An Introduction to LLVM Infrastructure

高庆

2014.10.08

About LLVM

- LLVM： **L**ow-**l**evel **v**irtual **m**achine

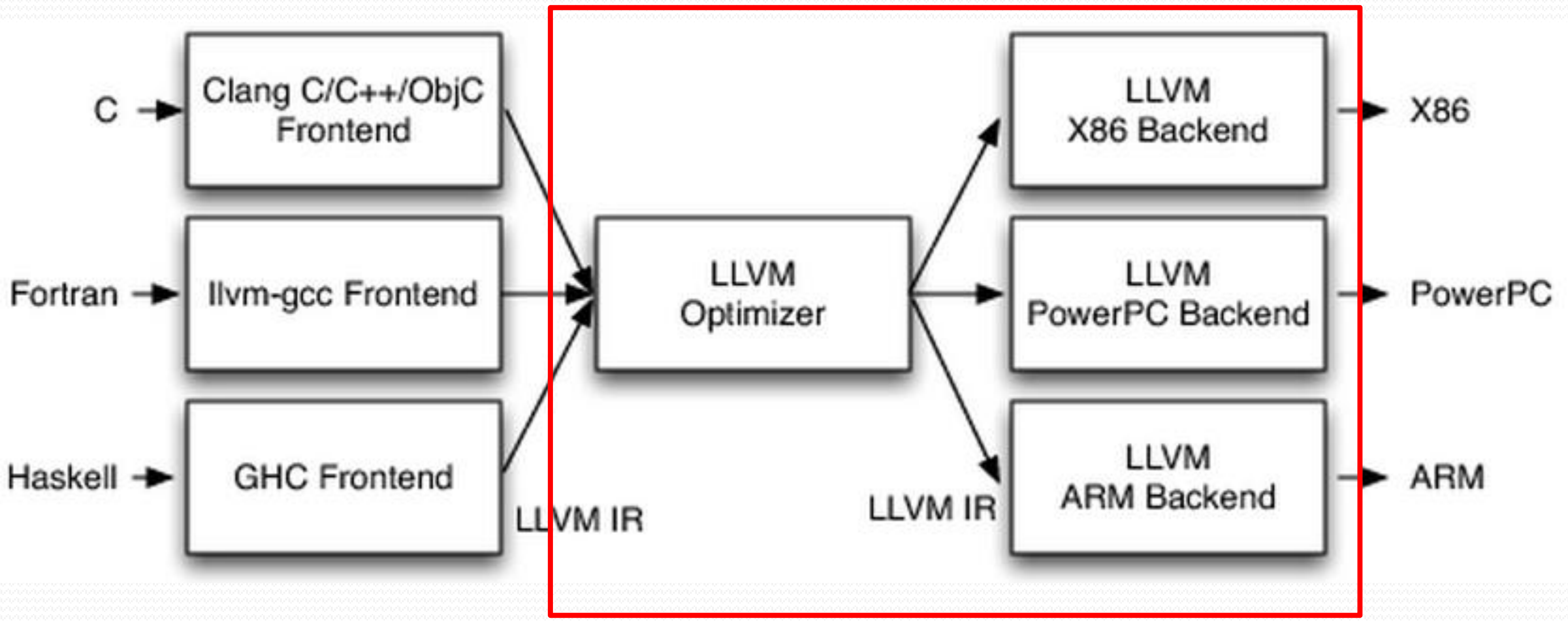
- A framework for writing compilers (including tools for static analysis)

- Written in C++

- Main author: Chris Lattner

LLVM IR: Intermediate Representation

- Input to LLVM



Example 1

```c
#include <stdio.h>

int main(){
  printf("Hello World!\n");
  return 0;
}
```

```llvm
; ModuleID = 'hello.o'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  %2 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([14 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.4 (tags/RELEASE_34/final)"}
```

Example 2

- Partial-SSA form

```c
void f(){
  int *p;
  int *q = malloc(1);
  int *r = malloc(2);
  p = q;
  p = r;
  *p = 1;
}
```

```llvm
define void @f() #0 {
entry:
  %call = call i8* @malloc(i64 1)
  %0 = bitcast i8* %call to i32*
  %1 = bitcast i32* %0 to i32*
  %call1 = call i8* @malloc(i64 2)
  %2 = bitcast i8* %call1 to i32*
  %3 = bitcast i32* %2 to i32*
  %4 = bitcast i32* %3 to i32*
  store i32 1, i32* %4, align 4
  ret void
}
```

Example 3

```c
size1 = strlen (s1);
size2 = strlen (s2);
ret_val = xmalloc (size1 + size2 + 1);
strcpy (ret_val, s1);
strcpy (&ret_val[size1], s2);
return ret_val;
```

```llvm
%call = call i64 @strlen(i8* %s1.addr.0)
%conv = trunc i64 %call to i32
%4 = bitcast i32 %conv to i32
%call4 = call i64 @strlen(i8* %s2.addr.0)
%conv5 = trunc i64 %call4 to i32
%5 = bitcast i32 %conv5 to i32
%add = add nsw i32 %4, %5
%add6 = add nsw i32 %add, 1
%call7 = call i8* @xmalloc(i32 %add6)
%6 = bitcast i8* %call7 to i8*
%call8 = call i8* @strcpy(i9* %6, i8* %s1.addr.0)
%idxprom = sext i32 %4 to i64
%arrayidx = getelementptr inbounds i8* %6, i64 %idxprom
%call9 = call i8* @strcpy(i8* %arrayidx, i8* %s2.addr.0)
ret i8* %6
```

How do LLVM work – analyzed object

- Input: IR
  - Analyzing unit: Module

- Modules can be combined to a larger module
  - Useful for linking

How do LLVM work - implementation

- Composition
  - Header files

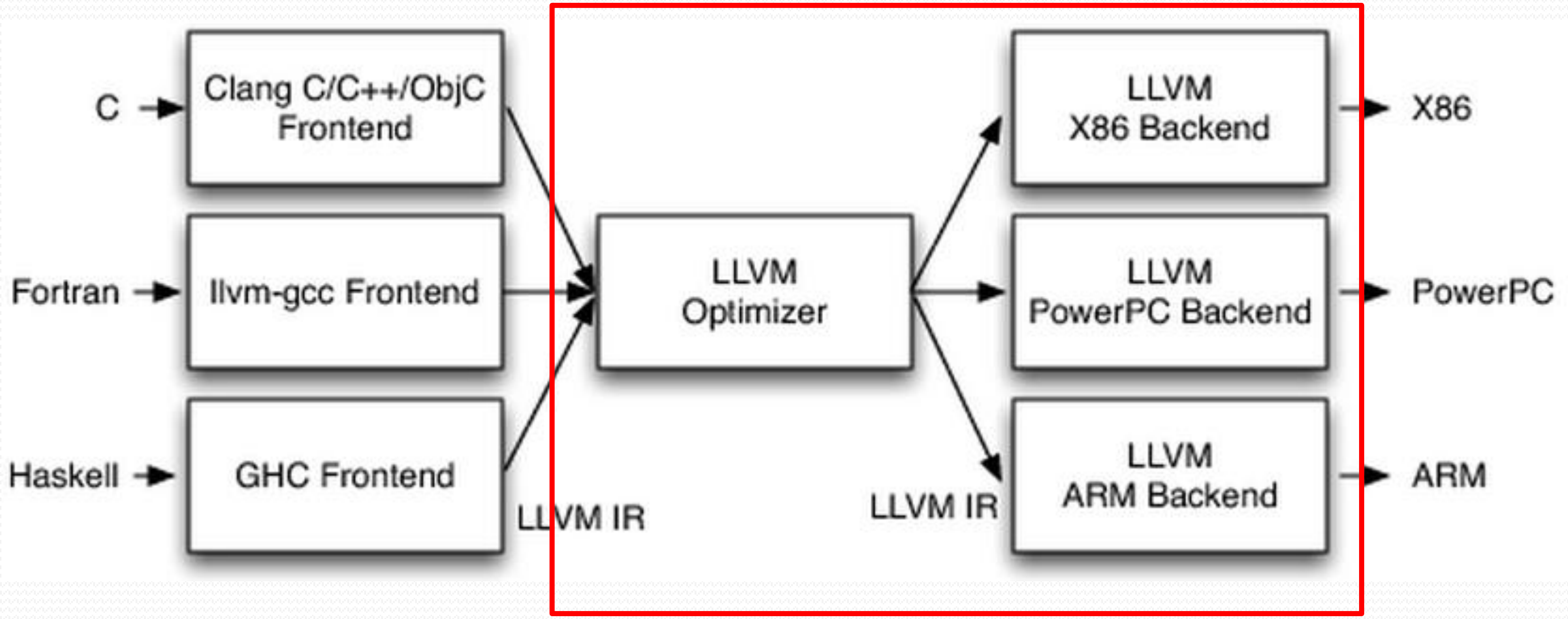
```
root@ubuntu: /mnt/data/llvm/include/llvm# ls
ADT             CodeGen          GVMaterializer.h    LinkAllPasses.h  PassAnalysisSupport.h  Support
Analysis        Config           InitializePasses.h  Linker.h         Pass.h                 TableGen
Assembly        DebugInfo        InstVisitor.h       LTO              PassManager.h          Target
AutoUpgrade.h   DebugInfo.h      IR                  MC               PassRegistry.h         Transforms
Bitcode         DIBuilder.h      IRReader            Object           PassSupport.h
CMakeLists.txt  ExecutionEngine  LinkAllIR.h         Option           Summary
```

  - Source files

```
root@ubuntu: /mnt/data/llvm/lib# ls
Analysis    CMakeLists.txt  ExecutionEngine  Linker          Makefile  Option   TableGen
AsmParser   CodeGen         IR               LLVMBuild.txt   MC        Summary  Target
Bitcode     DebugInfo       IRReader         LTO             Object    Support  Transforms
```

Using LLVM

- Writing frontend compilers
- Writing backend tools
- Wrting Tools using both frontend and backend



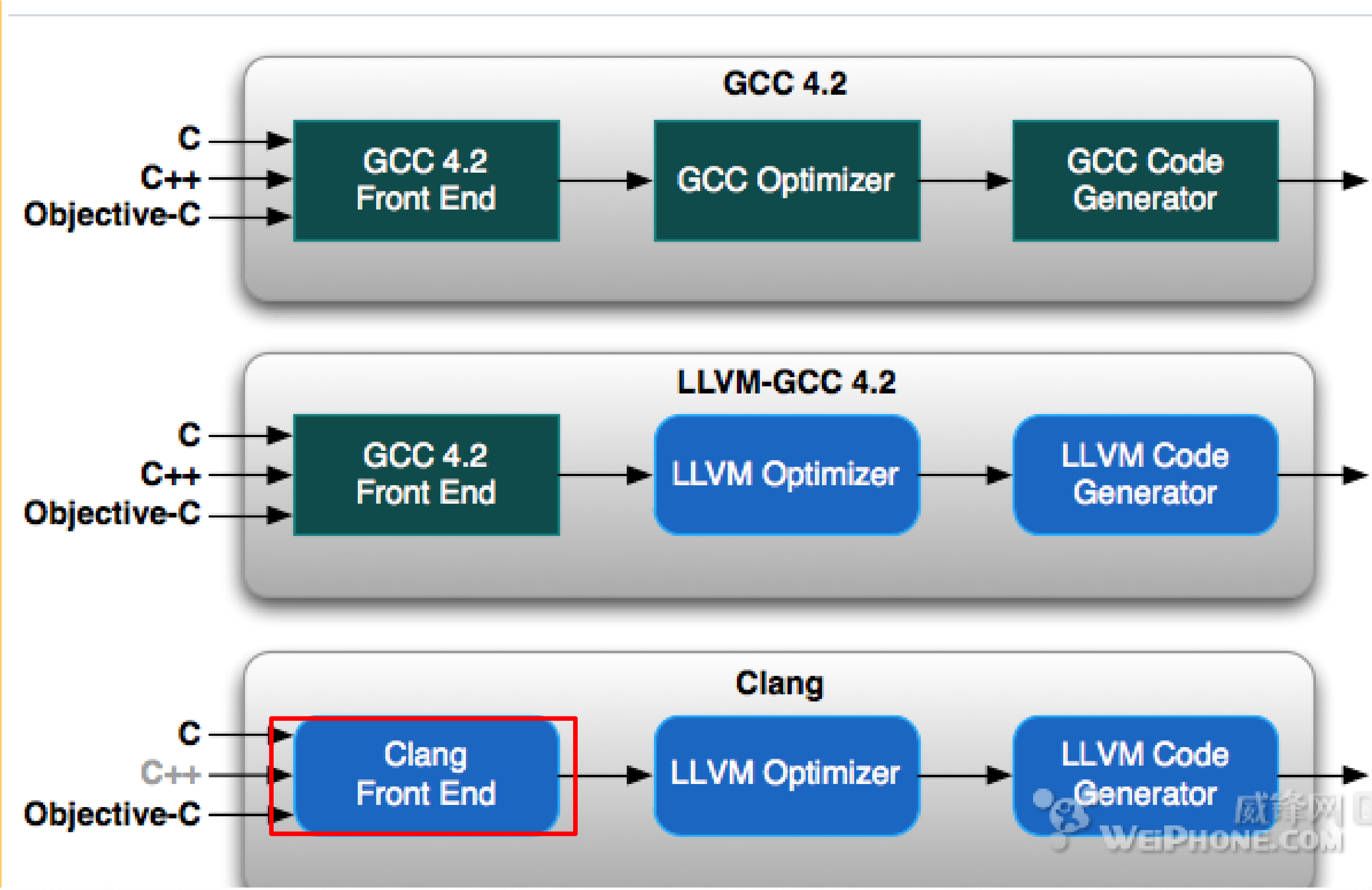
Writing frontend compilers

- Only need to compile source code to IR

- Existing compilers that compile to IR:
  - C/C++
  - Ruby
  - Python
  - Haskell
  - Java
  - D
  - PHP
  - Pure
  - Lua
  - etc.

Clang: LLVM Frontend C/C++ Compiler

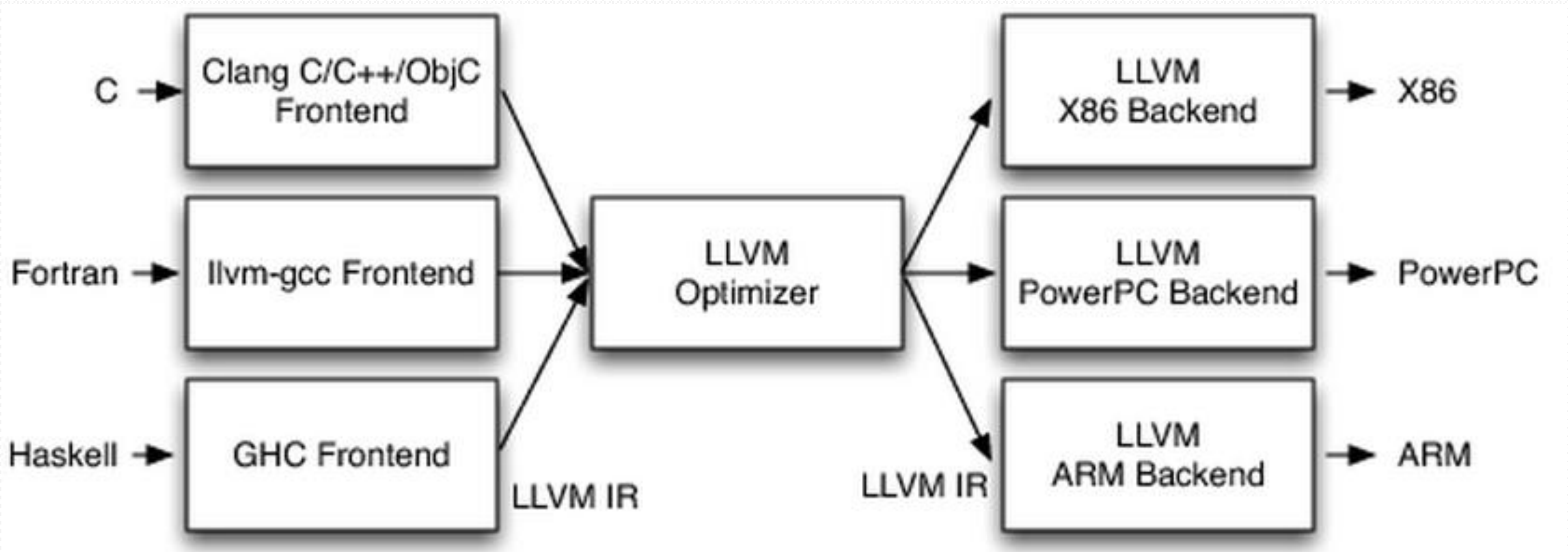
- Similar to gcc: easy to use
- Faster speed
- Better modularity
- Can be used separately from LLVM: Compile to executables



GCC 4.2
C
C++
Objective-C
GCC 4.2 Front End
GCC Optimizer
GCC Code Generator
LLVM-GCC 4.2
C
C++
Objective-C
GCC 4.2 Front End
LLVM Optimizer
LLVM Code Generator
Clang
C
C++
Objective-C
Clang Front End
LLVM Optimizer
LLVM Code Generator

Writing backend tools

- Simpliest way: Using LLVM Passes
  - Module pass
  - Function pass
  - BasicBlock pass



Writing backend tools

- All passes are registered and managed by pass manager
  - Each pass is identified by its field address: ID
  - Running order of passes are written by tool developer

```
void AllocIdentify::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.addRequired<LoopInfo>();
  AU.setPreservesAll();
}

char AllocIdentify::ID = 0;
static RegisterPass<AllocIdentify>
X("alloc-identify", "Identify allocator wrapper functions");
```

Program analysis using LLVM

- Writing intra-procedure analysis tools
  - Using Clang CFG
  - Using LLVM passes
- Define-use chains are already provided
  - Value::use_iterator
- Alias analysis
  - Inherit alias analysis base class
- Pointer analysis
  - DSA

Combining Clang and LLVM

- Clang provides ASTs in source level code
- LLVM provides more powerful program analysis tools
- LLVM gold plugin
  - Used to perform link-time optimization
  - Based on GNU gold linker

Example: Memory-leak fixing

- Pointer analysis is performed at IR during linking
  - Existing tool: DSA

- Data flow analysis is performed via Clang CFG
  - Contain the information of source code location for fixing

Thanks!