
WELCOME
TO
15-213
14-513
18-213
18-613
15-513

Today

- Cache organization and operation
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

The Memory Mountain

- **Read throughput (read bandwidth)**
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain: Measured read throughput as a function of spatial and temporal locality.**
  - Compact way to characterize memory system performance.

Memory Mountain Test Function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride",
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

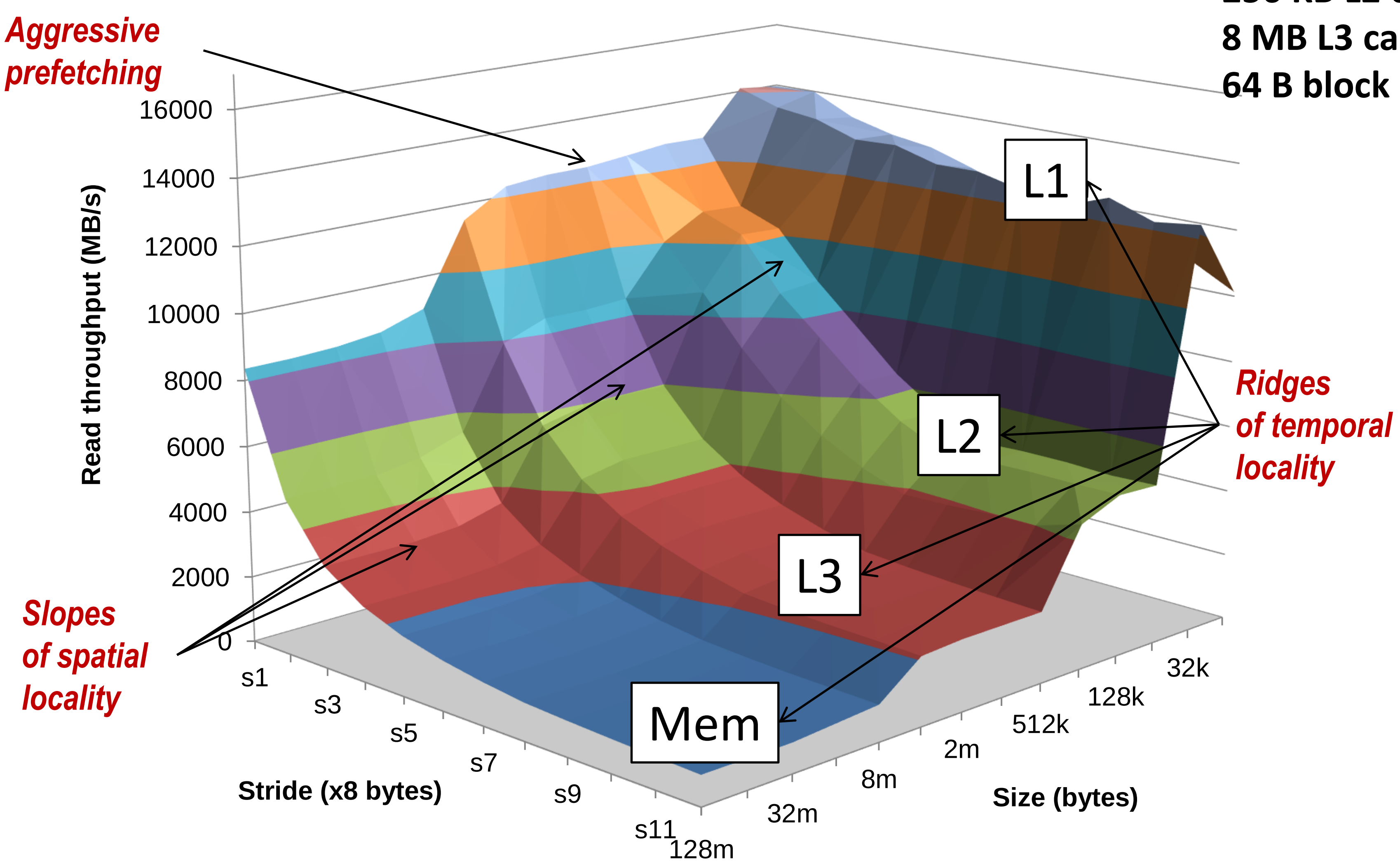
For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

2. Call `test()` again and measure the read throughput(MB/s)

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Matrix Multiplication Example

- **Description:**
  - Multiply *N* x *N* matrices
  - Matrix elements are doubles (8 bytes)
  - O(N^3) total operations
  - *N* reads per source element
  - *N* values summed per destination
    - but may be able to hold in register

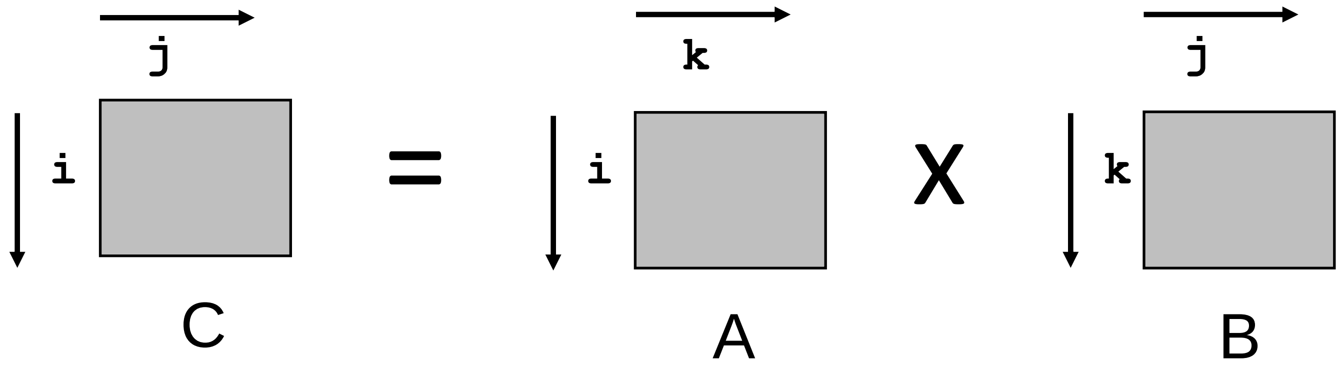
Variable ***sum*** *held in register*

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- **Analysis Method:**
  - Look at access pattern of inner loop



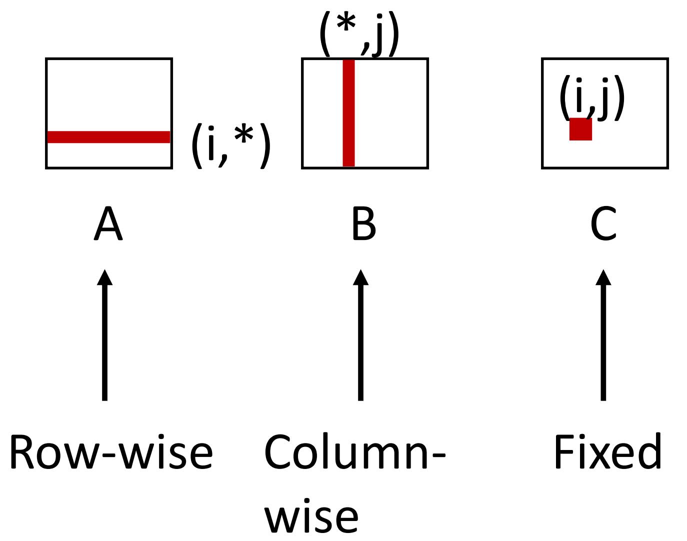
Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations
- **Stepping through columns in one row:**
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- **Stepping through rows in one column:**
  - ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (`ijk`)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}                                matmult/mm.c
```

Inner loop:



Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

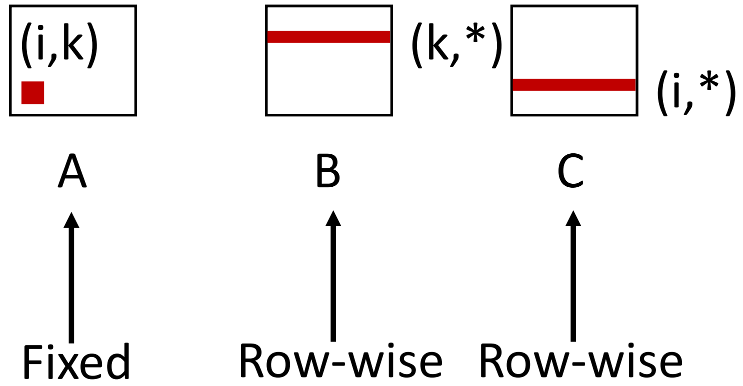
Block size = 32B (four doubles)

Matrix Multiplication (`kij`)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c

Inner loop:



Miss rate for inner loop iterations:

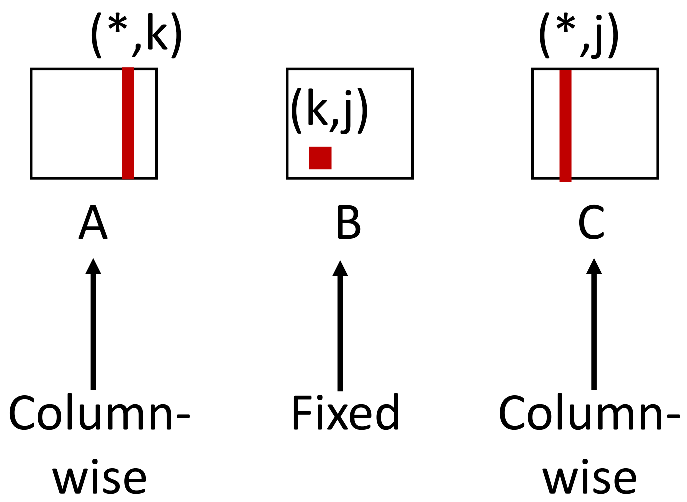
A	B	C
0.0	0.25	0.25

Block size = 32B (four doubles)

Matrix Multiplication (`jki`)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
                         matmult/mm.c
```

Inner loop:



Miss rate for inner loop iterations:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
  }
}
```

ijk (& jik):

- 2 loads, 0 stores
- avg misses/iter = **1.25**

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

kij (& ikj):

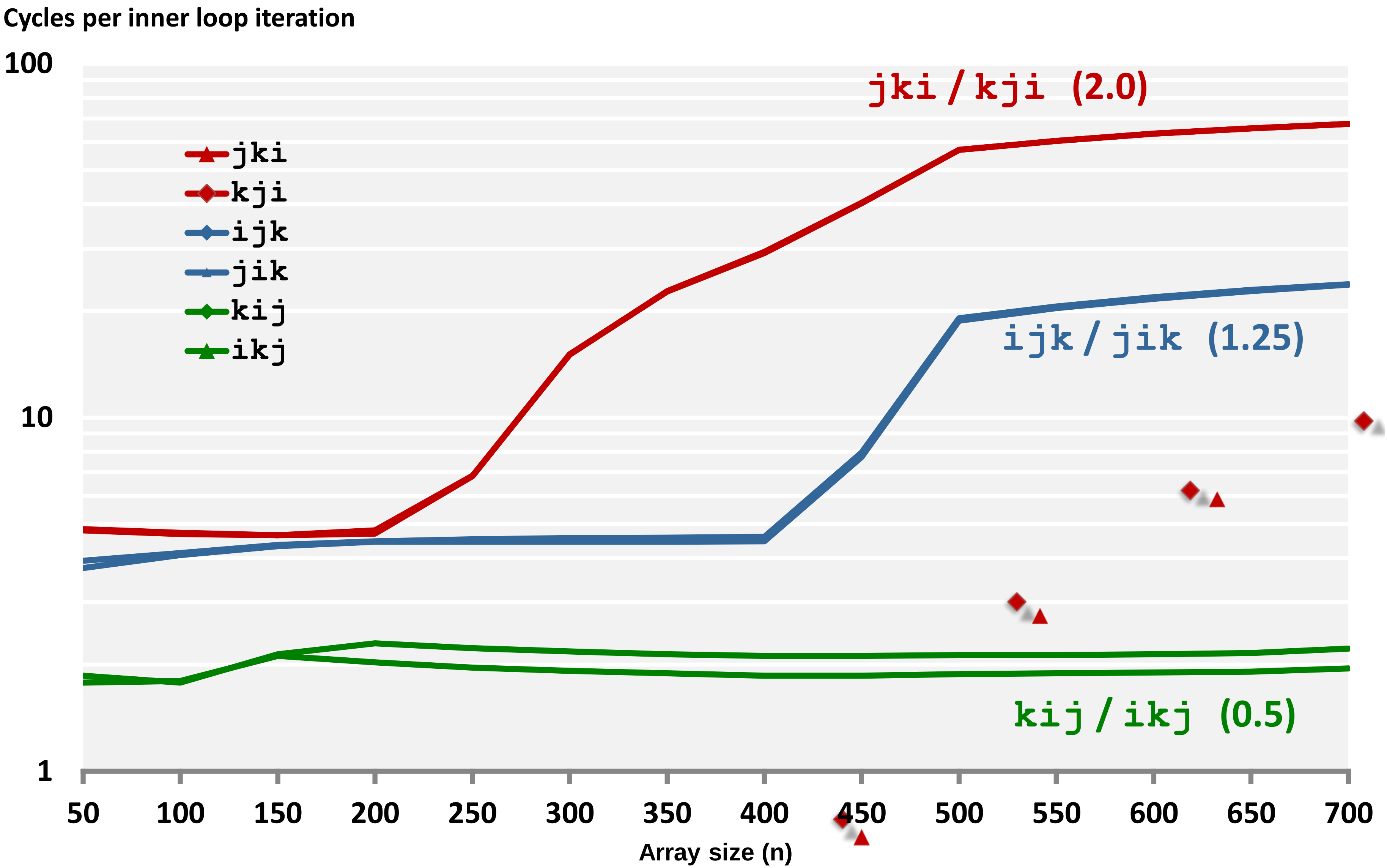
- 2 loads, 1 store
- avg misses/iter = **0.5**

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki (& kji):

- 2 loads, 1 store
- avg misses/iter = **2.0**

Core i7 Matrix Multiply Performance



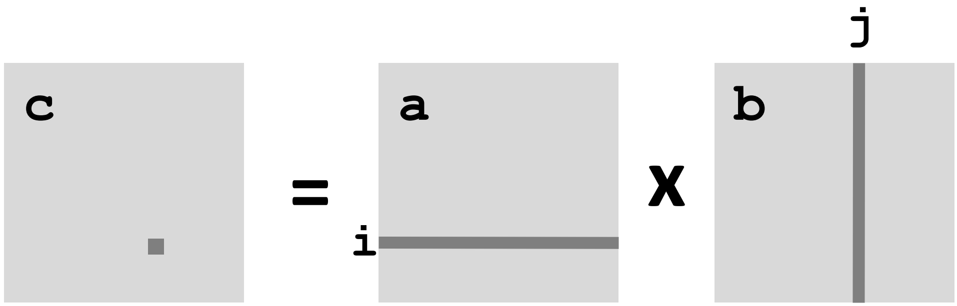
Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

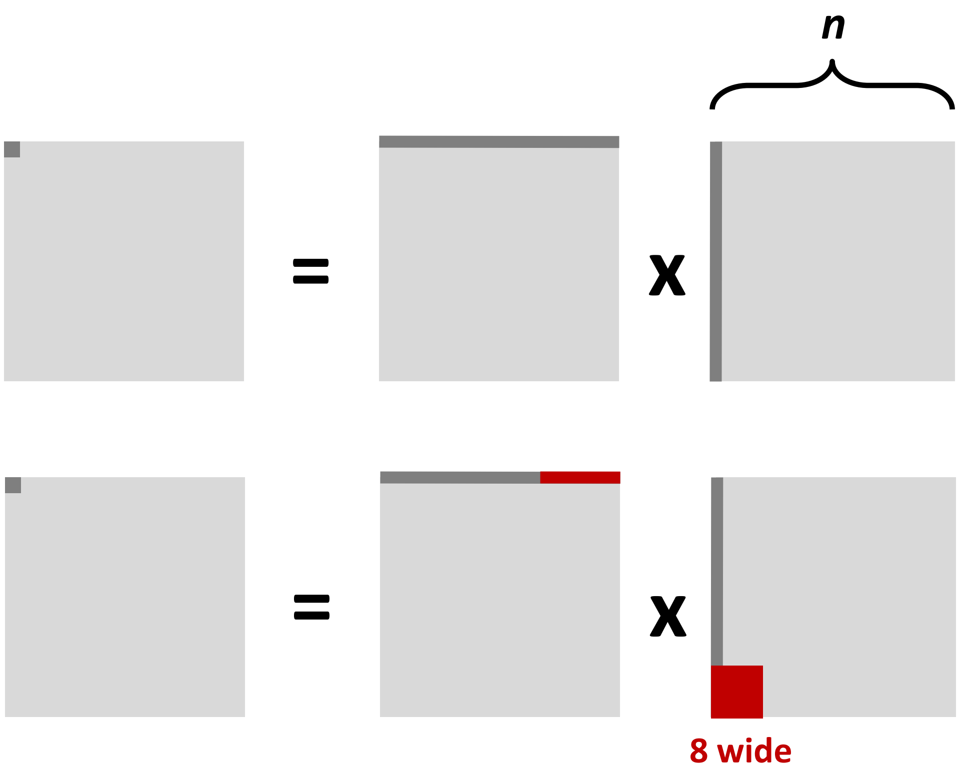


Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache line = 8 doubles
  - Cache size C << *n* (much smaller than *n*)

- **First iteration:**
  - $n/8 + n = 9n/8$ misses

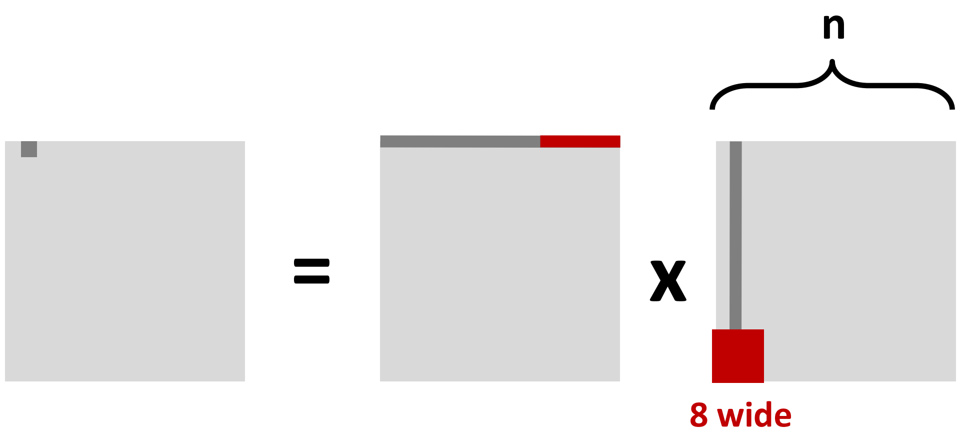
  - Afterwards in cache: (schematic)



Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache line = 8 doubles
  - Cache size C << *n* (much smaller than *n*)

- **Second iteration:**
  - Again:
    $n/8 + n = 9n/8$ misses



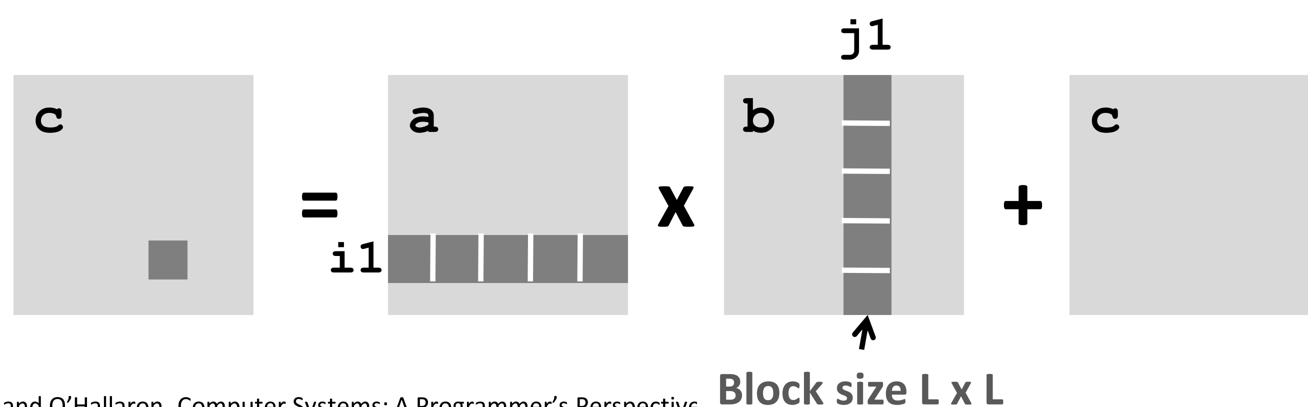
- **Total misses:**
  - $9n/8\ n^2 = (9/8)\ n^3$

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=L)
        for (j = 0; j < n; j+=L)
            for (k = 0; k < n; k+=L)
                /* L x L mini matrix multiplications */
                for (i1 = i; i1 < i+L; i1++)
                    for (j1 = j; j1 < j+L; j1++)
                        for (k1 = k; k1 < k+L; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

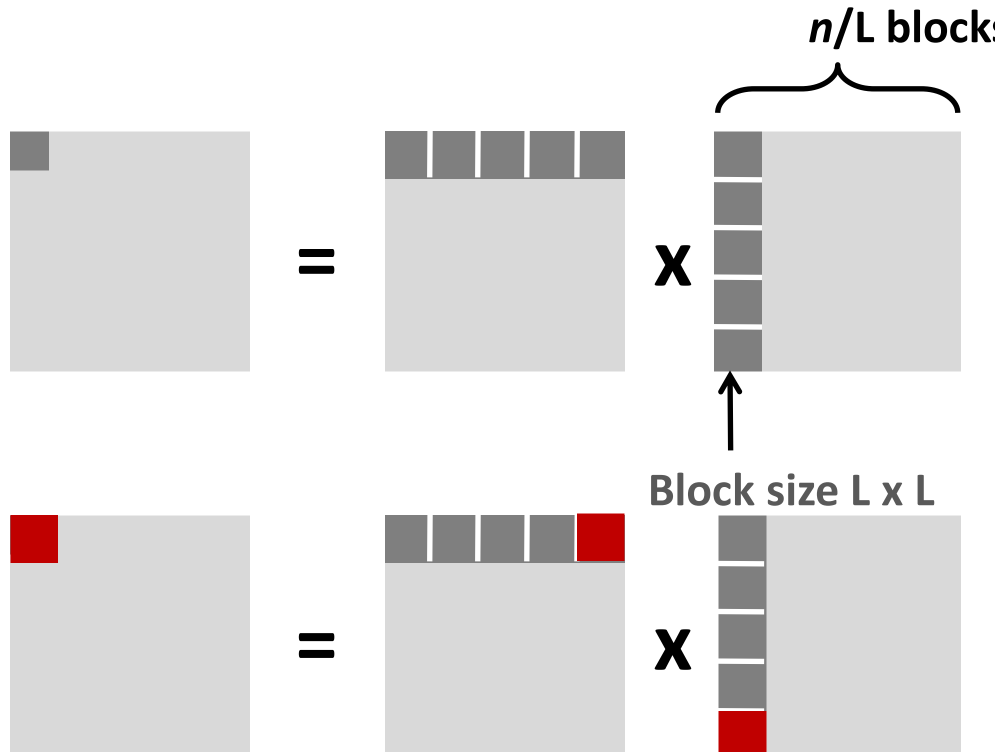
matmult/bmm.c



Cache Miss Analysis

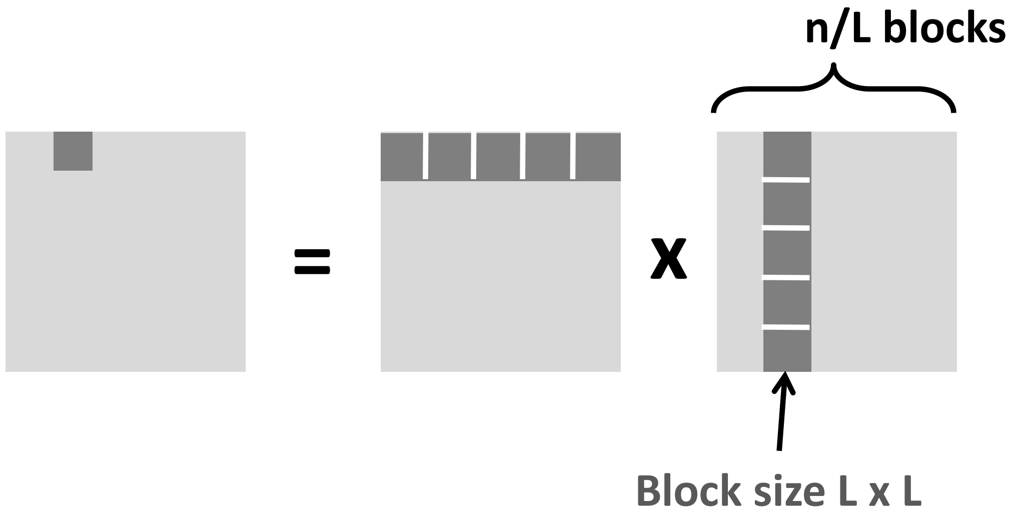
- **Assume:**
  - Cache line = 8 doubles. Blocking size $L \geq 8$
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks ■ fit into cache: $3L^2 < C$

- **First (block) iteration:**
  - **Misses per block:** $L^2/8$
  - **Blocks per Iteration:** $2n/L$ (omitting matrix c)
  - **Misses per Iteration:** $2n/L \times L^2/8 = nL/4$
  - Afterwards in cache (schematic)



Cache Miss Analysis

- **Assume:**
  - Cache line = 8 doubles. Blocking size $L \geq 8$
  - Cache size $C \ll n$ (much smaller than n)
  - Three blocks fit into cache: $3L^2 < C$
- **Second (block) iteration:**
  - Same misses as first iteration
  - $2n/L \times L^2/8 = nL/4$



- **Total misses:**
  - $nL/4$ misses per iteration x $(n/L)^2$ iterations = $n^3/(4L)$ misses

Blocking Summary

- **No blocking: (9/8) n^3 misses**
- **Blocking: (1/(4L)) n^3 misses**

- **Use largest block size L, such that L satisfies $3L^2 < C$**
  - Fit three blocks in cache! Two input, one output.

- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

Cache Summary

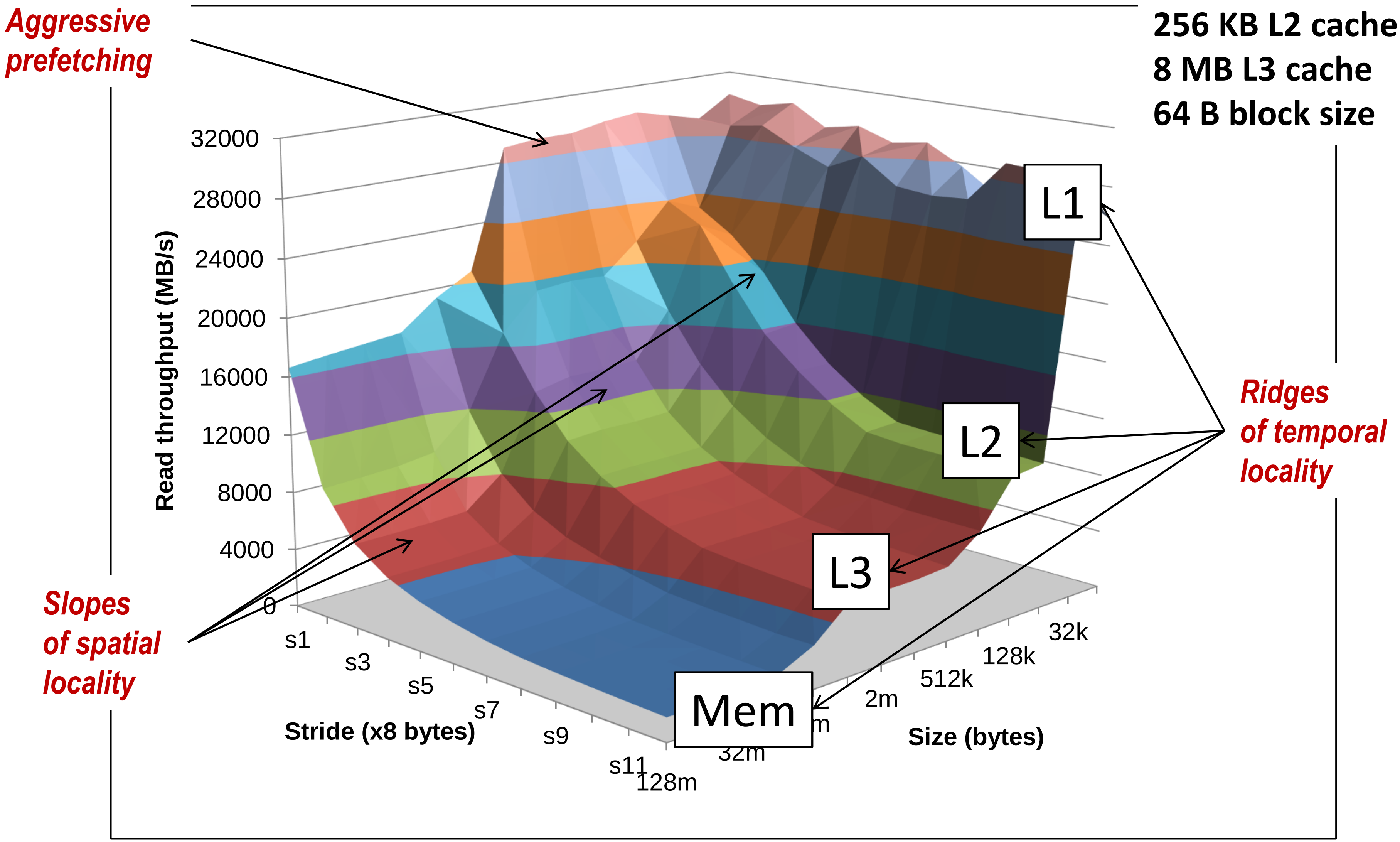
- **Cache memories can have significant performance impact**

- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.

Supplemental slides

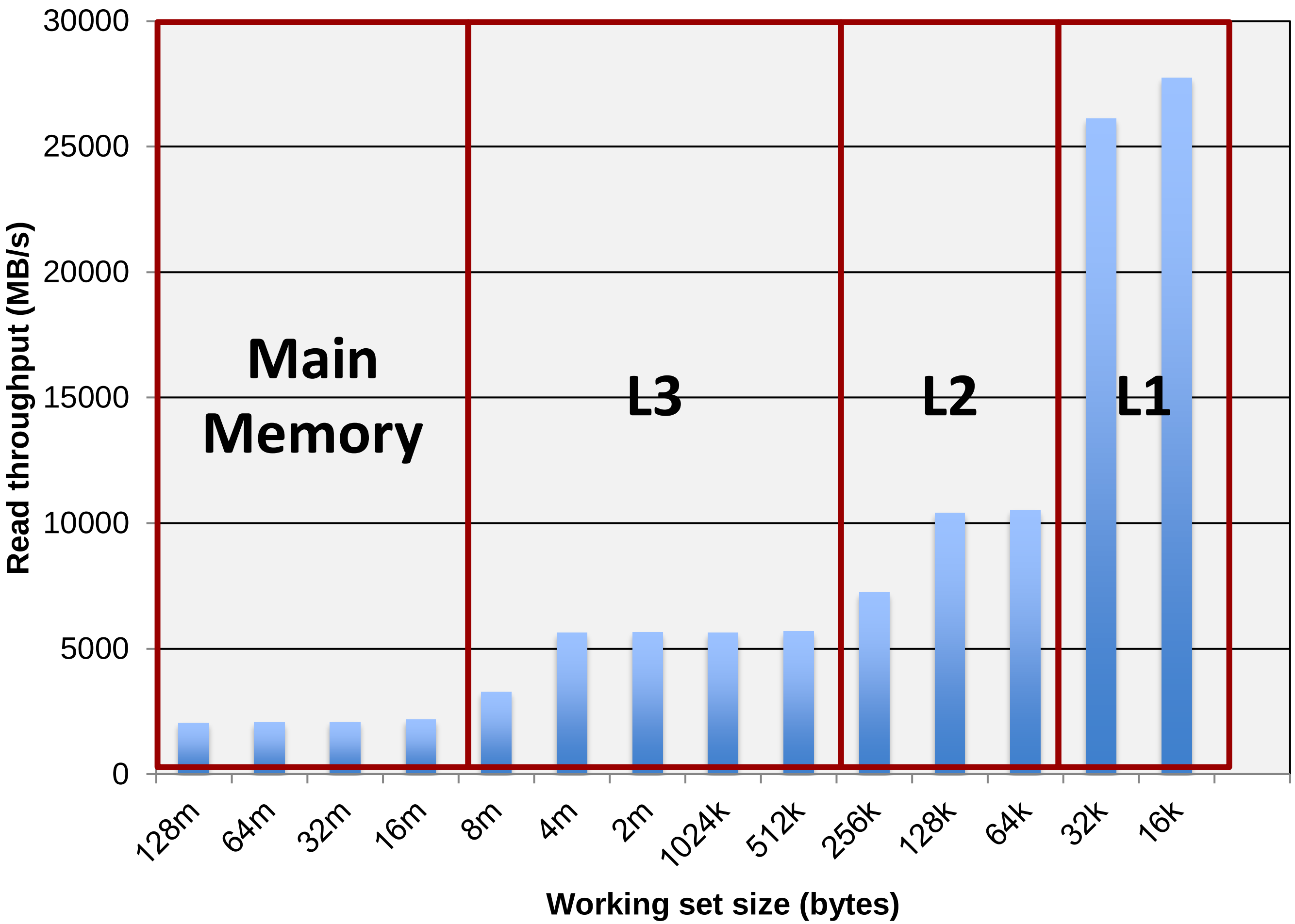
The Memory Mountain

Core i5 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Cache Capacity Effects from Memory Mountain

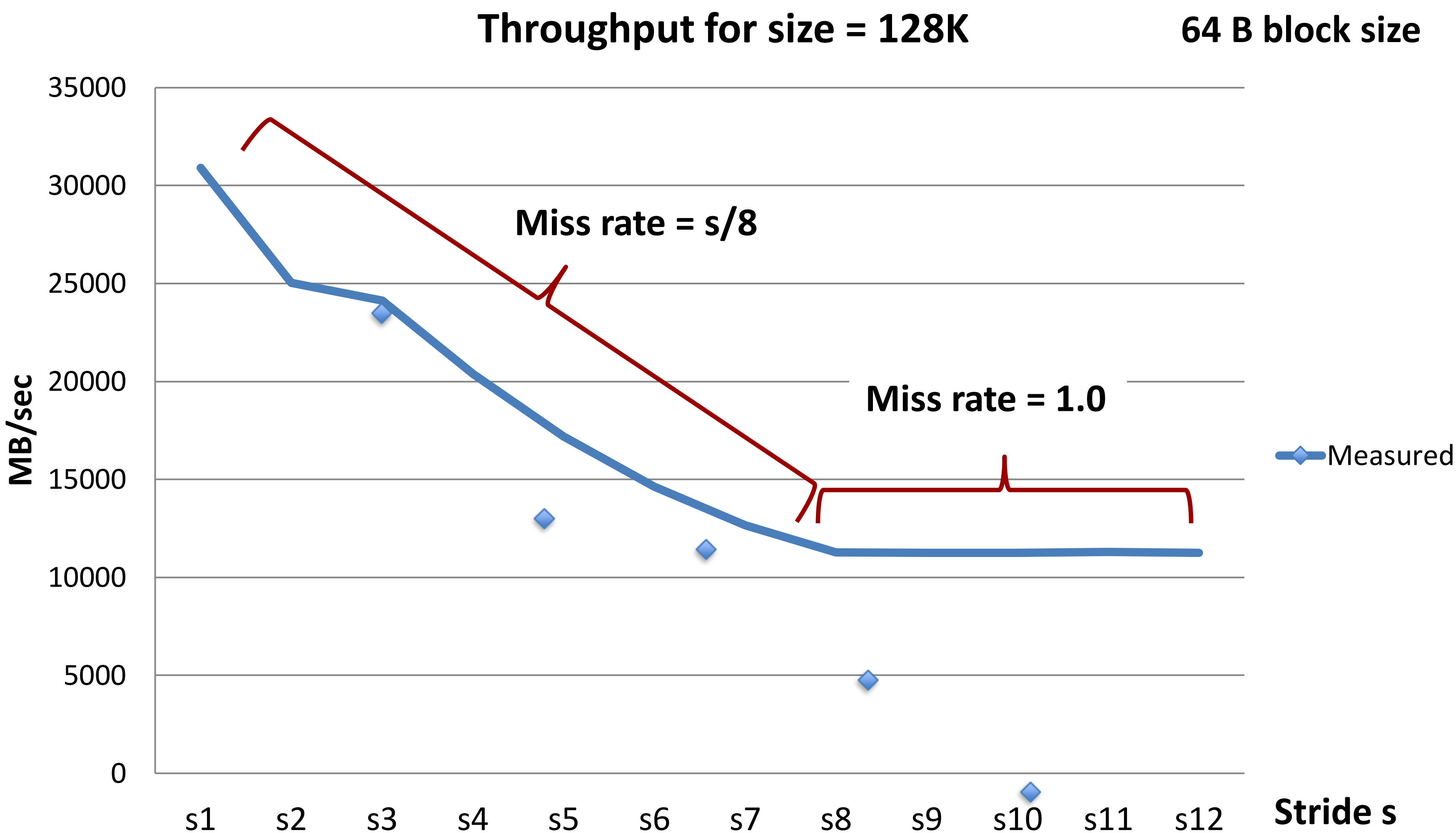
Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Slice through memory mountain with stride=8

Cache Block Size Effects from Memory Mountain

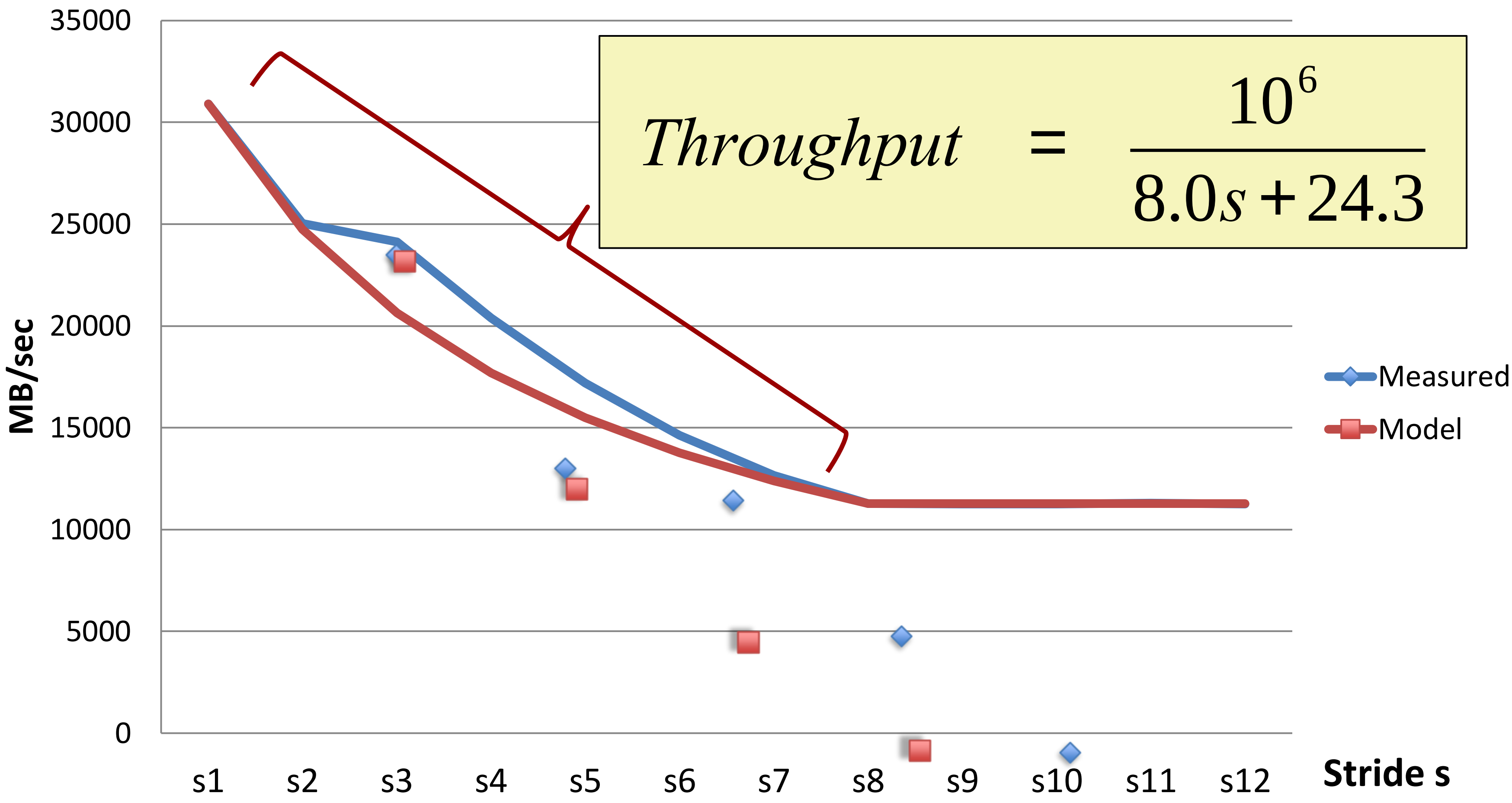
Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Modeling Block Size Effects from Memory Mountain

Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

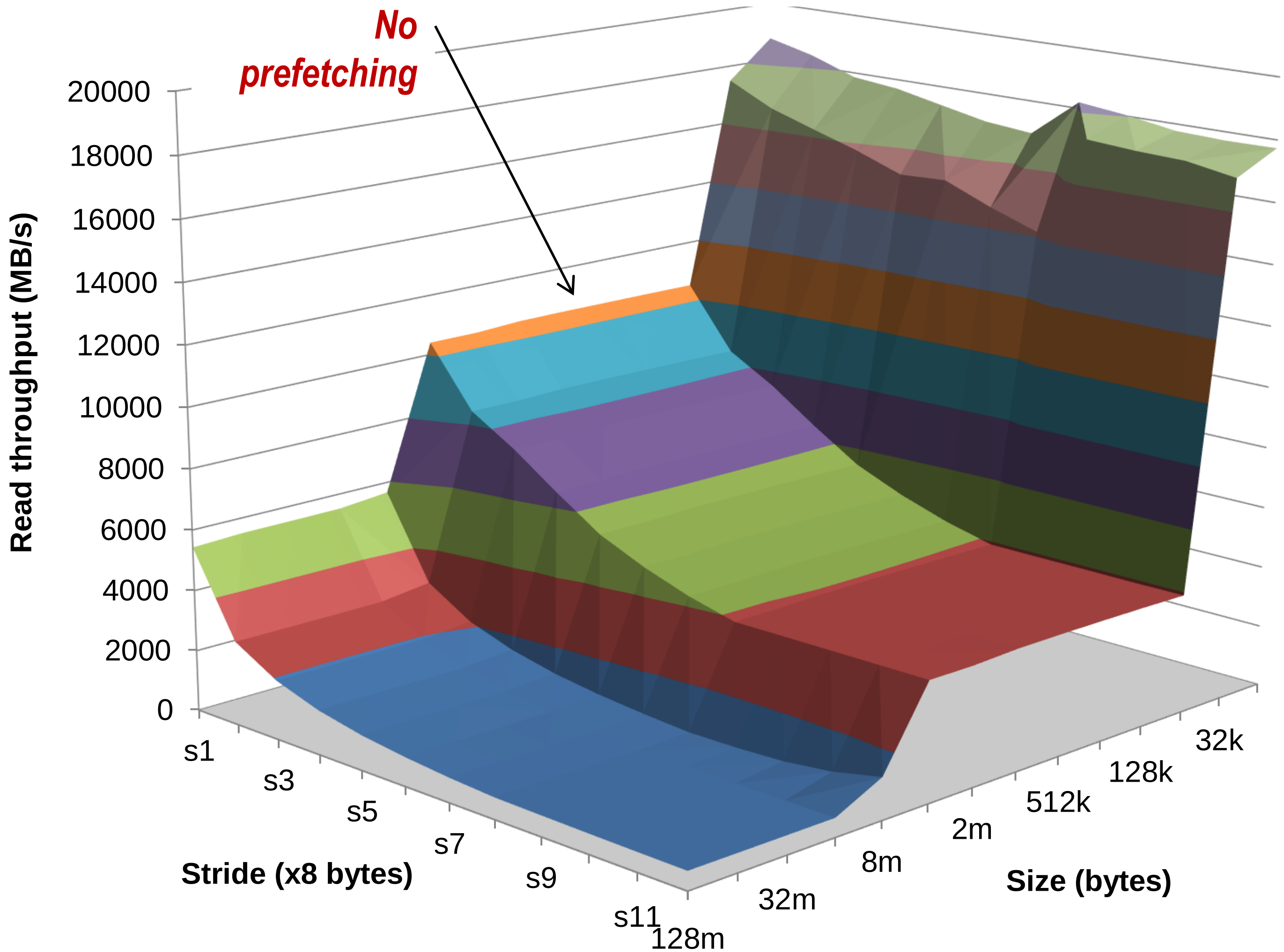
Throughput for size = 128K



$$Throughput = \frac{10^6}{8.0s + 24.3}$$

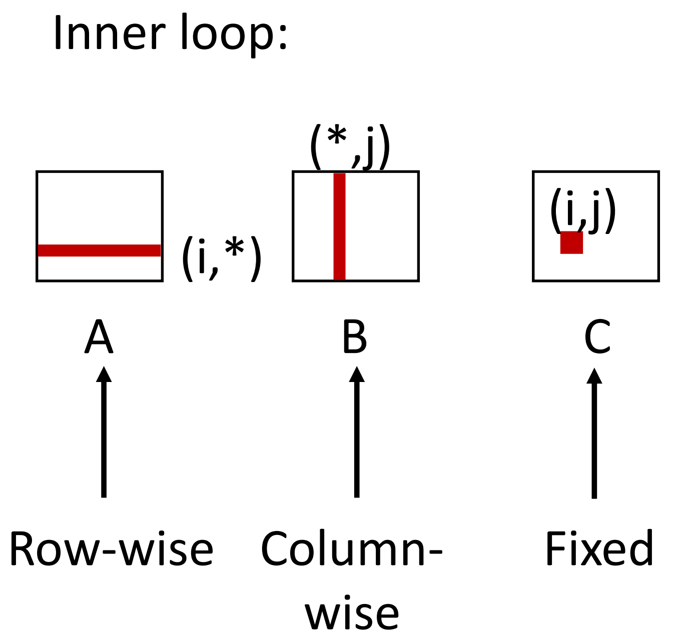
2008 Memory Mountain

Core 2 Duo
2.4 GHz
32 KB L1 d-cache
6MB L2 cache
64 B block size



Matrix Multiplication (`jik`)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}                                    matmult/mm.c
```



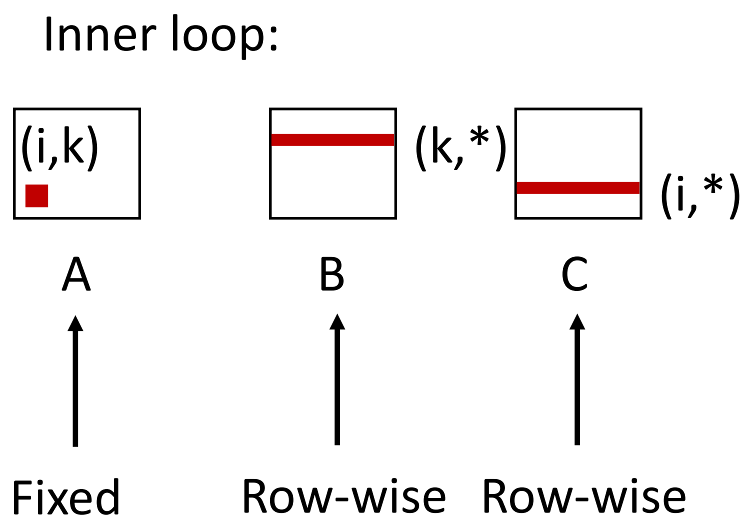
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)

Matrix Multiplication (`ikj`)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
                                matmult/mm.c
```



Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

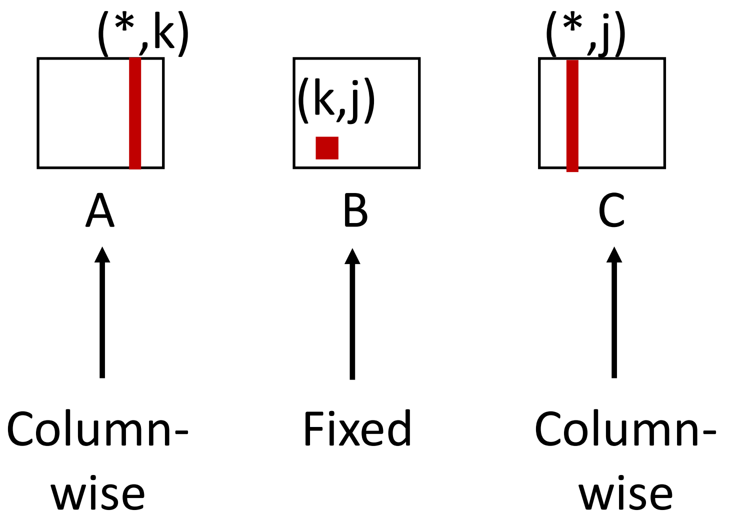
Block size = 32B (four doubles)

Matrix Multiplication (`kji`)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)

Cache and Memory Wrap-Up

15-213/18-213/15-513/18-613: Introduction to Computer Systems
16 March 2021

Design and Debugging

15-213/18-213/15-513/18-613: Introduction to Computer Systems
16 March 2021

After this lecture

- **You will be able to:**
  - Describe the steps to debug complex code failures
  - Identify ways to manage the complexity when programming
  - State guidelines for communicating the intention of the code

Outline

- **Debugging**
  - **Defects and Failures**
  - **Scientific Debugging**
  - **Tools**
- Design
  - Managing complexity
  - Communication
  - Naming
  - Comments

Defects and Infections

1. The programmer creates a defect
2. The defect causes an infection
3. The infection propagates
4. The infection causes a failure

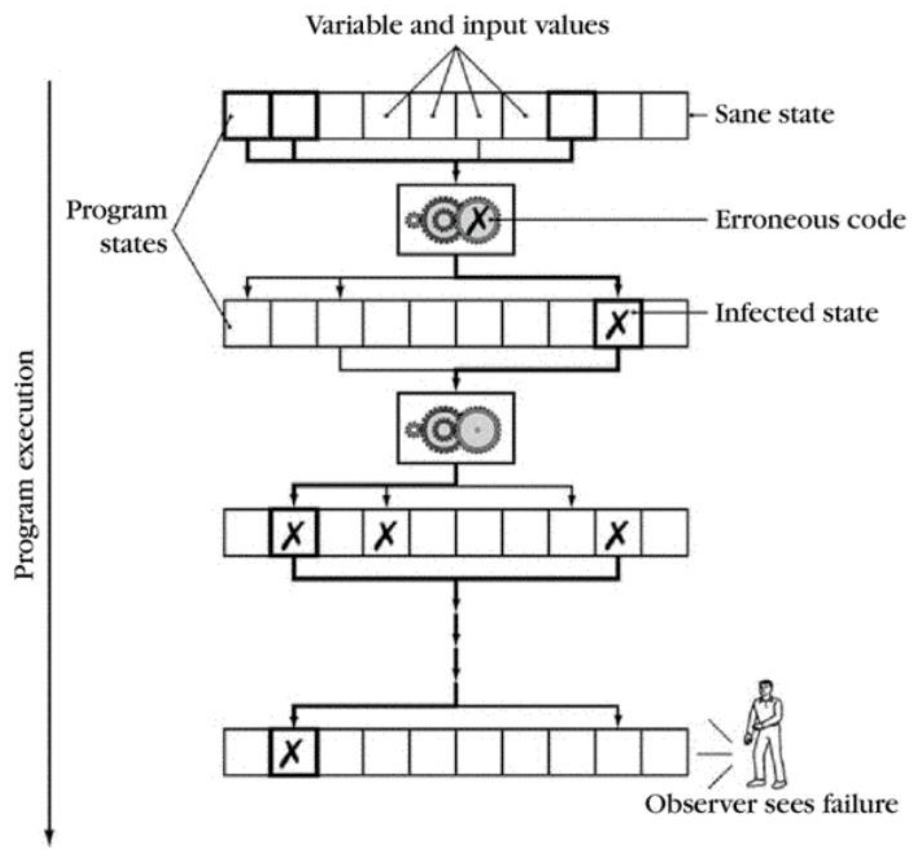
Curse of Debugging

- **Not every defect causes a failure!**

- **Testing can only show the presence of errors – not their absence. (Dijkstra 1972)**

Defects to Failures

- **Code with defects will introduce erroneous or "infected" state**
  - Correct code may propagate this state
  - Eventually an erroneous state is observed
- **Some executions will not trigger the defect**
  - Others will not propagate "infected" state
- **Debugging sifts through the code to find the defect**

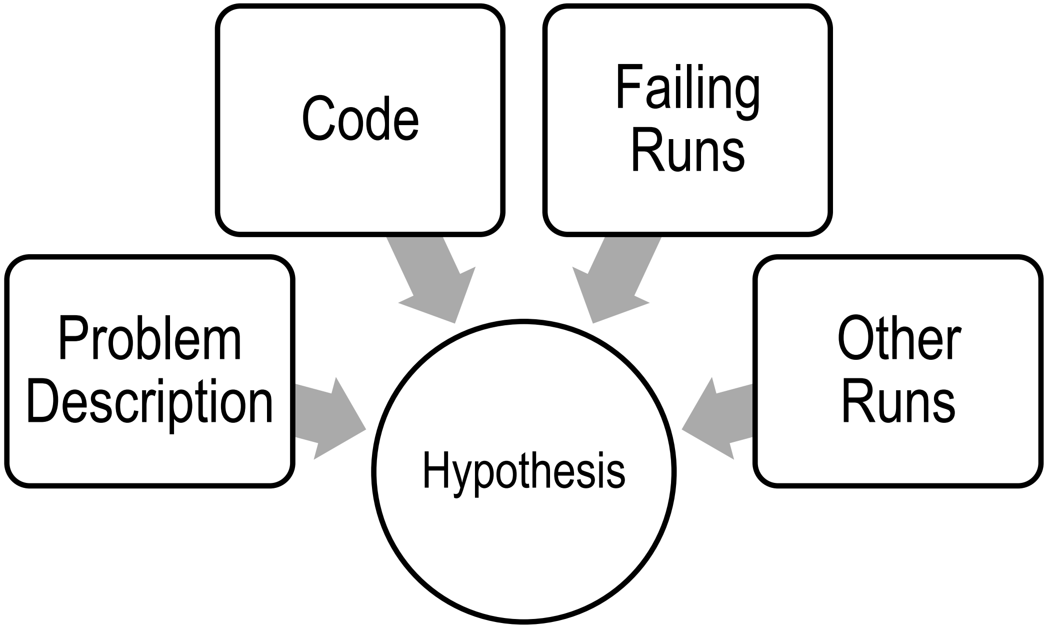


Explicit Debugging

- **Stating the problem**
  - Describe the problem aloud or in writing
    - A.k.a. “Rubber duck” or “teddy bear” method
  - Often a comprehensive problem description is sufficient to solve the failure

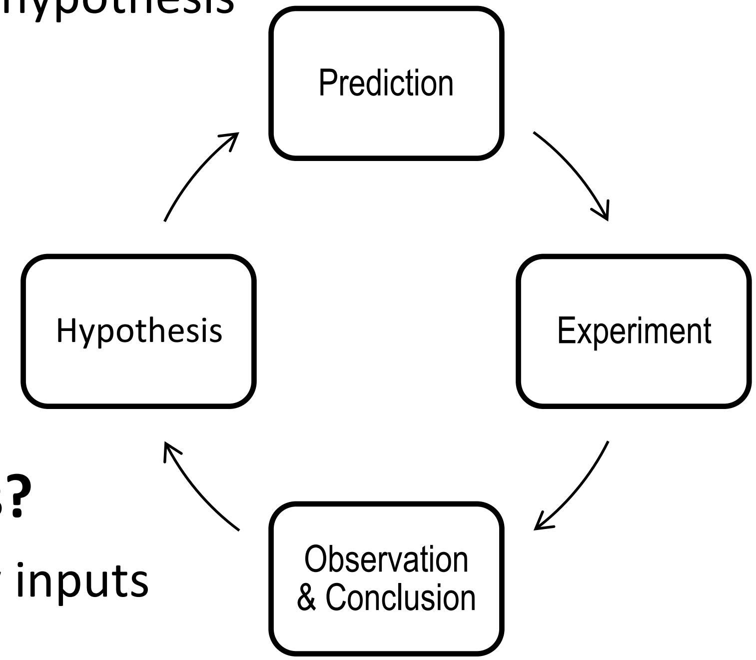
Scientific Debugging

- **Before debugging, you need to construct a hypothesis as to the defect**
  - Propose a possible defect and why it explains the failure conditions
- **Ockham's Razor – given several hypotheses, pick the simplest / closest to current work**



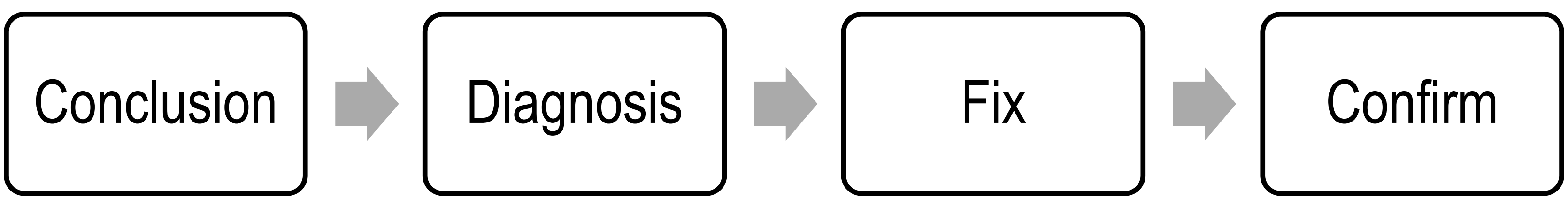
Scientific Debugging

- **Make predictions based on your hypothesis**
  - What do you expect to happen under new conditions
  - What data could confirm or refute your hypothesis
- **How can I collect that data?**
  - What experiments?
  - What collection mechanism?
- **Does the data refute the hypothesis?**
  - Refine the hypothesis based on the new inputs



Scientific Debugging

- **A set of experiments has confirmed the hypothesis**
  - This is the diagnosis of the defect
- **Develop a fix for the defect**
- **Run experiments to confirm the fix**
  - Otherwise, how do you know that it is fixed?



Code with a Bug

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main(..) {
..
  for (i = 9; i > 0; i--)
    printf("fib(%d)=%d\n",
            i, fib(i));
```

```
$ gcc -o fib fib.c
fib(9)=55
fib(8)=34
...
fib(2)=2
fib(1)=134513905
```

A defect has caused a failure.

Constructing a Hypothesis

- **Specification defined the first Fibonacci number as 1**
  - We have observed working runs (e.g., fib(2))
  - We have observed a failing run
  - We then read the code
- **fib(1) failed // Hypothesis**

Code	Hypothesis
for (i = 9; …)	Result depends on order of calls
while (n > 1) {	Loop check is incorrect
int f;	f is uninitialized

Prediction

- **Propose a new condition or conditions**
  - What will logically happen if your hypothesis is correct?
  - What data can be

- **fib(1) failed // Hypothesis**
  - // Result depends on order of calls
    - If fib(1) is called first, it will return correctly.
  - // Loop check is incorrect
    - Change to n >= 1 and run again.
  - // f is uninitialized
    - Change to int f = 1;

Experiment

- **Identical to the conditions of a prior run**
  - Except with one condition changed
- **Conditions**
  - Program input, using a debugger, altering the code

- **fib(1) failed // Hypothesis**
  - If fib(1) is called first, it will return correctly.
    - Fails.
  - Change to n >= 1
    - fib(1)=2
    - fib(0)=...
  - Change to int f = 1;
    - Works. Sometimes a prediction can be a fix.

Observation

- **What is the observed result?**
  - Factual observation, such as “Calling fib(1) will return 1.”
  - The conclusion will interpret the observation(s)
- **Don’t interfere.**
  - printf() can interfere
  - Like quantum physics, sometimes observations are part of the experiment
- **Proceed systematically.**
  - Update the conditions incrementally so each observation relates to a specific change

Debugging Tools

- **Observing program state can require a variety of tools**
  - Debugger (e.g., gdb)
    - What state is in local / global variables (if known)
    - What path through the program was taken
  - Valgrind
    - Does execution depend on uninitialized variables
    - Are memory accesses ever out-of-bounds

Diagnosis

- **A scientific hypothesis that explains current observations and makes future predictions becomes a theory**
  - We'll call this a diagnosis
- **Use the diagnosis to develop a fix for the defect**
  - Avoid "post hoc, ergo propter hoc" fallacy
    - Latin for: "After this, therefore because of this"
  - Or correlation does not imply causation
- **Understand why the defect and fix relate**

Once there was a program that only worked on Wednesday…

Fix and Confirm

- **Confirm that the fix resolves the failure**
- **If you fix multiple perceived defects, which fix was for the failure?**
  - Be systematic

Learn

- **Common failures and insights**
  - Why did the code fail?
  - What are my common defects?

- **Assertions and invariants**
  - Add checks for expected behavior
  - Extend checks to detect the fixed failure

- **Testing**
  - Every successful set of conditions is added to the test suite

Quick and Dirty

- **Not every problem needs scientific debugging**
  - Set a time limit: (for example)
    - 0 – 10 minutes – Informal Debugging
    - 10 – 60 minutes – Scientific Debugging
    - > 60 minutes – Take a break / Ask for help

Code Smells

Common ways in which code is likely to have bugs, either already or in the future

- Use of uninitialized variables
- Unused values
- Unreachable code
- Duplicated code
- Bloated functions/methods
- Memory leaks
- Interface misuse
- Null pointers
- Etc

Quiz Time!

Check out:

https://canvas.cmu.edu/courses/10968

Outline

- Debugging
  - Defects and Failures
  - Scientific Debugging
  - Tools
- **Design**
  - **Managing complexity**
  - **Communication**
  - **Naming**
  - **Comments**

Design

- **A good design needs to achieve many things:**
  - Performance
  - Availability
  - Modifiability, portability
  - Scalability
  - Security
  - Testability
  - Usability
  - Cost to build, cost to operate

Design

Good Design does:

Complexity Management &

Communication

Complexity

- There are well known limits to how much complexity a human can manage easily.

VOL. 63, NO. 2 MARCH, 1956

THE PSYCHOLOGICAL REVIEW

THE MAGICAL NUMBER SEVEN, PLUS OR MINUS TWO: SOME LIMITS ON OUR CAPACITY FOR PROCESSING INFORMATION [1]

GEORGE A. MILLER

Harvard University

Complexity Management

- However, patterns can be very helpful...

COGNITIVE PSYCHOLOGY 4, 55–81 (1973)

Perception in Chess[1]

WILLIAM G. CHASE AND HERBERT A. SIMON
Carnegie–Mellon University

This paper develops a technique for isolating and studying the perceptual structures that chess players perceive. Three chess players of varying strength — from master to novice — were confronted with two tasks: (1) A perception task, where the player reproduces a chess position in plain view, and (2) de Groot's (1965) short-term recall task, where the player reproduces a chess position after viewing it for 5 sec. The successive glances at the position in the perceptual task and long pauses in the memory task were used to segment the structures in the reconstruction protocol. The size and nature of these structures were then analyzed as a function of chess skill.

Complexity Management

Many techniques have been developed to help manage complexity:

- Separation of concerns
- Modularity
- Reusability
- Extensibility
- DRY
- Abstraction
- Information Hiding
- ...

Managing Complexity

- **Given the many ways to manage complexity**
  - Design code to be testable
  - Try to reuse testable chunks

Complexity Example

- **Split a cache access into three+ testable components**
  - State all of the steps that a cache access requires

  - Which steps depend on the operation being a load or a store?

Complexity Example

- **Split a cache access into three+ testable components**
  - State all of the steps that a cache access requires

    Convert address into tag, set index, block offset

    Look up the set using the set index

    Check if the tag matches any line in the set

    If so, hit

    If not a match, miss, then

    Find the LRU block

    Evict the LRU block

    Read in the new line from memory

    Update LRU

    Update dirty if the access was a store

  - Which steps depend on the operation being a load or a store?

Designs need to be testable

- **Testable design**
  - Testing versus Contracts
  - These are complementary techniques

- **Testing and Contracts are**
  - Acts of design more than verification
  - Acts of documentation

Testing Example

- **For your cache simulator, you can write your own traces**
  - Write a trace to test for a cache hit

    L 50, 1
    L 50, 1

  - Write a trace to test dirty bytes in cache

    S 100, 1

Communication

When writing code, the author is communicating with:

- The machine
- Other developers of the system
- Code reviewers
- Their future self

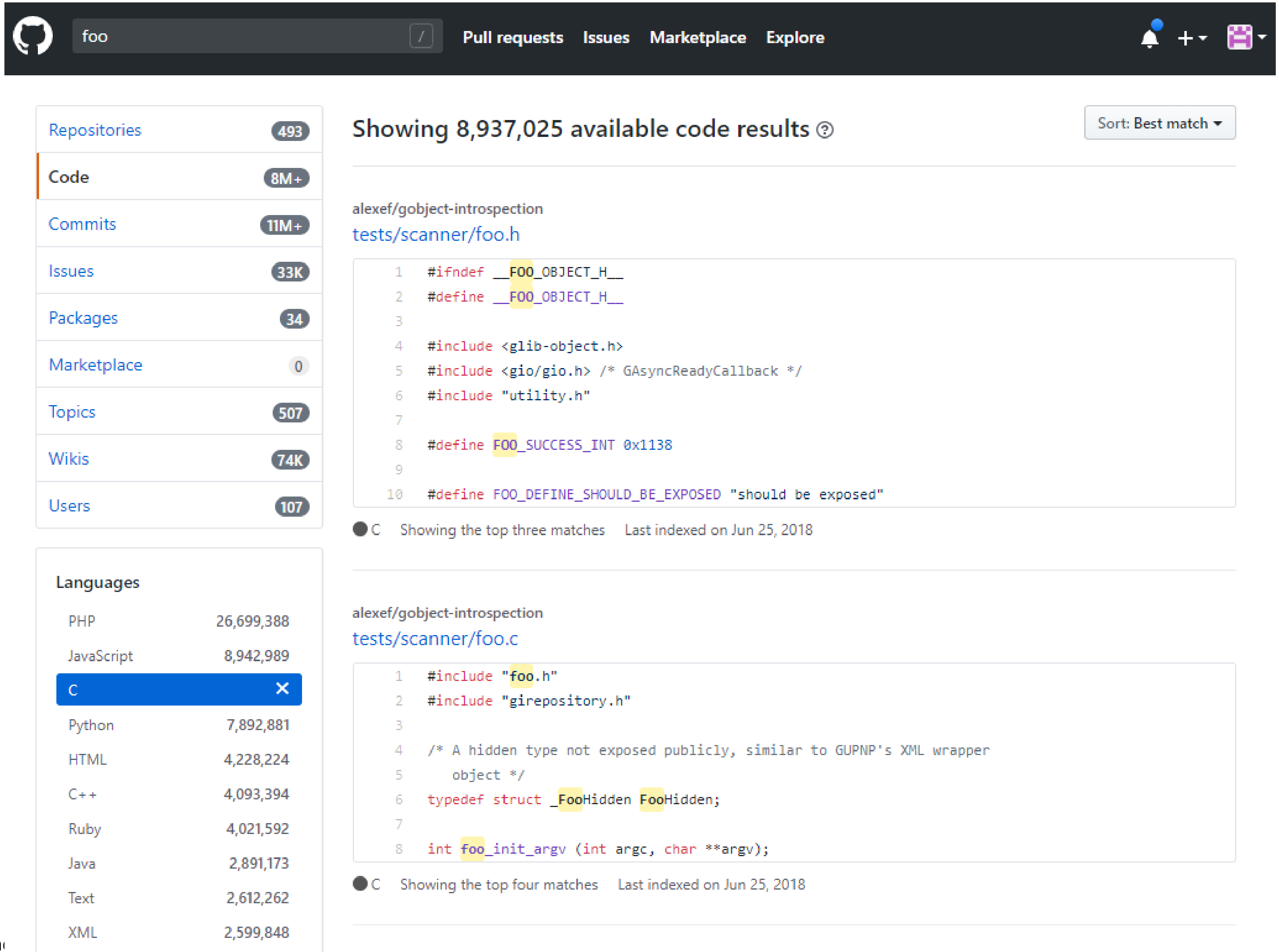
Communication

There are many techniques that have been developed around code communication:

- Tests
- Naming
- Comments
- Commit Messages
- Code Review
- Design Patterns
- …

Naming

Avoid deliberately meaningless names:



Naming is understanding

"If you don't know what a thing should be called, you cannot know what it is.

***If you don't know what it is, you cannot sit down and write the code."* - Sam Gardiner**

Better naming practices

1. Start with meaning and intention
2. Use words with precise meanings (avoid “data”, “info”, “perform”)
3. Prefer fewer words in names
4. Avoid abbreviations in names
5. Use code review to improve names
6. Read the code out loud to check that it sounds okay
7. Actually rename things

Naming guidelines – Use dictionary words

- **Only use dictionary words and abbreviations that appear in a dictionary.**
  - For example: FileCpy -> FileCopy
  - Avoid vague abbreviations such as acc, mod, auth, etc..

Avoid using single-letter names

- **Single letters are unsearchable**
  - Give no hints as to the variable's usage

- **Exceptions are loop counters**
  - Especially if you know why i, j, etc were originally used

Limit name character length

“Good naming limits individual name length, and reduces the need for specialized vocabulary” – Philip Relf

Limit name word count

- Keep names to a four word maximum
- Limit names to the number of words that people can read at a glance.

- Which of each pair do you prefer?

```
a1) arraysOfSetsOfLinesOfBlocks

a2) cache



b1) evictedData

b2) evictedDataBytes
```

Describe Meaning

- Use descriptive names.
- Avoid names with no meaning: a, foo, blah, tmp, etc

- There are reasonable exceptions:

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Use a large vocabulary

- **Be more specific when possible:**
  - Person -> Employee

- **What is size in this binaryTree?**

```
struct binaryTree {
  int size;
  …
};
```

Use problem domain terms

- **Use the correct term in the problem domain's language.**
  - Hint: as a student, consider the terms in the assignment

- **In cachelab, consider the following:**

```
line

element
```

Use opposites precisely

- **Consistently use opposites in standard pairs**
  - first/end -> first/last

Comments

Don’t Comments

- **Don’t say what the code does**
  - because the code already says that
- **Don’t explain awkward logic**
  - improve the code to make it clear
- **Don’t add too many comments**
  - it’s messy, and they get out of date

Awkward Code

- **Imagine someone (TA, employer, etc) has to read your code**
  - Would you rather rewrite or comment the following?

```
(*(void **)((*(void **)(bp)) + DSIZE)) = (*(void **)(bp + DSIZE));
```

  - How about?

```
bp->prev->next = bp->next;
```

  - Both lines update program state in the same way.

Do Comments

- Answer the question: why the code exists

- When should I use this code?
- When shouldn't I use it?
- What are the alternatives to this code?

Why does this exist?

- **Explain why a magic number is what it is.**

```
// Each address is 64-bit, which is 16 + 1 hex characters
const int MAX_ADDRESS_LENGTH = 17;
```

- **When should this code be used? Is there an alternative?**

```
unsigned power2(unsigned base, unsigned expo){
    unsigned i;
    unsigned result = 1;
    for(i=0;i<expo;i++){
        result+=result;
    }
    return result;
}
```

How to write good comments

1. **Write short comments of what the code will do.**
    1. Single line comments
    2. Example: Write four one-line comments for quick sort

```
// Initialize locals
// Pick a pivot value
// Reorder array around the pivot
// Recurse
```

How to write good comments

1. **Write short comments of what the code will do.**
    1. Single line comments
    2. Example: Write four one-line comments for quick sort

2. **Write that code.**

3. **Revise comments / code**
    1. If the code or comments are awkward or complex
    2. Join / Split comments as needed

4. **Maintain code and comments**

Commit Messages

- **Committing code to a source repository is a vital part of development**
  - Protects against system failures and typos:
    - cat foo.c versus cat > foo.c
  - The commit messages are your record of your work
    - Communicating to your future self
    - Describe in one line what you did

  “Parses command line arguments”

  “fix bug in unique tests, race condition not solved”

  “seg list finished, performance is …”

Summary

- **Programs have defects**
  - Be systematic about finding them
- **Programs are more complex than humans can manage**
  - Write code to be manageable
- **Programming is not solitary, even if you are communicating with a grader or a future self**
  - Be understandable in your communication

Acknowledgements

- **Some debugging content derived from:**
  - http://www.whyprogramsfail.com/slides.php
- **Some code examples for design are based on:**
  - "The Art of Readable Code". Boswell and Foucher. 2011.
- **Lecture originally written by**
  - Michael Hilton and Brian Railing