Workshop on Essential Abstractions in GCC

A Summary of Essential Abstractions

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

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

3 July 2012

Workshop Coverage

<table>
<thead>
<tr>
<th>Compiler Specifications</th>
<th>External View</th>
<th>Internal View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler Generator</td>
<td>Machine descriptions</td>
<td>Front end hooks</td>
</tr>
<tr>
<td>Generated Compiler</td>
<td>Configuration and building</td>
<td>Retargetability mechanism</td>
</tr>
<tr>
<td></td>
<td>Gray box probing</td>
<td>Pass structure</td>
</tr>
<tr>
<td></td>
<td>Pass structure and IR</td>
<td>Control flow</td>
</tr>
<tr>
<td></td>
<td>Data Flow Analysis</td>
<td>Static and dynamic plugin mechanisms</td>
</tr>
<tr>
<td></td>
<td>Parallelization, Vectorization</td>
<td></td>
</tr>
</tbody>
</table>

Notes
Compilation Models

Aho Ullman Model

- Front End
- AST
- Optimizer

Aho Ullman: Instruction selection
- over optimized IR using
- cost based tree pattern matching

Davidson Fraser Model

- Front End
- AST
- Expander
- Optimizer

Davidson Fraser: Instruction selection
- over AST using
- structural tree pattern matching
- naive code which is
  - target dependent, and
  - optimized subsequently

Notes

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
The Architecture of GCC

- Input Language
- Compiler Generation Framework
- Language Specific Code
- Language and Machine Independent Generic Code
- Machine Dependent Generator Code
- Machine Descriptions
- Target Name
- Development Time
- Build Time
- Use Time

Configuring GCC

- configure
- config/*
- config.guess
- config.cache
- config.log
- config.status
- config.h
- Makefile
- Makefile.in
- config.h.in
- config.sub
Bootstrapping: The Conventional View

Essential Abstractions in GCC

A Native Build on i386

Notes

Essential Abstractions in GCC

Notes
Build for a Given Machine

This is what actually happens!

- **Generation**
  - Generator sources
    - $(SOURCE_D)/gcc/gen*.c are read and generator executables are created in $(BUILD)/gcc
  - MD files are read by the generator executables and back end source code is generated in $(BUILD)/gcc

- **Compilation**
  - Other source files are read from $(SOURCE_D) and executables created in corresponding subdirectories of $(BUILD)

- **Installation**
  - Created executables and libraries are copied in $(INSTALL)

Genattr
gencheck
gencheckgencondsgenconstants
genflags
genopinit
genpredsgenattrtab
genemit
genrntlgensimdpegsenoutput
genrecog
genautomata
gencoles
genconfig
genextract
genotype
genmodes
genpeep

More Details of an Actual Stage 1 Build for C

- native cc, binutils, libraries
- fixincl
- gen*
- ccl
- xgcc
- libgcc
- libiberty
- libraries
- GCC sources

cc, binutils, libraries for stage 2

Notes
Building a MIPS Cross Compiler on i386: A Closer Look

Requirement: \( BS = HS = \text{i386}, \ TS = \text{mips} \)

• Stage 1 cannot build gcc but can build only cc1
• Stage 1 build cannot create executables
• Library sources cannot be compiled for mips using stage 1 build

we have not built libraries for mips
Building a MIPS Cross Compiler on i386: A Closer Look

Requirement: BS = HS = i386, TS = mips

- Stage 1 cannot build gcc but can build only cc1
- Stage 1 build cannot create executables
- Library sources cannot be compiled for mips using stage 1 build
- Stage 2 build is not possible

Difficulty in Building a Cross Compiler

requires

libgcc

uses

gcc for target

target libraries

Notes
Generated Compiler Executable for All Languages

- Main driver: $BUILD/gcc/xgcc
- C compiler: $BUILD/gcc/cc1
- C++ compiler: $BUILD/gcc/cc1plus
- Fortran compiler: $BUILD/gcc/f95i
- Ada compiler: $BUILD/gcc/gnat1
- Java compiler: $BUILD/gcc/jcl
- Java compiler for generating main class: $BUILD/gcc/jvgenmain
- LTO driver: $BUILD/gcc/lto1
- Objective C: $BUILD/gcc/cc1obj
- Objective C++: $BUILD/gcc/cc1objplus

Basic Transformations in GCC

Transformation from a language to a different language

Target Independent

Parse → Simplify → Tree SSA → Optimize → Generate RTL → Optimize RTL → Generate ASM

Target Dependent

GIMPLE → RTL
RTL → ASM

GIMPLE Passes
RTL Passes
Instruction Specification and Translation: A Recap

- Target Independent
  - Parse
  - Simplify
  - Tree SSA
  - Optimize
  - Generate RTL

- Target Dependent
  - Optimize RTL
  - Generate ASM

GIMPLE: target independent
RTL: target dependent
Need: associate the semantics
⇒ GCC Solution: Standard Pattern Names

GIMPLE ASSIGN

(define_insn "movsi"
  [(set (match_operand 0 "register_operand" "r")
        (match_operand 1 "const_int_operand" "k"))
   "" /* C boolean expression, if required */
   "li %0, %1"
  ]
)

D.1283 = 10;
(set
  (reg:SI 58 [D.1283])
  (const 10: [0xa])
  li $t0, 10
)
Retargetability Mechanism of GCC

Input Language → Compiler Generation Framework → Target Name

- Language Specific Code
- Language and Machine Independent Generic Code
- Machine Dependent Generator Code
- Machine Descriptions

Selected → Copied → Generated

Selected → Copied → Generated

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

GIMPLE → PN + PN → IR-RTL + IR-RTL → ASM

GIMPLE → IR-RTL + IR-RTL → ASM

Plugin Structure in cc1

toplevel → main → frontend → pass manager → pass 1 → code for pass 1

langhook → code for language 1 → pass expand → recognizer code

code for pass 1 → code for pass 2 → expander code

optab_table → generated code for machine 1

MD n → MD 2 → MD 1

Plugin Structure in cc1
The Mechanism of Dynamic Plugin

- pass manager
- code for pass
- code for dynamic plugin
- expander code optab table
- recognizer code for

Runtime initialization of the appropriate linked list of passes
Made possible by dynamic linking
Execution Order in Intraprocedural Passes

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

Notes
LTO Support in GCC

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Analysis 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>

Notes:
- `-flto`
- `-flto -flto-partition=none`
- WHOPR mode

cc1 and Single Process lto1

toplev_main
... compile_file
... cgraph_analyze_function
cgraph_optimize
... ipa_passes
... cgraph_expand_all_functions
... tree_rest_of_compilation

cc1
cpp and Single Process lto1

toplev_main
... compile_file
... cgraph_analyze_function

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

lto1_main
... read_cgraph_and_symbols
... materialize_cgraph

The GNU Tool Chain for Single Process LTO Support

gcc

"Fat" .s files
as

"Fat" .o files
cc1'
cc1
lto1'
lto1
common

Single .s file
as

Single .o file + glibc/newlib
ld

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
The GNU Tool Chain for Single Process LTO Support

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

```c

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

Multi Process LTO (aka WHOPR LTO)

Option `-flto -c`

Option `-flto -o out`

Notes
Multi Process LTO (aka WHOPR LTO)

Option `-flto -c`

External View

<table>
<thead>
<tr>
<th>File</th>
<th>Source</th>
<th>Compiler</th>
<th>LTO</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1.c</td>
<td>cc1</td>
<td>lto1</td>
<td>common</td>
<td>f1.o</td>
</tr>
<tr>
<td>f2.c</td>
<td>cc1</td>
<td>lto1</td>
<td>common</td>
<td>f2.o</td>
</tr>
<tr>
<td>f3.c</td>
<td>cc1</td>
<td>lto1</td>
<td>common</td>
<td>f3.o</td>
</tr>
</tbody>
</table>

Internal View

Option `-flto -o out`

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

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

/GEN

WPA

Option `-flto -o out`

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

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

/GEN

LTRANS

Notes
Option `-flto -c`

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

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

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

```
out
```

---

**External View**

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

---

**Internal View**

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

```
out
```
Single Process LTO

Option `-flto -c`

```plaintext
f1.c → cc1 → lto1 → f1.o

f2.c → cc1 → lto1 → f2.o

f3.c → cc1 → lto1 → f3.o
```

IPA + LTRANS

Option

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

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

This WPA can examine function bodies also

LGEN

Redundancy in MIPS Machine Descriptions: Example 3

```plaintext
[(set (match_operand: m 0 "register_operand" "c0") (plus: m (mult: m (match_operand: m 1 "register_operand" "c1") (match_operand: m 2 "register_operand" "c2")))) (match_operand: m 3 "register_operand" "c3")]
```

RTL Template

Structure

Details

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>m</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>*mul acc si</td>
<td>SI</td>
<td>=l*?*,d?</td>
<td>d, d</td>
<td>d, d</td>
<td>0, d</td>
</tr>
<tr>
<td>*mul acc si r3900</td>
<td>SI</td>
<td>=l*?<em>,d</em>?,d?</td>
<td>d, d, d</td>
<td>d, d, d</td>
<td>0,1, d</td>
</tr>
<tr>
<td>*macc</td>
<td>SI</td>
<td>=l,d</td>
<td>d, d</td>
<td>d, d</td>
<td>0,1</td>
</tr>
<tr>
<td>*madd4&lt;mode&gt;</td>
<td>ANYF</td>
<td>=f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>*madd3&lt;mode&gt;</td>
<td>ANYF</td>
<td>=f</td>
<td>f</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>
Hooking up Back End Details

Runtime initialization of data structure

And the final realization...