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

- Compiler Specifications
  - Compiler Generator
  - Generated Compiler

External View
- Machine descriptions
- Configuration and building
- Gray box probing
- Pass structure and IR
- Data Flow Analysis
- Parallelization, Vectorization

Internal View
- Front end hooks
- Retargetability mechanism
- Pass structure
- Control flow
- Static and dynamic plugin mechanisms

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Compilation Models

Aho Ullman Model

Front End → AST → Optimizer → Target Indep. IR → Code Generator → Target Program

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

Davidson Fraser Model

Front End → AST → Expander → Register Transfers → Optimizer → Register Transfers → Recognizer → Target Program

Davidson Fraser: Instruction selection
- over AST using
- structural tree pattern matching
- naive code which is
  - target dependent, and is
  - optimized subsequently
The GNU Tool Chain for C

Source Program

\[
gcc
\]

Target Program

\[
glibc/newlib
\]

\[
lc
\]

\[
cc1
\]

\[
cpp
\]

\[
as
\]
The Architecture of GCC

Input Language

Compiler Generation Framework

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Selected

Copied

Copied

Generated

Generated

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer

Source Program

Generated Compiler (cc1)

Assembly Program

Target Name

Development Time

Build Time

Use Time
Configuring GCC

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

Level n C

C_{n-1}

C_n

C_{n-2}

m/c

input language

output language

implementation language
A Native Build on i386

**Requirement:** \( BS = HS = TS = i386 \)

- Stage 1 build compiled using \( cc \)
- Stage 2 build compiled using \( gcc \)
- Stage 3 build compiled using \( gcc \)
- Stage 2 and Stage 3 Builds must be identical for a successful native build
Build for a Given Machine

This is what actually happens!

- **Generation**
  - Generator sources
    - \( $(\text{SOURCE}_D)/\text{gcc/gen*.c} \) are read and generator executables are created in \( $(\text{BUILD})/\text{gcc/build} \)
  - MD files are read by the generator executables and back end source code is generated in \( $(\text{BUILD})/\text{gcc} \)

- **Compilation**
  - Other source files are read from \( $(\text{SOURCE}_D) \) and executables created in corresponding subdirectories of \( $(\text{BUILD}) \)

- **Installation**
  - Created executables and libraries are copied in \( $(\text{INSTALL}) \)
More Details of an Actual Stage 1 Build for C

GCC sources → native cc, binutils, libraries → libraries → libiberty → fixincl → gen* → cc1 → cpp → xgcc → libgcc

cc, binutils, libraries for stage 2

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Building a MIPS Cross Compiler on i386: A Closer Look

Stage 2 build is infeasible for cross build

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
- Stage 2 build is not possible

we have not built libraries for mips
Difficulty in Building a Cross Compiler

```
require

gcc for target
requires
libgcc
uses
target libraries
```
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/f951
- 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 \textit{different} language

- Target Independent
  - Parse \rightarrow Gimplify
  - Tree SSA
  - Optimize
  - GIMPLE \rightarrow RTL
- Target Dependent
  - Generate RTL
  - Optimize RTL
  - Generate ASM
  - RTL \rightarrow ASM

GIMPLE Passes

RTL Passes
Instruction Specification and Translation: A Recap

- Target Independent
  - Parse
  - Gimplify
  - Tree SSA
  - Optimize
- Target Dependent
  - Generate RTL
  - Optimize RTL
  - Generate ASM

- GIMPLE: target independent
- RTL: target dependent
- Need: associate the semantics

GCC Solution: Standard Pattern Names

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

(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_int 10: [0xa])
) ➞ li $t0, 10
Retargetability Mechanism of GCC

Compiler Generation Framework

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

Target Name
- GIMPLE → PN
- PN → IR-RTL
- IR-RTL → ASM
- GIMPLE → IR-RTL
- IR-RTL → ASM

Development Time
- GIMPLE → PN
- PN → IR-RTL

Build Time
- IR-RTL → ASM

Use Time
- GIMPLE → IR-RTL
- IR-RTL → ASM

Generated Compiler
- Parser
- Gimplifier
- Tree SSA Optimizer
- RTL Generator
- Optimizer
- Code Generator

Selected
Copied
Copied
Generated
Generated
Generated
The Mechanism of Dynamic Plugin

Runtime initialization of the appropriate linked list of passes

Made possible by dynamic linking
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
Execution Order in Interprocedural 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>
## LTO Support in GCC

<table>
<thead>
<tr>
<th>Whole Program Analysis</th>
<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>Call graph without function bodies</td>
<td>Single partition of the program</td>
<td>Single partition of the program</td>
<td>Multiple partitions of the program</td>
</tr>
<tr>
<td>Not supported</td>
<td>Suppported in GCC-4.6.0</td>
<td>Will be supported in future</td>
<td></td>
</tr>
<tr>
<td>Call graph with function bodies</td>
<td>Suppported in GCC-4.6.0</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

- `-flto`
- `-flto -flto-partition=none`
- WHOPR mode
**cc1 and Single Process**

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

```
cgraph_optimize
...
ipa_passes
...
cgraph_expand_all_functions
...
tree_rest_of_compilation
```
cc1 and Single Process lto1

toplev_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
The GNU Tool Chain for Single Process LTO Support

gcc

collect2

cc1′ common

lto1′

as

"Fat" .s files

"Fat" .o files

Single .s file

Single .o file + glibc/newlib

collect2

ld

a.out file

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)

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`

f1.c → c1' → lto1' → f1.o
common

f2.c → c1' → lto1' → f2.o
common

f3.c → c1' → lto1' → f3.o
common

Option `-flto -o out`

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

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

WPA

out ← c1' ← lto1'
common

LGEN

LTRANS
Single Process LTO

Option `-flto -c`

f1.c → ccl’ → lto1’ → common → f1.o

f2.c → ccl’ → lto1’ → common → f2.o

f3.c → ccl’ → lto1’ → common → f3.o

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

IPA + LTRANS

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

This WPA can examine function bodies also

LGEN

out
Redundancy in MIPS Machine Descriptions: Example 3

\[
\begin{align*}
\text{(set (match_operand: } m \text{ 0 "register_operand" "} c0\text{") (plus: } m \\
\text{(mult: } m \text{ (match_operand: } m \text{ 1 "register_operand" "} c1\text{") \\
\text{(match_operand: } m \text{ 2 "register_operand" "} c2\text{")})]) \\
\text{(match_operand: } m \text{ 3 "register_operand" "} c3\text{")})])
\end{align*}
\]

**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*???,d*??,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

$(SOURCE)/gcc/optabs.h
$(SOURCE)/gcc/optabs.c

optab_table

mov_optab

Runtime initialization of data structure

OTI_mov

insn_data

"movsi"

1280

gen_movsi

insn_code

CODE_FOR_movsi

CODE_FOR_nothing

$(BUILD)/gcc/insn-codes.h

CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing

$(BUILD)/gcc/insn-opinit.c

...
And the final realization . . .
And the final realization . . .
And the final realization . . .

Work hard 😊