[Article Directory](http://www.dev9.com/article-directory)

[Performance 101: Causes of Bad Performance](file:///C:\Stefano\Git\SFiorenza\Docs\Frameworks\Disruptor\Performance%20101_%20Causes%20of%20Bad%20Performance%20—%20Dev9_files\Performance%20101_%20Causes%20of%20Bad%20Performance%20—%20Dev9.html)

[April 16, 2014](file:///C:\Stefano\Git\SFiorenza\Docs\Frameworks\Disruptor\Performance%20101_%20Causes%20of%20Bad%20Performance%20—%20Dev9_files\Performance%20101_%20Causes%20of%20Bad%20Performance%20—%20Dev9.html)

In the previous articles, we talked about some language and concepts, and then we got into tools and some more math. This has allowed us to identify places we should spend our time.

Why?

The first question we ask ourselves when we're tasked with performance enhancements is *why?* There are a lot of factors to consider, however, there are some basic rules you can start with.

The first breakdown we have is identifying if the problem is a contention or algorithm performance issue. This maps almost directly to our throughput vs. latency talks from the first article. These are the vast majority of performance issues you might encounter. Contention is what prevents us from getting the throughput that we want.

Algorithm performance affects latency. Everything works as expected, with little contention, but it's still not fast enough. These problems, while they're more fun to fix, can also mean huge changes. Making a sorting algorithm or a queue implementation or a face detection algorithm faster is an example of algorithmic performance.

*NOTE: While this article is mostly about identifying performance issues in your application, do not overlook the importance of identifying and fixing performance issues in software that your app depends on. Database servers are a common source of issues, for example. You should be running the contention checks on these services if your tests slowness in that area*

Algorithm Performance

Since we are going to spend the majority of our time talking about contention, let's spend a few minutes talking about algorithm performance. I consider an algorithm performance problem one where increasing the speed of the computer is the only way to improve performance. All contentions are minimized, but the code is running into fundamental limits from the hardware.

Big-O

What we are talking about is the basis for [Big-O notation](http://en.wikipedia.org/wiki/Big_O_notation). When you see O(n) or O(1) or O(n^2), these are Big-O examples. It helps us figure out the behavior of the algorithm with no restrictions. A merge sort (O(n log n)) is faster than a bubble sort (O(n\*n)) because it needs to consider fewer things.

Data Structures

This type of performance issue can also arise when you use improper data structures. This is why it's important for programmers to at least be aware of the differences between arrays, linked lists, trees, and maps.

Caching

The last bit of note here is one we have all probably done: caching. If you've ever used ehcache or terracotta or memcached or even Hibernate, you've taken advantage of caching. The ability to skip a lengthy computation if the value doesn't change too frequently can mean the difference between 2 and 100 servers to handle the load you want. This can often be used for a quick performance gain if it's your first pass through the application.

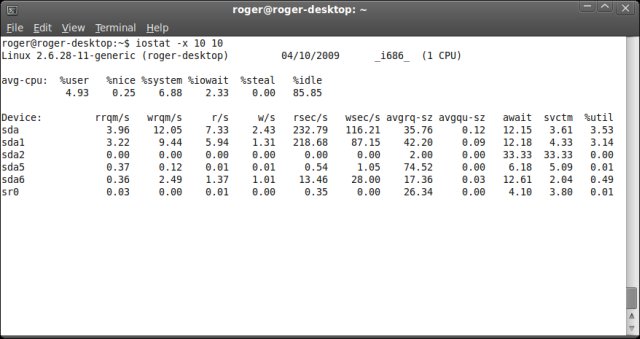
Contention

A computer has a finite amount of resources. Contention is when one of these resources is utilized to its maximum capacity, and there is a queue of work waiting to be done. That resource simply can't handle any more work. Common sources of contention are discussed below, as well as some ways to mitigate the effects.

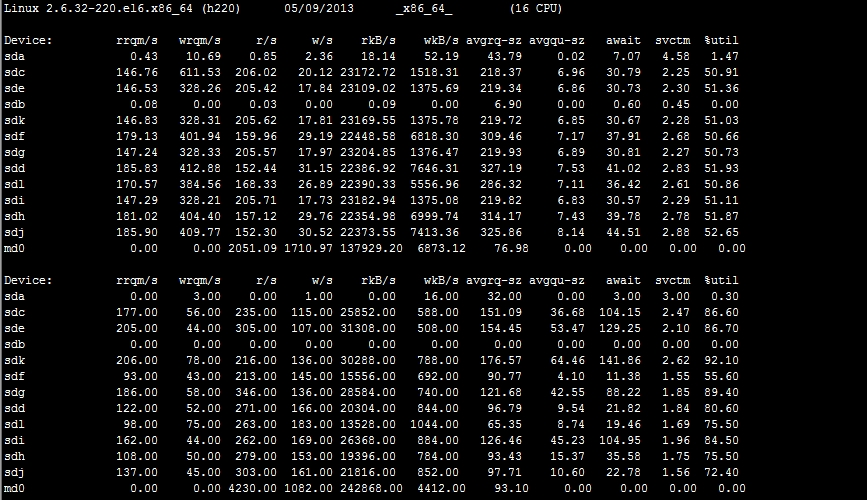
Disk Contention

Disk contention is when we spend too much time reading and writing data on disk. Because a spinning disk needs to get to the point on disk where you dsata lives, it takes some amount of time to get data back. If you have a lot of read and write tasks, your request may queue up behind others who haven't been served yet. This will result in long i/o wait times.

To investigate this issue on \*nix-like systems, use the tool [iostat](http://en.wikipedia.org/wiki/Iostat). An example of a healthy disk setup:



You can see all the different devices in this example (thanks to Roger for putting it on the internet). The interesting things are the avgqu-sz and await. avgqu-sz is the average size of the request queue. In a mostly-idle system, this will be somewhere around zero, which means all requests are served quickly. In a system with disk contention, this number will grow rapidly. Await is the average time (in milliseconds) that a request took to process -- including queue time and serving time. Let's look at an unhealthy iostat:



There are a lot more devices on this machine, but you can see certain devices have a large queue size, and you can see the await times. If you want to serve a webpage in 100ms, but your disk is taking 105ms to service a request, you're going to have a bad time.

**Blocking vs. Non-Blocking IO**

You've surely heard about node.js by now. It's famous for its non-blocking IO. Whenever your application does something that needs to talk to disk (or something that would cause a wait), it yields to another task, and lets that original task finish later when the data is available.

Java also has some of this. There's a java.io package for blocking IO, and it's probably what most of us use. There's also the java.nio package for non-blocking IO. If you find that waiting for disk becomes an issue, see if you can use non-blocking IO calls to speed up your server's ability to handle more.

**Reducing Contention**

To reduce disk contention, there are four common strategies. First, you can buy your way out. Solid-state drives (SSD) are significantly faster than spinning disks for random seeks. If your access pattern is a lot of small reads and writes, this will get you very significant benefits in performance. If, instead, your performance problem is one of reading and writing very large chunks of data, an investment in an enterprise storage system may be warranted. A RAID setup combines multiple disks into one, providing (usually) both fault tolerance and increased throughput, since multiple drives contribute to the end result.

The second common strategy is to simply reduce the amount of data that is on the disk. If you can get by with smaller files, that may increase speeds. Compression can be used on large files, as expanding it in-memory can be faster than reading uncompressed from the disk. A different serialization format can result in significant disk savings (JSON, for example, is quite verbose compared to CSV).

The third common strategy is to do batch reads/writes. This is what your hard drive controller does, which is why it's so dangerous to just unplug your computer. There is a disk cache that waits for some amount of data, then writes it all to disk at once. This is how some high-performance NoSQL engines work. They keep as much working data in memory as possible, and flush to disk periodically.

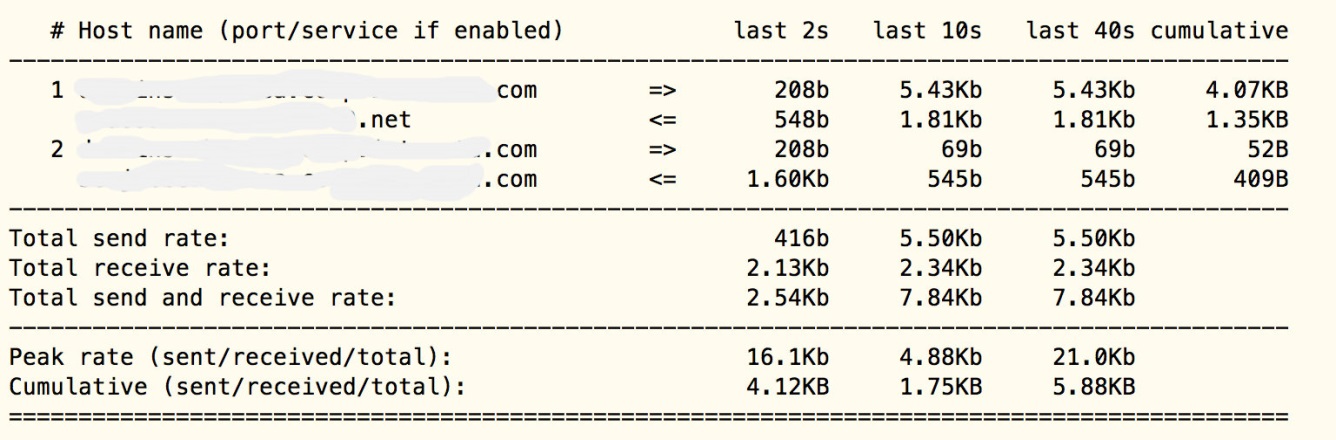
The fourth common strategy is to do more caching. Your operating system likely does some of this for you, by caching frequently-used files in memory. Your application can do the same. If the data on disk changes infrequently, you might want to read it to memory when your app starts up. If it's very large, and you do a lot of searching, see if you can index the file, or sort the file, so you can use faster algorithms like a binary search to get to your data.

This problem can be magnified if you use disk stores backed by networks. NFS, NAS, Samba, SAN -- these are all disks backed by network. While they may offer unparalleled data security and storage capacity and data mobility, you incur some overhead since it needs to communicate over a network. That leads us to our next step...

Network Contention

Networks have a lot of the same issues we have been discussing (latency, throughput, queueing). There's also the issue of network card capacity and network connection capacity. Most servers these days should have a gigabit ethernet card, and high-end servers should be using 10gb ethernet. But if your switches/routers are only capable of 10mbit, you will have a problem.

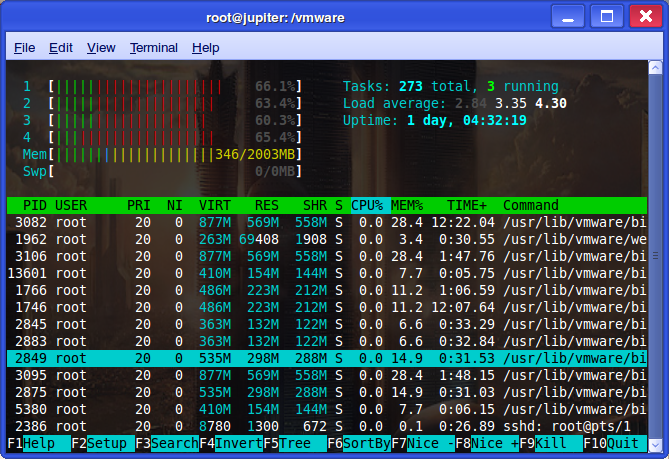
To look for potential network issues, there are two tools that are useful. First, there's the venerable [netstat](http://en.wikipedia.org/wiki/Netstat). This tool is useful to inspect network connections on the server. This can help you diagnose if you need additional threads to handle connections, if you're leaking connections, or if your system is being overwhelmed by connections. Second, there's a little utility called [iftop](http://en.wikipedia.org/wiki/Iftop). It periodically tracks the throughput of all network devices. It can drill down to a single connection, aggregate all connections, and track peaks and means. If you have a 100mbit network card and see ~10MB/sec in iftop, there's a good chance you're maxing out your network. Here's a little screenshot of it running with -t (to just print text to the console instead of an interactive app):



Solving network issues can be tricky business. Lucky for us, it's also probably not the source of problems. I have seen it be the source of problems one or two times. In those cases, if you took a thread dump of the process under load, you would see multiple threads blocking on a process that is waiting in the function socketRead0 or socketWrite0 -- native code that actually does the socket communication. Sadly, thread dumps are the most reliable way I've found to differentiate between network issues and other forms of contention.

CPU Contention

CPU contention is fun, in a way. We all know the command 'top' to display system load and running processes. In the modern world, with lots of threads and processes, it can be helpful to get a bit more detailed in our analysis. The tool htop breaks down each core's load, and provides a lot more than the default top:



Notice how all 4 cores are showing a high level of usage in this screenshot. I highly recommend this tool to verify that your CPU is overloaded.

While this is one of the easiest metrics to inspect, it can be very difficult to fix. The system just has too much to compute to do much more. This is where algorithmic fixes and better data structures come into play. If you notice the system bogging down but the CPUs are not maxed, you have some other contention slowing it down.

**System Load**

The load average from the above screen shot shows the 1-, 5-, and 15-minute averages of load. The load average is an exponentially dampened snapshot of running and runnable processes correlated with wait queue length. With multi-core machines, it gets even trickier. If it was truly just CPU, a 4-core machine with a load average of 1.00 is ambiguous. Is that 1 core fully maxed (and thus ripe for paralellization), or all 4 cores running at 25%? Maybe 2 cores at 50%? With multiple cores, a load average of 4 doesn't necessarily mean the system is under any stress. This is where tools like htop above help diagnose the issue

Memory Contention

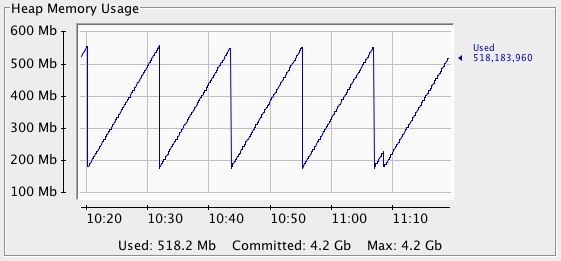
Memory contention is when there is more memory being used than available on your system. Thanks to swap partitions, this shouldn't crash a machine. However, it's likely to destroy your performance. The more tuned your app is, the more detrimental swapping will be. Memory contention can also come when you run into out-of-memory (OOM) problems, or issues with garbage collection in managed apps.

Swap-based issues are easy to spot, as top or htop will tell you the amount of swap they're using. For a production system, you ideally want no swap used. Putting 128GB or more on one machine is perfectly doable these days.

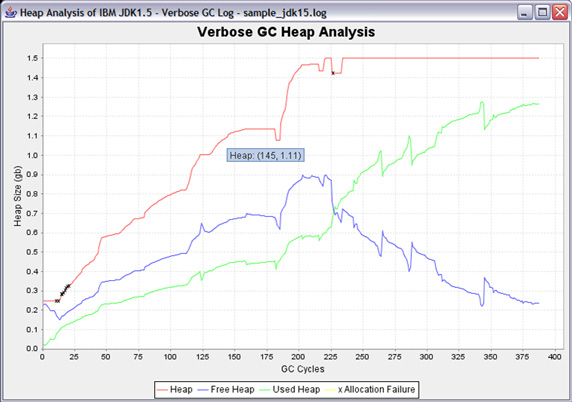
Out-of-memory usually surfaces as a process that dies unexpectedly with no messages. The only way to fix this is to consume fewer resources. This, again, is where better data structures or more compact object representations may help. You may also have memory leaks.

**Garbage Collection**

Garbage collection tuning, especially in Java, is almost a job unto itself. There are a lot of tunable parameters. There is the permgen vs. the heap. There's the issue of heap resizing vs. fixed-size heap. When you profile your Java app, you really want to see the classic saw-tooth pattern:



This is a generally healthy system. The garbage collector kicks in and returns the heap to about the same size. A more unstable or growing-memory system looks like this:



Notice how the "free heap" blue line climbs, then eventually drops, despite garbage collection happening on the green line. When the free heap is near zero, Java will spend a lot of its time trying to free up the heap, including multiple stop-the-world pauses. These can range from a second or so to multiple minutes, depending on heap size and object counts. If you let it run like this long enough, it will probably become unresponsive and eventually crash with an out-of-memory error.

Tuning the GC is beyond the scope of this article, but it can dramatically improve performance.

**Memory Leaks**

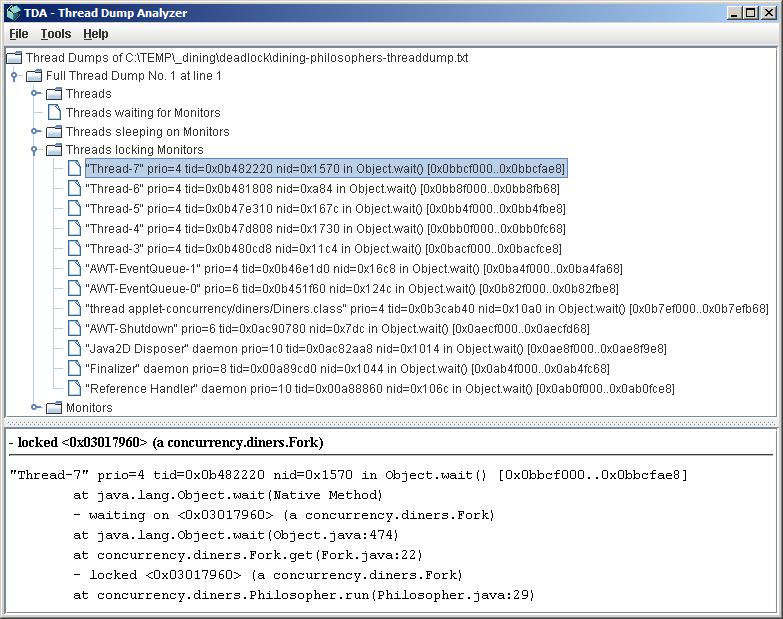
The last thing to look out for is memory leaks. Technically it's impossible to do this in Java if you're not using unsafe libraries, but in practice it's a problem. The biggest issue is dangling references to objects that you no longer need. It keeps these objects around in perpetuity, and that could eventually kill your heap. Using tools described in the previous article (visualvm, jprofiler, yourkit), you can inspect where these objects are created, which ones take the most space, and what types of objects they are. This can be very helpful in tracking down excessive memory usage.

There is a fascinating issue in Java that has cropped up occasionally. If you have a very large string (say, a JSON or XML document, or a large text file you've put into a string), then take a substring of it, that substring is just a windowed view of the larger text string. This is good when your strings are small, as it prevents a lot of reallocation. However, when the source is very large, your substring holds a reference to that large document, meaning your memory usage is far larger than normal. This "leak" was fixed in OpenJDK 7u6. If you're still on JDK 5 or 6, you're probably being affected by this.

Lock Contention

This is where things get really tricky. I've also found that it's the source of a lot of issues, so it's good to get well versed in this kind of contention. Because multiple threads accessing the same resource (variable, array, database connection, etc) can stomp on each other, there needs to be a way to control access to these sensitive bits. In Java, we often put synchronized on the method definition, or put a critical block inside a synchronized block. Behind the scenes, it creates a lock so that only one thread can execute here at a time.

When there are a lot of processes vying for the same lock, you are slowing them all down. If your locks are implemented as spin locks, it may exhibit 100% cpu usage while it waits. If you take a stack trace, you will see "Waiting on 0xXXXXXXX" for threads that are waiting for a lock. A useful tool for some users is TDA, and this is how it presents there:



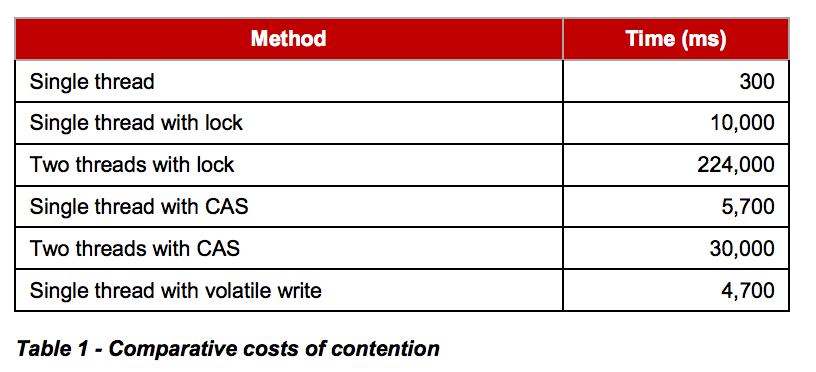
Not all locks are deadlocks. Deadlocks are two threads waiting for locks that the other holds, with no way to make progress. Most profilers offer tools to automatically detect deadlocks, and will make your life significantly easier.

Lock contention is so expensive that multiple languages/frameworks/patterns are designed to avoid it. Functional programming languages often avoid this issue because they do little-to-no variable mutation. Immutable data structures simplify your life significantly. Because they can't change, it's safe for multiple threads to access the data. Lock-free queues are popular in some circles, but they can be nasty to code up correctly if you're not extremely well-versed in this specialty.

**Mechanical Empathy**

If you're going the ultra-high-performance route, there's a concept you should be aware of called mechanical empathy. This describes a way of organizing your parallelism to minimize the burden on the CPU, especially context-swaps. When you're trying to design sub-millisecond responses in a managed language, it can be difficult to achieve using traditional methods.

Mechanical empathy was invented and popularized by the [LMAX Disruptor](http://lmax-exchange.github.io/disruptor/). From the PDF of the whitepaper, this graphic comes to explain the cost of locks:



You see that as contention (and thus arbitration) increases, your throughput gets tanked. LMAX Disruptor is an attempt to design a system to minimize locking and minimize expensive context switches.

This is not a simple implementation, and it's not a drop-in replacement for, say, ConcurrentHashMap. Read the link above to get a lot more information and see if it's the right approach for you.

Conclusion

This wraps up our tour of the most common cases of poor performance, and some things to look out for. Becoming proficient at analyzing and tuning the JVM as well as analyzing and tuning your own applications, you will have a much deeper understanding of your own code, as well as a much deeper understanding of how the JVM works, which might make for better code. Go forth and profile!