DSO Tuning Guide

About Terracotta Documentation
This documentation is about Terracotta DSO, an advanced distributed-computing technology aimed at meeting special clustering requirements.

Terracotta products without the overhead and complexity of DSO meet the needs of almost all use cases and clustering requirements. To learn how to migrate from Terracotta DSO to standard Terracotta products, see Migrating From Terracotta DSO. To find documentation on non-DSO (standard) Terracotta products, see Terracotta Documentation. Terracotta release information, such as release notes and platform compatibility, is found in Product Information.

Release: 3.6
Publish Date: November, 2011

Documentation Archive »

Terracotta DSO Tuning Guide

Introduction

This tuning guide is an introduction to the important concepts needed to tune an application with Terracotta Distributed Shared Objects (DSO). It also provides a discussion on actual tuning practices that will help you get the best performance out of DSO.

Tuning Best Practices

Don't Jump To Conclusions

The first rule of performance testing and tuning is: Don't pre-optimize. The second rule is: Don't jump to conclusions (which is a variation of the first rule). Until performance tests are run, it's almost impossible to know where the bottlenecks are. Once you start running your performance tests, don't jump to conclusions about the cause of these bottlenecks, and about what the solution is. Proper bottleneck analysis is almost certainly cheaper than optimizing code that isn't the cause of your worst bottleneck.

Turn Off Profiling

Disable profiling hooks and tools while running performance tests to prevent them from skewing your results. However, make sure you keep verbose GC settings on. You will need to refer to them to do your analysis. Quantify the overhead of your monitoring tools so that you can factor them out of any results.

Set Min And Max Heap Settings The Same

Make sure you set your min and max heap settings to the same value. This prevents the cost of resizing the heap as your application starts.

Take Good Notes

While you are in the performance testing and tuning cycle, make sure you take good notes. You may think you don't have time to take good notes, but the truth is you don't have time *not* to. When performance tuning, and/or tracking something down that requires multiple runs or configurations of your software always, always, always take notes on each run. You should write down all the details you can think of. Some examples include:

- test settings
- CPU usage and other machine stats
- what problems you ran into.

You should also archive your test output and other artifacts and always keep a date/time stamp on your test results. This will prevent the inevitable rerunning of tests because you forgot the results, or mixing up what you have tried and not tried. It only takes one mistake to use up more time than tons of note taking would require.

Understand Memory Usage and Garbage Collection

One of the most important factors for keeping a cluster running and optimizing performance involves the efficient use of memory. Not too much memory should remain unused on a node, but having *too little* free memory can lead to node failures and even stop entire clusters. In a Terracotta cluster, the main tools to managing memory are the following:

Terracotta Virtual Memory Manager
Tune the Terracotta Virtual Manager (VMM) to balance memory needs against performance. Understanding the memory-usage characteristics of your clustered application helps you tune VMM for optimal performance. More information on tuning the VMM is given below.

- #VMM on Terracotta Server Instances – VMM causes flushing of objects from heap (the Terracotta server cache) to disk store. When set to be aggressive, VMM attempts to free memory before OutOfMemoryErrors (OOMEs) occur. Tuned to be less aggressive, VMM allows more objects to remain in the heap, allowing them to be faulted to clients more quickly (from server memory instead of disk). A table of VMM properties for Terracotta servers appears below.
- #VMM on Terracotta Clients – VMM causes objects to be cleared from the client heap by Java garbage collection. When set to be aggressive, VMM attempts to free memory before OutOfMemoryErrors (OOMEs) occur. Tuned to be less aggressive, VMM allows more objects to remain in the heap, preserving locality of reference by preventing the client from having to fault in those objects from the Terracotta server. A table of VMM properties for Terracotta clients appears below.

Java Garbage Collection

Optimizing Java garbage collection (GC) is crucial to keeping nodes from bogging down in GC cycles or running out of memory. While an efficient application does not create huge amounts of garbage needlessly, a well-configued and tune Java GC can keep up with garbage without going into long cycles. More information on tuning Java GC is given below.

Distributed Garbage Collector

The Distributed Garbage Collector (DGC) collects *shared* objects on Terracotta servers when these objects no longer exist on any client's heap, which keeps a Terracotta server's cache and disk store from filling up with garbage from cluster operations. Tuning the DGC involves balancing the need to keep a server from being burdened with too much garbage against the performance costs of DGC cycles. More information on tuning DGC is given below.

JVM Heap Settings

Appropriately sizing the heaps on Terracotta servers and clients helps ensure that there is enough memory to start and that it is allocated efficiently. More information on JVM heap settings is given below.

These tools are interrelated and each should be observed and tuned while keeping the others in mind. For example, the DGC cannot collect (from a Terracotta server) shared objects that are no longer referenced on any Terracotta client heaps until those objects are first collected on all those heaps by Java GC. An efficiently tuned Java GC helps DGC complete its task more efficiently.

A Simple Tuning Process

Make sure you can answer the following questions:

1. **Are You CPU Bound?** Make sure you run your test with a system monitor like vmstat, iostat, or similar to see if you are CPU bound on any of the machines in your cluster. Don't forget to tee the output (if possible) to a file for later reference.
2. **Are you GC bound?** Make sure you run your test with some variant of the -verbose:gc flag turned on. Remember to tee the output (if possible) to a file for later reference. If you are GC bound, it's probably time to start GC tuning.

If you are CPU bound, but not GC bound, then you can start looking for hotspots in your code. One of the best ways to do this is to take a series of thread dumps and examine them to find out where the application is spending most of its time. Start trying to optimize the worst offender, then try the test again.

Once you are not CPU bound anymore, it's time to start searching for the bottleneck. It is best to think about a non CPU bound performance problem as thread starvation rather than a network problem or a disk problem (at least at first). Continue to take thread dumps and try to understand where threads are blocked and why.

To get a clear picture of where your bottlenecks are, wrap large swaths of your slow code like so:

```
long start = System.currentTimeMillis();

//... your code here ...

long t = System.currentTimeMillis() - start;

count++;
total += t;
if(count % 1000 == 0){
  // obviously the number you divide by here is dependent on how often the code
  // is called
 System.out.println("T1 Average: " + (t/count) + " count:" + count);
}
```

Do this in multiple parts of your code and narrow in tighter and tighter until you have found the part or parts of your code that are taking the longest.

Lock Tuning

The cluster-wide locking you do in your application can have a big impact on cluster performance. The following are some rules of thumb to consider when trying to improve the performance of your locking.

Cross-Node Contention

Try to avoid lock contention across cluster nodes as much as possible. Try to partition your work to avoid heavy lock contention. As in a single JVM, you don't want highly contended locks if you can avoid them.

Lock Hopping

If you do have a highly contended lock, try making your lock acquisition more coarse-grained to batch operations within the scope of a single lock acquisition and release. This will reduce the lock hopping between threads and JVMs. An example of when to do this would be doing an operation that performs multiple gets and puts on a synchronized Map. You would be better off synchronizing at the higher level operation than auto-locking the Map operations.

Very Coarse Locking

Avoid locks around very large operations. Terracotta doesn't currently fragment transactions, so you may see poor performance or even exceed available memory if you try to do huge operations all within the scope of a single lock boundary.

Example:

```
synchronized (myClusteredObject) {
  for (long i=0; i<99999999999; i++) {
    myClusteredObject.makeAVeryLargeNumberOfChanges();
  }
}
```

This code might blow your heap.

Very Fine Locking

You might need to avoid locking within long, tight loops as the overhead of creating lots of very small transactions may negatively affect performance.

Example:

```
for (long i=0; i<99999999999; i++) {
  synchronized (myClusteredObject) {
    myClusteredObject.makeAVeryLargeNumberOfChanges();
  }
}
```

This code might be slow.

Obviously, this advice sounds like the opposite of the previous point about not locking around very large operations. The point is that you need to strike a balance between too coarse and too fine grained locking that is appropriate for your application and what you are doing within the scope of those locks. (As Terracotta implements more sophisticated batching algorithms, this will become less of a user-facing issue).

Remember that locks serve as the boundaries of transactions for Terracotta, so the amount of locking you do and the amount of change occurs within the scope of those locks will affect the throughput and the memory consumption of your application. When you've identified a locking problem, try different lock granularities systematically to find the optimal lock granularity for what you are trying to do in your application.

Unnecessary Locking

Avoid unnecessary cluster-wide synchronization. Choose carefully and only declare Terracotta autolocks around code that you want to be locked

cluster-wide.

Always Acquire Nested Locks In The Same Order

As a general concurrent programming practice, if you acquire nested locks, make sure you always acquire them in the same order or you will deadlock.

Bad Example:

```
public void doStuff() {
  synchronized (objectA) {
    // ... do stuff
    synchronized (objectB) {
      // ... do stuff
    }
    // ... do stuff
  }
}

public void doOtherStuff() {
  synchronized (objectB) {
    // do other stuff
    synchronized (objectA) {
    // Yikes! I've grabbed locks in a different order here than in
    // doStuff(). I've just coded a deadlock!
    // ... attempt to do other stuff, but maybe deadlock
    }
    // ...
  }
}
```

This code will lead to a deadlock

Good Example:

```
public void doStuff() {
  synchronized (objectA) {
    // ... do stuff
    synchronized (objectB) {
      // ... do stuff
    }
    // ... do stuff
  }
}

public void doOtherStuff() {
  synchronized (objectA) {
    // do other stuff
    synchronized (objectB) {
    // This is much better.  I'm always acquiring locks in the
    // same order.
    // ... do other stuff
    }
    // ...
  }
}
```

This code doesn't have the out-of-order lock acquisition problem.

Use Lock Logging And The Lock Profiler

It's a good idea to turn on lock logging to see what exactly is happening with your locks so you can make judgements about whether you are seeing what you want to see. See the "lock-debug" section of the Terracotta Configuration Guide and Reference for instructions on turning on lock debugging.

You should also use the lock profiler available in the Terracotta Developer Console. It is a very useful tool for tuning your locking.

Performance Tuning Tools

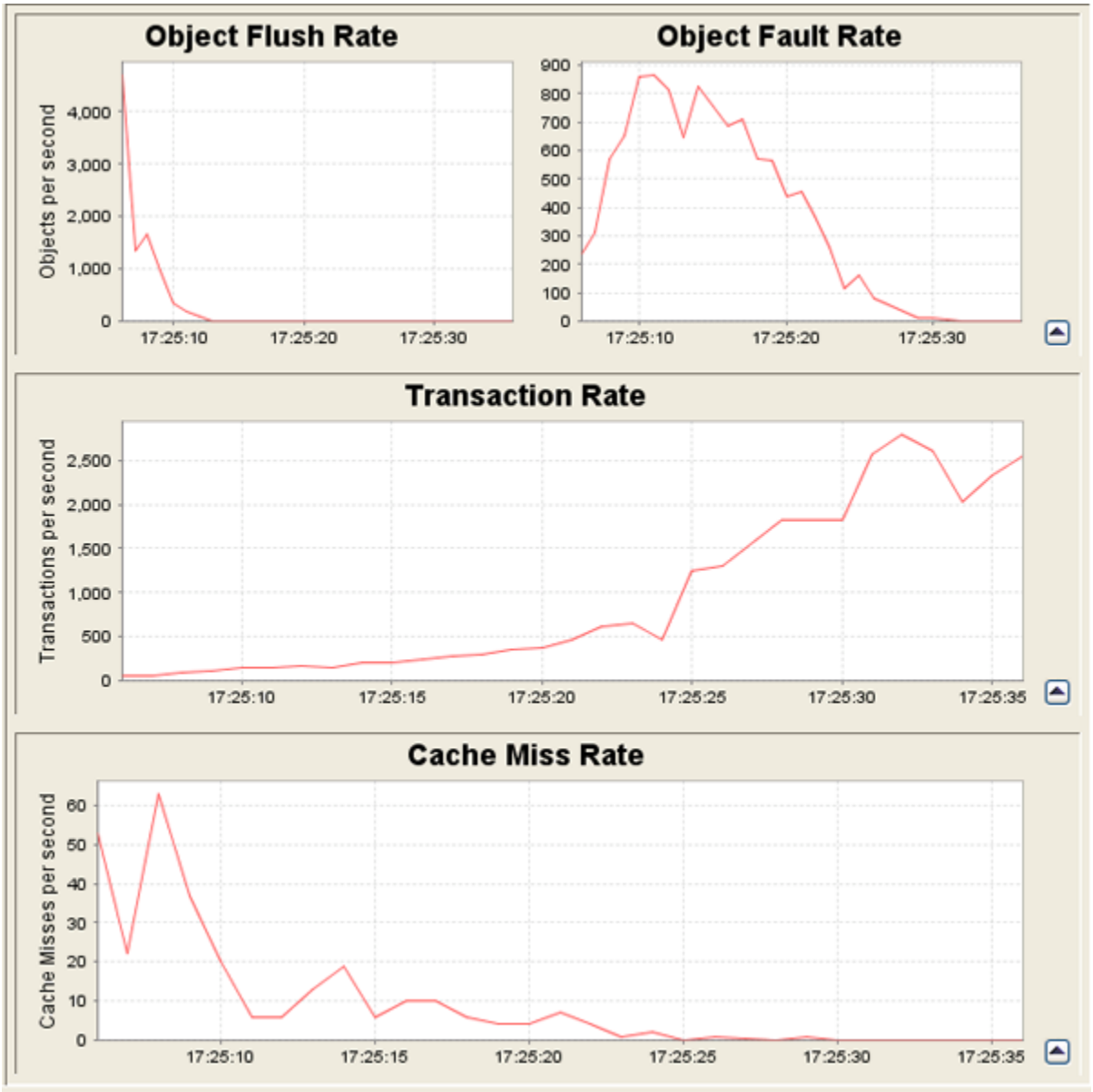
1. The Terracotta Developer Console in the Terracotta kit displays information invaluable to quantitative profiling and debugging. Along with the JConsole that ships with the JDK, it provides visibility into the inherent JMX instrumentation available within the Terracotta implementation to help you understand the runtime characteristics of your clustered application. The Developer Console also displays the following cluster-wide and per-client information:
    - Fault/flush rates
    - Number of object puts/gets
    - Number of times an object was retrieved off the disk of a Terracotta server instance
2. Increasing Terracotta debug levels (refer to the "debugging" section of the Terracotta Configuration Guide and Reference) provides detailed information around Instrumentation, Locking, Data replication, Client-reconnects etc.
3. Network monitoring tools help profile bandwidth consumption. Terracotta is expected to be low, given the fine-grained replication. Choose from several tools such as:
    - Ntop - http://www.ntop.org. Ntop is a free and useful tool that shows the network usage, similar to the Unix top command does.
    - IPTraf - http://www.iptraf.com
    - Netstat, TcpDump etc.
4. #JVM Tuning - Tools such as jstat and Sun's Visual GC are extremely useful for monitoring and profiling JVM memory usage.
5. System monitoring tools help to determine if the performance issue is due to system resources being pegged, specifically - CPU, memory, disk I/O. Nmon is a reliable system monitoring tool:
    - Nmon - http://www.ibm.com/developerworks/aix/library/au-analyze_aix/.

Terracotta Developer Console

The Terracotta Developer Console provides a wealth of information and insight into the runtime characteristics of your cluster, including:

- Fault rate – the rate at which objects are faulted into client JVMs (per application server and per Terracotta server instance).

- Flush rate – The rate at which objects are removed from client JVMs (per application server and per Terracotta server instance).
- Cache miss rate – The number of times the desired object is not in memory in a Terracotta server instance.
- Transaction rate – the rate at which locks are acquired and released for each application server, each Terracotta server instance, and the entire cluster.



In addition, the Developer Console provides detailed information on all of the shared objects by class, including the number of times each has been created, and can provide the same information in a package view or map view.

Tablular | Hierarchical | Map

Class	Creation count
org.terracotta.test.Cookie	44,500
java.util.HashMap	446
org.terracotta.test.Session	445
org.terracotta.test.UserIdCacheKey	445
java.util.LinkedList	3
com.tcclient.object.Client	3
org.terracotta.test.SharedNodeCounter	1
org.terracotta.test.SessionCache	1
java.util.IdentityHashMap	1

The latest Developer Console provides:

- a runtime view of the clustered object state
- a view of shared objects by class
- a lock profiler
- a distributed garbage collection stats viewer
- a cluster-wide thread dump tool that will allow you to take and view thread dumps simultaneously on all JVMs in the cluster.
- a statistics recorder and visualization tool

See the Terracotta Developer Console documentation for more information on the console's latest features.

JVM Heap and Garbage Collection

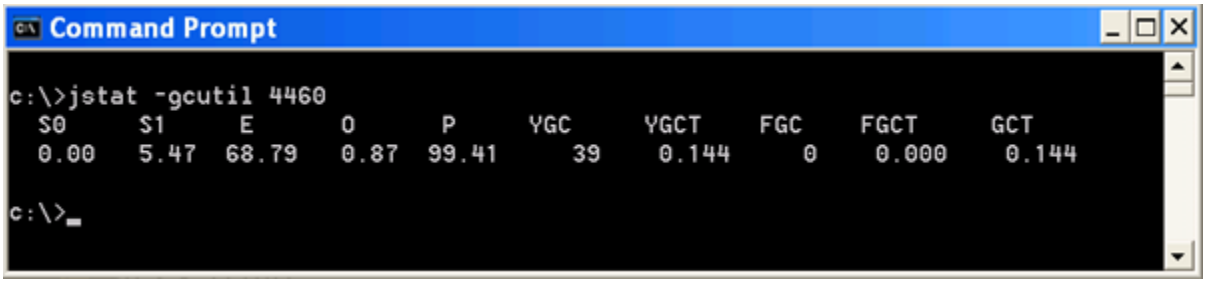
Because Terracotta effectively makes clustering a service of the runtime, tuning the runtime, or the JVM, is a very appropriate target for increasing performance. While JVM tuning is an exhaustive topic in its own right, our focus will be memory and garbage collection. We will identify a core set of tuning parameters that can be used and experimented with to arrive at optimal settings.

The first step in tuning the JVM is to get a look at how your application uses memory in real-time. There are several open source and commercial tools that provide that functionality. We'll use jstat, a free command-line tool included in the Sun JDK. Sun's Visual GC, essentially a graphical version of that tool, can also be downloaded from the Sun website.

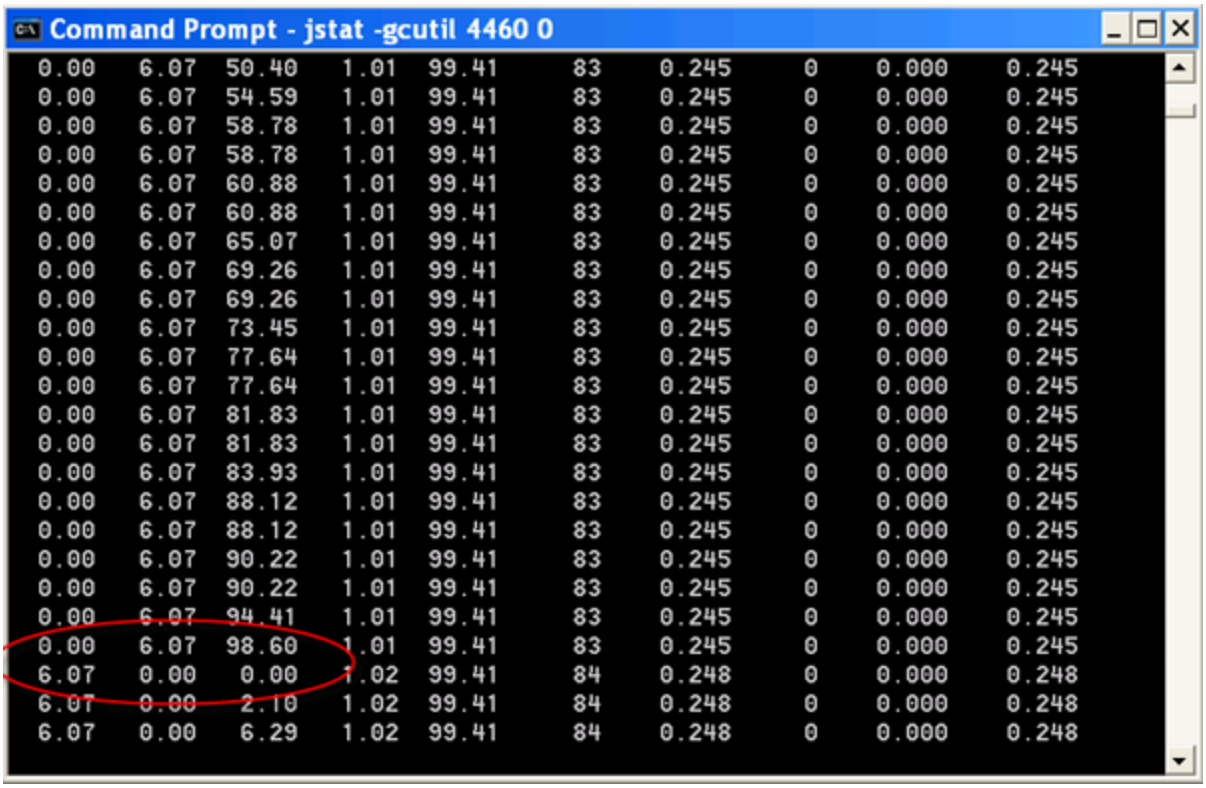
Again, to use jstat, you need a full JDK (not just the JRE). jstat is located in the bin directory of the JDK installation. For our purposes, we'll need to use two jstat parameters – the "-gcutil" option followed by the process id of the target java process.

You can retrieve the process id of a process on Windows using the graphical Task Manager tool and on Linux/Solaris using the ps command-line utility. Many Linux/Solaris boxes will have jps installed, a command-line utility that conveniently lists only Java processes.

With just those settings, we get a snapshot of the various memory regions associated with the heap in a JVM – survivor one and two, eden, old and permanent. You can see these in the following screen capture:



A third numeric parameter specifies the number of snapshots we'd like to see; a value of 0 tells jstat to run continuously. In this snapshot, we see where Eden had filled up, resulting in a brief, non-blocking garbage collection operation.



As you see your application run in real-time, you can get a sense of where heap size and/or garbage collection settings may need to be changed.

Heap settings

In general, the first consideration when tuning Terracotta is heap memory. Terracotta adds overhead to shared data structures on the order to 10-15%. As a result, an application with adequate heap settings may no longer have adequate heap settings as a clustered application. Consider increasing your heap size accordingly.

Normally, you should set the initial and maximum heap sizes to the same value. This can save the JVM from having to perform multiple memory allocations, particularly at startup. It is also recommended that you increase memory with the adding of processors to a given system. This allows for greater parallelization.

Garbage Collection (GC)

The following discussion focuses on GC tuning in the presence of Terracotta.

Since the Eden space in the heap is collected more frequently than the tenured space, and GCs in Eden are minor (not full "stop-the-world" collections), settings that allow all throwaway garbage to remain in Eden lead to more efficient GC. If your application generates a lot of throwaway garbage, increasing the size of the Eden space allows Eden to fill up but not overflow into the tenured space. If Eden is too small, throwaway garbage spills to the tenured space, which is collected only when the full GC runs.

If an application generates a lot of permanent objects, but little throwaway garbage, having a smaller Eden space can reduce overall garbage-collection time.

On a Terracotta server instance, you can enable parallel GC. Parallel GC takes longer and usually decreases the instantaneous throughput, but does not have the stop-the-world effect. The overall effect of the parallel GC is to increase the average overall throughput, as the stop the world type of GC really affects average throughput measurements.

Example GC settings

Here are some GC settings you can use:

-verbose:gc

verbose:gc is a Java command-line switch that causes detailed GC information to be output to standard out.

-XX:SurvivorRatio=8 – large survivor spaces for short-lived objects

This example sets survivor space ratio to 1:8, which results in larger survivor spaces. Smaller numbers mean larger survivor spaces. When the survivor space is large, it gives short-lived objects more time to die in the young generation (Eden and Survivor).

-XX:+UseParallelGC

This setting specifies that the parallel garbage collector should be used for the new generation, the default on server machines. See below for the -server switch.

-XX:ParallelGCThreads=20

This setting specifies the number of threads to use for parallel GC operations.

-XX:+UseParallelOldGC

This setting insures that certain portions of an old generation collection will be performed in parallel, resulting in an overall speedup of the process.

-Xmn1g

In this example, a 1GB heap size is configured for the young generation. The young generation can be collected in parallel. This helps keep short lived objects out of the old generation, which is more expensive to garbage collect.

-server switch

As of Java 5, the collector strategy used is based on the class of the machine on which the application runs. Windows boxes are considered "client" machines (meaning they use the client VM vs. the server VM). It's generally advisable to force Java to consider the box on which a Terracotta client is running to be a "server". This requires the addition of the "-server" switch to the Java command-line.

Tuning the Terracotta Distributed Garbage Collector

The Terracotta Distributed Garbage Collector (DGC) finds objects that are eligible for collection in a Terracotta server instance, then removes them from the server instance and from persistent storage. The DGC requires tuning both for optimization reasons and to remedy failures. Sometimes, for example, the DGC process is unable to reclaim objects fast enough, resulting either in poor performance, a full disk, or both. For more information on what the DGC is and how it operates, see the Concept and Architecture Guide.

Important DGC Tuning Properties

l2.objectmanager.dgc.throttle.timeInMillis

Specifies the time in milliseconds that DGC pauses (during the MARK stage) each time it reaches the l2.objectmanager.dgc.throttle.requestsPerThrottle value. Increasing this value may benefit transactions per second (TPS), while decreasing it may allow for quicker garbage collection by the DGC.

Default Value: 0 (DGC does not pause.)

l2.objectmanager.dgc.throttle.requestsPerThrottle

Specifies the number of object lookups performed by the DGC before it pauses for the duration set by l2.objectmanager.dgc.throttle.timeInMillis.

Default Value: 1000

l2.objectmanager.dgc.young.enabled

Enables (True) or disables (False) DGC collection of young-generation objects. For DGC, objects qualify as "young generation" if they have never been evicted from a Terracotta server instance's cache. If your application creates a large number of short-lived objects that are not flushed from the Terracotta server cache quickly, you can reduce the duration of more costly full DGC sweeps by enabling DGC collection of young-generation objects. By setting an optimal interval in l2.objectmanager.dgc.young.frequencyInMillis for running DGC young-generation collections, you can free memory sooner and reduce the load produced by full DGC collections.

Default Value: False

l2.objectmanager.dgc.young.frequencyInMillis

If l2.objectmanager.dgc.young.enabled is enabled, the interval (in milliseconds) between collections of young-generation objects. This property should be tested with your application and adjusted based on observed garbage-generation characteristics. If most of the objects created by your application do not become garbage or are flushed from the Terracotta server cache within the interval set for DGC young-generation collections, you should increase the interval. This interval should remain within the configured interval of normal DGC runs .

Default Value: 180000 (Collection occurs every three minutes.)

Symptoms of Badly Tuned DGC

There are three main symptoms of a badly tuned DGC:

1. A single DGC cycle runs for longer than the configured period. For example, DGC is set to run every three minutes, but it takes five minutes to complete a single cycle. Records of completed DGC cycles are listed in the Developer Console.
2. The clustered object count, also called *managed* object count or live object count, is continually increasing. The object count in a cluster is tracked by the Developer Console.
3. The amount of disk space used by the Terracotta server instance being served by the suspect DGC is continually increasing.

Identifying the Cause of the Problem and Determining a Solution

The following are common causes of cluster performance problems.

Too Much Garbage

One of the most common problems with Java applications is code that creates too much garbage. Everything referencable by a clustered object automatically becomes clustered unless it is explicitly declared transient. Because there is overhead to managing clustered data within the limited resources of a JVM, only objects that actually need to be part of the cluster should be added to distributed object graphs. In addition, while Terracotta can handle *very* large numbers of shared objects, burdening a cluster with too much garbage taxes its performance. Reducing the volume of shared objects reduces the volume of garbage, easing the burden on cluster resources.

The following are some of the ways you can decrease the volume of shared objects:

- Declare transient classes that do not need to be shared.
  See the Terracotta Configuration Guide and Reference for more information on configuring transience in Terracotta.
- If a class must be shared, where possible reduce the number of times it is instantiated in your code.

One way to discover candidate classes is to analyze the class creation count reported in the Developer Console. Begin by investigating the top most-instantiated classes listed in the Terracotta Developer Console classes browser.

DGC Interval Too Long

Sometimes, the default DGC interval is not fast enough to keep up with the rate of garbage creation. Try making the DGC interval smaller (see the Configuration Guide and Reference for details).

You can also take reduce the duration of full DGC collections by enabling the collection of DGC young generations. This type of collection is effective when there are many short-lived objects that have been garbage-collected quickly from all client heaps and have never been flushed from the Terracotta server's cache. Using the DGC young-generation collection, these objects are removed before the full DGC collection is run.

Garbage Objects Not Removed From Client JVMs Fast Enough

The DGC algorithm can't safely declare an object garbage if any client JVM has it in heap. The sooner a clustered object is collected by Java GC on the connected client JVMs, the sooner it can be collected by the Terracotta DGC, making DGC more effective. Tuning the Terracotta Virtual Memory Manager can improve the performance of garbage collection.

The Underlying Data Store Can't Keep Up

Once you have the DGC interval and the client JVM GC settings tuned such that the managed object count is not ever-increasing, you may still see the disk attached to a Terracotta server instance fill up. Although the DGC algorithm may be identifying garbage objects in a timely manner, the underlying data store is not cleaning up its records from the disk fast enough.

The relevant data-store tuning properties are as follows:

- l2.berkeleydb.je.cleaner.bytesInterval=100000000 (decrease - e.g. 20Million) - more aggressive cleaning.
- l2.berkeleydb.je.checkpointer.bytesInterval=100000000 (decrease it e.g. ~ 20MB) - this forces more frequent checkpoints.
- l2.berkeleydb.je.cleaner.lookAheadCacheSize=32768 (increase it - e.g. 65536) - lookahead cachesize for cleaning. This will reduce #Btree lookups.
- l2.berkeleydb.je.cleaner.minAge=5 (decrease it - e.g. 1) - files get considered for cleaning sooner.
- l2.berkeleydb.je.cleaner.maxBatchFiles=100 (e.g. set to 100) - Upper bounds the cleaner's backlog
- l2.berkeleydb.je.cleaner.rmwFix=true disable this e.g. false) -
- l2.berkeleydb.je.cleaner.threads=4 (increase this e.g. 8) - more threads cleaning.

The relevant Terracotta properties are as follows:

- l2.objectmanager.deleteBatchSize = 5000 (increase it - e.g. 40000) - more batched deletes
- l2.objectmanager.loadObjectID.checkpoint.maxlimit = 1000 (increase it - e.g. 4Million) - product will default to this value in the next release.

Tuning the Terracotta VMM

The Terracotta Virtual Memory Manager (VMM) frees referenced objects for garbage collection by the local JVM garbage collector. Tuning the VMM involves setting certain properties to control its behavior under specified conditions that more closely match your application's needs. The most important VMM properties are defined below. To learn how to set VMM properties using the Terracotta configuration file, see the Configuration Guide and Reference.

VMM on Terracotta Server Instances

The following are the most important VMM properties to tune on a Terracotta server instance:

l2.cachemanager.percentageToEvict

Specifies the amount of heap memory, as a percentage of the total size of the shared object set, that VMM should reclaim when l2.cachemanager.threshold is reached. Reaped objects are flushed to the Terracotta disk store and faulted back as needed.

Default Value: 10 (Evict ten percent of the total size of the shared object set.)

l2.cachemanager.criticalThreshold

When the percentage of used heap memory exceeds the specified value, triggers VMM to aggressively reclaim memory until the percentage of used heap memory is less than the value specified by l2.cachemanager.threshold.

Default Value: 90

l2.cachemanager.threshold

When the percentage of used heap memory exceeds the specified value, triggers l2.cachemanager.percentageToEvict.

Default Value: 70

l2.cachemanager.leastCount

The rise, in percent, of heap memory usage that triggers VMM.

Default Value: 2 (A rise of 2% in heap memory usage triggers VMM.)

l2.cachemanager.sleepInterval

The maximum time the VMM sleeps between two inspections of memory.

Default Value: 3000 (milliseconds)

l2.cachemanager.monitorOldGenOnly

Determines whether VMM monitors just the Old Generation heap space (true) or the entire heap space (false). Try setting to "false" (monitor entire heap space) to improve the way VMM works with certain types of Java garbage collectors.

Default Value: true (Monitor Old Generation heap space only.)

l2.cachemanager.criticalObjectThreshold

Determines whether VMM monitors the number of objects in heap memory instead of heap memory usage, which is useful when object memory requirements are well understood. If a positive integer value is specified for this property, VMM evicts objects from heap memory whenever this value is exceeded.

Default Value: -1 (VMM monitors heap memory usage.)

l2.cachemanager.logging.enabled

Enables (disables) logging of VMM statistics. Set to "true" to enable logging. Note that logging VMM statistics may impact performance.

Default Value: false (logging disabled)

When the VMM evicts objects from memory in a Terracotta server instance, the objects are flushed to the Terracotta disk store. If they are still available in the disk store, these objects can be faulted back into the server instance's heap as needed.

VMM on Terracotta Clients

The following are the most important VMM properties to tune on a Terracotta client:

l1.cachemanager.percentageToEvict

Specifies the amount of heap memory VMM should reclaim when l1.cachemanager.threshold is reached. Reaped objects are flushed to a Terracotta server instance and faulted back as needed.

Default Value: 10 (Evict ten percent of the total size of the shared object set.)

l1.cachemanager.criticalThreshold

When the percentage of used heap memory exceeds the specified value, triggers VMM to aggressively reclaim memory until the percentage of used heap memory is less than the value specified by l1.cachemanager.threshold.

Default Value: 90

l1.cachemanager.threshold

When the percentage of used heap memory exceeds the specified value, triggers l1.cachemanager.percentageToEvict.

Default Value: 70

l1.cachemanager.leastCount

The rise, in percent, of heap memory usage that triggers VMM.

Default Value: 2 (A rise of 2% in heap memory usage triggers VMM.)

l1.cachemanager.sleepInterval

The maximum time the VMM sleeps between two inspections of memory.

Default Value: 3000 (milliseconds)

l1.cachemanager.monitorOldGenOnly

Determines whether VMM monitors just the Old Generation heap space or the entire heap space. If set to false (monitor entire heap space.), may work better with certain types of garbage collectors.

Default Value: true (Monitor Old Generation heap space only.)

l1.cachemanager.criticalObjectThreshold

Determines whether VMM monitors the number of objects in heap memory instead of heap memory usage, which is useful when object memory requirements are well understood. If a positive integer value is specified for this property, VMM evicts objects from heap memory whenever this value is exceeded.

Default Value: -1 (VMM monitors heap memory usage.)

l1.cachemanager.logging.enabled
Enables (disables) logging of VMM statistics. Set to "true" to enable logging. Note that logging VMM statistics may impact performance.
Default Value: false (logging disabled)

When the VMM evicts objects from memory in a Terracotta client, the objects are flushed to a Terracotta server instance. If they are still available in the Terracotta server array, these objects can be faulted back into the client's heap as needed.

Symptoms of Badly Tuned Terracotta VMM

In an application with heavy usage of heap memory, a well-tuned VMM can keep a JVM's heap from becoming full and prevent a Terracotta client or server instance from failing. Problems with the VMM manifest themselves by throwing OutOfMemoryErrors. An OutOfMemoryError (OOME) can appear either in the client JVMs or in a Terracotta server instance as certain very large objects grow too large to fit in the heap of either the client JVM or the server instance.

Identifying the Cause of the Problem and Determining a Solution

An OOME in a Terracotta server instance may be an indication that a collection is too large to fit in the server instance's heap. A server instance can also throw an OOME when there is a very large transaction sent by a client. Whatever the cause, the problem can often be fixed by increasing the heap size of the server instance's JVM.

An OOME in a Terracotta client JVM may also an indication that a collection is too large to fit in the server instance's heap. Other causes of OOMEs in the client may be a memory leak in your application unrelated to Terracotta, or a very large transaction. To determine if there are very large logically managed objects that won't fit in memory, try the following:

- Look for big collections by inspecting the clustered object graphs in the Developer Console. Using the tree control in the Developer Console can be tedious, however, if your object graphs are complex.
- Turn on the JVM dump-heap option (-XX:+HeapDumpOnOutOfMemory). This will give you a snapshot of the Java heap at the time of an OOME that you can examine to see if any of your collections have grown to be very large. You should always run with this option on during testing and tuning. Use in production, too, if practical. For more information on the JVM heap-dump option and analysis, see this blogs.sun.com entry.

The Terracotta VMM Evictor Is Not Keeping Up

As memory pressure increases in both the client JVMs and the Terracotta server array, the VMM subsystem will attempt to reclaim memory by evicting "less frequently used" objects from the heap. This evictor thread may fall behind in some cases, allowing ever more clustered objects that should be evicted to stay in the heap. If the evictor thread falls far enough behind, you will see an OOME in either clients or server instances.

To determine if this is the problem, turn on VMM logging in the Terracotta clients and server instances. VMM log messages show the number of evictable objects and the number of objects evicted. If the number of evictable objects is large and growing, this is an indicator that the evictor is not keeping up.

Using the Developer Console, you can record and view VMM statistics as graphs.

To tune the evictor in VMM, try making the threshold and the criticalThreshold more aggressive (for example, dial down to 30). Also try increasing the percentage to evict.

Non-Partially Faulted Object Is Too Large To Fit In Memory

If a non-partially faulted object is too large to fit in memory, there are two options:

- Switch to a class than can be partially faulted.
- Partition that object into multiple separate objects instead of one big collection.

A Partially Faulted Collection Still Won't Fit In Memory

Under certain conditions, even partially faulted objects may not fit in memory:

- A partially faulted Map has a key set to large to fit in memory.
  The entire key set of partially faulted Maps is always kept in memory. If the key set is impractically large, consider partitioning that collection into multiple separate objects.
- A partially faulted collection holds literal-value types such as String.
  When a collection is partially faulted, if the value objects are references and not literal values, only the Terracotta object IDs are stored in the values instead of pulling in all of the value objects. This allows only a fraction of the data contained in the collection to be instantiated on heap. However, if the values are literal values, the entire value set is instantiated on the heap as well. This effect is amplified when the values themselves are large, such as when a Map holds large string values. For a discussion of the difference between reference and literal values, see the Concept and Architecture Guide.

If your keys or values are literals, you can replace them with wrapper objects that make reference to the value instead of the literal value itself. This indirection allows the VMM to fault the actual key or value separately from the collection.

See the Clustered Data Structures Guide for more information on how partitioning with specific data structures is handled in Terracotta.

Resources

The following are helpful resources for tuning Java

- Java Champion and Terracotta engineer Geert Bevin demonstrates how to use the Terracotta visualization tools to tune a Terracotta clustered application
- Java Tuning Whitepaper
- Tuning Garbage Collection
- Scalability Considerations

Next Steps

When you feel you understand how to tune your clustered application, you're ready to prepare for deployment.

Deploy Your Clustered Application with Terracotta »