Code analysis and transformation

LLVM

Simone Campanoni
simonec@eecs.northwestern.edu
Problems with Canvas?

Problems with slides?

Any problems?

Makefile tutorial:
http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor
Outline

• Introduction to LLVM

• CAT steps

• Hacking LLVM
LLVM

• LLVM is a great, hackable compiler for C/C++ languages
  • C, C++, Objective-C

• But it’s also
  • A dynamic compiler
  • A compiler for bytecode languages (e.g., Java, CIL bytecode)

• LLVM IR: bitcode

• LLVM is modular and well documented

• Started from UIUC, it’s now the research tool of choice

• It’s an industrial-strength compiler
  Apple, AMD, Intel, NVIDIA
LLVM tools

- **clang**: compile C/C++ code as well as OpenMP code
- **clang-format**: to format C/C++ code
- **clang-tidy**: to detect and fix bug-prone patterns, performance, portability and maintainability issues
- **clangd**: to make editors (e.g., vim) smart
- **clang-rename**: to refactor C/C++ code
- **SAFECode**: memory checker
- **lldb**: debugger
- **lld**: linker
- **polly**: parallelizing compiler
- **libclc**: OpenCL standard library
- **dragonegg**: integrate GCC parsers
- **vmkit**: bytecode virtual machines
- ... and many more
LLVM common use at 10000 feet

Source files

clang

Binary
LLVM common use at 10000 feet

Source files

Binary

```
$ clang hello_world.c -o hello_world
$ ./hello_world
hello world
```
LLVM common use at 10000 feet

Source files

Lib/tool...

Lib/tool...

Lib/tool 1

Lib/tool 2

clang

Lib/tool 3

Lib/tool 4

Lib/tool...

Lib/tool...

Lib/tool...

Binary

LLVM They all talk bitcode
LLVM internals

• A component is composed of pipelines
  • Each stage: reads something as input and generates something as output
  • To develop a stage: specify how to transform the input to generate the output

• Complexity lies in linking stages

• In this class: we’ll look at concepts and internals of middle-end
  But some of them are still valid for front-end/back-end
LLVM and other compilers

- LLVM is designed around its IR
  - Multiple forms (human readable, bitcode on-disk, in memory)

Diagram:

- Front-end (Clang)
  - IR
  - Pass manager
  - Middle-end
    - IR
    - Back-end
      - Machine code
  - IR
  - Pass
  - IR
  - ...
Pass manager

- The pass manager orchestrates passes
- It builds the pipeline of passes in the middle-end
- The pipeline is created by respecting the dependences declared by each pass
  Pass X depends on Y
  Y will be invoked before X
Learning LLVM

• Login to hanlon.wot.eecs.northwestern.edu and play with LLVM
  • LLVM 8.0.0 is installed in /home/software/llvm
  • Add the following code in your ~/.bash_profile file
    LLVM_HOME=/home/software/llvm
    export PATH=$LLVM_HOME/bin:$PATH
    export LD_LIBRARY_PATH=$LLVM_HOME/lib:$LD_LIBRARY_PATH

• Read the documentation
• Read the documentation
• Read the documentation

• Get familiar with LLVM documentation
  • Doxygen pages (API docs)
  • Language reference manual (IR)
  • Programmer’s manual (LLVM-specific data structures, tools)
  • Writing an LLVM pass
Pass types

Use the “smallest” one for your CAT

• CallGraphSCCPass
• ModulePass
• FunctionPass
• LoopPass
• BasicBlockPass

```c
int bar (void){
    return foo(2);
}

int foo (int p){
    return p+1;
}
```
Adding a pass

• Internally
  
  clang  vmkit  ...

• Externally
  
  • More convenient to develop (compile-debug loop is much faster!)

  clang  vmkit  ...
Homework: build your own compiler

Source files

- Your work
- CAT
- LLVM IR
- clang

A bash script

cat-c

Binary
You will start from CAT-c.tar.bz2 (Canvas)
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/IR/LegacyTargetPassManager.h"
#include "llvm/IR/LegacyAnalysisPass.h"
#include "llvm/IR/Transforms/InstCombine/InstCombine.h"
#include "llvm/IR/Transforms/FunctionPass.h"
#include "llvm/IR/FunctionPass.h"

namespace {
    struct CAT : public FunctionPass {
    static char ID;

    CAT() : FunctionPass(ID) {
    }

    bool doInitialization(Module &M) override {
        errs() << "Hello LLVM World at " << doInitialization\n";  
        return false;
    }

    bool runOnFunction(static RegisterPass CAT X("CAT", "Homework for the CAT class"),
    errs() << "Hello World at " << runOnFunction\n";  
    return false; 
    }

    static RegisterStandardPasses _RegPass1(PassManagerBuilder::EP_OptimizerLast,
    static CAT * _PassMaker = NULL;
    static RegisterStandardPasses _RegPass2(PassManagerBuilder::EP_OptEnabledOnOptLevel0,
    void getAnalysisAUX(const PassManagerBuilder& PM, legacy::PassManagerBase& PM) {
        if(!_PassMaker) { PM.add(_PassMaker = new CAT()); } // ** for -Ox
        if(!_PassMaker) { PM.add(_PassMaker = new CAT()); } // ** for -O0
    }
Your cat-c compiler

Source files

Your work

CAT

Your work

clang

cat-c

Binary

A bash script
Using your cat-c compiler

To do more than a hello world pass: modify
10 assignments: from H0 to H9

- Hi depends on Hi-1
- For every assignment:
  - You have to modify your previous CatPass.cpp
  - You have to pass all tests distributed
- Assignment i: Hi.tar.bz2
  - The description of the homework (Hi.pdf)
  - The tests you have to pass (tests)
- Each assignment is an LLVM pass
Passes

• A compilation pass reads and (sometime) modifies the bitcode (LLVM IR)

• If you want to understand code properties: you need to understand the bitcode

• If you want to modify the bitcode: you need to understand the bitcode first
LLVM IR (a.k.a. bitcode)

• RISC-based
  • Instructions operate on variables
  • Load and store to access memory

• Include high level instructions
  • Function calls \texttt{(call)}
  • Pointer arithmetics \texttt{(getelementptr)}
LLVM IR (2)

• Strongly typed
  • No assignments of variables with different types
  • You need to explicitly cast variables
  • Load and store to access memory

• Variables
  • Global  (@myVar)
  • Local to a function  (%myVar)
  • Function parameter (define i32 @myF (i32 %myPar))
LLVM IR (3)

• 3 different (but 100% equivalent) formats
  • Assembly: human-readable format (FILENAME.ll)
  • Bitcode: machine binary on-disk (FILENAME.bc)
  • In memory: in memory binary

• Generating IR
  • Clang for C-like languages (similar options w.r.t. GCC)
  • Different front-ends available
It’s a Static Single Assignment (SSA) representation
  • A variable is set only by one instruction in the function body
    %myVar = ...
  • A static assignment can be executed more than once

We’ll study SSA later
SSA and not SSA example

float myF (float par1, float par2, float par3){
    return (par1 * par2) + par3;
}

define float @myF(float %par1, float %par2, float %par3) {
    %1 = fmul float %par1, %par2
    %1 = fadd float %1, %par3
    ret float %1
}

define float @myF(float %par1, float %par2, float %par3) {
    %1 = fmul float %par1, %par2
    %2 = fadd float %1, %par3
    ret float %2
}
SSA and not SSA

• CATs applied to SSA-based code are faster!
  • Old compilers aren’t SSA-based
  • Transforming IR in its SSA-form takes time

• When designing your CAT, think carefully about SSA
  • Take advantage of its properties
LLVM tools to read/generate IR

• **clang** to compile/optimize/generate LLVM IR code
  • To generate binaries from source code or IR code
  • Check Makefile you have in LLVM.tar.bz2 (Canvas)

• **lli** to execute (interpret/JIT) LLVM IR code
  lli FILE.bc

• **llc** to generate assembly from LLVM IR code
  llc FILE.bc
LLVM tools to read/generate IR

- **opt** to analyze/transform LLVM IR code
  - Read LLVM IR file
  - Load external passes
  - Run specified passes
  - Respect pass order you specify as input
    - `opt -pass1 -pass2 FILE.ll`
  - Optionally generate transformed IR

- **Useful passes**
  - `opt -view-cfg FILE.ll`
  - `opt -view-dom FILE.ll`

- `opt -help`
LLVM summary

• LLVM is an industrial-strength compiler also used in academia
  • Very hard to know in detail every component
  • Focus on what’s important to your goal
  • Become a ninja at jumping around the documentation

• It’s well organized, documented with a large community behind it

• Basic C++ skills are required
Final tips

• LLVM includes A LOT of passes
  • Analyses
  • Transformations
  • Normalization

• Take advantage of existing code

• I have a pointer to something. What is it?
  
  getName() works on most things
  
  errs() << TheThingYouDon’tKnow ;
Now you are ready for your first assignment!

homework/H0.tar.bz2

Test your code in
hanlon.wot.eecs.northwestern.edu
Outline

• Introduction to LLVM

• CAT steps

• Hacking LLVM
Code analysis and transformation

• Code normalization

• Analysis

• Transformation
CAT example: loop hoisting

Do {
    Work(varX);
    varY = varZ + 1;
    varX++;
} while (varX < 100);

varY = varZ + 1;
Do {
    Work(varX);
    varX++;
} while (varX < 100);
CAT example: loop hoisting (2)

Do {
    Work(varX);
    varY = varZ + 1;
    varX++;
} while (varX < 100);

while (varX < 100) {
    Work(varX);
    varY = varZ + 1;
    varX++;
}

And now?
Loop normalization

• What: loop normalization pass

• When: before running loop hoisting
  Declare a dependence to your pass manager

• Advantages?

• Disadvantages?
CAT design

• Understand the problem
• Create representative code examples you expect to optimize
• Optimize them by hand to test the best benefits of your optimization
• Identify the common case
• Define the normalized input code
• Define the information you need to make your transformation safe
• Design the analyses to automatically generate this information
• Design the transformation
• Test, test, test
Improving CAT

• Improve your CAT by better handling your common cases

• Improve your CAT by improving the normalization passes

• Handle corner cases
  Before we simply ignored them (i.e., no transformation)
As Linus Torvalds says ...

*Talk is cheap. Show me the code.*

Let’s start hacking LLVM

**LLVM use examples:** LLVM_introduction.tar.bz2  
code/LLVM.tar.bz2