The LLVM Compiler System

LLVM: Low Level Virtual Machine

Chris Lattner
sabre@nondot.org

Bossa International Conference
on Open Source, Mobile Internet and Multimedia
March 12, 2007
LLVM Talk Overview

- The LLVM Approach to Compilers
- The LLVM C/C++/ObjC Compiler
- Optimizing OpenGL with LLVM
- Using LLVM with Scripting Languages
Open Source Compiler Technology

• “Scripting” Language Interpreters: Python, Ruby, Perl, Javascript, etc
  − Traditionally interpreted at runtime, used by highly dynamic languages

• Java and .NET Virtual Machines
  − Run time compilation for any language that can target the JVM

• GCC: C/C++/ObjC/Ada/FORTRAN/Java Compiler
  − Static optimization and code generation at build time
OSS Compiler Technology Strengths 1/2

• Scripting/Dynamic Language Strengths:
  − Interpreters are extremely portable and small (code size)
  − Many interesting advanced compilation models (pypy, Parrot, jruby, etc)
  − Dynamic languages are very expressive and powerful

• Java Virtual Machine Strengths:
  − JVM bytecode is portable, JVMs available for many systems
  − Many languages can be compiled to JVM
  − Provides runtime/JIT optimization, high level optimizations
OSS Compiler Technology Strengths 2/2

• GCC Strengths:
  – Support for important languages like C/C++
  – Other projects can emit C/C++ code and compile with GCC
  – Good code generation/optimization
  – Supports many different CPUs

• Common strengths:
  – Each has a large and vibrant community!
  – We support tons of existing code written in many languages!

With so many strengths, what could be wrong?
• **Common Problem:**
  - The tools only work together at a very coarse grain

- Each arrow/box is a completely separate project from the others
- Very little sharing (e.g.) between ruby and python interpreter
- Advanced optimizer projects don’t share code (e.g. jruby vs shedskin)
OSS Compiler Technology Weaknesses 2/2

• Scripting Language Weaknesses:
  – Efficient execution: poor “low level” performance, memory use

• Java Virtual Machine Weaknesses:
  – Must use all of JVM or none of it: GC, JIT, class library, etc
  – Forced to mold your language into the Java object model
  – Huge memory footprint and startup time

• GCC Weaknesses:
  – Old code-base and architecture: Very steep learning curve
  – Doesn’t support modern compiler techniques (JIT, cross file optimization)
  – Slow compile times

Each approach has mostly disjoint strengths and weaknesses!
LLVM Compiler Vision and Approach

• Basic mission: **build a set of modular compiler components** that:
  – ... implement aggressive and modern techniques
  – ... integrate well with each other
  – ... have few dependencies on each other
  – ... are language- and target-independent where possible
  – ... integrate closely with existing tools where possible

• Second: **Build compilers** that use these components
Value of a library-based approach

- **Reduces the time & cost** to construct a particular compiler
  - A new compiler = glue code plus any components not yet available
- Components are **shared across different compilers**
  - Improvements made for one compiler benefits the others
- Allows choice of the **right component for the job**
  - Don’t force “one true register allocator”, scheduler, or optimization order

Examples:
- **Initial bringup of llvm-gcc4** took 2-3 months (GCC is very complex!)
  - Required building “GCC tree to LLVM” converter
  - Including support for many targets, aggressive optimizations, etc
- **First OpenGL JIT** built in two weeks:
  - Required building “OpenGL to LLVM” converter
  - Replaced existing JIT, much better optimizations and performance

**Key LLVM Feature:**
IR is small, simple, easy to understand, and is well defined
Example Client: llvm-gcc4
C/C++/ObjC/...
Standard GCC 4.x Design

• Standard compiler organization: front-end, optimizer, codegen
  - Parser and front-ends work with language-specific “trees”
  - Optimizers use trees in “GIMPLE” form, modern SSA techniques, etc.
  - RTL code generator use antiquated compiler algorithms/data structures

- Pros: Excellent front-ends, support for many processors, defacto standard
- Cons: Very slow, memory hungry, hard to retarget, no JIT, no LTO, no aggressive optimizations, ...

http://llvm.org/
**llvm-gcc4 Design**

- Use GCC front-end with LLVM optimizer and code generator
  - Reuses parser, runtime libraries, and some GIMPLE lowering
  - Requires a new GCC “tree to llvm” converter
LLVM optimizer features used by llvm-gcc

- Aggressive and fast optimizer built on modern techniques
  - SSA-based optimizer for light-weight (fast) and aggressive xforms
  - Aggressive loop optimizations: unrolling, unswitching, mem promotion, ...
  - Many InterProcedural (cross function) optimizations: inlining, dead arg elimination, global variable optimization, IP constant prop, EH optzn, ...

http://llvm.org/
Other LLVM features used by llvm-gcc

- LLVM Code Generator
  - Modern retargetable code generator, easier to retarget than GCC
- Write LLVM IR to disk for codegen after compile time:
  - link-time, install-time, run-time

Language Front-end ➔ LLVM Optimizer ➔ LLVM Code Generator ➔ .s file

C ➔ LLVM IR ➔ LLVM IR ➔ .s file
C++ ➔ LLVM IR ➔ LLVM IR ➔ .s file
ObjC ➔ LLVM IR ➔ LLVM IR ➔ .s file
...

DISK ➔ LLVM IR ➔ LLVM IR ➔ LLVM IR ➔ LLVM IR ➔ LLVM IR

JIT ➔ LTO ➔ exe file

Ship ➔ Code Gen ➔ Install Time Code Generation

http://llvm.org/
Link-Time Optimization (LTO)

- Link-time is a natural place for interprocedural optimizations
  - Cross-module optimization is natural and trivial (no makefile changes)
  - LLVM is safe with partial programs (dynamically loaded code, libraries, etc)
  - LTO has been available since LLVM 1.0!
Example Client: OpenGL JIT
OpenGL Vertex/Pixel Shaders
OpenGL Pixel/Vertex Shaders

• Small program, provided at run-time, to be run on each vertex/pixel:
  – Written in one of a few high-level graphics languages (e.g. GLSL)
  – Executed millions of times, extremely performance sensitive
• Ideally, these are executed on the graphics card:
  – What if hardware doesn’t support some feature? (e.g. laptop gfx)
  – Interpret or JIT on main CPU

```c
void main() {
  vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
  vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
  vec3 lightVec = normalize(LightPosition - ecPosition);
  vec3 reflectVec = reflect(-lightVec, tnorm);
  vec3 viewVec = normalize(-ecPosition);
  float diffuse = max(dot(lightVec, tnorm), 0.0);
  float spec = 0.0;
  if (diffuse > 0.0) {
    spec = max(dot(reflectVec, viewVec), 0.0);
    spec = pow(spec, 16.0);
  }
  LightIntensity = DiffuseContribution * diffuse +
                  SpecularContribution * spec;
  MCposition = gl_Vertex.xy;
  gl_Position = ftransform();
}
```

GLSL Vertex Shader
Traditional OpenGL Impl - Before LLVM

• Custom JIT for X86-32 and PPC-32:
  – Very simple codegen: Glued chunks of Altivec or SSE code
  – Little optimization across operations (e.g. scheduling)
  – Very fragile, hard to understand and change (hex opcodes)

• OpenGL Interpreter:
  – JIT didn’t support all OpenGL features: fallback to interpreter
  – Interpreter was very slow, 100x or worse than JIT
OpenGL JIT built with LLVM Components

- At runtime, build LLVM IR for program, optimize, JIT:
  - Result supports any target LLVM supports
  - Generated code is as good as an optimizing static compiler
- Other LLVM improvements to optimizer/codegen improves OpenGL
- Key question: How does the “OpenGL to LLVM” stage work?

http://llvm.org/
Structure of an Interpreter

• Simple opcode-based dispatch loop:

    while (...) {
        ...
        switch (cur_opcode) {
            case dotproduct:      result = opengl_dot(lhs, rhs); break;
            case texturelookup: result = opengl_texlookup(lhs, rhs); break;
            case ...
        }
    }

• One function per operation, written in C:

    double opengl_dot(vec3 LHS, vec3 RHS) {
        #ifdef ALTIVEC
            ... altivec intrinsics ...
        #elif SSE
            ... sse intrinsics ...
        #else
            ... generic c code ...
        #endif
    }

• In a high-level language like GLSL, each op can be hundreds of LOC

Key Advantage of an Interpreter: Easy to understand and debug, easy to write each operation (each operation is just C code)
• At OpenGL build time, compile each opcode to LLVM bytecode:
  - Same code used by the interpreter: easy to understand/change/optimize
OpenGL to LLVM: At runtime

1. Translate OpenGL AST into LLVM call instructions: one per operation
2. Use the LLVM inliner to inline opcodes from precompiled bytecode
3. Optimize/codegen as before

vec3 viewVec = normalize(-ecPosition);
float diffuse = max(dot(lightVec, tnorm), 0.0);
...
%tmp1 = call opengl_negate(ecPosition)
%tmp3 = shuffle <4 x float> %tmp1, ...
%tmp4 = mul <4 x float> %tmp3, %tmp3

GLSL → OpenGL Parser → OpenGL to LLVM → LLVM Optimizer → LLVM JIT

OpenGL to LLVM

...
Benefits of this approach

• Key features of this approach:
  – Each opcode is written/debugged for a simple interpreter, in standard C
  – Retains all advantages of an interpreter: debugability, understandability, etc
  – Easy to make algorithmic changes to opcodes
  – OpenGL runtime is independent of opcode implementation
  – LLVM provides high performance: optimizations, regalloc, scheduling, etc

• Continuing benefits of using LLVM:
  – Support for new platforms for free
  – LLVM codegen continues to improve as it is used by other projects (e.g. llvm-gcc)
  – OpenGL group doesn’t maintain their own JIT!
Example Client: a Scripting Language
Loosely based on “pypy” Python Compiler
LLVM and Dynamic Languages

• Dynamic languages are very different than C:
  – Extremely polymorphic, reflective, dynamically extensible
  – Standard compiler optzns don’t help much if “+” is a dynamic method call

• Observation: in many important cases, dynamism is eliminable
  – Solution: Use dataflow and static analysis to infer types:
    
    ```
    var i;
    for (i = 0; i < 10; ++i)
    ... A[i] ...
    ```
    
    ‘i’ starts as an integer
    ++ on integer returns integer
    i isn’t modified anywhere else

  – We proved “i” is always an integer: change its type to integer instead of object
  – Operations on “i” are now not dynamic
  – Faster, can be optimized by LLVM (e.g. loop unrolling)
Advantages and Limitations of Static Analysis

• Works on **unmodified programs** in scripting languages:
  – No need for user annotations, no need for sub-languages

• Many approaches for doing the analysis (with cost/benefit tradeoffs)

• Most of the analyses could work with many scripting languages:
  – Parameterize the model with info about the language operations

• Limitation: cannot find all types in general!
  – That’s ok though, the more we can prove, the faster it goes
Scripting Language Performance

- Ahead-of-Time Compilation provides:
  - Reduced memory footprint (no ASTs in memory)
  - Eliminate (if no ‘eval’) or reduce use of interpreter at runtime (save code size)
  - Much better performance if type inference is successful

- JIT compilation provides:
  - Full support for optimizing eval’d code (e.g. json objects in javascript)
  - Runtime “type profiling” for speculative optimizations

- LLVM provides:
  - Both of the above, with one language -> LLVM translator
  - Install-time codegen
  - Continuously improving set of optimizations and targets
  - Ability to inline & optimize code from different languages
  - inline your runtime library into the client code?
Call for help!

- OSS community needs to unite work on various scripting languages
- Common module to represent/type infer an arbitrary dynamic language
- Who will provide this? pypy? parrot? llvm itself someday ("hlvm")?
LLVM Summary

• LLVM is a set of modular compiler components:
  – LLVM can be used for many things other than simple static compilers!
  – LLVM can be used to make a great static compiler! (llvm-gcc)

• LLVM components are language- and target-independent:
  – Does not force use of JIT, GC, or a particular object model
  – Code from different languages can be linked together and optimized

• LLVM provides aggressive functionality and is industrial strength:
  – Interprocedural optimization, link-time, install-time optimization today!
  – LLVM has compiled and optimized millions of lines of code

• LLVM 2.0 due out in May:
  – Huge number of new features, codegen improvements
  – Full ARM support, contributed by INdT, enhanced by Apple

sabre@nondot.org

http://llvm.org/