CS107: Register Allocation

Guannan Wei
guannan.wei@tufts.edu

March 26, 2026

Spring 2026

Tufts University

Recap

Middle-end phases of compilation:

- continuation-passing style
- machine-independent optimizations (Project 6)
- analysis over the IR

Recap

Middle-end phases of compilation:

- continuation-passing style
- machine-independent optimizations (Project 6)
- analysis over the IR

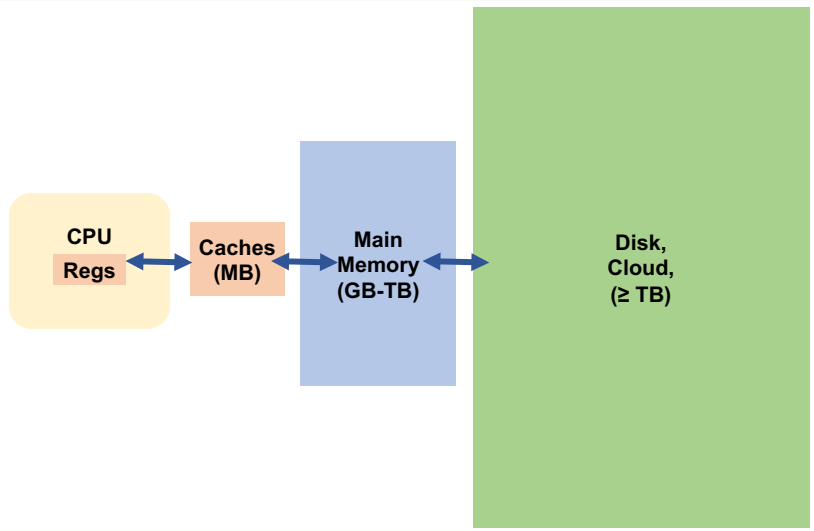
Back-end phases of compilation:

- register allocation
- instruction scheduling

Runtime:

- garbage collection

Memory Hierarchy



Register allocation

The problem of **register allocation** is to transform a program that makes use of an unbounded number of local variables (**virtual** or **pseudo-registers**) into one that only makes use of machine registers.

- If there are not enough machine registers to store all variables, some 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 very 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, such as 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 the source language:

```
def gcd(a: Int, b: Int) =
  if (0 == b)
   a
  else
   gcd(b, a % b)
```

In register-transfer language:

```
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, ... result is in R_1.

Register Allocation Example

Before register allocation

```
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
```

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 RLK
```

Allocation:

$v_0 \rightarrow R_{LK}$

$v_1 \rightarrow R_1$

$v_2 \rightarrow R_2$

$v_3, v_4, v_5 \rightarrow R_3$

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: Allocation By Graph Coloring

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

- Build the *interference graph*:
  - assign a register (physical or virtual) for each node,
  - and two nodes are connected by an edge iff their registers are simultaneously live.
- The interference graph is colored with at most K colors, so that all nodes have a different color than their neighbors.
  - K is the number of available registers

Technique #1: Allocation By Graph Coloring

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

- Build the *interference graph*:
  - assign a register (physical or virtual) for each node,
  - and two nodes are connected by an edge iff their registers are simultaneously live.
- The interference graph is colored with at most K colors, so that all nodes have a different color than their neighbors.
  - K is the number of available registers

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

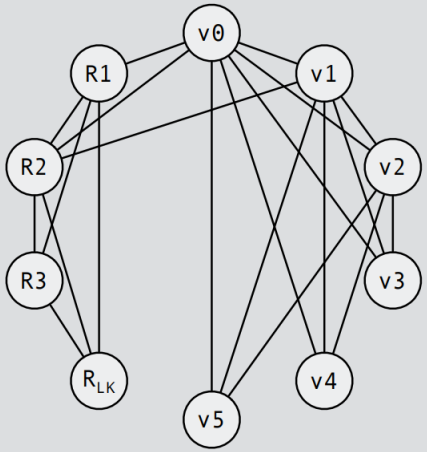
{in}{out}

{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

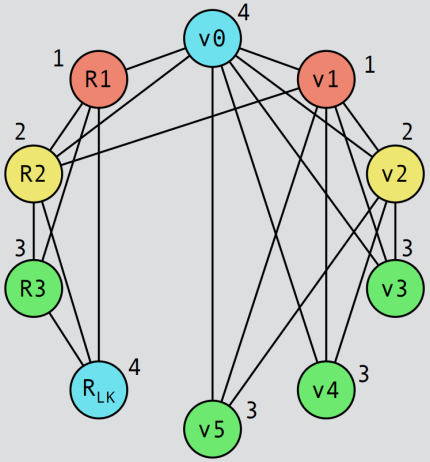


Coloring Example

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.

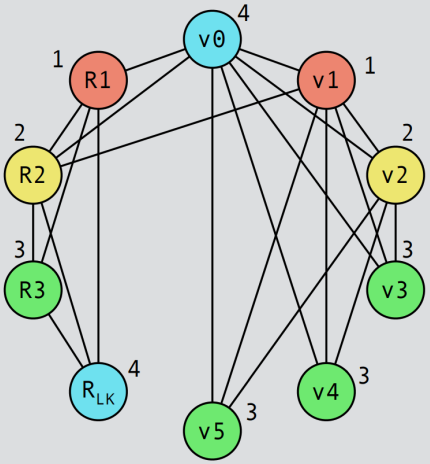
```
gcd:
  RLK ← RLK
  R1 ← R1
  R2 ← R2
loop:
  R3 ← done
  if R2=0 goto R3
  R3 ← R2
  R2 ← R1 % R2
  R1 ← R3
  R3 ← loop
  goto R3
done:
  R1 ← R1
  goto RLK
```

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.

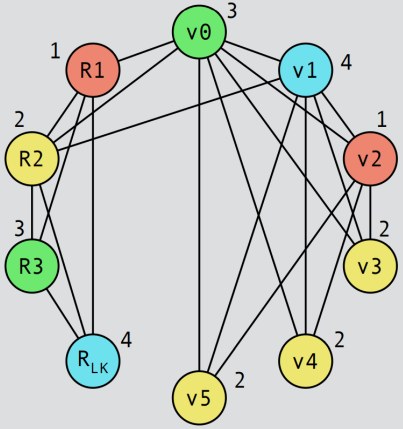
```
gcd:
  RLK ← RLK
  R1 ← R1
  R2 ← R2
loop:
  R3 ← done
  if R2=0 goto R3
  R3 ← R2
  R2 ← R1 % R2
  R1 ← R3
  R3 ← loop
  goto R3
done:
  R1 ← R1
  goto RLK
```

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

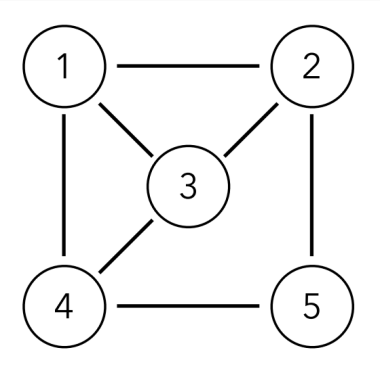
Coloring by simplification is a heuristic technique to (try to) color a graph G with K colors.

- 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

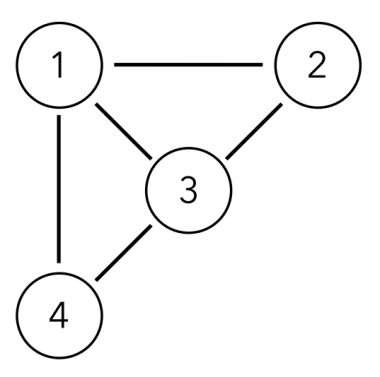
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes:

Coloring By Simplification

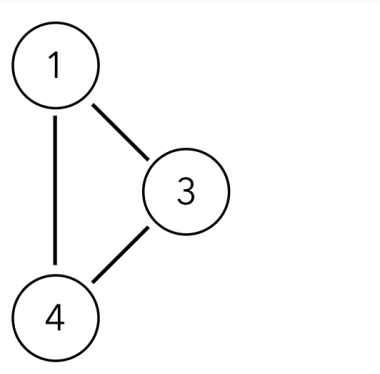
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5

Coloring By Simplification

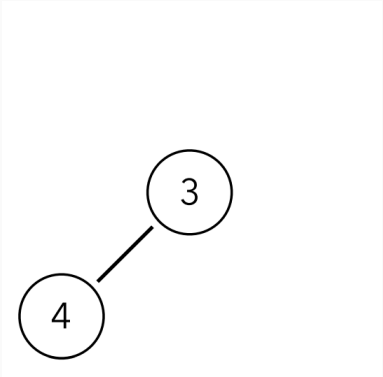
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5 2

Coloring By Simplification

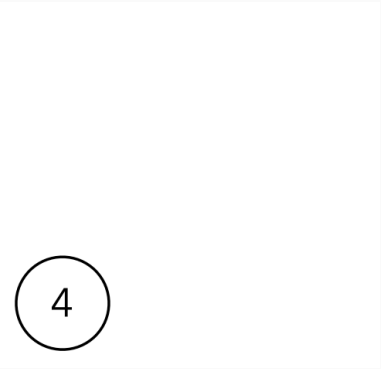
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5 2 1

Coloring By Simplification

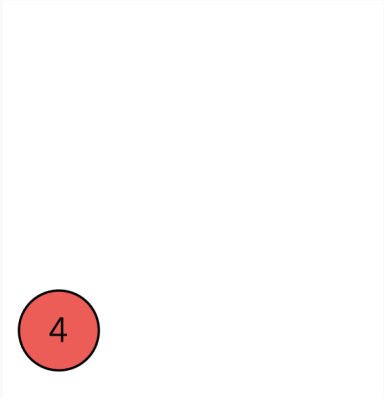
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5 2 1 3

Coloring By Simplification

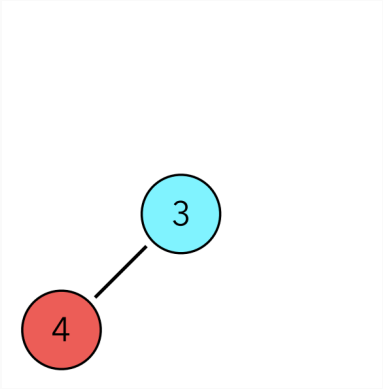
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5 2 1 3

Coloring By Simplification

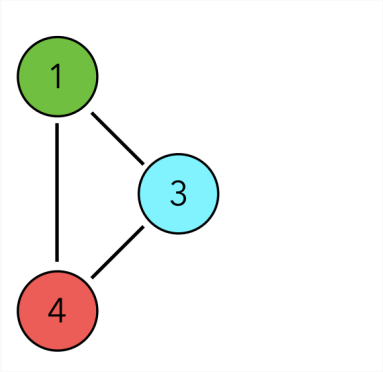
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5 2 1

Coloring By Simplification

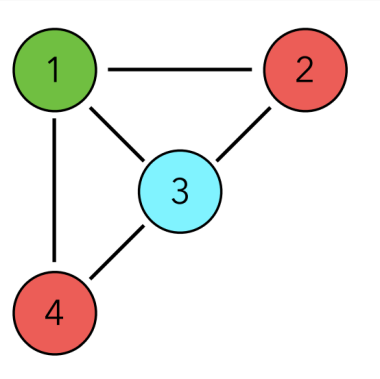
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5 2

Coloring By Simplification

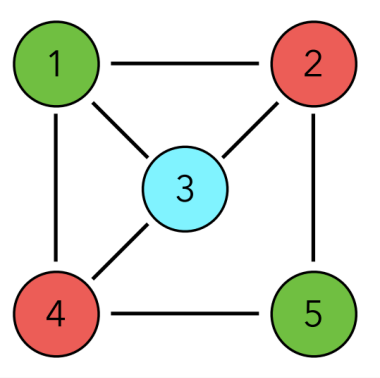
To illustrate coloring by simplification, we color the following graph with K=3 colors.



Stack of removed nodes: 5

Coloring By Simplification

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



Stack of removed nodes:

(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.

(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.

$$\texttt{cost(n)} = (\texttt{rw}_0\texttt{(n)} + 10\ \texttt{rw}_1\texttt{(n)} + \ldots + 10^k\ \texttt{rw}_k\texttt{(n))} / \texttt{degree(n)}$$

where n is the node, $\texttt{rw}_i\texttt{(n)}$ 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.

The node with the lowest cost should be spilled first.

Spill Costs Example

```
int i, j;
int k = 0;
for (i = 0 ; i < N; i++) {
  k += 5;
  for (j = 0 ; j < N; j++)
    k += 3;
}
```

```
cost(i) = rw0(i) + 10 * rw1(i) = 1 + 10 * 3 = 31
cost(j) =
cost(k) =
```

Spill Costs Example

```
int i, j;
int k = 0;
for (i = 0 ; i < N; i++) {
  k += 5;
  for (j = 0 ; j < N; j++)
    k += 3;
}
```

```
cost(i) = rw0(i) + 10 * rw1(i) = 1 + 10 * 3 = 31
cost(j) = rw0(j) + 10 * rw1(j) + 100 * rw2(j) = 0 + 10 * 1 + 100 * 3 = 310
cost(k) = rw0(k) + 10 * rw1(k) + 100 * rw2(k) = 1 + 10 * 2 + 100 * 2 = 211
```

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 3 colors. There is no node with degree less than 3, so the one with the lowest cost must be spilled.

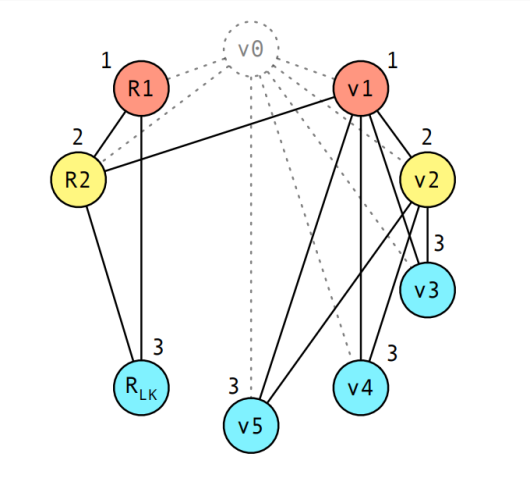
```
gcd:
  v0 ← RLK
  v1 ← R1
  v2 ← R2
loop:
  v3 ← done
  if v2=0 goto v3
  v4 ← v2
  v2 ← v1 % v2
  v1 ← v4
  v5 ← loop
  goto v5
done:
  R1 ← v1
  goto v0
```

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:

- 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.

Consequences Of Spilling

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

- 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 ← 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
```

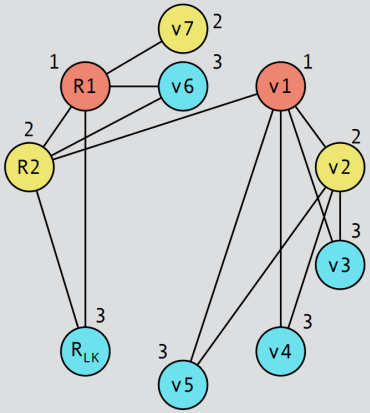
spilling of v_0

Rewritten program

```
gcd:
  v6 ← RLK
  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

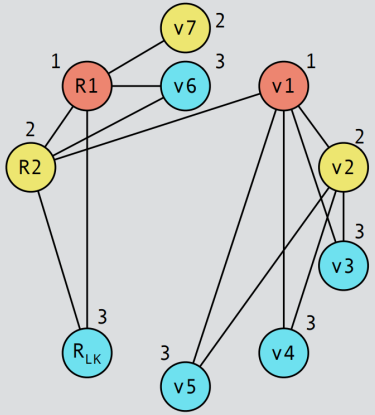


Final program

```
gcd:
  R_LK ← R_LK
  push R_LK
  R_1 ← R_1
  R_2 ← R_2
loop:
  R_LK ← done
  if R_2 = 0 goto R_LK
  R_LK ← R_2
  R_2 ← R_1 % R_2
  R_1 ← R_LK
  R_LK ← loop
  goto R_LK
done:
  R_1 ← R_1
  pop R_2
  goto R_2
```

New Interference Graph

Interference graph w/ spilling



Final program

```
gcd:
  R_LK ← R_LK
  push R_LK
  R_1 ← R_1
  R_2 ← R_2
loop:
  R_LK ← done
  if R_2 = 0 goto R_LK
  R_LK ← R_2
  R_2 ← R_1 % R_2
  R_1 ← R_LK
  R_LK ← loop
  goto R_LK
done:
  R_1 ← R_1
  pop R_2
  goto R_2
```

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.

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

$v_1 \leftarrow v_2$

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).

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

$v_1 \leftarrow v_2$

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 (not live at the same time), 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 useless copy/move 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,
- it 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:

1) Briggs: coalesce nodes n_1 and n_2 to $n_{1\&2}$ iff $n_{1\&2}$ has less than K neighbors of degree $\geq K$,

2) 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 degree $< K$.

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 degree $\geq K$.

Rationale: during simplification, all the neighbors of $n_{1\&2}$ that are of degree $< K$ 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.

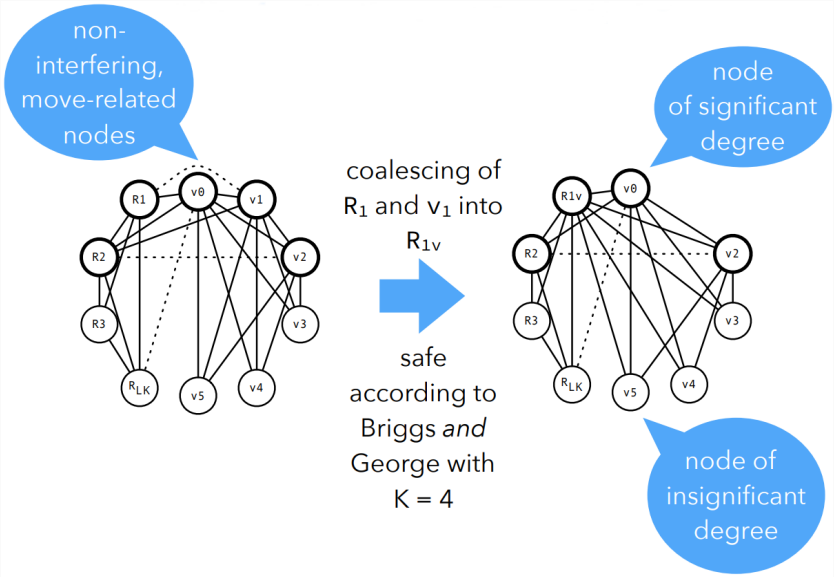
Heuristic #2: George

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

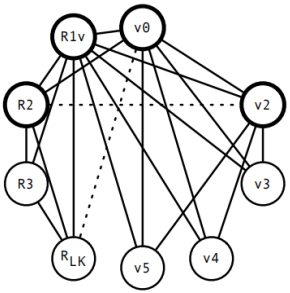
Rationale: the neighbors of $\mathsf{n}_{1\&2}$ will be the same as the neighbors of n_2, plus all neighbors of n_1 that are of insignificant degree. The latter ones will all be simplified, at which point the graph will be a sub-graph of the original one.

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

Coalescing Example

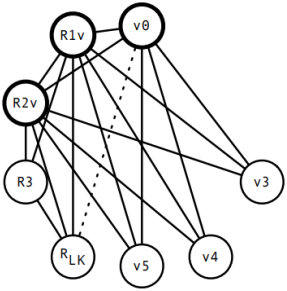


Coalescing Example (2)

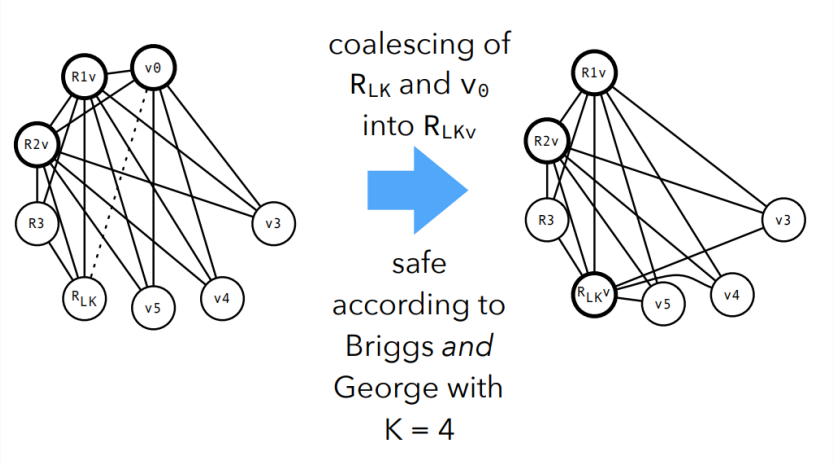


coalescing of R_2 and v_2 into R_{2v}

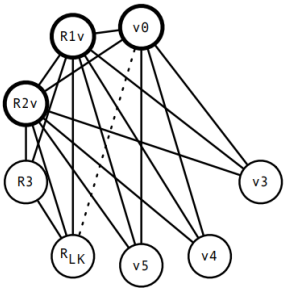
safe according to Briggs *and* George with K = 4



Coalescing Example (3)

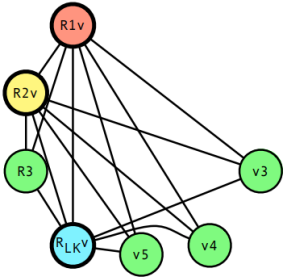


Coalescing Example (3)



coalescing of R_{LK} and v_0 into R_{LKv}

safe according to Briggs *and* George with K = 4



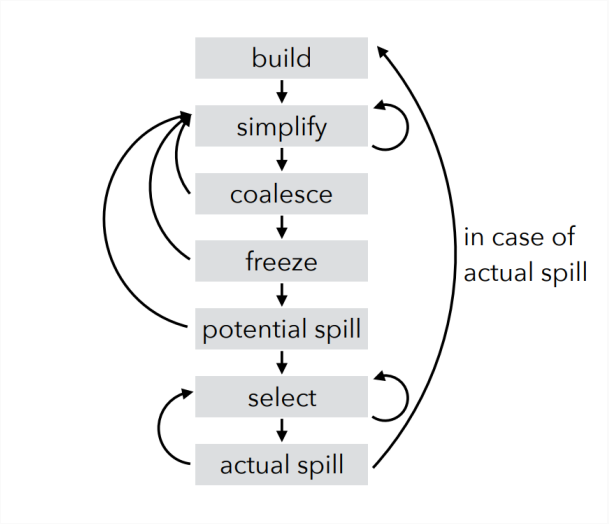
Summary: Iterated Register Coalescing

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 nodes not move-related (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, thus subject to simplification).
5) The process is restarted at 2.

Reference: https://ropas.snu.ac.kr/lib/dock/GeAp1996.pdf

Iterated Register Coalescing



Summary

Register allocation:

- Graph coloring
- Coloring by simplification
- Spilling and spill costs
- Coalescing and heuristics
- Combining simplification and coalescing