The Low Level Virtual Machine

Mattias Holm, MSc
holm at liacs.nl

Universiteit Leiden
The Netherlands

What is it?

- Developed at University of Illinois at Urbana-Champaign
- Compiler infrastructure
- Virtual instruction set
- Compilation strategy
  - Allows for compile-, link-, run- and install-time optimisation

What is it?

- Platform dependent bit code (pointer sizes may differ between targets)
- Single Static Assignment (SSA)
- C++ API for writing optimisation passes
  - Very clean C++
- A bit code assembly language
- Tools, lots of them
- Libraries
- Code generation (ARM, PPC, SPARC, x86 etc)

What is it NOT?

- High Level Virtual Machine (JVM/CLI)
  - Does not require GC
  - Does not require runtime codegen

LLVM Usage

- Apple
  - MacOS X OpenGL stack
  - iPhone OS compiler tool chain
- Adobe
  - Alchemy (ActionScript for Flash)
  - Hydra
- Google
  - Unladen Swallow (JIT for Python)
- Others (LDC etc)

The Tools

- LLC: The bitcode compiler
- LLI: The bitcode interpreter
- LLVM-LD: The LLVM linker
- LLVM-PROF: The llvm code profiler
- OPT: Debugging tool for running optimisation passes manually
- LLVMC: LLVM compiler driver
- LLVM-GCC: GCC based frontend (C, Objective C, C++ Ada, Fortran etc)
- CLANG: Custom C-frontend (C, Objective C and soon C++)

The Libraries

- Analysis Passes
- Instrumentation Passes
- Optimisation Passes
- Code Generation
- IR-builders
- JIT

LLVM Assembly Language

- Load-Store like model (RISC)
  - Load / store to get memory into an unlimited set of registers
  - All other instructions use registers

LLVM Instructions

- Arithmetic instructions for integers, floats and vectors
  - Add, sub, mul, udiv, sdiv, urem, srem
  - Shl, lshr, ashr, and, or, xor
- Memory instructions
  - Alloca, load, store
  - getelementptr: Computes addresses of elements in aggregated types

LLVM Instructions

- Casts and conversion
  - trunk, zext, sext, bitcast, ptrtoint*, uitofp*, sitofp*
- Other Instructions
  - Icmp, fcmp
  - Phi
  - Select
  - Call, br, switch, invoke, unwind, indirectbr, ret

LLVM Types

- Integers: i1, i2, i3, i8, i16, i32, i64...
- Reals: float, double, fp128, x86_fp80
- Arrays: [20 x i32], [4 x [4 x float]]
- Records: {i32, float, \2*}
- Packed records: <{i8, i32}>
- Vectors: <4 x float>
- Pointers: i8*
- Typedefs: %foo = type { %foo*, i32 }
- Metadata: metadata !{i32 0}

Assembly Example 1

Simple C-code

```
double
multiply(double x, double y)
{
  return x * y;
}
```

LLVM Assembly

```
define double @multiply(double %x, double %y) nounwind ssp {
entry:
  %retval = alloca double                         ; <double*> [#uses=2]
  %x.addr = alloca double                         ; <double*> [#uses=2]
  %y.addr = alloca double                         ; <double*> [#uses=2]
  store double %x, double* %x.addr
  store double %y, double* %y.addr
  %tmp = load double* %x.addr                     ; <double> [#uses=1]
  %tmp1 = load double* %y.addr                    ; <double> [#uses=1]
  %mul = fmul double %tmp, %tmp1                  ; <double> [#uses=1]
  store double %mul, double* %retval
  %0 = load double* %retval                       ; <double> [#uses=1]
  ret double %0
}
```

Assembly Example 2

```
#include <stdio.h>

int main (int argc, char const *argv[])
{
  int clargs = argc - 1;
  printf("You specified %d command line "
         "arguments\n", clargs);
  return 0;
}
```

Assembly Example 2

```
; ModuleID = 'foo.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-f32:32:32-
f64:32:64-v64:64:64-v128:128:128-a0:0:64-f80:128:128"
target triple = "i386-apple-darwin9.0"
@"\01LC" = internal constant [41 x i8] c"You specified %d command line arguments\0A\00"
          ; <[41 x i8]*> [#uses=1]

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
          %retval = alloca i32                       ; <i32*> [#uses=2]
          %argc.addr = alloca i32                    ; <i32*> [#uses=2]
          %argv.addr = alloca i8**                   ; <i8***> [#uses=1]
          %clargs = alloca i32, align 4              ; <i32*> [#uses=2]
          store i32 %argc, i32* %argc.addr
          store i8** %argv, i8*** %argv.addr
          %tmp = load i32* %argc.addr                ; <i32> [#uses=1]
          %sub = sub i32 %tmp, 1                     ; <i32> [#uses=1]
          store i32 %sub, i32* %clargs
          %tmp1 = load i32* %clargs                  ; <i32> [#uses=1]
          %call = call i32 (i8*, ...)*
                      @printf(i8* getelementptr ([41 x i8]* @"\01LC", i32 0, i32 0),
                              i32 %tmp1)             ; <i32> [#uses=0]
          store i32 0, i32* %retval
          %0 = load i32* %retval                     ; <i32> [#uses=1]
          ret i32 %0
}

declare i32 @printf(i8*, ...)
```

Extending LLVM

- Pass chains
  - Module, function, basic block, loop passes
  - Analysis passes
  - Optimisation passes
- Adding intrinsic functions or instructions
  - Can have special treatment of special *normal* functions
- Metadata

LLVM-GCC and DRAGONEGG

- LLVM-GCC (pre 4.5) based frontend to LLVM
  - C, C++, Objective-C
  - Fortran
  - Ada
- GCC 4.5 use plugins
  - Dragonegg

Clang

- Frontend for the C language family
  - C, C++ and Objective C
  - C++ reasonably robust, but misses C++0x support
- Clang is self hosting as of 2010-02-04
- Libified architecture
- Easy to get started with
  - Hackable
  - Modular
- Faster than GCC in most cases

Clang

- Libraries
  - Lexer
  - Parser
  - Sema
  - AST
  - Codegen
  - Rewrite
  - etc

Demo Time

- Using Clang to generate bitcode
- Using the LLVM linker to link modules
- Using the opt tool to run an optimisation pass
- Using LLVM to generate native code

How to Get Started

- Documentation
  - http://llvm.org/docs/
- Kaleidoscope Tutorial
  - http://llvm.org/docs/tutorial/
- Official Getting Started Guide
  - http://llvm.org/docs/GettingStarted.html
- How to Write an Optimisation Pass
  - http://llvm.org/docs/WritingAnLLVMPass.html

Questions & discussion