

JAVA ENTERPRISE PERFORMANCE

ALOIS REITBAUER
KLAUS ENZENHOFER
ANDREAS GRABNER
MICHAEL KOPP
STEPHEN PIERZCHALA
STEVE WILSON

Java Enterprise Performance

Chapter 2 Java Memory Management

By
Michael Kopp

© 2012 Compuware Corporation

All rights reserved under the Copyright Laws of the United States.



This work is licensed under a
[Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License](https://creativecommons.org/licenses/by-nc-nd/3.0/).

Java Memory Management

What's in this chapter?

1. [How Garbage Collection Really Works](#)
 2. [The Impact of Garbage Collection on Application Performance](#)
 3. [Reducing Garbage-Collection Pause Time](#)
 4. [Making Garbage Collection Faster](#)
 5. [Not All JVMs Are Created Equal](#)
 6. [Analyzing the Performance Impact of Memory Utilization and Garbage Collection](#)
 7. [Tuning](#)
 8. [GC-Configuration Problems](#)
-

JAVA MEMORY MANAGEMENT, with its built-in garbage collection, is one of the language's finest achievements. It allows developers to create new objects without worrying explicitly about memory allocation and deallocation, because the garbage collector automatically reclaims memory for reuse. This enables faster development with less boilerplate code, while eliminating memory leaks and other memory-related problems. At least in theory.

Ironically, Java garbage collection seems to work too well, creating and removing too many objects. Most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system. In order to wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

How Garbage Collection Really Works

Many people think garbage collection collects and discards dead objects. In reality, Java garbage collection is doing the opposite! Live objects are tracked and everything else designated garbage. As you'll see, this fundamental misunderstanding can lead to many performance problems.

Let's start with the heap, which is the area of memory used for dynamic allocation. In most configurations the operating system allocates the heap in advance to be managed by the JVM while the program is running. This has a couple of important ramifications:

- Object creation is faster because global synchronization with the operating system is not needed for every single object. An allocation simply claims some portion of a memory array and moves the offset pointer forward (see Figure 2.1). The next allocation starts at this offset and claims the next portion of the array.
- When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system.

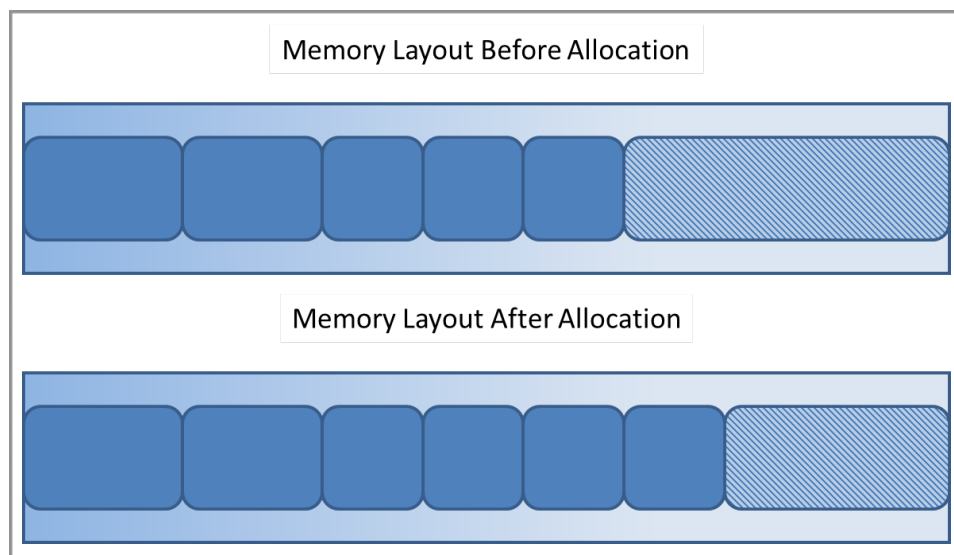


Figure 2.1: New objects are simply allocated at the end of the used heap.

All objects are allocated on the heap area managed by the JVM. Every item that the developer uses is treated this way, including class objects, static variables, and even the code itself. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory. As simple as this sounds, it raises a question: what is the first reference in the tree?

Garbage-Collection Roots—The Source of All Object Trees

Every object tree must have one or more root objects. As long as the application can reach those roots, the whole tree is reachable. But when are those root objects considered reachable? Special objects called garbage-collection roots (GC roots; see Figure 2.2) are always reachable and so is any object that has a garbage-collection root at its own root.

There are four kinds of GC roots in Java:

- **Local variables** are kept alive by the stack of a thread. This is not a real object virtual reference and thus is not visible. For all intents and purposes, local variables are GC roots.
- **Active Java threads** are always considered live objects and are therefore GC roots. This is especially important for thread local variables.
- **Static variables** are referenced by their classes. This fact makes them de facto GC roots. Classes themselves can be garbage-collected, which would remove all referenced static variables. This is of particular importance when we use application servers, [OSGi containers](#), or class loaders in general, which are discussed below in the [Problem Patterns section](#).
- **Java Native Interface (JNI) references** are Java objects that the native code has created as part of a JNI call. Objects thus created are treated specially because the JVM does not know if it is being referenced by the native code or not. Such objects represent a very special form of GC root, which we will examine in more detail in the [Problem Patterns section](#) below.

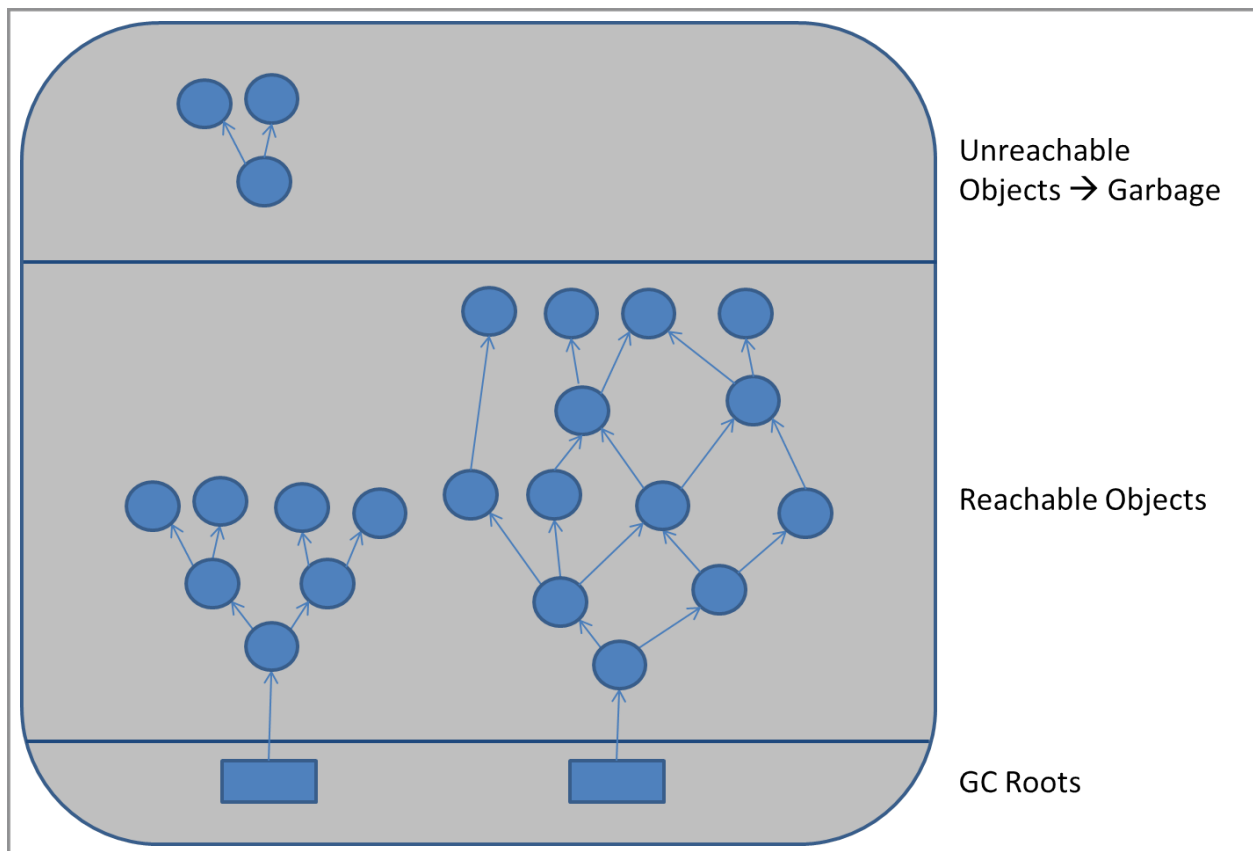


Figure 2.2: GC roots are objects that are themselves referenced by the JVM and thus keep every other object from being garbage-collected.

Therefore, a simple Java application has the following GC roots:

- Local variables in the main method
- The main thread
- Static variables of the main class

Marking and Sweeping Away Garbage

To determine which objects are no longer in use, the JVM intermittently runs what is very aptly called a [mark-and-sweep algorithm](#). As you might intuit, it's a straightforward, two-step process:

1. The algorithm traverses all object references, starting with the GC roots, and marks every object found as alive.
2. All of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept clean of unused objects.

Garbage collection is intended to remove the cause for classic memory leaks: unreachable-but-not-deleted objects in memory. However, this works only for memory leaks in the original sense. It's possible to have unused objects that are still reachable by an application because the developer simply forgot to dereference them. Such objects cannot be garbage-collected. Even worse, such a logical memory leak cannot be detected by any software (see Figure 2.3). Even the best analysis software can only highlight suspicious objects. We will examine memory leak analysis in the [Analyzing the Performance Impact of Memory Utilization and Garbage Collection](#) section, below.

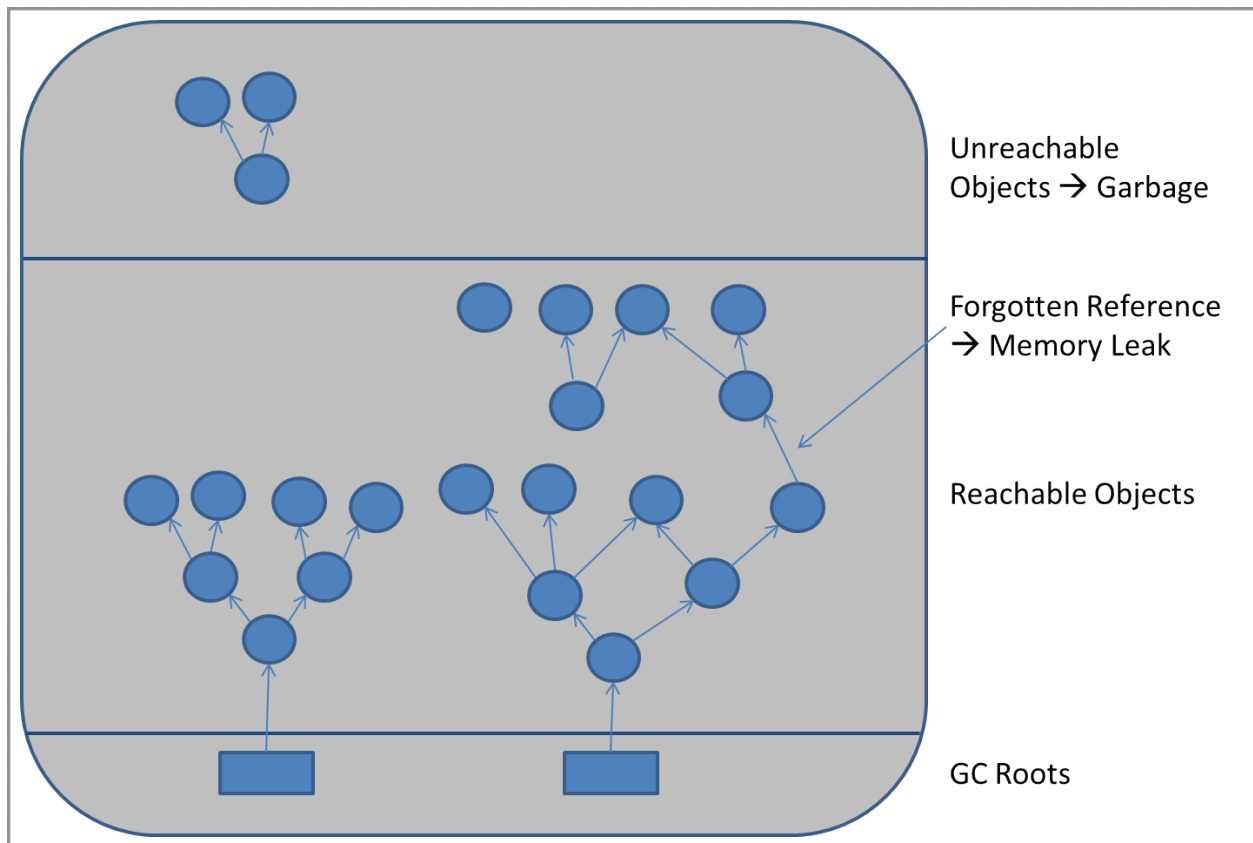


Figure 2.3: When objects are no longer referenced directly or indirectly by a GC root, they will be removed. There are no classic memory leaks. Analysis cannot really identify memory leaks; it can only point out suspicious objects.

The Impact of Garbage Collection on Application Performance

As we've seen, the performance of the garbage collector is not determined by the number of dead objects, but rather by the number of live ones. The more objects die, the faster garbage collection is. If every object in the heap were to be garbage-collected, the GC cycle would be nearly instantaneous. Additionally, the garbage collector must suspend the execution of the application to ensure the integrity of the object trees. The more live objects are found, the longer the suspension, which has a direct impact on response time and throughput.

This fundamental tenet of garbage collection and the resulting effect on application execution is called the garbage-collection pause or GC pause time. In applications with multiple threads, this can quickly lead to scalability problems.

Figure 2.4 illustrates the impact that GC suspensions have on the throughput of multithreaded applications. An application that spends 1% of its execution time on garbage collection will lose more than 20% throughput on a 32-processor system. If we increase the GC time to 2%,

the overall throughput will drop by another 20%. Such is the impact of suspending 32 executing threads simultaneously!

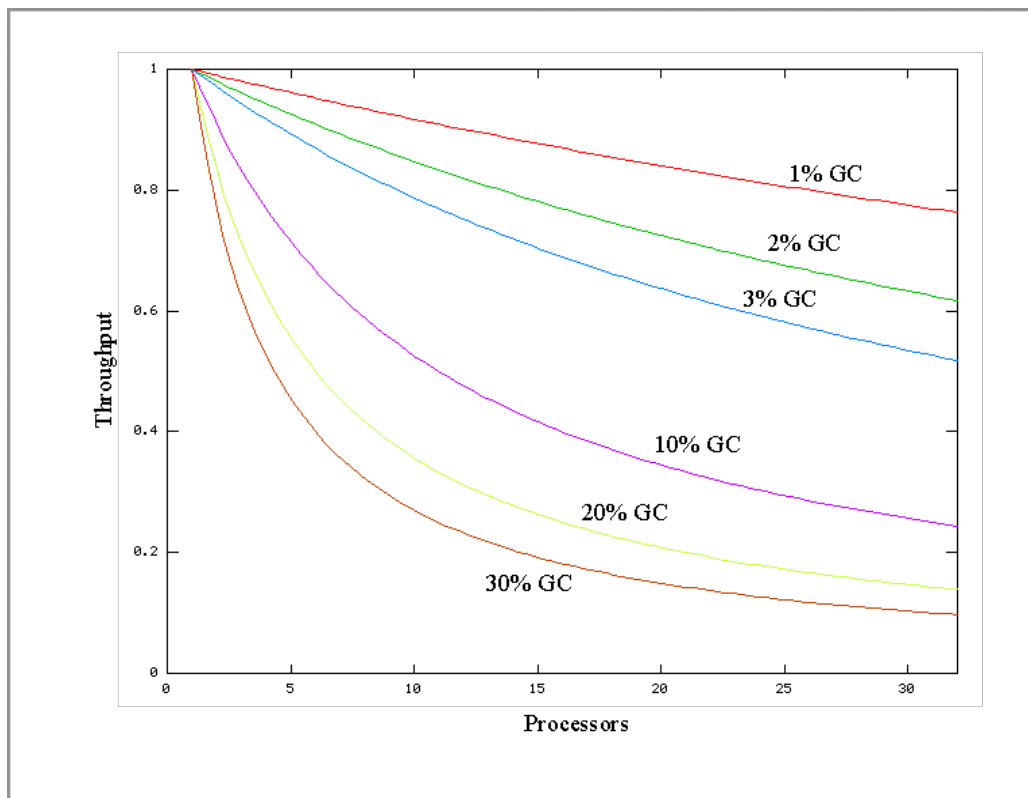


Figure 2.4: This graphic from an [Oracle GC Tuning article](#) illustrates the performance hit of GC suspensions with increasing number of CPUs. (Source: Oracle)

There are two commonly used methods to reduce GC pause time:

- Reducing suspension time by adjusting the mark-and-sweep algorithm
- Limiting the number of objects that need to be marked

But before we examine ways to improve garbage-collection performance, you should understand memory fragmentation, which impacts suspension time and application performance.

Fragmented Memory and Finding a Big Enough Hole

Whenever we create a new object in Java, the JVM automatically allocates a block of memory large enough to fit the new object on the heap. Repeated allocation and reclamation leads to memory fragmentation, which is similar to disk fragmentation. Memory fragmentation leads to two problems:

- **Reduced allocation speed:** The JVM tracks free memory in lists organized by block size. To create a new object, Java searches through the lists to select and allocate an optimally sized block. Fragmentation slows the allocation process, effectively slowing the application execution.
- **Allocation errors:** Allocation errors happen when fragmentation becomes so great that the JVM is unable to allocate a sufficiently large block of memory for a new object.

Java does not rely on the operating system to solve these problems and must deal with these itself. Java avoids memory fragmentation by executing *compaction* (Figure 2.5) at the end of a successful GC cycle. The process is very similar to hard-disk defragmentation.

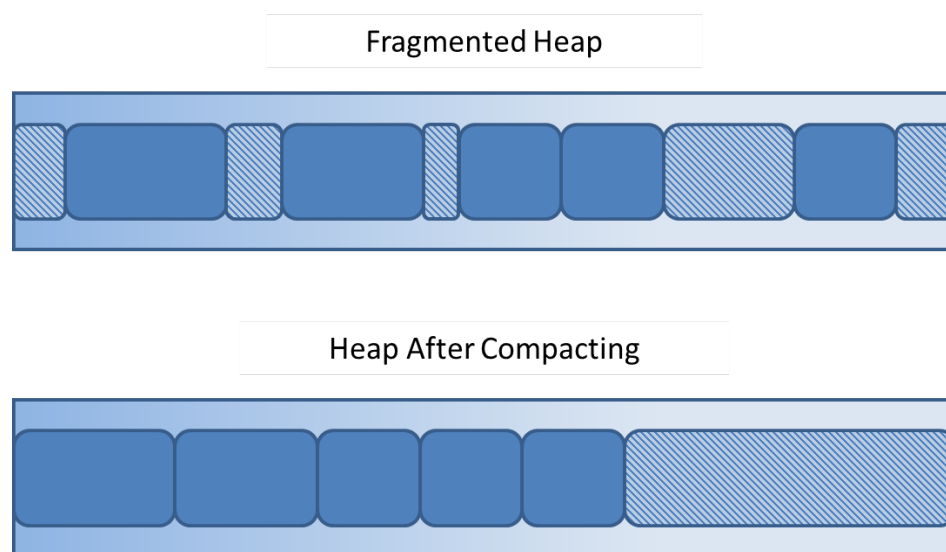


Figure 2.5: When the heap becomes fragmented due to repeated allocations and garbage collections, the JVM executes a compaction step, which aligns all objects neatly and closes all holes.

Compacting simply moves all live objects to one end of the heap, which effectively closes all holes. Objects can be allocated at full speed (no free lists are needed anymore), and problems creating large objects are avoided.

The downside is an even longer GC cycle, and since most JVMs suspend the application execution during compaction, the performance impact can be considerable.

Reducing Garbage-Collection Pause Time

There are two general ways to reduce garbage-collection pause time and the impact it has on application performance:

- The garbage collection itself can leverage the existence of multiple CPUs and be executed in parallel. Although the application threads remain fully suspended during this time, the garbage collection can be done in a fraction of the time, effectively reducing the suspension time.
- The second approach is to leave the application running, and execute garbage collection concurrently with the application execution.

These two logical solutions have led to the development of serial, parallel, and concurrent [garbage-collection strategies](#), which represent the foundation of all existing Java garbage-collection implementations (see Figure 2.6).

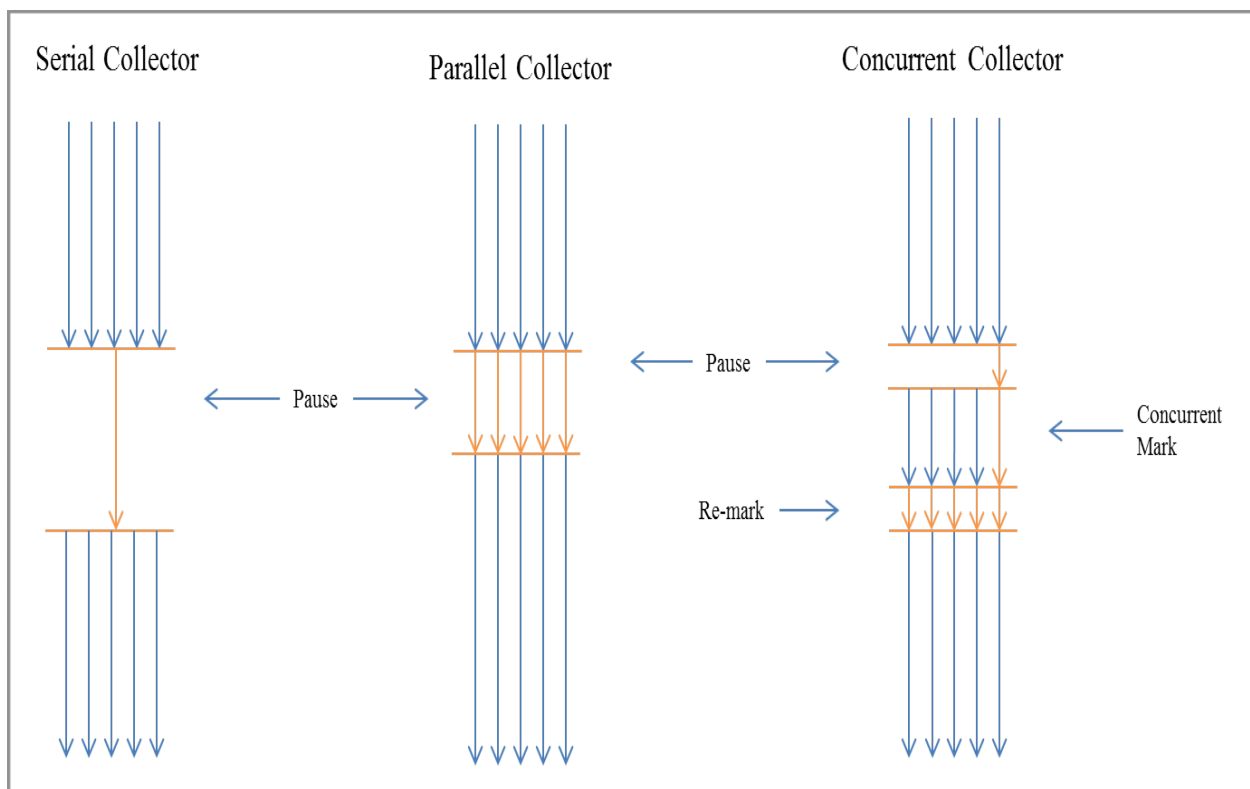


Figure 2.6: The differences between garbage-collection algorithms becomes clearest when comparing garbage-collection suspensions.

The serial collector suspends the application and executes the mark-and-sweep algorithm in a single thread. It is the simplest and oldest form of garbage collection in Java and is still the default in the Oracle HotSpot JVM.

The parallel collector uses multiple threads to do its work. It can therefore decrease the GC pause time by leveraging multiple CPUs. It is often the best choice for throughput applications.

The concurrent collector does the majority of its work concurrent with the application execution. It has to suspend the application for only very short amounts of time. This has a big benefit for response-time-sensitive applications, but is not without drawbacks.

(Mostly) Concurrent Marking and Sweeping (CMS)

Concurrent garbage-collection strategies complicate the relatively simple mark-and-sweep algorithm a bit. The mark phase is usually sub-divided into some variant of the following:

1. In the initial marking, the GC root objects are marked as alive. During this phase, all threads of the application are suspended.
2. During concurrent marking, the marked root objects are traversed and all reachable objects are marked. This phase is fully concurrent with application execution, so all application threads are active and can even allocate new objects. For this reason there might be another phase that marks objects that have been allocated during the concurrent marking. This is sometimes referred to as pre-cleaning and is still done concurrent to the application execution.
3. In the final marking, all threads are suspended and all remaining newly allocated objects are marked as alive. This is indicated in Figure 2.6 by the re-mark label.

The concurrent mark works mostly, but not completely, without pausing the application. The tradeoff is a more complex algorithm and an additional phase that is not necessary in a normal stop-the-world GC: the final marking.

The Oracle JRockit JVM improves this algorithm with the help of a keep area, which, if you're interested, is described in detail in the [JRockit documentation](#). New objects are kept separately and not considered garbage during the first GC. This eliminates the need for a final marking or re-mark.

In the sweep phase of the CMS, all memory areas not occupied by marked objects are found and added to the free list. In other words, the objects are swept by the GC. This phase can run at least partially concurrent to the application. For instance, JRockit divides the heap into two areas of equal size and sweeps one then the other. During this phase, no threads are stopped, but allocations take place only in the area that is not actively being swept.

The downsides of the CMS algorithm can be quickly identified:

- As the marking phase is concurrent to the application's execution, the space allocated for objects can surpass the capacity of the CMS, leading to an allocation error.
- The free lists immediately lead to memory fragmentation and all this entails.
- The algorithm is more complicated than the other two and consequently requires more CPU cycles.

- The algorithm requires more fine tuning and has more configuration options than the other approaches.

These disadvantages aside, the CMS will nearly always lead to greater predictability and better application response time.

Reducing the Impact of Compacting

Modern garbage collectors execute their compacting processes in parallel, leveraging multiple CPUs. Nevertheless, nearly all of them have to suspend the application during this process. JVMs with several gigabytes of memory can be suspended for several seconds or more. To work around this, the various JVMs each implements a set of parameters that can be used to compact memory in smaller, incremental steps instead of as a single big block. The parameters are as follows:

- Compacting is executed not for every GC cycle, but only once a certain level of fragmentation is reached (e.g., if more than 50% of the free memory is not continuous).
- One can configure a target fragmentation. Instead of compacting everything, the garbage collector compacts only until a designated percentage of the free memory is available as a continuous block.

This works, but the optimization process is tedious, involves a lot of testing, and needs to be done again and again for every application to achieve optimum results.

Making Garbage Collection Faster

Short of avoiding garbage collection altogether, there is only one way to make garbage collection faster: ensure that as few objects as possible are reachable during the garbage collection. The fewer objects that are alive, the less there is to be marked. This is the rationale behind the generational heap.

The Generation Conflict—Young vs. Old

In a typical application most objects are very short-lived. On the other hand, some objects last for a very long time and even until the application is terminated. When using generational garbage collection, the heap area is divided into two areas—a young generation and an old generation—that are garbage-collected via separate strategies.

Objects are usually created in the young area. Once an object has survived a couple of GC cycles it is tenured to the old generation. (The JRockit and IBM JVM make exceptions for very large objects, as I will explain later.) After the application has completed its initial startup phase (most applications allocate caches, pools, and other permanent objects during startup), most allocated objects will not survive their first or second GC cycle. The number of live objects that need to be considered in each cycle should be stable and relatively small.

Allocations in the old generation should be infrequent, and in an ideal world would not happen at all after the initial startup phase. If the old generation is not growing and therefore not running out of space, it requires no garbage-collection at all. There will be unreachable objects in the old generation, but as long as the memory is not needed, there is no reason to reclaim it.

To make this generational approach work, the young generation must be big enough to ensure that all temporary objects die there. Since the number of temporary objects in most applications depends on the current application load, the optimal young generation size is load-related. Therefore, sizing the young generation, known as generation-sizing, is the key to achieving peak load.

Unfortunately, it is often not possible to reach an optimal state where all objects die in the young generation, and so the old generation will often require a concurrent garbage collector. Concurrent garbage collection together with a minimally growing old generation ensures that the unavoidable, stop-the-world events will at least be very short and predictable.

On the other hand, while there is a very high number of allocations in the young generation at the beginning of each GC cycle, there is only a small portion of objects alive after each GC cycle. This leads to a high level of fragmentation using the GC strategies we have discussed so far. You might think that using free lists would be a good option, but this will slow down allocations. An alternative strategy of executing a full compaction every time has a negative effect on pause time. Instead, most JVMs implement a strategy in the young generation, known as copy collection.

When Copying Is Faster than Marking

Copy garbage collection divides the heap into two (or more) areas, only one of which is used for allocations. When this area is full, all live objects are copied to the second area, and then the first area is simply declared empty (see Figure 2.7).

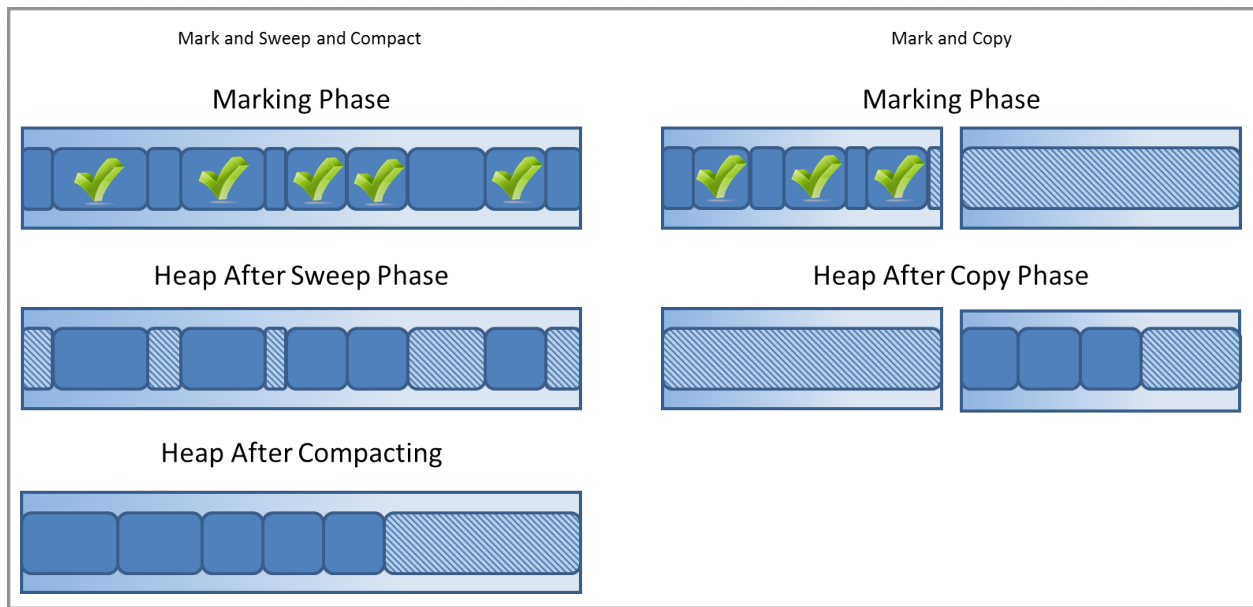


Figure 2.7: Instead of sweeping away garbage and compacting the heap, copy collection simply copies live objects someplace else and declares the old region empty.

The advantage here is that no fragmentation occurs, and thus there is no need for free lists and compaction. Allocations are always fast and the GC algorithm is simple. This strategy is effective only if most objects die, which is the default in the young generation. However, this can lead to a problem when the JVM is executing a high number of concurrent transactions.

If the young generation is too small, objects are tenured prematurely to the old generation. If the young generation is too large, too many objects are alive (undead) and the GC cycle will take too long. Contrary to what most people think, these young-generation GCs, often termed *minor* GCs, are full of stop-the-world events. Their negative impact on response time can be more severe than the occasional old-generation GC.

Not surprisingly, the generational heap does not provide a silver-bullet solution to the garbage-collection problem. The optimal configuration is often a compromise, with an adequately sized young generation to avoid overly long minor GCs, and a concurrent GC in the old generation to deal with prematurely tenured objects.

The Case for a Non-generational Heap

The Oracle HotSpot JVM uses a generational heap exclusively, while Oracle JRockit also supports a non-generational heap, and IBM WebSphere defaults to a non-generational heap and recommends that [JVMs smaller than 100 MB always use a non-generational heap](#). in terms of both CPU and memory, a generational GC and the associated copy collection have a certain overhead, which makes sense.

If your application is designed for pure throughput and the number of temporary objects is relatively small, a non-generational GC has advantages. A full parallel GC has a better tradeoff in terms of CPU usage if you don't care about the response time of a single transaction. On the other hand, if the number of temporary objects is relative small, a concurrent GC in a non-generational heap might do the job and with less suspension time than a generational GC. Only a thorough performance test can determine this for sure.

Improving Allocation Performance

Two factors have a negative impact on allocation speed: fragmentation and concurrency. We have already dealt with fragmentation. Concurrency's problem is that all threads in the JVM share memory resources, and all memory allocations must be synchronized. When many threads try to allocate simultaneously, this quickly becomes a choke point. The solution to this is [thread-local allocation](#).

Each thread receives a small, but exclusive, memory allotment where objects are allocated without the need for synchronization. This increases concurrency and the speed of application execution. (It's important that this not be confused with the heap area for thread-local variables, where allocated objects are accessible by all threads.)

A single thread-local heap (TLH) to accommodate a large number of threads can still be quite small. The TLH is not treated as a special heap area, but instead is usually a part of the young generation, which creates its own problems.

A generational heap with TLH requires a larger young generation than without a TLH. The same number of objects simply uses more space. On the other hand, a non-generational heap with active TLH is likely to become more fragmented and will need more-frequent compaction.

Not All JVMs Are Created Equal

Many developers know only a single JVM (Java Virtual Machine), the Oracle HotSpot JVM (formerly Sun JVM), and speak of garbage collection in general when they are referring to Oracle's HotSpot implementation specifically. It may seem as though there is an industry default, but such is not the case! In fact, the two most popular application servers, IBM WebSphere and Oracle WebLogic, each come with their own JVM. In this section, we will examine some of the enterprise-level garbage-collection specifics of the three most prominent JVMs, the Oracle HotSpot JVM, the IBM WebSphere JVM, and the Oracle JRockit JVM.

Oracle HotSpot JVM

The [Oracle HotSpot JVM](#) uses a generational garbage-collection scheme exclusively (see Figure 2.8). (We'll discuss Oracle's plans to implement a G1, Garbage First, collector below.)

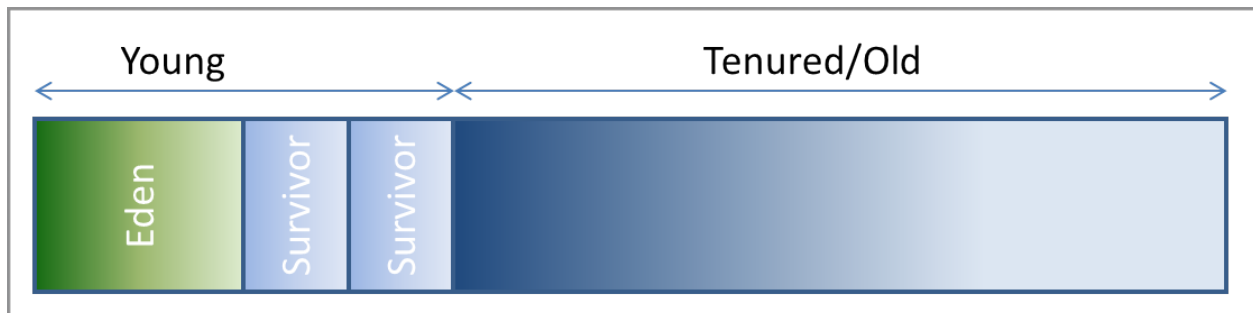


Figure 2.8: The Oracle HotSpot JVM always uses a generational garbage-collection heap layout.

Objects are allocated in the Eden space, which is always considerably larger than the survivor spaces (default ratio is 1:8, but can be configured). The copy GC algorithm is executed either single-threaded or in parallel. It always copies surviving objects from the Eden, and currently used survivor into the second (currently unused) survivor space. Copying an object is, of course, more expensive than simply marking it, which is why the Eden space is the biggest of the three young-generation spaces. The vast majority of objects die in their infancy. A bigger Eden will ensure that these objects will not survive the first GC cycle, and thus not be copied at all. Once an object has survived multiple GC cycles (how many can be configured) it is tenured to the old generation (the old generation is also referred to as tenured space).

It is, of course, entirely possible for objects to tenure prematurely. The size and ratio of the areas has a big influence on allocation speed, GC efficiency, and frequency and depends completely on the application behavior. The optimal solution can be found only by applying this knowledge and a lot of testing.

The old generation can be configured to use either a serial (default), parallel, or concurrent garbage collection. The parallel collector is also a compacting one, but the way it does the compaction can be configured with a wide variety of options. The concurrent GC, on the other hand, does not compact at all. This can lead to allocation errors due to fragmentation (no large-enough blocks available). If this happens the CMS triggers a full GC which effectively uses the normal GC (and collects the young generation as well).

All these combinations lead to a variety of options and configuration possibilities and can make finding an optimal GC configuration seem quite complicated. In the [Tuning Section](#) later in this chapter, we will cover the most important optimization: determining the optimal size for the young and old generations. This optimization ensures that only long-lived objects get tenured, while avoiding too many young-generation GCs and thus too many suspensions.

The HotSpot JVM has an additional unique feature, called the permanent generation, to help make the garbage-collection process more efficient. Java maintains the application code and classes as objects within the heap. For the most part these objects are permanent and need not be considered for garbage collection. The HotSpot JVM therefore improves the garbage-

collection performance by placing class objects and constants into the permanent generation. It effectively ignores them during regular GC cycles.

For better or for worse, the proliferation of application servers, OSGi containers, and dynamically-generated code has changed the game, and objects once considered permanent are not so permanent after all. To avoid out-of-memory errors, the permanent generation is garbage-collected during a major, or full, GC only.

In the [Problem Pattern Section](#) below, we will examine the issue of out-of-memory errors in the permanent collection, which wasn't designed to handle modern use cases. Today's application servers can load an amazing number of classes—often more than 100,000, which pretty much busts whatever default allocation has usually been set. We will also discuss the memory-leak problems caused by dynamic bytecode libraries.

Note: For further information, here's a link to a detailed description of [Oracle HotSpot JVM memory management](#).

Garbage First: G1 Collection

Oracle's Java 7 will implement G1 garbage-collection (with backports to Java 6), using what is known as a *garbage first* algorithm. The underlying principle is very simple, and it is expected to bring substantial performance improvements. Here's how it works.

The heap is divided into a number of fixed subareas. A list with references to objects in the area, called a *remember set*, is kept for each subarea. Each thread then informs the GC if it changes a reference, which could cause a change in the remember set. If a garbage collection is requested, then areas containing the most garbage are swept first, hence garbage first. In the best case (likely the most common, as well), an area will contain no living objects and is simply defined as free—no annoying mark-and-sweep process and no compacting. In addition, it is possible to define targets with the G1 collector, such as overhead or pause times. The collector then sweeps only as many areas as it can in the prescribed interval.

In this way, G1 combines the advantages of a generational GC with the flexibility of a continuous garbage collector. G1 also supports thread-local allocation and thus combines the advantages of all three garbage collection methods we've discussed—at least theoretically.

For instance, where a generational heap helps find the the correct size for the young generation, often a source of problems, G1 is intended to obviate this sizing problem entirely. However, Oracle has indicated that G1 is not yet ready for production, and still considers the G1 experimental and not ready for production use.

IBM WebSphere JVM

As of Java 5, the [IBM WebSphere JVM has added generational GC configuration](#) option to its classic mark-and-sweep algorithm. The default setup still uses a single big heap with either a parallel or a concurrent GC strategy. This is recommended for applications with a small heap

size, not greater than 100 MB, but is not suitable for large or more-complex applications, which should use the generational heap.

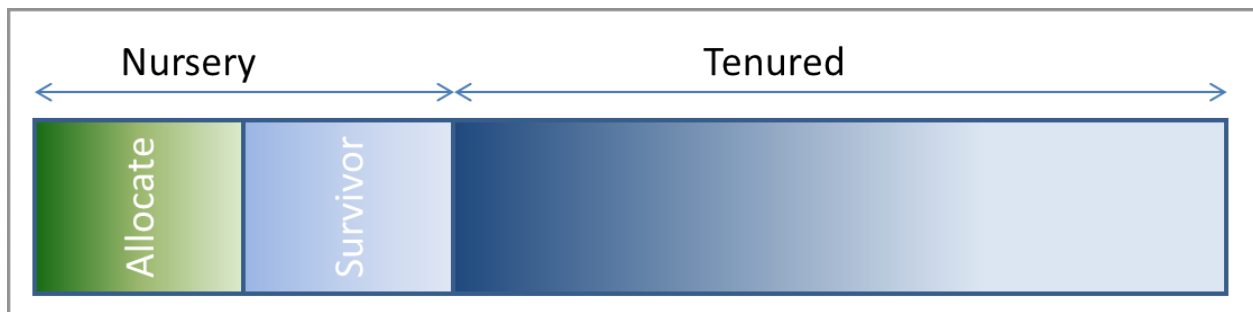


Figure 2.9: When using the generational GC, the IBM WebSphere JVM separates the heap into nursery and tenured spaces. The nursery space is divided into two equal-sized areas for infant and surviving objects.

The layout of the generational heap (see Figure 2.9) is slightly different from that of the Oracle HotSpot JVM. The WebSphere nursery is equivalent to the HotSpot young generation, and the WebSphere tenured space is equivalent to the HotSpot old generation. The nursery is divided into two parts of equal size—allocate and survivor areas. Objects are always allocated in the nursery and copy garbage collection is used to copy surviving objects to the survivor area. After a successful GC cycle, the former survivor area becomes the nursery.

The IBM WebSphere JVM omits the Eden space and does not treat infant objects specially. It does, however, differentiate between [small and large objects](#). Large objects, usually more than 64k, are allocated in a specific area for the non-generational heap or directly in the tenured generation space. The rationale is simple. Copying (generational GC) or moving (compacting) large objects is more expensive than considering them during the marking phase of a normal GC cycle.

And unlike the HotSpot JVM, the WebSphere JVM treats classes like any other object, placing them in the “normal” heap. There is no permanent generation and so classes are subjected to garbage collection every time. Under certain circumstances when classes are repeatedly reloaded, this can lead to performance problems. Examples of this can be found in the [Problem Pattern Section](#) below.

Oracle JRockit

Oracle’s Weblogic Application Server uses the [Oracle JRockit JVM](#), and, like the IBM WebSphere JVM, can use a single continuous heap or generational GC. Unlike the other two other JVMs we’ve discussed, JRockit does not use a copy garbage collection strategy within the nursery. It simply declares a block within the nursery (size is configurable and the placement changes after every GC) as a keep area (see Figure 2.10).

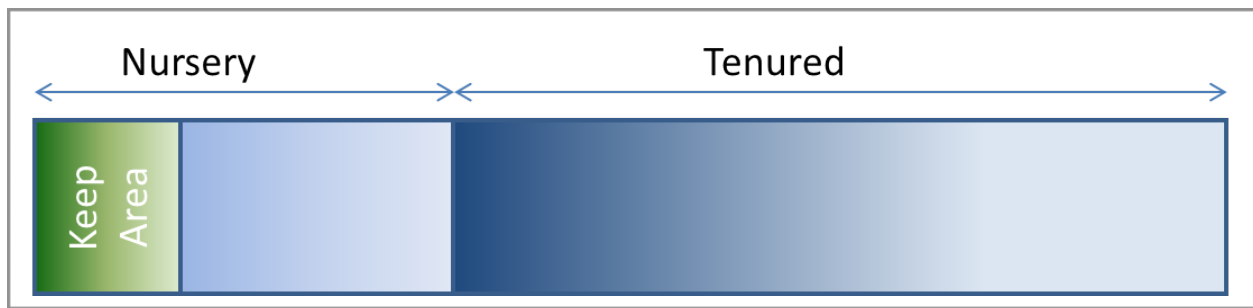


Figure 2.10: The generational GC of the Oracle JRockit JVM does not split its nursery into separate spaces; it attempts to keep infant objects from being treated by the garbage collector at all.

The keep area leads to the following object allocation and garbage collection semantics:

- Objects are first allocated anywhere outside the keep area. Once the nursery fills up, the keep area gets used as well.
- The keep area automatically contains the most-recently allocated objects once the GC is triggered.
- All live objects outside the keep area are promoted to the tenured space. Objects in the keep area are considered alive and left untouched.
- After the GC, the nursery is empty apart from objects within the former keep area. A new equally-sized memory block within the nursery is now declared as a keep area and the cycle starts again.

This is much simpler and more efficient than the typical copy garbage collection, due to these two main points:

- An object is never copied more than once.
- Recently allocated objects are too young to tenure, and most likely are alive and thus simply left untouched.

To avoid unnecessary promotions, the JRockit JVM young generation has to be larger than the corresponding generations in the other two main JVMs. It is important to note that the young-generation GC is a stop-the-world scenario—the more objects get promoted, the longer the application is suspended.

While the JRockit handles the young generation differently, the tenured space is then using the same either a parallel or concurrent GC strategies as the two other JVMs.

The following are some important points that distinguish the JRockit JVM from others:

- Thread-local allocation (TLA) is active by default (which we'll discuss in the [next section of this chapter](#)) and is part of the nursery.
- JRockit distinguishes between small and large objects, with large objects allocated directly to the old generation.
- Classes are considered normal objects and are placed on the heap and subject to garbage collection (which is also true of the IBM WebSphere JVM).

"Special" Garbage-Collection Strategies

There are some situations when standard garbage collection is not sufficient. We will examine the [Remote Garbage Collector](#), which deals with distributed object references, and the [Real Time Garbage Collector](#), which deals with real-time guarantees.

The Remote Garbage Collector

With a Remote Method Invocation (RMI) we can use a local Java object (client-side stub) represent another object residing on a different JVM (server-side). Obviously, RMI calls to the server-side object require that this object exists. Therefore RMI makes it necessary to consider the server-side object being referenced by the client-side stub. Since the server has no way of knowing about this reference, we need remote garbage collection remedies. Here's how it works:

1. When a client receives a stub from a server, it acquires a lease for it. The server-side object is considered referenced by the client stub.
2. A server-side object is kept alive by the RMI implementation itself until the lease expires, which is a simple timeout.
3. Existing client-side stubs execute regular heartbeats (known informally as *dirty* calls) to renew their leases. (This is done automatically by the RMI implementation.)
4. The server side checks periodically for expired leases.
5. Once a lease expires (because no clients exist anymore to reference the object), the RMI implementation simply forgets the object. It can then be garbage-collected like any other object.

Server objects, though no longer used by the client, can therefore survive garbage collection (no client objects are alive). An otherwise-inactive client might hold onto a remote object for a long time even if the object is ready for garbage collection. If the client object is not garbage-collected the server object is held onto as well. In extreme cases, this means that a lot of inactive clients lead to a lot of unused server objects that cannot be garbage-collected. This can effectively crash the server with an out-of-memory error.

To avoid this, the distributed garbage collector (RMI garbage collector) forces a major client garbage collection (with all the negative impact on performance) at regular intervals. This interval is controlled using the [GCInterval system property](#).

The same setting exists for the server side and do the same thing. (Until Java 6, both settings defaulted to a minute. In Java 6, the server-side default changed to one hour.) It effectively triggers a major garbage collection every minute, a performance nightmare. The setting makes sense in general on the client side (to allow the server to remove remote objects), but it's unclear why it exists on the server side. A server remote object is freed for garbage collection either when the lease is up or when the client explicitly cleans it. The explicit garbage collection has no impact on this, which is why I recommend setting this property as high as possible for the server.

I also recommend that RMI be restricted to stateless service interfaces. Since there would exist only one instance of such a server interface and it would never need to be garbage-collected (or at least as long as the application is running), we do not need remote garbage collection to remove it. If we restrict RMI in this way, we can also set the client-side interval very high and effectively remove the distributed garbage collector from our equation by negating its impact on application performance.

Real-Time Garbage Collectors

Real-time systems guarantee nearly instantaneous (in the single-digit-milliseconds range) execution speed for every request being processed. However, the precious time used for Garbage collection runtime suspensions can pose problems, especially since the frequency and duration of GC execution is inherently unpredictable. We can optimize for low pause time, but we can't guarantee a maximum pause time. Thankfully, there are multiple solutions to these problems.

Sun originally specified the [Java Real-Time System](#) (Java RTS; see [JSR-1](#) and [JSR-282](#)) with a specific real-time garbage collector called Henriksson GC that attempts to ensure strict thread scheduling. The algorithm is intended to make sure garbage collection does not occur while critical threads (defined by priority) are executing tasks. However, it is a best-effort algorithm, and there is no way to guarantee that no critical threads are suspended.

In addition, the Java RTS specification includes scoped and immortal memory areas. A scope is defined by marking a specific method as the start of a scoped memory area. All objects allocated during the execution of that method are considered to be part of the scoped memory area. Once the method execution has finished, and thus the scoped memory area is left, all objects allocated within it are simply considered deleted. No actual garbage collection occurs, objects allocated in a scoped memory area are freed, and all used memory is reclaimed immediately after the defined scope has been exited.

Immortal objects, objects allocated via the immortal memory area, are never garbage collected, an enormous advantage. As such, they must never reference scoped objects, which would lead to inconsistencies because the scoped object will be removed without checking for references.

These two capabilities give us a level of memory control that is otherwise not possible in Java, which allows us to minimize the unpredictable impact of the GC on our response time. The disadvantage is that this is not part of the standard JDK, so it requires a small degree of code change and an intrinsic understanding of the application at hand.

The IBM WebSphere and Oracle JRockit JVMs both provide real-time garbage collectors. IBM promotes its real-time garbage collector by guaranteeing ≤ 1 ms pause time. Oracle JRockit provides a [deterministic garbage collector](#), in which the maximum GC pause can be configured. Other JVMs, such as Zing, from Azul Systems, try to solve this issue by completely removing the stop-the-world event from the garbage collector. (There are a number of [Real Time Java implementations](#) available).

Analyzing the Performance Impact of Memory Utilization and Garbage Collection

The goal of any Java memory analysis is to optimize garbage collection (GC) in such a way that its impact on application response time or CPU usage is minimized. It is equally important to ensure the stability of the application. Memory shortages and leaks often lead to instability. To identify memory-caused instability or excessive garbage collection we first need to monitor our application with appropriate tools. If garbage collection has a negative impact on response time, our goal must be to optimize the configuration. The goal of every configuration change must be to decrease that impact. Finally, if configuration changes alone are not enough we must analyze the allocation patterns and memory usage itself. So let's get to it.

Memory-Monitoring Tools

Since Java 5, the standard JDK monitoring tool has been [JConsole](#). The Oracle JDK also includes [jStat](#), which enables the monitoring of memory usage and garbage-collector activity from the console, and [Java VisualVM](#) (or `jvisualvm`), which provides rudimentary memory analyses and a profiler. The Oracle JRockit JDK includes [JRockit Mission Control](#) and the `verbose:gc` flag of the JVM. Each JVM vendor includes its own monitoring tools, and there are numerous commercial tools available that offer additional functionality.

Monitoring Memory Use and GC Activity

Memory shortage is often the cause of instability and unresponsiveness in a Java applications. Consequently, we need to monitor the impact of garbage collection on response time and memory usage to ensure both stability and performance. However, monitoring memory utilization and garbage collection times is not enough, as these two elements alone do not tell us if the application response time is affected by garbage collection. Only GC suspensions affect

response time directly, and a GC can also run concurrent to the application. We therefore need to correlate the suspensions caused by garbage collection with the application's response time. Based on this we need to monitor the following:

- Utilization of the different memory pools (Eden, survivor, and old). Memory shortage is the number-one reason for increased GC activity.
- If overall memory utilization is increasing continuously despite garbage collection, there is a memory leak, which will inevitably lead to an out-of-memory error. In this case, a memory heap analysis is necessary.
- The number of young-generation collections provides information on the churn rate (the rate of object allocations). The higher the number, the more objects are allocated. A high number of young collections can be the cause of a response-time problem and of a growing old generation (because the young generation cannot cope with the quantity of objects anymore).
- If the utilization of the old generation fluctuates greatly without rising after GC, then objects are being copied unnecessarily from the young generation to the old generation. There are three possible reasons for this: the young generation is too small, there's a high churn rate, or there's too much transactional memory usage.
- High GC activity generally has a negative effect on CPU usage. However, only suspensions (aka stop-the-world events) have a direct impact on response time. Contrary to popular opinion, suspensions are not limited to major GCs. It is therefore important to monitor suspensions in correlation to the application response time.

The JVM memory dashboard (Figure 2.11) shows that the tenured (or old) generation is growing continuously, only to drop back to its old level after an old-generation GC (lower right). This means that there is no memory leak and the cause of the growth is prematurely-tenured objects. The young generation is too small to cope with the allocations of the running transactions. This is also indicated by the high number of young-generation GCs (Oracle/Sun Copy GC). These so-called minor GCs are often ignored and thought to have no impact.

The JVM will be suspended for the duration of the minor garbage collection; it's a stop-the-world event. Minor GCs are usually quite fast, which is why they are called minor, but in this case they have a high impact on response time. The root cause is the same as already mentioned: the young generation is too small to cope. It is important to note that it might not be enough to increase the young generation's size. A bigger young generation can accommodate more live objects, which in turn will lead to longer GC cycles. The best optimization is always to reduce the number of allocations and the overall memory requirement.

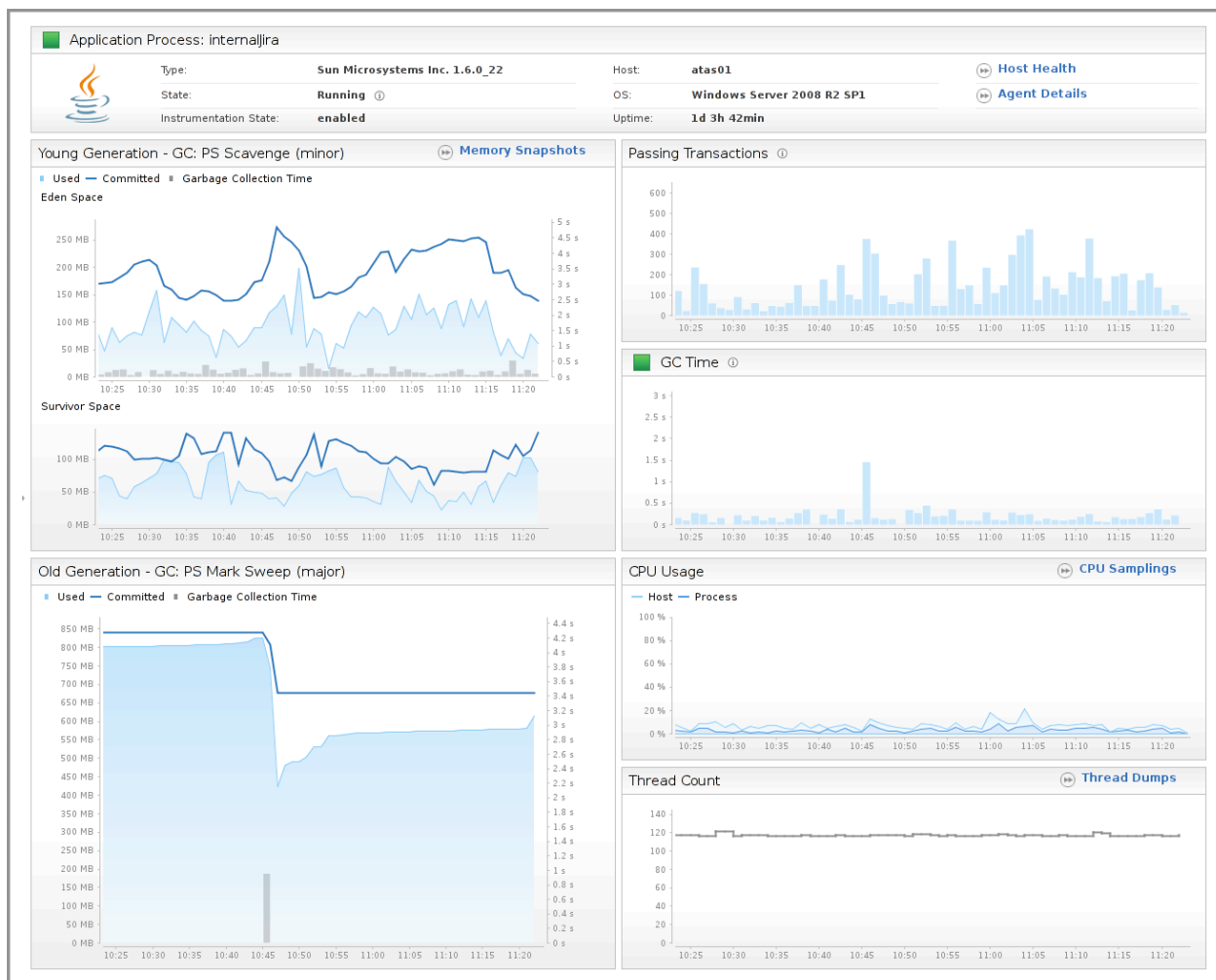


Figure 2.11: This shows how one can visualize JVM memory activity. The GC activity is shown as bars in the respective generations and the actual suspension time is shown separately.

Of course, we cannot avoid GC cycles, and we would not want to. However, we can optimize the configuration to minimize the impact of GC suspensions on response time.

How to Monitor and Interpret the Impact of GC on Response Time

GC suspensions represent the only direct impact on response time by the garbage collector. The only way to monitor this is via the JVM tool interface (JVM TI), which can be used to register callbacks at the start and end of a suspension. During a stop-the-world event, all active transactions are suspended. We can correlate these suspensions to application transactions by identifying both the start and end of the suspension. Effectively, we can tell how long a particular transaction has been suspended by the GC. (See Figure 2.12.)

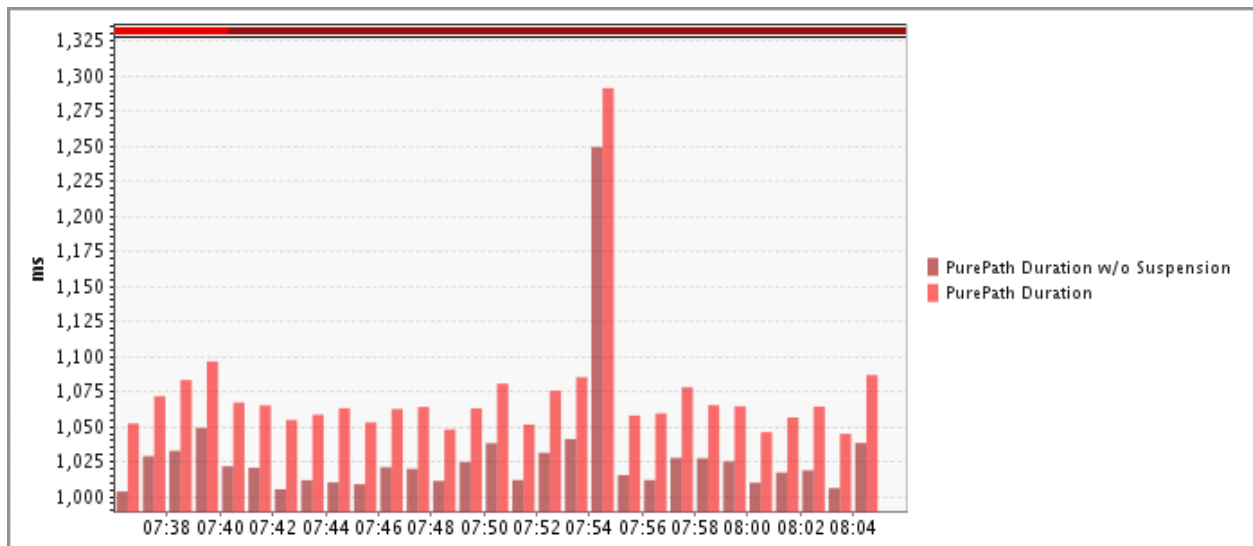


Figure 2.12: The two bars represent the transaction (labeled PurePath in the picture) duration with and without GC suspension. The difference represents the performance impact of garbage collection.

Only a handful of tools allow the direct monitoring of suspensions, dynaTrace being one of them. If you do not have such a tool you can use jStat, JConsole, or a similar tool to monitor GC times. The metrics reported in jStat are also directly exposed by the JVM via JMX. This means you can use any JMX-capable monitoring solution to monitor these metrics.

It is important to differentiate between young- and old-generation GCs (or minor and major, as they are sometimes called; see more in the next section) and equally important to understand both the frequency and the duration of GC cycles. Young-generation GCs will be mostly short in duration but under heavy load can be very frequent. A lot of quick GCs can be as bad for performance as a single long-lasting one (remember that young-generation GCs are always stop-the-world events).

Frequent young-generation GCs have two root causes:

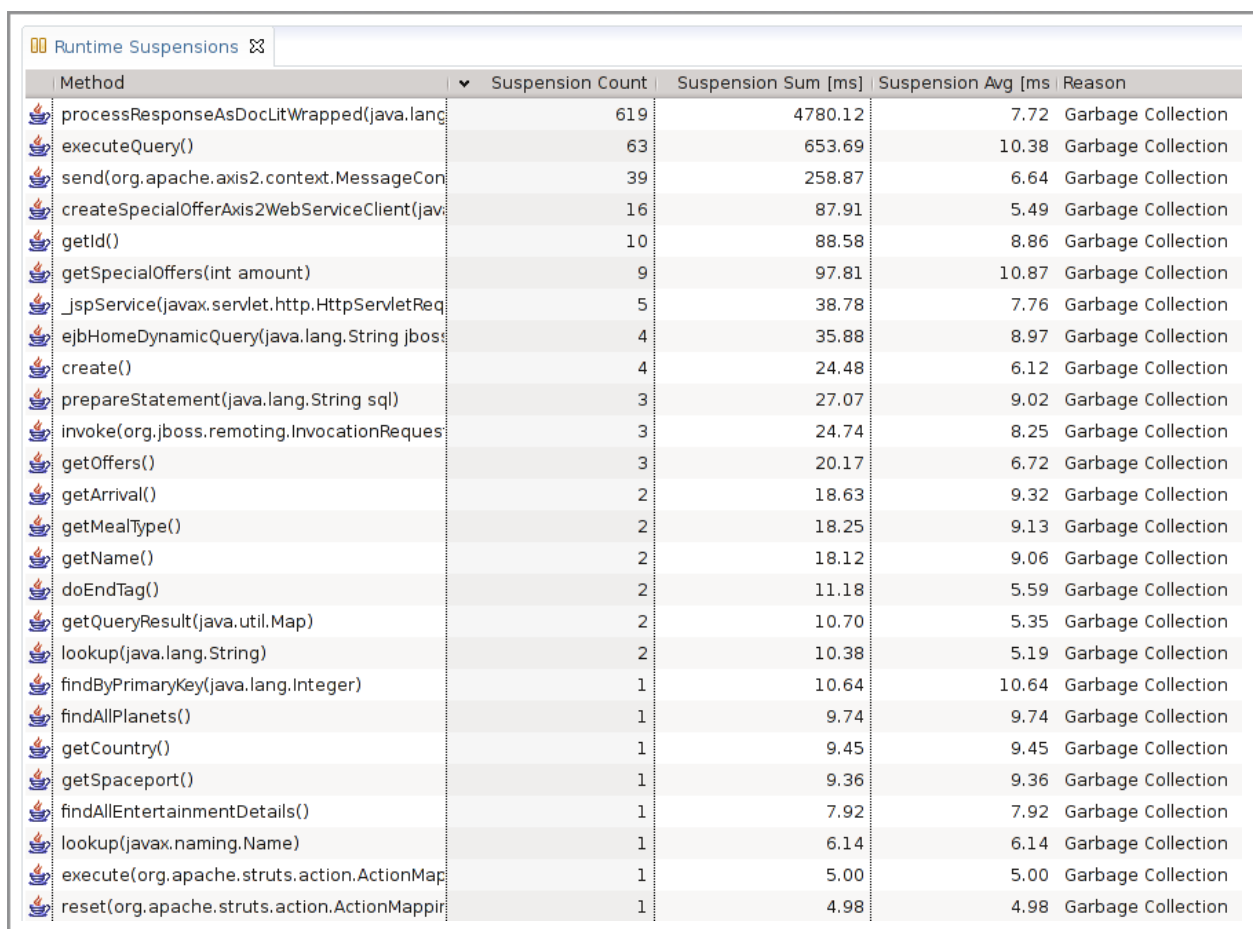
- A too-small young generation for the application load
- High churn rate

If the young generation is too small, we see a growing old generation due to prematurely tenured objects.

If too many objects are allocated too quickly (i.e., if there's a high churn rate), the young generation fills up as quickly and a GC must be triggered. While the GC can still cope without overflowing to the old generation, it has to do so at the expense of the application's performance.

A high churn rate might prevent us from ever achieving an optimal generation sizing, so we must fix such a problem in our code before attempting to optimize garbage collection itself. There is a relatively easy way to identify problematic code areas.

The runtime suspensions shown in Figure 2.13 report a massive statistical concentration of garbage collections in one particular function. This is very unusual, because a GC is not triggered by a particular method, but rather by the fillstate of the heap. The fact that one method is suspended more than others suggests that the method itself allocates enough objects to fill up the young generation and thus triggers a GC. Whenever we see such a statistical anomaly we find a prime candidate for allocation analysis and optimization (which we'll examine further in the Allocation Analysis section of this chapter, below).



Method	Suspension Count	Suspension Sum [ms]	Suspension Avg [ms]	Reason
processResponseAsDocLitWrapped(java.lang...	619	4780.12	7.72	Garbage Collection
executeQuery()	63	653.69	10.38	Garbage Collection
send(org.apache.axis2.context.MessageCon...	39	258.87	6.64	Garbage Collection
createSpecialOfferAxis2WebServiceClient(jav...	16	87.91	5.49	Garbage Collection
getId()	10	88.58	8.86	Garbage Collection
getSpecialOffers(int amount)	9	97.81	10.87	Garbage Collection
_jspService(javax.servlet.http.HttpServletReq...	5	38.78	7.76	Garbage Collection
ejbHomeDynamicQuery(java.lang.String jboss...	4	35.88	8.97	Garbage Collection
create()	4	24.48	6.12	Garbage Collection
prepareStatement(java.lang.String sql)	3	27.07	9.02	Garbage Collection
invoke(org.jboss.remoting.InvocationReques...	3	24.74	8.25	Garbage Collection
getOffers()	3	20.17	6.72	Garbage Collection
getArrival()	2	18.63	9.32	Garbage Collection
getMealType()	2	18.25	9.13	Garbage Collection
getName()	2	18.12	9.06	Garbage Collection
doEndTag()	2	11.18	5.59	Garbage Collection
getQueryResult(java.util.Map)	2	10.70	5.35	Garbage Collection
lookup(java.lang.String)	2	10.38	5.19	Garbage Collection
findByPrimaryKey(java.lang.Integer)	1	10.64	10.64	Garbage Collection
findAllPlanets()	1	9.74	9.74	Garbage Collection
getCountry()	1	9.45	9.45	Garbage Collection
getSpaceport()	1	9.36	9.36	Garbage Collection
findAllEntertainmentDetails()	1	7.92	7.92	Garbage Collection
lookup(javax.naming.Name)	1	6.14	6.14	Garbage Collection
execute(org.apache.struts.action.ActionMap...	1	5.00	5.00	Garbage Collection
reset(org.apache.struts.action.ActionMappir...	1	4.98	4.98	Garbage Collection

Figure 2.13: This figure shows how often and for how long a certain method has been suspended by garbage collection.

Major vs. Minor Garbage Collections

What I have been referring to as young- and old-generation GCs, are commonly referred to as minor and major GCs. Similarly, it is common knowledge that major GCs suspend your JVM, are

bad for performance, and should be avoided if possible. On the other hand, minor GCs are often thought to be of no consequence and are ignored during monitoring. As already explained, minor GCs suspend the application and neither they nor major GCs should be ignored. A major GC is often equated with garbage collection in the old generation; this is, however, not fully correct. While every major GC collects the old generation, not every old-generation GC is a major collection. Consequently, the reason we should minimize or avoid major GCs is misunderstood. Looking at the output of verbose:GC explains what I mean:

```
[GC 325407K->83000K(776768K), 0.2300771 secs] [GC 325816K-
>83372K(776768K), 0.2454258 secs] [Full GC 267628K->83769K(776768K),
1.8479984 secs]
```

A major GC is a full GC. A major GC collects all areas of the heap, including the young generation and, in the case of the Oracle HotSpot JVM, the permanent generation. Furthermore, it does this during a stop-the-world event, meaning the application is suspended for a lengthy amount of time, often several seconds or minutes. On the other hand, we can have a lot of GC activity in the old generation without ever seeing a major (full) GC or having any lengthy GC-caused suspensions. To observe this, simply execute any application with a concurrent GC. Use jstat –gcutil to monitor the GC behavior of your application.

This output is reported by jstat -gcutil. The first five columns in Table 2.1 show the utilization of the different spaces as a percentage—survivor 0, survivor 1, Eden, old, and permanent—followed by the number of GC activations and their accumulated times—young GC activations and time, full GC activations and time. The last column shows the accumulated time used for garbage collection overall. The third row shows that the old generation has shrunk and jStat reported a full GC (see the bold numbers).

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
90.68	0.00	32.44	85.41	45.03	134	15.270	143	21.683	36.954
90.68	0.00	32.45	85.41	45.03	134	15.270	143	21.683	36.954
90.68	0.00	38.23	85.41	45.12	134	15.270	143	21.683	36.954
90.68	0.00	38.52	85.39	45.12	134	15.270	144	21.860	37.131
90.68	0.00	38.77	85.39	45.12	134	15.270	144	21.860	37.131
90.68	0.00	49.78	85.39	45.13	134	15.270	145	21.860	37.131

Table 2.1: jstat -gcutil output report

On the other hand, monitoring the application with JConsole (Figure 2.14), we see no GC runs reported for the old generation, although we see a slightly fluctuating old generation.

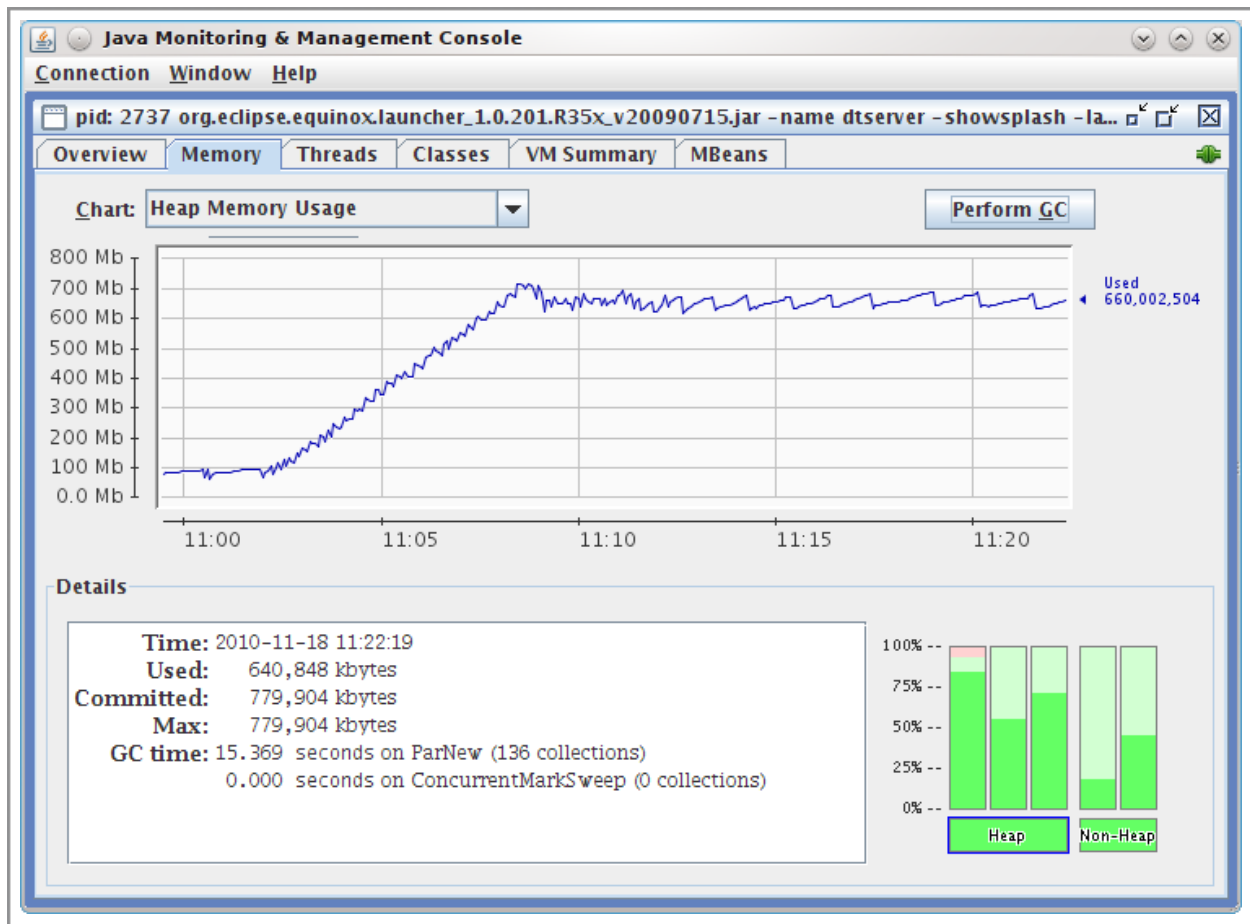


Figure 2.14: JConsole shows that the amount of used memory in the old generation is fluctuating. At the same time it shows that not a single garbage collection was executed in the old generation.

Which of the two tools, jStat (Figure 2.13) or JConsole (Figure 2.14) is correct?

In fact, both are only partially correct, which is rather misleading. As reported by jStat, the concurrent GC has executed, but it has done so asynchronously to the application and for the old generation only, which is its purpose. It was not a full GC! JStat reports old-generation GCs as full GCs, which is wrong. This is most likely a legacy from days prior to concurrent and incremental GCs.

JConsole reports activations via Java Management Extensions (JMX) and managed beans (MBeans). Previously, these MBeans reported only real major GCs. In the case of the CMS, these occur only when the CMS is not able to do its work concurrent to the application, due to memory fragmentation or high churn rate. For this reason, JConsole would not show any activations.

In a recent release the CMS, Memory MBean has been changed to report only activations of the CMS itself; the downside is that we have no way to monitor real major GCs anymore.

In the IBM WebSphere JVM, major and minor GCs are indistinguishable via JConsole. The only way to distinguish between them is by using the `verbose:gc` flag.

For these reasons, we mistakenly ignore minor GCs and overrate old-generation GCs by considering them to be major. The truth is we need to monitor JVM suspensions and understand whether the root causes lie in the young generation, due to object churn, or the old generation, due to wrong sizing or a memory leak.

Analyzing GC for Maximum Throughput

We have discussed the impact of suspensions have on application response time. However, transaction rates or more important for throughput or batch applications. Consequently we do not care about the pause-time impact to a particular transaction, but about the CPU usage and overall suspension duration.

Consider the following. In a response-time application it is not desirable to have a single transaction suspended for 2 seconds. It is, however, acceptable if each transaction is paused for 50 ms. If this application runs for several hours the overall suspension time will, of course, be much more than 2 seconds and use a lot of CPU, but no single transaction will feel this impact! The fact that a throughput application cares only about the overall suspension and CPU usage must be reflected when optimizing the garbage-collection configuration.

To maximize throughput the GC needs to be as efficient as possible, which means executing the GC as quickly as possible. As the whole application is suspended during the GC, we can and should leverage all existing CPUs. At the same time we should minimize the overall resource usage over a longer period of time. The best strategy here is a parallel full GC (in contrast to the incremental design of the CMS).

It is difficult to measure the exact CPU usage of the garbage collection, but there's an intuitive shortcut. We simply examine the overall GC execution time. This can be easily monitored with every available free or commercial tool. The simplest way is to use `jstat -gc 1s`. This will show us the utilization and overall GC execution on a second-by-second basis (see Table 2.2).

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
0.00	21.76	69.60	9.17	75.80	63	2.646	20	9.303	11.949
0.00	21.76	71.24	9.17	75.80	63	2.646	20	9.303	11.949
0.00	21.76	71.25	9.17	75.80	63	2.646	20	9.303	11.949
0.00	21.76	71.26	9.17	75.80	63	2.646	20	9.303	11.949
0.00	21.76	72.90	9.17	75.80	63	2.646	20	9.303	11.949
0.00	21.76	72.92	9.17	75.80	63	2.646	20	9.303	11.949
68.74	0.00	1.00	9.17	76.29	64	2.719	20	9.303	12.022
68.74	0.00	29.97	9.17	76.42	64	2.719	20	9.303	12.022
68.74	0.00	31.94	9.17	76.43	64	2.719	20	9.303	12.022
68.74	0.00	33.42	9.17	76.43	64	2.719	20	9.303	12.022

Table 2.2: Rather than optimizing for response time, we need only look at the overall GC time.

Using graphical tools, the GC times can be viewed in charts and thus correlated more easily with other metrics, such as throughput. In Figure 2.15 we see that while the garbage collector is executed, suspending quite often in all three applications, it consumes the most time in the GoSpaceBackend it this does not seem to have an impact on throughput.

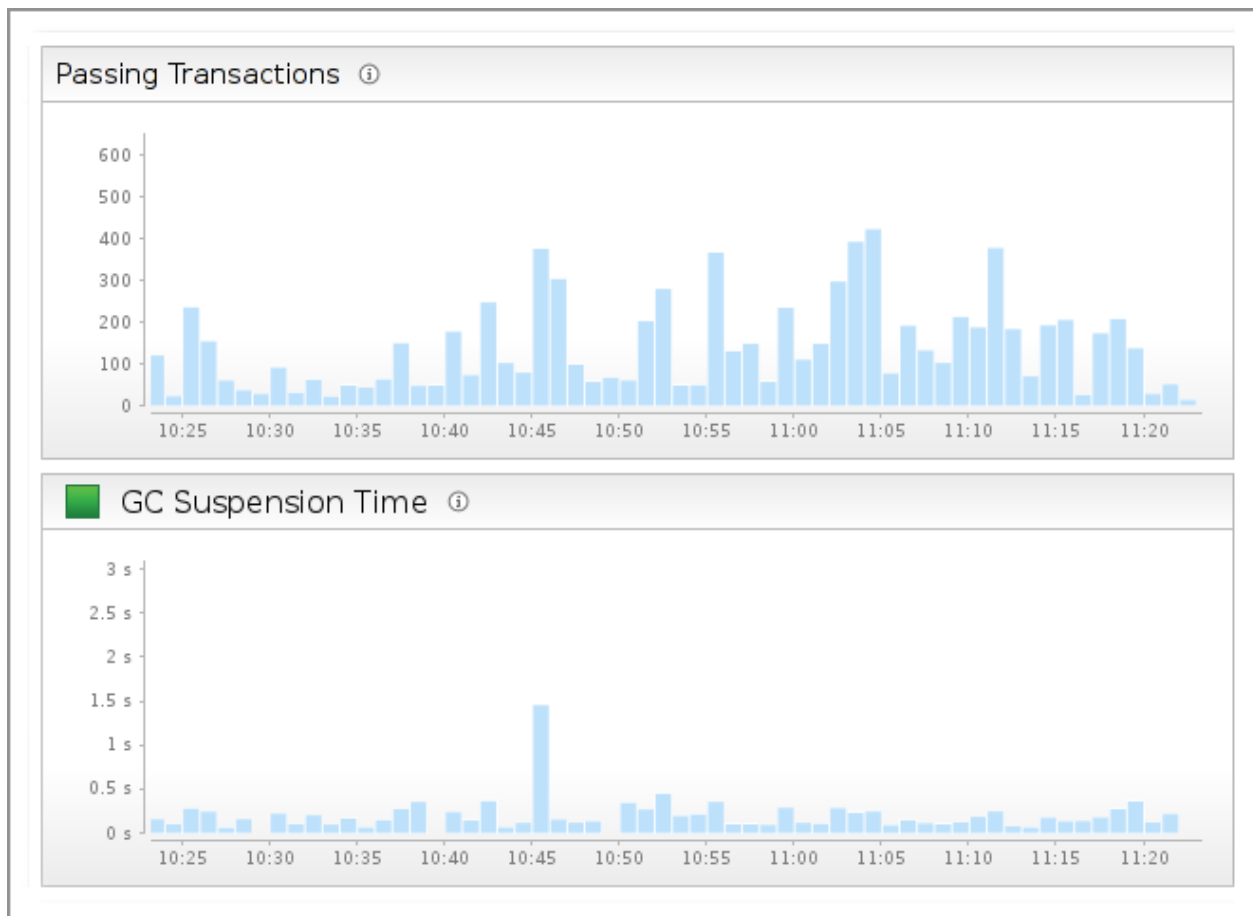


Figure 2.15: The chart shows the duration of GC suspensions of three different JVMs in context to throughput for a particular JVM.

It can be safely assumed that it also requires far more CPU resources than in the other JVMs and has a higher negative impact. With increasing load, the GC times will also rise; this is normal! However, the GC time should never take up more than 10% of the total CPU time used by the application. If CPU usage rises disproportionately to application load, or if the usage is disproportionately heavy, one must consider a change in the GC configuration or an improvement in the application.

Allocation Analysis

Allocating objects in Java is relatively inexpensive compared to C++, for instance. The generational heap is specifically geared for fast and frequent allocations of temporary objects. But it's not free! Allocations can quickly become a choke point when a lot of concurrent threads are involved. Reasons include memory fragmentation, more frequent GCs due to too many allocations, and synchronization due to concurrent allocations. Although you should refrain from implementing object pools for the sole sake of avoiding allocations, not allocating an object is the best optimization.

The key is to optimize your algorithms and avoid making unnecessary allocations or allocating the same object multiple times. For instance, we often create temporary objects in loops. Often it is better to create such an object once prior to the loop and just use it. This might sound trivial, but depending on the size of object and the number of loop recursions, it can have a high impact. And while it might be the obvious thing to do, a lot of existing code does not do this. It is a good rule to allocate such temporary "constants" prior to looping, even if performance is no consideration.

Allocation analysis is a technique of isolating areas of the application that create the most objects and thus provide the highest potential for optimization. There are a number of free tools, such as JvisualVM/JVisualVM, for performing this form of analysis. You should be aware that these tools all have enormous impact on runtime performance and cannot be used under production-type loads. Therefore we need to do this during QA or specific performance-optimization exercises.

It is also important to analyze the correct use cases, meaning it is not enough to analyze just a unit test for the specific code. Most algorithms are data-driven, and so having production-like input data is very important for any kind of performance analysis, but especially for allocation analysis.

Let's look at a specific example. I used JMeter to execute some load tests against one of my applications. Curiously, JMeter could not generate enough load and I quickly found that it suffered from a high level of GC activity. The findings pointed towards a high churn rate (lots of young-generation GCs). Consequently I used an appropriate tool (in my case, JVisualVM) to do some allocation hot-spot analysis. Before I go into the results, I want to note that it is important to start the allocation analysis after the warm-up phase of a problem. A profiler usually shows hotspots irrespective of time or transaction type. We do not want to skew those hot spots with data from the initial warm-up phase (caching and startup). I simply waited a minute after the start of JMeter before activating the allocation analysis. The actual execution timing of the analysis application must be ignored; in most cases the profiler will lead to a dramatic slowdown.

The analysis (Figure 2.16) shows which types of objects are allocated most and where these allocations occur. Of primary interest are the areas that create many or large objects (large

objects create a lot of other objects in their constructor or have a lot of fields). We should also specifically analyze code areas that are known to be massively concurrent under production conditions. Under load, these locations will not only allocate more, but they will also increase synchronization within the memory management itself. In my scenario I found a huge number of Method objects were allocated (Figure 2.16). This is especially suspicious because a Method object is immutable and can be reused easily. There is simply no justification to have so many new Method objects.

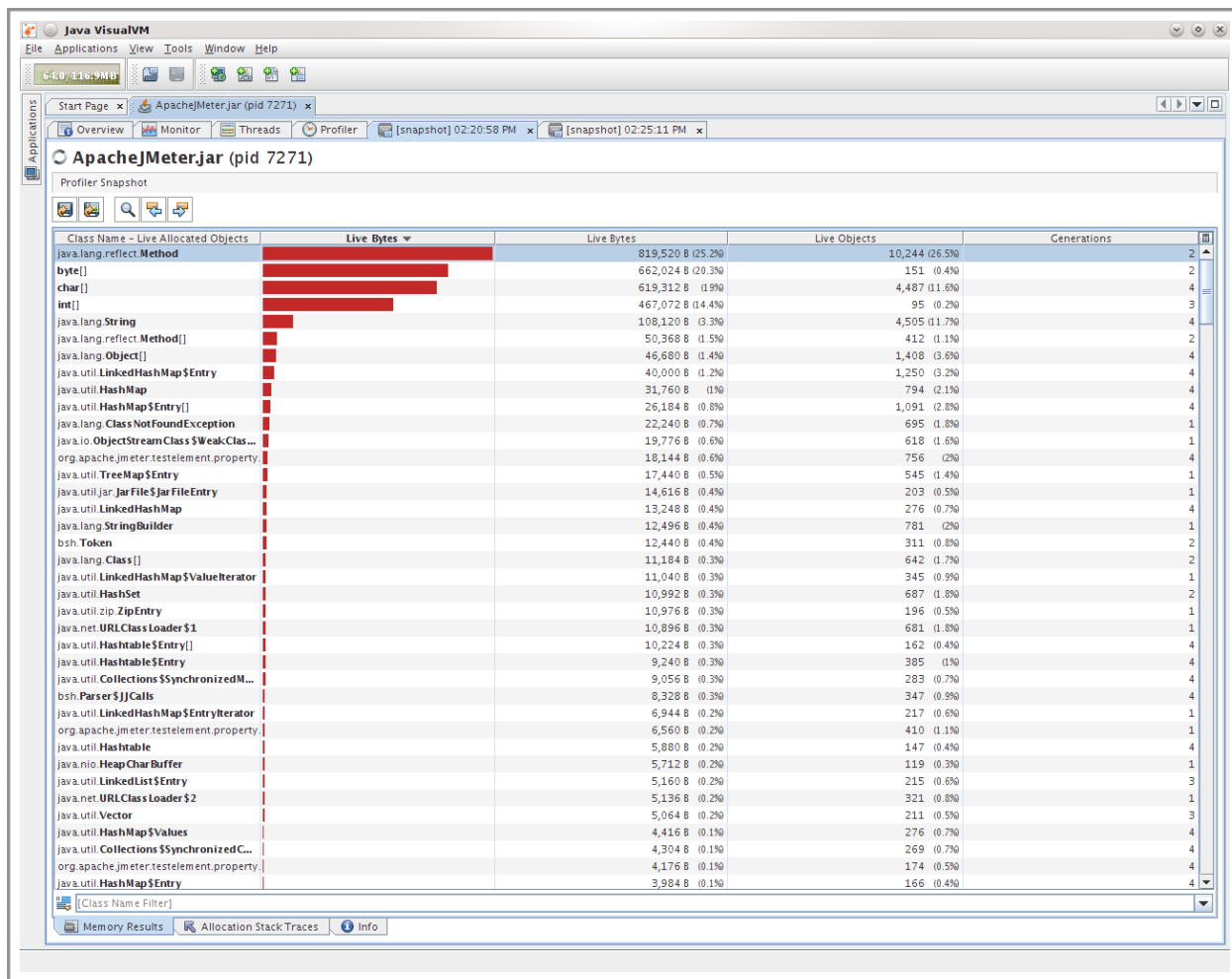


Figure 2.16: JVisualVM tells me that Method objects are the most allocated ones in my application.

By looking at the origin of the allocation (Figure 2.17), I found that the Interpreter object was created every time the script was executed instead of being reused. This led to the allocations of the Method objects. A simple reuse of that Interpreter object removed more than 90% of those allocations.

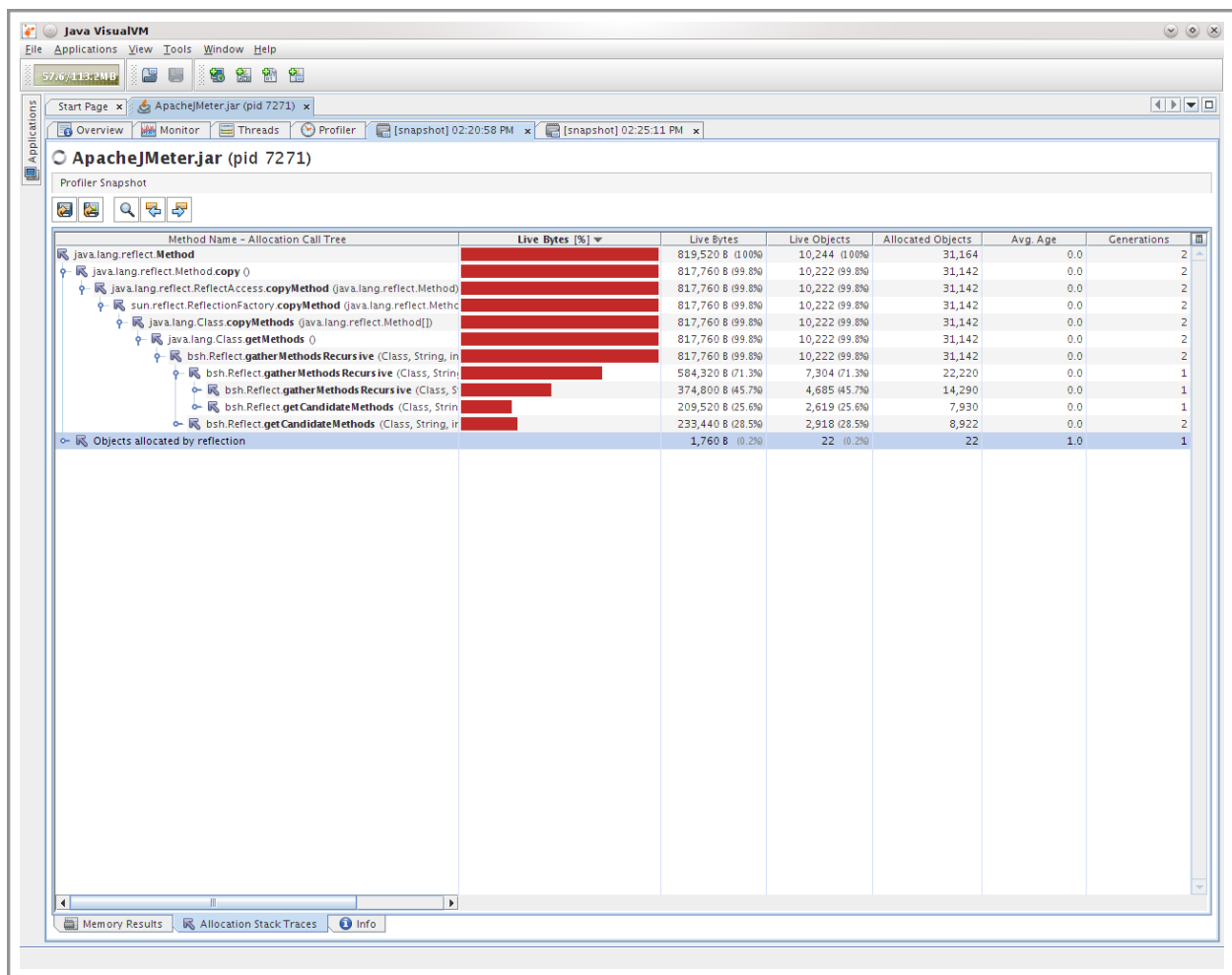


Figure 2.17: JVisualVM tells where the allocations of the method object happen in my application code.

To avoid going quickly astray, it is important to check each optimization for a positive effect. The transaction must be tested before and after the optimization under full load and, naturally, without the profiler, and the results compared against each other. This is the only way to verify the optimization.

Some commercial tools, such as dynaTrace (see Figures 2.18 and 2.19), also offer the ability to test allocations on a transaction basis under load conditions. This has the added advantage of taking into account data-driven problems and concurrency issues, which are otherwise difficult to reproduce in a development setup.

Class	Instance	Allocating Class	Allocating Method	Agent
com.loadtrace.gospace.shared.domain.RoomType	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Spaceport	createWrapper	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Spaceport	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$MealType	parse	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.shared.domain.Ship	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Ship	createWrapper	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$MealType	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Flight	parse	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Ship	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Offer	createWrapper	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.shared.domain.MealType	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$RoomType	parse	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Flight	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Accommodation	createWrapper	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.shared.domain.Spaceport	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	parse	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Offer	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	createWrapper	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.shared.domain.Offer	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	parse	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$RoomType	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	createWrapper	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.shared.domain.Accommodation	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	parse	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Accommodation	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	createWrapper	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.shared.domain.Planet	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	parse	GoSpaceFrontend@gospace-fe2
com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	3900	com.loadtrace.gospace.frontend.webservices.axis2.SpecialOfferServiceStub\$Planet	createWrapper	GoSpaceFrontend@gospace-fe2

Figure 2.18: dynaTrace tells me how often a particular object has been allocated in a load-test scenario.

Method	Class	Argument	Return	Agent	API	Total [ms]
doFilter(ServletRequest request, ServletResponse response, FilterChain chain)	ReplyHeaderFilter			GoSpaceFrontend	Servlet	1149.27
doGet(HttpServletRequest request, HttpServletResponse response)	ActionServlet			GoSpaceFrontend	Struts, Serv	1149.02
selectModule(HttpServletRequest request, ServletContext context)	RequestUtils			GoSpaceFrontend	Struts	0.08
process(HttpServletRequest request, HttpServletResponse response)	RequestProcessor			GoSpaceFrontend	Struts	1148.79
findActionConfig(String path)	ModuleConfigImpl			GoSpaceFrontend	Struts	0.04
createActionForm(HttpServletRequest request, ActionMapping mapping, ModuleConfigImpl)	RequestUtils			GoSpaceFrontend	Struts	0.04
execute(ActionMapping mapping, ActionForm form, HttpServletRequest req, HttpServletResponse res)	MenuAction			GoSpaceFrontend	Struts	1128.70
createSpecialOfferAxis2WebServiceClient(URL url, boolean bottleneckEnabled)	SpecialOfferWebS			GoSpaceFrontend	WebService:	24.12
new SpecialOfferLogic(java.net.URL;java.net.URL;java.net.URL;java.net.URL;boolean)	SpecialOfferLogic	java.net.URL;java.1		GoSpaceFrontend	Memory	-
getOffers()	SpecialOfferAxis2v			GoSpaceFrontend	WebService:	1100.17
send(MessageContext)	AxisEngine			GoSpaceFrontend	Java Web Se	-
send(MessageContext msgContext)	AxisEngine			GoSpaceFrontend	Java Web Se	964.10
new SpecialOfferServiceStub\$GetSpecialOffersResponse()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Offer()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Accommodation()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Flight()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Ship()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Spaceport()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$MealType()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Planet()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$RoomType()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Offer()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Accommodation()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Flight()	SpecialOfferServic			GoSpaceFrontend	Memory	-
new SpecialOfferServiceStub\$Ship()	SpecialOfferServic			GoSpaceFrontend	Memory	-

Figure 2.19: In addition, dynaTrace can tell me where in my transaction the object was allocated.

Memory Analysis—Heap Dump

So far we have dealt with the influence of garbage collection and allocations on application performance. I have stated repeatedly that high memory utilization is a cause for excessive garbage collection. In many cases, hardware restrictions make it impossible to simply increase the heap size of the JVM. In other cases increasing the heap size does not solve but only delays the problem because the utilization just keeps growing. If this is the case, it is time to analyze the memory usage more closely by looking at a heap dump.

A heap dump is a snapshot of main memory. It can be created via JVM functions or by using special tools that utilize the JVM TI. Unfortunately, JVM-integrated heap dumps are not standardized. Within the Oracle HotSpot JVM, one can create and analyze heap dumps using [jmap](#), [jhat](#), and [VisualVM](#). JRockit includes its [JRockit Mission Control](#), which offers a number of functions besides memory analysis itself.

The heap dump itself contains a wealth of information, but it's precisely this wealth that makes analysis hard—you get drowned easily in the quantity of data. Two things are important when analyzing heap dumps: a good Tool and the right technique. In general, the following analyses are possible:

- Identification of memory leaks
- Identification of memory-eaters

Identifying Memory Leaks

Every object that is no longer needed but remains referenced by the application can be considered a memory leak. Practically we care only about memory leaks that are growing or that occupy a lot of memory. A typical memory leak is one in which a specific object type is created repeatedly but not garbage-collected (e.g., because it is kept in a growing list or as part of an object tree). To identify this object type, multiple heap dumps are needed, which can be compared using trending dumps.

Trending dumps only dump a statistic of the number of instances on the class level and can therefore be provided very quickly by the JVM. A comparison shows which object types have a strongly rising tendency. Every Java application has a large number amount of String, char[], and other Java-standard objects. In fact, char[] and String will typically have the highest number of instances, but analyzing them would take you nowhere. Even if we were leaking String objects, it would most likely be because they are referenced by an application object, which represents the root cause of the leak. Therefore concentrating on classes of our own application will yield faster results.

Figure 2.20 shows a view on multiple trending dumps that I took from a sample application I suspected of having a memory leak. I ignored all standard Java objects and filtered by my application package. I immediately found an object type that constantly increased in number of instances. As the object in question was also known to my developer, we knew immediately that we had our memory leak.

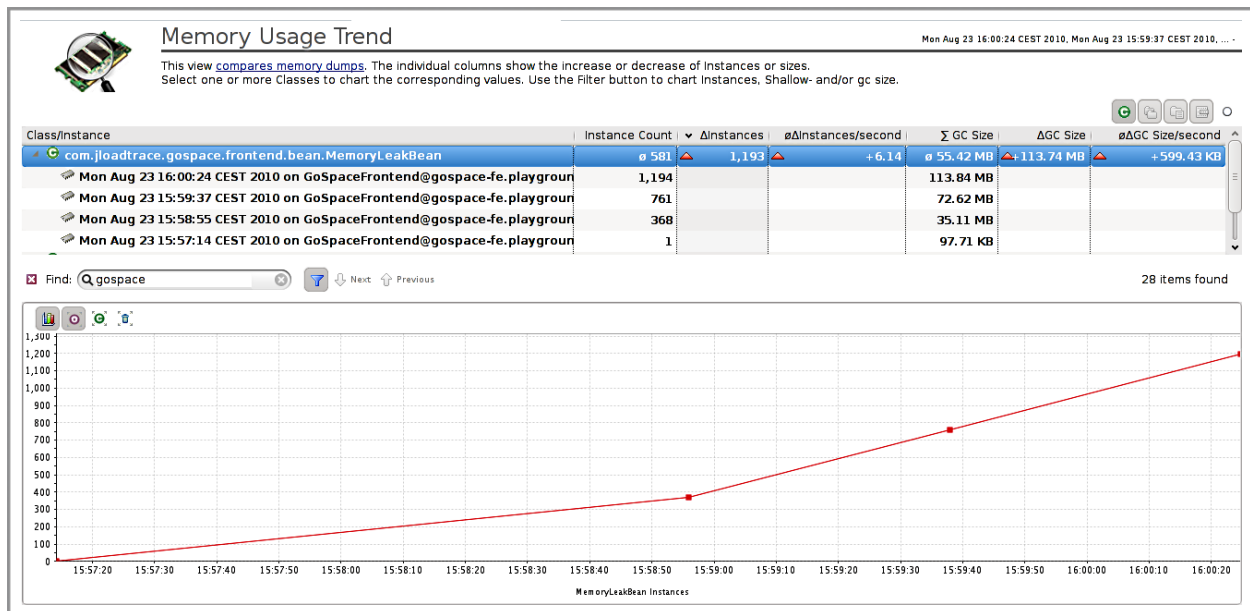


Figure 2.20: The trending dump shows which objects increase in number over time.

Identifying Memory-Eaters

There are several cases in which you want to do a detailed memory analysis.

- The Trending Analysis did not lead you to your memory leak.
- Your application uses too much memory but has no obvious memory leak, and you need to optimize.
- You could not do a trending analysis because memory is growing too fast and your JVM is crashing.

In all three cases the root cause is most likely one or more objects that are at the root of a larger object tree. These objects prevent a whole lot of other objects in the tree from being garbage-collected. In case of an out-of-memory error it is likely that a handful of objects prevent a large number of objects from being freed, hence triggering the out-of-memory error. The purpose of a heap-dump analysis is to find that handful of root objects.

The size of the heap is often a big problem for a memory analysis. Generating a heap dump requires memory itself. If the heap size is at the limit of what is available or possible (32-bit JVMs cannot allocate more than 3.5 GB), the JVM might not be able to generate one. In addition, a heap dump will suspend the JVM. Depending on the size, this can take several minutes or more. It should not be done under load.

Once we have the heap dump we need to analyze it. It is relatively easy to find a big collection. However, manually finding the one object that prevents a whole object tree (or network) from being garbage-collected quickly becomes the proverbial needle in a haystack.

Fortunately solutions like dynaTrace are able to identify these objects automatically. To do this they use a dominator algorithm that stems from graph theory. This algorithm is able to calculate the root of an object tree. In addition to calculating object-tree roots, the memory-analysis tool calculates how much memory a particular tree holds. This way it can calculate which objects prevent a large amount of memory from being freed—in other words, which object dominates memory (see Figure 2.21).

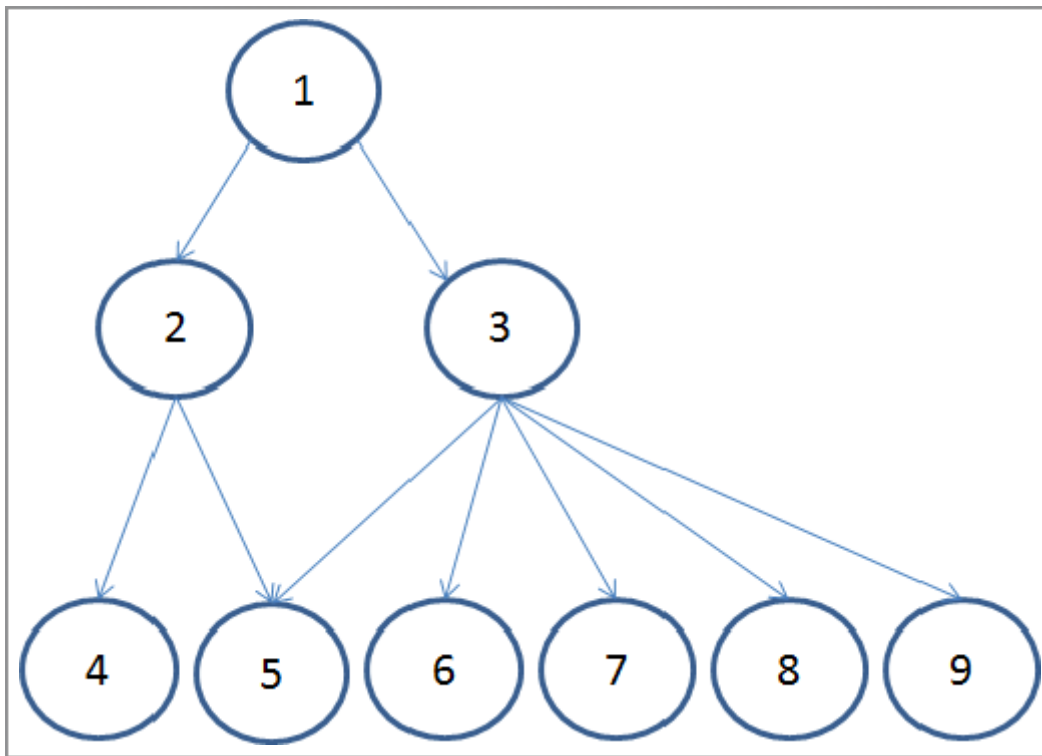


Figure 2.21: This is a schematic representation of an object tree. In this presentation object 1 is the root of the tree and thus dominates all the others.

Let us examine this using the example in Figure 2.21. Object 1 is the root of the displayed object tree. If object 1 no longer references object 3, all objects that would be kept alive by 3 are garbage-collected. In this sense, object 3 dominates objects 6, 7, 8, and 9. Objects 4 and 5 cannot be garbage-collected because they are still referenced by one other (object 2). However, if object 1 is dereferenced, all the objects shown here will be garbage-collected. Therefore, object 1 dominates them all.

A good tool can calculate the dynamic size of object 1 by totaling the sizes of all the objects it dominates. Once this has been done, it is easy to show which objects dominate the memory. In Figure 2.21, objects 1 and 3 will be the largest hot spots. A practical way of using this is in a memory hotspot view as we have in Figure 2.22:

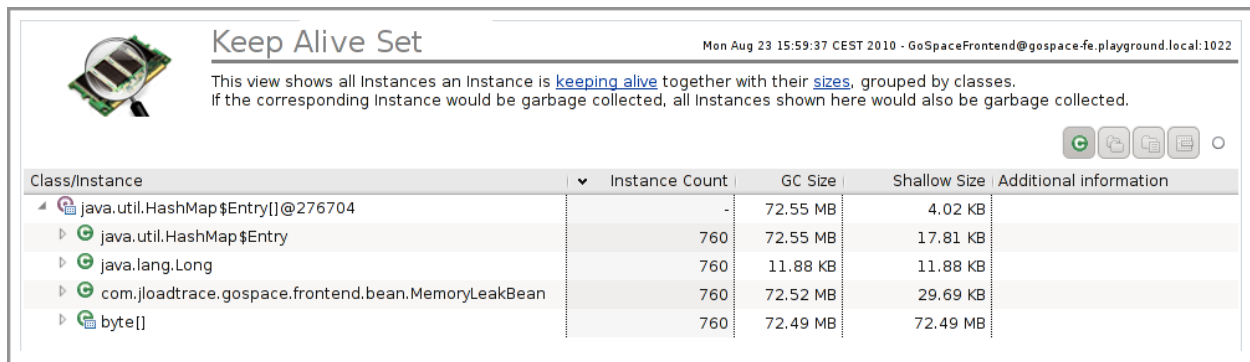


Figure 2.22: This hotspot view shows that the `HashMap$Entry` array dominates, with more than 80 percent of the heap.

The `HashMap$Entry` array dominates—it accounts for more than 80 percent of the heap. Were it to be garbage-collected, 80% of our heap would be empty. From here it is a relative simple matter to use the heap dump to figure out why. We can then easily understand and show why that object itself has not been garbage-collected. Even more useful is to know which objects are being dominated and make up that hot spot. In dynaTrace we call this the Keep Alive Set (see Figure 2.23).

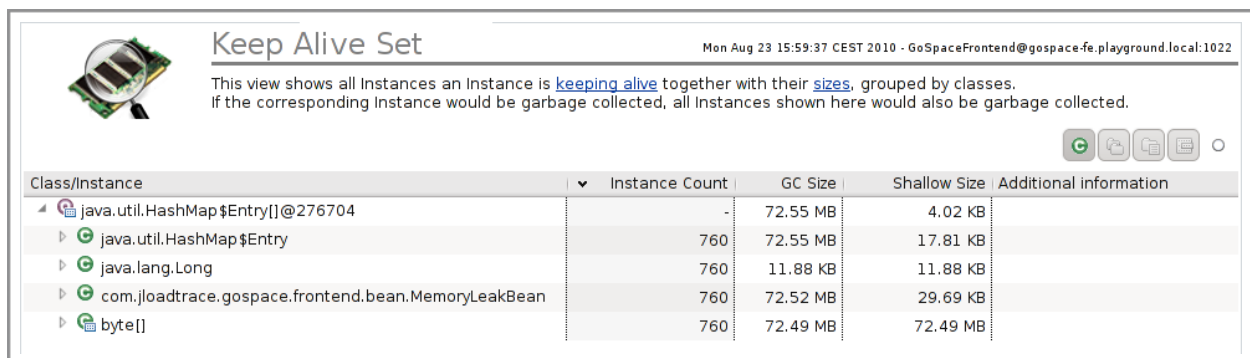


Figure 2.23: The keep alive set shows the objects that are being dominated or kept alive by the `HashMap$Entry` Array.

With the aid of such analyses, it is possible to determine within a few minutes whether there is a large memory leak or whether the cache uses too much memory.

Tuning

Sizing the young generation is the most important tuning step in a generational GC setup. First, it must be large enough to handle all threads of the concurrent working set without tenuring short-lived objects to the old generation. Second, we want long-lived objects tenured as quickly as possible. Remember, the more objects remain live, the more expensive the GC cycle!

Young-Generation Sizing

We're trying to achieve an efficient balance between two extremes, and we'll start by defining a large enough young generation to prevent the old generation growing too fast under load (at least for the 90% of the time that matters most).

For the Oracle HotSpot JVM, try using a throughput collector with adaptive sizing:

- `-XX:+UseAdaptiveSizePolicy`
- Pause Goal: `-XX:MaxGCPauseMillis`
- Throughput Goal: `-XX:GCTimeRatio`

You need to optimize for the desired survivor size, so keep an eye on the survivor spaces. After a young-generation GC, one of the two survivor spaces should not be filled more than 75%. At the same time, the old generation should not grow. If adaptive sizing isn't working, adjust the young generation's size manually.

The IBM WebSphere JVM and Oracle JRockit need to be sized manually. The IBM WebSphere JVM has two areas in the young generation instead of three, but the same target is still valid. The survivor area should not be filled more than 75% after a young-generation GC and the old generation should not grow.

Next, we need to assure that there are not so many live objects in the young generation that a minor GC becomes expensive. For the generational GC, middle-lived objects can be problematic. If they stick around too long in the young generation, they lead to long GC cycles. If we tenure them, the old generation grows, eventually bringing about its own excessive GCs.

By monitoring garbage collections under full load we can analyze the situation and find out what's going on. If young-generation GCs become too expensive while the old generation does not grow, we have too many live objects in the young generation. In the HotSpot JVM we can tune the tenure threshold, which defines how often an object must survive a GC cycle before it gets tenured. This enables us to tenure objects sooner while making sure that temporary objects die in the young generation. The effect should be a faster young-generation GC and a slightly growing old generation. In the IBM WebSphere and JRockit JVMs, all we can do is shrink the young generation by resizing.

If we have to accept a spillover of middle-lived objects to the old generation, we should use the CMS old-generation GC to deal with it. However if that spillover becomes too great we will experience very expensive major GCs. In such a case GC tuning alone does not suffice anymore and we need to optimize the transactional memory usage of the application with an allocation analysis.

Old-Generation Tuning

Once you've achieved near perfection in the young-generation size, you know the old generation isn't going to grow under load. This makes it relatively easy to discover the optimal size of the old generation. Check your application's utilization of the old generation after the initial warm-up phase and note the value. Reconfigure your JVM so that the old generation's size is 25% bigger than this observed value to serve as a buffer. For the Oracle HotSpot JVM, you must size the buffer to be at least as big as Eden plus one survivor to accommodate [HotSpot's young-generation guarantee](#) (check the HotSpot documentation for further details).

If you were unable to achieve optimal young-generation sizing—sometimes it simply is not possible—things become more complicated.

If your application is response time-oriented, you will want to use a concurrent GC in the old generation. In this case you need to tune both the concurrent GC thresholds and the size of the old generation so that the average fill state is never higher than 75%. Again, the 25% extra represents headroom needed for the concurrent GC. If the old generation fills up too much, the CMS will not be able to free up memory fast enough. In addition, fragmentation can easily lead to allocation errors. Both cases will trigger a real major GC, stopping the entire JVM.

If your application is throughput-oriented, things are not as bad. Use a parallel GC with compaction and make the old generation big enough to accommodate all concurrent running transactions. Without this, the GC will not free sufficient memory to support this level of application concurrency. If necessary, it's okay to expand the headroom a bit, keeping in mind that more memory only delays a GC—it cannot prevent it.

GC-Configuration Problems

The Problems:

1. [Incorrect GC Strategy and Configuration](#)
 2. [Java Memory Leaks](#)
 3. [Excessive Memory Use](#)
 4. [ClassLoader-Related Memory Issues](#)
 5. [Out-of-Memory Errors](#)
 6. [Churn Rate and High Transactional Memory Usage](#)
 7. [Incorrect Implementation of Equals and Hashcode](#)
-

Incorrect GC Strategy and Configuration

Often applications rely on the default settings and behavior of the JVM for garbage collection. Unfortunately while some JVMs, like Oracle JRockit, attempt to dynamically deploy the best strategy, they cannot detect whether your application cares more about response time or throughput. Default garbage-collection settings are a performance compromise to make sure that most applications will work, but such settings are not optimized for either type of application. Don't expect many of your applications to run their fastest and jump their highest without some careful testing and analysis on your part.

Performance of both response time-bound and throughput-oriented applications suffers from frequent garbage collection; however, the strategies for fixing this problem are different for the two kinds of application. Determining the correct strategy and sizing will allow you to remedy the majority of GC-related performance problems, so here are some tools and tactics for finding optimal solutions to the most common GC issues:

- To optimize for response time, use the GC suspensions monitor (or activations monitor if it's all that's available). Always use a parallel young-generation collector to ensure short minor GCs (remember they still suspend everything). This will ensure that not more than a few temporary objects get tenured to the old generation. If you cannot prevent an overflow to the old generation, you might require concurrent old-generation garbage collection to avoid longer suspensions (See the [Tuning section](#) earlier in this chapter).
- To optimize for throughput, it is important to make the GC as fast as possible. We've discussed how longer suspensions due to high concurrency lead to exponentially worse throughput. Utilizing all of the CPUs in parallel will usually yield the shortest suspensions, which is why you want to choose a parallel collector for the young and the old generation. In case you have a lot of CPUs and the old-generation GCs still take too long (several hundred milliseconds) you might need to switch to an incremental or even a concurrent GC to avoid the negative effect that long suspensions have on highly concurrent applications.
- Massively concurrent applications can be burdened by the fact that allocations need to be synchronized within the JVM. One can turn on thread-local allocation to remedy this (see the [Making Garbage Collection Faster section](#) earlier in this chapter). Thread-local allocation should not be confused with thread-local variables; the former is a JVM option to improve allocation concurrency. While this can help a lot, reducing the number of allocations will improve things even more. To achieve this we need to do an allocation analysis (see the [Analyzing Performance Impact section](#) earlier in this chapter).

When your application has a more variable load pattern, it's quite difficult to achieve proper young-generation sizing. There are a couple of workarounds:

- Use adaptive sizing and let the JVM size the young and the old generation. In many cases, this will take care of the matter.

- The Oracle HotSpot JVM allows you to test the garbage first (G1) GC, a tool specifically designed for applications with a variable load pattern. Instead of having one young and one old generation, the G1 splits the heap into many smaller areas and dynamically assigns those spaces the duty of being a young or old generation. This way it can more easily adapt to a changing load behavior.

Frequent Major Garbage Collections

Suspending your JVM for a Major GC simply takes longer than for any other GC cycle. Therefore, frequent major GCs can quickly become a big performance problem.

Causes and Solutions

Most applications experience memory constraints from improper young-generation sizing, which results in premature object tenuring (See the [Tuning section](#) earlier in this chapter).

General memory constraints are just as often at the root of this problem. Increasing the total memory allocation for the JVM is the usual remedy. However, if the problem persists and you've already assigned several gigabytes of memory, the cause is more likely a memory leak, which can lead to an ever-growing old generation and, eventually, an out-of-memory situation.

Summary

Incorrect or non-optimized garbage-collection configurations are the most common cause for GC-related performance problems. These are also the easiest to fix, since they do not require any code changes. Memory leaks, as you will see in the next chapter, are a very different story.

Java Memory Leaks

We're all familiar with the ever-growing memory leak, the sort where excess objects accumulate until all available memory is filled. This sort of problem is easy to track down with simple trending or histogram dumps. Figure 2.24 shows an example of the dynaTrace trending-dump facility, but you can achieve similar results manually by using `jmap -histo` multiple times and then comparing the results. If your results show ever-growing instance numbers, you've found your leak.

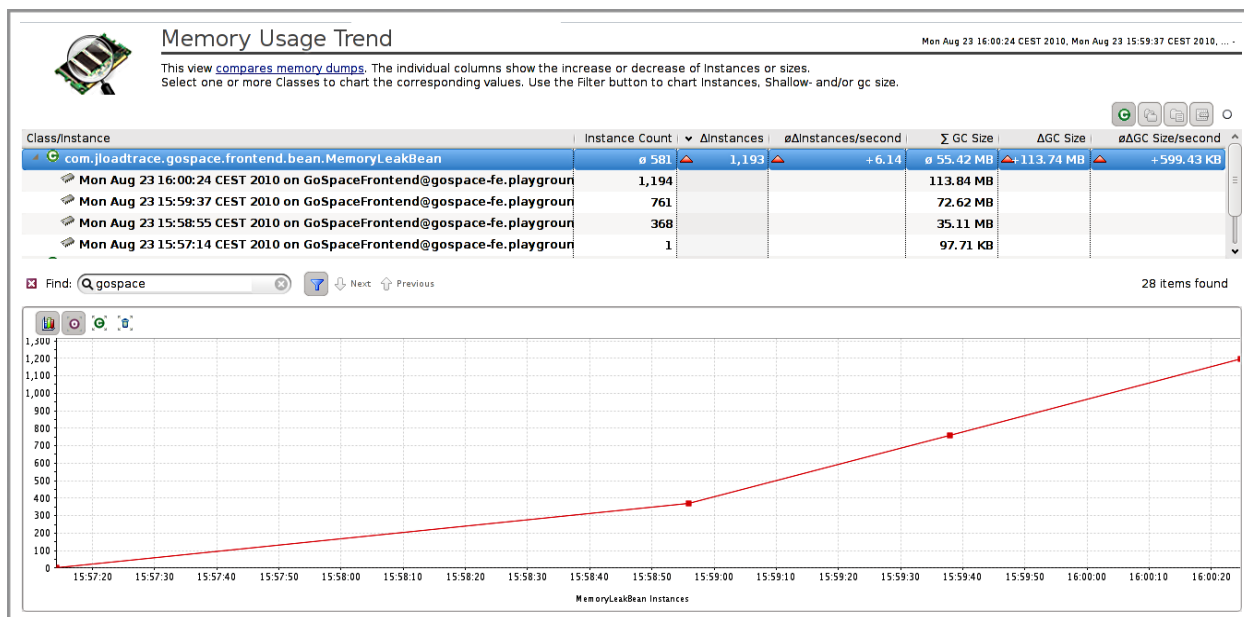


Figure 2.24: The trending dump shows which objects increase in number over time.

On the other hand, a non-growing or slow-growing memory leak is easily overlooked because it can seem so minor as to be unimportant. However, when such a leak is triggered by a single large object structure, the missing memory can pose a significant threat to the stability of the application. By using a modern heap-analyzer tool, we can track down this type of leak fairly quickly as is visualized in figure 2.25 (see the [Analyzing Performance Impact](#) section earlier in this chapter).

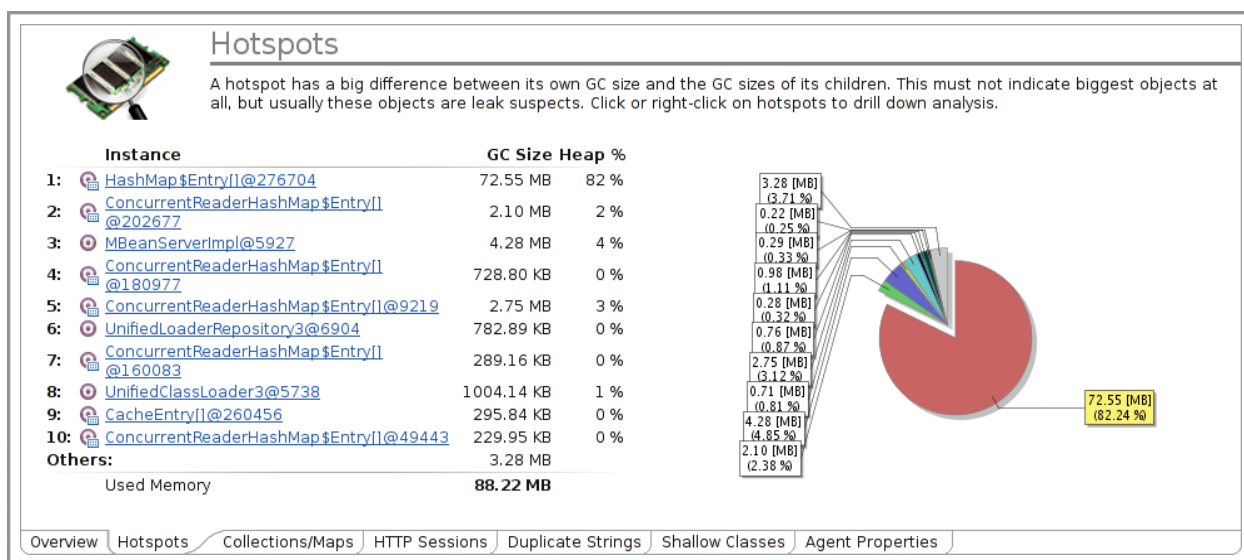


Figure 2.25: This hotspot view shows that the `HashMap$Entry` Array dominates over 80 percent of the heap and is most likely responsible for our memory problems.

It's also possible to have many small, unrelated memory leaks in a single application. This situation is rarely seen, because it can be caused only by some seriously bad programming work. It's as if the leaking objects were rats leaving a sinking ship. But let's not worry too much about such unlikely occurrences, and instead let's concentrate on the most common causes for memory leaks.

Mutable Static Fields and Collections

Static fields are de facto GC roots (see the [How Garbage Collection Works section](#) earlier in this chapter), which means they are never garbage-collected! For convenience alone, static fields and collections are often used to hold caches or share state across threads. Mutable static fields need to be cleaned up explicitly. If the developer does not consider every possibility (a near certainty), the cleanup will not take place, resulting in a memory leak. This sort of careless programming means that static fields and collections have become the most common cause of memory leaks!


In short, never use mutable static fields—use only constants. If you think you need mutable static fields, think about it again, and then again! There's always a more appropriate technique.

Thread-Local Variables

A thread-local variable is basically a member field in the Thread class. In a multithreaded application, every thread instance has its own instance of such a variable. Therefore it can be very useful to bind a state to a thread. But this can be dangerous because thread-local variables will not be removed by the garbage collector as long as the thread itself is alive. As threads are often pooled and thus kept alive virtually forever, the object might never be removed by the garbage collector!

Since an active thread must be considered a GC root, a thread-local variable is very similar to a static variable. The developer, more specifically his code, needs to explicitly clean it up, which goes against the idea of a garbage collector. Just like a mutable static variable, it should be avoided unless there are very few good reasons to use it.

These kinds of memory leaks can be discovered with a heap dump. Just take a look at the ThreadLocalMap in the heap dump and follow the references (see Figure 2.26). Also look at the name of the thread to figure out which part of your application is responsible for the leak.



References by Keep Alive

This view shows the keep alive references of an instance or, for classes, the accumulated references.
[See Documentation for further information.](#)

Class/Instance	Referenced Instance	GC Size
java.lang.Thread	99	10.01 MB
java.lang.ThreadLocal\$ThreadLocalMap	160	9.97 MB
java.lang.ThreadLocal\$ThreadLocalMap\$Entry[]	160	9.96 MB
java.lang.ThreadLocal\$ThreadLocalMap\$Entry	4,784	9.91 MB
org.apache.lucene.index.SegmentTermEnum	1,065	8.21 MB
org.apache.jasper.runtime.JspFactoryImpl\$PageContextPool	47	767.79 KB
com.dynatrace.diagnostics.agent.TraceTag[]	80	326.09 KB
org.apache.xerces.jaxp.DocumentBuilderImpl	5	120.47 KB
java.util.HashSet	741	98.41 KB
org.apache.catalina.core.ApplicationContext\$DispatchData	47	64.99 KB
org.dom4j.DocumentFactory	24	62.11 KB
java.util.HashMap	416	53.02 KB
com.opensymphony.xwork.ActionContext	47	23.73 KB
java.util.LinkedList	100	23.28 KB
java.util.GregorianCalendar	49	20.29 KB
java.util.Hashtable	70	9.28 KB

Figure 2.26: The heap dump shows more than 4K objects, which amount to about 10 MB, held by thread-locals.

Circular and Complex Bi-directional References

Memory leaks due to overly complex and circular object structures are my personal favorites, because it's as if they've got their own personalities. The developer seems to be trying to trick the garbage collector so that it can't do its job correctly.

Let me explain this particular problem by example:

```
org.w3c.dom.Document doc = readXmlDocument();
org.w3c.dom.Node child = doc.getDocumentElement().getFirstChild();
doc.removeNode(child);
doc = null;
```

The first code line reads an XML document from somewhere. The subsequent code, however, does not need the whole document, but rather just a specific portion (e.g. just the first child element). I know that the document holds a reference to the child document, so I remove it (removeNode) to allow the garbage collector to do its work. At the end of the code snippet one might think the DOM document will be garbage-collected; this is not the case!

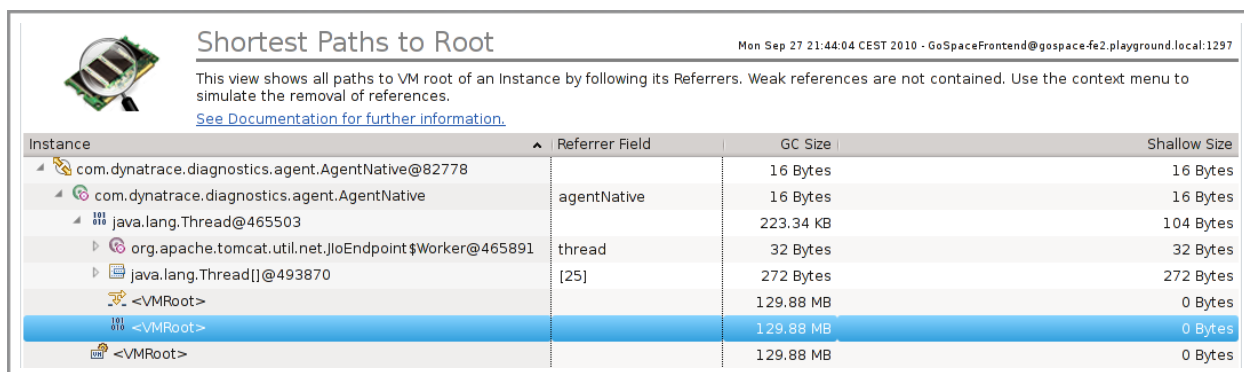
A DOM *Node* object always *belongs* to a DOM *Document*. Even when *removed* from the document the node object retains a reference to its owning document. As long as we keep that child object, neither the document nor any of the nodes it refers to will be removed.

I've seen this and similar cases quite often, and they are rather hard to track down. If you do not know the code in question by heart (which allows you to see the problem just by reading the code), you need to do a heap analysis and figure out why the document object has not been garbage-collected.

JNI Memory Leaks

Java Native Interface (JNI) memory leaks are particularly nasty and hard to find. JNI is used to call native code from Java. This native code can handle, call, and also *create* Java objects. Every Java object created in a native method begins its life as a *local reference*, which means that the object is referenced until the native method returns. We could say the native method references the Java object, so you won't have a problem unless the native method runs forever. In some cases you want to keep the created object even after the native call has ended. To achieve this you can either ensure that it is referenced by some other Java object or you can change the local reference into a *global reference*. A global reference is a GC root and will never be removed until it is explicitly freed by the native code (see Figure 2.27).

The only way to discover JNI memory leaks is to use a heap-dump tool that explicitly marks native references. If possible, you should not use any global references. It's better to assign the desired object to the field of a normal Java class.



Shortest Paths to Root Mon Sep 27 21:44:04 CEST 2010 - GoSpaceFrontend@gospace-fe2.playground.local:1297

This view shows all paths to VM root of an Instance by following its Referrers. Weak references are not contained. Use the context menu to simulate the removal of references.
[See Documentation for further information.](#)

Instance	Referrer Field	GC Size	Shallow Size
com.dynatrace.diagnostics.agent.AgentNative@82778		16 Bytes	16 Bytes
com.dynatrace.diagnostics.agent.AgentNative	agentNative	16 Bytes	16 Bytes
java.lang.Thread@465503		223.34 KB	104 Bytes
org.apache.tomcat.util.net.JIoEndpoint\$Worker@465891	thread	32 Bytes	32 Bytes
java.lang.Thread[]@493870	[25]	272 Bytes	272 Bytes
<VMRoot>		129.88 MB	0 Bytes
<VMRoot>		129.88 MB	0 Bytes
<VMRoot>		129.88 MB	0 Bytes

Figure 2.27: The highlighted line indicates that the object is kept alive due to a JNI global reference.

Excessive Memory Use

Even though an average server might have 16 GB or more memory, excessive memory usage in enterprise applications has become an increasingly frequent and critical problem. For one thing, a high degree of parallelism and a lack of awareness on the part of the developer can quickly lead to memory shortages. At other times, there may be sufficient memory available, but with JVMs using gigabytes of memory, GC pauses can be unacceptably long.

Increasing memory is the obvious workaround for memory leaks or badly written software, but done thoughtlessly this can actually make things worse in the long run. After all, more memory means longer garbage-collection suspensions.

The following are the two most common causes for high memory usage.

Incorrect Cache Usage

It may seem counterintuitive, but excessive cache usage can easily lead to performance problems. In addition to the typical problems, such as misses and high turnaround, an overused cache can quickly exhaust available memory. Proper cache sizing can fix the problem, assuming you can identify the cache as the root cause. The key problem with caches is [soft references](#), which have the advantage that they can be released at any time at the discretion of the garbage collector. It's this property that makes them popular in cache implementations. The cache developer assumes, correctly, that the cache data is released in the event of a potential memory shortage; in essence, to avoid an out-of-memory error (in contrast to [weak references](#), which never prevent the garbage collection of an object and would not be useful in a cache).

If improperly configured, the cache will grow until the available memory is exhausted, which causes the JVM to trigger a GC, clearing all soft references and removing their objects. Memory usage drops back to its base level, only to start growing again. The symptoms are easily mistaken for an incorrectly configured young generation and often trigger a GC tuning exercise. This exercise can never succeed. This situation requires proper monitoring of the cache metrics or a heap dump to identify the root cause.

Session Caching Antipattern

When an HTTP session is misused as a data cache, we refer to it as the session caching antipattern. Correctly implemented, the HTTP session is used to store user data or a state that needs to survive beyond a single HTTP request. This *conversational state* is found in most web applications dealing with nontrivial user interactions, but there are potential problems.

First, when an application has many users, a single web server may end up with many active sessions. Most obviously, it's important that each session be kept small to avoid using up all available memory. Second, these sessions are not explicitly released by the application! Instead, web servers use a session timeout, which can be set quite high to increase the perceived comfort of the users. This can easily lead to large memory demands and HTTP sessions that are on the order of multiple megabytes in size.

Session caches are convenient because it is easy for developers to add objects to the session without considering other solutions that might be more efficient. This is often done in *fire-and-forget mode*, meaning data is never removed. The session will be removed after the user has left the page anyway, or so we may think, so why bother? What we ignore is that session timeouts from 30 minutes to several hours are not unheard of.

One currently popular version of this antipattern is the misuse of hibernate sessions to manage the conversational state. The hibernate session is stored in the HTTP session in order to facilitate quick access to data. This means storage of far more state than necessary, and with only a couple of users, memory usage immediately increases greatly.

If we couple a big HTTP session with session replication, we get large object trees that are expensive to serialize, and a lot of data to be transferred to the other web servers. We just added a severe performance problem on top of the fact that we run out of memory quickly.

ClassLoader-Related Memory Issues

Sometimes I think the Java classloader is to Java what *dll-hell* was to Microsoft Windows (Read [here](#), [here](#) and [here](#) if you want to know more about this sort of hell). However, modern enterprise Java applications often load thousands of classes, use different isolated classloaders, and generate classes on the fly. While Java classloader issues lead to the same runtime issues as the aforementioned *dll-hell* (multiple versions of the same class/method), they also lead to memory leaks and shortages that need to be addressed in any book about Java performance.

When there are memory problems, one thinks primarily of normal objects. However, in Java classes, objects are managed on the heap, as well. In the HotSpot JVM, classes are located in the *permanent* generation or PermGen (See the section [Not All JVMs Are Created Equal](#)). It represents a separate memory area, and its size must be configured separately. If this area is full, no more classes can be loaded and an out-of-memory error occurs in the PermGen. The other JVMs do not have a permanent generation, but that does not solve the problem, as classes can still fill up the heap. Instead of a PermGen out-of-memory error, which at least tells us that the problem is class-related, we get a generic out-of-memory error.

Here we'll cover the most common classloader-related memory issues (plus one rare but educational one), and how to identify and solve them.

Large Classes

A class is an object and consumes memory. Depending on the number of fields and class constants, the needed memory varies. Too many large classes, and the heap is maxed out.

Quite often the root cause of large classes is too many static class constants. Although it is a good approach to keep all literals in static class constants, we should not keep them all in a single class. In one case a customer had one class per language to contain all language-specific literals. Each class was quite large and memory-hungry. Due to a coding error the application was loading not just one, but every language class during startup. Consequently, the JVM crashed.

The solution is quite simply to split large classes into several smaller ones, especially if you know that not all of the constants are needed at the same time. The same is true for class members. If

you have a class with 20 members, but depending on the use case you use only a subset, it makes sense to split the class. Unused members still increase the size of the class!

Single Class in Memory Multiple Times

It is the very purpose of a classloader to load classes in isolation to each other. Application servers and OSGi containers use this feature of classloaders to load different applications or parts of applications in isolation. This makes it possible to load multiple versions of the same library for different applications. Due to configuration errors, we can easily load the same version of a library multiple times. This increases the memory demand without any added value and can lead to performance problems, as well.

Symptom

A customer ran a service-oriented architecture application on Microsoft Windows 32-bit JVMs. His problem: he needed to assign 700 MB to the PermGen, but 32-bit Windows does not allow more than ~1500 MB per Java process. This did not leave him with enough room for the application itself.

Each service was loaded in a separate classloader without using the shared classes jointly. All common classes, about 90% of them, were loaded up to 20 times. The result was a PermGen out-of-memory error 45 minutes after startup.

I was able to identify this by getting a histogram memory dump from the JVM in question (jmap -histo). If a class is loaded multiple times, its instances are also counted multiple times. If we see a single class several times with different counters, we know that it was loaded multiple times. I subsequently requested a full heap dump and analyzed the references to the classes that were loaded multiple times. I found that the same JAR file was loaded via different classloaders!

Another symptom is that calls from one service to the other would serialize and deserialize the service parameters, even if those calls happened within the same JVM (I could see this as a hot spot in the performance analysis of those service calls). Although the different applications did use the same classes, they resided in different classloaders. Hence the service framework had to treat them as different. The service framework solved this by passing the parameters per-value and not per-reference. In Java this is done by serializing and deserializing.

Solution

I remedied the problem by changing the configuration switch in the JBoss deployment files of the services. The deployment file defined which JAR files should be loaded in isolation and which should be shared. By simply setting the common JAR files to shared, the memory demand of the PermGen dropped to less than 100 MB.

ClassLoader Leaks

Especially in application servers and OSGi containers, there is another form of memory leak: the classloader leak. As classes are referenced by their classloaders, they get removed when the classloader is garbage-collected. That will happen only when the application gets unloaded. Consequently, there are two general forms of classloader leak:

ClassLoader Cannot Be Garbage-Collected

A classloader will be removed by the garbage collector only if nothing else refers to it. All classes hold a reference to their classloader and all objects hold references to their classes. As a result, if an application gets unloaded but one of its objects is still being held (e.g., by a cache or a thread-local variable), the underlying classloader cannot not be removed by the garbage collector!

Symptom

This will happen only if you redeploy your application without restarting the application server. The JBoss 4.0.x series suffered from just such a classloader leak. As a result I could not redeploy our application more than twice before the JVM would run out of PermGen memory and crash.

Solution

To identify such a leak, un-deploy your application and then trigger a full heap dump (make sure to trigger a GC before that). Then check if you can find any of your application objects in the dump. If so, follow their references to their root, and you will find the cause of your classloader leak. In the case of JBoss 4.0 the only solution was to restart for every redeploy.

Leaking Class Objects

The second classloader leak version is even nastier. It first came into existence with now-popular bytecode-manipulation frameworks, like BCEL and ASM. These frameworks allow the dynamic creation of new classes. If you follow this thought you will realize that classes, just like objects, can be created and subsequently forgotten by the developer. The code might create new classes for the same purpose multiple times. You will get a nice classloader leak if either the class or its object remains referenced. The really bad news is that most heap-analyzer tools do not point out this problem; we have to analyze it manually, the hard way. This form of memory leak became famous due to an issue in an old version of Hibernate and its usage of CGLIB (see this [discussion on Hibernate](#) for details).

Symptom-Solution

One way to identify such a problem is to check a full heap dump for the leaked classes. If the generated classes share a common naming pattern, you should be able to search for them. You can then check if you find multiple classes with the same pattern, where you know you should

only have one. From there you should be able to find the root reference easily by traversing the references.

Same Class Being Loaded Again and Again

Lastly I want to describe a phenomenon where the same class is loaded repeatedly without it being in memory multiple times. It is not a common phenomenon, but neatly shows why it is important to know about the behaviors of different JVMs.

Contrary to popular belief, classes will be garbage-collected! The HotSpot JVM does this only during a real major GC (see the earlier discussion of [major vs. minor GCs](#)), whereas both IBM WebSphere JVM and JRockit JVM might do it during every GC. If a class is used for only a short time it might be released immediately (like every other temporary object). Loading a class is not exactly cheap and usually not optimized for concurrency. In fact, most JVMs will synchronize this, which can really kill performance!

Symptom

I have seen this twice in my career so far. In one specific case, the classes of a script framework (Bean Shell) were loaded and garbage-collected repeatedly while the system was under heavy load. Since multiple threads were doing this it led to a global synchronization point that I could identify by analyzing the locking behavior of my threads (leveraging multiple thread dumps). However, the development happened exclusively on the Oracle HotSpot JVM. As mentioned, the HotSpot JVM garbage-collects classes only in a major GC, therefore the problem never occurred during the development cycle. The production site, on the other hand, used an IBM WebSphere JVM and there the issue happened immediately. The lesson learned was that [not all JVMs are equal](#).

Solution

As a solution I simply cached the main object (the BeanShell Interpreter). By ensuring that the main object was not garbage-collected, I made sure that all required classes were kept alive and the synchronization issue was gone.

Conclusion

ClassLoader problems are hard to spot. The reason is not really a technical one. Most developers simply never have to deal with this topic and tool support is also poorest in this area. Once you know what to look for, you can easily identify classloader problems with a combination of trending and full heap dumps.

Out-of-Memory Errors

Out-of-memory errors occur when there is not enough space available in the heap to create a new object. A JVM will always trigger a garbage collection and try to reclaim memory before

raising an out-of-memory error. In most cases the underlying cause is an insufficient memory configuration or a memory leak, but there are some others.

Here is a non-exhaustive list of the most important ones:

- Insufficient memory configuration
- Memory leak
- Memory fragmentation
- Excess GC overhead
- Allocating oversized temporary objects

Besides the first two, memory fragmentation can also cause out-of-memory errors even when there appears to be enough free memory, because there isn't a single continuous area in the heap that's large enough for the allocation request. In most cases, compaction assures that this does not occur, however there are GC strategies that do not use compaction.

Some JVM implementations, such as the Oracle HotSpot, will throw an out-of-memory error when GC overhead becomes too great. This feature is designed to prevent near-constant garbage collection—for example, spending more than 90% of execution time on garbage collection while freeing less than 2% of memory. Configuring a larger heap is most likely to fix this issue, but if not you'll need to analyze memory usage using a heap dump.

The last issue is often a matter of developer thoughtlessness: program logic that attempts to allocate overly large temporary objects. Since the JVM can't satisfy the request, an out-of-memory error is triggered and the transaction will be aborted. This can be difficult to diagnose, as no heap dump or allocation-analysis tool will highlight the problem. You can only identify the area of code triggering the error, and once discovered, fix or remove the cause of the problem.

Churn Rate and High Transactional Memory Usage

The churn rate describes the number of allocations of temporary objects per transaction or time slice. Java allows us to allocate a large number of objects very quickly, but high concurrency and high throughput quickly lead to churn rates beyond what the JVM can sustain. The transactional memory usage on the other hand describes how much memory a transaction keeps alive until it is finished. (e.g. a single Transaction might need at least 5MB memory and creates 1000 temporary objects). High concurrency means that there are many active transactions and each of them needs some memory. If the sum of that (100 concurrent Transactions at 5 MB = 500 MB) is beyond the capacity of the young generation, temporary objects will be tenured to the old generation! The problem is easily overlooked during development because it is visible only under high-load situations.

Symptom

A high churn rate alone will slow down the applications because the JVM needs to execute young-generation GCs more frequently. Young-generation GCs are inexpensive only if most objects die! In a high-load situation with many concurrent transactions, many objects will be alive at the time of the GC. Therefore a high transactional memory usage, meaning a lot of live objects in the young generation, leads to longer and more expensive young-generation GCs. If the transactional memory usage exceeds the young generation's capacity (as described earlier), objects are tenured prematurely and the old generation will grow. This then leads to more-frequent old-generation GCs, further hurting performance.

In extreme cases the old generation's capacity gets exceeded and we end up with an out-of-memory error. The tricky part is that the out-of-memory error will abort all running transactions and the subsequent GC will remove the root cause from memory. Most memory tools look at the Java memory every couple of seconds and will never see 100% utilization. As a result there seems to be no explanation for the out-of-memory error.

Solution

The indicators for this problem are clear: frequent minor and expensive GCs, eventually leading to a growing old generation. The solution is also clear, but does involve a lot of testing and subsequent code changes.

1. Do a thorough allocation analysis to bring down the churn rate.
2. Take several heap dumps under full load. Analyze how much memory a single transaction keeps alive and try to bring that down. The more concurrency you expect in your production system, the less memory a single transaction should use.
3. Make sure you follow up your optimizations with a young-generation-sizing exercise and extensive load testing.

Optimizing churn rate issues is not a simple task, but the performance and scalability improvement can be substantial.

Incorrect Implementation of Equals and Hashcode

The relationship between the hashcode and equals methods and memory problems is not obvious at first. But we only need to think about hashmaps to make it more clear. An object's hashcode is used to insert and find objects in hashmaps. However, the hashcode is not unique, which is why it only selects a bucket that can potentially contain multiple objects. Because of this, the equals method is used to make sure that we find the correct object. If the hashcode method is wrong (which can lead to a different result for otherwise-equal objects), we will never find an object in a hashmap. The consequence is often that the application inserts an object again and again.

Although the growing collection can easily be identified by most tools, the root cause is not obvious in a heap dump. I have seen this case over and over through the years, and one

extreme case led the customer to run his JVMs with 40 GB of memory. The JVM still needed to be restarted once a day to avoid out-of-memory errors. We fixed the problem and now the application runs quite stable at 800 MB!

A heap dump—even if complete information on the objects is available—rarely helps in this case. One simply would have to analyze too many objects to identify the problem. The best approach is to be proactive and automatically unit-test comparative operators. A few free frameworks (such as [EqualsVerifier](#)) ensure that the equals and hashCode methods conform to the contract.