

# The LLVM compiler framework

## Introduction

Daniele Cattaneo

Politecnico di Milano

2020-05-13

*These slides were originally written by Michele Scandale, Ettore Speziale and Stefano Cherubin for the “Code Transformation and Optimization” course.*

# Contents

**Introduction**

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

# Compilers and compilers

Not all compilers are the same... The traditional distinction was:

## Toy Compiler

- ▶ small codebase
- ▶ easy to modify
- ▶ limited capabilities

## Production-Quality Compiler

- ▶ huge codebase
- ▶ hard to modify
- ▶ produces high-quality code

Working with a production-quality compiler is *initially* **hard**...  
...however it provides a huge set of tools that toy compilers **miss**!

# LLVM: Low Level Virtual Machine

Initially started as a small research project at Urbana-Champaign.

Now it has grown to a huge size...

- ▶ Key technology in the **industry**: AMD, Apple, Google, Intel, NVIDIA...
- ▶ Still intensively used in **research** about compilers

# GCC vs LLVM

**LLVM** [1] is Open Source

If you are familiar with Linux you might have used **GCC** [2]...

GCC is older than LLVM

- ⇒ GCC produces better code
- ⇒ LLVM is generally faster
- ⇒ LLVM is more modular and *clean*

# Contents

Introduction

**Compiler organization**

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

# Compiler pipeline

Typically a compiler is a **pipeline**.

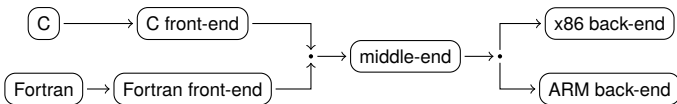
Advantages of the pipeline model:

- ▶ **simplicity** – read something, produce something
- ▶ **locality** – no superfluous state

Complexity lies on **chaining** together stages.

# Compiler pipeline

High-level pipeline structure of a compiler:



There are three main components:

**Front-end** **translate** a source file into the intermediate representation

**Middle-end** **analyze** the intermediate representation, **optimize** it

**Back-end** **generate** target machine assembly from the intermediate representation



# The LLVM compiler pipeline

We will consider only the *middle-end*.

Same concepts are valid also for {front,back}-end.

**IR** (a.k.a. Intermediate Representation) the **language** used in the middle-end

**Pass** a **pipeline stage**  
a Pass may have **dependencies** on other Passes.

**Pass Manager** component that **schedules** passes according to their **dependencies** and **executes** them  
(*builds the pipeline*)

# First insights

A compiler is **complex**:

- ▶ passes are the **elementary unit of work**
- ▶ Pass Manager must be **advised** about pass chaining
- ▶ pipeline shapes are **not fixed** – they can change from one compiler execution to another  
e.g. optimized/not optimized builds, compiler options, ...

# A word of warning!

Compilers must be **conservative**:



All passes **must preserve the program semantics**



Compiler passes must be designed **very carefully!**

# Contents

Introduction

Compiler organization

**Algorithm design**

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

# Classical Algorithm Design

In algorithm design, a good approach is the following:

1. study the problem
2. make some example
3. identify the **common case**
4. derive the algorithm for the common case
5. add handling for **corner cases**
6. improve performing **optimizing the common case**

Weakness of the approach:

- ▶ **corner cases** – a *correct* algorithm **must** consider *all the corner cases*!

# Compiler Algorithm Design

Corner cases are difficult to handle, but they cannot be ignored

Compiler algorithms must be **proven** to preserve  
program semantic **at all times**

As an aid, a *standard methodology* is employed.

Compiler algorithms are built combining **three** kinds of passes:

Analysis, Optimization, Normalization

# Compiler Algorithm Design

Corner cases are difficult to handle, but they cannot be ignored

Compiler algorithms must be **proven** to preserve  
program semantic **at all times**

As an aid, a *standard methodology* is employed.

Compiler algorithms are built combining **three** kinds of passes:

Analysis, Optimization, Normalization

We now consider a simple example: *loop hoisting*.

# Loop Hoisting

It is a transformation that:

- ▶ looks for statements (inside a loop) not depending on the loop state
- ▶ move them outside the loop body

## Loop Hoisting – Before

```
do {  
    a += i;  
    b = c;  
    i++;  
} while (i < k);
```

## Loop Hoisting – After

```
b = c;  
do {  
    a += i;  
    i++;  
} while (i < k);
```



# Loop Hoisting

## The general idea:

- ▶ move “good” statement outside of the loop

This **pass** modifies the code, thus it is an **optimization pass**.  
It needs to know:

- ▶ which pieces of code are loops
- ▶ which statements are “good” statements

This information is computed by the the **analysis** passes:

- ▶ detecting loops in the program
- ▶ detecting loop-independent statements

The loop hoisting pass declares which analyses it needs:

- ▶ pipeline automatically built: **analysis** → **optimization**

# Loop Hoisting

The **proof** is trivial:

- ▶ the transformation shall preserve program semantics
- ▶ the analyses shall be correct

Analysis passes are usually built starting from other analyses already implemented inside the compiler, or are already present in LLVM

- ▶ often, no proof is necessary for the analyses

## However...

You also have to prove that the combination of analysis + transformation is correct!

"Beware of bugs in the above code;  
I have only proved it correct, not tried it."

— Knuth

# Loop Hoisting

We have spoken about loops, but which kind of loop?

**do-while? while? for?**

Almost all loops are different forms of the **same exact thing**



We can convert a lot of loops to a loop of another kind!

To account for the various kinds of loops, we choose a **normal** kind of loop, and then we write a **normalization** pass.

Usually, **do-while** loops are chosen to be the *normal* loops.

Sometimes, normalization is also called **canonicalization**.

The more loops we recognize, the higher the potential **optimization impact!**

# Compiler Algorithm Design

You have to:

1. analyze the problem
2. make some examples
3. detect the common case
4. determine the **input conditions**
5. determine which **analyses** you need
6. design the **optimization** pass
7. proof its **correctness**
8. improve algorithm performance on the common case
9. improve the effectiveness of the algorithm by adding **normalization passes**

# Compiler Algorithm Design

Something is missing...

## Corner Cases!

Why?

1. It makes no sense to optimize code that is seldom executed
2. Your optimization will be based on **properties of the code that are true only in the common case you are considering**
  - ▶ If the code does not fit the common case, it shall stay as-is
  - ▶ Otherwise you **risk breaking program semantics!**

# Contents

Introduction

Compiler organization

Algorithm design

**Inside LLVM**

LLVM-IR language

Conclusions

Bibliography

# Using LLVM

LLVM is a **compiler construction framework**  
It operates on the **LLVM-IR** language.



Using LLVM *by itself* does not make much sense!  
Writing LLVM-IR by hand is unfeasible.

# Terminology

LLVM-IR comes in 3 different flavours:

**assembly** on-disk human-readable format  
(file extension: `.ll`)

**bitcode** on-disk machine-oriented binary format  
(file extension: `.bc`)

**in-memory** in-memory binary format  
(used during compilation process)

All formats have the same expressiveness!



# Frontends and Drivers

The LLVM-IR input to LLVM is provided by **frontends**.

## Example

**Clang**[3] is the frontend for the C language family

The **compiler driver** is the program that:

- ▶ Provides the interface to the user
- ▶ Performs setup of the front end and LLVM itself.

## Example

The driver of *Clang* is the `cclang` executable (compatible with GCC)

# Frontends and Drivers

We can generate LLVM-IR assembly using the clang driver:

```
clang -emit-llvm -S -o out.ll in.c
```

If you want to generate bitcode instead:

```
clang -emit-llvm -o out.bc in.c
```

The compiler driver can also generate native code starting from  
LLVM-IR assembly

(Like compiling an assembly file with GCC)

# Tools

Run one or more passes on the LLVM-IR on-demand by using `opt`:

- ▶ Syntax is like `clang` (supports even `-O1`, `-O2...`)
- ▶ One command line argument per pass to run
- ▶ Order of execution is the same as the argument order
  - ▶ Different order, different results! (**phase/stage ordering**)

Some useful passes for debugging (they do not transform anything):

```
print CFG opt -view-cfg input.ll
```

```
print dominator tree opt -view-dom input.ll
```

```
print current IR opt -print-module input.ll
```

## Example

- ▶ Run *mem2reg*, then view the CFG:
  - ▶ `opt -mem2reg -view-cfg input.ll`

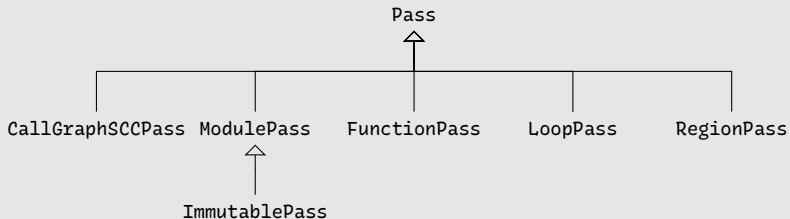
# Pass Hierarchy

LLVM provides a lot of passes...

- Try `opt -help!`

For performance reasons there are different kind of passes:

## LLVM Passes



# LLVM Passes

Each kind of pass visits particular elements of a module:

- ImmutablePass** compiler configuration – never run
- CallGraphSCCPass** post-order visit of CallGraph SCCs
- ModulePass** visit the whole module
- FunctionPass** visit functions
- LoopPass** post-order visit of loop nests
- RegionPass** visit a custom-defined region of code

Specializations come with restrictions:

- ▶ e.g. a **FunctionPass** cannot add or delete functions
- ▶ refer to “Writing a LLVM Pass” [4] for documentation on features and limitations of each kind of pass

# Normalization passes in LLVM

LLVM provides several **normalization/canonicalization** passes:

- ▶ Variable-to-register promotion (`Mem2Reg`)
- ▶ Loop canonicalization (`LoopSimplify`)
- ▶ CFG canonicalization & simplification (`SimplifyCFG`)
- ▶ ...

They are useful to make your life easier!

# Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

**LLVM-IR language**

Conclusions

Bibliography

# LLVM IR

The LLVM IR [5] language is RISC-based:

- ▶ instructions operates on **variables**
  - ▶ aka virtual registers, temporary values
- ▶ only **load** and **store** access memory
- ▶ **alloca** used to reserve memory on the stack

There are also a few **high level instructions**:

- ▶ function call – **call**
- ▶ pointer arithmetics – **getelementptr**
- ▶ ...



# LLVM IR

LLVM IR is **strongly typed**:

- ▶ e.g. you cannot assign a floating point value to an integer variable without an explicit cast

**Almost everything is typed**:

<b>functions</b>	@fact	→	i32 (i32)
<b>statements</b>	%3 = icmp eq i32 %2, 0	→	i1

A variable can be:

<b>global</b>	@var = common global i32 0, align 4
<b>argument</b>	define i32 @fact(i32 %n)
<b>local</b>	%2 = load i32, i32* %1, align 4

Local variables are defined by statements

# LLVM IR

```
define i32 @fact(i32 %n) {  
entry:  
    %retval = alloca i32, align 4  
    %n.addr = alloca i32, align 4  
    store i32 %n, i32* %n.addr, align 4  
    %0 = load i32, i32* %n.addr, align 4  
    %cmp = icmp eq i32 %0, 0  
    br i1 %cmp, label %if.then, label %if.end  
  
if.then:  
    store i32 1, i32* %retval, align 4  
    br label %return  
  
if.end:  
    %1 = load i32, i32* %n.addr, align 4  
    %2 = load i32, i32* %n.addr, align 4  
    %sub = sub nsw i32 %2, 1  
    %call = call i32 @fact(i32 %sub)  
    %mul = mul nsw i32 %1, %call  
    store i32 %mul, i32* %retval, align 4  
    br label %return  
  
return:  
    %3 = load i32, i32* %retval, align 4  
    ret i32 %3  
}
```

# LLVM IR Language

LLVM IR is SSA-based:

- ▶ every variable is **statically assigned** exactly **once**

Statically means that:

- ▶ inside each function...
- ▶ ...for each variable `%foo`...
- ▶ ...there is **only one** statement in the form `%foo = ...`

**Static** (compile time)  $\neq$  **dynamic** (runtime)

- ▶ Single *Dynamic* Assignment:  
*in the execution trace* there is only one assignment to a variable `x`
- ▶ Single *Static* Assignment:  
*in the code listing* there is only one assignment to a variable `x`
  - ▶ Assignments **can** be performed multiple times (in a loop for example)

# Static Single Assignment

## Scalar SAXPY

```
float saxpy(float a, float x, float y) {  
    return a * x + y;  
}
```

## Scalar LLVM SAXPY

```
define float @saxpy(float %a, float %x, float %y) {  
    %1 = fmul float %a, %x  
    %2 = fadd float %1, %y  
    ret float %2  
}
```

Temporary %1 not reused! %2 is used for the second assignment!

# Static Single Assignment

## Array SAXPY

```
void saxpy(float a, float x[4], float y[4], float z[4]) {  
    for(unsigned i = 0; i < 4; ++i)  
        z[i] = a * x[i] + y[i];  
}
```

## Array LLVM SAXPY

```
for.cond:  
    %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]  
    %cmp = icmp ult i32 %i.0, 4  
    br i1 %cmp, label %for.body, label %for.end  
[...]  
for.inc:  
    %inc = add i32 %i.0, 1  
    br label %for.cond
```

One assignment for loop counter %i.0

# Static Single Assignment

## Max

```
float max(float a, float b) {  
    return a > b ? a : b;  
}
```

## LLVM Max – WRONG

```
%1 = fcmp ogt float %a, %b  
br i1 %1, label %if.then, label %if.else  
if.then:  
    %2 = %a  
    br label %if.end  
if.else:  
    %2 = %b  
    br label %if.end  
if.end:  
    ret float %2
```

Why is it **wrong**?

# Static Single Assignment

The %2 variable must be statically assigned once!

## LLVM Max

```
%1 = fcmp ogt float %a, %b
br i1 %1, label %if.then, label %if.end
if.then:
    br label %if.end
if.else:
    br label %if.end
if.end:
    %2 = phi float [ %a, %if.then ], [ %b, %if.else ]
    ret float %2
```

The **phi** instruction is a *conditional move*:

- ▶ it takes  $(\text{variable}_i, \text{label}_i)$  pairs
- ▶ if coming from predecessor identified by  $\text{label}_i$ , its value is  $\text{variable}_i$

# Static Single Assignment

Each SSA variable is assigned only once:

- ▶ variable **definition**

Each SSA variable can be referenced by multiple instructions:

- ▶ variable **uses**

Algorithms and technical language abuse of these terms!

*Let %foo be a variable. If the definition of %foo does not have side-effects nor uses, the aforementioned %foo variable can be erased from the CFG without altering program semantics.*



# Static Single Assignment

Old compilers are not SSA-based:

- ▶ converting non-SSA input into SSA form is expensive
- ▶ cost must be amortized

New compilers are SSA-based:

- ▶ SSA easier to work with
- ▶ SSA-based analysis/optimizations are faster

# Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

**Conclusions**

Bibliography

# Conclusions

LLVM is a **production-quality** compiler framework:

⇒ impossible knowing all details

But:

- ▶ it is well organized
- ▶ if you know compiler theory, it is relatively easy to find what you need inside the source code

Remember it's written in C++!

- ▶ To hack around LLVM you need at least basic C++ skills
- ▶ C++  $\neq$  C

# Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

**Bibliography**

# Bibliography I



University of Illinois at Urbana-Champaign.

Low Level Virtual Machine.

<http://www.llvm.org>.



GNU.

GNU Compiler Collection.

<http://gcc.gnu.org>.



University of Illinois at Urbana-Champaign.

Clang: a C language family frontend for LLVM.

<http://clang.llvm.org>.



Chris Lattner and Jim Laskey.

Writing an LLVM Pass.

<http://llvm.org/docs/WritingAnLLVMPass.html>.

# Bibliography II



Chris Lattner and Vikram Adve.  
LLVM Language Reference Manual.  
<http://lvm.org/docs/LangRef.html>.



Linus Torvalds.  
Re: SCO: "thread creation is about a thousand times faster than  
onnative."  
<https://lkml.org/lkml/2000/8/25/132>.



Bruce Eckel.  
Thinking in C++ – Volume One: Introduction to Standard C++.  
<http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>.



Bruce Eckel and Chuck Allison.  
Thinking in C++ – Volume Two: Practical Programming.  
<http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>.

# Bibliography III



AMD.

Open64.

<http://developer.amd.com/tools-and-sdks/cpu-development/x86-open64-compiler-suite>.



John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner.

LLVM Testing Infrastructure Guide.

<http://llvm.org/docs/TestingGuide.html>.



LLVM Community.

LLVM Coding Standards.

<http://llvm.org/docs/CodingStandards.html>.



LLVM Community.

LLVM Passes.

<http://llvm.org/docs/Passes.html>.

# Bibliography IV



LLVM Community.  
Autovectorization in LLVM.  
<http://llvm.org/docs/Vectorizers.html>.



LLVM Community.  
LLVM Programmer's Manual.  
<http://llvm.org/docs/ProgrammersManual.html>.



Ettore Speziale.  
Compiler Optimization and Transformation Passes.  
<https://github.com/speziale-ettore/COTPasses>.



Scott Chacon.  
Pro Git.  
<http://git-scm.com/book>.