数理・計算科学特論C
プログラミング言語処理系の最先端実装技術

LLVM Compiler Framework
What is LLVM?

- LLVM is an open-source **compiler infrastructure** (e.g. COINS, Eclipse OMR):
  - Its official page says: “The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.”
  - It is originated from a research project at U. Illinois “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, Chris Lattner and Vikram Adve (University of Illinois at Urbana-Champaign), CGO 2004
  - It can support both static and dynamic (JIT) compiler
  - It uses non-copyleft (i.e. commercial-use friendly) University of Illinois/NCSA license
LLVM sub projects

- LLVM Core
  - LLVM IR (intermediate representation) design and utilities
  - optimizers, backends etc (using LLVM IR)

- Clang
  - C/C++ frontend (translate C/C++ into LLVM IR)

- Compiler-RT
  - low-level intrinsics (e.g. software floating point library)
  - code testing runtimes

- libc++, lldb, lld, OpenMP, OpenCL, polly etc…
LLVM users

- Many research projects (IMO, it is much more popular than GCC now)

- Many open source projects (such as FreeBSD, Rust, WebAssembly, Pyston, TensorFlow XLA)

- Many products
  - Apple: macOS (clang), iOS (Swift)
  - NVIDIA: CUDA tool chain
  - Sony: Play Station 4 development tool
  - IBM: XL C/C++ (clang frontend combined with own backend)
LLVM and Dragon

- It is Wyvern, a kind of dragon with wings
- “Dragons have connotations of power, speed and intelligence, and can also be sleek, elegant, and modular (err, maybe not).”
- It also relates to Dragon book, a famous text book on compiler
- “This dragon image is owned by Apple Inc.”
**Overall structure of LLVM infrastructure**

- **LLVM Pass**: in LLVM, each optimizer, analyzer, checker etc. is implemented as a *pass* that takes input and generates output in LLVM-IR (or low-level Machine IR)

- Types of passes: module pass, function pass, loop pass, basic block pass, machine function pass
LLVM IR (Intermediate Representation)

- LLVM IR is a code representation used in optimization phases of LLVM
  - based on Static Single Assignment (SSA)
  - low enough level to cover various language features
  - typed language
    (see language reference: https://llvm.org/docs/LangRef.html)

- LLVM IR can be dumped into a file as bitcode (.bc) or human readable assembly file (.ll)
  - A human readable IR file (.ll file) can be generated from C program using clang by clang -O -S -emit-llvm input.c
An example of LLVM IR

```c
int foo(int *p, int a) {
  int x = 0;
  if (p == nullptr)
    x = func(a);
  else
    x = a + *p;
  return x;
}
```

```llvm
define signext i32 @_Z3fooPii(i32* %p, i32 signext %a) #0 {
  entry:
  %cmp = icmp eq i32* %p, null
  br i1 %cmp, label %if.then, label %if.else

  if.then:
    %call = call signext i32 @_Z4funci(i32 signext %a)
    br label %if.end

  if.else:
    %0 = load i32, i32* %p, align 4, !tbaa !3
    %add = add nsw i32 %a, %0
    br label %if.end

  if.end:
    %x.0 = phi i32 [ %call, %if.then ], [ %add, %if.else ]
    ret i32 %x.0
}
```
define signext i32 @_Z3fooPii(i32* %p, i32 signext %a) #0 {
   entry:
      %p.addr = alloca i32*, align 8
      %a.addr = alloca i32, align 4
      %x = alloca i32, align 4
      store i32* %p, i32** %p.addr, align 8, !tbaa !3
      store i32 %a, i32* %a.addr, align 4, !tbaa !7
      %0 = bitcast i32* %x to i8*
      call void @llvm.lifetime.start.p0i8(i64 4, i8* %0) #3
      store i32 0, i32* %x, align 4, !tbaa !7
      %1 = load i32*, i32** %p.addr, align 8, !tbaa !3
      %cmp = icmp eq i32* %1, null
      br i1 %cmp, label %if.then, label %if.else

   if.then: ; preds = %entry
      %2 = load i32, i32* %a.addr, align 4, !tbaa !7
      %call = call signext i32 @_Z4funci(i32 signext %2)
      store i32 %call, i32* %x, align 4, !tbaa !7
      br label %if.end

   if.else: ; preds = %entry
      %3 = load i32, i32* %a.addr, align 4, !tbaa !7
      %4 = load i32*, i32** %p.addr, align 8, !tbaa !3
      %5 = load i32, i32* %4, align 4, !tbaa !7
      %add = add nsw i32 %3, %5
      store i32 %add, i32* %x, align 4, !tbaa !7
      br label %if.end

   if.end: ; preds = %if.then, %if.else
      %6 = load i32, i32* %x, align 4, !tbaa !7
      %7 = bitcast i32* %x to i8*
      call void @llvm.lifetime.end.p0i8(i64 4, i8* %7) #3
      ret i32 %6
}

Tools to handle LLVM IR files

System tools included in LLVM

- **llc**: compiler for LLVM IR (.bc/.ll file → .s)
- **lli**: interpreter for LLVM IR (directly execute .bc/.ll file)
- **opt**: a tool to apply specific pass to LLVM IR file (.bc/.ll file → .bc/.ll file)

These tools can be installed by `apt install llvm` (not `apt install clang`) on Ubuntu

Clang

- `clang` accepts LLVM IR files as well as .c and .s files
Intermediate Representations used in LLVM

1) LLVM IR
   (Machine-independent)
   High-level IR

2) Selection DAG

3) Machine IR
   (Machine SSA)

4) Machine IR
   (non-SSA)

"Lowering"

(Machine-dependent)
Low-level IR
Selection DAG (entry BB)

Initial selection DAG: BB#0 '_Z3fooPii:entry'
SelectionDAG has 19 nodes:

t0: ch = EntryToken

t4: i64,ch = CopyFromReg t0, Register:i64 %vreg4

  t6: i64 = AssertSext t4, ValueType:ch:i32

  t7: i32 = truncate t6

  t9: ch = CopyToReg t0, Register:i32 %vreg5, t7

  t2: i64,ch = CopyFromReg t0, Register:i64 %vreg3

    t12: i1 = setcc t2, Constant:i64<0>, seteq:ch

    t14: i1 = xor t12, Constant:i1<-1>

    t16: ch = brcond t9, t14, BasicBlock:ch<if.else 0x1002bb99de8>

    t18: ch = br t16, BasicBlock:ch<if.then 0x1002bb99d38>
Machine SSA (for PowerPC)

# Machine code for function _Z3fooPii: IsSSA, TracksLiveness

Function Live Ins: %X3 in %vreg3, %X4 in %vreg4

BB#0: derived from LLVM BB %entry Live Ins: %X3 %X4

%vreg4<def> = COPY %X4; G8RC:%vreg4
%vreg3<def> = COPY %X3; G8RC_and_G8RC_NOX0:%vreg3
%vreg5<def> = COPY %vreg4:sub_32; GPRC:%vreg5 G8RC:%vreg4
%vreg6<def> = CMPLDI %vreg3, 0; CRRC:%vreg6 G8RC_and_G8RC_NOX0:%vreg3
BCC 68, %vreg6<kill>, <BB#2>; CRRC:%vreg6

B <BB#1>

Successors according to CFG: BB#1(0x30000000 / 0x80000000 = 37.50%)
BB#2(0x50000000 / 0x80000000 = 62.50%)

BB#1: derived from LLVM BB %if.then Predecessors according to CFG: BB#0

ADJCALLSTACKDOWN 32, 0, %R1<imp-def,dead>, %R1<imp-use>

%vreg8<def> = EXTSW_32_64 %vreg5; G8RC:%vreg8 GPRC:%vreg5
%X3<def> = COPY %vreg8; G8RC:%vreg8
# Machine code for function _Z3fooPii: NoPHIs, TracksLiveness, NoVRegs

Function Live Ins: %X3, %X4

BB#0: derived from LLVM BB %entry Live Ins: %X3 %X4

%CR0<def> = CMPLDI %X3, 0
BCC 76, %CR0<killed>, <BB#2>

Successors according to CFG: BB#2(0x30000000 / 0x80000000 = 37.50%)
BB#1(0x50000000 / 0x80000000 = 62.50%)

BB#1: derived from LLVM BB %if.else Live Ins: %X3 %X4

Predecessors according to CFG: BB#0

%R3<def> = LWZ 0, %X3<killed>; mem:LD4[%p](tbaa=!4)
%R3<def> = ADD4 %R3<killed>, %R4<killed>, %X4<imp-use>, %X3<imp-def>
%X3<def> = EXTSW_32_64 %R3<killed>, %X3<imp-use>
BLR8 %LR8<imp-use>, %RM<imp-use>, %X3<imp-use,killed>

BB#2: derived from LLVM BB %if.then Live Ins: %X4
Optimization passes

- LLVM provides most of the well-known (e.g. explained in this course) optimizations and analysis
  - You can observe what optimization passes are applied and how each pass works by `clang -03 -mllvm -print-after-all input.c`
  - Often similar (or same) optimization is implemented twice for LLVM IR and Machine IR
  - LLVM provides powerful vectorizers compared to GCC (loop vectorizer, SLP vectorizer, polyhedral optimization as Polly sub project)
Optimization passes for –O3 (partial)

- Module Verifier
- Simplify the CFG
- SROA
- Early CSE
- Lower 'expect' Intrinsics
- Force set function attributes
- Infer set function attributes
- Interprocedural Sparse Conditional Constant Propagation
- Global Variable Optimizer
- Promote Memory to Register
- Dead Argument Elimination
- Combine redundant instructions
- Simplify the CFG
- Remove unused exception handling info
- Function Integration/Inlining
- Deduce function attributes
- Remove unused exception handling info
- Function Integration/Inlining
- Deduce function attributes
- SROA
- Early CSE w/ MemorySSA
- Jump Threading
- Value Propagation
- Simplify the CFG
- Combine redundant instructions
- Conditionally eliminate dead library calls
- PGOMemOPSize
- Tail Call Elimination
- Simplify the CFG
- ...
- PowerPC CTR Loops
- Safe Stack instrumentation pass
- Module Verifier
- PowerPC CTR Loops Verify
- PowerPC VSX Copy Legalization
- Expand ISel Pseudo-instructions
- Tail Duplication
- Optimize machine instruction PHIs
- Slot index numbering
- Merge disjoint stack slots
- Local Stack Slot Allocation
- Remove dead machine instructions
- Early If-Conversion
- Machine InstCombiner
- Machine Loop Invariant Code Motion
- Machine Common Subexpression Elimination
- …
Supported architectures

- AArch64
- MSP430
- AMDGPU
- MIPS
- ARM
- NVPTX
- AVR
- Nios2
- BPF
- PowerPC
- WebAssembly
- Hexagon
- RISCV
- X86
- Sparc
- XCore
- Lanai
- SystemZ
TableGen

- LLVM provides a tool called TableGen to create a source code from a table of records
  - LLVM code generator heavily uses TableGen to manage platform-dependent information, e.g. instructions and registers, in a human-friendly manner
  - Each target has own tablegen (.td) files in its Target directory; e.g. .td files for PowerPC
    - processor models (PPC.td)
    - instructions (PPCInstrInfo.td, PPCInstr64Bit.td, PPCInstrAltivec.td, ..)
    - registers (PPCRegisterInfo.td)
    - calling convention (PPCCallingConv.td)
    - instruction scheduler (PPCSchedule.td, PPCScheduleP8.td, ..)
    - binary encoding (PPCInstrFormat.td)
Directory structure

- `/` (root of LLVM repository)
  - `lib`
    - IR, ADT, … (IR and data types)
    - Transforms, Analysis (optimization and analysis passes for LLVM IR)
    - CodeGen (platform-common part of the backend)
      - SelectionDAG (DAG-based instruction selection)
    - Target (platform-dependent part of the backend)
      - X86, PowerPC, ARM, NVPTX, AMDGPU …
  - `include/llvm` (header files)
    - similar to lib subtree (each target has own header files under lib)
  - `test`, `unittests` (unit tests)
  - `tools`
    - `llc`, `lli`, `opt`, …
    - `clang`, `lld` (sub projects)
  - `projects`
    - `compiler-rt`, `libomp`, `libcxx`, … (sub projects)
test case - for optimization pass

; RUN: opt -dce -S < %s | FileCheck %s

; CHECK-LABEL: @test
define void @test()
{
; CHECK-NOT: add
  %add = add i32 1, 2
; CHECK-NOT: sub
  %sub = sub i32 %add, 1
  ret void
}
test case - for backend

; RUN: llc -relocation-model=static -verify-machineinstrs -mcpu=pwr7 -mtriple=powerpc64-unknown-linux-gnu -mattr=+vsx < %s | FileCheck %s
; RUN: llc -relocation-model=static -verify-machineinstrs -mcpu=pwr8 -mtriple=powerpc64le-unknown-linux-gnu -mattr=+vsx < %s | FileCheck -check-prefix=CHECK-LE %s

define void @test33u(<4 x float>* %a, <4 x float> %b) {
  store <4 x float> %b, <4 x float>* %a, align 8
  ret void

; CHECK-LABEL: @test33u
; CHECK: stxvw4x 34, 0, 3
; CHECK: blr

; CHECK-LE-LABEL: @test33u
; CHECK-LE: xxswapd [[V1:[0-9]+]], 34
; CHECK-LE: stxvd2x [[V1]], 0, 3
; CHECK-LE: blr
}
How to contribute?

- You can get the latest development tree from SVN repository ([https://llvm.org/svn/llvm-project/cfe/trunk/](https://llvm.org/svn/llvm-project/cfe/trunk/))
  - There is a mirror on github ([https://github.com/llvm-mirror/llvm](https://github.com/llvm-mirror/llvm))
- You can submit a patch for review in phabricator ([https://reviews.llvm.org](https://reviews.llvm.org))
  - You cannot submit pull requests in github mirror!
- You can submit a bug report in bugzilla ([https://bugs.llvm.org](https://bugs.llvm.org))
- You can ask commit access after (typically) four contributions