Individual Exercise 2

Tan Yee Jian

November 1, 2021

Contents

Each section corresponds to each of the problems raised in the Assignment.

1 General flow of compilation

The main idea in the design of GCC is "**passes**"[2]. GCC compiles source code (eg. from `C`) to assembly (eg. to `x86-64` assembly) by using many, many passes. On the top level, passes translate one representation of the code to another representation, that gets closer and closer to machine code, and eventually assembly code.

Compiling code in GCC while dumping all passes yield all the passes as files:

```
a-test.c.005t.original
a-test.c.006t.gimple
a-test.c.009t.omplower
a-test.c.010t.lower
a-test.c.013t.eh
a-test.c.015t.cfg
a-test.c.017t.ompexp
a-test.c.022t.fixup_cfg1
a-test.c.023t.ssa
a-test.c.025t.nothrow
...
a-test.c.327r.shorten
a-test.c.328r.nothrow
a-test.c.329r.dwarf2
a-test.c.330r.final
a-test.c.331r.dfinish
a-test.c.332t.statistics
a-test.c.333t.earlydebug
a-test.c.334t.debug
test.c
test.s
```

Listing 1: Files generated in the passes.

1.1 Source code

This is the user input in any of the GCC-supported language such as `C` and `Fortran`.

1.2 GENERIC (language-independent Abstract Syntax Tree)

User-input source code is first parsed using recursive descent into an Abstract Syntax Tree. For example, the `C` code is parsed in the function below as an entry point, in This is parsed in `gcc/c/c-parser.c:21961`:

```c
21961 // Begin C-parser entry point
21962 void c_parse_file (void)
21963 {
21964   /* Use local storage to begin.  If the first token is a pragma, parse it.
21965      If it is #pragma GCC pch_preprocess, then this will load a PCH file
21966      which will cause garbage collection.  */
21967   c_parser tparser;
21968 
21969   memset (&tparser, 0, sizeof tparser);
21970   tparser.translate_strings_p = true;
21971   tparser.tokens = &tparser.tokens_buf[0];
```

```c
  the_parser = &tparser;

  if (c_parser_peek_token (&tparser)->pragma_kind == PRAGMA_GCC_PCH_PREPROCESS)
    c_parser_pragma_pch_preprocess (&tparser);
  else
    c_common_no_more_pch ();

  the_parser = ggc_alloc<c_parser> ();
  *the_parser = tparser;
  if (tparser.tokens == &tparser.tokens_buf[0])
    the_parser->tokens = &the_parser->tokens_buf[0];

  /* Initialize EH, if we've been told to do so.  */
  if (flag_exceptions)
    using_eh_for_cleanups ();

  c_parser_translation_unit (the_parser);
  the_parser = NULL;
}
// End C-parser entry point
```

Since each language has a different syntax, hence different syntax tree structures. **GENERIC** specifies a **language-independent AST** structure to abstract all the AST of different languages into one common **AST**[1], facilitating translation to further IRs (such as GIMPLE, Tree-SSA, RTL) and optimization.

There are differences in how **GENERIC** is used in parsing.

- `C` parses directly into **GENERIC** but
- `Fortran` however parses first into a private representation to **GENERIC**, which is later then "lowered" into **GENERIC** and **GIMPLE**[2].

1.3 GIMPLE ("three-address codes" that has repeated assignment to variables)

This process of translating from other representations, such as **GENERIC** to **GIMPLE** is called "**gimplification**". The entry point for this pass is the function `gimplify_function_tree()` in `gcc/gimplify.c:15447`:

```
void
gimplify_function_tree (tree fndecl) {
  gimple_seq seq;
  gbind *bind;

  gcc_assert (!gimple_body (fndecl));

  if (DECL_STRUCT_FUNCTION (fndecl))
    push_cfun (DECL_STRUCT_FUNCTION (fndecl));
  else
    push_struct_function (fndecl);

  /* OMITTED CODE */

      /* Replace the current function body with the body
         wrapped in the try/finally TF.  */
      seq = NULL;
      gimple_seq_add_stmt (&seq, new_bind);
      gimple_set_body (fndecl, seq);
      bind = new_bind;
    }

  if (sanitize_flags_p (SANITIZE_THREAD)
      && param_tsan_instrument_func_entry_exit)
    {
      gcall *call = gimple_build_call_internal (IFN_TSAN_FUNC_EXIT, 0);
      gimple *tf = gimple_build_try (seq, call, GIMPLE_TRY_FINALLY);
      gbind *new_bind = gimple_build_bind (NULL, tf, NULL);
      /* Replace the current function body with the body
         wrapped in the try/finally TF.  */
      seq = NULL;
      gimple_seq_add_stmt (&seq, new_bind);
      gimple_set_body (fndecl, seq);
    }

  DECL_SAVED_TREE (fndecl) = NULL_TREE;
  cfun->curr_properties |= PROP_gimple_any;

  pop_cfun ();

  dump_function (TDI_gimple, fndecl);
}
```

Listing 2: Entry point to gimplification.

1.4 Tree-SSA (Single Static Assignment)

From this point onwards, the passes are mainly optimizations. It is managed by `gcc/passes.c` which executes passes as listed in `gcc/passes.def:29`.

```
29 /* All passes needed to lower the function into shape optimizers can
30    operate on.  These passes are always run first on the function, but
31    backend might produce already lowered functions that are not processed
32    by these passes.  */
33  INSERT_PASSES_AFTER (all_lowering_passes)
34  NEXT_PASS (pass_warn_unused_result);
35  NEXT_PASS (pass_diagnose_omp_blocks);
36  NEXT_PASS (pass_diagnose_tm_blocks);
37  NEXT_PASS (pass_omp_oacc_kernels_decompose);
38  NEXT_PASS (pass_lower_omp);
39  NEXT_PASS (pass_lower_cf);
40  NEXT_PASS (pass_lower_tm);
41  NEXT_PASS (pass_refactor_eh);
42  NEXT_PASS (pass_lower_eh);
43  NEXT_PASS (pass_coroutine_lower_builtins);
44  NEXT_PASS (pass_build_cfg);
45  NEXT_PASS (pass_warn_function_return);
46  NEXT_PASS (pass_coroutine_early_expand_ifns);
47  NEXT_PASS (pass_expand_omp);
48  NEXT_PASS (pass_warn_printf);
49  NEXT_PASS (pass_walloca, /*strict_mode_p=*/true);
50  NEXT_PASS (pass_build_cgraph_edges);
51  TERMINATE_PASS_LIST (all_lowering_passes)
52 // many more passes
```

Listing 3: Some of the passes specified in `gcc/passes.def`.

1.5 RTL (Register Transfer Language) RTL is a machine code for an abstract

machine with inifinitely many registers. RTL's syntax and several peephole optimizations will be included in the sections below.

1.6 Target assembly code

This is the assembly code in the target machine architecture, such as `ARMv7`, `RISCV` or `aarch64`. GCC's compilation end here, and the rest of the job is given to the `as` GNU Assembler.

There are passes which do not further "compile" the language into the next, more low-level version, which are known as "optimization passes".

2 Parsing of C

Recursive descent is used. Found in `gcc/c/c-parser.c:1800`

```
1800 static void
1801 c_parser_declaration_or_fndef (c_parser *parser, bool fndef_ok,
1802                                bool static_assert_ok, bool empty_ok,
1803                                bool nested, bool start_attr_ok,
1804                                tree *objc_foreach_object_declaration,
1805                                vec<c_token> omp_declare_simd_clauses,
1806                                bool have_attrs, tree attrs,
1807                                struct oacc_routine_data *oacc_routine_data,
1808                                bool *fallthru_attr_p)
1809 { ... }
```

Listing 4: Recursive descent function to parse declarations

which parses function declarations and more. Recursive Descent allows for better parsing error reporting as all cases can have individualized error messages. StackExchange post Example at `gcc/c/c-parser.c:1937`:

```
        {
          /* This is not C++ with its implicit typedef.  */
          richloc.add_fixit_insert_before ("struct ");
          error_at (&richloc,
                    "unknown type name %qE;"
                    " use %<struct%> keyword to refer to the type",
                    name);
        }
      else if (tag_exists_p (UNION_TYPE, name))
        {
          richloc.add_fixit_insert_before ("union ");
          error_at (&richloc,
                    "unknown type name %qE;"
                    " use %<union%> keyword to refer to the type",
                    name);
        }
      else if (tag_exists_p (ENUMERAL_TYPE, name))
        {
          richloc.add_fixit_insert_before ("enum ");
          error_at (&richloc,
                    "unknown type name %qE;"
                    " use %<enum%> keyword to refer to the type",
                    name);
        }
      else
        {
          auto_diagnostic_group d;
          name_hint hint = lookup_name_fuzzy (name, FUZZY_LOOKUP_TYPENAME,
                                              here);
          if (const char *suggestion = hint.suggestion ())
            {
              richloc.add_fixit_replace (suggestion);
              error_at (&richloc,
                        "unknown type name %qE; did you mean %qs?",
                        name, suggestion);
            }
          else
            error_at (here, "unknown type name %qE", name);
        }
```

Listing 5: Some detailed error messages in the C-parser

3 Intermediate code formats

In this section and the next, we use this running example of `C` code:

```c
int main() {
  int x = 1 + 2;
  char c = 's';
  while (x < 10) {
    x += 1;
  }
  return x;
}
```

Listing 6: Running example in this section, in `C`

3.1 Motivation

- Before GENERIC and GIMPLE were invented, the parsed AST for every language were immediately translated into Register Transfer Language (RTL), which is like "assembly language with infinite number of registers".
- The lack of a common structure caused each language to have to write a **AST to RTL** compiler, on top of the **parser**. The use of **GENERIC** abstracted this common work done by each language front-end.
- Directly translating from AST to RTL loses many properties of the code, "for example, array references, data types, references to program variables, control flow structures"[5]. Even function calls are expanded to more than one instructions, loosing the structure of functions, hence losing opportunities for function optimizations such as *folding* and *dead code elimination*.
- Solution: "GENERIC addresses the lack of a common tree representation among the various front ends. GIMPLE solves the complexity problems that facilitate the discovery of data and control flow in the program."[5]
- **GIMPLE** in three-address codes provides a good structure for optimizations, developed actively in research([4],[3]).

3.2 GENERIC

GENERIC is a common AST structure for all gcc-compatible languages. There are no optimizations happening at this point, until the code is "gimplified" or converted to **GIMPLE**, which is in three-address code.

3.3 GIMPLE

The `C` code compiles to the following **GIMPLE** code:

```
int main ()
{
  int D.1950;

  {
    int x;
    char c;

    x = 3;
    c = 115;
    goto <D.1947>;
    <D.1948>:
    x = x + 1;
    <D.1947>:
    if (x <= 9) goto <D.1948>; else goto <D.1946>;
    <D.1946>:
    D.1950 = x;
    return D.1950;
  }
  D.1950 = 0;
  return D.1950;
}
```

Listing 7: **GIMPLE** code generated by `C`

GIMPLE is basically three-address code, however many optimizations rely on code being in the **Static Single Assignment** form, called **Tree-SSA** or **SSA** in GCC. After some passes, GCC obtains the **SSA** as below:

```
int main ()
{
  char c;
  int x;
  int _4;

  <bb 2> :
  x_2 = 3;
  c_3 = 115;
  goto <bb 4>; [INV]

  <bb 3> :
  x_6 = x_1 + 1;

  <bb 4> :
  # x_1 = PHI <x_2(2), x_6(3)>
  if (x_1 <= 9)
```

```
    goto <bb 3>; [INV]
  else
    goto <bb 5>; [INV]

  <bb 5> :
  _4 = x_1;
  return _4;

}
```

At this point, many interesting peephole optimizations can already be done, such as “Tail-call optimization” and “Conditional constant propogation (CCP)”, which we will discuss in the last section.

3.4 Register Transfer Language (RTL)

RTL is described as an “assembly language with infinite number of registers”, inspired by LISP lists and describes the instructions to be outputed eventually in a slightly higher level. Internally it is a graph-like structure with references, but printed as nested brackets similar to LISP.

This will be explained in detail in the next section.

4 Explain RTL representation

We will explain using this RTL code, taken from the previous section. All the in-code comments are mine.

```
(note 1 0 3 NOTE_INSN_DELETED) ;; notes are just debugging information.
(note 3 1 13 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 13 3 2 2 NOTE_INSN_PROLOGUE_END)
(note 2 13 9 2 NOTE_INSN_FUNCTION_BEG)
(insn 9 2 10 2
;; insn are instructions that do not jump and are not function calls
    (set (reg/i:SI 0 ax)
    ;; sets the register with the constant below.
    ;; (reg/i:SI 0 ax):
    ;;   reg means registers
    ;;   /i here means the value is a scalar that is not part of an aggregate
    ;;   SI here represents the Single Integer mode of 4 bytes.
    ;;   0 is a hard register number.
        (const_int 10 [0xa])) "test.c":8:1 75 {*movsi_internal}
        ;; this is a constant integer of value 10, with representation 0xa.
     (nil))
(insn 10 9 14 2 (use (reg/i:SI 0 ax)) "test.c":8:1 -1
;; this is an instruction that loads data from the register for the return below.
     (nil))
(note 14 10 15 2 NOTE_INSN_EPILOGUE_BEG)
(jump_insn 15 14 16 2 (simple_return) "test.c":8:1 837 {simple_return_internal}
;; this is a jump-instruction.
     (nil)
 -> simple_return)
;; simple_return is the return instruction without the function-return epilogue.
(barrier 16 15 12)
;; barrier denotes the end of control flow.
(note 12 16 0 NOTE_INSN_DELETED)
```

Listing 8: RTL compiled, with in-code comment explanations

5 Three peephole optimizations in GCC

Peephole optimizations are found in the directory =gcc/config= and are sorted by each architecture. Peephole optimizations are defined with the following structure:

```
(define_peephole2
  [(set (match_operand:SI 0 "arm_general_register_operand" "")
        (plus:SI (match_operand:SI 1 "arm_general_register_operand" "")
                 (const_int -1)))
   (set (match_operand 2 "cc_register" "")
        (compare (match_dup 0) (const_int -1)))
   (set (pc)
        (if_then_else (match_operator 3 "equality_operator"
                        [(match_dup 2) (const_int 0)])
                      (match_operand 4 "" "")
                      (match_operand 5 "" "")))]
  "TARGET_32BIT && peep2_reg_dead_p (3, operands[2])"
  [(parallel[
    (set (match_dup 2)
         (compare:CC
          (match_dup 1) (const_int 1)))
    (set (match_dup 0) (plus:SI (match_dup 1) (const_int -1)))])
   (set (pc)
        (if_then_else (match_op_dup 3 [(match_dup 2) (const_int 0)])
                      (match_dup 4)
                      (match_dup 5)))]
  "operands[2] = gen_rtx_REG (CCmode, CC_REGNUM);
   operands[3] = gen_rtx_fmt_ee ((GET_CODE (operands[3]) == NE
                                  ? GEU : LTU),
                                 VOIDmode,
                                 operands[2], const0_rtx);"
)
```

Listing 10: Compacting 3 instructions into 2.

```
(define_peephole2
  [insn-pattern-1
   insn-pattern-2
   ...]
  "condition"
  [new-insn-pattern-1
   new-insn-pattern-2
   ...]
  "preparation-statements")
```

Listing 9: Structure of peephole definition.

5.1 Instruction compaction in =gcc/config/arm/arm.md:1183

This matches the instructions

```
sub  rd, rn, #1
cmn  rd, #1        (equivalent to cmp rd, #-1)
bne  dest
```

```
subs rd, rn, #1
bcs  dest        ((unsigned)rn >= 1)
```

into
which is a common loop idiom in cases such as `while(n--)`.

5.2 Replacing instruction in =gcc/config/arm/arm.md:9827

```
(define_peephole2
  [(set (reg:CC CC_REGNUM)
        (compare:CC (match_operand:SI 1 "register_operand" "")
                    (const_int 0)))
   (cond_exec (ne (reg:CC CC_REGNUM) (const_int 0))
              (set (match_operand:SI 0 "register_operand" "") (const_int 0)))
   (cond_exec (eq (reg:CC CC_REGNUM) (const_int 0))
              (set (match_dup 0) (const_int 1)))
   (match_scratch:SI 2 "r")]
  "TARGET_32BIT && peep2_regno_dead_p (3, CC_REGNUM)"
  [(parallel
    [(set (reg:CC CC_REGNUM)
          (compare:CC (const_int 0) (match_dup 1)))
     (set (match_dup 2) (minus:SI (const_int 0) (match_dup 1)))])
   (set (match_dup 0)
        (plus:SI (plus:SI (match_dup 1) (match_dup 2))
                 (geu:SI (reg:CC CC_REGNUM) (const_int 0))))]
)
```

Listing 11: Replacing conditional instructions with parallel ones.

This code replaces patterns of the form `Rd = (eq (reg1) (const_int0))` into the instructions:

```
negs Rd, reg1
adc  Rd, Rd, reg1
```

which is more efficient.

5.3 Parallelization of operations in `gcc/config/arm/arm.md:11401`

```
(define_peephole2
  [(set (match_operand:SI 0 "arm_general_register_operand" "")
        (match_operand:SI 1 "arm_general_register_operand" ""))
   (set (reg:CC CC_REGNUM)
        (compare:CC (match_dup 1) (const_int 0)))]
  "TARGET_ARM"
  [(parallel [(set (reg:CC CC_REGNUM) (compare:CC (match_dup 1) (const_int 0)))
              (set (match_dup 0) (match_dup 1))])]
  ""
)
```

Listing 12: Parallelizing two `set` instructions in `ARM`.

This peephole optimization basically takes two "set" (store) operations and paralellize them, utilizing the special instruction in ARM architecture.

6 Extra: Three non-peephole optimizations in GCC

Other than peephole optimizations, there are many other optimizations in GCC that are worthy of noting, occuring at both the RTL level and the Tree SSA level. We list 3 below.

6.1 Tail-call optimization (in `gcc/tree-tailcall.c`)

- Tail-calls occur when a recursive function returns a call to itself as part of explicit recursion. Since it happens at the end of a function, it is named as tail-call.
- Tail-call optimization aims to remove unnecessary recursion (which takes up the stack) with loops.
- Furthermore, functions where the tail-call is not explicit is also optimized by GCC using accumulators. Below is an example of the recursive `sum()` function:

```
int sum (int n)
{
  if (n > 0)
    return n + sum (n - 1);
  else
    return 0;
}
```

Listing 13: Before tail-call optimization

is transformed into

```
int sum (int n)
{
  int acc = 0;

  while (n > 0)
    acc += n--;

  return acc;
}
```

Listing 14: After tail-call optimization

Example function in `gcc/tree-tailcall.c:1086`:

```
1086 static unsigned int
1087 tree_optimize_tail_calls_1 (bool opt_tailcalls)
1088 {
1089   edge e;
1090   bool phis_constructed = false;
1091   struct tailcall *tailcalls = NULL, *act, *next;
1092   bool changed = false;
1093   basic_block first = single_succ (ENTRY_BLOCK_PTR_FOR_FN (cfun));
1094   tree param;
1095   gimple *stmt;
1096   edge_iterator ei;
1097
1098   if (!suitable_for_tail_opt_p ())
1099     return 0;
1100   if (opt_tailcalls)
1101     opt_tailcalls = suitable_for_tail_call_opt_p ();
1102
1103   FOR_EACH_EDGE (e, ei, EXIT_BLOCK_PTR_FOR_FN (cfun)->preds)
1104     {
1105       /* Only traverse the normal exits, i.e. those that end with return
1106          statement.  */
1107       stmt = last_stmt (e->src);
1108
1109       if (stmt
1110           && gimple_code (stmt) == GIMPLE_RETURN)
1111         find_tail_calls (e->src, &tailcalls);
1112     }
```

Listing 15: Entry point to tail-call optimization (`gcc/tree-tailcall.c`)

6.2 Conditional constant propogation & Folding built-in functions (`tree-ssa-ccp.c`)

Conditional constant propagation (CCP) propogates statements of the pattern `VAR = CONSTANT` to the rest of the program. The algorithm is based on the assumption that the code is SSA. The common operation of propogation can be seen in this code snippet in `gcc/tree-ssa-propagate.c:1444`:

```
1444 static void
1445 replace_exp_1 (use_operand_p op_p, tree val,
1446                    bool for_propagation ATTRIBUTE_UNUSED)
1447 {
1448   if (flag_checking)
1449     {
1450       tree op = USE_FROM_PTR (op_p);
1451       gcc_assert (!(for_propagation
1452                   && TREE_CODE (op) == SSA_NAME
1453                   && TREE_CODE (val) == SSA_NAME
1454                   && !may_propagate_copy (op, val)));
1455     }
1456 
1457   if (TREE_CODE (val) == SSA_NAME)
1458     SET_USE (op_p, val);
1459   else
1460     SET_USE (op_p, unshare_expr (val));
1461 }
```

Listing 16: Entry point to CCP

After constants are propagated, built-in functions are also simplified if the arguments are constants. In `gcc/tree-ssa-ccp.c:3255`

```
unsigned int
pass_fold_builtins::execute (function *fun)
{
  bool cfg_changed = false;
  basic_block bb;
  unsigned int todoflags = 0;

  FOR_EACH_BB_FN (bb, fun)
    {
      gimple_stmt_iterator i;
      for (i = gsi_start_bb (bb); !gsi_end_p (i); )
        {
          gimple *stmt, *old_stmt;
          tree callee;
          enum built_in_function fcode;

          stmt = gsi_stmt (i);
      // --snip--
    }
    // --snip--
    }
// --snip--
}
```

Listing 17: Entry point to built-in function folding

6.3 Delayed branch scheduling (in `reorg.c`)

Recall in CS2100, we know that branch instructions incur Branch Penalty, which are extra cycles that are needed compared to a, say, ADD instruction. These extra cycles are made use of by rearranging non branch-critical instructions to execute during the Branch Penalty to offset the cost.

From `gcc/reorg.c:3738`:

```
/* Try to find insns to place in delay slots.  */

static void
dbr_schedule (rtx_insn *first)
{
  rtx_insn *insn, *next, *epilogue_insn = 0;
  int i;
  bool need_return_insns;

  /* If the current function has no insns other than the prologue and
     epilogue, then do not try to fill any delay slots.  */
  if (n_basic_blocks_for_fn (cfun) == NUM_FIXED_BLOCKS)
    return;
```

```

  /* Find the highest INSN_UID and allocate and initialize our map from
     INSN_UID's to position in code.  */
  for (max_uid = 0, insn = first; insn; insn = NEXT_INSN (insn))
    {
      if (INSN_UID (insn) > max_uid)
        max_uid = INSN_UID (insn);
      if (NOTE_P (insn)
          && NOTE_KIND (insn) == NOTE_INSN_EPILOGUE_BEG)
        epilogue_insn = insn;
    }
  // CODE OMITTED FOR BREVITY
}
```

References

[1] GCC. Parsing pass (gnu compiler collection (gcc) internals), 2021. https://gcc.gnu.org/onlinedocs/gccint/Parsing-pass.html#Parsing-pass, last accessed November 1, 2021.

[2] GCC. Passes (gnu compiler collection (gcc) internals), 2021. https://gcc.gnu.org/onlinedocs/gccint/Passes.html#Passes, last accessed November 1, 2021.

[3] GCC. Rtl passes(gnu compiler collection (gcc) internals), 2021. https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html#RTL-passes, last accessed November 1, 2021.

[4] GCC. Tree ssa passes (gnu compiler collection (gcc) internals), 2021. https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA-passes.html#Tree-SSA-passes, last accessed November 1, 2021.

[5] Diego Novillo. From source to binary: The inner workings of gcc, Dec 2004.