# LLVM - discover secrets of the dragon

Dominik Adamski

29 November 2017

# Agenda

# Overview

## LLVM features

- It is a collection of modular and reusable compiler and toolchain technologies [1]
- Name is not an acronym
- Started as a student project in 2001
- Large community support
- Awarded the 2013 ACM Sotware System Award
- Main open source competitor for GCC

FIGURE – LLVM logo

# Details

## What is the presentation about ?

- General overview of LLVM project
- LLVM internal design
- Middle-end optimization

## What is skipped ?

- Front-end input code transformation
- Back-end binary code generation

# LLVM vs GCC

**LLVM advantages**

- Available under permissive license
- Modular design
- Source code written entirely in C++
- Default compiler for Apple products
- Reusable components

**GCC advantages**

- Linux kernel compilation
- Default compiler for multiple platforms
- Variety of supported languages
- Numerous supported target platforms
- 30 years of development

**Performance**

The race is on and it is hard to indicate the winner

# LLVM Architecture

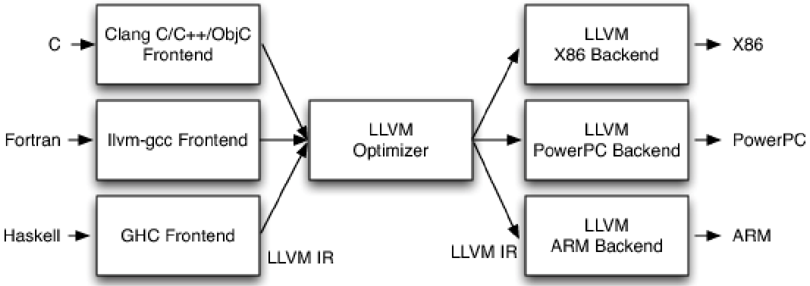


FIGURE – Architecture of LLVM compiler [2]

# LLVM IR Features

- Simplified syntax - similar to assembly language
- Common for multiple input languages
- Aiming at being target-independent as much as possible
- Strongly typed
- Infinite number of registers
- Compliant with Static Single Asssignment principle
- Full language specification available on LLVM's webpage

Exemplary source code :

```
int hello(int a) {
  float b = 2;
  char c[10];
  c[0] = 3;
  return a * b * c[0];
}
```

Corresponding IR code

# LLVM tools

Generate IR code :

```
clang -S -emit-llvm -g -Xclang -disable-O0-optnone hello.c -o hello.ll
```

Corresponding IR code

Run optimization passes :

```
opt -O3 -S hello.ll -o hello-opt.ll
```

Optimized IR code

Generate assembly language :

```
llc -filetype=asm Examples/hello-opt.ll
```

Assembly code

# LLVM passes

**Type of activity :**

- Analysis passes
- Transform passes
- Utility passes

**Scope of operation :**

- Module passes
- Function passes
- Region passes
- Loop passes
- Basic block passes
- Call graph SCC passes

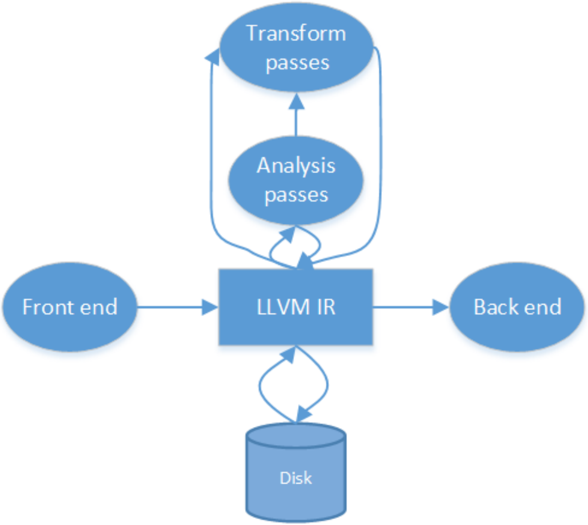


FIGURE – Cooperation of LLVM passes [3]

# View control flow graph

Use cases of utility passes :

- IR code verification
- Narrowing source of LLVM bug

Exemplary source code :

```
int process(int a) {
  int res = 0;
  if (a > 0)
    res++;
  else
    res--;
  return res;
}
```

Corresponding IR code
View-cfg-pass invocation :
`opt -view-cfg cfg.ll`

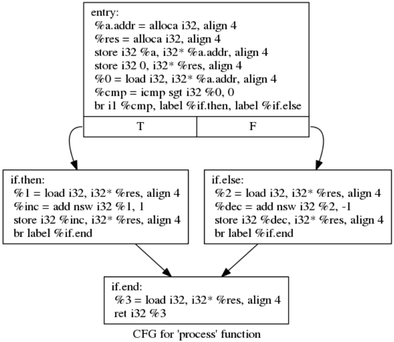


FIGURE – Control flow graph generated by view-cfg pass

# Features of analysis passes

High level description of analysis passes :

- Providing useful information for transform passes
- No modification of IR code
- Single responsibility design

# Basic alias analysis

High level description of alias analysis :

- Checking possible immediate dependence between two pointers (possible outputs : must, partial, may, or partial alias)
- Possibly computation intensive
- Required for some transformation passes
- Trade off between complexity and accuracy

Exemplary source code :

```
char glob[40];

int analysis(char **ptr) {
  char local[40];
  local[1] = 'a';
  int res;
  res = **ptr + 2 + local[1];
  return res;
}
```

Corresponding IR code
Basic alias analysis invocation :
`opt -basicaa -aa-eval -S -print-all-alias-modref-info alias.ll`
Result of alias analysis

# Loops detection

Loops :

- Detected on the basis of LLVM IR
- Basic units for multiple code optimisation passes

Exemplary source code : loops.c
Corresponding IR code
Loop detection analysis invocation :
`opt loops.ll -loops -S -analyze`
Result of loop detection analysis

# Features of transform passes

High level description of transform passes :

- Modification of IR code
- Limited range of modification
- Possible invalidation of analysis result
- Single responsibility design

# Memory to Registers pass

High level description of promotion memory variable into scalar registers :

- Function pass
- Replacement of specified list of `alloca` instruction by scalar registers
- PHI node insertion
- No modification of control flow graph

Exemplary source code :

```
int foo(int a) {
  int x;
  if (a)
    x = 2;
  else
    x = 3;
  return x;
}
```

Corresponding IR code
Memory to registers invocation :
`opt -S -mem2reg mem.ll`
Optimisation result

# Induction Variable pass

High level description of induction variable pass :

- Loop pass
- Canonicalisation of the loop
- Simplification of the loop for further optimisation
- All loop-dependent variables dependent on induction variable

Exemplary source code :

```
void foo() {
  int a[10];
  for (int i = 2;
         i * i < 100;
         i++) {
    a[i] = i;
  }
}
```

Corresponding IR code
Invocation of induction variable optimisation :
`opt -mem2reg -indvars -S preindvar.ll`
Result of introduction induction variable

# Loop Invariant Code Motion pass

High level description of loop invariant code motion pass :

- Loop pass
- Motion of loop invariant code outside the loop
- Strong dependence on alias analysis

Exemplary source code :

```c
int foo(int n) {
  int res;
  int a[10];
  for (int i = 0; i < 10;
                      i++) {
    res = n + 5;
    a[i] = i;
  }
  return res + a[2];
}
```

Corresponding IR code
Invocation of `licm` optimisation :
`opt -mem2reg -licm -S licm.ll`
Result of `licm` optimisation

# Polly optimizer

Features of Polly optimizer :

- Part of LLVM project
- Set of LLVM IR passes
- Started by Tobias Grosser as student project - 2011
- Abstract mathematical model used for code analysis and optimisation
- Loop optimizer
- Automatical code parallelisation - OpenMP
- Data locality improvement - tiling
- Code vectorisation - SIMD

# Example

Source code of two matrix multiplications taken from Polybench benchmark [4]

```
clang -O3 -I utilities -I linear-algebra/kernels/2mm
utilities/polybench.c linear-algebra/kernels/2mm/2mm.c -DPOLYBENCH_TIME
-o 2mm_polly -mllvm -polly -mllvm -polly-tiling -mllvm -polly-parallel
-mllvm -polly-optimized-scops
```

Mathematical description of optimized code by Polly

# Test Polly

Target platform :
`AMD Ryzen 5 1600, 16GB DDR4, Ubuntu 16.04`

Tested compilers :

```
gcc v5.4, -O3 -> 34s
clang (master branch, latest commit - 20.09.2017), -O3 -> 38s
clang (master branch, latest commit - 20.09.2017), -polly -tiling
-parallel -> 0.37s
```

# Further reading

- LLVM documentation
- Polly official webpage
- LLVM Developers' Meeting

# Bibliography


ADMIN TEAM, L.
LLVM official webpage.
http://llvm.org/.
Accessed : 23.11.2017.

LATTNER, C.
LLVM's implementation of the three-phase design.
http://www.aosabook.org/en/llvm.html.
Accessed : 23.11.2017.

LOPEZ, B. C., AND AULER, R.
Introducing LLVM Intermediate Representation.
https://www.packtpub.com/books/content/introducing-llvm-intermediate-representation.
Accessed : 24.11.2017.

POUCHET, L. N.
Polybench/c version 3.2.
http://web.cs.ucla.edu/~pouchet/software/polybench/.
Accessed : 25.11.2017.

# Presentation style

If you like this presentation template, you can find it here.

It is distributed under GNU-GPL v3 Licence.

Thank you for your attention☺