# CBOE Report

Jarod Jenson
Chief Systems Architect
Aeysis, Inc.
jarod@aeysis.com

## Summary

The initial purpose of the engagement was to determine the source of latency being experienced in production in the FE in timely delivery of data received from the BC. Although, this was not reproducible in the testing environment, analysis of the application has given us several likely candidates.

In addition, there were a number of stability issues being experienced on the new v40z servers that have a planned deployment into the production environment. Since these issues were also deemed critical, and several occurred while on-site, assistance in root causing and working with Sun to provide remediation was also undertaken. The source of the problems appear to have been identified were attributable to highly rare and recently filed bugs. Sun has provided fixes for these in the form of a kernel patch with two additional specific fixes for the issue observed.

A brief analysis was also undertaken on the other core pieces of the platform – the BC and the mdcas – to look for potential tuning opportunities in these applications. In both cases, potential opportunities were noted that could provide improved performance.

## Detailed Analysis

The first priority of the analysis of the FE was to determine the possible causes of the latency being experienced on messages being processed in the FE that were received from the BC. Since this particular phenomenon could not actually be reproduced in the test environment, only likely hypothesis based on observation of the application's processing can be made.

The most likely candidate is based upon the fact that there is one thread that receives the data returned by the BC that is competing for a synchronization object with a high number of threads on the mdcas side. Since the number of threads competing for lock exceeds the number of CPUs, and it is a highly contended lock, it is possible that the thread could be briefly starved from the lock and this would be the source of the latency.

There are a number of factors that come into play in this hypothesis. The first is that – due to the default timeshare nature of the scheduler - the single thread will have a lower priority from an operating system perspective since it will singularly consume more cycles per time quantum that each of the other threads. As threads consume their entire time quantum, they slowly have their priority lowered as they are deemed to be a CPU *hog*. This means that the OS has to make a determination about the order of a thread in a particular CPU runqueue (we expect some queueing since there are more active threads than CPUs), the single thread may find himself further back in that runqueue. This would equate to latency to get on CPU and potentially miss the opportunity to acquire the lock – potentially several times. Secondly, Java will "escalate" highly contended locks to the operating system in the form of a system call to attempt to allow the OS to deal with lock acquisition. This system call overhead (the trap) as well as the significant code paths associated with it, could add additional latency itself to the queuing thread.

Because of the high contention on a single synchronization object, there are only a couple of possible options to mitigate this issue. The first (and best) is to attempt to find a new algorithm for accomplishing the same goal. This includes looking at lockless protocols (structures), breaking out the lock to multiple locks, change the number of enqueuing threads, or to use a interval based approach to prevent starvation. Each of these has pros and cons, and the final solution could be some combination of the aforementioned. As the CBOE developers are quite skilled, a discussion of each of these items is likely not warranted. Additionally, additional application semantics are required to fully understand the nature of the architecture to provide firm recommendations on which approach would work best. This discussion should be scheduled for the near future with the potential for additional DTrace analysis to determine the efficiency of any prototype implementation.

However, there is one possible feature of Solaris 10 that may be employed to potentially mitigate the issue to some extent. In Solaris 10, there is what is known as the *fixed priority* scheduling class. The fixed priority (FX) scheduling class does not participate the raising and lower of thread priorities based on CPU usage and sleep time. Threads are merely maintained at the same priority level regardless of any of their behavior characteristics. This means that as a thread becomes runnable, it is merely placed next in the queue and as the scheduler looks for the best thread to run, it has no chance of being moved to the back of the queue (this of course is in the context of the process in the FX class, but this is by far the number of runnable threads on this system). This will greatly reduce the possibility of starvation for the single thread. To move the thread to the FX class, the priocntl(2) command is used. By default, only the root user can change a processes scheduling class. However, on Solaris 10 with *Process Rights Management* the ability to change the scheduling class can be granted to a non-root user. This is done by adding the user to /etc/user_attr and specifying the proc_priocntl privilege. As an example, the following command could be used to set the class and priority of an already running JVM:

```
      priocntl -s -c FX -m 58 -p 58 -i pid <pid of JVM>
```

This would give every thread in the process a high priority that will never change. This potential work-around could be tested to determine what (if any) benefit it provides. This should only be used temporarily as a more correct option is investigated.

Further investigation of the FE yielded a number of additional tuning opportunities to improve the overall performance of the application. First and foremost, it was noted that a significant amount of CPU cycles were being consumed in performance message non-repudiation. In addition, the only disk I/O from the application was originating from the non-repudiation logging. Since the application no longer needed this functionality, it was determined that disabling this feature would provide significant benefit. In fact, disabling non-repudiation dropped overall CPU consumption of the application by almost 50%. As a side note, this recovery of CPU cycles could have a significant impact on the issue described above.

The next item discovered was another high CPU and object allocation in a particular HashMap key and toString methods. Code inspection of this method revealed a very interesting implementation (bug). The key method was mis-coded in a way in which variable arguments were assigned the value of themselves (such as `flavor=flavor` instead of `this.flavor=flavor`). This meant that the key function was providing no functionality effectively and since everything hashed to the same value the size of the HashMap was 1. Since this was a significant amount of work for no value, the code was modified to make this static and non-calculated entities. This change also resulted in CPU cycle recovery and combined with the aforementioned change reduced the amount of time between garbage collection cycles from every 4 seconds to every 8 seconds. Clearly a major improvement.

From a code perspective, the last item shown as a significant outlier from other methods is the constructor for the CDRBuffer. There was insufficient time to do a code analysis and determine the cause of the high CPU utilization in this method (which could also be attributed to the object's frequency of creation), but it seems a likely candidate for further work.

After the changes mentioned above, the overall CPU utilization of the FE at the same workload level was approximately 60% of the initial value.

For the BC, there were only a couple of observations made – although they could have significant implications. The first was the use of the Java method notifyAll. notifyAll will wakeup all threads waiting on an object. For an application with many threads waiting on the same object, this can have significant implications in the form of the thundering herd problem. All of the threads will be awoken, compete for the same locks and (generally) only one thread will have work to do. This activity consumes significant unnecessary CPU cycles and creates a heavy burden on the OS scheduler. The two places where notifyAll was found was in the object editor of objectwave and the QueueBasedImpl. Eliminate one or both of these if possible would provide valuable CPU cycles back to the application.

Secondly, it was noted that the garbage collector was having to perform a GC about one each second. This is a very high rate of object turnover. *Significant* work has been done to tune garbage collection and, in fact, CBOE could have some of the most advanced GC tuning I have seen. However, additional analysis is warranted to try identify the root causes of the high object allocation rates and remediate them were possible. This can easily be accomplished using the DTrace DVM provider, but the time was not available during this engagement. If CBOE is inclined, they can start the BC application using the instructions provided for the DTrace DVM provider at https://solaris10-dtrace-vm-agents.dev.java.net/ and specify the startup option as '`-Xrundvmti:allocs`'. This will mitigate the significant performance overhead of also monitoring method entry/return. With this running, the following DTrace script will help identify top offenders:

```
      dtrace -n dvm<pid of JVM>:::object-alloc'{@[jstack(10, 1024)] =
sum(arg1)}END{trunc(@, 25)}'
```

This will give the top 25 Java stack frames responsible for the highest byte count of allocated objects and the sum of the total bytes allocated. From this, it should be trivial to examine the code for possible ways to reduce the allocations required.

For mdcas, there was a single issue noted simply due to the fact that the overhead of the operation accounted for a *vast* majority of the work performed. Simply, the amount of synchronization (lock) contention in the application was severe. The CPU utilization, context switch rate, and involuntary context switch rate was almost purely a result of the contended locks. It was determined that the magnitude of the issue was exacerbated by the format of the test, however, a modification to the test to attempt to alleviate some of the issue still exhibited the problem. Specifically, the threadCommandQueue.getNextCommand and the threadPool.dispatch methods were the two sources of the high lock contention. If the test harness is flawed, it will require modification before more useful data can be collected. If it is indicative of the actual underlying issue, then the root of the problem must be addressed.

As a sanity check, the production machine should be monitored using mpstat(1M). If the context switches, involuntary context switches, and system CPU time are all quite high in production, it is most likely that the same problem is being observed there.

Due to the nature of the locking protocols in Java noted above (the escalation of the locks to the kernel), and attempt to work around some of the effect of the problem was tried. This was to use the Java option `'-XX:UseLWPSynchronization'` to force the synchronization to be handled in user land only using tradition libc based mutexes. This change dropped CPU utilization by approximately 40%. However, this is not a clear indicator that the performance of the application was improved by this factor. More testing should be done to determine what (if any) improvement is noted at an application level. Again, this is not a solution, but merely an interim measure until the code and architecture can be further evaluated.

## System Observations

As part of investigating system stability issues that were being encountered during the timeframe of the engagement, it was noted that on at least two production systems there was a single CPU spinning with 100% system time. DTrace and system dump analysis identified that CBOE was experiencing a bug I had recently identified as part of a benchmark. This is bug 6418995: htable_scan() never completes. As this bug had been previously analyzed, an IDR was rapidly produced and provided by Sun to address the issue. The effect of this issue without the IDR is the loss of a full CPU from use by the applications residing on the affected system.

It should be noted that I was impressed in the rapid production of the IDR and its delivery to CBOE by Sun.