Course Overview

CSCI 2400/ ECE 3217: Computer Architecture

Instructors:
David Ferry

Slides adapted from Bryant & O’Hallaron’s slides
Overview

- Course theme
- Five realities
- Logistics
Course Theme:

Abstraction Is Good But Don’t Forget Reality

■ Most CS and CE courses emphasize abstraction
  ▪ Abstract data types
  ▪ Asymptotic analysis

■ These abstractions have limits
  ▪ Especially in the presence of bugs
  ▪ Need to understand details of underlying implementations

■ Useful outcomes
  ▪ Become more effective programmers
    ▪ Able to find and eliminate bugs efficiently
    ▪ Able to understand and tune for program performance
  ▪ Prepare for later “systems” classes in CS & ECE
    ▪ Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems
Great Reality #1:
Ints are not Integers, Floats are not Reals

Example 1: Is $x^2 \geq 0$?

- Float’s: Yes!
- Int’s:
  - $40000 \times 40000 \Rightarrow 1600000000$
  - $50000 \times 50000 \Rightarrow ??$

Example 2: Is $(x + y) + z = x + (y + z)$?

- Unsigned & Signed Int’s: Yes!
- Float’s:
  - $(1e20 + -1e20) + 3.14 \Rightarrow 3.14$
  - $1e20 + (-1e20 + 3.14) \Rightarrow ??$

Source: xkcd.com/571
Consequences of Abstraction: Code Security

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD’s implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs
Typical Usage

/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

#define MSIZE 528
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}

- AKA “buffer over-read”
- This is essentially what happened with the Heartbleed bug (2014)
Computer Arithmetic

■ Does not generate random values
  ▪ Arithmetic operations have important mathematical properties

■ Cannot assume all “usual” mathematical properties
  ▪ Due to finiteness of representations
  ▪ Integer operations satisfy “ring” properties
    ▪ Commutativity, associativity, distributivity
  ▪ Floating point operations satisfy “ordering” properties
    ▪ Monotonicity, values of signs

■ Observation
  ▪ Need to understand which abstractions apply in which contexts
  ▪ Important issues for compiler writers and serious application programmers
Great Reality #2: You’ve Got to Know Assembly

- Chances are, you’ll never write programs in assembly
  - Compilers are much better & more patient than you are

- But: Assembly is key to understanding machine-level execution
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!
Assembly Code Example

**Time Stamp Counter**
- Special 64-bit register in Intel-compatible machines
- Incremented every clock cycle
- Read with rdtsc instruction

**Application**
- Measure time (in clock cycles) required by procedure

```c
double t;
start_counter();
P();
t = get_counter();
printf("P required %f clock cycles\n", t);
```
Code to Read Counter

- Write small amount of assembly code using GCC’s asm facility
- Inserts assembly code into machine code generated by compiler

```c
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits
   of the cycle counter.
*/
void access_counter(unsigned *hi, unsigned *lo)
{
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
         : "=r" (*hi), "=r" (*lo)
         : : "edx", "eax");
}
```
Great Reality #3: Memory Matters
Random Access Memory Is an Unphysical Abstraction

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
  - Memory access is relatively slow

- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space

- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) => 3.14
fun(1) => 3.14
fun(2) => 3.1399998664856
fun(3) => 2.00000061035156
fun(4) => 3.14, then segmentation fault

■ Result is architecture specific
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) => 3.14
fun(1) => 3.14
fun(2) => 3.1399998664856
fun(3) => 2.00000061035156
fun(4) => 3.14, then segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Saved State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>4</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>3</td>
</tr>
<tr>
<td>a[1]</td>
<td>2</td>
</tr>
<tr>
<td>a[0]</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Referencing Errors

- **C and C++ do not provide any memory protection**
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free

- **Can lead to nasty bugs**
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated

- **How can I deal with this?**
  - Program in Java, Python, etc.
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)
Memory System Performance Example

- Hierarchical memory organization
  - Caches trade off speed and size
- Performance depends on access patterns
  - Including how step through multi-dimensional array

```c
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```c
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

21 times slower (Pentium 4)
The Memory Mountain

Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
Great Reality #4: There’s more to performance than asymptotic complexity

- Constant factors matter too!
- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Example Matrix Multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- What is going on?
MMM Plot: Analysis

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz Gflop/s

- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice

- Effect: fewer register spills, L1/L2 cache misses, and TLB misses
Great Reality #5:
Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance
  - Hard drives, SSDs, USB, CD/DVD, keyboards, mice, etc.
  - What’s important? Response time, latency, or throughput?
  - Large writes vs. Small (4Kb) writes

- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues
Course Perspective

Most Systems Courses are Builder-Centric

- Computer Architecture
  - Design pipelined processor in Verilog
- Operating Systems
  - Implement large portions of operating system
- Compilers
  - Write compiler for simple language
- Networking
  - Implement and simulate network protocols
This Course is more Programmer-Centric

- Purpose is to show how by knowing more about the underlying system, one can be more effective as a programmer
- Enable you to
  - Understand how C/C++ executes in hardware
  - Write programs that are more reliable and efficient
  - Incorporate features that require hooks into OS
    - E.g., concurrency, signal handlers
- Not just a course for dedicated hackers
  - We bring out the hidden hacker in everyone
  - Imagine a zoologist that didn’t know biology or chemistry
- Cover material in this course that you won’t see elsewhere
Course Website

- **Class Website:** [http://cs.slu.edu/~dferry/courses/csci2400](http://cs.slu.edu/~dferry/courses/csci2400)
  - Detailed class information and policies
  - Full Schedule, including:
    - lecture topics and code examples
    - assignments
    - exam dates
  - All assignments posted on website
  - Most lecture slides posted on website
  - Class website is the official syllabus and takes precedence over anything that is said in slides or in class

- **SLU Blackboard**
  - Blackboard is not used for this course
Textbook

- Randal E. Bryant and David R. O’Hallaron,
  - Textbook’s website: http://csapp.cs.cmu.edu
  - Recommend getting a hardcopy, since exams are often open book & notes
    - Laptops, tablets, etc. not allowed during exams

- C reference textbook
  - “C Programming”
  - a free online reference text for C programming, that may prove beneficial for those who haven’t used C (or C++) before
Grading

- **Exams (50%)**
  - mid-semester exams: 15% each
  - final 20%

- **Assignments (45%):** approximately 7-9 assignments

- **Class Participation (5%)**
  - for participation in hands-on work during class

- **Curving policy on website**

- **Late Policy:**
  - 10% penalty for up to 24 hours late
  - 20% penalty for up to 48 hours late
  - Penalty waived or accepted after 48 hours only at instructor’s discretion
Attendance and Class Guidelines

- Attendance is at students’ discretion, but highly recommended

- Questions and Participation highly encouraged
  - If you have a question or need clarification, it’s very likely that other students will likewise benefit from your question

- Laptops / computers may be used during class
  - But NOT during exams
Policy for Collaborating on Assignments

- Collaboration allowed, even encouraged, PROVIDED that:
  - you only discuss the problem, not the solution
  - students may help guide each other in the process of solving the problem, BUT each student MUST turn in their own answer
  - students MUST indicate who they collaborated with on their cover sheet
Cheating

■ What is cheating?
  ▪ *Sharing code or answers*: copying, retyping, looking at, or supplying a file
  ▪ *Detailed coaching*: helping your friend to write code or an answer, line by line
  ▪ *Copying code* from previous course or from elsewhere on WWW
    ▪ only allowed to use code supplied in class or on course website
    ▪ must cite such code

■ What is NOT cheating?
  ▪ Explaining how to use systems or tools
  ▪ Helping others understand design issues or the process for solving a problem

■ Penalty for cheating:
  ▪ Ranges, based on severity, from zero on assignment to being sent before Academic Honesty Committee
  ▪ Records saved for all incidents of cheating

■ Detection of cheating:
  ▪ Instructor is (unfortunately) extremely experienced at detecting cheating
Topic: Programs and Data

- **Topics**
  - Assembly (and machine) language vs. High-level languages (HLLs)
  - Instruction set architecture
    - CPU (core)
    - register file
    - processing units (ALU, FPU, etc.)
  - Types of instructions
    - arithmetic
    - logical
    - shifts and bit manipulation
    - memory
    - compares
    - branches and jumps
    - procedure calls & returns
  - Representation of variables, arrays and data structures
Topic: Computer Architecture

- **Topics**
  - Fundamentals of Logic Design (gates & circuits)
  - Processor Organization
  - CPU (core) Organization
  - Fetch-Decode-Execute Cycle and Datapath Flow
  - Sequential (single-cycle) Datapath
  - Pipelined Datapath
    - purpose / benefit
    - data and control dependencies
    - hazards
    - bypassing / forwarding
    - branch prediction
Topic: Memory and the Memory Hierarchy

- **Topics**
  - Data representation
  - Memory technology (disk vs. RAM vs. ROM vs. cache)
  - Loads & Stores (reads & writes)
  - Physical vs. Virtual memory
    - page tables, address translation, and TLB
    - how memory organized within a process
      - global vs. heap vs. stack memory
  - Cache memory
    - purpose / benefit
    - locality
    - how it works
Topic: Performance and Optimization

Topics

- How simple modifications in assembly / machine code can dramatically affect execution time
- Co-optimization (control and data)
- Measuring time on a computer
- Related to architecture, compilers, and OS
Welcome
and Enjoy!