Overview

Copyright 2019, Keith D. Cooper, Linda Torczon and Zoran Budimlić, all rights reserved.
Students enrolled in Comp 506 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Read Chapter 1 of EaC2e.
SPECIAL THANKS
Keith Cooper and Linda Torczon
Summary

COMP 506 — Compiler Construction for Graduate Students

TOPICS in the design of compilers, including scanning, parsing, optimization, and code generation. COMP 506 will focus on practical issues in compiler construction.

Instructor: Zoran Budimlić zoran@rice.edu DH 3134
Assistant: Annepha Hurlock annepha@rice.edu DH 3122
TA’s: Bumjin Im, others TBA

• Office Hours: Tue 11AM, Thu 2PM
• TA’s Office Hours: TBA
• Text: Engineering a Compiler, 2nd Edition
  ♦ Royalties for sales to Rice students go to the Torczon Family Fellowship
• Media: Class web site is http://www.clear.rice.edu/comp506
  Discussion site is on Piazza; you should receive an invitation
• Programming: All code will be written in C; it must run on CLEAR.
• Syllabus: posted on Esther, Piazza, and the web site
Logistics

COMP 506 will have two exams and three programming assignments

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Description</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mid-term exam</td>
<td>Scheduled exam, Feb. 19th, 6-9PM</td>
<td>25%</td>
</tr>
<tr>
<td>Final exam</td>
<td>Scheduled exam, during finals period</td>
<td>25%</td>
</tr>
<tr>
<td>Lab one</td>
<td>Scanner &amp; Parser</td>
<td>16%</td>
</tr>
<tr>
<td>Lab two</td>
<td>Generate ILOC\textsuperscript{1} code from Lab 1</td>
<td>17%</td>
</tr>
<tr>
<td>Lab three</td>
<td>Optimize ILOC code from Lab 2</td>
<td>17%</td>
</tr>
</tbody>
</table>

• To pass the course, you must take both exams and hand in all three labs
• Brief questionnaire on each lab, instead of a lab report

Notice: Any student who needs accommodations for a disability in COMP 506 should contact the instructor or contact Alan Russell, Rice’s Director of Disability Support Services. Alan’s office is on the 1\textsuperscript{st} floor of Allen Center.

\textsuperscript{1} See Appendix A in EaC2e
Logistics

What will we cover in COMP 506?

• A quick tour of scanning, parsing, and IR generation
  ♦ I assume that you have an undergraduate degree with some exposure to Computer Science. This section will focus on practice rather than theory. If you don’t know the theory of DFA-based scanners and PDA-based parsers, read chapters 2 & 3 in the book
  ♦ Learn by doing it, rather than by listening to me talk about it

• Scalar code optimization
  ♦ Focus on best-practice algorithms for classical optimizations
  ♦ An abbreviated, practical tour of material from COMP 512
  ♦ Will supplement the book with technical papers for some techniques

• Back-end compiler algorithms
  ♦ Instruction selection
  ♦ Instruction scheduling
  ♦ Register allocation

Projects:
• Scanner / Parser for a toy language
• Simple code generation
• Simple scalar optimization
Logistics

Suggestions for your class-taking techniques

• I will use projected material extensively
  ♦ Questions are welcome; interrupt the lecture
  ♦ PowerPoint materials should be online before class
  ♦ See (also) the COMP 412 lecture notes (http://www.clear.rice.edu/comp412)

• Read the book
  ♦ Not all material will be covered in class
  ♦ Think through the section review questions

• Come to class
  ♦ The simplest advice I can give
  ♦ Students attend a tier-1 research university for the lectures, not the PowerPoints

• Start on the programming assignments promptly
  ♦ Procrastination kills more grades than lack of knowledge or skill
Logistics

We have set up a course discussion site on Piazza

• Piazza is the official forum for asking questions and receiving advice and clarifications
  ♦ You should have received an email inviting you to join.
  ♦ If you have not received an invite by next class, let us know.

• Please post questions and post answers to the Piazza group
  ♦ We will monitor the forum
  ♦ You can post anonymously to the forum
  ♦ You can post (wiki-like) a communal answer to any question
  ♦ We can post an official answer to any question
  ♦ This is a community forum, not a “ask the teacher, get an answer” forum

• This forum only helps those who use it.
Logistics

• All programming in COMP 506 will be done in C
• All code that you submit must build and run correctly on CLEAR

These two statements have several consequences
• Make sure that you have an account on CLEAR and that you know how to access it
• Make sure that you know how to use the C tools on CLEAR
  ♦ CLEAR supports both gcc and clang
  ♦ Your labs will include makefile to build the code for the graders
• If you intend to develop code on your own system, make sure that the versions of the various tools on your system are compatible with those on CLEAR
  ♦ We will build ("make"), execute, and evaluate your code on CLEAR
  ♦ “It worked on my laptop” is not an acceptable excuse

1 make is an antiquated but useful technology.
Honor Policy

COMP 506 is subject to the rules of Rice’s Honor Code

• Exams are scheduled, closed-book, closed-notes tests.
  ♦ You cannot use electronic devices during the exam. Period.
  ♦ If you cannot take an exam at the scheduled time, talk with me before that time.

• Projects are also covered by the honor code.
  ♦ You may discuss algorithms, techniques, and problems with other students.
  ♦ You may not look at another student’s code. You may not write code that is used in another student’s project.
  ♦ You are expected to write all of the code for your project yourself. Period.
  ♦ We will likely use sophisticated automatic tools to detect plagiarism. I don’t want to go there. You don’t want to go there. Trust me.
This Course Covers Compiler Construction

• What is a **compiler**?
  ♦ A program that translates an *executable* program in one language into an *executable* program, usually in another language
  ♦ The compiler should improve the program, *in some way*

• What is an **interpreter**?
  ♦ A program that reads an *executable* program and produces the results of executing that program

• C and C++ are typically compiled

• Python & Scheme are typically interpreted

• Java is complicated
  ♦ compiled to bytecode (*code for the Java VM*)
  ♦ which are then interpreted
  ♦ or a hybrid strategy is used
    → Just-in-time compilation

**Common misstatement:**  *x* is an interpreted language, or *x* is a compiled language
How does a compiler work?

- Front end analyzes the input program in the source language & builds some internal representation for the program ("IR")
- Optimizer analyzes & rewrites the IR to improve the final code
  ♦ The connection between the IR and the final code may be subtle
- Back end translates the IR into the target language
  ♦ Target language is usually the instruction set of some target processor
Compiler Structure

How does this structure affect the compiler writer?

- Separate passes make the compiler easier to debug
- Three “languages” for compiler writer to master (source, IR, and target)
  - The IR may be very different than either the source or target languages
- The front end & optimizer “shape” the code for the back end
  - Different than shaping it for the target language
How does this structure relate to the syllabus?

• Lecture will correspond (roughly) to a linear walk through the chapters

• Roughly speaking:
  – First 1/3 of course on chapters 1 through 7 (Labs 1 and 2)
  – Second 1/3 of course on chapters 8, 9, and 10 (Lab 3)
  – Last third of the course on chapters 11, 12, and 13

• The undergraduate course, comp 412, covers the front & back ends
  – COMP 506 will spend at least 1/3 of the semester on the optimizer
Compiler Structure

**Front End**
- **Scanner**: Translate stream of characters into stream of classified words.
- **Parser**: Determine if stream of words is a sentence in the source language.
- **Semantic Elaboration**: Check deeper meaning and build appropriate IR(s).

**Intermediate Representation (IR)**
- **Optimizer**: Rewrite the IR form of the code to “improve” it in some way.

**Back End**
- **Instruction Selector**: Rewrite the IR into target language operations.
- **Register Allocator**: Rewrite the code to fit into the finite register set of the target.
- **Instruction Scheduler**: Reorder the operations so that they run faster.

**Program in Source Code** → **Scanner → Parser → Semantic Elaboration → IR** → **Pass 1, Pass 2, …, Pass n** → **Optimizer** → **Selector, Allocator, Scheduler** → **Program in Target Code**
Why study compilers?

- Compilers are *interesting*
  - Large complicated software systems that must efficiently tackle hard algorithmic problems — approximate solutions to NP complete problems
  - Application of theory to practice

- Compilers are *fundamental*
  - Primary responsibility for application performance
    - Performance becomes more difficult as processors become more complex
  - The alternative (assembly language) is much less attractive

- Compilers (& interpreters) are *everywhere*
  - Many applications have embedded languages
    - XML, HTML, macros, commands, Visual Basic in Excel, ...
  - Many applications have input formats that look like languages
The Big Question

Why study compilers?
In other COMP courses, you are taught to use a variety of abstractions, ranging from object orientation to hash maps to closures to ...
• Each of these abstractions has a price
• You need to understand that price before you implement
  ♦ Abstraction is critical to successful construction of interesting programs, but you must understand the costs and make intelligent decisions about when to replace an abstraction with a more efficient & concrete implementation
  ♦ Careful choice of abstractions & where to use them can be the difference between a fast system & a slow (or infeasible) one
• Examples:
  ♦ Use of virtual function calls in performance-critical kernels
  ♦ Use of scripting languages, such as PHP, for back-end server applications
  ♦ Use of hash maps over enumerated types as array indices in lab 2
Simple Example

Which loop is fastest?

```c
for (x=0; x<n; x++)
    for (y=0; y<n; y++)
        A[x][y] = 0;

for (y=0; y<n; y++)
    for (x=0; x<n; x++)
        A[x][y] = 0;

p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
    *p++ = 0;
```
Simple Example

Which loop is fastest?

```
for (x=0; x<n; x++)
    for (y=0; y<n; y++)
        A[x][y] = 0;
```

```
for (y=0; y<n; y++)
    for (x=0; x<n; x++)
        A[x][y] = 0;
```

```
p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
    *p++ = 0;
```

All three loops have distinct performance

- 0.51 seconds on 10,000 x 10,000 array
- 1.65 seconds on 10,000 x 10,000 array
- 0.11 seconds on 10,000 x 10,000 array

All data collected on a quiescent, multiuser Intel T9600 @ 2.8 GHz using code compiled with gcc 4.1 –O3
Simple Example

Which loop is fastest?

```
for (x=0; x<n; x++)
    for (y=0; y<n; y++)
        A[x][y] = 0;

for (y=0; y<n; y++)
    for (x=0; x<n; x++)
        A[x][y] = 0;

p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
    *p++ = 0;
```

Conventional wisdom suggests using

```
bzero((void *) &A[0][0],(size_t) n * n * sizeof(int));
```

All three loops have distinct performance

- 0.51 seconds on 10,000 x 10,000 array
- 1.65 seconds on 10,000 x 10,000 array
- 0.11 seconds on 10,000 x 10,000 array

All data collected on a quiescent, multiuser Intel T9600 @ 2.8 GHz using code compiled with gcc 4.1 -O3

COMP 506, Spring 2019
Simple Example

Which loop is fastest?

```plaintext
for (x=0; x<n; x++)
    for (y=0; y<n; y++)
        A[x][y] = 0;

for (y=0; y<n; y++)
    for (x=0; x<n; x++)
        A[x][y] = 0;

p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
    *p++ = 0;
```

Conventional wisdom suggests using

```c
bzero((void *) &A[0][0],(size_t) n * n * sizeof(int));
```

All three loops have distinct performance:

- 0.51 seconds on 10,000 x 10,000 array
- 1.65 seconds on 10,000 x 10,000 array
- 0.11 seconds on 10,000 x 10,000 array

A good compiler should know these tradeoffs on each target and generate the best code. Few real compilers do.

All data collected on a quiescent, multiuser Intel T9600 @ 2.8 GHz using code compiled with gcc 4.1 –O3

COMP 506, Spring 2019
Simple Example

Which loop is fastest?

for (x=0; x<n; x++)
  for (y=0; y<n; y++)
    A[x][y] = 0;

for (y=0; y<n; y++)
  for (x=0; x<n; x++)
    A[x][y] = 0;

p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
  *p++ = 0;

Another point

• Why did the code in the third example compute \( t \) separately?
• Why not fold that multiply into the test that ends the loop?
• The compiler should move the computation of \( n^2 \) out of the loop

What does it take to move \( n \times n \)?

1. Compiler must recognize that \( n \) does not change in the loop
   ♦ Implies \( n \) is an unambiguous scalar reference

2. Must be to a location where \( n \times n \) can safely execute
   ♦ Implies that \( n \geq 0 \) is known at compile time
Simple Example

Which loop is fastest?

```
for (x=0; x<n; x++)
    for (y=0; y<n; y++)
        A[x][y] = 0;
```

```
for (y=0; y<n; y++)
    for (x=0; x<n; x++)
        A[x][y] = 0;
```

```
p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
    *p++ = 0;
```

Understanding performance will make you a better implementer

• The choices in this simple example require that you understand
  ♦ How the code is translated
  ♦ How the code is optimized
  ♦ How the cache & TLB work

• An explicit goal for COMP 506 is to help you develop this level of understanding of computers & their software systems.

If you are going to implement the next Facebook, or Netflix, or Google, I want you to know this stuff and use it.
Simple Example

Which loop is fastest?

```c
for (x=0; x<n; x++)
    for (y=0; y<n; y++)
        A[x][y] = 0;

for (y=0; y<n; y++)
    for (x=0; x<n; x++)
        A[x][y] = 0;

p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
    *p++ = 0;
```

One more point

• The code in the third loop nest replaces array addressing with a simple pointer dereference and a pointer auto-increment

• Is that fair? Can a compiler recognize that situation and rewrite the code this way?

1. The optimizer flattened the two nested loops into one single loop.
   ♦ Done in Fortran I compiler (1959)

2. The optimizer replaced `A[x][y]` with a pointer auto-increment
   ♦ Well understood by 1981. See [19].
   ♦ See “Operator Strength Reduction” in Chapter 10 or in [107]
The Big Question

Why study compilers?

In many applications, performance matters. Students (and many software engineers) often lack a clear understanding of how to approach performance problems.

One useful strategy to improve application performance

• Design at the appropriate level of abstraction
• If performance is an issue
  ♦ Measure where the application spends time
  ♦ In those places, replace the abstract implementation with a semantically equivalent implementation that is faster and more concrete
• Repeat until you are happy with the results

This strategy only works if implementers understand the costs of abstractions and what compilers can (and cannot) reasonably do to improve those costs
Why Does This Matter Today?

The era of clock-speed improvements is over

- $n^2$ impact of clock speed on energy consumption
- Smaller wires introduce higher resistances
- We cannot cool high-clock rate chips made at small feature geometries

However, we are wasting another massive factor in performance due to the way that we build systems

- Leiserson has estimated the waste at a factor of $> 30,000 \times$
- That is equivalent to 14 or 15 generations of Moore’s law improvements
- To get that performance back, we need better language choices (not Python), better implementations of those languages (good optimizing compilers), and programmers who understand the performance cost of the abstractions that they use (all of you)
Price of Abstraction

In COMP 412, students can work in any programming language, except PERL.

- We have performance data, for the last 6 years, on the best labs, by language.
- The data shows a marked difference, by language.

![Graph showing performance data by language](COMP 412, Lab 1 Data)

**Watch the time scale change**
Price of Abstraction

In COMP 412, students can work in any programming language, except PERL.

- We have performance data, for the last 6 years, on the best labs, by language.
- The data shows a marked difference, by language.

Watch the time scale change.

70x difference: Python to C.

COMP 412, Lab 1 Data.
Price of Abstraction

In COMP 412, students can work in any programming language, except PERL.

- We have performance data, for the last 6 years, on the best labs, by language.
- The data shows a marked difference, by language.

Watch the time scale change
Next class, we will dive into techniques for building a scanner using a classic, LEX-style scanner generator.

- Read Chapter 1
- Read Chapter 2, Sections 2.1 to 2.3 (2.4 if you have time)
- Make sure that you have a working account on CLEAR