CBOE Report

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

Analysis

The third installment of the analysis of CBOE applications focused only only the HybridTradeServer. It was intended to look at other applications in the environment, but issues prevented the correct operation of the servers (specifically the CAS) in the perf environment we were using for testing.

The continuing analysis of the HTS unveiled additional issues that will prevent anything near linear scalability of the application as volume increases. The most prominent issues are serialization of the application through extremely hot synchronization points as well as object allocation.

For the former, there are a number of underlying resources for which there is only a single instance. By having a single instance accessed by numerous threads, the scalability of the application will be severely impacted. The critical regions that are synchronized are very small, but the number of threads accessing them at any given point in time can cause exponential degradation in performance.

For instance, the TradingClassCommandQueue has only a single instance of which as many as two dozen threads were observed to be accessing via synchronized get() and put() methods. There are serious implications in having highly contended locks by a large number of threads. The most obvious is that there will be a significant number of threads blocked from forward progress at any given point in time when the lock is held by another thread. However, there are other side-effects that can escalate this issue quickly.

The Java implementation for synchronized blocks is to adaptively spin and then block on contended sections. The algorithm is fairly intelligent and uses information such as whether the owner of the lock is on CPU, the recent successes of spinning and a backoff algorithm. However, with a high number of threads, there is still the significant probability that a necessary artifact of SMP implementation will become an inhibitor of performance – e-cache line contention. The lock is essentially an address in memory. Only one CPU at any given point in time can “own” the portion of memory in which the lock resides – known as an e-cache line. When greater than two threads are attempting to spin (or otherwise access) the relevant portion of memory, the CPU's will continually steal the memory from each other in the underlying compare and swap instruction. This means that for small durations of time, the e-cache line will be swapped between processors at interconnect speeds and steal valuable resources that could otherwise be used for forward progress of other threads in the application.

TradingClassCommandQueue is only a single example of this issue. The ElasticObjectPool synchronization and TransactionLog and MarketDataHomeImpl hash tables are others. The ElasticObjectPool is a significant area of concern in this respect (and difficult to remedy as it is

implemented as a singleton). For each of these, an alternative must be identified to address the scalability issues that will eventually become a cause of significant application performance degradation at high volumes.

It is understood that a replacement for the ElasticObjectPool is currently under development. The replacement will need to directly address the serialization issues currently plaguing the ElasticObjectPool to be an effective replacement. If this is accomplished, then this concern is mitigated.

For the TradingClassCommandQueue, I am unaware of any current plans to alter the implementation. It is recommended that, if possible, there be multiple TradingClassCommandQueues. Based on conversations, it appears that this may be possible by having a TradingClassCommandQueue per trading class. Minimally, there should be several of these (perhaps one per CPU defined at runtime) that the classes can "hash" to for using this queue. By breaking the locks out and having fewer threads competing for the same locks, the scalability of the application can be significantly improved.

For the two hash tables referenced, each of them has a particular method that is responsible for the majority of the contention. For the TransactionLog, it is the notifyListeners method and for MarketDataHomeImpl it is the getMarketData method.

For hashtable issues, there are generally a couple of approaches that we use. The first is to do an analysis to determine if the hashCode() is doing an effective job of eliminating collisions. If there is a high collision rate, then the chain length of the individual buckets can become sufficiently long to introduce long hold times during insertion and retrieval. The simplest way to test this is to use the current hashCode algorithm on a large subset of the universe of items that are hashed to determine how many collisions occur. This can be used to extrapolate potential chain lengths.

Secondly is to identify the access pattern of the hashtable. If the table is read heavy and the semantics of the application are relaxed such that an update seemingly initiated after a reader has called its get() method is acceptable as returned data, then using a ConcurrentHashMap may be an acceptable alternative. A ConcurrentHashMap is thread-safe, but retrieval operations do not entail locking and there is no mechanism to prevent all accesses to the ConcurrentHashMap. Based on current knowledge of the usage, this may be an acceptable alternative for MarketDataHomeImpl. If indeed it is acceptable, then this should be an easy drop in replacement and the methods and usage are virtually identical to HashTables. Empirical evidence of a comparison between the Java HashTable and ConcurrentHashMap implementation shows that for 10M operations, the ConcurrentHashMap required only 7% of the wall clock time of the HashTable to complete the operations. However, it may or may not be suitable for the TransactionLog.

If not, the TransactionLog implementation could be modified to support multiple instances and the listeners could be "hashed" across the various instances to remove the single lock nature of the hashtable.

A significant (but unrelated to serialization) issue was also identified. A tremendous amount of time (almost 40% of the top 100 hottest CPU users was observed) was spent in isAssignableFrom and its inducers. This is not directly a code related issue, but the source of the usage is queued to be addressed. ObjectWave is the source of this, and there is already work underway to limit or remove its usage. This will be the best way to address this item as the poor performance of isAssignableFrom is known (see Sun CR 6461827), but there will not be a fix for this in the near future (probably an update or two away

in JDK 6). Removing the use of ObjectWave will recover significant CPU cycles for the HTS.

From an overall perspective, the HTS clearly shows signs that scalability will be a key issue as volumes increase. In the profiling that has been done to date, there is clear evidence that the code itself is not a major source of CPU consumption or latency. Rather, the semantics of the application with regard to synchronization, object allocation and the thread interaction model (high wait() and notify() usage) will be the limiting factors. Several suggestions have been made to address the synchronization issues and each one of them that is addressed will work to improve scalability.

For the object allocation issues (which really means latency due to garbage collection), it is highly unlikely that tuning garbage collection parameters will bring substantial relief. The highest allocation rate objects have been identified and measures should be taken to address them. For instance, the buffers used for sending/receiving data have already been modified to be more adaptive with regard to message size. However, analysis shows that there are still a substantial number of messages that exceed the currently configured maximum size at which a pre-allocated buffer is re-used. Wisely, this interface was designed to be runtime tunable and that threshold should be increased substantially to 32k or larger. The potential for wastage is small (250 threads with 32k pre-allocated buffers would require less than 8MB of memory) and the benefit is substantial. Similar steps should be taken for each of the high rate, transient objects that are allocated. In this manner, the frequency of the garbage collections can be significantly reduced which are adding between 20ms and 100ms latency to messages impacted.

If it is not possible to drastically reduce the GC rates in the current configuration, it may be necessary to entertain the idea of increasing the number of servers (JVM's) where possible. The production systems are configured with sufficient memory to allow for this increase in footprint (the JVM's would probably be sized and configured the same as the current number of instances).

In terms of the current wait()/notify() architecture in which threads routinely do a small amount of work before passing the transaction to the next thread, there will need to be an logic analysis done to determine how much (if any) of this work can be combined and reduce the number of threads involved to complete the transaction. Unfortunately, this is not merely a technical analysis. The semantics of the application with regard to ordering of messages is the key factor. Changing the architecture without introducing new serialization points would be a wasteful exercise. Generally, financial systems similar to this one can allow parallization of messages if the independent streams are divided by message class, symbol, or other differentiator.

Similarly, breaking the input streams (from a network perspective) into multiple streams instead of the current single one would likely provide a benefit. With a single thread processing all incoming network packets on both “sides” of the application, any latency in that thread (OS scheduling or otherwise) would likely result in increased latency or queueing. Much like above, this could be accomplished by sending different classes of messages to different network ports. In Solaris 10, independent TCP endpoints can be processed in multiple threads across CPUs and allow for greater concurrency at the network layer.

Moving forward, the skills transferred from the on-site training course should be applied to determine the most beneficial of these issues to attack based on their relative impact to the application (for HTS and other applications). DTrace can be used to give an easily quantifiable view of which contended monitors are causing the highest latencies through excessive hold times or call rates and which objects are allocated at the highest rates. As each of these are addressed, a new profile can be created that will show what benefit has been achieved as well as the next most beneficial target of optimization.