Why software developers should care about CPU caches

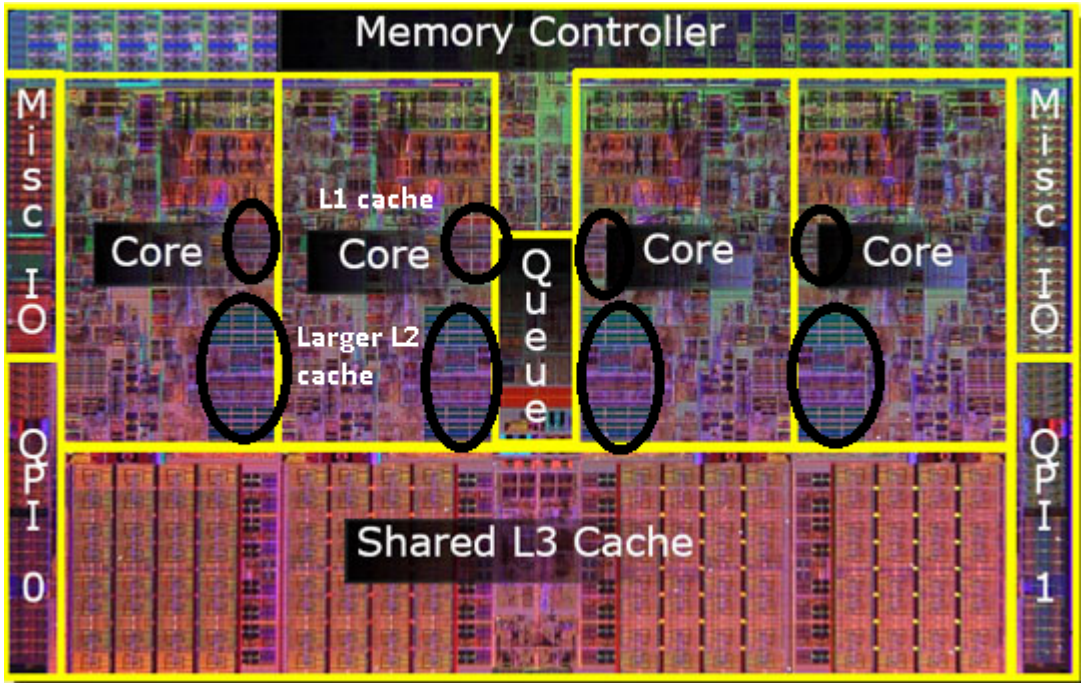
EventHelix Follow
Jul 9, 2017 · 7 min read

The memory subsystem on modern computer systems has not kept pace with the increasing processor speed. This has resulted in processor designers adding very fast cache memory to reduce the penalty of main memory access.

In many scenarios, cache access may be 27 times faster than accessing main memory. This performance difference requires a rethink on traditional optimization techniques.

Cache organization — L1, L2 and L3 cache

Let's start by looking at the layout of CPU cores and caches on a typical processor die. The figure below shows a processor with four CPU cores.



L1, L2 and L3 cache in a four core processor (credit)

Each processor core sports two levels of cache:

- 2 to 64 KB Level 1 (L1) cache very high speed cache
- ~256 KB Level 2 (L2) cache medium speed cache

All cores also share a Level 3 (L3) cache. The L3 cache tends to be around 8 MB.

Performance difference between L1, L2 and L3 caches

- L1 cache access latency: **4 cycles**
- L2 cache access latency: **11 cycles**
- L3 cache access latency: **39 cycles**
- Main memory access latency: **107 cycles**

Note here that the accessing data or code from the L1 cache is 27 times faster than accessing the data from the main memory! Due to this lopsided nature of memory access, an O(N) algorithm may perform better than an O(1) algorithm if the latter causes more cache misses.

Now lets look at performance implications of the cache structure. This will be particularly important for high performance computing and game development.

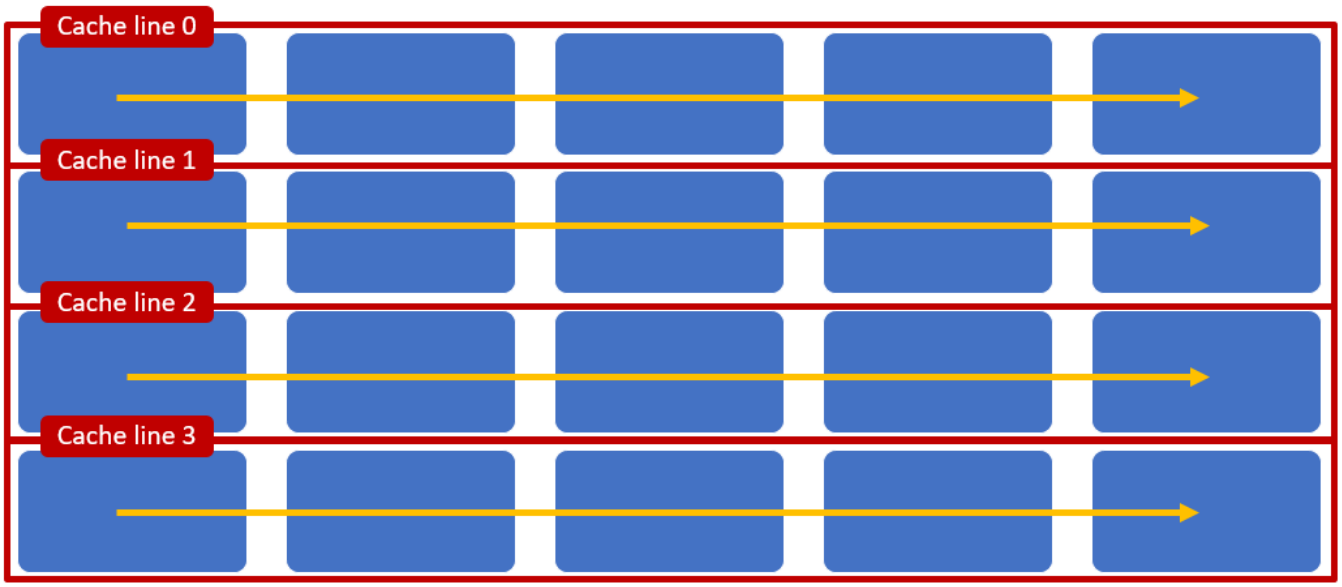
Cache line — the unit of data transfer between cache and memory

A cache line is the unit of data transfer between the cache and main memory. Typically the cache line is 64 bytes. The processor will read or write an entire cache line when any location in the 64 byte region is read or written. The processors also attempt to prefetch cache lines by analyzing the memory access pattern of a thread.

The organization of access around cache lines has important consequences for application performance. Consider the example in the following figure, an application is accessing a two dimensional array that happens to fit cache lines as shown below. A row wise access will result in:

1. Processor fetching the first row in to `Cache line 0` on the first access to the two dimensional array. Once `Cache line 0` is in the cache, the processor will be able to read subsequent items directly from the cache.

2. The processor would also prefetch `Cache line 1` even before any access is attempted to the memory area corresponding to the `Cache line 1`.

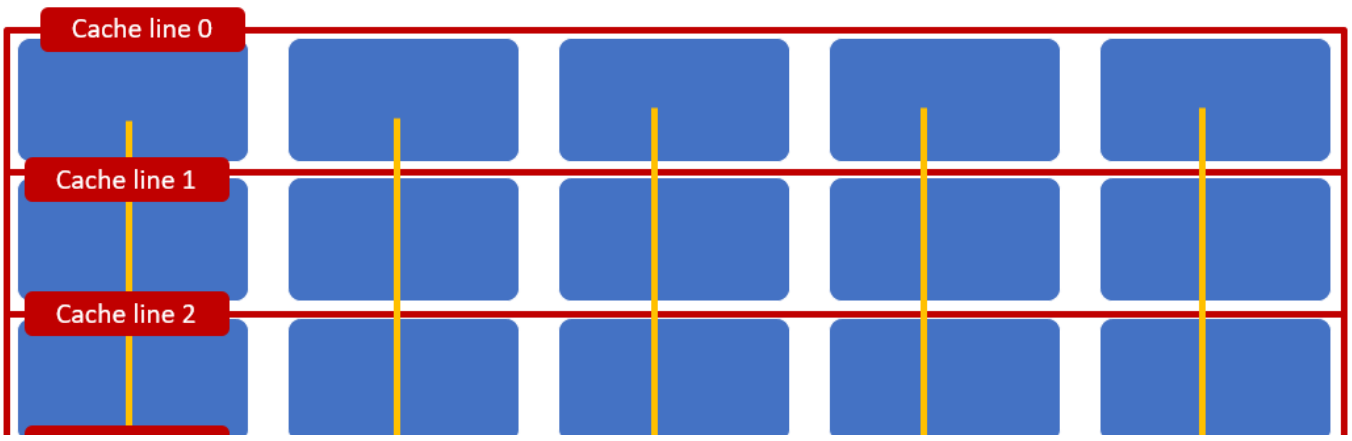


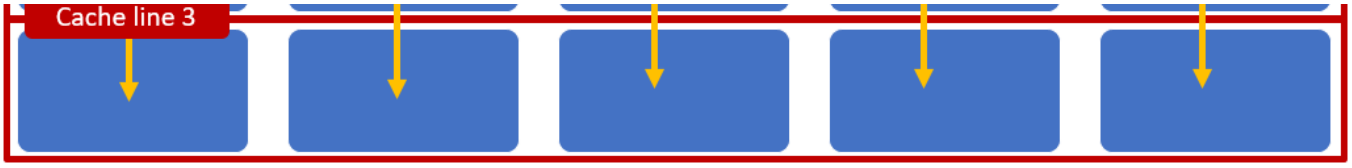
Row wise access efficiently uses the cache organization

Things do not go so well if the same data structures were accessed column wise.

1. The process starts with `Cache line 0` but soon needs to read in `Cache line 1` to `Cache line 3`. Each access incurs the overhead of main memory access.

2. If the cache is in full, it is quite possible that subsequent column wise interactions will result in repeated fetching of the cache lines.

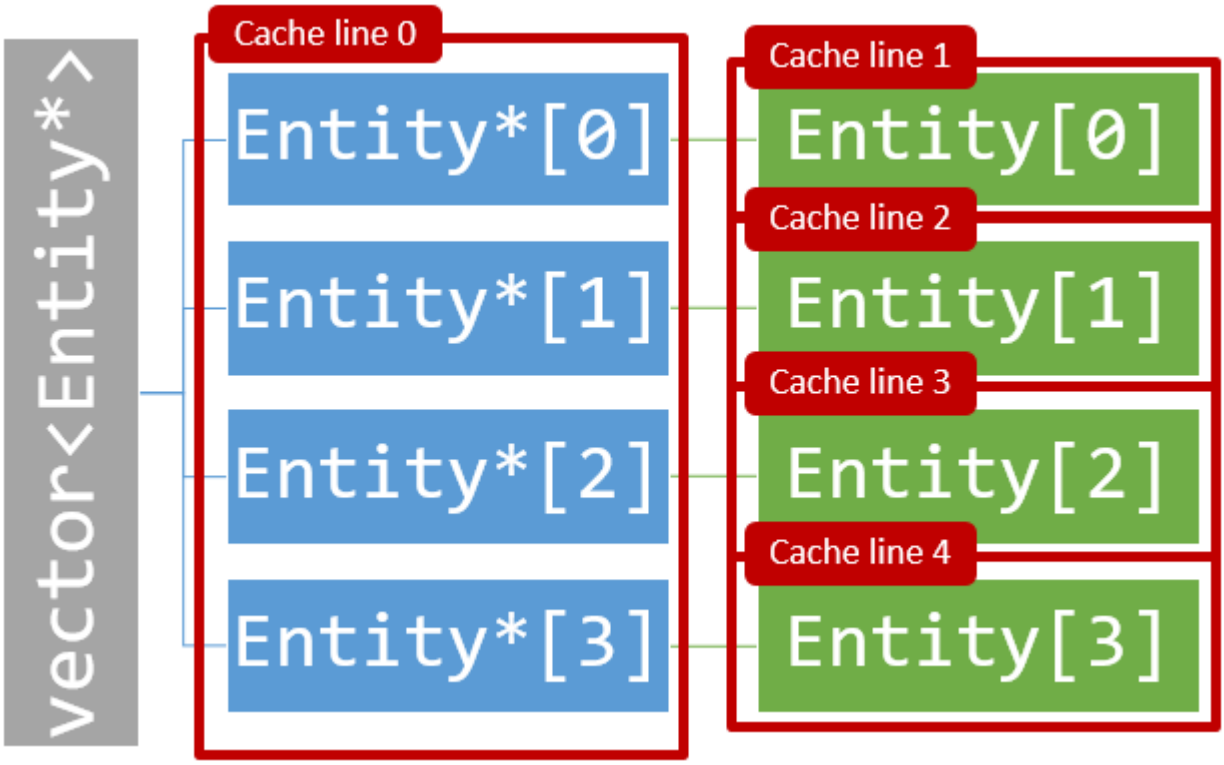




Column wise access is inefficient as it results in swapping of cache lines

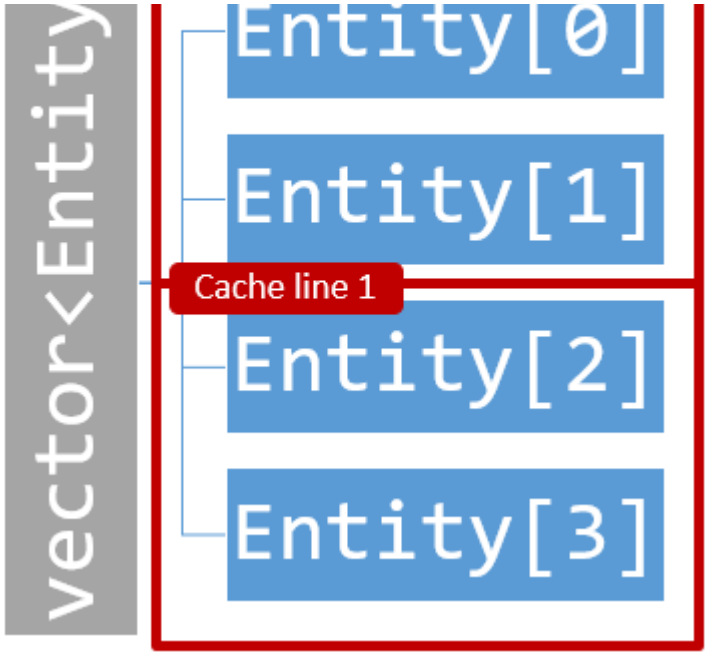
Prefer arrays and vectors stored by value

Cache organization also works better for data structures that are stored by value. Consider the cache line allocation for a vector pointers: `vector<Entity*>`. If individual `Entity` objects are allocated from the heap, it is quite likely that the `Entity` objects are all allocations are far apart in memory. This would mean access to each `Entity` will result in loading of a complete cache line.



Cache line usage in a vector of pointers

Contrast the above scenario with a vector that stores entities by value: `vector<Entity>`. In this case the `Entity` objects are organized in contiguous memory. This minimizes the load of cache lines.



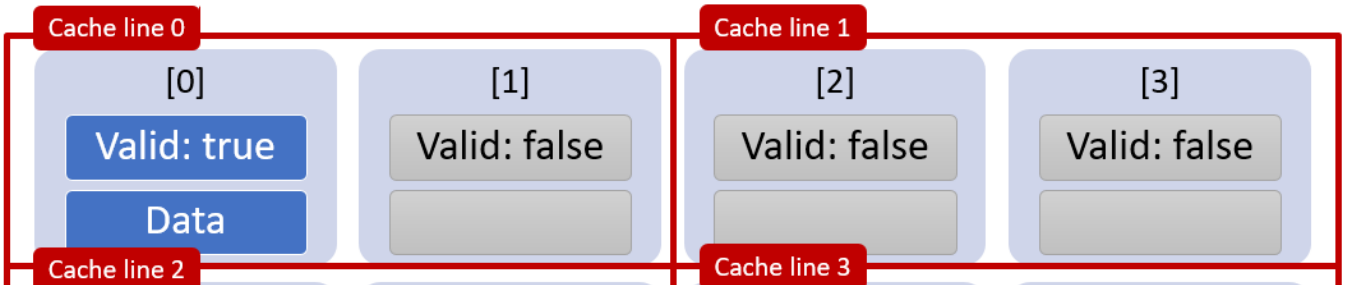
Cache line usage in a vector that stored data by value

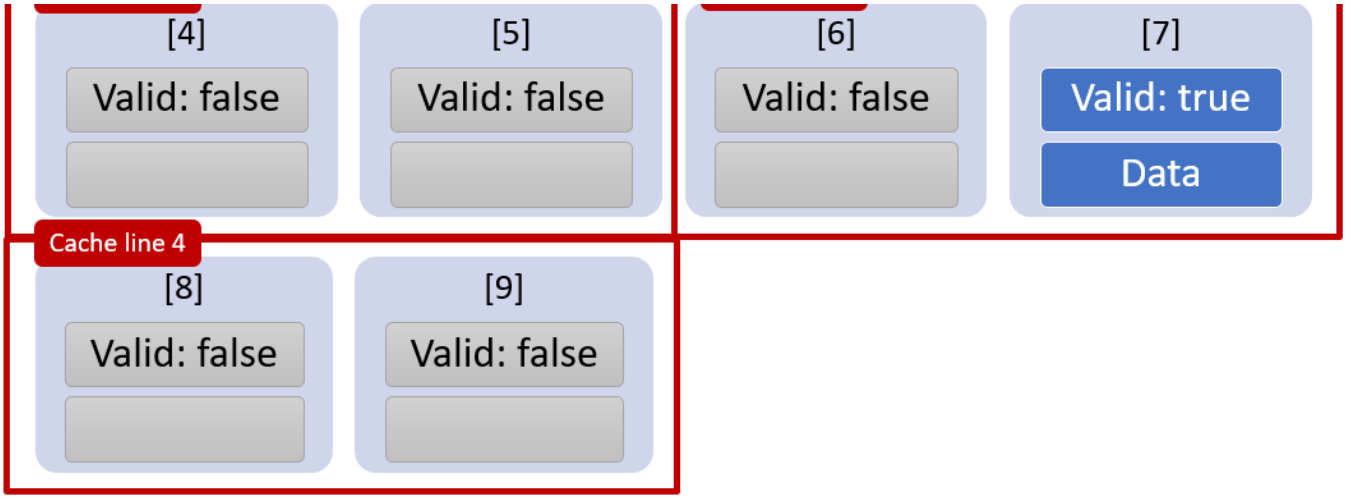
Keep array entry validity flags in separate descriptor arrays

Cache lines might also require a reevaluation of conditional access to array elements. In the following example, validity of an array element is determined by a flag inside the array element.

```
for (int i=0; i < 10; i++)
{
   if (array[i].Valid)
   {
      /* Act on array entity */
      ...
   }
}
```

The above `for` loop will result in cache line loads even for cache entries that need to be skipped (see the figure below).



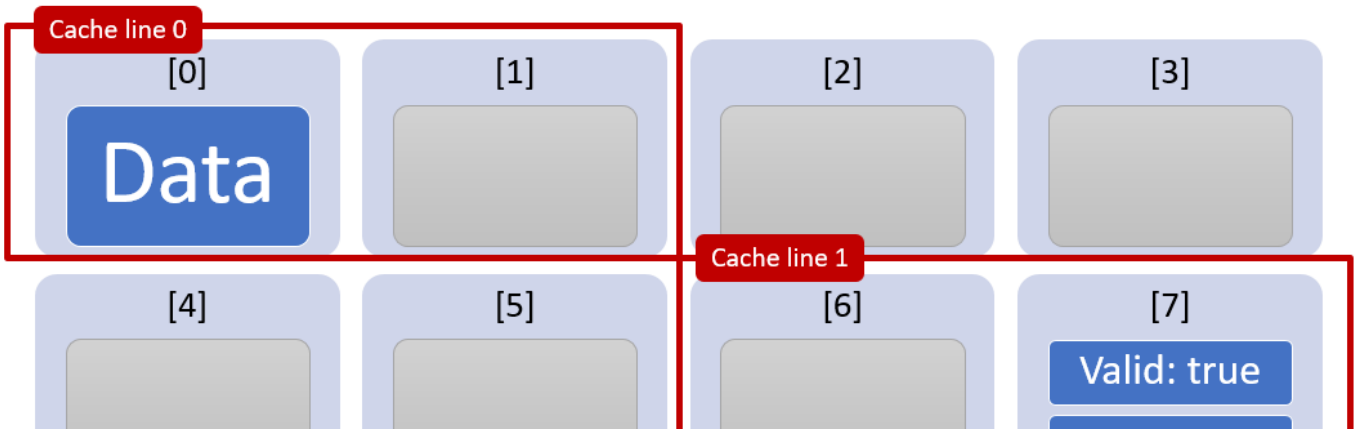


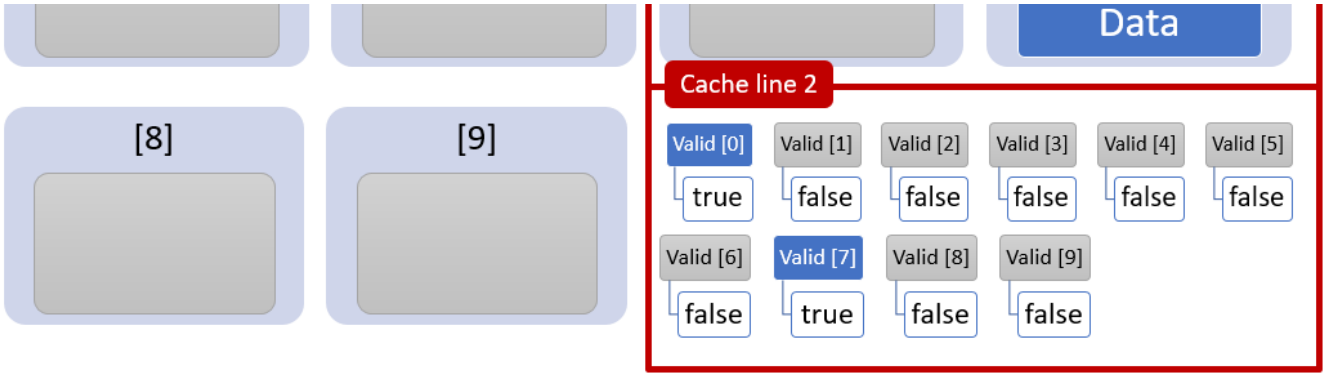
Cache behavior when the validity flag is stored inside an array

If the validity flags are moved into a separate descriptor arrays, the cache performance will improve significantly. The validity descriptors are loaded from a single cache line `Cache line 2`. The application ends up loading far fewer cache lines as cache lines for invalid array entries are skipped.

```
for (int i=0; i < 10; i++)
{
  /* The validity flags have been moved to a descriptor array */
  if (array_descriptor[i].Valid)
   {
      /* Act on array entity */
      ...
   }
}
```

You can see below, separation of the descriptor and the array content results in reduced cache line loading.





Cache behavior when the validity flags are stored in a separate descriptor

Avoid cache line sharing between threads (false sharing)

Lets have another look at the CPU die. Notice that L1 and L2 caches are per core. The processor has a shared L3 cache. This three tier cache architecture causes cache coherency issues between the caches.

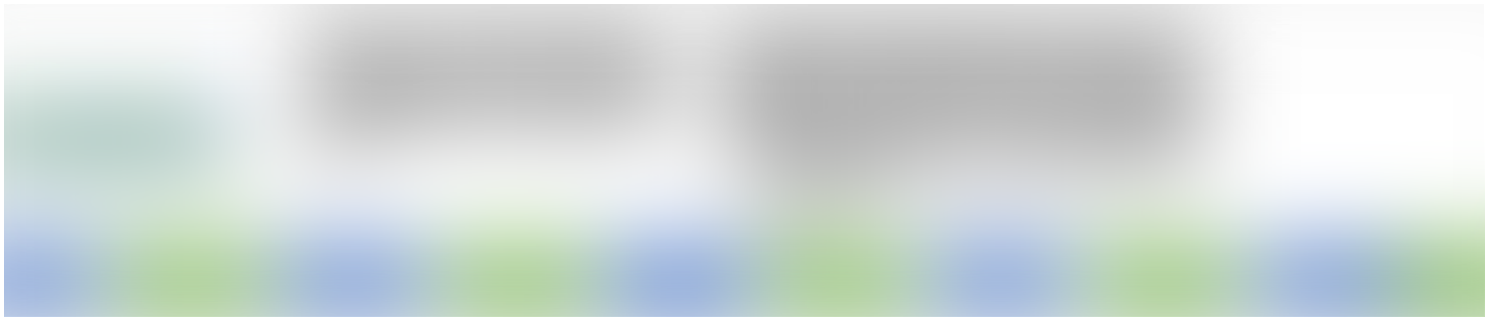
Let's assume that the left most processor reads a memory location that results in loading of `Cache line 0` into the L1 cache of the left most core. If (say) the right most core attempts to access a memory location that also happens to be on `Cache line 0`. When this access takes place, the processor will have to trigger the cache coherency procedure to ensure that `Cache line 0` is coherently represented across all caches. This is an expensive procedure that wastes a lot of memory bandwidth, thus reducing the benefit of caching memory.

L1, L2 and L3 cache (credit)

The following example shows how a caching optimized application may suffer from the cache coherency induced lag. Consider the array show below that has been optimized so that two cores (blue and green) are operating on alternate entries in the array.

This approach would give us a double the throughput of a single core but for the shared descriptors. Here the descriptor array is small so it fits on a single cache line. The green and the blue processor are working on alternate entries in the descriptor and accessing distinct memory addresses. The problem is that they are sharing a cache line. This triggers cache coherency procedures between green and blue cores. This approach is referred to as false sharing.

This problem can be easily solved by making sure the descriptors for green and blue cores occupy different cache lines.



Example of false sharing due to two cores sharing a single cache line

What about code caching?

We have focused on data caching in this article. Code caching plays an equally important role in improving performance.

Short loops fit in code caches

Code with tight `for` loops is likely to fit into the L1 or L2 cache. This speeds up program executing as CPU cycles are not wasted in repeatedly fetching the same instructions from main memory.

Function in-lining

Consider the following code fragment. Here `foo()` is calling a a function `bar()` in a loop.

```
void bar(int index)
{
    // Code Block 1
    // Code Block 2: Uses the index parameter
}

void foo()
{
   for (int i=0; i < MAX_VALUE; i++)
   {
      bar(i);
      // More code
   }
}
```

If the compiler in-lines the function `bar()` it will remove a function call from the loop, thus improving the locality of the code. As an added bonus, the compiler might also bring `//Code Block 1` outside the loop if it finds that the computation in the code block does not change in the loop.

The code after these optimizations is shown below. This code in the for loop might be small enough to execute from the L1 cache.

```
void foo()
{
   // Code Block 1
   for (int i=0; i < MAX_VALUE; i++)
   {
      // Code Block 2: Uses the loop variable i
      // More code
   }
}
```

Explore more

@Scott__Meyers's talk about caching was an inspiration for this article. Watch it to learn more.

Why software developers should care about CPU cache organization

Programming Cache Optimization Game Development