Introducing Computer Systems from a Programmer’s Perspective

Randal E. Bryant, David R. O’Hallaron
Computer Science and Electrical Engineering
Carnegie Mellon University
Outline

Introduction to Computer Systems
- Course taught at CMU since Fall, 1998
- Some ideas on labs, motivations, ...

Computer Systems: A Programmer’s Perspective
- Our textbook, now in its second edition
- Ways to use the book in different courses
Background

1995-1997: REB/DROH teaching computer architecture course at CMU.

- Good material, dedicated teachers, but students hate it
- Don’t see how it will affect their lives as programmers

Course Evaluations

CS Average

REB: Computer Architecture
Computer Arithmetic
Builder’s Perspective

- How to design high performance arithmetic circuits
Numbers are represented using a finite word size
Operations can overflow when values too large
  • But behavior still has clear, mathematical properties

void show_squares()
{
  int x;
  for (x = 5; x <= 5000000; x*=10)
    printf("x = %d x^2 = %d\n", x, x*x);
}
Memory System
Builder’s Perspective

Builder’s Perspective

- Must make many difficult design decisions
- Complex tradeoffs and interactions between components
Hierarchical memory organization

Performance depends on access patterns
- Including how step through multi-dimensional array
The Memory Mountain

Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
Background (Cont.)

1997: OS instructors complain about lack of preparation

- Students don’t know machine-level programming well enough
  - What does it mean to store the processor state on the run-time stack?
- Our architecture course was not part of prerequisite stream
Birth of ICS

1997: REB/DROH pursue new idea:
- Introduce them to computer systems from a programmer's perspective rather than a system designer's perspective.
- Topic Filter: What parts of a computer system affect the correctness, performance, and utility of my C programs?

1998: Replace architecture course with new course:
- 15-213: Introduction to Computer Systems

Curriculum Changes
- Sophomore level course
- Eliminated digital design & architecture as required courses for CS majors
15-213: Intro to Computer Systems

Goals
- Teach students to be sophisticated application programmers
- Prepare students for upper-level systems courses

Taught every semester to 400+ students
- All CS undergrads (core course)
- All ECE undergrads (core course)
- Many masters students
  - To prepare them for upper-level systems courses
- Variety of others from math, physics, statistics, ...

Preparation
- Optional: Introduction to CS in Python or Ruby
- Imperative programming in C subset
ICS Feedback

Students

Faculty

- Prerequisite for most upper level CS systems courses
- Also required for ECE embedded systems, architecture, and network courses
Lecture Coverage

Data representations [3]
- It’s all just bits.
- int’s are not integers and float’s are not reals.

IA32 & x86-64 machine language [5]
- Analyzing and understanding compiler-generated machine code.

Program optimization [2]
- Understanding compilers and modern processors.

Memory Hierarchy [3]
- Caches matter!

Linking [1]
- With DLL’s, linking is cool again!
Lecture Coverage (cont)

Exceptional Control Flow [2]
- The system includes an operating system that you must interact with.

Virtual memory [4]
- How it works, how to use it, and how to manage it.

Application level concurrency [3]
- Processes and threads
- Races, synchronization

I/O and network programming [4]
- Programs often need to talk to other programs.

Total: 27 lectures, 14 week semester
Labs

Key teaching insight:
- Cool Labs $\Rightarrow$ Great Course

A set of 1 and 2 week labs define the course.

Guiding principles:
- Be hands on, practical, and fun.
- Be interactive, with continuous feedback from automatic graders.
- Find ways to challenge the best while providing worthwhile experience for the rest.
- Use healthy competition to maintain high energy.
Lab Exercises

Data Lab (2 weeks)
- Manipulating bits.

Bomb Lab (2 weeks)
- Defusing a binary bomb.

Buffer Lab (1 week)
- Exploiting a buffer overflow bug.

Performance Lab (2 weeks)
- Optimizing kernel functions.

Shell Lab (1 week)
- Writing your own shell with job control.

Malloc Lab (2-3 weeks)
- Writing your own malloc package.

Proxy Lab (2 weeks)
- Writing your own concurrent Web proxy.
Data Lab

Goal: Solve some “bit puzzles” in C using a limited set of logical and arithmetic operators.

- Examples: absval(x), greaterthan(x,y), log2(x)

Lessons:

- Information is just bits in context.
- C int’s are not the same as integers.
- C float’s are not the same as reals.

Infrastructure

- Configurable source-to-source C compiler that checks for compliance.
- Instructor can automatically select from 45 puzzles.
- Automatic testing using formal verification tools
Let’s Solve a Bit Puzzle!

/*
 * abs - absolute value of x (except returns TMin for TMin)
 *   Example: abs(-1) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 10
 *   Rating: 4
 */
int abs(int x) {
    int mask = x >> 31;
    return ____________________________;
}

\[
\begin{align*}
-1 & \quad x < 0 \\
0 & \quad x \geq 0
\end{align*}
\]

\[
\begin{align*}
& (x^\text{mask}) + 1+\text{~mask} \\
& = \\
& -x, \quad x < 0 \\
& x, \quad x \geq 0
\end{align*}
\]
Verifying Solutions

Do these functions produce identical results?

How could you find out?

```c
int abs(int x) {
    int mask = x>>31;
    return (x ^ mask) + ~mask + 1;
}

int test_abs(int x) {
    return (x < 0) ? -x : x;
}
```
Bit-Level Program Model

- View computer word as 32 separate bit values
- Each output becomes Boolean function of inputs

```c
int abs(int x) {
    int mask = x>>31;
    return (x ^ mask) + ~mask + 1;
}
```
Bit-Level Program Verification

- Determine whether functions equivalent for all outputs $j$
- Exhaustive checking:
  - Single input: $2^{32}$ cases $\times$ 50 cycles $\approx 60$ seconds
  - Two input: $2^{64}$ cases $\Rightarrow 8,800$ years!
- Other approaches
  - BDDs, SAT solvers
  - Easily handle these functions (< 1.0 seconds)
**Verification Example**

```c
int iabs(int x) {
    if (x == 1234567) x++;  
    int mask = x>>31;       
    return (x ^ mask) + ~mask + 1;
}
```

**Almost Correct**

- **Valid for all but one input value**
- **Overlooked by our test suite**
Counterexample Generation

\[
\text{int iabs(int x) \{} \\
\quad \text{if (x == 1234567) x++;} \\
\quad \text{int mask = x>>31;} \\
\quad \text{return (x ^ mask) + ~mask + 1;} \\
\text{\}}
\]

Detected By Checking Code

- Since covers \textit{all} cases
- Generate counterexample to demonstrate problem

\[
\text{int main() \{} \\
\quad \text{int val1 = iabs(1234567);} \\
\quad \text{int val2 = test_iabs(1234567);} \\
\quad \text{printf("iabs(1234567) --> %d [0x%x]\n", val1, val1);} \\
\quad \text{printf("test_iabs(1234567) --> %d [0x%x]\n", val2, val2);} \\
\quad \text{if (val1 == val2) \{} \\
\quad \quad \text{printf(".. False negative\n");} \\
\quad \text{\} else} \\
\quad \text{printf(".. A genuine counterexample\n");} \\
\text{\}}
\]
Bomb Lab

- Idea due to Chris Colohan, TA during inaugural offering

*Bomb*: C program with six *phases*.

Each phase expects student to type a specific string.
- Wrong string: bomb *explodes* by printing BOOM! (- ½ pt)
- Correct string: phase *defused* (+10 pts)
- In either case, bomb sends message to grading server
- Server posts current scores anonymously and in real time on Web page

Goal: Defuse the bomb by defusing all six phases.
- For fun, we include an unadvertised seventh *secret phase*

The challenge:
- Each student get only binary executable of a *unique* bomb
- To defuse their bomb, students must disassemble and reverse engineer this binary
Properties of Bomb Phases

Phases test understanding of different C constructs and how they are compiled to machine code

- Phase 1: string comparison
- Phase 2: loop
- Phase 3: switch statement/jump table
- Phase 4: recursive call
- Phase 5: pointers
- Phase 6: linked list/pointers/structs
- Secret phase: binary search (biggest challenge is figuring out how to reach phase)

Phases start out easy and get progressively harder
Let’s defuse a bomb phase!

08048b48 <phase_2>:

```assembly
... # function prologue not shown
8048b50:    mov 0x8(%ebp),%edx             # edx = &str
8048b53:    add $0xfffffffff8,%esp      # eax = &num[] on stack
8048b56:    lea 0xfffffffffe8(%ebp),%eax # eax = &num[] on stack
8048b59:    push %eax                    # push function args
8048b5a:    push %edx                    # edx = &str
8048b5b:    call 8048f48 <read_six_nums> # rd 6 ints from str 2 num
8048b60:    mov $0x1,%ebx                 # i = 1
8048b68:    lea 0xffffffffffe8(%ebp),%esi # esi = &num[] on stack
8048b70:    mov 0xfffffffffffec(%esi,%ebx,4),%eax # LOOP: eax = num[i-1]
8048b74:    add $0x5,%eax               # eax = num[i-1] + 5
8048b77:    cmp %eax,(%esi,%ebx,4)       # if num[i-1] + 5 == num[i]
8048b7a:    je 8048b81 <phase_2+0x39>   # then goto OK:
8048b7c:    call 804946c <explode_bomb> # else explode!
8048b81:    inc %ebx                   # OK: i++
8048b82:    cmp $0x5,%ebx             # if (i <= 5)
8048b85:    jle 8048b70 <phase_2+0x28> # then goto LOOP:
... # function epilogue not shown
8048b8f:    ret # YIPPEE!
```

# LOOP: eax = num[i-1]
# edx = &str
# eax = &num[] on stack
# push function args
# rd 6 ints from str 2 num
# i = 1
# esi = &num[] on stack
# LOOP: eax = num[i-1]
# eax = num[i-1] + 5
# if num[i-1] + 5 == num[i]
# then goto OK:
# else explode!
# OK: i++
# if (i <= 5)
# then goto LOOP:
# YIPPEE!
Source Code for Bomb Phase

```c
/*
 * phase2b.c - To defeat this stage the user must enter arithmetic
 * sequence of length 6 and delta = 5.
 */
void phase_2(char *input)
{
    int ii;
    int numbers[6];

    read_six_numbers(input, numbers);

    for (ii = 1; ii < 6; ii++) {
        if (numbers[ii] != numbers[ii-1] + 5)
            explode_bomb();
    }
}
```
The Beauty of the Bomb

For the Student

- Get a deep understanding of machine code in the context of a fun game
- Learn about machine code in the context they will encounter in their professional lives
  - Working with compiler-generated code
- Learn concepts and tools of debugging
  - Forward vs backward debugging
  - Students *must* learn to use a debugger to defuse a bomb

For the Instructor

- Self-grading
- Scales to different ability levels
- Easy to generate variants and to port to other machines
Buffer Bomb

```c
int getbuf()
{
    char buf[12];
    /* Read line of text and store in buf */
    gets(buf);
    return 1;
}
```

**Task**

- Each student assigned “cookie”
  - Randomly generated 8-digit hex string
- Type string that will cause `getbuf` to return cookie
  - Instead of 1
Calling function `gets(p)` reads characters up to `\n`
Stores string + terminating null as bytes starting at `p`
Assumes enough bytes allocated to hold entire string
Buffer Code: Good case

**void test()**{  
  int v = getbuf();  
  ...  
}

**void getbuf()** {  
  char buf[12];  
  gets(buf);  
  return 1;  
}

- Fits within allocated storage
  - String is 11 characters long + 1 byte terminator

Input string
“01234567890”

Stack Frame for test

<table>
<thead>
<tr>
<th>Saved %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

Return address

Return address

Increasing addresses

%ebp

buf
Buffer Code: Bad case

- Overflows allocated storage
  - Corrupts saved frame pointer and return address
- Jumps to address 0x00383736 when `getbuf` attempts to return
  - Invalid address, causes program to abort

```c
void test() {
    int v = getbuf();
    ...
}

void getbuf() {
    char buf[12];
    gets(buf);
    return 1;
}
```

Input string
“0123456789012345678”

Stack Frame for test

<table>
<thead>
<tr>
<th>Return address</th>
<th>Saved</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>30</td>
<td>39</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Increasing addresses

Input string:

```
0123456789012345678
```

Increasing addresses:

```
00 38 37 36
```

Return address

Saved:

```
%ebp
```

%ebp

buf
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `getbuf()` executes return instruction, will jump to exploit code

```c
void getbuf() {
    char buf[12];
    gets(buf);
    return 1;
}

void test() {
    int v = getbuf();
    ...
}
```

Exploit string for cookie 0x12345678 (not printable as ASCII)

```
void test() {
    int v = getbuf();
    ...
}
```
## Exploit Code

```c
void getbuf() {
    char buf[12];
    gets(buf);
    return 1;
}
```

- Repairs corrupted stack values
- Sets 0x12345678 as return value
- Reexecutes return instruction
- As if getbuf returned 0x12345678

```assembly
pushl $ 0x80489ee  # Restore return pointer
movl $ 0x12345678 ,%eax  # Alter return value
ret
.long 0xbfffb8c8  # Saved value of %ebp
.long 0xbfffb89c  # Location of buf
```

After executing code:

- Stack Frame for test
- Return address
- Saved %ebp
- %ebp
- buf (0xffffb89c)
Why Do We Teach This Stuff?

Important Systems Concepts

- Stack discipline and stack organization
- Instructions are byte sequences
- Making use of tools
  - Debuggers, assemblers, disassemblers

Computer Security

- What makes code vulnerable to buffer overflows
- The most exploited vulnerability in systems

Impact

- CMU student teams consistently win international Capture the Flag Competitions
Performance Lab

Goal: Make small C kernels run as fast as possible
- Examples: DAG to UDG conversion, convolution, rotate, matrix transpose, matrix multiply

Lessons:
- Caches and locality of reference matter.
- Simple transformations can help the compiler generate better code.
- Improvements of 3–10X are possible.

Infrastructure
- Students submit solutions to an evaluation server.
- Server posts sorted scores in real-time on Web page
Shell Lab

Goal: Write a Unix shell with job control
- (e.g., ctrl-z, ctrl-c, jobs, fg, bg, kill)

Lessons:
- First introduction to systems-level programming and concurrency
- Learn about processes, process control, signals, and catching signals with handlers
- Demystifies command line interface

Infrastructure
- Students use a scripted autograder to incrementally test functionality in their shells
Malloc Lab

Goal: Build your own dynamic storage allocator

```c
void *malloc(size_t size)
void *realloc(void *ptr, size_t size)
void free(void *ptr)
```

Lessons

- Sense of programming underlying system
- Large design space with classic time-space tradeoffs
- Develop understanding of scary “action at a distance” property of memory-related errors
- Learn general ideas of resource management

Infrastructure

- Trace driven test harness evaluates implementation for combination of throughput and memory utilization
- Evaluation server and real time posting of scores
Proxy Lab

Goal: write concurrent Web proxy.

Lessons: Ties together many ideas from earlier

- Data representations, byte ordering, memory management, concurrency, processes, threads, synchronization, signals, I/O, network programming, application-level protocols (HTTP)

Infrastructure:

- Plugs directly between existing browsers and Web servers
- Grading is done via autograders and one-on-one demos
- Very exciting for students, great way to end the course
ICS Summary

Proposal

- *Introduce students to computer systems from the programmer's perspective rather than the system builder's perspective*

Themes

- What parts of the system affect the correctness, efficiency, and utility of my C programs?
- Makes systems fun and relevant for students
- Prepare students for builder-oriented courses
  - Architecture, compilers, operating systems, networks, distributed systems, databases, …
  - Since our course provides complementary view of systems, does not just seem like a watered-down version of a more advanced course
  - Gives them better appreciation for what to build
CMU Courses that Build on ICS

CS
- Compilers
- Secure Coding
- Software Engin.

Parallel Systems
- Dist. Systems
- Networks
- Operating Systems

Storage Systems
- Databases

Robotics
- Cog. Robotics
- Comp. Photo.
- Computer Graphics

ECE
- Embedded Control
- Real-Time Systems
- Embedded Systems
- Computer Arch.

ICS
Fostering “Friendly Competition”

Desire

- Challenge the best without frustrating everyone else

Method

- Web-based submission of solutions
- Server checks for correctness and computes performance score
  - How many stages passed, program throughput, ...
- Keep updated results on web page
  - Students choose own *nom de guerre*

Relationship to Grading

- Students get full credit once they reach set threshold
- Push beyond this just for own glory/excitement
Shameless Promoting

- http://csapp.cs.cmu.edu
- Second edition Published 2010
- In use at 186 institutions worldwide
International Editions

Computer Systems
A Programmer's Perspective
Second Edition
Randal E. Bryant • David R. O'Hallaron
Overall Sales

- First + Second Editions
- As of 12/31/2011
- 116,574 total
Worldwide Adoptions

186 total
North American Adoptions

114 total
CS:APP2e

Vital stats:

- 12 chapters
- 233 practice problems (solutions in book)
- 180 homework problems (solutions in instructor’s manual)
- 475 figures, 282 line drawings
- Many C & machine code examples

Turn-key course provided with book:

- Electronic versions of all code examples.
- Powerpoint, EPS, and PDF versions of each line drawing
- Password-protected Instructors Page, with Instructor’s Manual, Lab Infrastructure, Powerpoint lecture notes, and Exam problems.
Coverage

Material Used by ICS at CMU

- Pulls together material previously covered by multiple textbooks, system programming references, and man pages

Greater Depth on Some Topics

- Dynamic linking
- I/O multiplexing

Additional Topic

- Computer Architecture
- Added to cover all topics in “Computer Organization” course
Architecture

Material

- Y86 instruction set
  - Simplified/reduced IA32

- Implementations
  - Sequential
  - 5-stage pipeline

Presentation

- Simple hardware description language to describe control logic
- Descriptions translated and linked with simulator code

Labs

- Modify / extend processor design
  - New instructions
  - Change branch prediction policy
- Simulate & test results
Web Asides

- Supplementary material via web
- Topics either more advanced or more arcane

Examples

- Boolean algebra & Boolean rings
- Combining assembly & C code
- x87 and SSE floating point code
- Using SIMD instructions
- Asynchronous signal safety
Courses Based on CS:APP

**Computer Organization**

ORG  Topics in conventional computer organization course, but with a different flavor

ORG+  Extends computer organization to provide more emphasis on helping students become better application programmers

**Introduction to Computer Systems**

ICS  Create enlightened programmers who understand enough about processor/OS/compilers to be effective

ICS+  What we teach at CMU. More coverage of systems software

**Systems Programming**

SP  Prepare students to become competent system programmers
# Courses Based on CS:APP

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Topic</th>
<th>ORG</th>
<th>ORG+</th>
<th>ICS</th>
<th>ICS+</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2</td>
<td>Data representations</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>3</td>
<td>Machine language</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>4</td>
<td>Processor architecture</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Code optimization</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>6</td>
<td>Memory hierarchy</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>7</td>
<td>Linking</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>8</td>
<td>Exceptional control flow</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>9</td>
<td>Virtual memory</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>10</td>
<td>System-level I/O</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Concurrent programming</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Network programming</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>

- **Partial Coverage**
- **Complete Coverage**
Conclusions

ICS Has Proved Its Success

- Thousands of students at CMU over 13 years
- Positive feedback from alumni
- Positive feedback from systems course instructors

CS:APP is International Success

- Supports variety of course styles
- Many purchases for self study