TERRACOTTA

TERRACOTTA TECHNICAL WHITEPAPER:

Maximize Scale and Performance with BigMemory

Abstract

As your application's data needs grow, maintaining scalability and performance becomes a growing challenge. Caching data in memory helps enormously, but this often results in diminishing returns due to latencies caused by garbage collection pauses. As the Java heap size grows in a complex application, continuous, tedious, garbage collection tuning activities are often required. However, even this strategy has its limitations, leaving you with very few options.

Terracotta's BigMemory eliminates the need to tune Java's garbage collector and its associated performance bottlenecks because it eliminates full garbage collection pauses caused by large, fully occupied heaps. The result is an easy-to-deploy, 100% Java software-only solution for all data-related performance and scalability problems that can easily be used with your data-intensive application today. BigMemory gives Java applications instant access to a large memory footprint, but without the garbage collection cost.

Highlights

The following are highlights of what you will learn about BigMemory from Terracotta in this paper:

- **Remove database-related delays**
  - Ehcache keeps critical data in-memory, eliminating database, web-server, or other data access delays
  - Performance gains and latency improvements in the 10x to 100x range
- **Maximize memory usage and eliminate unpredictable GC latencies**
  - BigMemory keeps heap sizes small with no GC demands
  - GC pause time reductions from minutes to microseconds are common
- **Predictably meet your service-level agreements (SLAs)**
  - Dramatically decrease maximum latency and increase throughput
  - Eliminate latencies due to stop-world garbage collection pauses with large heaps
- **Simplify deployment and management**
  - Eliminate the need to distribute large amounts of data across multiple Java instances
- **Simple, snap-in configuration**
  - With Hibernate, BigMemory can be integrated with just a command-line switch
- **A 100% pure Java solution**
  - Seamlessly works with Java SE 5 and Java SE 6 applications

TERRACOTTA®

Contents

Overview – The Problem

Applications need speed and scale in today's hyper-fast, need-it-now world. Competitive pressures and escalating customer expectations have put a premium on application speed. Successful businesses need to scale up rapidly, which can stress traditional multi-tier application architectures.

Many applications use data stored in a central resource, such as a database, web-service, or other type of server. Accessing this data indirectly can become a bottleneck that slows application performance dramatically. However, the closer data is to an application, the faster the application executes. Bringing data directly onto the machine where the application runs offers the best performance. This is achieved by implementing an application data cache.

A cache stores data closer to the application for faster access, increased application throughput and reduced overall latency. Storing more data in cache requires a larger Java heap, which helps overall performance, but only up to a point. Problems arise as Java's heap grows, and so do the demands on Java's garbage collector.

What Is Garbage Collection?

Java helps productivity by managing memory on the programmer's behalf through automatic memory management, or garbage collection. At the most fundamental level, garbage collection involves two deceivingly simple steps:

1 – Determine which objects can no longer be referenced by an application. In Java, object graphs (chains of references) are traced to determine live and dead objects.

2 – Reclaim the memory used by dead objects (the garbage).

The work to check object "live-ness" and then reclaim dead objects takes time and must be executed from time to time to ensure that enough free memory is consistently made available for the application. Complexities arise in determining when, for how long and how often, garbage collection activities are to take place. This work directly impacts the performance and determinism of the running application.

The unpredictable nature of garbage collection makes it especially hard to manage; it's impossible to predict when it will occur and how long it will last. With most collectors, application threads are paused for some or all of the time that the garbage collector runs. Generally, as both the heap size and the number of live objects increases, the garbage collector runs more often and takes longer to complete. Our work with many enterprise customers has shown this is manageable up to an occupied heap in the range of 2GB – 4GB in size. With exhaustive tuning, larger heaps are possible, but the costs of tuning and re-tuning ever-larger heaps to house growing data caches quickly out-weights the benefits of caching.

In this paper, we discuss BigMemory, which offers a solution to this problem. BigMemory allows data caches hundreds of gigabytes in size without requiring or increasing the demands on Java's garbage collector. The result allows you to use all of your server's physical memory, with complete control over application performance and latency, free of large GC pauses.

Snap in Performance

With more than 500,000 deployments, including the majority of the Global 2000, the de facto caching standard for enterprise Java is Terracotta's Ehcache. It ships as the default cache of many popular applications, containers and frameworks including Atlassian, ColdFusion, Grails, Hibernate, Liferay, Salesforce and Spring, among others. If it isn't already in your application, you can download the library, snap it in and configure it in a matter of hours to remove performance bottlenecks and improve application response times.

What Is a Cache?

At one end of the scalability continuum—in applications that run on a single machine—adding capacity means maximizing raw performance. Caching is usually the easiest and most effective way to reduce latency and increase throughput. A cache stores results that are relatively expensive or time-consuming to retrieve or compute so that subsequent work that relies on those results may complete without incurring the cost of repeated operations. At a stroke, adding effective caching can improve application performance by orders of magnitude with minimal code impact.

In recent tests performed by a customer, Ehcache delivered 90% database load reduction and orders-of-magnitude lower latency at completely linear throughput from 10 threads to 100 in a single application server. At 10 worker threads, the application performed one million transactions/second. At 50 threads, the application performed six million transactions/second. This linear scale illustrates the internal efficiency and inherent parallelism of the Ehcache library.

Add BigMemory to Any Enterprise Ehcache Edition

To maximize memory use and performance, enhance your deployment with BigMemory, which snaps into Enterprise Ehcache to provide an in-process, off-heap cache that's not subject to Java Garbage Collection. BigMemory provides fast, local access for large amounts of data, without GC pauses or tuning. BigMemory supports both standalone and distributed off-heap caching.

Let's take a closer look at how BigMemory revolutionizes Java's memory model for large application data caches. Simplify your deployment by consolidating the number of JVMs needed to run your application and/or the Terracotta Server Array.

A Deeper Look at Garbage Collection

Garbage collection activity is typically a source of an unpredictable amount of latency in a Java application. There are different algorithms and approaches to garbage collection; the most common are:

- Concurrent collection: the work can be performed in parallel to application threads (referred to as concurrent GC)
- Parallel collection: GC work is performed by parallel GC worker threads (and not necessarily concurrent with application threads).

These algorithms can be combined, but aren't always (i.e. a parallel collector may not support concurrent collection). Additionally, a concurrent collector may use multiple GC threads to perform parallel collection. Java SE offers both parallel and concurrent collectors, both of which halt application threads from time to time (referred to as a stop-world pause). To understand why, let's examine how the Java heap is used and collected.

In Java applications, almost all objects have a very short lifetime, and often the most recently created objects have the shortest lives. To optimize GC activity based on this, the collector maintains multiple areas of the heap broken down by object age: a young generation of objects, and an old generation of objects. In fact, even the generations are composed of other spaces, but we'll ignore this detail for simplicity.

Most objects are initially created in the young generation. When GC work is performed, it takes place in either the young or old generations. GC in either area does not occur until free space in that region drops below a certain threshold. The spaces are sized such that collections occur in just the young generation most frequently, where surviving live objects are marked. This is referred to as a minor collection, which occurs often, and is tuned to be very fast.

When an object in the young generation survives a certain number of minor collections, it's moved to the old generation. When the old generation is mostly occupied, GC will occur in this area (referred to as a major collection). Given the larger size of the space, major collections occur less frequently, but take longer to complete. In fact, a different collector and algorithm is used in this space as it focuses on memory fragmentation and allocation time.

Typically, many young generation collections will occur before an old generation collection is needed. Occasionally, however, a young generation collection will be halted because the old generation is too full to move survivors to it. Instead, the old generation collector will be used to sweep and collect objects across the entire heap (including the young, old and permanent generations). This is referred to as a full collection and usually results in relatively large stop-world pauses. Even Java's Concurrent Mark-Sweep (CMS) collector requires stop-world pauses at certain times.

The Garbage Collection Myth

Most people incorrectly think that collecting dead objects takes time when, in fact, it's really the number of live objects that affects GC performance most. As the Java heap becomes occupied with an increasing number of live objects, full collections occur more often and will each require more time to complete. The result is an increasing number of stop-world pauses in your application for increasing lengths of time. The larger the heap—and the more occupied it becomes—the greater the latencies in your application. BigMemory helps to avoid large, occupied heaps due to large data caches and therefore eliminates full GC, stop-world pauses. Let's examine this now.

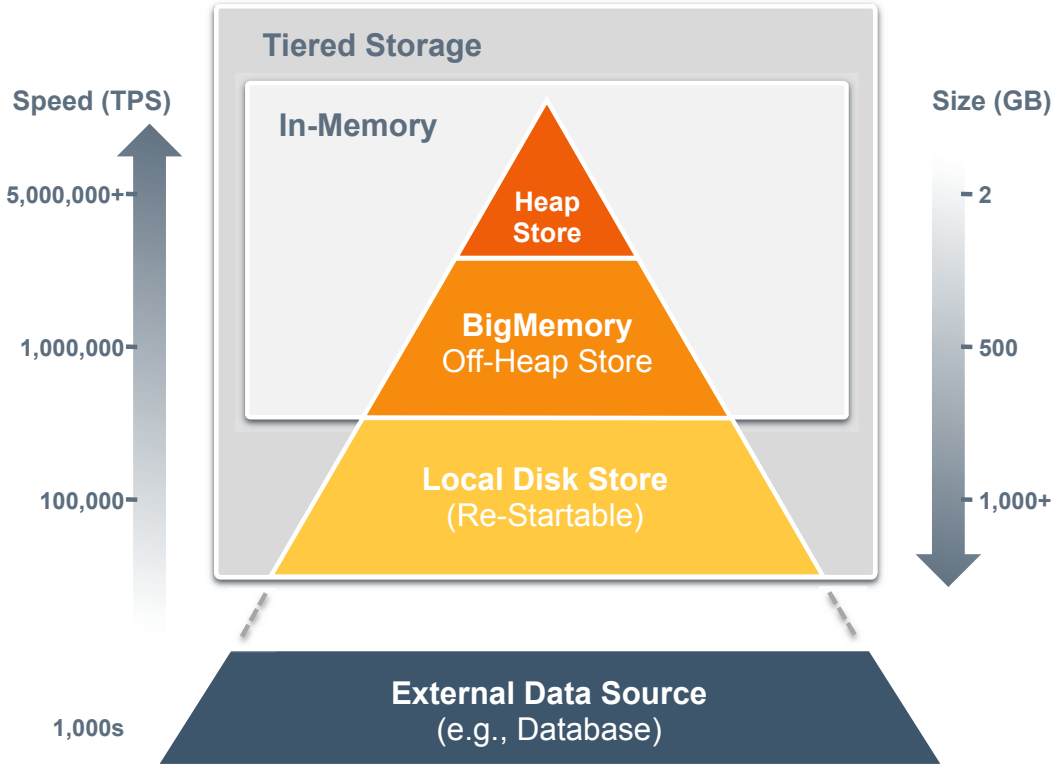
Maximize Memory Use with BigMemory

Conventional in-process Java caches store cached data in the Java heap and are therefore subject to garbage collection pauses. BigMemory is pure Java and enables data caching off the Java heap. As a result, the cached data is not visible to Java's garbage collector, and reduces the need for a large Java heap. Further, the Java heap can remain consistently small, further controlling and reducing garbage collection related pauses and latencies, even while your Ehcache size grows. And since the cache resides in-process (within your Java application), accessing the data remains very fast—around three orders of magnitude faster than a cache distributed across multiple Java virtual machines.

BigMemory gives Java applications instant access to a large memory footprint, but without the garbage collection cost. BigMemory solves three main problems commonly seen in Java applications with high data demands:

1. Database-related delays: BigMemory keeps critical data in-memory, eliminating costly database, web-server, or other data access delays.
2. Unpredictable GC latencies: BigMemory keeps heap sizes small with no GC demands.
3. Complicated deployment and management: BigMemory eliminates the need to distribute large amounts of data across multiple Java instances, but can still keep hundreds of gigabytes or more of data cached in-process.

To achieve this, BigMemory is designed as a tiered storage engine, which looks like the following:



This figure shows a system where data is consistently stored as close to the application code as possible, but without overloading the Java heap and its associated garbage collector. At the lowest layer, data is stored within an external database, which represents the slowest access times. BigMemory aims to eliminate as much access to this layer as possible to improve application performance.

The top-most layer represents the area within the Java heap that BigMemory keeps the most frequently used data, allowing for read/write latencies less than 1 microsecond. The layer immediately below the heap represents BigMemory's in-memory cache that's a bit further away from the heap, and hidden from the garbage collector so that it never causes a pause in the JVM while it sits there resident. Caches hundreds of gigabytes in size can be accessed in around 100 microseconds with no garbage collection penalties.

For applications using the Terracotta Server Array as a distributed cache, BigMemory maximizes the memory available to each node in the server array. With more memory at the disposal of each Terracotta server node, a terabyte-scale distributed cache is delivered with a fraction of the number of nodes. In customer deployments, we typically see the number of servers consolidate by a factor of four or more.

Note: with Ehcache, data is cached automatically according to most-frequently-used algorithms, and placed within the proper layer in the tiered storage engine as your application runs and its data patterns analyzed. To avoid re-creating this cache when your application is restarted, Ehache can be configured with a backing disk store to keep large caches on disk in addition to in memory. When you restart any application node, the persistent disk store is then ready to use as soon as your application begins execution.

A Data Experiment

The tiered combination of configurable in-memory caches backed by durable on-disk storage allows high-performance access to very large caches without requiring hundreds of servers to fit all of the cache in memory.

To test the benefits of BigMemory to the extreme, we tried an experiment that required caching up to 350GB of data in memory within a Java application. We began the experiment with 100GB of data cached into a 150GB Java heap. Without tuning, the result was unusable, as back-to-back full garbage collection cycles consumed all of the processing time. After much tuning, and increasing the heap size to 200GB to reduce the total heap occupancy, the application ran but with terrible performance, as it was often interrupted by long garbage collection pauses. Clearly keeping large caches in heap is not a scalable solution.

This is the problem scenario that motivated the development of BigMemory, which uses an off-heap memory store to cache data, free of garbage collection. To consider the wide variations of cache use, from write-through to read-only, both are taken into account. The test scenario is a combination of the two cases: a cache where access is split evenly between reads and writes.

The Test Setup

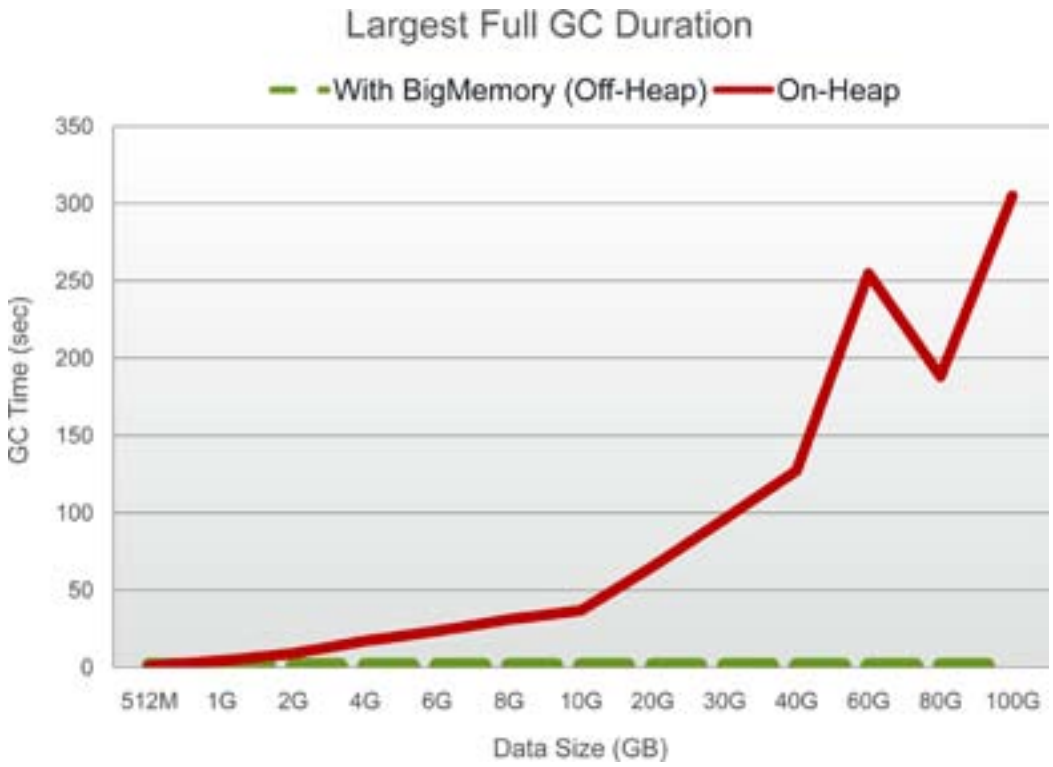
Our test environment consists of a server with 24 cores and 378GB of RAM, running Java SE 1.6.0_21 on Red Hat Enterprise Linux. All software, including the OS and Java virtual machine, is 64bit, allowing for a large memory address space. The test application was run in two test cases:

1. With a large Java heap (250GB) for on-heap cache.

2. With a small Java heap (2GB), and a 350GB BigMemory cache.

In both cases, we modeled a cache scenario where a “hot set” of 10% of the data is accessed through the cache 90% of the time. The results were astonishing, as the application performance without BigMemory degraded quickly and consistently as the cache size increased beyond 4GB. However, the BigMemory test case maintained very good performance and latency, consistently, as the cache reached 350GB in size.

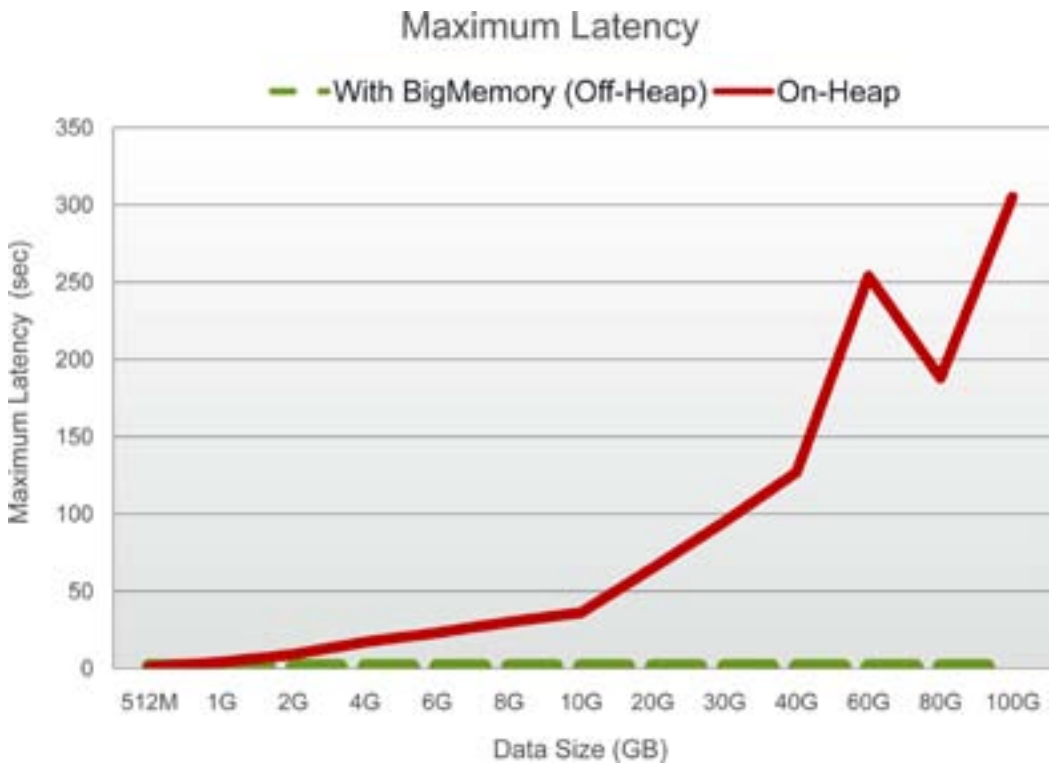
In both test cases, we measured key performance parameters such as total time spent garbage collecting as the cache size grew, as well as application throughput (i.e. transactions per second) and maximum latency (presumably caused by garbage collection activity). The chart below compares the garbage collection activity for both test cases:



With the on-heap data cache, garbage collection times increase dramatically with the growth in heap occupancy until the application is non-responsive. With BigMemory, however, since the data cache is kept in-memory but off of the Java heap, GC times remain constant and small even as cache size increases. This is because BigMemory cache is not subject to garbage collection.

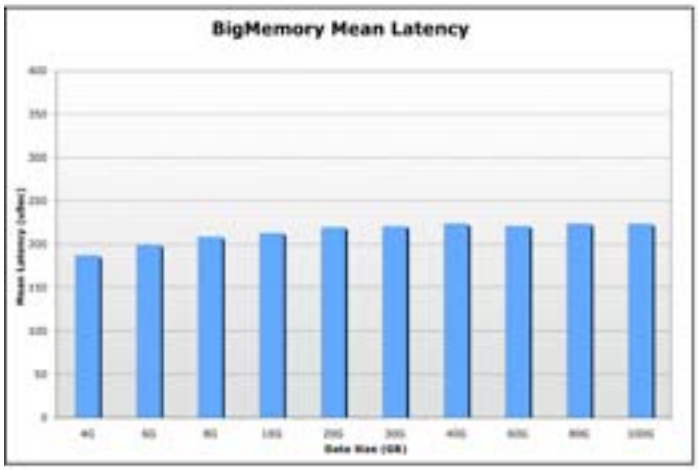
> *"We are able to reduce the heap size and get fast, local access for large amounts of data."*
>
> – Joe Caisse, CTO
> New Digital Media

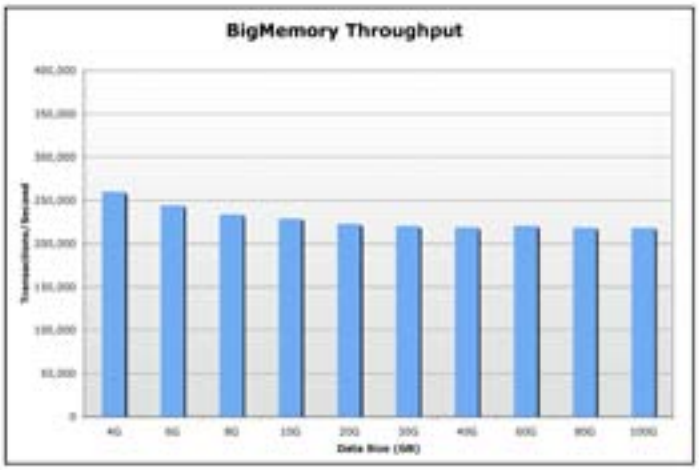
We see the same result with latency, as shown in the chart below:



As expected, as GC durations increase in the on-heap cache test case, so does maximum latency. However, since BigMemory cache size doesn't affect the Java heap, GC times remain consistent, and mostly non-existent, even as the cache size grows.

As we focus and expand on the BigMemory results for throughput and latency times in this test, we see the numbers remain consistent with excellent performance as the cache grows:





These charts prove that BigMemory provides consistent performance as the cache grows from 4GB in size to 150GB in size, and beyond. It also proves that BigMemory enables the following for your data-centric applications:

- Maximize memory usage in your servers
- Predictably meet your service-level agreements (SLAs) in terms of maximum latency and throughput even with very large data cache sizes (350GB or more)
- Simplify your deployment, as you don't need to divide and distribute your cache

> *"It's almost ridiculous how easy it is to deploy… and the scalability is awesome."*
>
> – Shane Shelton, Director of IT Operations
> Archipelago Learning

BigMemory Use Cases

BigMemory is broadly applicable, suitable for many types of applications across the enterprise. BigMemory can improve the speed, scalability, and predictability of any application constrained by garbage collection. This includes cases where:

- Garbage collection tuning is an ongoing, time-consuming, task
- Applications run on 32-bit or 64-bit versions of the Java virtual machine
- Cache sizes are as small as 2GB to as large as 350GB
- Applications use local or distributed caches

BigMemory gives Java applications instant access to a large memory footprint, but without the garbage collection cost. Terracotta BigMemory provides the following benefits:

- Eliminate latencies due to stop-world garbage collection pauses with large heaps
    - o Pause time reductions from minutes to microseconds are common
- Maximize the memory utilization of your existing servers
- Simple, snap-in configuration
- With Hibernate, BigMemory can be integrated with a simple switch
- Performance gains and latency improvements in the 15x to 100x range:
    - o 100x improvement measured using disk-based databases over a LAN
    - o 15x improvement using RAM-based solid-state drives (SSD).
    - o Greater performance improvements with Flash-based SSDs.
- A 100% pure Java solution

The BigMemory Tutorial

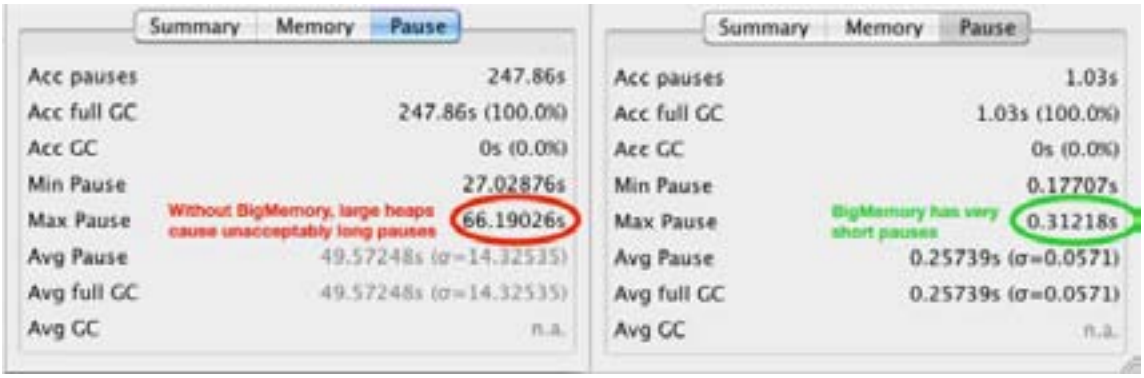
To experience the benefits of BigMemory first-hand, download the Enterprise Ehcache software with BigMemory, the trial license key, and the tutorial application, all available on Terracotta.org. The tutorial comes with configurations to test BigMemory with a range of cache sizes. Choose one that matches your caching needs and the physical memory available in your deployment hardware as closely as possible—configurations range from a 1GB up to a 500GB cache.

For each cache size configuration, the tutorial contains two test scripts: the first to run with the cache completely on the Java heap, the second with the cache off the Java heap using BigMemory. As you run the tutorial, all throughput and latency data, along with GC activity output, is stored in log files for further examination. A summary of the results is presented at the end of the test for quick comparison.

In our tests, running the on-heap configuration for a 20GB cache yielded an average throughput of 261,618 transaction-per-second (TPS), with a total run time of 229,623 milliseconds (ms). This includes a warm-up phase and several test runs to ensure peak performance. Running with BigMemory in the same test configuration (20GB cache, same hardware) yielded an average throughput of 819,998 TPS, with a run time of only 73,127ms. That's more than a three-fold increase in throughput and performance over the on-heap test.

GC Activity Comparison

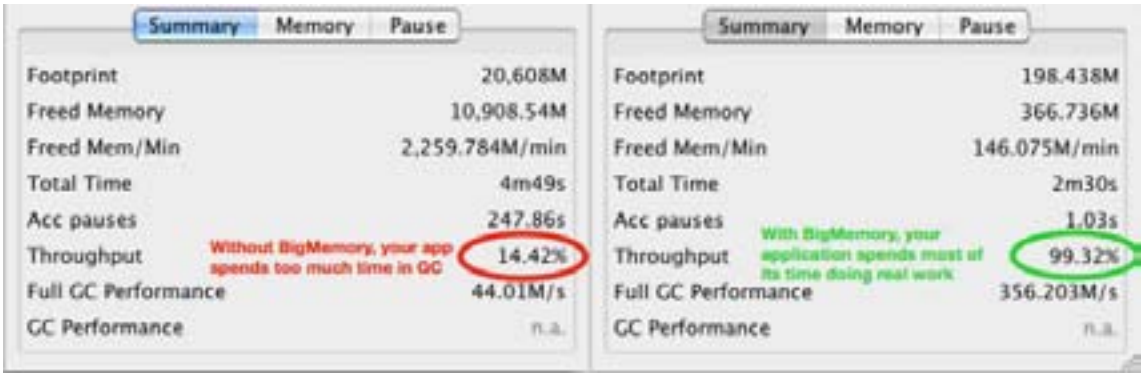
The garbage collection activity shows just how much time is wasted in the on-heap test configuration, and just how much of a savings BigMemory yields. Using the GC logs from the tests, along with an open-source analysis tool called GCViewer, you can see the dramatic differences between the two. For instance, the following diagram summarizes the results from each test configuration (on-heap test results on the left; BigMemory on the right):



The output above shows the total time spent garbage collecting during the test, along with the longest and shortest GC pause durations, and the average duration. On the left, with a 20GB Java heap to hold the 20GB cache, we see that over 247 seconds were spent garbage collecting, with a maximum pause time of 66 seconds, and an average of almost 50 seconds per pause. Contrast this with the BigMemory test run with a much smaller heap since the 20GB cache is kept in

BigMemory's off-heap memory cache. Here, the average pause was _ of a second, and the total garbage collection time was only 1 second for the same test.

Looking at the data to measure throughput further proves the benefits of BigMemory. For instance, the following diagram summarizes the throughput results from each test configuration, as discussed above:



The output above clearly shows that, with BigMemory, Java spends most of its time running application code versus performing GC activity. With the on-heap cache, the results are reversed: most of the time is spent performing GC activity, with application code running less than 15% of the time. In these tests, in terms of throughout and latency, BigMemory clearly provides superior performance when compared with a cache that resides on the Java heap.

Get Started

Download the free trial of BigMemory from Terracotta.org and try it with your application today, or run it with the tutorial discussed above to see the results first hand. For more information on evaluating BigMemory or for pricing, contact Terracotta sales.

About Terracotta

Terracotta's software products provide snap-in performance and scale for enterprise applications. Our flagship product, Enterprise Ehcache, extends the capabilities of Ehcache, the de facto caching standard for enterprise Java and the default cache for Hibernate, Spring, Grails and other leading frameworks.

With more than 500,000 enterprise deployments, including the majority of the Fortune 2000, Terracotta is behind some of the most widely used software for application scalability, availability and performance. Terracotta is a wholly owned subsidiary of Software AG. For more information, see www.terracotta.org.

Terracotta, Inc.
575 Florida St. Suite 100
San Francisco, CA 94110
USA

Product, Support, Training and Sales Information
sales@terracottatech.com

USA Toll Free
+1-888-30-TERRA

International
+1 415 738-4000

Terracotta China
china@terracottatech.com
+1 415 738-4088