The C2 Register Allocator

Introduction

Todo

General flow

Let's first show the general flow of the register allocator (RA).

```
PhaseLive live
PhaseIFG ifg

// assign an LRG index for each node
de_ssa()

// create all the corresponding LRGs
gather_lrg_masks()

// compute the LIVE_OUT for each block in the CFG
live.compute()

// optimize phase; stretch the base pointers
// for derived pointers at each safepoint
if stretch_base_pointer_live_ranges() {
// if nodes are added,
// rebuild the LRGs and recompute the liveness
gather_lrg_masks()
```

```
live.compute()
}

// build an IFG
build_ifg_virtual()

// make the triangular IFG squared
ifg.square_up()

// optimize phase; coalesce phi and two address
// instructions with their inputs, if possible
PhaseAggressiveCoalesce coalesce
coalesce.coalesce_driver()

// if coalescing insert explicit copies were the coalescing failed
coalesce.insert_copies()

// rebuild the LRGs and recompute the liveness
gather_lrg_masks()
live.compute()

// build an IFG also computing the area and cost for each LRG,
// also compute the High Register Pressure (HRP) and Low Register Pressure (LRP) area.
// See if any nodes must spill from start
uint must_spill = build_ifg_physical()

if must_spill {

// add spill code for spilled nodes
split()

// re-number the LRG indices to make the IFG smaller
compact()

// rebuild the LRGs and recompute the liveness
gather_lrg_masks()
live.compute()

// build the IFG again, ignore the return value
build_ifg_physical()

// square the IFG up again
ifg.square_up()

// compute the degree, i.e. the
ifg.compute_effective_degree()

// optimize phase; conservatie coalescing, try to
// coalesce live ranges but don't make a colorable
// graph un-colorable.
if OptoCoalesce {
PhaseConservativeCoalesce coalesce
coalesce.coalesce_driver()
}

// make all Nodes map directly to their final LRG idx
compress_uf_map_for_nodes()

} else {
// make the triangular IFG squared
ifg.square_up()

// compute the effective degree for each LRG
ifg.compute_effective_degree()
}

// compute the cost / area ratio, if we need to spill
// Also build the lo-degree list
cache_lrg_info()

// remove all lo-degree LRGs from the IFG
```

```
// and push them onto the stack
simplify()

// assign color (physical registers) to each LRG
uint spills = select()
while spills {

// add spill code for spilled nodes
split()

// re-number the LRG indices to make the IFG smaller
compact()

// rebuild the LRGs and recompute the liveness
gather_lrg_masks()
live.compute()

// build the IFG again, ignore the return value
build_ifg_physical()

// make the triangular IFG squared
ifg.square_up()

// compute the effective degree for each LRG
ifg.compute_effective_degree()

// optimize phase; conservatie coalescing, try to
// coalesce live ranges but don't make a colorable
// graph un-colorable.
if OptoCoalesce {
PhaseConservativeCoalesce coalesce
coalesce.coalesce_driver()
}

// make all Nodes map directly to their final LRG idx
compress_uf_map_for_nodes()

// compute the cost / area ratio, if we need to spill
// Also build the lo-degree list
cache_lrg_info()

// remove all lo-degree LRGs from the IFG
// and push them onto the stack
simplify()

// assign color (physical registers) to each LRG
spills = select()
}

post_allocate_copy_removal()

fixup_spills()
```

PhaseChaitin::de_ssa

Mission

Assign each node a virtual register.

Needs

The CFG with nodes and an empty LRG_List _name.

Produces

Fills the LRG_List _names with a node to virtual register mapping.

Detailed

The CFG is iterated and each node is assigned a virtual register id, we call this lrg_idx, a live range index. The lrg_idx is a uint. All machine nodes are assigned a unique lrg_idx. All other non machine nodes are assigned lrg_idx 0.

de_ssa() looks for nodes with a non empty out_RegMask(). Each node that fullfills this criteria is assigned a unique number. The numbering starts from 1. The 0 is used for all nodes with empty out_RegMask().

It it possible to get a lrg_idx for a node using the method:

```
uint n2lidx(const Node *n) const
```

or directly by accessing _names:

```
uint lrg = _names[node->_idx]
```

PhaseLive::compute

Mission

Compute a LIVE_OUT set for each block in the CFG.

Needs

The CFG with nodes and non empty LRG_List has an array of LIVE_OUT sets, one for each block in the CFG.

Produces

The instance object of PhaseLive will have an array of LIVE_OUT sets, one for each block in the CFG.

Detailed

When computing the liveness analysis, the LRG_List _names and the CFG are used. The liveness analysis is an incremental algorithm and computes the LIVE_OUT for each block in the CFG.

Each LIVE_OUT set is represented as an IndexSet. An IndexSet can be thought of as an array of bools. Each bool in the array corresponds to a lrg_idx (which corresponds to a node. The bool at index X corresponds to the lrg_idx == X. You can fetch a LIVE_OUT set from a block from a PhaseLive instance accordingly.

```
PhaseLive live(_cfg, _names, &live_arena);
live.compute(_maxlrg);
IndexSet *live_set = live.live(block);
```

Chaitin::gather_lrg_masks

Mission

Create a live range struct, LRG, for each node in the CFG that has a unique live range index, lrg_idx.

Needs

The CFG with all nodes and an empty array of LRG instances.

Produces

Fills the LRG instances with relevant data for each node with a unique virtual register.

Detailed

The CFG is iterated and each node with a unique lrg_idx is processed. The LRG is filled with a lot of information, such as type (gpr of float), number of registers, register pressure, which registers the node value can be in, etc.

PhaseChaitin::stretch_base_pointer_live_ranges

Mission

Stretch the base pointers of derived pointer at each safepoint.

Needs

The CFG with all nodes and the LIVE_OUT for each block.

Produces

If any base pointers are stretched, new nodes may be added.

Detailed

TODO

Chaitin::build_ifg_virtual

Mission

Build an IFG in order to do Aggressive Coalesing later on.

Needs

The CFG with an LRG for each node with a unique lrg_idx.

Produces

An IFG with info on which LRGs that interfere with each other.

Detailed

The class PhaseIFG contains an array _adjs. It's an array of IndexSets; one IndexSet for each LRG. Each IndexSet for a LRG L, contains the indices of the LRGs that interfere with L. The _adjs is a two-dimensional array with number_of_LRGs * number_of_LRGs as area. An initial representation of _adjs is shown below:

```
......L1 L2 L3 L4
  L1   0  0  0  0
  L2   0  0  0  0
  L3   0  0  0  0
  L4   0  0  0  0
```

When the _adjs array is built, an example of the representation could be:

```
......L1 L2 L3 L4
  L1   0  1  0  0
  L2   0  0  0  1
  L3   1  0  0  0
  L4   0  1  0  0
```

Note that the IFG is still triangular, meaning that the edge from L1 to L3 as shown above doesn't guarantee that there is an edge from L3 to L1.
In order to do a faster Coalescing later on, the IFG is squared. This means that if there is an edge between L1 and L3, there must also be an edge from L3 to L1 in the _adjs array.

```
......L1 L2 L3 L4
  L1   0  1  1  0
  L2   1  0  0  1
  L3   1  0  0  0
  L4   0  1  0  0
```

The general algorithm for building the IFG is very straight forward, and also shows why the IFG is triangular at first. The general algorithm is shown below. For each block in any order, we step through the nodes in reverse order. At each node we check if it's a virtual register, if so, we remove its corresponding live range from the LIVE_OUT. At the same time, we also look at which live ranges that are present in LIVE_OUT. These are the live ranges that the current live range interfere with. All live range that uses the same type of registers as this live range, are edges to this live range in the IFG.

At each node we also look at the input nodes (except at phi nodes). If they are virtual registers and their corresponding live ranges are not present in the LIVE_OUT, they are added to the LIVE_OUT.

```
// in any block order
foreach block in CFG {
  LIVE_OUT live = get_live_out(block)
  // in reserver order in block
  foreach node in block {

    if node.is_virtual_reg {
      LRG l = n2lidx(node)
      live.remove(l)
      add_edges(l, liveout)
    }

    if node.is_phi == false {
      foreach input in node {
        if input.is_virtual_reg {
          LRG l = n2lidx(input)
          if live.exist(l) == false {
            live.add(l)
          }
        }
      }
    }
  }
```

PhaseAggressiveCoalesce::coalesce_driver

Mission

Try to coalesce phi nodes and two-address instructions with its input.

Needs

The CFG with nodes with the IFG. The IFG must be squared.

Produces

Merged LRGs for phi nodes and their inputs if the coalescing succeeds. Same with two-address nodes and their inputs.

Detailed

Up to this point we have virtual copies. Virtual copies means that we have not put copies explicitly in the code, instead the incoming variables to a phi, and the phi itself have unique live-ranges. The goal is to coalesce these, so that they will have the same live range. If this is not possible, actual copies will be inserted later on. Let's have a look at the algorithm for aggressive coalescing.

```
COALESCE_DRIVER() {
  foreach block in CFG {
    COALESCE(block)
  }
}
```

```
COALESCE(block) {
  foreach successor in block {
    // phi coalescing
    foreach node in successor {
      if node.is_phi == false {
        break
      }

      Node i_node = phi.in_from_block(block)
      COMBINE_THESE_TWO(node, i_node)
    }

    // check for two-address nodes
    foreach node in block {
      if node.is_two_address {
        Node i_node = node.two_address_in()
        COMBINE_THESE_TWO(node, i_node)
      }
    }
  }
}
```

```
COMBINE_THESE_TWO(node, i_node) {

  uint lrg_idx = n2lidx(node)
  uint i_lrg_idx = n2lidx(i_node)
  // have the same idx
  if lrg_idx == i_lrg_idx {
    return
  }

  // interfere
  if ifg.interfers(lrg_idx, i_lrg_idx) {
    return
  }

  LRG lrg = lrg(lrg_idx)
  LRG i_lrg = lrg(i_lrg_idx)

  bool legit_cast = lrg.is_oop || i_lrg.is_oop == false
  bool common_registers = lrg.mask().overlap(i_lrg->mask())

  if legit_cast && common_registers {
    // merge larger into smaller.
    if lrg_idx > i_lrg_idx {
      swap(node, i_node)
      swap(lrg_idx, i_lrg_idx)
      swap(lrg, i_lrg)
    }

    // set i_node lrg_idx to node lrg_idx in _uf_map
    union(node, i_node)

    if lrg._maxfreq < i_lrg._maxfreq {
      lrg._maxfreq = i_lrg._maxfreq
    }

    // merge in the IFG
    ifg.union(lrg, i_lrg)

    // combine register restrictions
    lrg.AND(i_lrg.mask())
  }
}
```

Coalescing that involves casting is most of the time ok.

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

PhaseAggressiveCoalesce::insert_copies

Mission

Insert explicit copies between the phi nodes and their inputs where coalescing failed. Same for two-address nodes. Also insert copies at safepoints for high frequency nodes in low frequency blocks to reduce spilling.

Needs

The CFG nodes with the IFG.

Produces

Extra copies for moves between phi nodes and their inputs. Same for two-address nodes. Also added copies, i.e. moves, between high frequent nodes at safepoints in low frequent blocks.

Detailed

When encountering a phi node in the CFG, the lrg_idx of the phi node is checked and compared to the lrg_idxs of the phi node inputs. If the lrg_idx differs, a copy must be inserted. The copy represents a move from the npute node to the phi node.

If the input is a constant node which is set to rematerializable, we clone the input instead of copying it. This way we re-compute the value instead of introducing a copy. If the input is not a constant node that is rematerializable, we have to create a copy node.

The copy is inserted in the predecessing block from which the input comes from. It could be tricky to insert a copy if there are several more phi copies to be inserted in the same block. Imagine for example that L1 -> L2 is a copy and also L2 -> L3. We need to think of the order in which we put the copies. It could also be even trickier. L1 -> L2, L2 -> L3 and L3 -> L1 for example, now we have to introduce an extra copy to prevent any value to be lost.

The copy node and the phi node will have the lrg_idx. The copy will be the input of the phi node, replacing the old input. The old input will instead be input to the copy node.

Chaitin::build_ifg_physical

Mission

Creates a new IFG that is based on actual used registers. It also produces gpr and float frequencies for each block, they indicate if the block is a High Pressure Region (HRP) or Low Register Pressure (LRP) region. Each block also contains a gpr and float HRP index. If the HRP index is 0, the whole block is a HRP region. If the HRP index is greater than the number of instructions in the block, the block is a LRP region. If the HRP index is in between 0 and number of instructions for the block, the block transitions from LRP to HRP at the HRP index.
An area for each LRG is also calculated, togheter with a cost. The area and cost are then used to determine a score for the LRG. A high score keeps the LRG from being spilled.

Needs

The CFG with nodes annd a new liveness since the copies might been added and the virtual IFG phase made the LIVE_OUT a LIVE_IN.

Produces

Returns if we need to spill or not. Also creates an IFG and per block register pressures for both gpr and float, as well as HRP indices for gpr and float. The HPR indices determines if the block is HRP, LRP or has a transition from LRP to HRP.
Also produces an area and a cost for each LRG, that will be used to determine a score for each LRG.

Detailed

Each LRG has a _register_pressure and a _num_regs.

```
uint _register_pressure;
uint _num_regs;
```

The _register_pressure is platform DEPENDENT. The register pressure is used in the splitting heuristics. The _register_pressure tells how many platform registers that are needed to hold the value of the node corresponding to the LRG. Here is the list register pressure list:

```
............RegL RegI RegFlags RegF RegD
  IA32       2    1    1        1    1
  IA64       1    1    1        1    1
  SPARC      2    2    2        2    2
  SPARCV9    2    2    2        2    2
  AMD64      1    1    1        1    1
```

The _num_regs is platform INDEPENDENT.
Used for:
Here is the list:

```
............RegL RegI RegFlags RegF RegD FatProj
  ALL        2    1    1        1    2    >2
```

On top of this, there is also an _area for each LRG. The _area is the sum of all blocks simultaneously live values. The _area is combined with a _cost to define a score.

The score() method is used to determine which LRG is more important than the other. A higher score makes the LRG more important and prevents it from being spilled.

The score is based on two things.

```
double _cost;
double _area;
```

When traversing the blocks (in any order) and the nodes (in backwards order), every node definition that is mapped to a LRG increases the cost of that LRG with the block frequency, _freq. If the node however is rematerializable, a cost will not be added.

Also, every use of a node mapped to a LRG, found when traversing the block, increases the LRG cost with 2 * _freq.

If the _area is big, i.e. the LRG is covering a large area, it's better to spill because more LRGs get freed up. The cost function looks like this:

```
double static raw_score(double cost, double area) {
  // 1.52588e-5 = 1/65536 == 1/INT
  return cost - (area * RegisterCostAreaRatio) * 1.52588e-5;
}


double LRG::score() const {
  // initial score
  double score = raw_score( _cost, _area);

  if (_area == 0.0) {
    return 1e35; // max double value
  }

  // If spilled once before,
  // we are unlikely to make progress again.
  if (_was_spilled2) {
    return score + 1e30;
  }

  // Tiny area relative to cost,
  // probably bad to spill
  if(_cost >= _area * 3.0) {
    return score + 1e17;
  }

  // Small area relative to cost,
  // probably bad to spill
  if ((_cost+_cost) >= _area * 3.0) {
    return score + 1e10;
  }

  return score;
}
```

Each block has a _reg_pressure, _freg_pressure, _ihrp_index and _fhrp_index. These are used for the splitting heuristics.

```
uint _freq; // Use frequency from previous executions
            // of this method. If this block has been
            // entered at every run, the value is 1.0
            // if it has never been entered on
            // previous runs, it's 0.0.

uint _reg_pressure;
uint _ihrp_index;
uint _freg_pressure;
uint _fhrp_index;
```

There is also a suggested register pressure used in the register allocator when coloring. Here is the list:

```
.....................INTPRESSURE FLOATPRESSURE
  IA32                6            6
  IA64               50           41
  SPARC              48 (24)      52 (26)
  SPARCV9            48 (24)      52 (26)
  AMD64              14           15
```

The blocks are iterated and the nodes are iterated within each block in backwards order.

If the gpr or float register pressures at a certain point in the block are greater than the INTPRESSURE or FLOATPRESSURE, that point is in a HRP region for that type of registers. If the register pressure is lower or equal to the INTPRESSURE and/or FLOATPRESSURE at a certain in the block, that point is in a LRP region.

An initial block frequency is calculated for each block. We use two temporary arrays through-out the build IFG process to keep track of the block frequencies and possible HRP indices.

```
uint pressure[2];
uint hrp_index[2];
```

When the nodes are iterated, they are checked if they are within the LIVE_OUT set. If they are, they are removed from the LIVE_OUT set and the block register pressure is lowered with the LRG's register pressure.
If the node was not found in the LIVE_OUT, and is not a projection node, it is removed from the CFG because it is dead. (This due to previous spilling and rematerialization.)

Each time the block's register pressure is lowered, it is compared to the INTPRESSURE or FLOATPRESSURE, depending on the type of the LRG.

If the register pressure is now equal to the INTPRESSURE or FLOATPRESSURE, the current node index - 1 in the block is added as the HRP index for the block. This means we have found a transition from LRP to HRP in the block. Only the latest LRP to HRP transition is stored. (Since we go backwards in the block, we come from a LRP to a HRP).

At each node that is not a phi node, all the inputs are added to the LIVE_OUT, if they are virtual registers. The block's register pressure is also increased with the register pressure of all uses that are virtual registers.

When a node with a LRG is encountered in the traversing of nodes, the cost of the block, that is the block frequency, _freq, times the instruction count, is subtracted from the area of the lrg.

```
double cost = b->_freq * instruction_count
LRG._area -= cost;
```

If the node is a spill copy and only used once immediately after its define, it is given an LRG area of 0.0. This will ensure it to not be spilled.

As mentioned earlier, we now remove the node from the live out and reduce the register pressure of the block. We also map this nodes index - 1 to be a HRP index if the block pressure is now equal to INTPRESSURE or FLOATPRESSURE depending on type.

Finally, if the node is a copy (not necessarily a MachSpillCopy, it could also be a SSI, SSL, SSP, SSF or SSD node, which are stack slots nodes used only on x86_32 and x86_64. This is an old thing when no SSE support is available) then we should find the input for that copy and fetch its possible LRG and reduce the block cost of area of the LRG.

To be continued..