


Computer Organization

– Overview

Kai Shen


1



Introduction to Computer Systems

- Abstraction and theory are critically important:
 - Abstract data types, asymptotic analysis
- They don't address many practical problems:
 - In the presence of bugs
 - Different systems components interact in complex ways
 - Performance is affected by subtle implementation details
 - Malicious parties exploit vulnerabilities that do not follow conventional system abstraction and design
- The study of computer systems confronts reality:
 - Real implementation, real hardware artifacts, real security vulnerability, real performance anomalies, ...


2



Reality #1: int vs. integer, float vs. real

- Example 1: Is $x^2 \geq 0$?
 - $40000 * 40000 = 1600000000$
 - $50000 * 50000 = ??$
- Example 2: Is $(x + y) + z = x + (y + z)$?
 - **Unsigned & Signed int's:** Yes!
 - **Float's:**
 - $(1e20 + -1e20) + 3.14 = 3.14$
 - $1e20 + (-1e20 + 3.14) = ??$

3



Security Vulnerability

```

/* 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);
}

```

- Real example: similar to code found in FreeBSD's implementation of getpeername

4

Malicious 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);
    . . .
}
```

- **-MSIZE**, when interpreted as **unsigned int** by **memcpy**, becomes a very large integer

5

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: understanding assembly is useful and important
 - Implementing system software
 - Compiler has machine code as target
 - Operating systems do weird things (save/restore process state)
 - Access special hardware features
 - Processor model-specific registers
 - Tuning program performance
 - Understand optimizations done / not done by the compiler
 - Understand sources of program inefficiency
 - Creating / fighting malware
 - x86 assembly is the language of choice!

6

Example: Read Timestamp Counter

- Timestamp Cycle Counter (useful for fine-grained time tracking)
 - Special 64-bit register on many processors
 - Incremented every clock cycle
 - Read with rdtsc instruction
- Write small amount of assembly code using GCC's asm facility

```
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");
}
```

7

Reality #3: Memory Matters

- Random Access Memory is a deceiving abstraction
- Memory is not unbounded
 - It must be allocated and managed
 - Memory referencing bugs are hard to track down (silently propagate and corrupt before manifesting symptoms appear)
- 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

8

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) returns 3.14
 fun(1) returns 3.14
 fun(2) returns 3.1399998664856
 fun(3) returns 2.00000061035156
 fun(4) returns 3.14, then segmentation fault

Explanation:

Saved State	4
d[0] upper 32-bit	3
d[0] lower 32-bit	2
a[1]	1
a[0]	0

} Location accessed by fun(i)

9

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 bug has any effect depends on system/compiler
 - Consequence 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?
 - Understand and debug!
 - Use tools to detect referencing errors (e.g. Valgrind)
 - Program in language with stronger memory protection (Java)

10

Memory System Performance Example

```
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];
}
```

```
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)

- Hierarchical memory organization
- Performance depends on access patterns
 - In this case, how step through multi-dimensional array

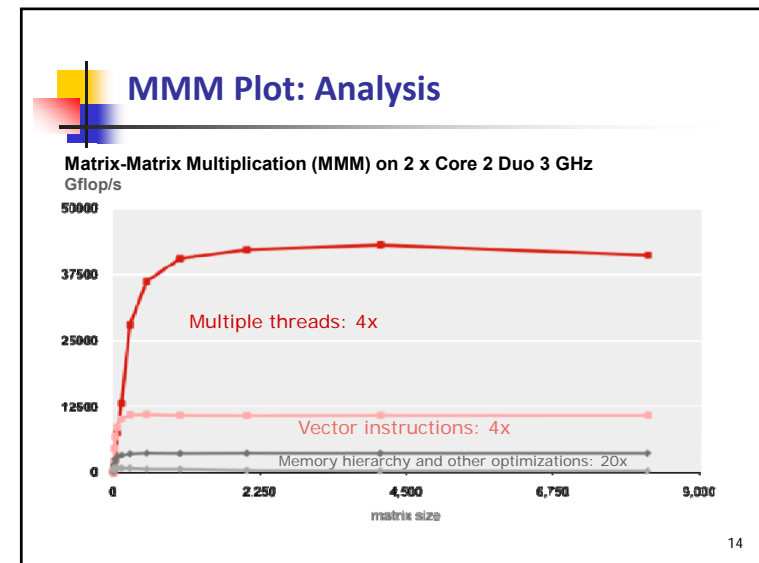
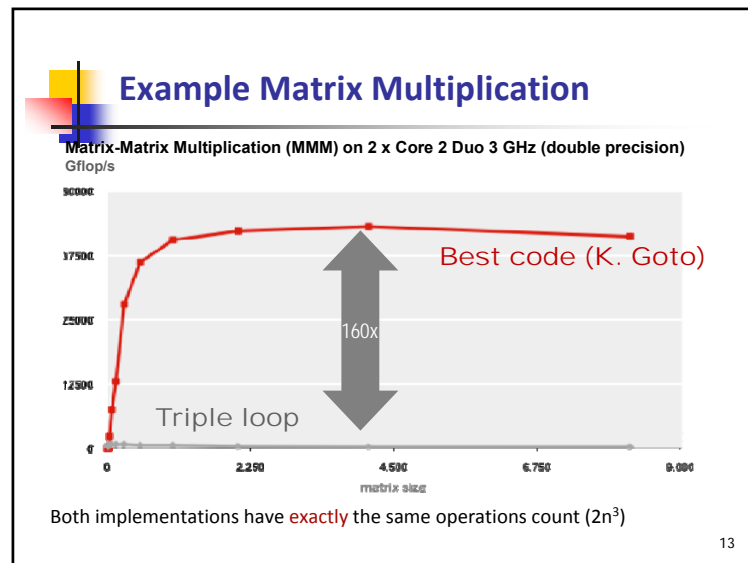
A[0][0] A[0][2047] A[1][0] A[1][2047]

11

Reality #4: 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 is 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

12



Reality #5: Computers do more than execute programs

- They need to get data in and out
 - Computing is increasingly driven by data
- They communicate with each other over networks with new system issues
 - Concurrent operations by autonomous processes
 - Coping with unreliable communication media
 - Cross platform compatibility
 - Complex performance issues
- An I/O operation takes milliseconds to complete – enough time to run tens of millions of instructions (even on a smartphone)

15

Course Goals

- Acquire a broad understanding of computer systems, its components, and how they work together.
- Become more effective programmers
 - Able to find and eliminate bugs efficiently
 - Able to understand and tune for program performance
- Prepare for later “systems” courses
 - Compilers, Operating Systems, Networks, Computer Architecture, Parallel and Distributed Systems

16



Course Perspective

- Later Systems Courses are Builder-Centric
 - Computer Architecture
 - Design pipelined processor
 - Operating Systems
 - Implement large portions of operating system
 - Compilers
 - Write compiler for high-level language
 - Networking
 - Implement and simulate network protocols

17



Course Perspective (Cont.)

- This Course is Programmer-Centric
 - By knowing more about the underlying system, one can be more effective as a programmer
 - Enable you to
 - Write programs that are more reliable and efficient
 - Incorporate features that require hooks into OS (e.g., concurrency, signal handlers)

18



General Course Information

- Course Web page
 - <http://www.cs.rochester.edu/courses/252/spring2015/>
- Textbook
 - “Computer Systems: A Programmer’s Perspective”, by Bryant and O’Hallaron, Second Edition, 2011.
- Instructor and TAs

19



Course Components

- Lectures
 - Higher level concepts
 - Presence in lectures: voluntary, recommended
 - I encourage participation in class discussion
- Lab assignments (5)
 - The heart of the course
 - About 2 weeks each
 - Provide in-depth understanding of an aspect of systems
 - Programming and measurement
- Exams (midterm + final)
 - Test your understanding of concepts & mathematical principles

20



Lab assignments

- Substantial C programming
- Late submissions (not accepted or with substantial penalty)
- Groups
- Accounts in computer science labs
 - If you don't have one, sign up on a sheet AND see Marty

21



Academic Honesty

- What is cheating?
 - Sharing code: by copying, retyping, looking at, or supplying a file
 - Coaching: helping your friend to write a lab
 - Copying code from previous course or from elsewhere on WWW
 - Only allowed to use code we supply
- What is NOT cheating (in fact, encouraged)?
 - Explaining how to use general systems, libraries, or tools for programming and/or debugging
 - Learning through online sources with respect to general, high-level knowledge that is not specific to your lab assignments
 - Helping each other on the understanding of such materials

22



Disclaimer

- These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

23