Introducing LLVM

Ettore Speziale

Politecnico di Milano

Contents

Contents

Welcome

Introducing the Lab

What will we see? How to ...

- play with compilers
- design compiler algorithms
- implement algorithms inside a production-quality compiler

A production-quality compiler?

- of course – toy compilers are almost useless!

Welcome
Toy vs Production-Quality

Don't be afraid from a production-quality compiler:

Toy Compiler

- small code-base
- easy doing tiny edits
- impossible doing normal/big edits

Production-Quality Compiler

- huge code-base
- difficult performing any kind of edits
- compiler-code extremely optimized

Key concepts:

- working with a production-quality compiler is initially hard, but . . .
- . . . an huge set of tools for analyzing/transforming/testing code is provided – toy compilers miss these things!

Initially started as a research project at Urbana-Champaign:

- now intensively used for researches involving compilers
- key technology for leading industries – AMD, Apple, Intel, NVIDIA, . . .

If you are there, then it is your key-technology:

- open-source compilers: Open64 [6], GCC [5], LLVM [10]
- LLVM is young – GCC performances are better –, but . . .
- . . . it is kept *clean* by developers – easier working with it

What About the Exam?

To get a pass grade:

- oral test – Professor Crespi
- LLVM-based homework with short presentation – me

During the lab we will see some examples:

- checkout the examples repository [11]

Examples distributed as an LLVM-based project:

- please start from it
- please version sources – Git tutorial here [1]

Note: LLVM is written in C++ [3, 4]:

- you can follow "Principi dei Linguaggi di Programmazione" lab classes for an intro to C++

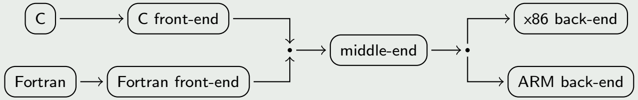
Contents

How Does a Compiler Works?

Recalling Formal Languages and Compilers

A compiler is just a pipeline:

Compiler Pipeline



Three main components:

- **Front-end** take and input file, translate it to an intermediate representation
- **Middle-end** analyze intermediate representation, optimize it
- **Back-end** take intermediate representation, translate it into target machine assembly

Inside {Front,Middle,Back}-ends there are sub-pipelines:

- simple model of computations: read something, produce something
- only needed to specify how to transform input data into output data

Complexity lies on chaining together stages

Terminology

Intermediate Representation and Pass

From now on, we will consider only the middle-end:

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

Let me introduce:

Pass a pipeline stage is called pass

IR Intermediate Representation is the language describing data read/written by passes. Usually, inside middle-ends only one kind of IR is used

Given a set of passes, the pass manager:

- build the compilation pipeline – schedule –, by chaining passes together according to dependencies

Dependencies are hints:

- advise pass manager about passes scheduling

First Insights

A compiler is complex:

- passes are the elementary unit of work
- pass manager must be advisee about pass chaining
- pipeline shapes are not fixed – it can change from one compiler execution to another [1]

Moreover, compilers must be conservative:

- apply a transformation only if program semantic is preserved

Compiler algorithms are designed differently!

[1] e.g. optimized/not optimized builds

Contents

Usually, you [2] act like this:

1. study the problem
2. make some examples
3. identify the common case
4. sketch a first algorithm for the common case
5. consider corner cases
6. improve algorithm performance by optimizing the common case

Weakness of the approach:

- corner cases

A correct algorithm must consider all corner cases!

[2] For sure me

Corner cases are difficult to handle:

- compiler algorithms must be proved to preserve program semantic
- having a common methodology helps on that

Compiler algorithms are built combining three kind of passes:

- analysis
- optimization
- normalization

Let me take *loop hoisting* as a simple example

It is a transformation that:

1. look for statements not depending on loop state
2. move them outside of the loop

Loop Hoisting – Before

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

Loop Hoisting – After

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

Loop Hoisting

Focus on the Transformation

The transformation is trivial:

- move "good" statement outside of the loop

This is the optimization pass. It needs to known:

- loops
- "good" statements

They are analysis passes:

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

When registering loop hoisting, also declare needed analysis:

- pipeline automatically built – analysis → optimization

The proof is trivial:

- transformation is correct if analysis are correct, but ...
- ... usually analysis are built starting from other analysis already implemented inside the compiler

You have to prove that combining all analysis information gives you a correct view of the code:

- analysis information cannot induce optimization passes applying a transformation not preserving program semantic

Loop Hoisting

More Loops

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

- **do** loops?
- **while** loop?
- **for** loops?

Loop hoisting only works with one kind of loop:

- **while** loops

What about other kinds of loops?

- they must be normalized – i.e. transformed to **while** loops

Normalization passes do that:

- before running loop hoisting, you must tell pass manager loop normalization must be run before

This allows to recognize more loops, thus potentially improving optimization impact!

You have to:

1. analyze the problem
2. make some examples
3. detect the common case
4. declare the input format
5. declare analysis you need
6. design an optimization pass
7. proof its correctness
8. improve algorithm perfomance by acting on common case – the only considered up to now. Please notice that corner cases are not considered – just do not optimize
9. improve the effectiveness of the algorithm by adding normalization passes

Contents

Introducing LLVM
Ettore Speziale
Introduction
Compiler Organization
Compiler Algorithms
LLVM Quick Tour
Conclusions
Bibliography

LLVM tools work with modules:

- lists of global objects

A global object can be:

- a type declaration
- a variable declaration
- a function

A functions is:

- a list of basic blocks

A basic block is:

- a list of statements

Please notice that in LLVM a lot of things are just lists!

LLVM IR language [7] is RISC-based:

- instructions operates on variables [3]
- only **load** and **store** access memory
- **alloca** used to reserve memory on function stacks

Factorial – 1

```
define i32 @fact(i32 %n) nounwind {
  %1 = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  %2 = load i32* %1, align 4
  %3 = icmp eq i32 %2, 0
  br i1 %3, label %4, label %5

; <label>:4
  br label %11
```

[3] Virtual registers

Introducing LLVM
Ettore Speziale
Introduction
Compiler Organization
Compiler Algorithms
LLVM Quick Tour
Conclusions
Bibliography

Factorial – 2

```
; <label>:5
  %6 = load i32* %1, align 4
  %7 = load i32* %1, align 4
  %8 = sub i32 %7, 1
  %9 = call i32 @fact(i32 %8)
  %10 = mul i32 %6, %9
  br label %11

; <label>:11
  %12 = phi i32 [ 1, %4 ], [ %10, %5 ]
  ret i32 %12
}
```

In addition, some high level instructions:

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

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 – e.g.:

functions	@fact – i32 (i32)
statements	%3 = **icmp eq** i32 %2, 0 – i1

Notice that a variable is:

global	@var = **common global** i32 0, **align** 4
function parameter	**define** i32 @fact(i32 %n) **nounwind**
local	%2 = **load** i32* %1, **align** 4

Local variables are defined by statements

Terminology
Speaking About LLVM IR

LLVM IR comes with 3 different flavours:

assembly	human-readable format
bitcode	binary on-disk machine-oriented format
in-memory	binary in-memory format, used during compilation process

All formats have the same expressiveness!

File extensions:

.ll	for assembly files
.bc	for bitcode files

Writing LLVM assembly by hand is unfeasible:

- different front-ends available for LLVM
- use clang [9] for the C family

The clang driver is compatible with GCC:

- same command line options

To generate LLVM IR:

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

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

It can also generate native code starting from LLVM assembly or LLVM bitcode – like compiling an assembly file with GCC

LLVM IR can be manipulated using `opt`:

- read an input file
- run specified LLVM passes on it
- respecting user-provided order

Useful passes:

- print CFG with `opt -view-cfg input.ll`
- print dominator tree with `opt -view-dom input.ll`
- ...

Pass chaining:

- run *mem2reg* [4], then view the CFG with `opt -mem2reg -view-cfg input.ll`

[4] More on this later

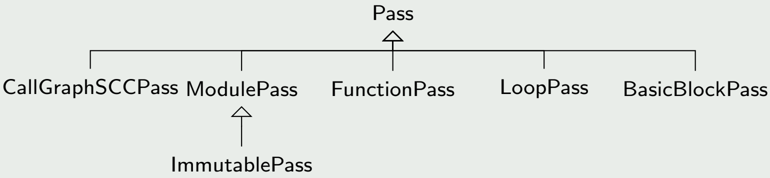
LLVM Passes
Start Looking at Code

LLVM provides a lot of passes:

- try `opt -help`

For performance reasons there are different kind of passes:

LLVM Pass Hierarchy [5]



See [8] for an intro

[5] Forget about RegionPass

LLVM Passes

Each pass kind 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
- **BasicBlockPass** visit basic blocks

Specializations comes with restrictions:

- e.g. a FunctionPass cannot add or delete functions
- refer to [8] for accurate description of features and limitations of each kind of pass

Your LLVM Pass
Sources

Showing code on slides is both boring and error-prone, so I will use as much as possible vi *and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.*

"Talk is cheap, show me the code" [12]

The passes we will see are very simple:

- some of them are meaningless
- goal is to show you the LLVM API

Each pass is "tested" using the LLVM testing framework [2]:

- look at the test subdirectory

Look at the following passes:

instruction-count
: simple instruction counting analysis

hello-llvm
: optimization pass building an hello-world program

function-eraser
: optimization pass removing “small” functions

Please take the LLVM pass writing tutorial [8]

Contents

Conclusions

LLVM is a production-quality compiler:

- impossible knowing all details

But:

- is well organized
- if you known compilers theory is easy finding what you need inside sources

Please take into account C++:

- basic skills required

Contents

Bibliography I

Scott Chacon.
Pro Git.
https://github.com/speziale-ettore/COTPasses.

John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner.
LLVM Testing Infrastructure Guide.
http://llvm.org/releases/3.0/docs/TestingGuide.html.

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

Bibliography II

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

GNU.
GNU Compiler Collection.
http://gcc.gnu.org.

Open64 Steering Group.
Open64.
http://www.open64.net.

Chris Lattner and Vikram Adve.
LLVM Language Reference Manual.
http://llvm.org/releases/3.0/docs/LangRef.html.

Bibliography III

Chris Lattner and Jim Laskey.
Writing an LLVM Pass.
http://llvm.org/releases/3.0/docs/WritingAnLLVMPass.html.

University of Illinois at Urbana-Champaign.
clang: a C language family frontend for LLVM.
http://clang.llvm.org.

University of Illinois at Urbana-Champaign.
Low Level Virtual Machine.
http://www.llvm.org.

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

Bibliography IV

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