A Summary of Essential Abstractions

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

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

13 June 2014
Part 2

Methodology
Our Padagogy

- Compiler Specifications
  - Compiler Generator
    - Generated Compiler
      - Gray box probing
        - Pass structure and IR
          - Parallelization, Vectorization
      - Configuration and building
        - Front end hooks
      - Retargetability mechanism
    - Machine descriptions
  - Internal View
    - Pass structure
      - Control flow
        - Static and dynamic plugin mechanisms
Gray Box Probing

Black Box Probing

Observe

Observe
Gray Box Probing

White Box Probing
Gray Box Probing

Phase 1

Phase 2

... 

Phase n

Observe

Observe

Observe

Observe
Systematic Development of Machine Descriptions

- Conditional control transfers
- Function Calls
- Arithmetic Expressions
  - Sequence of Simple Assignments involving integers
    - MD Level 1
    - MD Level 2
    - MD Level 3
    - MD Level 4
Part 3

The Framework
The GNU Tool Chain for C

Source Program

- gcc

Target Program

- cc1
- cpp
- as
- ld
- glibc/newlib
The Architecture of GCC

Input Language → Compiler Generation Framework → Target Name


Parser → Gimplifier → Tree SSA Optimizer → Expander → Optimizer → Recognizer

Source Program → Generated Compiler (cc1) → Assembly Program

Development Time → Build Time → Use Time

Selected → Copied → Generated
Part 4

The Generated Compiler
**Compilation Models**

**Aho Ullman Model**

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

**Davidson Fraser Model**

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

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

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

Uday Khedker
GRC, IIT Bombay
Basic Transformations in GCC

Transformation from a language to a different language

Target Independent → Target Dependent

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

GIMPLE → RTL

RTL → ASM

GIMPLE Passes

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

- **toplevel**
- **main**
- **front end**
- **pass manager**
- **langhook**
- **pass 1**
  - code for pass 1
  - MD 1
  - MD 2
  - MD n
- **pass 2**
  - code for pass 2
- **pass expand**
  - expander code
  - insn_data
  - optab_table
  - code for machine 1
- **pass n**
  - recognizer code
  - code for language 1
  - code for language 2
  - code for language n
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>Pass 1</th>
<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 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part 5

LTO
Partitioned and Non-Partitioned LTO

- **Analysis**: Load complete call graph
  - Load function summaries but not bodies
  - Load all function bodies

- **Transformation**: Load groups of function bodies
  - Load all function bodies one by one
  - Load all function bodies

**Partitioned Mode**

- No need to load the entire program in memory
- IPA possible (multiple function bodies)
- Parallel transformations possible
- Analysis and transformations in independent processes

Sequential Analysis

Uday Khedker

GRC, IIT Bombay
Partitioned and Non-Partitioned LTO

Analysis

Transformation

Load complete call graph

Load function summaries but not bodies

Load all function bodies

Partitioned Mode

Load all function bodies

Load groups of function bodies

Sequential Analysis

Load all function bodies

Load function bodies one by one

All function bodies already loaded

Balanced partitions: `-flto -flto-partitions=balanced`

One Partition per file: `-flto -flto-partitions=1to1`

Partitions by number: `-flto --params lto-partitions=n`

Partitions by size: `-flto --params lto-min-partition=s`

Uday Khedker

GRC, IIT Bombay
### Partitioned and Non-Partitioned LTO

#### Analysis
- Load complete call graph
- Load function summaries but not bodies
- Load all function bodies

#### Transformation
- Load all function bodies
- Load function bodies one by one
- Load groups of function bodies
- All function bodies already loaded

### Non-Partitioned Mode
- Entire program needs to be loaded in memory
- No partitions
  - `flto -flto-partitions=none`
- Strictly sequential transformations
- Analysis and transformations in the same processes
cc1 and Single Process lto1

toplevel_main
...
  compile_file
...
  cgraph_analyze_function

cc1
...
  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

lto

The GNU Tool Chain for Single Process LTO Support

The process involves several components and steps:

1. **gcc** takes the source code as input.
2. **collect2** is used to collect the intermediate files.
3. **cc1** is used for the compilation step.
4. **lto1** is used for the linking step.
5. **as** is used for assembling the files.
6. **ld** is used for linking the object files.
7. **collect2** is used to collect the final object files.
8. **ld** is used again to link the final object files.
9. The **a.out** file is generated.

The arrows indicate the flow of the process, with the terms "Fat" .s files and "Fat" .o files indicating the types of files generated at different stages.
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
      execute_ipa_pass_list(all_late_ipa_passes)
   cgraph_expand_all_functions
      cgraph_expand_function
         /* Intraprocedural passes on GIMPLE, */
         /* expansion pass, and passes on RTL. */
Partitioned LTO (aka WHOPR LTO)

Option `-flto -c`

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

```
f2.c  →  cc1'  →  lto1'  →  f2.o
  common
```

```
f3.c  →  cc1'  →  lto1'  →  f3.o
  common
```

Option `-flto -o out`

```
out  ←  cc1'  ←  lto1'  ←  ltrans0.o
   common
```

WPA

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

LGEN

LTRANS

/tmp/ccdKEyVB.ltrans0.o
(possibly multiple files)
Non-Partitioned LTO

LGEN

Option -flto -c
f1.c → cc1' → lto1' → f1.o
f2.c → cc1' → lto1' → f2.o
f3.c → cc1' → lto1' → f3.o

IPA + Transformations
Option
-flto -o out
-flto-partition=none

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

This IPA can examine function bodies also

out→
Part 6

The Build Process
Configuring GCC
Bootstrapping: The Conventional View

Level n C

C_{n-1} \quad C_{n-2}

implementation language

input language

output language

m/c

m/c
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
    - $(SOURCE_D)/gcc/gen*.c) are read and generator executables are created in $(BUILD)/gcc/build
  - 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)
More Details of an Actual Stage 1 Build for C

GHC sources → native cc, binutils, libraries

libraries

libiberty

fixincl

gen*

cc1

cpp

xgcc

libgcc

target binutils, libraries

cc, binutils, libraries for stage 2
Building a MIPS Cross Compiler on i386: A Closer Look

Stage 1 Build

Stage 2 build is infeasible for cross build

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

we have not built libraries for mips

Uday Khedker

GRC, IIT Bombay
Difficulty in Building a Cross Compiler

- gcc for target
- target libraries
- requires libgcc
- uses

require

Uday Khedker
GRC, IIT Bombay
### Generated Compiler Executable for All Languages

<table>
<thead>
<tr>
<th>Component</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main driver</td>
<td>$BUILD/gcc/xgcc</td>
</tr>
<tr>
<td>C compiler</td>
<td>$BUILD/gcc/cc1</td>
</tr>
<tr>
<td>C++ compiler</td>
<td>$BUILD/gcc/cc1plus</td>
</tr>
<tr>
<td>Fortran compiler</td>
<td>$BUILD/gcc/f951</td>
</tr>
<tr>
<td>Ada compiler</td>
<td>$BUILD/gcc/gnat1</td>
</tr>
<tr>
<td>Java compiler</td>
<td>$BUILD/gcc/jcl</td>
</tr>
<tr>
<td>Java compiler for generating main class</td>
<td>$BUILD/gcc/jvgenmain</td>
</tr>
<tr>
<td>LTO driver</td>
<td>$BUILD/gcc/lto1</td>
</tr>
<tr>
<td>Objective C</td>
<td>$BUILD/gcc/cc1obj</td>
</tr>
<tr>
<td>Objective C++</td>
<td>$BUILD/gcc/cc1objplus</td>
</tr>
</tbody>
</table>
Part 7

Retargetability
Examples of Influences on the Machine Descriptions

- **Source Language**
  - INT_TYPE_SIZE
  - Activation Record

- **Build System**

- **Host System**

- **Target System**
  - Instruction Set Architecture
  - Assembly and executable formats

- **GCC Architecture**
  - Generation of `nop`
  - Tree covers for instruction selection
  - `define Predicate`

- **Build System**
  - `hwint.h`

- **Host System**
  - `<target>.h`

- **Target System**
  - `<target>.md`
  - `<target>.h`
  - Other headers

- **Machine Description**
  - `<target>.h`
Redundancy in MIPS Machine Descriptions: Example 3

\[
((\text{set} \ (\text{match} \ \text{operand}: m \ 0 \ "\text{register} \ \text{operand}" \ "c0") \ (\text{plus:} m \\
(\text{mult:} m \ (\text{match} \ \text{operand}: m \ 1 \ "\text{register} \ \text{operand}" \ "c1") \\
(\text{match} \ \text{operand}: m \ 2 \ "\text{register} \ \text{operand}" \ "c2") ))) \\
(\text{match} \ \text{operand}: m \ 3 \ "\text{register} \ \text{operand}" \ "c3")))
\]

**RTL Template**

- Pattern name
- $m$  
- $c0$  
- $c1$  
- $c2$  
- $c3$

<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>
Instruction Specification and Translation: A Recap

- **Parse → Gimplify → Tree SSA Optimize → Generate RTL → Optimize RTL → Generate ASM**

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

GCC Solution Example:

```c
(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"
)
```

Uday Khedker
GRC, IIT Bombay
Translation Sequence in GCC

```lisp
(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"
)
```

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

Input Language

Parser
Gimplifier
Tree SSA Optimizer
RTL Generator
Optimizer
Code Generator

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

Selected
Copied
Copied
Generated
Generated

GIMPLE \rightarrow PN
PN \rightarrow IR-RTL
IR-RTL \rightarrow ASM

GIMPLE \rightarrow IR-RTL
IR-RTL \rightarrow ASM

Use Time
Build Time
Development Time

Generated Compiler
### Hooking up Back End Details

#### $(\text{SOURCE})$/gcc/optabs.h
- `optab_table`
- `OTI_mov`
- `SI`
- `SF`
- `insn_code`
  - `CODE_FOR_movsi`
  - `CODE_FOR_nothing`

#### $(\text{BUILD})$/gcc/insn-output.c
- `insn_data`
  - `"movsi"`
  - `1280`
  - `gen_movsi`

#### Code Generation
- $(\text{SOURCE})$/gcc/optabs.c
- $(\text{BUILD})$/gcc/insn-codes.h
  - `CODE_FOR_movsi=1280`
  - `CODE_FOR_movsf=CODE_FOR_nothing`

#### Runtime Initialization
- `init_all_optabs`
- `cc1`

---

**Uday Khedker**

**GRC, IIT Bombay**
The Process of Expansion

GIMPLE

match SPN

Search optab_table[] and extract CODE_FOR_<SPN>

Search insn_data[] and extract gen_<SPN>

invoke gen_<SPN>

gimple_expand_cfg

RTL

Generated