CS451 – Software Analysis

Lecture 18

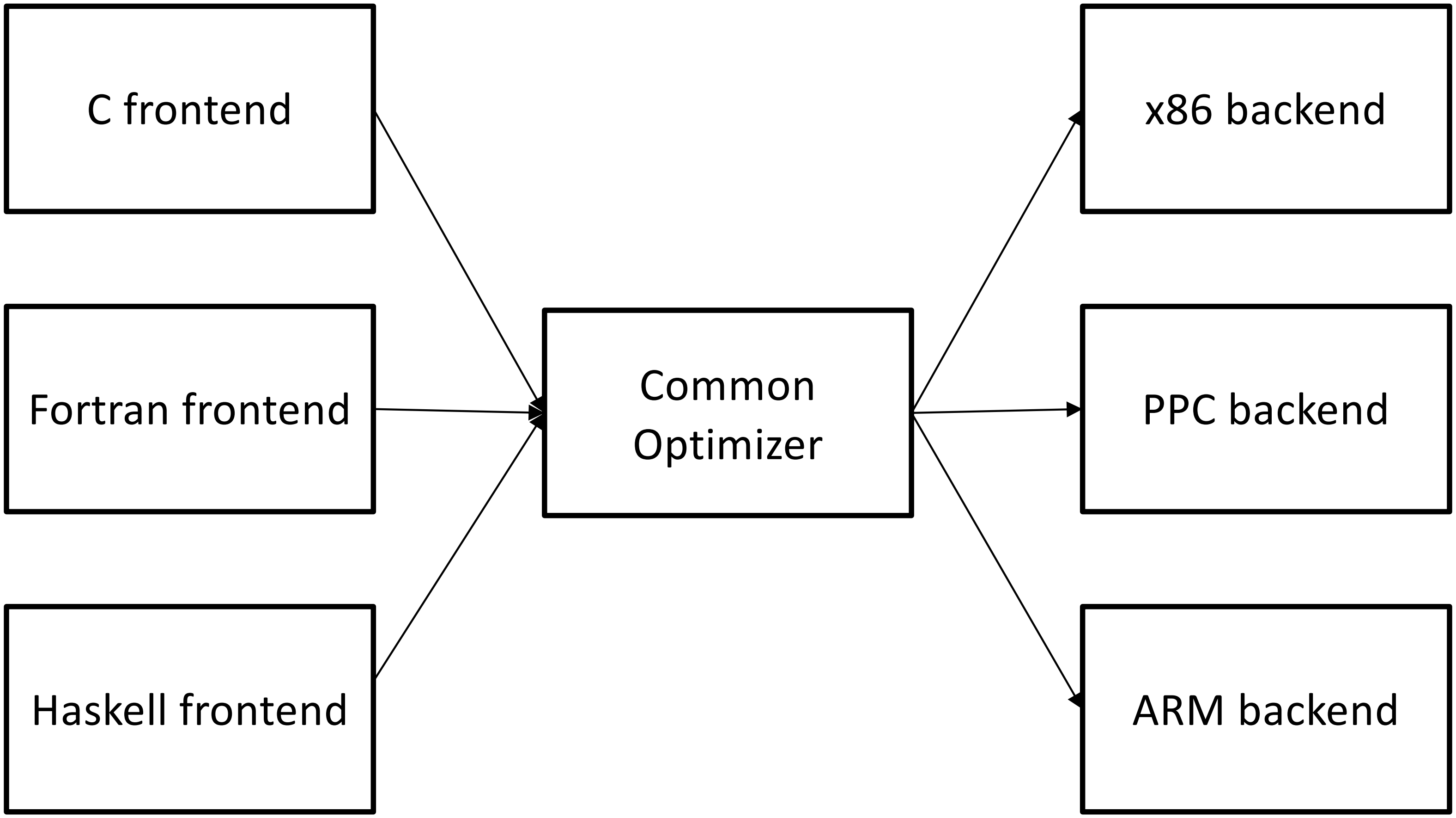
Low Level Virtual Machine (LLVM)

Elias Athanasopoulos
elathan@cs.ucy.ac.cy

LLVM

- Started in December 2000 as a set of reusable libraries with a well-defined interface
- Main motivation was to replace special-purpose tools
  - Difficult to reuse the parser of an existing compiler (e.g., GCC) for doing other tasks like static analysis
- Little code sharing between different compilers and programming languages
  - Everybody was *reinventing the wheel*
- LLVM is now one of the very well-established compiler frameworks
  - Used for many other analysis tasks

Compiler design



C frontend

Fortran frontend

Haskell frontend

Common Optimizer

x86 backend

PPC backend

ARM backend

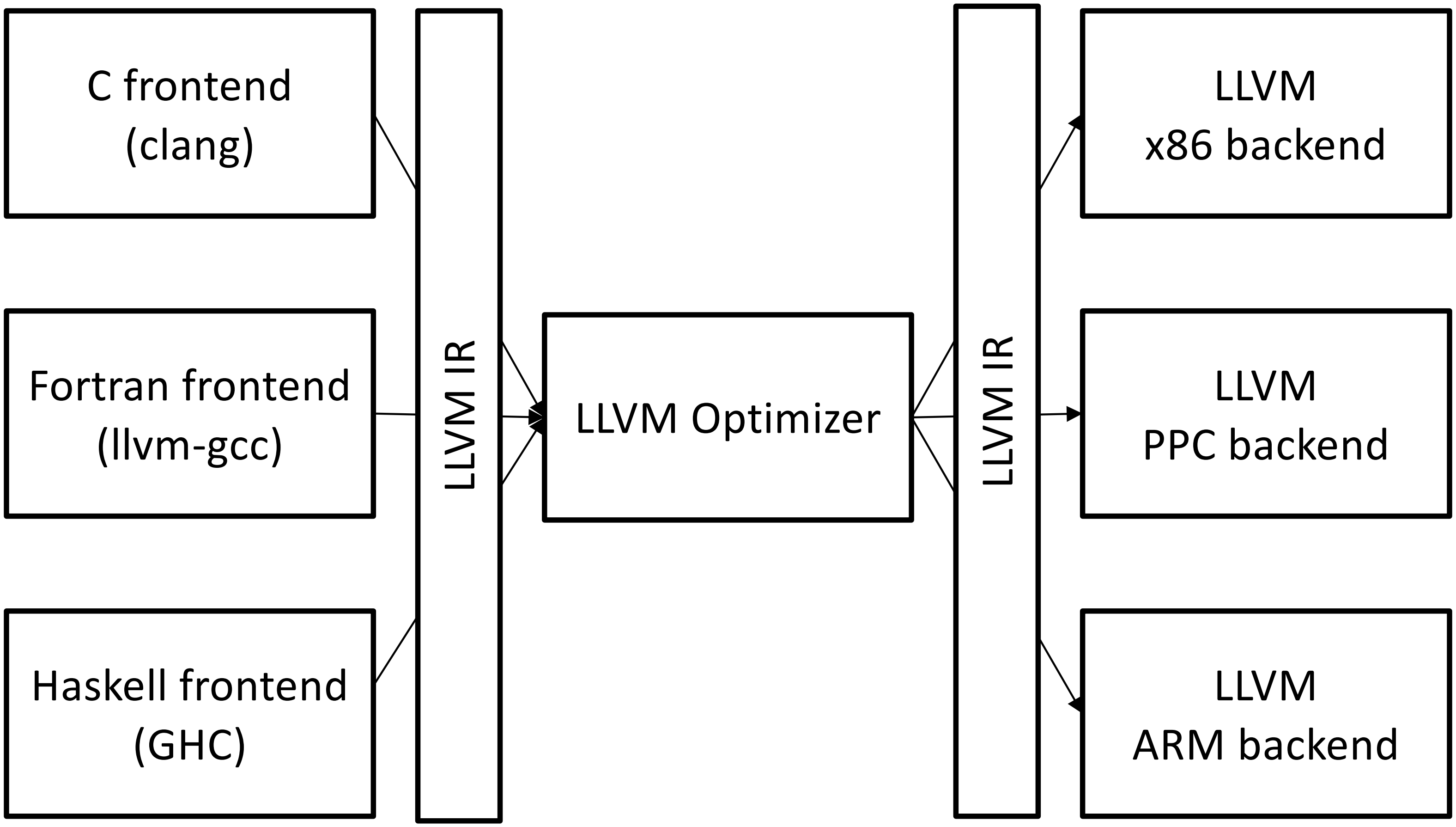
Existing implementations

- Although the three-phase design has clear benefits, it is not actually used in practice
- Implementations of Perl, Python, Ruby and Java share no code
- Compilers, such as the Glasgow Haskell Compiler (GHC) may be retargetable to multiple different CPUs
  - However, their design is very specific to the language specification
  - Java and .NET virtual machines are exceptions
- GCC offers support the three-phase design, with several frontends/backends but the design is not clean

How LLVM contributes?

- LLVM offers a common intermediate representation format
  - LLVM IR
- LLVM offers several libraries and tools that can work with the LLVM IR
  - Parse, analyze, extend, etc.
- By using LLVM, it is much easier to
  - Create new frontends or backends
  - Extend the compiler pipeline of an existing frontend
  - Create analysis tools that work with the LLVM IR

Compiler design with LLVM



LLVM IR

```c
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}
```

```
~/llvm-project/llvm/build/bin/clang -S -emit-llvm example1.c
```

Output is on example1.ll

```llvm
define dso_local i32 @add1(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add i32 %5, %6
  ret i32 %7
}
```

LLBM IR properties

- LLVM IR is a low-level RISC-like virtual instruction set
  - Supports linear sequences of simple instructions like add, subtract, compare, and branch
  - Instructions are in three-address form
- Compared to real RISC machine, LLVM
  - Is strongly typed with a simple type system, for example i32 is a 32-bit integer and i32**is a pointer to pointer to 32-bit integer
  - Supports calling convention through call/ret instructions and explicit arguments
  - Uses an infinite set of registers named with a % character
- Easy for a front end to generate
- Expressive enough to allow important optimizations to be performed for real targets

LLVM IR representation

- LLVM IR is defined in three isomorphic forms
  - textual format (example in slide)
  - in-memory data structure inspected and modified by optimizations themselves
  - efficient and dense on-disk binary "bitcode" format
- LLVM provides tools to convert the on-disk format from text to binary
  - `llvm-as` assembles the textual `.ll` file into a `.bc` file
  - `llvm-dis` turns a `.bc` file into an `.ll` file

Optimizations on LLVM IR

```
# X-X
%example1 = sub i32 %a, %a
# X-0
%example2 = sub i32 %b, 0
# (X*2)-X
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
```

```
// X - 0 -> X
if (match(Op1, m_Zero()))
  return Op0;
// X - X -> 0
if (Op0 == Op1)
  return Constant::getNullValue(Op0->getType());
// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
  return Op1;
```

Complete code representation

- LLVM IR is well specified and serves an interface to the optimizer
  - Writing a new front end for a new programming language is as simple as generating LLVM IR
  - The LLVM tools can then take the produced IR and compiled it to machine code for several different architectures
- Doing this with different compiler toolchains (e.g., GCC) is not straightforward

Collection of libraries

- LLVM is not another compiler or run-time, but a collection or shared libraries for processing LLVM IR
  - This makes it possible to implement new tools or extend *existing* compilers
- As an example, consider the optimization phase of clang
  - It takes LLVM IR and produces new optimized LLVM IR
  - Clang with –O0 uses no optimization passes, but with –O3 uses 67 passes (LLVM 2.8)
- Each LLVM pass is written as a C++ class the derives from the Pass class
  - New classes can be written by users

Example of a pass

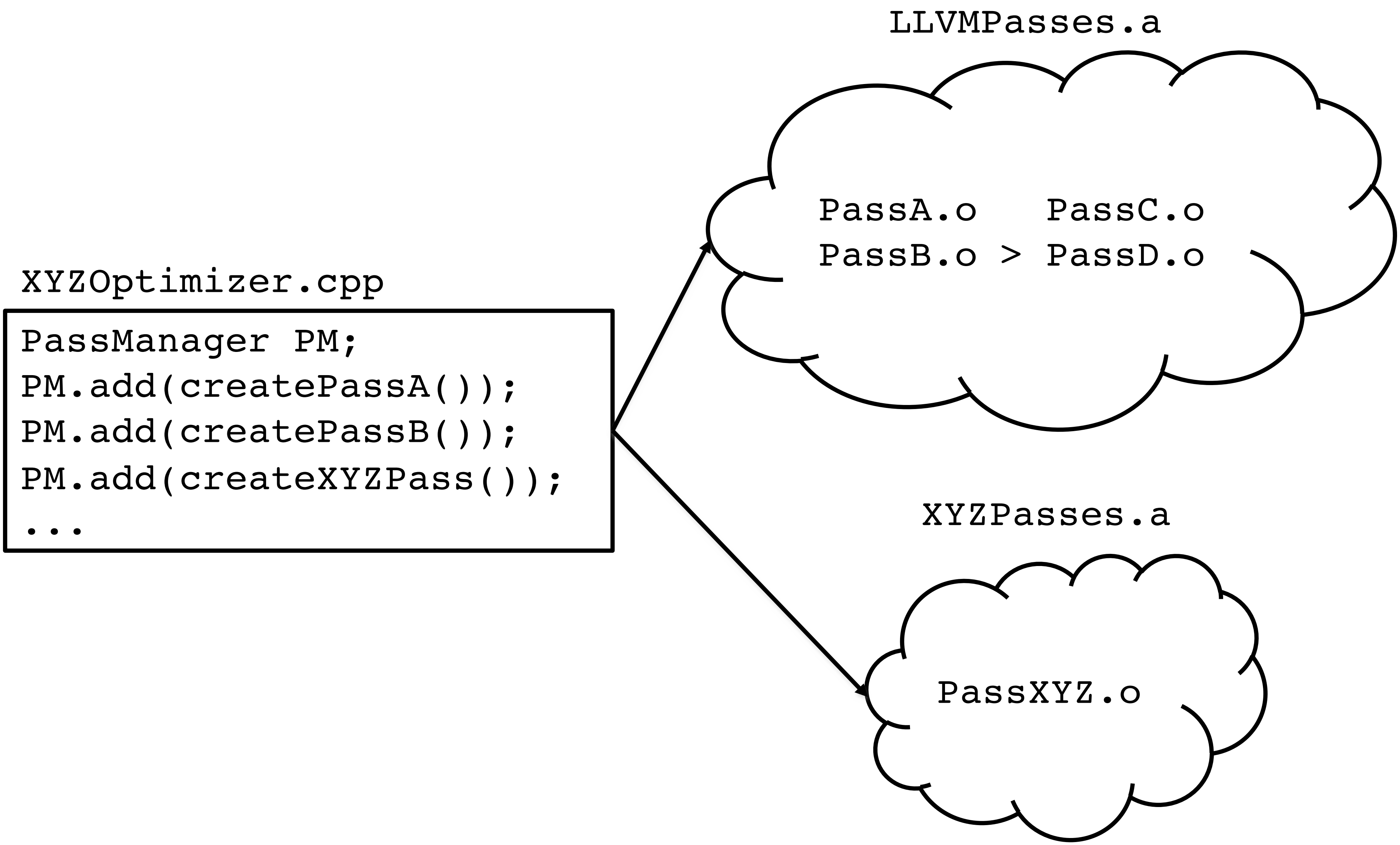
```
namespace {
  class Hello : public FunctionPass {
  public:
    // Print out the names of functions in
    // the LLVM IR being optimized.
    virtual bool runOnFunction(Function &F) {
      cerr << "Hello: " << F.getName() << "\n";
      return false;
    }
  };
}


FunctionPass *createHelloPass() { return new Hello(); }
```

Available passes

- LLVM provides dozens of passes
  - Compiled into one or more .o files and then assembled to archives (.a files on Unix systems)
  - Passes provide all sorts of analysis and transformation capabilities
  - Passes are expected to be standalone, or declare their dependencies if they depend on other passes
- The LLVM PassManager is responsible for running all passes during compilation
- Users can enable new passes and define the order
  - Custom compilation for different applications

Custom compilation



XYZOptimizer.cpp

```
PassManager PM;
PM.add(createPassA());
PM.add(createPassB());
PM.add(createXYZPass());
...
```

LLVMPasses.a

```
PassA.o   PassC.o
PassB.o > PassD.o
```

XYZPasses.a

```
PassXYZ.o
```

Retargetable code generator

- The LLVM code generator is responsible for transforming LLVM IR into target specific machine code
- Code generators are specific for each target, but solve very similar problems
  - Assign values to registers using common algorithms
- LLVM splits the code generation problem into individual passes
  - Instruction selection, register allocation, etc.
  - Support passes in code generation
- The target author can use default passes or override the defaults and implement new ones
  - The x86 back end uses a register-pressure-reducing scheduler
  - The PPC back end uses a latency optimizing scheduler

Target description files

- LLVM supports a “mix and match” approach
  - A set of passes can be enabled for different architectural targets
  - Consider a pass that optimizes register usage, when each target has a different set of registers
- LLLVM provides a target description in a declarative domain-specific language (a set of .td files) processed by the tblgen tool

```
def GR32 : RegisterClass<[i32], 32, [EAX,
ECX, EDX, ESI, EDI, EBX, EBP, ESP, R8D,
R9D, R10D, R11D, R14D, R15D, R12D, R13D]>
{ … }
```

Modular design

- LLVM IR can be (de)serialized to/from LLVM bitcode
- Partial compilation, save our progress to disk, then continue work at some point in the future
  - Link-time and install-time optimization, both of which delay code generation from "compile time"
- Traditional compilers process one translation unit (.c file)
  - Link-Time Optimization (LTO) perform optimizations, e.g., *inlining*, across file boundaries
- LLVM compilers like Clang support (-flto or -O4)
  - Instructs the compiler to emit LLVM bitcode to the .o file
  - Delays code generation to link time

Link-time optimizations (LTO)

- The linker detects that it has LLVM bitcode in the .o files instead of native object files
  - Reads all the bitcode files into memory, links them together, and runs the LLVM optimizer
- The optimizer can now see across a much larger portion of the code
  - Can inline, propagate constants, do more aggressive dead code elimination, and more across file boundaries
- Many modern compilers support LTO by having an expensive and slow serialization process
  - In LLVM, LTO falls out naturally from the design of the system, and works across different source languages

Install-time optimizations

- Delay code generation even later than link time, all the way to install time
  - Install time is a when you find out the specifics of the device you're targeting
  - In the x86 family for example, there is a broad variety of chips and characteristics

Unit testing

- The traditional approach to testing this is to write a .c file that is run through the compiler, and to have a test harness that verifies that the compiler doesn't crash
  - The compiler consists of many different subsystems
  - Many different passes in the optimizer cane interfere with the buggy code in question
- The LLVM test suite has highly focused regression tests that can load LLVM IR from disk, run it through exactly one optimization pass, and verify the expected behavior

Unit test example

- The RUN line specifies the command to execute
- The opt program is a simple wrapper around the LLVM pass manager
- The FileCheck tool verifies that its standard input matches a series of CHECK directives

```
; RUN: opt < %s -constprop -S | FileCheck %s
define i32 @test() {
    %A = add i32 4, 5
    ret i32 %A ;
    CHECK: @test() ;
    CHECK: ret i32 9
}
```

Reproducing the bug

- BugPoint uses the IR serialization for bug reproduction
  - Takes as input a file .ll/.bc and the optimization passes that causes an optimizer crash
  - It then outputs a reduced test case and the opt command used to reproduce the failure
  - It finds this by using "delta debugging”
  - BugPoint knows the structure of LLVM IR and can send valid IR inputs to the optimizer

References

- The Architecture of Open Source Applications
  - http://www.aosabook.org/en/llvm.html