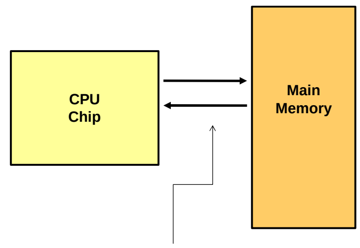
Architecture: Caching Issues in Performance

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University

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

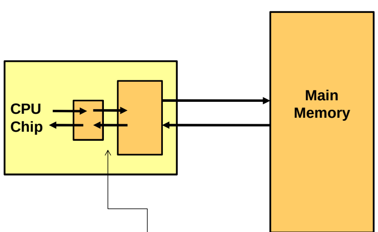


This path is relatively slow, forcing the CPU to wait for up to 200 clock cycles just to do a store to, or a load from, memory.

Depending on your CPU's ability to process instructions out-of-order, it might go idle during this time.

This is a *huge* performance hit!

Solution: Hierarchical Memory Systems, or "Cache"



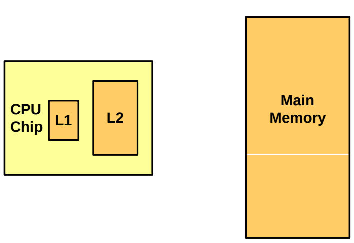
The solution is to add intermediate memory systems. The one closest to the CPU is small and fast. The memory systems get slower and larger as they get farther away from the CPU.

How is Cache Memory Actually Defined?

In computer science, a ***cache*** *is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (due to longer access time) or to compute, compared to the cost of reading the cache. In other words, a cache is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter. Cache, therefore, helps expedite data access that the CPU would otherwise need to fetch from main memory.*

-- Wikipedia

Cache and Memory are Named by "Distance Level" from the CPU



Storage Level Characteristics

	L1	L2	Memory	Disk
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 David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, 2007. (4th Edition)

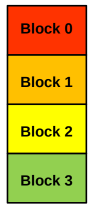
Usually there are two L1 caches – one for Instructions and one for Data. You will often see this referred to in data sheets as: "L1 cache: 64KB + 64KB" or "I and D cache"

Cache Hits and Misses

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

When the CPU asks for a value from memory, and that value is not already in the cache, it will have to go off the chip to get it.
This 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.



Memory is arranged in **blocks**. The size of a block is typically **64 Kbytes**.

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**.

Possible Cache Architectures

1. Fully Associative – cache lines from any block of memory can appear anywhere in cache.

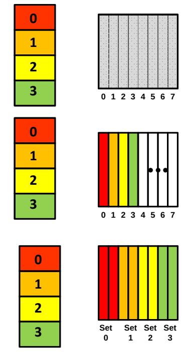
2. Direct Mapped – a cache line from a particular block of memory has only one place it could appear in cache. A memory block's cache line is:

Cache line # = Memory block # % # lines the cache has

3. N-way Set Associative – a cache line from a particular block of memory can appear in a limited number of places in cache. Each "limited place" is called a **set** of cache lines. A set contains **N** cache lines. A memory block's set number is:

Set # = Memory block # % # sets the cache has

The memory block can appear in any cache line in its set. N is typically 4 for L1 and 8 or 16 for L2.



Note that Direct Mapped can be thought of as 1-Way Set Associative.
Note that Fully Associative can be thought of as 1-Set Set-Associative.

What Happens When the Cache is Full and a New Piece of Memory Needs to Come In?

1. **Random** – randomly pick a cache line to remove
2. **Least Recently Used** (LRU) – remove the cache line which has gone unaccessed the longest
3. **Oldest** (FIFO, First-In-First-Out) – remove the cache line that has been there the longest

Actual Caches Today are N-way Set Associative

This is like a good-news / bad-news joke:

Good news: This keeps one specific block of memory from hogging all the cache lines.

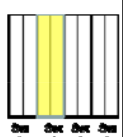
Bad news: It also means that a block that your program uses much more than the other blocks will need to over-use those N cache lines assigned to that block, and underuse the other cache lines assigned to the other blocks.

Actual Cache Architecture

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

$$\frac{512K}{64} = 8192$$

cache lines available all together in the cache. The "16-way" means that there are 16 cache lines per set, so that the cache must be divided into:



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

sets, which makes each memory block need to 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 block 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 from just one block, and they are coming from all over the block 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 from just one block, and they are coming from just a (16 cache-lines) * (64 bytes/cache-line) = 1024-byte portion of the block, 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 the 1024-situation is somewhat unrealistic. However, if you can arrange it so that most of the time you are traipsing through your data in-order, then at least you will not have nearly as many cache misses. It is the jumping around in memory that kills you.

What does this tell you about linked lists?

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

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

[j] →

[i] ↓

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

float f = Array[i][j] ;

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];  // access across a row
                }
        }
        double finish = Time( );

        double row_secs = finish – start;
```

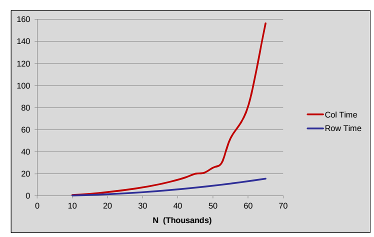

Demonstrating the Cache-Miss Problem

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

        double col_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];
```

X0 Y0 Z0 | X1 Y1 Z1 | X2 Y2 Z2 | X3 Y3 Z3

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

X0 X1 X2 X3 . . . | Y0 Y1 Y2 Y3 . . . | Z0 Z1 Z2 Z3 . . .

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 Good Object-Oriented Programming Style 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 a large 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.

When you need more, allocate another large array and link to it.

What is Brian Apgar's Complaint About Cache?

```
int ( *funcptr[ ] )( void ) =
{
        func0,
        func1,
        func2
};

. . .

for( int k = 0; k < MAX; k++ )
{
        . . .
        int i = ( funcptr[ j ] )( );
        . . .
}
```

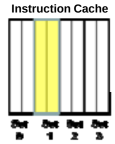
A function jump table requires two instruction cache lines:

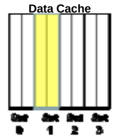
1. The for-loop statements
2. The function that really gets called

and one data cache line:

1. Your *funcptr* array

Because *you* create this table, it is in data memory.





The PS2, PSP, and Xbox 360 have 2-way associative caches . Brian talks about a tight loop of indirect function calls being a problem for these systems.

But, those systems have 2-way associative caches for *both* the instructions and the data. So, there should be enough room.

What, then, does Brian warn us about?

A Class That Inherits from a Virtual Class uses a Jump Table ("vtable") that is Contained in *Instruction* Memory

```
class Bird
{
    public:
        virtual void f( );
};

class Raven : public Bird
{
    public:
        void f( );              // overrides Bird::f( )
};

. . .

Bird    *b = new Bird( );
Raven *r = new Raven( );

Bird *birds[2] = { b, r };

for( int k = 0; k < MAX; k++ )
{
    for( int i = 0; i < 2; i++ )
    {
        birds[i]->f( );
    }
}
```

So that the correct f( ) can be called at runtime, each class has a **virtual function table**, or **vtable**, containing a pointer to its own version of f( ). Because this **vtable** is built by the compiler, it ends up living where the compiler wants it, not where you want it.

So, the virtual function jump table requires three *instruction* cache lines:

1. The for-loop statements
2. The f( ) function that really gets called
3. The virtual function table

The PS2 and PSP have 2-way associative caches. If all of these 3 items are in the same memory block, they will attempt to use the same set in the cache. But, there are only two cache lines in each set. So, every pass through the k for-loop will cause a cache miss.

(The Wii has an 8-way associative cache, so there is no similar problem there.)