

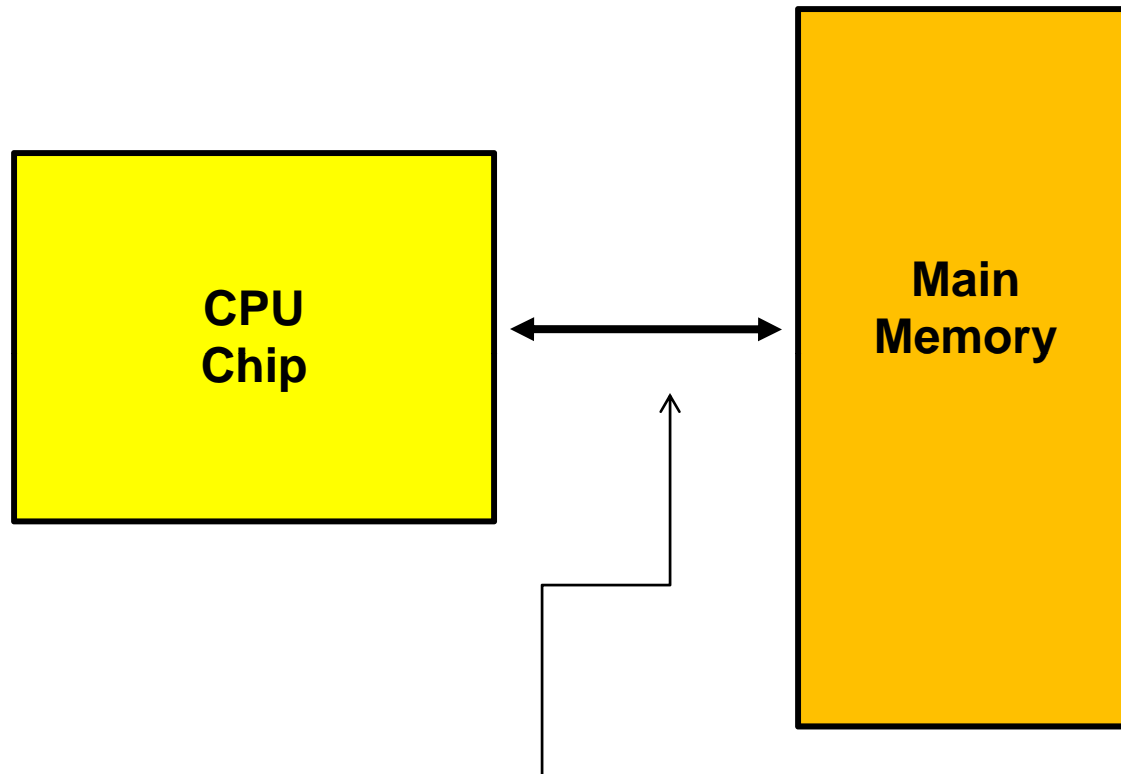
Architecture: Caching Issues in Performance

Mike Bailey

Oregon State University

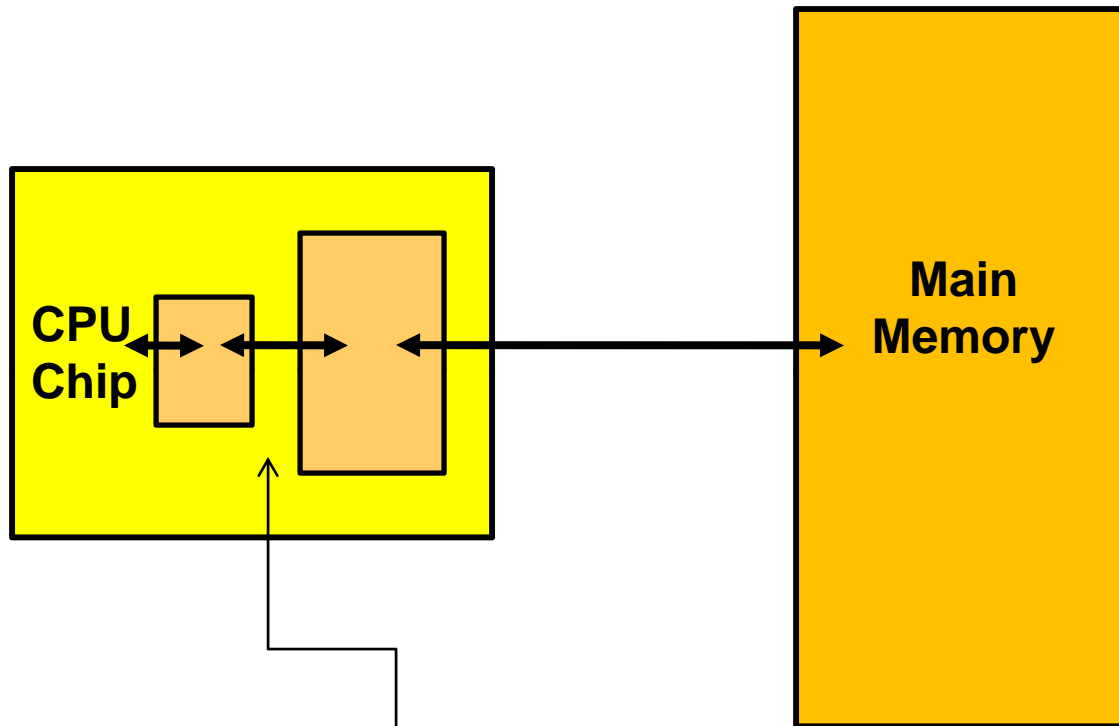


Problem: The Path Between a CPU Chip and Off-chip Memory is Slow



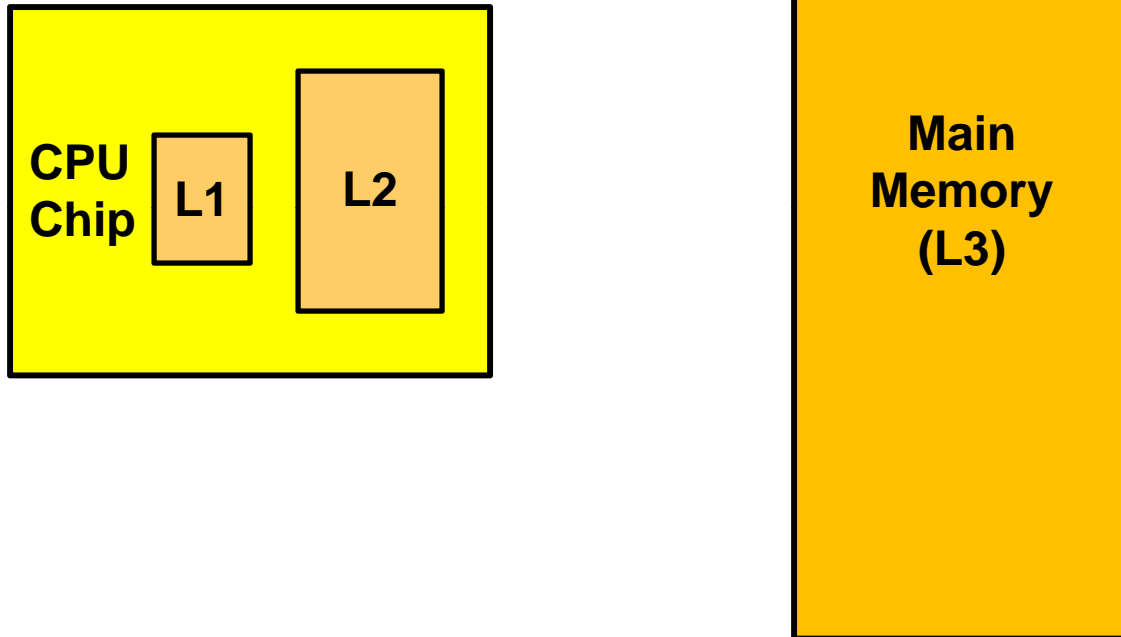
This path is relatively slow, forcing the CPU to wait for up to 200 clock cycles just to do a store to, or a load from, memory. This is a *huge* performance hit!

Solution: Hierarchical Memory Systems



The solution is to add intermediate memory systems. The one closest to the CPU is small and fast. The memory systems get slower and larger as they get farther away from the CPU.

Memory is Named by “Distance Level” from the CPU



Storage Level Characteristics

	L1	L2	L3	L4
Type of Storage	On-chip Registers	On-chip CMOS memory	Off-chip DRAM main memory	Disk
Typical Size	< 100 KB	< 8 MB	< 10 GB	Many GBs
Typical Access Time (ns)	.25 - .50	.5 – 25.0	50 - 250	5,000,000
Bandwidth (MB/sec)	50,000 – 500,000	5,000 – 20,000	2,500 – 10,000	50 - 500
Managed by	Compiler	Hardware	OS	OS

Adapted from: John Hennessy and Davis Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, 2007. (4th Edition)

Usually there are two L1 caches - one for Instructions and one for Data. You will often see this referred to in data sheets as: "L1 cache: 64KB + 64KB"



Cache Hits and Misses

When the CPU asks for a value from memory, and that value is already in the cache, it is called a **cache hit**

When the CPU asks for a value from memory, and that value is not already in the cache, it is called a **cache miss**

Performance programming should strive to avoid as many cache misses as possible. That's why it is very helpful to know the cache structure of your CPU.

Actual Cache Architecture

While cache might be multiple kilo- or megabytes, the bytes are transferred in much smaller quantities, each called a **cache line**. The size of a cache line is typically just 64 bytes.

These days, cache lines are usually arranged in something called an **n-way Set Associative Cache**. This means that each “chunk” of memory has a set of dedicated (usually 2 or 4 for L1 and 8 or 16 for L2) cache lines that it can use to page memory into and out of.

This keeps one specific chunk of memory from hogging all the cache lines. However, it also means that a chunk that your program uses much more than the other chunks will need to over-use those 2-4 or 8-16 cache lines assigned to that chunk, and underuse the other cache lines assigned to the other chunks.



Actual Cache Architecture

Let's try some typical numbers. Assume there is 1 GB of main memory, 512 KB of L2 cache, and 64-byte 16-way cache lines into the L2 cache. This means that there are

$$\frac{524,288}{64} = 8192$$

cache lines available. The “16-way” means that there are 16 cache lines per chunk of memory, so that main memory must be divided into:

$$\frac{8192}{16} = 512$$

chunks, which makes each chunk contain:

$$\frac{1GB}{512} = \frac{1,073,741,824}{512} = 2,097,152 = 2MB$$

of main memory. What does this mean to you?

Actual Cache Architecture – Take Home Message

A 2 MB memory chunk could be contained in

$$\frac{2MB}{64} = \frac{524,288}{64} = 32,768$$

cache lines, but it only has 16 available. If your program is using data and instructions from just one chunk, and they are coming from all over the chunk and going back and forth, you will have many cache misses and those 16 cache lines will keep being re-loaded and re-loaded. Performance will suffer.

If your program is using data and instructions from just one chunk, and they are coming from just a $16 \times 64 = 1024$ -byte portion of the chunk, those 16 cache lines will never need to be re-loaded. Your cache hit rate will be 100%. Performance will be much better.

To expect that is somewhat unrealistic. However, if you can arrange it so that most of the time you are traipsing through instructions or data in-order, then at least you will not have nearly as many cache misses.



How Bad Is It? -- Demonstrating the Cache-Miss Problem

C and C++ store 2D arrays a row-at-a-time, like this:

20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

For large arrays, would it be better to process the elements by row, or by column?
Which will avoid the most cache misses?

Demonstrating the Cache-Miss Problem

```
#include <stdio.h>
#include <ctime>
#include <cstdlib>

#define N    10000

float      Array[N][N];

double     Time( );

int
main( int argc, char *argv[ ] )
{
    float sum = 0.;
    double start = Time( );
    for( int i = 0; i < N; i++ )
    {
        for( int j = 0; j < N; j++ )
        {
            sum += Array[i][j];
        }
    }
    double finish = Time( );

    double col_secs = finish - start;
```



Demonstrating the Cache-Miss Problem

```
sum = 0.;
start = Time( );
for( int j = 0; j < N; j++ )
{
    for( int i = 0; i < N; i++ )
    {
        sum += Array[i][j];
    }
}
finish = Time( );

double row_secs = finish - start;
fprintf( stderr, "N = %5d ; By rows = %lf ; By cols = %lf\n", N, row_secs, col_secs );
}

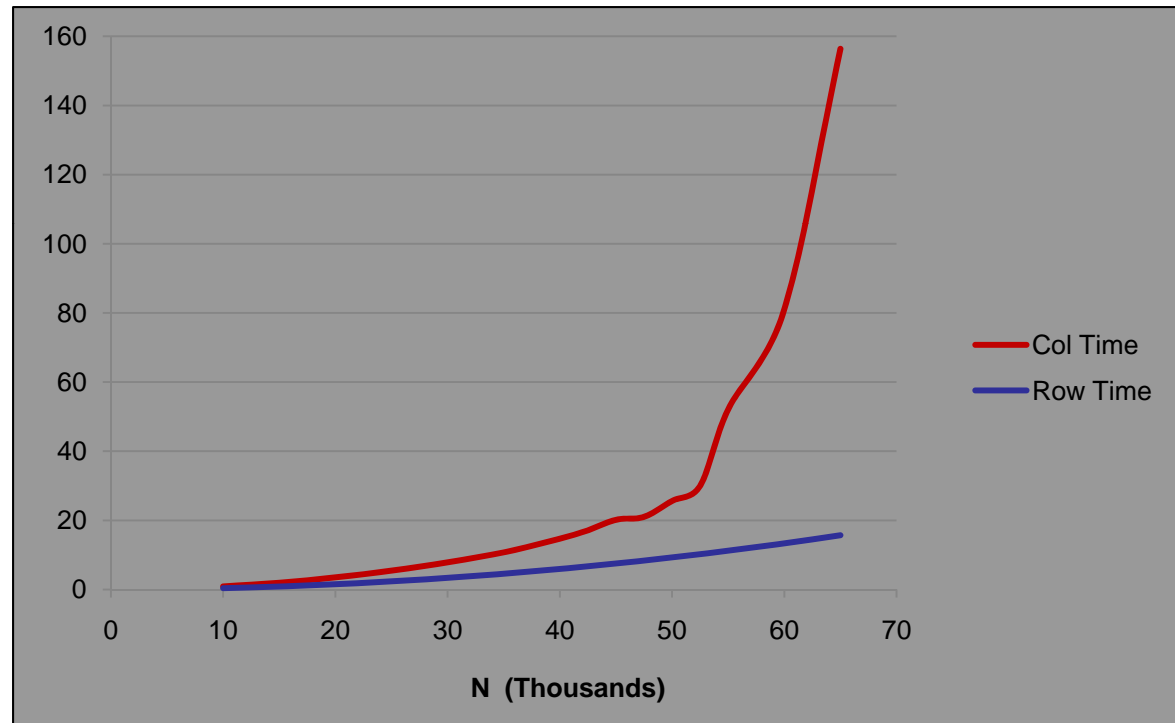
double
Time( )
{
    return (double)clock( ) / CLOCKS_PER_SEC;
}
```



Demonstrating the Cache-Miss Problem

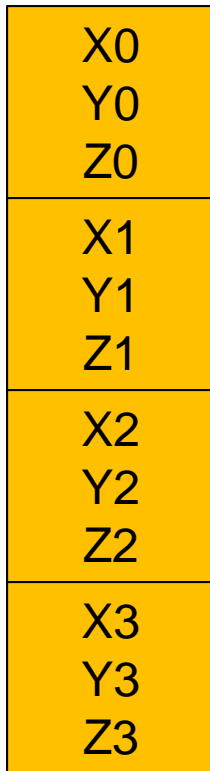
Time, in seconds, to compute the array sums, based on row-first versus column-first order:

N	Col Time	Row Time
10000	0.87	0.36
15000	1.94	0.84
20000	3.47	1.49
25000	5.45	2.32
30000	7.85	3.34
35000	10.71	4.56
40000	14.69	5.94
42500	17.12	6.7
45000	20.16	7.51
47500	20.99	8.36
50000	25.61	9.27
52500	29.99	10.22
55000	52.12	11.23
60000	80.95	13.33
65000	156.4	15.71

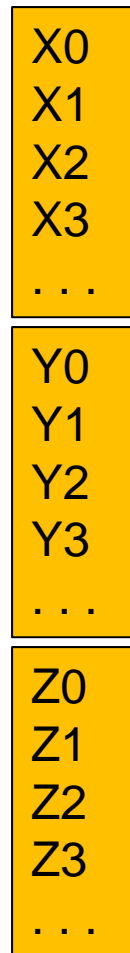


Array-of-Structures vs. Structure-of-Arrays:

```
struct xyz  
{  
    float x, y, z;  
} Array[N];
```



```
float X[N], Y[N], Z[N];
```



1. Which is a better use of the cache if we are going to be using X-Y-Z triples a lot?
2. Which is a better use of the cache if we are going to be looking at all X's, then all Y's, then all Z's?

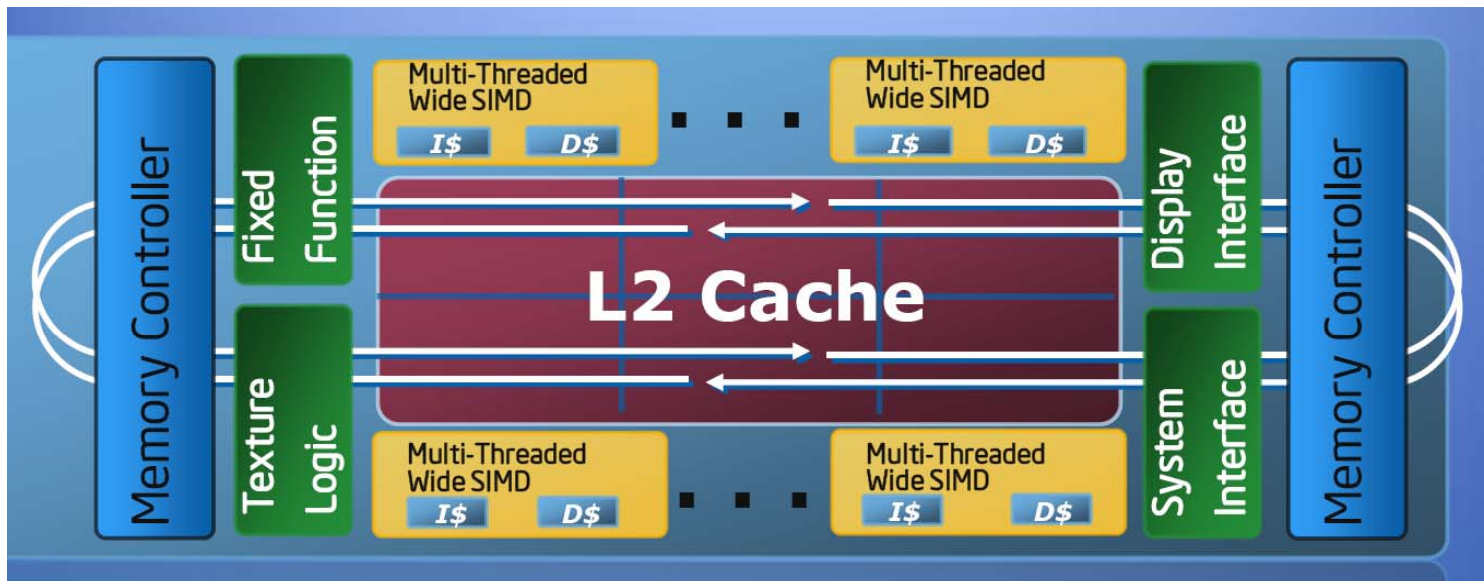
Sometimes Object-Oriented Programming is Inconsistent with Good Cache Use:

```
class xyz
{
    public:
        float x, y, z;
        xyz *next;
        xyz ( );
        static xyz *Head = NULL;
};

xyz::xyz( )
{
    xyz *n = new xyz;
    n->next = Head;
    Head = n;
};
```

This is good OO style - it encapsulates and isolates the data for this class. Once you have created a linked list whose elements could be all over memory, is it the best use of the cache?

It might be better to create an array of xyz structures and then have the constructor method pull new ones from that list. That would keep many of the elements close together while preserving the flexibility of the linked list.



Future Trend: Adding one more layer of caching on the CPU

