CBOE Report

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

Summary

In this follow-up engagement, the main focus was to better understand the internal implementation of the application as opposed to the broader view of the application's impact/requirements of the operating system/hardware. To that end, a significant amount of time was spent profiling the interaction of the various threads and their notification mechanisms. In addition, there was an analysis done of object allocation types and rates as the effects of garbage collection can induce significant latency bubbles in the applications.

Overall, a much better understanding of the applications and their (significant) thread usage was gained. In addition – specifically on the FE – there was a much clearer understanding of the objects allocated and in many cases, unnecessary allocations were remediated while on-site.

For the MDCAS, their was a marked reduction in CPU utilization through thread synchronization modification both based upon DTrace analysis and the application team's own analysis done out of band. The overall CPU utilization of MDCAS dropped for a given workload from a high of almost 80% to a low of about 37%. This is a significant recovery of CPU cycles that can be better utilized by the application. However, there are still some serious obstacles for MDCAS to support significant increases in volume in light of the penny spread changes that will occur in January of 2007. These will be discussed in detail in following sections.

For the FE, most time was spent in profiling the object allocation patterns in the application. In several instances, it was noted that an internally developed thread specific data object was being over allocated and these instances were removed. In addition, the unnecessary allocation of String (character array) objects was being created for a debug message that was not actually being logged. Remediation of these over-allocations resulted in improved performance as well as less work for the garbage collector. Also, modification of the generation sizes showed marked improvements in reducing both the frequency and the duration of garbage collection. In total, these changes reduced the unwanted latency of GC to less than half of their original frequency.

Detailed Analysis

For the FE, there was a detailed analysis of the objects allocated during a stress test run. The majority of the objects were character ([C) and byte ([B) arrays. Obviously, most of these are being allocated as part of a more complex object. The most frequently allocated application objects are the CDRBuffer, CDRInputStream, and CDROutputStream. These are the top inducers of memory usage and CPU

cycles associated with the object creation. As mentioned previously, it was determined that a number of the allocations were being unnecessarily done with an internally developed thread specific structure. Kevin was able to address some of these by replacing them with Java ThreadLocal's. In this manner, the object is only created once and is associated with a particular thread.

In addition to merely identifying the objects allocated, we collected the top allocation stacks for each of the above referenced types. Each of these hot allocation points can be examined to determine if there is wastage or unnecessary allocations. Addressing just the top handful of each of these should greatly reduce the amount of work the garbage collector must do. For instance, removing a generated, but never logged debugMessage method call had a noticeable impact on GC by itself. Although a very small percentage increase in the time between GC's was evident, the repetition of this exercise just a handful of times would easily result in double digit percentage gains.

An analysis of the GC logs was also done that indicated that a larger young generation would be beneficial since most objects were very short lived. Because of the speed of these systems, GC' ing as much as 500MB will only take trivially longer than 200MB. However, we can more than double the interval between the collections resulting in fewer pauses of the application threads. These pauses are effectively 15-22ms latency bubbles added to every transaction over that interval for the system.

For MDCAS, there were a number of items identified that resulted measurable improvements to performance (almost a 50% drop in CPU utilization at the same workload). For instance, in the channelAdapter, there was a 50ms wait time being used in the synchronization of **many** threads. in fact, there were about 2100 threads created to manage 100 users. With this many threads waking up at a rapid interval to compete for locks and work, the system was extremely pressured in merely trying to schedule this number of threads. Removing the 50ms wait and decreasing the number of threads had the most significant improvement in performance.

There were other items identified that resulted in decreased CPU utilization, but none on the order of magnitude as the above issues. One item that required no code changes was to place the MDCAS process in the fixed priority scheduling class. As surmised, it dramatically reduced the involuntary context switch rate and the *system* CPU time. This is due to the reduced overhead of managing thread priorities and the elimination of the side-effects of the same application threads cannibalizing each other as their priorities shift.

Other minor improvements include identifying a method (isTransportSupported) associated with client connection establishment that made thousands of calls per connection to identify the hostname and the IP address of the server system. As this should never change during the course of the application running, these calls can be eliminated by using a single call and a static object to hold the results.

In order for the MDCAS to be able to support significant increase in load or users, I believe it will require a fundamental architectural change to the application. The number of threads (and synchronization/serialization points) in the application will need to move from a model where the thread count is based on a multiplier of the connected user count. Instead, a more generic thread model that can service many simultaneous users will be required. Even with the reduced thread counts in the application, there are still over 2000 threads in the MDCAS when supporting only 100 connected users. This high thread count and dedicated nature of the threads will have significant issues with scaling.