GCC Internals
Passes

Diego Novillo
dnovillo@google.com

November 2007

Passes

- Scheduled in `passes.c:init_optimization_passes`
- Three levels of processing: IPA, GIMPLE, RTL
- Three kinds of passes
  - Initializers: `pass_referenced_vars`, `pass_build_cfg`
  - Analysis: `pass_build_ssa`
  - Optimizations: `pass_vrp`
- TODO items determine cleanup actions to perform before/after a pass: `TODO_update_ssa`, `TODO_dump_func`

Passes

- Passes hierarchically grouped in families
- `all_lowering_passes`
- `all_ipa_passes`
  - `pass_early_local_passes`
- `all_passes`
  - `pass_all_optimizations`
    - `pass_tree_loop`
  - `pass_rest_of_compilation`
    - `pass_loop2`
    - `pass_post_reload`

all_lowering_passes

Google

- Process the IL to be ready for optimization
- pass_remove_useless_stmts
  - Simplistic dead code eliminator that needs no data flow
- pass_lower_{omp,cf,eh}
  - Put IL in low GIMPLE form
- pass_build_cfg
- pass_build_cgraph_edges

all_ipa_passes

Google

- pass_ipa_early_inline
  - Simplistic inlining using local info
  - Used with profiling to reduce instrumentation cost
- pass_ipa_cp
- pass_ipa_inline
  - Analyze cgraph and decide inlining plan
  - Greedy algorithm favouring small functions and functions called once
- pass_ipa_pta
- pass_ipa_struct_reorg

pass_early_local_passes

Google

- Put function in SSA form and clean it up
- `pass_tree_profile`
- `pass_cleanup_cfg`
- `pass_referenced_vars`
- `pass_build_ssa`
- Several scalar cleanups: `pass_ccp`, `pass_forwprop`, `pass_simple_dse`, `pass_dce`, ...

all_passes

- GIMPLE scalar optimizations

```
pass_apply_inline  pass_ccp      pass_fre            pass_dce
pass_copy_prop     pass_vrp      pass_dominator      pass_ch
pass_sra           pass_reassoc  pass_dse            pass_pre
...
```

- GIMPLE loop optimizations

```
pass_tree_loop_init      pass_lim
pass_predcom             pass_tree_unswitch
pass_empty_loop          pass_linear_transform
pass_iv_canon            pass_if_conversion
pass_vectorize           pass_parallelize_loops
pass_iv_optimize         ...
```

- RTL optimizations

```
pass_expand        pass_into_cfg_layout_mode  pass_cse
pass_gcse          pass_loop2                 pass_inc_dec
pass_combine       pass_sms                   pass_sched
pass_local_alloc   pass_global_alloc          pass_post_reload
...
```

Adding a new pass

Google

- To implement a new pass
  - Add a new file to `trunk/gcc` or edit an existing pass
  - Add a new target rule in `Makefile.in`
  - If a flag is required to trigger the pass, add it to `common.opt`
  - Create an instance of `struct tree_opt_pass`
  - Declare it in `tree-pass.h`
  - Sequence it in `init_optimization_passes`
  - Add a gate function to read the new flag
  - Document pass in `trunk/gcc/doc/invoke.texi`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (voi

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

Extension for dump file is `.<static_pass_number>[itr].<name>`

e.g., `prog.c.158r.greg`

`i` for IPA passes
`t` for GIMPLE (tree) passes
`r` for RTL passes

`static_pass_number` is automatically assigned by pass manager

Letter used by the -d switch to enable a specific RTL dump (backward compatibility)

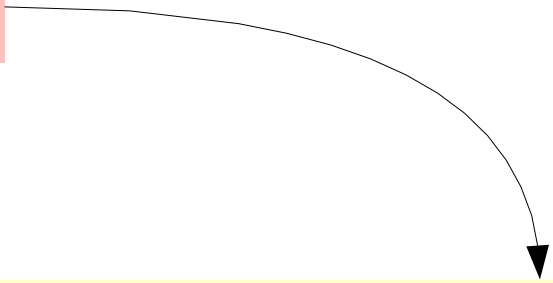
Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

If function `gate()` returns true, then the pass entry point function `execute()` is called

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_requi
  unsigned int properties_provi
  unsigned int properties_destr
  unsigned int todo_flags_start
  unsigned int todo_flags_finis
  char letter;
};
```



Passes may be organized hierarchically

`sub` points to first child pass

`next` points to sibling class

Passes are chained together with `NEXT_PASS` in `init_optimization_passes`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

Each pass can define its own separate timer

Timers are started/stopped automatically by pass manager

Timer handles (`timevars`) are defined in `timevar.def`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

Properties required, provided and destroyed are defined in `tree-pass.h`

Common properties
`PROP_cfg`
`PROP_ssa`
`PROP_alias`
`PROP_gimple_lcf`

Describing a pass

```
struct tree_opt_pass
{
  const char *name;

  bool (*gate) (void);

  unsigned int (*execute) (void);

  struct tree_opt_pass *sub;
  struct tree_opt_pass *next;

  int static_pass_number;

  unsigned int tv_id;

  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
  char letter;
};
```

Cleanup or bookkeeping actions that the pass manager should do before/after the pass

Defined in `tree-pass.h`

Common actions
`TODO_dump_func`
`TODO_verify_ssa`
`TODO_cleanup_cfg`
`TODO_update_ssa`

Available features

Google

- APIs available for
  - CFG: block/edge insertion, removal, dominance information, block iterators, dominance tree walker.
  - Statements: insertion in block and edge, removal, iterators, replacement.
  - Operands: iterators, replacement.
  - Loop discovery and manipulation.
  - Data dependency information (scalar evolutions framework).

Available features

Google

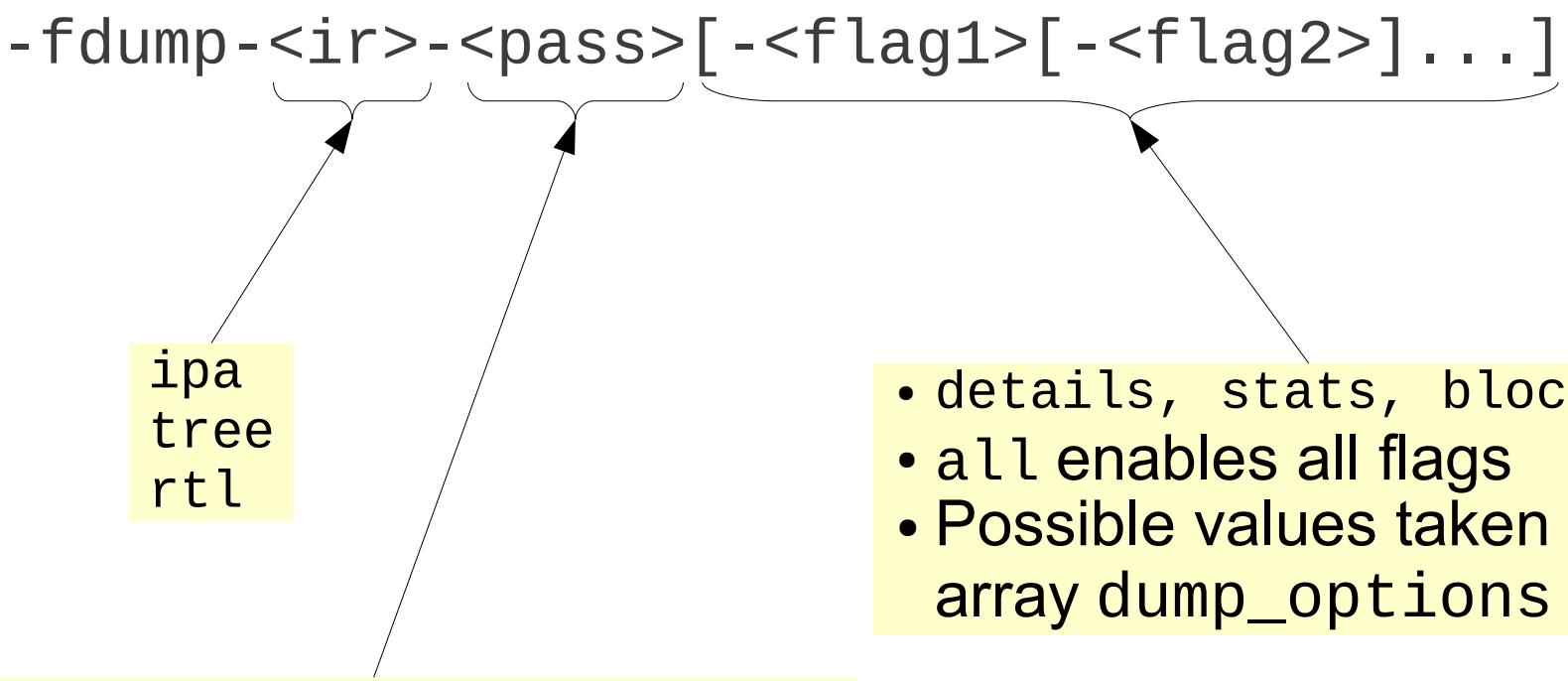
- Other available infrastructure
  - Debugging dumps (`-fdump-tree-...`)
  - Timers for profiling passes (`-ftime-report`)
  - CFG/GIMPLE/SSA verification (`--enable-checking`)
  - Generic value propagation engine with callbacks for statement and Φ node visits.
  - Generic use-def chain walker.
  - Support in test harness for scanning dump files looking for specific transformations.
  - Pass manager for scheduling passes and describing interdependencies, attributes required and attributes provided.

Debugging

Debugging dumps

Google

Most passes understand the `-fdump` switches



Debugging dumps

Google

- Adding dumps to your pass
  - Specify a name for the dump in `struct tree_opt_pass`
  - To request a dump at the end of the pass add `TODO_dump_func` in `todo_flags_finish` field
- To emit debugging information during the pass
  - Variable `dump_file` is set if dumps are enabled
  - Variable `dump_flags` is a bitmask that specifies what flags were selected
  - Some common useful flags: `TDF_DETAILS`, `TDF_STATS`

Using gdb

- Never debug the `gcc` binary, that is only the driver
- The real compiler is one of `cc1`, `jc1`, `f951`, ...

```
$ <bld>/bin/gcc -O2 -v -save-temps -c a.c
  Using built-in specs.
  Target: x86_64-unknown-linux-gnu
  Configured with: [ ... ]
  [ ... ]
  End of search list.
  <path>/cc1 -fpreprocessed a.i -quiet -dumpbase a.c
  -mtune=generic -auxbase a -O2 -version -o a.s
```

```
$ gdb --args <path>/cc1 -fpreprocessed a.i -quiet -dumpbase
  a.c -mtune=generic -auxbase a -O2 -version -o a.s
```

Using gdb

- The build directory contains a `.gdbinit` file with many useful wrappers around debugging functions
- When debugging a bootstrapped compiler, try to use the stage 1 compiler
- The stage 2 and stage 3 compilers are built with optimizations enabled (may confuse debugging)
- To recreate testsuite failures, cut and paste command line from

```
<bld>/gcc/testsuite/{gcc,gfortran,g++,java}/*.log
```

Timing passes

- Timers defined in `timevar.def`
- Start timer with `timevar_push`
- Stop timer with `timevar_pop`
- Timings are reported if compiling with `-ftime-report`
- Timers use the best standard mechanism they can find

`times` » `getrusage` » `clock`

Memory usage statistics

Google

- Enabled with -fmem-report
- To gather extremely detailed memory usage, configure with

```
--enable-gather-detailed-mem-stats
```

Debug counters

Google

- Mechanism for tracing and counting events
- `dbg_cnt` increments counter
  - Returns false when threshold is crossed
  - Returns true otherwise
- Allows to control when to apply a transformation
- Counters are defined in dbgcnt.def
- Thresholds are set with

```
-fdbg-cnt=name1:N1,name2:N2, ...
```

Case study - VRP

Value Range Propagation

Google

- Based on Patterson's range propagation for jump prediction [PLDI'95]
  - No branch probabilities (only taken/not-taken)
  - Only a single range per SSA name.

```
for (int i = 0; i < a->len; i++)
  {
    if (i < 0 || i >= a->len)
      throw 5;
    call (a->data[i]);
  }
```

- Conditional inside the loop is unnecessary.

Value Range Propagation

Google

Two main phases

Range assertions

Conditional jumps provide info on value ranges

```
if (a_3 > 10)
  a_4 = ASSERT_EXPR <a_3, a_3 > 10>
  ...
else
  a_5 = ASSERT_EXPR <a_4, a_4 <= 10>
```

Now we can associate a range value to a_4 and a_5.

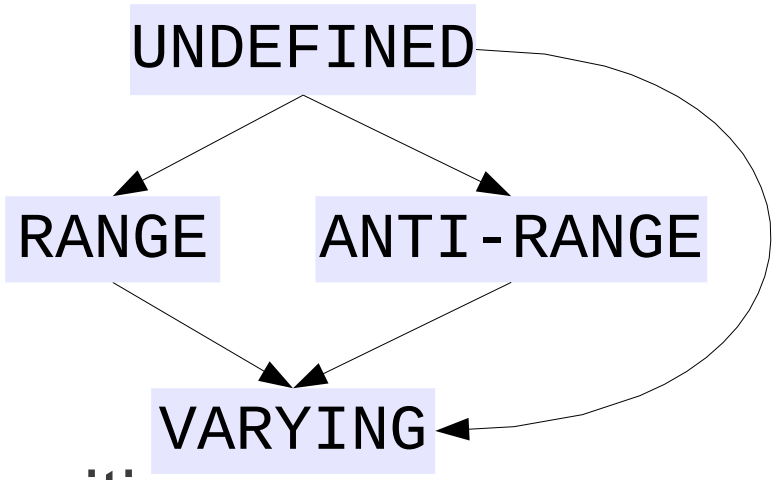
Range propagation

Generic propagation engine used to propagate value ranges from ASSERT_EXPR

Value Range Propagation

Google

- Two range representations
  - Range [MIN, MAX] → MIN <= N <= MAX
  - Anti-range ~[MIN, MAX] → N < MIN or N > MAX
- Lattice has 4 states



- No upward transitions

Propagation engine

- Generalization of propagation code in SSA-CCP
- Simulates execution of statements that produce "*interesting*" values
- Flow of control and data are simulated with work lists.
  - CFG work list → control flow edges.
  - SSA work list → def-use edges.
- Engine calls-back into VRP at every statement and PHI node

Propagation engine

Usage

```
ssa_propagate (visit_stmt, visit_phi)
```

Returns 3 possible values for statement **S**

SSA_PROP_INTERESTING

S produces an interesting value

If **S** is not a jump, `visit_stmt` returns name N_i holding the value

Def-use edges out of N_i are added to SSA work list

If **S** is jump, `visit_stmt` returns edge that will always be taken

SSA_PROP_NOT_INTERESTING

No edges added, **S** may be visited again

SSA_PROP_VARYING

Edges added, **S** will *not* be visited again

Propagation engine

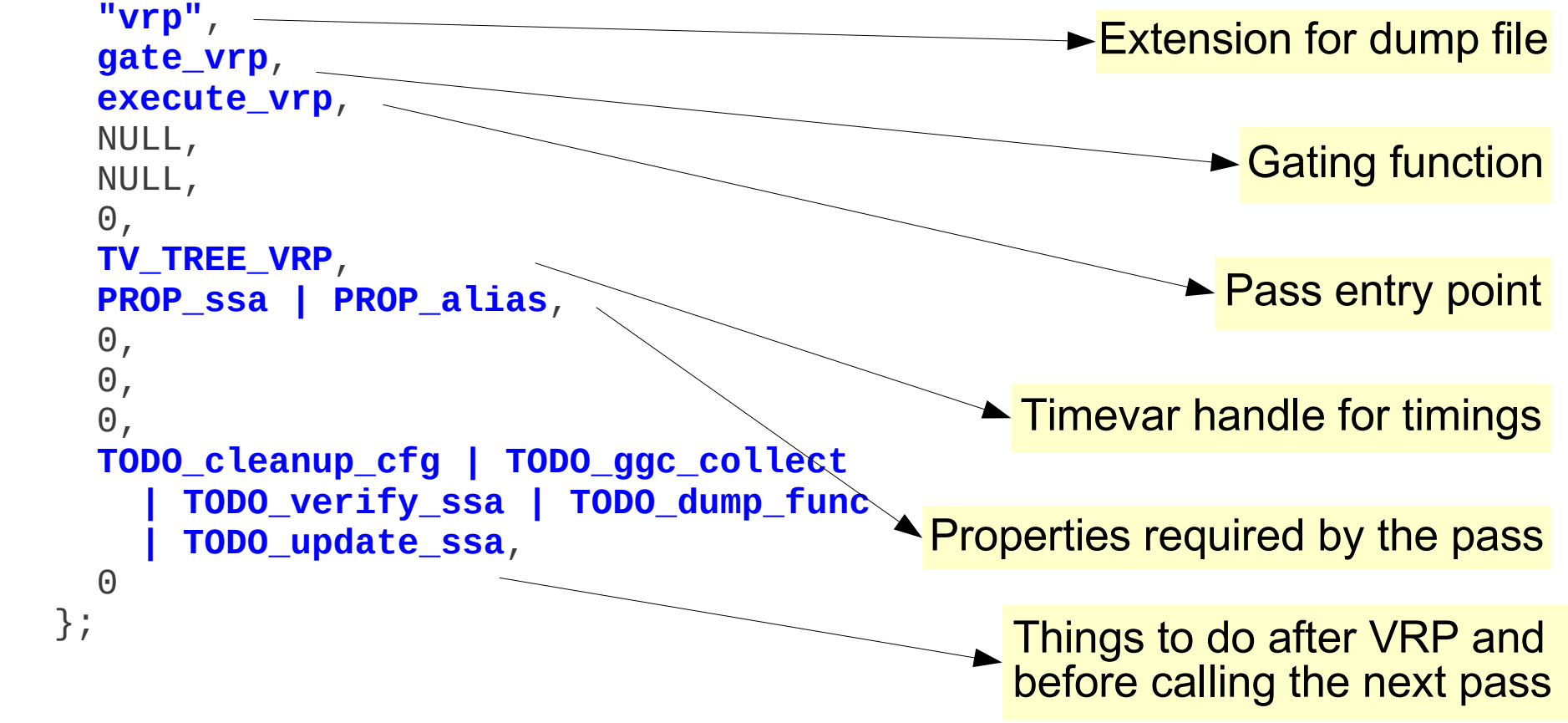
Google

- `visit_phi` has similar semantics to `visit_stmt`
  - PHI nodes are merging points, so they need to “intersect” all the incoming arguments
- Simulation terminates when both SSA and CFG work lists are drained
- Values should be kept in an array indexed by SSA version number
- After propagation, call `substitute_and_fold` to do final replacement in IL

Google

Pass declaration in gcc/tree-vrp.c

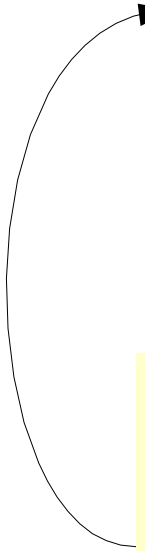
```
struct tree_opt_pass pass_vrp =
  {
    "vrp",
    gate_vrp,
    execute_vrp,
    NULL,
    NULL,
    0,
    TV_TREE_VRP,
    PROP_ssa | PROP_alias,
    0,
    0,
    0,
    TODO_cleanup_cfg | TODO_ggc_collect
      | TODO_verify_ssa | TODO_dump_func
      | TODO_update_ssa,
    0
  };
```



Implementing VRP

Add `-ftree-vrp` to `common.opt`

```
ftree-vrp
Common Report Var(flag_tree_vrp) Init(0) Optimization
Perform Value Range Propagation on trees
```



`Common`	This flag is available for all languages
`Report`	`-fverbose-asm` should print the value of this flag
`Var`	Global variable holding the value of this flag
`Init`	Initial (default) value for this flag
`Optimization`	This flag belongs to the optimization family of flags

Implementing VRP

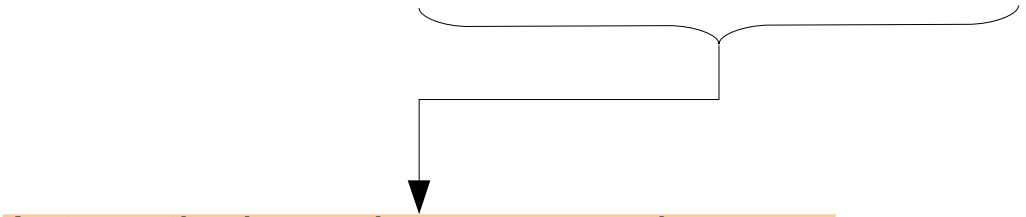
Google

Add gating function

```
static bool
gate_vrp (void)
{
  return flag_tree_vrp != 0;
}
```

Add new entry in Makefile.in

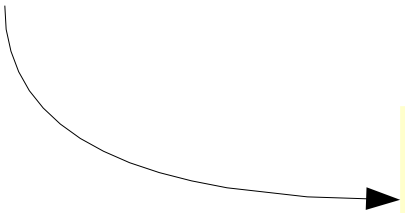
- Add `tree-vrp.o` to `OBJS-common` variable
- Add rule for `tree-vrp.o` listing **all** dependencies



Google

Add entry point function

```
static unsigned int
execute_vrp (void)
{
  insert_range_assertions ();
  ...
  ssa_propagate (vrp_visit_stmt,vrp_visit_phi_node);
  ...
  remove_range_assertions ();

  return 0;
}
```



If the pass needs to add TODO items, it should return them here

Implementing VRP

Schedule VRP in init_optimization_passes

```
init_optimization_passes (void)
{
    ...
    NEXT_PASS (pass_merge_phi);
    NEXT_PASS (pass_vrp);
    ...
    NEXT_PASS (pass_reassoc);
    NEXT_PASS (pass_vrp);
    ...
}
```