Building High-Level Compiler Optimizers and Code Generators for the Multicore Era

Introduction to LLVM compiler framework

Stefano Cherubin

Politecnico di Milano

12-12-2016

These slides contain material by M. Scandale and E. Speziale
About the dragon

- The LLVM logo [1] is a stylized wyvern (mythological creature often mistaken for a dragon). Dragons have connotations of power, speed and intelligence, and can also be sleek, elegant, and modular (err, maybe not).
About the dragon

- The LLVM logo [1] is a stylized wyvern (mythological creature often mistaken for a dragon). Dragons have connotations of power, speed and intelligence, and can also be sleek, elegant, and modular (err, maybe not).

- There is a series of compiler books dating back to the 1970s showing illustrations with dragons and knights [2] [3] [4]
About me

Stefano Cherubin

- stefano.cherubin@polimi.it
- 2nd year PhD student @ Politecnico di Milano (Italy)
- working on compilers since relatively short time
- definitely not an experienced knight...
About me

Stefano Cherubin
- stefano.cherubin@polimi.it
- 2nd year PhD student @ Politecnico di Milano (Italy)
- working on compilers since relatively short time
- definitely not an experienced knight...
- ...I’m more like a lazy Hobbit
In order to fully understand the content of this course you should have:

- knowledge of what a compiler is
- proficiency in most common data structures
- proficiency in Object-Oriented Programming
- at least some experience with C++
About this talk

- Compiler design
- LLVM structure overview
- LLVM-IR language
- LLVM Documentation
- LLVM quick start
Goal of this talk

At the end of this speech you should:

- understand the LLVM compiler infrastructure
- be able to read a .ll file (LLVM-IR)
- know where to look for documentation
- navigate the LLVM code base
Contents

1 Introduction

2 Compiler organization

3 Algorithm design

4 Inside LLVM

5 LLVM-IR language

6 Documentation

7 Using the LLVM framework: quick start

8 Conclusions
Compilers and compilers

Approaching to compilers, we need to understand the difference between a *toy-compiler* and *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 [5], GCC [6], LLVM [7]
- LLVM is relatively young – GCC performances are better – but . . .
- . . . it is highly modular, well written, kept clean by developers.
Contents

1 Introduction

2 Compiler organization

3 Algorithm design

4 Inside LLVM

5 LLVM-IR language

6 Documentation

7 Using the LLVM framework: quick start

8 Conclusions
Typically a compiler is a pipeline:

- **Front-end** translate a source file in the intermediate representation
- **Middle-end** analyze intermediate representation, optimize it
- **Back-end** generate target machine assembly from the intermediate representation
Compiler pipeline

Internal pipelines

Each component is composed internally by 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.
Compiler pipeline

We will consider only the *middle-end*: same concepts are valid also for \{front,back\}-end.

Technical terms:

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

The **pass manager** manages a set of passes:

- build the compilation pipeline: **schedule** passes together according to **dependencies**.

Dependencies are **hints** used by the pass manager in order to schedule passes.
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 w.r.t. standard algorithms!

\(^1\)e.g. optimized/not optimized builds, compiler options, ...
Contents

1 Introduction

2 Compiler organization

3 Algorithm design

4 Inside LLVM

5 LLVM-IR language

6 Documentation

7 Using the LLVM framework: quick start

8 Conclusions
Dealing with algorithm design, a naive 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 performance optimizing the common case

Weakness of the approach:

- corner cases — a correct algorithm must consider all the corner cases!
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
- normalization
- optimization

We now consider a simple example: loop hoisting.
Loop Hoisting

It is a transformation that:

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

```c
Loop Hoisting – Before

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

```c
Loop Hoisting – After

b = c;

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

The transformation is trivial:
- move “good” statement outside of the loop

This is the optimization pass. It needs to know:
- which pieces of code are loops
- which statements are “good” statements

They are analysis passes:
- detecting loops in the program
- detecting loop-independent statements

When registering loop hoisting, also declare needed analysis:
- pipeline is then automatically built: analysis $\rightarrow$ 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-while loops?
- while loop?
- for loops?

We have seen loop hoisting on:

- do-while loops

What about other kinds of loops?

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

Normalization passes do that:

- before running loop hoisting, you must tell to the pass manager that 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 performance by acting on common case – the only considered up to now. Please notice that corner cases are not considered – just do not optimize corner cases
9. improve the effectiveness of the algorithm by adding normalization passes
Contents

1 Introduction
2 Compiler organization
3 Algorithm design
4 Inside LLVM
5 LLVM-IR language
6 Documentation
7 Using the LLVM framework: quick start
8 Conclusions
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 [8] for the C family

The clang driver is compatible with GCC:

- \( \approx \) 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, then view the CFG with
  opt -mem2reg -view-cfg input.ll
- potentially different results using different option order
  (phase/stage ordering)
Pass Hierarchy

LLVM provides a lot of passes:

- try opt -help

For performance reasons there are different kind of passes:

```
Pass
  CallGraphSCCPass
    ModulePass
      ImmutablePass
  FunctionPass
  LoopPass
  BasicBlockPass
```
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 come with restrictions:

- e.g. a **FunctionPass** cannot add or delete functions
- refer to [9] for accurate description of features and limitations of each kind of pass
What is Available Inside LLVM?

 LLVM provides passes performing basic transformations:
  - variables promotion
  - loops canonicalization
  - ... 

They can be used to normalize/canonicalize the input:
  - transform into a form analyzable for further passes

Input normalization is essential:
  - keep passes implementation manageable
Contents

1 Introduction
2 Compiler organization
3 Algorithm design
4 Inside LLVM
5 LLVM-IR language
6 Documentation
7 Using the LLVM framework: quick start
8 Conclusions
LLVM IR [10] language is RISC-based:

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

There are also few high level instructions:

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

\[2\text{Virtual registers}\]
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
```

A variable can be:

```
global  @var = common global i32 0, align 4

function parameter  define i32 @fact(i32 %n)

local  %2 = load i32, i32* %1, align 4
```

Local variables are defined by statements
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

Static Single Assignment

LLVM IR is SSA-based:

- every variable is \textit{statically assigned} exactly once

Statically means that:

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

Static is different from dynamic:

- a static assignment can be executed more than once
Scalar SAXPY

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

Scalar LLVM SAXPY

```llvm
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!
## Array SAXPY

```c
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

```llvm
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 \( \%i.0 \)
Max

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

LLVM Max – Bad

```llvm
%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 bad?
The %2 variable must be statically set once

LLVM Max

```llvm
%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 \((variable_i, label_i)\) pairs
- if coming from predecessor identified by \(label_i\), its value is \(variable_i\)
Static Single Assignment
Definition and Uses

Each SSA variable is set only once:

- variable definition

Each SSA variable can be used by multiple instructions:

- variable uses

Algorithms and technical language abuse of these terms.

Let \( \%\text{foo} \) be a variable. If \( \%\text{foo} \) definition has not side-effects, and no uses, dead-code elimination can be efficiently performed by erasing \( \%\text{foo} \) definition from the CFG.
Static Single Assignment

Rationale

Old compilers are not SSA-based:
- putting input into SSA-form is expensive
- cost must be amortized

New compilers are SSA-based:
- SSA easier to work with
- SSA-based analysis/optimizations faster

All modern compilers are SSA-based:
- exception are old version of the HotSpot Client compiler
Contents

1 Introduction

2 Compiler organization

3 Algorithm design

4 Inside LLVM

5 LLVM-IR language

6 Documentation

7 Using the LLVM framework: quick start

8 Conclusions
LLVM official documentation

llvm.org/docs
A lot of documentation...

llvm.org/docs mentions:

- 5 references about Design & Overview
- 19 references about User Guides
- 12 references about Programming Documentation
- 32 references about Subsystem Documentation
- 7 references about Development Process Documentation
- 5 Mailing Lists
- 5 IRC bots

Most of the above references are OUT-OF-DATE.

You probably need documentation about the documentation itself.
**Essential documentation**

*Intro to LLVM* [11] gives a quick and clear introduction to the compiler infrastructure. It is mostly up-to-date.

*Writing an LLVM pass* [9] explains step by step how to implement a Pass for those who never did anything like that.

*Doxygen* [12] *The best code documentation is the code itself.* Sometimes the generated doxygen documentation is enough. It also contains links to the web version of the source code. It is always up-to-date.

*llvm-dev* Mailing List. Last resource: ask other developers. Warning: 24/7 many people are posting in this ML.

---

3 at the time I am writing
Contents

1 Introduction
2 Compiler organization
3 Algorithm design
4 Inside LLVM
5 LLVM-IR language
6 Documentation
7 Using the LLVM framework: quick start
8 Conclusions
Commands

llvm-as  LLVM assembler
llvm-dis  LLVM disassembler
opt   LLVM optimizer
llc   LLVM static compiler
lli   directly execute programs from LLVM bitcode
llvm-link  LLVM bitcode linker
llvm-lib  LLVM lib.exe compatible library tool
llvm-nm  list LLVM bitcode and object file’s symbol table
llvm-config  Print LLVM compilation options
llvm-stress  generate random .ll files
llvm-dwarfdump  print contents of DWARF sections

For a complete reference, see LLVM command guide ⁴

.c source

```
<table>
<thead>
<tr>
<th>gcc / clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bc / .ll</td>
</tr>
<tr>
<td>llvm-link</td>
</tr>
<tr>
<td>.bc / .ll</td>
</tr>
<tr>
<td>opt</td>
</tr>
<tr>
<td>.bc / .ll</td>
</tr>
<tr>
<td>llc</td>
</tr>
<tr>
<td>.s</td>
</tr>
<tr>
<td>llvm-mc / as</td>
</tr>
<tr>
<td>.o</td>
</tr>
<tr>
<td>ld</td>
</tr>
<tr>
<td>dynLibWhatever.o</td>
</tr>
<tr>
<td>executable</td>
</tr>
</tbody>
</table>
```

```
| .ll          |
| llvm-as      |
| .bc          |
```

Stefano Cherubin <stefano.cherubin@polimi.it>
Writing a LLVM pass

There are a lot of tutorials available:

- Official developer guide
  llvm.org/docs/WritingAnLLVMPass

- Out-of-source pass
  github.com/quarkslab/llvm-dev-meeting-tutorial-2015
Testing

LLVM has an internal testing infrastructure. ³ Please use it.

`llvm-lit` LLVM Integrated Tester

1. Forge a proper LLVM-IR input file (.ll) for your test case
2. Instrument it with `lit` script comments
3. Run `lit` on your test
   - `llvm-lit /llvm/test/myTests/singleTest.ll`
     run a single test
   - `llvm-lit /llvm/test/myTests`
     run the test suite (folder)
4. Run `lit` on the LLVM test suite (regression testing)

To submit a bug report to LLVM developers you will be asked to write a `lit` test case that highlights the bug.

³http://llvm.org/docs/TestingGuide.html
Contents

1 Introduction
2 Compiler organization
3 Algorithm design
4 Inside LLVM
5 LLVM-IR language
6 Documentation
7 Using the LLVM framework: quick start
8 Conclusions
Understanding LLVM

LLVM is not a compiler.
Understanding LLVM

LLVM is not a compiler.

LLVM is a collection of components which is useful to build a compiler.
What LLVM is made of

- C++ libraries
  - src/include/llvm/...
  - src/lib/...

- small application (tools)
  - src/tools/...
  - src/utils/...

You can find binaries of them in the installation directory under root/bin/...
clang is a compiler based on LLVM.

It compiles all major C-like languages.

It can be added as a tool in the LLVM framework but must be manually cloned in the tool directory:

1. cd src/tools
2. git clone http://llvm.org/git/clang

You can easily see on a production quality compiler the impact of changes you made on your local copy of LLVM.
Conclusions

LLVM is a production-quality compiler framework:

⇒ 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
Bibliography I

Apple Inc.
Llvm logo.
http://llvm.org/Logo.html.

Alfred V. Aho and Jeffrey D. Ullman.
Principles of Compiler Design (Addison-Wesley Series in Computer
Science and Information Processing).
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA,
1977.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA,
1986.
Bibliography II


Bibliography III

- **University of Illinois at Urbana-Champaign.**
  Clang: a C language family frontend for LLVM.

- **Chris Lattner and Jim Laskey.**
  Writing an LLVM Pass.

- **Chris Lattner and Vikram Adve.**
  [http://llvm.org/docs/LangRef.html](http://llvm.org/docs/LangRef.html).

- **Chris Lattner.**
  Intro to LLVM.
Bibliography IV

LLVM Community.
Doxygen annotations.

Think Geek.
Relaxing with a pipe full.

Linus Torvalds.
Re: SCO: "thread creation is about a thousand times faster than onnative.

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

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

John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner. 
LLVM Testing Infrastructure Guide. 

LLVM Community. 
LLVM Coding Standards. 
http://llvm.org/docs/CodingStandards.html.

LLVM Community. 
LLVM Passes. 
http://llvm.org/docs/Passes.html.
Bibliography VI

LLVM Community.
Autovectorization in LLVM.

LLVM Community.

Ettore Speziale.
Compiler Optimization and Transformation Passes.

Scott Chacon.
Pro Git.