Computer Systems

CSE 410 Spring 2012
1st Lecture, March 26

Instructor:

Hal Perkins

Teaching Assistants:

Cortney Corbin, Anton Devore, Soumya Vasisht

Today's Agenda

- **Administrivia**
  - Course overview
  - Staff
  - General organization
  - Requirements, assignments, grading
  - Texts and references
  - Policies

- **The course**
  - What it's about, our perspective

Organization and Administrivia

Everything is on the course web page:

http://www.cs.washington.edu/410

Including

- General information, policies, syllabus
- Staff information, office hours (still working on that)
- **Office hour doodle!!** Please fill in your best times – help us schedule
- Links to discussion board (please use), archive of email announcements from course staff (you are subscribed to this list at your @uw email address if you are enrolled)
- Calendar(s) with lecture slides, links to assignments, etc.
- Information and links to computing resources and reference info
- Etc., etc., etc. & let us know if there's anything else we should add

Us

Instructor

Hal Perkins, CSE 548, perkins@cs

TAs

Cortney Corbin

Anton Devore

Soumya Vasisht

Use the discussion board for most communications, but if you need to contact us via email please use cse410-staff@cs

Who are you?

- **60+ students (wow!)**

- **Who has written programs in assembly before?**
- **Written a threaded program before?**

- **Who knows C? Linux?**
  (You'll learn a lot about how C programs execute, but this isn't a C programming course – don't worry if you didn't take CSE 374)

- **What is hardware? Software?**
- **What is an operating system?**
  (Lazowska's answer: "I don't know. Nobody knows. They're programs – big hairy programs.")

Course Registration

If you don't plan to take the course, please drop soon to open up room for people who want in

If you're still trying to get in (how many of you are there?) please sign up before you leave today. We'll do what we can depending on how many people there are.

C vs. Assembler vs. Machine Programs

```
if ( x != 0 ) y = (y+z) / x;
```

```
        cmpl    $0, -4(%ebp)
        je      .L2
        movl    -12(%ebp), %eax
        movl    -8(%ebp), %edx
        leal    (%edx,%eax), %eax
        movl    %eax, %edx
        sarl    $31, %edx
        idivl   -4(%ebp)
        movl    %eax, -8(%ebp)
.L2:
```

```
1000001101111110000100100000111000000000
0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
100011010000010000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011100
10001001010001000010010000011000
```

C vs. Assembler vs. Machine Programs

```
if ( x != 0 ) y = (y+z) / x;
```

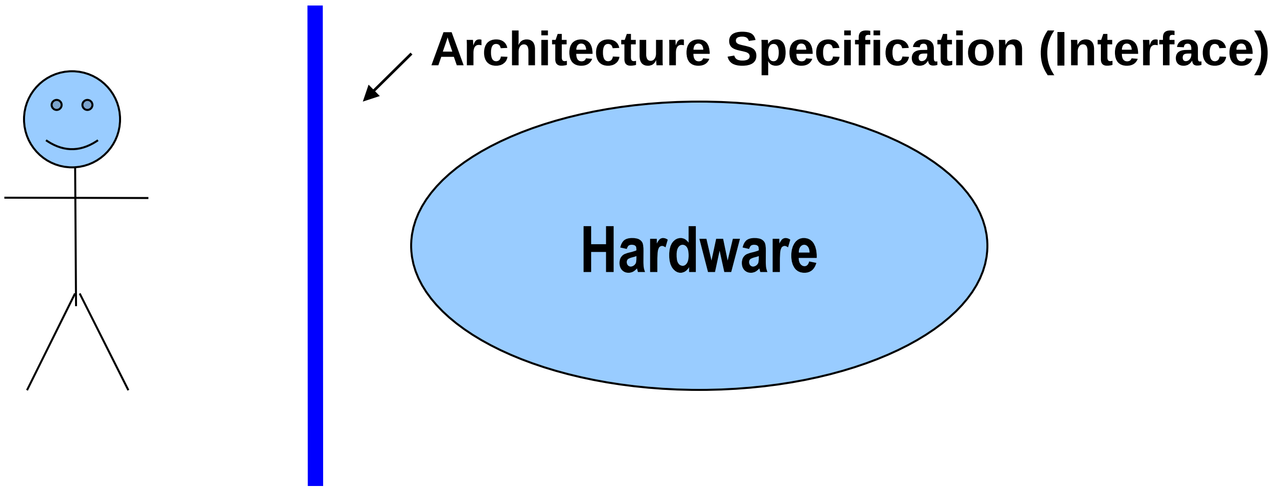
```
        cmpl    $0, -4(%ebp)
        je      .L2
        movl    -12(%ebp), %eax
        movl    -8(%ebp), %edx
        leal    (%edx,%eax), %eax
        movl    %eax, %edx
        sarl    $31, %edx
        idivl   -4(%ebp)
        movl    %eax, -8(%ebp)
.L2:
```

```
1000001101111100001001000001110000000000
0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
100011010000010000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011100
10001001010001000010010000011000
```

- **The three program fragments are equivalent**
- **You'd rather write C!**
- **The hardware likes bit strings!**
  - The machine instructions are actually much shorter than the bits required to represent the characters of the assembler code

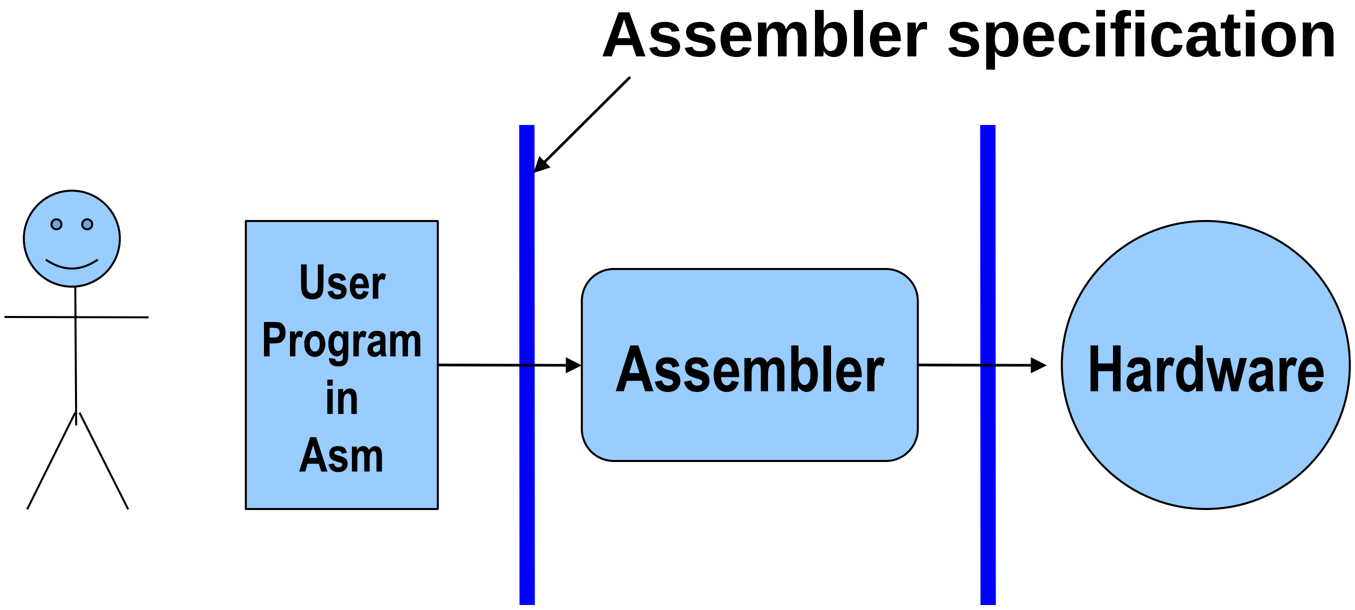
HW/SW Interface: The Historical Perspective

- **Hardware started out quite primitive**
  - Design was expensive $\Rightarrow$ the instruction set was very simple
    - E.g., a single instruction can add two integers
- **Software was also very primitive**



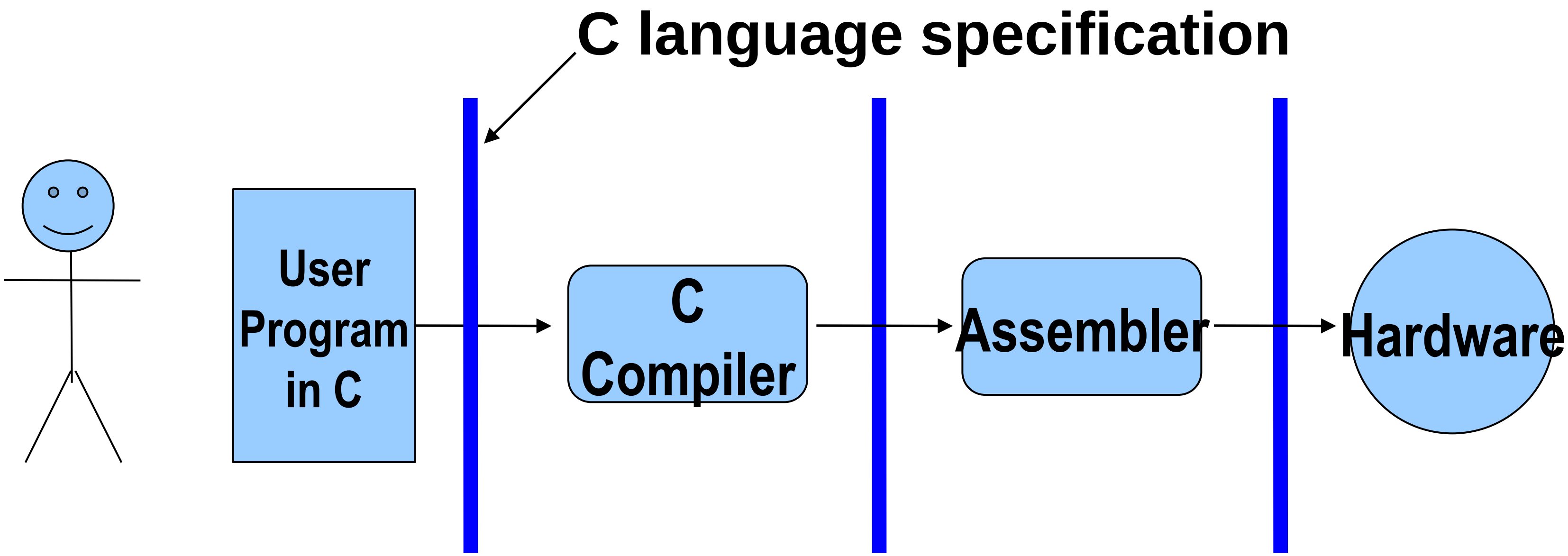
HW/SW Interface: Assemblers

- **Life was made a lot better by assemblers**
  - 1 assembly instruction = 1 machine instruction, but...
  - different syntax: assembly instructions are character strings, not bit strings

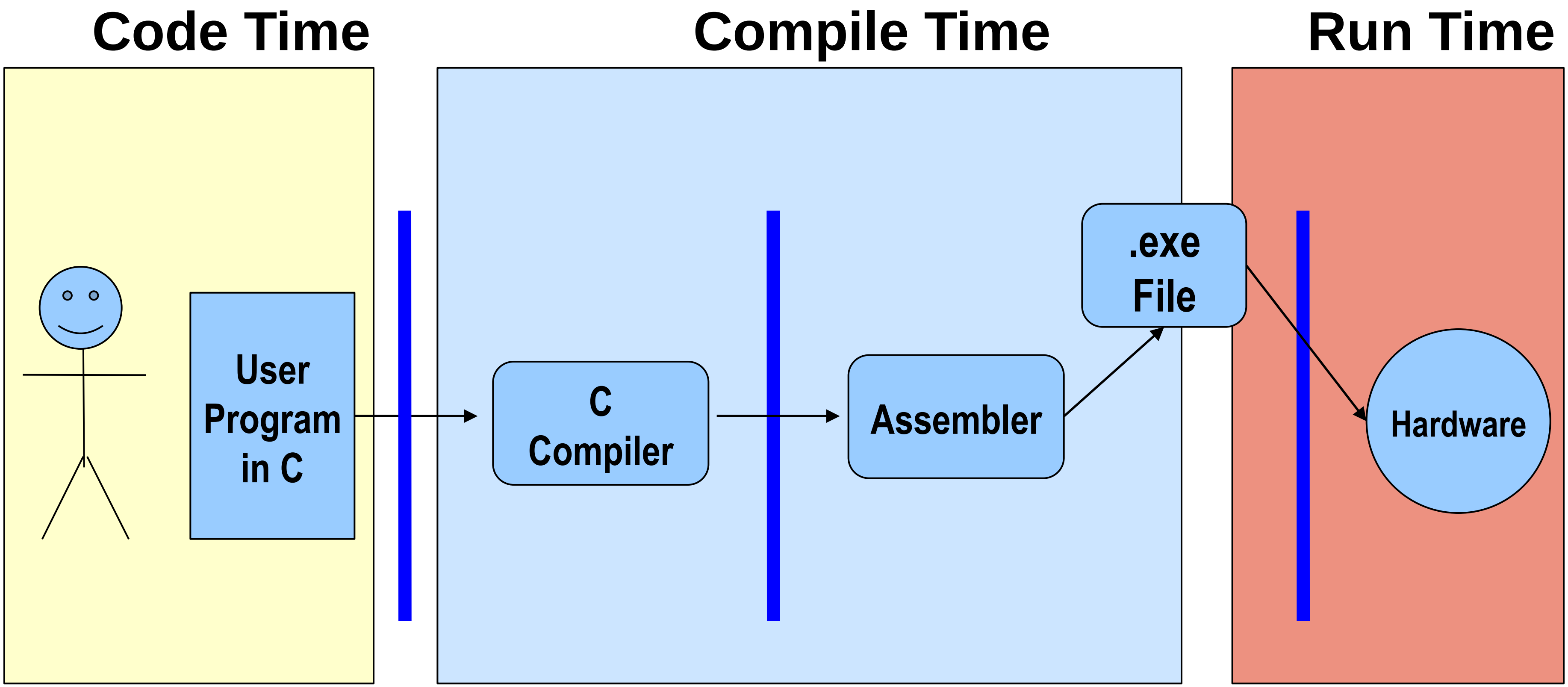


HW/SW Interface: Higher Level Languages (HLL's)

- **Higher level of abstraction:**
  - 1 HLL line is compiled into many (many) assembler lines

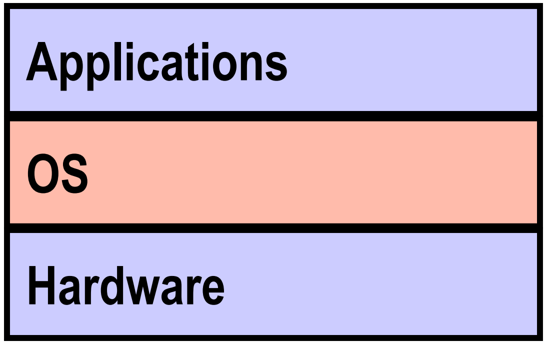


HW/SW Interface: Code / Compile / Run Times



Note: The compiler and assembler are just programs, developed using this same process.

Operating Systems – The Traditional Picture



- **"The OS is everything you don't need to write in order to run your application"**
- **This depiction invites you to think of the OS as a library; we'll see that**
  - In some ways, it is:
    - all operations on I/O devices require OS calls (syscalls)
  - In other ways, it isn't:
    - you use the CPU/memory without OS calls
    - it intervenes without having been explicitly called

Overview

- Course themes: big and little
- Four important realities
- What's new this year
- (More) Logistics

- (ready? ☺)

The Big Theme

- **WHAT’S REALLY HAPPENING WHEN YOUR PROGRAM RUNS??**
- **How does the hardware (0s and 1s, processor executing instructions) relate to the software (C/Java programs)?**
- **Computing is about abstractions (but don’t forget reality)**
- **What are the abstractions that we use?**
- **What do YOU need to know about them?**
  - When do they break down and you have to peek under the hood?
  - What bugs can they cause and how do you find them?
- **Become a better programmer and begin to understand the thought processes that go into building computer systems**

Little Theme 1: Representation

- **All digital systems represent everything as 0s and 1s**
- **Everything includes:**
  - Numbers – integers and floating point
  - Characters – the building blocks of strings
  - Instructions – the directives to the CPU that make up a program
  - Pointers – addresses of data objects in memory
- **These encodings are stored in registers, caches, memories, disks, etc.**
- **They all need addresses**
  - A way to find them
  - Find a new place to put a new item
  - Reclaim the place in memory when data no longer needed

Little Theme 2: Translation

- **There is a big gap between how we think about programs and data and the 0s and 1s of computers**
- **Need languages to describe what we mean**
- **Languages need to be translated one step at a time**
  - Word-by-word
  - Phrase structures
  - Grammar
- **We know Java as a programming language**
  - Have to work our way down to the 0s and 1s of computers
  - Try not to lose anything in translation!
  - We'll encounter the C language, assembly language, and machine code (for the x86 family of CPU architectures)

Little Theme 3: Control Flow

- **How do computers orchestrate the many things they are doing – seemingly in parallel**
- **What do we have to keep track of when we call a method, and then another, and then another, and so on**
- **How do we know what to do upon "return"**
- **User programs and operating systems**
  - Multiple user programs
  - Operating system has to orchestrate them all
    - Each gets a share of computing cycles
    - They may need to share system resources (memory, I/O, disks)
  - Yielding and taking control of the processor
    - Voluntary or by force?

Course Outcomes

- **Foundation: basics of high-level programming (Java, C)**
- **Understanding of some of the abstractions that exist between programs and the hardware they run on, why they exist, and how they build upon each other**
- **Knowledge of some of the details of underlying implementations**
- **Become more effective programmers**
  - More efficient at finding and eliminating bugs
  - Understand the many factors that influence program performance
  - Facility with some of the many languages that we use to describe programs and data

Reality 1: Ints ≠ Integers & Floats ≠ Reals

- **Representations are finite**
- **Example 1: Is $x^2 \geq 0$?**
  - Floats: Yes!
  - Ints:
    - 40000 * 40000 --> 1600000000
    - 50000 * 50000 --> ??
- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Ints: Yes!
  - Floats:
    - (1e20 + -1e20) + 3.14 --> 3.14
    - 1e20 + (-1e20 + 3.14) --> ??

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE]; int len = 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 */
    if (KSIZE > maxlen) len = 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]; int len = 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 */
    if (KSIZE > maxlen) len = 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);
}
```

Malicious Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE]; int len = 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 */
    if (KSIZE > maxlen) len = maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

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

Reality #2: You've Got to Know Assembly

- **Chances are, you'll never write a program in assembly code**
  - Compilers are much better and more patient than you are
- **But: Understanding assembly is the key to the machine-level execution model**
  - Behavior of programs in presence of bugs
    - High-level language model breaks down
  - Tuning program performance
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Operating systems must manage process state
  - Creating / fighting malware
  - x86 assembly is the language of choice
  - Use special thingees inside processor!

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

```
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**

```
/* 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) /* output    */
        :                        /* input     */
        : "%edx", "%eax");       /* clobbered */
}
```

Reality #3: Memory Matters

- **Memory is not unbounded**
  - It must be allocated and managed
  - Many applications are memory-dominated
- **Memory referencing bugs are 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
```

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:

`Saved State`	4
`d7 … d4`	3
`d3 … d0`	2
`a[1]`	1
`a[0]`	0

Location accessed by `fun(i)`

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 (or C#, or ML, or …)
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors

Memory System Performance Example

- **Hierarchical memory organization**
- **Performance depends on access patterns**
  - Including how program steps through multi-dimensional array

```
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)

Reality #4: Performance isn’t counting ops

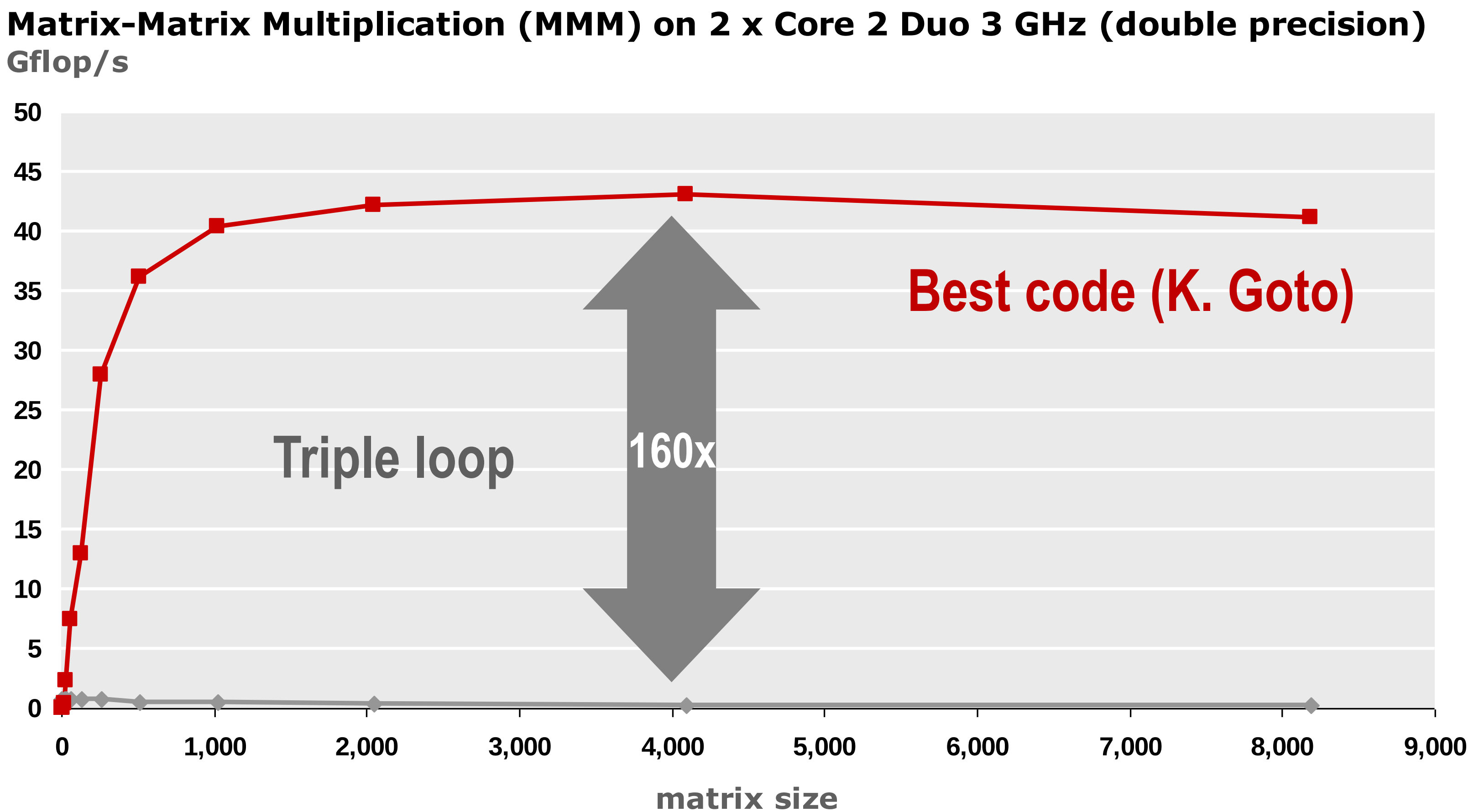
- Can you tell how fast a program is just by looking at the code?

Reality #4: Performance isn’t counting ops

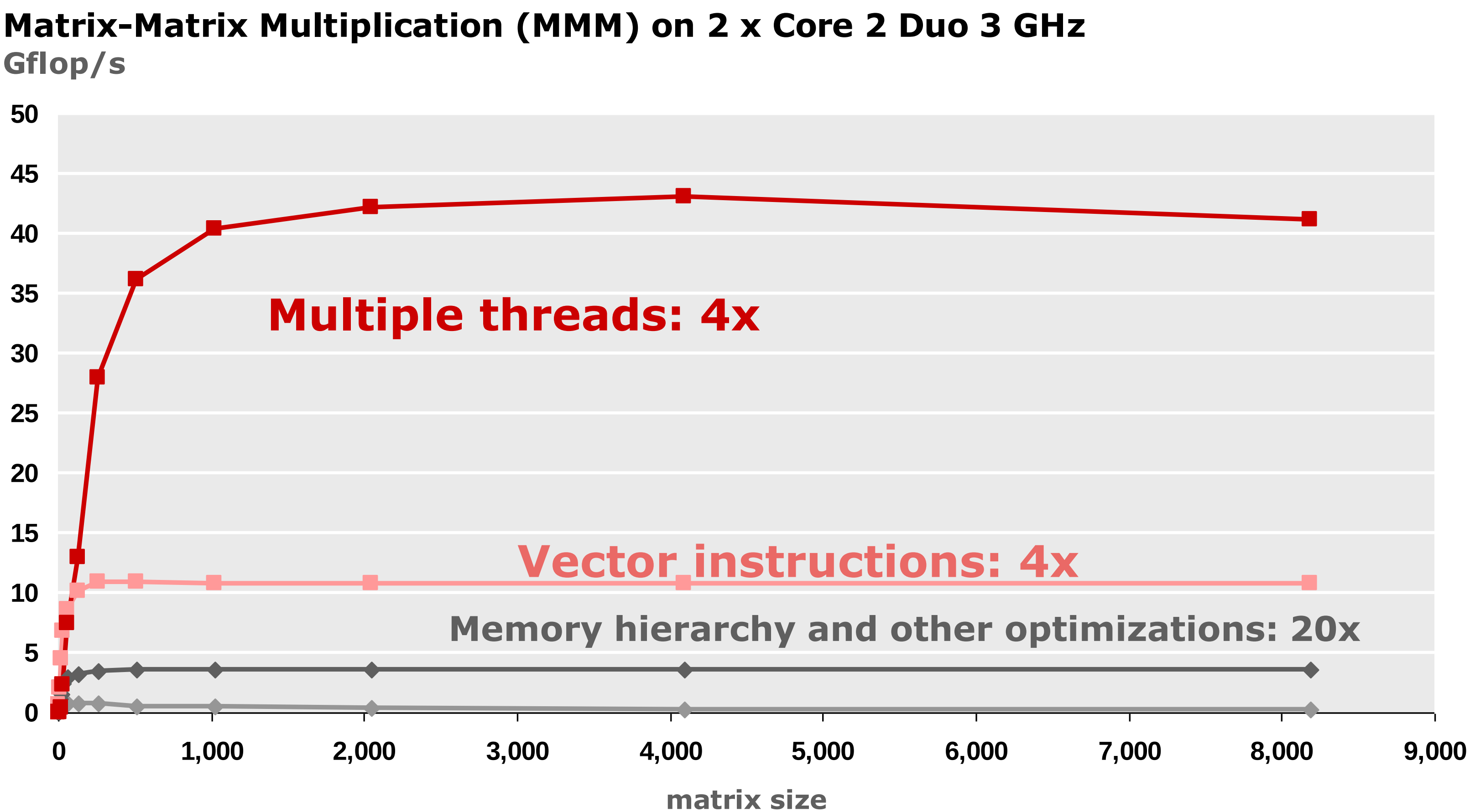
- **Exact op count does not predict performance**
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs compiled and executed
  - How memory system is organized
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Example Matrix Multiplication

- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)



MMM Plot: Analysis



- Reason for 20x: blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- *Effect: less register spills, less L1/L2 cache misses, less TLB misses*

CSE 410: What's new this year?

- **Big changes to bring the course up to date:**
  - Old: write assembly language, learn how processor is built
  - New: look at assembly language generated by compilers
  - Using x86-64/Linux/C instead of MIPS assembler as the vehicle
  - Matches changes in CSE core curriculum – borrowing a lot from CSE 351

- **But major topics are much the same:**
  - Instructions and data at the machine level
  - Memory hierarchy (caches, virtual memory)
  - What the OS does and how it interacts with programs

Course Perspective

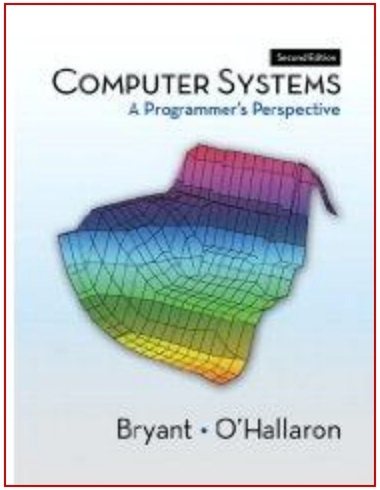
- **Traditional systems courses are Builder-Centric**
  - Computer Architecture
    - Design pipelined processor in Verilog
  - Operating Systems
    - Implement large portions of operating system
  - Compilers
    - Write compiler for simple language
  - Networking
    - Implement and simulate network protocols

Course Perspective (Cont.)

- **This course is Programmer-Centric**
  - Purpose is to show how software really works
  - By understanding the underlying system,
    one can be more effective as a programmer
    - Better debugging
    - Better basis for evaluating performance
    - How multiple activities work in concert (e.g., OS and user programs)
  - Not just a course for dedicated hackers
    - What every programmer needs to know

Textbooks

- **Computer Systems: A Programmer's Perspective, 2nd Edition**
  - Randal E. Bryant and David R. O'Hallaron
  - Prentice-Hall, 2010
  - http://csapp.cs.cmu.edu
  - This book really matters for the course!
    - How to solve labs
    - Practice problems typical of exam problems



- **OS: Online book by Anderson & Dhalin**
- **A good C book (not required, maybe useful)**
  - C: A Reference Manual (Harbison and Steele)
  - The C Programming Language (Kernighan and Ritchie)
- **Linux (if you want a book)**
  - Linux Pocket Guide (Barrett – new edition this week!)

Course Components

- **3 lectures per week (~30 total)**
- **Written assignments (~3 or 4)**
  - Problems from text to solidify understanding
- **Labs (~4 or 5)**
  - Provide in-depth understanding (via practice) of an aspect of systems
- **Exams (midterm + final)**
  - Test your understanding of concepts and principles

- **Late policy: 4 total "late days"; at most 2 per assignment**
  - Save 'em for later!

Policies: Grading

- **Exams: midterm 15%, final 25% of total grade**
- **Written assignments: weighted according to effort**
  - We'll try to make these about the same
- **Labs assignments: weighted according to effort**
  - These will likely increase in weight as the quarter progresses
- **Grading:**
  - 25% written assignments
  - 35% lab assignments
  - 40% exams
- **Academic integrity: policy on course web – Read it!**
  - Do your own work – explain any unconventional action on your part
  - I trust you completely
  - I have no sympathy for trust violations – nor should you
  - Honest work is the most important feature of a university. Show respect for your colleagues and for yourself.

Welcome to CSE 410!

- **Let's have fun**
- **Let's learn – together**
- **Let's communicate**
- **Let's set the bar for a useful and interesting class**

- **Many thanks to the many instructors who have shared their lecture notes – I will be borrowing liberally through the quarter – they deserve all the credit, the errors are all mine**
  - UW: Gaetano Borriello, Luis Ceze (CSE 351)
  - CMU: Randy Bryant, David O'Halloran, Gregory Kesden, Markus Püschel
  - Harvard: Matt Welsh
  - UW: Ed Lazowska, Steve Gribble, Tom Anderson, John Zahorjan, Hank Levy (CSE 451)

Work to do

Fill out the office hour doodle on the course web

Read:

- **Ch. 1 (background)**
- **Sec. 2.1 (storage – we'll talk about this Wednesday)**
- **(Aside: we expect you to read things before class. Lectures won't present things from scratch.)**

If you're trying to add the course please sign up on the sheet before leaving today