[**False Sharing-the silent performance killer**](http://robsjava.blogspot.com/2014/03/what-is-false-sharing.html)

#### Cache Line Overview

CPU's don't read memory in single bytes, rather they read 'chunks of memory' usually in blocks of 64 bytes, these chunks are referred to as cache lines.  
  
If you had two threads ( let's call them Thread 1 and Thread 2  ) both modifying a volatile variable, which we shall call ‘x' : 

volatile long x;

If Thread 1 was to change the value of ‘x’, and then Thread 2  was to read it :

**Thread 1:**x=3;

**Thread 2:**System.out.print(x);

For the value of x to be passed between the two threads ( Thread 1 to Thread 2  ) a whole 64 bytes will be exchanged, as cores only exchange data in cache lines. It is possible that Thread 1 and Thread 2, may actually be processed on the same core, but for this over simplified example lets assume that each thread is processed on its own core.

Given that long values are stored in 8 bytes, and in our example our cache line is 64 bytes, then the cache line could store 8 longs, we already have one long value of 'x' stored in the cache line, lets assume that the rest of the cache line was full of 7 other longs, for example v1 to v7

x, v1, v2, v3, v4, v5 ,v6 ,v7

#### False Sharing

This cache line could be used by a number of different threads. If another thread was to modify v2, this would then force Thread 1 and Thread 2  to reload the cache line. You maybe wondering why should Thread 1 and Thread 2 reload this cache line, as the update to v2 should not effect them. Well, even though these updates are logically independent of each other, coherence is maintained on a cache-line basis, and not on individual elements. This apparent unnecessary sharing of data is referred to as false sharing.

A core can execute hundreds of instructions in the time taken to fetch a single cache line.  
If a core has to wait for a cache line to be reloaded, the core will run out of things to do, this is called a stall. Stalls can be avoided by reducing false sharing, one technique to reduce false sharing is to pad out data structure so that threads working on independent variables fall in separate cache lines.  
 Another problem about false sharing :

|  |
| --- |
| <http://1.bp.blogspot.com/-MYso5dR5ItE/TjOy1Mbr3NI/AAAAAAAAAAk/uLPjSeGhbmg/s1600/cache-line.png> |
| Figure 1. |

Figure 1. above illustrates the issue of false sharing.  A thread running on core 1 wants to update variable X while a thread on core 2 wants to update variable Y.  Unfortunately these two hot variables reside in the same cache line.  Each thread will race for ownership of the cache line so they can update it.  If core 1 gets ownership then the cache sub-system will need to invalidate the corresponding cache line for core 2.  When Core 2 gets ownership and performs its update, then core 1 will be told to invalidate its copy of the cache line.  This will ping pong back and forth via the L3 cache greatly impacting performance.  The issue would be further exacerbated if competing cores are on different sockets and additionally have to cross the socket interconnect.

#### Padding

An example of a padded class, attempting to place 'x' and 'v1' on separate cache lines :

**public class** FalseSharingWithPadding {

**public volatile long** x;

**public volatile long** p2; *// padding*

**public volatile long** p3; *// padding*

**public volatile long** p4; *// padding*

**public volatile long** p5; *// padding*

**public volatile long** p6; *// padding*

**public volatile long** p7; *// padding*

**public volatile long** p8; *// padding*

**public volatile long** v1;

}

Before you go ahead an pad all your data structures its worth bearing in mind that the JVM can eliminate or re-order unused fields, thus re-introducing false sharing. Also there is no guarantee where objects will be placed on the heap.   
  
To reduce the chance of your unused padding fields from being eliminated, it usually helps if you set them volatile. I suggest you only apply padding to highly contended concurrent classes and then, only if profiling on your target architecture, actually shows a difference. Usually best to do this after at least 10,000 iterations, to eliminate the effects of JVM realtime optimisations.

**Results**  
  
Running the above code while ramping the number of threads and adding/removing the cache line padding,  I get the results depicted in Figure 2. below.  This is measuring the duration of test runs on my 4-core Nehalem.

|  |
| --- |
| <http://2.bp.blogspot.com/-gi_3N6LC26E/TjOzSGIUMxI/AAAAAAAAAAo/5-UaNKmGZzY/s1600/duration.png> |
| Figure 2. |

The impact of false sharing can clearly be seen by the increased execution time required to complete the test.  Without the cache line contention we achieve near linear scale up with threads.  
  
This is not a perfect test because we cannot be sure where the *VolatileLongs*will be laid out in memory.  They are independent objects.  However experience shows that objects allocated at the same time tend to be co-located.  
  
So there you have it.  False sharing can be a silent performance killer.