Introduction to LLVM compiler framework

Stefano Cherubin

Politecnico di Milano

03-05-2019

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

Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

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**!

LLVM: Low Level Virtual Machine

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: GCC [**?**], LLVM [**?**]
- LLVM is relatively **young** – GCC performances may be better – …
- … LLVM is more modular, well written, kept *clean* by developers.

Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

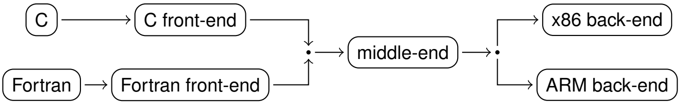
LLVM-IR language

Conclusions

Bibliography

Compiler pipeline

Typically a compiler is a **pipeline**:



There are three main components:

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

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

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

Compiler pipeline

Each component is composed internally by pipelines:

- simple model – read something, produce something
- specify only 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

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

Classical Algorithm Design

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 performancing **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:

- 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)

Compiler Algorithm Design

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 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

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**, which have to be implemented by other passes:

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

When registering loop hoisting, also declare needed analysis:

- pipeline automatically built: **analysis** $\rightarrow$ **optimization**

Loop Hoisting

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

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**!

Compiler Algorithm Design

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 try to optimize the corner cases
9. improve the effectiveness of the algorithm by adding **normalization passes**

Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

Terminology

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

Tools

Writing LLVM assembly by hand is unfeasible:

- different front-ends available for LLVM
- use Clang [**?**] 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

Tools

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**)

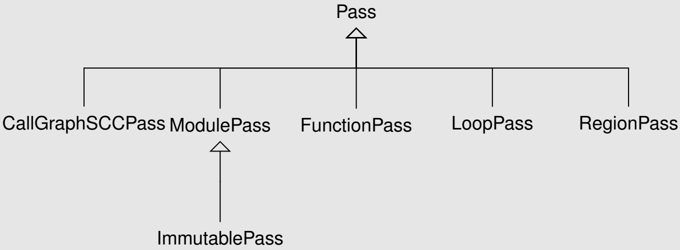
Pass Hierarchy

LLVM provides a lot of passes:

- try `opt -help`

For performance reasons there are different kind of passes:

LLVM Passes



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
RegionPass	visit a custom-defined region of code

Specializations comes with restrictions:

- e.g. a **FunctionPass** cannot add or delete functions
- refer to "Writing a LLVM Pass" [**?**] 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
- it is essential because keeps passes implementation manageable

Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

LLVM IR

LLVM IR [**?**] language is RISC-based:

- instructions operates on **variables** [2]
- 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] Virtual registers

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** – 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

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
```

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 is different from dynamic:

- a static assignment can be executed more than once

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:
```

Static Single Assignment

Max

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

LLVM Max – Bad

```
  %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:
```

Static Single Assignment

The `%2` variable must be statically set 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 $(variable_i, label_i)$ pairs
- if coming from predecessor identified by $label_i$, its value is $variable_i$

Static Single Assignment

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.

Static Single Assignment

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

Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

Conclusions

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

$\Rightarrow$ impossible knowing all details

But:

- it is well organized
- given you known compilers theory, it is relatively easy to find what you need inside its sources

Please take into account C++:

- basic skills required

Contents

Introduction

Compiler organization

Algorithm design

Inside LLVM

LLVM-IR language

Conclusions

Bibliography

Bibliography I