Java Application Performance Tuning for AMD EPYC™ Processors

Publication #	**56245** Revision: **0.70**
Issue Date:	**January 2018**

Advanced Micro Devices

Contents

List of Figures

Revision History

Date	Revision	Description
January 2018	0.70	Initial release.

Overview

Java is a widely accepted language used by programmers for many data-oriented applications, popular for its simple, secure, robust, interpreted, and multithreaded features. From its inception, the language was designed to provide the flexibility for developers to code with features including architecture neutral, automatic garbage collection, object oriented and inheritance.

However performance becomes the key factor once your Java application is ready after testing and debugging. Tuning is required to achieve the expected performance if an application fails to perform well.

Garbage collection (GC) tuning, tuning for architectural features like NUMA (non-uniform memory access), runtime/threads tuning, hotspot compiler, and kernel parameter tuning are some of the important items that may affect Java server performance.

Using SPECjbb® 2015 as a real world application, this document explains each of these tuning methods.

Sun Hotspot JVM comes with three options:

1. Classic – Disables the Hotspot JIT compiler.
2. Client – Mainly for client applications. Gives instant performance.
3. Server – For server applications, takes time for profiling for HOT methods, then delivers the best performance.

Use option –client or -server according to application deployed as client or server.

GC Tuning

For a complete description of Java garbage collection basics, see *http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html*. A brief summary is included here.

Allocating the heap as minimum as required helps the application to run smoother. The JVM heap is mainly divided in two generations called *young generation* (“young gen”) and *old generation* (“old gen”). For most of the Java server class of applications, 90% of the objects die very young and can be collected during young gen garbage collection (GC).

The young generation can further have an *Eden Space* and two survivor spaces called *From* and *To*. Survivor spaces contains the tenured objects, once the object reaches certain threshold (again a parameter), it gets transferred to “old” gen.

When the Eden space is full, the young gen GC recovers the memory, occurring for a very short period of time, hence the application experiences a short pause that affects performance.

An object surviving for longer periods is kept in old gen. Full GC occurs when the whole heap is full and has no space for new objects, but experiences a longer pause time to clean the full heap memory. To avoid longer pauses by full GC, the old gen size should be minimized as low as possible.

GC tuning is typically performed by studying the GC profiles from the application. These profiles are written by the JVM with the specific flags like `–Xloggc` or `–XX:+PrintGCDetails`. Such profiles help us to understand the behavior of the GC, like amount of memory freed, amount of time spent in doing minor (Eden) and major (full) GCs, amount of heap space wasted (unused), etc. Using such profiles, the developer has to reduce/increase the sizes of the different pieces of the heaps (like Eden, Survivor space, Old Gen) optimally so that the amount of GC time spent in the application is less than 1% of the total time spent in the application. For example, if the application runs for about an hour, the amount of GC time spent in the application (for a healthy application) should not be more than two minutes. This may vary depending on the application. As a case study, we have provided the tuning parameters for SPECjbb® 2015 in Appendix A.

Tuning for Exploiting NUMA Architecture

NUMA architecture is prevalent in servers which have many cores and large memory in today's current technology. The AMD EPYC™ processor also exploits this to get the best performance out of all cores available (1-128T). However, the application has to be aware of NUMA architecture and tune the application accordingly to get the best performance.

The AMD EPYC 7601 processor architecture (see Figure 1) has four dies in each socket (1P). Each die has two CCX (Core-complex) units. Each CCX has 4C/8T with its own L1 and L2 (512kB). L3 (8MB) cache is shared across cores of a CCX. Since a thread running on one die runs optimally when it accesses memory from the memory channels connected to the same die, it is best to ensure each Java instance (for example) runs on a single die with memory allocations/accesses occurring from the channels connected to that die. In fact, we have seen that if the Java application is run as multiple instances and each Java instance is bound at the die or the CCX level and bind the memory to the local memories (connected to the local channels), the performance on the AMD EPYC processor peaks.

Using the AMD EPYC 7601 processor architecture (64C/128T, 512GB) as an example, having one Java instance per CCX (4C/8T, 32 GB)/total 16 JVMs improves performance on the overall system. Hence, binding the JVM process to CCX (4C/8T, 32GB) or Node (2CCX-8C/16T, 64GB) can provide the best performance.

The following figure shows how the SPECjbb® 2015 Multi JVM instance is run on each CCX on the AMD EPYC 7601 processor. In the figure, G1, G2 etc refers to the group of the SPECjbb 2015. When we run in this configuration, we also make sure we bind the memory of each group to the local memory.

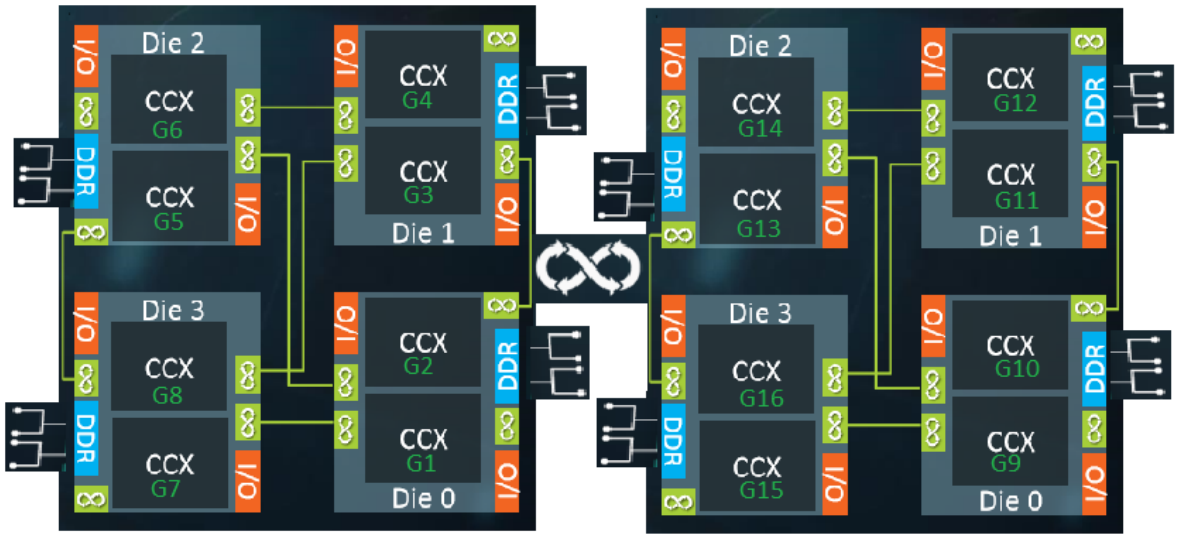

Figure 1. AMD EPYC™ 7601 Processor (2P) showing the groups of SPECjbb® 2015

You can see more details of our experience with SPECjbb 2015 in Appendix B.

Compiler and Kernel Settings

The HotSpot JVM contains two JIT-compilers, namely the C1 (client) compiler and the C2 (server) compiler. C1 and C2 runs within the context of the application (along with the application) and compiles the "hot" methods of the application at various compilation levels. C1 compiles the interpreted methods (after some threshold of "hotness") at the optimization levels, 1, 2, and 3, while C2 compiles the methods at the highest level of optimization of level 4. Level 1 is simple optimization, Level 2 is with minimal profiles, Level 3 collects at full profile and Level 4 performs full optimization using the full profiles (collected at Level 3). This is called "tiered" compilation and research has shown that this helps optimize the runtime of the application significantly.

There are many compiler flags which may be useful for tuning the Java server application. For example, we can compile any method at any levels using the options given. We can also exclude compiling methods if there are issues in compiling those methods. We can also change the amount of inlining, loop optimization, etc., as applied by the compiler using different JVM flags. In general, it is not recommended to change the flow of the compiler (since Hotspot does a good job of compilation), however if we find any opportunities/reasons to give specific hints to the compiler, we should tune the application using the flags provided. For example, for SPECjbb® 2015, we found that inlining a few methods (big methods) at early stage improves the performance of AMD EPYC™ processors.

There are several kernel parameters which are known to work well with Java applications. For example, if the Java application runs large number of threads, it may be useful to avoid too many context switches. The Appendix C gives our experience with such parameters with SPECjbb 2015.

Runtime Settings

One of the tuning factors affecting performance can be number of threads the application runs. If the number of threads the application creates is much more than the hardware threads available, then such an environment may need tuning. Also, there may be different types of threads: producer, consumer, network threads, etc. Depending on how loaded they are, and how performance critical they are, it may be best to tune the number of such threads to get optimal performance from the system. The current Linux® kernels provide all such parameters for tuning.

For SPECjbb® 2015, we performed the following tuning for number of threads and other parameters. A few of them are shown below:

1) Selector Runner Threads: These threads are spawned for servicing the network activity from the application. For max-JOPS (one of the scores in SPECjbb 2015) we set the value as 1 or 2. This number creates the multiple of 6 worker thread for controller, multiple of 5 for BE, and multiple of 5 for TxI. Setting it to a large number severely affects performance of max-jops.
2) Map-Reducer Pool size: These threads perform the job of report generation by map reducing the work with the help of fork join worker thread. Optimal value is 5 for max-JOPS. Decreasing to lower values affects the performance.
3) Fork-join Worker Threads: These are worker threads in the backend instance which service the requests coming from simulated workload. This works in multiple tiers. These are set in three tiers, found Tier1 = 170 Tier2 = 48 and Tier3=28 to be optimal.

Client pool size is for the number of socket connection to be opened for each agent. The worker pool threads binds to each socket and keeps on waiting for the responses from fork join worker Tier1 threads. If the Tier1 threads are not able to respond quickly, the performance is affected. So the problem cannot be resolved by decreasing or increasing the Tier1 threads, optimal number of threads are required. For increasing the Tier1 threads, the heap also required to increase so performance goes down by increasing the memory to more than 31G, as it requires more CPU time to free up the memory.

References

https://www.cubrid.org/blog/the-principles-of-java-application-performance-tuning

http://www.oracle.com/technetwork/articles/javase/turbo-141254.html

https://dzone.com/articles/java-performance-tuning

https://oakbytes.wordpress.com/2012/06/06/linux-scheduler-cfs-and-latency/

Appendix A Garbage Collection

Analysis

Java application performance can be analyzed with GC logs. For analysis any online gc analyzing tool can be used. If the application throughput is as per the expectation, we do not need to tune else few GC flag values are required to change. For SPECjbb® 2015, we tune based on GC logs. The logs can be collected by running the application along with the following flags-

-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -XX:+PrintTenuringDistribution(Options are available in JAVA 8)

Tuning

Heap setting flags *-Xmx -Xms –Xmn* (Maximum heap, Minimum heap, Young gen heap respectively) should be set according to the requirement. For example for SPECjbb® 2015 we the values as *–Xmx28g –Xms28g –Xmn26g*(“g” is for GB).

GC threads are important to set. For e.g. 1Node (Die) – 8C/16T we set the value 16 as the optimal value for GC thread is equal to or slightly higher to the number of available core. Increasing to large value degrades the performance.

Survivor Ratio decides the size of Eden and two survivor spaces so for the value of 28(*SurvivorRatio)* the size of survivor spaces calculated as

One Survivor Space= -Xmn / (28+2).

Max Tenuring Threshold decides the age of object survival in young gen. For example, if an object has this count as 15 then after next young gen GC, the object will move to the old gen. So it remains in the young gen survivor spaces till it reaches to this threshold. Optimal value for survivor threshold for SPECjbb 2015 seem to be 15.

The flags we use for SPECjbb 2015:

-XX:+UseParallelOldGC -XX:SurvivorRatio=28 -XX:TargetSurvivorRatio=95 -XX:ParallelGCThreads=16 -XX:MaxTenuringThreshold=15

Among different available GC’s (Parallel, CMS and G1GC) the best for SPECjbb 2015 is ParallelOldGC. The other GCs do not work well for SPECjbb 2015 since their purposes are different. Garbage collector also should be chosen based on the type of application.

(For further details refer to: *http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html*)

Appendix B Node Level Binding

As explained in the earlier sections, in cases where the application can be split into multiple Java instances, binding each Java process at Node level or CCX level (along with binding the memory channels to those dies/CCX) gives significant performance boost. For example for SPECjbb® 2015, we bind the processes at CCX Level or Node (die) Level.

For NUMA binding, we use the command called "numactl" on Linux® as:

Core level/CCX level binding:

```
numactl –physcpubind=0,1,2,3,4,5,6,7 –membind=0 <JAVA Process>
```

Node Level binding:

```
numactl –cpunodebind=0 –membind=0 <JAVA Process>
```

GCThreads should be changed according to the binding. For example if binding to a specific node then threads should be equal to no of cores in that node. Similarly, for any other parameter which depends on no of available processor, should be changed according to the binding.

Memory should not be allocated more than the available on particular node.

(Use numactl –H command to check the available memory on particular node)

Initiating Your Java Process by Binding to Nodes

The **numactl -H** command gives the NUMA node structure of the system (for more information, see *https://linux.die.net/man/8/numactl*).

1) --cpunodebind: This option binds to the particular node.
2) --physcpubind: This binds to physical core. So use this if the binding is according to cores. Check the core numbering before binding.
3) --interleave=all: This option sets the memory to be used across all NUMA nodes, so the allocation will be done on different nodes the system finds optimal to allocate.
4) –membind=0: This option binds the memory to particular node. So binding the process to specific node where the memory allocation is restricted to happen on that node.
5) –localalloc: To allocate the memory locally according to the cpunodebind.

Appendix C Kernel Parameters

As explained in the earlier section, a few kernel parameters (which affect the thread scheduling, page swapping, etc.) are useful for some Java server processes running for longer time.

Scheduler Tunable:
On Linux® systems **CFS scheduler** comes with different parameter which can be tuned according to the type of application. Few parameters we change for SPECjbb® 2015 are:

a) **sched_migration_cost_ns**: It determines how long a migrated process has to be running before the kernel will consider migrating it again to another core. Default value 50000 (that's ns so 0.5 ms), we reduce it to 1000 to have high migration.
b) **sched_rt_runtime_us**: A global limit on how much time real-time scheduling may use. We increase the value to 990000, so this gives 0.01s out of total period of 1s to be used by other non-RT tasks.
c) **sched_latency_ns**: Period over which CFQ tries to fairly schedule the tasks on the run queue. All of the tasks on the run queue are guaranteed to be scheduled once within this period. By default, this is set to 20ms. For SPECjbb 2015 we set it to 24ms.
d) **sched_min_granularity_ns**: This decides the minimum time a task will be allowed to run on CPU before being pre-empted out. By default, it is set to 4ms. So by default, any task will run at least 4ms before getting pre-empted out. We found the optimal value for SPECjbb 2015 is 10ms.

Other Tunable:

a) **dirty_background_ratio**: Contains, as a percentage of total available memory that contains free pages and reclaimable pages, the number of pages at which the background kernel flusher threads will start writing out dirty data. For SPECjbb 2015 we set it to 10.
b) **dirty_ratio**: Contains, as a percentage of total available memory that contains free pages and reclaimable pages, the number of pages at which a process which is generating disk writes will itself start writing out dirty data. For SPECjbb2015 we set it to 40.
c) **dirty_writeback_centisecs**: The pdflush writeback daemons will periodically wake up and write "old" data out to disk. This tunable expresses the interval between those wakeups, in 100’Th of a second. Setting this to zero disables periodic write back altogether. For SPECjbb2015 we set it to 1500.
d) **dirty_expire_centisecs**: This tunable is used to define when dirty data is old enough to be eligible for writeout by the pdflush daemons. It is expressed in 100’Th of a second. Data which has been dirty in memory for longer than this interval will be written out next time when a pdflush daemon wakes up. For SPECjbb 2015 we set it to 10000
e) **swappiness**: Sets the kernel's balance between reclaiming pages from the page cache and swapping process memory. The default value is 60.For SPECjbb 2015 we set it to 10.

These kernel parameter setting comes from some level of experimentation and knowledge of Linux kernel.