EECS 583 – Advanced Compilers
Course Overview,
Introduction to Control Flow Analysis

Fall 2016, University of Michigan

September 12, 2016
About Me

v Lingjia Tang
v Research area: compiler/system/architecture
  » Dynamic compiler
  » Datacenter
  » Clarity-lab: http://clarity-lab.org/
v Joined Michigan in 2013
v Before: UCSD, UVa
v Industry: Google
Papers

2016

[Bibtex]

[Bibtex]
Class Overview

☐ This class is NOT about:
  » Programming languages
  » Compiler Frontend: Parsing, syntax checking, semantic analysis
  » Debugging
  » Simulation
  » Handling advanced language features – virtual functions, …

☐ Compiler backend
  » Mapping applications to processor hardware
  » Analysis, optimizations, code generation
  » Retargetability – work for multiple platforms (not hard coded)
  » Work at the assembly-code level (but processor independent)
  » Speed/Efficiency
    • How to make the application run fast
    • Use less memory, efficiently execute
    • Parallelize, prefetch, optimize using profile information
Compilation Phases

- Source Code
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization
- Machine Code
Background You Should Have

- 1. Programming
  » Good C++ programmer (essential)
  » Linux, gcc, emacs
  » Debugging experience – hard to debug with printf’s alone – gdb!

- 2. Computer architecture
  » EECS 370 is good, 470 is better but not essential
  » Basics – caches, pipelining, function units, registers, virtual memory, branches, multiple cores, assembly code

- 3. Compilers
  » Frontend stuff is not very relevant for this class
  » Basic backend stuff we will go over fast
    • Non-EECS 483 people will have to do some supplemental reading
Textbook

- No required text – Lecture notes, papers
- 2 reference books: the Dragon book and Muchnick
Other Material

❖ Course webpage + piazza
  » [http://www.eecs.umich.edu/courses/eecs583](http://www.eecs.umich.edu/courses/eecs583)
  » Lecture notes – available the night before class
  » Piazza – ask/answer questions, GSI and I will try to check regularly but may not be able to do so always
    • [http://www.piazza.com](http://www.piazza.com)

❖ LLVM compiler system
  » LLVM webpage: [http://www.llvm.org](http://www.llvm.org)
  » Read the documentation!
  » LLVM users group
What the Class Will be Like

- Class meeting time – 10:30 – 12:30, MW
  - 2 hrs is hard to handle
  - We’ll stop at 12:00, most of the time
- Core backend stuff
  - Text book material – some overlap with 483
  - 2 homeworks to apply classroom material
- Research papers
  - Last 1/3rd of the semester, students take over
  - I will recommend papers of several topics
  - Select paper related to your project – entire class is expected to read the paper
  - Each project team - presents 1 paper. 20 min presentation + 5 min Q&A.
What the Class Will be Like (2)

- Learning compilers
  - No memorizing definitions, terms, formulas, algorithms, etc
  - Learn by doing – Writing code
  - Substantial amount of programming
    - Fair learning curve for LLVM compiler
  - Reasonable amount of reading

- Classroom
  - Attendance – You should be here
  - Discussion important
    - Work out examples, discuss papers, etc
  - Essential to stay caught up
  - Extra meetings outside of class to discuss projects
Course Grading

- Yes, everyone will get a grade
  » Most (hopefully all) will get A’s and B’s
  » Slackers will be obvious

- Components
  » Midterm exam – 25%
  » Project – 45%
  » Homeworks – 15%
  » Paper presentation – 10%
  » Class participation – 5%
Homeworks

- 2 of these
  - 1 small & 1 hard programming assignment
  - Design and implement something we discussed in class

- Goals
  - Learn the important concepts
  - Learn the compiler infrastructure so you can do the project

- Grading
  - Working testcases?, Does anything work? Level of effort?

- Working together on the concepts is fine
  - Make sure you understand things or it will come back to bite you
  - Everyone must do and turn in their own assignment
Projects – Most Important Part of the Class

- Design and implement an “interesting” compiler technique and demonstrate its usefulness using LLVM

- Topic/scope/work
  - 2-4 people per project (1 person, 5 persons allowed in some cases)
  - You will pick the topics (I have to agree)
  - You will have to
    - Read background material
    - Plan and design
    - Implement and debug

- Deliverables
  - Working implementation
  - Project report: ~5 page paper describing what you did/results
  - 15-20 min presentation at end (demo if you want)
  - Project proposal (late Oct) and status report (late Nov) scheduled with each group during semester
Types of Projects

v New idea
  » Small research idea
  » Design and implement it, see how it works

v Extend existing idea (most popular)
  » Take an existing paper, implement their technique
  » Then, extend it to do something interesting
    • Generalize strategy, make more efficient/effective

v Implementation
  » Take existing idea, create quality implementation in LLVM
  » Try to get your code released into main LLVM system

v Using other compilers/systems (GPUs, mobile phone, etc.) is possible but need a good reason
Topic Areas (You are Welcome to Propose Others)

- Memory system performance
  - Instruction/data prefetching
  - Use of scratchpad memories

- Automatic parallelization
  - Loop parallelization
  - Vectorization/SIMDization

- Reliability
  - Reducing AVF
  - Application-specific techniques

- Power
  - Identification of power-intensive computation
  - Instruction scheduling techniques to reduce power

- Multicore/manycore/GPU
  - Cache contention

- For the adventurous - Dynamic optimization
  - DynamoRIO
  - Protean Code
    - Run-time parallelization or other optimizations are interesting
    - Hybrid processors: Transmeta style processor (Nvidia’s Denver)

- Approximate computing
- Machine learning
Class Participation

- Interaction and discussion is essential in a graduate class
  - Be here
  - Don’t just stare at the wall
  - Be prepared to discuss the material
  - Have something useful to contribute

- Opportunities for participation
  - Research paper discussions – thoughts, comments, etc
  - Saying what you think in project discussions outside of class
  - Solving class problems
  - Asking intelligent questions
Paper Reading

❖ How to read a research paper?
  » What problem does the paper solve?
    • Is it an important problem?
  » Context of the paper?
  » What new insights does the paper provide?
    • Here’s some data that shows something that we didn’t know before about programs/architecture/compiler
  » What is the mechanism proposed in the paper?
  » What is the conclusion?
  » Are you convinced that the paper presents a good idea?
  » Does the paper raise any questions?
  » How to improve the paper?
Compiler conferences

- CGO: Code Generation and Optimization
- PLDI: Programming Language Design and Implementation
- MICRO: International Symposium on Microarchitecture
- ASPLOS:
- PACT:
- PPoPP
GSI

- Animesh Jain (anijain@umich.edu)

- Office hours
  - ??
  - Location: ??

- LLVM help/questions

- But, you will have to be independent in this class
  - Read the documentation and look at the code
  - Come to him when you are really stuck or confused
  - He cannot and will not debug everyone’s code
  - Helping each other is encouraged
  - Use the piazza group (Animesh and I will monitor this)
Contact Information

- Office: 4609 CSE
- Email: lingjia@umich.edu
- Office hours
  - Mon/Wed, 12-12:30 (right after class)
  - Or send me an email for an appointment
- Visiting office hrs
  - Mainly help on classroom material, concepts, etc.
  - I am an LLVM novice, so likely I cannot answer any non-trivial question
  - See Animesh for LLVM details
## Tentative Class Schedule

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No class</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Sept 12</td>
<td>Course intro, Control flow analysis Intro</td>
</tr>
<tr>
<td></td>
<td>Sept 14</td>
<td>Control flow analysis/LLVM Intro <a href="#">HW #1 out</a></td>
</tr>
<tr>
<td>3</td>
<td>Sept 19</td>
<td>Control flow – region formation</td>
</tr>
<tr>
<td></td>
<td>Sept 21</td>
<td>Dataflow analysis - intro</td>
</tr>
<tr>
<td>4</td>
<td>Sept 26</td>
<td>Dataflow analysis + optimization</td>
</tr>
<tr>
<td></td>
<td>Sept 28</td>
<td>SSA form <a href="#">HW #1 due HW #2 out</a></td>
</tr>
<tr>
<td>5</td>
<td>Oct 3</td>
<td>Classic optimization</td>
</tr>
<tr>
<td></td>
<td>Oct 5</td>
<td>Code generation - basics</td>
</tr>
<tr>
<td>6</td>
<td>Oct 10</td>
<td>Compiler Research Highlight 1 – Dynamic Compiler</td>
</tr>
<tr>
<td></td>
<td>Oct 12</td>
<td>Compiler Research Highlight 2 – Machine-learning based approaches</td>
</tr>
<tr>
<td>7</td>
<td>Oct 17</td>
<td>No class – Fall Break</td>
</tr>
<tr>
<td></td>
<td>Oct 19</td>
<td>Compiler Research Highlight 3 – Approximate Computation</td>
</tr>
<tr>
<td>8</td>
<td>Oct 24</td>
<td>Code generation – Superblock scheduling</td>
</tr>
<tr>
<td></td>
<td>Oct 26</td>
<td>Project proposals</td>
</tr>
<tr>
<td>9</td>
<td>Oct 31</td>
<td>Project proposals</td>
</tr>
<tr>
<td></td>
<td>Nov 2</td>
<td>Code generation – Register allocation</td>
</tr>
<tr>
<td>10</td>
<td>Nov 7</td>
<td>Midterm Exam – in class</td>
</tr>
<tr>
<td></td>
<td>Nov 9</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td>11</td>
<td>Nov 14</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td></td>
<td>Nov 16</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td>12</td>
<td>Nov 21</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td></td>
<td>Nov 23</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td>13</td>
<td>Nov 28</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td></td>
<td>Nov 30</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td>14</td>
<td>Dec 5</td>
<td>Research paper presentations</td>
</tr>
<tr>
<td></td>
<td>Dec 7</td>
<td>Project Demos</td>
</tr>
<tr>
<td>15</td>
<td>Dec 12</td>
<td>Project demos</td>
</tr>
</tbody>
</table>
Target Processors: 1) Multicore

- Sequential programs – 1 core busy, 3 sit idle
- How do we speed up sequential applications?
  - Switch from ILP to TLP as major source of performance
  - Contention for shared resources
Target Processors: 2) SIMD

- Do the same work on different data: GPU, SSE, etc.
- Energy-efficient way to scale performance
- Must find “vector parallelism”
Target Processors: 3) VLIW/EPIC Architectures

- VLIW = Very Long Instruction Word
  - Aka EPIC = Explicitly Parallel Instruction Computing
  - Compiler managed multi-issue processor

- Desktop
  - IA-64: aka Itanium I and II, Merced, McKinley, Transmeta

- Embedded processors
  - All high-performance DSPs are VLIW
    - Why? Cost/power of superscalar, more scalability
  - TI-C6x, Philips Trimedia, Starcore, ST-200
So, let’s get started… Compiler Backend IR – Our Input

- Variable home location
  - Frontend – every variable in memory
  - Backend – maximal but safe register promotion
    - All temporaries put into registers
    - All local scalars put into registers, except those accessed via &
    - All globals, local arrays/structs, unpromotable local scalars put in memory. Accessed via load/store.

- Backend IR (intermediate representation)
  - machine independent assembly code – really resource indep!
  - aka RTL (register transfer language), 3-address code
  - r1 = r2 + r3 or equivalently add r1, r2, r3
    - Opcode (add, sub, load, …)
    - Operands
      - Virtual registers – infinite number of these
      - Literals – compile-time constants
Architecture of LLVM

- LLVM (Low-level Virtual Machine)
  » Developed at UIUC (2000~ )

![Diagram of LLVM architecture](image-url)
LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

ABSTRACT
This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, language-independent type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and \texttt{setjmp/longjmp} in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for optimizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs (“idle time”) using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run well on new systems.

This paper presents LLVM — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) a code representation with several novel features that serves as a common representation for analysis, transformation, and code
Architecture of GCC
First Topic: Control Flow Analysis

- Control transfer = branch (taken or fall-through)
- Control flow
  - Branching behavior of an application
  - What sequences of instructions can be executed
- Execution → Dynamic control flow
  - Direction of a particular instance of a branch
  - Predict, speculate, squash, etc.
- Compiler → Static control flow
  - Not executing the program
  - Input not known, so what could happen
- Control flow analysis
  - Determining properties of the program branch structure
  - Determining instruction execution properties
Basic Block (BB)

- Group operations into units with equivalent execution conditions

- **Defn: Basic block** – a sequence of consecutive operations in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
  - Straight-line sequence of instructions
  - If one operation is executed in a BB, they all are

- Finding BB’s
  - The first operation starts a BB
  - Any operation that is the target of a branch starts a BB
  - Any operation that immediately follows a branch starts a BB
Identifying BBs - Example

L1: r7 = load(r8)
L2: r1 = r2 + r3
L3: beq r1, 0, L10
L4: r4 = r5 * r6
L5: r1 = r1 + 1
L6: beq r1 100 L3
L7: beq r2 100 L10
L8: r5 = r9 + 1
L9: jump L2
L10: r9 = load (r3)
L11: store(r9, r1)
Control Flow Graph (CFG)

- **Defn Control Flow Graph** – Directed graph, \( G = (V,E) \) where each vertex \( V \) is a basic block and there is an edge \( E \), \( v_1 (BB1) \rightarrow v_2 (BB2) \) if BB2 can immediately follow BB1 in some execution sequence
  - A BB has an edge to all blocks it can branch to
  - Standard representation used by many compilers
  - Often have 2 pseudo vertices
    - entry node
    - exit node
CFG Example

\[
x = z - 2; \\
y = 2 * z; \\
\text{if (c)} \{ \\
    x = x + 1; \\
    y = y + 1; \\
\} \\
\text{else} \{ \\
    x = x - 1; \\
    y = y - 1; \\
\} \\
z = x + y
\]
Weighted CFG

- **Profiling** – Run the application on 1 or more sample inputs, record some behavior
  - Control flow profiling
    - edge profile
    - block profile
  - Path profiling
  - Cache profiling
  - Memory dependence profiling
- Annotate control flow profile onto a CFG → weighted CFG
- Optimize more effectively with profile info!!
  - Optimize for the common case
  - Make educated guess
Property of CFGs: Dominator (DOM)

- **Defn: Dominator** – Given a CFG(V, E, Entry, Exit), a node x dominates a node y, if every path from the Entry block to y contains x

- 3 properties of dominators
  - Each BB dominates itself
  - If x dominates y, and y dominates z, then x dominates z
  - If x dominates z and y dominates z, then either x dominates y or y dominates x

- **Intuition**
  - Given some BB, which blocks are guaranteed to have executed prior to executing the BB
Dominator Examples
Dominator Analysis

- Compute \( \text{dom}(BB_i) = \text{set of BBs that dominate } BB_i \)
- Initialization
  - \( \text{Dom(entry)} = \text{entry} \)
  - \( \text{Dom(everything else)} = \text{all nodes} \)
- Iterative computation
  - while change, do
    - change = false
    - for each BB (except the entry BB)
      - \( \text{tmp}(BB) = BB + \{\text{intersect of Dom of all predecessor BB’s}\} \)
      - if (\( \text{tmp}(BB) \neq \text{dom}(BB) \))
        - \( \text{dom}(BB) = \text{tmp}(BB) \)
        - change = true
Immediate Dominator

- **Defn:** Immediate dominator (idom) – Each node \( n \) has a unique immediate dominator \( m \) that is the last dominator of \( n \) on any path from the initial node to \( n \)
  - Closest node that dominates
**Dominator Tree**

First BB is the root node, each node dominates all of its descendants.

<table>
<thead>
<tr>
<th>BB</th>
<th>DOM</th>
<th>BB</th>
<th>DOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1,4,5</td>
</tr>
<tr>
<td>2</td>
<td>1,2</td>
<td>6</td>
<td>1,4,6</td>
</tr>
<tr>
<td>3</td>
<td>1,3</td>
<td>7</td>
<td>1,4,7</td>
</tr>
<tr>
<td>4</td>
<td>1,4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Class Problem

Draw the dominator tree for the following CFG
If You Want to Get Started …

- Go to http://llvm.org
- Download and install LLVM 3.4 on your favorite Linux box
  » Read the installation instructions to help you
  » Will need gcc 4.x
- Try to run it on a simple C program
- Will be the first part of HW 1 that goes out next week.