 loop
     JEQ V5 R0 R0
done: MOVE R1 V1
     JEQ V0 R0 R0
```

R_0 : zero
 R_1, R_2 : parameters
 R_{LK} : return address

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



After register allocation

```
gcd: loop: LINT R3 done
     JEQ R3 R2 R0
     MOVE R3 R2
     MOD R2 R1 R2
     MOVE R1 R3
     LINT R3 loop
     JEQ R3 R0 R0
done: JEQ R1K R0 R0
```

Allocation:

$V_0 \rightarrow R_{LK}$
 $V_1 \rightarrow R_1$
 $V_2 \rightarrow R_2$
 $V_3, V_4, V_5 \rightarrow R_3$

5

Register allocation techniques

We will study the two most 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 slightly worse results – at least in its standard form.

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.

6

Technique #1: Register allocation by graph coloring

Allocation by graph coloring

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

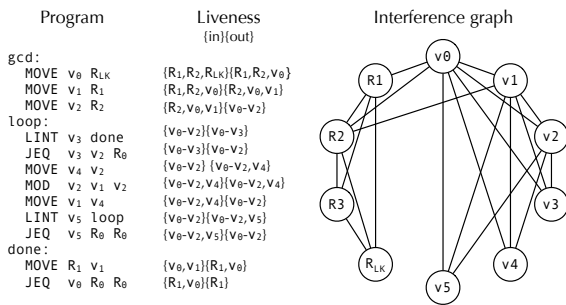
1. The **interference graph** is built. It has one node per register (real or virtual), and two nodes are connected by an edge iff their registers are simultaneously live.
2. The interference graph is colored with at most K colors – K being the number of available registers – so that all nodes have a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

Problems:

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

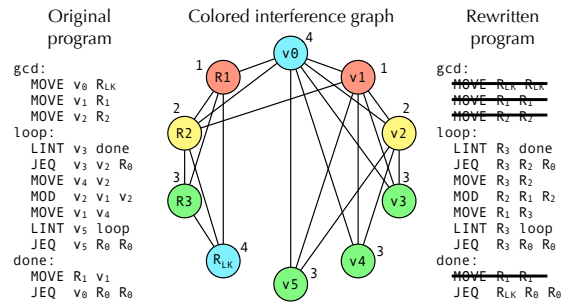
8

Interference graph example



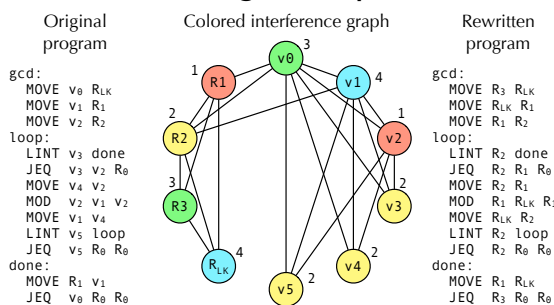
9

Coloring example



10

Coloring example (2)



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

11

Coloring by simplification

Coloring by simplification is a heuristic technique to (try to) color a graph 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.

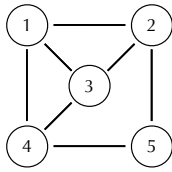
There is always at least one color available for n , because its neighbors use at most $K-1$ colors.

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.

12

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

13

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.

15

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.

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

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

16

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.

17

Spilling example

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

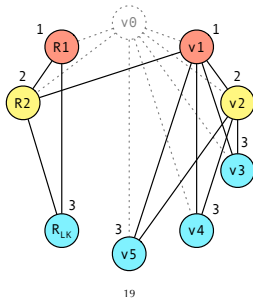
The graph does not contain a node with degree less than three, so the one with the lowest cost must be spilled.

	node	rw_0	rw_1	degree	cost
gcd:	v_0	2	0	7	0.29
MOVE	v_1	2	2	6	3.67
MOVE	v_2	1	4	6	6.83
loop:	v_3	0	2	3	6.67
LINT	v_4	0	2	3	6.67
JEQ	v_5	0	2	3	6.67
done:		$\text{cost} = (rw_0 + 10 rw_1) / \text{degree}$			
MOVE	R_1				
JEQ	R_0				

18

Spilling example

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



19

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.

20

Spilling code integration

Original program

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



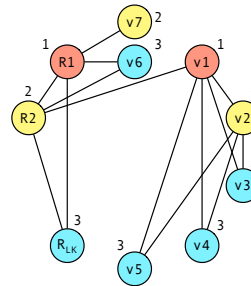
Rewritten program

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

21

New interference graph

Interference graph w/ spilling



Final program

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

22

Coalescing

Coloring quality

As we have seen in our first example, two valid K -colorings of the same interference graph are not necessarily 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 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.

24

Coalescing

Given a MOVE instruction of the form

MOVE v_1 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 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.

25

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.

26

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.

27

Heuristic #2: George

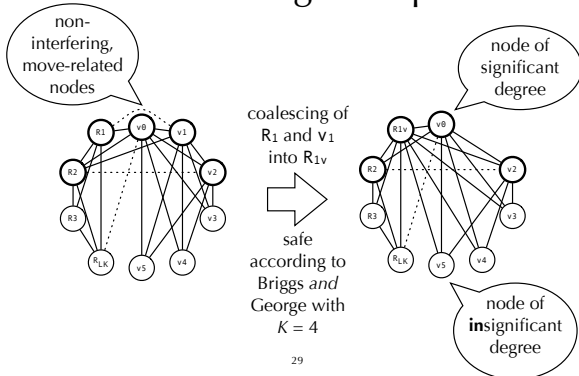
George's heuristic: coalesce nodes n_1 and n_2 to $n_{1\&2}$ iff all neighbors of n_1 either already interfere with n_2 or are of insignificant degree.

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.

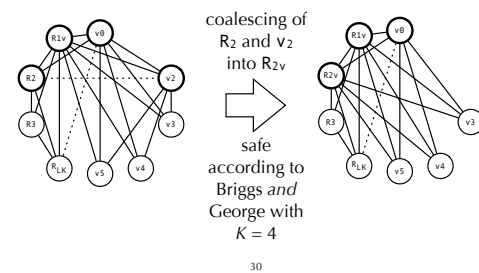
Like Briggs', George's heuristic is safe but conservative.

28

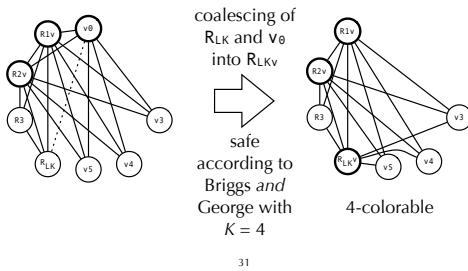
Coalescing example



Coalescing example (2)



Coalescing example (3)



Putting it all together

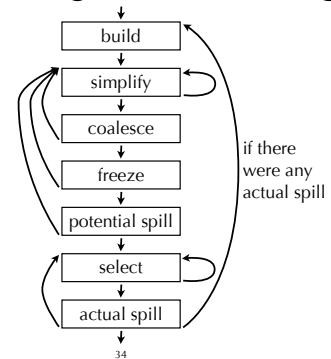
Iterated register coalescing

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

- the nodes of the interference graph are partitioned in two classes, depending on whether they are **move-related** or not – a node is move-related if its register is the source or target of a MOVE instruction,
- simplification is done only on nodes that are *not* move-related – the idea being that move-related nodes could be coalesced and should not be simplified (yet),
- coalescing is performed conservatively,
- when neither simplification nor coalescing can proceed further, some move-related nodes are **frozen**, i.e. marked as non-move-related so that they can be simplified.

33

Iterated register coalescing



Handling 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 architecture 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.

36

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.

37

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, e.g.:

```
fact:
```

```
    MOVE 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:

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

Whenever possible, these MOVE instructions will be removed by coalescing.

38

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?

39

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!

40

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.

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

```
entry:  
    MOVE v1 R8 ; save callee-saved R8 in v1  
    ; ... function body  
    MOVE R8 v1 ; restore callee-saved R8  
    RET
```

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.

41

Technique #2 Linear scan register allocation

Linear scan

The basic linear scan technique is very simple:

1. the program is linearized – *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.

43

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	Live ranges
1 gcd: MOVE v0 R _{LK}	v0: [1 ⁺ ,12 ⁻]
2 MOVE v1 R ₁	v1: [2 ⁺ ,11 ⁻]
3 MOVE v2 R ₂	v2: [3 ⁺ ,10 ⁺]
4 loop: LINT v3 done	v3: [4 ⁺ ,5 ⁻]
5 JEQ v3 v2 R ₀	v4: [6 ⁺ ,8 ⁻]
6 MOVE v4 v2	v5: [9 ⁺ ,10 ⁻]
7 MOD v2 v1 v2	
8 MOVE v1 v4	
9 LINT v5 loop	
10 JEQ v5 R ₀ R ₀	Notation:
11 done: MOVE R ₁ v1	<i>i</i> ⁺ entry of instr. <i>i</i>
12 JEQ v0 R ₀ R ₀	<i>i</i> ⁻ exit of instr. <i>i</i>

44

Linear scan example (4 regs)

	1	2	3	4	5	6	7	8	9	10	11	12
v0												
v1												
v2												
v3												
v4												
v5												
R1												
R2												
R3												
R _{LK}												

time	active intervals	allocation
1 ⁺	[1 ⁺ ,12 ⁻]	v0 → R ₃
2 ⁺	[2 ⁺ ,11 ⁻], [1 ⁺ ,12 ⁻]	v0 → R ₃ , v1 → R ₁
3 ⁺	[3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻], [1 ⁺ ,12 ⁻]	v0 → R ₃ , v1 → R ₁ , v2 → R ₂
4 ⁺	[4 ⁺ ,5 ⁻], [3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻], [1 ⁺ ,12 ⁻]	v0 → R ₃ , v1 → R ₁ , v2 → R ₂ , v3 → R _{LK}
6 ⁺	[6 ⁺ ,8 ⁻], [3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻], [1 ⁺ ,12 ⁻]	v0 → R ₃ , v1 → R ₁ , v2 → R ₂ , v4 → R _{LK}
9 ⁺	[9 ⁺ ,10 ⁻], [3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻], [1 ⁺ ,12 ⁻]	v0 → R ₃ , v1 → R ₁ , v2 → R ₂ , v5 → R _{LK}

Result: no spilling

Linear scan example (3 regs)

	1	2	3	4	5	6	7	8	9	10	11	12
v0												
v1												
v2												
v3												
v4												
v5												
R1												
R2												
R _{LK}												

time	active intervals	allocation
1 ⁺	[1 ⁺ ,12 ⁻]	v0 → R _{LK}
2 ⁺	[2 ⁺ ,11 ⁻], [1 ⁺ ,12 ⁻]	v0 → R _{LK} , v1 → R ₁
3 ⁺	[3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻], [1 ⁺ ,12 ⁻]	v0 → R _{LK} , v1 → R ₁ , v2 → R ₂
4 ⁺	[4 ⁺ ,5 ⁻], [3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻]	v0 → S, v1 → R ₁ , v2 → R ₂ , v3 → R _{LK}
6 ⁺	[6 ⁺ ,8 ⁻], [3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻]	v0 → S, v1 → R ₁ , v2 → R ₂ , v4 → R _{LK}
9 ⁺	[9 ⁺ ,10 ⁻], [3 ⁺ ,10 ⁺], [2 ⁺ ,11 ⁻]	v0 → S, v1 → R ₁ , v2 → R ₂ , 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.

47

Summary

Register allocation is probably the most important compiler optimization.

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

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

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

48