This Blog Has Moved!

February 04, 2021

Right, so yes, five years ago I moved to github pages, and never bothered to redirect any of these pages there. Now I've moved on from there, and... Finally I am using my real domain, trishagee.com . My blog is now at trishagee.com/blog . See you there!

Post a Comment

Dissecting the Disruptor: Why it's so fast (part two) - Magic cache line padding

July 22, 2011

We mention the phrase Mechanical Sympathy quite a lot, in fact it's even Martin's blog title. It's about understanding how the underlying hardware operates and programming in a way that works with that, not against it.

We get a number of comments and questions about the mysterious cache line padding in the RingBuffer, and I referred to it in the last post. Since this lends itself to pretty pictures, it's the next thing I thought I would tackle.

Comp Sci 101

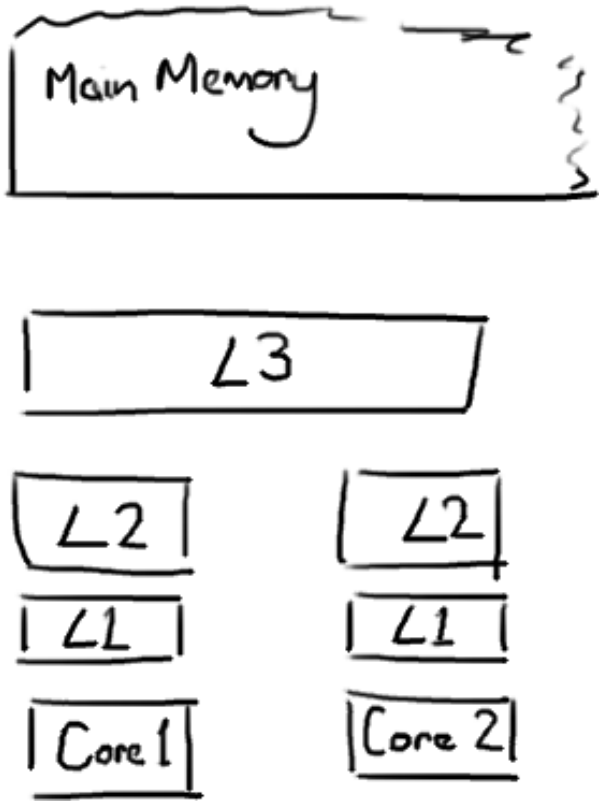
One of the things I love about working at LMAX is all that stuff I learnt at university and in my A Level Computing actually means something. So often as a developer you can get away with not understanding the CPU, data structures or Big O notation - I spent 10 years of my career forgetting all that. But it turns out that if you do know about these things, and you apply that knowledge, you can come up with some very clever, very fast code.

So, a refresher for those of us who studied this at school, and an intro for those who didn't.

Beware - this post contains massive over-simplifications.

The CPU is the heart of your machine and the thing that ultimately has to do all the operations, executing your program. Main memory (RAM) is where your data (including the lines of your program) lives. We're going to ignore stuff like hard drives and networks here because the Disruptor is aimed at running as much as possible in memory.

The CPU has several layers of cache between it and main memory, because even accessing main memory is too slow. If you're doing the same operation on a piece of data multiple times, it makes sense to load this into a place very close to the CPU when it's performing the operation (think a loop counter - you don't want to be going off to main memory to fetch this to increment it every time you loop around).



The closer the cache is to the CPU, the faster it is and the smaller it is. L1 cache is small and very fast, and right next to the core that uses it. L2 is bigger and slower, and still only used by a single core. L3 is more common with modern multi-core machines, and is bigger again, slower again, and shared across cores on a single socket. Finally you have main memory, which is shared across all cores and all sockets.

When the CPU is performing an operation, it's first going to look in L1 for the data it needs, then L2, then L3, and finally if it's not in any of the caches the data needs to be fetched all the way from main memory. The further it has to go, the longer the operation will take. So if you're doing something very frequently, you want to make sure that data is in L1 cache.

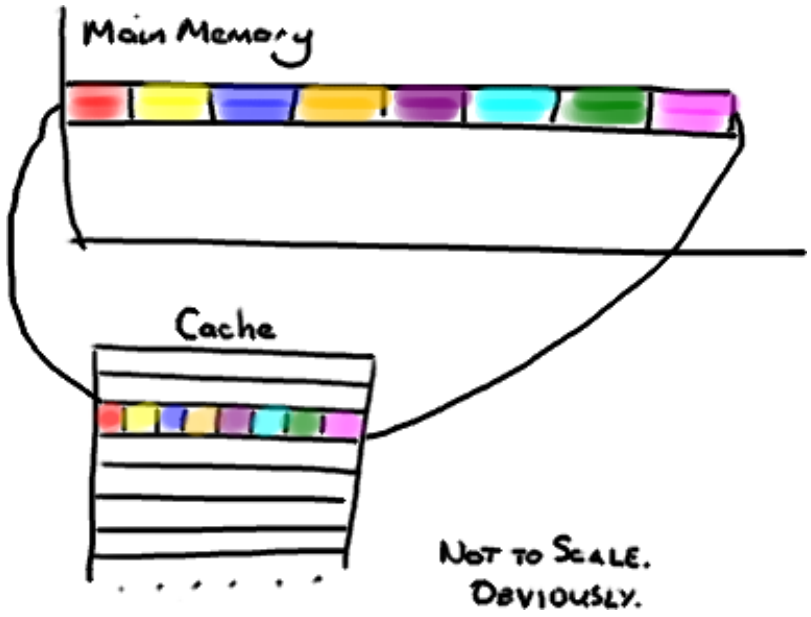
Martin and Mike's QCon presentation gives some indicative figures for the cost of cache misses:

Latency from CPU to...	*Approx. number of CPU cycles*	*Approx. time in nanoseconds*
Main memory		~60-80ns
QPI transit (between sockets, not drawn)		~20ns
L3 cache	~40-45 cycles,	~15ns
L2 cache	~10 cycles,	~3ns
L1 cache	~3-4 cycles,	~1ns
Register	1 cycle	

If you're aiming for an end-to-end latency of something like 10 milliseconds, an 80 nanosecond trip to main memory to get some missing data is going to take a serious chunk of that.

Cache lines

Now the interesting thing to note is that it's not individual items that get stored in the cache - i.e. it's not a single variable, a single pointer. The cache is made up of cache lines, typically 64 bytes, and it effectively references a location in main memory. A Java `long` is 8 bytes, so in a single cache line you could have 8 `long` variables.



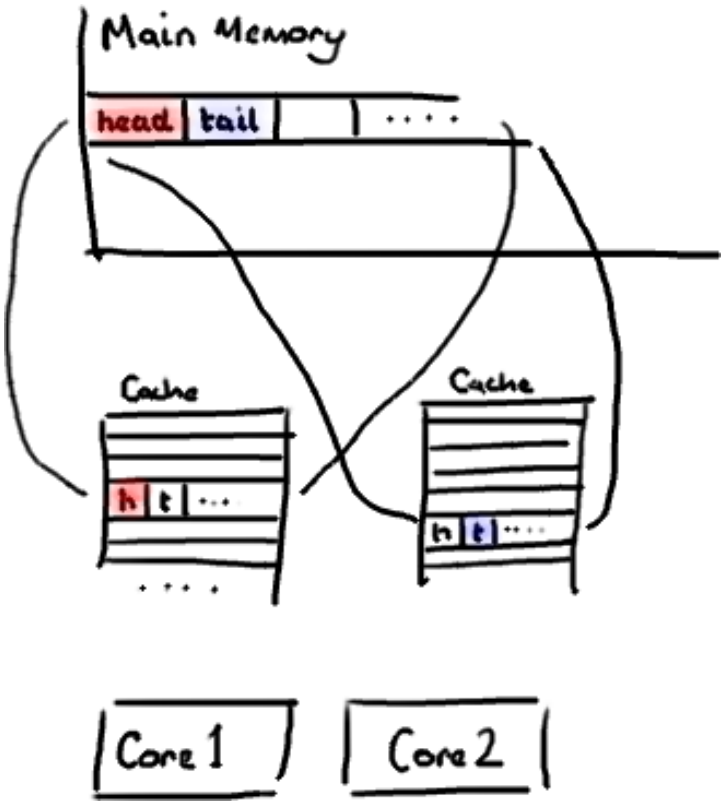
(I'm going to ignore the multiple cache-levels for simplicity)

This is brilliant if you're accessing an array of longs - when one value from the array gets loaded into the cache, you get up to 7 more for free. So you can walk that array very quickly. In fact, you can iterate over any data structure that is allocated to contiguous blocks in

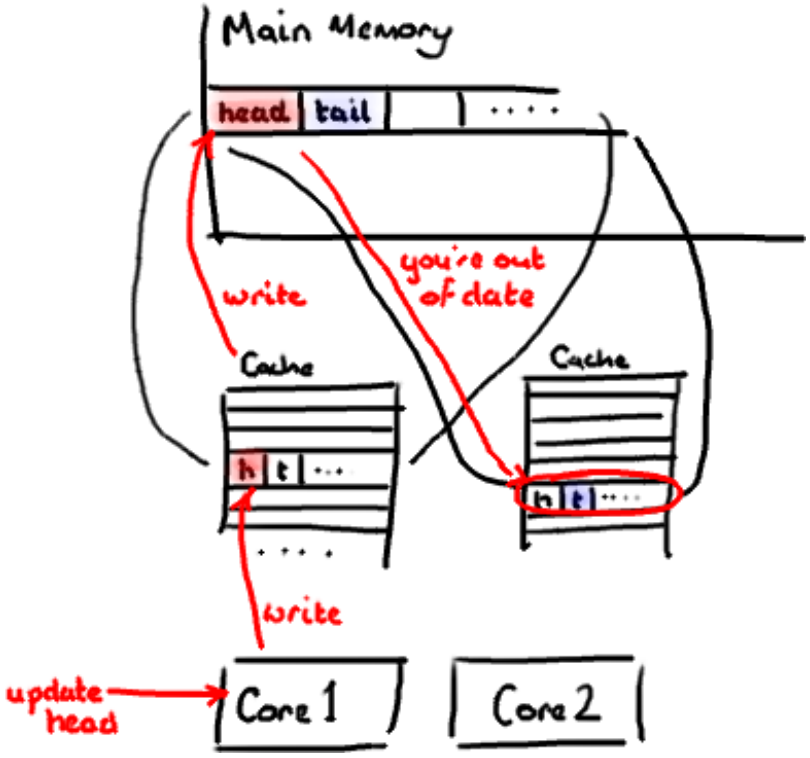
memory very quickly. I made a passing reference to this in the very first post about the ring buffer, and it explains why we use an array for it.

So if items in your data structure aren't sat next to each other in memory (linked lists, I'm looking at you) you don't get the advantage of freebie cache loading. You could be getting a cache miss for every item in that data structure.

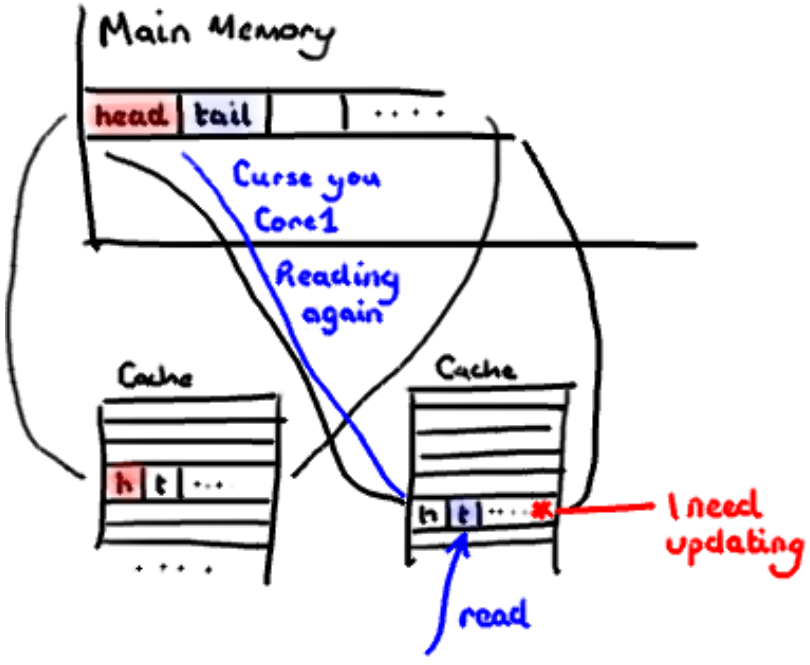
However, there is a drawback to all this free loading. Imagine your `long` isn't part of an array. Imagine it's just a single variable. Let's call it `head`, for no real reason. Then imagine you have another variable in your class right next to it. Let's arbitrarily call it `tail`. Now, when you load `head` into your cache, you get `tail` for free.



Which sounds fine. Until you realise that `tail` is being written to by your producer, and `head` is being written to by your consumer. These two variables aren't actually closely associated, and in fact are going to be used by two different threads that might be running on two different cores.



Imagine your consumer updates the value of `head`. The cache value is updated, the value in memory is updated, and any other cache lines that contain head are invalidated because other caches will not have the shiny new value. And remember that we deal with the level of the whole line, we can't just mark `head` as being invalid.



Now if some process running on the other core just wants to read the value of `tail`, the whole cache line needs to be re-read from main memory. So a thread which is nothing to do with your consumer is reading a value which is nothing to do with `head`, and it's slowed down by a cache miss.

Of course this is even worse if two separate threads are writing to the two different values. Both cores are going to be invalidating the cache line on the other core and having to re-read it every time the other thread has written to it. You've basically got write-contention between the two threads even though they're writing to two different variables.

This is called false sharing, because every time you access `head` you get `tail` too, and every time you access `tail`, you get `head` as well. All this is happening under the covers, and no compiler warning is going to tell you that you just wrote code that's going to be very inefficient for concurrent access.

Our solution - magic cache line padding

You'll see that the Disruptor eliminates this problem, at least for architecture that has a cache size of 64 bytes or less, by adding padding to ensure the ring buffer's sequence number is never in a cache line with anything else.

```
    public long p1, p2, p3, p4, p5, p6, p7; // cache line padding

    private volatile long cursor = INITIAL_CURSOR_VALUE;

    public long p8, p9, p10, p11, p12, p13, p14; // cache line
padding
```

So there's no false sharing, no unintended contention with any other variables, no needless cache misses.

It's worth doing this on your `Entry` classes too - if you have different consumers writing to different fields, you're going to need to make sure there's no false sharing between each of the fields.

EDIT: Martin wrote a more technically correct and detailed post about false sharing, and posted performance results too.

concurrency disruptor disruptor-docs java lmax mechanical sympathy

---

Yeroc *22 July 2011 at 19:23*

Thanks for the very informative post. Are you able to show performance results before and after adding cache line padding? How much of a difference did it make to the overall performance?

Thanks,
Corey

REPLY

Trisha *23 July 2011 at 19:31*

That's a good question, I'd forgotten I meant to do that. Will see if I can get some figures.

REPLY

ipc *23 July 2011 at 19:52*

thank you!

REPLY

Stephen Souness *24 July 2011 at 03:22*

I understand what the purpose of the cache line padding is, but still consider it strange that a system running in a virtual machine is bound so tightly to the underlying CPU architecture (why not just write in C?).

From your example it seems to be a counter-intuitive situation of taking the fastest accessible memory and filling it up with 87.5% junk so that the two items that make up the other 12.5% in this instance won't be neighbours.

REPLY

Kimble *24 July 2011 at 11:38*

First of all, thanks for taking the time to write all these blog posts! Very interesting reading material!

I'm also very interested in the performance with and without the padding. And by the way, do really the Java memory model guarantee that the variables are

laid out the same way in memory as they're declared in the source file? That sounds a bit strange to me. One should think that the JIT compiler would detect that the padding variables are unused (if I'm not mistaken..?) and avoid putting them into the precious CPU caches.

REPLY

Anonymous *24 July 2011 at 21:11*

I have summed up a few comments at http://www.heisenbug.info/archives/11-Cinderella-vs.-Disruptor.html

@Kimble: making the padding public should hopefully inhibit this optimization, because the JVM cannot know that you will never load a class which actually accesses these fields, but that's just a guess ...

REPLY

Trisha *25 July 2011 at 09:27*

@Stephen we could have written it in C. There was nothing to stop us taking that solution. But we wanted to prove that you don't *have* to write high performance code in C.

Two reasons to use Java spring to mind:
1) You don't spend hours chasing the kind of bugs that happen with C and by definition don't happen with Java (e.g. problems with pointers I guess - I've never written C so I don't know).

2) There's a massive pool of talent to recruit from if you're a Java shop.

And since you can write it in Java, why not? We could have picked C#, or probably a bunch of other languages. The point is the language doesn't actually matter if you can get the performance you need.

And the cache line padding isn't junk, because it's required for the performance. If some of the variables are never accessed because they're not actually in the same line as the sequence number, that doesn't matter - we don't really care about objects sat in main memory, it's the contents of the CPU caches that are important.

REPLY

Stephen Souness *25 July 2011 at 23:32*

Hi Trisha,

Excuse me if I have missed something obvious, but my interpretation of your description of cache lines is that the entire contiguous block - including the padding - is stored in the CPU cache.

REPLY

Trisha *26 July 2011 at 17:47*

Seven of the padding variables will be in the same cache line as the sequence

Seven of the padding variables will be in the same cache line as the sequence number, yes (assuming a 64 byte cache line). Which seven will vary depending upon where in main memory the sequence number is loaded from.

Those that are in the cache might not be read, but are obviously doing an important job preventing false sharing with the sequence number.

The other seven may, or may not, be in the CPU cache next to some other variable. Whether the left over ones are in the cache or in main memory, the predictable latency provided by reducing cache misses on the sequence number is worth the cost of 7 empty variables. Predictability is actually more important than small latency numbers in a lot of cases.

I'll get the numbers for performance with and without padding, and if that doesn't prove anything I've been saying then you can all point and laugh.

REPLY

Jan *26 July 2011 at 20:38*

It's a pity that you cannot tell JVM to align objects continuously in memory. With buffer array only object pointers will be optimally cached, the speed gain could doubled if the objects are located next to each other. The CPU caches can recognize that you are reading memory in a loop and will speculatively preload caches.

Suggested video: http://skillsmatter.com/podcast/home/cpu-caches-and-why-you-care

I think that memory layout is one of the reasons why microbenchmark art is so hard.

Now I see you use volatile to force cache invalidation between different threads. I don't know how is volatile compiled into code. Speed comparison between volatile and CAS would be interesting. I would not be surprised if there is less then order of magnitude difference.

Benjie *3 February 2012 at 15:00*

"Now I see you use volatile to force cache invalidation between different threads."

volatile doesn't invalidate caches, it only makes sure data is read from memory instead of stored locally in a register. The cache coherency protocol that runs at the hardware level will make sure a cacheline is up-to-date or needs to be refetched. If the memory address was recently written, the data is probably still in L2 or L3 cache, and doesn't have to go all the way out to main memory.

Paul *4 March 2012 at 16:56*

> With buffer array only object pointers will be optimally cached, the speed gain could doubled if the objects are located next to each other.

Has anyone has tried to allocate Event objects contiguously using something like http://highlyscalable.wordpress.com/2012/02/02/direct-memory-access-in-java/ (see Direct Memory Management). I wonder if it would work...?

REPLY

Stephen Souness *27 July 2011 at 03:12*

Feel free to post performance numbers (along with context about CPU affinity if relevant), but nobody was questioning the potential performance benefits of this cache line padding approach (on the appropriate CPU architecture).

I, for one, was certainly not aiming to mock the approach (though unused public fields anywhere else in the codebase would warrant a severe tongue lashing).

REPLY

Trisha *27 July 2011 at 10:02*

Curse this text-based communication! That was meant to be much more tongue-in-cheek :) I thought it would be funny if, after all my defense of the code, the tests proved it made no difference at all.

REPLY

Trisha *30 July 2011 at 09:59*

Martin's written a post with more (technically correct!) details about cache lines and false sharing, and created a performance test that shows the impact:

http://mechanical-sympathy.blogspot.com/2011/07/false-sharing.html

REPLY

Shane Tolmie *4 August 2011 at 23:31*

Great post, your diagrams are the best!

REPLY

0x1e *12 August 2011 at 17:52*

Awesome! Agree with Shane, your style is awesome! :)

REPLY

达伦王 *10 October 2011 at 09:51*

enlightening

REPLY

Unknown *27 June 2012 at 09:01*

hi trisha:

as you said,avoid false sharing when read head and tail,but how about data in ring buffer?they all are next to each other...

Trisha *27 June 2012 at 09:11*

But they should be larger than a single long, they should be objects. But you're right, there's a chance of false sharing in any part of an application you write.

Unknown *27 June 2012 at 10:30*

pointer isn't a big one if put into the ring buffer.so any idea to minimum the impact? or it is impossible or unworthy to do that?

Trisha *27 June 2012 at 10:36*

I would first write some perf tests and profile to see if this is a real problem for you

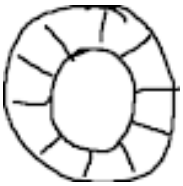
REPLY

Comments have been disabled since this blog is no longer active.

Popular posts from this blog

Dissecting the Disruptor: What's so special about a ring buffer?

June 22, 2011



Recently we open sourced the LMAX Disruptor , the key to what makes our exchange so fast. Why did we open source it? Well, we've realised that conventional wisdom around high performance programming ...

READ MORE

Dissecting the Disruptor: Writing to the ring buffer

July 04, 2011

This is the missing piece in the end-to-end view of the Disruptor. Brace yourselves, it's quite long. But I decided to keep it in a single blog so you could have the context in one place. The important areas are: ...

READ MORE

Dissecting the Disruptor: Why it's so fast (part one) - Locks Are Bad

July 16, 2011

Martin Fowler has written a really good article describing not only the Disruptor , but also how it fits into the architecture at LMAX . This gives some of the context that has been missing so far, but the mos ...

READ MORE

TRISHA

VISIT PROFILE

Archive