CBOE Report 5
XTP, MDX, OHS, et al.

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

Summary

For this engagement, several applications were lined up for analysis. This primarily included XTP, MDX, and OHS. The persistence server was not analyzed, but there was significant discussion about potential mechanisms for ensuring that messages are not lost in the event of a server failure.

For the XTP analysis, a resulting increase in message rate would be seen as a good thing; however, the primary issue that needed to be addressed was the inability to recover and avoid significant queueing in the event of a Concurrent Mark and Sweep garbage collection. After the analysis, we were able to achieve both of these goals with good results in both categories.

For OHS and MDX, the message rate was the key focus. On the OHS side, there were several observations made that allowed for an increased message rate and with less deviation. For MDX, several potential performance inhibitors were identified; although no runs (that I am aware of) were made with all the changes necessary to address the issues to determine the improvement (if any).

One item that was noted that will have an impact across many applications is a key piece of the infrastructure code used to instrument methods and codepaths of interest within the applications. Performance and scalability issues within this instrumentor framework were addressed and showed significant performance improvement.

Detailed Analysis

The first application that was profiled was XTP. The initial discussions included talk of the high transaction rates, however; the biggest current issue is that if a Concurrent Mark and Sweep (CMS) garbage collection occurs while the application is under heavy load, there is a high probability the the pause (and CPU consumption) associated with this collection could cause a significant enough backlog that the application would queue heavily and potentially not recover from the situation cleanly enough causing significant instability.

To address this, we looked at historical data (GC logs) and statistics associated with the application. Based on the data, it appeared that if a CMS collection happened, it took longer than the time of a young generation collection and this would cause the application to reach a point that the “promotion guarantee” could not be met. When this happened, the GC pause was significant enough that the application became unrecoverably behind.

It was decided that the best way to address this issue was to cause a CMS collection to begin earlier, giving enough headroom so that a young generation promotion would be a non-issue. The default time to begin a CMS collection is when about 68% of the tenured generation is used. Based on the initial startup costs and slow tenured growth of the XTP application, it was decided that a value of 40% would be ideal. This gives some room on top of the initial startup allocation that will be permanent, yet it starts soon enough that we meet our promotion guarantee requirements.

Setting the value too low (like to 1 for instance), would cause CMS threads to run constantly. This can have two potential negative side-effects. The first is that one CPU (of four) would constantly be busy doing CMS collections. This lack of resources could cause the same queuing issue we are attempting to address. Also, it is possible that this can increase the rate at which tenured memory space fragments (CMS is a non-compacting collector). This could mean that we would have enough free space in the tenured generation, but not enough contiguous space to allow for a young generation promotion. In this event, a full collection would be necessary (concurrent mode failure) and the queuing issue would again happen.

Tests of XTP with the above change showed that a CMS collection was a non-event (even multiple over the life of the application). The application continued processing messages normally with only a small impact to the rate a latency.

Next was to observe the application to see if there were any opportunities to increase the message rate. Using a DTrace based approach, we identified a handful of items that could easily throttle the message rate.

The first item noted was that the “method instrumentor” was having a significant impact on the application in terms of both performance and scalability. The method instrumentor is an infrastructure software component that is used to provide high resolution timing of critical methods in the application. These logs can be used to determine performance as well as to investigate potential production issues should they arise. It is therefore a highly useful and critical component of this application as well as others.

The method instrumentor has not previously been identified as an issue. It is well written, but as the performance of the applications increase; it slowly has drifted into observability. Essentially, as application can call internal methods “faster” or more frequently; the method instrumentor's impacts become more obvious. The biggest issue was the use of traditional Java monitors to provide synchronization of critical sections. Once identified, the infrastructure team quickly addressed the method instrumentor and provided a version that was about 3X faster -a significant improvement.

However, for XTP we wanted to continue testing without waiting for remediation (the fix was done quickly, but we had only a couple of hours). To meet this goal, the method instrumentor class was interposed with a version that did nothing but return. This would allow us to measure performance without the impacts of the method instrumentor.

The next item noted for XTP was that a fairly significant portion of the time was being spent in a particular hash map. In looking at the hash map, it was noted that the hash chain lengths were longer than desired (up to a chain length of 5) and that the comparison method was non-optimal (i.e. it tested two fields that would have almost always been equal before testing the third field that would be the most likely indicator of equality). Although not tested, it might also be wise to investigate the usage of a concurrent hashmap with Java 1.5 as these generally perform much better when heavily contended.

With these changes in place, the message rate moved from about 550K/s to about 637K/s. A very measurable increase. However, this includes the caveat that the method instrumentor was still essentially a null class. Testing with the new method instrumentor should be (and probably has been) done to determine the real new rate (although it should still be improved significantly). And with the increase, the CMS collections are still a non-event.

The last item noted – and still left to address – is that we are spending a significant portion of one CPU in kernel time dealing with the UDP traffic. While this is anticipated due to the high multicast volume, the lock contention observed in the UDP module was not. With the new UDP stack in Solaris 10, it is highly unlikely that there should be high lock contention across disparate UDP endpoints. This is apparently what we are seeing. I will be working with kernel engineering to determine the source of this (potentially a multicast only) issue and the best way to address it. This should further increase XTP performance.

Another application profiled was MDX. The time spent with MDX was a small fraction on the time spent with XTP and OHS. However, one thing was abundantly clear – MDX can really work the Solaris scheduler.

First and foremost, MDX has an extremely high context switch rate. The number approached 40K context switches per second per CPU. This is usually indicative of a few underlying issues, and I believe that all of them are present in MDX. High context switch rates are not always causes for concern, even though there is some amount of inherent latency associated with them. However, for applications that measure latency down to the microsecond, they generally are a leading indicator of other underlying issues that will contribute heavily to latency and lower scalability.

The first thing we did is move MDX to the fixed scheduling class (FX). This scheduling class is used by a number of applications (and should be used by all potentially) at CBOE. The FX class essentially prevents a highly threaded application from *cannibalizing* itself as priorities of the individual threads continually have their priorities adjusted. A thread with a higher priority will preempt a thread of a lower priority when it is runnable. As threads consume CPU cycles, their priorities are lowered. This causes unwanted preemption for what are actually very short CPU run times. the FX class avoids this by placing all threads at the same priority and allowing the application threads to complete their work (as long as they don't exceed their time quantum) without the preemption. With the FX class, the context switches of MDX dropped substantially – including the near absence of involuntary context switches.

A second leading cause of high context switch rates is the use of the notifyAll() method. notifyAll() will wake all threads waiting on a particular condition instead of waking a single thread when work is ready. This is generally known as the thundering herd problem. In profiling MDX, there were several uses of notifyAll() found in the code. Some (if not all) have been addressed and this should contribute to further lowering the context switch rate and avoiding spurious wakeups.

The third performance issue identified – and another source of context switching – is high lock contention. Contended locks are clearly a scalability inhibitor, and the overhead associated with these contended locks results in performance degradation as well. MDX is a highly threaded application and lock contention will be heavily compounded. The contention for e-cache lines (locks are atomic and e-cache lines have to be owned) has a hard to observe penalty, but it is significant.

DTrace analysis showed several points in which there where heavily contended synchronized blocks. The most significant – in terms of a statistical sampling - was a synchronized run method in a heavily used class. The stacks associated with contention events have been collected. However, a inconsistency between the Java 1.5 (or 1.6) implementations and DTrace, prevents us from getting a full Java stack. This is reported as Sun CR# 6276264. In any event, Kevin Dolan is addressing the contention points as best as can be done with the stack information that is obtainable. This appears to be (on the surface) sufficient information to find the most egregious issues, but addressing the Sun CR will provide even more useful information. The following two stack fragement show what are the most heavily contended synchronized methods:

```
java/lang/Object.wait*
com/cboe/util/ThreadCommandQueue.getNextCommand*
com/cboe/util/WorkThread.run*

java/lang/Object.wait*
com/cboe/mdx/common/supplier/MDXUserSessionMarketDataBaseSupplier.run
com/cboe/mdx/common/supplier/MDXUserSessionMarketDataBaseSupplier.run
```

Lastly for MDX, is the issue that there were many threads (hundreds) that were blocked on conditions (wait) with timeouts. A 50 millisecond timeout would normally not be an issue, but when there are a high number of threads with the 50 millisecond timeout, there is an increase in the number of threads that are experiencing their timeout and becoming runnable. This plays into the context switch issue, and also causes undue CPU utilization by threads that are spuriously waking to perform no work. Usually, this paradigm is used in cases where there is the potential for a missed wakeup. For MDX, this is not likely (it would be a bug), and therefore the timeout can be extended significantly to avoid the timeout of threads that will do no work. Moving from 50 to 250 milliseconds for instance will reduce the potential of spurious wakeups by 1/5 for what should be idle threads. In addition, it may be advantageous to increase the clock rate on the system housing the MDX application. By default, the clock thread runs 100 times per second. Timers (such as those used in Object.wait()) will cluster their wakeups around the clock thread. By increasing the clock rate, this would give MDX more opportunities to evenly distribute the wakeup events across a second interval. To do this, the following line can be added to /etc/system:

```
set hires_tick=1
```

The last application observed with DTrace was OHS. OHS was encumbered by a handful of previously mentioned or previously reported issues. The first if the “method instrumentor” performance and scalability issues reported for XTP in this report. These should be easily corrected with the same infrastructure code patch. Secondly was the priocntl(1) issue (having the application processes run in the FX scheduling class) to reduce the interference of rapidly changing priorities of threads within the application. And lastly wasa previously reported issue regarding libumem(3LIB) for rtserver – in which the default Solaris memory allocator is not MT-Hot and with the high memory allocation by numerous threads, severe scalability issues can be observed. However, there were more significant issues at play.

The biggest issue by far was the fact that OHS errantly was sending messages that should have been destined for MDH queues directly to the rtserver. This was in addition to sending the message to the correct recipient. The effect of this was most detrimental due to the fact that the rtserver was incapable of sustaining the message rate of OHS. What was observed is that the rtserver was not able to receive the message as quick as OHS was placing them in queue (in the form of a TCP connection). Once the

TCP window filled, OHS writes to rtserver would block for upwards of 8-10 milliseconds while the TCP window was consumed.

Increasing TCP window sizes could somewhat address the issue and could assist with performance in general. A simple ndd boot time script (I believe CBOE already has) could be used to incease the maximum TCP send and receive window sizes. The real bug at play, however, was that the OHS server should not have been sending the data to rtserver in the first place. This would prove to be a significant performance inhibitor that once addressed increased the message rate significantly.

There were additional issue outside of OHS that contributed to a reduced message rate as well. For instance, the DN adapter suffered from a previously reported Talarian exception issue in which messages were identified using Java exceptions. This causes a significant performance degradation due to the overhead of the exception and stack resolution. This issue is addressed in a Talarian patch that was applied.

Also, it was determined during the testing that the test driver was saturating the driver server (Solaris 8 on an older SPARC machine) due to the timeouts for the artificial delay being clustered around the clock thread – as mentioned previously here - and was not able to completely or repeatedly stress the OHS server. Essentially, the server appeared to be idle when in fact the threads heavily competed for CPU time between accounting clock cycles. Once this was identified, the test driver was moved to a more “capable” system and stressed the OHS server as expected.

Although not a direct performance analysis, there was much discussion regarding the persistence server. Primarily, the persistence server needs to be able to survive and insure transactional integrity in the event of a server failure. To this end, it is necessary the the persistence server persist (sounds redundant, I know) accepted transactions before committing them to the Oracle instance that serves as the central repository for this data.

There are several options available, and those discussed include a local (and seemingly faster) database or a proprietary filesystem database based approach. Regardless of the decision, there needs to be an underlying storage architecture that will ensure fast and reliable access to the data.

Technologies discussed include NFSv4, QFS (from Sun), iSCSI, or a CFS (Symantec) based solution. Each of these have some cost associated with them, but NFS or iSCSI is by far the cheapest if it can be configured to meet the goals. That is not to say that the more expensive options are better solutions, but if a free (or close to) option meets requirements, then it seems only logical to consider it first.

The main area of concern (for me at least) with regard to this approach is that the persistence server will have to provide the same ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) properties that Oracle provides, and do it faster than Oracle to be of benefit. This is possible because the persistence server will not be as feature rich while ensuring the ACID properties, and needs only to provide two streams of sequential modifications (one to record the transaction and one to mark it complete). However, the usage and the configuration of the underlying file system and storage will be critical in guaranteeing that this can be done in a manner faster than Oracle.