Tuning {brandname} 10.0 Performance

Table of Contents

This guide will give you information and tweaks about tuning {brandname} performance (both server and library mode).

Chapter 1. Capacity planning

Data in {brandname} is either stored as plain Java objects or in a serialized form, depending on operating mode (embedded or server) or on specific configuration options such as store-as-binary. For more information, see store-as-binary. Data size can be estimated using sophisticated tools like Java Object Layout and the total amount of required memory can be roughly estimated using the following formulas:

Term `overhead` is used here as an average amount of additional memory (e.g. expiration or eviction data) needed for storing an Entry in a Cache.

In case of Local or Replicated mode, all data needs to fit in memory, so calculating the amount of required memory is trivial.

Calculating memory requirements for Distributed mode is slightly more complicated and requires using the following:

Where:

- `Total Data Set` - Estimated size of all data
- `Nodes` - The number of nodes in the cluster
- `Node Failures` - Number of possible failures (also `number of owners - 1`)

Calculated amount of memory should be used for setting `Xmx` and `Xms` parameters.

JVM as well as {brandname} require additional memory for other tasks like searches, allocating network buffers etc. It is advised to allocate no more than 50% of memory with living data when using {brandname} solely as a caching data store, and no more than 33% of memory with living data when using {brandname} to store and analyze the data using querying, distributed execution or distributed streams.

When considering large heaps, make sure there's enough CPU to perform garbage collection efficiently.

1.1. Java Virtual Machine settings

Java Virtual Machine tuning might be divided into sections like memory or GC. Below is a list of helpful configuration parameters and a guide how to adjust them.

1.1.1. Memory settings

Adjusting memory size is one of the most crucial step in {brandname} tuning. The most commonly used JVM flags are:

- `-Xms` - Defines the minimum heap size allowed.
- `-Xmx` - Defines the maximum heap size allowed.
- `-Xmn` - Defines the minimum and maximum value for the young generation.

- `-XX:NewRatio` - Define the ratio between young and old generations. Should not be used if -Xmn is enabled.

Using `Xms` equal to `Xmx` will prevent JVM from dynamically sizing memory and might decrease GC pauses caused by resizing. It is a good practice to specify `Xmn` parameter. This guaranteed proper behavior during load peak (in such case {brandname} generates lots of small, short living objects).

1.1.2. Garbage collection

The main goal is to minimize the amount of time when JVM is paused. Having said that, CMS is a suggested GC for {brandname} applications.

The most frequently used JVM flags are:

- `-XX:MaxGCPauseMillis` - Sets a target for the maximum GC pause time. Should be tuned to meet the SLA.
- `-XX:+UseConcMarkSweepGC` - Enables usage of the CMS collector.
- `-XX:+CMSClassUnloadingEnabled` - Allows class unloading when the CMS collector is enabled.
- `-XX:+UseParNewGC` - Utilize a parallel collector for the young generation. This parameter minimizes pausing by using multiple collection threads in parallel.
- `-XX:+DisableExplicitGC` - Prevent explicit garbage collections.
- `-XX:+UseG1GC` - Turn on G1 Garbage Collector.

1.1.3. Other settings

There are two additional parameters which are suggested to be used:

- `-server` - Enables server mode for the JVM.
- `-XX:+ UseLargePages` - Instructs the JVM to allocate memory in Large Pages. These pages must be configured at the OS level for this parameter to function successfully.

1.1.4. Example configuration

In most of the cases we suggest using CMS. However when using the latest JVM, G1 might perform slightly better.

32GB JVM

```
-server
-Xmx32G
-Xms32G
-Xmn8G
-XX:+UseLargePages
-XX:+UseConcMarkSweepGC
-XX:+UseParNewGC
-XX:+DisableExplicitGC
```

32GB JVM with G1 Garbage Collector

```
-server
-Xmx32G
-Xms32G
-Xmn8G
-XX:+UseG1GC
```

1.2. Network configuration

{brandname} uses TCP/IP for sending packets over the network (for both cluster communication when using TCP stack or when communication with Hot Rod clients)

In order to achieve the best results, it is recommended to increase TCP send and receive window size (refer to you OS manual for instructions). The recommended values are:

- send window size - 640 KB
- receive window size - 25 MB

1.3. Number of threads

{brandname} tunes its thread pools according to the available CPU cores. Under Linux this will also take into consideration taskset / CGroup quotas. It is possible to override the detected value by specifying the system property `infinispan.activeprocessorcount`.

Java 10 and later can limit the number of active processor using the VM flag `-XX:ActiveProcessorCount=xx`.

1.4. Number of threads (Server mode only)

Hot Rod Server uses worker threads which are activated by a client's requests. It's important to match the number of worker threads to the number of concurrent client requests:

Hot Rod Server worker thread pool size

```
<hotrod-connector socket-binding="hotrod" cache-container="local" worker-threads="200
">
   <!-- Additional configuration here -->
</hotrod-connector>
```

1.5. Cache Store performance

In order to achieve the best performance, please follow the recommendations below when using Cache Stores:

- Use async mode (write-behind) if possible

- Prevent cache misses by preloading data
- For JDBC Cache Store:
  - Use indexes on `id` column to prevent table scans
  - Use PRIMARY_KEY on `id` column
  - Configure batch-size, fetch-size, etc

1.6. Hints for program developers

There are also several hints for developers which can be easily applied to the client application and will boost up the performance.

1.6.1. Ignore return values

When you're not interested in returning value of the `#put(k, v)` or `#remove(k)` method, use `Flag.IGNORE_RETURN_VALUES` flag as shown below:

Using Flag.IGNORE_RETURN_VALUES

```
Cache noPreviousValueCache = cache.getAdvancedCache().withFlags(Flag
.IGNORE_RETURN_VALUES);
noPreviousValueCache.put(k, v);
```

It is also possible to set this flag using ConfigurationBuilder

Using ConfigurationBuilder settings

```
ConfigurationBuilder cb = new ConfigurationBuilder();
cb.unsafe().unreliableReturnValues(true);
```

1.6.2. Use `Externalizer` for marshalling

{brandname} uses JBoss Marshalling to transfer objects over the wire. The most efficient way to marshall user data is to provide an AdvancedExternalizer. This solutions prevents JBoss Marshalling from sending class name over the network and allows to save some bandwidth:

User entity with Externalizer

```
import org.infinispan.marshall.AdvancedExternalizer;

public class Book {

   final String name;
   final String author;

   public Book(String name, String author) {
      this.name = name;
      this.author = author;
   }

   public static class BookExternalizer
            implements AdvancedExternalizer<Book> {

      @Override
      public void writeObject(ObjectOutput output, Book book)
            throws IOException {
         output.writeObject(book.name);
         output.writeObject(book.author);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), (String) input.readObject());
      }

      @Override
      public Set<Class<? extends Book>> getTypeClasses() {
         return Util.<Class<? extends Book>>asSet(Book.class);
      }

      @Override
      public Integer getId() {
         return 2345;
      }
   }
}
```

The Externalizer must be registered in cache configuration. See configuration examples below:

Adding Externalizer using XML

```
<cache-container>
   <serialization>
      <advanced-externalizer class="Book$BookExternalizer"/>
   </serialization>
</cache-container>
```

Adding Externalizer using Java

```
GlobalConfigurationBuilder builder = ...
builder.serialization().addAdvancedExternalizer(new Book.BookExternalizer());
```

1.6.3. Storing Strings efficiently

If your strings are mostly ASCII, convert them to `UTF-8` and store them as `byte[]`:

- Using `String#getBytes("UTF-8")` allows to decrease size of the object
- Consider using G1 GC with additional JVM flag `-XX:+UseStringDeduplication`. This allows to decrease memory footprint (see JEP 192 for details).

1.6.4. Use simple cache for local caches

When you don't need the full feature set of caches, you can set local cache to "simple" mode and achieve non-trivial speedup while still using {brandname} API.

This is an example comparison of the difference, randomly reading/writing into cache with 2048 entries as executed on 2x8-core Intel® Xeon® CPU E5-2640 v3 @ 2.60GHz:

Table 1. Number of operations per second (± std. dev.)

Cache type	single-threaded cache.get(...)	single-threaded cache.put(...)	32 threads cache.get(...)	32 threads cache.put(...)
Local cache	14,321,510 ± 260,807	1,141,168 ± 6,079	236,644,227 ± 2,657,918	2,287,708 ± 100,236
Simple cache	38,144,468 ± 575,420	11,706,053 ± 92,515	836,510,727 ± 3,176,794	47,971,836 ± 1,125,298
CHM	60,592,770 ± 924,368	23,533,141 ± 98,632	1,369,521,754 ± 4,919,753	75,839,121 ± 3,319,835

The CHM shows comparison for ConcurrentHashMap from JSR-166 with pluggable equality/hashCode function, which is used as the underlying storage in {brandname}.

Even though we use JMH to prevent some common pitfalls of microbenchmarking, consider these results only approximate. Your mileage may vary.