GCC Internals
Control and data flow support

Diego Novillo
`dnovillo@google.com`

November 2007

Control/Data Flow support

- Call Graph (cgraph)
- Control Flow Graph (CFG)
- Static Single Assignment in GIMPLE (SSA)
- Loop Nest Optimizations
  - Natural loops
  - Scalar evolutions
  - Data dependency tests
- Data-flow analysis in RTL (DF)

Call Graph

- Every internal/external function is a node of type `struct cgraph_node`
- Call sites represented with edges of type `struct cgraph_edge`
- Every cgraph node contains
  - Pointer to function declaration
  - List of callers
  - List of callees
  - Nested functions (if any)
- Indirect calls are not represented

Call Graph

- Callgraph manager drives intraprocedural optimization passes
- For every node in the callgraph, it sets `cfun` and `current_function_decl`
- IPA passes must traverse callgraph on their own
- Given a cgraph node

  `DECL_STRUCT_FUNCTION (node->decl)`

  points to the `struct function` instance that contains all the necessary control and data flow information for the function

Control Flow Graph

- Built early during lowering
- Survives until late in RTL
  - Right before machine dependent transformations (`pass_machine_reorg`)
- In GIMPLE, instruction stream is physically split into blocks
  - All jump instructions replaced with edges
- In RTL, the CFG is laid out over the double-linked instruction stream
  - Jump instructions preserved

Using the CFG

- Every CFG accessor requires a `struct function` argument
- In intraprocedural mode, accessors have shorthand aliases that use `cfun` by default
- CFG is an array of double-linked blocks
- The same data structures are used for GIMPLE and RTL
- Manipulation functions are callbacks that point to the appropriate RTL or GIMPLE versions

Using the CFG - Callbacks

- Declared in `struct cfg_hooks`

```
create_basic_block
redirect_edge_and_branch
delete_basic_block
can_merge_blocks_p
merge_blocks
can_duplicate_block_p
duplicate_block
split_edge
...
```

- Mostly used by generic CFG cleanup code
- Passes working with one IL may make direct calls

Using the CFG - Accessors

`basic_block_info_for_function(fn)` `basic_block_info`	Sparse array of basic blocks
`BASIC_BLOCK_FOR_FUNCTION(fn, n)` `BASIC_BLOCK (n)`	Get basic block N
`n_basic_blocks_for_function(fn)` `n_basic_blocks`	Number of blocks
`n_edges_for_function(fn)` `n_edges`	Number of edges
`last_basic_block_for_function(fn)` `last_basic_block`	First free slot in array of blocks (≠ `n_basic_blocks`)
`ENTRY_BLOCK_PTR_FOR_FUNCTION(fn)` `ENTRY_BLOCK_PTR`	Entry point
`EXIT_BLOCK_PTR_FOR_FUNCTION(fn)` `EXIT_BLOCK_PTR`	Exit point

Using the CFG - Traversals

- The block array is sparse, never iterate with

```
for (i = 0; i < n_basic_blocks; i++)
```

- Basic blocks are of type `basic_block`
- Edges are of type `edge`
- Linear traversals

```
FOR_EACH_BB_FN (bb, fn)
  FOR_EACH_BB (bb)

FOR_EACH_BB_REVERSE_FN (bb, fn)
  FOR_EACH_BB_REVERSE (bb)

FOR_BB_BETWEEN (bb, from, to, {next_bb|prev_bb})
```

Using the CFG - Traversals

- Traversing successors/predecessors of block bb

```
edge e;
edge_iterator ei;
FOR_EACH_EDGE (e, ei, bb->{succs|preds})
    do_something (e);
```

- Linear CFG traversals are essentially random
- Ordered walks possible with dominator traversals
  - Direct dominator traversals
  - Indirect dominator traversals via walker w/ callbacks

Using the CFG - Traversals

- Direct dominator traversals
  - Walking all blocks dominated by **bb**

```
for (son = first_dom_son (CDI_DOMINATORS, bb);
    son;
    son = next_dom_son (CDI_DOMINATORS, son))
```

  - Walking all blocks post-dominated by **bb**

```
for (son = first_dom_son (CDI_POST_DOMINATORS, bb);
    son;
    son = next_dom_son (CDI_POST_DOMINATORS, son)
```

  - To start at the top of the CFG

```
    FOR_EACH_EDGE (e, ei, ENTRY_BLOCK_PTR->succs)
      dom_traversal (e->dest);
```

Using the CFG - Traversals

- `walk_dominator_tree()`
- Dominator tree walker with callbacks
- Walks blocks and statements in either direction
- Up to six walker callbacks supported

  Before **and** after dominator children

  x2

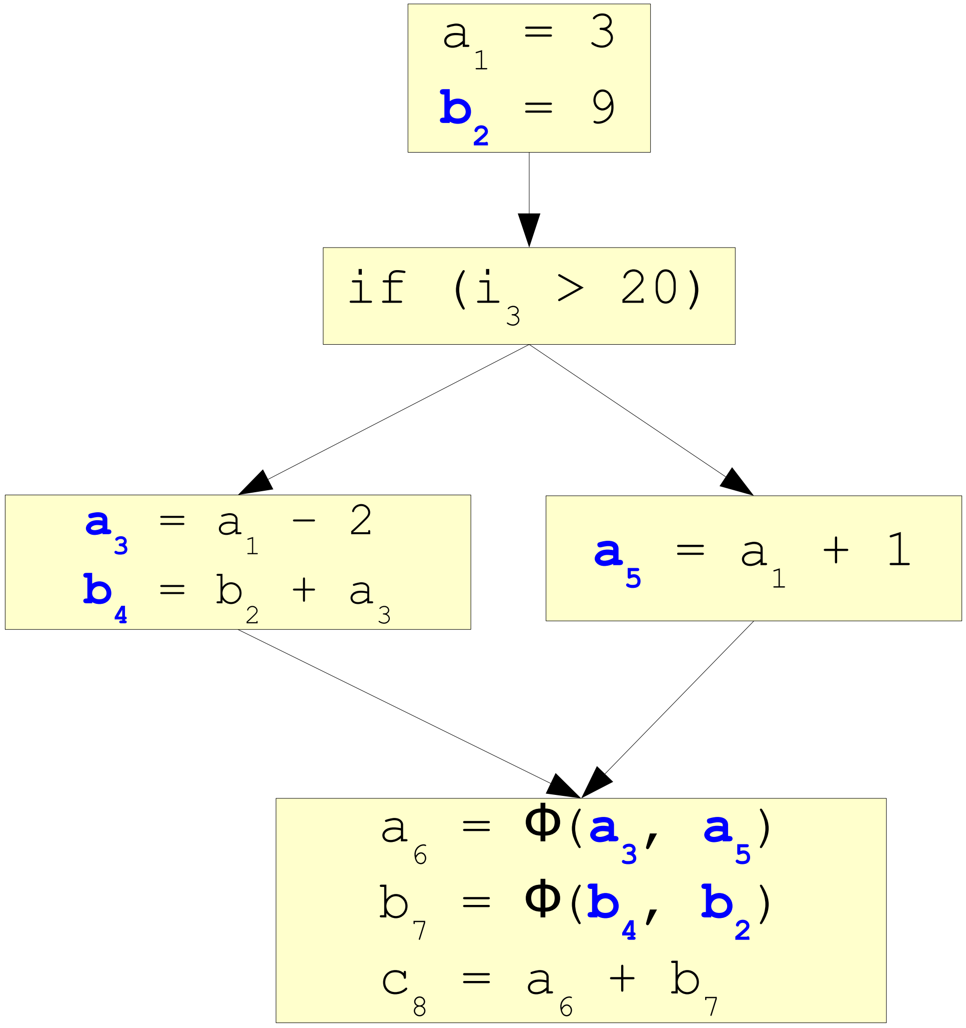
  1. Before walking statements
  2. Called for every GIMPLE statement in the block
  3. After walking statements

- Walker can also provide block-local data to keep pass-specific information during traversal

SSA Form

Static Single Assignment (SSA)

- Versioning representation to expose data flow explicitly
- Assignments generate new versions of symbols
- Convergence of multiple versions generates new one (Φ functions)



SSA Form

- Rewriting (or standard) SSA form
  - Used for real operands
  - Different names for the same symbol are *distinct objects*
  - overlapping live ranges (OLR) are allowed
  - Program is taken out of SSA form for RTL generation (new symbols are created to fix OLR)

$$\texttt{if } (\mathbf{x_2} > 4)$$
$$z_5 = \mathbf{x_3} - 1$$

$x_2 \rightarrow$ x.1

$x_3 \rightarrow$ x

$z_5 \rightarrow$ z

```
if (x.1 > 4)
  z = x - 1
```

Conversion to normal form

SSA Form

- Factored Use-Def Chains (FUD Chains)
  - Also known as Virtual SSA Form
  - Used for virtual operands.
  - All names refer to the *same object*.
  - Optimizers may **not** produce OLR for virtual operands.

Memory copies!

$\texttt{if } (\mathbf{x_2[2]} > 4)$
$z_5 = \mathbf{x_3[5]} - 1$

$\mathbf{x_2} \rightarrow \mathbf{x.1}$
$\mathbf{x_3} \rightarrow \mathbf{x}$
$z_5 \rightarrow z$

$\texttt{if } (\mathbf{x.1[2]} > 4)$
$z = \mathbf{x[5]} - 1$

Conversion to normal form

Virtual SSA Form

- VDEF operand needed to maintain DEF-DEF links
- They also prevent code movement that would cross stores after loads
- When alias sets grow too big, static grouping heuristic reduces number of virtual operators in aliased references

```
foo (i, a, b, *p)
{
  p_2 = (i_1 > 10) ? &a : &b

  # a_4 = VDEF <a_11>
  a = 9;

  # a_5 = VDEF <a_4>
  # b_7 = VDEF <b_6>
  *p_2 = 3;

  # VUSE <a_5>
  t1_8 = a;

  t3_10 = t1_8 + 5;
  return t3_10;
}
```

Incremental SSA form

SSA forms are kept up-to-date incrementally

Manually

- As long as SSA property is maintained, passes may introduce new SSA names and PHI nodes on their own
- Often this is the quickest way

Automatically using `update_ssa`

- Marking individual symbols (`mark_sym_for_renaming`)
- name → name mappings (`register_new_name_mapping`)
- Passes that invalidate SSA form must set `TODO_update_ssa`
- Symbols with OLRs must not be marked for renaming

SSA Implementation

- `tree-into-ssa.c`
  - Pass to put function in SSA form (`pass_build_ssa`)
  - Helpers to incrementally update SSA form (`update_ssa`)
- `tree-outof-ssa.c`
  - Pass to take function out of SSA form (`pass_del_ssa`)
- `tree-ssa.c`
  - Helpers for maintaining SSA data structures
  - SSA form verifiers

Loop Nest Optimization

- Based on natural loops
- Works on GIMPLE and RTL
- Number of iterations
- Induction variables (scalar evolutions)
- Data dependences
  - Single/Multiple/Zero IV generalized Banerjee tests
  - Omega test

LNO – Loop Analysis and Manipulation

- Loop discovery
  - `loop-init.c:loop_optimizer_init` builds loop tree
  - `loop-init.c:loop_optimizer_finalize` releases loop structures
- Loop discovery can enforce certain properties
  - Force loops to have only one/many latch blocks
  - Force loops to have preheader blocks
  - Mark irreducible regions

Useful for unrolling, peeling, etc

- Loop closed SSA form (`rewrite_into_loop_closed_ssa`)
  - Additional PHI nodes ensure that no SSA name is used outside the loop that defines it

LNO – Loop analysis

- Number of loops: `number_of_loops`, `get_loop`
- Loop nesting: `flow_loop_nested_p`, `find_common_loop`
- Loop bodies: `flow_bb_inside_loop_p`, `get_loop_body`, `get_loop_body_in_dom_order`, `get_loop_body_in_bfs_order`
- Exit edges and exit blocks: `loop_exit_edge_p`, `get_loop_exit_edges`, `single_exit`
- Pre-header and latch edges: `loop_preheader_edge`, `loop_latch_edge`
- Loop iteration: `FOR_EACH_LOOP`

LNO – Scalar Evolutions

- Based on chains of recurrences (chrec)

```
chrec(v) = {init, +, step}
```

- Given an SSA name N and loop L
  - `analyze_scalar_evolution (l, n)` returns the chrec for N in loop L
  - `instantiate_parameters (l, chrec)` tries to give values to the symbolic expressions `init` and `step`
  - `initial_condition_in_loop_num` retrieves initial value
  - `evolution_part_in_loop_num` retrieves step value
- Affine induction variable support in `tree-affine.c`

LNO – Dependence Analysis

- `compute_data_dependences_for_loop`
  - Returns list of memory references in the loop
  - Returns list of data dependence edges for the loop
- Given a data dependence edge
  - `DDR_A`, `DDR_B` are the two memory references
  - `DDR_ARE_DEPENDENT` is
    - `chrec_known` No dependence
    - `chrec_dont_know` Could not analyze dependence
    - `NULL` They are dependent

LNO – Linear transformations

- Based on lambda-code representation
- Suitable for transformations that can be expressed as linear transformations of iteration space (interchange, reversal)
- Support functions in `lambda-*.[ch]`
- Loop nest must be converted to/from a lambda loop nest for applying transformations

1. `gcc_loopnest_to_lambda_loopnest`
2. `lambda_loopnest_transform`
3. `lambda_loopnest_to_gcc_loopnest`

LNO - Optimizations

- GIMPLE
  - Loop invariant motion, unswitching, interchange, unrolling (`pass_lim`, `pass_tree_unswitch`, `pass_linear_transform`, `pass_iv_optimize`)
  - Predictive commoning (`pass_predcom`)
  - Vectorization (`pass_vectorize`)
  - Array prefetching (`pass_loop_prefetch`)
  - IV optimizations (`pass_iv_optimize`)
- RTL
  - Loop invariant motion, unswitching, unrolling, peeling (`pass_rtl_move_loop_invariants`, `pass_rtl_unswitch`, `pass_rtl_unroll_and_peel_loops`)
  - Decrement and branch instructions (`pass_rtl_doloop`)

Data Flow Analysis on RTL

- General framework for solving dataflow problems
- A separate representation of each RTL instruction describes sets of defs and uses in each insn
- Representation is kept up-to-date as the IL is modified
- Available between `pass_df_initialize` and `pass_df_finish`
- Implemented in `df-core.c`, `df-problems.c` and `df-scan.c`

Data Flow Analysis on RTL

- Three main steps
  - `df_*_add_problem`
    Adds a new problem to solve: reaching defs (`rd`), live variables (`live`), def-use or use-def chains (`chain`).
  - `df_analyze`
    Solves all the problems added
    Each basic block ends up with the corresponding IN and OUT sets (`DF_*_BB_INFO`)
  - `df_finish_pass`
    Removes data-flow problems
- Data flow analysis may be done globally or on a subset of nodes

Data Flow Analysis on RTL

- Scanning allocates a descriptor for every register defined or used in each instruction
  – Changes to the instruction need to be reflected into the descriptor
- Rescanning support exists for
  – Immediate updates
  – Deferred updates
  – Total updates
  – Manual updates