Outline

- Motivation
- Plugins in GCC
- GCC Control Flow
- Link time optimization in GCC
- Conclusions
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 (e.g. 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
  cc1  \rightarrow  cpp
  as
  glibc/newlib
  ld
```

Target Program

Notes

Plugin for a translator in the driver gcc
Static Plugins in the Generated Compiler

Parser → Gimplifier → Tree SSA Optimizer → RTL Generator → Optimizer → Code Generator

Input Language

Compiler Generation Framework

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Selected → Copied → Generated

Source Program

Generated Compiler (cc1)

Assembly Program

Notes

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
### 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>Build 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>
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. */
};

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},
{ "i", "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", "C-compiler", 0, 1, 0},
{ "s", "Assembler", 0, 1, 0}
};
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))\n  %(E|M|M):\n  %(traditional|ftraditional:\n   %{E\n      %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n        %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n         cc1 -fpreprocessed %{save-temps:%b.i} %{!save-temps:%g.i} \n        ccc1 %(cpp_unique_options) %{cc1_options})\n  %{!E:%{!M:%{!MM:\n      %{traditional|ftraditional:\n       %{E\n          %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n            %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n            ccc1 %(cpp_unique_options) %{cc1_options})\n       %{!E:%{!M:%{!MM:\n         %{traditional|ftraditional:\n          %{E\n            %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n              %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n              ccc1 %(cpp_unique_options) %{cc1_options})\n         %{!E:%{!M:%{!MM:\n            %{traditional|ftraditional:\n             %{E\n               %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n                 %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n                 ccc1 %(cpp_unique_options) %{cc1_options})\n               %{!E:%{!M:%{!MM:\n                  %{traditional|ftraditional:\n                   %{E\n                     %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n                       %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n                       ccc1 %(cpp_unique_options) %{cc1_options})\n                   %{!E:%{!M:%{!MM:\n                      %{traditional|ftraditional:\n                       %{E\n                          %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n                            %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n                            ccc1 %(cpp_unique_options) %{cc1_options})\n                          %{!E:%{!M:%{!MM:\n                           %{traditional|ftraditional:\n                            %{E\n                                %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n                                  %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n                                  ccc1 %(cpp_unique_options) %{cc1_options})\n                                %{!E:%{!M:%{!MM:\n                                 %{traditional|ftraditional:\n                                  %{E\n                                    %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n                                      %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n                                      ccc1 %(cpp_unique_options) %{cc1_options})\n                                    %{!E:%{!M:%{!MM:\n                                     %{traditional|ftraditional:\n                                      %{E\n                                        %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n                                          %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n                                          ccc1 %(cpp_unique_options) %{cc1_options})\n                                        %{!E:%{!M:%{!MM:\n                                         %{traditional|ftraditional:\n                                          %{E\n                                            %(save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \n                                              %(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n                                              ccc1 %(cpp_unique_options) %{cc1_options})\n                                            %{!E:%{!M:%{!MM:\n                                               %{traditional|ftraditional:\n                                                 %(E|
".cc", "@c++", 0, 0, 0},
{".cp", "@c++", 0, 0, 0},
{".cxx", "@c++", 0, 0, 0},
{".cpp", "@c++", 0, 0, 0},
{".cpp", "@c++", 0, 0, 0},
{".cpp", "@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},
{".hxx", "@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}\n cc1plus %{save-temps|no-integrated-cpp:-fpreprocessed %{save-temps:%b.ii} %{!save
 %{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},

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Populated Plugin Data Structure for LTO:
gcc/lto/lang-specs.h

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

{"@lto", "lto1 %c1_options %i %!fsyntax-only:%t{invoke_as}}",
/*cpp_spec=*/NULL, /*combinable=*/1, /*needs_preprocessing=*/0,}

What about the Files to be Processed by the Linker?

- Linking is the last step
- Every file is passed on to the linker unless it is suppressed
- If a translator is not found, the input file is assumed to be a file for the linker
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_HOOKSPARSE_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 on a Translation Unit

Pass variable: all_simple_ipa_passes

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

Plugins for Interprocedural Passes across a Translation Unit

Pass variable: all_regular_ipa_passes

```c
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 *);
    unsigned int (*variable_transform) (struct varpool_node *);
};
```
Predefined Pass Lists

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-passes</td>
<td>all-passes</td>
</tr>
<tr>
<td>all-lower-passes</td>
<td>all-lower-passes</td>
</tr>
<tr>
<td>all-regular-passes</td>
<td>all-regular-passes</td>
</tr>
<tr>
<td>ipa-passes</td>
<td>ipa-passes</td>
</tr>
<tr>
<td>lto-passes</td>
<td>lto-passes</td>
</tr>
</tbody>
</table>

Notes

1. Write the driver function in your file.
2. Declare your pass in the tree-pass.h file.
3. Add your pass to the appropriate pass list in initoptimizations passes().
4. Add your file details to $SOURCE/gcc/Makefile.in.
5. Configure and build gcc.
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
The Mechanism of Dynamic Plugin

Runtime initialization of the appropriate linked list of passes
Made possible by dynamic linking
Specifying an Example Pass

```c
struct simple_ipa_opt_pass pass_plugin = {
    SIMPLE_IPA_PASS,
    "dynamic_plugin",
    0,
    execute_pass_plugin,
    NULL,
    info,
    0,
    TV_INTEGRATION,
    0,
    properties_required,
    0,
    properties_provided,
    0,
    properties_destroyed,
    0,
    todo_flags_start,
    0
};
```

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

    return 0;
}
```

Makefile for Creating and Using a Dynamic Plugin

```make
CC = $(INSTALL_D)/bin/gcc
PLUGIN_SOURCES = new-pass.c
PLUGIN_OBJECTS = $(patsubst %.c,%.o,$(PLUGIN_SOURCES ))
GCCPLUGINS_DIR ... -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
```
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
main
/* In file gcc.c */
validate_all_switches
lookup_compiler
do_spec
  do_spec_2
    do_spec_1 /* Get the name of the compiler */
execute
  pex_init
  pex_run
    pex_run_in_environment
    obj->funcs->exec_child

Observations
• All compilers are invoked by this driver
• Assembler is also invoked by this driver
• Linker is invoked in the end by default
main
toplev_main /* In file toplev.c */
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

Observations
- The entire compilation is driven by functions specified in language hooks
- Not a good design!
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_DECLspecs /* parse declarations */
    c_parser_compound_statement
    finish_function  /* finish parsing */
    c_genericize
    cgraph_finalize_function
    /* finalize AST of a function */

Observations

- GCC has moved to a recursive descent parser from version 4.1.0
- Earlier parser was generated using Bison specification
lang_hooks.decls.final_write_globals =>
    c_write_global_declarations
    cgraph_finalize_compilation_unit
    cgraph_analyze_functions /* Create GIMPLE */
    cgraph_analyze_function
gimplify_function_tree
    gimplify_body
    gimplify_stmt
    gimplify_expr
cgraph_lower_function /* Intraprocedural */
tree_lowering_passes
evaluate_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!
## Organization of Passes

<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>

---

**Essential Abstractions in GCC**

GCC Resource Center, IIT Bombay

---

```
cgraph_analyze_function /* Create GIMPLE */
```

```
cgraph_optimize
  ipa_passes
    execute_ipa_pass_list(all_small_ipa_passes) /*!in_lto_p*/
    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. */
    tree_rest_of_compilation
    execute_pass_list (all_passes)
```
cgraph_analyze_function /* Create GIMPLE */
...
/* previous slide */
cgraph_optimize
ipa_passes
execute_ipa_passes()
execute_ipa_summary_passes()
execute_ipa_summary_passes(all_lto_gen_passes)
icom三位一体
ipa_write_summaries

cgraph_expand_all_functions

cgraph_expand_function

tree_rest_of_compilation
exectute_pass_list (all_passes)

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>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Pass 2</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Pass 3</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Pass 4</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Pass 5</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
Execution Order in Intraprocedural Passes

Function 1  Function 2  Function 3  Function 4  Function 5
Pass 1
Pass 2
Pass 3
Pass 4
Pass 5

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
Pass 2
Pass 3
Pass 4
Pass 5

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

Pass 2

Pass 3

Pass 4

Pass 5

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
Motivation for Link Time Optimization

- Default `cgraph` creation is restricted to a translation unit (i.e. a single file)
  ⇒ Interprocedural analysis and optimization is restricted to a single file

- All files (or their equivalents) are available only at link time
  (assuming static linking)

- LTO enables interprocedural optimizations across different files
Link Time Optimization

- LTO framework supported in GCC-4.6.0
- Use `-flto` option during compilation
- Generates conventional `.o` files with GIMPLE level information inserted
  Complete translation is performed in this phase
- During linking all object modules are put together and `lto1` is invoked
- `lto1` 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");
}

Assembly Output without LTO Information (1)

```
.file "t0.c"
.LC0:
.string "hello, world"
.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
  call puts
  leave
  .cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 4.6.0"
```

Assembly Output with LTO Information (2)

```
.ascii "\007"
.text
.section .gnu.lto..refs.6a5c5521,"",@progbits
.string \"\x234cb''\006b&\006\030\text
.string ""
.string ""
.string "t"
.ascii "\b"
.text
.section .gnu.lto..statics.6a5c5521,"",@progbits
.string \"\x234cb\'\'\030\016\342\214\020\330\260\213\237\242\336\207\b{\204}B\222p\032\277F8\037"
```
Assembly Output with LTO Information (5)

```
.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
```
Whole program optimization needs to see the entire program

- Does it need the entire program together in the memory?

- Load only the call graph without function bodies
  - Independent computation of summary information of functions
  - “Adjusting” summary information through whole program analysis over the call graph
  - Perform transformation independently on functions

Multi process LTO

- Process the entire program together

Single process LTO

Why Avoid Loading Function Bodies?

- Practical programs could be rather large and compilation could become very inefficient

- Many optimizations decisions can be taken by looking at the call graph alone
  - Procedure Inlining: just looking at the call graph is sufficient
    - Perhaps some summary size information can be used
  - Procedure Cloning: some additional summary information about actual parameters of a call is sufficient
Multi Process LTO (aka WHOPR Mode of LTO)

- Three steps
  - LGEN: Local generation of summary information and translation unit information \(\text{Potentially Parallel}\)
  - WPA: Whole Program Analysis \(\text{Sequential}\)
    - Reads the call graph and not function bodies
    - Summary information for each function
  - LTRANS: Local Transformations \(\text{Potentially Parallel}\)
- Why do we call this LTO \textit{Multi Process LTO}?
  - \texttt{gcc} executes LGEN
  - Subsequent process of \texttt{lto1} executes WPA
  - Subsequent independent processes of \texttt{lto1} execute LTRANS

---

Single Process LTO

- Three steps
  - LGEN: Local Generation of translation unit information (no summary) \(\text{Potentially Parallel}\)
  - IPA: Inter-Procedural Analysis \(\text{Sequential}\)
    - Reads the call graph and function bodies
  - LTRANS: Local Transformations \(\text{Sequential}\)
- Why do we call this LTO \textit{Single Process LTO}?
  - \texttt{gcc} executes LGEN
  - Subsequent process of \texttt{lto1} executes both IPA and LTRANS
  - When \texttt{-flto-partition=none}, IPA = WPA
struct ipa_opt_pass_d {
  struct opt_pass pass;
  void (*generate_summary) (void);
  void (*read_summary) (void);
  void (*write_summary) (struct cg_node_set_def *,
                           struct varpool_node_set_def *);
  void (*write_optimization_summary)(struct cg_node_set_def *,
                                      struct varpool_node_set_def *);
  void (*read_optimization_summary) (void);
  void (*stmt_fixup) (struct cg_node *, gimple *);
  unsigned int function_transform_todo_flags_start;
  unsigned int (*function_transform) (struct cg_node *);
  void (*variable_transform) (struct varpool_node *);
};
struct ipa_opt_pass_d
{
    struct opt_pass pass;
    void (*generate_summary) (void);
    void (*read_summary) (void);
    void (*write_summary) (struct cg_node_set_def *,
                           struct varpool_node_set_def *);
    void (*write_optimization_summary)(struct cg_node_set_def *,
                                        struct varpool_node_set_def *);
    void (*read_optimization_summary) (void);
    void (*stmt_fixup) (struct cg_node *, gimple *);
    unsigned int (*function_transform)(todo_flags_start;
    unsigned int (*function_transform) (struct cg_node *);
    void (*variable_transform) (struct varpool_node *);
};
### LTO Pass Hooks

```c
struct ipa_opt_pass_d
{
    struct opt_pass pass; (member void (*execute)(void));
    void (*generate_summary)(void);
    void (*read_summary)(void);
    void (*write_summary)(struct cgraph_node_set_def *,
                          struct varpool_node_set_def *,
                          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 *);
    void (*stmt_fixup) (struct cgraph_node *, gimple *);
};
```

IPA for Single Process LTO

### Notes

LTRANS for Multi Process LTO
```c
struct ipa_opt_pass_d
{
    struct opt_pass pass;
    void (*generate_summary) (void);
    void (*read_summary) (void);
    void (*write_summary) (struct cg_node_set_def *,
                           struct vp_node_set_def *);
    void (*write_optimization_summary)(struct cg_node_set_def *,
                                        struct vp_node_set_def *);
    void (*read_optimization_summary) (void);
    void (*stmt_fixup) (struct cg_node *, gimple *);
    unsigned int function_transform_todoFlags_start;
    unsigned int (*function_transform)(struct cg_node *);
    void (*variable_transform) (struct vp_node *);
};
```

---

### LTRANS for Single Process LTO

### LTO Support in GCC

<table>
<thead>
<tr>
<th>Transformation</th>
<th>In the same process as that of analysis</th>
<th>In an independent process (possibly multiple processes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single partition of the program</td>
<td>Single partition of the program</td>
<td>Multiple partitions of the program</td>
</tr>
<tr>
<td>Whole Program Analysis</td>
<td>Call graph without function bodies</td>
<td>Not supported</td>
</tr>
<tr>
<td></td>
<td>Call graph with function bodies</td>
<td>Supported in GCC-4.6.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Will be supported in future</td>
</tr>
</tbody>
</table>

- `-flto`
- `-flto -flto-partition=none`
- WHOPR mode
lto_main
lto_process_name
lto_init_reader
read_cgraph_and_symbols
if (flag_wpa)
/* WPA for multi process LTO */
do_whole_program_analysis
  materialize_cgraph
  execute_ipa_pass_list (all_regular_ipa_passes)
  lto_ipa_write_files
else
/* WPA and LTRANS for single process LTO */
/* Only LTRANS for multi process LTO */
  materialize_cgraph
  cgraph_optimize

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. */
Plugins: Link Time Optimization

**GCC Resource Center, IIT Bombay**

```
toplevel_main
  ...
  compile_file
  ...
  cgraph_analyze_function

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

```
cc1

  lto_main
  ...
  read_cgraph_and_symbols
  ...
  materialize_cgraph

  lto

  ...
  ipa_passes
  ...
  cgraph_expand_all_functions
  ...
  tree_rest_of_compilation
```
The GNU Tool Chain for Single Process LTO Support

cc1
cc1′
lto1′
common

“Fat” .s files
as

“Fat” .o files
cc1
cc1′
lto1
common

Single .s file
as

Single .o file + glibc/newlib

collect2
ld

ld

a.out file

Common Code (executed twice for each function in the input program for single process LTO. Once during LGEN and then during WPA + LTRANS)

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. */

a.out file
Multi Process LTO (aka WHOPR LTO)

Option `-flto -c`

External View

Option `-flto -o out`

Internal View

large call graph without procedure bodies (Interproc. analysis: √ Transformation: X)

/tmp/ccdKEyVB.ltrans0.o (possibly multiple files)
**Multi Process LTO (aka WHOPR LTO)**

**Option** `-f.lto -c`

- `f1.c` → `cc1` → `lto1` → `f1.o`
- `f2.c` → `cc1` → `lto1` → `f2.o`
- `f3.c` → `cc1` → `lto1` → `f3.o`

**WPA**

**Option** `-f.lto -o` `out`

Large call graph without procedure bodies (Interproc. analysis: √ Tranformation: ×)

/tmp/ccdKEyVB.ltrans0.o (possibly multiple files)

```
out common
```

**LGEN**

Essential Abstractions in GCC

**Essential Abstractions in GCC**

GCC Resource Center, IIT Bombay

---

**Single Process LTO**

**Option** `-f.lto -c`

- `f1.c` → `cc1` → `lto1` → `f1.o`
- `f2.c` → `cc1` → `lto1` → `f2.o`
- `f3.c` → `cc1` → `lto1` → `f3.o`

**Option** `-f.lto` `-o` `out`

```
out common
```

**LTRANS**

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

---

Notes
Single Process LTO

Option `-flto -c`

External View

f1.c → `cc1` → `lto1` → `common` → `f1.o`

f2.c → `cc1` → `lto1` → `common` → `ccl` → `ho1` → `f2.o`

f3.c → `cc1` → `lto1` → `common` → `ccl` → `ho1` → `f3.o`

Internal View

Option `-flto -o out
-flto-partition=none`

large call graph with procedure bodies
(Interproc. analysis: √
Transformation: √)

out

Notes

LGEN

IPA + LTRANS

This WPA can examine function bodies also
1 July 2012

Part 6

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 is a good support for interprocedural analysis and optimization
  It would be useful to support
  – a single process LTO mode that
  – creates a large call graph of the entire program with
  – on-demand loading of procedure bodies for
  – enabling examining procedure bodies for interprocedural analysis