Workshop on Essential Abstractions in GCC

GCC Control Flow and Plugins

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

1 July 2011

Outline

• Motivation
• Plugins in GCC
• GCC Control Flow
• Link time optimization in GCC
• Conclusions
Part 1

Motivation

Module Binding Mechanisms

• The need for adding, removing, and maintaining modules relatively independently

• The mechanism for supporting this is called by many names:
  ▶ Plugin, hook, callback, …
  ▶ Sometimes it remains unnamed (eg. compilers in gcc driver)

• It may involve
  ▶ Minor changes in the main source
    Requires static linking
    We call this a static plugin
  ▶ No changes in the main source
    Requires dynamic linking
    We call this a dynamic plugin
Plugin as a Module Binding Mechanisms

- We view plugin at a more general level than the conventional view. Adjectives “static” and “dynamic” create a good contrast.

- Most often a plugin in a C-based software is a data structure containing function pointers and other related information.

Static Vs. Dynamic Plugins

- Static plugin requires static linking:
  - Changes required in gcc/Makefile.in, some header and source files
  - At least cc1 may have to be rebuilt. All files that include the changed headers will have to be recompiled.

- Dynamic plugin uses dynamic linking:
  - Supported on platforms that support -ldl -rdynamic
  - Loaded using dlopen and invoked at pre-determined locations in the compilation process
  - Command line option
    - -fplugin=/path/to/name.so
  - Arguments required can be supplied as name-value pairs.
### GCC's Solution

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Implementation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translator in gcc</td>
<td>Array of C structures</td>
<td>Development time</td>
</tr>
<tr>
<td>Front end in cc1</td>
<td>C structure</td>
<td>Development time</td>
</tr>
<tr>
<td>Passes in cc1</td>
<td>Linked list of C structures</td>
<td>Development time</td>
</tr>
<tr>
<td>Back end in cc1</td>
<td>Arrays of structures</td>
<td>Build time</td>
</tr>
</tbody>
</table>
Plugin Data Structure in the GCC Driver

```c
struct compiler {
    const char *suffix;  /* Use this compiler for input files whose names end in this suffix. */
    const char *spec;    /* To use this compiler, run this spec. */
    const char *cpp_spec; /* If non-NULL, substitute this spec for '%C', rather than the usual cpp_spec. */
    const int combinable; /* If nonzero, compiler can deal with multiple source files at once (IMA). */
    const int needs_preprocessing; /* If nonzero, source files need to be run through a preprocessor. */
};
```

Notes

- `@`: Aliased entry
- `#`: Default specs not available

Default Specs in the Plugin Data Structure in `gcc.c`

All entries of Objective C/C++ and some entries of Fortran removed.

```c
static const struct compiler default_compilers[] = {
    {".cc", "#C++", 0, 0, 0},  {".cxx", "#C++", 0, 0, 0},
    {".cpp", "#C++", 0, 0, 0},  {".cp", "#C++", 0, 0, 0},
    {".c++", "#C++", 0, 0, 0},  {".C", "#C++", 0, 0, 0},
    {".CPP", "#C++", 0, 0, 0},  {".ii", "#C++", 0, 0, 0},
    {".ads", "#Ada", 0, 0, 0},  {".adb", "#Ada", 0, 0, 0},
    {".f", "#Fortran", 0, 0, 0}, {".F", "#Fortran", 0, 0, 0},
    {".for", "#Fortran", 0, 0, 0}, {".FUR", "#Fortran", 0, 0, 0},
    {".f90", "#Fortran", 0, 0, 0}, {".F90", "#Fortran", 0, 0, 0},
    {".p", "#Pascal", 0, 0, 0},  {".pas", "#Pascal", 0, 0, 0},
    {".java", "#Java", 0, 0, 0},  {".class", "#Java", 0, 0, 0},
    {".c", "@C", 0, 1, 1},
    {".h", "@c-header", 0, 0, 0}, {".i", "@cpp-output", 0, 1, 0},
    {".s", "@assembler", 0, 1, 0}
};
```
Complete Entry for C in gcc.c

同胞 capability in GCC

Populated Plugin Data Structure for C++:
 gcc/cp/lang-specs.h

Notes
Populated Plugin Data Structure for C++:

gcc/cp/lang-specs.h

{"@c++-header",
  "%{E|M|MM:cc1plus -E (%(cpp_options) %2 (%(cpp_debug_options))\n  %{E:%{M:%{MM: %{save-t twins\no-integrated-cpp:cc1plus -E\\%{cpp_options} %2 -o %{save-twins:%b.ii} %{!save-twin\nts:%g.ii} \n  cc1plus %{save-twins:no-integrated-cpp:-fpreprocessed %{save-twins:%b.ii} %{!save-t\ntwins:%!no-integrated-cpp:%(cpp_unique_options)}\\%{c\n1_options} %2\\%{!fsyntax-only:%{!fdump-ada-spec*:-%t %g.s %{!o*:--output-pch=%i.gch}\n  %W{o*:--output-pch=%*}}%V}}}}",
  CPLUSPLUS_CPP_SPEC, 0, 0},

{".ii", "@c++-cpp-output", 0, 0, 0},

{"@c++-cpp-output",
  %{!M:%{!MM:%{!E:
  cc1plus -fpreprocessed %i %(cc1_options) %2%{!fsyntax-only:%(invoke_as)}}}}, 0, 0, 0},

{"$c++-cpp-output",
  "%{!M:%{!MM:%{!E:
  cc1plus --fpreprocessed %i %!{cc1_options} %2\%
  %{!fsyntax-only:%{!fdump-ada-spec*:-%t %g.s %{!o*:--output-pch=%i.gch}\n  %W{o*:--output-pch=%*}}%V}}}}", 0, 0, 0},
/* LTO contributions to the "compilers" array in gcc.c. */

{"@lto", "lto1 %{ccl_options} %i %{!fsyntax-only:%{invoke_as}}", /*cpp_spec=*/NULL, /*combinable=*/1, /*needs_preprocessing=*/0},

Notes

What about the Files to be Processed by the Linker?

- Linking is the last step
- Every file is passed on to linker unless it is suppressed
- If a translator is not found, input file is assumed to be a file for linker
Plugin Structure in cc1

- toplevel main
- frontend
- pass manager
- pass 1
- pass 2
- . . .
- pass expand
- . . .
- pass n

Notes

- double arrow represents control flow whereas single arrow represents pointer or index
- code for pass 1
- code for pass 2
- expander code
- optab_table
- code for pass n
Plugin Structure in \textit{cc1}

- Top level main
- Front end
- Pass manager
  - Pass 1 → Code for pass 1
  - Pass 2 → Code for pass 2
  - Pass expand → Expander code
    - \texttt{optab\_table}
  - Pass n → Code for pass n

- Language hooks
  - Code for language 1
  - Code for language 2
  - Code for language n

Notes

- Insns data
  - Generated code for machine 1
  - MD 1
  - MD 2
  - MD n
Important fields of struct `lang_hooks` instantiated for C

```c
#define LANG_HOOKS_FINISH c_common_finish
#define LANG_HOOKS_EXPAND_EXPR c_expand_expr
#define LANG_HOOKS_PARSE_FILE c_common_parse_file
#define LANG_HOOKS_WRITEGLOBALS c_write_global_declarations
```

```c
struct opt_pass {
    enum opt_pass_type type;
    const char *name;
    bool (*gate) (void);
    unsigned int (*execute) (void);
    struct opt_pass *sub;
    struct opt_pass *next;
    int static_pass_number;
    timevar_id_t tv_id;
    unsigned int properties_required;
    unsigned int properties_provided;
    unsigned int properties_destroyed;
    unsigned int todo_flags_start;
    unsigned int todo_flags_finish;
};
```

```c
struct gimple_opt_pass {
    struct opt_pass pass;
};
```

```c
struct rtl_opt_pass {
    struct opt_pass pass;
};
```
### Plugins for Interprocedural Passes

```c
struct ipa_opt_pass_d{
    struct opt_pass pass;
    void (*generate_summary) (void);
    void (*write_summary) (struct cgraph_node_set_def *);
    void (*function_read_summary) (struct cgraph_node *);
    void (*function_read_summary) (struct cgraph_node *);
    void (*stmt_fixup) (struct cgraph_node *, gimple *);
    unsigned int function_transform_todo_flags_start;
    unsigned int (*function_transform) (struct cgraph_node *);
    void (*variable_transform) (struct varpool_node *);
};

struct simple_ipa_opt_pass{
    struct opt_pass pass;
};
```

### Predefined Pass Lists

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_lowering_passes</td>
<td>Lowering</td>
</tr>
<tr>
<td>all_small_ipa_passes</td>
<td>Early optimization passes. Invokes intraprocedural passes over the call graph.</td>
</tr>
<tr>
<td>all_regular_ipa_passes</td>
<td></td>
</tr>
<tr>
<td>all_lto_gen_passes</td>
<td></td>
</tr>
<tr>
<td>all_passes</td>
<td>Intraprocedural passes on GIMPLE and RTL</td>
</tr>
</tbody>
</table>
Registering a Pass as a Static Plugin

1. Write the driver function in your file
2. Declare your pass in file tree-pass.h:
   ```c
   extern struct gimple_opt_pass your_pass_name;
   ```
3. Add your pass to the appropriate pass list in
   ```c
   init_optimization_passes() using the macro NEXT_PASS
   ```
4. Add your file details to $SOURCE/gcc/Makefile.in
5. Configure and build gcc
   (For simplicity, you can make cc1 only)
6. Debug cc1 using ddd/gdb if need arises
   (For debugging cc1 from within gcc, see:
Dynamic Plugins

- Supported on platforms that support `-ldl -rdynamic`
- Loaded using `dlopen` and invoked at pre-determined locations in the compilation process
- Command line option
  `-fplugin=/path/to/name.so`
  Arguments required can be supplied as name-value pairs

Specifying an Example Pass

```c
struct simple_ipa_opt_pass pass_plugin = {
    SIMPLE_IPA_PASS,  /* name */
    "dynamic_plug",  /* gate */
    execute_pass_plugin,  /* execute */
    NULL,  /* sub */
    NULL,  /* next */
    0,  /* static pass number */
    TV_INTEGRATION,  /* tv_id */
    0,  /* properties required */
    0,  /* properties provided */
    0,  /* properties destroyed */
    0,  /* todo_flags start */
    0  /* todo_flags end */
};
```
Registering Our Pass as a Dynamic Plugin

```c
struct register_pass_info pass_info = {
  &(pass_plugin.pass), /* Address of new pass, here, the 
  struct opt_pass field of 
  simple_ipa_opt_pass defined above */
  "pta", /* Name of the reference pass (string 
  in the structure specification) for 
  hooking up the new pass. */ 
  0, /* Insert the pass at the specified 
  instance number of the reference 
  pass. Do it for every instance if 
  it is 0. */ 
  PASS_POS_INSERT_AFTER /* how to insert the new pass: 
  before, after, or replace. Here we 
  are inserting our pass the pass 
  named pta */ 
};
```

Registering Callback for Our Pass for a Dynamic Plugins

```c
int plugin_init(struct plugin_name_args *plugin_info,
struct plugin_gcc_version *version)
{ /* Plugins are activated using this callback */

  register_callback ( 
    plugin_info->base_name, /* char *name: Plugin name, 
    could be any name. 
    plugin_info->base_name gives this filename */
    PLUGIN_PASS_MANAGER_SETUP, /* int event: The event code. 
    Here, setting up a new 
    pass */
    NULL, /* The function that handles 
    the event */
    &pass_info); /* plugin specific data */

  return 0;
} 
```
Makefile for Creating and Using a Dynamic Plugin

CC = $(INSTALL_D)/bin/gcc
PLUGIN_SOURCES = new-pass.c
PLUGIN_OBJECTS = $(patsubst %.c,%.o,$(PLUGIN_SOURCES ))
GCCPLUGINS_DIR = $(shell $(CC) -print-file-name=plugin)
CFLAGS+= -fPIC -O2
INCLUDE = -Iplugin/include

%.o : %.c
$(CC) $(CFLAGS) $(INCLUDE) -c $<

new-pass.so: $(PLUGIN_OBJECTS)
    $(CC) $(CFLAGS) $(INCLUDE) -shared $^ -o $@

test_plugin: test.c
    $(CC) -fplugin=./new-pass.so $^ -o $@ -fdump-tree-all

Part 4

Flow of Control in the Generated Compiler
Walking the Maze of a Large Code Base

- If you use conventional editors such as vi or emacs
  - Use cscope
    
    ```
    cd $SOURCE
cscope -R
    ```
  - Use ctags
    
    ```
    cd $SOURCE
ctags -R
    ```
  - Make sure you use exeburant-ctags
- Or use IDE such as eclipse

Notes

```c
main  /* In file gcc.c */
  validate_all_switches
  lookup_compiler
  do_spec
    do_spec_2  /* Get the name of the compiler */
    do_spec_1
  execute
  pex_init
  pex_run
    pex_run_in_environment
    obj->funcs->exec_child
```
main  /* In file gcc.c */
  validate_all_switches
  lookup_compiler
  do_spec
    do_spec_2
    do_spec_1  /*
  execute
    pex_init
    pex_run
    pex_run_in
    obj->fu

Observations
  • All compilers are invoked by this driver
  • Assembler is also invoked by this driver
  • Linker is invoked in the end by default

main  /* In file toplev.c */
toplev_main
  decode_options
  do_compile
    compile_file
    lang_hooks.parse_file => c_common_parse_file
    lang_hooks.decls.final_write Globals =>
    c_write_global_declarations
    targetm.asm_out.file_end
  finalize
**Top Level Control Flow**

```c
main
toplev_main  /* In file toplev.c */
decode_options
do_compile
  compile_file
    lang_hooks.p
  lang_hooks.d
  targetm.asm_out
  finalize
```

**Observations**
- The entire compilation is driven by functions specified in language hooks
- Not a good design!

**Control Flow: Parsing for C**

```c
lang_hooks.parse_file => c_common_parse_file
c_parse_file
  c_parser_translation_unit
  c_parser_external_declaration
  c_parser_declaration_or_fndef
  c_parser_decls
  finish_function  /* finish parsing */
c generalize
  cgraph_finalize_function
  /* finalize AST of a function */
```
lang_hooks.parse_file => c_common_parse_file
  c_parse_file
    c_parser_translation_unit
      c_parser_enter
        c_parse
          c_parse
          finish_function
  c_genericize
  cgraph_finalize_function

Observations

• GCC has moved to a recursive descent parser from version 4.1.0
• Earlier parser was generated using Bison specification

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Notes
lang_hooks.decls.final_write_globals => c_write_global_declarations
cgraph_finalize_compilation_unit
 /* Lowering passes are language independent */
 cgraph_analyze
 cgraph_anal
 gimpli
 gim

cgraph_lower_function
 /* Intraprocedural */
 tree_lowering_passes
 execute_pass_list (all_lowering_passes)

Observations

• Lowering passes are language independent
• Yet they are being called from a function in language hooks
• Not a good design!

<table>
<thead>
<tr>
<th>Order</th>
<th>Task</th>
<th>IR</th>
<th>Level</th>
<th>Pass data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lowering</td>
<td>GIMPLE</td>
<td>Intra</td>
<td>gimple_opt_pass</td>
</tr>
<tr>
<td>2</td>
<td>Optimizations</td>
<td>GIMPLE</td>
<td>Inter</td>
<td>ipa_opt_pass</td>
</tr>
<tr>
<td>3</td>
<td>Optimizations</td>
<td>GIMPLE</td>
<td>Intra</td>
<td>gimple_opt_pass</td>
</tr>
<tr>
<td>4</td>
<td>RTL Generation</td>
<td>GIMPLE</td>
<td>Intra</td>
<td>rtl_opt_pass</td>
</tr>
<tr>
<td>5</td>
<td>Optimization</td>
<td>RTL</td>
<td>Intra</td>
<td>rtl_opt_pass</td>
</tr>
</tbody>
</table>
**Observations**

- Optimization and code generation passes are language independent
- Yet they are being called from a function in language hooks
- Not a good design!
## Execution Order in Intraprocedural Passes

<table>
<thead>
<tr>
<th>Function 1</th>
<th>Function 2</th>
<th>Function 3</th>
<th>Function 4</th>
<th>Function 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Execution Order in Interprocedural Passes

Function 1  Function 2  Function 3  Function 4  Function 5

Pass 1  O  O  O  O  O
Pass 2  O  O  O  O  O
Pass 3  O  O  O  O  O
Pass 4  O  O  O  O  O
Pass 5  O  O  O  O  O

Notes
gimple_expand_cfg
   expand_gimple_basic_block(bb)
   expand_gimple_cond(stmt)
   expand_gimple_stmt(stmt)
   expand_gimple_stmt_1(stmt)
   expand_expr_real_2
   expand_expr  /* Operands */
   expand_expr_real
   optab_for_tree_code
   expand_binop /* Now we have rtx for operands */
   expand_binop_directly
   /* The plugin for a machine */
   code=optab_handler(binoptab,mode);
   GEN_FCN
   emit_insn
Link Time Optimization

- Default `cgraph` creation is restricted to a translation unit (i.e., a single file)
- Interprocedural analysis and optimization across files is not possible by default
- All files (or their equivalents) are available at link time (assuming static linking)
- LTO in GCC is basically interprocedural optimizations of functions across different files

---

Notes

- LTO framework supported in GCC-4.6.0
- Use `-f1to` option during compilation
- Generates conventional `.o` files and inserts GIMPLE level information in them
- Complete transition is performed in this phase
- During linking all object modules are put together and `lto1` is invoked
- It re-executes optimization passes from the function `cgraph_optimize`

```
Basic Idea: Provide a larger call graph to regular ipa passes
```
main ()
{
    printf ("hello, world\n");
}

main ()
{
    printf ("hello, world\n");
}
Assembly Output with LTO Information (3)

```
.asci "/342\312\254\204\323\307\035\207[w\230qN\204\032gB2\335p"
.asci "\025\304\033\365\241\341\033\314\255a\225\376\237\326"
.asci ",&\121\253.Y021q/ /320\310\0166\230qN\204\032gB2\335p"
.asci ",@progbits
.asci "/017\f'005\227D\267\340\333\365\241\341\033\314\255a\225\376\237\326"
.asci ",@progbits
```

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
andl $-16, %esp
subl $16, %esp
movl $.LC0, (%esp)
call puts
.leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size main, -.main
.comm __gnu_lto_v1,1,1
.ident "GCC: (GNU) 4.6.0"
.section .note.GNU-stack,"",@progbits
```c
lto_main
lto_process_name
lto_init_reader
read_cgraph_and_symbols
  if (flag_wpa)
  {
    do_whole_program_analysis
    materialize_cgraph
    execute_ipa_pass_list (all_regular_ipa_passes)
  }
  else
  {
    materialize_cgraph
    cgraph_optimize
  }
```

```c
cc1 Control Flow: A Recap

toplev_main  /* In file toplev.c */
compile_file
  lang_hooks.parse_file=>c_common_parse_file
  lang_hooks.decls.final_write_globals=>c_write_global_declarations
  cgraph_finalize_compilation_unit
  cgraph_analyze_functions  /* Create GIMPLE */
  cgraph_analyze_function  /* Create GIMPLE */
  ...
  cgraph_optimize
  ipa_passes
    execute_ipa_pass_list(all_small_ipa_passes)  /*!in lto*/
    execute_ipa_summary_passes(all_regular_ipa_passes)
    execute_ipa_summary_passes(all_lto_gen_passes)
    ipa_write_summaries
  cgraph_expand_all_functions
    cgraph_expand_function
    /* Intraprocedural passes on GIMPLE, */
    /* expansion pass, and passes on RTL. */
```
cc1 and lto1

toplev_main...
  compile_file...
  cgraph_analyze_function

  cgraph_optimize...
    ipa_passes...
    cgraph_expand_all_functions...
    tree_rest_of_compilation

c1

lto1

lto_main...
  read_cgraph_and_symbols...
  materialize_cgraph

cc1 and lto1

toplev_main...
  compile_file...
  cgraph_analyze_function

  cgraph_optimize...
    ipa_passes...
    cgraph_expand_all_functions...
    tree_rest_of_compilation

c1
Our Pictorial Convention

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

The GNU Tool Chain: Our First Picture

Source Program

Target Program

Notes
The GNU Tool Chain for LTO Support

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay

Common Code (executed twice for each function in the input program)

```c
cc1' lto1' cc1

“Fat” .s files
as as

Single .s file
as as

Single .o file + glibc/newlib

a.out file
```

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
WHOPR Mode of Link Time Optimization (1)

• "Fat" files could be real fat
• For large programs with thousands of functions, the entire program may not fit in the memory
• It would be useful to read only
  ▶ the call graph and not function bodies
  ▶ summary information for each function
• This would enable independent processing of functions at the interprocedural level
  Parallel analysis on multiple CPUs analysis would be an added advantage

WHOPR Mode of Link Time Optimization (2)

Three steps
• LGEN: Local Generation of summary information Potentially Parallel
• WPA: Whole Program Analysis Sequential
  ▶ Reads the call graph and not function bodies
  ▶ summary information for each function
• LTRANS: Local Transformations Potentially Parallel
Part 6

Conclusions

1 July 2011 Plugins: Conclusions 52/52

Conclusions

• Excellent mechanism of plugging in different
  ◦ translators in the main driver
  ◦ front ends, passes, and back ends in the main compiler

• However, the plugins have been used in an adhoc manner

• LTO provides a good support for real interprocedural analysis and optimization