CENG331 - Computer Organization

Introduction

Fall 2020

Instructor:

Murat Manguoglu (Sections 1-2)

Unless otherwise noted adapted from slides of the textbook: http://csapp.cs.cmu.edu/

Introduction

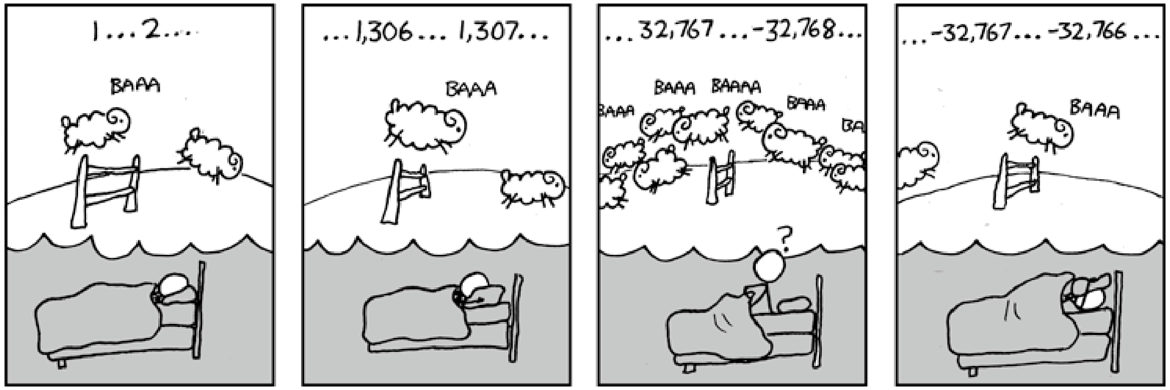
- Course theme
- Five realities

Course Theme:
Abstraction Is Good But Reality Matters

- **Most CENG courses emphasize abstractions**
  - Abstract data types
  - Asymptotic analysis
- **These abstractions have limits**
  - Especially in the presence of bugs, performance constraints
  - Need to understand details of underlying implementations
- **Useful outcomes of CENG331**
  - Become more effective programmers and computer scientists
    - *Able to find and eliminate bugs efficiently*
    - *Able to **understand** and **tune** your code for **performance***
  - Prepare for later "systems" classes in CS/CENG
    - *Intro. to Parallel Computing, Compilers, Operating Systems, Networks, Embedded Systems, etc.*

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 --> ??
- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Ints: Yes!
  - Floats:
    - (1e20 + -1e20) + 3.14 --> 3.14
    - 1e20 + (-1e20 + 3.14) --> ??

Source: xkcd.com/571

Computer Arithmetic

- ~~Does not generate random values~~ ("most of the time")
  - Arithmetic operations have important mathematical properties
  - Random errors, bit flips, etc. are possible on large scale computers or quantum computers!
- **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

Reality #2: You've Got to Know Assembly (and the processor)

- **Chances are, you'll never write programs in assembly**
  - Compilers are much better & more patient than you
- **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: you can never only rely on compiler optimizations, usually you can do much better!
    - Understand optimizations done / not done by the compiler (*or can not be done*)
    - 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!

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

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  →    3.14
fun(1)  →    3.14
fun(2)  →    3.1399998664856
fun(3)  →    2.00000061035156
fun(4)  →    3.14
fun(6)  →    Segmentation fault
```

- Result is system specific

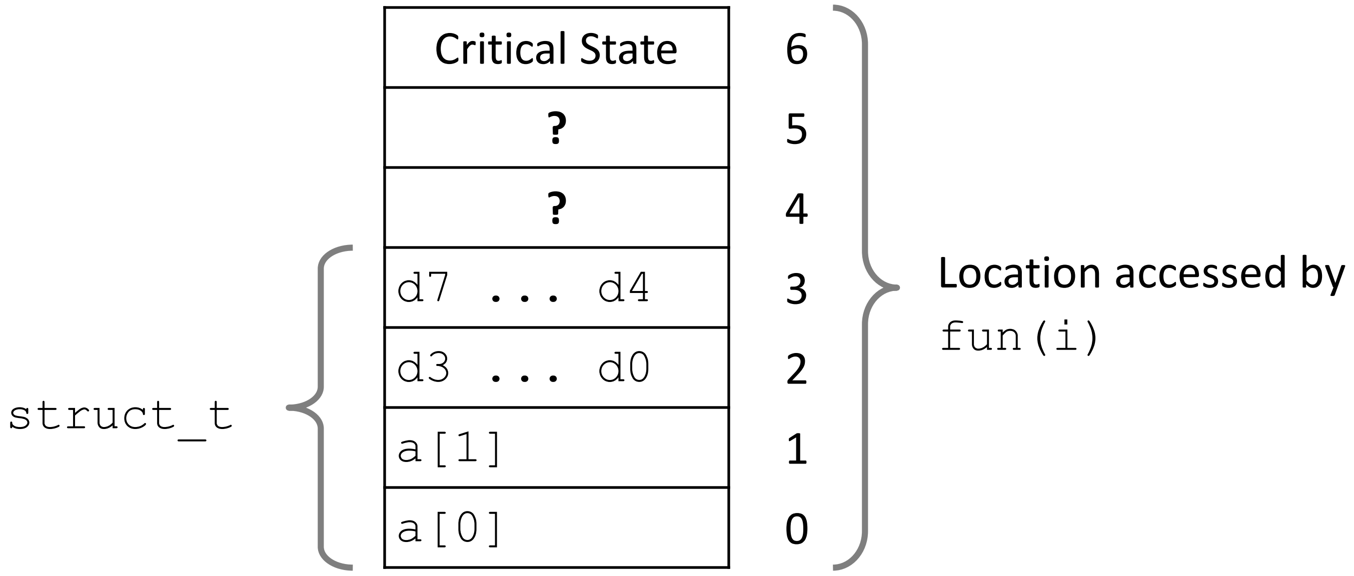
volatile keyword indicates that a value may change between different accesses, even if it does not appear to be modified. This keyword prevents an optimizing compiler from optimizing away subsequent reads or writes and thus incorrectly reusing a stale value or omitting writes

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  →   3.14
fun(1)  →   3.14
fun(2)  →   3.1399998664856
fun(3)  →   2.00000061035156
fun(4)  →   3.14
fun(6)  →   Segmentation fault
```

Explanation:



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

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

Memory System Performance Example

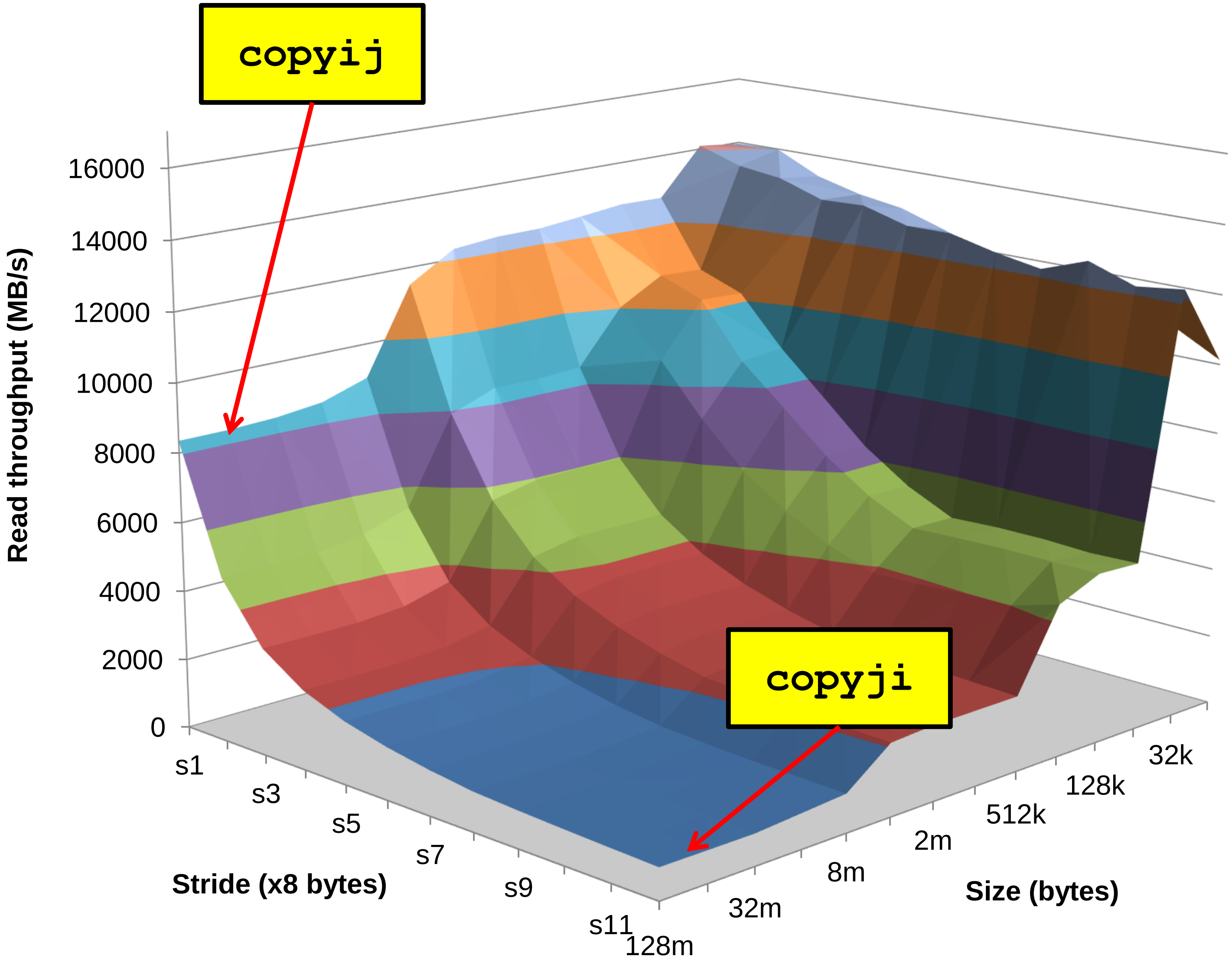
```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
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];
}
```

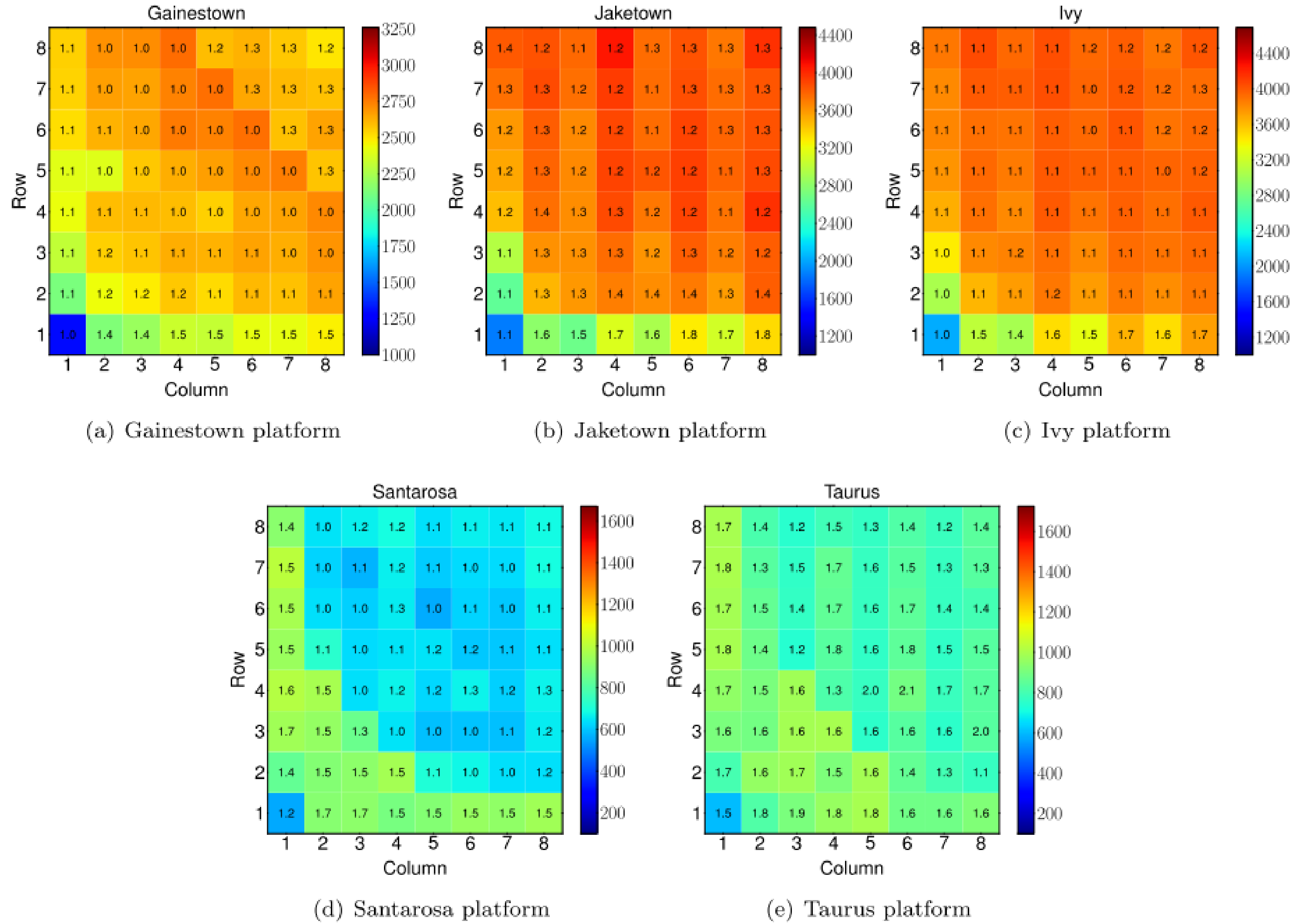
4.3ms **2.0 GHz Intel Core i7 Haswell** **81.8ms**

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

Why The Performance Differs



Another performance figure (matrix mult.)



(a) Gainestown platform (b) Jaketown platform (c) Ivy platform

(d) Santarosa platform (e) Taurus platform

Figure 8: SpMV Performance Profiles for optimized-BCSR on a dense matrix on a single core. On each platform, each square is an $r \times c$ implementation, $1 \leq r, c \leq 8$, colored by its performance in MFlops/sec, and labeled by its speedup over OSKI. The top of the speed range for each platform is the performance bound from the Roofline model, in the last column of Table 5.

Source: Byun, J. H., Lin, R., Yelick, K. A., & Demmel, J. (2012). Autotuning sparse matrix-vector multiplication for multicore. *EECS, UC Berkeley, Tech. Rep*

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

CENG435 Data Communications and Networking

CENG478 Introduction to Parallel Computing

Thank you !