Plugin Mechanisms in GCC

Uday Khedker

(www.cse.iitb.ac.in/~uday)

GCC Resource Center,
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

13 June 2014

Outline

- Motivation
- Plugins in GCC

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
  - No changes in the main source
    Requires dynamic linking

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 rebuild
    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

Static Plugins in the GCC Driver

Source Program

gcc/g++

cc1/cc1plus

cpp

as

glibc/newlib

ld

Target Program

Static Plugins in the GCC Driver

Source Program

gcc/g++

cc1/cc1plus

cpp

Plugin for
a translator in the
driver gcc

as

glibc/newlib

ld

Target Program

Static Plugins in the Generated Compiler

Static Plugins in the Generated Compiler

Static Plugins in the Generated Compiler

Static Plugins in the Generated Compiler

Part 2

Static Plugins in GCC

GCC's Solution

Plugin	Implementation	
	Data Structure	Initialization
Translator in gcc/g++	Array of C structures	Development time
Front end in cc1/cc1plus	C structure	Build time
Passes in cc1/cc1plus	Linked list of C structures	Development time
Back end in cc1/cc1plus	Arrays of structures	Build time

Plugin Data Structure in the GCC Driver

```
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.  */
};
```

Default Specs in the Plugin Data Structure in gcc.c

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

```
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},
  {".FOR", "#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}
}
```

Default Specs in the Plugin Data Structure in gcc.c

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

```
static const struct compiler default_compilers[] =
{
  {".cc", "#C++", 0, 0, 0},
  {".cpp", "#C++", 0, 0, 0},
  {".c++", "#C++", 0, 0, 0},
  {".CPP", "#C++", 0, 0, 0},
  {".ads", "#Ada", 0, 0, 0},
  {".f", "#Fortran", 0, 0, 0},
  {".for", "#Fortran", 0, 0, 0},
  {".f90", "#Fortran", 0, 0, 0},
  {".p", "#Pascal", 0, 0, 0},
  {".java", "#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}
  {".cxx", "#C++", 0, 0, 0},
  {".cp", "#C++", 0, 0, 0},
  {".C", "#C++", 0, 0, 0},
  {".ii", "#C++", 0, 0, 0},
  {".adb", "#Ada", 0, 0, 0},
  {".F", "#Fortran", 0, 0, 0},
  {".FOR", "#Fortran", 0, 0, 0},
  {".F90", "#Fortran", 0, 0, 0},
  {".pas", "#Pascal", 0, 0, 0},
  {".class", "#Java", 0, 0, 0},
}
```

- @: Aliased entry

Default Specs in the Plugin Data Structure in gcc.c

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

```
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},     {".FOR", "#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}
}
```

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

Complete Entry for C in gcc.c

```
  {"@c",
   /* cc1 has an integrated ISO C preprocessor.  We should invoke the
      external preprocessor if -save-temps is given.  */
     "%{E|M|MM:%(trad_capable_cpp) %(cpp_options) %(cpp_debug_options)}\
      %{!E:%{!M:%{!MM:\
          %{traditional|ftraditional:\
%eGNU C no longer supports -traditional without -E}\
       %{!combine:\
  %{save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \
%(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n\
    cc1 -fpreprocessed %{save-temps:%b.i} %{!save-temps:%g.i} \
%(cc1_options)}\
  %{!save-temps:%{!traditional-cpp:%{!no-integrated-cpp:\
cc1 %(cpp_unique_options) %(cc1_options)}}}\
          %{!fsyntax-only:%(invoke_as)}} \
      %{combine:\
  %{save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \
%(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i}}\
  %{!save-temps:%{!traditional-cpp:%{!no-integrated-cpp:\
cc1 %(cpp_unique_options) %(cc1_options)}}\
                %{!fsyntax-only:%(invoke_as)}}}}}}", 0, 1, 1},
```

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

```
{".cc",  "@c++", 0, 0, 0},
{".cp",  "@c++", 0, 0, 0},
{".cxx", "@c++", 0, 0, 0},
{".cpp", "@c++", 0, 0, 0},
{".c++", "@c++", 0, 0, 0},
{".C",   "@c++", 0, 0, 0},
{".CPP", "@c++", 0, 0, 0},
{".H",   "@c++-header", 0, 0, 0},
{".hpp", "@c++-header", 0, 0, 0},
{".hp",  "@c++-header", 0, 0, 0},
{".hxx", "@c++-header", 0, 0, 0},
{".h++", "@c++-header", 0, 0, 0},
{".HPP", "@c++-header", 0, 0, 0},
{".tcc", "@c++-header", 0, 0, 0},
{".hh",  "@c++-header", 0, 0, 0},
```

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

```
  {"@c++-header",
    "%{E|M|MM:cc1plus -E %(cpp_options) %2 %(cpp_debug_options)}\
     %{!E:%{!M:%{!MM:\
       %{save-temps|no-integrated-cpp:cc1plus -E\
%(cpp_options) %2 -o %{save-temps:%b.ii} %{!save-temps:%g.ii} \n}\
      cc1plus %{save-temps|no-integrated-cpp:-fpreprocessed %{save-temps:%
      %{!save-temps:%{!no-integrated-cpp:%(cpp_unique_options)}}\
%(cc1_options) %2\
%{!fsyntax-only:%{!fdump-ada-spec*:-o %g.s %{!o*:--output-pch=%i.gch}\
        %W{o*:--output-pch=%*}}%V}}}}",
     CPLUSPLUS_CPP_SPEC, 0, 0},
```

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

```
  {"@c++",
    "%{E|M|MM:cc1plus -E %(cpp_options) %2 %(cpp_debug_options)}\
     %{!E:%{!M:%{!MM:\
       %{save-temps|no-integrated-cpp:cc1plus -E\
%(cpp_options) %2 -o %{save-temps:%b.ii} %{!save-temps:%g.ii} \n}\
      cc1plus %{save-temps|no-integrated-cpp:-fpreprocessed %{save-temps:%
      %{!save-temps:%{!no-integrated-cpp:%(cpp_unique_options)}}\
%(cc1_options) %2\
       %{!fsyntax-only:%(invoke_as)}}}}",
     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},
```

Populated Plugin Data Structure for LTO: gcc/lto/lang-specs.h

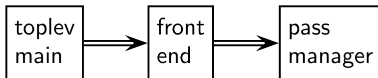
```
/* LTO contributions to the "compilers" array in gcc.c.  */

  {"@lto", "lto1 %(cc1_options) %i %{!fsyntax-only:%(invoke_as)}",
   /*cpp_spec=*/NULL, /*combinable=*/1, /*needs_preprocessing=*/0},
```

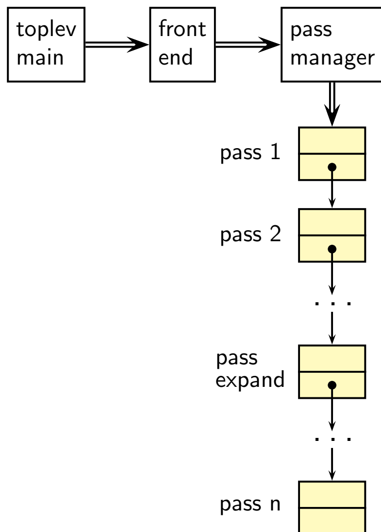
What about the Files to be Procecced by the Linker?

- 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



Plugin Structure in cc1



Double arrow represents control flow whereas single arrow represents pointer or index

For simplicity, we have included all passes in a single list. Actually passes are organized into five lists and are invoked as five different sequences

Plugin Structure in `cc1`

Plugin Structure in cc1

Plugin Structure in cc1

Plugin Structure in `cc1`

Plugin Structure in `cc1`

Plugin Structure in cc1

Plugin Structure in `cc1`

Front End Plugin

Important fields of `struct lang_hooks` instantiated for C

```
#define LANG_HOOKS_FINISH c_common_finish
#define LANG_HOOKS_PARSE_FILE c_common_parse_file
#define LANG_HOOKS_WRITE_GLOBALS c_write_global_declarations
```

Plugins for Pass Specifications

Intraprocedural Passes	Interprocedural Passes
struct gimple_opt_pass struct opt_pass	struct simple_ipa_opt_pass struct opt_pass
struct rtl_opt_pass struct opt_pass	struct ipa_opt_pass_d struct opt_pass /* Additional fields */

struct opt_pass

Plugins for Intraprocedural Passes

```
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;
};
```

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

struct rtl_opt_pass
{
  struct opt_pass pass;
};
```

Plugins for Interprocedural Passes on a Translation Unit

Pass variable: `all_simple_ipa_passes`

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

Plugins for Interprocedural Passes across a Translation Unit

Pass variable: `all_regular_ipa_passes`

```
struct ipa_opt_pass_d
{
  struct opt_pass pass;
  void (*generate_summary) (void);
  void (*read_summary) (void);
  void (*write_summary) (struct cgraph_node_set_def *,
                         struct varpool_node_set_def *);
  void (*write_optimization_summary)(struct cgraph_node_set_def *,
                                     struct varpool_node_set_def *);
  void (*read_optimization_summary) (void);
  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 *);
};
```

Predefined Pass Lists

Pass List	Purpose
all_lowering_passes	AST to CFG translation
all_small_ipa_passes	Interprocedural passes restricted to a single translation unit
all_regular_ipa_passes	Interprocedural passes on a translation unit as well as across translation units (during WPA/IPA of LTO)
all_late_ipa_passes	Interprocedural passes on partitions created by LTO (after WPA/IPA)
all_lto_gen_passes	Passes to encode program for LTO
all_passes	Intraprocedural passes on GIMPLE and RTL

Registering a Pass as a Static Plugin

1. Write the driver function in your file
2. Declare your pass in file `tree-pass.h`:
   `extern struct gimple_opt_pass your_pass_name;`
3. Add your pass to the appropriate pass list in
   `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 debuging `cc1` from within `gcc`, see:
   `http://gcc.gnu.org/ml/gcc/2004-03/msg01195.html`)

Part 3

Dynamic Plugins in GCC

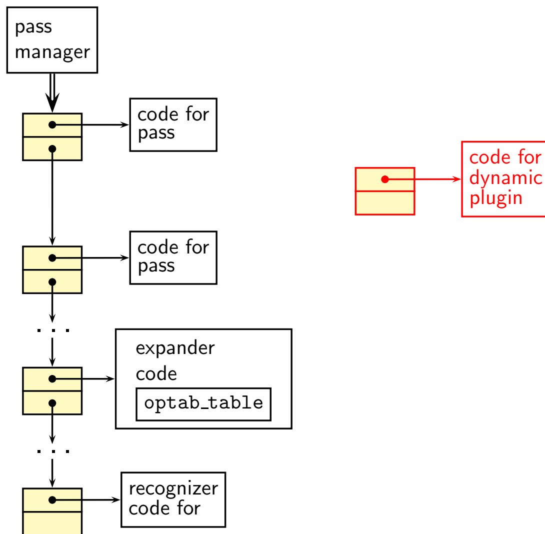
Dynamic Plugins

- 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

The Mechanism of Dynamic Plugin



The Mechanism of Dynamic Plugin

pass manager

code for pass

code for dynamic plugin

code for pass

expander code

`optab_table`

recognizer code for

Runtime initialization of the appropriate linked list of passes

Made possible by dynamic linking

Specifying an Example Pass

```
struct simple_ipa_opt_pass pass_plugin = {
  {
    SIMPLE_IPA_PASS,
    "dynamic_plug",                 /*  name */
    0,                              /*  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

```
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

```
int plugin_init(struct plugin_name_args *plugin_info,
    struct plugin_gcc_version *version)
{ /*  Plugins are activiated 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/g++
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
```