Computer Systems

CS107

Cynthia Lee

Stanford University

Today's Topics

Today's Lecture:

- Performance optimization and the memory hierarchy

Announcements:

- **Assign7** due Friday 10th wk
- **Assign6** (optional small assignment for small amount of extra credit) goes out later today, due Friday 10th wk
  - Assign6 & 7 NO late days allowed!
  - **Submit early and often**!
    - Submit whatever you have that could be worth any points at all as soon as you have it, re-submit any time you improve it
  - That way if a catastrophe happens at deadline time and you can't submit, at least you have some points from your earlier submissions!

Valgrind for instruction profiling

Your code is awaited at the gates of Valgrindhalla! It shall ride eternal, shiny and chrome.

Remember this slide? dynamic instruction counts

For loop construction

```
# straightforward assembly
Initialization
Test
Branch past loop if fails
Body
Increment
jmp to Test
```

5n

```
# what gcc actually emits
Initialization
jmp to Test
Body
Increment
Test
Branch to Body if succeeds
```

4n

Automated tool for creating dynamic instruction counts

```
                /* Simple selection sort algorithm, O(N^2) */
                void selsort(int *arr, int n)
                {
    4,005           for (int i = 0; i < n; i++) {
    2,000               int min = i;
2,005,000               for(int j = i+1; j < n; j++)
7,003,974                   if (arr[j] < arr[min]) min = j;
   13,000               swap(&arr[i], &arr[min]);
                    }
                }
```

Valgrind can show you dynamic instruction counts

- Valgrind is a tool for memory profiling!
- But wait, there's more! Can also do other kinds of performance profiling

```
% valgrind --tool=callgrind --simulate-cache=yes ./array
% callgrind_annotate --auto=yes callgrind.out.[procID]
```

- The last argument in the first call is the name of the executable to profile *(array is only an example)*
- The last argument in the second call is the name of the output file created by the first call (procID is a number, will be different each time)
- Add --inclusive=yes if you want to include the cost of function calls (by default it only counts code actually in the current function)

Interpreting cycle counts

SORTING 1000 ELEMENTS:

- Alg-A 0.88M cycles
- Alg-B 5.31M cycles
- Alg-B 5.07M cycles (on already-sorted data)
- Alg-C 7.14M cycles
- Alg-C 0.01M cycles (on already-sorted data)

- Guess the algorithm:

A. A = quicksort, B = selection sort, C = insertion sort
B. A = quicksort, B = insertion sort, C = selection sort
C. Something else

Sorting algorithms reminder

```
/* Simple selection sort algorithm, O(N^2) */
void selsort(int *arr, int n)
{
    for (int i = 0; i < n; i++) {
        int min = i;
        for(int j = i+1; j < n; j++)
            if (arr[j] < arr[min]) min = j;
        swap(&arr[i], &arr[min]);
    }
}

/* Simple insertion sort algorithm, O(N^2) */
void inssort(int *arr, int n)
{
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && arr[j-1] > arr[j]; j--) {
            swap(&arr[j], &arr[j-1]);
        }
    }
}
```

Performance example: array access

We'll look at a few different patterns of array access

We have code that accesses an array in different ways

```
for (int i = 0; i < n; i++)      //normal access
        sum += a[i];

for (int i = n-1; i >= 0; i--) //backwards access
        sum += a[i];

for (int i = 0; i < n; i+=2)    //every other one (evens)
        sum += a[i];
for (int i = 1; i < n; i+=2)    //then odds
        sum += a[i];

for (int i = 0; i < n; i++)      //random order
        sum += a[indexes[i]];
```

You can probably guess that the punchline is that these have very different performance profiles (they do).

We have code that accesses an array in different ways

1

```
for (int i = 0; i < n; i++) //forwards
        sum += a[i];
```

2

```
for (int i = n-1; i >= 0; i--) //backwards
        sum += a[i];
```

3

```
for (int i = 0; i < n; i+=2) //evens/odds
        sum += a[i];
for (int i = 1; i < n; i+=2)
        sum += a[i];
```

4

```
for (int i = 0; i < n; i++) //random order
        sum += a[indexes[i]];
```

How will their cycle counts rank? (guess)

A. 1 < 2 < 3 < 4
B. 1 = 2 < 3 < 4
C. 1 < 2 = 3 < 4
D. 1 = 2 = 3 < 4
E. Something else

Taking a look at the assembly: FORWARD

```
0000000000400a35 <sum_forward>:
  400a35:       55                      push   %rbp
  400a36:       48 89 e5                mov    %rsp,%rbp
  400a39:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  400a3d:       89 75 e4                mov    %esi,-0x1c(%rbp)
  400a40:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400a47:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  400a4e:       eb 1d                   jmp    400a6d <sum_forward+0x38>
  400a50:       8b 45 f8                mov    -0x8(%rbp),%eax
  400a53:       48 98                   cltq
  400a55:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400a5c:       00
  400a5d:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400a61:       48 01 d0                add    %rdx,%rax
  400a64:       8b 00                   mov    (%rax),%eax
  400a66:       01 45 fc                add    %eax,-0x4(%rbp)
  400a69:       83 45 f8 01             addl   $0x1,-0x8(%rbp)
  400a6d:       8b 45 f8                mov    -0x8(%rbp),%eax
  400a70:       3b 45 e4                cmp    -0x1c(%rbp),%eax
  400a73:       7c db                   jl     400a50 <sum_forward+0x1b>
  400a75:       8b 45 fc                mov    -0x4(%rbp),%eax
  400a78:       5d                      pop    %rbp
  400a79:       c3                      retq
```

Taking a look at the assembly: BACKWARD

```
0000000000400a7a <sum_backward>:
  400a7a:       55                      push   %rbp
  400a7b:       48 89 e5                mov    %rsp,%rbp
  400a7e:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  400a82:       89 75 e4                mov    %esi,-0x1c(%rbp)
  400a85:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400a8c:       8b 45 e4                mov    -0x1c(%rbp),%eax
  400a8f:       83 e8 01                sub    $0x1,%eax
  400a92:       89 45 f8                mov    %eax,-0x8(%rbp)
  400a95:       eb 1d                   jmp    400ab4 <sum_backward+0x3a>
  400a97:       8b 45 f8                mov    -0x8(%rbp),%eax
  400a9a:       48 98                   cltq
  400a9c:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400aa3:       00
  400aa4:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400aa8:       48 01 d0                add    %rdx,%rax
  400aab:       8b 00                   mov    (%rax),%eax
  400aad:       01 45 fc                add    %eax,-0x4(%rbp)
  400ab0:       83 6d f8 01             subl   $0x1,-0x8(%rbp)
  400ab4:       83 7d f8 00             cmpl   $0x0,-0x8(%rbp)
  400ab8:       79 dd                   jns    400a97 <sum_backward+0x1d>
  400aba:       8b 45 fc                mov    -0x4(%rbp),%eax
  400abd:       5d                      pop    %rbp
  400abe:       c3                      retq
```

Taking a look at the assembly: EVEN-ODD

```
0000000000400b71 <sum_evenodd>:
  400b71:       55                      push   %rbp
  400b72:       48 89 e5                mov    %rsp,%rbp
  400b75:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  400b79:       89 75 e4                mov    %esi,-0x1c(%rbp)
  400b7c:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400b83:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  400b8a:       eb 1d                   jmp    400ba9 <sum_evenodd+0x38>
  400b8c:       8b 45 f8                mov    -0x8(%rbp),%eax
  400b8f:       48 98                   cltq
  400b91:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400b98:       00
  400b99:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400b9d:       48 01 d0                add    %rdx,%rax
  400ba0:       8b 00                   mov    (%rax),%eax
  400ba2:       01 45 fc                add    %eax,-0x4(%rbp)
  400ba5:       83 45 f8 02             addl   $0x2,-0x8(%rbp)
  400ba9:       8b 45 f8                mov    -0x8(%rbp),%eax
  400bac:       3b 45 e4                cmp    -0x1c(%rbp),%eax
  400baf:       7c db                   jl     400b8c <sum_evenodd+0x1b>
  400bb1:       c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)
  400bb8:       eb 1d                   jmp    400bd7 <sum_evenodd+0x66>
  400bba:       8b 45 f4                mov    -0xc(%rbp),%eax
  400bbd:       48 98                   cltq
  400bbf:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400bc6:       00
  400bc7:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400bcb:       48 01 d0                add    %rdx,%rax
  400bce:       8b 00                   mov    (%rax),%eax
  400bd0:       01 45 fc                add    %eax,-0x4(%rbp)
  400bd3:       83 45 f4 02             addl   $0x2,-0xc(%rbp)
  400bd7:       8b 45 f4                mov    -0xc(%rbp),%eax
  400bda:       3b 45 e4                cmp    -0x1c(%rbp),%eax
  400bdd:       7c db                   jl     400bba <sum_evenodd+0x49>
  400bdf:       8b 45 fc                mov    -0x4(%rbp),%eax
  400be2:       5d                      pop    %rbp
  400be3:       c3                      retq
```

Taking a look at the assembly: RANDOM

```
0000000000400be4 <sum_random>:
  400be4:       55                      push   %rbp
  400be5:       48 89 e5                mov    %rsp,%rbp
  400be8:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  400bec:       89 75 e4                mov    %esi,-0x1c(%rbp)
  400bef:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
  400bf3:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400bfa:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  400c01:       eb 30                   jmp    400c33 <sum_random+0x4f>
  400c03:       8b 45 f8                mov    -0x8(%rbp),%eax
  400c06:       48 98                   cltq
  400c08:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400c0f:       00
  400c10:       48 8b 45 d8             mov    -0x28(%rbp),%rax
  400c14:       48 01 d0                add    %rdx,%rax
  400c17:       8b 00                   mov    (%rax),%eax
  400c19:       48 98                   cltq
  400c1b:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400c22:       00
  400c23:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400c27:       48 01 d0                add    %rdx,%rax
  400c2a:       8b 00                   mov    (%rax),%eax
  400c2c:       01 45 fc                add    %eax,-0x4(%rbp)
  400c2f:       83 45 f8 01             addl   $0x1,-0x8(%rbp)
  400c33:       8b 45 f8                mov    -0x8(%rbp),%eax
  400c36:       3b 45 e4                cmp    -0x1c(%rbp),%eax
  400c39:       7c c8                   jl     400c03 <sum_random+0x1f>
  400c3b:       8b 45 fc                mov    -0x4(%rbp),%eax
  400c3e:       5d                      pop    %rbp
  400c3f:       c3                      retq
```

And the winner is…..

```
myth10:/usr/class/cs107/samples/lect16> ./array
This program sums a 1000000-elem array. It times the
traversal forward, backward, even/odd, vs randomly.

Sum array forward:     9.40M cycles
Sum array backward:    9.63M cycles
Sum array even/odd:    9.64M cycles
Sum array random:     36.29M cycles
```

Looking for answers in the code

```
for (int i = 0; i < n; i++) //forwards
       sum += a[i];

for (int i = n-1; i >= 0; i--) //backwards
       sum += a[i];

for (int i = 0; i < n; i+=2) //evens/odds
       sum += a[i];
for (int i = 1; i < n; i+=2)
       sum += a[i];

for (int i = 0; i < n; i++) //random order
       sum += a[indexes[i]];
```

```
forward:     9.40M cycles
backward:    9.63M cycles
even/odd:    9.64M cycles
random:     36.29M cycles
```

Does anything about this comparison strike you as particularly <u>unfair</u>?

Making the code more fair ("apples to apples" comparison)

```
for (int i = 0; i < n; i++) //forwards
        sum += a[i];

for (int i = 0; i < n; i++) //random order
        sum += a[indexes[i]];  // must access memory TWICE!

for (int i = 0; i < n; i++) //forwards, but with 2-step index
        sum += a[indexes[i]];  // must still access TWICE
```

Does anything about this comparison strike you as particularly unfair?

Taking a look at the assembly: RANDOM

```
0000000000400be4 <sum_random>:
  400be4:       55                      push   %rbp
  400be5:       48 89 e5                mov    %rsp,%rbp
  400be8:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  400bec:       89 75 e4                mov    %esi,-0x1c(%rbp)
  400bef:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
  400bf3:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400bfa:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  400c01:       eb 30                   jmp    400c33 <sum_random+0x4f>
  400c03:       8b 45 f8                mov    -0x8(%rbp),%eax
  400c06:       48 98                   cltq
  400c08:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400c0f:       00
  400c10:       48 8b 45 d8             mov    -0x28(%rbp),%rax
  400c14:       48 01 d0                add    %rdx,%rax
  400c17:       8b 00                   mov    (%rax),%eax
  400c19:       48 98                   cltq
  400c1b:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400c22:       00
  400c23:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400c27:       48 01 d0                add    %rdx,%rax
  400c2a:       8b 00                   mov    (%rax),%eax
  400c2c:       01 45 fc                add    %eax,-0x4(%rbp)
  400c2f:       83 45 f8 01             addl   $0x1,-0x8(%rbp)
  400c33:       8b 45 f8                mov    -0x8(%rbp),%eax
  400c36:       3b 45 e4                cmp    -0x1c(%rbp),%eax
  400c39:       7c c8                   jl     400c03 <sum_random+0x1f>
  400c3b:       8b 45 fc                mov    -0x4(%rbp),%eax
  400c3e:       5d                      pop    %rbp
  400c3f:       c3                      retq
```

Taking a look at the assembly: FWD-INDEX

```
0000000000400c40 <sum_fwd_ind>:
  400c40:       55                      push   %rbp
  400c41:       48 89 e5                mov    %rsp,%rbp
  400c44:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  400c48:       89 75 e4                mov    %esi,-0x1c(%rbp)
  400c4b:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
  400c4f:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400c56:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  400c5d:       eb 30                   jmp    400c8f <sum_fwd_ind+0x4f>
  400c5f:       8b 45 f8                mov    -0x8(%rbp),%eax
  400c62:       48 98                   cltq
  400c64:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400c6b:       00
  400c6c:       48 8b 45 d8             mov    -0x28(%rbp),%rax
  400c70:       48 01 d0                add    %rdx,%rax
  400c73:       8b 00                   mov    (%rax),%eax
  400c75:       48 98                   cltq
  400c77:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400c7e:       00
  400c7f:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400c83:       48 01 d0                add    %rdx,%rax
  400c86:       8b 00                   mov    (%rax),%eax
  400c88:       01 45 fc                add    %eax,-0x4(%rbp)
  400c8b:       83 45 f8 01             addl   $0x1,-0x8(%rbp)
  400c8f:       8b 45 f8                mov    -0x8(%rbp),%eax
  400c92:       3b 45 e4                cmp    -0x1c(%rbp),%eax
  400c95:       7c c8                   jl     400c5f <sum_fwd_ind+0x1f>
  400c97:       8b 45 fc                mov    -0x4(%rbp),%eax
  400c9a:       5d                      pop    %rbp
  400c9b:       c3                      retq
```

We have code that accesses an array in different ways

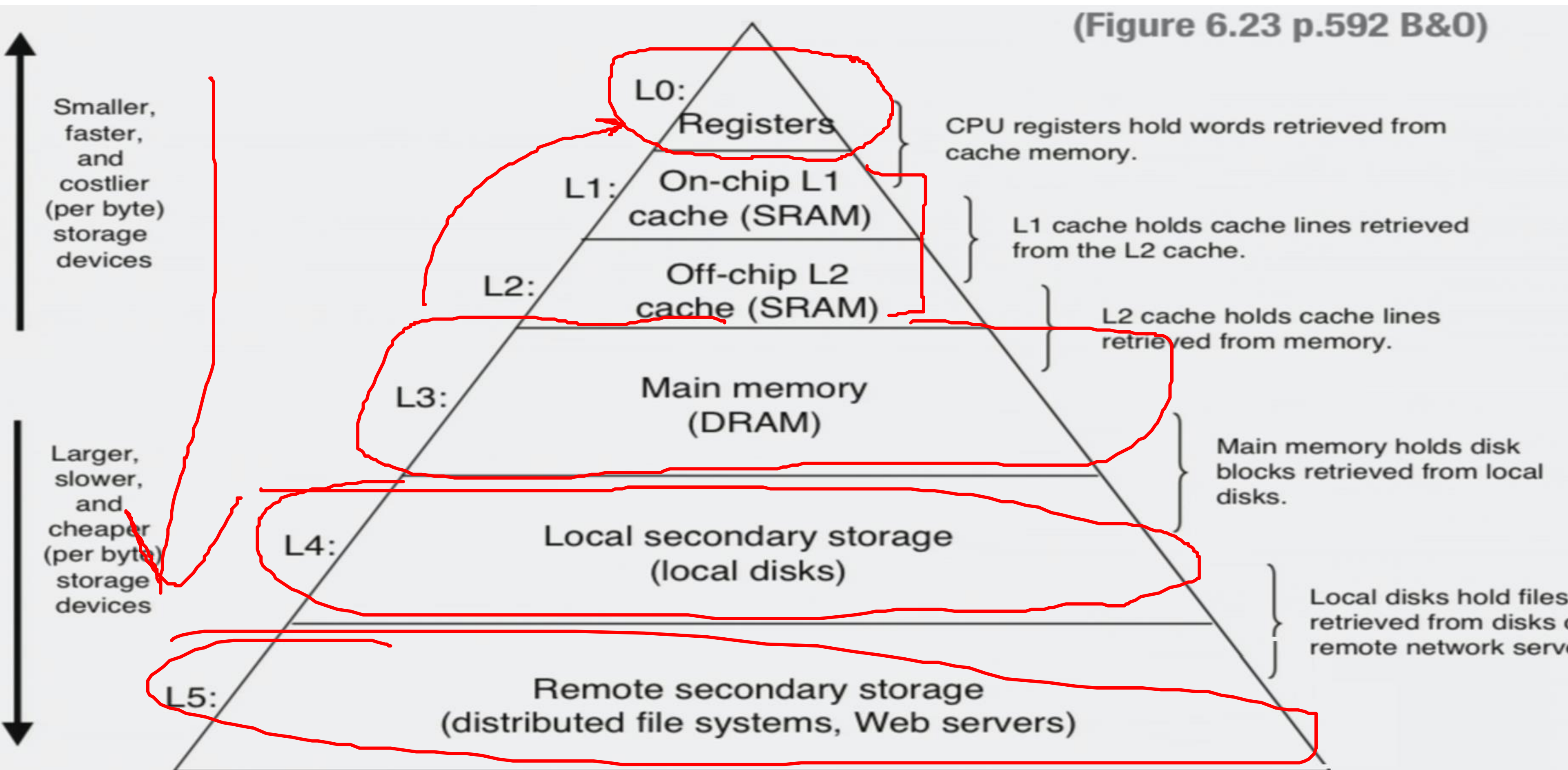
```
myth10:/usr/class/cs107/samples/lect16> ./array
This program sums a 1000000-elem array. It times the
traversal forward, backward, even/odd, vs randomly.

Sum array forward:     9.40M cycles
Sum array backward:    9.63M cycles
Sum array even/odd:    9.64M cycles
Sum array random:     36.29M cycles
Sum array fwd ind:     11.89M cycles
Sum array unrolled:    4.78M cycles
```

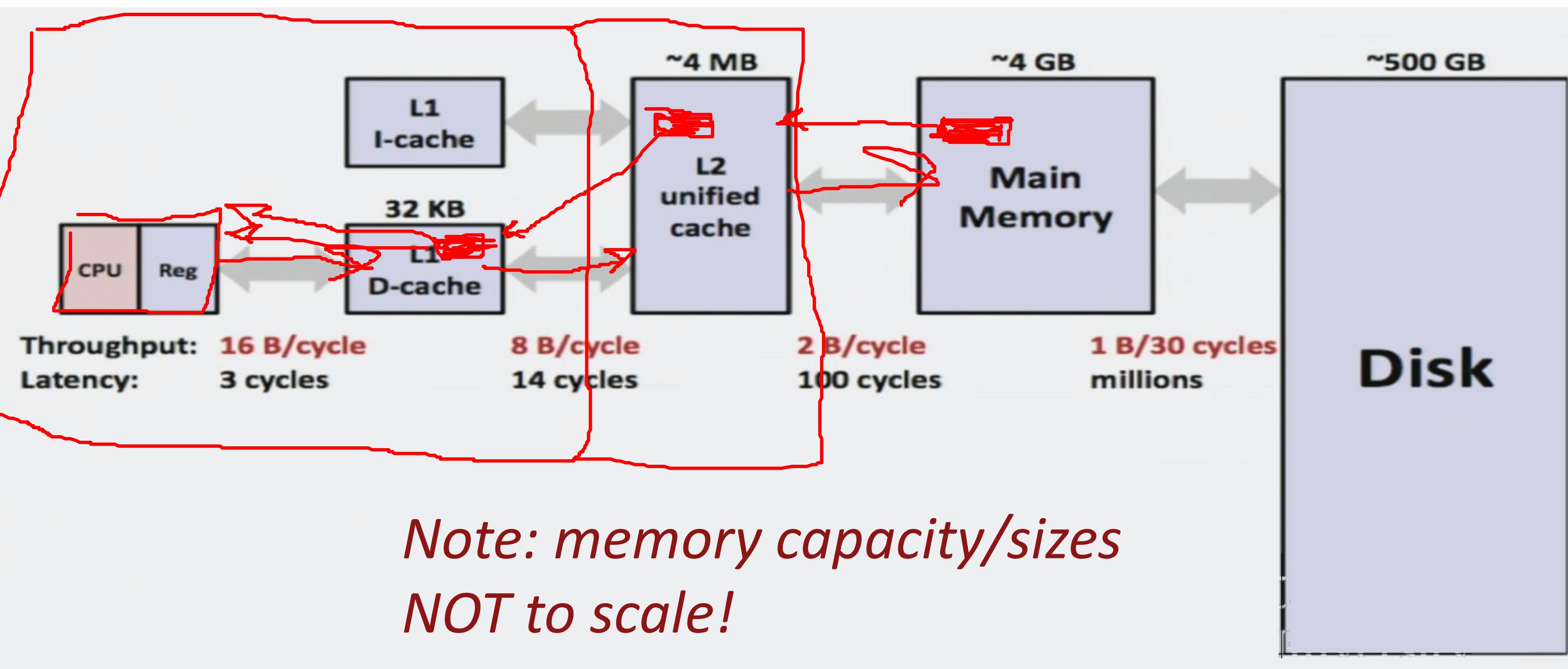
Q: Why??

A: THE MEMORY HIERARCHY

Why?? The memory hierarchy



The memory hierarchy



Mapping the Memory Hierarchy

Next few slides courtesy of my late PhD advisor Dr. Allan Snavely

Red Shift: Data keeps moving further away from the CPU with every step of Moore's Law

1000
100
10
1
0.1
0.01

nanoseconds

CPU Cycle Time

Multi Core Effective Cycle Time

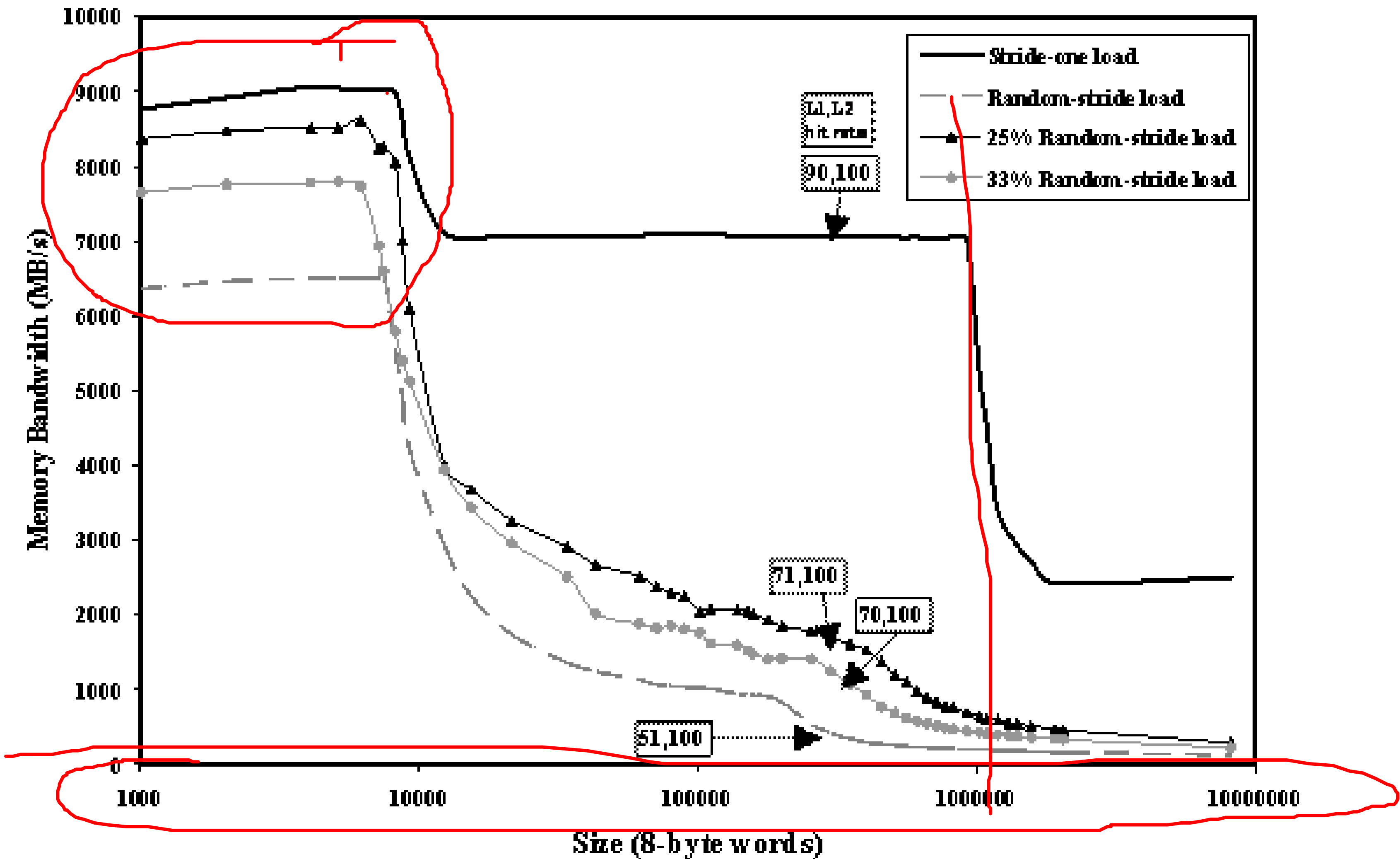
Memory Access Time

Disk Access Time

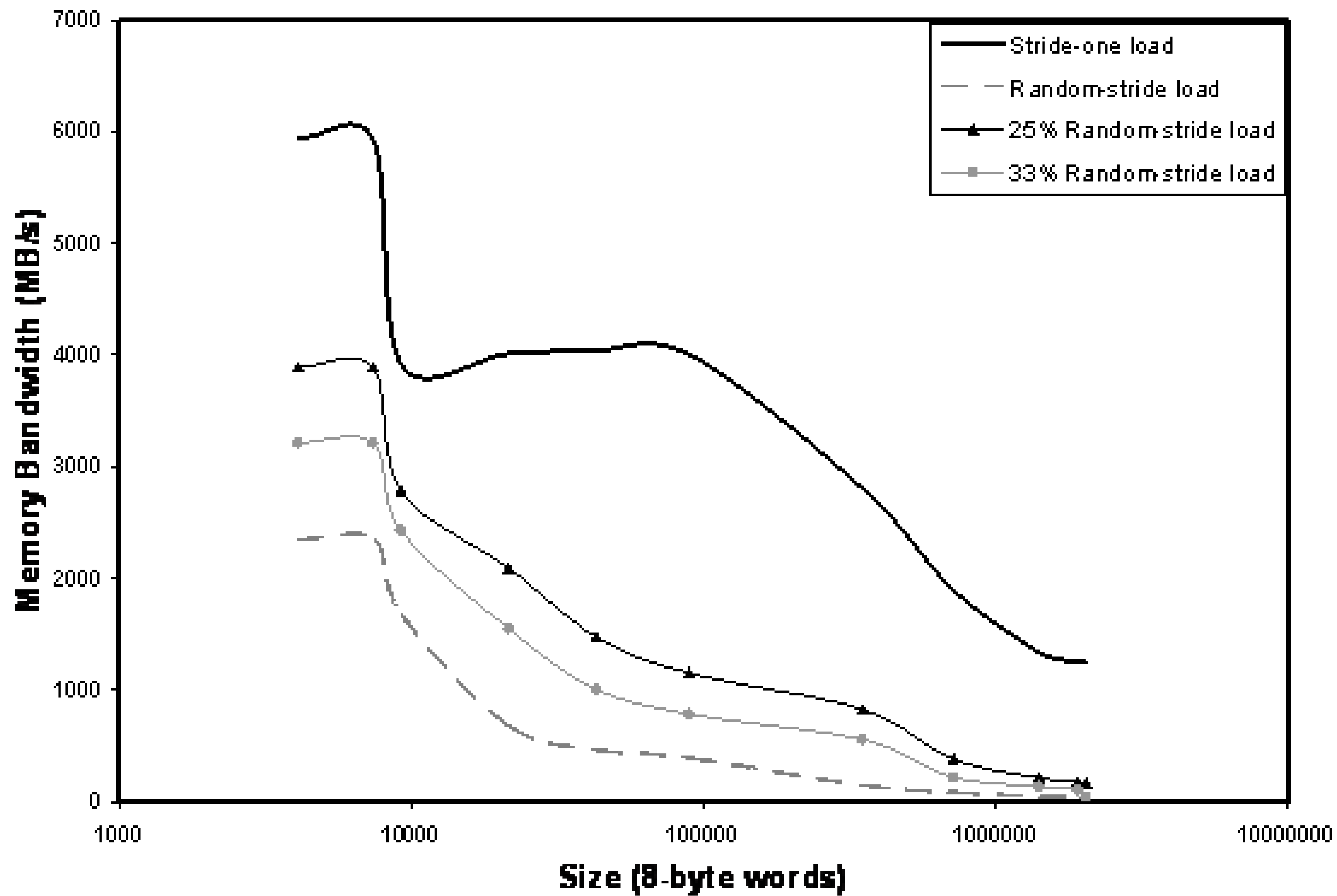
1982 1984 1985 1987 1989 1991 1993 1995 1997 1999 2001 2003 2005 2007

data due to Dean Klein of Micron

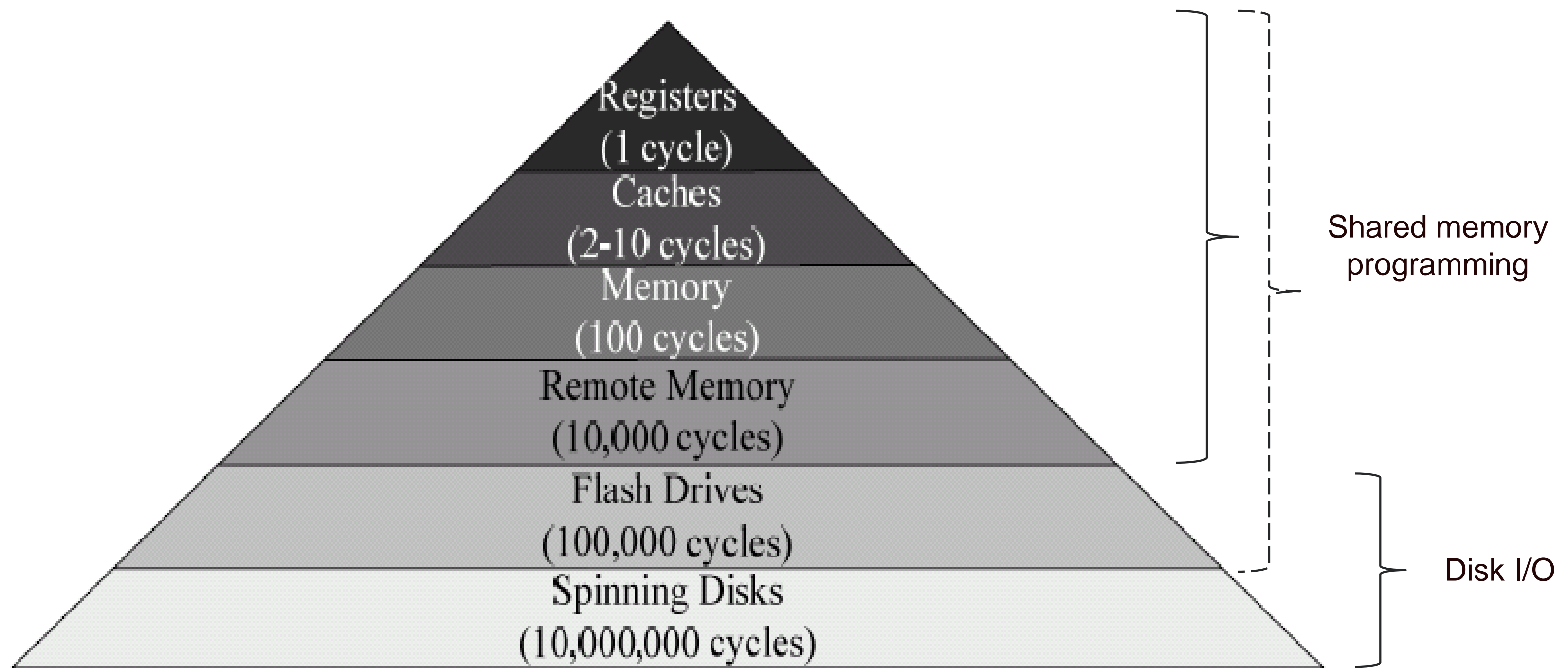
Memory Map of a Compaq Alphaserver



Memory Map of an IBM SP-3



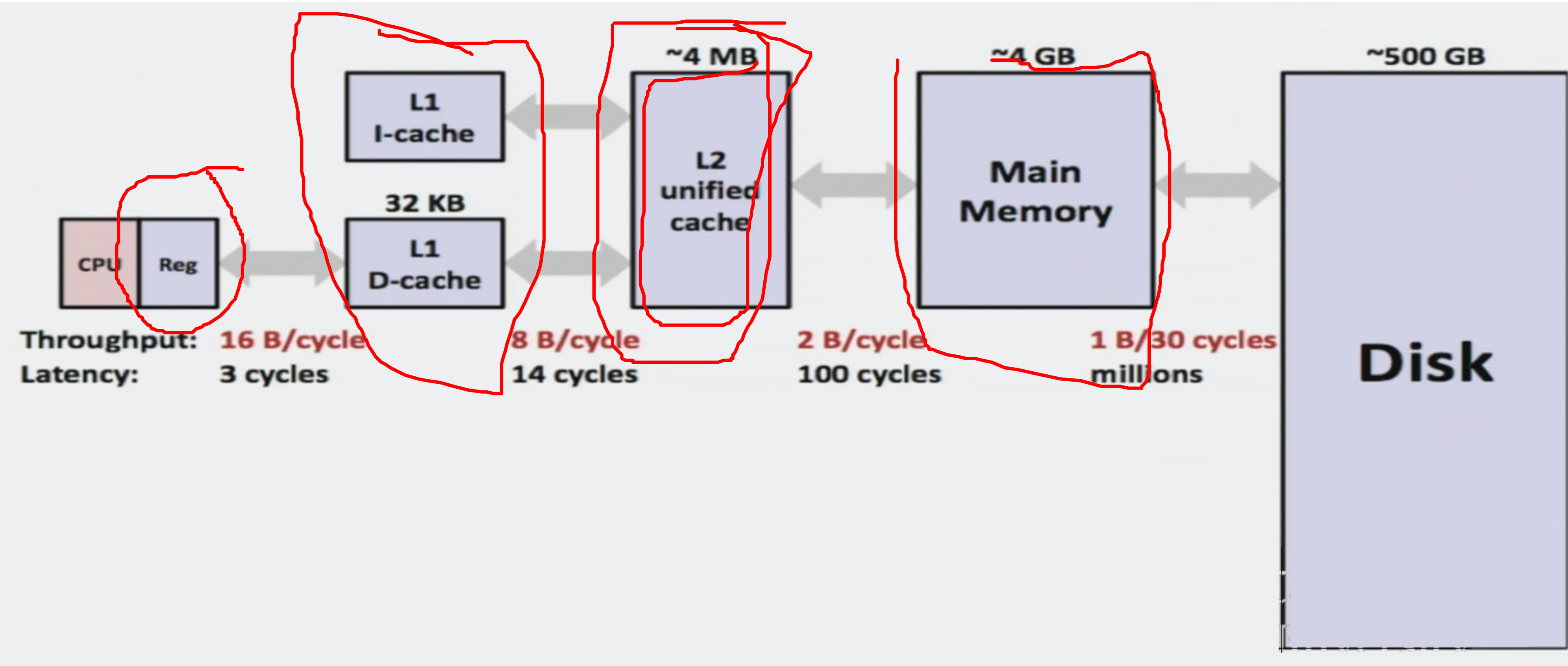
The Memory Hierarchy of Gordon (a large "supercomputer")



Caching basics

Principles of caching

The memory hierarchy



All of caching relies on LOCALITY

- **Temporal locality**:
  - Uses of the same piece of data tend to be near each other in TIME
  - Things I have recently used, I am more likely to use again
  - *Ugliest shirts that I should probably just give away are at the very bottom of my shirt drawer, becasue I never wear them*

- **Spatial locality**:
  - Uses of pieces of data tend to be near each other in SPACE
  - Even if I have never used something, if it is near a used item, it is more likely to be used soon
  - *Coat closet gets ignored during summer, but once autumn hits and I use one item from there (scarf) it is likely I will soon use other items from there (various jackets)*

Cache outcomes

- **Cache hit**:
  - What I wanted is in cache—lucky me!
  - Hit rate: % of accesses that are cache hits

- **Cache miss**:
  - What I wanted is not in cache—sad times!
  - Go to main memory (or lower level of cache) to get the item, will take much longer

Example: cache performance

- Pretty realistic scenario:
  - 97% cache hit rate
  - Cache hit: 1 cycle to access
  - Cache miss: 100 cycles to access

- **What percent of your total memory access time is spent on the 3% of memory accesses that are cache misses?**
  A. <=3%
  B. 30%
  C. 50%
  D. >50%
  - Bonus discussion: How much does this change if your cache hit rate goes up slightly to 99%?

Valgrind for cache profiling

Your code is awaited at the gates of Valgrindhalla! It shall ride eternal, shiny and chrome.

Track cache outcomes in your code with Valgrind

- To track cache hits and misses, use this command:

```
% valgrind --tool=callgrind --simulate-cache=yes ./array_opt
% callgrind_annotate --auto=yes callgrind.out.[procID]
```