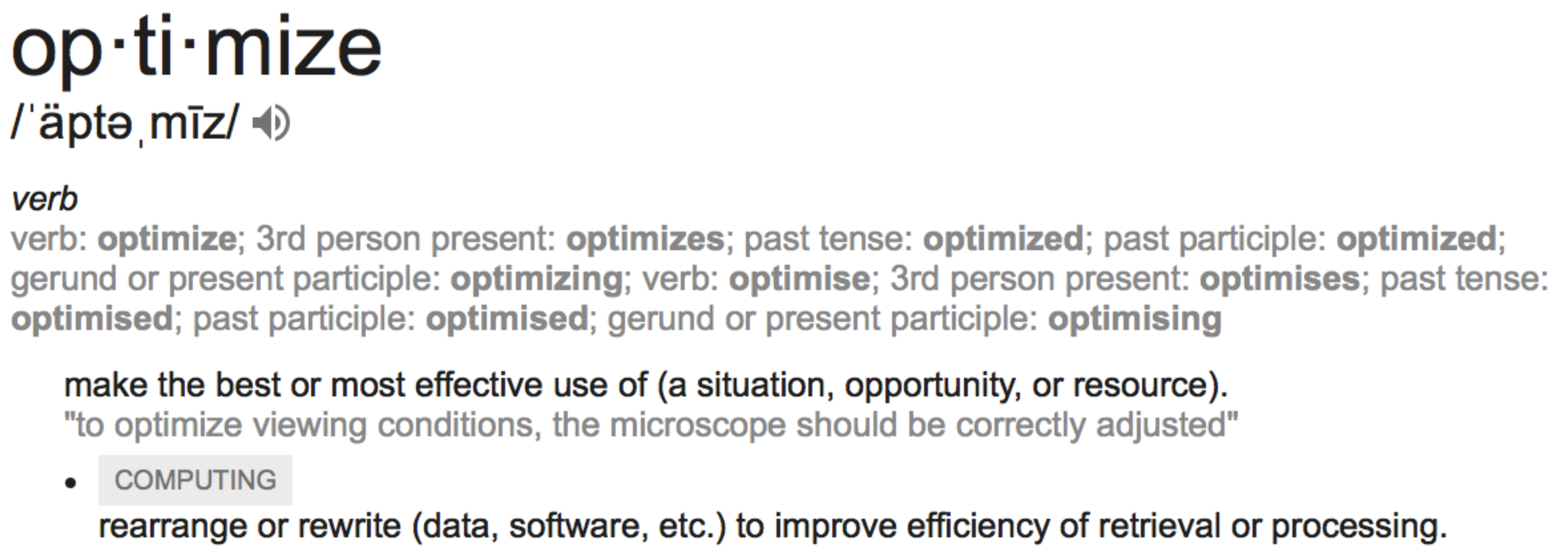
CS 107 Lecture 16: Optimization

Monday, March 6, 2023

Computer Systems
Winter 2023
Stanford University
Computer Science Department

Reading: Chapter 5, textbook

Lecturer: Chris Gregg



op·ti·mize
/ˈäptəˌmīz/

verb
verb: **optimize**; 3rd person present: **optimizes**; past tense: **optimized**; past participle: **optimized**; gerund or present participle: **optimizing**; verb: **optimise**; 3rd person present: **optimises**; past tense: **optimised**; past participle: **optimised**; gerund or present participle: **optimising**

make the best or most effective use of (a situation, opportunity, or resource).
"to optimize viewing conditions, the microscope should be correctly adjusted"

- COMPUTING
  rearrange or rewrite (data, software, etc.) to improve efficiency of retrieval or processing.

Today's Topics

- Programs from class: `/afs/ir/class/cs107/samples/lect15`
- Optimization:
  - What optimizing compilers do and don't do
  - GCC explorer: https://godbolt.org/g/3p91t2
- Memory Performance
  - How memory is organized
  - Caching
  - Impact of temporal and spatial locality
- Profiling tools
  - Measuring runtime and memory performance

A few quotes on optimization

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. ”

— Donald Knuth

“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.”

— W.A. Wulf (University of Virginia)

“Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is.”

— Rob Pike (Google, created UTF-8, the Go programming language)

Optimization Considerations

Measure, measure, measure! Time your code to see if there is even an issue. We optimize code to make it faster (or smaller) — if there isn't a problem already, don't optimize. In other words, if it works okay at the scale you care about, don't try and optimize.

For example, if the code scales well already, it probably doesn't need to be optimized further.

Use the correct algorithm and design — optimization won't change Big O or fix a bad design, and your biggest win will be because you've chosen the correct algorithms to begin with.

Optimization Considerations

Keep it simple! — Simple code that is easy to understand and debug is generally best. In this case you are optimizing the programmer's time. This is especially true for the parts of your code that aren't providing the bottleneck.

Let gcc do its optimizations — don't pre-optimize, and after you compile with a high optimization in gcc, look at the assembly code and analyze it to see where you may be able to optimize.

Optimize explicitly as a last resort — measure again, and attack the bottlenecks first.

Optimization Blockers

Programmers need to be careful to write code that can be optimized!

Although this isn't always possible, it is a good goal to have.

Let's look at two functions:

```
void twiddle1(long *xp,
              long *yp)
{
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle2(long *xp,
              long *yp)
{
    *xp += 2 * *yp;
}
```

The functions perform the same thing, right?

Optimization Blockers

```
void twiddle1(long *xp,
              long *yp)
{
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle2(long *xp,
              long *yp)
{
    *xp += 2 * *yp;
}
```

```
$ ./twiddle 2 3
a: 2, b: 3
after twiddle1(&a, &b), a = 8

a: 2, b:3
after twiddle2(&a, &b), a = 8

a: 2
after twiddle1(&a, &a), a = 8

a: 2
after twiddle2(&a, &a), a = 6
```

Oops — if the pointer is the same, we have a problem! Pointers can be *optimization blockers*. gcc won't optimize twiddle1 to twiddle2, because it could lead to incorrect code.

Optimization Examples: Constant Folding

GCC explorer: https://gcc.godbolt.org/z/vbfTjehPa

```
unsigned long CF(unsigned long val)
{
    unsigned long ones = ~0U/UCHAR_MAX;
    unsigned long highs = ones << (CHAR_BIT - 1);
    return (val - ones) & highs;
}
```

-O0

```
pushq %rbp
movq %rsp, %rbp
movq %rdi, -24(%rbp)
movq $16843009, -8(%rbp)
movq -8(%rbp), %rax
salq $7, %rax
movq %rax, -16(%rbp)
movq -24(%rbp), %rax
subq -8(%rbp), %rax
andq -16(%rbp), %rax
popq %rbp
ret
```

-O2

```
leaq -16843009(%rdi), %rax
andl $2155905152, %eax
ret
```

The compiler doesn't need to do as many real-time calculations, and *folds* the constants into two calculations.

Optimization Ex: Common Subexpression Elimination

GCC explorer: https://gcc.godbolt.org/z/5oK5PxY5s

```
int CSE(int num, int val)
{
    int a = (val + 50);
    int b = num * a - (50 + val);
    return (val + (100 / 2)) + b;
}
```

-O0

```
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl %esi, -24(%rbp)
movl -24(%rbp), %eax
addl $50, %eax
movl %eax, -4(%rbp)
movl -20(%rbp), %eax
imull -4(%rbp), %eax
movl -24(%rbp), %edx
addl $50, %edx
subl %edx, %eax
movl %eax, -8(%rbp)
movl -24(%rbp), %eax
leal 50(%rax), %edx
movl -8(%rbp), %eax
addl %edx, %eax
popq %rbp
ret
```

-O2

```
leal 50(%rsi), %eax
imull %edi, %eax
ret
```

The complier is able to *eliminate subexpressions* by determining that they are the same.

Optimization Ex: Strength Reduction

GCC explorer: https://gcc.godbolt.org/z/cW7n4G3hT

```
int SR(int a, int val)
{
    unsigned int b = 5*val;
    int c = b / (1 << val);
    return (b + c) % 2;
}
```

-O0

```
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %edx
movl %edx, %eax
sall $2, %eax
addl %edx, %eax
movl %eax, -4(%rbp)
movl -20(%rbp), %eax
movl -4(%rbp), %edx
movl %eax, %ecx
shrl %cl, %edx
movl %edx, %eax
movl %eax, -8(%rbp)
movl -8(%rbp), %edx
movl -4(%rbp), %eax
addl %edx, %eax
andl $1, %eax
popq %rbp
ret
```

-O2

```
leal (%rdi,%rdi,4), %eax
movl %edi, %ecx
movl %eax, %edx
shrl %cl, %edx
addl %edx, %eax
andl $1, %eax
ret
```

The complier replaces expensive (strong) operations (e.g., divides) with equivalent expressions that are *less strong*.

Optimization Ex: Code Motion

GCC explorer: https://gcc.godbolt.org/z/zh6YMjjb8

```c
int CM(int val)
{
    int sum = 0;
    do {
        sum += 6 + 14 * val;
    } while (sum < (9 / val));
    return sum;
}
```

-O0

```
    pushq %rbp
    movq %rsp, %rbp
    movl %edi, -20(%rbp)
    movl $0, -4(%rbp)
.L2:
    movl -20(%rbp), %eax
    addl %eax, %eax
    leal 0(,%rax,8), %edx
    subl %eax, %edx
    movl %edx, %eax
    addl $6, %eax
    addl %eax, -4(%rbp)
    movl $9, %eax
    cltd
    idivl -20(%rbp)
    cmpl -4(%rbp), %eax
    jg .L2
    movl -4(%rbp), %eax
    popq %rbp
    ret
```

-O2

```
    movl $9, %eax
    xorl %ecx, %ecx
    cltd
    idivl %edi
    imull $14, %edi, %esi
    addl $6, %esi
.L2:
    addl %esi, %ecx
    cmpl %eax, %ecx
    jl .L2
    movl %ecx, %eax
    ret
```

The compiler *moves code* out of loops if it can: it only needs to perform the operation once, so it does.

Optimization Ex: Dead Code Elimination

GCC explorer:

https://gcc.godbolt.org/z/r1eeWEarG

-O0

```
.LC0:
.string "The end of the world is near!"
DC:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movl %edi, -20(%rbp)
    movl %esi, -24(%rbp)
    movl -20(%rbp), %eax
    cmpl -24(%rbp), %eax
    jge .L2
    movl -20(%rbp), %eax
    cmpl -24(%rbp), %eax
    jle .L2
    movl $.LC0, %edi
    call puts
.L2:
    movl $0, -8(%rbp)
    jmp .L3
.L4:
    movl -4(%rbp), %eax
    imull -8(%rbp), %eax
    movl %eax, -4(%rbp)
    addl $1, -8(%rbp)
.L3:
    cmpl $999, -8(%rbp)
    jle .L4
    movl -20(%rbp), %eax
    cmpl -24(%rbp), %eax
    jne .L5
    addl $1, -20(%rbp)
    jmp .L6
.L5:
    addl $1, -20(%rbp)
.L6:
    cmpl $0, -20(%rbp)
    jne .L7
    movl $0, %eax
    jmp .L8
.L7:
    movl -20(%rbp), %eax
.L8:
    leave
    ret
```

```c
int DC(int param1, int param2)
{
    if (param1 < param2 && param1 > param2) // can this test ever be true?
    printf("The end of the world is near!\n");

    int result;
    for (int i = 0; i < 1000; i++)
        result *= i;

    if (param1 == param2) // if/else obviously same on both paths
        param1++;
    else
        param1++;

    if (param1 == 0) // if/else no-so-obviously same on both paths
        return 0;
    else
        return param1;
}
```

-O2

```
DC:
    leal 1(%rdi), %eax
    ret
```

The compiler realizes that most of the code does not perform useful work, so it just removes it!

gcc and optimization

`-O0` // faithful, literal match to C, and the best for debugging (but everything goes on the stack)

`-Og` // streamlined, but debug-friendly (we used this for most assignments and bank vault)

`-O2` // apply all acceptable optimizations

`-O3` // even more optimizations, but relies strongly on exact C specification (e.g., if you assume, for instance that signed numbers wrap, your code might break with this optimization level)

`-Os` // optimize for code size; performs the `-O2` optimizations that don't increase the code size (e.g., no function alignment)

You can see all optimizations that will be run by compiling with the following flags:

```
gcc -O3 prog.c -o prog -Q --help=optimizers
```

gcc and optimization

gcc knows the hardware you are running on, including:

- Register allocation
- Instruction choice
- Alignment

All transformations made by gcc during optimization should be legal and equivalent to your original C program.

- The compiler knows about compile time, not run time.
- The optimizations are conservative (e.g., it rarely tries to perform too much optimization with pointers, and it rarely removes a function unless it knows enough about it to do so).

gcc and optimization

How do we measure performance?

- Timers! There are a number of different ways to do it. One timer, `rtdsc`, is only available in assembly, although we can write a C program to access it by using "inline assembly" or linking to an assembly function.
- We can also use `valgrind --tool=callgrind`

You should time unoptimized vs. optimized

- mult.c
- sorts.c
- fact.c
- array.c

What gcc cannot do

gcc cannot fix algorithmic weaknesses, or big-O!

If you optimize too early yourself, gcc may not be able to figure out any further optimizations

gcc generally cannot remove function calls, nor can it "see through" pointers to determine if aliasing has occurred.

What gcc can do

Example: summing the char values in a string

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

1. What is going to cause the bottleneck for this function?

What gcc can do

Example: summing the char values in a string

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

1. What is going to cause the bottleneck for this function?

   `strlen(s)` -- it must search one character at a time!

What gcc can do

Example: summing the char values in a string

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

1. What is going to cause the bottleneck for this function?

   `strlen(s)` -- it must search one character at a time!

2. What can the compiler do about it?

What gcc can do

Example: summing the char values in a string

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

1. What is going to cause the bottleneck for this function?

   `strlen(s)` -- it must search one character at a time!

2. What can the compiler do about it?

```
int charsum2(char *s)
{
    int sum = 0;
    size_t len = strlen(s);
    for (size_t i = 0; i < len; i++) {
        sum += s[i];
    }
    return sum;
}
```

What gcc cannot do

Example: converting a string to lowercase

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

1. What is going to cause the bottleneck for this function?

What gcc cannot do

Example: converting a string to lowercase

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

1. What is going to cause the bottleneck for this function?

   `strlen(s)` -- it must search one character at at time!

What gcc cannot do

Example: converting a string to lowercase

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

1. What is going to cause the bottleneck for this function?

   `strlen(s)` -- it must search one character at at time!

2. Can the compiler move this out of the loop?

What gcc cannot do

Example: converting a string to lowercase

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

1. What is going to cause the bottleneck for this function?

   `strlen(s)` -- it must search one character at at time!

2. Can the compiler move this out of the loop?

   It cannot! Because **`s`** is changing, the compiler won't risk moving **`strlen()`** outside the loop. It can't figure out that a zero won't ever be put into the string (changing the string's length).

Hand-optimization

Let's look at some code for a vector -- this is the textbook example in chapter 5
handout: /afs/ir/class/cs107/lect15/vector_handout.pdf

Let's look at the combine functions (there are four versions). The first version is this:

```
void combine1(vec_ptr v, data_t *dest)
{
    *dest = IDENT; // 0
    for (long i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val; // OP is +
    }
}
```

Original clock-ticks per call to combine1:

- `-O1`: 11.3
- `-O2`: 9.3

What might we do to improve this function?

Hand-optimization

Let's look at some code for a vector -- this is the textbook example in chapter 5 handout: /afs/ir/class/cs107/samples/lect16/vector_handout.pdf

Let's look at the combine functions (there are four versions). The first version is this:

```
void combine1(vec_ptr v, data_t *dest)
{
    *dest = IDENT;
    for (long i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Original clock-ticks per call to combine1:

- `-O1`: 11.3
- `-O2`: 9.3

What might we do to improve this function?

Let's move the call to **`vec_length`** out of the function.

Hand-optimization

Here is combine2:

```
// move call to vec_length out of loop
void combine2(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);

    *dest = IDENT;
    for (long i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Clock-ticks for combine1 and combine2:

	comb1	comb2
`-O1`:	11.3	5.8
`-O2`:	9.3	5.8

Better! Both optimizations ended up about the same for combine2. Can we do better?

Hand-optimization

Here is combine2:

```
// move call to vec_length out of loop
void combine2(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);

    *dest = IDENT;
    for (long i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Clock-ticks for combine1 and combine2:

	comb1	comb2
`-O1`:	11.3	5.8
`-O2`:	9.3	5.8

Better! Both optimizations ended up about the same for combine2. Can we do better?

Maybe the call to **`get_vec_element`** seems like it may be causing a bottleneck. Let's look at that function.

Hand-optimization

```
// move call to vec_length out of loop
void combine2(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);

    *dest = IDENT;
    for (long i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Clock-ticks for combine1 and combine2:

	comb1	comb2
-O1:	11.3	5.8
-O2:	9.3	5.8

Here is the `get_vec_element` function:

```
/*
 * Retrieve vector element and store at dest.
 * Return 0 (out of bounds) or 1 (successful)
 */
int get_vec_element(vec_ptr v, long index,
data_t *dest)
{
    if (index < 0 || index >= v->len)
        return 0;
    *dest = v->data[index];
    return 1;
}
```

Hmm...maybe the bounds checking is the issue. Let's just get rid of the function altogether and directly access the data.

This does break some abstraction, but it is in the name of speed!

Hand-optimization

Here is combine3:

```
// direct access to vector data
void combine3(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (long i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

Clock-ticks for combine1 and combine2:

	comb1	comb2	comb3
`-O1`:	11.3	5.8	6.02
`-O2`:	9.3	5.8	1.9

It looks like the `-O1` actually got worse! It probably is about the same, actually, but our timing isn't perfect. The `-O2` did get much better, and we should look at the assembly code to see why.

Is there anything else we can do to make this faster?

Hand-optimization

Here is combine3:

```
// direct access to vector data
void combine3(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (long i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

Clock-ticks for combine1 and combine2:

	comb1	comb2	comb3
`-O1`:	11.3	5.8	6.02
`-O2`:	9.3	5.8	1.9

There are three memory references in the following line:

```
*dest = *dest OP data[i];
```

We really only need to update memory at the end of the function, so let's do it.

Hand-optimization

Here is combine4:

```
// accumulate result in local variable
void combine4(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    for (long i = 0; i < length; i++) {
        acc = acc OP data[i];
    }

    *dest = acc;
}
```

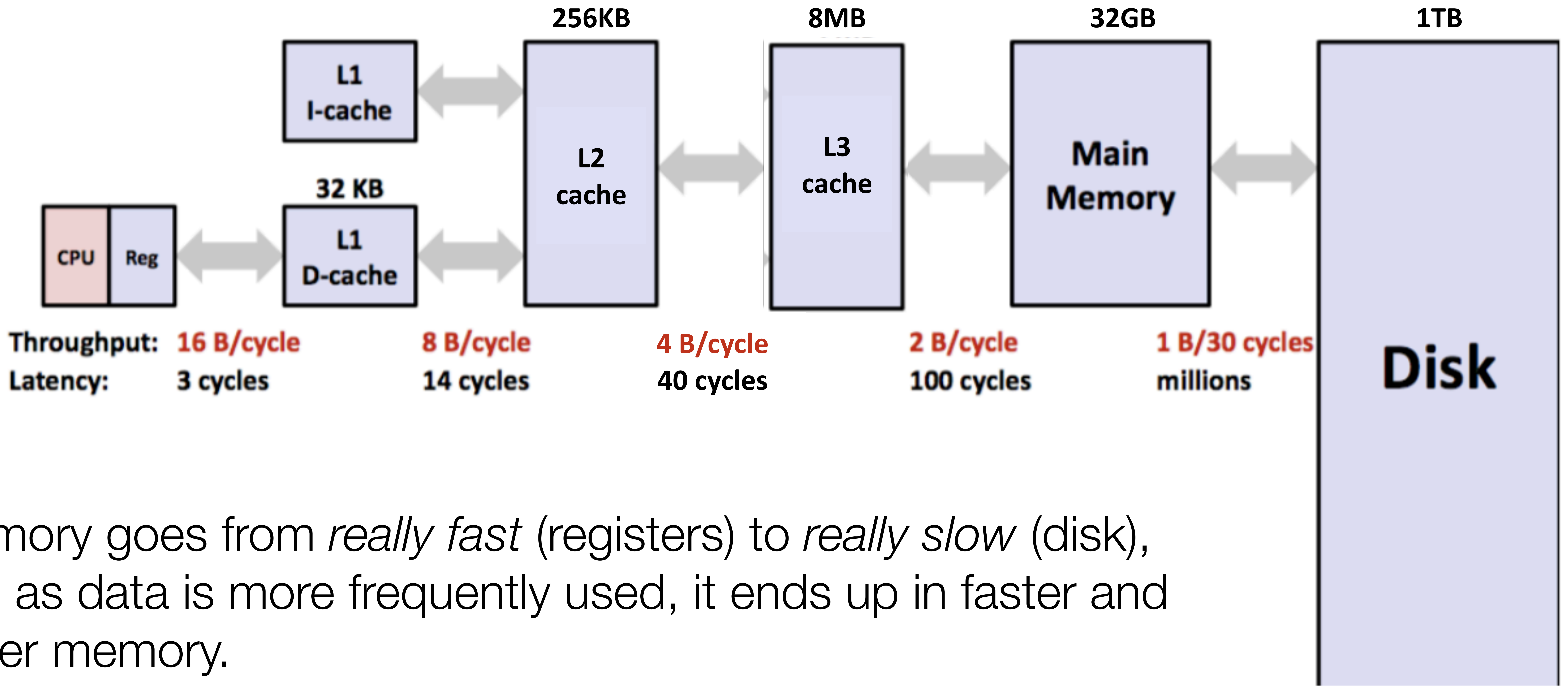
Clock-ticks for combine1 and combine2:

	comb1	comb2	comb3	comb4
`-O1`:	11.3	5.8	6.02	2.3
`-O2`:	9.3	5.8	1.9	1.5

This is much better! We were able to save 5x time in the `-O1` version, and 6x time in the `-O2` version. From the comb1 `-O1` version to the comb4 `-O2` version, we have a 7.5x improvement in time.

Caching

Computers these days have many *levels* of memory, from registers, to *cache*, to main memory, to disk memory. The myth machines, with a Core i7 CPU have the following memory structure (find out by typing `lscpu`)



Memory goes from *really fast* (registers) to *really slow* (disk), and as data is more frequently used, it ends up in faster and faster memory.

Caching and Locality

All caching depends on locality.

Temporal locality

Repeat access to same data tends to be co-located in TIME

Things I have used recently, I am likely to use again soon

Spatial locality

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

Realistic scenario:

97% cache hit rate
Cache hit costs 1 cycle
Cache miss costs 100 cycles
How much of your memory access time spent on 3% of accesses that are cache misses?

array.c

Using Callgrind

Run program under callgrind, creates file callgrind.out.pid

```
valgrind --tool=callgrind ./array_opt
```

Process file to see source annotated with count per line

```
callgrind_annotate --auto=yes callgrind.out.<pid>
```