CS 526

Advanced
Compiler
Construction

http://misailo.cs.Illinois.edu/courses/cs526

Goals of the Course

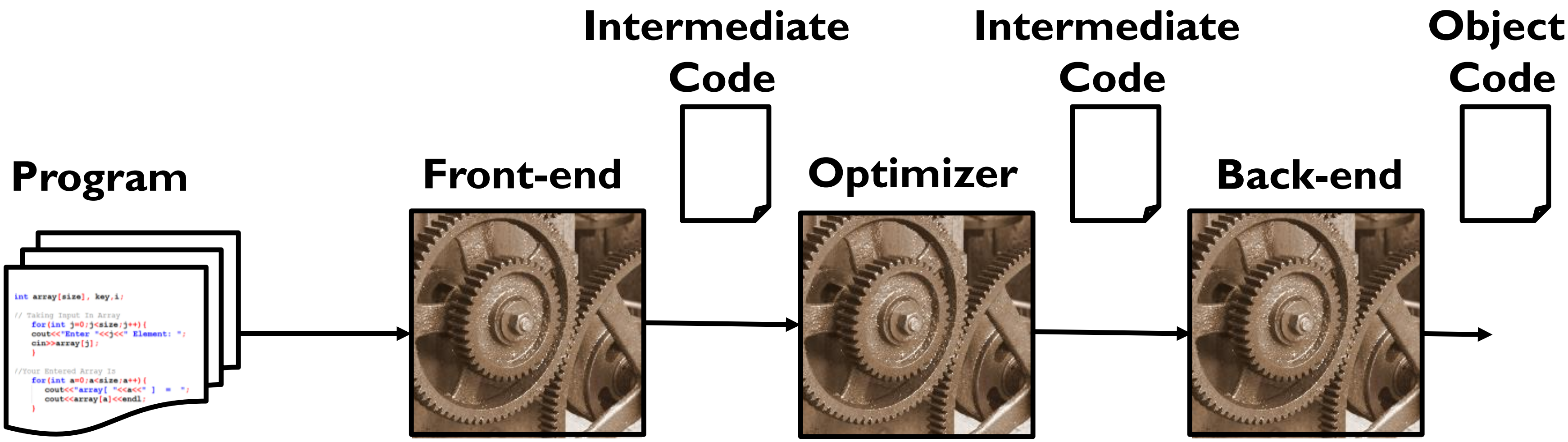
Develop a fundamental understanding of the major approaches to program analysis and optimization

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

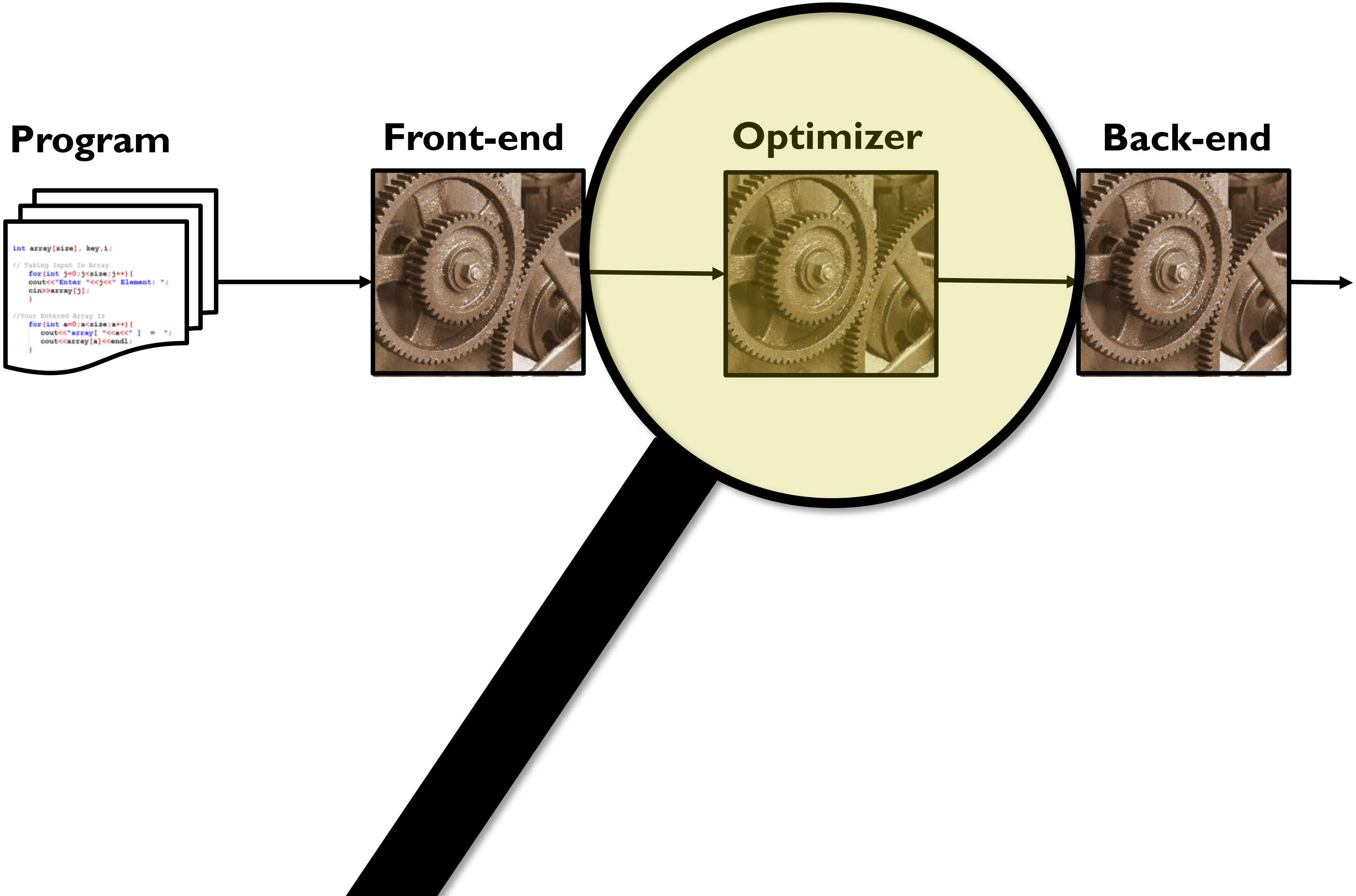
- JIT code generation
- Runtime optimization
- Fault tolerance

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."

-- John Backus, Fortran I, II and III, Annals of the History of Computing}, vol. 1, no. 1, July 1979

Why is Optimization Important?

For expressive 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

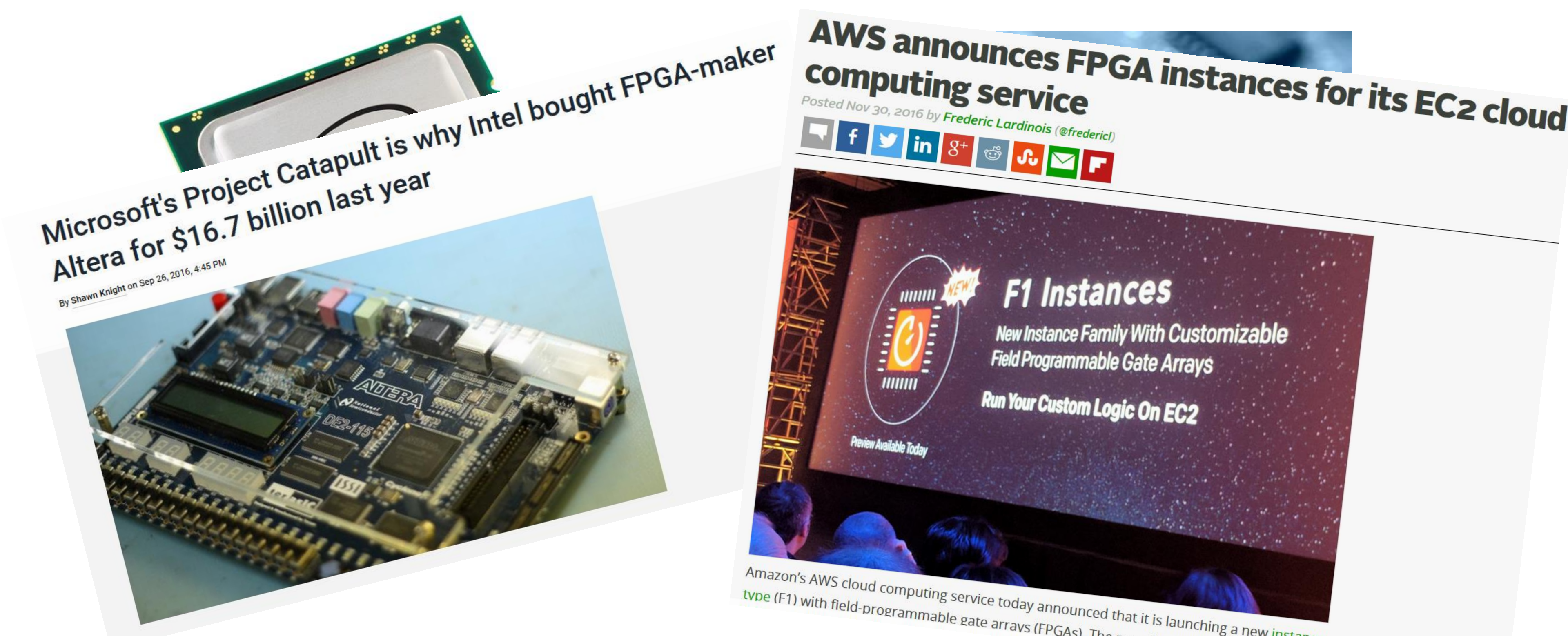
Improve programmer productivity, software reliability without unduly sacrificing performance

- **Type-safe languages:** type checking, array bounds checking, garbage collection (GC)
- **Object-oriented programming:** encapsulation; reuse; polymorphic dispatch
- **Managed runtimes:** just-in-time compilation; code verification
- **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 better performance and portability

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?

Because Moore's Law is Dead

Intelligent Machines

DARPA has an ambitious $1.5 billion plan to reinvent electronics

The US military agency is worried the country could lose its edge in semiconductor chips with the end of Moore's Law.

by Martin Giles July 30, 2018

Last year, the Defense Advanced Research Projects Agency (DARPA), which funds a range of blue-sky research efforts relevant to the US military, launched a $1.5 billion, five-year program known as the Electronics Resurgence Initiative (ERI) to support work on advances in chip technology. The agency has just unveiled the first set of research teams selected to explore unproven but

How are we going to leverage new post-Moore architectures?

Why is Optimization Important?

For new applications

Wearable computing (e-textiles)

Analog nano-computing (Bio)

Self Driving Cars

Edge intelligence

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 Driver Verifier:** 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

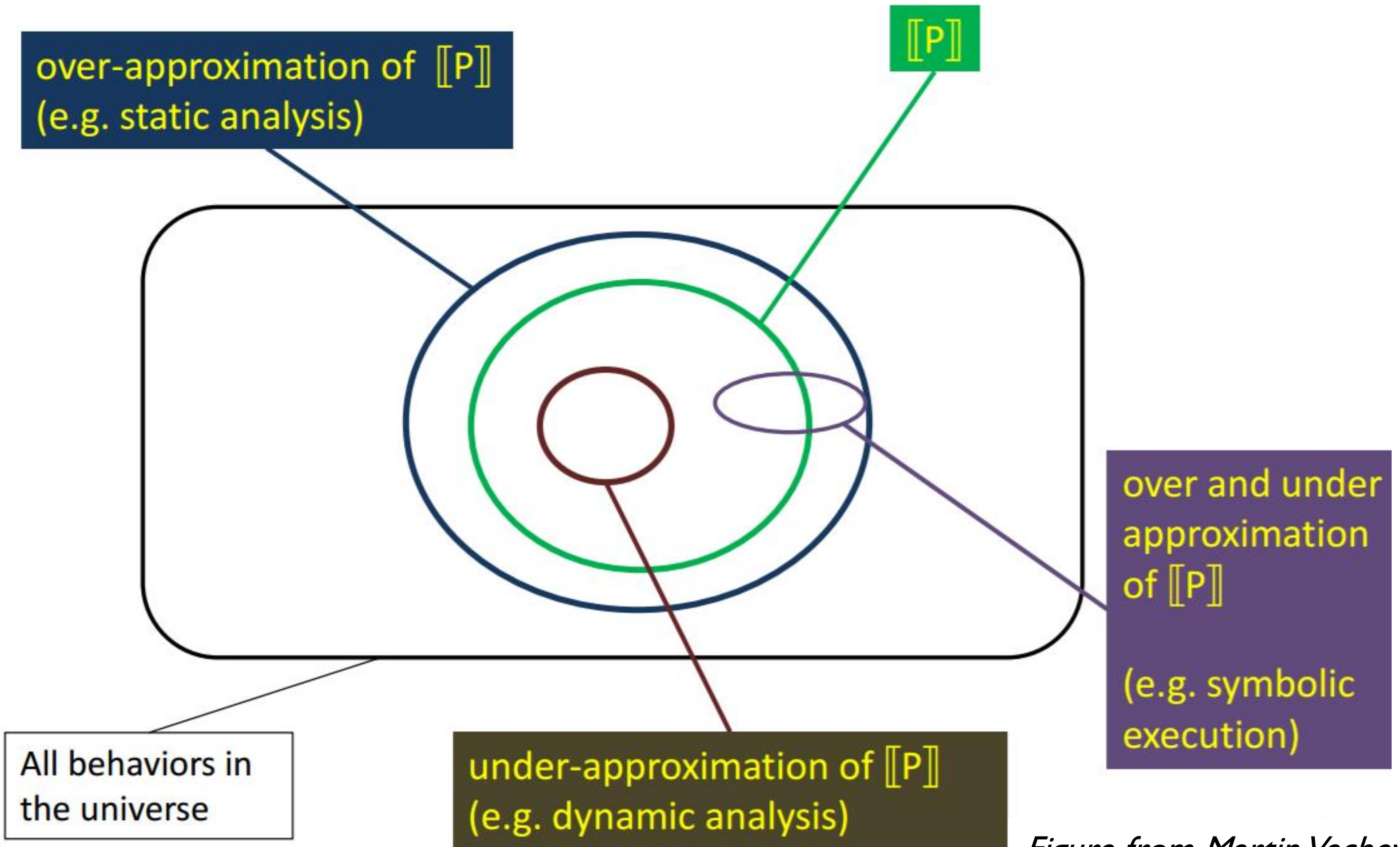


Figure from Martin Vechev, ETH

How Coverity built a bug-finding tool, and a business, around the unlimited supply of bugs in software systems.

BY AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPEAK, AND DAWSON ENGLER

A Few Billion Lines of Code Later

Using Static Analysis to Find Bugs in the Real World

IN 2002, COVERITY commercialized[3] a research static bug-finding tool.[6,9] 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 naïveté. Here, we document some of the more important examples of what we learned

the fact that programming rules often map clearly to source code; thus static inspection can find many of their violations. For example, to check the rule "acquired locks must be released," a checker would look for relevant operations (such as `lock()` and `unlock()`) and inspect the code path after flagging rule disobedience (such as `lock()` with no `unlock()` and double locking).

For those who keep track of such things, checkers in the research system typically traverse program paths (flow-sensitive) in a forward direction, going across function calls (inter-procedural) while keeping track of call-site-specific information (context-sensitive) and toward the end of the effort had some of the support needed to detect when a path was infeasible (path-sensitive).

A glance through the literature reveals many ways to go about static bug finding.[1,2,4,7,8,11] For us, the central religion was results: If it worked, it was good, and if not, not. The ideal: check millions of lines of code with little manual setup and find the maximum number of serious true errors with the minimum number of false reports. As much as possible, we avoided using annotations or specifications to reduce manual labor.

Like the PREfix product,[2] we were also unsound. Our product did not verify the absence of errors but rather tried to find as many of them as possible. Unsoundness let us focus on handling the easiest cases first, scaling up as it proved useful. We could ignore code constructs that led to high rates of false-error messages (false positives) or analysis complexity, in the extreme skipping problematic code entirely (such as assembly statements, functions, or even entire files). Circa 2000, unsoundness was

CS 598 SM

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)

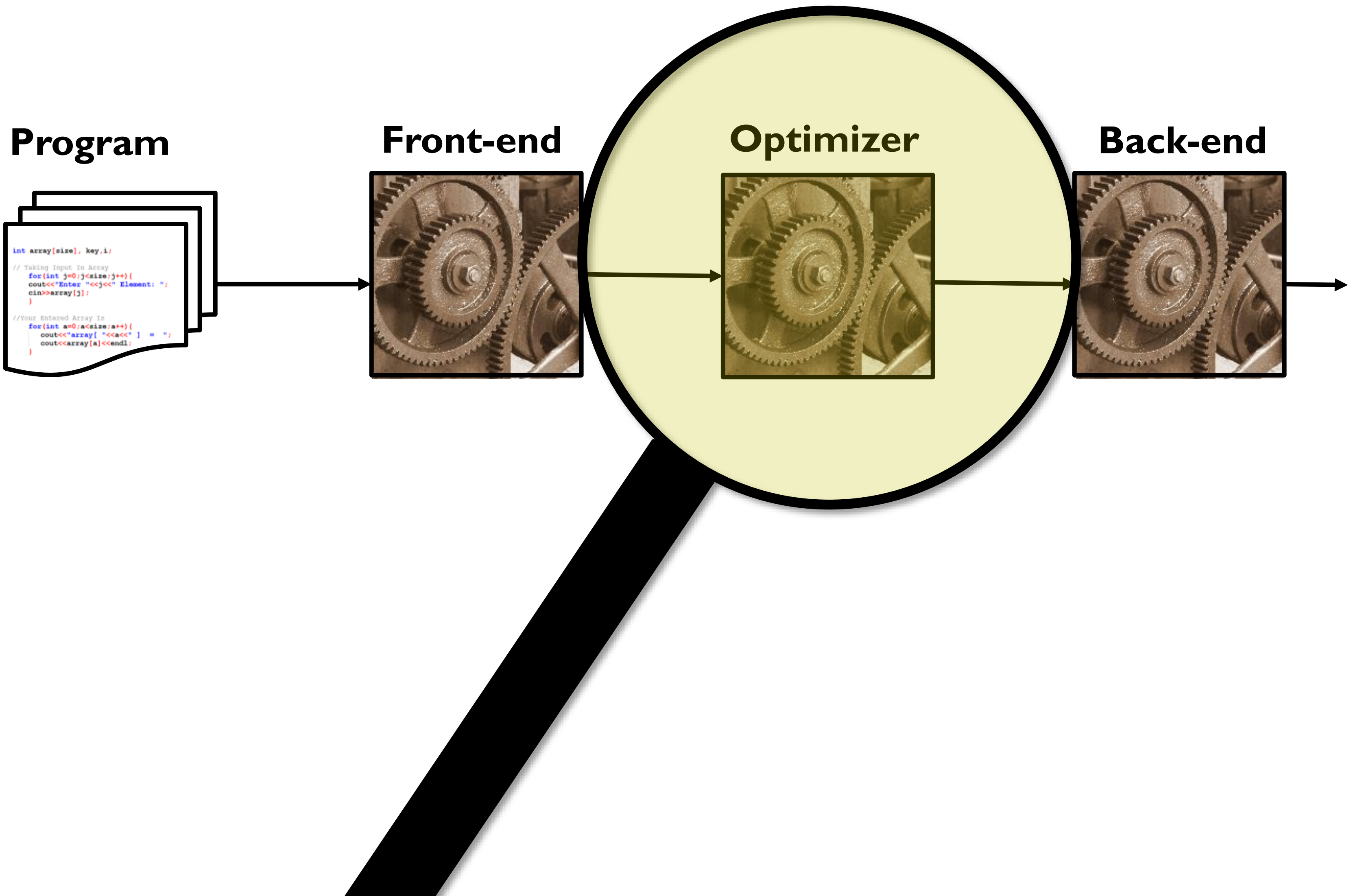
Optimizations

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

Advanced topics

- Basics of static analysis
- Checking correctness of compilers
- Compilers for Machine Lenarning

Compiler Overview



Topics We Will Not Cover

- Back-end code generation, e.g., scheduling, allocation, software pipelining (CS 426)
- Automatic vectorization, parallelization (CS 598dp)
- Compilers for Machine Learning (CS 598lce)
- New heterogeneous architectures (CS 598sa)
- Program verification (CS 476, CS 477…)
- LLVM hacking (although we have the project ☺)

CS 526 SM

COURSE LOGISTICS

Schedule

Twice a week – Tuesdays and Thursdays 11:00am-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

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)

Most important: commitment to learn as you go

Grading

Optimization Project	10%
Midterm ~~Exam~~ Quiz	20%
Final ~~Exam~~ Quiz	20%
Open-ended Project	50%

Exams

First

- Take home (March 12; before the break)
- Focuses on analysis (SSA, dataflow, dependency)
- 75 minutes (within 24 hour time)

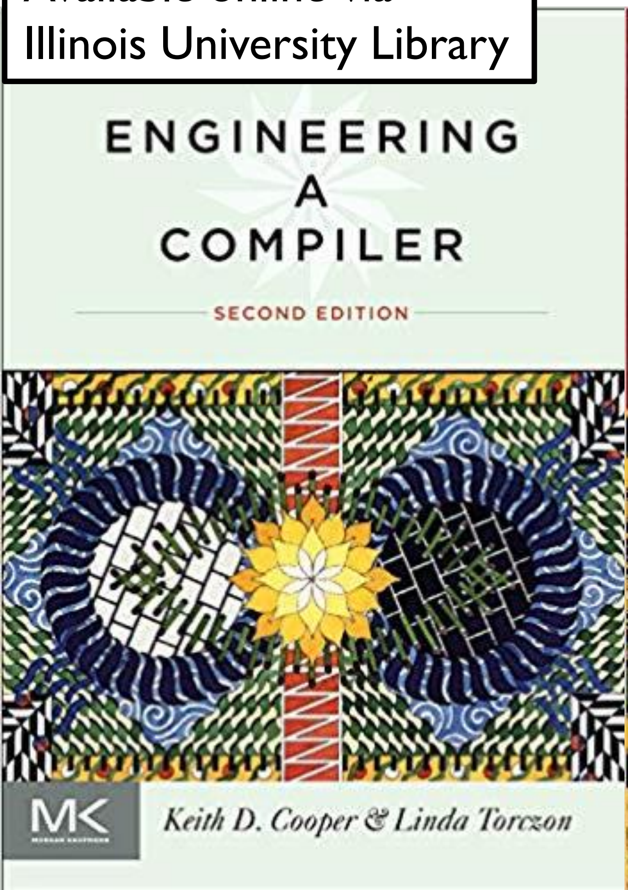
Second

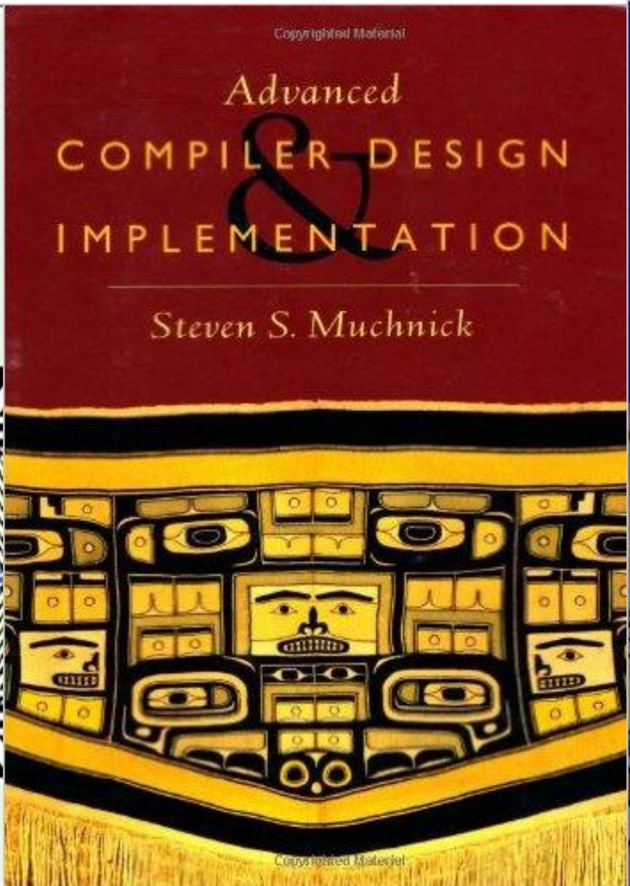
- Take home
- Pointer analysis, optimization and special topics
- Also includes the materials from the first one
- 90 minutes (within 24 hour time)

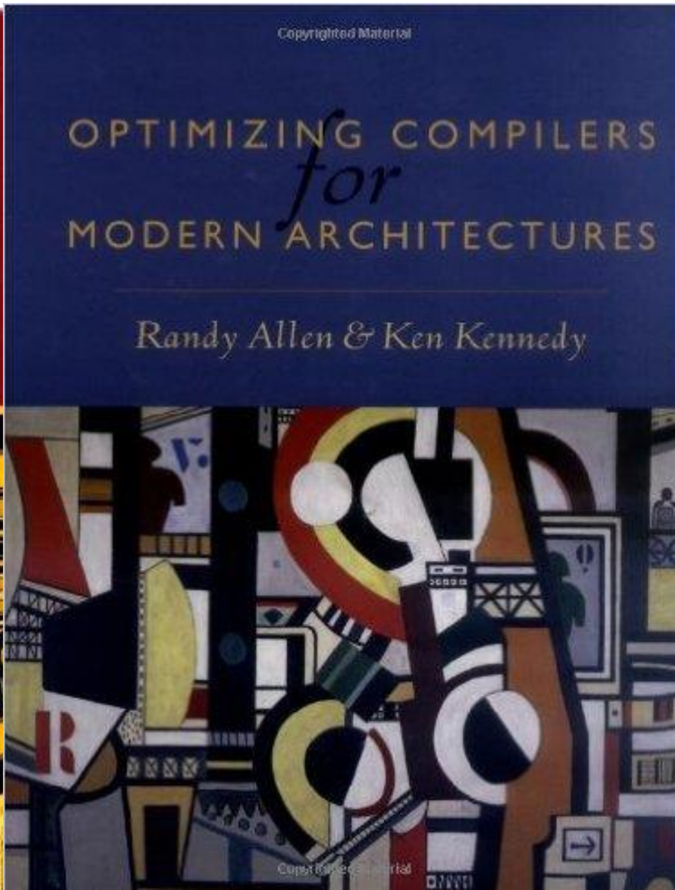
Books

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

Available online via
Illinois University Library



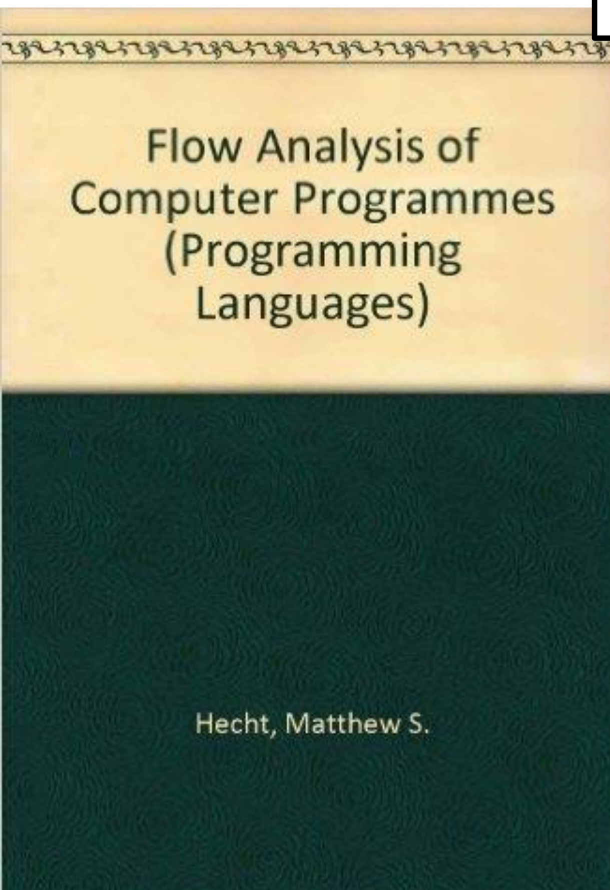


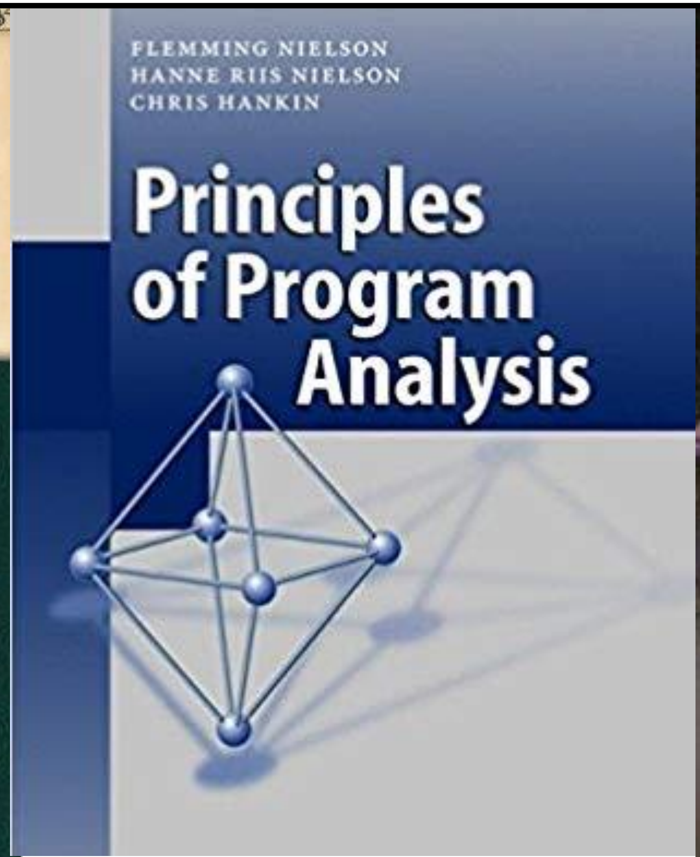


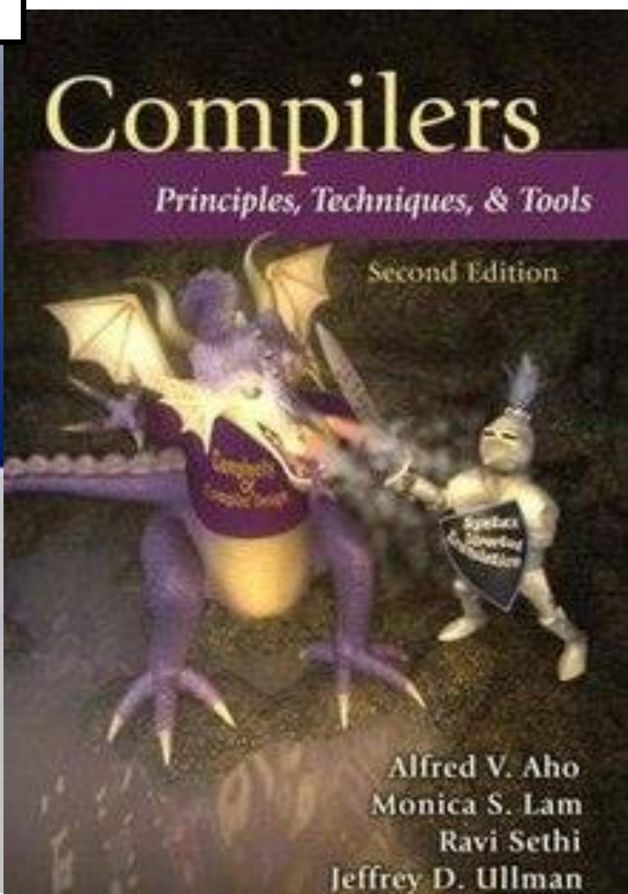
And More Books

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

Available online via Publisher







And More …

We will point our several classical papers that introduced the analysis and/or optimization techniques

To access the papers from ACM/IEE prepend the link with the following:

```
http://www.library.illinois.edu/proxy/go.php?url=
```

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,*** 2 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

- 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

Get in Touch

Email: misailo@illinois.edu

- Please include “[CS 526]” in the subject line

Office: Siebel Center, office 4110

Office Hours:

- By appointment (send me an email)
- I am typically free right after the class
- We can organize dedicated office hours before the exams

CS 526

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 ∈ N is a designated "initial" node, and there is a path from node s to every node n ∈ 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?*

Control Flow Graph (CFG)

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 $n \in N$.

Control Flow Graph (CFG) is a flow graph that represents all ***paths*** (sequences of statements) that might be traversed during program execution.

- Nodes in CFG are program statements, and edge (S_1,S_2) denotes that statement S_1 can be followed by S_2 in execution.
- In CFG, a 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 ... S_n such that execution control must reach S_1 before S_2, and, if S_1 is executed, then S_2 . . . 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)

Let us refine our previous definition

CFG is a directed graph in which:

- Each node is a single basic block
- There is an edge b1 → b2 if block b2 *may* be executed after block b1 in *some* execution

We typically define it for a single procedure

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

Example

Source Code

```
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 (ver 3.9.1)

```
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
}
```

See You Next Time!

Review in the next few weeks:

Muchnick, Chapter 21: Case Studies of Compilers

Review by next Tuesday:

Cytron, Ferrante, Rosen, Wegman, and Zadeck,

"Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,"

ACM Trans. on Programming Languages and Systems, 13(4), Oct. 1991, pp. 451–490.

If you see this, I pressed a wrong button