



COMP 412
FALL 2009

COMP 412
Overview of the Course

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 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.

Critical Facts



COMP 412 — *Introduction to Compiler Construction*

Topics in the design of programming language translators, including parsing, run-time storage management, error recovery, code generation, and optimization

- Instructor: Keith Cooper, keith@rice.edu, x6013
- Office Hours: 1 PM, Monday, DH 2065
- Text: *Engineering a Compiler* (Cooper and Torczon)
 - Published by Morgan-Kaufmann, an Elsevier imprint
 - Royalties for sales to COMP 412 go to the Torczon Fellowship at Rice, which awards fellowships to Rice CS undergrads
- Web Site: <http://owlnet.rice.edu/~comp412>
 - Lab handouts, homework, slides, practice exams, ...
 - I will not have handouts in class; get them from the web



Basis for Grading

- Exams
 - Midterm 20%
 - Final 25%
- Homework 5%
- Projects
 - Register allocator 15%
 - Parser generator 20%
 - Instruction scheduler 15%

This stuff will total 50% of the course credit. If I assign more homework than in the past, I may shift the percentages by up to 5%

Notice: Any student with a disability requiring accommodations in this class is encouraged to contact me after class or during office hours, and to contact Rice's Coordinator for Disabled Student Services.

Basis for Grading



<ul style="list-style-type: none">• Exams<ul style="list-style-type: none">– Midterm– Final	<ul style="list-style-type: none">◆ Closed-notes, closed-book take home◆ Old exam on web site as an example◆ Questions from low attendance days
<ul style="list-style-type: none">• Homework	<ul style="list-style-type: none">◆ Reinforce concepts, provide practice◆ Number of assignments <i>t.b.d.</i>
<ul style="list-style-type: none">• Projects<ul style="list-style-type: none">– Register allocator– Parser generator– Instruction scheduler	<ul style="list-style-type: none">◆ We don't build a traditional compiler◆ High ratio of thought to programming◆ Activities you might encounter on job◆ Choose your own language (<i>not PERL</i>)

Rough Syllabus

Friday, Lab 1



- Overview § 1
- Local Register Allocation § 13.2
- Scanning § 2
- Parsing § 3 *new*
- Context Sensitive Analysis § 4
- Inner Workings of Compiled Code § 6, 7
- Introduction to Optimization § 8
- Code Selection § 11
- Instruction Scheduling § 12
- Register Allocation § 13
- More Optimization (*time permitting*)

If it looks like the course follows the text, that's because the text was written from the course.

What about the missing chapters?

5 : We'll fit it in
9, 10: see CS 512

Lab Schedule is on the web site



Class-taking Technique for COMP 412

- I will use projected material extensively
 - I welcome questions
 - PowerPoint materials online before class
- Read the book
 - Not all material will be covered in class
- Come to class
 - The tests will cover both lecture and reading
 - I take test questions from low-attendance classes
- Do the programming assignments
 - COMP 412 is not a programming course
 - Projects are graded on functionality, documentation, and lab reports, not style (*results matter*)
 - Correctness is assumed *
- Do the homework
 - Good practice for the tests

This aspect of CS 412 bothers some students. In lower-level programming courses, we give generous partial credit on projects. In 412, we demand correctness. You would never buy a compiler that did not produce correct code. Thus, you don't get partial credit for a lab that is written but doesn't compile or that produces incorrect code for most examples.

Welcome to the real world.



About the Book

- Textbook: "Engineering a Compiler"
 - By Keith D. Cooper and Linda M. Torczon
 - If you find something that looks like a typo, it may well be one
 - Second printing is better than first printing
 - Second edition will be much cleaner (we hope)
- Book presents modern material
 - Considers problems of post-1986 computers
 - Addresses modern techniques
 - Discards lots of less relevant material
- Other textbooks on reserve in Fondren Library
 - Consult them for alternate views

We will use some material from EaC 2e.
I will provide hardcopies.

Compilers



- What is a **compiler**?
 - A program that translates an *executable* program in one language into an *executable* program 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 is typically compiled, Scheme is typically interpreted
- Java is compiled to bytecodes (code for the Java VM)
 - which are then interpreted
 - Or a hybrid strategy is used
 - Just-in-time compilation

Common mis-statement:
X is an interpreted language
(or a compiled language)



Why Study Compilation?

- Compilers are **important**
 - Responsible for many aspects of system performance
 - Attaining performance has become more difficult over time
 - In 1980, typical code got 85% or more of peak performance
 - Today, that number is closer to 5 to 10% of peak
 - Compiler has become a prime determiner of performance
- Compilers are **interesting**
 - Compilers include many applications of theory to practice
 - Writing a compiler exposes algorithmic & engineering issues
- Compilers are **everywhere**
 - Many practical applications have embedded languages
 - Commands, macros, formatting tags ...
 - Many applications have input formats that look like languages



Reducing the Price of Abstraction

Computer Science is the art of creating virtual objects and making them useful.

- We invent abstractions and uses for them
- We invent ways to make them efficient
- Programming is the way we realize these inventions

Well written compilers make abstraction affordable

- Cost of executing code should reflect the underlying work rather than the way the programmer chose to write it
- Change in expression should bring small performance change
- Cannot expect compiler to devise better algorithms
 - Don't expect bubblesort to become quicksort

Making Languages Usable



It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus on the subject of the 1st FORTRAN compiler



Simple Examples

Which is faster?

```
for (i=0; i<n; i++)  
    for (j=0; j<n; j++)  
        A[i][j] = 0;
```

All three loops have distinct performance.

0.51 sec on 10,000 x 10,000 array

```
for (i=0; i<n; i++)  
    for (j=0; j<n; j++)  
        A[j][i] = 0;
```

1.65 sec on 10,000 x 10,000 array

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

0.11 sec 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.

Conventional wisdom suggests using

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

0.52 sec on 10,000 x 10,000 array



Simple Examples

Abstraction has its price (& it is often higher than expected)

- In the 1980's we built a system called the Rⁿ Programming Environment
 - Bitmap displays and mice were new & poorly supported
 - SUN Workstation (& others) had no window systems
 - Predated the Mac, Windows, and so on.
- We built our own window system
 - It had to represent rectangles on the screen
 - Rectangle was a pair of points
 - Mouse tracking was difficult (10 MHz Motorola 68010)
 - Each mouse movement generated an interrupt
 - At each movement, had to repaint old mouse location, save new mouse location, xor mouse into location, and paint result to screen
 - We hit performance problems due to the point abstraction



Simple Examples

This code shows the point abstraction in use

```
struct point {      /* Point on the plane of windows */
    int x; int y;
}

void Padd(struct point p, struct point q, struct point * r)
{
    r->x = p.x + q.x;
    r->y = p.y + q.y;
}

int main( int argc, char *argv[] )
{
    struct point p1, p2, p3;

    p1.x = 1; p1.y = 1;
    p2.x = 2; p2.y = 2;

    Padd(p1, p2, &p3);

    printf("Result is <%d,%d>.\n", p3.x, p3.y);
}
```

Simple Example (point add)

gcc 4.1, -S option



`_main:` (some boilerplate code ellided for brevity's sake)
`L5:`

```
popl    %ebx
movl    $1, -16(%ebp)
movl    $1, -12(%ebp)
movl    $2, -24(%ebp)
movl    $2, -20(%ebp)
leal    -32(%ebp), %eax
movl    %eax, 16(%esp)
movl    -24(%ebp), %eax
movl    -20(%ebp), %edx
movl    %eax, 8(%esp)
movl    %edx, 12(%esp)
movl    -16(%ebp), %eax
movl    -12(%ebp), %edx
movl    %eax, (%esp)
movl    %edx, 4(%esp)
call    _PAdd
movl    -28(%ebp), %eax
movl    -32(%ebp), %edx
movl    %eax, 8(%esp)
movl    %edx, 4(%esp)
leal    LC0-"L00000000001$pb"(%ebx), %eax
movl    %eax, (%esp)
call    L_printf$stub
addl    $68, %esp
popl    %ebx
leave
ret
```

Code for Intel Core 2 Duo

Assignments to p1 and p2

Setup for call to PAdd

Setup for call to printf

Address calculation for format string in printf call

Simple Example (point add)

gcc 4.1, -S option



_PAdd:

```
pushl   %ebp
movl    %esp, %ebp
subl    $8, %esp
movl    8(%ebp), %edx
movl    16(%ebp), %eax
addl    %eax, %edx
movl    24(%ebp), %eax
movl    %edx, (%eax)
movl    12(%ebp), %edx
movl    20(%ebp), %eax
addl    %eax, %edx
movl    24(%ebp), %eax
movl    %edx, 4(%eax)
leave
ret
```

Code for PAdd

Actual work

The code does a lot of work to execute two add instructions.

→ Factor of 10 in overhead

→ And a window system does a lot of point adds

Code optimization (careful compile-time reasoning & transformation)
can make matters better.

Simple Example (point add)

gcc 4.1, -S -O3 option



`_main:` (some boilerplate code ellided for brevity's sake)
`L5:`

```
    popl    %ebx
    subl    $20, %esp
    movl    $3, 8(%esp)
    movl    $3, 4(%esp)
    leal    LC0-"L00000000001$pb"(%ebx), %eax
    movl    %eax, (%esp)
    call    L_printf$stub
    addl    $20, %esp
    popl    %ebx
    leave
    ret
```

It inlined PAdd and folded the known constant values of p1 and p2.

With the right information, a good compiler can work wonders.
→ It kept the body of PAdd because it could not tell if it was dead

What if it could not discern the values of p1 and p2?

Simple Example (point add)

gcc 4.1, -S -O3 option



`_main:` (some boilerplate code ellided for brevity's sake)
`L5:`

```
    popl    %ebx
    subl    $20, %esp
    movl    _one-"L00000000001$pb"(%ebx), %eax
    addl    _two-"L00000000001$pb"(%ebx), %eax
    movl    %eax, 8(%esp)
    movl    %eax, 4(%esp)
    leal    LC0-"L00000000001$pb"(%ebx), %eax
    movl    %eax, (%esp)
    call    L_printf$stub
    addl    $20, %esp
    popl    %ebx
    leave
    ret
```

I put 1 and 2 in global variables named "one" and "two".

The optimizer inlined PAdd

The optimizer recognized that
 $p1.x = p1.y$ and $p2.x = p2.y$
so

$p1.x + p2.x = p1.y + p2.y.$

If I make PAdd static, it
deletes the code for PAdd

This code shows the more general version. It inlined PAdd and subjected the arguments to local optimization. It still had to perform the adds, but it recognized that the second one was redundant.

→ Gcc did a good job on this example.



Simple Example (point add)

I lied to you

- The R^n point abstraction used short ints rather than ints
 - Added conversion from short to int at call & back in PAdd
 - Added conversion on return value in both PAdd and caller
- A compiler cannot, in general, change the data types of programmer-specified variables
 - Original code compiled where short \cong int
 - Moved to machine where they differed
 - Had to rewrite the code before it could track the mouse
- Morals of the story
 - A good compiler can overcome many problems
 - A good compiler cannot, in general, overcome bad design
 - Sometimes, you need to understand the compiler to debug code

Z80

Replaced most PAdd and PSub calls with inline macros

Intrinsic Merit



- Compiler construction poses challenging and interesting problems:
 - Compilers must process large inputs, perform complex algorithms, but also **run quickly**
 - Compilers have primary responsibility for **run-time performance**
 - Compilers are responsible for making it acceptable to use the **full power** of the programming language
 - Computer architects perpetually create new challenges for the compiler by building more **complex machines**
 - Compilers must hide that complexity from the programmer
- A successful compiler requires mastery of the many complex interactions between its constituent parts

Intrinsic Interest



- Compiler construction involves ideas from many different parts of computer science

<i>Artificial intelligence</i>	Greedy algorithms Heuristic search techniques
<i>Algorithms</i>	Graph algorithms, union-find Dynamic programming
<i>Theory</i>	DFAs & PDAs, pattern matching Fixed-point algorithms
<i>Systems</i>	Allocation & naming, Synchronization, locality
<i>Architecture</i>	Pipeline & hierarchy management Instruction set use



Why Does This Matter Today?

In the last 3 years, most processors have gone multicore

- The era of clock-speed improvements is drawing to an end
 - Faster clock speeds mean higher power (n^2 effect)
 - Smaller wires mean higher resistance for on-chip wires
- For the near term, performance improvement will come from placing multiple copies of the processor (*core*) on a single die
 - Classic programs, written in old languages, are not well suited to capitalize on this kind of multiprocessor parallelism
 - Parallel languages, some kinds of OO systems, functional languages
 - Parallel programs require sophisticated compilers
- Think of the Intel/AMD bet on multicore as a full-employment act for well-trained compiler writers

Revenge of
COMP 210



About the Instructor

My own research program

- Compiling for advanced microprocessor systems
 - Optimization for *space, power, & speed*
- Relationship between compiler structure & effectiveness
- Nitty-gritty things that happen in compiler back ends
- Whole program analysis & optimization

Thus, my interests lie in

- Quality of generated code
- Interplay between compiler and architecture
- Static analysis to discern program behavior
- Run-time performance analysis

Major focus of
the PACE Project



Next class

- The view from 35,000 feet
 - How a compiler works
 - What is important
 - What is hard and what is easy

- Things to do
 - Read Chapter 1
 - Make sure you have a working OwlNet account
 - Find the web site

<http://www.owlnet.rice.edu/~comp412>



Extra Slides Start Here



Taking a Broader View

- Compiler Technology = Off-Line Processing
 - **Goals:** improved performance and language usability
 - Making it practical to use the full power of the language
 - **Trade-off:** preprocessing time versus execution time (or space)
 - **Rule:** performance of both compiler and application must be acceptable to the end user
- Examples
 - Macro expansion
 - PL/I macro facility – 10x improvement with compilation
 - Database query optimization
 - Emulation acceleration
 - TransMeta “code morphing”