Contemporary Compilers
by Aaron Myles Landwehr
LLVM

- Formally “Low Level Virtual Machine”
- A Compiler written in C++ (no exceptions or RTTI) – see [here](#).
  - Started in 2000 at University of Illinois at Urbana–Champaign.
  - BSD-Style License (not a Copyleft license: no restrictions on how code is used)
  - Started by Chris Lattner (now at Apple)
  - Compiles IR into target ASM (or Machine Code)
    - No linking though – *yet*: must use a separate linker (gnu ld, msvc link.exe, gold, OSX Linker, MCLinker).

- Primary compiler for OSX user-land and IOS (OSX Kernel is still GCC)
  - Apple took interest for a number of reasons:
    - LLVM has a less restrictive license than GCC.
    - Objective-C: low priority for gcc - stagnant.
    - GCC more difficult to hack.
Clang

- Compiler Front end for LLVM.
- Compiles C, C++, Objective-C, and Objective-C++ into LLVM IR.
- Using Clang in conjunction with LLVM replaces the GCC stack.
Why use LLVM?

- Modern Compiler (with an arguably modular design).
- Language Agnostic.
- Better documentation (compared to alternatives).
- Less restrictive license.
- Easier to extend, add optimizations, add new targets, etc.
LLVM Toolchain at a High-Level
LLVM ASM (Intermediate Representation)

- A Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly.

- Contains many instructions normally found in target assemblies:
  - Binary operations:
    - ret, br, add, sub, mul, udiv, sdiv, urem, srem, fadd, fsub, fmul, fdiv.
  - Bitwise operations:
    - shl, lshr(logical), ashr (arithmetic), and, or, xor
  - Comparisons
    - icmp, fcmp (perhaps, ASMs don’t normally have this form).
  - Memory operations
    - load, store, cmpxchg
Other Instructions in the LLVM IR

- Contains many other operations:
  - phi, select, call, va_arg, fence, getelementptr, switch, *et cetera*.

- Conversion operations:
  - trunc, zext, sext, fpext, fptoui, fptosi, uitofp, sitofp, ptrtoint, inttoptr, bitcast

- Intrinsic functions
  - memcpy, cos, sin, log, exp, pow, *et cetera*. 
IR Type System

- The IR is strongly typed.
- Instructions use these types:
  - Integer
    - $i_1, i_2, i_3$, ..., $i_8$, ..., $i_{16}$, ..., $i_{32}$, ..., $i_{64}$, ...
  - Float
    - Half, float, double,
    - fp128 (128-bit floating point value (112-bit mantissa)),
    - x86_fp80 (80-bit floating point value (X87)),
    - ppc_fp128 (128-bit floating point value (two 64-bits))
  - Pointer, vector, structure, array, label, meta data.
  - Others...
LLVM IR Closer to High Level

- The IR supports global variables, functions, aliases, linkage types.
- Has more in common with a high level language than a normal assembly language. Organized into modules that can be linked together:

```assembly
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {
    ; i32()
    ; Convert [13 x i8] to i8 *
    %cast210 = getelementptr [13 x i8]* @.str, i64 0, i64 0

    ; Call puts function to write out the string to stdout.
call i32 @puts(i8* %cast210)
    ret i32 0
}
LLVM IR Example Module (Using ExampleOne)

- How to compile into LLVM IR:
  - clang -O3 -emit-llvm -S exampleOne.c -o exampleOne.ll

- OR
  - View the exampleOne.c and exampleOne.ll files in the additional materials.
LLVM Infrastructure at a Low Level View

- Different Sections to be explained...

- LLVM IR
- IR Passes
- Instruction Selection
- Scheduling and Formation
- SSA-based Machine Code Optimizations
- Code Emission
- Late Machine Code Optimizations
- Prolog/Epilog Code Insertion
- Register Allocation
For Optimizations, Analysis, and Transformations

LLVM Passes
LLVM Analysis and Transform Passes

- Passes perform transformations and optimizations that make up the compiler.
- Perform analysis (to aid other transformations, or to aid the programmer).
- They can operate in two distinct phases:
  - Before instruction selection (Operating on the LLVM IR).
    - For applying machine independent optimizations and transformations.
  - After Instruction Selection and Scheduling and Formation
    - Operating on the Machine dependent Representation.
      - Three types: SSA-based/Pre-RF, RA, non-SSA/Post-RF.
      - For applying machine specific optimizations and transformations.
- Support for different types of passes: function, basic block, loop, regions, call graph, etc.
- Mechanisms to handle pipelining passes, dependencies and interactions.
Pass Phases

- One that operates on the high level IR.
- One that operates on the machine representation (Machine Passes).

Diagram:
- LLVM IR
  - IR Passes
  - Code Emission
  - LLVM IR
- Normal Passes
  - Instruction Selection
  - Late Machine Code Optimizations
- Machine post-RA Code Passes
  - Scheduling and Formation
  - Prolog/Epilog Code Insertion
- Machine RA Code Passes
  - SSA-based Machine Code Optimizations
- Machine Pre-RA Code Passes
  - Register Allocation
Example Pass (using `exampleTwo` and `exampleThree`)

1. `clang -emit-llvm exampleTwo.c -S -o exampleTwo.ll`

2. Demo CFG
   - **As a Loadable Module** (AKA Not in Windows ;-) ) – See [here](#).
     - `opt -load /path/to/llvm/lib/LLVMAViewCFG.so -a-view-cfg exampleTwo.ll > /dev/null`
   - **Integrated into Opt:**
     - `opt -a-view-cfg exampleTwo.ll > /dev/null`

3. Demo Dom
   - `opt -view-dom exampleTwo.ll > /dev/null`

4. Demo phi nodes
   1. `clang -O1 -emit-llvm exampleThree.c -S -o exampleThree.ll`
   2. `opt -a-print-phi exampleThree.ll > /dev/null`
Example Pass (using `exampleTwo` and `exampleThree`) Cont.

- View the additional materials:
  - `exampleTwo_CFG.dot` – Control Flow Graph.
  - `exampleTwo_DOM.dot` – Dominator Tree.
  - `exampleThree_PHI.txt` – Phi Nodes.
  - Additionally, look at the corresponding `.ll` files for the LLVM IR.
The Bulk of LLVM

LLVM Target Independent Code Generator
LLVM Target Independent Code Generator

- A framework that provides a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target.
Instruction Selection

- Instruction Selection is the process of translating LLVM code presented to the code generator into target-specific machine instructions.

- LLVM uses a SelectionDAG based instruction selector.
  - The nodes are of type SDNode (e.g. specialized classes inheriting from it).
    - e.g. LoadSDNode, StoreSDNode, ...
  - Instruction Selection is done programmatically and with pattern matching.
Example SelectionDAG (Uses exampleOne)

- View the additional materials:
  - exampleOne_DAG.dot

- Programmatically:
  - `cgdb --args llc exampleOne.ll`
  - `b DAGCombiner.cpp:Run`
  - `run`
  - `call DAG.viewGraph()`
Phases that Use the SelectionDAG

- Only two phases operate on the Selection DAG.

Diagram:

1. LLVM IR
2. IR Passes
3. Instruction Selection
4. Scheduling and Formation
5. SSA-based Machine Code Optimizations
6. Code Emission
7. Late Machine Code Optimizations
8. Prolog/Epilog Code Insertion
9. Register Allocation
Instruction Selection Cont.

- **Build initial DAG**
  - Simple translation into a DAG from the input IR (Contains illegal Ops).

- **Optimize SelectionDAG**
  - Simplify the DAG. Programmatically done (and ad-hoc)
  - See CodeGen/SelectionDAG/DAGCombiner.cpp

- **Legalize SelectionDAG Types**
  - Eliminate any types that are not supported by the target.
  - E.g. if the target doesn’t support 32 bit types, it may promote them to 64 bit types.
  - See lib/Target/TARGETNAME/TARGETNAMEISelLowering.cpp
Instruction Selection Cont. 2

- Optimize SelectionDAG
- Legalize SelectionDAG Ops
  - Eliminate operations not natively supported by the target.
  - See `lib/Target/TARGETNAME/TARGETNAMEISellLowering.cpp`
- Optimize SelectionDAG
- Select instructions from the DAG
  - Takes a legal Target-independent SelectionDAG as input and outputs a Target SelectionDAG.
  - Done via Pattern Matching (mostly).
  - In some cases it is easier to eliminate non-native operations during this phase.
  - See `lib/Target/TARGETNAME/*\.td` files.
Scheduling and Formation

- This phase takes a Target SelectionDAG and assigns an order to the operations.
  - The scheduler can pick an order depending on various constraints of the machines.
- Once the order is established, the SelectionDAG is converted into a list of Machine Instructions.
LLVM Infrastructure at a Low Level View

- Where we are next...

LLVM IR

IR Passes → Instruction Selection → Scheduling and Formation → SSA-based Machine Code Optimizations

Code Emission → Late Machine Code Optimizations → Prolog/Epilog Code Insertion → Register Allocation
SSA-based Machine Code Optimizations

- Modulo-scheduling* and peephole optimizations.
- Implemented as machine passes.
- See `lib/CodeGen/PeepholeOptimizer.cpp`
- This stage is where targets can and have implemented their own SSA-based/pre-register allocation machine passes.

- * Doesn’t exist anymore – The original implementation was SPARC specific and eventually was clobbered.
LLVM Infrastructure at a Low Level View

- Where we are next...

Diagram:

- LLVM IR
  - IR Passes
  - Instruction Selection
  - Scheduling and Formation
  - SSA-based Machine Code Optimizations
  - Register Allocation
  - Code Emission
  - Late Machine Code Optimizations
  - Prolog/Epilog Code Insertion
Register Allocation

- Transform the code from using an infinite virtual register file in SSA form to a concrete register file used by the target.
- Introduces register spilling (including spill code).
- Removed unnecessary copy instructions and replaces Phi instructions.
- Implemented as machine passes.

Register Allocators

- Fast – for debug builds, keeps values in registers and reuses registers as appropriate.
- Basic – Uses live ranges per register one at a time.
- Greedy – Highly tuned version of Basic that incorporates global live range spilling. (default)
- PBQP (Partitioned Boolean Quadratic Programming) – Uses a PBQP solver?
- Linear Scan – Old default register allocator (pre LLVM 3.0).

See Lib/CodeGen/PhiElimination.cpp & lib/CodeGen/RegAlloc*.cpp
LLVM Infrastructure at a Low Level View

- Where we are next...

1. LLVM IR
2. IR Passes
3. Instruction Selection
4. Scheduling and Formation
5. SSA-based Machine Code Optimizations
6. Prolog/Epilog Code Insertion
7. Register Allocation
8. Late Machine Code Optimizations
9. Code Emission
Prolog/Epilog Code Insertion

- At this point the machine code has been generated for functions and the amount of stack pass required is known.
- The compiler inserts the prolog and epilog code for functions.
- Frame-pointer elimination and stack packing optimizations are done here.
- See `lib/Target/TARGETNAME/TARGETNAMEFrameLowering.cpp`
LLVM Infrastructure at a Low Level View

- Where we are next...
Late Code Optimizations

- Optimizations that operate on the final machine code go here.
- Spill code scheduling and peephole optimizations.
- Implemented by the Target in `lib/Target/TARGETNAME/*` in different files as machine passes.
- This stage is where targets can and have implemented their own non-SSA based/post-register allocation machine passes.
LLVM Infrastructure at a Low Level View

- Where we are next...
Code Emission

- The stage where the code is emitted as either assembly or machine code.
- See `lib/Target/TARGETNAME/TARGETNAMEASMPrinter.cpp` (for `asm`)
- See `lib/Target/TARGETNAME/TARGETNAMEMCInstLower.cpp` (for `obj`)
- See `lib/Codegen/TargetLoweringObjectFileImpl.cpp`
- Etc.
LLVM Testing
LLVM Testing

- Contains two types:
  - Regression
    - Found under the test directory and organized under many different categories.
    - Target specific tests are under test/CodeGen/TARGETNAME/*
    - Can be run individually using llvm-lit or to check all tests run “make check”.
  - Whole Program
    - Uses the llvm test-suite.
    - Found in a separate SVN.
    - Programs written in C or C++.
      - Single source, multisource, and external benchmarks (SPEC2000, etc).
    - The suite contains reference outputs of the programs.
Regression Test Format

; RUN: llvm-as < %s | llc -march=x86-64 | FileCheck %s

define void @sub1(i32* %p, i32 %v) {
    entry:
        ; CHECK: sub1:
        ; CHECK: subl
        %0 = tail call i32 @llvm.atomic.load.sub.i32.p0i32(i32* %p, i32 %v)
        ret void
}
Regression Test Format

; RUN: llvm-as < %s | llc -march=x86-64 | FileCheck %s

define void @sub1(i32* %p, i32 %v) {
  entry:
    ; CHECK: sub1:
    ; CHECK: subl
    %0 = tail call i32 @llvm.atomic.load.sub.i32.p0i32(i32* %p, i32 %v)
    ret void
}

Normal LLVM IR
 Regression Test Format

; RUN: llvm-as < %s | llc -march=x86-64 | FileCheck %s

define void @sub1(i32* %p, i32 %v) {
  entry:
    ; CHECK: sub1:
    ; CHECK: subl
      %0 = tail call i32 @llvm.atomic.load.sub.i32.p0i32(i32* %p, i32 %v)
  ret void
}
Regression Test Format

; RUN: llvm-as < %s | llc -march=x86-64 | FileCheck %s

define void @sub1(i32* %p, i32 %v) {
  entry:
    ; CHECK: sub1:
    ; CHECK: subl
    %0 = tail call i32 @llvm.atomic.load.sub.i32.p0i32(i32* %p, i32 %v)
  ret void
}
Close to the end
LLVM Tools

- **clang**
  - Frontend for c, c++, obj-c, obj-c++.

- **llc**
  - Backend – i.e. LLVM.

- **opt**
  - Tool to run and debug passes.

- **llvm-lit**
  - Tool to run tests.
Building LLVM (and Clang)

1. Choose a wise location for your source since it cannot be moved after compilation.
2. Install g++ and cmake (from a package manager).
3. Checkout LLVM
   - `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
4. Checkout Clang
   - `cd llvm/tools`
   - `svn co http://llvm.org/svn/llvm-project/cfe/trunk clang`
5. Create a build directory (not inside of the src directory)
   - `mkdir build_dir`
   - `cd build_dir`
6. Run cmake from the build directory
   - `cmake -DCMAKE_BUILD_TYPE:STRING=Debug /path/to/llvm/src`
7. Compile
   - `make all`
   - `make check`
8. There should now be bin and lib directories (found in the main directory).
   1. Add the bin and lib directories to your PATH and LD_LIBRARY_PATH variables.
Explanation about Building LLVM (and Clang)

- Why ‘make all’?
  - We want llvm-lit to run individual tests and other developer tools.
  - Normally the internal **utils** are not built by llvm which means you would manually have to install python modules and tools to get llvm-lit to work.
  - Trust me, you don’t want to have to do that.

- Why ‘make check’?
  - This generates a configuration file for llvm-lit.
  - You technically don’t even need to wait for this command to complete beyond the first few steps.

- Why NOT ‘make install’?
  - None of the **utils** will install and only the stuff needed for running llvm will.
  - So you would need to add the bin and lib directories to your path variables anyway.
What to take away

- A contemporary compiler infrastructure eases programmer burden for newbies and seasoned veterans alike.

- Through providing well-defined mechanisms to
  - Implement new targets (target description (td, c++)).
  - Implement transformations and optimizations (passes).
  - Implement new reg schedulers (register as pass, see lib/CodeGen/RegAllocBasic.cpp)
  - Test regressions (llvm-lit) and whole programs (test-suite).
  - Visualize data (CFGs, DAGS, Dom trees).

- Documentation
  - This gives you structure and methodology.

- You can too!
Bibliography

- http://llvm.org/docs/
- http://llvm.org/docs/Passes.html
- http://stackoverflow.com/questions/5134975/what-can-make-c-rtti-undesirable
- http://edll.sourceforge.net/
- Jürgen Ributzka
- Ryan Taylor