

## Back-end

- Register allocation
  - Assign registers to all the vars
  - Graph coloring problem
  - Architecture-independent (assume  $K$  registers)



---

---

---

---

---

---

---

---

## FIR register allocation

- Sub-optimal, but easy
- Each function is an expression tree
- No subexpression can have more than  $K$  free variables
- Standard calling convention
  - If  $>K$  arguments, the rest are passed in memory ( $ebp+index$ )
  - If a subexpression has  $>K$  free vars, spill the vars that are not used for the longest time
- Have to be careful when generating assembly, because more temporaries will be needed (for example, let  $K=4$ , and let Codegen use  $eax,edx$  as additional temps)



---

---

---

---

---

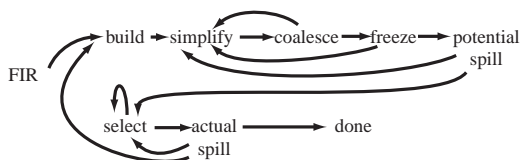
---

---

---

## General register allocation

- Global analysis
- Register assignment is arbitrary, so that callers and callees may optimize argument usage
- On x86, many vars will be spilled



---

---

---

---

---

---

---

---

## Graph coloring algorithm

- Keep a stack of removed nodes
- Simplify: remove and push all non-move related nodes with degree  $< K$  (these nodes can always be colored)
- Coalesce: coalesce (join) any two nodes in a move that pass the coalescing test, return to simplify
- freeze: if neither simplify or coalesce applies, find a move-related node of low-degree, and remove it from consideration for coalescing, return to simplify
- spill: if there are no more moves, and all nodes have degree  $\geq K$ , select a node, and remove/push it, resume simplify
- select: pop all vars on stack, assigning colors
  - *If there were spills, start over*



---

---

---

---

---

---

---

---

---

---

## Moves and coalescing

- If two nodes are used in a Move instruction, and they do not interfere, they may be *coalesced* into a new node with the union of the edges of the nodes being replaced (and the Move can be deleted)
- Unfortunately, graph may become uncolorable with aggressive coalescing
- Two conservative tests:
  - Briggs: nodes *a* and *b* can be coalesced if the resulting node has fewer than  $K$  neighbors with degree  $\geq K$
  - George: nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, *t* already interferes with *b*, or *t* has degree  $< K$



---

---

---

---

---

---

---

---

---

---

## Precolored nodes

- Some vars correspond to machine registers
  - For example, the IDIV instruction uses registers *eax,edx*
  - Calling-convention will pass arguments in pre-assigned registers
  - Precolored nodes are in *Frame.registers*



---

---

---

---

---

---

---

---

---

---

## Graph coloring algorithm

- Dataflow module builds interference graph, including all var<-var moves
- Nodes are always classified in one of eight classes
  - Precolored (a register)
  - Initial (at start)
  - SimpWL (non-move related, degree is less than K)
  - FreezeWL (involved in a move that may be frozen)
  - SpillWL (possible spill)
  - Spilled (real spill)
  - Coalesced (was coalesced with another var)
  - Colored (color has been assigned)
  - Stack (removed from the graph, on the stack)



---

---

---

---

---

---

---

---

## Nodes

- Each node belongs to exactly one class
- Node info:
  - *node\_degree*: number of active neighbors
  - *node\_neighbors*: all the incident edges
  - *node\_alias*: alias if the node is coalesced
  - *node\_color*: assigned color (if there is one)
  - *node\_moves*: moves the node is involved in



---

---

---

---

---

---

---

---

## Nodes

```
type node =  
{ node_node : IGraph.node;  
  mutable node_class : node_class;  
  mutable node_degree : int;  
  mutable node_alias : node option;  
  mutable node_color : var option;  
  mutable node_moves : move list;  
  mutable node_neighbors : node list;  
  
  (* Linked list *)  
  mutable node_pred : node option ref;  
  node_succ : node option ref  
}
```



---

---

---

---

---

---

---

---

## Moves

- Moves are also classified into exactly one class
  - Coalesced (move has been coalesced)
  - Constrained (nodes in the move interfere)
  - Frozen (no longer considered for coalescing)
  - Normal (can be considered for coalescing)
  - Active (can't be coalesced using conservative strategy)



---

---

---

---

---

---

---

---

## Moves

- Move needs
  - *move\_dst*, *move\_src*: dst/src of the move
  - *move\_class*: class of the move



---

---

---

---

---

---

---

---

## Moves

```
type move =  
  { move_dst : node;  
    move_src : node;  
    mutable move_class : move_class;  
  
    (* Linked list *)  
    mutable move_pred : move option ref;  
    move_succ : move option ref  
  }
```



---

---

---

---

---

---

---

---

## Register alloc struct

- Needs to support two operations on nodes/moves efficiently
  - *reclassify*: change the classification of a node/move
  - *get\_all*: get all the nodes/moves in a particular class
- Data structure uses doubly-linked lists
  - Constant-time insertion removal
  - Imperative and ugly
- Interference graph edges are saved in a SymbolMatrix



---

---

---

---

---

---

---

---

---

---

## Register alloc struct (nodes)

```
type ra =
{ (* Node worklists *)
  ra_precolored : node option ref;
  ra_initial : node option ref;
  ra_simp_wl : node option ref;
  ra_freeze_wl : node option ref;
  ra_spill_wl : node option ref;
  ra_spilled : node option ref;
  ra_coalesced : node option ref;
  ra_colored : node option ref;
  ra_stack : node option ref;
```



---

---

---

---

---

---

---

---

---

---

## Register alloc struct (moves)

```
(* Move worklists *)
mv_coalesced : move option ref;
mv_constrained : move option ref;
mv_frozen : move option ref;
mv_wl : move option ref;
mv_active : move option ref;

(* Explicit representation of the graph *)
ra_edges : SymbolMatrix.t
}
```



---

---

---

---

---

---

---

---

---

---

## Reclassify

- Remove node from current linked list

```
let node_reclassify ra node c1 =
  (* Delete the node from its current list *)
  let pred = node.node_pred in
  let succ = !(node.node_succ) in
  let _ =
    pred := succ;
    match succ with
    | Some next -> next.node_pred <- pred
    | None -> ()
  in
```



---

---

---

---

---

---

---

---

## Reclassify

- Add it to new linked list, update class

```
(* Add to the new list *)
let l = node_worklist ra c1 in
let next = !l in
l := Some node;
assert (node.node_class <> NodePrecolored);
node.node_class <- c1;
node.node_pred <- l;
node.node_succ := next;
match next with
| Some next -> next.node_pred <- node.node_succ
| None -> ()
```



---

---

---

---

---

---

---

---

## Register allocation

- Generate ASM code from FIR code
- Create dataflow graph
- Classify nodes/moves
- Do:
  - If *simp\_wl* then Simplify
  - If *move\_wl* then Coalesce
  - If *freeze\_wl* then Freeze
  - If *spill\_wl* then Spill
- Assign colors
- If there are spills, rewrite code and start over



---

---

---

---

---

---

---

---

## Classify

- Create a node for each var
- If node is a register then
  - classify as *Precolored*
- if node\_degree  $\geq$  K then
  - classify as *SpillWL*
- if node\_moves  $\leq$  [] then
  - classify as *FreezeWL*
- else
  - classify as *SimpWL*
- All moves are classified as *MoveWL*
- Save interference graph edges in ra\_edges



---

---

---

---

---

---

---

---

---

---

## Simplify

- Choose a node in ra\_simp\_wl
- Reclassify as *Stack*
- Decrement degree of all the neighbors

```
let simplify ra =  
  let node = node_list_head ra NodeSimpWL in  
  node_reclassify ra node NodeStack;  
  List.iter (decrement_degree ra) (node_neighbors ra node)
```



---

---

---

---

---

---

---

---

---

---

## Neighbors

- Neighbors include all nodes excepted *Coalesced* and *Stack* nodes

```
let node_neighbors ra node =  
  List.filter (fun node ->  
    match node.node_class with  
    | NodeStack  
    | NodeCoalesced ->  
      false  
    | _ ->  
      true) node.node_neighbors
```



---

---

---

---

---

---

---

---

---

---

## Decrement degree

- If degree becomes  $K - 1$ :
  - Reconsider all Active moves related to this node and all adjacent nodes
  - If the node is move-related
    - reclassify as FreezeWL
    - else, reclassify as SimpWL



---

---

---

---

---

---

---

---

## Decrement degree

```
let decrement_degree ra node =  
  if node.node_class <> NodePrecolored then  
    let degree = node.node_degree in  
    node.node_degree <- pred degree;  
    if degree = max_colors then  
      begin  
        enable_moves ra (node :: node_neighbors ra node);  
        if node_is_move_related node then  
          node_reclassify ra node NodeFreezeWL  
        else  
          node_reclassify ra node NodeSimpWL  
        end  
      end  
  end
```



---

---

---

---

---

---

---

---

## Coalesce

- This is the most complex
- Choose a move (dst, src) in mv\_wl
- Get aliases (x, y) for (dst, src) (nodes may have been coalesced)
- If  $x = y$ , reclassify move as Coalesced
- If both (x, y) are precolored, or (x, y) interfere, reclassify as Constrained
- If (x is precolored && George(y)), Combine
- If (x is not precolored && Briggs(x, y)), Combine
- Otherwise, reclassify as Active



---

---

---

---

---

---

---

---

## Coalesce (first part)

```
let coalesce ra =
  let move = move_list_head ra MoveWL in
  let x = node_alias move.move_dst in
  let y = node_alias move.move_src in
  let u, v =
    if y.node_class = NodePrecolored then
      y, x
    else
      x, y
  in
  if node_eq u v then
    begin
      move_reclassify ra move MoveCoalesced;
      add_worklist ra u
    end
```



---

---

---

---

---

---

---

---

---

---

## Coalesce (interfering nodes)

```
else if v.node_class = NodePrecolored || query ra u v then
  begin
    move_reclassify ra move MoveConstrained;
    add_worklist ra u;
    add_worklist ra v
  end
```



---

---

---

---

---

---

---

---

---

---

## Coalesce (precolored nodes)

```
else if u.node_class = NodePrecolored && george_test ra u v then
  begin
    move_reclassify ra move MoveCoalesced;
    combine_precolored ra u v
  end
else if u.node_class <> NodePrecolored && briggs_test ra u v then
  begin
    move_reclassify ra move MoveCoalesced;
    combine_normal ra u v;
    add_worklist ra u
  end
else
  move_reclassify ra move MoveActive
```



---

---

---

---

---

---

---

---

---

---

## Combine(*u*, *v*)

- reclassify *v* as Coalesced
- Set `node_alias` for *v* to *u*
- Add all `node_moves` for *v* to `nodes_moves` for *u*
- For each edge incident on *u*, make a corresponding edge on *v*



---

---

---

---

---

---

---

---

## Combine

```
let combine_normal ra u v =  
  u.node_moves <- u.node_moves @ v.node_moves;  
  List.iter (fun t ->  
    add_edge ra t u;  
    decrement_degree ra t) (node_neighbors ra v);  
  node_reclassify ra v NodeCoalesced;  
  v.node_alias <- Some u;  
  if u.node_degree >= max_colors && u.node_class = NodeFreezeWL then  
    node_reclassify ra u NodeSpillWL
```



---

---

---

---

---

---

---

---

## Freeze

- Choose a node *u* from FreezeWL
- reclassify as SimpWL
- Freeze moves:
  - For each move (*u*, *v*) related to *u*
  - reclassify as Frozen
  - If *v* become non-move related, and degree < *K*, reclassify as SimpWL



---

---

---

---

---

---

---

---

## Freeze

```
let freeze ra =  
  let node = node_list_head ra NodeFreezeWL in  
  node_reclassify ra node NodeSimpWL;  
  freeze_moves ra node
```



---

---

---

---

---

---

---

---

## Freeze\_moves

```
let freeze_moves ra node =  
  let u = node_alias node in  
  let moves = node_moves node in  
  List.iter (fun move ->  
    let y = node_alias move.move_src in  
    let v =  
      if node_eq y u then  
        node_alias move.move_dst  
      else  
        y  
    in  
    move_reclassify ra move MoveFrozen;  
    if not (node_is_move_related v) && v.node_degree < max_colors then  
      node_reclassify ra node NodeSimpWL) moves
```



---

---

---

---

---

---

---

---

## Spill

- Choose a node from SpillWL
- Reclassify as SimpWL
- Freeze moves
- Be careful!
  - When a node is spilled, the assembly has to be rewritten to fetch the value from memory just before it is used, and store it after it is defined
  - This defines a new temporary register with a short live range
  - Don't want to spill these temporaries!



---

---

---

---

---

---

---

---

## Spill

```
let spill ra vars =  
  let rec search node =  
    let v = var_of_node ra node in  
    if SymbolSet.mem v vars then  
      begin  
        node_reclassify ra node NodeSimpWL;  
        freeze_moves ra node  
      end  
    else  
      search (node_succ node)  
  in  
  search (node_list_head ra NodeSpillWL)
```



---

---

---

---

---

---

---

---

## Spill choices

- The given spill algorithm chooses spills randomly
- However, the spill choice is critical to performance
  - *What is a good spill metric?*



---

---

---

---

---

---

---

---

## Spill metrics

- Option 1: spill the var with the highest degree
  - *Removes a lot of interferences at once*
- Option 2: spill the var with the lowest cost
  - *Basic cost: #times the variable is accessed per program execution*
  - *Balance against number of other spills if this var is assigned to a register*
  - *Heuristic: don't spill vars in the inner loop*



---

---

---

---

---

---

---

---

## Final step:

- once there are no SimpWL, MoveWL, FreezeWL, SpillWL, it is time to color the nodes
- Pop nodes from the Stack in order, assigning colors
- For each node: the possible colors are colors not already given to the neighbors
- If there is a color, pick a color, any color
- Otherwise, add a *real* spill
- Once finished, if there are any real spills, start over



---

---

---

---

---

---

---

---

## Assign\_colors

```
let rec assign_colors ra =
  node_iter ra NodeStack (fun node ->
    let remove_color colors w = ... in
    match List.fold_left remove_color
      Backend.registers node.node_neighbors
    with
    color :: _ ->
      node.node_color <- Some color;
      node_reclassify ra node NodeColored
    | [] ->
      node_reclassify ra node NodeSpilled)
```



---

---

---

---

---

---

---

---

## Remove color

- Start with the full set of registers
- Remove the color of each neighboring node

```
let remove_color colors w =
  let w' = node_alias w in
  match w'.node_class with
  NodeColored
  | NodePrecolored ->
    (match w'.node_color with
     Some color ->
       List_util.remove color colors
     | None ->
       raise (Invalid_argument "assign_colors"))
  | _ ->
    colors
```



---

---

---

---

---

---

---

---

## Spill color assignment

- Once all the colors have been assigned, the Spilled nodes have to be spilled
- The easy way:
  - Just call (*Backend.spset\_spill spset v*) for any node that was spilled
  - This will assign *v* to a new spill register *i(%ebp)*, in sequence *0(%ebp)*, *4(%ebp)*, *8(%ebp)*, ...



---

---

---

---

---

---

---

---

---

---

## A better spill method

- Treat spills as an alternative register set, and spill the same way as `assign_colors`
- Start with an empty register file
  - Make up new colors as you go along



---

---

---

---

---

---

---

---

---

---

## Assigning spill colors

```
(* Generate new colors *)
let colors =
  node_fold ra NodeSpilled (fun colors node ->
    let remove_color colors w = ... in
    match List.fold_left remove_color
      colors node.node_neighbors with
    color :: _ ->
      node.node_color <- Some color;
      colors
    | [] ->
      let color = new_symbol_string "spill" in
      node.node_color <- Some color;
      color :: colors) []
in
```



---

---

---

---

---

---

---

---

---

---

## Choosing spill colors

```
(* Generate new colors *)
let remove_color colors w =
  let w' = node_alias w in
  match w'.node_class with
  | NodeSpilled ->
    (match w'.node_color with
     | Some color ->
       List_util.remove color colors
     | None ->
       colors)
  | _ ->
    (* Others do not interfere *)
    colors
```

in



---

---

---

---

---

---

---

---

---

---

## Spilling spill colors

```
(* Spill all the new colors *)
let spset = List.fold_left Backend.spset_spill spset colors in

(* Assign all the spills *)
node_fold ra NodeSpilled (fun spset node ->
  let v = var_of_node ra node in
  match node.node_color with
  | Some v' ->
    Backend.spset_add spset v v'
  | None ->
    raise (Invalid_argument ...)
```



---

---

---

---

---

---

---

---

---

---