COMP 512
Rice University
Spring 2015

Welcome to COMP 512

COMP 512

This class is COMP 512 — "Advanced Compiler Construction"

- Subject Matter
  - ♦ Compiler-based code-improvement techniques
    - → Sometimes called "optimization"
    - → Transformations that rewrite the code
    - → Analyses needed to support transformation
  - ♦ No vector or multiprocessor parallelism
    - → See COMP 515 for that material
- Required Work
  - ♦ Mid-term exam (30%), Final Exam (30%), and Project (40%)
  - ♦ Project will be an optimizer for an **ILOC** subset (see Appendix A, EaC2e)

Notice: *Any student with a disability that requires accommodations in this class is encouraged to contact me after class or during office hours. Students may also contact Alan Russell, Rice's Director of Disability Support Services.*

Reading Materials

- We will use multiple sources for reading materials
  - ♦ Chapters 8, 9, & 10 of *Engineering a Compiler, 2nd Edition*
  - ♦ Technical papers available on the net (ACM Digital Library) or on the web site
- You will learn much more if you actually read the material
  - ♦ Part of a graduate education is learning to read technical papers and think critically about their contents
  - ♦ Reading original work is a critical part of your training that is tested in the oral exams for the Masters and Ph.D.
- Slides from lecture will be available on the course web site
  - ♦ http://www.clear.rice.edu/comp512
  - ♦ I will post them before class

You are responsible for reading the material and coming to class

COMP 512

My goals for the course (*version 1*)

- Convey a fundamental understanding of the current state-of-the-art in code optimization and code generation
- Develop a mental framework for approaching these techniques
- Differentiate between the past & the present
- Motivate current research areas (*and expose dead problems*)

Explicit non-goals

- Cover every transformation in the "catalog"
- Teach every data-flow analysis algorithm
- Cover issues related to multiprocessor parallelism

COMP 512

Rough syllabus

- Introduction to optimization — EaC2e § 8
  - ♦ Examples at different scopes
  - ♦ Principles of optimization
- Static analysis — EaC2e § 9
  - ♦ Iterative data-flow analysis
  - ♦ SSA construction
- Classic scalar optimization — EaC2e § 10 + papers
  - ♦ Best-practice techniques
  - ♦ Combining optimizations
- Analyzing and improving whole programs

Safety, opportunity, & profitability
Decision complexity
Potential sources of improvement

For next class, start reading Chapter 8 in EaC2e

COMP 512

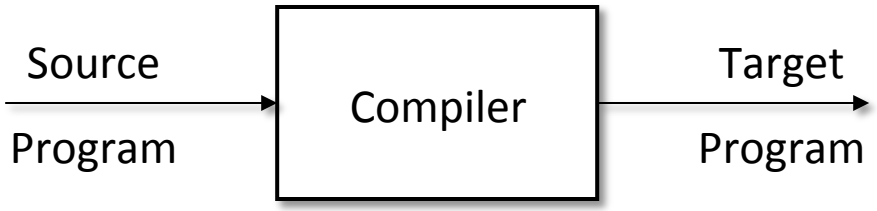
Many educated computer scientists have serious misperceptions about compiler-based code optimization

- Principles are not well elucidated
- Sources of improvement are not always clear
  - ♦ Dasgupta's unrolling example
- Practitioners can get so wrapped up in the minutia that they fail to paint the big picture
  - ♦ Interview talks in this area tend to do a mediocre job of introducing and explaining the work — *"… just a grab bag collection of tricks …"*

By the end of COMP 512, you will be literate in the field of scalar code optimization, to the point where you should be able to do independent implementation and research in the area.

I used to complain here about the Wikipedia page on compiler optimization, but it has improved markedly.

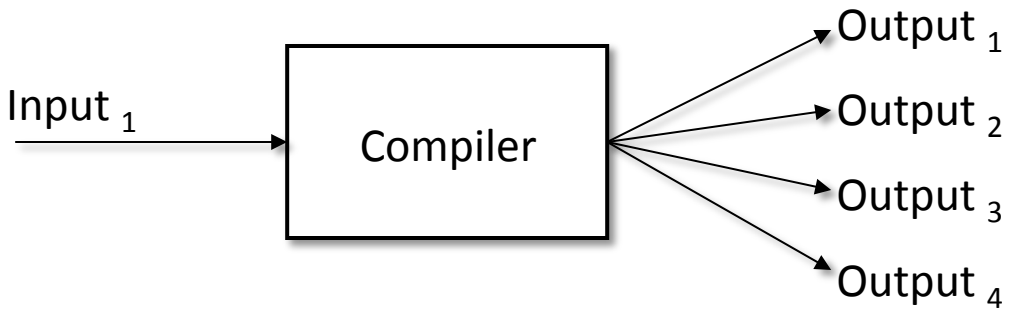
How does optimization change the program?



Optimizer tries to

1. Eliminate overhead from language abstractions
2. Map source program onto hardware efficiently
   - ♦ Hide hardware weaknesses, utilize hardware strengths
3. Equal the efficiency of a good assembly programmer

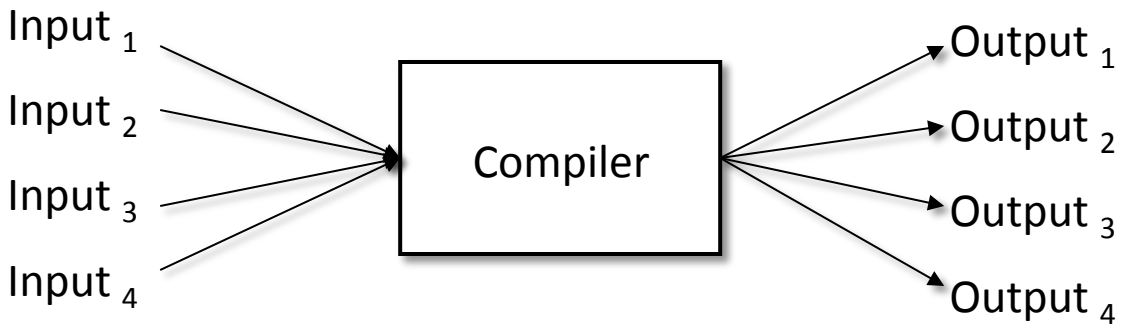
What does optimization do?



- The compiler can produce many outputs for a given input
  - The user might want the fastest code
  - The user might want the smallest code
  - The user might want the code that pages least
  - The user might want the code that …
- Optimization tries to reshape the code to better fit the user's goal

COMP 512

- Some inputs have <u>always</u> produced good code
  - ♦ First Fortran compiler focused on loops
  - ♦ PCC did well on assembly-like programs



- The compiler should provide <u>robust</u> optimization
  - ♦ Small changes in the input should not produce wild changes in the output
  - ♦ Create (& fulfill) an expectation of excellent code quality
  - ♦ Broaden the set of inputs that produce good code
- Routinely attain large fraction of peak performance (not 10%)

COMP 512

Good optimizing compilers are crafted, not assembled

- Consistent philosophy
- Careful selection of transformations
- Thorough application of those transformations
- Careful use of algorithms and data structures
- Attention to the output code

Compilers are engineered objects

- Try to minimize running time of compiled code
- Try to minimize compile time
- Try to limit use of compile-time space
- With all these constraints, results are sometimes unexpected

COMP 512

One strategy may not work for all applications

- Compiler may need to adapt its strategies to fit specific programs
  - ♦ Choice and order of optimizations
  - ♦ Parameters that control decisions & transformations
- Field of "autotuning" or "adaptive compilation"
  - ♦ Compiler writer cannot predict a single answer for all possible programs
  - ♦ Use learning, models, or search to find good strategies
- Lots of recent work in this area
  - ♦ Talk at Rice last fall by Mary Hall, Utah CS Professor
  - ♦ We'll talk about some of the problems & solutions

A Quick Look At Real Compilers

Consider inline substitution

- Replace procedure call with body of called procedure
  - ♦ Rename to handle naming issues
  - ♦ Widely used in optimizing OOPs
- How well do compilers handle inlined code?

We will talk about inlining later. For now, focus on how compilers handle inlined code.

Characteristics of inline substitution

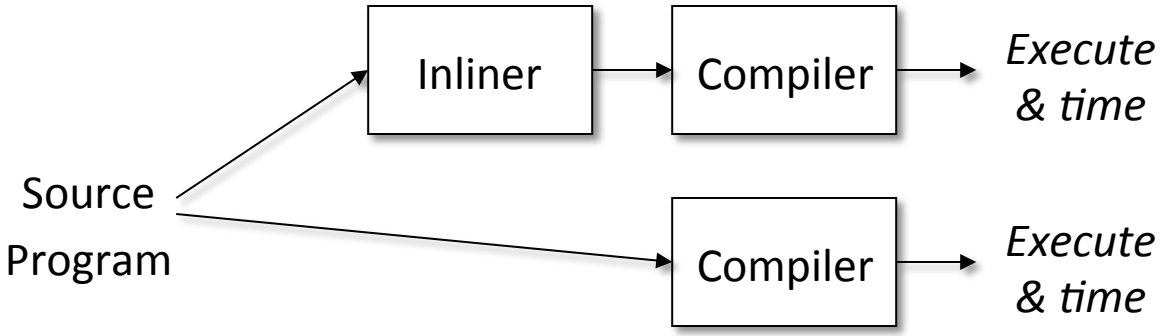
- *Safety:* almost always safe
- *Profitability:* expect improvement from avoiding the overhead of a procedure call and from specialization of the code
- *Opportunity:* inline leaf procedures, procedures called once, others where specialization seems likely

A Quick Look At Real Compilers

The Study

Five good compilers!

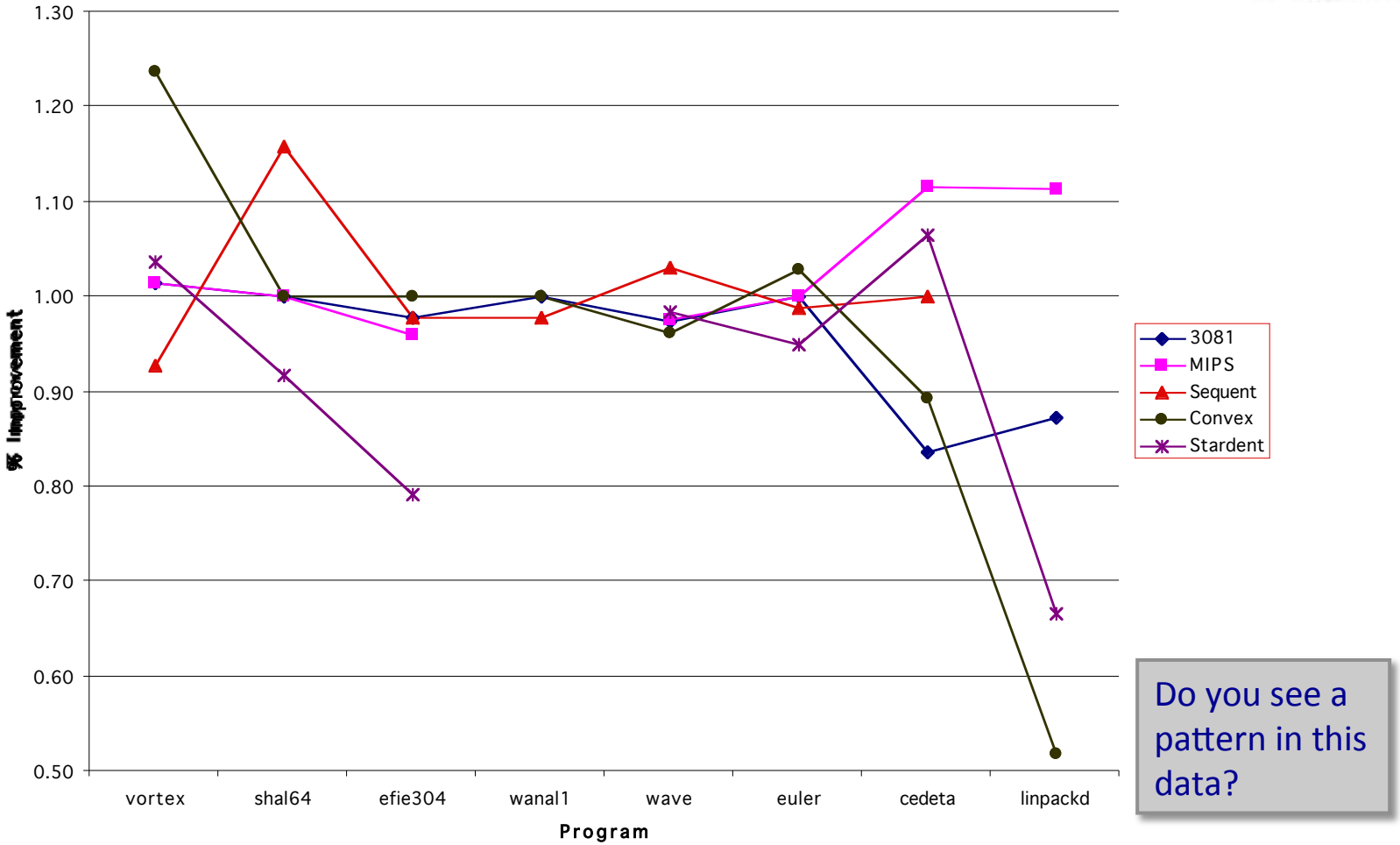
- Eight programs, five compilers, five processors
- Eliminated over 99% of dynamic calls in 5 of programs
- Measured speed of original versus transformed code



Experimental Setup

- We expected uniform speed up, at least from call overhead
- What really happened?

A Quick Look At Real Compilers



Do you see a pattern in this data?

A Quick Look At Real Compilers

And this happened with (what were then) good compilers!

What happened?

- Input code violated assumptions made by compiler writers
  - Longer procedures
  - More names
  - Different code shapes
- Exacerbated problems that are unimportant on "normal" code
  - Imprecise analysis
  - Algorithms that scale poorly
  - Tradeoffs between global and local speed
  - Limitations in the implementations

The compiler writers were surprised (most of them)

A Quick Look At Real Compilers

One standout story

- MIPS M120/5, 16 MB of memory
- Running standalone, *wanal1* took > 95 hours to compile
  - ♦ Original code, not the transformed code
  - ♦ 1,252 lines of Fortran (not a large program)
  - ♦ COMP 512 met twice during the compilation
- Running standalone with 48 MB of memory, it took < 9 minutes
- The compiler swapped for over 95 hours !?!

- For several years, *wanal1* was a popular benchmark
  - ♦ Compiler writers included it to show their compile times!

COMP 512

My Goals for COMP 512 (*version 2*)

- Theory & practice of scalar optimization
  - ♦ The underpinning for all modern compilers
  - ♦ Influences the practice of computer architecture
- Learn not only "what" but also "how" and "why"
- Provide a framework for thinking about compilation
- Class will emphasize transformations
- Analysis should be driven by needs of transformations

Register windows as an example

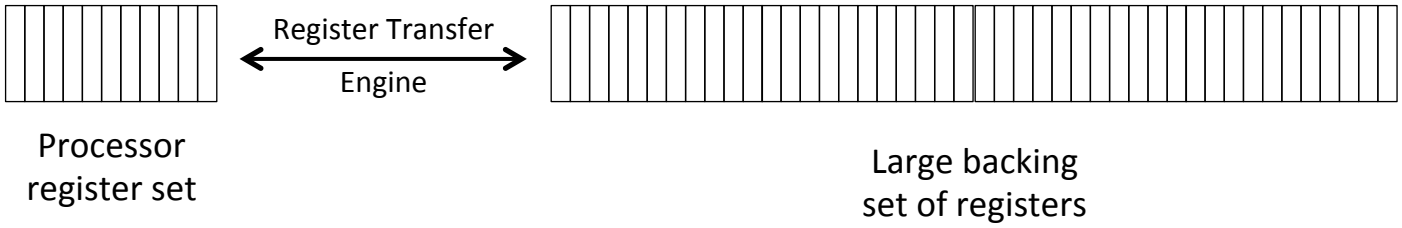
Role of the lab

- Provide experience working with multiple optimizations & their interactions
  - ♦ You will build an optimizer for an **ILOC** subset
  - ♦ Details next week

DIGRESSION ON REGISTER WINDOWS

The Idea

- Provide hardware support for the en masse register spills at call sites



- Hardware transfer engine invoked by single instruction
  - ♦ Savings in code space & execution time
- Idea died out quickly
  - ♦ Once backing register set is full, it must spill to RAM
  - ♦ Hardware solution failed to take into account program context
  - ♦ Compiler could do it as well, in general, and better in extreme cases
    - → Recursive **fibonacci** in **scheme** was orders of magnitude faster than in **C** on the early **SPARC** chips

Caller saves
Callee saves

FURTHER DIGRESSION: It is hard to make the large register set into addressable registers because of instruction-size constraints. The set of names is limited by word length.

COMP 512

My Goals for COMP 512 (*version 2*)

- Theory & practice of scalar optimization
  - ♦ The underpinning for all modern compilers
  - ♦ Influences the practice of computer architecture
- Learn not only "what" but also "how" and "why"
- Provide a framework for thinking about compilation
- Class will emphasize transformations
- Analysis should be driven by needs of transformations

Register windows as an example

Role of the lab

- Provide experience working with multiple optimizations & their interactions
  - ♦ You will build an optimizer for an **ILOC** subset
  - ♦ Details next week

Optimization

Content from § 8 EaC2e

Compilers operate at many ***granularities*** or ***scopes***

- Local techniques
  - ♦ Work on a single basic block
  - ♦ Maximal length sequence of straight-line code
- Regional techniques
  - ♦ Consider multiple blocks, but less than whole procedure
  - ♦ Single loop, loop nest, dominator region, …
- Intraprocedural (or global) techniques
  - ♦ Operate on an entire procedure (*but just one*)
  - ♦ Common unit of compilation
- Interprocedural (or whole-program) techniques
  - ♦ Operate on > 1 procedure, up to whole program
  - ♦ Logisitical issues related to accessing the code (*link time?*)

Optimization

At each of these scopes, the compiler uses different graphs

- Local techniques
  - ♦ Dependence graph (*instruction scheduling*)
- Regional Techniques
  - ♦ Control-flow graph (*natural loops*)
  - ♦ Dominator tree
- Intraprocedural (or global) techniques
  - ♦ Control-flow graph
  - ♦ Def-use chains, sparse evaluation graphs, SSA as graph
- Interprocedural (or whole-program) techniques
  - ♦ Call (multi) graph

Compiler writers must be able to perform graph traversals in their sleep – preorder, postorder, reverse postorder, depth first, …

Optimization

At each of these scopes, the compiler uses kinds of techniques

- Local techniques
  - ♦ Simple walks of the block
- Regional Techniques
  - ♦ Find a way to treat multiple blocks as a single block (*EBBs, dominators*)
  - ♦ Work with an entire loop nest
- Intraprocedural (or global) techniques
  - ♦ Data-flow analysis to determine safety and opportunity
  - ♦ Separate transformation phase to rewrite the code
- Interprocedural (or whole-program) techniques
  - ♦ Need a compilation framework where optimizer can see all the relevant code
  - ♦ Sometimes, limit to all procedures in a file
  - ♦ Sometimes, perform optimization at link time

Optimization

We want to differentiate between analysis and transformation

- Analysis reasons about the code's behavior
- Transformation rewrites the code to change its behavior

Local techniques can interleave analysis and transformation

- Property of basic block: operations execute in defined order

Over larger regions, the compiler typically must complete its analysis before it transforms the code

- Analysis must consider all possible paths, including cycles
  - ♦ Cycles typically force compiler into offline analysis
- Leads to confusion in terminology between "optimization", "analysis", and "transformation"

Optimization

Terminology

Optimization

- We will use "optimization" to refer to a broad technique or strategy, such as code motion or dead code elimination

Transformation

- We will use "transformation" to refer to algorithms & techniques that rewrite the code being compiled

Analysis

- We will use the term "analysis" to refer to algorithms & techniques that derive information about the code being compiled

This subtle distinction in usage was suggested by Vivek Sarkar.