Revision 1.2
Randy Rorden
Software Development Director
PTC, Inc.

Who Stopped the World? A Tale of Java Garbage Collector Delays

Java Garbage Collection

One of the best features of the Java[1] Virtual Machine is automatic garbage collection. For the most part, Java programmers don't have to worry about managing memory for objects created by their applications. When the application no longer has any references to an object, the JVM automatically deletes it and reclaims the memory it occupied for future allocations. Compare this to C and C++ applications, which must keep track of memory allocated with malloc or new and return it to the system explicitly using free or delete. Failing to de-allocate memory can cause the application's memory footprint to increase over time (a memory leak), resulting in a crash or the process getting killed by the operating system. The worst kind of memory leak is a small one because it adds just a few bytes at a time, often under circumstances that are not easily reproduced or noticed during testing. The worst place for a memory leak is in an embedded system, where the application may run for years without restarting, slowly absorbing a critical resource, until it fails.

How a Java GC Works

There are many different Garbage Collector (GC) algorithms in use today by Java VMs as well as other language runtimes such as Python, Go, .NET, and Ruby. The Java 11 VM offers a choice of four different GC algorithms, but they all perform the same basic tasks:

- Identify unused (dead) objects in memory
- Reclaim the memory occupied by the dead objects
- Defragment memory by moving live objects

The most common technique used to identify unused objects and reclaim memory is some form of the Mark and Sweep algorithm. Starting from a set of "root" pointers to known-live objects in heap memory, such as a list of active threads and their local variables, the garbage collector scans each object for reference pointers to other objects and marks the referents as live. Then it scans each of those live objects, and so on, until all live objects have been reached, scanned and marked. Encountering a previously marked object means it has already been scanned and no further work on it is required. After scanning, the garbage collector "sweeps" sequentially through heap memory looking for unmarked objects, reclaiming the memory they occupied.

The last task, defragmenting memory, is typically done by moving live objects to new locations so they are contiguous to one another, leaving larger free blocks available for new allocations. It is during object relocation when most delays occur. Imagine you are the garbage collector. You've found and reclaimed all the dead objects in the heap, but now there are lots of "holes" in the memory between live objects. If you just left things alone, the heap would become fragmented over time. Eventually a new object allocation will fail because it can't find a large enough free block even though the total free space is more than enough. You must move some of the live objects together in order to coalesce free space. But there's a catch. If you move an object, what about all the other objects

[1] Oracle and Java are registered trademarks of Oracle Corporation

that reference it? Those pointers need to be updated to the new location. You will have to go back and fix those stale references. What if a Java thread happens to use a stale reference while you are moving objects and fixing pointers? Boom. Crash. Traditional Java garbage collectors "stop the world" during critical phases by pausing all Java threads while objects in heap memory are being modified.

Tricks of the Trade

There are some tricks that can be used to minimize pause times. One is to move fewer objects at a time, letting Java threads run between moves. Of course, if objects are allocated and de-allocated rapidly, fragmentation may overwhelm the system and the garbage collector must run longer with Java threads paused to catch up.

Another trick is to segment the heap into generations, where objects are first allocated in a young generation area and migrated to an old generation area after they survive one or more garbage collection passes. Since most Java objects are short lived, it pays to collect from the young generation more frequently. The JVM or the user must decide how big the young generation should be for optimal performance. Figure 1 shows how a generational heap is organized.

Today three of the four garbage collectors in Oracle Java and OpenJDK are generational collectors, although the underlying structures in which the generations are stored may differ. Collecting dead objects from the young generation is called a minor collection, while collecting from the old generation is called a major collection. Minor collections are more frequent. Heuristics monitor heap usage and fragmentation to determine when a major collection is required.

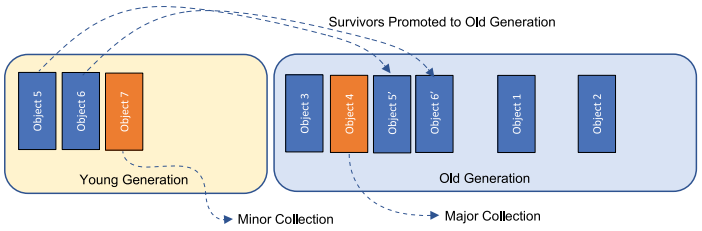

Fig. 1

One more trick adds load and/or store barriers. A barrier is a snippet of code that is executed by Java programs on the way to reading (a load) or writing (a store) through a reference to an object in heap. The barrier examines the reference to see if the referent is waiting to be moved by the garbage collector. If so, the barrier may perform the object move itself and return the updated pointer. This can help reduce pauses by allowing Java threads and garbage collector threads to run concurrently more often. A barrier may also help mark live objects. The actions of the barrier will depend on the phase of the garbage collector. Barriers reduce the performance of Java applications by adding more instructions to read or write operations, so they must be designed to be as fast and efficient as possible.

Traditional Java Garbage Collectors

As mentioned above, the Java 11 Virtual Machine has four garbage collection algorithms from which to choose. One is experimental and a fifth is present but deprecated. Each algorithm is optimized for different hardware features (i.e. single vs. multicore), heap sizes, application behavior, and sensitivity to delay.

- The Serial Collector is the simplest and most efficient. It runs in a single garbage collector thread and all Java threads are paused while it performs a collection, either on the young or old generation. It is a compacting collector - it compacts live objects at the front of the old generation area and moves them from a young "Eden" area to a young "Survivor" area and from there to the old generation. Compacting allows faster allocation of new objects from a contiguous free block. The Serial Collector is recommended when CPU resources are limited, and applications aren't sensitive to delays.

- The Parallel Collector runs multiple garbage collector threads coordinating with each other to speed up collection of the young generation. By default, the old generation is collected using a single thread, but an option is available to do parallel collection there too. All collections pause all Java threads, but minor collections go faster using multiple threads and major collections are expected to be less frequent. The Parallel Collector is recommended when there are plenty of available CPU resources. It is the default collector in Java 8.

- The G1 or Garbage First Collector divides heap into equal sized regions, some of which are designated for the young generation (Eden and Survivor) and others for the old generation. It uses multiple threads like the Parallel Collector, but they can do much of their work concurrently with Java threads. To accomplish this feat, the G1 Collector introduces a store barrier. When Java threads write to an object, the action may be recorded to help the garbage collector identify references between regions. The G1 Collector focuses on meeting a pause time target. If it completes a minor collection of the young regions within the pause time, it may proceed to collect one or more old regions as well. Live objects in collected old regions are compacted to contiguous space in other old regions. This provides shorter pauses than the Parallel Collector, which must collect and compact the entire old generation for each major collection. G1 is better for very large heaps. Figure 2 shows a G1 region map.

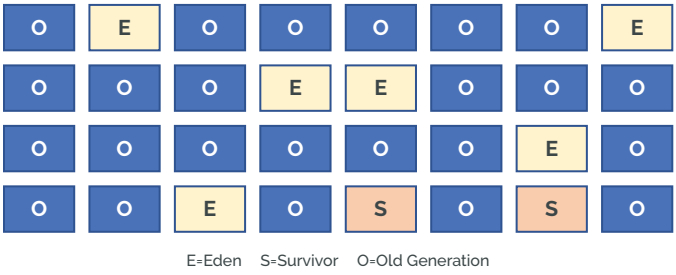

Fig. 2

- The Z Garbage Collector (ZGC) is experimental in Java 11 through 14. It has a single generation for all objects and has three region sizes (2MB, 32MB, and 2MB x n variable). It uses a load barrier instead of a store barrier for Java threads to help the collector keep track of referenced objects. ZGC keeps some extra status bits in each reference. It can do this by using full 64-bit addresses into virtual memory and limiting the maximum heap size to 4 TB of memory (42 bits). Four of the remaining bits, called metadata bits, track references to finalizable objects, track if the reference has been remapped to a new object location, and track reachable objects with a pair of mark bits. If the reference hasn't been remapped and the object is in a region being relocated, the load barrier itself will move the

object and set the remap bit. Requiring 64-bit references means ZGC is only available on 64-bit systems. Note that ZGC has some stop-the-world pauses, such as to scan local variables for threads, so large numbers of threads can increase pause times.

There are vast numbers of technical articles, how-tos, YouTube videos, and conference presentations dedicated to selecting and tuning Java garbage collectors. A google search for "tuning Java garbage collection" yields over 850,000 results. You can spend a career learning how to optimize garbage collection for your application. Most developers and IT professionals pick settings that someone else recommended and if they experience excessive delays they often "throw memory" at the problem, increasing heap memory 2x or more to see if that helps. Unfortunately, sometimes there isn't that much memory to spare, as in the case of embedded systems.

A Real-Time Garbage Collector

The first Java Virtual Machine was publicly released by Sun Microsystems in 1996. That same year, an Iowa State University professor, Dr. Kelvin Nilsen, founded NewMonics, Inc. and began developing a clean-room JVM based on his research in real-time garbage collection. That work resulted in the Perc[2] Virtual Machine. Perc has been used by customers worldwide since 1998. Since then, new releases have improved performance, supported new Java versions and operating systems, new compilers, and new processors. Today, PTC Perc supports Java SE 8 on Linux for both 32 and 64-bit Intel x86 and ARM multicore processors. While much has changed in the implementation of the Perc garbage collector, the basic design continues to follow Dr. Nilsen's original patent.

The Perc garbage collector is region-based, like Java's G1 and ZGC collectors. Region size is selectable by the user (default 1MB). Objects larger than a region use Special regions that are a multiple of the region size. The Perc collector is not generational, but it does maintain a set of static regions to hold objects larger than 256 KB as well as long-lived internal objects for Java classes, methods, and Just-in-Time (JIT) compiled code. Unlike traditional Java, JIT code is kept in the heap rather than in a separate code cache. Static regions are garbage collected, but they are not defragmented. The remaining "Normal" regions are garbage collected and defragmented. Figure 3 shows a region map for Perc.

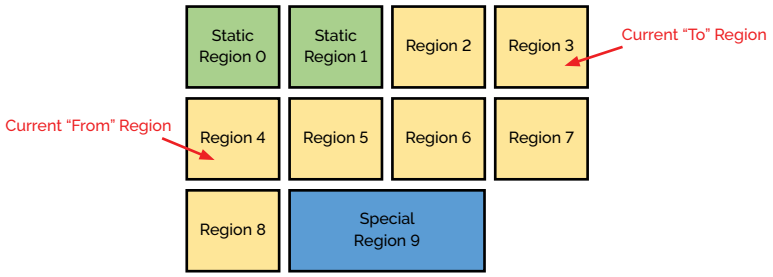

Fig. 3

In addition to the three region types, at the beginning of each cycle the Perc garbage collector selects a current "From" and "To" region from among the Normal regions based on the fragmentation and free space of each region. These will become the source and target of the copy phase of the collector in that cycle. All objects in the "From" region will be copied to contiguous space in the "To" region, leaving the "From"

[2] Perc is a registered trademark of PTC Inc.

region empty. It often becomes the "To" region of the next cycle. In this way, at least one empty region will always be available as a target for copying.

Most of the Perc collector's work is done by parallel threads running concurrently with Java threads. It accomplishes this using an "indirect pointer" kept with each object. During object scanning, when the collector finds an object in the current "From" region, it reserves space in the "To" region for the copy phase. A barrier allows Java threads to do the same when they encounter an object that will be copied. The collector and threads update references to the object to point to the new reserved location and the indirect pointer in the reserved location points back to the original object so that any reads or writes to the object continue to be made to the original. Once all scanning and updating of references is complete, the object contents are copied, and the indirect pointer is "flipped" to the new location.

What is unique about the Perc collector is that even during a phase when it is unable to operate concurrently with Java threads, it can be preempted by a higher priority Java thread and then continue operation where it left off. This isn't the case for traditional Java. This is due to another real-time feature of Perc, namely explicitly controlled thread scheduling. When a traditional JVM starts a new thread, it runs under the scheduling control of the operating system, not the JVM. The OS decides when and on what CPU core it should run. If 10 running Java threads are sharing 4 cores, the JVM has no say about who runs next, regardless of Java thread priority. There are some 'java -XX' options that offer Linux "nice" level assignments based on Java priority, but these require root privilege and the OS can choose to override nice levels anyway. What this means is that in traditional Java, thread priorities are essentially meaningless.

In Perc, all Java thread scheduling is controlled by the Perc VM Dispatcher. Only the N highest priority ready-to-run threads will be allowed to run on N CPU cores. Threads at the same priority will be scheduled "round robin" to receive equal amounts of CPU time. The Perc garbage collector threads have a Java priority and they participate in the scheduling system alongside Java threads. The Perc VM can preempt a thread at any time to schedule a higher priority ready thread. See Figure 4.

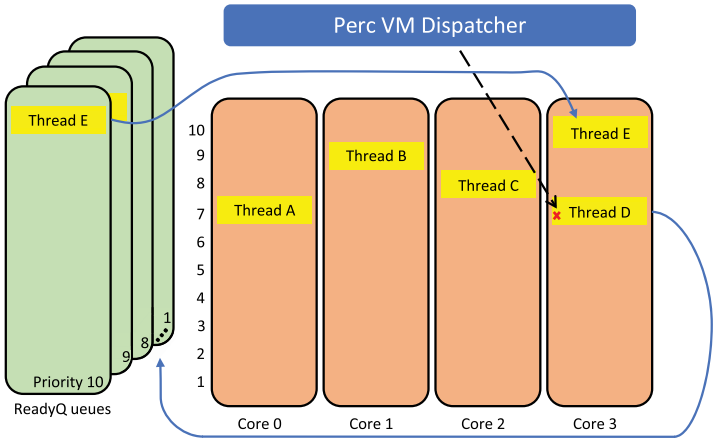

Fig. 4

By default, one collector thread is assigned to each available core. During a phase when only collector threads must run, all lower priority Java threads are preempted until the phase completes. If a higher priority thread needs to run, all garbage collector threads are preempted until the high priority Java thread finishes its task. Then the collector threads continue. Collector threads maintain state in queues, allowing a different thread to pick up the work left behind during preemption.

To take advantage of Perc's scheduling system, developers need to carefully select priorities for their application threads and decide what priority to assign to the garbage collector threads. If a task can accommodate some delay, it should run below the collector. If it must respond to a timer or external event in real time, it should be assigned a priority above the collector.

In addition to being able to set the priority of the Perc collector threads, the user can specify how much CPU time should be dedicated to background garbage collection. This is done with two values: a GC period and a GC timeslice. The period specifies the number of ticks over which to measure GC CPU usage and the timeslice specifies the number of ticks within each period to run the collector threads. A tick is the resolution of all timing functions in Perc (default 1 millisecond). By default, the GC period is set to the number of ticks in 250 milliseconds and the timeslice is set to one tenth of those ticks, for a target of 10% CPU usage during normal operation. In low memory conditions, it may use more. The user can change the period and timeslice on the Perc VM command line. There is also a GC threshold value that specifies the percentage of the heap that must be full before the collector is enabled (default 50%).

Watching Java Garbage Collection in Action

GC algorithms are wonderful things to talk about, but at some point, you have to ask how they really perform. To answer that question, we wrote a "GCStress" program, which is available for free and can be accessed by sending an email to: developer-tools-support@ptc.com with subject Line "Request GCStress". You will receive a reply containing a secure download link to the "gcstressmaster.zip" archive.

The archive has a README that explains how to install it on a Linux/x86_64 host, and run it under Oracle Java or OpenJDK. You may also request an evaluation version of PTC Perc to run with GCStress by sending an email to developer-tools-sales@ptc.com. As explained in the README, GCStress runs two threads:

- A Timer thread runs at the highest Java priority and contains a loop that samples System.nanoTime(), sleeps for 100 milliseconds, and samples nanoTime() again. The actual sleep time is compared to the expected time to get a positive (late) or negative (early) delay. The delay and the current heap usage are recorded and the loop repeats for 300 iterations (30 seconds).
- A Hammer thread runs at normal Java priority and creates a cache using a LinkedHashMap with a capacity of 2,000,000 objects, keyed by an Integer value. Then it runs a loop, generating a random key, checking the cache for an entry for that key, and if already present, it removes the entry. Otherwise, a byte array of random size from 0 to 255 bytes is created and added to the cache for that key. The purpose of the Hammer thread is to cause fragmentation of the heap, forcing the garbage collector to collect many dead objects and move many live objects. The Hammer loop continues until the Timer thread is done.

At the end of 300 samples, the recorded data is written to a CSV file and a summary printed. A separate GCDelayGraph program converts the CSV file into a JPG chart for visualization of all 300 delay samples in blue and heap usage samples in red.

Figure 5 is a chart for OpenJDK 8 running with a 400 MB maximum heap (you need to limit the max size or the JVM can cheat by allocating more memory):

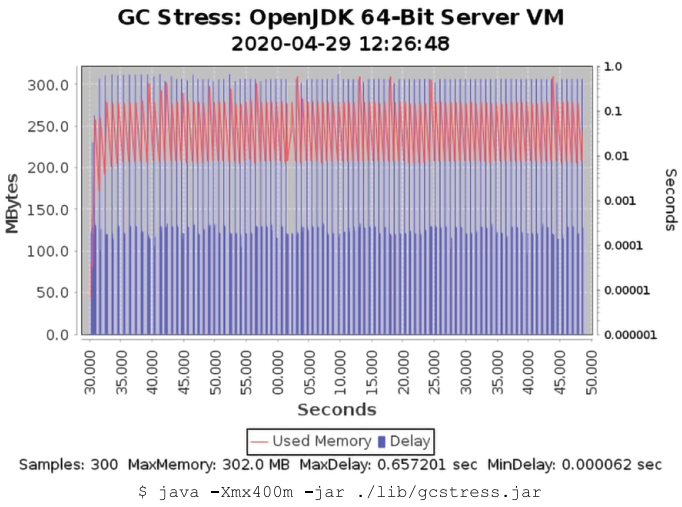

Fig. 5

Remember that OpenJDK 8 uses the Parallel Collector. Looking first at the red graph showing heap usage, you can see that it bounces between 200 MB and 300 MB about once every second. Zooming in on the top-left corner in Figure 6, you can see each decrease in memory usage is followed by a spike in the delay graph in blue. The delay axis is logarithmic on the right side of the chart. The maximum delay is 657 milliseconds and occurs each time the garbage collector runs. When the GC isn't running, the delays are around 200 microseconds.

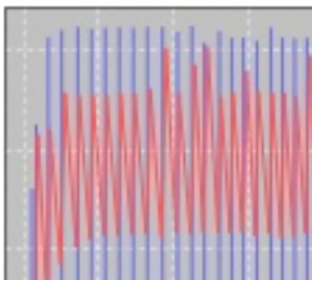

Fig. 6

Figure 7 is OpenJDK 11 with the G1 collector enabled:

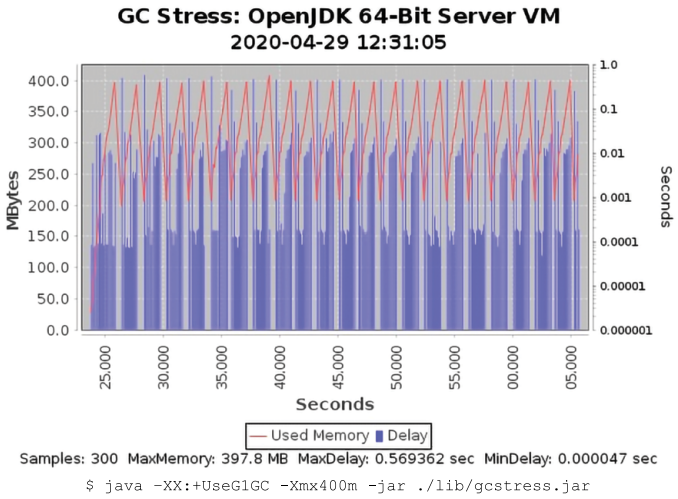

Fig. 7

Notice the memory usage graph peaks roughly every 2-3 seconds, and there are three distinct delay levels: one at about 200 microseconds as with OpenJDK 8, another at about 10 milliseconds, and a third at around 500 milliseconds that no doubt corresponds to major collections. We can speculate that the 10 millisecond delays are for minor collections and only after the heap became overly fragmented did it resort to a major collection. Either way, the worst-case delay is like the Parallel Collector in OpenJDK 8.

Now let's look at PTC Perc running GCStress. If you would like to run this yourself, you may request an evaluation via email to developer-tools-sales@ptc.com. The results, using Perc command-line options to limit maximum heap to 400 MB, are shown in Figure 8:

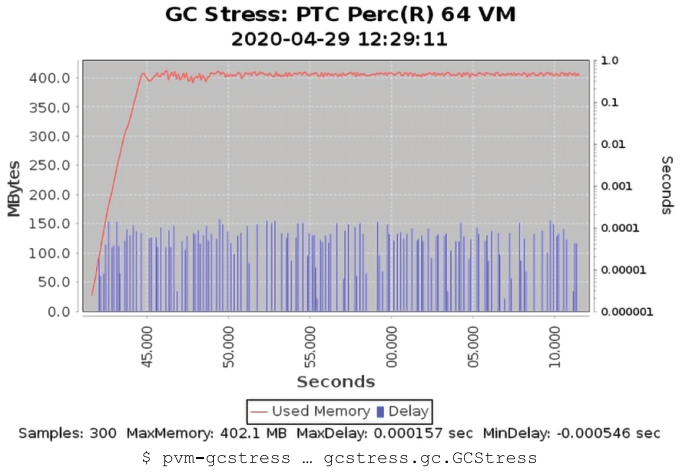

Fig. 8

Note the maximum delay measured by the Timer thread is 157 microseconds and many samples are at or near zero. The heap usage graph rises to the available memory and stays steady. There are no big decreases because the Hammer thread is allocating faster than the collector is reclaiming memory in the background. When the Hammer thread hits the maximum heap memory, it gives up scheduling timeslices to the collector until it can complete the allocation. Thus, memory usage "bounces" a few megabytes below the 400 MB ceiling.

Figure 9 shows the same Perc VM running with 25% CPU allocation to the garbage collector:

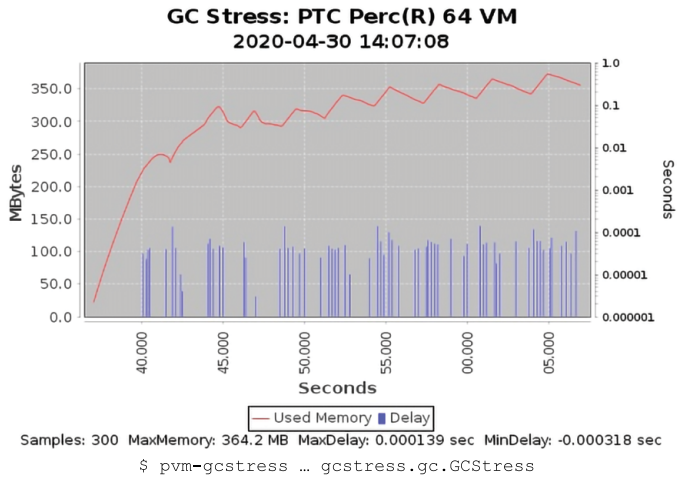

Fig. 9

Now memory usage is rising and falling between collection cycles like OpenJDK. Note the reduction in the number of delays above zero because the Timer thread doesn't have to preempt garbage collector threads when the collector is quiescent.

What Did I Learn?

A real-time Java Virtual Machine can make a big difference if you have a Java application that needs fast, deterministic response for timed events or external inputs. While traditional JVMs have reduced pause times, they still suffer from stop-the-world collection phases. PTC Perc allows high priority threads to preempt the garbage collector to run critical tasks. There are additional features of Perc that are required for real time, such as priority inheritance, jitter-free timing APIs, ahead-of-time compilation, and memory page locking.

If you would like to learn more or evaluate Perc for your real-time application needs, feel free to go to the PTC Perc homepage:

https://www.ptc.com/en/products/developer-tools/perc

Then click on the "Contact Us" button. An account representative will get in touch with you to set it up. We look forward to discussing how Perc can help you build Java-based mission-critical systems with sub-millisecond response times.