Introduction to LLVM compiler framework

Stefano Cherubin

Politecnico di Milano

06-05-2016

This material is strongly based on material produced by Michele Scandale and Ettore Speziale for the course ‘Code Optimizations and Transformations’.
Contents

1 Introduction
2 Compiler organization
3 Algorithm design
4 Inside LLVM
5 LLVM-IR language
6 Conclusions
7 Bibliography
### Compilers and compilers

Approaching to compilers, we need to understand the difference between a *toy-compiler* and *production-quality compiler*.

<table>
<thead>
<tr>
<th>Toy Compiler</th>
<th>Production-Quality Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>small code-base</td>
<td>huge code-base</td>
</tr>
<tr>
<td>easy doing tiny edits</td>
<td>difficult performing any kind of edits</td>
</tr>
<tr>
<td>impossible doing normal/big edits</td>
<td>compiler-code extremely optimized</td>
</tr>
</tbody>
</table>

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 [1], GCC [2], LLVM [3]
- 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 Conclusions

7 Bibliography
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
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.
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!

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

1. Introduction
2. Compiler organization
3. Algorithm design
4. Inside LLVM
5. LLVM-IR language
6. Conclusions
7. Bibliography
Dealing with 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 performance by 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
- optimization
- normalization

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

**Loop Hoisting – Before**

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

**Loop Hoisting – After**

```plaintext
b = c;
do {
    a += i;
    i++;
} while (i < k);
```
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-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
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 Conclusions
7 Bibliography
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 [4] 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`, then view the CFG with `opt -mem2reg -view-cfg input.ll`

^2More on this later
LLVM provides a lot of passes:

- try opt -help

For performance reasons there are different kind of passes:

**LLVM Passes**

![Pass Hierarchy Diagram]

- CallGraphSCCPass
- ModulePass
- FunctionPass
- LoopPass
- BasicBlockPass
- ImmutablePass

Stefano Cherubin <stefano.cherubin@polimi.it>
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 [5] for accurate description of features and limitations of each kind of pass
Introduction to LLVM compiler framework

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
Introduction to LLVM compiler framework

Contents

1 Introduction
2 Compiler organization
3 Algorithm design
4 Inside LLVM
5 LLVM-IR language
6 Conclusions
7 Bibliography
LLVM IR [6] 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`
- ...

---

3Virtual 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 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
Introduction to LLVM compiler framework

Static Single Assignment

Examples

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
Introduction to LLVM compiler framework

Static Single Assignment
Handling Multiple Assignments

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 \(variable_i\)
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 %foo be a variable. If %foo definition has not side-effects, and no uses, dead-code elimination can be efficiently performed by erasing %foo definition from the CFG.
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 Conclusions
7 Bibliography
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
Contents

1 Introduction

2 Compiler organization

3 Algorithm design

4 Inside LLVM

5 LLVM-IR language

6 Conclusions

7 Bibliography
Introduction to LLVM compiler framework

Bibliography I

- **AMD.**
  Open64.

- **GNU.**
  GNU Compiler Collection.

- **University of Illinois at Urbana-Champaign.**
  Low Level Virtual Machine.

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

Chris Lattner and Jim Laskey.
Writing an LLVM Pass.

Chris Lattner and Vikram Adve.
http://llvm.org/docs/LangRef.html.

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 III


Bibliography IV