COMP201

Computer Systems & Programming

Lecture #24 – Debugging, Design and Optimization

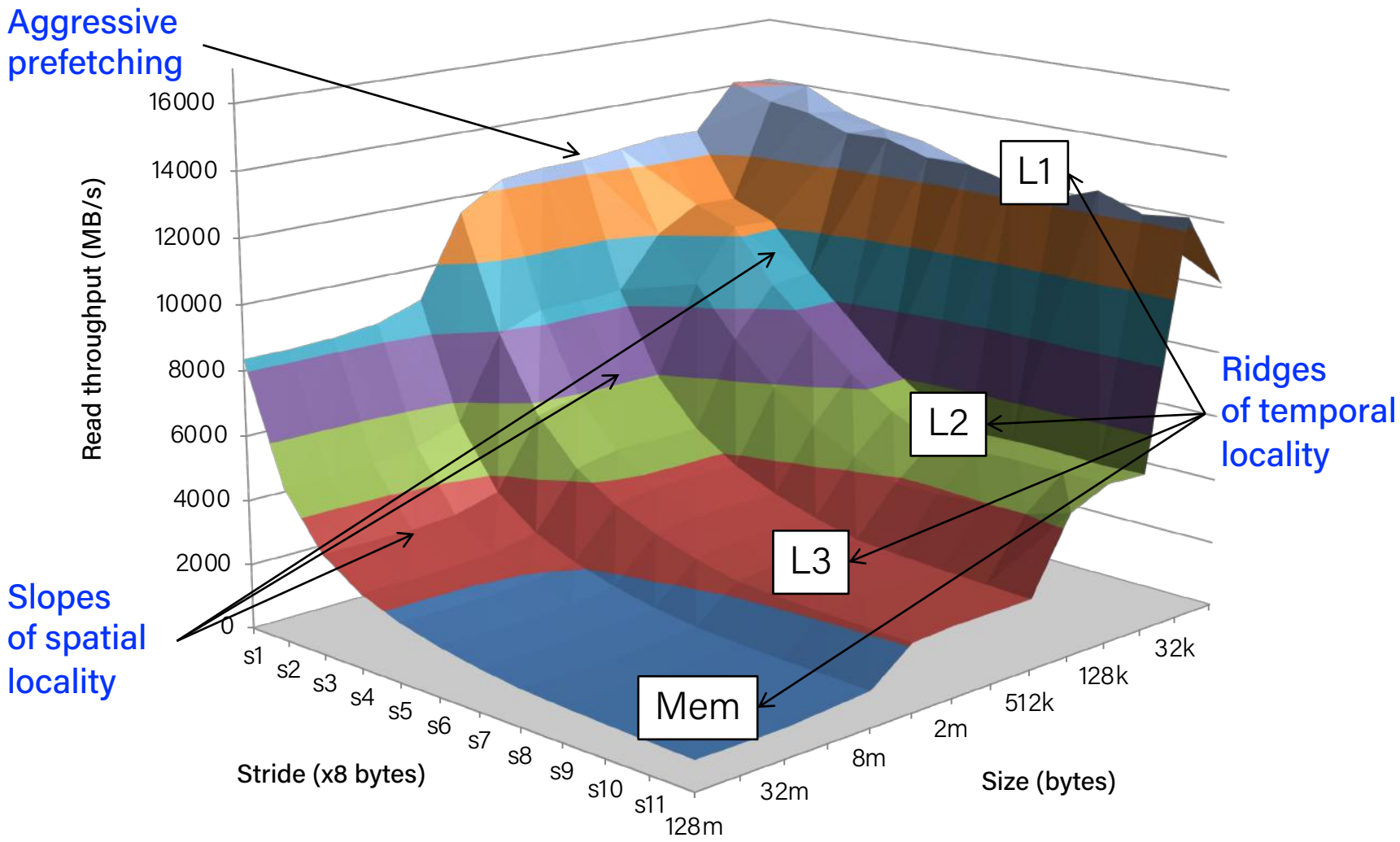
KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Recap

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

Recap: 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

Recap: 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;
 }
}
```

```
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];
 }
}
```

```
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;
 }
}
```

ijk (& jik):

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

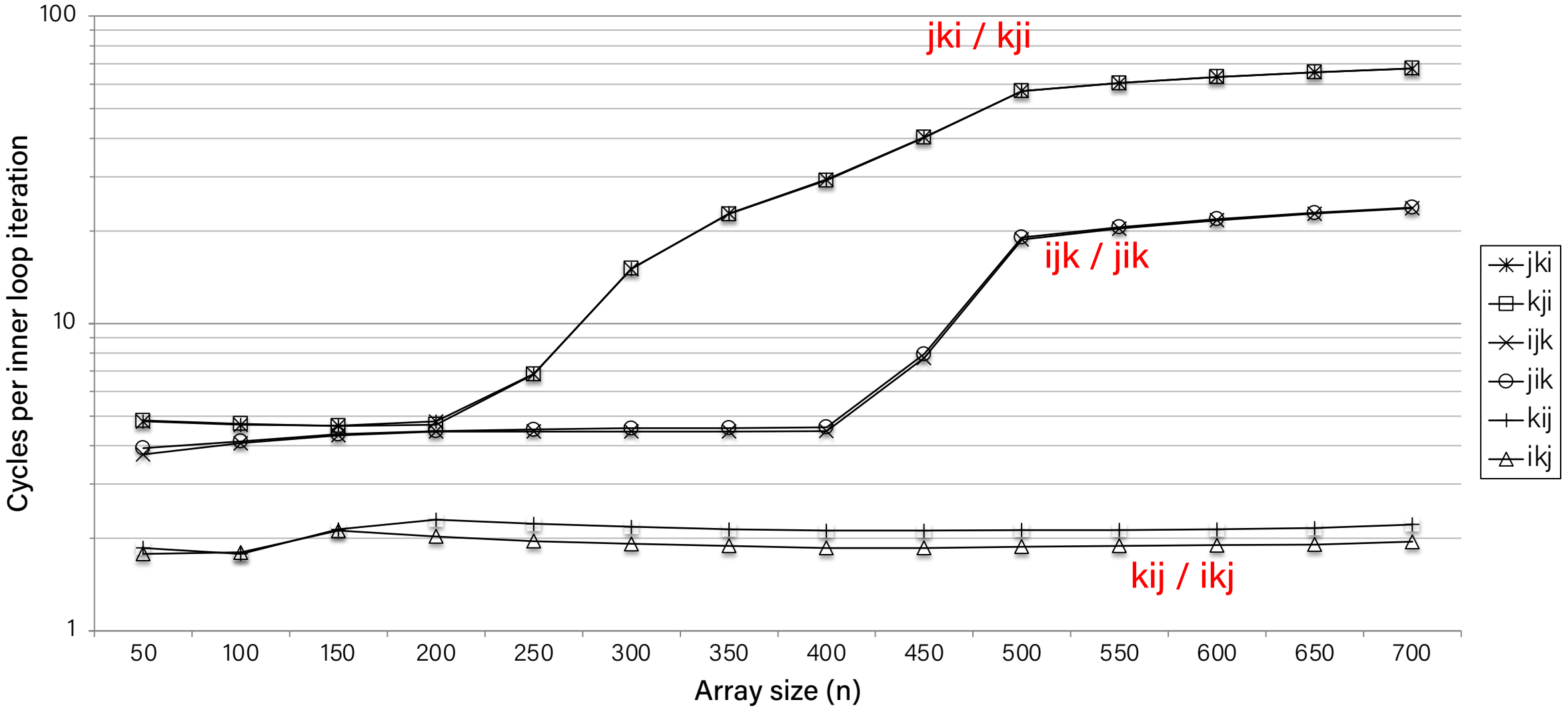
kij (& ikj):

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

jki (& kji):

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

Recap: Matrix Multiplication

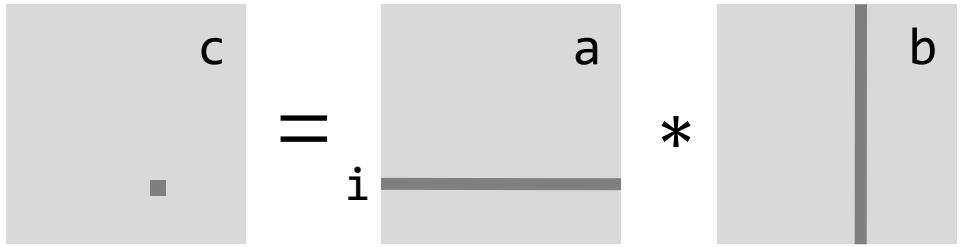


Recap: Matrix Multiplication

```
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];
}
```

- **No blocking:** $(9/8) * n^3$



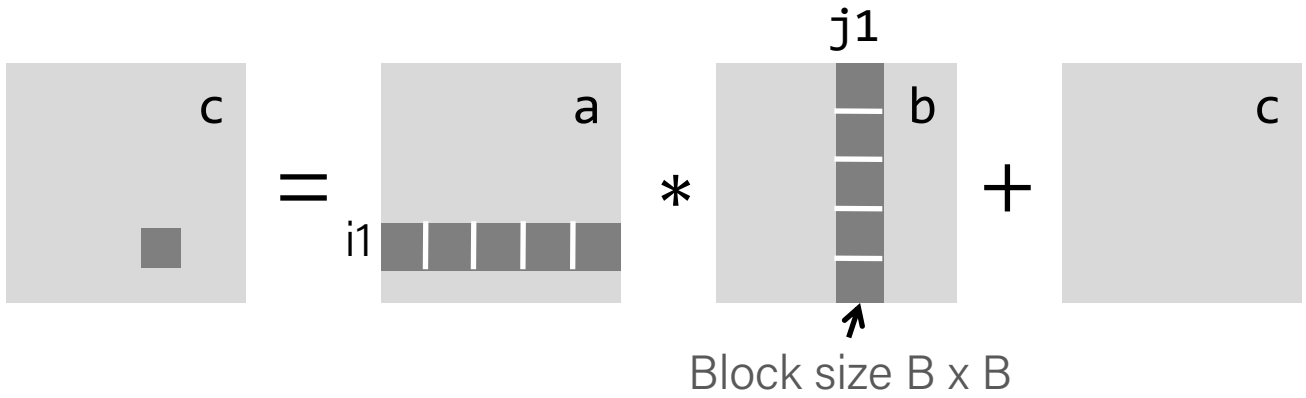
Recap: Blocked Matrix Multiplication

```
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+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

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



Plan for Today

- Debugging
- Design
- Optimization

Disclaimer: Slides for this lecture were borrowed from

—Michael Hilton and Brian Railing's CMU 15-213 class

—Nick Troccoli's Stanford CS107 class

Learning Goals

- 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
- Understand how we can optimize our code to improve efficiency and speed
- Learn about the optimizations GCC can perform

Lecture Plan

- Debugging
  - Defects and Failures
  - Scientific Debugging
  - Tools
- Design
- Optimization

The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.

Denial

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."

Bargaining/Self-Blame

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"

Anger

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.

Depression

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.

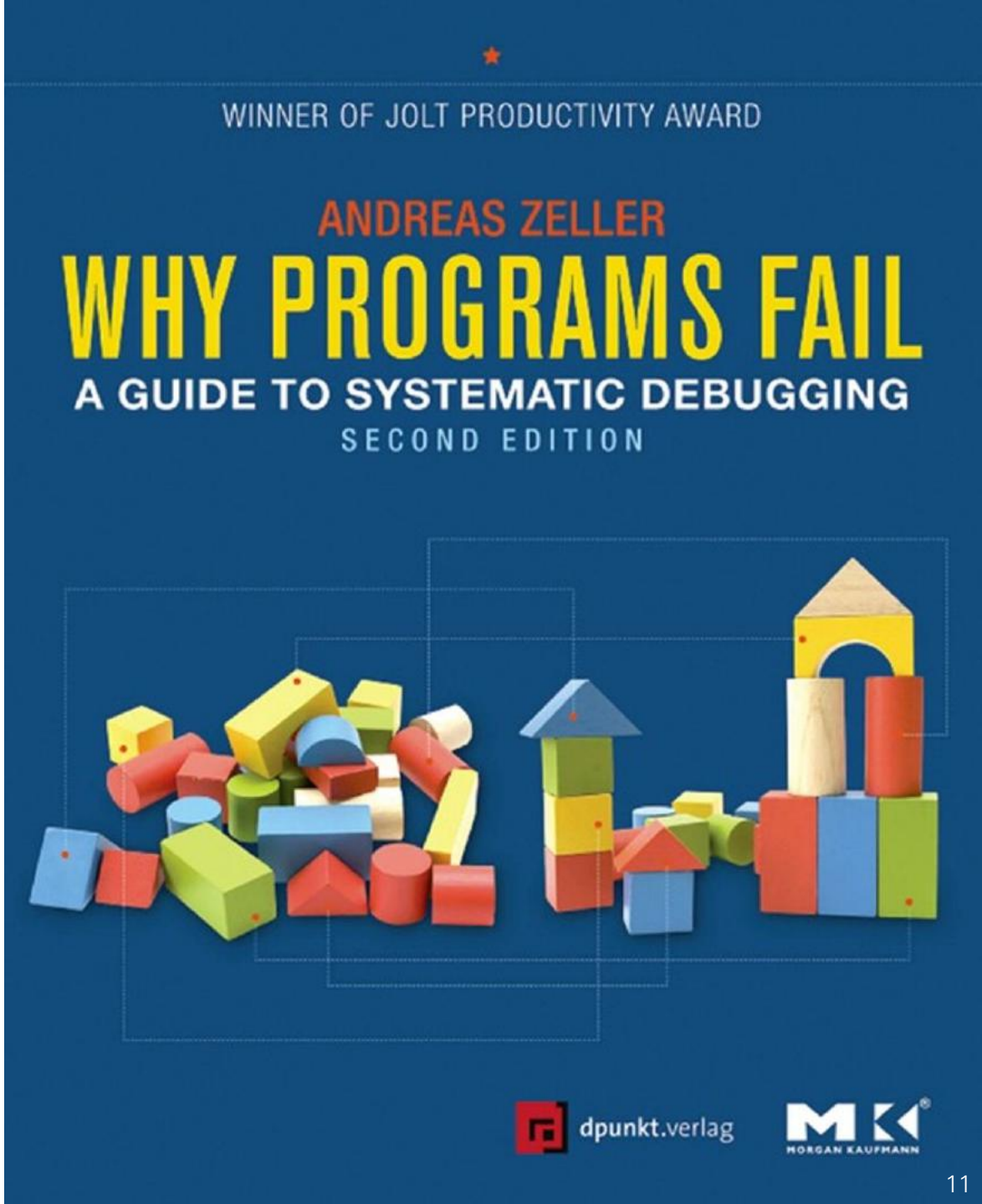
Acceptance

The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake.

https://www.reddit.com/r/ProgrammerHumor/comments/1r0cw7/the_5_stages_of_debugging/

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

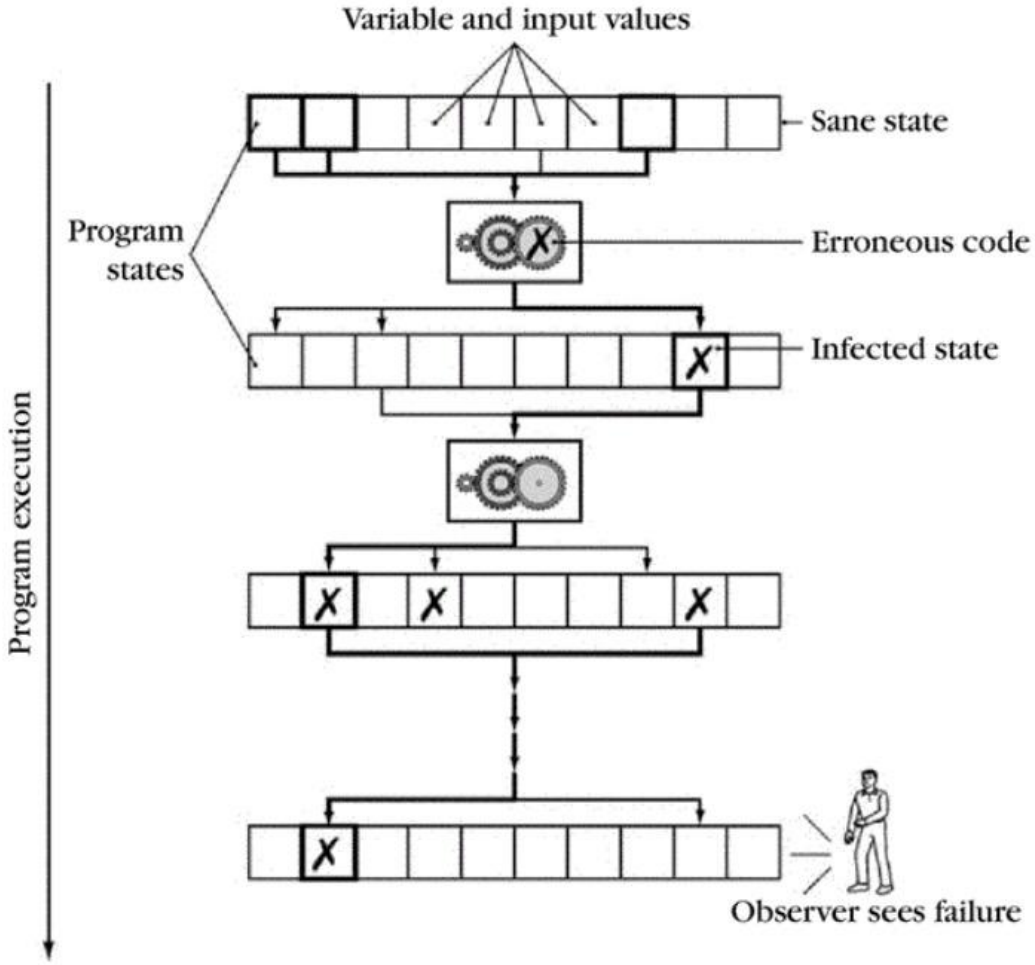
- Not every defect causes a failure!

- “Testing can only show the presence of errors – not their absence.”

  – Edsger W. 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

Explicit Debugging

- Stating the p
  - Describe tl
    - A.k.a. "Ru
  - Often a co :he failure

Rubber Duck Debugging

Debugging software with a rubber ducky

About Talk to a Duck

Rubber Duck Debugging

The rubber duck debugging method is as follows:

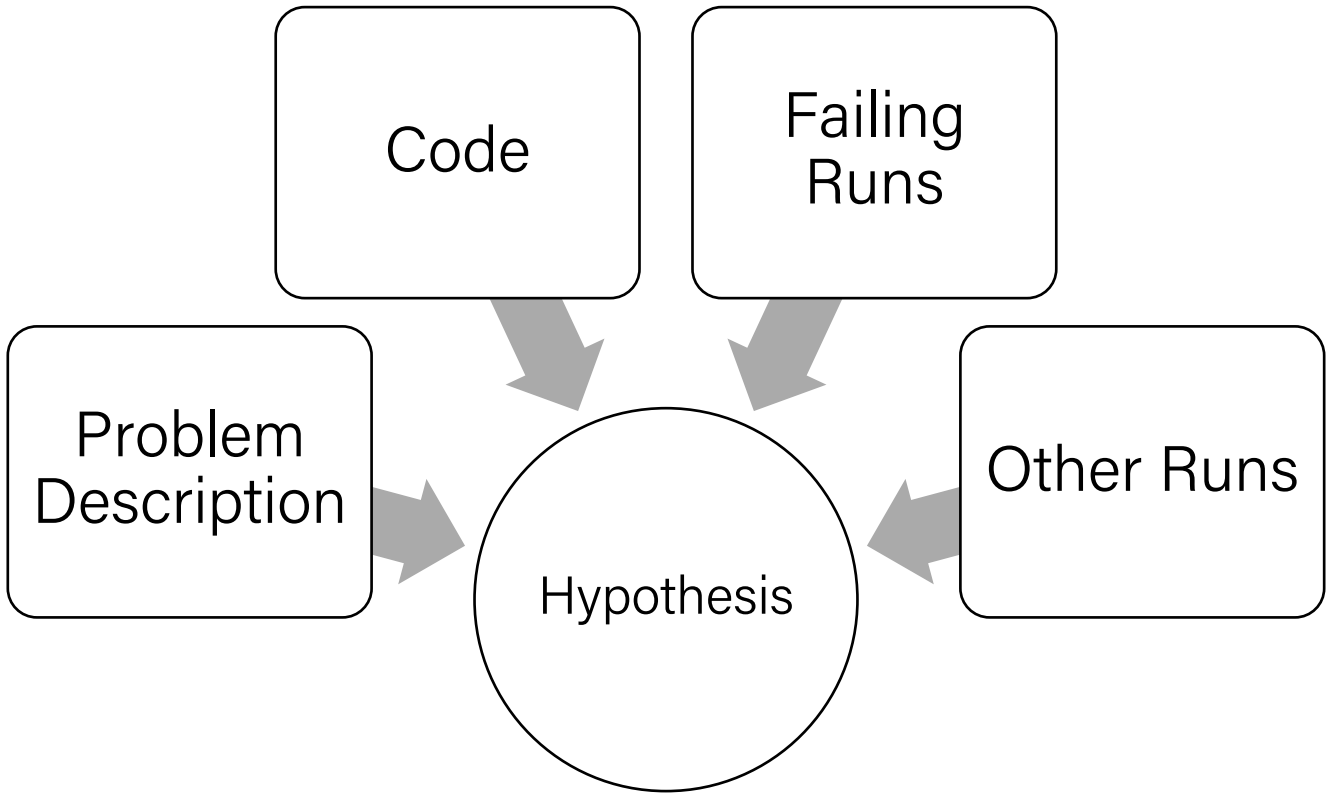
1. Beg, borrow, steal, buy, fabricate or otherwise obtain a rubber duck (bathtub variety).
2. Place rubber duck on desk and inform it you are just going to go over some code with it, if that's all right.
3. Explain to the duck what your code is supposed to do, and then go into detail and explain your code line by line.
4. At some point you will tell the duck what you are doing next and then realise that that is not in fact what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.

Note: In a pinch a coworker might be able to substitute for the duck, however, it is often preferred to confide mistakes to the duck instead of your coworker.

Original Credit: ~Andy from lists.ethernal.org

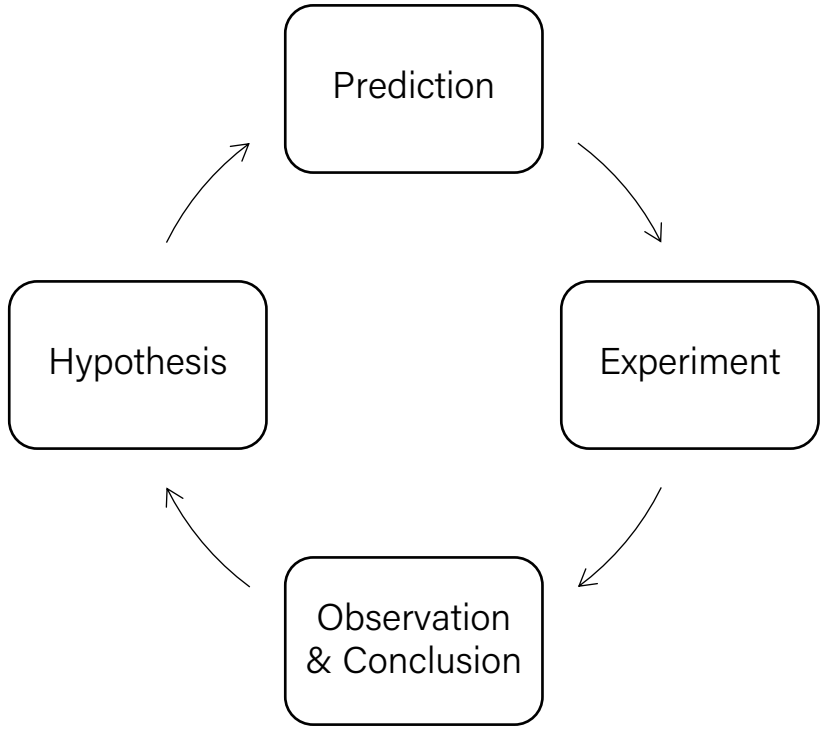
Scientific Debugging

- 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



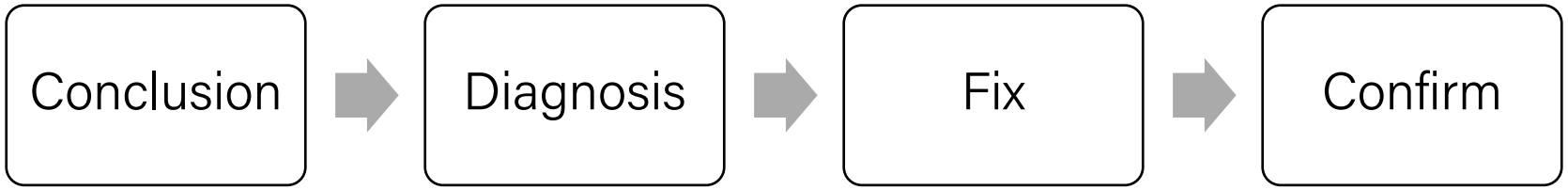
Scientific Debugging

- 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
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

```
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));
```

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

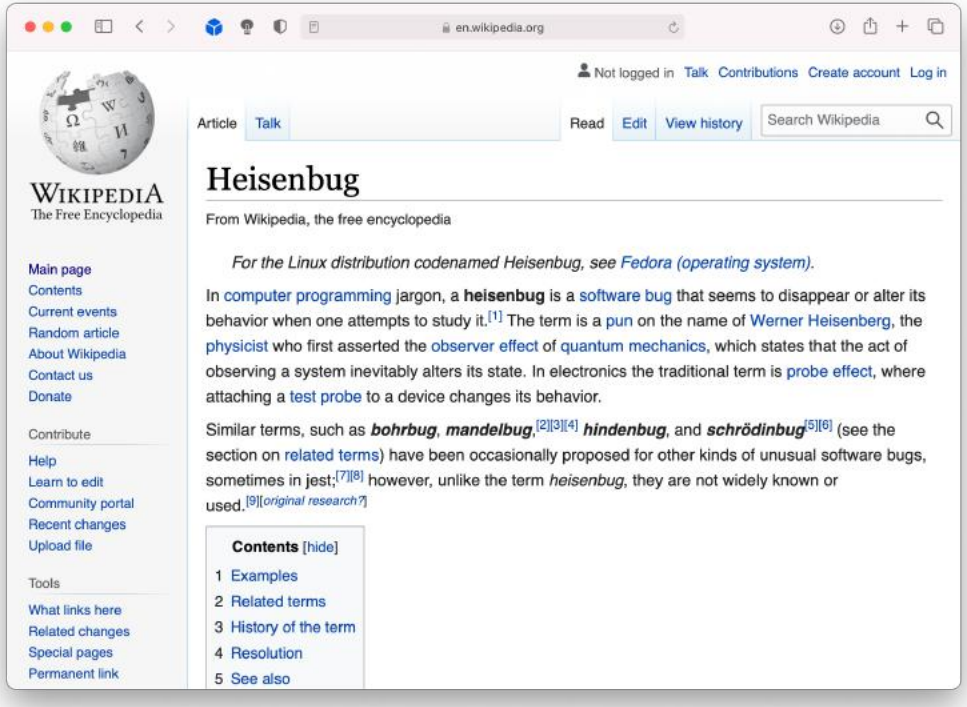
- `fib(1)` failed // Hypothesis
  - If `fib(1)` is called first, it will return correctly.
    - Fails.
  - Change to `n >= 1`
    - `fib(1)=2`
    - `fib(0)=...`
  - Change to `int f = 1;`
    - Works. Sometimes a prediction can be a fix.

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
  - Or correlation does not imply causation
- Understand why the defect and fix relate

Fix and Confirm

- Confirm that the fix resolves the failure

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

David Amador
@DJ_Link

when a "simple" bug is found and we start looking at the code

12:22 PM · Jul 3, 2017

2,181 Retweets **3,400** Likes

Hal replacing a light bulb - Malcolm in the Middle, S03E06

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

- Use of uninitialized variables
- Unused values
- Unreachable code
- Memory leaks
- Interface misuse
- Null pointers

Lecture Plan

- Debugging
- Design
  - Managing complexity
  - Communication
  - Naming
  - Comments
- Optimization

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

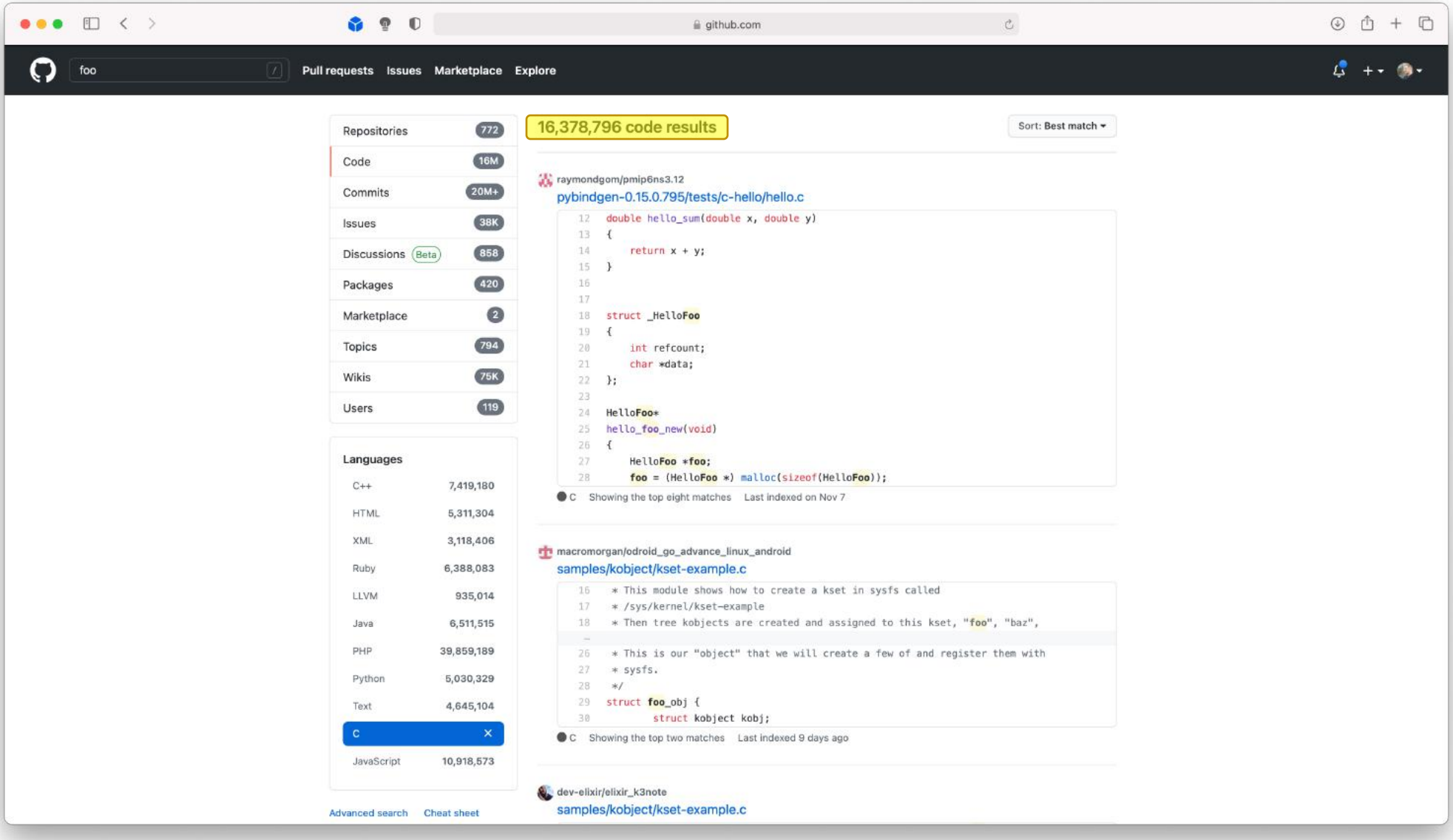
Communication

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

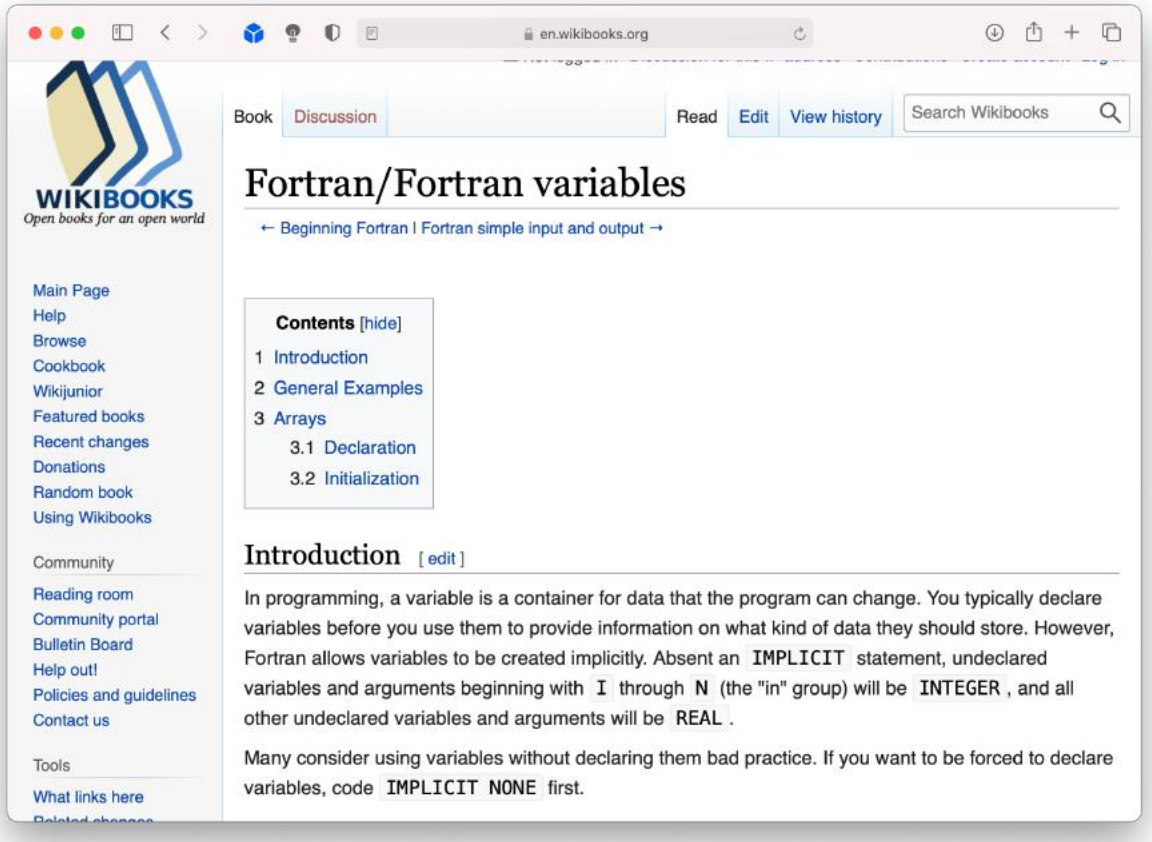
Naming guidelines – Use dictionary words

- 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 words 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 Assignment 5, consider the following:

```
username
```

```
secret_pin
```

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 …”

Debugging and Design

- 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

Lecture Plan

- Debugging
- Design
- Optimization
  - What is optimization?
  - GCC Optimization
  - Limitations of GCC Optimization
  - Caching revisited

Optimization

- Optimization is the task of making your program faster or more efficient with space or time. Later you will learn about explorations of efficiency with Big-O notation!
- *Targeted, intentional* optimizations to alleviate bottlenecks can result in big gains. But it's important to only work to optimize where necessary.

Optimization

Most of what you need to do with optimization can be summarized by:

1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
2) If doing things thing a lot, or on big inputs, make the primary algorithm's Big-O cost reasonable
3) **Let `gcc` do its magic from there**
4) Optimize explicitly as a last resort

GCC Optimization

- Today, we'll be comparing two levels of optimization in the gcc compiler:
  - `gcc -O0  // mostly just literal translation of C`
  - `gcc -O2  // enable nearly all reasonable optimizations`
  - (we use `-Og`, like `-O0` but with less needless use of the stack)

- There are other custom and more aggressive levels of optimization, e.g.:
  - `-O3     //more aggressive than -O2, trade size for speed`
  - `-Os     //optimize for size`
  - `-Ofast  //disregard standards compliance (!!)`

- **Exhaustive list of gcc optimization-related flags:**
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Example: Matrix Multiplication

Here's a standard matrix multiply, a triply-nested for loop:

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

```
./mult         // -O0 (no optimization)
matrix multiply  25^2: cycles   0.43M
matrix multiply  50^2: cycles   3.02M
matrix multiply 100^2: cycles  24.82M
```

```
./mult_opt     // -O2 (with optimization)
matrix multiply  25^2: cycles   0.13M (opt)
matrix multiply  50^2: cycles   0.66M (opt)
matrix multiply 100^2: cycles   5.55M (opt)
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- The Force

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling
- ~~The Force~~

(may be not 🤓)

GCC Optimizations

Optimizations may target one or more of:

- Static instruction count
- Dynamic instruction count
- Cycle count / execution time

GCC Optimizations

- **Constant Folding**
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Constant Folding

Constant Folding pre-calculates constants at compile-time where possible.

```
int seconds = 60 * 60 * 24 * n_days;
```

What is the consequence of this for you as a programmer?
What should you do differently or the same knowing that compilers can do this for you?

Constant Folding

```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = sqrt(2.0);
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
```

Constant Folding: Before (-O0)

```
0000000000400626 <fold>:
  400626:        55                          push   %rbp
  400627:        53                          push   %rbx
  400628:        48 83 ec 08                 sub    $0x8,%rsp
  40062c:        89 fd                       mov    %edi,%ebp
  40062e:        f2 0f 10 05 da 00 00        movsd  0xda(%rip),%xmm0
  400635:        00
  400636:        e8 d5 fe ff ff              callq  400510 <sqrt@plt>
  40063b:        f2 0f 2c c8                 cvttsd2si %xmm0,%ecx
  40063f:        69 ed 07 01 00 00           imul   $0x107,%ebp,%ebp
  400645:        b8 15 00 00 00              mov    $0x15,%eax
  40064a:        99                          cltd
  40064b:        f7 f9                       idiv   %ecx
  40064d:        8d 98 07 01 00 00           lea    0x107(%rax),%ebx
  400653:        bf 04 07 40 00              mov    $0x400704,%edi
  400658:        e8 93 fe ff ff              callq  4004f0 <strlen@plt>
  40065d:        48 69 c0 23 05 00 00        imul   $0x523,%rax,%rax
  400664:        48 63 db                    movslq %ebx,%rbx
  400667:        48 8d 44 18 c9              lea    -0x37(%rax,%rbx,1),%rax
  40066c:        48 c1 e8 02                 shr    $0x2,%rax
  400670:        01 e8                       add    %ebp,%eax
  400672:        48 83 c4 08                 add    $0x8,%rsp
  400676:        5b                          pop    %rbx
  400677:        5d                          pop    %rbp
  400678:        c3                          retq
```

Constant Folding: After (-O2)

```
0000000004004f0 <fold>:
  4004f0:         69 c7 07 01 00 00          imul   $0x107,%edi,%eax
  4004f6:         05 a5 06 00 00             add    $0x6a5,%eax
  4004fb:         c3                         retq
  4004fc:         0f 1f 40 00                nopl   0x0(%rax)
```

GCC Optimizations

- Constant Folding
- **Common Sub-expression Elimination**
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

This optimization is done even at **–O0**!

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

```
0000000004004f0 <subexp>:
  4004f0:  81 c6 07 01 00 00     add    $0x107,%esi
  4004f6:  0f af fe              imul   %esi,%edi
  4004f9:  8d 04 77              lea    (%rdi,%rsi,2),%eax
  4004fc:  0f af c6              imul   %esi,%eax
  4004ff:  c3                    retq
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- **Dead Code**
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Dead Code

Dead code elimination removes code that doesn't serve a purpose:

```
if (param1 < param2 && param1 > param2) {
    printf("This test can never be true!\n");
}

// Empty for loop
for (int i = 0; i < 1000; i++);

// If/else that does the same operation in both cases
if (param1 == param2) {
    param1++;
} else {
    param1++;
}

// If/else that more trickily does the same operation in both cases
if (param1 == 0) {
    return 0;
} else {
    return param1;
}
```

Dead Code: Before (-O0)

```
0000000000004004d6 <dead_code>:
  4004d6:    b8 00 00 00 00          mov    $0x0,%eax
  4004db:    eb 03                   jmp    4004e0 <dead_code+0xa>
  4004dd:    83 c0 01                add    $0x1,%eax
  4004e0:    3d e7 03 00 00          cmp    $0x3e7,%eax
  4004e5:    7e f6                   jle    4004dd <dead_code+0x7>
  4004e7:    39 f7                   cmp    %esi,%edi
  4004e9:    75 05                   jne    4004f0 <dead_code+0x1a>
  4004eb:    8d 47 01                lea    0x1(%rdi),%eax
  4004ee:    eb 03                   jmp    4004f3 <dead_code+0x1d>
  4004f0:    8d 47 01                lea    0x1(%rdi),%eax
  4004f3:    f3 c3                   repz retq
```

Dead Code: After (-O2)

```
0000000000004004f0 <dead_code>:
  4004f0: 8d 47 01                lea    0x1(%rdi),%eax
  4004f3: c3                      retq
  4004f4: 66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4004fb: 00 00 00
  4004fe: 66 90                   xchg   %ax,%ax
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- **Strength Reduction**
- Code Motion
- Tail Recursion
- Loop Unrolling

Strength Reduction

Strength reduction changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

```
int a = param2 * 32;
int b = a * 7;
int c = b / 3;
int d = param2 % 2;

for (int i = 0; i <= param2; i++) {
    c += param1[i] + 0x107 * i;
}
return c + d;
```

Strength Reduction: After (-O3)

```
unsigned udiv19(unsigned arg) {
return arg / 19;
}
```

```
udiv19(unsigned int):
    mov   eax, edi
    mov   edx, 2938661835
    imul   rax, rdx
    shr   rax, 32
    sub   edi, eax
    shr   edi
    add   eax, edi
    shr   eax, 4
    ret
```

https://godbolt.org/z/Wq8ra3

What really happens here?

$$a \cdot \frac{1}{19} \approx \frac{a \cdot \frac{2938661835}{2^{32}} + \frac{a - a \cdot \frac{2938661835}{2^{32}}}{2^1}}{2^4}$$

$$a \cdot \frac{1}{19} \approx \left(a \cdot 2938661835 \cdot 2^{-32} + \left(a - a \cdot 2938661835 \cdot 2^{-32}\right) \cdot 2^{-1}\right) \cdot 2^{-4}$$

$$a \cdot \frac{1}{19} \approx a \cdot \frac{7233629131}{137438953472}$$

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **Code Motion**
- Tail Recursion
- Loop Unrolling

Code Motion

Code motion moves code outside of a loop if possible.

```
for (int i = 0; i < n; i++) {
  sum += arr[i] + foo * (bar + 3);
}
```

Common subexpression elimination deals with expressions that appear multiple times in the code. Here, the expression appears once, but is calculated each loop iteration.

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- **Tail Recursion**
- Loop Unrolling

Tail Recursion

Tail recursion is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

```
long factorial(int n) {
  if (n <= 1) {
      return 1;
  }
  else return n * factorial(n - 1);
}
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- **Loop Unrolling**

Loop Unrolling

Loop Unrolling: Do **n** loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

```
for (int i = 0; i <= n - 4; i += 4) {
    sum += arr[i];
    sum += arr[i + 1];
    sum += arr[i + 2];
    sum += arr[i + 3];
} // after the loop handle any leftovers
```

Limitations of GCC Optimization

GCC can't optimize everything! You ultimately may know more than GCC does.

```
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck? **`strlen` called for every character**
What can GCC do? **code motion – pull `strlen` out of loop**

Limitations of GCC Optimization

GCC can't optimize everything! You ultimately may know more than GCC does.

```
void lower1(char *s) {
    for (size_t i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] -= ('A' - 'a');
        }
    }
}
```

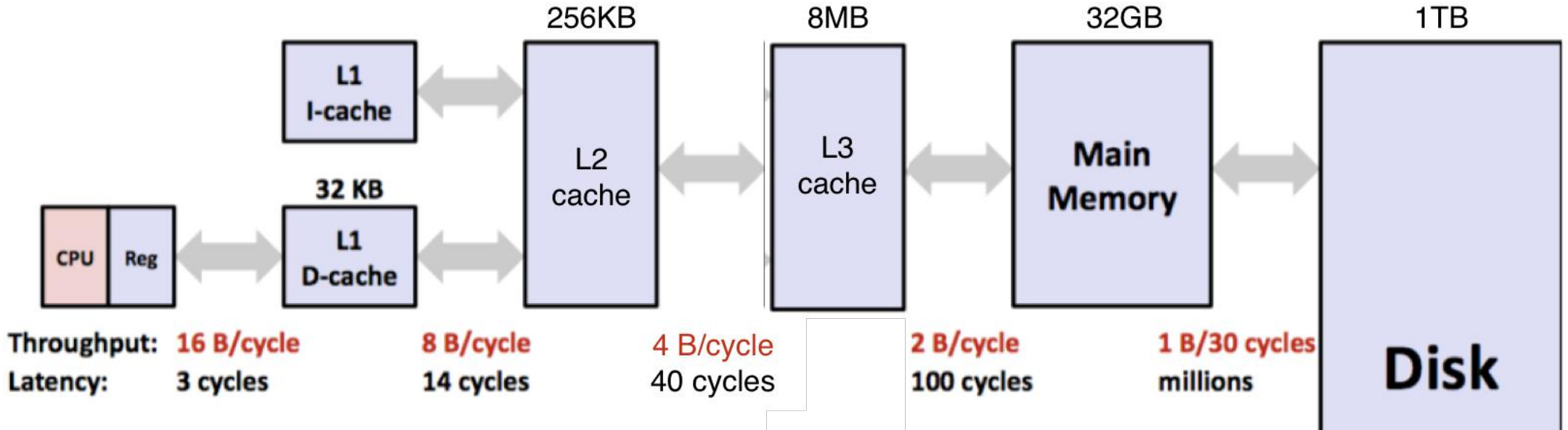
What is the bottleneck?
What can GCC do?

`strlen` called for every character

nothing! `s` is changing, so GCC doesn't know if length is constant across iterations. But we know its length doesn't change.

Caching

- Processor speed is not the only bottleneck in program performance – memory access is perhaps even more of a bottleneck!
- Memory exists in levels and goes from *really fast* (registers) to *really slow* (disk).
- As data is more frequently used, it ends up in faster and faster memory.



Caching

All caching depends on locality.

Temporal locality

- Repeat access to the same data tends to be co-located in TIME
- Intuitively: things I have used recently, I am likely to use again soon

Spatial locality

- Related data tends to be co-located in SPACE
- Intuitively: data that is near a used item is more likely to also be accessed

Optimizing Your Code

- Explore various optimizations you can make to your code to reduce instruction count and runtime.
  - More efficient Big-O for your algorithms
  - Explore other ways to reduce instruction count
    - Look for hotspots using `callgrind`
    - Optimize using `-O2`
    - And more...

Compiler Optimizations

Why not always just compile with `-O2`*?*

- Difficult to debug optimized executables – only optimize when complete
- Optimizations may not *always* improve your program. The compiler does its best, but may not work, or slow things down, etc. Experiment to see what works best!

Why should we bother saving repeated calculations in variables if the compiler has common subexpression elimination?

- The compiler may not always be able to optimize every instance. Plus, it can help reduce redundancy!

Recap

- Debugging
- Design
- Optimization

Next time: Linking