Gcc under the hood
Siddhesh Poyarekar
The Next Hour

- The journey from source to binary
  - Understand how your source code goes through gcc, as and finally ld
- Digging deeper
  - We explore how intermediate code generated during compilation can help us understand our program. And the compiler.
- Hacking
  - We look at some use cases for enhancing the compiler
- Am Toolchain Tech Lead at Linaro Developer Services
  - We provide professional toolchain services for the Arm ecosystem
- Maintain the GNU C Library
- Contribute to GCC/binutils
- Hack on LuaJIT
From Source to Binary
The 30 second view of a toolchain

1. Read Program Code
2. Optimize Program Logic
3. Generate Assembly Code
4. Generate Binary Code
5. Link
What this means for us

```c
#include <stdio.h>
int main ()
{
    printf("Hell Oh World!\n");
    return 0;
}
```

- Read Program Code
- Optimize Program Logic
- Generate Assembly Code
- Generate Binary Code
- Link

```
$ Hell Oh World!
```
The 60 second view of the compiler

The front end: A dozen or so programming languages!

The middle end: A few HUNDRED optimization passes!

Intermediate Representation!

The back end: Translate code into the machine instructions.

- Read Program Code
- Optimize Program Logic
- Generate Assembly Code
The 90 second view

Read Program Code

Optimize Program Logic

Generate Assembly Code

gcc/passes.def
gcc/tree-*
...

gcc/c-family/*
gc/cp/*
gcc/fortran/*
...

GIMPLE
RTL

gcc/config/<arch>/*
The 120 second view

Read Program Code

Optimize Program Logic

Generate Assembly Code

The front end: A dozen or so programming languages!

The middle end: A few HUNDRED optimization passes!

GCC CONFIG/arch/*.c

GIMPLE RTL

The back end: Translate code into the machine instructions.

GCC CONFIG/arch/*.md
Digging Deeper
Intermediate Representations

- **GENERIC**
  - Tree structure representation of a function
  - Interface between the parser and optimiser

- **GIMPLE**
  - Three address, machine and language independent format
  - Lowered from GENERIC
  - More restrictive than GENERIC

- **RTL**
  - Lowest Intermediate representation
  - Sequential instruction descriptions lowered from GIMPLE
  - Expressed as Lisp-like S-expressions
GENERIC

- Tree structure with connected tree_nodes (gcc/tree-core.h)
- Everything based on the tree_base struct
  - Look for struct GTY(()) tree_base

```
struct GTY(())
tree_<specialization> {  
  struct tree<type it is based on>;  
  <The contents of the tree>;  
};

e.g.

struct GTY(()) tree_string {  
  struct tree_typed typed;  
  int length;  
  char str[1];  
};
```
GIMPLE

- The optimizer workhorse
- Linear statements with no more than 3 operands in most cases
- Tuples defined in gcc/gimple.def
- Control flow described by the Control Flow Graph (CFG)
Control Flow Graphs

- Overlays on GIMPLE and RTL
- Graph that connects basic blocks (BB) of sequential code
  - Each BB may have one or more GIMPLE tuples
- Edges describe flow of control from one BB to another
- See gcc/cfg.* for more details
- Loops get special treatment
  - See gcc/cfgloop.h for details
GIMPLE Single Static Assignment (SSA)

- Variables are assigned in exactly one location
- Multiple assignments result in multiple copies of the variable
  
  \[
  x = 10; \ x += 20; \\
  \text{Becomes} \\
  x_1 = 10; \ x_2 = x_1 + 20;
  \]

- Conditional assignments result in mysterious entities called PHI nodes
  
  \[
  \text{if (n > 10) } x = 10; \\
  \text{else } x = 20; \\
  \text{return } x; \\
  \text{Becomes} \\
  \text{if (n > 10) } x_1 = 10; \\
  \text{else } x_2 = 20; \\
  \text{# x_3 = PHI<x_1, x_2>;} \\
  \text{return } x_3;
  \]
Register Transfer Language (RTL)

- Low level representation intended to map directly to one or more instructions
- Internal structure form as well as a textual form made of Lisp-like S-expressions
- RTL expressions are listed in rtl.def
- Textual form used to write a machine description
Machine Description

- We want assembly in the end
- *.md files with RTL instruction descriptions
  - A gcc preprocessing tool parses it and generates code
- An RTL instruction may expand into one or more machine instructions
- One machine description file per architecture
Extending the Machine Description

- Additional sources per architecture to make more intelligent decisions about generated code
- Source files in config/<arch>/*.c
Optimisation Passes

- Tree Level Optimisers
  - Static data flow analysis on tree IR (GIMPLE)
  - Machine Independent
  - E.g. DCE, CSE, IV optimisation, vectorisation

- RTL Optimisers
  - Static analysis on sequential IR
  - Machine dependent
  - E.g. register allocation, instruction scheduling, etc.

- Plug Your Own Optimiser
  - Add to passes.def
Peeking and Poking
Studying Intermediate Outputs

- `-fdump-tree-*` options to study GIMPLE IR outputs for every pass
- `-fdump-rtl-*` options to study RTL IR outputs for every pass
Squeezing the last drop

- Microarchitecture descriptions
- Machine descriptions with pipeline information
- Used by the instruction scheduler pass to select or reorder instructions
- Per-cpu cost tables
  - Loop alignment
  - Function alignment
  - Costs of operations (e.g. unaligned access)
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org