INTRODUCTION

Jo, Heeseung

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

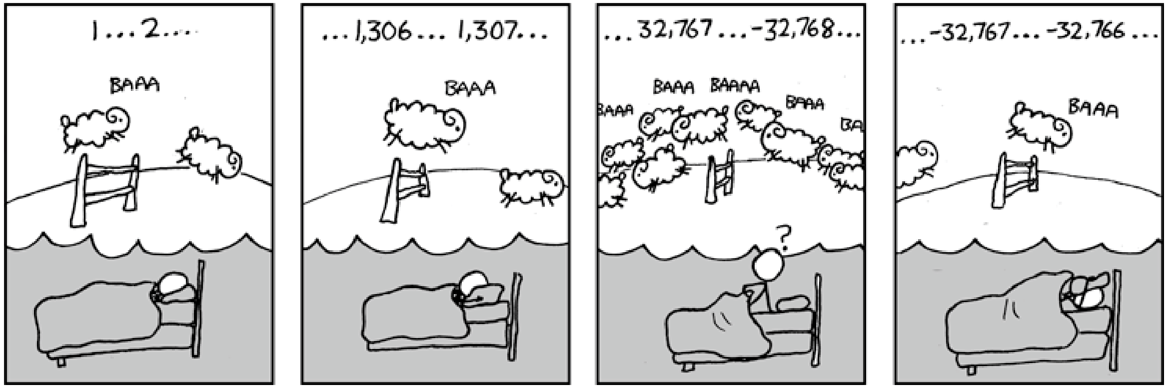
Useful outcomes

- Become more effective programmers
  - Able to find and eliminate bugs efficiently
  - Able to understand and tune for program performance
- Prepare for later "systems" classes
  - 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 * 40000 = 1600000000
  - 50000 * 50000 = -1794967296 (overflow)

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) --> ??

Source: xkcd.com/571

Code Security Example

```
/* 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);
}
```

Computer Arithmetic

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 Need 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 key to machine-level execution model

- 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

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

```
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 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];
}
```

volatile*: Don't be optimized by compiler*

```
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];
}
```

volatile*: Don't be optimized by compiler*

```
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, Ruby or ML
- Understand what possible interactions may occur
- Use or develop tools to detect referencing errors (e.g. Valgrind)

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];
}
```

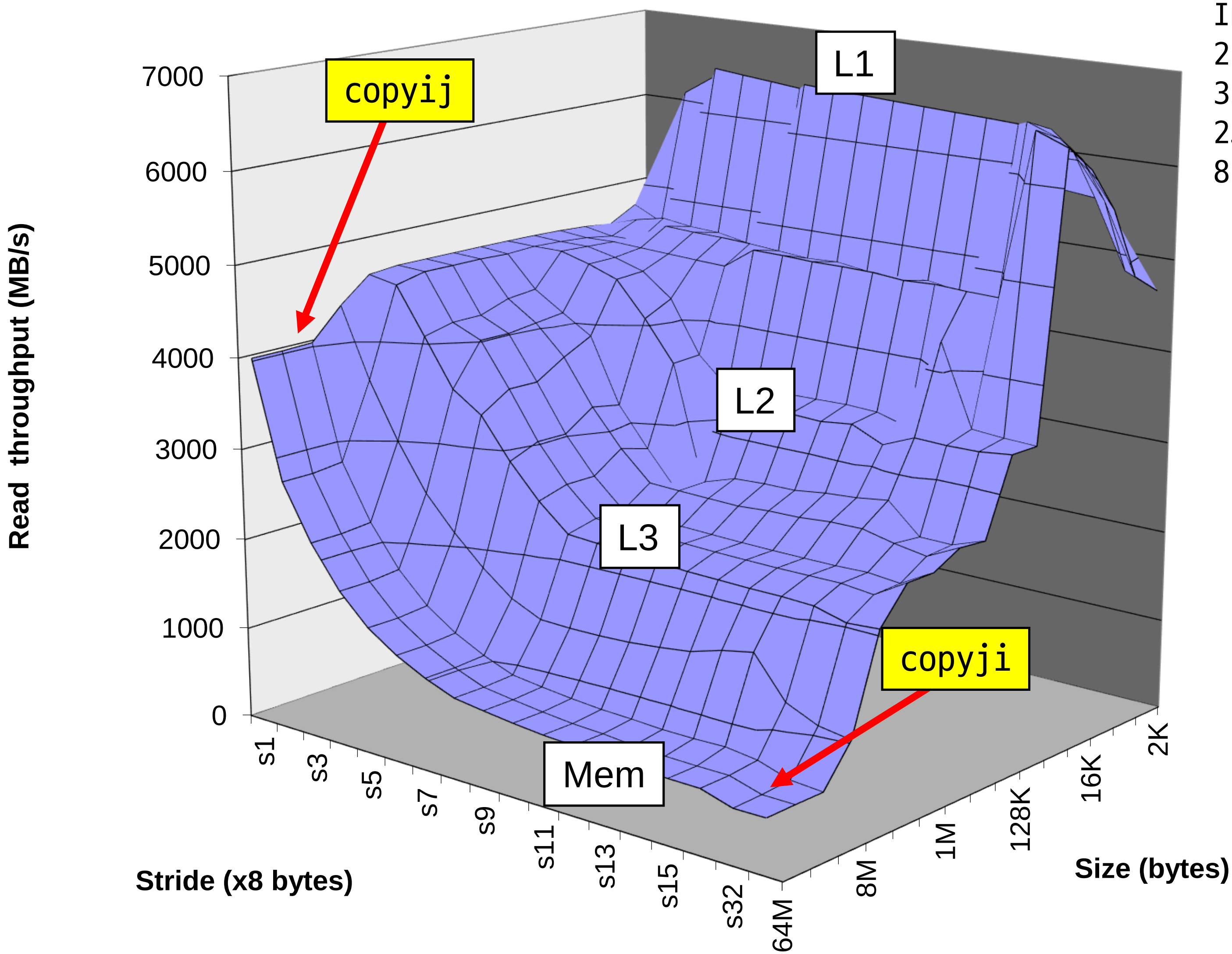
```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

Need to understand hierarchical memory organization

Performance depends on access patterns

- Including how step through multi-dimensional array

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

Must optimize at multiple levels: algorithm, data representations, procedures, and loops

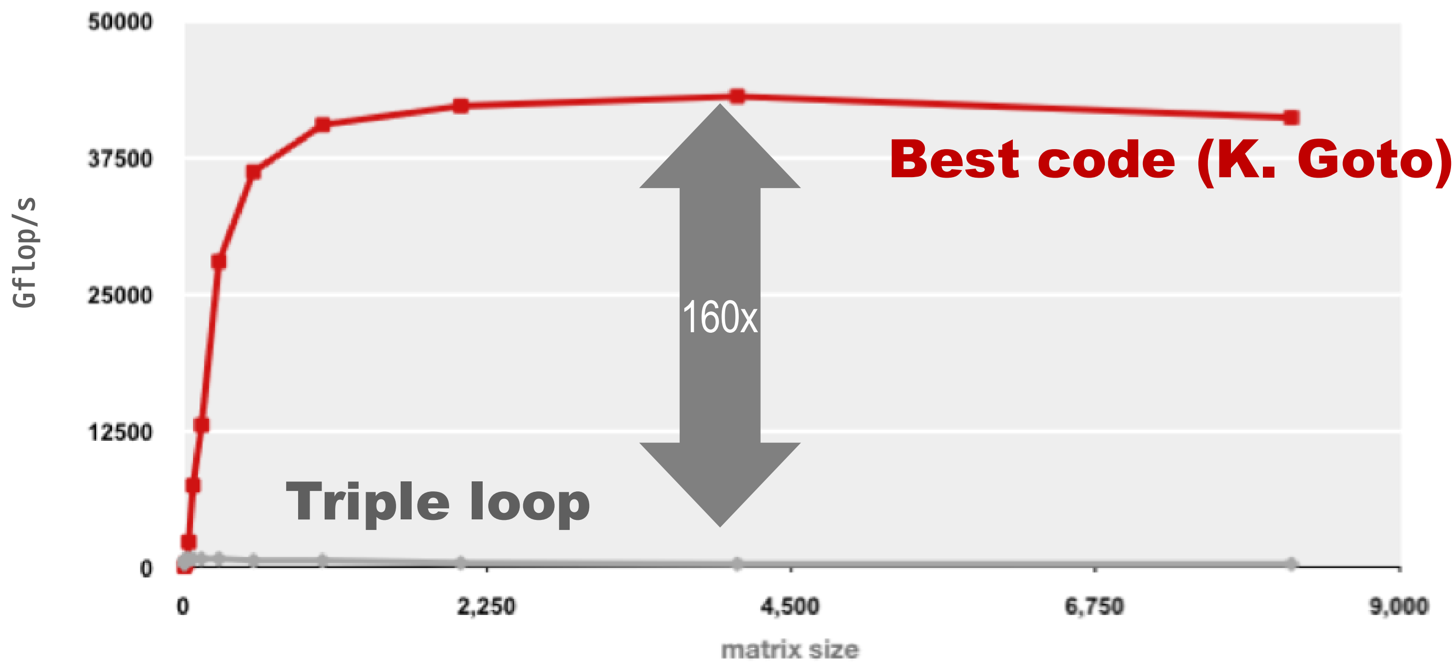
- Easily see 10:1 performance range depending on how code written

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)



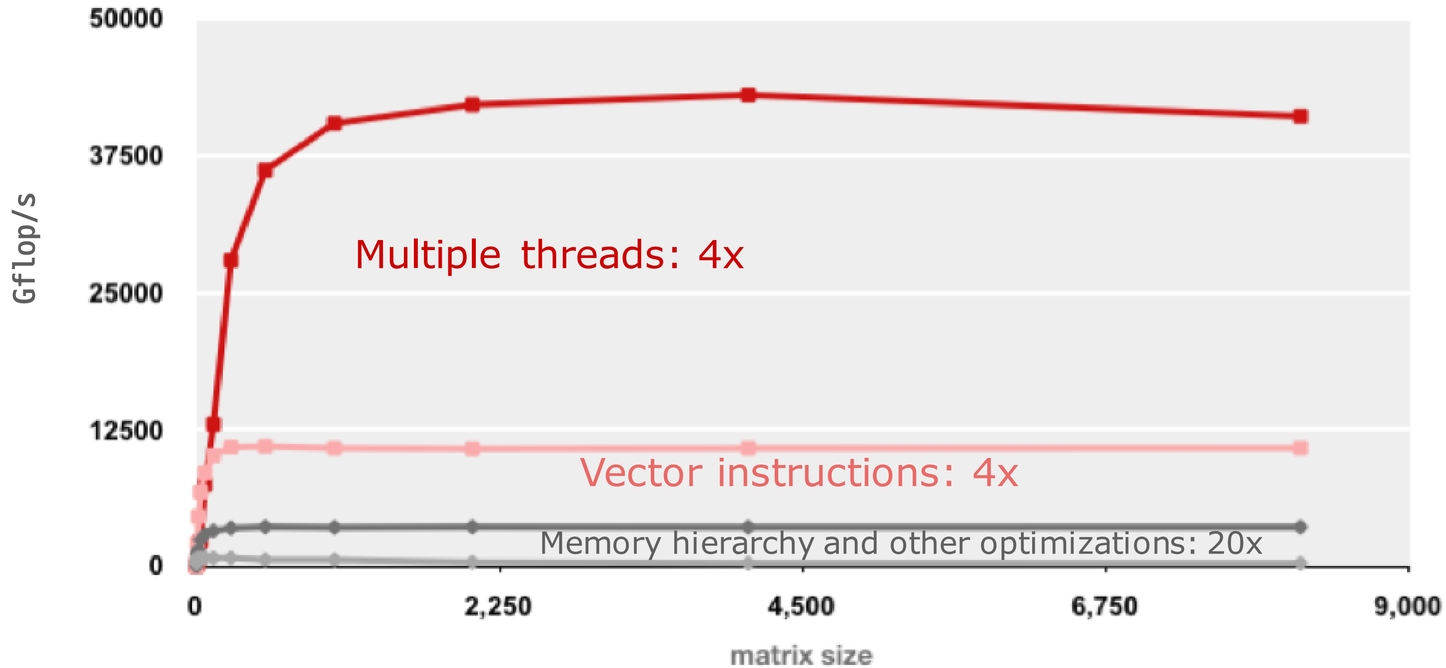
Standard desktop computer, vendor compiler, using optimization flags

- 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 (double precision)



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

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