The C2 Register Allocator

Niclas Adlertz

I

Safe Harbor Statement

Outline

- Register allocation using graph coloring
- The structure of the C2 register allocator
- The future of the C2 register allocator

Register allocation using graph coloring

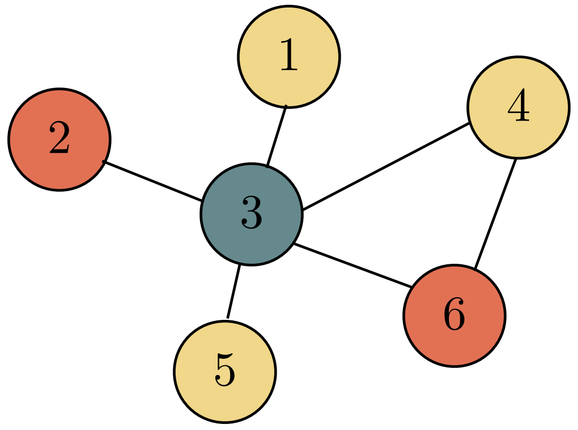
What is register allocation?

- Decides which variables that reside in registers at every program point
- Keep as many variables in registers as possible
  - i.e. avoid spilling variables to memory
- An important step in the compiler
  - Time consuming with big impact on code quality

Graph coloring

- Main idea
  - Try to K-color a graph containing N nodes with K colors
  - Assign each node a color, adjacent nodes cannot be assigned the same color
- Applied to register allocation
  - Nodes are variables (virtual registers)
  - Colors are registers (physical registers)
  - Adjacent nodes are variables that are live at the same time

Interference graph



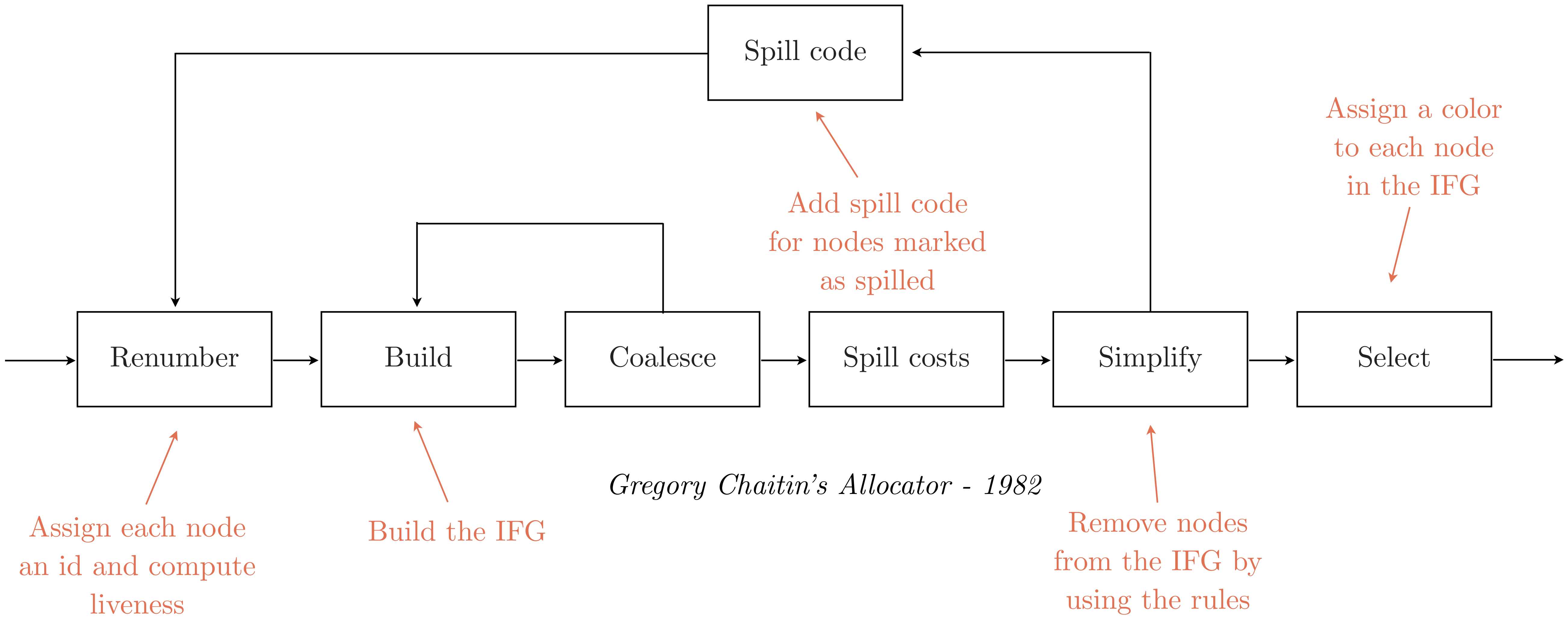
Nodes
{1, 2, 3, 4, 5, 6}

Colors
🟥 🟨 🟦

Graph coloring - basic algorithm

- Build IFG
  - Build an interference graph (IFG) using liveness data
- Simplify
  - Remove all nodes from the IFG and push them onto a stack using two rules:
    - *degree* $< K$ rule, where the *degree* of a node is its *number of edges*.
    - A cost function, mark the node as spilled
- Spill
  - If we had to use the cost function, insert spill code for the nodes marked as spilled and redo everything
- Select
  - Pop all nodes from the stack and add them to the IFG again
  - Assign a color to each node that is not used by any adjacent neighbor

Chaitin’s Allocator

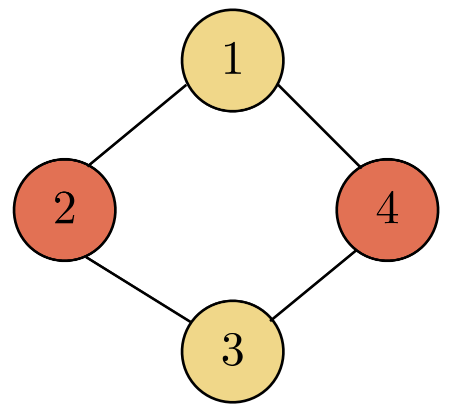

Spill code
Renumber
Build
Coalesce
Spill costs
Simplify
Select
Add spill code for nodes marked as spilled
Assign a color to each node in the IFG
Assign each node an id and compute liveness
Build the IFG
Remove nodes from the IFG by using the rules

Gregory Chaitin’s Allocator - 1982

Optimistic coloring

- Tries to assign colors in Select even if Simplify cannot remove all the nodes by using the first rule ($degree < K$)
- It will color any graph that Chaitin can color, and it will color some graphs that Chaitin can not
  - This automatically means reduced spilling on some graphs

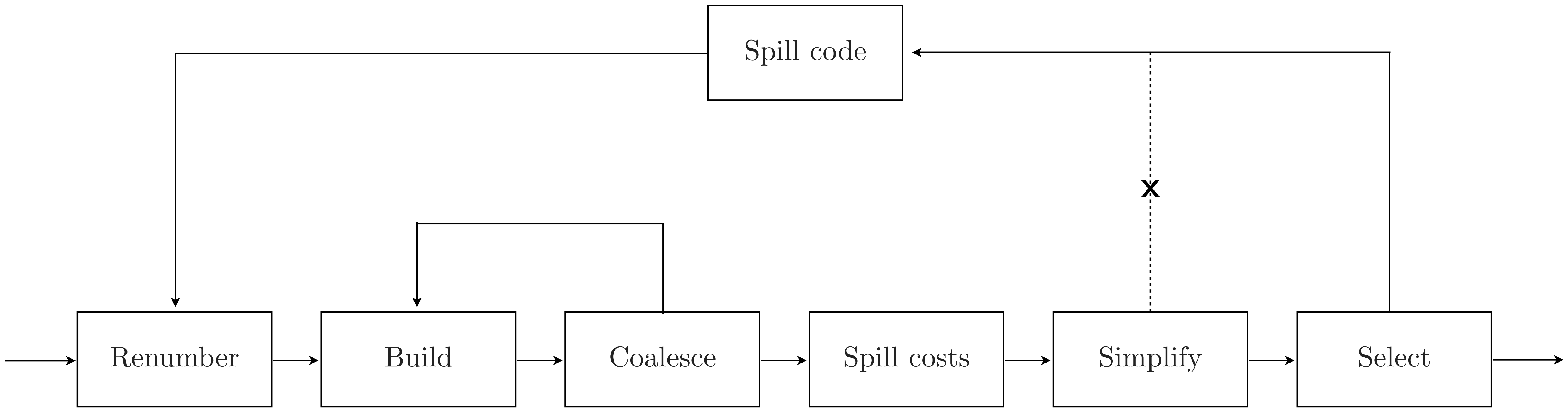
Diamond graph



Nodes
{1, 2, 3, 4}

Colors
🟧 🟨

Optimistic coloring



Chaitin-Briggs Allocator - 1989

The structure of the C2 register allocator

The C2 intermediate representation

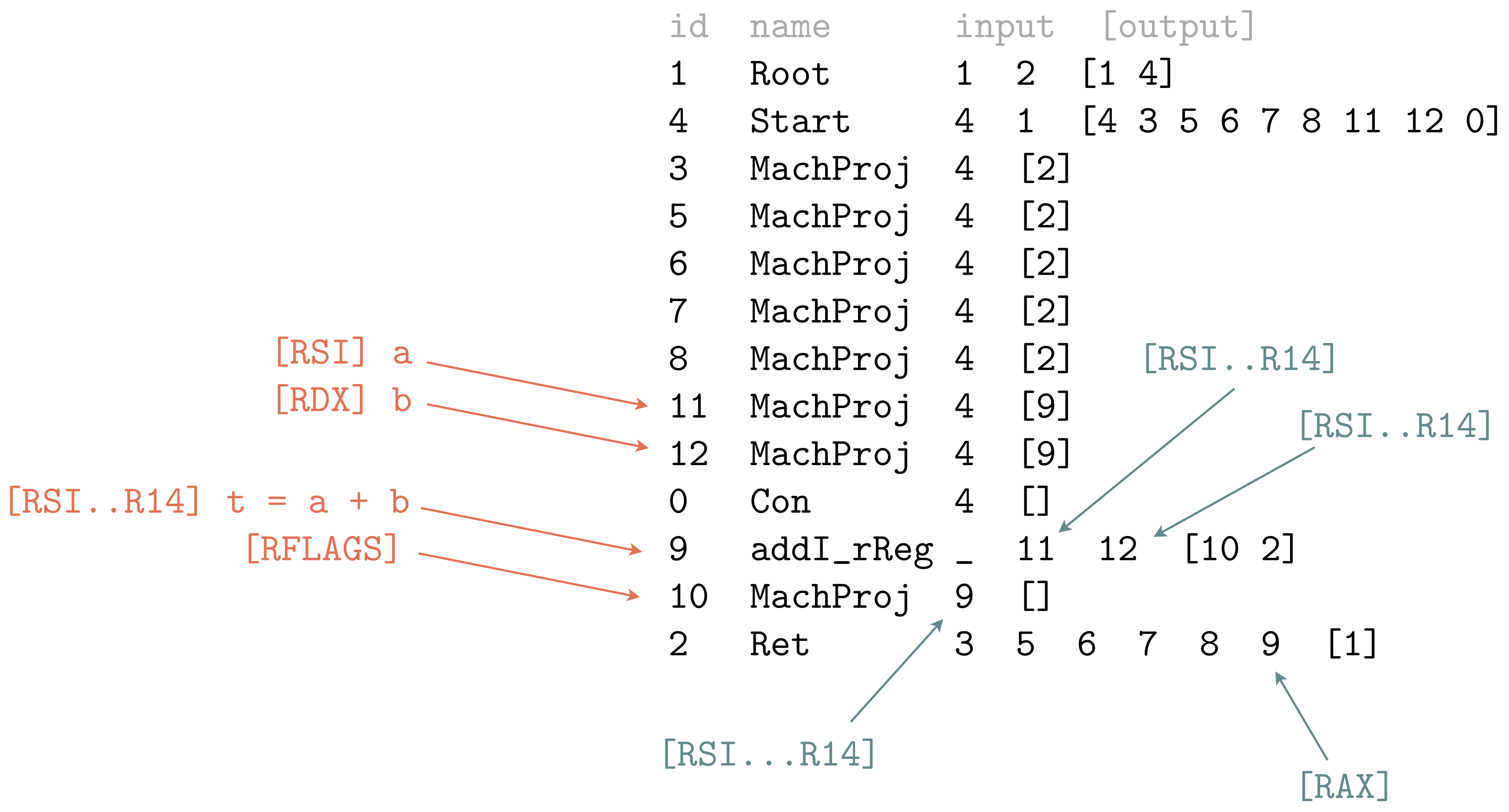
Java code

```
static int add(int a, int b) {
  return a + b;
}
```

C2 IR

```
id  name       input  [output]
1   Root       1  2  [1 4]
4   Start      4  1  [4 3 5 6 7 8 11 12 0]
3   MachProj   4  [2]
5   MachProj   4  [2]
6   MachProj   4  [2]
7   MachProj   4  [2]
8   MachProj   4  [2]
11  MachProj   4  [9]
12  MachProj   4  [9]
0   Con        4  []
9   addI_rReg _  11  12  [10 2]
10  MachProj   9  []
2   Ret        3  5  6  7  8  9  [1]
```

Register masks



```
id  name       input  [output]
1   Root       1  2  [1 4]
4   Start      4  1  [4 3 5 6 7 8 11 12 0]
3   MachProj   4  [2]
5   MachProj   4  [2]
6   MachProj   4  [2]
7   MachProj   4  [2]
8   MachProj   4  [2]
11  MachProj   4  [9]
12  MachProj   4  [9]
0   Con        4  []
9   addI_rReg  _  11  12  [10 2]
10  MachProj   9  []
2   Ret        3  5  6  7  8  9  [1]
```

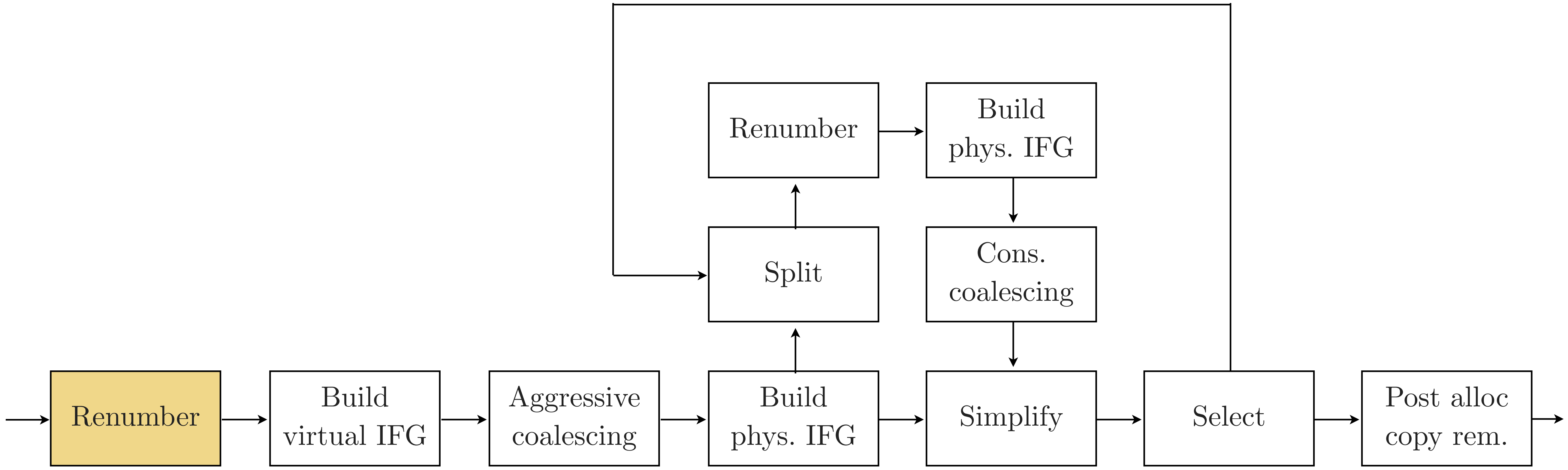
C2 uses register masks to represent real world limitations. Each definition has an `out` register mask and every use has an `in` register mask.

Finding virtual registers

- All nodes with a non-empty `out` register mask are considered virtual registers
- Each virtual register is assigned a unique live range id (`lid`)
  - all non virtual registers are assigned the `lid` 0

```
lid     node

0       1   Root       1  2  [1 4]
0       4   Start      4  1  [4 3 5 6 7 8 11 12 0]
0       3   MachProj   4  [2]
0       5   MachProj   4  [2]
0       6   MachProj   4  [2]
0       7   MachProj   4  [2]
0       8   MachProj   4  [2]
1       11  MachProj   4  [9]
2       12  MachProj   4  [9]
0       0   Con        4  []
3       9   addI_rReg _  11  12  [10 2]
4       10  MachProj   9  []
0       2   Ret        3  5  6  7  8  9  [1]
```

The register allocator of C2



The register allocator of C2

Structure of the C2 register allocator

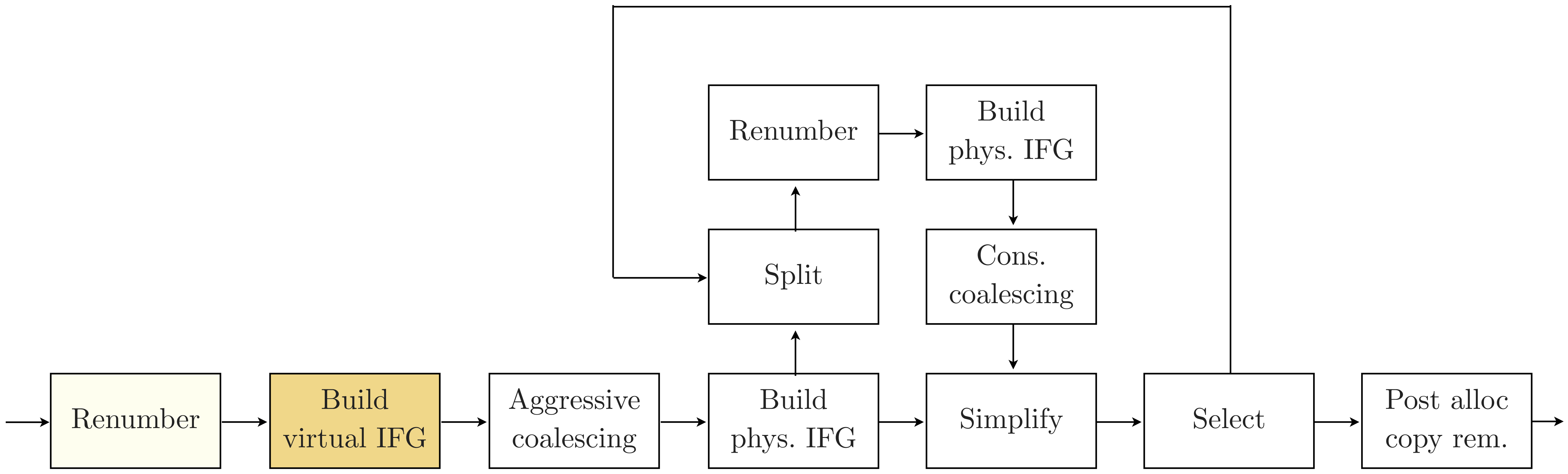
Renumber

- For every `lid` we create a live range (`LRG`)
- The `LRG` stores information such as
  - `_mask` - register mask that contains which registers this the live range can reside in
  - `_reg` - chosen register (or memory position) for this live range
  - `_def` - the node this live range corresponds to (could be more than one node)
  - `score()` - the higher score, more costly to spill
- The `_mask` of a `LRG` is $\texttt{out} \cap \texttt{in}_1 \cap .. \cap \texttt{in}_n$ (register masks at **n** all uses)
  - If the `LRG` is multi defined (due to coalescing) we will also intersect with every `out` register mask
  - If `_mask = ∅` when intersecting it with the register masks at the uses, we set `_reg = SPILL_REG`
- The `LIVE_OUT` for each block is computed
  - Contains all `lids` which are live at the exit of the block, `1 = live, 0 = dead`
  - Used to build the `IFG`

1	0	0	1	0	1	1

`LIVE_OUT`

The register allocator of C2



The register allocator of C2

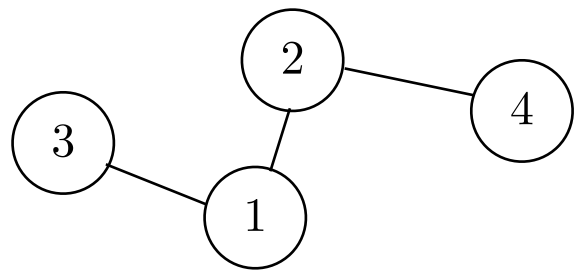
Build virtual IFG

```
foreach block in CFG
  live_out = get_live_out(block)
  foreach_reverse node in block
    // definition
    if has_lid(node)
      lid = get_lid(node)
      live_out.remove(lid)
      find_interference(lid, live_out)
    // uses
    if !is_phi(node)
      foreach input in node
        if has_lid(input)
          input_lid = get_lid(input)
          if live.exist(input_lid) == false
            live.add(input_lid)
```

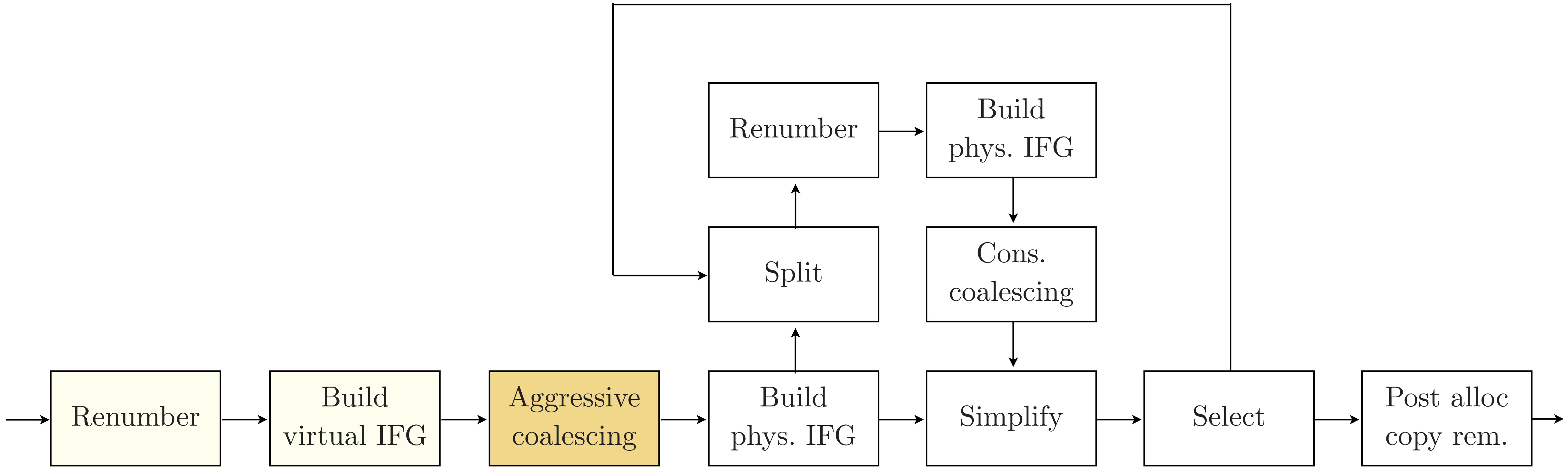
Internal data structure

	lid1	lid2	lid3	lid4
lid1	0	1	1	0
lid2	1	0	0	1
lid3	1	0	0	0
lid4	0	1	0	0

Graphical representation



The register allocator of C2



The register allocator of C2

Aggressive coalescing

- Coalesce phi nodes and two-address instructions with their inputs

```
foreach block in CFG
  foreach successor in block
    // phi coalescing
    foreach node in successor
      if !is_phi(node)
        break
      input = phi.in_from_block(block)
      coalesce(node, input)
    // check for two-address nodes
    foreach node in block
      if is_two_address(node)
        input = node.get_two_address_input()
        coalesce(node, input)
```

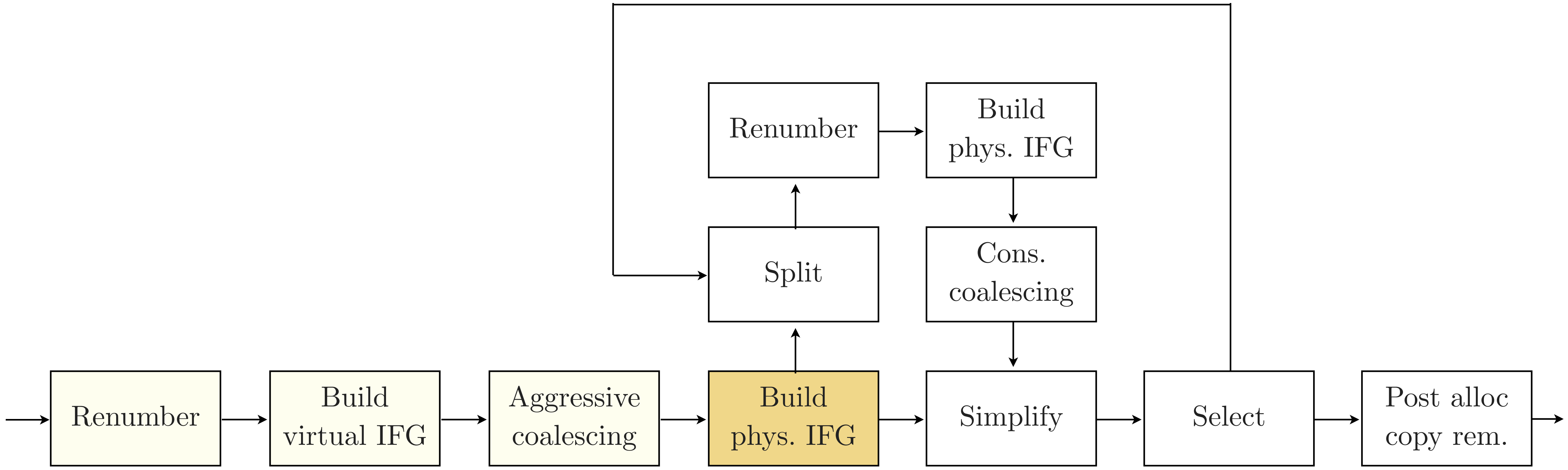
Restrictions - casting

```
int -> oop | OK
int -> int | OK
oop -> oop | OK
oop -> int | NOT OK!
```

Restrictions - register masks overlapping

`node._mask ∩ input._mask ≠ ∅`

The register allocator of C2



The register allocator of C2

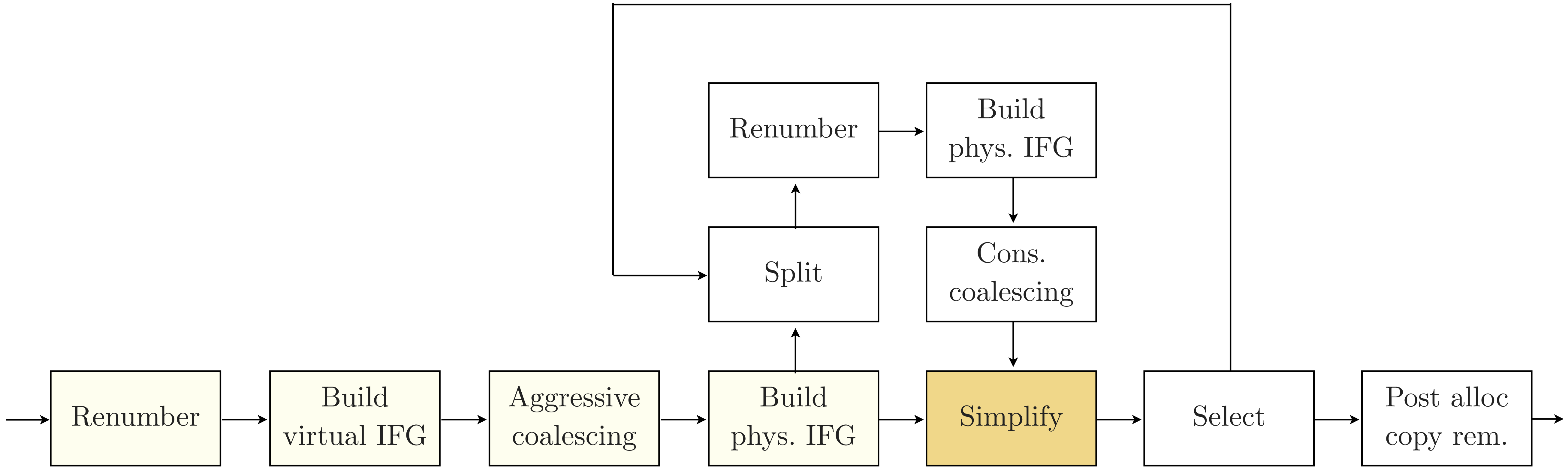
Structure of the C2 register allocator

Build physical IFG

- Build an IFG (as in the “Build virtual IFG” step)
- Estimate the register pressure for each block
  - Used for splitting heuristics
  - Record the index of the first transition from low to high register pressure (if any)
- For all bound[0] live ranges remove the bound register from interfering live ranges register mask
  - If `_mask` of an interfering register becomes empty, set `_reg = SPILL_REG` of the interfering mask
- Compute score (spill cost) for each live range
  - Higher score, more painful to spill
  - Defined/used at frequently executed paths → raises the score
  - Live during a big area in the code → lowers the score

[0] A bound live range has only one register in its register mask, which it’s bound to use. An example of such a live range would be a live range corresponding to an incoming argument.

The register allocator of C2



The register allocator of C2

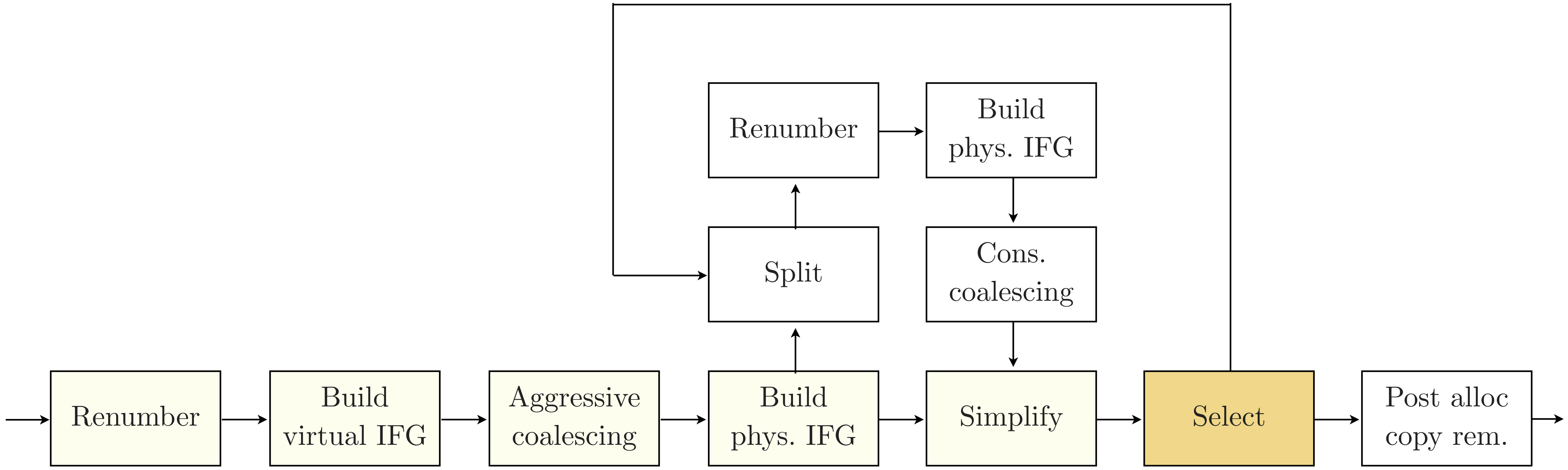
Simplify

```
while true
  while !low_degree_list.empty()
    lrg = low_degree_list.pop()
    simplify_list.push(lrg)
    // Remove lrg from IFG and lower neighbors
    // degree. Move neighbors that become low
    // degree from high to low degree list
    remove_from_ifg(lrg)

    if (high_degree_list.empty())
      break

    hi_lrg = find_the_best_spill(high_degree_list)
    high_degree_list.remove(hi_lrg)
    low_degree_list.push(hi_lrg)
```

The live range that will be picked from the high degree list will

1. Have the lowest score
2. If the scores are equal, have the biggest area.
3. If the areas are equal, be a bound live range
4. If all are bound/not bound, have the lowest cost

The register allocator of C2

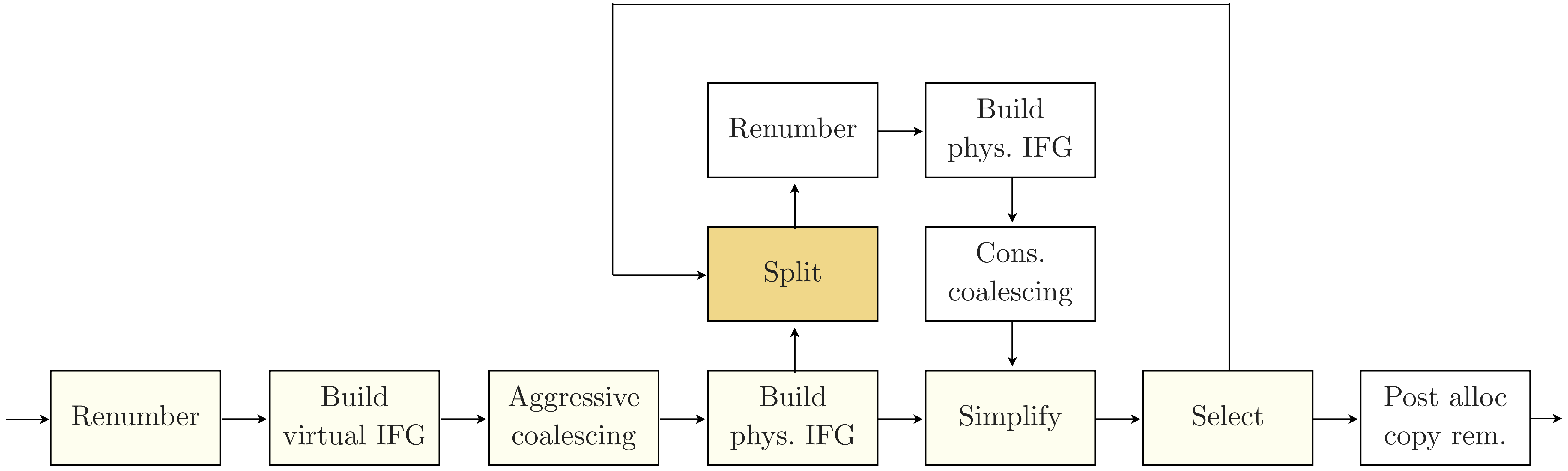


The register allocator of C2

Select

- Re-insert the the removed live ranges in reverse order of removal
  - If nothing from the high degree list was yanked, coloring will be guaranteed
  - If not, it might still color (optimistic coloring)
- For each re-inserted live range `l`, assign a register to `l` that is not used by any adjacent live ranges
  - If no register is available, set `_reg = SPILL_REG` of `l`
- When picking a register, use biased coloring
  - Try to assign the same register to two live ranges connected by a copy

The register allocator of C2



The register allocator of C2

Split

- Split all live ranges with `_reg = SPILL_REG` into multiple shorter live ranges
  - Reduces interference, but does not reduce register pressure
- Each time a live range is split, the new live range will be assigned a register mask containing both registers and stack slots
  - This will make the new live range pick a color in the Select phase (even if no registers are available)
- The low-to-high register pressure (HRP) transition indices computed in "Build IFG physical" are used to decide where to split.
  - Assume that a live range that will be split starts out in register
  - Split down to memory before entering HRP regions if live range is considered to be in register
  - Do not split in low register pressure (LRP) regions if the live range and use slots register masks overlap
- Use re-materialization if it is cheaper
  - Recompute the value instead of splitting it

The future of the C2 register allocator

Problems

- The code is very complex
  - The very important Split phase consists of one big method with about 1000 lines of code
  - A lot of information got lost. E.g. unknown splitting algorithm
- Poorly abstracted code
  - Most classes have references to each other.
  - Manually have to keep track of internal states of data structures. E.g. depending on the phase in the register allocator, we use different getters to retrieve the live range id for a node.
- Hard to test
  - No unit tests, no IR injection
- Hard to visualize
  - The current solution is to look at log files
- The current splitting heuristics is not aware of control flow
  - Splits can happen within loops and the allocator will not know it

Possible improvements

- Improving the current implementation
  - Code cleanups (work in progress) http://wiki.se.oracle.com/display/JPG/Register+Allocator+of+the+Server+Compiler
  - Visualization (work in progress)
  - Look at existing benchmarks and see how the spilling could be improved (to be done)
- Extensive additions to the current implementation
  - Passive splitting (1998). POC done by Intel, 2k+ lines of code, no major improvements. http://ecee.colorado.edu/ecen4553/fall09/live-range-splitting.pdf
  - Improved passive splitting (2005). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.78&rep=rep1&type=pdf
  - Interference Region Spilling (1997). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.3362&rep=rep1&type=pdf
- Implementing a new register allocator in C2
  - Graph fusion (1996) used in JRockit. Could re-use parts of the current allocator. http://reports-archive.adm.cs.cmu.edu/anon/1996/CMU-CS-96-106.pdf
  - Linear scan with lifetime holes and live range splitting (used in Graal and C1)
  - Greedy allocator in LLVM (2011) http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html
- Focus on the register allocators of the COMPILER.NEXT