



Java Memory Management - Internals and performance -- Webcast topic transcript

Hello everyone, and to all participants who are attending remotely today, thank you for joining us, and welcome.

Our topic for this session is Java Memory Management internals and performance, The main goal of today's presentation is to explain how the different GC policies on HP-UX actually work, this is a fundamental first step for anyone trying to analyze and improve the performance of a Java application, while this presentation does not primarily focus on the tools HP provides for analyzing GC related performance problems, the level of understanding you will gain here today is an essential pre-cursor to using any GC analysis tools in the future, you must have a good understanding of the internals of GC in order to interpret the data presented by the tools and take appropriate action toward tuning your application.

[NEXT SLIDE]

So, we will start with a simple introduction and overview of memory management for Java, we will then take a closer look at generational garbage collectors to make sure we understand how they work and what exactly happens when your application goes through a garbage collection cycle, after that, we will cover the details of all supported GC policies in the HP Java Virtual Machine, we will also explain the defaults for each GC policy, this will help you understand which part of the heap you need to tune for your application, and we will end today's talk with a review of fundamental guidelines for ensuring good memory management performance for your application.

[NEXT SLIDE]

Let's start with an overview of memory management for applications written in Java. The common term for Java Memory Management is Garbage Collection, you will hear me use both terms during today's presentation. The main premise behind memory management for Java is that it is handled automatically by the JVM, unlike C or C++ where developers need to explicitly manage the allocation and freeing of memory with malloc/free or new/delete, Java developers don't have to do anything special, the memory requirements for their application are managed by the Garbage Collector in the JVM.

There are many types of garbage collectors out there, each have their own advantages and disadvantages, for example Reference Counting garbage collectors which rely on maintaining a reference counter stored in the header of each object, the counter is incremented/decremented every time an object is referenced/de-referenced, objects with a zero reference count are considered dead and are garbage collected, while this is a simple and reasonably fast way of managing memory, one immediately realizes it does not handle space reclamation very well for example, it does not do compaction.

Another class of garbage collectors is known as Mark Sweep collectors, unlike reference counting collectors, mark sweep collectors periodically bring a running application to a complete stop, quickly scan all objects to identify and mark the ones that are no longer reachable, then do a sweep to reclaim their space.

A third class of collectors is the copying mark sweep compact collectors, unlike mark sweep only collectors which manage the application's memory in one dedicated space, copying collectors divide the space where objects are allocated into two distinct areas, where one of these two areas operates as a regular mark sweep collector, but instead of only identifying and removing dead objects, it also copies all live or surviving objects into the second space, the sequential copying of live objects into the second space, has the nice effect of removing fragmentation and compacting memory. After copying is done, the application is resumed and new object allocation is directed to take place in the compacted space. The collector alternates copying surviving objects between the two different spaces.

Both the mark sweep and mark sweep compact collectors are what we call stop-the-world collectors, they need to stop the running application before doing marking and sweeping,

a fourth class of garbage collectors is the Concurrent Mark Sweep collectors, these strive to reduce the length of time an application is interrupted for the purpose of doing garbage collection, they are known as Concurrent collectors, however in reality they are mostly concurrent since they too need to stop or pause the running application, however the length of the pause is very short.

The garbage collector you get with the HP JVM is a generational copying collector, at its core it is a mark sweep compact collector, but unlike traditional mark sweep compact collectors it divides the Java heap into more than two spaces.

[NEXT SLIDE]

So let's take a closer look at generational garbage collectors and understand how exactly they work

[NEXT SLIDE]

Our generational garbage collector divides the Java heap into five distinct spaces or generations, we have a new/young generation which is further divided into an eden space, and two survivor spaces known as the TO and FROM spaces, an old space, and a separate permanent space.

We divide the heap this way because we know from examining a large sample of applications that most objects are short lived, and since the cost of doing a garbage collection cycle is directly proportional to the number of objects which need to be examined, in other words, the larger the number of objects we need to mark and sweep the higher the cost, we aim at optimizing the cost of garbage collection by speeding up the common case, namely speeding up the cost of marking and sweeping newly allocated objects in a small part of the Java heap (in the new/young generation).

The overall size of the heap is controlled by two JVM flags or command-line options (-Xms and -Xmx), -Xms controls the initial heap size, and -Xmx controls the maximum size of the heap as we show here. When these two JVM flags are not set to the same value, the JVM will initialize the heap based on the value set by -Xms, then periodically expands the heap if needed until the maximum size is reached.

The size of the new/young generation can also be set on the command-line by specifying -Xmn, and, the

permanent space size can be set by specifying `-XX:PermSize` on the command-line.

The new/young generation is where all newly created objects reside, objects that survive collection (i.e long lived objects) are copied into the old generation. The permanent space is set aside for holding information about the application's classes and these are used directly by the JVM.

[NEXT SLIDE]

As we mentioned already, the To and FROM spaces are known as the Survivor spaces, their size is set as a ratio to the size of the new generation, the java command-line flag `-XX:SurvivorRatio` sets the size of one survivor space compared to the size of eden.

One more basic command-line to keep in mind is `-XX:MaxTenuringThreshold`, the value associated with this command-line flag specifies the age of a surviving object in term of the number of times it survives a minor collection.

All of this will become clear as we step through the inner workings of generational garbage collectors in the next set of slides.

[NEXT SLIDE]

Based on our description of the heap so far, it should become apparent that we have two types of garbage collections with two very different costs, a minor collection corresponding to the new/young generation becoming full, and a major collection which is triggered when the entire heap becomes full.

[NEXT SLIDE]

OK, now that we understand how the generational garbage collector divides the heap, let's take a closer look at how it actually works.

As we already mentioned, all new objects are initially allocated in the new generation, more specifically, all new objects are allocated directly into the eden portion of the new generation. In the following view of the Java heap, we show several "red" objects allocated in eden as a result of running an application for a certain length of time. And for the purpose of this example, let's assume that we have set our tenuring threshold to a low value of only 2.

So our application has been running for a while, and it has been creating new objects all of which are initially placed into eden, what happens when eden becomes full?

[NEXT SLIDE]

When eden becomes full, the JVM pauses the application (this is what we also called stop-the-world when we were describing the different types of garbage collectors)

After pausing the Java application, the JVM iterates over all objects in the eden and marks the ones that are live (basically objects which are referenced from other objects in the application), the live objects in our view of the heap here are indicated with an arrow pointing to the.

Now, What happens after the JVM marks all live objects?

[NEXT SLIDE]

All live objects are then copied to one of the two survivor spaces.

[NEXT SLIDE]

The objects remaining in eden after live objects are copied to one survivor space are the ones which we consider to be dead, these are the objects we need to garbage collect

[NEXT SLIDE]

Dead objects which were in eden are garbage collected or scavenged with what we call a minor collection which only operates on a small part of the heap, namely the eden.

Live objects which survived the scavenge (the objects copied into one of the survivor spaces) will now be assigned an age of 1, this age represents their tenuring threshold

Once a scavenge is completed, the JVM resumes the halted application, and more objects are newly created in the new/young generation

[NEXT SLIDE]

These new objects shown here as yellow blocks, eventually fill-up the eden portion of the new generation, what happens next?

The JVM pauses the application again and prepares for another minor collection (or scavenge)

[NEXT SLIDE]

All live objects in eden "AND" the first survivor space are marked, then copied over to the second survivor space and their age or tenuring threshold is updated accordingly

[NEXT SLIDE]

Note how after a scavenge or a minor collection, we always end up with eden and one survivor space completely empty

Also note that one of the objects at this point has survived two minor collections, and hence reached the tenuring threshold, which means if this object remains live after the third minor collection, it will have to be promoted into the old generation.

OK, so our second scavenge or minor collection is done, the JVM resumes the application and more new objects are placed into eden again

[NEXT SLIDE]

These new objects are shown here as light blue blocks in eden

[NEXT SLIDE]

Once eden fills up, the JVM pauses the application for a third time, and all live objects are identified and marked, then all objects which already reached their tenuring threshold are first copied into the old space

[NEXT SLIDE]

What we show here is that one object which already survived three scavenges or minor collections is now placed in the old space, from here on, this object will be considered long living and it will not be scanned or copied

around when the next scavenge kicks-in

[NEXT SLIDE]

After copying the long living object into the old space, all live objects in eden and one survivor space are identified and copied over into the second empty survivor space.

[NEXT SLIDE]

And just as before, the age of surviving objects is updated

[NEXT SLIDE]

And a scavenge removes all remaining dead objects from eden and one of the two survivor spaces

The JVM then resumes the application, and more objects are allocated into the new/young generation

[NEXT SLIDE]

After running for a certain length of time, the old generation becomes almost full, and eden becomes full, when this happens, the JVM halts the application one more time, and starts a major garbage collection cycle (or a Full GC), during a major collection cycle, all objects in the heap (the ones in the new/young space as well as the ones in the old space) are scanned and marked

[NEXT SLIDE]

The garbage collector then removes all dead objects, and compacts the old space as shown here.

Major collections or Full GCs are very expensive compared to minor collections, mainly because they have to process all objects in the heap, and because they have the added overhead of old space compaction

So, we now have a good understanding of how generation mark-sweep-compact garbage collectors work, but before moving on to look into the internals of all supported GC policies in our JVM, let's look into one corner case and clarify how the JVM internally accounts for it.

[NEXT SLIDE]

In this corner case, after the new generation fills-up, we could get into a situation where all objects in eden and one survivor space are live

A normal scavenge would try to copy all live objects into the second survivor space, as we show here, clearly this is not possible in this case

In order to handle this type of corner cases, the JVM by default sets aside a portion of the old space equal in size to eden plus one survivor space, for the most part this space is wasted, mainly because it is rare that all objects in eden and one survivor space remain live after a minor collection.

If you know your application does not exhibit this condition, you can reduce or completely eliminate the amount of space set aside in the old generation for handling this case, this can be done by specifying `-XX:MaxLiveObjectEvacuationRatio` which allows you to specify a percentage of the old generation size to set aside, or by specifying `-Xoptgc` which essentially sets no space at all in the old generation for handling this case.

Please keep in mind that maximizing the size of the old generation has the benefit of delaying or reducing the number of expensive Full garbage collections

[NEXT SLIDE]

Let's now take a look at the internals of supported GC policies on HP-UX all of which are to a large extent implemented as generational mark-sweep-compact garbage collectors, and for the most part they all work the same way we already explained. There are some differences in the way the Concurrent Mark Sweep policy works, and we will point these differences out during our examination of CMS.

[NEXT SLIDE]

The java virtual machine on HP-UX supports four different garbage collection policies, a single threaded serial GC policy, a high-throughput parallel new generation GC policy, a high throughput parallel old and new generation policy, and a low pause, mostly concurrent mark sweep GC policy.

Having different policies reflects the fact that it is not suitable to have one garbage collection policy which would perform well for all types of applications.

Let's take a look at how each of these policies is implemented

[NEXT SLIDE]

The Serial GC policy can be activated by specifying `-XX:+UseSerialGC` on the Java command-line, prior to JDK 1.4.2 this used to be the default GC policy on our HP-PA and Integrity platforms

As we show here, both minor and major collections are single threaded with this policy

[NEXT SLIDE]

The second GC policy is the parallel new generation GC policy, this was introduced in the later versions of JDK 1.4.1, and has become the default GC policy on servers with 2 or more CPUs starting with JDK 1.4.2.

Users can explicitly activate this GC policy by specifying `-XX:+UseParNewGC` on the java command-line.

With the throughput parallel new generation GC policy Scavenges (or minor collections of the new generation) are done in parallel. Collection of the old generation (when a major or a Full GC kicks-in) remains single threaded similar to the way it happens with the Serial GC policy.

[NEXT SLIDE]

Specifying `-XX:+UseParallelOldGC` on the java command-line activates a high throughput parallel collector where both scavenges (or minor collections) as well as old generation major collections are done in parallel on machines with 2 or more cores. We will explain how to control the number of parallel GC threads when we talk about policy defaults in the next section.

[NEXT SLIDE]

Specifying `-XX:+UseConcMarkSweepGC` on the java command-line activates the low pause concurrent mark sweep garbage collection policy in our JVM, this policy employs a multi threaded young generation collector and a low pause mostly concurrent old generation collector, the old generation collector is mostly concurrent because it contains two extremely short stop-the-world type pauses as we show here.

The concurrent phase in this policy is a bit involved and it consists of 4 major phases.

[NEXT SLIDE]

Phase 1 as we show here pauses the running application for a short period of time and invokes a single thread to do an initial mark, the initial mark phase identifies a subset of all live objects

When looking at the output of verbosegc with our tools, the pause in this phase can be identified as STW 1 which stands for Stop The World 1.

[NEXT SLIDE]

After the initial subset of live objects has been identified, the JVM resumes the application and invokes a concurrent thread to identify all live objects reachable from the set of live objects identified in the initial mark phase. This phase also does pre-cleaning which is essentially a proactive scan of all objects updated during the concurrent mark phase. Pre-cleaning is intended to shorten the pause which is to follow in phase 3

[NEXT SLIDE]

In phase 3, the JVM pauses the application one more time and invokes multiple threads to do a quick parallel re-mark, the purpose of the re-mark is to identify as many of the live objects missed during the concurrent mark phase (keep in mind that the application is still running and creating more objects during the concurrent mark phase).

when looking at the verbosegc data with our tools, this phase can be identified as STW 2 (Stop The World 2)

[NEXT SLIDE]

After the parallel re-mark phase, the JVM resumes the application for the second time and invokes a single concurrent thread to sweep all dead objects.

[NEXT SLIDE]

Our current tools don't have a single view that shows how the different CMS stop-the-world phases correlate to heap usage, in order to do this, I usually super-impose the heap usage view on top of the duration view as shown here, visually lining up the time axis from both views allows you to get a good view of what is happening in the java heap during each phase.

[NEXT SLIDE]

For those of you who have started using JDK 6, you should be aware of two new CMS policy features. Beginning with JDK 6, users will be able to specify multiple concurrent sweep threads by adding -XX:ParallelCMSThreads on the java command-line. This could result in speeding up the concurrent mark sweep garbage collector considerably when running on multi-core servers.

Also in JDK 6, the default handling of explicit GC (these are calls to System.gc()) from within the application for example) can now be overridden to invoke a low-pause CMS collection instead of an expensive Full-blown garbage collection with expensive old space compaction, this new feature can be activated by adding -XX:+ExplicitGCInvokesConcurrent on the java command-line

[NEXT SLIDE]

One other change in JDK 6 directly affecting the CMS collector has to do with how tenuring and the survivor spaces are handled, prior to JDK 6, specifying the CMS GC policy essentially resulted in disabling the survivor spaces (basically when CMS is specified, the survivor ratio was set to 1024 which effectively disables the survivor spaces by making them extremely small), this had the effect of tenuring objects pre-maturely and increasing the pressure on the old generation. Beginning with JDK 6, the survivor spaces are re-activated

allowing users to tune them if necessary.

[NEXT SLIDE]

Before concluding our discussion on Concurrent Mark Sweep we need to explain how the “young generation guarantee” is handled when CMS is specified. If you recall, with any of the non concurrent collectors, not having enough space set aside in the old generation for handling the situation where all objects in the new generation happen to survive a minor collection leads to the JVM terminating with an out of memory error, with CMS however, when this happens, the garbage collector will try a last ditch minor collection first to see if some objects might have died since the promotion was attempted, if the last ditch collection does not result in freeing up enough space in the old generation, the garbage collector will then do a heavy, full-blown collection of the old generation followed by compaction, and if that also fails to free up enough space, then the JVM will terminate with an out of memory error.

[NEXT SLIDE]

Next we'll take a closer look at the defaults for each GC policy to better understand how the garbage collector would behave if we don't explicitly change any to them.

[NEXT SLIDE]

If no GC parameters are specified on the command-line, the initial java heap size will be automatically set to one sixty fourth of the total available physical memory on the box but not to exceed 1 GB (so it would be capped at 1 GB), similarly, the maximum java heap size will be automatically set to one fourth of the total available physical memory on the box, and that would also be capped at 1 GB.

When running on machines with more than 2 cores (or processors), the parallel new GC policy will be selected by default.

Also by default, the JVM will enable adaptive size policy, which effectively tries to vary the size of eden and survivor spaces as well as tenuring thresholds according to some internal JVM heuristics.

[NEXT SLIDE]

Next we'll try to understand how does the JVM set the number of parallel GC threads when one of the throughput collectors is activated,

if the server on which Java is deployed has 8 or less cores (or processors), then the JVM will initialize as many parallel GC threads as there are cores on the box. For cases where Java is started on servers with more than 8 cores, the number of parallel GC threads is set by the JVM according the formula shown here.

[NEXT SLIDE]

The JVM uses the same mechanism to determine the number of parallel GC threads for the Low-pause CMS collector.

[NEXT SLIDE]

With the low-pause CMS collector, the JVM defaults to automatically sizing the new generation spaces, and as we already mentioned, with JDK 5.0 and prior, the JVM will strive to promote objects prematurely in an attempt to shorten pauses caused by minor collections.

The automatic settings of the JVM will be disturbed if users specify explicit values for the initial or maximum heap size, or if they explicitly specify a survivor ratio or a tenuring threshold.

[NEXT SLIDE]

Now that we have a good grasp of how generational garbage collectors work, and a good understanding of the different GC policies and how their defaults are set, we will spend some time talking about garbage collection performance:

when should you use one GC policy and not the other?

what are the general guidelines that ensure good GC performance for your application?

and we will also briefly talk about the Java performance analysis tools available to you for free on HP-UX.

Coupled with your new deeper understanding of the inner workings of our garbage collector, you will now have a renewed appreciation for the tools and the type of insight they provide into where the performance bottlenecks are in your application. I encourage you to enable JVM profiling and use the tools to look into both, your application and the Java Virtual Machine and get a big picture of where the problems are and decide how to resolve them.

[NEXT SLIDE]

We start with general guidelines related to correctly sizing the java heap. Your aim here should be to minimize the number of costly Full garbage collections, make sure short lived objects do not get tenured pre-maturely, so keep the tenuring threshold high, and make sure your new generation is sized correctly,

a new generation that is too large could result in long pauses due to abnormally long minor collections, and a new generation that is too small could result in filling up the old space too fast leading to frequent costly Full garbage collections.

[NEXT SLIDE]

Also make sure you avoid unnecessary costly Full Garbage collections caused by explicit calls to `System.gc()` or `Runtime.gc()` from within your application, you can specify `-XX:+DisableExplicitGC` on the java command-line to instruct the JVM to ignore calls to these two explicit GC routines.

Again, use the tools to see if your application is calling these routines.

And if your application uses RMI, make sure to set the GC interval for both the server and the client to the highest possible value as shown here, failing to do so will cause an expensive Full garbage collection every 60 seconds.

[NEXT SLIDE]

Use the right GC policy according to the type of application at hand. If you have a small application running on a small box, then you might want to use the single threaded serial garbage collector. If on the other hand you have a large throughput-oriented application running on a system with more than 2 processors, then use one of our parallel throughput collectors, and if your application is extremely sensitive to GC pauses, then you might want to use the concurrent mark sweep collector.

When running with Montecito Hyper Threading enabled which is now available on 11i v3, make sure to manually set the number of parallel GC threads equal to the number of physical cores on the box, also make sure to explicitly disable binding of GC task threads to processors by adding `-XX:-BindGCTaskThreadsToCPUs` on the Java command-line.

[NEXT SLIDE]

If you are not sure what your application is doing

or if you are not sure which value to tune next,

it is time to download and run our performance analysis tools.

[NEXT SLIDE]

The tools will provide you with instant answers to basic questions.

For example:

What are the various threads in the application doing?

Which threads are running slow?

Where is each thread spending the most time?

Is the problem you are seeing caused by the garbage collector or by something else?

[NEXT SLIDE]

Once you have identified the problem at the high level, the tools will enable you to dig deep into the application and pin-point the exact source of the slowdown.

[NEXT SLIDE]

This brings us to the end of our presentation today, but before we conclude, let's summarize the key points:

Key point number 1- Make sure you have a through understanding of how each of our garbage collection policies work

Key point number 2- Understand each policy's defaults

Key point number 3- Use an appropriate GC policy for the type of workload at hand

Key point number 4- Follow the general tuning guidelines

and when your application does not perform well, use the tools to quickly identify and resolve any performance related problems.

This concludes today's presentation.

For more information:

www.hp.com/go/knowledgeondemand