CS 526
Advanced Compiler Construction
http://misailo.web.engr.Illinois.edu/courses/cs526
Goals of the Course

Develop a fundamental understanding of the major approaches to program analysis and optimization

Understand published research on various novel compiler techniques

Solve a significant compiler problem by reading the literature and implementing your solution in LLVM

Learn about current research in compiler technology
Compiler Overview

Preprocessing Source
- Automatic Parallelization
- Vectorization
- Cache Management
- Performance Modeling

Code Generation
- Source Code Portability
- Back-end Optimizations
- Static Profiling
- Power Management

Linking/Loading
- Interprocedural optimization
- Load-time optimization
- Security checking

Runtime compilation
- JIT code generation
- Runtime optimization
- Fault tolerance
COMPILER = Program Analysis + Program Transformation
Compiler Overview
Why is Optimization Important?

For source-level programming languages

Liberate programmer from machine-related issues and enable portable programming without unduly sacrificing performance.

John Backus on the first FORTRAN compiler:

“It is our belief that if FORTRAN, during its first months, were to translate any reasonable scientific program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger.”

“To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.”

Why is Optimization Important?

For simple language features

Allow programmer to focus on clean, easy-to-understand programs; avoid detailed hand-optimizations:

- **Expression simplification**: Constant folding, associativity, commutativity
- **Redundancy elimination**: Loop-invariant code motion, common subexpressions, equivalent subexpressions
- **Dead code elimination**: Unreachable code, unused computations
- **Control flow simplification**: Branch folding, branch elimination
- **Procedure call elimination**: Single-use functions, frequent function calls
- **Bounds check elimination**: Array expressions
Why is Optimization Important?
For more powerful language features

Improve programmer productivity, software reliability without unduly sacrificing performance

• **Type-safe languages**: type checking, array bounds checking, garbage collection (GC)
• **Object-oriented programming**: encapsulation; reuse; polymorphic dispatch
• **Managed runtimes**: GC; code verification; just-in-time compilation
• **Scripting languages**: interpreters; dynamic typing; domain-specific languages
• **Generic programming**: polymorphic algorithms and data types
• **First-class functions**: functional programming; lambdas/blocks
Why is Optimization Important?

For high performance

Current processors rely heavily on compilers for performance and domain specific processors and FPGAs require automated compilation of general purpose software.
Why is Optimization Important?

For new applications

Wearable computing (e-textiles)

Analog computing (Bio)

Sensor Networks
Why is Optimization Important?

To Understand

In discussing any optimization, look for three properties:

**Safety** — Does it change the results of the program?  
(static analysis, e.g., dataflow, dependence)

**Profitability** — Is it expected to speed up execution?  
(static or dynamic analysis)

**Opportunity** — Can we easily locate sites to modify?  
(find all sites; updates and orderings)
Why is Program Analysis Important?

Software Reliability and Security

Improve programmer productivity, software reliability without unduly sacrificing performance

• **PREFix, PREFast**: Identify many common bugs, vulnerabilities in Windows, .NET applications

• **Microsoft DriverVerifier**: Finds memory corruption, deadlocks and other bugs in Windows drivers

• **CodeSonar, Coverity, Fortify, PolySpace**: Find a wide range of programming errors in several different languages

**Most tools are based on program analysis, often flow-sensitive, context-sensitive, interprocedural**
Program Analysis Techniques

- over-approximation of $[P]$ (e.g. static analysis)
- under-approximation of $[P]$ (e.g. dynamic analysis)
- All behaviors in the universe

Figure from Martin Vechev, ETH
A Few Billion Lines of Code Later
Using Static Analysis to Find Bugs in the Real World

IN 2002, COVERITY COMMERCIALIZED a research static bug-finding tool.\(^5,9\) Not surprisingly, as academics, our view of commercial realities was not perfectly accurate. However, the problems we encountered were not the obvious ones. Discussions with tool researchers and system builders suggest we were not alone in our naiveté. Here, we document some of the more important examples of what we learned.
COURSE TOPICS
List of Topics (Part 1)

The order of topics is subject to change

Static Program Analysis

• Natural loops, intervals, reducibility (refresher)
• Static single assignment (SSA)
• Dataflow analysis
• Pointer analysis
• Array dependence analysis
• Interprocedural analysis
List of Topics (Part II)

Optimizations

• Code motions and redundancy elimination
• Induction variable optimizations
• Loop transformations and memory hierarchy optimizations
• Basic interprocedural optimizations

Advanced topics

• Basics of static analysis
• Ensuring correctness of compilers
• Approximate and probabilistic compilers
Compiler Overview

Program

Front-end

Optimizer

Back-end
Topics We Will Not Cover

• Back-end code generation, e.g., scheduling, allocation, software pipelining
• Automatic vectorization and parallelization
• Storage management and garbage collection
• Program verification (static and runtime)
• LLVM hacking
CS 598 SM

COURSE LOGISTICS
Schedule

Twice a week – Tuesdays and Thursdays 11-12:15 pm

Course Format

• Lectures – most of the weeks (sometimes guest)
• Projects – two programming assignments (LLVM)
• Exams – midterm and final exams
• Mini-quizzes – before (almost) every lecture
Prerequisites

Very helpful (I will assume you took it):
Basic **compilers** course (e.g., CS 426)

Also helpful:
Basic **programming languages** course (e.g., CS 421)

Basic **computer architecture** (e.g., CS 233)

(or commitment to learn as you go)
Grading

Miniquizzes  10%
Midterm Exam  15%
Final Exam  25%
Projects  50%
Miniquizzes

**Test background** knowledge (like the one today)
- **5 minutes** at the beginning of the class
- Concept from compiler theory, something that was covered in previous courses or lectures
- We will discuss solution immediately afterwards

**Each** miniquiz is **worth ~0.5%** (up to 10%).
- **Self-graded**, the main purpose is to bring everyone to the same page before we start the discussions
- In total 25 quizzes; can miss 5 without penalty
Exams

Midterm
• In class (March 16; before the break)
• Focuses on analysis (SSA, dataflow, dependency)
• 75 minutes

Final
• In the finals week
• Pointer analysis, optimization and special topics
• Also include the materials from the midterm
• Up to 3 hours
Books

No official book, but many times you will need to look into one of these:
Projects

Gain experience solving existing compiler problems
• Read the literature for the problems
• Find or develop a solution
• Implement the solution in a realistic compiler
• Test it on realistic benchmarks
Projects

P1 – Warm-up exercise:
• *Individual*, 3 weeks but do it sooner
• Scalar replacement of aggregates via SSA (Muchnick, Chapter 12)
• Goal: become familiar with the infrastructure

P2 – Main problem
• *Groups of two*, 12 weeks, also do it sooner!
• Choose and solve a harder problem (Suggestions coming soon)
Infrastructure

**LLVM: Low Level Virtual Machine** [http://llvm.org](http://llvm.org)

- Virtual instruction set: RISC-like, SSA-form
- Powerful link-time (interprocedural) optimization system
- Many front-ends: C/C++, D, Fortran, Julia, Haskell, Objective-C, OpenMP, OpenCL, Python, Swift, ...
- Software: 1.3M+ lines of C++
- Open source: In use at many universities and major companies
Infrastructure

Prepare for project, **during this week:**
Read LLVM Documentation at http://llvm.org/docs:
*Introduction to the LLVM Compiler Infrastructure*

Follow instructions in the **Getting Started** and **Writing an LLVM Pass** guides to:
(a) Download LLVM and Clang 3.9.1 and test-suite
(b) Do a full build (no need to run "make install")
(c) Compile and run the “Hello” pass

Install on **your EWS:** ssh <netid>@linux.ews.illinois.edu
Grading Summary

Miniquizzes 10%
Reviews & Discussion 15%
Paper Presentation 25%
Project 50%

Grading on an absolute scale (no curve!)
QUESTIONS SO FAR?
CONTROL FLOW ANALYSIS

The slides adapted from Vikram Adve
**Flow Graphs**

**Flow Graph:** A triple $G=(N,A,s)$, where $(N,A)$ is a (finite) directed graph, $s \in N$ is a designated “initial” node, and there is a path from node $s$ to every node $n \in N$.

- An *entry node* in a flow graph has no predecessors.
- An *exit node* in a flow graph has no successors.
- There is exactly one entry node, $s$. We can modify a general DAG to ensure this. *How?*
- In a control flow graph, any node unreachable from $s$ can be safely deleted. *Why?*
- Control flow graphs are usually *sparse*. I.e., $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$. 
Control Flow Graph (CFG)

**Basic Block** is a sequence of statements $S_1 \ldots S_n$ such that execution control must reach $S_1$ before $S_2$, and, if $S_1$ is executed, then $S_2 \ldots S_n$ are all executed in that order

- Unless a statement causes the program to halt

**Leader** is the first statement of a basic block

**Maximal Basic Block** is a basic block with a maximum number of statements ($n$)
Control Flow Graph (CFG)

CFG is a directed graph in which:
• Each node is a single basic block
• There is an edge \( b_1 \rightarrow b_2 \) if block \( b_2 \) may be executed after block \( b_1 \) in some execution

We define it typically for a single procedure

A CFG is a conservative approximation of the control flow! Why?
Example

Source Code

```c
unsigned fib(unsigned n) {
    int i;
    int f0 = 0, f1 = 1, f2;
    if (n <= 1) return n;
    for (i = 2; i <= n; i++) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
    }
    return f2;
}
```

LLVM bitcode

```llvm
define i32 @fib(i32) {
    %2 = icmp ult i32 %0, 2
    br i1 %2, label %12, label %3

    ; <label>:3:
    br label %4

    ; <label>:4:
    %5 = phi i32 [ %8, %4 ], [ 1, %3 ]
    %6 = phi i32 [ %5, %4 ], [ 0, %3 ]
    %7 = phi i32 [ %9, %4 ], [ 2, %3 ]
    %8 = add i32 %5, %6
    %9 = add i32 %7, 1
    %10 = icmp ugt i32 %9, %0
    br i1 %10, label %11, label %4

    ; <label>:11:
    br label %12

    ; <label>:12:
    %13 = phi i32 [ %0, %1 ], [ %8, %11 ]
    ret i32 %13
}
```
Dominance in Flow Graphs

Let $d, d_1, d_2, d_3, n$ be nodes in $G$.

$d$ dominates $n$ ("$d$ dom $n$") iff every path in $G$ from $s$ to $n$ contains $d$

$d$ properly dominates $n$ if $d$ dominates $n$ and $d \neq n$

$d$ is the immediate dominator of $n$ ("$d$ idom $n$") if $d$ is the last proper dominator on any path from initial node to $n$,

$\text{DOM}(x)$ denotes the set of dominators of $x$. 