

# *Workshop on Essential Abstractions in GCC*

## **GCC Control Flow and Plugins**

GCC Resource Center  
([www.cse.iitb.ac.in/grc](http://www.cse.iitb.ac.in/grc))

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



1 July 2012

# 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
  - ▶ 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



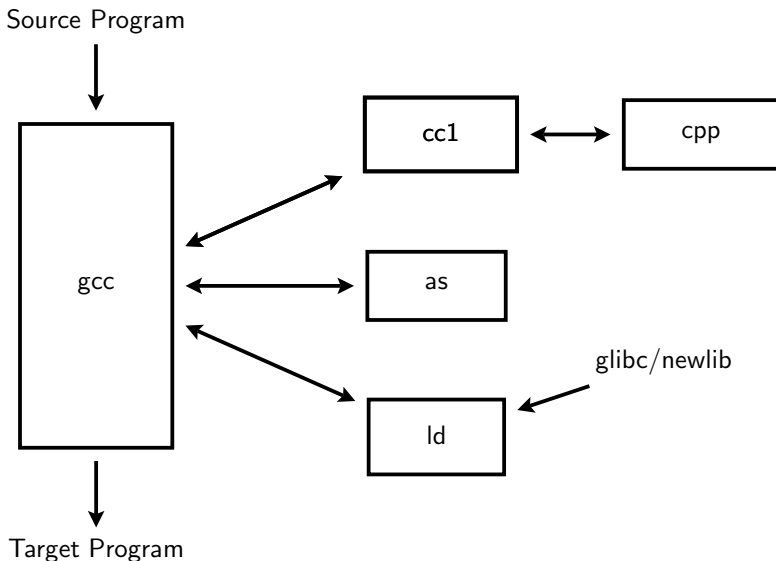
## Static Vs. Dynamic Plugins

- 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

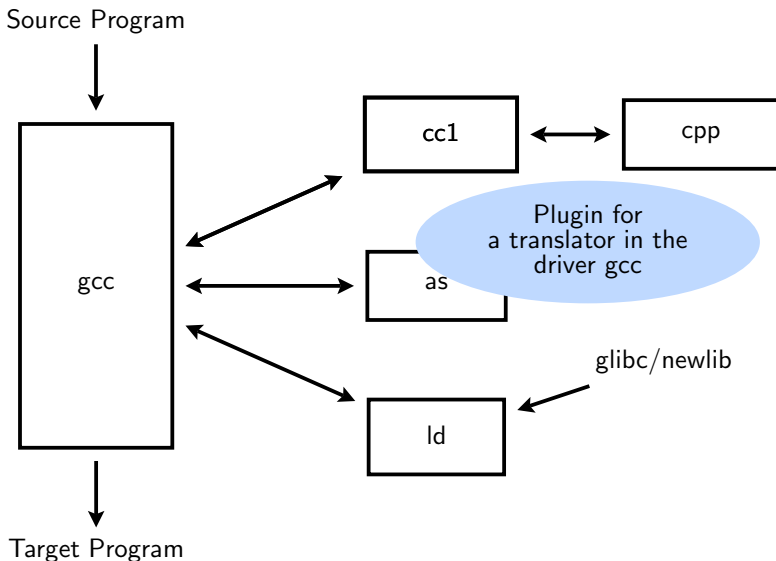


## Static Plugins in the GCC Driver

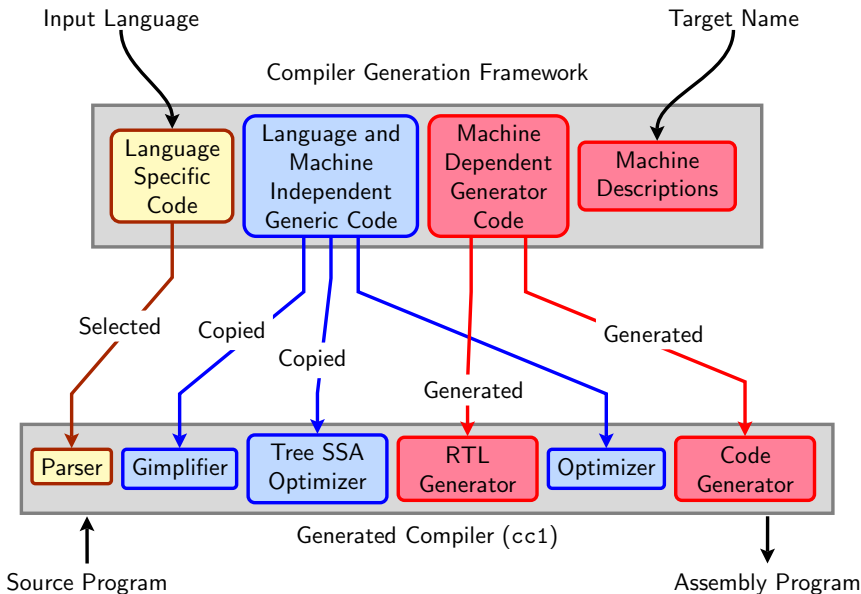




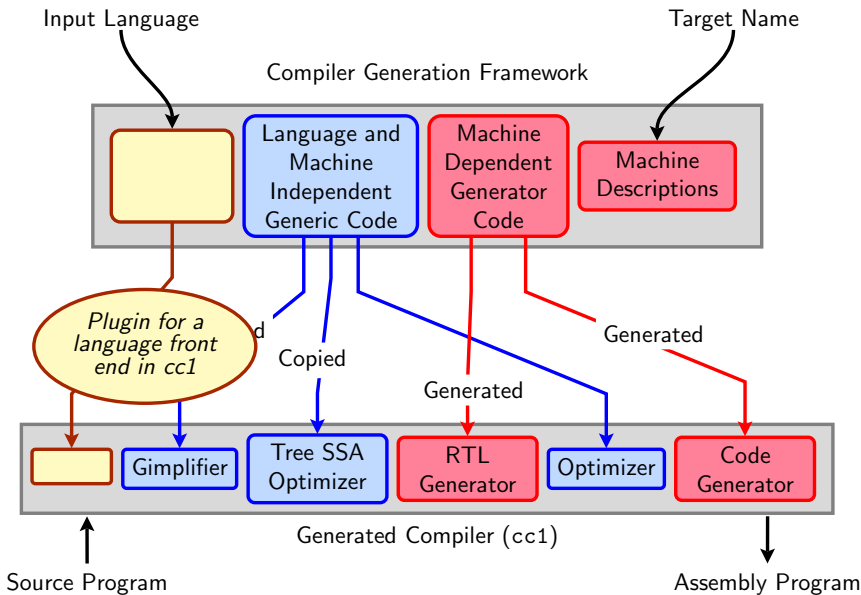
## Static Plugins in the GCC Driver



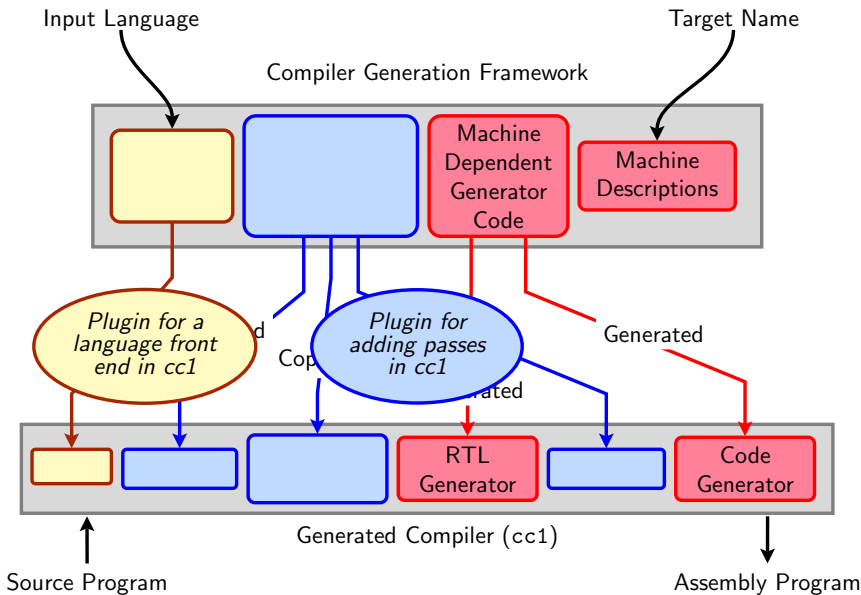
# Static Plugins in the Generated Compiler



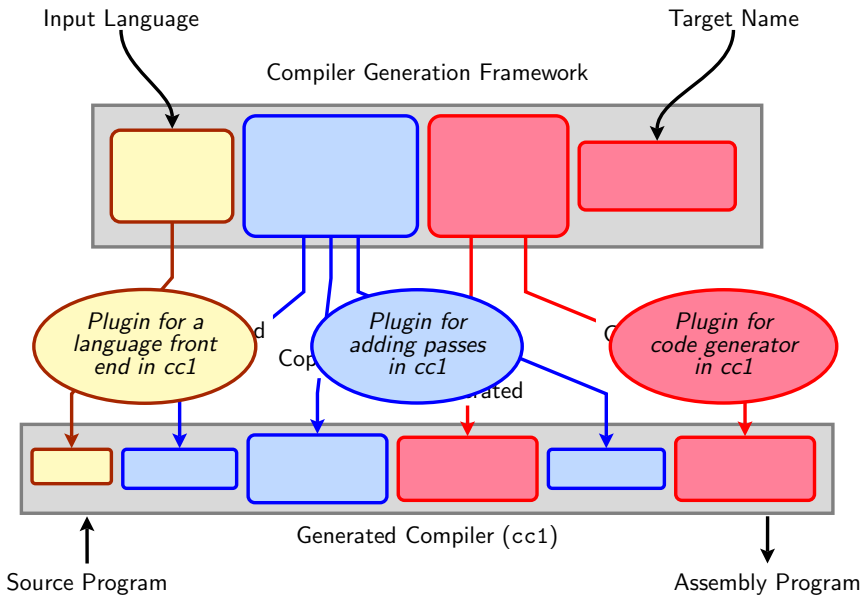
# Static Plugins in the Generated Compiler



# 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	Array of C structures	Development time
Front end in cc1	C structure	Build time
Passes in cc1	Linked list of C structures	Development time
Back end in cc1	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},
  {".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 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},
```



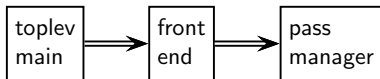


## What about the Files to be Procecded by the Linker?

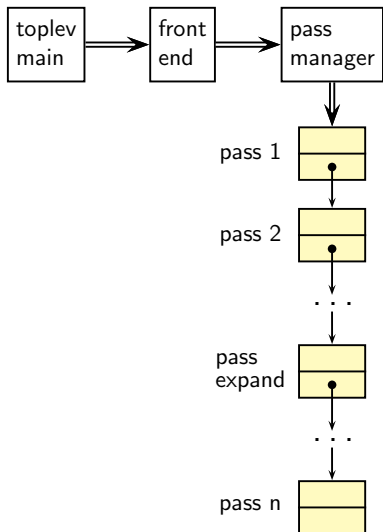
- Linking is the last step
- Every file is passed on to linker unless it is suppressed
- If a translator is not found, input file is assumed to be a file for linker



## Plugin Structure in cc1



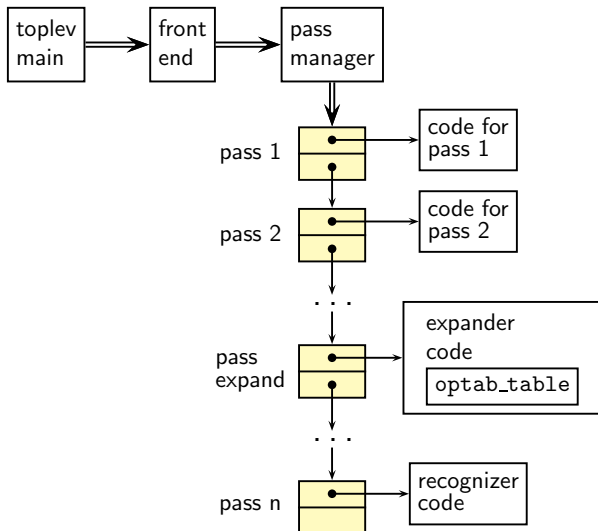
## Plugin Structure in cc1



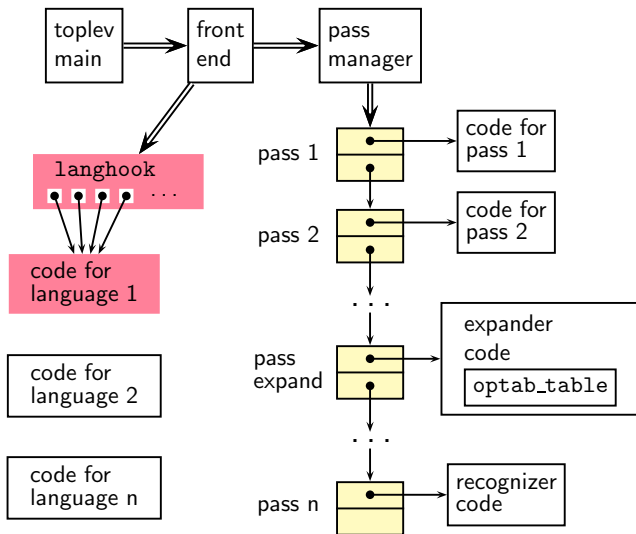
double arrow represents control flow whereas single arrow represents pointer or index



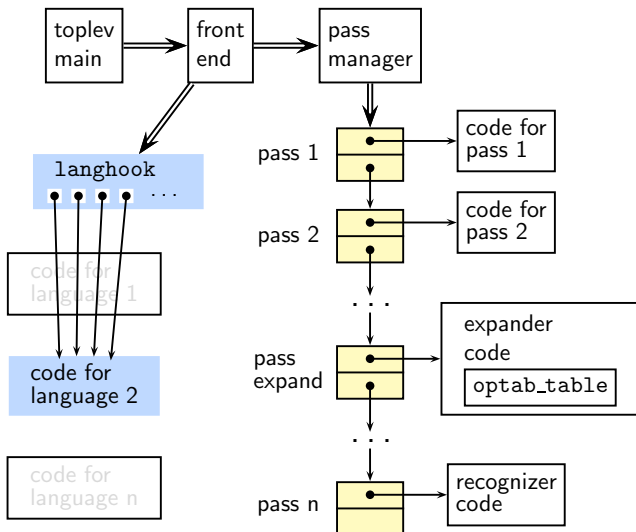
## Plugin Structure in cc1



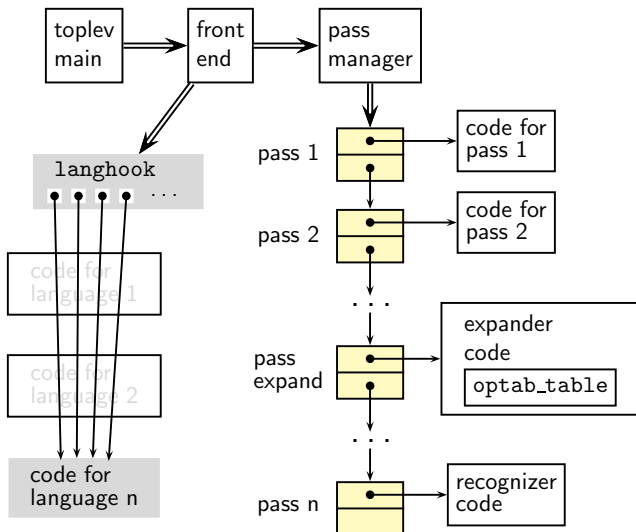
# Plugin Structure in cc1



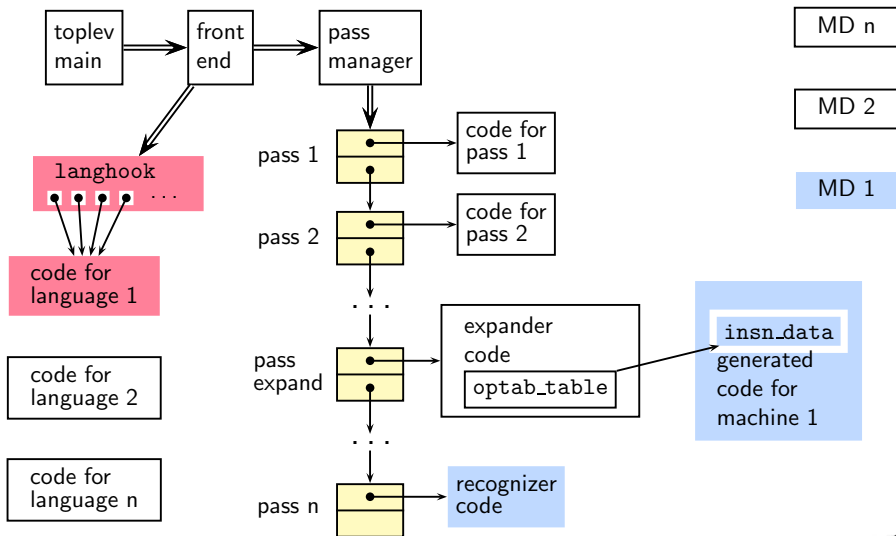
## Plugin Structure in cc1



## Plugin Structure in cc1

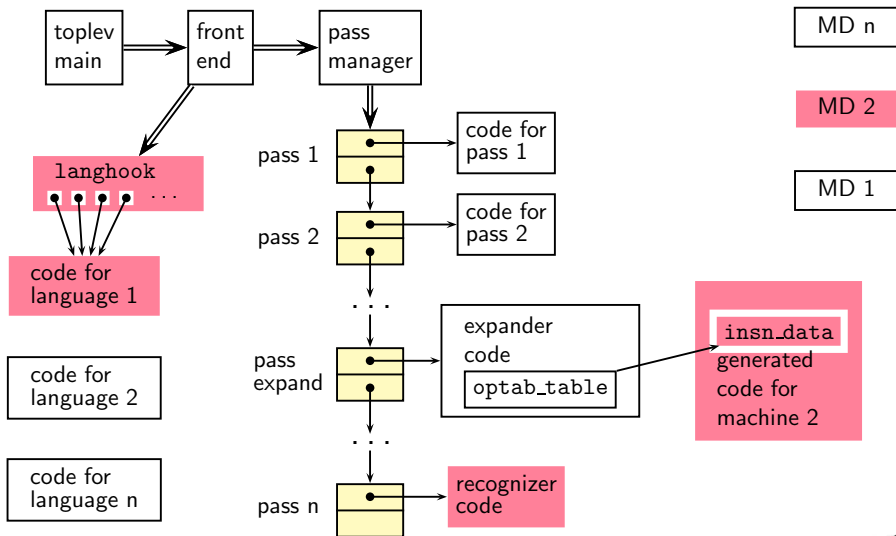


## Plugin Structure in cc1

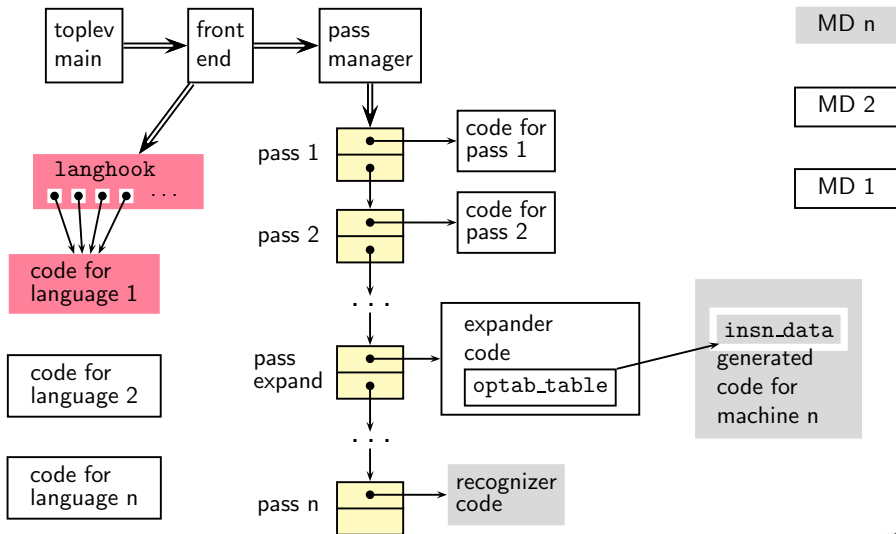




## Plugin Structure in cc1



## 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_EXPAND_EXPR c_expand_expr
#define LANG_HOOKS_PARSE_FILE c_common_parse_file
#define LANG_HOOKS_WRITE_GLOBS c_write_global_declarations
```



## 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 Name	Purpose
<code>all_lowering_passes</code>	Lowering
<code>all_small_ipa_passes</code>	Early optimization passes. Invokes intraprocedural passes over the call graph.
<code>all_regular_ipa_passes</code>	
<code>all_lto_gen_passes</code>	
<code>all_passes</code>	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 debugging `cc1` from within `gcc`, see:  
<http://gcc.gnu.org/ml/gcc/2004-03/msg01195.html>)





*Part 3*

# *Dynamic Plugins in GCC*

## Dynamic Plugins

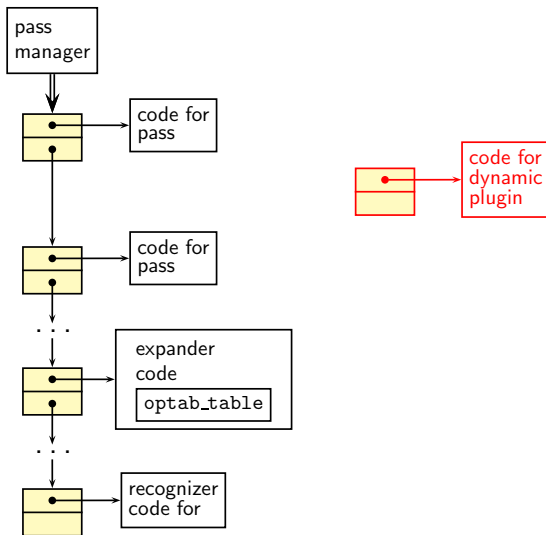
- Supported on platforms that support `-ldl -rdynamic`
- Loaded using `dlopen` and invoked at pre-determined locations in the compilation process
- Command line option

`-fplugin=/path/to/name.so`

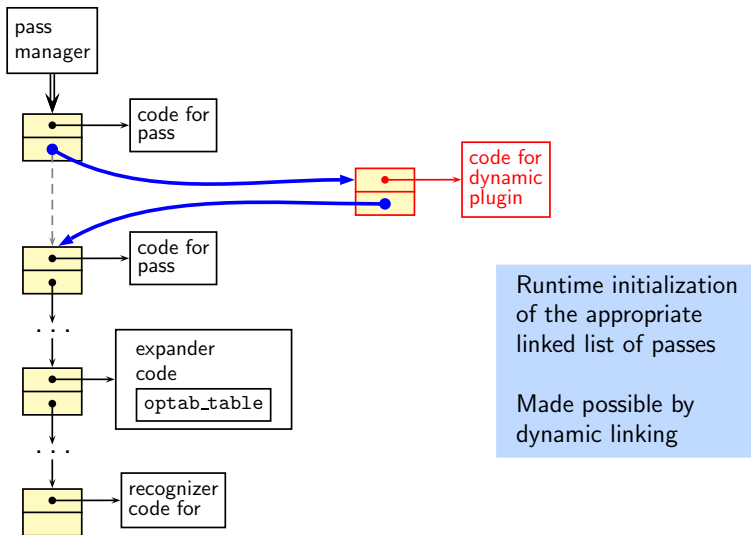
Arguments required can be supplied as name-value pairs



# The Mechanism of Dynamic Plugin



# The Mechanism of Dynamic Plugin



## 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 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`



## gcc Driver Control Flow

```
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
```



## gcc Driver Control Flow

```
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



## cc1 Top Level Control Flow

```
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
```



## cc1 Top Level Control Flow

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

### Observations

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

declarations



## cc1 Control Flow: Parsing for 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_declspecs /* parse declarations */
          c_parser_compound_statement
          finish_function /* finish parsing */
          c_genericize
          cgraph_finalize_function
          /* finalize AST of a function */
```



## cc1 Control Flow: Parsing for C

```
lang_hooks.parse_file => c_common_parse_file
  c_parse_file
    c_parser_translation_unit
```

```
  c_parser_e
```

```
    c_parse
```

```
      c_pa
```

```
      c_pa
```

```
      fini
```

```
      c
```

```
      c
```

```
      /
```

### Observations

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

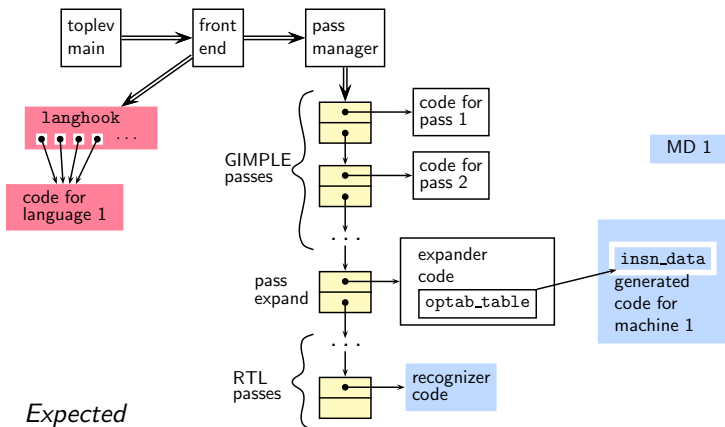
```
ions */
```

```
 */
```

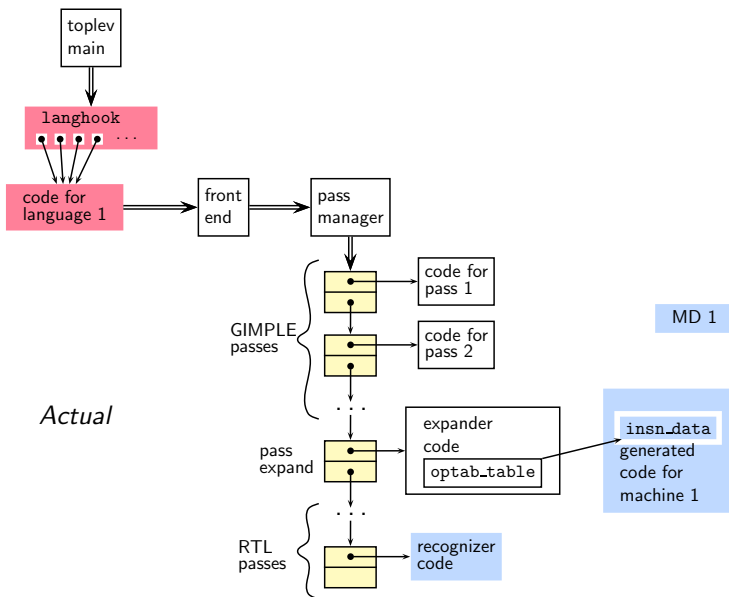




## Expected Vs. Actual Schematic



# Expected Vs. Actual Schematic



## cc1 Control Flow: Lowering Passes for C

```
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
    execute_pass_list (all_lowering_passes)
```



## cc1 Control Flow: Lowering Passes for C

```

lang_hooks.decls.final_write_globals =>
                                c_write_global_declarations
cgraph_finalize_compilation_unit
  cgraph_analyze                */
  cgraph_analyze                */
  gimplify                      */
  gimplify                      */
                                */
cgraph_lower                    */
  tree_lower                    */
  execute                       */

```

### Observations

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



## Organization of Passes

Order	Task	IR	Level	Pass data structure
1	Lowering	GIMPLE	Intra	<code>gimple_opt_pass</code>
2	Optimizations	GIMPLE	Inter	<code>ipa_opt_pass</code>
3	Optimizations	GIMPLE	Intra	<code>gimple_opt_pass</code>
4	RTL Generation	GIMPLE	Intra	<code>rtl_opt_pass</code>
5	Optimization	RTL	Intra	<code>rtl_opt_pass</code>



## cc1 Control Flow: Optimization and Code Generation Passes

```
cgraph_analyze_function      /* Create GIMPLE */  
...                          /* previous slide */  
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)
```



# cc1 Control Flow: Optimization and Code Generation Passes

```

cgraph_analyze_function      /* Create GIMPLE */
...                          /* previous slide */
cgraph_optimize
  ipa_passes
    execute_ipa_passes      /* in_lto_p */
    execute_ipa_summary    /* (es)
    execute_ipa_summary
    ipa_write_summary
cgraph_expand_all
  cgraph_expand
  /* Intraprocedural
  /* expansion pass
    tree_restoration
    execut

```

## Observations

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



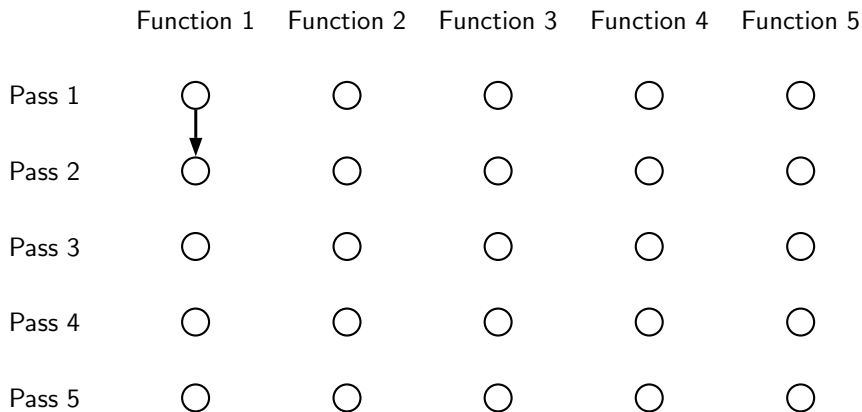
## Execution Order in Intraprocedural Passes

	Function 1	Function 2	Function 3	Function 4	Function 5
Pass 1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 3	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 4	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 5	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

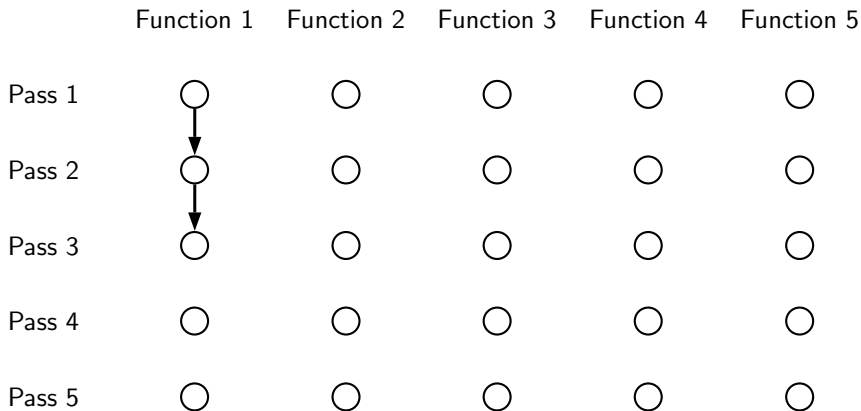




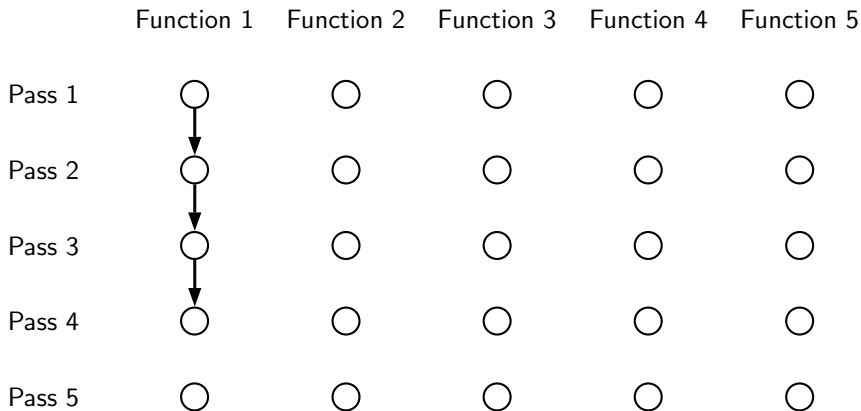
## Execution Order in Intraprocedural Passes



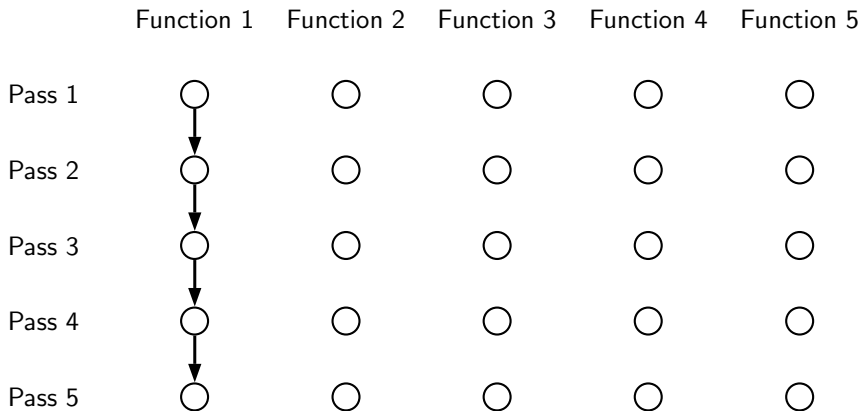
## Execution Order in Intraprocedural Passes



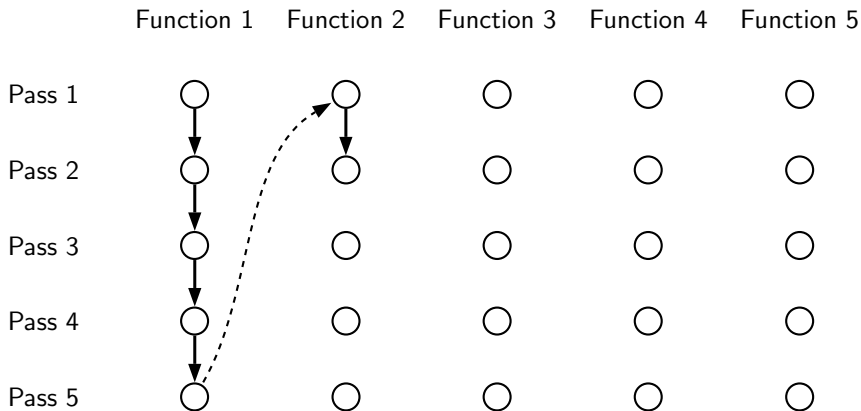
## Execution Order in Intraprocedural Passes



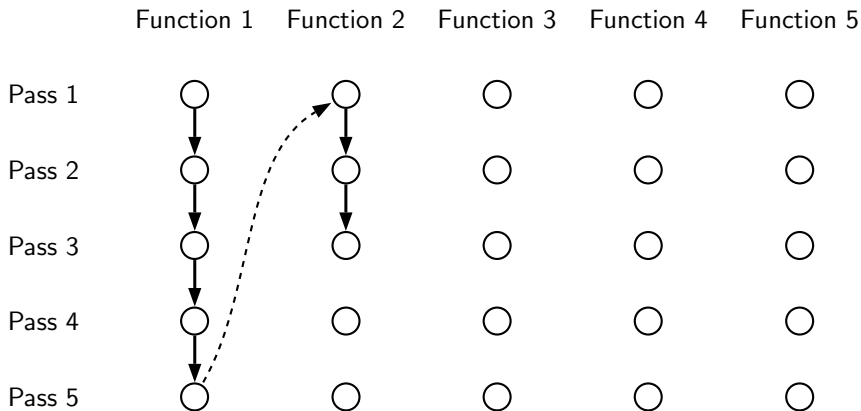
## Execution Order in Intraprocedural Passes



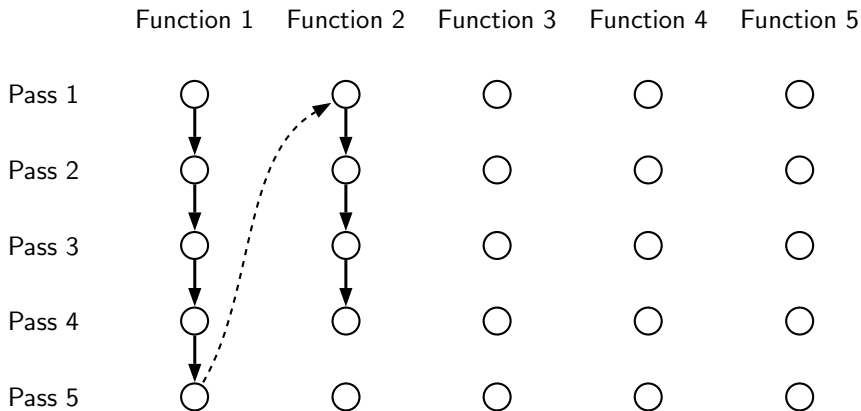
## Execution Order in Intraprocedural Passes



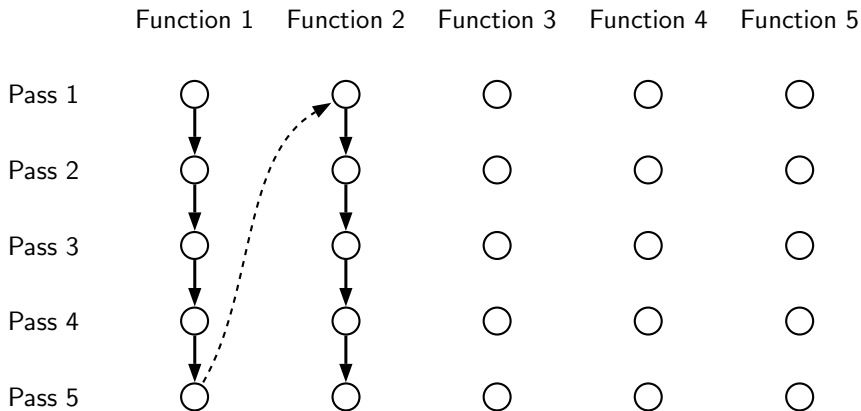
## Execution Order in Intraprocedural Passes



## Execution Order in Intraprocedural Passes

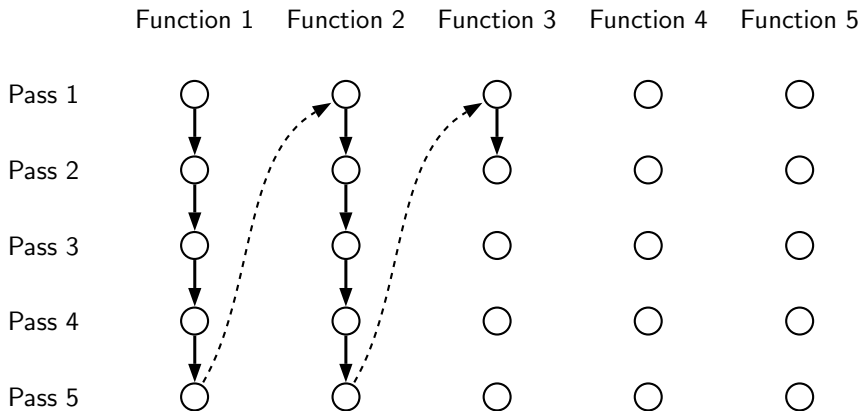


## Execution Order in Intraprocedural Passes

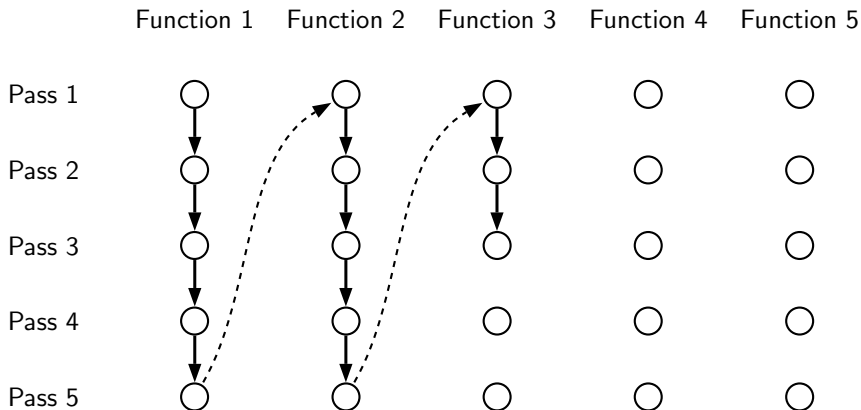




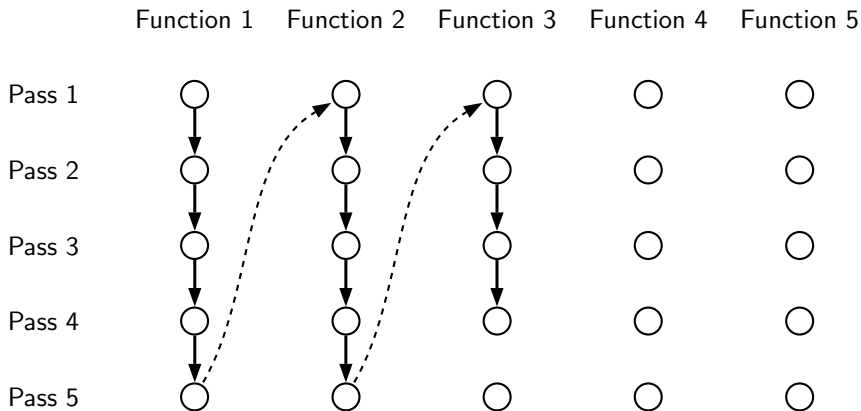
## Execution Order in Intraprocedural Passes



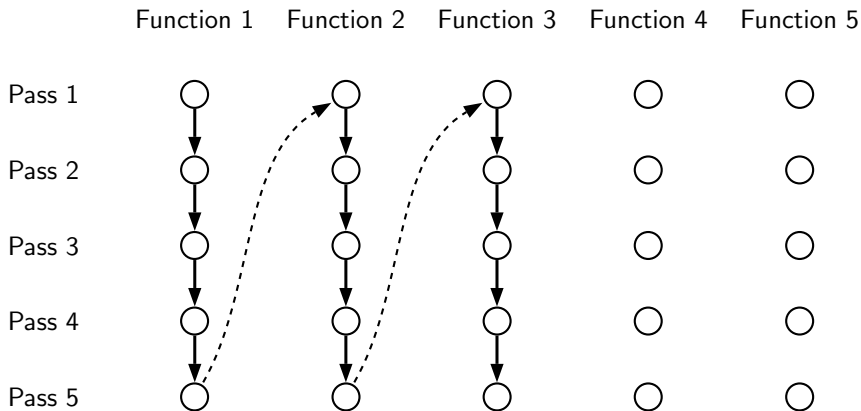
## Execution Order in Intraprocedural Passes



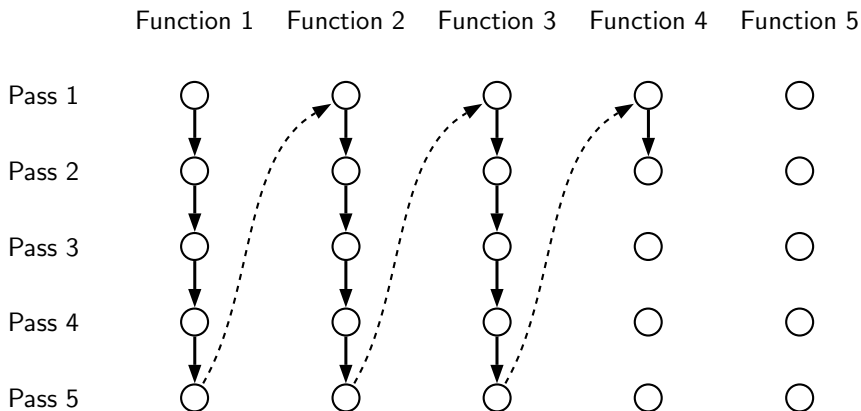
## Execution Order in Intraprocedural Passes



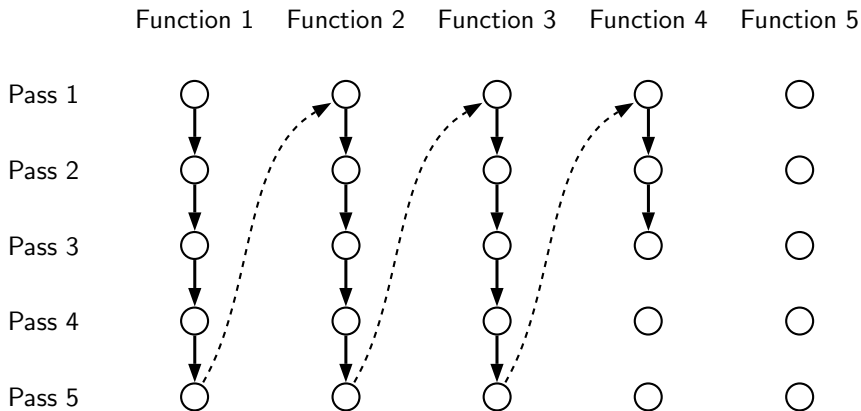
# Execution Order in Intraprocedural Passes



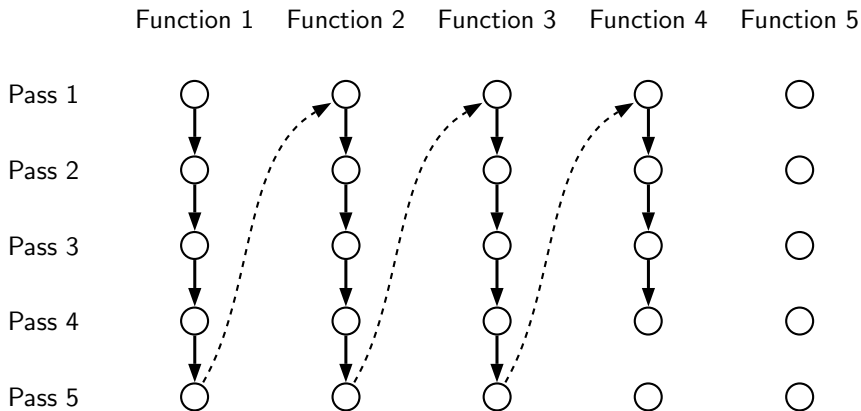
## Execution Order in Intraprocedural Passes



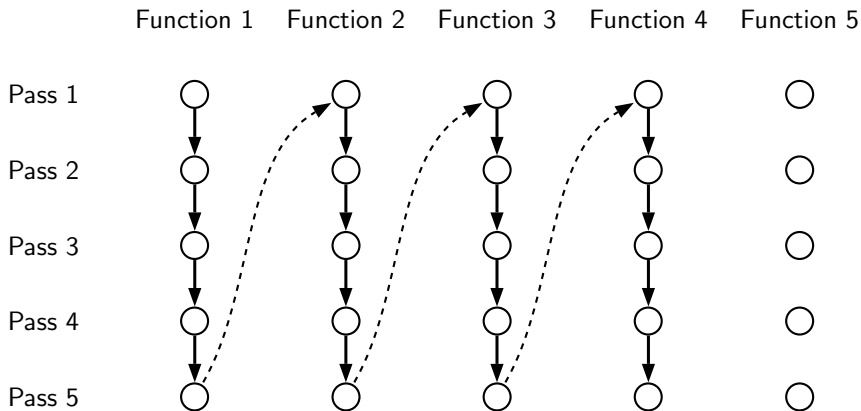
## Execution Order in Intraprocedural Passes



## Execution Order in Intraprocedural Passes

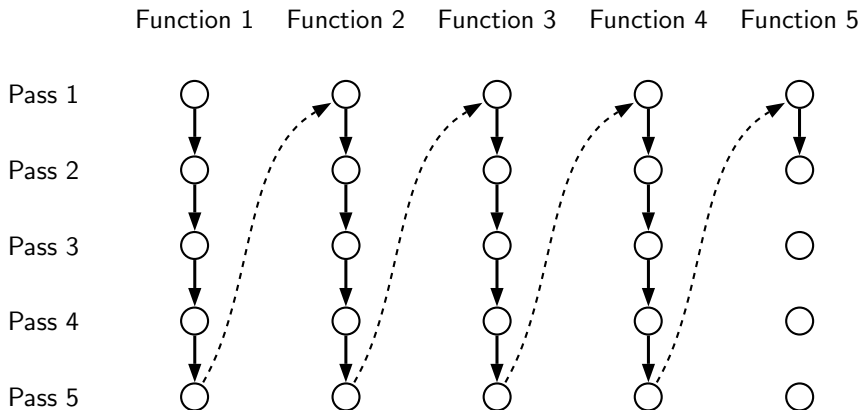


# Execution Order in Intraprocedural Passes

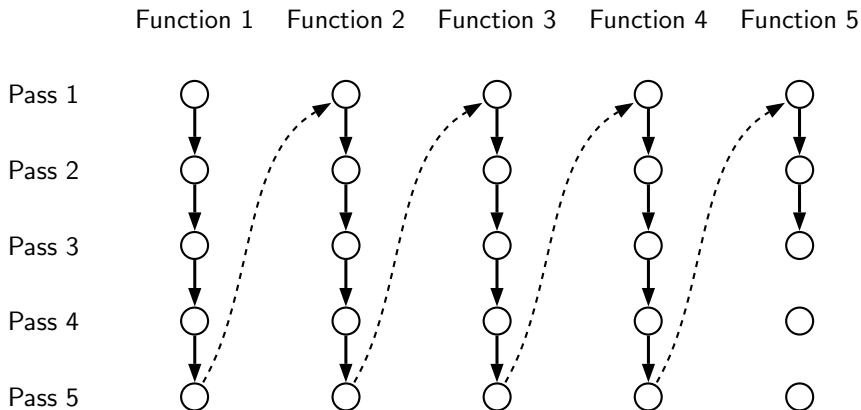




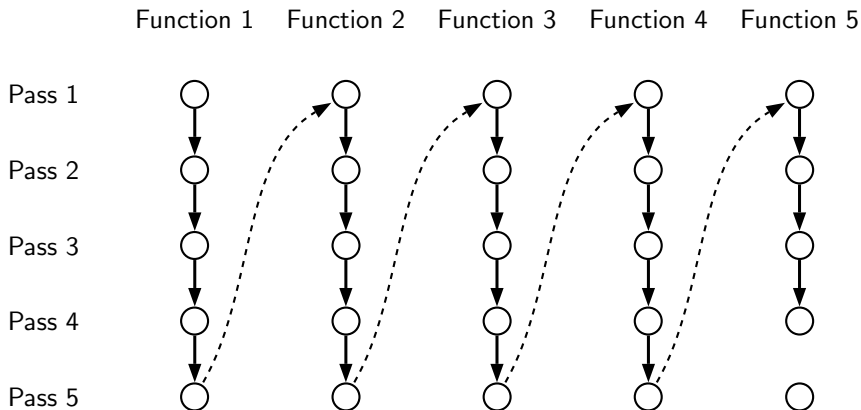
## Execution Order in Intraprocedural Passes



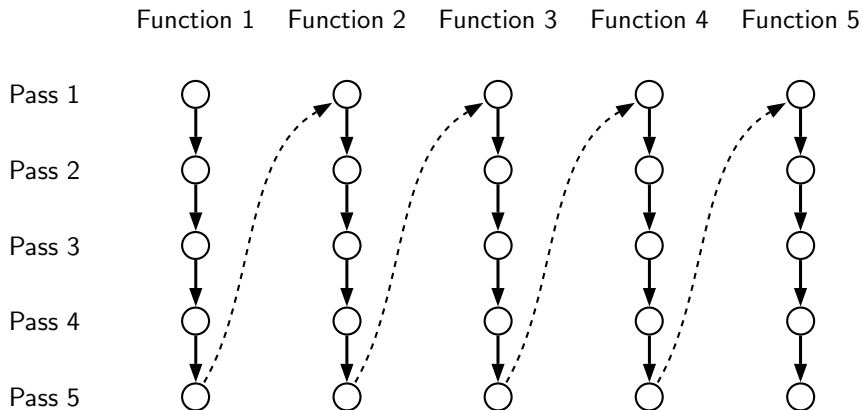
## Execution Order in Intraprocedural Passes



# Execution Order in Intraprocedural Passes



# Execution Order in Intraprocedural Passes

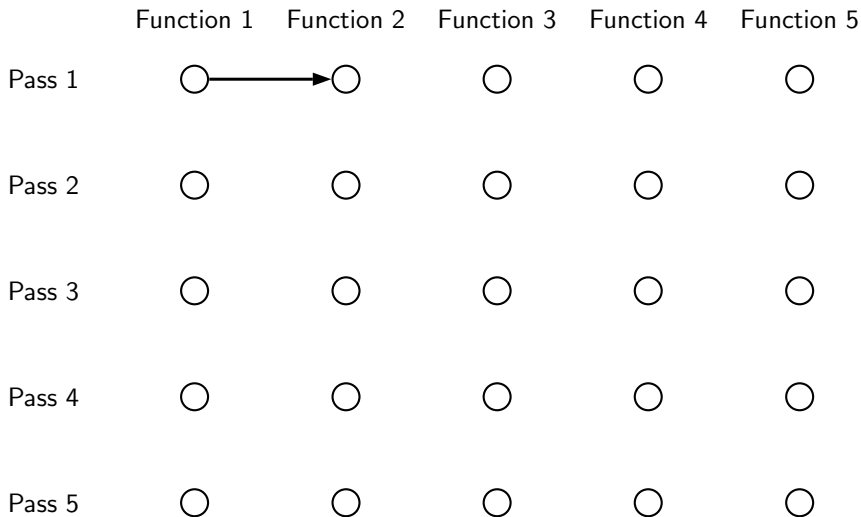


## Execution Order in Interprocedural Passes

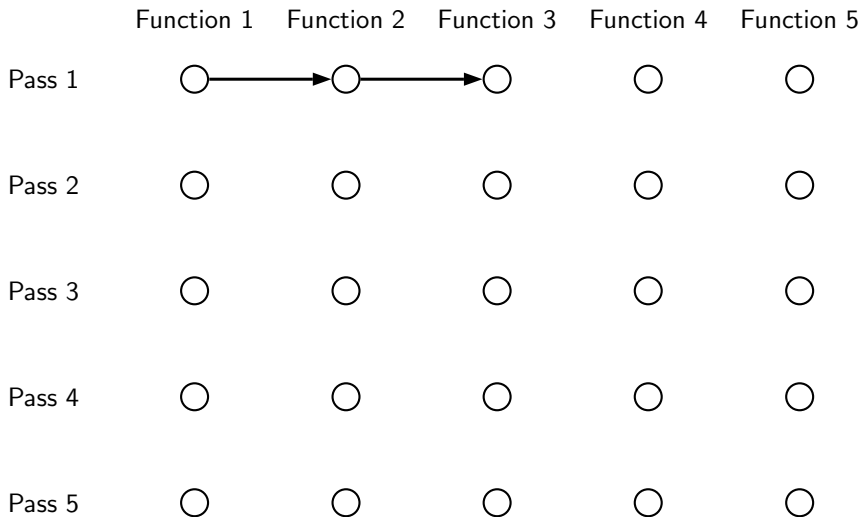
	Function 1	Function 2	Function 3	Function 4	Function 5
Pass 1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 3	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 4	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pass 5	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>



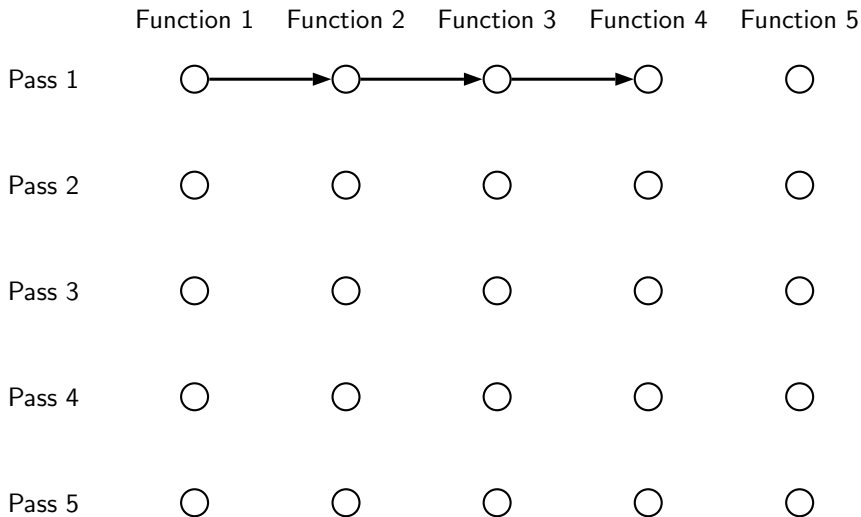
## Execution Order in Interprocedural Passes



## Execution Order in Interprocedural Passes

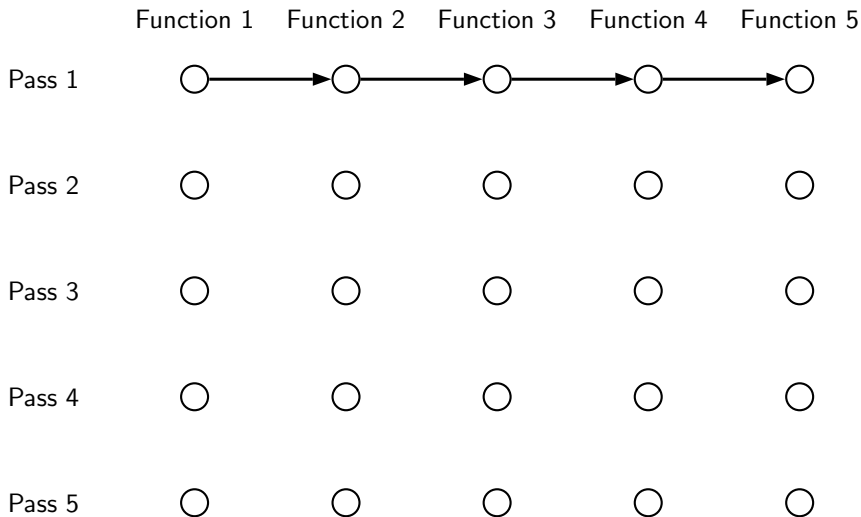


## Execution Order in Interprocedural Passes

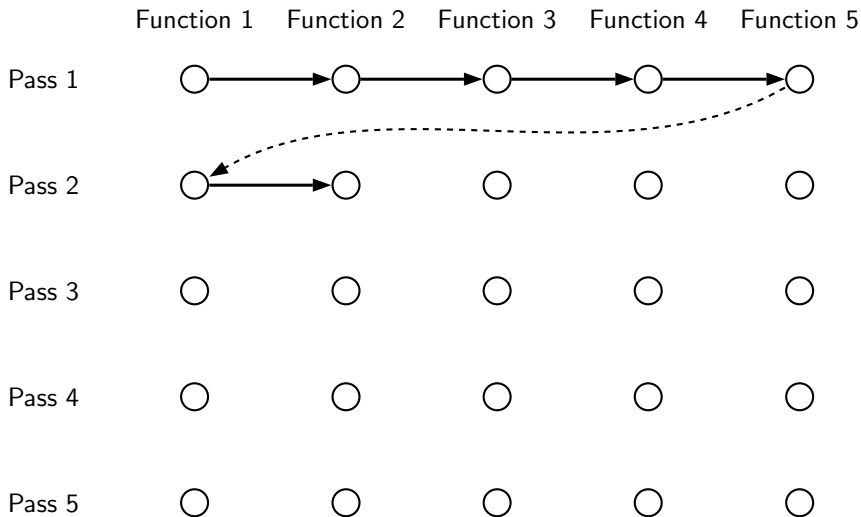




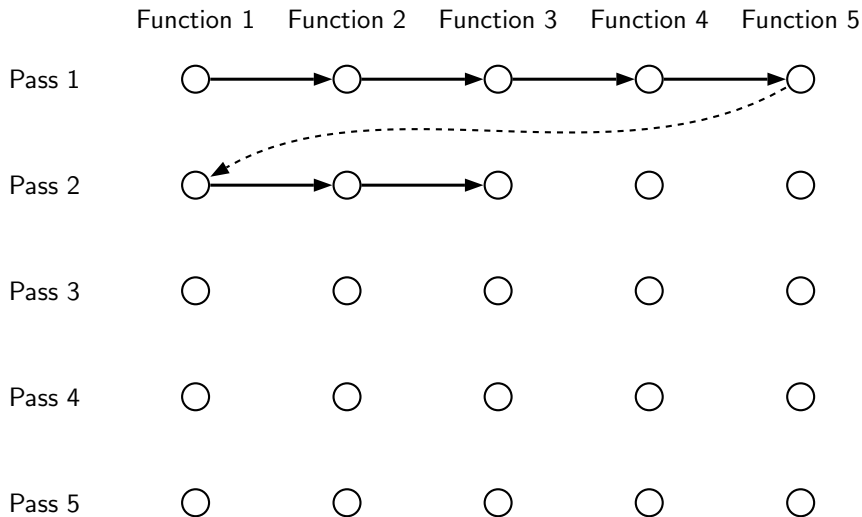
## Execution Order in Interprocedural Passes



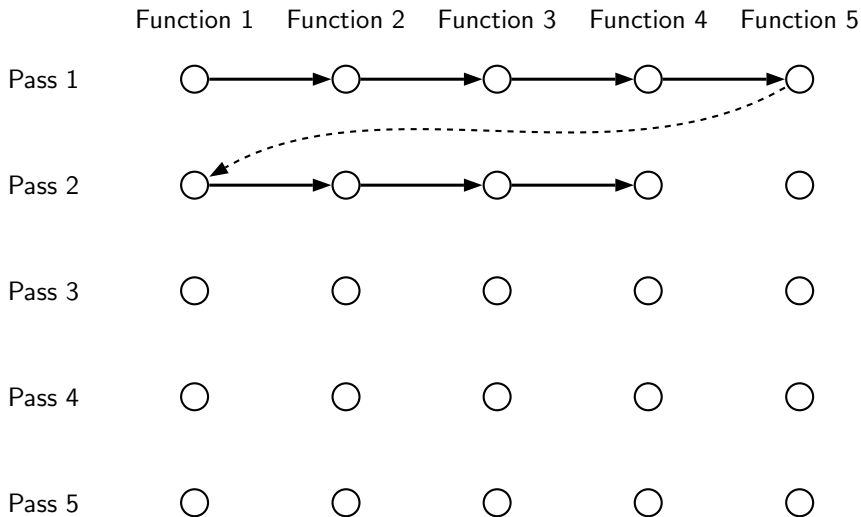
## Execution Order in Interprocedural Passes



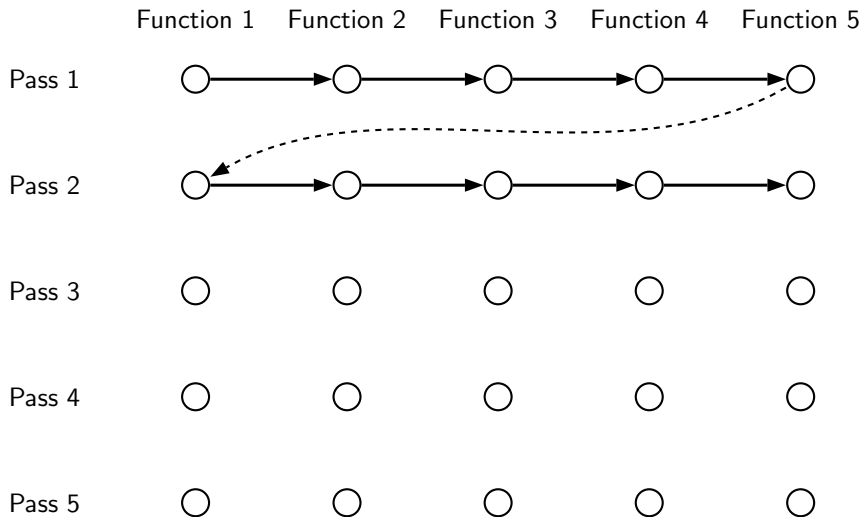
## Execution Order in Interprocedural Passes



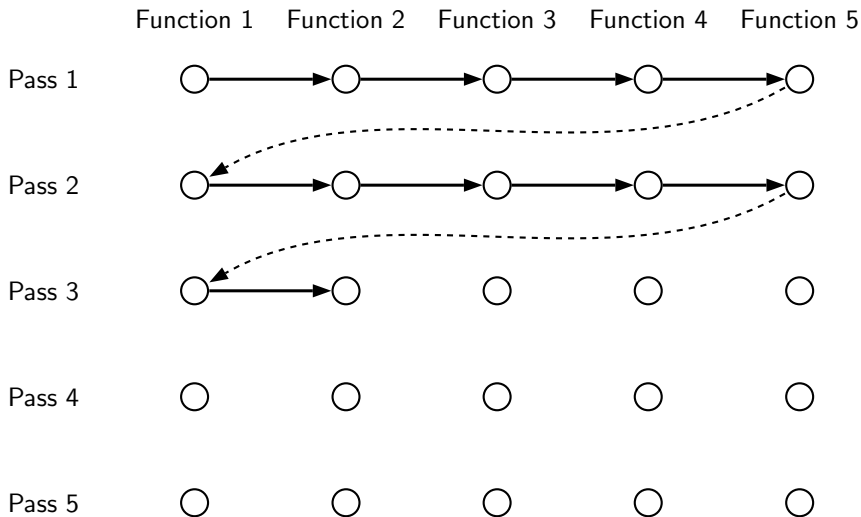
## Execution Order in Interprocedural Passes



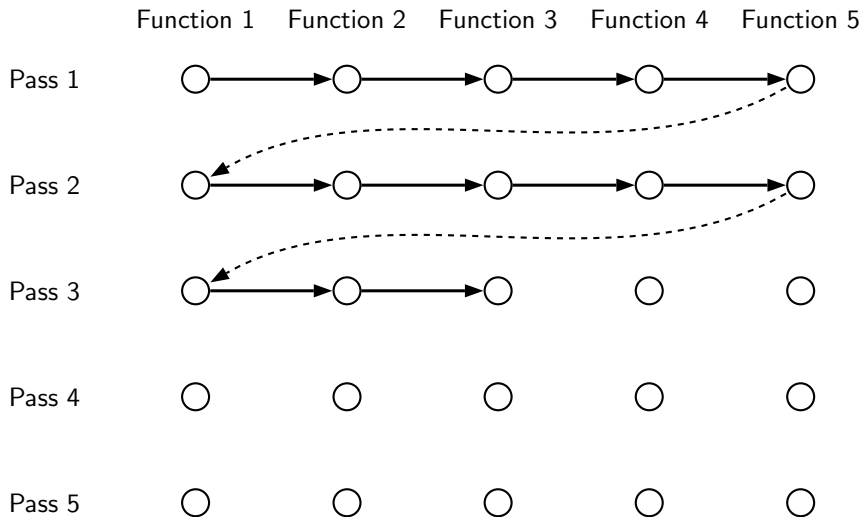
## Execution Order in Interprocedural Passes



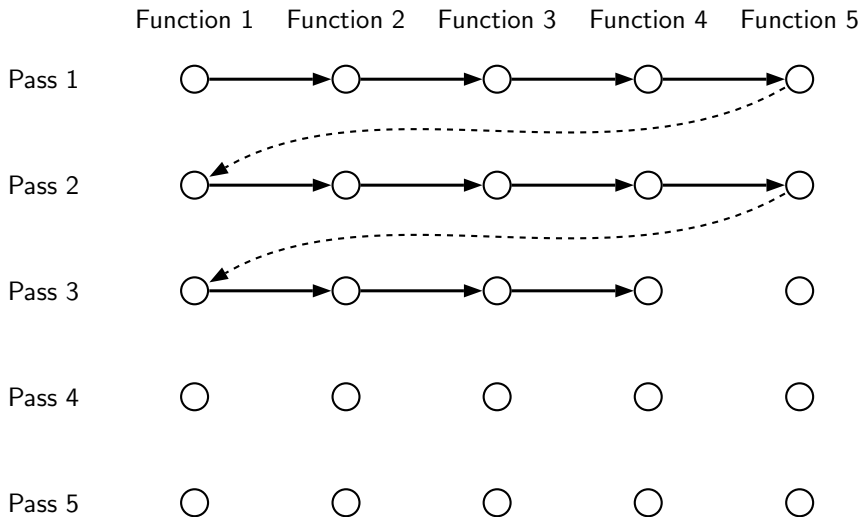
## Execution Order in Interprocedural Passes



## Execution Order in Interprocedural Passes

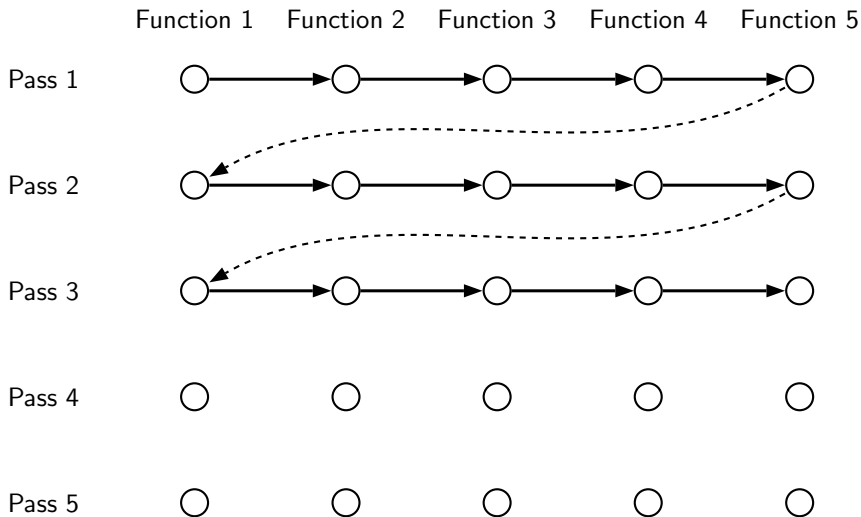


## Execution Order in Interprocedural Passes

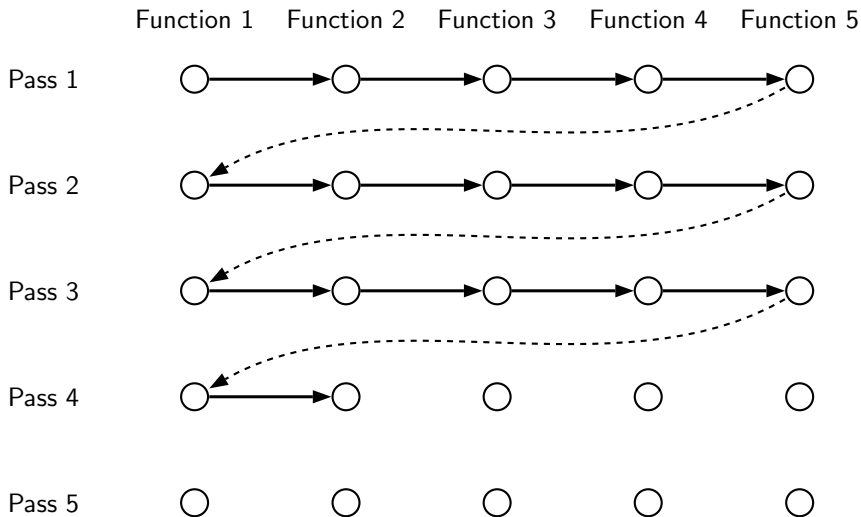




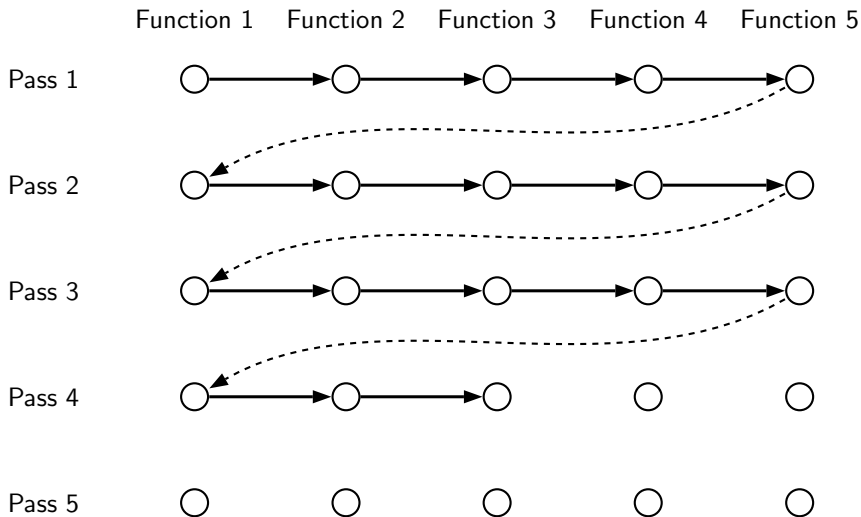
## Execution Order in Interprocedural Passes



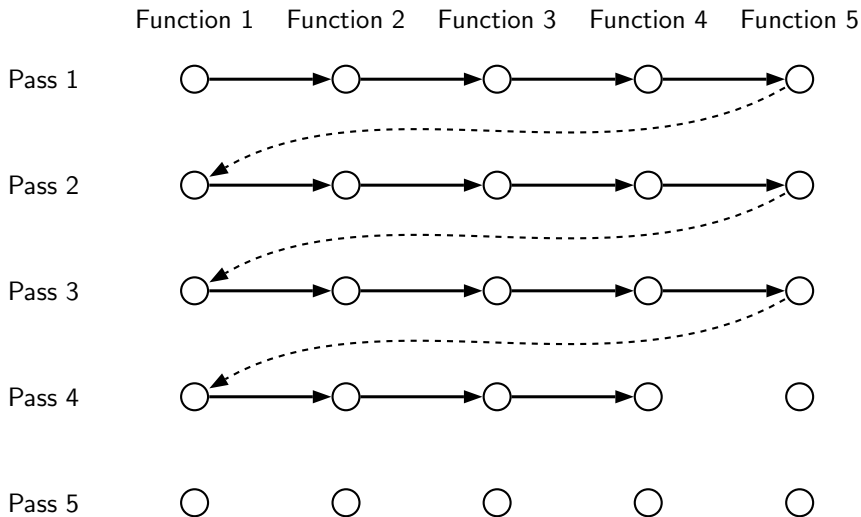
## Execution Order in Interprocedural Passes



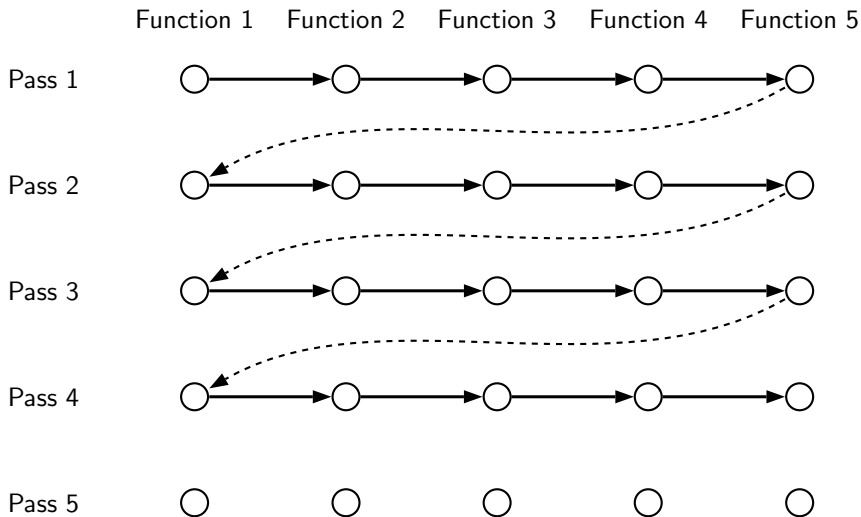
## Execution Order in Interprocedural Passes



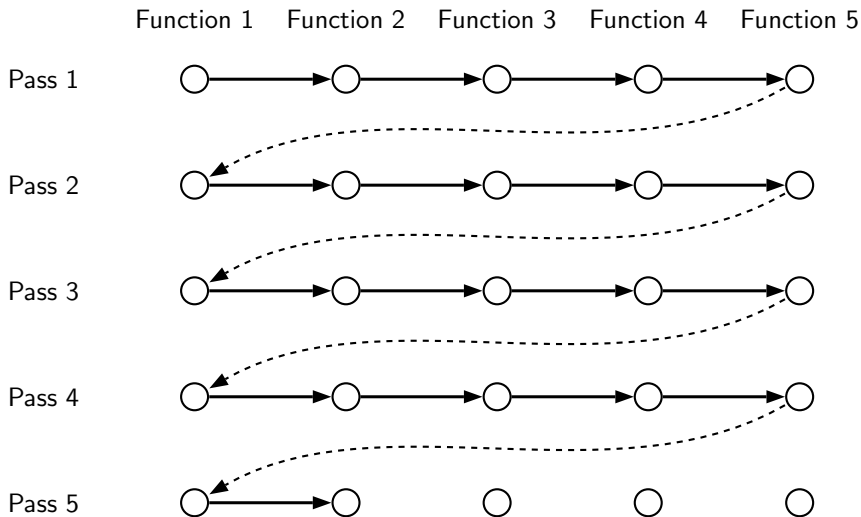
## Execution Order in Interprocedural Passes



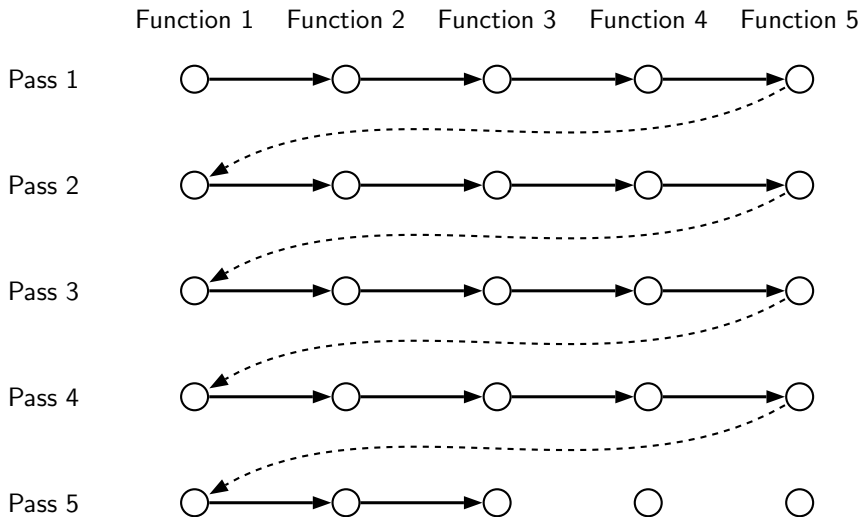
## Execution Order in Interprocedural Passes



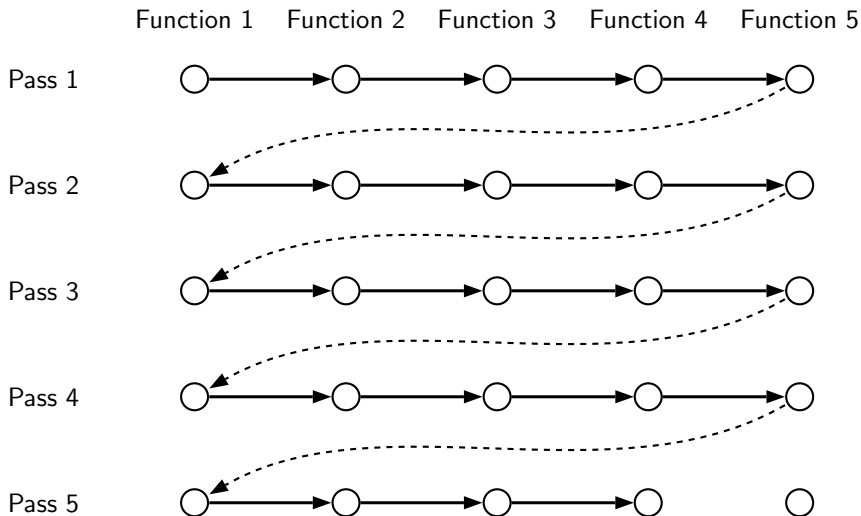
## Execution Order in Interprocedural Passes



## Execution Order in Interprocedural Passes

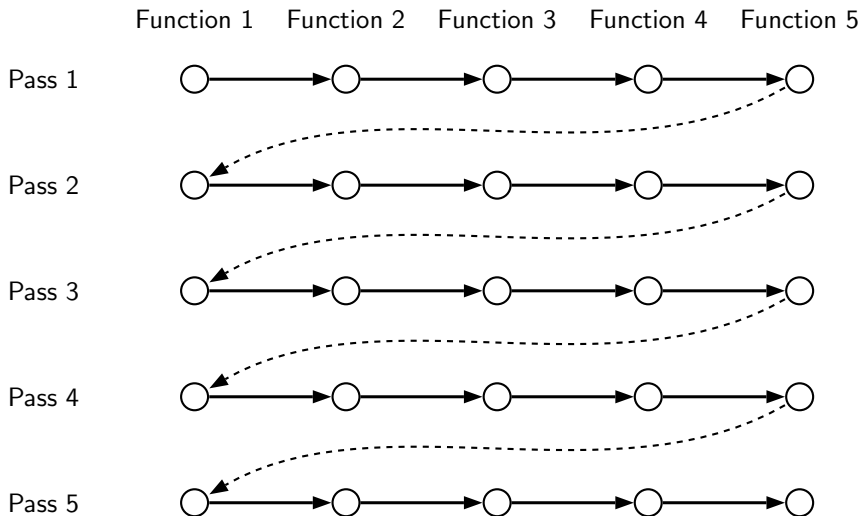


## Execution Order in Interprocedural Passes





## Execution Order in Interprocedural Passes



## cc1 Control Flow: GIMPLE to RTL Expansion (pass\_expand)

```
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
```



*Part 5*

# *Link Time Optimization*

## 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*



# Understanding LTO Framework

```
main ()
{
    printf ("hello, world\n");
}
```



## Assembly Output without LTO Information (1)

```
.file "t0.c"
.section .rodata
.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
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
.ident "GCC: (GNU) 4.6.0"
.section .note.GNU-stack,"",@pro
```



## Assembly Output with LTO Information (2)

```
.ascii "\007"
.text
.section .gnu.lto_.refs.6a5c5521,"",@progbits
.string "x\234cb' '\006b&\006\030"
.string ""
.string ""
.string "t"
.ascii "\b"
.text
.section .gnu.lto_.statics.6a5c5521,"",@progbits
.string "x\234cb' '\b\300\016@\342\214\020&"
.string ""
.string "\330"
.ascii "\b"
.text
.section .gnu.lto_.decls.6a5c5521,"",@progbits
.string "x\234\225R=0\002A\020}\273w\352\236\247(Q/\!\026\!\!F-\214\
.ascii "\021A\360\003\254\355\314jG\207\263w\007\334E\2058\311\333
.ascii "\331\371|s\307\341I\206\320&\251s'\226t\272\260\210\236({\
.ascii "\260\213\237\242\336\207\b{\204}B\222p@\320}\277F8\3
```





## Assembly Output with LTO Information (3)

```
.ascii "\342\312)\254G\204\323j\307\035\207[w\230qN\204\032gB2\33
.ascii "\025\304$\033\365U\241\f\341\033\314\255a\225\376\237#Y\t\
.ascii "&|}\215\273\276\245{\342\255\374n\f\035b\332\213\236/#\221
.ascii "\321\253.Y\021q/ \320\310\0166\322\303\305\275^\357L\373\3
.ascii "\017'f\005\227D\267\3400\333\365Z\325_8h\217j\367f-\034j\3
.ascii "!r\237y[\f\344\231x\302\034\335\222\301{\343\317@\204\371\
.ascii "\\ \211u}p\324\351\252\201\307\213^\262\027\3757S\311j0\257
.ascii "\277\302$[\325\006\r\247\275\0207\376\nLu\246\221\254\n+\3
.ascii "\007\367\251\3001\251\244h\003\223\216\350\354\254\016\343
.ascii "\033M\210\356\242\272\211\375\352\005\314\2201F\215\2320\3
.ascii "zx\236t0f\334\237\273\201\350\255\356}\334\017\376F\344\20
.ascii "v\222\366\006\206\316V\226S\320S\351\243\323\221\354q6{\23
.ascii "| \003\262q\030\362"
.text
.section .gnu.lto_ .symtab.6a5c5521, "", @progbits
.string "main"
.string ""
.string ""
.string ""
.string ""
```



## Assembly Output with LTO Information (4)

```
.string ""
.string ""
.string ""
.string ""
.string ""
.string ""
.string ""
.string "K"
.string ""
.string ""
.text
.section .gnu.lto_.opts,"",@progbits
.string "x\234cb'\340\002bs\006\b'\002\021\r\f\f\273\230\031\030\
.ascii "\002\370\tL"
.text
.section .rodata
.LC0:
.string "hello, world"
```



## 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
```



## Assembly Output with LTO Information (6)

```
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
```



# Single Process and Multi Process LTO

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
  - ▶ WPA: Whole Program Analysis
    - Reads the call graph and not function bodies
    - Summary information for each function
  - ▶ LTRANS: Local Transformations



# Multi Process LTO (aka WHOPR Mode of LTO)

- Three steps
  - ▶ LGEN: Local generation of summary information and translation unit information **Potentially Parallel**
  - ▶ WPA: Whole Program Analysis
    - Reads the call graph and not function bodies
    - Summary information for each function
  - ▶ LTRANS: Local Transformations





# Multi Process LTO (aka WHOPR Mode of LTO)

- Three steps
  - ▶ LGEN: Local generation of summary information and translation unit information **Potentially Parallel**
  - ▶ WPA: Whole Program Analysis **Sequential**
    - Reads the call graph and not function bodies
    - Summary information for each function
  - ▶ LTRANS: Local Transformations



# Multi Process LTO (aka WHOPR Mode of LTO)

- Three steps
  - ▶ LGEN: Local generation of summary information and translation unit 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**



## Multi Process LTO (aka WHOPR Mode of LTO)

- Three steps
  - ▶ LGEN: Local generation of summary information and translation unit 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**
- Why do we call this LTO *Multi Process* LTO?



## Multi Process LTO (aka WHOPR Mode of LTO)

- Three steps
  - ▶ LGEN: Local generation of summary information and translation unit 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**
- Why do we call this LTO *Multi Process* LTO?
  - ▶ gcc executes LGEN
  - ▶ Subsequent process of lto1 executes WPA
  - ▶ Subsequent independent processes of lto1 execute LTRANS



# Single Process LTO

- Three steps
  - ▶ LGEN: Local Generation of translation unit information (no summary)
  - ▶ IPA: Inter-Procedural Analysis
    - Reads the call graph and function bodies
  - ▶ LTRANS: Local Transformations



# Single Process LTO

- Three steps
  - ▶ LGEN: Local Generation of translation unit information (no summary) **Potentially Parallel**
  - ▶ IPA: Inter-Procedural Analysis
    - Reads the call graph and function bodies
  - ▶ LTRANS: Local Transformations



# Single Process LTO

- Three steps
  - ▶ LGEN: Local Generation of translation unit information (no summary) **Potentially Parallel**
  - ▶ IPA: Inter-Procedural Analysis **Sequential**
    - Reads the call graph and function bodies
  - ▶ LTRANS: Local Transformations



# Single Process LTO

- Three steps
  - ▶ LGEN: Local Generation of translation unit information (no summary) **Potentially Parallel**
  - ▶ IPA: Inter-Procedural Analysis **Sequential**
    - Reads the call graph and function bodies
  - ▶ LTRANS: Local Transformations **Sequential**





## Single Process LTO

- Three steps
  - ▶ LGEN: Local Generation of translation unit information (no summary) **Potentially Parallel**
  - ▶ IPA: Inter-Procedural Analysis **Sequential**
    - Reads the call graph and function bodies
  - ▶ LTRANS: Local Transformations **Sequential**
- Why do we call this LTO *Single Process* LTO?



## Single Process LTO

- Three steps
  - ▶ LGEN: Local Generation of translation unit information (no summary) **Potentially Parallel**
  - ▶ IPA: Inter-Procedural Analysis **Sequential**
    - Reads the call graph and function bodies
  - ▶ LTRANS: Local Transformations **Sequential**
- Why do we call this LTO *Single Process* LTO?
  - ▶ gcc executes LGEN
  - ▶ Subsequent process of lto1 executes both IPA and LTRANS



# Single Process LTO

- Three steps
  - ▶ LGEN: Local Generation of translation unit information (no summary) **Potentially Parallel**
  - ▶ IPA: Inter-Procedural Analysis **Sequential**
    - Reads the call graph and function bodies
  - ▶ LTRANS: Local Transformations **Sequential**
- Why do we call this LTO *Single Process* LTO?
  - ▶ gcc executes LGEN
  - ▶ Subsequent process of lto1 executes both IPA and LTRANS
- When `-flto-partition=none`, IPA = WPA



## LTO Pass Hooks

```
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 *);
};
```



## LTO Pass Hooks

```
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 *);
};
```

LGEM for Multi Process LTO



## LTO Pass Hooks

```
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 *);
};
```

LGEM for Single Process LTO



## LTO Pass Hooks

```
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 *);
  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 *);
};
```

WPA for Multi Process LTO



## LTO Pass Hooks

```
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 *);
  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 *);
};
```

IPA for Single Process LTO





## LTO Pass Hooks

```
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 *);
};
```

LTRANS for Multi Process LTO



## LTO Pass Hooks

```
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 *);
};
```

LTRANS for Single Process LTO



## LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies			
	Call graph with function bodies			



## LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported		
	Call graph with function bodies			



## LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0	
	Call graph with function bodies			



## LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0	Will be supported in future
	Call graph with function bodies			



## LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0	Will be supported in future
	Call graph with function bodies	Supported in GCC-4.6.0		



## LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0	Will be supported in future
	Call graph with function bodies	Supported in GCC-4.6.0	Not supported	Not supported





## LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0	Will be supported in future
	Call graph with function bodies	Supported in GCC-4.6.0	Not supported	Not supported

-flto



# LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0	Will be supported in future
	Call graph with function bodies	Supported in GCC-4.6.0	Not supported	Not supported

-flto

WHOPR mode



# LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0	Will be supported in future
	Call graph with function bodies	Supported in GCC-4.6.0	Not supported	Not supported

`-flto`

`-flto -flto-partition=none`

WHOPR mode



# LTO Support in GCC

		Transformation		
		In the same process as that of analysis	In an independent process (possibly multiple processes)	
			Single partition of the program	Single partition of the program
Whole Program Analysis	Call graph without function bodies	Not supported	Supported in GCC-4.6.0 ⇒	Will be supported in future
	Call graph with function bodies	Supported in GCC-4.6.0	Not supported	Not supported

`-flto`

`-flto -flto-partition=none`

WHOPR mode



## lto1 Control Flow

```
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_wpa_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. */
```



## cc1 and Single Process lto1

```
toplev_main
```

```
...
```

```
  compile_file
```

```
  ...
```

```
    cgraph_analyze_function
```

```
  cgraph_optimize
```

```
  ...
```

```
    ipa_passes
```

```
cc1
```

```
  ...
```

```
    cgraph_expand_all_functions
```

```
  ...
```

```
    tree_rest_of_compilation
```



## cc1 and Single Process lto1

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

```
lto_main
...
  read_cgraph_and_symbols
  ...
    materialize_cgraph
```

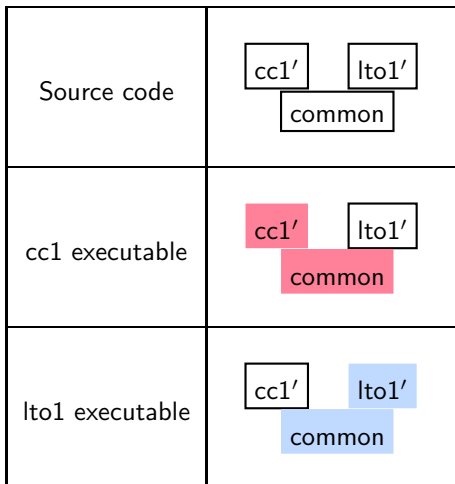
```
cgraph_optimize
...
  ipa_passes
  ...
    cgraph_expand_all_functions
    ...
      tree_rest_of_compilation
```

lto1

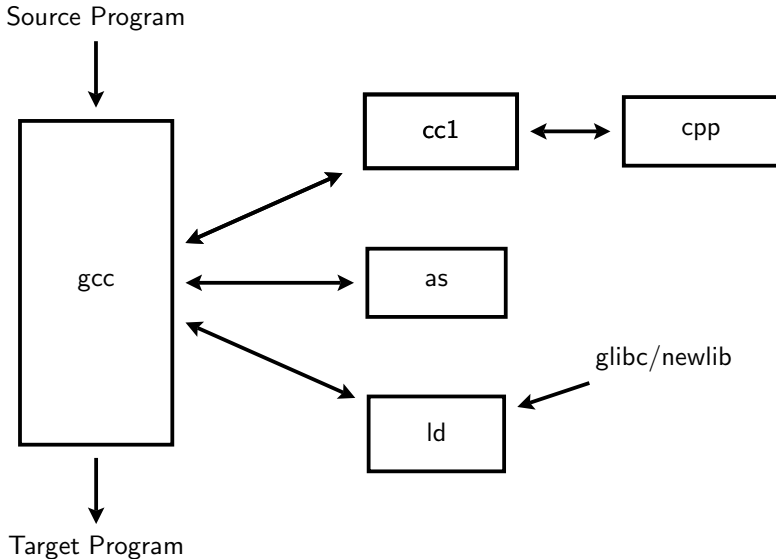




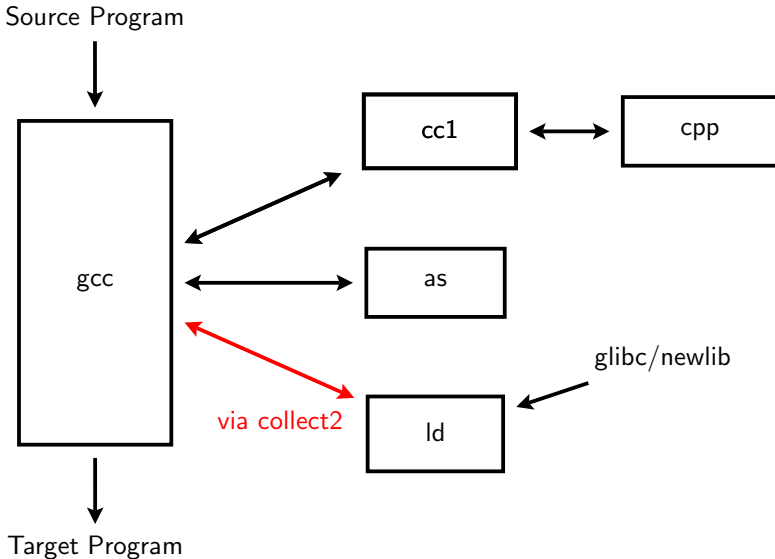
## Our Pictorial Convention



# The GNU Tool Chain: Our First Picture



# The GNU Tool Chain: Our First Picture



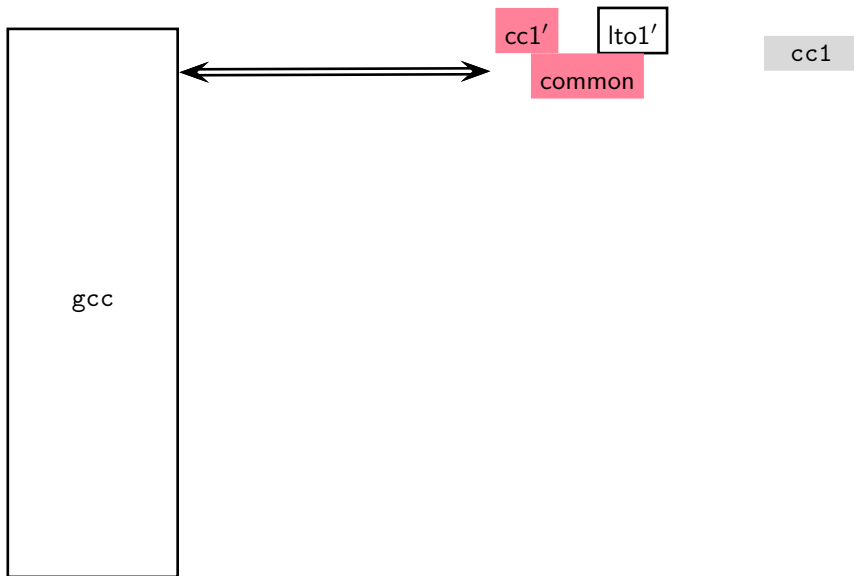
# The GNU Tool Chain for Single Process LTO Support



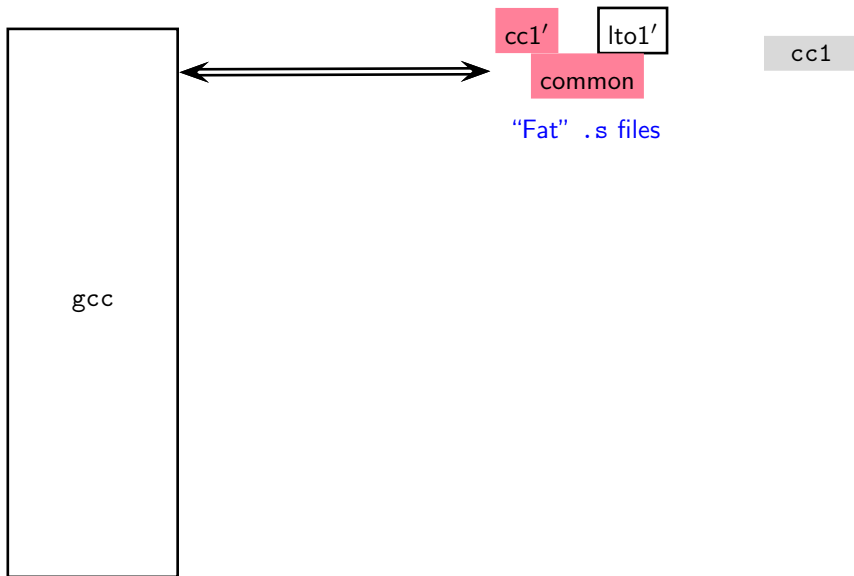
gcc



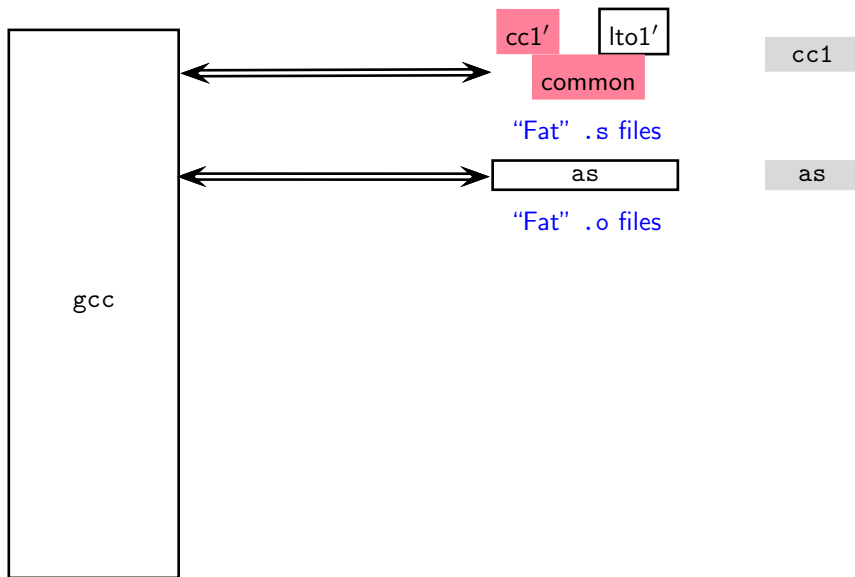
# The GNU Tool Chain for Single Process LTO Support



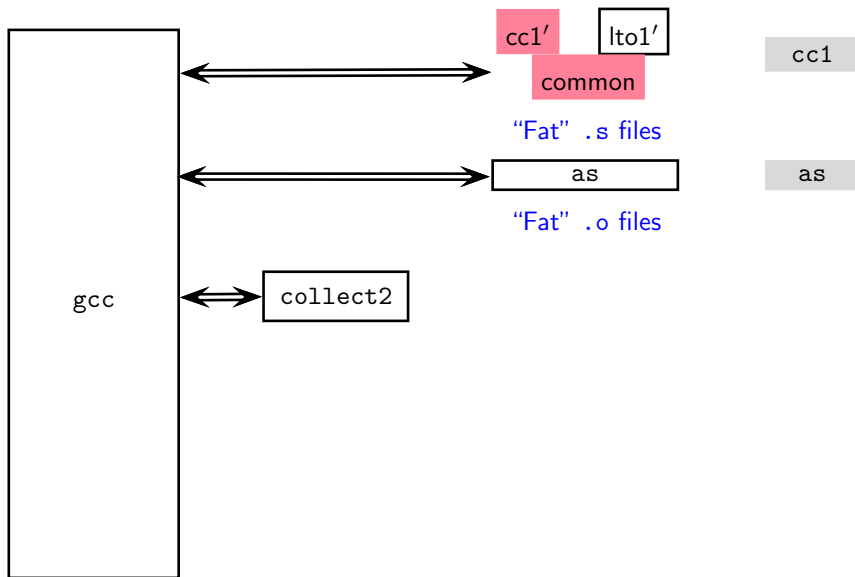
# The GNU Tool Chain for Single Process LTO Support



# The GNU Tool Chain for Single Process LTO Support

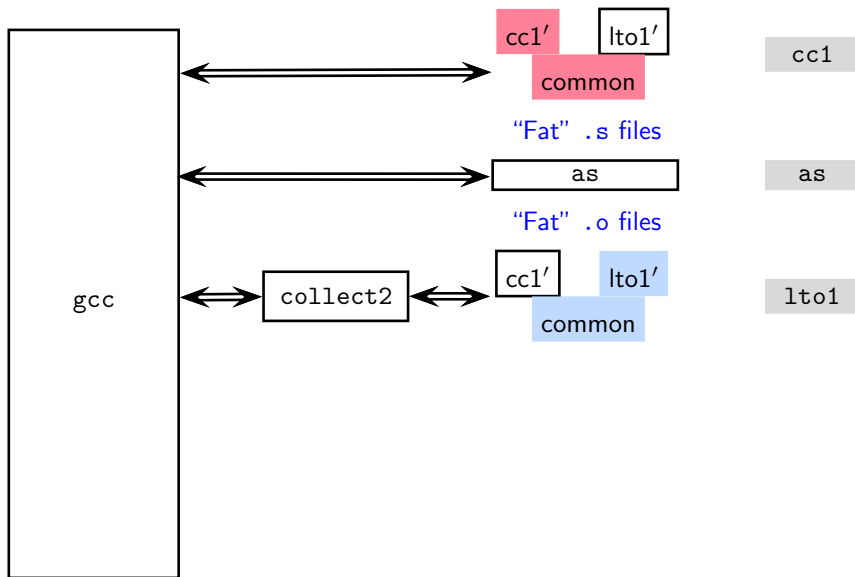


# The GNU Tool Chain for Single Process LTO Support

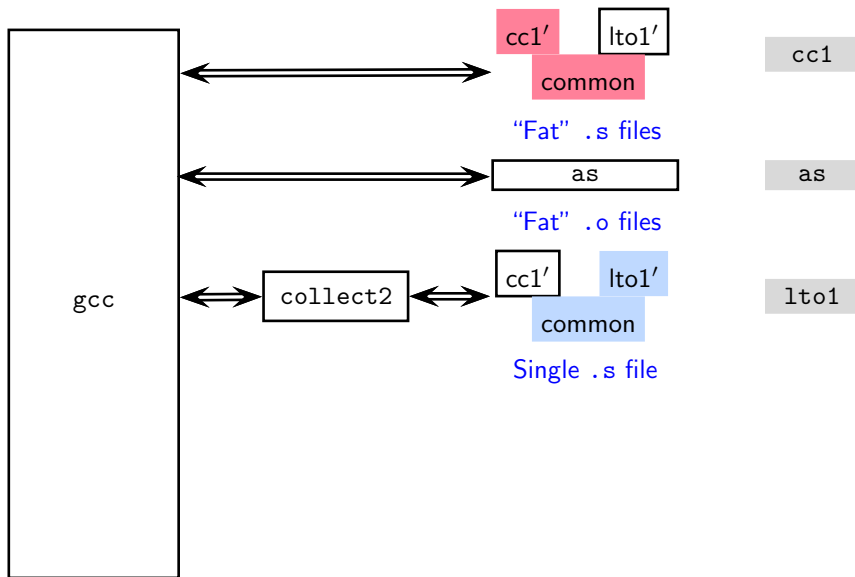




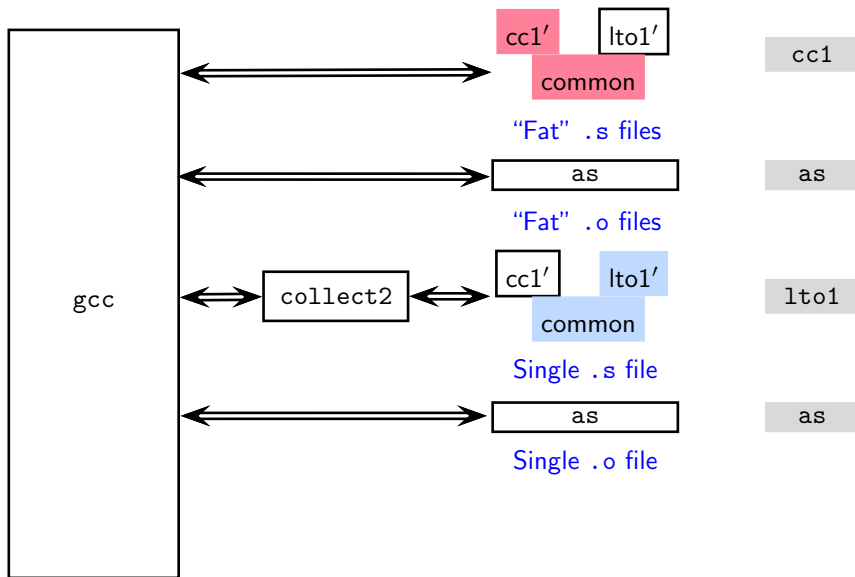
# The GNU Tool Chain for Single Process LTO Support



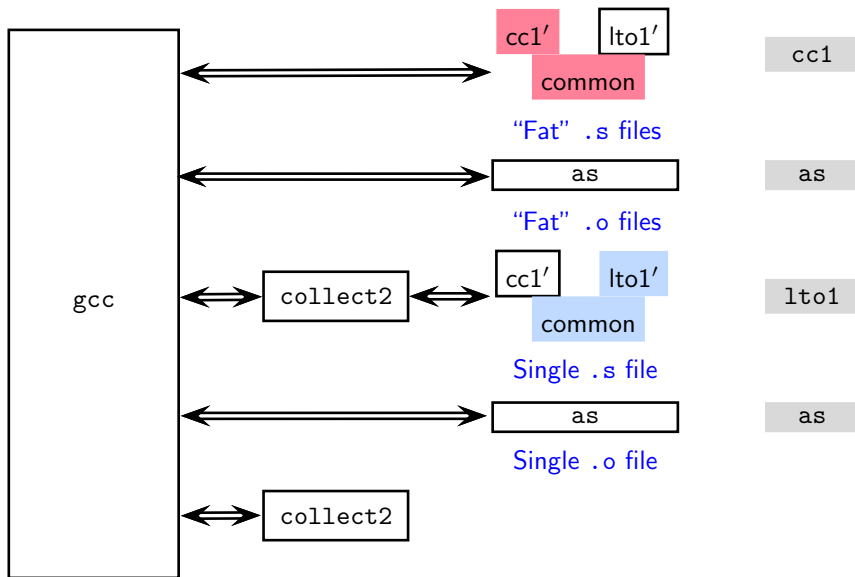
# The GNU Tool Chain for Single Process LTO Support



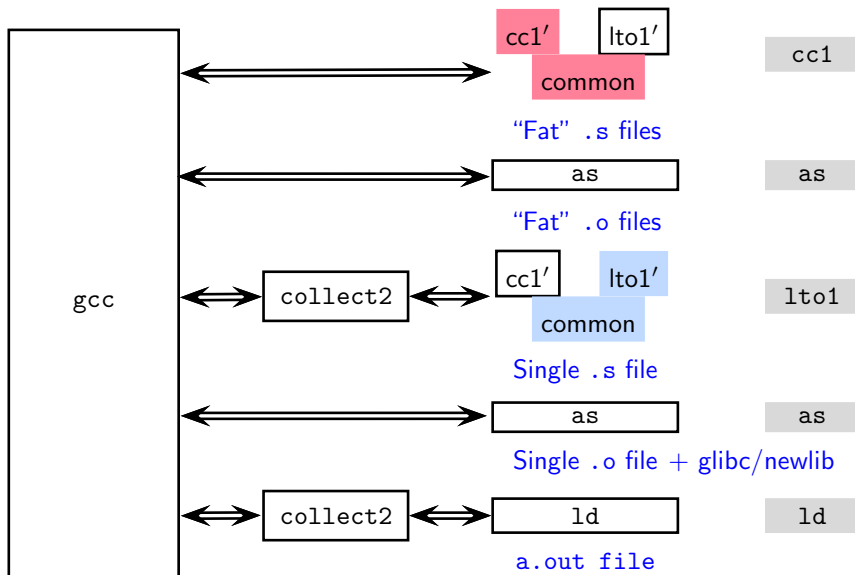
# The GNU Tool Chain for Single Process LTO Support



# The GNU Tool Chain for Single Process LTO Support



# The GNU Tool Chain for Single Process LTO Support



# The GNU Tool Chain for Single Process LTO Support

cc1'

lto1'

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

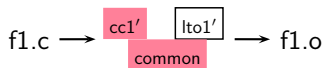
```
cgraph_optimize
  ipa_passes
    execute_ipa_pass_list(all_small_ipa_passes)/*!in lto*/
    execute_ipa_summary_passes(all_regular_ipa_passes)
    execute_ipa_summary_passes(all_lto_gen_passes)
    ipa_write_summaries
  cgraph_expand_all_functions
  cgraph_expand_function
  /* Intraprocedural passes on GIMPLE, */
  /* expansion pass, and passes on RTL. */
```

a.out file



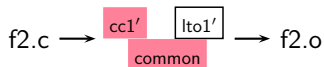
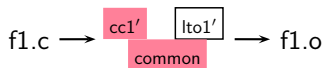
# Multi Process LTO (aka WHOPR LTO)

Option `-flto -c`



# Multi Process LTO (aka WHOPR LTO)

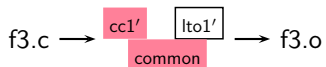
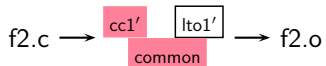
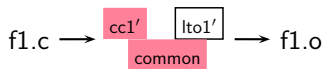
Option `-flto -c`



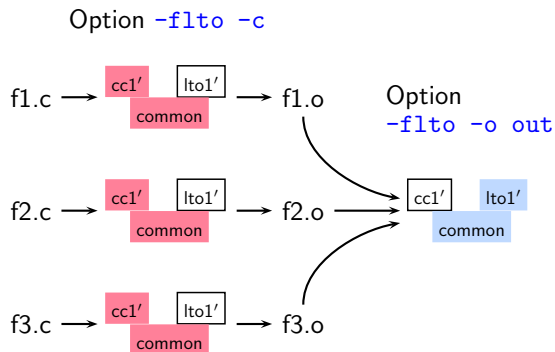


# Multi Process LTO (aka WHOPR LTO)

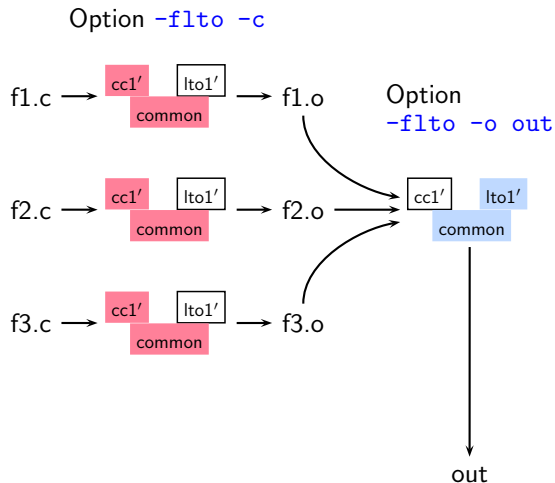
Option `-flto -c`



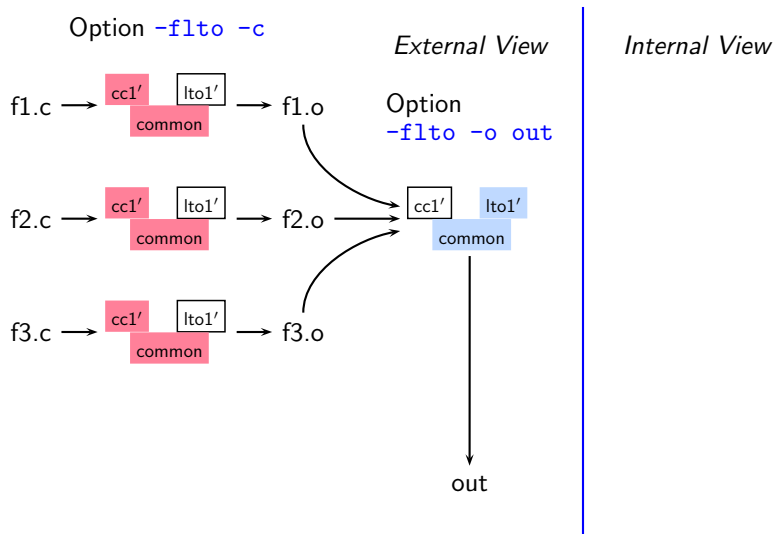
# Multi Process LTO (aka WHOPR LTO)



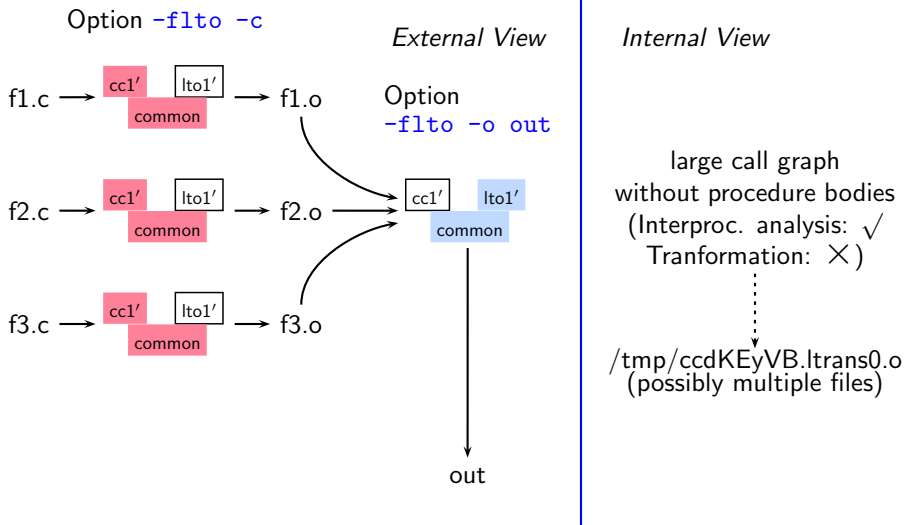
# Multi Process LTO (aka WHOPR LTO)



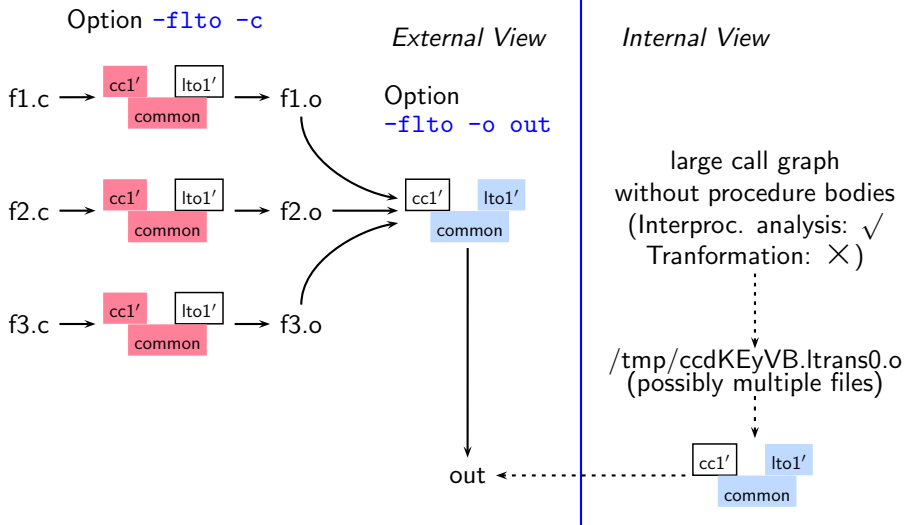
# Multi Process LTO (aka WHOPR LTO)



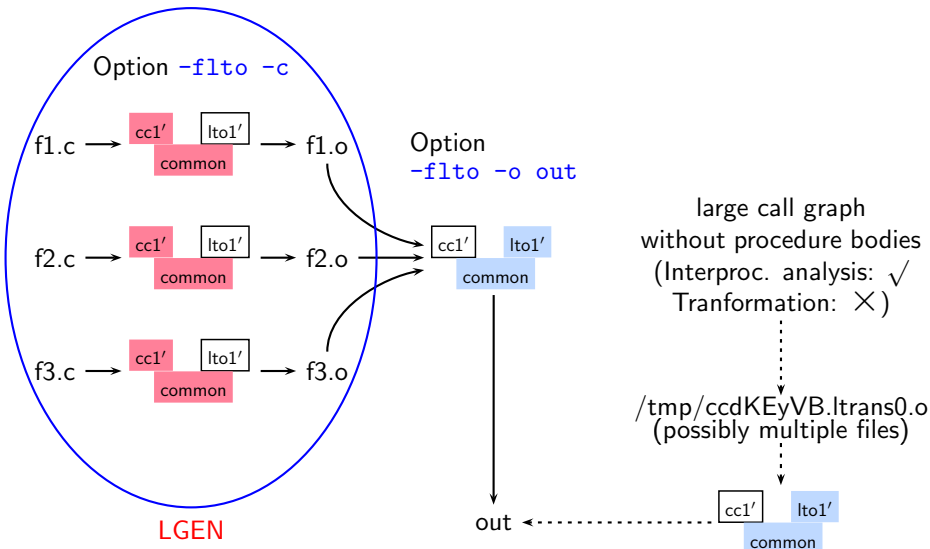
# Multi Process LTO (aka WHOPR LTO)



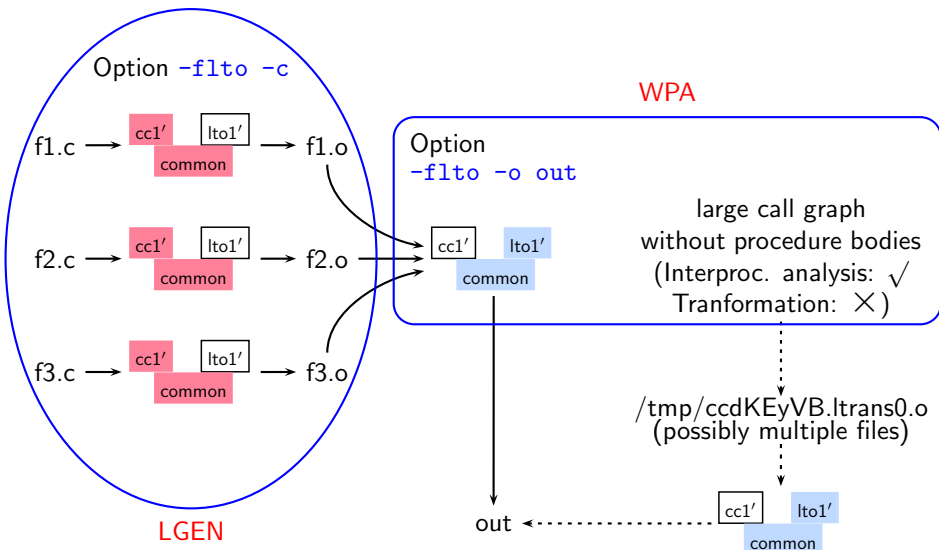
# Multi Process LTO (aka WHOPR LTO)



# Multi Process LTO (aka WHOPR LTO)

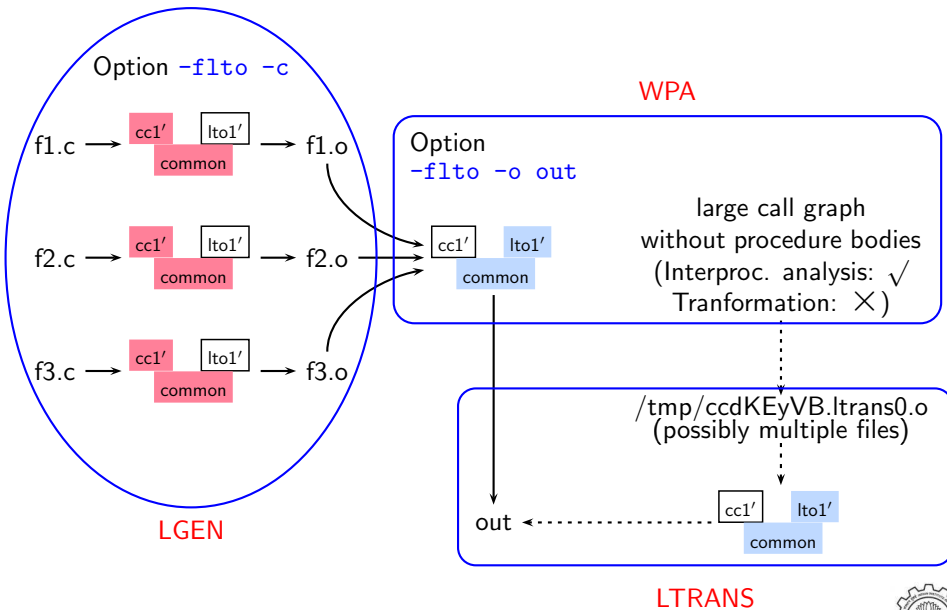


# Multi Process LTO (aka WHOPR LTO)



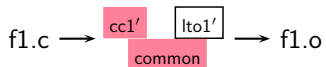


# Multi Process LTO (aka WHOPR LTO)



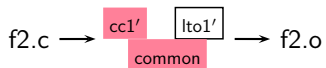
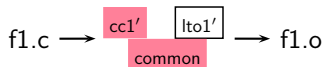
# Single Process LTO

Option `-flto -c`



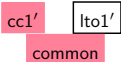
# Single Process LTO

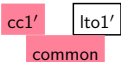
Option `-flto -c`

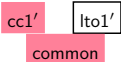


# Single Process LTO

Option `-flto -c`

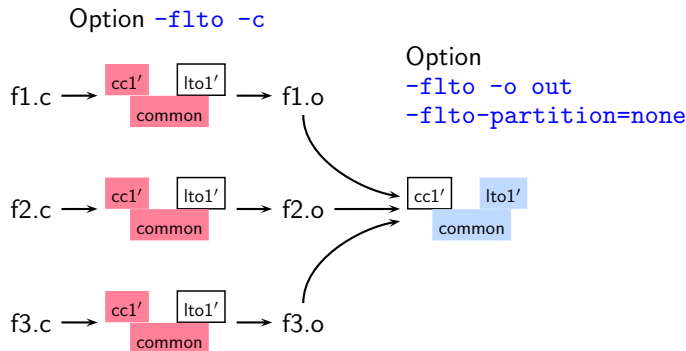
f1.c →  → f1.o

f2.c →  → f2.o

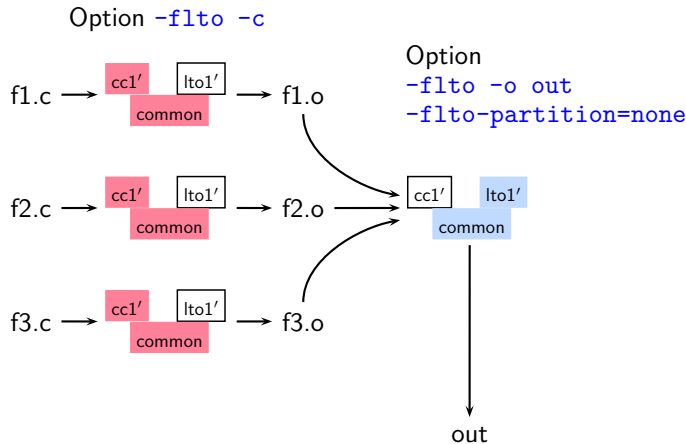
f3.c →  → f3.o



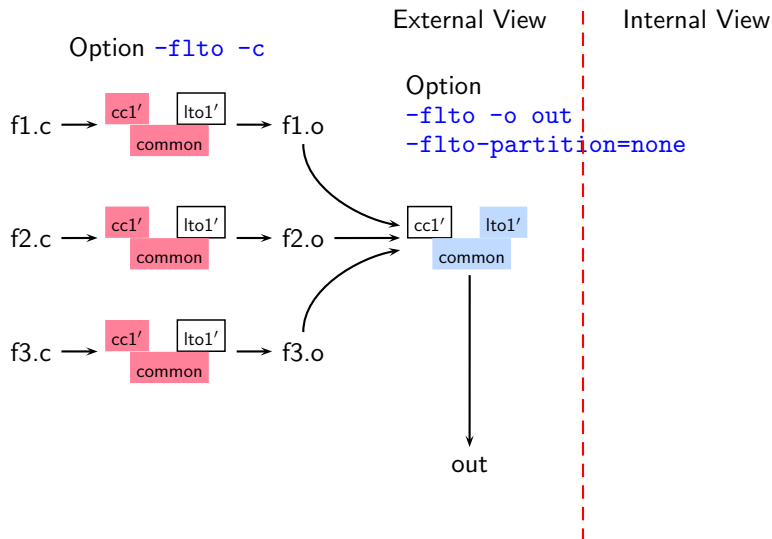
# Single Process LTO



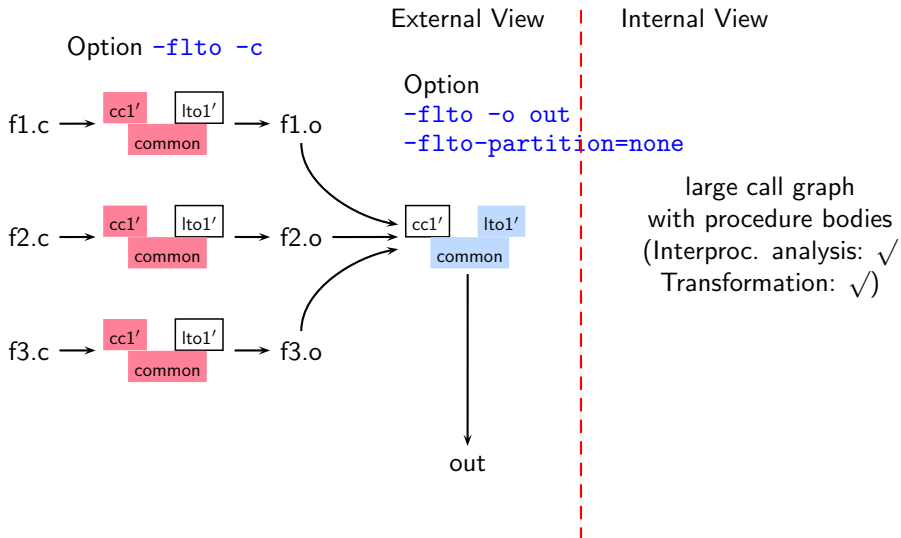
# Single Process LTO



# Single Process LTO

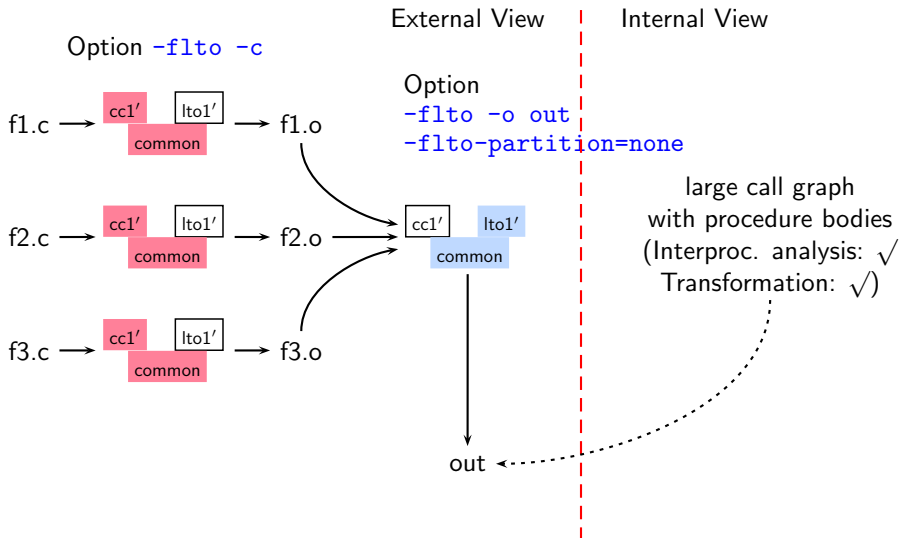


# Single Process LTO

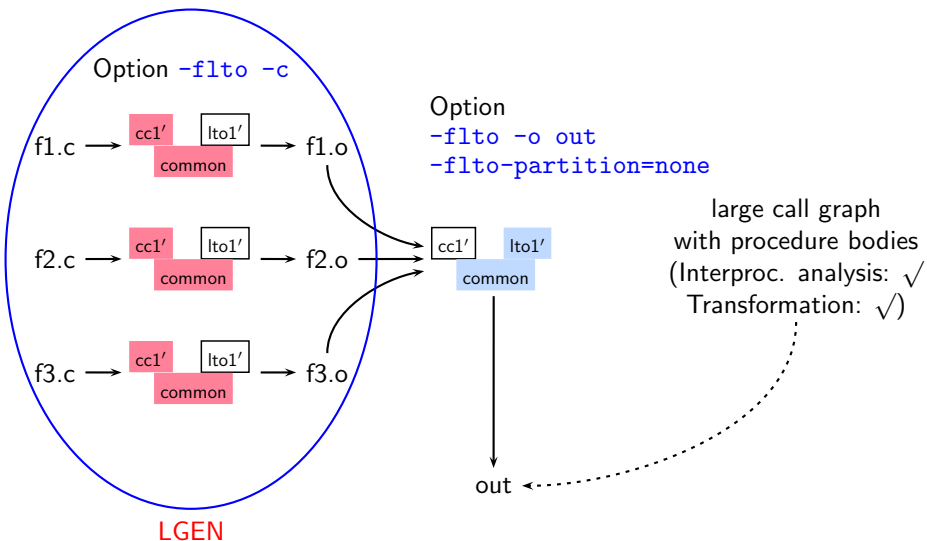




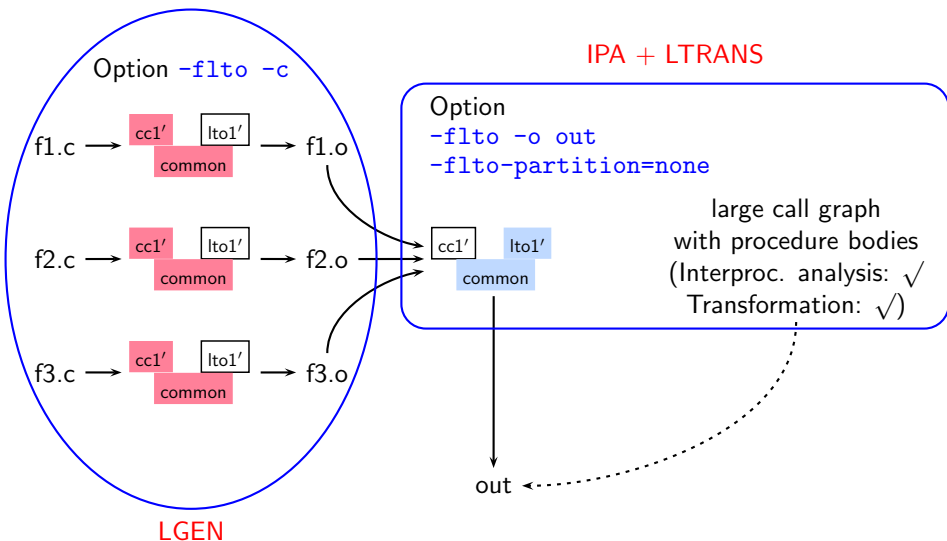
# Single Process LTO



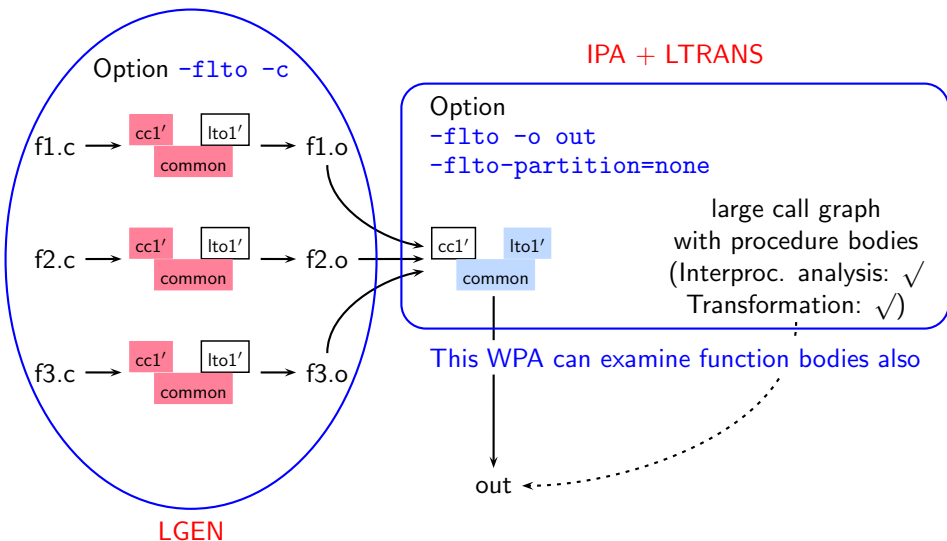
# Single Process LTO



# Single Process LTO



# Single Process LTO



*Part 6*

# *Conclusions*

## 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

