Workshop on Essential Abstractions in GCC

Introduction to Gimple IR

GCC Resource Center

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

July 2009

Outline

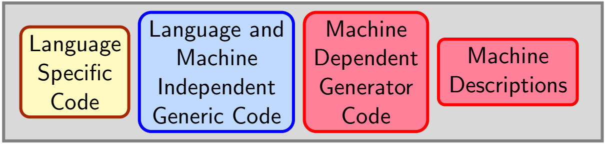
- Introduction to Gimple IR
- Adding a pass to GCC
- Working with the Gimple API

Part 1

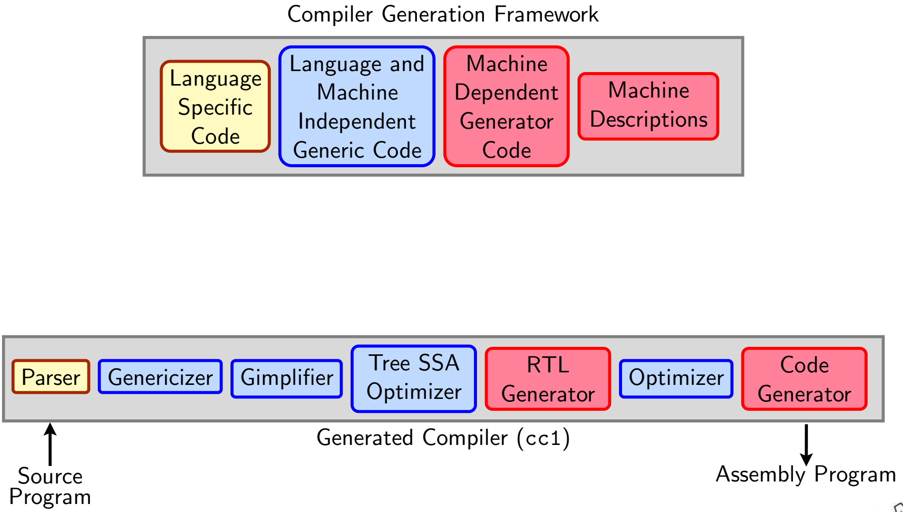
Introduction to GIMPLE

Recall GCC CGF

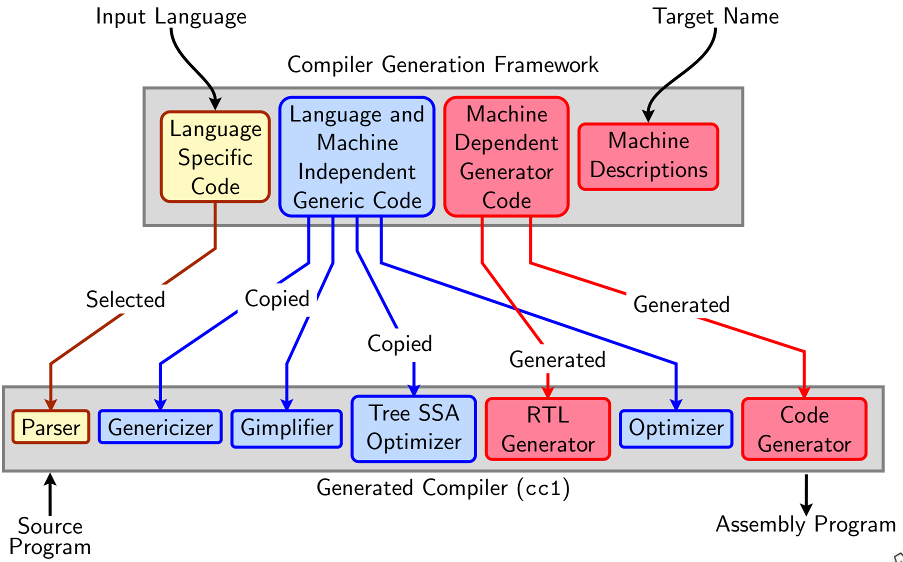
Compiler Generation Framework



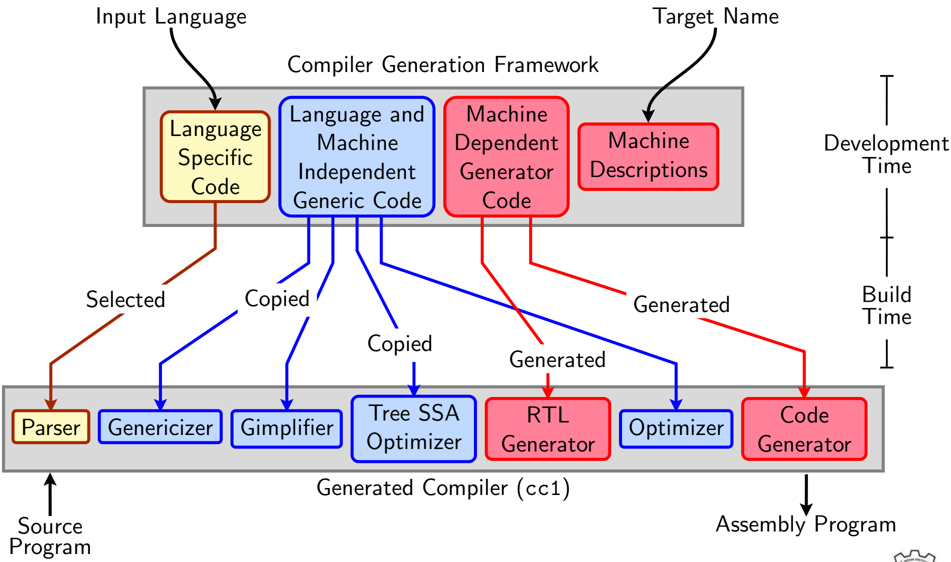
Recall GCC CGF



Recall GCC CGF



Recall GCC CGF



Basics of GIMPLE

- GIMPLE is a language-independent IR for GCC.
- It is based on *tree* data structure.
- GIMPLE is *simple.*

Motivation behind GIMPLE

- Previously, the only common IR was RTL (Register Transfer Language)
- Drawbacks of RTL for performing high-level optimizations :
  - RTL is a low-level IR, works well for optimizations close to machine (e.g., register allocation)
  - Some high level information is difficult to extract from RTL (e.g. array references, data types etc.)
  - Optimizations involving such higher level information are difficult to do using RTL.
  - Introduces stack too soon, even if later optimizations dont demand it.

Notice

Inlining at tree level could partially address the the last limitation of RTL.

Why not ASTs for optimization ?

- ASTs contain detailed function information but are not suitable for optimization because
  - Lack of a common representation
    - No single AST shared by all front-ends
    - So each language would have to have a different implementation of the same optimizations
    - Difficult to maintain and upgrade so many optimization frameworks
  - Structural Complexity
    - Lots of complexity due to the syntactic constructs of each language

Need for a new IR

- In the past, compiler would only build up trees for a single statement,and then lower them to RTL before moving on to the next statement.
- For higher level optimizations, entire function needs to be represented in trees in a language-independent way.
- Result of this effort - GENERIC and GIMPLE

What is GENERIC ?

- Language independent IR for a complete function in the form of trees
- Obtained by removing language specific constructs from ASTs
- All tree codes defined in `$(SOURCE)/gcc/tree.def`

- Each language frontend may still have its own AST.
- Once parsing is complete they must emit GENERIC

What is GIMPLE ?

- GIMPLE is influenced by SIMPLE IR of McCat compiler
- But GIMPLE is not same as SIMPLE (Gimple supports GOTO)
- It is a simplified subset of GENERIC
  - 3 address representation
  - Control flow lowering
  - Cleanups and simplification, restricted grammar
- Benefit : Optimizations become easier

GIMPLE Phase sequence in cc1 and GCC

Converting GENERIC to GIMPLE

```
c_genericize()                                   c-gimplify.c
   gimplify_function_tree()                        gimplify.c
      gimplify_body()                              gimplify.c
         gimplify_stmt()                           gimplify.c
            gimplify_expr()                        gimplify.c
lang_hooks.callgraph.expand_function()
 tree_rest_of_compilation()                  tree-optimize.c
  tree_register_cfg_hooks()                        cfghooks.c
   execute_pass_list()                               passes.c
 /* TO: Gimple Optimisations passes */
                ...
          NEXT_PASS(pass_lower_cf)
```

GIMPLE Goals

The Goals of GIMPLE are

- Lower control flow
  Program = sequenced statements + unrestricted jump
- Simplify expressions
  Typically: two operand assignments!
- Simplify scope
  move local scope to block begin, including temporaries

Notice

Lowered control flow → nearer to register machines + Easier SSA!

High GIMPLE

- GIMPLE that is not fully lowered.
- Consists of Intermediate Language before the pass *pass_lower_cf*.
- Contains some container statements like lexical scopes and nested expressions.
- High GIMPLE Instruction Set : GIMPLE_BIND, GIMPLE_CALL, GIMPLE_CATCH, GIMPLE_GOTO, GIMPLE_EH_FILTER, GIMPLE_RETURN, GIMPLE_SWITCH, GIMPLE_TRY, GIMPLE_ASSIGN

Low GIMPLE

- Gimple that is fully lowered after the pass *pass_lower_cf*.
- Exposes all of the implicit jumps for control and exception expressions.
- Low GIMPLE Instruction Set : GIMPLE_CALL, GIMPLE_GOTO, GIMPLE_RETURN, GIMPLE_SWITCH, GIMPLE_ASSIGN
- Lowered Instruction Set : GIMPLE_BIND, GIMPLE_CATCH, GIMPLE_EH_FILTER, GIMPLE_TRY

Some GIMPLE Node types

Binary Operator	`MAX_EXPR`
Comparison	`EQ_EXPR`, `LT_EXPR`
Constants	`INTEGER_CST`, `STRING_CST`
Declaration	`FUNCTION_DECL`, `LABEL_DECL` , `VAR_DECL`
Expression	`PLUS_EXPR`, `ADDR_EXPR`
Reference	`COMPONENT_REF`, `ARRAY_RANGE_REF`
Statement	`GIMPLE_MODIFY_STMT`, `RETURN_EXPR`, `COND_EXPR`, `INIT_EXPR`
Type	`BOOLEAN_TYPE`, `INTEGER_TYPE`
Unary	`ABS_EXPR`, `NEGATE_EXPR`

Tip :

All tree nodes ($\sim$ 152) in GCC are listed in: `$(SOURCE)/gcc/tree.def`.

Journey through GIMPLE

Generic Code (gimple.c)

```
int main()
{
    int a;
    if (a)
    {
        int b;
        b = 2 + a + b;
    }
    return 0;
}
```

Journey through GIMPLE

High GIMPLE (`gimple.c.004t.gimple`)

```
main ()
{
  int D.1195;
  int D.1196;
  int a;

  if (a != 0)
    {
      {
        int b;

        D.1195 = a + 2;
        b = D.1195 + b;
      }
    }
  else
    {

    }
  D.1196 = 0;
  return D.1196;
}
```

Journey through GIMPLE

Low GIMPLE (`gimple.c.013t.cfg`) : Lexical scopes removed

```
main ()
{
  int b;
  int a;
  int D.1196;
  int D.1195;

  # BLOCK 2
  # PRED: ENTRY (fallthru)
  if (a != 0)
    goto <bb 3>;
  else
    goto <bb 4>;
  # SUCC: 3 (true) 4 (false)

  # BLOCK 3
  # PRED: 2 (true)
  D.1195 = a + 2;
  b = D.1195 + b;
  # SUCC: 4 (fallthru)

  # BLOCK 4
  # PRED: 2 (false) 3 (fallthru)
  D.1196 = 0;
  # SUCC: 5 (fallthru)

  # BLOCK 5
  # PRED: 4 (fallthru)
  return D.1196;
  # SUCC: EXIT
}
```

Important Dump Files

- Compile using `./gcc -fdump-tree-all <file-name >.c`
- Examine `<file-name >.c.013t.cfg`

Resolving doubts by inspecting GIMPLE

Inspect GIMPLE when in doubt

```
int main(void)
{
  int x=2,y=3;
  x= y++ + ++x + ++y ;
  printf("\nx = %d", x);
  printf("\ny = %d", y);
  return 0;
}
```

Resolving doubts by inspecting GIMPLE

Inspect GIMPLE when in doubt

```
int main(void)
{
  int x=2,y=3;
  x= y++ + ++x + ++y ;
  printf("\nx = %d", x);
  printf("\ny = %d", y);
  return 0;
}
```

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
printf (&"\nx = %d"[0], x);
printf (&"\y = %d"[0], y);
```

Resolving doubts by inspecting GIMPLE

Inspect GIMPLE when in doubt

```
int main(void)
{
  int x=2,y=3;
  x= y++ + ++x + ++y ;
  printf("\nx = %d", x);
  printf("\ny = %d", y);
  return 0;
}
```

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
printf (&"\nx = %d"[0], x);
printf (&"\y = %d"[0], y);
```

x = 10 , y =5

Part 2

Adding a Pass to GCC

Adding a Pass on Gimple IR

- Step 0. Write function `gccwk09_main()` in file `gccwk09.c`.
- Step 1. Create the following data structure in file `gccwk09.c`.

```
struct tree_opt_pass pass_gccwk09 =
{ "gccwk09",  /* name */
  NULL,       /* gate, for conditional entry to this pass */
  gccwk09_main, /* execute, main entry point */
  NULL,       /* sub-passes, depending on the gate predicate */
  NULL,       /* next sub-passes, independ of the gate predicate */
  0,          /* static_pass_number , used for dump file name*/
  0,          /* tv_id */
  0,          /* properties_required, indicated by bit position */
  0,          /* properties_provided , indicated by bit position*/
  0,          /* properties_destroyed , indicated by bit position*/
  0,          /* todo_flags_start */
  0,          /* todo_flags_finish */
  0           /* letter for RTL dump */
};
```

Adding a Pass on Gimple IR

- Step 2. Add the following line to `tree-pass.h`

```
extern struct tree_opt_pass pass_gccwk09;
```

Adding a Pass on Gimple IR

- Step 2. Add the following line to `tree-pass.h`
  `extern struct tree_opt_pass pass_gccwk09;`
- Step 3. Include the following call at an appropriate place in the function `init_optimization_passes()` in the file `passes.c`
  `NEXT_PASS (pass_gccwk09);`

Adding a Pass on Gimple IR

- Step 2. Add the following line to `tree-pass.h`
  `extern struct tree_opt_pass pass_gccwk09;`
- Step 3. Include the following call at an appropriate place in the function `init_optimization_passes()` in the file `passes.c`
  `NEXT_PASS (pass_gccwk09);`
- Step 4. Add the file name in the Makefile
  - Either in `$SOURCE/gcc/Makefile.in`
    Reconfigure and remake
  - Or in `$BUILD/gcc/Makefile`
    Remake

Adding a Pass on Gimple IR

- Step 2. Add the following line to `tree-pass.h`
  `extern struct tree_opt_pass pass_gccwk09;`
- Step 3. Include the following call at an appropriate place in the function `init_optimization_passes()` in the file `passes.c`
  `NEXT_PASS (pass_gccwk09);`
- Step 4. Add the file name in the Makefile
  - Either in `$SOURCE/gcc/Makefile.in`
    Reconfigure and remake
  - Or in `$BUILD/gcc/Makefile`
    Remake
- Step 5. Build the compiler

Adding a Pass on Gimple IR

- Step 2. Add the following line to `tree-pass.h`
  `extern struct tree_opt_pass pass_gccwk09;`
- Step 3. Include the following call at an appropriate place in the function `init_optimization_passes()` in the file `passes.c`
  `NEXT_PASS (pass_gccwk09);`
- Step 4. Add the file name in the Makefile
  - Either in `$SOURCE/gcc/Makefile.in`
    Reconfigure and remake
  - Or in `$BUILD/gcc/Makefile`
    Remake
- Step 5. Build the compiler
- Step 6. Debug using gdb if need arises

Part 3

Working with the GIMPLE API

GIMPLE Statements

- GIMPLE Statements are nodes of type tree
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through iterators

GIMPLE Statements

- GIMPLE Statements are nodes of type tree
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through iterators

```
block_statement_iterator bsi;
basic_block bb;
```

GIMPLE Statements

- GIMPLE Statements are nodes of type tree
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through iterators

```
block_statement_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
```

Basic Block Iterator

GIMPLE Statements

- GIMPLE Statements are nodes of type tree
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through iterators

```
block_statement_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
   for ( bsi =bsi_start(bb); !bsi_end_p(bsi); bsi_next(&bsi))
```

Block Statement Iterator

GIMPLE Statements

- GIMPLE Statements are nodes of type tree
- Every basic block contains a doubly linked-list of statements
- Processing of statements can be done through iterators

```
block_statement_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
   for ( bsi =bsi_start(bb); !bsi_end_p(bsi); bsi_next(&bsi))
      print_generic_stmt (stderr, bsi_stmt(bsi), 0);
```

A simple application

Counting the number of assignment statements in GIMPLE

```
#include <stdio.h>
int m,q,p;
int main(void)
{
  int x,y,z,w;
  x = y + 5;
  z = x * m;
  p = m + q + w ;
  return 0;
}
```

```
x = y + 5;
m.0 = m;
z = x * m.0;
m.1 = m;
q.2 = q;
D.1580 = m.1 + q.2;
p.3 = D.1580 + w;
p = p.3;
D.1582 = 0;
return  D.1582;
```

The statements in blue are the assignments corresponding to the source.

A simple application

Counting the number of assignment statements in GIMPLE

```
struct tree_opt_pass pass_gccwk09 =
{
    "gccwk09",
     NULL,
     gccwk09_main,
     NULL,
     NULL,
     0,
     0,
     0,
     0,
     0,
     0,
     0,
     0
};
```

A simple application

Counting the number of assignment statements in GIMPLE

```
static unsigned int gccwk09_main(void)
{  basic_block bb;
   block_stmt_iterator si;

   initialize_stats();

   FOR_EACH_BB (bb)
   {
       for (si=bsi_start(bb); !bsi_end_p(si); bsi_next(&si))
           {
             tree stmt = bsi_stmt(si);
             process_statement(stmt);
           }
   }
   return 0;
}
```

A simple application

Counting the number of assignment statements in GIMPLE

```
void process_statement(tree stmt)
{  tree lval,rval;
    switch (TREE_CODE(stmt))
    {    case GIMPLE_MODIFY_STMT:
              lval=GIMPLE_STMT_OPERAND(stmt,0);
              rval=GIMPLE_STMT_OPERAND(stmt,1);
              if(TREE_CODE(lval) == VAR_DECL)
              {   if(!DECL_ARTIFICIAL(lval))
                   {  print_generic_stmt(stderr,stmt,0);
                      numassigns++;
                   }
                  totalassigns++;
              }
              break;
        default :
              break;
    }
}
```

A simple application

Counting the number of assignment statements in GIMPLE

- Add the following in `$(SOURCE)/gcc/common.opt` :
- `fpass_gccwk09`
- Common Report Var (`flag_pass_gccwk09`)
- Enable pass named `pass_gccwk09`

Compile using `./gcc -fdump-tree-all -fpass_gccwk09 test.c`

Assignment and Reference

API Reference

- http://gcc.gnu.org/onlinedocs/gccint.pdf Pg- 233-235
- Refere the same document for some detailed documentation

Assignments (by traversing the GIMPLE IR)

- Count the number of copy statements in a program
- Count the number of variables declared "const" in the program
- Count the number of occurances of arithmatic operators in the program
- Count the number of references to global variables in the program