Advanced Topics in C Compilers

LLVM

Simone Campanoni
simone.campanoni@northwestern.edu
Outline

- Summary of 323: LLVM

- Summary of 323: LLVM IR

- Summary of 323: Dependences
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
• llpd: debugger
• llld: 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

```
$ clang hello_world.c -o hello_world

$ ./hello_world

hello world
```

Binary
LLVM common use at 10000 feet

Source files

Lib/tool...

clang

Lib/tool 1

Lib/tool 3

Lib/tool...

Lib/tool 2

Lib/tool 4

Lib/tool...

Lib/tool...

Lib/tool...

Lib/tool...

LLVM They all talk bitcode

Binary
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
LLVM and other compilers

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

```
IR
  ↓
Pass
  ↓
  IR
  ↓
Pass
  ↓
  IR
  ↓
  ...
```

```
Front-end (Clang)
  ↓
  IR
  ↓
Middle-end
  ↓
  IR
  ↓
Back-end
  ↓
Machine code
```

Pass manager
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
Pass types

Use the “smallest” one for your project

- 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!)

https://github.com/scampanoni/LLVM_middleend_template
```cpp
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 (Function &F) override {
    errs() << "Hello CAT\n;  
    return false;
  }

};

using namespace llvm;

}  // namespace
```
Outline

• Summary of 323: LLVM

• Summary of 323: LLVM IR

• Summary of 323: Dependences
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 (call, invoke)
  • Pointer arithmetics (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
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

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

• When designing your pass, 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

- **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
Running LLVM passes

```cpp
// Next there is code to register your pass to "opt"
char CAT::ID = 0;
static RegisterPass<CAT> X("CAT", "Homework for the CAT class");

// Next there is code to register your pass to "clang"
static CAT * _PassMaker = NULL;
static RegisterStandardPasses _RegPass1(PassManagerBuilder::EP_OptimizerLast,
    const PassManagerBuilder&, legacy::PassManagerBase& PM) {
    if(!_PassMaker){ PM.add(_PassMaker = new CAT()); }); // ** for -Ox
static RegisterStandardPasses _RegPass2(PassManagerBuilder::EP_EnabledOnOptLevel0,
    const PassManagerBuilder&, legacy::PassManagerBase& PM) {
    if(!_PassMaker){ PM.add(_PassMaker = new CAT()); }); // ** for -00
```
Outline

• Summary of 323: LLVM

• Summary of 323: LLVM IR

• Summary of 323: Dependences
Dependences: the big picture

• Code transformations are designed to preserve the “semantics” of the code given as input
  • What is the “semantics” of a program?

• A dependence A -> B is satisfied if A will always execute before B

• If we satisfy all dependences in the code, then we will preserve I => O
Control dependence intuition

• Dependence: C will be executed depending on B

• How to identify C? (automatically)
  • Do we need a DFA?
    • We need a Control Flow Analysis

A: varX = 1;
B: if (par1 > 5)
C: varX = par1 + 1
D: print(varX)
Post-dominators

**Assumption:** Single exit node in CFG

**Definition:** Node $d$ post-dominates node $n$ in a graph if every path from $n$ to the exit node goes through $d$

B: if (par1 > 5)
C: varX = par1 + 1
C2: ...
D: print(varX)

**Immediate post-dominator tree**
Control dependences

A node $Y$ control-depends on another node $Y$ if and only if

1. There is a path from $X$ to $Y$ such that every node in that path other than $X$ and $Y$ is post-dominated by $Y$
2. $X$ is not strictly post-dominated by $Y$

```
B: while (par1 > 5)
C: varX = par1 + 1
C2: ...
D: print(varX)
```
Data dependences

• Gives constraints on parallelism that must be satisfied

• Must be satisfied to have correct program  
  • How can we satisfy data dependences?

• Any order that does not violate these dependences is correct!
Loop-carried data dependences

while(...){
    i: x = ...;
    j: *p = x + 1;
    ...
}

while(...){
    j: *p = x + 1;
    i: x = ...;
    ...
}
Loop-carried data dependences

while(...){
    j: *p = x + 1;
    i: x = ...;
    ... 
}

while(...){
    j: *p = A[i-2] + 1;
    i: A[i] = ...;
    k: i++; 
}
Program dependence graph (PDG)

- Program Dependence Graph = Control Dependence Graph + Data Dependences

- Facilitates performing most traditional optimizations
  - Constant folding, scalar propagation, common subexpression elimination, code motion, strength reduction, code parallelization, code vectorization, etc...

- Requires only single walk over PDG
Strongly Connected Component (SCC)

Often you need to partition instructions in groups

- Where each group is composed of instructions that depend on each other

Core 0

i0: while (i <= N)
i1:   X = Y + 1
i2:   K = Z * 5
i3:   Y = X * 42
i4:   Z = K + 2
i5:   i = i + 1

Core 0

i1:   X = Y + 1
i3:   Y = X * 42

Core 1

i2:   K = Z * 5
i4:   Z = K + 2

Different colors <-> different cycles in the PDG => different cores
Strongly Connected Component (SCC)

• A directed graph is strongly connected if there is a path between all pairs of vertices

• A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph
SCCDAG

• From the PDG

• To the SCC identifications
SCCDAG

• From the PDG

• To the SCC identifications

• To the SCCDAG

i0: while (i <= N)
i1: X = Y + 1
i2: K = Z * 5
i3: Y = X * 42
i4: Z = K + 2
i5: i = i + 1