15-213

"The Class That Gives CMU Its Zip!"

Introduction to Computer Systems

Randal E. Bryant

January 15, 2008

Topics:

- Theme
- Five great realities of computer systems
- How this fits within CS curriculum

Course Theme

- Abstraction is good, but don't forget reality!

Most CS courses emphasize abstraction

- Abstract data types
- Asymptotic analysis

These abstractions have limits

- Especially in the presence of bugs
- Need to understand underlying implementations

Useful outcomes

- Become more effective programmers
  - Able to find and eliminate bugs efficiently
  - Able to tune program performance
- Prepare for later "systems" classes in CS & ECE
  - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems

Great Reality #1

Int's are not Integers, Float's are not Reals

Examples

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

Computer Arithmetic

Does not generate random values

- Arithmetic operations have important mathematical properties

Cannot assume "usual" 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 program in assembly

- Compilers are much better & more patient than you are

Understanding assembly key to machine-level execution model

- Behavior of programs in presence of bugs
  - High-level language model breaks down
- Tuning program performance
  - 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 required by procedure
  - In units of clock cycles

```
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 un-physical 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

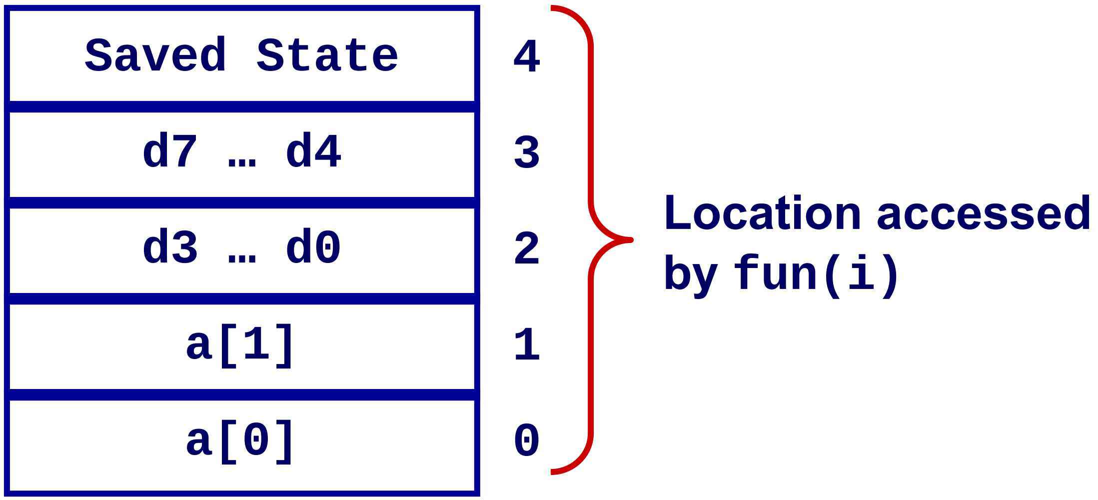
- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

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];
}
```

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

Referencing Bug Explanation



- C does not implement bounds checking
- Out of range write can affect other parts of program state

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 ML
- Understand what possible interactions may occur
- Use or develop tools to detect referencing errors

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];
}
```

59,393,288 clock cycles

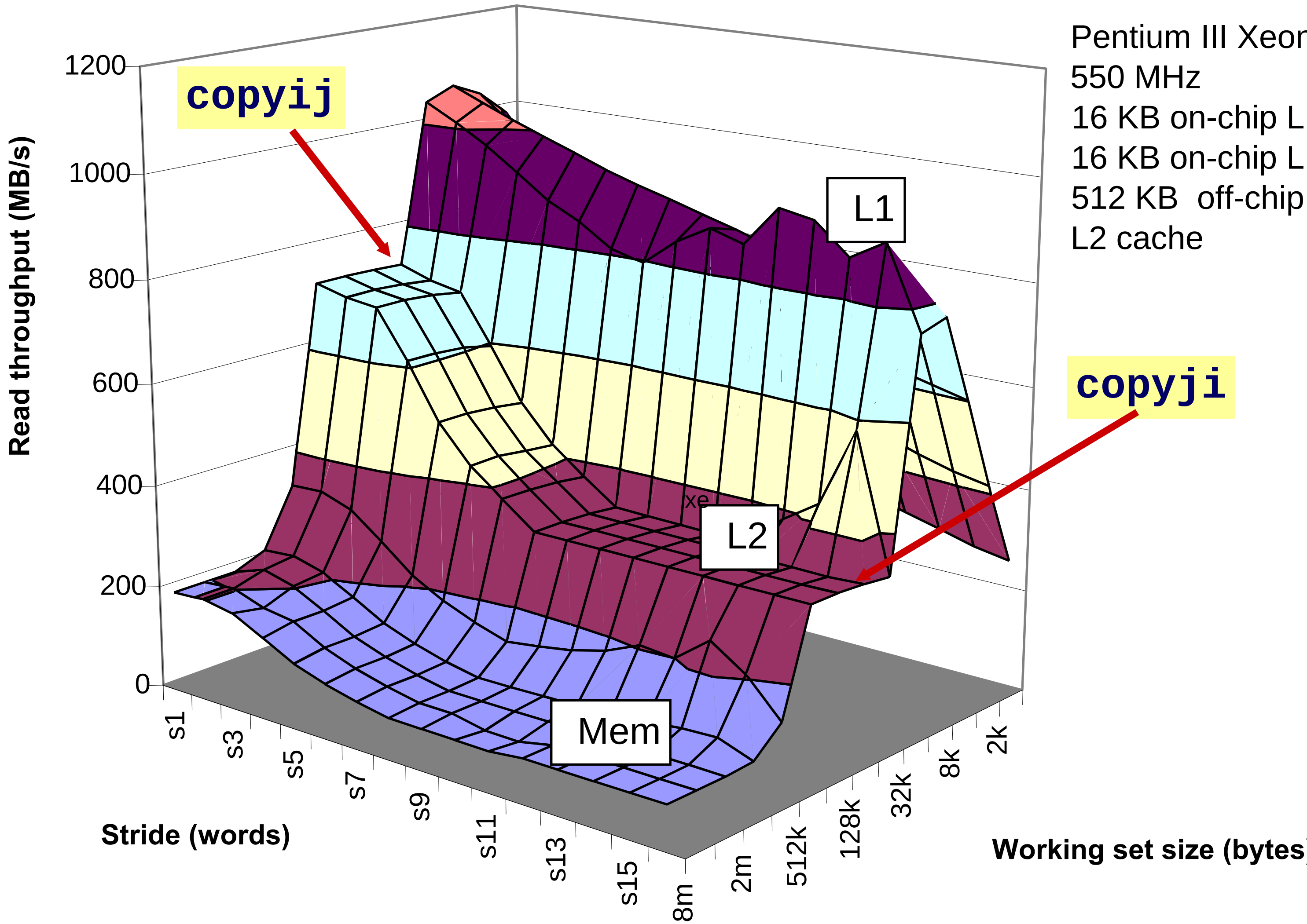
1,277,877,876 clock cycles

21.5 times slower!

(Measured on 2GHz Intel Pentium 4)

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

The Memory Mountain



Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

Great Reality #4

There's more to performance than asymptotic complexity

Constant factors matter too!

- 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

Code Performance Example

```
/* Compute product of array elements */
double product(double d[], int n)
{
    double result = 1;
    int i;
    for (i = 0; i < n; i++)
       result = result * d[i];
    return result;
}
```

- Multiply all elements of array
- Performance on class machines: ~7.0 clock cycles per element
  - Latency of floating-point multiplier

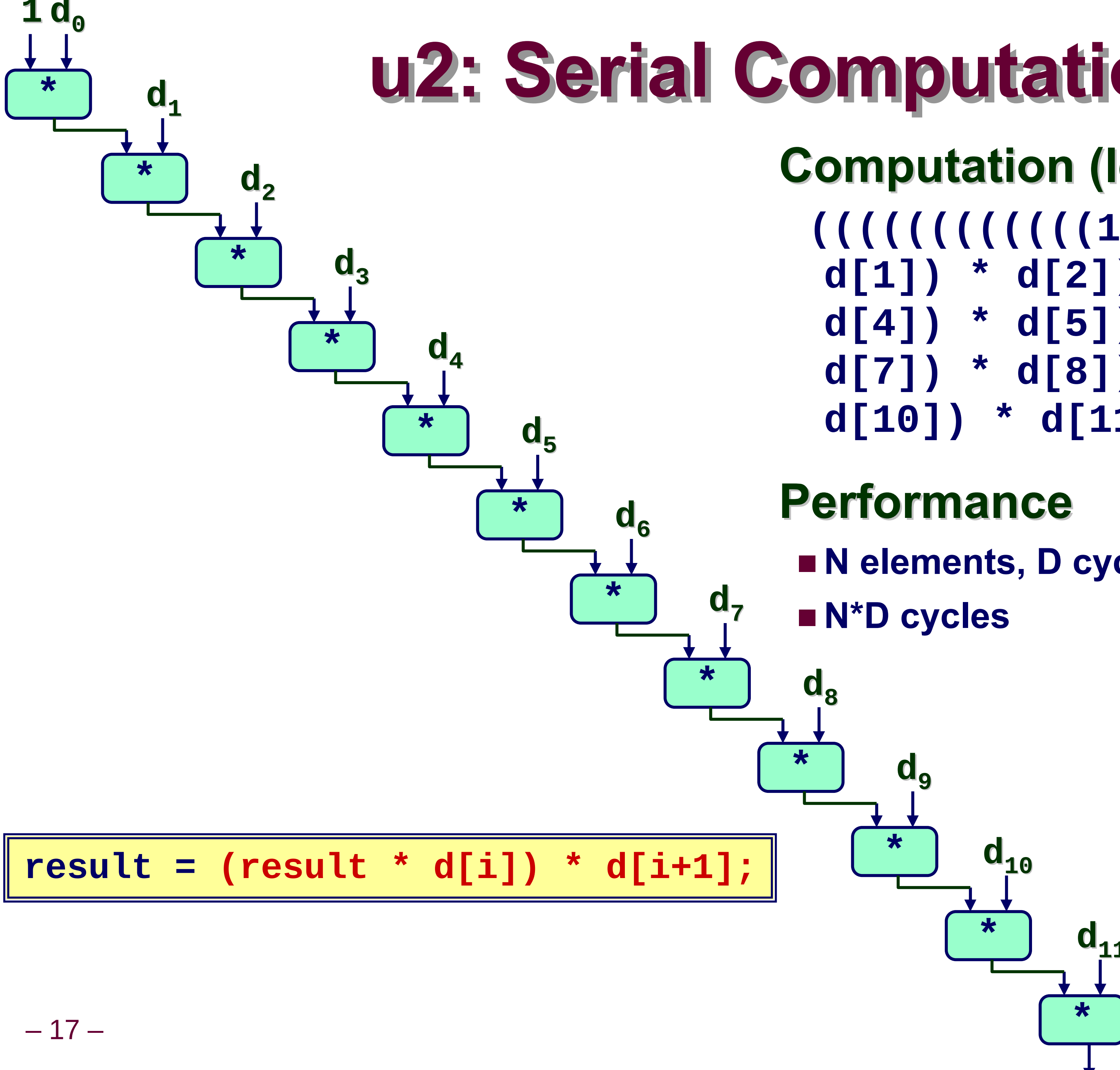
Loop Unrollings

```
/* Unroll by 2.  Assume n is even */
double product_u2(double d[], int n)
{
  double result = 1;
  int i;
  for (i = 0; i < n; i+=2)
    result = (result * d[i]) * d[i+1];
  return result;
}
```

```
/* Unroll by 2.  Assume n is even */
double product_u2r(double d[], int n)
{
  double result = 1;
  int i;
  for (i = 0; i < n; i+=2)
    result = result * (d[i] * d[i+1]);
  return result;
}
```

- **Do two loop elements per iteration**
  - **Reduces overhead**
- **Cycles per element:**
  - **u2: 7.0**
  - **u2r: 3.6**

u2: Serial Computation



Computation (length=12)

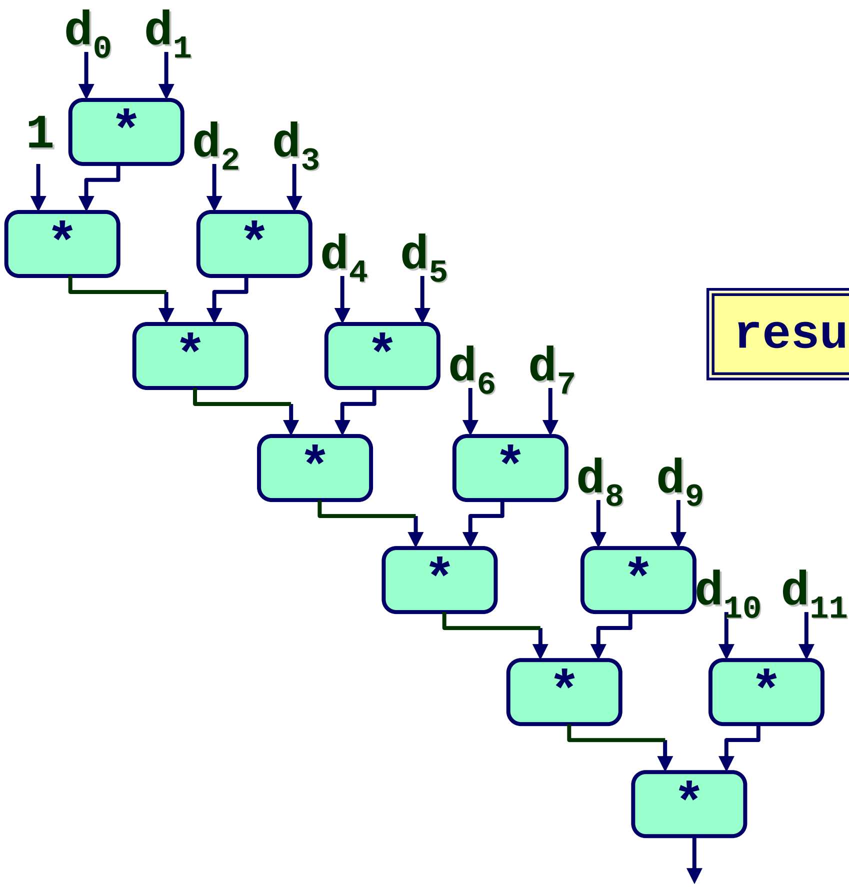
```
((((((((((((1 * d[0]) *
 d[1]) * d[2]) * d[3]) *
 d[4]) * d[5]) * d[6]) *
 d[7]) * d[8]) * d[9]) *
 d[10]) * d[11])
```

Performance

- N elements, D cycles/operation
- N*D cycles

```
result = (result * d[i]) * d[i+1];
```

u2r: Reassociated Computation



Performance

- N elements, D cycles/operation
- (N/2+1)*D cycles

```
result = result * (d[i] * d[i+1]);
```

Great Reality #5

Computers do more than execute programs

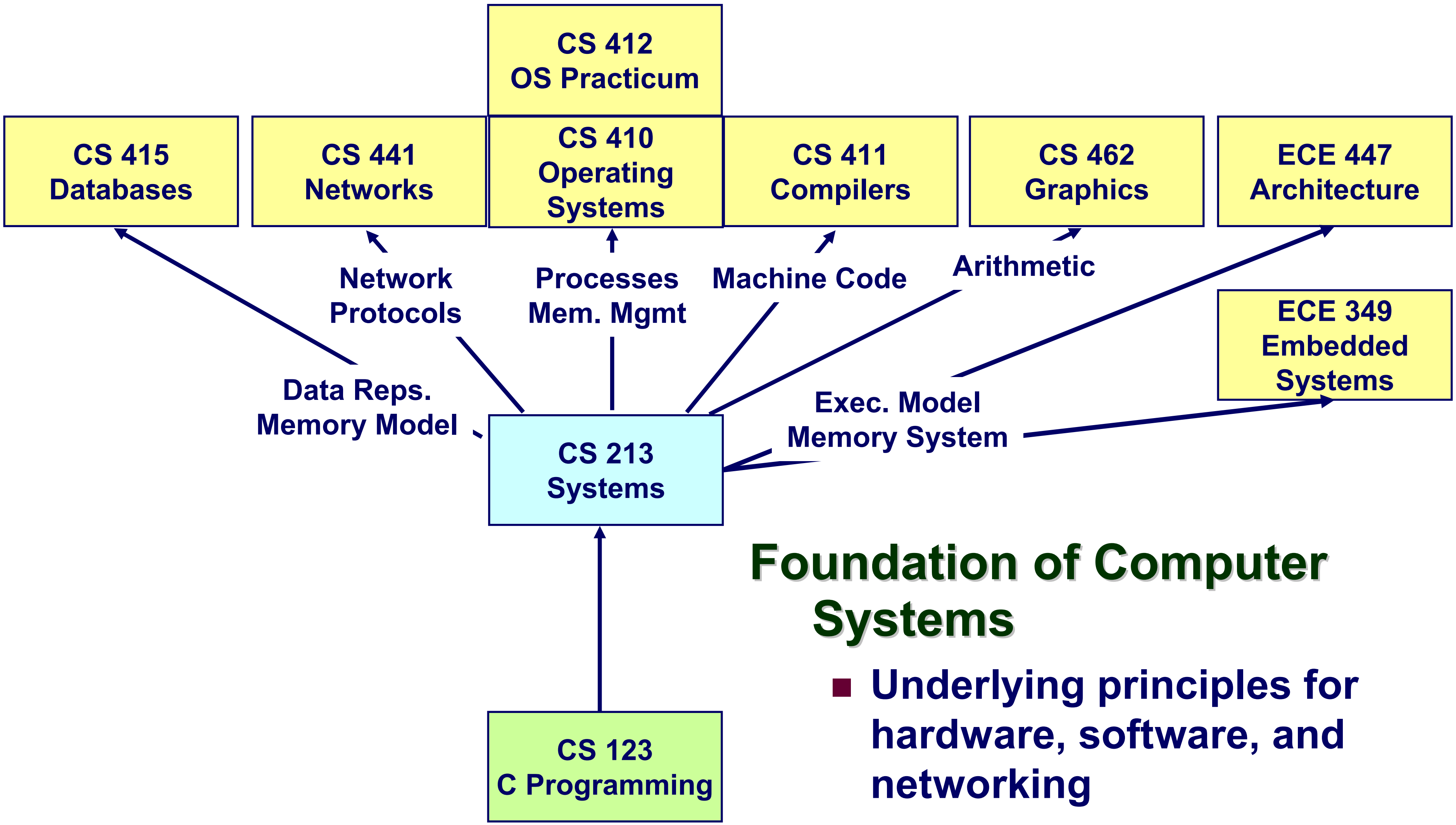
They need to get data in and out

- **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**

Role within Curriculum



Foundation of Computer Systems

- Underlying principles for hardware, software, and networking

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

Course Perspective (Cont.)

Our Course is 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
  - 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
- Cover material in this course that you won’t see elsewhere