Java Magazine

Written by the Java community for Java and JVM developers

Java 18, Java 8, JVM Internals

Java garbage collection: The 10-release evolution from JDK 8 to JDK 18

June 16, 2022 | 19 minute read

Thomas Schatzl

Thousands of enhancements improve throughput, latency, and memory footprint.

The general availability of JDK 18 marked the 10th release since the still-popular JDK 8 release in March 2014. This anniversary is a good opportunity to take pause and see what happened with the HotSpot JVM's garbage collectors along the way.

This article is based on my presentation "JDK 8 to JDK 18 in garbage collection: 10 releases, 2000+ enhancements."

Introducing garbage collection, metrics, and trade-offs

The component of the HotSpot JVM that manages the application heap of your application is called the *garbage collector* (GC). A GC governs the whole lifecycle of application heap objects, beginning when the application allocates memory and continuing through reclaiming that memory for eventual reuse later.

At a very high level, the most basic functionality of garbage collection algorithms in the JVM are the following:

- Upon an allocation request for memory from the application, the GC provides memory. Providing that memory should be as quick as possible.
- The GC detects memory that the application is never going to use again. Again, this mechanism should be efficient and not take an undue amount of time. This unreachable memory is also commonly called *garbage*.
- The GC then provides that memory again to the application, preferably "in time," that is, quickly.

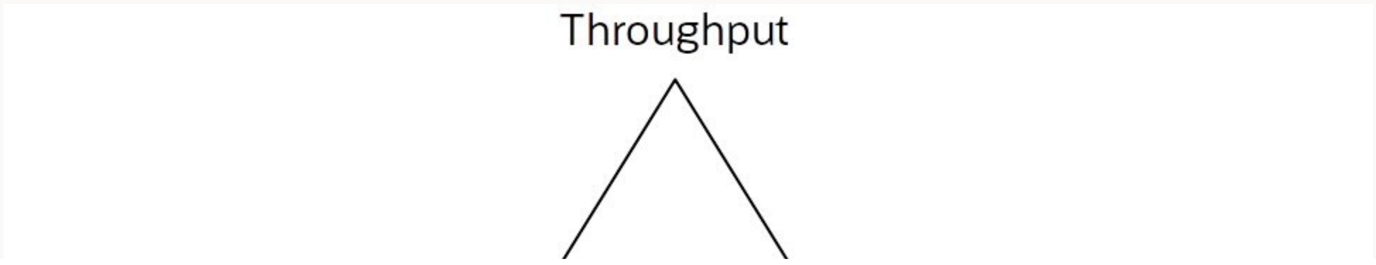
There are many more requirements for a good garbage collection algorithm, but these three are the most basic ones and sufficient for this discussion.

There are many ways to satisfy all these requirements, but unfortunately there is no silver bullet and no one-size-fits-all algorithm. For this reason, the JDK provides a few garbage collection algorithms to choose from, and each is optimized for different use cases. Their implementation roughly dictates behavior about one or more of the three main performance metrics of throughput, latency, and memory footprint and how they impact Java applications.

- *Throughput* represents the amount of work that can be done in a given time unit. In terms of this discussion, a garbage collection algorithm that performs more collection work per time unit is preferable, allowing higher throughput of the Java application.
- *Latency* gives an indication of how long a single operation of the application takes. A garbage collection algorithm focused on latency tries to minimize impacting latency. In the context of a GC, the key concerns are whether its operation induces pauses, the extent of any pauses, and how long the pauses may be.
- *Memory footprint* in the context of a GC means how much extra memory beyond the application's Java heap memory usage the GC needs for proper operation. Data used purely for the management of the Java heap takes away from the application; if the amount of memory the GC (or, more generally, the JVM) uses is less, more memory can be provided to the application's Java heap.

These three metrics are connected: A high throughput collector may significantly impact latency (but minimizes impact on the application) and the other way around. Lower memory consumption may require the use of algorithms that are less optimal in the other metrics. Lower latency collectors may do more work concurrently or in small steps as part of the execution of the application, taking away more processor resources.

This relationship is often graphed in a triangle with one metric in each corner, as shown in **Figure 1**. Every garbage collection algorithm occupies a part of that triangle based on where it is targeted and what it is best at.



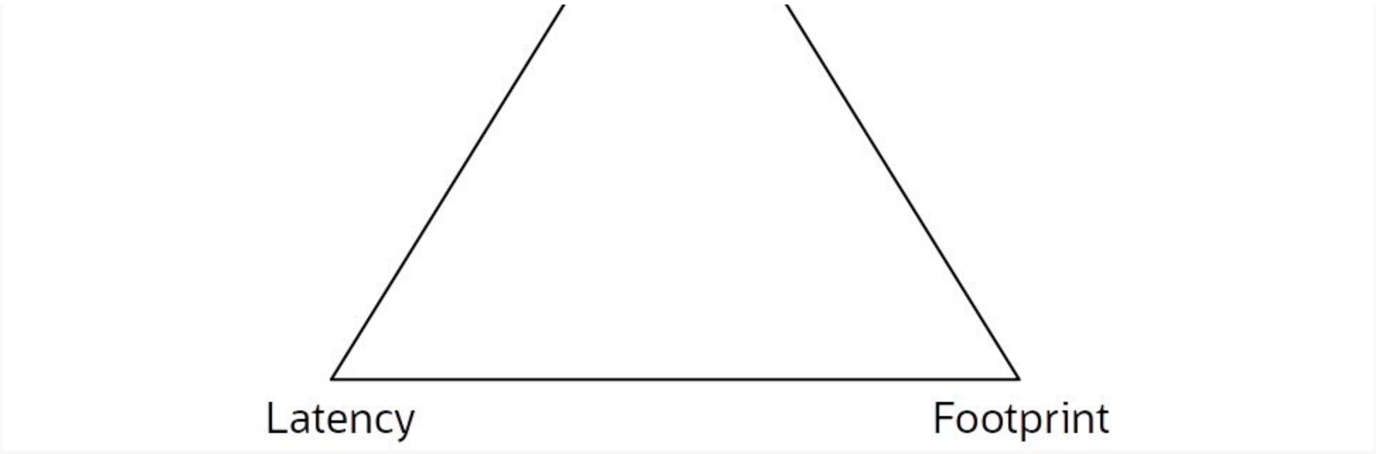


Figure 1. The GC performance metrics triangle

Trying to improve a GC in one or more of the metrics often penalizes the others.

The OpenJDK GCs in JDK 18

OpenJDK provides a diverse set of five GCs that focus on different performance metrics. **Table 1** lists their names, their area of focus, and some of the core concepts used to achieve the desired properties.

Table 1. OpenJDK's five GCs

Garbage collector	**Focus area**	**Concepts**
Parallel	Throughput	Multithreaded stop-the-world (STW) compaction and generational collection
Garbage First (G1)	Balanced performance	Multithreaded STW compaction, concurrent liveness, and generational collection
Z Garbage Collector (ZGC) (since JDK 15)	Latency	Everything concurrent to the application
Shenandoah (since JDK 12)	Latency	Everything concurrent to the application
Serial	Footprint and startup time	Single-threaded STW compaction and generational collection

The Parallel GC is the default collector for JDK 8 and earlier. It focuses on throughput by trying to get work done as quickly as possible with minimal regard to latency (pauses).

The Parallel GC frees memory by evacuating (that is, copying) the in-use memory to other locations in the heap in more compact form, leaving large areas of then-free memory within STW pauses. STW pauses occur when an allocation request cannot be satisfied; then the JVM stops the application completely, lets the garbage collection algorithm perform its memory compaction work with as many processor threads as available, allocates the memory requested in the allocation, and finally continues execution of the application.

The Parallel GC also is a generational collector that maximizes garbage collection efficiency. More on the idea of generational collection is discussed later.

The G1 GC has been the default collector since JDK 9. G1 tries to balance throughput and latency concerns. On the one hand, memory reclamation work is still performed during STW pauses using generations to maximize efficiency—as is done with the Parallel GC—but at the same time, it tries to avoid lengthy operations in these pauses.

G1 performs lengthy work concurrent to the application, that is, while the application is running using multiple threads. This decreases maximum pause times significantly, at the cost of some overall throughput.

The ZGC and Shenandoah GCs focus on latency at the cost of throughput. They attempt to do all garbage collection work without noticeable pauses. Currently neither is generational. They were first introduced in JDK 15 and JDK 12, respectively, as nonexperimental versions.

The Serial GC focuses on footprint and startup time. This GC is like a simpler and slower version of the Parallel GC, as it uses only a single thread for all work within STW pauses. The heap is also organized in generations. However, the Serial GC excels at footprint and startup time, making it particularly suitable for small, short-running applications due to its reduced complexity.

OpenJDK provides another GC, Epsilon, which I omitted from **Table 1**. Why? Because Epsilon only allows memory allocation and never performs any reclamation, it does not meet all the requirements for a GC. However, Epsilon can be useful for some very narrow and special-niche applications.

Short introduction to the G1 GC

The G1 GC was introduced in JDK 6 update 14 as an experimental feature, and it was fully supported beginning with JDK 7 update 4. G1 has been the default collector for the HotSpot JVM since JDK 9 due to its versatility: It is stable, mature, very actively maintained, and it's being improved all the time. I hope the remainder of this article will prove that to you.

How does G1 achieve this balance between throughput and latency?

One key technique is generational garbage collection. It exploits the observation that the most recently allocated objects are the most likely ones that can be reclaimed almost immediately (they "die" quickly). So G1, and any other generational GC, splits the Java heap into two areas: a so-called *young generation* into which objects are initially allocated and an *old generation* where objects that live longer than a few garbage collection cycles for the young generation are placed so they can be reclaimed with less effort.

The young generation is typically much smaller than the old generation. Therefore, the effort for collecting it, plus the fact that a tracing GC such as G1 processes only reachable (live) objects during young-generation collections, means the time spent garbage collecting the young generation generally is short, and a lot of memory is reclaimed at the same time.

At some point, longer-living objects are moved into the old generation.

Therefore, from time to time, there is a need to collect garbage and reclaim memory from the old generation as it fills up. Since the old generation is typically large, and it often contains a significant number of live objects, this can take quite some time. (For example, the Parallel GC's full collections often take many times longer than its young-generation collections.)

For this reason, G1 splits old-generation garbage collection work into two phases.

- G1 first traces through the live objects concurrently to the Java application. This moves a large part of the work needed for reclaiming memory from the old generation out of the garbage collection pauses, thus reducing latency. The actual memory reclamation, if done all at once, would still be very time consuming on large application heaps.
- Therefore, G1 incrementally reclaims memory from the old generation. After the tracing of live objects, for every one of the next few regular young-generation collections, G1 compacts a small part of the old generation in addition to the whole young generation, reclaiming memory there as well over time.

Reclaiming the old generation incrementally is a bit more inefficient than doing all this work at once (as the Parallel GC does) due to inaccuracies in tracing through the object graph as well as the time and space overhead for managing support data structures for incremental garbage collections, but it significantly decreases the maximum time spent in pauses. As a rough guide, garbage collection times for incremental garbage collection pauses take around the same time as the ones reclaiming only memory from the young generation.

In addition, you can set the pause time goal for both of these types of garbage collection pauses via the `MaxGCPauseMillis` command-line option; G1 tries to keep the time spent below this value. The default value for this duration is 200 ms. That might or might not be appropriate for your application, but it is only a guide for the maximum. G1 will keep pause times lower than that value if possible. Therefore, a good first attempt to improve pause times is trying to decrease the value of `MaxGCPauseMillis`.

Progress from JDK 8 to JDK 18

Now that I've introduced OpenJDK's GCs, I'll detail improvements that have been made to the three metrics—throughput, latency, and memory footprint—for the GCs during the last 10 JDK releases.

Throughput gains for G1. To demonstrate the throughput and latency improvements, this article uses the SPECjbb2015 benchmark. SPECjbb2015 is a common industry benchmark that measures Java server performance by simulating a mix of operations within a supermarket company. The benchmark provides two metrics.

- **maxjOPS** corresponds to the maximum number of transactions the system can provide. This is a throughput metric.
- **criticaljOPS** measures throughput under several service-level agreements (SLAs), such as response times, from 10 ms to 100 ms.

This article uses maxjOPS as a base for comparing the throughput for JDK releases and the actual pause time improvements for latency. While criticaljOPS values are representative of latency induced by pause time, there are other sources that contribute to that score. Directly comparing pause times avoids this problem.

Figure 2 shows maxjOPS results for G1 in composite mode on a 16 GB Java heap, graphed relative to JDK 8 for JDK 11 and JDK 18. As you can see, the throughput scores increase significantly simply by moving to later JDK releases. JDK 11 improves by around 5% and JDK 18 by around 18%, respectively, compared to JDK 8. Simply put, with later JDKs, more resources are available and used for actual work in the application.

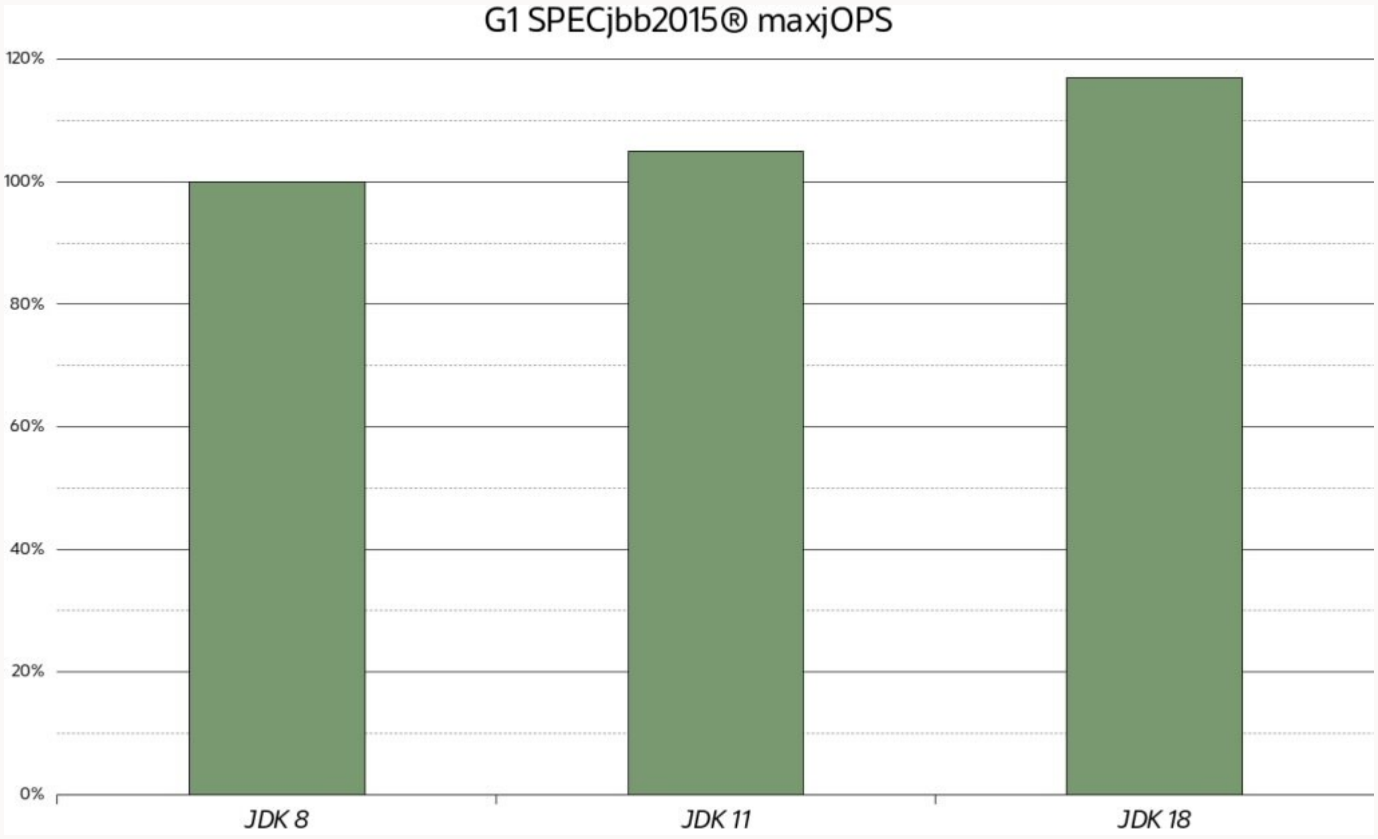


Figure 2. G1 throughput gains measured with SPECjbb2015 maxjOPS

The discussion below attempts to attribute these throughput improvements to particular garbage collection changes. However, garbage collection performance, particularly throughput, is also very amenable to other generic improvements such as code compilation, so the garbage collection changes are not responsible for all the uplift.

One significant improvement early in JDK 9 was how G1 starts the old-generation collection lazily, as late as possible.

In JDK 8 the user had to manually set the time when G1 started concurrent tracing of live objects for old-generation collection. If the time was set too early, the JVM did not use all the application heap assigned to the old generation before starting the reclamation work. One drawback was that this did not give the objects in the old generation as much time to become reclaimable. So G1 would not only take more processor resources to analyze liveness because more data was still live, but also G1 would do more work than necessary freeing memory for the old generation.

Another problem was that if the time to start old-generation collection were set to be too late, the JVM might run out of memory, causing a very slow full collection. Beginning with JDK 9, G1 automatically determines an optimal point at which to start old-generation tracing, and it even adapts to the current application's behavior.

Another idea that was implemented in JDK 9 is related to trying to reclaim large objects in the old generation that G1 automatically places there at a higher frequency than the rest of the old generation. Similar to the use of generations, this is another way the GC focuses on "easy pickings" work that has potentially very high gain—after all, large objects are called large objects because they take lots of space. In some (admittedly rare) applications, this even yields such large reductions in the number of garbage collections and total pause times that G1 beats the Parallel GC on throughput.

In general, every release includes optimizations that make garbage collection pauses shorter while performing the same work. This leads to a natural improvement in throughput. There are many optimizations that could be listed in this article, and the following section about latency improvements points out some of them.

Similar to the Parallel GC, G1 got dedicated nonuniform memory access (NUMA) awareness for allocation to the Java heap in JDK 14. Since then, on computers with multiple sockets where memory access times are nonuniform—that is, where memory is somewhat dedicated to the sockets of the computer, and therefore access to some memory can be slower—G1 tries to exploit locality.

When NUMA awareness applies, the G1 GC assumes that objects allocated on one memory node (by a single thread or thread group) will be mostly referenced from other objects on the same node. Therefore, while an object stays in the young generation, G1 keeps objects on the same node, and it evenly distributes the longer-living objects across nodes in the old generation to minimize access-time variation. This is similar to what the Parallel GC implements.

One more improvement I would like to point out here applies to uncommon situations, the most notable probably being full collections. Normally, G1 tries to prevent full collections by ergonomically adjusting internal parameters. However, in some extreme conditions this is not possible, and G1 needs to perform a full collection during a pause. Until JDK 10, the implemented algorithm was single-threaded, and so it was extremely slow. The current implementation is on par with the Parallel GC's full garbage collection process. It's still slow, and something you want to avoid, but it's much better.

Throughput gains for the Parallel GC. Speaking of the Parallel GC, **Figure 3** shows maxjOPS score improvements from JDK 8 to JDK 18 on the same heap configuration used earlier. Again, only by substituting the JVM, even with the Parallel GC, you can get a modest 2% to around a nice 10% improvement in throughput. The improvements are smaller than with G1 because the Parallel GC started off from a higher absolute value, and there has been less to gain.

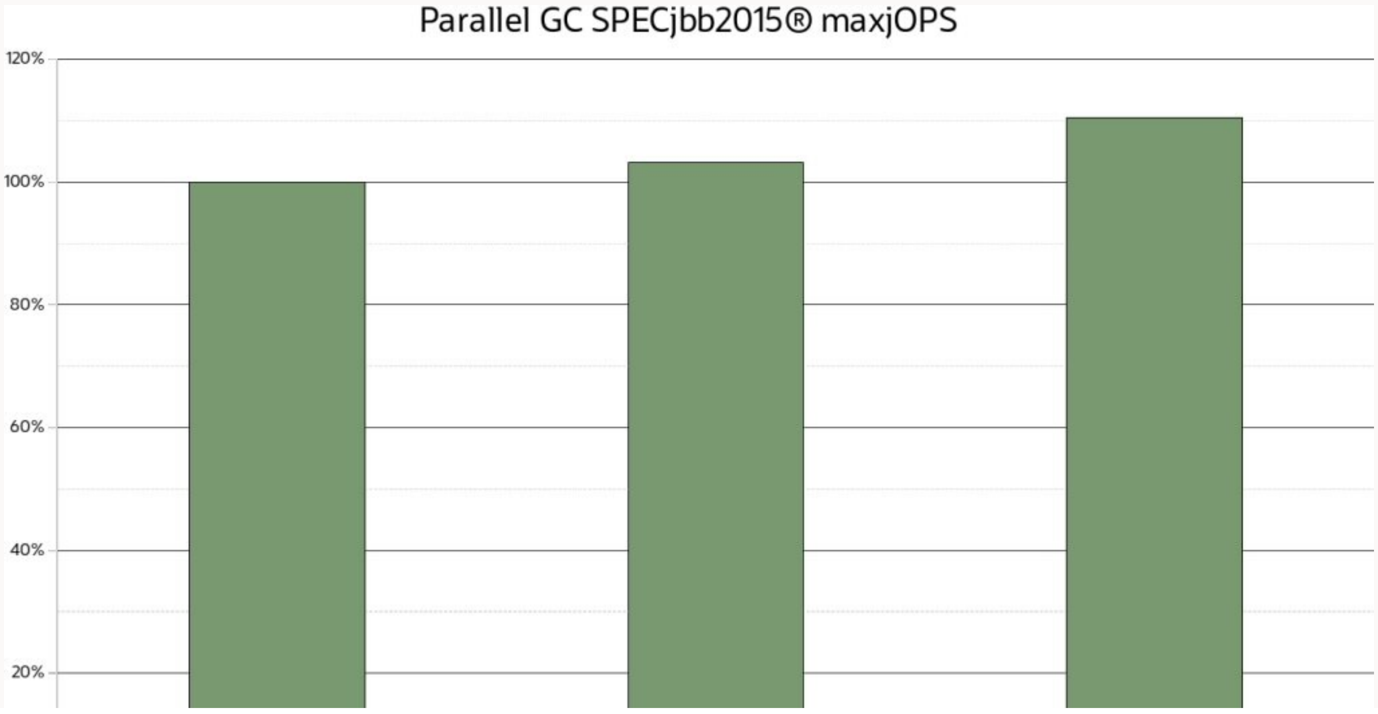


Figure 3. Throughput gains for the Parallel GC measured with SPECjbb2015 maxjOPS

Latency improvements on G1. To demonstrate latency improvements for HotSpot JVM GCs, this section uses the SPECjbb2015 benchmark with a fixed load and then measures pause times. The Java heap size is set to 16 GB. **Table 2** summarizes average and 99th percentile (P99) pause times and relative total pause times within the same interval for different JDK versions at the default pause time goal of 200 ms.

Table 2. Latency improvements with the default pause time of 200 ms

	JDK 8, 200 ms	**JDK 11, 200 ms**	**JDK 18, 200 ms**
Average (ms)	124	111	89
P99 (ms)	176	111	111
Relative collection time (%)	n/a	-15.8	-34.4

JDK 8 pauses take 124 ms on average, and P99 pauses are 176 ms. JDK 11 improves average pause time to 111 ms and P99 pauses to 134 ms—in total spending 15.8% less time in pauses. JDK 18 significantly improves on that once more, resulting in pauses taking 89 ms on average and P99 pause times taking 104 ms—resulting in 34.4% less time in garbage collection pauses.

I extended the experiment to add a JDK 18 run with a pause time goal set to 50 ms, because I arbitrarily decided that the default for `-XX:MaxGCPauseMillis` of 200 ms was too long. G1, on average, met the pause time goal, with P99 garbage collection pauses taking 56 ms (see **Table 3**). Overall, total time spent in pauses did not increase much (0.06%) compared to JDK 8.

In other words, by substituting a JDK 8 JVM with a JDK 18 JVM, you either get significantly decreased average pauses at potentially increased throughput for the same pause time goal, or you can have G1 keep a much smaller pause time goal (50 ms) at the same total time spent in pauses, which roughly corresponds to the same throughput.

Table 3. Latency improvements by setting the pause time goal to 50 ms

	JDK 8, 200 ms	**JDK 11, 200 ms**	**JDK 18, 200 ms**	**JDK 18, 50 ms**
Average (ms)	124	111	89	44
P99 (ms)	176	134	104	56
Relative collection time (%)	n/a	-15.8	-34.4	+0.06

The results in **Table 3** were made possible by many improvements since JDK 8. Here are the most notable ones.

A fairly large contribution to reduced latency was the reduction of the metadata needed to collect parts of the old generation. The so-called *remembered sets* have been trimmed significantly by both improvements to the data structures themselves as well as to not storing and updating never-needed information. In today's computer architectures, a reduction in metadata to be managed means much less memory traffic, which improves performance.

Another aspect related to remembered sets is the fact that the algorithm for finding references that point into currently evacuated areas of the heap has been improved to be more amenable to parallelization. Instead of looking through that data structure in parallel and trying to filter out duplicates in the inner loops, G1 now separately filters out remembered-set duplicates in parallel and then parallelizes the processing of the remainder. This makes both steps more efficient and much easier to parallelize.

Further, the processing of these remembered-set entries has been looked at very thoroughly to trim unnecessary code and optimize for the common paths.

Another focus in JDKs later than JDK 8 has been improving the actual parallelization of tasks within a pause: Changes have attempted to improve parallelization either by making phases parallel or by creating larger parallel phases out of smaller serial ones to avoid unnecessary synchronization points. Significant resources have been spent to improve work balancing within parallel phases so that if a thread is out of work, it should be cleverer when looking for work to steal from other threads.

By the way, later JDKs started looking at more uncommon situations, one of them being *evacuation failure*. Evacuation failure occurs during garbage collection if there is no more space to copy objects into.

Garbage collection pauses on ZGC. In case your application requires even shorter garbage collection pause times, **Table 4** shows a comparison with one of the latency-focused collectors, ZGC, on the same workload used earlier. It shows the pause-time durations presented earlier for G1 plus an additional rightmost column showing ZGC.

Table 4. ZGC latency compared to G1 latency

	JDK 8, 200 ms, G1	**JDK 18, 200 ms, G1**	**JDK 18, 50 ms, G1**	**JDK 18, ZGC**
Average (ms)	124	89	44	0.01
P99 (ms)	176	104	56	0.031

ZGC delivers on its promise of submillisecond pause time goals, moving all reclamation work concurrent to the application. Only some minor work to provide closure of garbage collection phases still needs pauses. As expected, these pauses will be very small: in this case, even far below the suggested millisecond range that ZGC aims to provide.

Footprint improvements for G1. The last metric this article will examine is progress in the memory footprint of the G1 garbage collection algorithm. Here, the footprint of the algorithm is defined as the amount of extra memory *outside of the Java heap* that it needs to provide its functionality.

In G1, in addition to static data dependent on the Java heap size, which takes up approximately 3.2% of the size of the Java heap, often the other main consumer of additional memory is remembered sets that enable generational garbage collection and, in particular, incremental garbage collection of the old generation.

One class of applications that stresses G1's remembered sets is object caches: They frequently generate references between areas within the old generation of the heap as they add and remove newly cached entries.

Figure 4 shows G1 native memory usage changes from JDK 8 to JDK 18 on a test application that implements such an object cache: Objects that represent cached information are queried, added, and removed in a least-recently-used fashion from a large heap. This example uses a Java heap of 20 GB, and it uses the JVM's native memory tracking (NMT) facility to determine memory usage.

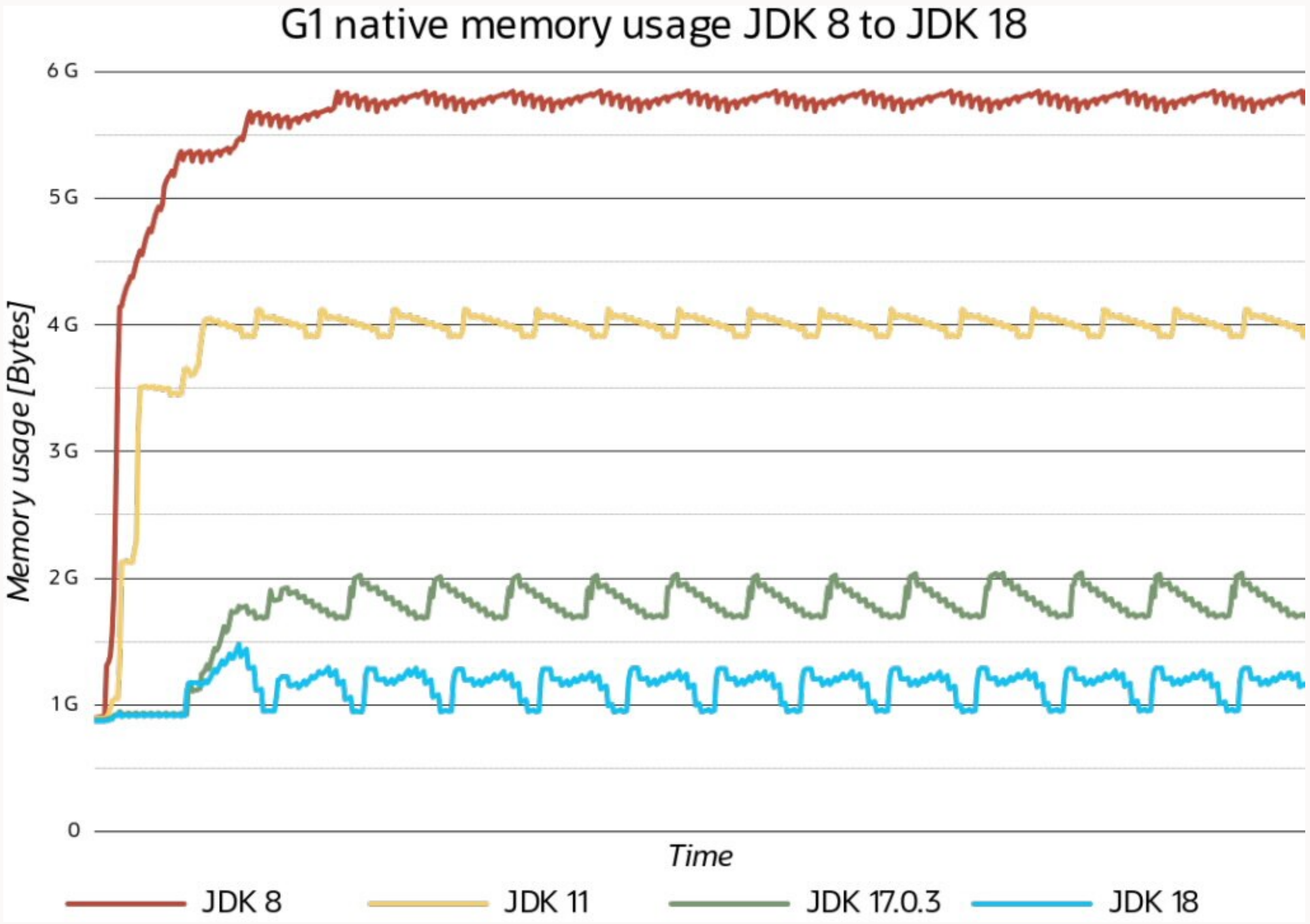


Figure 4. The G1 GC's native memory footprint

With JDK 8, after a short warmup period, G1 native memory usage settles at around 5.8 GB of native memory. JDK 11 improved on that, reducing the native memory footprint to around 4 GB; JDK 17 improved it to around 1.8 GB; and JDK 18 settles at around 1.25 GB of garbage collection native memory usage. This is a reduction of extra memory usage from almost 30% of the Java heap in JDK 8 to around 6% of extra memory usage in JDK 18.

There is no particular cost in throughput or latency associated with these changes, as previous sections showed. Indeed, reducing the metadata the G1 GC maintains generally improved the other metrics so far.

The main principle for these changes from JDK 8 through JDK 18 has been to maintain garbage collection metadata only on a very strict as-needed basis, maintaining only what is expected to be needed when it is needed. For this reason, G1 re-creates and manages this memory concurrently, freeing data as quickly as possible. In JDK 18, enhancements to the representation of this metadata and storing it more densely contributed significantly to the improvement of the memory footprint.

Figure 4 also shows that in later JDK releases G1 increased its aggressiveness, step by step, in giving back memory to the operating system by looking at the difference between peaks and valleys in steady-state operations—in the last release, G1 even does this process concurrently.

The future of garbage collection

Although it is hard to predict what the future holds and what the many projects to improve garbage collection and, in particular, G1, will provide, some of the following developments are more likely to end up in the HotSpot JVM in the future.

One problem that is actively being worked on is removing the need to lock out garbage collection when Java objects are used in native code: Java threads triggering a garbage collection must wait until no other regions are holding references to Java objects in native code. In the worst cases, native code may block garbage collection for minutes. This can lead to software developers choosing to not use native code at all, affecting throughput adversely. With the changes suggested in JEP 423 (Region pinning for G1), this will become a nonissue for the G1 GC.

Another known disadvantage of using G1 compared to the throughput collector, Parallel GC, is its impact on throughput—users report differences in the range of 10% to 20% in extreme cases. The cause of this problem is known, and there have been a few suggestions on how to improve this drawback without compromising other qualities of the G1 GC.

Fairly recently, it's been determined that pause times and, in particular, work distribution efficiency in the garbage collection pauses are still less than optimal.

One current focus of attention is removing one-half of G1's largest helper data structure, the mark bitmaps. There are two bitmaps used in the G1 algorithm that help with determining which objects are currently live and can be safely concurrently inspected for references by G1. An open enhancement request indicates that the purpose of one of these bitmaps could be replaced by other means. That would immediately reduce G1 metadata by a fixed 1.5% of the Java heap size.

There is much ongoing activity to change the ZGC and Shenandoah GCs to be generational. In many applications, the current single-generational design of these GCs has too many disadvantages regarding throughput and timeliness of reclamation, often requiring much larger heap sizes to compensate.

Conclusion

This article has shown that improvements to the HotSpot JVM garbage collection algorithms from JDK 8 through JDK 18 have been significant, as all three of the performance indicators—throughput, latency, and memory footprint—were improved by nontrivial amounts. Every new JDK release, even if it did not memory footprint—were improved by nontrivial amounts. Every new JDK release, even if it did not explicitly point out such improvements in JEPs, provided tangible improvements. It will likely stay that way for the foreseeable future, so keep up to date and enjoy the for-free improvements!

Thanks go to the many OpenJDK contributors who made all these great improvements possible over time.

Dig deeper

- Understanding the JDK's new superfast garbage collectors
- Understanding garbage collectors
- Epsilon: The JDK's do-nothing garbage collector
- Per Liden's garbage collector notes
- *HotSpot Virtual Machine Garbage Collection Tuning Guide*
- JDK 18 G1/Parallel/Serial GC changes

Thomas Schatzl
Thomas Schatzl, based in Austria, is a principal member of technical staff at Oracle. He has been contributing to the HotSpot JVM garbage collectors since 2012.

Previous Post

Security Practices

What is Talent Management?

What is VM?

Premier League

Oracle and Red Bull Racing Honda