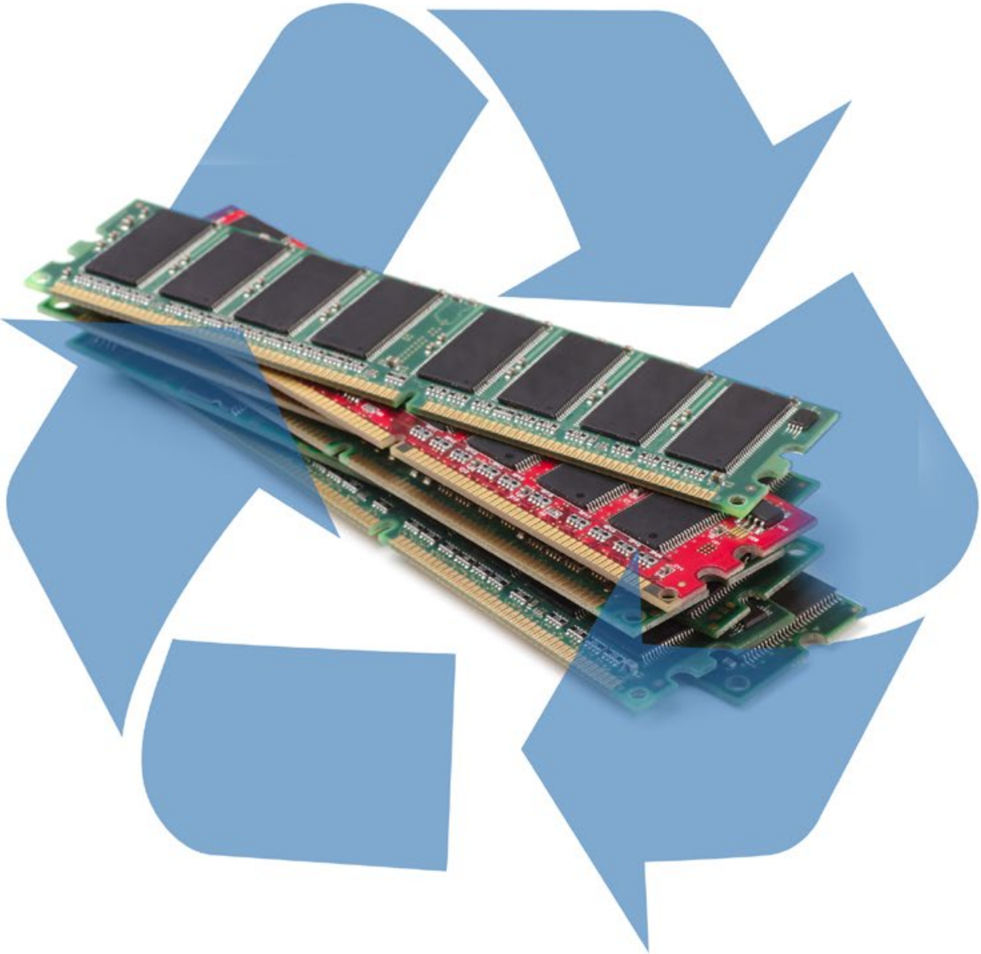


THE JAVA GARBAGE COLLECTION MINI-BOOK

Charles Humble

InfoQueue

ENTERPRISE SOFTWARE
DEVELOPMENT SERIES

The Java Garbage Collection Mini-book

Published by C4Media, publisher of InfoQ.com.

Production Editor: Ana Ciobotaru
Copy Editor: Lawrence Nyveen
Cover and Interior Design: Dragos Balasoiu

Library of Congress Cataloguing-in-Publication Data:
ISBN: 978-1-329-31238-8

Dedication

This book is dedicated to my parents who bought my first computer, a Commodore 64, when I was 10. They then not only tolerated but actively encouraged my tremendous enthusiasm and endless lengthy explanations as I slowly learned what it could be made to do.

Acknowledgements

I am enormously grateful to my wife, Krista, who has put up with me through the 16 years of our marriage so far, and who, despite having little interest in the subject, was kind enough to read several versions of this book, correcting typos and helping improve the clarity of the text. Taking on this project has meant a significant amount of weekend work and I'm hugely grateful to her for tolerating its impact on our family time.

A number of people whose algorithms I've briefly described took the time to explain them to me. I'm particularly grateful to Christine Flood of Red Hat and Gil Tene from Azul Systems, both of whom also took the trouble to read the complete draft and provide feedback.

John Galvez at IBM helped me to get answers to my questions on IBM's Balanced and Metronome collectors.

Ben Evans read an early draft of the book and provided a tremendous amount of invaluable feedback.

Dragos Balasoiu re-drew most of the diagrams from my hopeless originals.

Contents

Preface 2
Introduction 6

Foundations 9
The heap and pointers 10
Key terms 12
Collector mechanisms 18
General trade-offs 22

Two-region collectors 25
Heap structure 27
Serial collector 30
Parallel collector 31
Concurrent Mark Sweep (CMS) 32

Multi-region collectors 35
Heap structure 36
Garbage First 37
Balanced 39
Metronome 42
C4 44
Shenandoah 49

General monitoring and tuning advice 55
The major attributes 56
Getting started 57
Choosing a collector 57
Tuning a collector 58
Conclusion 83

Programming for less garbage 85
Reducing allocation rate 88
Weak references 92
Try-with-resources 94
Distributing programs 95
Other common techniques 95

Suggestions for further reading 96

Preface

Before I became full-time head of editorial for InfoQ.com[1], I was frequently asked to go in to organisations that were experiencing performance problems with Java-based applications in production. Where the problem was Java-related (as opposed to some external factor such as a poorly performing database), by far the most common cause was garbage collection.

Whilst it is often possible to correct these kinds of problems through tuning, it certainly isn't always the case. One system I was asked to look at was a web application that was also required to perform ad hoc batch-job-type functions. The system had been designed as a single monolithic application and could never possibly reach both goals – the response times needed for the web application necessitated a relatively small heap whilst the batch process needed to keep a large amount of data in memory and generated a huge amount of garbage. The two elements would also perform best with different collectors. It required some fundamental rework, at considerable expense, before it was able to obtain acceptable performance characteristics in production.

From talking to architects and senior developers at that and other organisations, it became apparent that the majority of them hadn't considered garbage collection at all, and those who had were often misinformed. Common misconceptions included the idea that garbage collection is inefficient (it's not; actually it's far more efficient than C's `malloc()` for example), that you can't have memory leaks in Java (you can), and that you can tune the HotSpot CMS collector so that it won't ever stop the world (you can't, in fact, roughly speaking: most collectors will pause for ~1 second per GB of live objects in the heap).

If any of these points surprise you, or if you aren't sure what the differences are between, say, parallelism and concurrency, or what a garbage-collection safe point is, or what the differences are between HotSpot CMS, G1, and C4, and when you would choose one instead of another, then this book can help. My goal in writing it is to provide a quick, accessible guide for Java developers and architects who want to understand what garbage collection is, how it works, and how it impacts the execution of their programs.

[1] http://www.infoq.com

There is already a definitive book on the subject, *The Garbage Collection Handbook* by Richard Jones et al., and I recommend it highly. However, it is a fairly dense, academic read, and I felt that a terser, lighter introduction to the subject was worthwhile. An InfoQ mini-book seemed like the ideal format for this. If reading this book makes you want to learn more about the subject, *The Garbage Collection Handbook* is a good place to go next, as are the many talks and other online resources I reference in the text. In the "Suggestions for further reading" section at the end of the book, I provide links to the academic papers that describe in more detail many of the algorithms I talk about here, and some other resources you may find useful.

What is in an InfoQ mini-book?

InfoQ mini-books are designed to be concise, intending to serve technical architects looking to get a firm conceptual understanding of a new technology or technique in a quick yet in-depth fashion. You can think of these books as covering a topic strategically or essentially. After reading a mini-book, the reader should have a fundamental understanding of a technology, including when and where to apply it, how it relates to other technologies, and an overall feeling that they have assimilated the combined knowledge of other professionals who have already figured out what this technology is about. The reader will then be able to make intelligent decisions about the technology once their projects require it, and can delve into sources of more detailed information (such as larger books, or tutorials) at that time.

Who this book is for

This book is aimed specifically at Java architects and senior developers who want a rapid introduction to how garbage collection works, the different underlying algorithms and memory structures, which collectors are commonly used, and how to choose one over another.

What you need for this book

Whilst there are no coding exercises in this book, you may want to try out some of the individual code samples on your own machine. To do this, you will need a computer running an up-to-date operating system (Windows, Linux, or Mac OS X). Java also needs to be installed. The book code was tested against JDK 1.8, but newer versions should also work.

Conventions

We use a number of typographical conventions within this book that distinguish between different kinds of information.

Important additional notes are shown using callouts like this.

Code in the text, including database table names, folder names, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

“The size of the Java heap can be typically controlled by two flags, `-Xms` for the initial size and `-Xmx` for the maximum size.”

A block of code is set out as follows:

```
public class GCInformation {

  public static void main(String[] args) {

    List<GarbageCollectorMXBean> gcMxBeans =
      ManagementFactory.getGarbageCollectorMXBeans();

    for (GarbageCollectorMXBean gcMxBean : gcMxBeans) {
       System.out.println(gcMxBean.getName());
    }
  }
}
```

When we want to draw your attention to a particular part of the code it is highlighted in bold:

```
[PSYoungGen: 542688K->253792K(628224K)]
1290819K->1185723K(1629184K), 0.3456630 secs] [Times:
user=1.01 sys=0.08, real=0.35 secs]
2015-03-07T22:11:25.540+0000: 7.239: [Full GC
[PSYoungGen: 253792K->0K(628224K)] [ParOldGen:
931931K->994693K(1595392K)] 1185723K->994693K(2223616K)
[PSPermGen: 3015K->3015K(21504K)], 3.1400810 secs]
[Times: user=8.78 sys=0.15, real=3.14 secs]
```

Reader feedback

We always welcome feedback from our readers. Let us know what you think about this book — what you liked or disliked. Reader feedback helps us develop titles that you get the most out of.

To send us feedback email us at `feedback@infoq.com`.

If you have a topic that you have expertise in and you are interested in either writing or contributing to a book, please take a look at our minibook guidelines on `http://www.infoq.com/minibook-guidelines`.

Introduction

The Java Language Specification mandates the inclusion of automatic storage management. "Typically," the spec states, "using a garbage collector, to avoid the safety problems of explicit deallocation (as in C's free or C++'s delete)."[2]

Automatic garbage collection is generally a Very Good Thing. It frees the programmer from much of the worry about releasing objects when they are no longer needed, which can otherwise consume substantial design effort. It also prevents some common types of bug occurring, including certain kinds of memory leaks, dangling-pointer bugs (which occur when a piece of memory is freed whilst there are still pointers to it and one of those pointers is then used), and double-free bugs (which occur when the program tries to free a region of memory that has already been freed and perhaps already been allocated again).

Whilst garbage collection clearly has a number of advantages, it does also create some problems. The most significant problem is that, with one exception, practical implementations of garbage collection in commercial Java runtimes involve an unpredictable pause during collection, generally referred to as a "stop-the-world event". Stop-the-world events have always been a problem for Java client programs, where even a short pause in the responsiveness of the UI can negatively impact how users feel about an application. However, as server-side Java programs have expanded in size and complexity and as more and more Java code is being used in environments such as financial exchanges and trading systems where performance is a primary concern, so the garbage collection pause has become an increasingly significant problem for Java software architects.

Java teams use a wide range of techniques to mitigate the problem: breaking programs down into smaller units and distributing them (a technique I call "premature distribution"), object pooling, having fixed-sized objects to avoid fragmentation, and using off-heap storage. If nothing else, the fact that so many workarounds exist demonstrates that garbage-collection pauses either are or are perceived to be a problem for many enterprise applications, and thus a basic understanding of garbage collection should be considered essential knowledge for a Java architect or senior programmer.

[2] http://docs.oracle.com/javase/specs/jls/se8/html/jls-1.html

A corollary is the typical size of the heap. In my experience most Java programs in production today are given heap sizes of 1 GB to 4 GB memory because 4 GB is about the most that they can cope with whilst having pauses of an acceptable length. Gil Tene did a quick informal survey of the audience for his talk[3] at SpringOne 2011 and saw results in line with this. More recently, Kirk Pepperdine did a similar exercise[4] at QCon New York, again with similar results.

Whilst 10 years ago, a 512-MB to 1-GB heap size would have been considered substantial, and a high-end commodity server might have shipped with 1-2 GB of RAM and a two-core CPU, a modern commodity-hardware server typically has around 96-256 GB of memory running on a system with 24 to 48 virtual or physical cores. Over a period of time during which commodity hardware memory capacities have increased a hundredfold, commonly used garbage-collector heap sizes have only doubled. To put this another way, the performance of garbage collectors has lagged significantly behind both the hardware and software demands of many larger enterprise applications.

[3] http://www.infoq.com/presentations/Understanding-Java-Garbage-Collection

[4] http://www.infoq.com/presentations/g1-gc-logs

PART ONE

Foundations

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean — neither more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

— Lewis Carroll, Alice's Adventures in Wonderland

One of the challenges for anyone wanting to understand garbage collection is the terminology. The same word may be used to mean different things, and different words may be used that mean the same thing. So to get started, we need first to define a set of terms that we'll use. I'll try to keep these to a minimum, and also highlight alternative terms that you may encounter if you read other articles and books on the subject.

It is entirely possible, particularly if Java is your only language, that you've never really thought about either the heap or pointers before. Given how fundamental these are to the discussion that follows, a quick definition seems like a sensible place to start.

The heap and pointers

The purpose of the garbage collector is to reclaim memory by cleaning up objects that are no longer used. Garbage collectors determine which memory to retain and which to reclaim, using reachability through chains of pointers.

In Java, an object is either a class instance or an array. The reference values (often called only "references") logically **point** to these objects from stack frames or other objects. There is also a special null reference which refers to no object.

Objects are stored in Java's heap memory. Created when the Java Virtual Machine (JVM) starts up, the Java heap is the run-time data area from which memory for all object instances (including arrays) is allocated. In addition, heap memory can be shared between threads. All instance fields, static fields, and array elements are stored in the Java heap. In contrast, local variables, formal-method parameters, and exception-handler parameters reside outside the Java heap; they are never shared between threads and are unaffected by the memory model. The heap is mutated by a mutator, which is just a fancy name for your application.

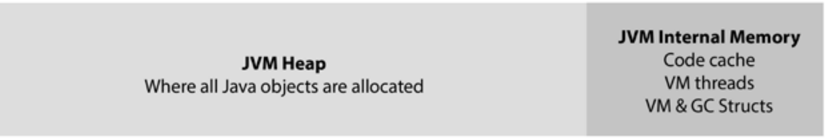


Figure 1: Heap overview

The size of the Java heap can be typically controlled by two flags, `-Xms` for the initial size and `-Xmx` for the maximum size. It is worth noting that most JVMs also use heaps outside of Java for storing material other than Java objects, such as the code cache, VM threads, VM and garbage-collection structures and so on, so the total size of the process memory will be larger than the maximum Java heap size you give it. Generally, you only need to worry about this if you are working in a very memory-constrained environment.

An example may help make this clearer. In figure 2, we have two stack frames, for Foo and Bar. They have immediate values like 42 and 3.1416, but they also have references to other objects allocated in the heap. As you can see, some of these objects are referenced by a single stack frame, some by multiple stack frames, and one of them is not referenced from anywhere. All the garbage collector is doing is finding the objects that are pointed to, compacting them into an area of memory so that you have better cache-line behaviour, and freeing up space for you to allocate new objects.

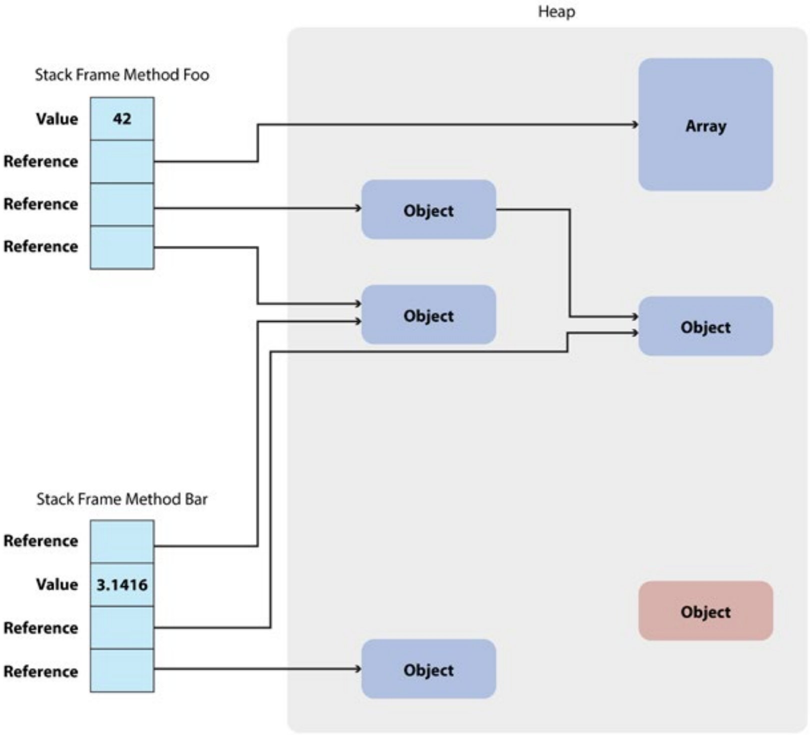


Figure 2: Heap and stack frames

Key terms

Before we move on to look at individual collectors, there are a few other terms you need to be familiar with.

Parallel vs. serial, concurrent vs. stop-the-world

Garbage collectors can be either single-threaded or parallel:

- A single-threaded collector uses a single CPU core to perform garbage collection.
- A parallel collector uses multiple threads that can perform collection work at the same time, and can therefore make simultaneous use of more than one CPU core.

Collectors are also either stop-the-world or concurrent:

- A stop-the-world (STW) collector performs garbage collection during a pause when the application is not running any code that could mutate the heap.
- A concurrent collector performs garbage collection concurrently with program execution, i.e. allowing program execution to proceed whilst collection is carried out (potentially using separate CPU cores).

Confusion between parallel and concurrent is widespread because the meaning of the terms has changed over time. In a number of older academic papers (25 to 30 years old), the term parallel was used as a shorthand for "in parallel with program execution" — i.e. what we now call concurrent.

Incremental vs. monolithic

An incremental collector performs garbage-collection operations as a series of smaller discrete steps with potentially large gaps between them. Generally, this means that it can let the application run in between the steps, and is usually used in conjunction with STW. Incremental is the opposite of monolithic — a monolithic STW collector algorithm always stops the application until everything is done.

About "mostly"…. The word "mostly" is often used to qualify technical terms in garbage-collection algorithms and to denote "not always". Whilst correctly used in academic papers to describe algorithms, the word “mostly” is often used in product descriptions to describe collectors in a positive light and to pull attention away from the opposite qualities they carry. Specifically, “mostly concurrent” should actually be read to mean “sometimes stop-the-world”, “mostly incremental” should be read to mean “sometimes monolithic”, and “mostly parallel” should be read to mean “sometimes serial”.

Precise vs. conservative

A collector is precise if it can fully identify and process all object references at the time of collection. A collector has to be precise if it is to move objects, since if you are going to take an object and move it from point A to point B, you have to be able to find all the pointers to it and point them to B to avoid corrupting the heap or other unfortunate mutational effects. Being precise is actually not a hard thing for a garbage collector, as the burden for having precise information usually lies elsewhere, and the collector just needs the relevant information. The compiler usually produces the necessary information. Indeed, the main burden for JIT compilers with regard to garbage collection is to produce accurate information about the location of references to the heap in registers and stack locations. In HotSpot-based JVMs, this information is tracked in the form of OopMaps; structures that record where object references (OOPs) are located on the Java stack for every piece of code it is safe to do a GC in. There are three types:

1. OopMaps for interpreted methods can be computed lazily, i.e. when garbage collection happens, by analysing byte-code flow.

2. OopMaps for JIT-compiled methods are generated during JIT compilation and kept with the compiled code so that the VM can quickly find, using the instruction address, the stack locations and registers where the object references are held.

3. OopMaps for generated shared run-time stubs are constructed manually by the developers/authors of these run-time stubs.

Compilers that don’t produce the relevant information, in whatever form, cannot support a precise collector and therefore cannot support a moving collector.

A collector is termed conservative if it is unaware of some object references at collection time, or is unsure about whether a field is a reference or not. This is actually the normal case for most languages and most systems. C++, for example, normally employs some form of conservative collector.

All commercial server JVMs use precise collectors and use a form of moving collector at some point in the garbage collection cycle.

Safe points

Garbage-collection events occur at safe points. A garbage-collection safe point is a point or range in a thread's execution when the collector can identify all the references in that thread's execution stack.

Bringing a thread to safe point is the act of getting a thread to reach a safe point and then not executing past it. This is not necessarily the same as stopping at a safe point; you can still be using CPU cycles. For example, if you make a call out to native code via JNI, that thread is at a safe point while you are running in native code, since JNI cannot mess with Java pointers.

A global safe point involves bringing all threads to a safe point. These global safe points represent the STW behaviour commonly needed by certain garbage-collection operations and algorithms. Their length depends on two things: the duration of the operation to take place during the global safe point and the time it takes to reach the safe point itself (also known as "time to safe point").

Safe-point opportunities should ideally occur frequently in executing code. A long time between safe-point opportunities in even a single thread's execution path can lead to long pauses when the code requires a global safe point. Long code paths between safe-point opportunities can lead to a situation where all threads but one have reached a safe point and are in a STW pause while the operation that requires the global safe point cannot proceed until the that one remaining thread reaches its next safe-point opportunity.

Unfortunately, depending on the JVM involved, certain optimisations and run-time code paths (e.g. counted-loop optimisations, memory copy operations, and object cloning) can result in many-millisecond periods of execution with no safe-point opportunity, leading to occasionally long time-to-safe-point-related pauses (ranging into large fractions of a second).

To further complicate the time-to-safe-point issue, several commonly used collectors do not consider time-to-safe-point pauses to be part of their garbage-collection pause time, starting their pause reporting only when the global safe point has been reached. This can lead to significant underreporting of pause lengths in most garbage-collection logs. On JVMs that support the `-XX:+PrintGCApplicationStoppedTime` flag, the related (and separate) pause-time output will generally report time inclusive of time-to-safe-point pauses, making it much more reliable.

It's worth saying that the terms "safe point" and "garbage-collection safe point" are commonly used interchangeably, but there are other reasons for taking a global safe point in a JVM — for instance, de-optimisation safe points. The JVM can de-optimise code because an assumption it made was wrong; a common case where this occurs is in class hierarchy analysis. Suppose the compiler has recognised that a certain method only has one implementer, so it optimises it, perhaps inlining it and making it a static call instead of a virtual call. Then you load a new class that overloads that reference, making the underlying assumption wrong; if we keep running, we'll end up calling the wrong function. When this happens, the compiler needs to take something that is compiled, throw away the frame, and reconstruct an interpretive stack with an equivalent JVM stack. A safe point for doing this is clearly broader than the garbage-collection safe point since you need to know where every variable is, including the non-pointer ones, and where all the state information is.

Generational collection

Almost all commercial Java collectors take advantage of generational collection in some way to achieve significantly more efficient collection. JVMs that do this segregate the heap between short-lived objects and long-lived objects. These two separate "generations" are typically physically distinct areas or sets of regions of the heap. The young (or "new") generation is collected in preference to old (or "tenured") generation, and objects that survive long enough are promoted (or "tenured") from the young generation to the old. Collection of the young generation is sometimes referred to as a "minor garbage-collection event".

The basis for generational collection is an observation that is commonly called "the weak generational hypothesis". It reflects an apparently nearly universal observation that, in the majority of programs, most allocated objects tend to live for a very short period of time and so relatively few dynamically created objects survive very long.

Generational collectors exploit this by ignoring the oldest objects whenever possible and collecting the young heap portions using algorithms whose complexity rises only with the amount of live young matter (as opposed to rising with the size of the young heap). As long as the generational hypothesis actually holds for the young generation, the sparseness of the younger heap portions will provide such algorithms with significantly improved efficiency in reclaiming empty space. This generational filter typically allows the collector to maximise recovered space (sometimes called “yield”) whilst minimising effort.

An important additional benefit of generational collectors is that the pauses involved in the collection of the young generation (in collectors that pause to collect the young generation) tend to be significantly shorter than pauses involved in collecting the old generation (in collectors that pause to collect the old generation). Thus, in such pausing collectors, generational collection helps reduce the frequency (but not the duration) of the larger old-generation (or “oldgen”) pauses.

Managed run times similar to Java have been leveraging the weak generational hypothesis since the mid 1980s and it appears to hold true for pretty much every program we run on JVMs today. However, it is important to understand that this powerful observation does not actually allow us to avoid collecting the old generation altogether; instead, it simply allows us to reduce the frequency of oldgen collections.

While statistically true for dynamically allocated objects, the weak generational hypothesis does not mean that all objects die young, or even that all objects that die end up dying young. Most applications produce plenty of long-lived (but eventually dying) objects on a dynamic basis. For example, the common practice of caching (using either homegrown or library solutions) will typically generate a lot of objects that live for a fairly long time, but which will eventually die and be replaced by other long-lived objects. This does not mean that the generational hypothesis does not apply to applications that use caching (it tends to strongly apply even there) but it does mean that most applications cannot put off dealing with the old-generation material indefinitely. In many JVMs, significant effort is spent on tuning the collectors to maximise the efficiency afforded by generational collection. We’ll look at this in more detail in the sections that follow.

Remembered sets and write barriers

A generational collector relies on what's called a "remembered set" as a way of keeping track of all references pointing from the old generation into the young generation. The remembered set is considered to be part of the roots for young-generation collecting, thus allowing young-generation collections to avoid scanning the old generation in its entirety. In most generational collectors, the remembered set is tracked in a card table which may use a byte or a bit to indicate that a range of words in the old-generation heap may contain a reference to the young generation.

Card tables are generally sparsely populated, making their scanning during young-generation collection quite fast. However, it is important to realise that the length of a card-table scan, even when the card table is completely clean, necessarily grows with the size of the old generation. Card-table scanning can therefore come to dominate young-generation collection time as heap sizes grow.

Card tables and remembered sets can be precise or imprecise. In a precise card table, each card tracks an individual word in the oldgen heap that may hold a reference. In an imprecise card table, each card provides an aggregate indication that an old-to-young reference may reside in a range of addresses in the oldgen heap, requiring the collector to scan the entire card and identify potential reference words in it.

The remembered set is typically kept up to date with a write barrier that intercepts the storing of every Java reference in order to track potential references from old objects into the young generation. For example, in the card table, the write barrier may dirty a card associated with the address the reference is being stored to. HotSpot uses a "blind" store for its write barrier — meaning it stores on every reference write rather than using some sort of conditional store. In other words, it doesn't only track a reference to new from old. The reason for this is that, at least for some processors, checking whether a reference is needed is more expensive than doing the card mark itself. There are non-blind "conditional" variations of write barriers which potentially deal with two separate efficiency issues: one variation actually checks the reference being stored and the reference of the object it is being stored into, to see if the store will create an old-to-young reference in the heap. This test may involve a little more work on every reference store, but it can result in a much cleaner card table that is much faster to use for collection when needed. Another variant checks the content of the card that is to be dirtied, avoiding a store into an already dirty card. This test can significantly reduce cache-line contention on hot cards, reducing the related scaling bottlenecks that are common in some concurrent operations and algorithms.

Collector mechanisms

Precise garbage collectors, including all collectors in current commercial JVMs, use tracing mechanisms for collection. Such precise collectors can (and will) freely move objects in the heap, and will identify and recycle any dead matter in the heap regardless of the topology of the object graph. Cyclic object graphs are trivially and safely collected in such collectors, allowing them to reliably collect dead matter from heaps with an efficiency that greatly exceeds those of inaccurate techniques like reference counting.

Tracing collectors use three techniques: mark/sweep/compact collection, copying collection, and mark/compact collection.

Commercial Java uses all three approaches in some form. Different collectors may combine these approaches in different ways, collecting one region of the heap with one method and another part of the heap with a second method. However, whilst the implementations of garbage collectors in Java runtimes vary, there are common and unavoidable tasks that all commercial JVMs and garbage collection modes perform, and others that are common to several of the algorithms.

Mark (also known as trace)

A mark or trace is an approach for finding all the live objects in a heap. Usually starting from the roots (which are things like static variables, registers, and content on the thread stacks), we take every reference, follow it, and paint everything we reach as “live”. Thus, at the time of the mark, everything we could reach is alive, and everything else is dead and can therefore be thrown away. The space occupied by unreachable objects is safe to recycle because (in a managed-code, managed-data environment like Java) once an object becomes unreachable, it can never again become reachable. This is the basic principle that makes tracing collectors work, and also explains why tracing collectors don’t have to worry about linked lists in a loop or reference counting and so on. Basically, if you have a graph and a linked list but nothing is pointing into it then it can be thrown away regardless what it is.

From a complexity point of view, the work performed during the mark phase increases linearly with the size of the live set rather than the size of the heap. Therefore, if you have a huge heap and a tiny live set, you are not going to need to do additional work in the tracing part of a collector.

Sweep

Collector algorithms that include a sweep pass (e.g. mark/sweep, mark/sweep/compact) scan through the entire heap, identify all the dead objects (those not marked live), and recycle their space in some way — e.g. by tracking their locations in free lists of some sort or by preparing the dead areas for later compaction. Sweeping work correlates with heap size since you have to look at the entire heap to find all the dead stuff; even if you have a large heap with very little that is alive, the sweep phase still has to cover the entire heap.

Compact/relocate

Compaction is a necessary evil in virtually all JVMs: without compaction, memory reclaimed from dead objects of variable sizes will tend to fragment over time. With such fragmentation, the heap will eventually reach a point where a large amount of available memory exists but is spread around in small chunks, meaning that there is no slot large enough to accommodate an object you want to create. Unless you have fixed-sized objects with fixed-population counts, this will eventually happen to any Java heap.

To combat this, collectors must periodically relocate live objects to create some amount of contiguous free space. Compaction is a major task since it has to correct all object references to point to new object locations, a process called “remapping” or “fix up”. The remap scan must cover all references that could point to any relocated objects. The work that needs to be performed increases linearly with the size of the live set.

Compacting collectors can be separated into two groups: in-place compactors and evacuating compactors. In-place compactors generally work by moving all live objects to one side of the heap and then filling in the empty gaps identified in sweep operations. Evacuating compactors generally compact the heap by evacuating live objects from regions of the heap into some external, empty space that has been kept aside for that purpose, thus freeing up the source regions being evacuated.

Copy

A copying collector uses a different technique. In its simplest form, a copying collector splits a heap it is managing into two equally sized spaces, which will alternately be referred to as “from” and “to”. The current “to” space is always kept completely empty except during collection, while all actual objects are in the “from” space. All allocations go into the designated “from” space until that space is full, which triggers a collection cycle. The copying collector performs a trace through all reachable objects, moving all encountered objects from the “from” space to the “to” space as it goes, and correcting all encountered references in all objects as it does so. The copy is completed in a single pass: at the start of the copy, all objects were in “from” space and all references pointed to “from” space, and at the end of the copy all live objects are in “to” space, and all live references point to “to” space. At that point, the collector reverses the roles of the “from” and “to” spaces, the collection is done, and allocation into the newly designated “from” space can proceed. The amount of work a copying collector performs generally rises linearly with the size of the live set.

Copying collectors are typically monolithic: all objects must move from the “from” to the “to” space for the collection to complete, and there is no way back nor any means of dealing with half-completed collections that cannot proceed due to lack of space in the “to” part of the heap. This is why, in a single-generation copying collector, the size of the “from” and “to” space must be equal, because at the beginning of the collection there is no way to tell how much of the “from” space is actually alive.

However, when used to collect a young generation in the context of a generational collector, a copying collector can take a slightly more complicated form since it has the option to overflow into the old generation in the unlikely event that the young generation is not mostly dead. This resolves the correctness issue of having to deal with a potentially completely live “from” space: instead of using two equal-sized spaces designated “from” and “to” and leaving half of all heap space unused at all times, a young-generation copying collector can choose to optimistically size its “to” space to a small fraction of the “from” space size. This optimism is strongly supported by the weak generational hypothesis.

Such young-generation copy collectors generally divide the young-generation heap into three regions. In HotSpot-based JVMs, these are referred to as "Eden" (usually the largest of the three) and two equally sized (but much smaller than Eden) "survivor" spaces, as shown below.

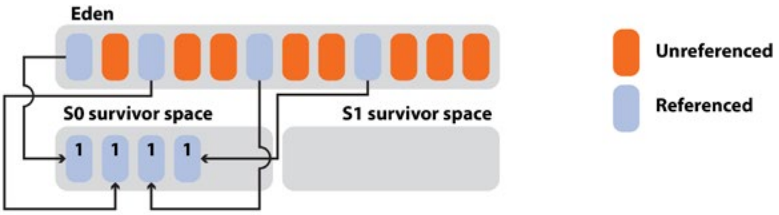


Figure 3: HotSpot copying collector

The collector performs all allocations into Eden, which is always considered part of the "from" space, whilst the survivors alternate between being "to" and another part of "from". Each collection starts with an empty "to" survivor space and a "from" space consisting of the "from" survivor space and Eden. During the collection, all live objects are either copied from the "from" (Eden + prior survivor) to the "to" (the current survivor) or promoted to the old generation. At the end of the collection, all surviving young objects will reside in the current "to" survivor space, while both Eden and the prior survivor space will be completely empty. A young-generation copying collector can reliably maintain these qualities even when the survivor space is much smaller than Eden, and even if much of Eden happens to be alive, because any surviving young objects that cannot fit into the current "to" survivor space will be promoted into the old generation heap. Such overflow promotion is sometimes called "premature promotion".

Aside from Azul's C4 collector, which uses mark/compact for the young generation, all other production-ready HotSpot collectors we will look at use a copying collector for the young generation.

A corollary of the above discussion is that young-generation collection can only be safely started when the old generation has as much empty space in it as the size of Eden. This is necessary in order to correctly handle the potential situation where all of Eden and the prior survivor space are full of still-alive objects.

General trade-offs

Each of the different collector mechanisms we've talked about has its own strengths and weaknesses. A copying collector works in a single pass; it prevents fragmentation so you never have to deal with compaction. However, copying is a monolithic operation: you have to copy the entire live set in one pass. Moreover, since you are copying the entire live set from one place to another, you need the heap to be double the maximum size of the live set in order for the collector to be reliable. This is somewhat costly. Still, copying collectors are commonly used for the young generation in commercial JVMs.

Like a copy, a mark/compact algorithm that doesn't have a sweep phase generally requires double the maximum heap size in order to fully recover garbage in each cycle.

Conversely, a mark/sweep collector only requires a little additional space on top of the maximum size of the live set, and it may be able to avoid some moving work. In other words, mark/sweep is better as far as heap occupancy is concerned and is generally more stable with regard to heap size whilst mark/compact and copying collectors are typically more efficient.

Because of this, depending on the collector's specific algorithm, these tasks may be performed in separate phases or as part of a single combined pass. For example, commonly used tracing collectors (popular for old-generation collection in commercial JVMs) often use separate mark, sweep, and compact phases to perform the identification, reclaiming, and relocation tasks. There are many variations of old-generation collectors. They may be stop-the-world, incremental stop-the-world, or mostly incremental stop-the-world. They may be concurrent, or mostly concurrent.

On the other hand, commonly used copying collectors (popular for young-generation collection in commercial JVMs) will typically perform all three tasks in a single copying pass (all live objects are copied to a new location as they are identified). Because the young generation has an old generation to promote into, it doesn't have to trigger a copy when the heap is only half full, as we might assume; it could actually get to completely full as long as there is room to put all the objects in the old generation. However, in current commercial JVMs that employ copying collection for the young generation, this is a monolithic, stop-the-world event.

Deciding the rate at which you promote objects from the young generation to the old generation is where a lot of time generally gets spent when tuning a HotSpot collector. Typically, you'll want to keep surviving objects in the young generation for at least one cycle before promotion, since immediate promotion can dramatically reduce the efficiency of the generational filter. Conversely, waiting too long can dramatically increase the copying work.

In the next part of the book, we will introduce the generational HotSpot collectors in more detail.

PART TWO

Two-region collectors

"When faced with two equally tough choices,

most people choose the third choice: to not choose."

— *Jarod Kintz,* This Book Title Is Invisible

The .NET CLR comes with two collectors: a client one and a server one. OpenJDK and the Oracle JDK, on the other hand, each come with four, and there are a number of other collectors available from different JVM providers.

Since Java 2 SE 5.0, default values for the garbage collector, heap size, and HotSpot virtual machine (client or server) are automatically chosen based on the platform and operating system on which the application is running. The JVM will often do a pretty decent job of selecting a garbage collector for you and it may be that you never have to consider the choices that it makes. You can, however, select the algorithm that the JVM is using for your program. Knowing what each collector does and how it works may help you in choosing the most appropriate one for your needs, though benchmarking is also important.

If you don't know which collector you are running then the following program will show you:

```
import java.lang.management.GarbageCollectorMXBean;
import java.lang.management.ManagementFactory;
import java.util.List;

public class GCInformation {

  public static void main(String[] args) {

    List<GarbageCollectorMXBean> gcMxBeans =
         ManagementFactory.getGarbageCollectorMXBeans();
    for (GarbageCollectorMXBean gcMxBean : gcMxBeans) {
     System.out.println(gcMxBean.getName());
    }
  }
}
```

This will output the younger generation first and show something like:

```
PS Scavenge
PS MarkSweep
```

PS Scavenge is the parallel Eden/survivor space collector. PS MarkSweep is the parallel old-generation collector (not the same as a concurrent collector).

CMS will show:

```
ParNew
ConcurrentMarkSweep
```

G1 will output:

```
G1 Young Generation
G1 Old Generation
```

Heap structure

The collectors we'll look at in this part of the book divide the heap into two regions — young/new and tenured — to exploit the weak generational hypothesis.

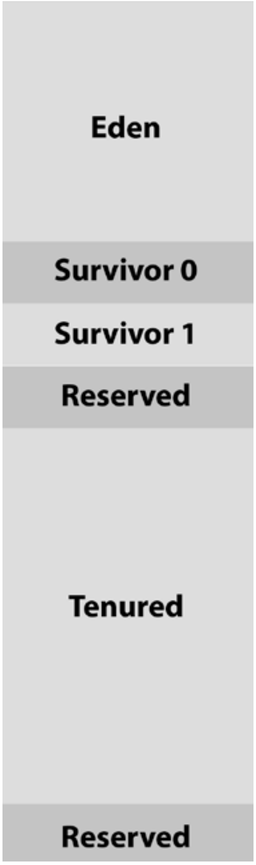


Figure 4: High-level generational heap structure

Eden is the region where most objects are initially allocated. The survivor spaces are a temporary store for objects that have survived a collection of the Eden space. Collectively, Eden and the survivor spaces are known as the "young" or "new" generation.

The action of promotion is called a "minor GC event" and the collection is triggered when Eden becomes full. Promotion is done by copying all the live objects in the new generation to either a survivor space or the tenured space as appropriate. Copying to the tenured space is known as "promotion" or "tenuring".

Promotion occurs for objects that are sufficiently old (`-XX:MaxTenuringThreshold`) or when the survivor space overflows. By doing this, we keep the assumption about the young generation true; the objects in the young generation will die young, because those that don't will be promoted.

To allow a degree of resizing of the pools without having to move everything, there are also two reserved areas (sometimes called virtual spaces) between the two heap spaces.

PermGen

Prior to Java 8, HotSpot also had a permanent generation ("PermGen") contiguous with the Java heap in which the runtime stored objects that it believed were effectively immortal, along with per-class metadata such as hierarchy information, method data, stack and variable sizes, the runtime constant pool, resolved symbolic reference, and Vtables.

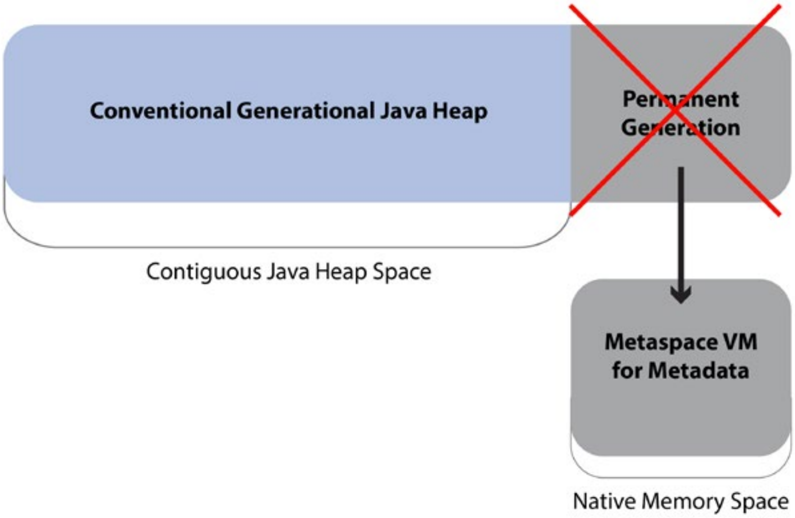

Conventional Generational Java Heap
Permanent Generation
Contiguous Java Heap Space
Metaspace VM for Metadata
Native Memory Space

Figure 5: PermGen moves to Metaspace

Unfortunately, many of the working assumptions behind PermGen are incorrect, particularly now that so many applications make use of custom class loaders. In Java 7, strings were moved from PermGen to tenured, and in Java 8 the permanent generation is no more, with the metadata moved to native memory in an area known as "Metaspace".

The move to Metaspace was necessary since the PermGen was really hard to tune. One problem was that its size was limited based on `-XX:MaxPermSize`. This had to be set on the command line before starting the JVM or it would default to 64 MB (85 MB for 64-bit scaled pointers). Sizing depended on a lot of factors, such as the total number of classes, the size of the constant pools, size of methods, etc., and was notoriously difficult. By contrast, the class metadata is now allocated out of native memory, which means that the max available space is the total available system memory.

The collection of the permanent generation was tied to the collection of the old generation, so whenever either was full, both the permanent generation and the old generation would be collected. If the class's metadata size was beyond the bounds of `-XX:MaxPermSize`, your application ran out of memory. There was a possibility that the metadata could move with every full garbage collection.

From an implementation standpoint, each garbage collector in HotSpot needed specialised code for dealing with metadata in the PermGen. Writing for InfoQ, Monica Beckwith[1] pointed out that "detaching metadata from PermGen not only allows the seamless management of Metaspace, but also allows for improvements such as simplification of full garbage collections and future concurrent deallocation of class metadata."

Most other commercial collectors also treat all long-living objects as tenured.

Object allocation

HotSpot uses the bump-the-pointer technique to get faster memory allocations, combined with thread-local allocation buffers (TLABs) in multithreaded environments.

The bump-the-pointer technique tracks the last object allocated to the Eden space, which is always allocated at the top. If another object is created afterwards, it checks only that the size of that object is suitable for the Eden space, and if it is, it will be placed at the top of the Eden space. So, when new objects are created, only the most recently added object needs to be checked, which allows much faster memory allocations.

To avoid contention in a multithreaded environment, each thread is assigned a TLAB from which it allocates objects. Using TLABs allows object allocation to scale with number of threads by avoiding contention on a single memory resource. When a TLAB is exhausted, a thread simply requests a new one from the Eden space. When Eden has been filled, a minor collection is triggered.

Large objects (`-XX:PretenureSizeThreshold=n`), for example a large array, may fail to be accommodated in the young generation and thus have to be allocated in the old generation. If the threshold is set below TLAB size then objects that fit in the TLAB will not be created in the old generation.

[1] Where Has the Java PermGen Gone? -
http://www.infoq.com/articles/Java-PERMGEN-Removed

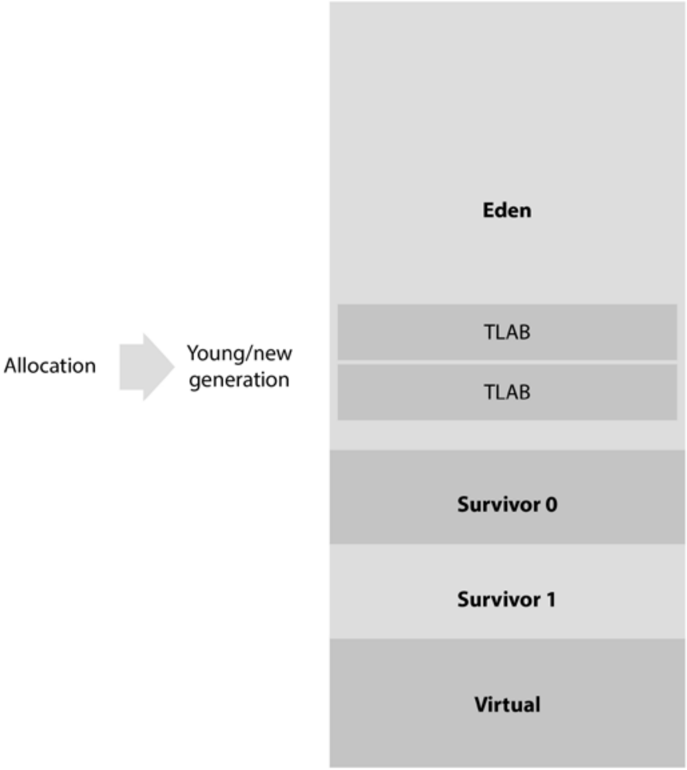


Figure 6: Object allocation into Eden

Object allocation via a TLAB is a very cheap operation; it simply bumps a pointer for the object size, which takes around 10 instructions on most platforms. As noted in the introduction, heap memory allocation for Java is cheaper than using `malloc()` from the C runtime. However, whilst individual object allocation is cheap, the rate at which minor collections occur is directly proportional to the rate of object allocation. In HotSpot, minor collections are still stop-the-world events. This is becoming a significant issue as our heaps get larger with more live objects.

Serial collector

The Serial collector (`-XX:+UseSerialGC`) is the simplest collector and is a good option for single-processor systems.

Serial GC has the smallest footprint of any collector. It uses a single thread for both minor and major collections. It uses a copying collector for the young generation, and a mark/sweep collection algorithm for the old and, where applicable, permanent generations.

The sweep phase identifies garbage, and the collector then performs sliding compaction, moving the objects towards the beginning of the old/permanent-generation space, and leaving any free space in a single contiguous free chunk at the opposite end. The compaction allows any future allocations to use the fast bump-the-pointer technique.

You might imagine that the Serial collector isn't that useful any more, but this isn't the case. The Serial collector is the collector of choice for most applications that are run on client-style machines that do not require low pause times. As a rough guide, at the time of writing, the Serial collector can manage heaps of around 64 MB with worst-case pauses of less than half a second for full collections. Perhaps more surprisingly, it is finding some server-side use in cloud environments. Speaking at JavaOne in 2014, Christine Flood, a principal software engineer at Red Hat, said:

"The Open Shift guys at Red Hat, who want to use a minimal amount of resources and a constrained footprint, are pretty happy with Serial GC right now. They can give their program a small, fixed amount of memory, and they can give it a goal, saying, 'I want you to come back here when you can.' So if your program ramps up your data, and then goes down again, Serial GC is really good at giving that memory back so that another JVM can use it."[2]

Parallel collector

The Parallel collector is the default server-side collector. It uses a monolithic, stop-the-world copying collector for the new generation, and a monolithic, stop-the-world mark/sweep for the old generation. It has, though, no impact on a running application until a collection occurs.

It comes in two forms: Parallel and Parallel Old. The Parallel collector (`-XX:+UseParallelGC`) uses multiple threads to run a parallel version of the young-generation-collection algorithm used by the Serial collector. It is still a stop-the-world copying collector, but performing the young-generation collection in parallel, using many threads, decreases garbage-collection overhead and hence increases application throughput. Since young-generation collection is very efficient, this stop-the-world

[2] https://www.parleys.com/play/543f8a2ce4b08dc7823e5418/about

pause typically completes very quickly, in hundreds of milliseconds or less.

Old-generation garbage collection for the Parallel collector is done using the same serial mark/sweep collection algorithm as the Serial collector, unless the Parallel Old collector (`-XX:+UseParallelOldGC`) is enabled. The Parallel Old collector has been the default since Java 7u4, and uses multiple threads for both minor and major collections. Objects are allocated in the tenured space using a simple bump-the-pointer algorithm. Major collections are triggered when the tenured space is full.

The Parallel Old collector is particularly suitable for tasks such as batch processing, billing, payroll, scientific computing, and so on — essentially anywhere where you might want a large heap and are happy to trade higher overall throughput for the occasional long pause.

> The Parallel collector is very good at what it does. On multi-processor systems, it will give the greatest throughput of any collector. If all you care about is end-to-end performance, Parallel GC is your friend.

The cost of collecting the old generations is affected to a greater extent by the number of objects that are retained than by the size of the heap. Therefore, the efficiency of the Parallel Old collector can be increased to achieve greater throughput by providing more memory and accepting larger, but fewer, collection pauses.

The Parallel collector only processes weak and soft references during a stop-the-world pause, and is therefore sensitive to the number of weak or soft references a given application uses.

Concurrent Mark Sweep (CMS)

HotSpot's Concurrent Mark Sweep or CMS (`-XX:+UseConcMarkSweepGC`) is a mostly concurrent generational collector that attempts to reduce, or delay, the old-generation pauses.

It uses the same parallel, monolithic, stop-the-world copying collector for the young generation as the Parallel collector does.

It has a mostly concurrent multipass marker that marks the heap while the mutator is running. Since CMS runs the mark phase concurrently, the object graph is changing whilst marking is happening. This results in an obvious race called, with no prizes for originality, the "concurrent marking race". To understand the problem, imagine that the mutator takes a reference that the collector hasn't seen yet and copies that reference into a place that the collector has already visited. As far as the collector is concerned, it already has this covered, so it never sees the reference, doesn't mark the object as alive, and thus the object gets collected during the sweep phase, corrupting the heap. This race must somehow be intercepted and closed.

There are two ways to deal with this problem: incremental update and "snapshot at the beginning" (SATB). CMS uses SATB, which takes a logical snapshot of the set of live objects in the heap at the beginning of the marking cycle. This algorithm uses a pre-write barrier to record and mark the objects that are a part of the logical snapshot.

HotSpot already has a generational collector and also has a blind write barrier to track every store of every reference. This means that all mutations are already tracked, and the card table reflects mutations. So if, during marking, we clean the card table in some way, then whatever accumulated in the card table whilst we were marking is stuff that was changed, and the collector can revisit these and repeat the marking. Of course, whilst it's doing this, the object graph is still shifting, so it has to repeat the process.

Eventually the collector will decide that the amount of work left to do is small enough that it can perform a brief stop-the-world pause to catch up and be done. As with the Parallel collector, CMS processes all weak and soft references in the stop-the-world pause, so programs that make extensive use of weak and soft references can expect to see a longer stop-the-world pause when using CMS.

This approach to concurrent marking works most of the time but, of course, if you are mutating the heap faster than the collector can keep up, it won't work. Because of this, the CMS collector, and almost any collector that has a concurrent marker, is sensitive to the mutation or transaction rate of your application. If CMS can't keep up with the mutator the problem will show up in the GC logs as a "concurrent mode failure".

CMS also has a concurrent sweep phase. Sweeping is actually fairly easy to do concurrently since, in marked contrast to any zombie film you may have watched, dead stuff doesn't come back to life, not on the JVM anyway.

Concurrent sweeping is implemented using a free list, and CMS attempts to satisfy old-generation allocations from this free list. However the free list is not compacted, so free space will inevitably fragment, and CMS falls back to a full stop-the-world pause to compact the heap when this happens.

When CMS does a promotion to tenured space, it again makes use of the free list, putting the objects into a place on the free list and recycling the memory. This works for a while. However, if you see the "promotion failure" message in the CMS GC log, or a pause of more than a second or two, then a promotion has failed. CMS will fall back to a full stop-the-world monolithic collection when it fails to promote objects because either the tenured space is too fragmented or it fails a concurrent load operation with a marker.

That CMS is mostly concurrent with the application has some other implications you should be aware of. First, CPU time is taken by the collector, thus reducing the CPU available to the application. The amount of time required by CMS grows linearly with the amount of object promotion to the tenured space. In addition, for some phases of the concurrent GC cycle, all application threads have to be brought to a safe point for marking GC roots and performing a parallel re-mark to check for mutation.

To sum up then, CMS makes a full GC event less frequent at the expenses of reduced throughput, more expensive minor collections, and a greater footprint. The reduction in throughput can be anything from 10%-40% compared to the Parallel collector, depending on promotion rate. CMS also requires a 20% greater footprint to accommodate additional data structures and floating garbage that can be missed during the concurrent marking and so gets carried over to the next cycle.

High promotion rates, and resulting fragmentation, can sometimes be reduced by increasing the size of both the young and old-generation spaces.

In the next part of the book, we'll take a look at Garbage First (or G1), which was intended to be a replacement for CMS in most cases, IBM's Balanced collector, and Azul's C4 pause-less commercial collector. We'll also look at IBM's Metronome real-time collector, and Red Hat's forthcoming Shenandoah.

PART THREE

Multi-region collectors

“Here is Edward Bear, coming down the stairs now, bump, bump, bump, on the back of his head, behind Christopher Robin. It is, as far as he knows, the only way of coming downstairs, but sometimes he feels that there really is another way, if only he could stop bumping for a moment and think of it. And then he feels that perhaps there isn’t.”

— *A. A. Milne,* Winnie-the-Pooh

Heap structure

The collectors in this part of the book use a hybrid heap structure. Here the heap is based on logical as opposed to physical generations, specifically a collection of non-contiguous regions of the young generation and a remainder in the old generation. A distinct advantage of this approach is that neither the young nor the old generation have to be contiguous, allowing for a more dynamic sizing of the generations. If it has a humongous object to process — an object that is larger than one of the regions — it can grab two or three adjacent regions and allocate the object to these.

The figure below, based on the Garbage First collector from Oracle, illustrates how this works.

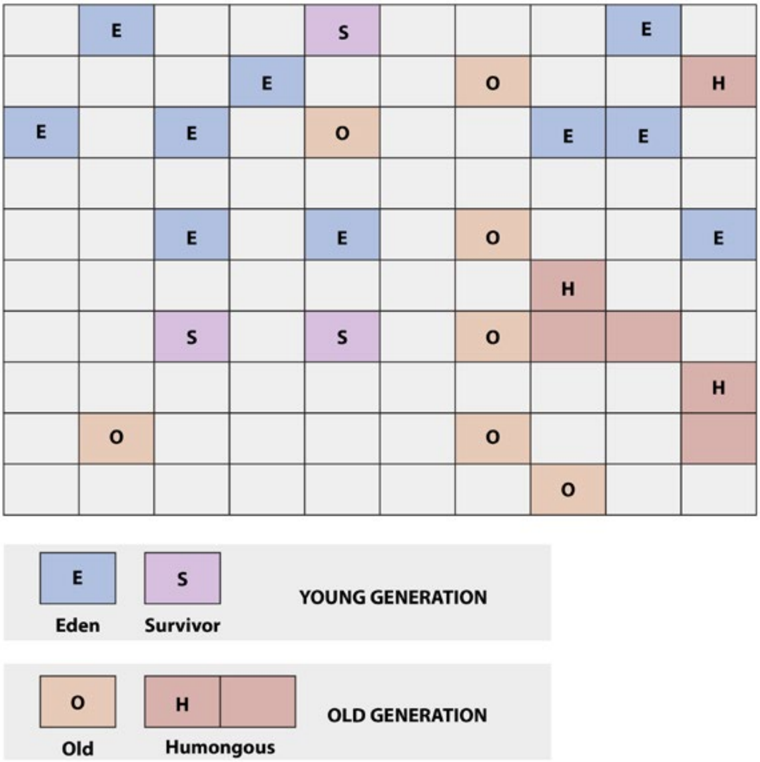


Figure 7: Garbage First heap structure

Garbage First

The first version of G1 (XX:+UseG1GC) was non-generational, but the current version is a generational collector that groups objects of a similar age. G1 is still comparatively new and thus in only limited deployment so far in the real world. Like CMS, G1 attempts to address the old-generation pauses. It also uses a monolithic, stop-the-world collector for the young generation. It has a mostly concurrent marker that is very similar to the one used in CMS and it has a concurrent sweep. It uses many other techniques we've already looked at — for example, the concepts of allocation, copying to survivor space, and promotion to old generation are similar to previous HotSpot GC implementations. Eden and survivor regions still make up the young generation. It uses SATB to close the concurrent marking race.

What's different is the way it handles compaction. Evacuation pauses exist to allow compaction, since object movement must appear atomic to mutators. I noted in part 1 that this atomicity makes compaction a major operation since, whilst moving objects is straightforward, we then have to remap all object references to point to new object locations. Both G1 and the Balanced collector from IBM use a technique called "incremental compaction", the purpose of which is to avoid doing a full GC as much as possible.

Incremental compaction exploits an interesting quality, which is that some regions of memory are more popular than others. The collector tracks pointers between regions — effectively a massive remembered set of what points to what recorded as regions rather than as objects. If it turns out that nothing in the heap points to a given region, then you can compact that one region and you don't have to remap at all. More commonly, when only a small number of stack frames point to a given region, you only have to remap a subset of the heap.

The remembered sets can get pretty large. Each region has an associated remembered set, which indicates all locations that might contain pointers to (live) objects within the region. Maintaining these remembered sets requires the mutator threads to inform the collector when they make pointer modifications that might create inter-region pointers.

This notification uses a card table (basically a hash table) in which every 512-byte card in the heap maps to a 1-byte entry in the table. Each thread has an associated-remembered-set log, a current buffer or sequence of modified cards. In addition, there is a global set of filled RS buffers.

Because of parallelism, each region has an associated array of several such hash tables, one per parallel GC thread, to allow these threads to update remembered sets without interference. The logical contents of the remembered set is the union of the sets represented by each of the component hash tables.

The remembered-set write barrier is performed after the pointer write. If the write creates a pointer from an object to another object in the same heap region, it is not recorded in a remembered set.

In many respects, G1 looks and feels like the other generational collectors we've looked at, but it behaves rather differently. For one thing, G1 has a pause prediction model built into the collector. When it goes through the mark phase of the collection, it calculates the live occupancy of a particular region and uses that to determine whether the region is ripe for collection. So when it goes to collect, it puts all of the young regions into the collection set (generally referred to as the "CSet") and then looks at how long it's going to take to collect all of these regions; if there is time left in the budget, based on the soft-defined pause-time goal, it can add some of the old-generation regions to the CSet as well. The pause-time goal here is a hint, not a guarantee — if the collector can't keep up, it will blow the pause-time goal.

In the original, non-generational version, the remembered set could end up larger than the heap itself in some circumstances. Going generational largely solved this problem, since the collector no longer has to keep track of the pointers from the old-generation objects into the newly allocated objects, because it has those via the card table.

G1 still has a couple of issues, however. The first is that the hypothesis on which it is based (that some regions are more popular than others) isn't always true, and the technique only works well if you have at least some non-popular regions. Moreover, as the heap grows, the popularity of regions grows. To illustrate this, imagine I have a heap with 100 regions and 20% of things point into any region, so 20 regions are pointing to my region. If I then increase the heap size and have a heap with 1,000 regions then I have 10 times

In my testing, I found that G1 sometimes performed better than CMS, was sometimes on a par, and was sometimes noticeably worse. In other words, as with everything here, the best bet is to benchmark it with your own application.

as many regions pointing into any one region. The complexity, and the amount of work I have to do, both grow considerably.

G1 is a good general-purpose collector for larger heaps that have a tendency to become fragmented, assuming your application can tolerate pauses of 0.5-1.0 second for incremental compactions. G1 tends to reduce the frequency of the worst-case, fragmentation-induced pauses seen by CMS at the cost of extended minor collections and incremental compactions of the old generation. Most pauses end up constrained to regional rather than full-heap compactions.

Balanced

IBM's WebSphere Application Server version 8 introduced the new region-based Balanced garbage collector. Though developed independently, it is similar to G1, at least at a high level. You enable it through the command line option `-Xgcpolicy:balanced`. The Balanced collector aims to even out pause times and reduce the overhead of some of the costlier operations typically associated with garbage collection.

Like G1, objects in a single region share certain characteristics, such as all being of a similar age. The region size is always a power of two (for example 512 kB, 1 MB, 2 MB, 4 MB, and so on) and is selected at startup based on the maximum heap size. The collector chooses the smallest power of two which will result in fewer than 2,048 regions, with a minimum region size of 512 kB. Except for small heaps (less than about 512 MB), the JVM aims to have between 1,024 and 2,047 regions.

Aside from arrays, objects are always allocated within the bounds of a single region so, unlike G1, the region size imposes a limit on the maximum size of an object. An array which cannot fit within a single region is represented using a discontiguous format known as an arraylet. Large array objects appear as a spine, which is the central object and the only entry that can be referenced by other objects. Actual array elements are then held as leaves which can be scattered throughout the heap in any position and order.

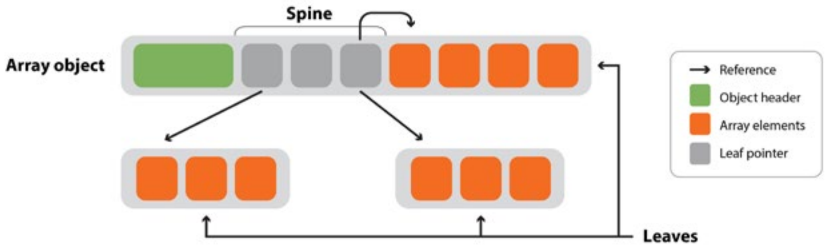


Figure 8: Balanced arraylet structure

The arraylet structure is also used in IBM's Metronome collector in the WebSphere Real Time product.

The use of arraylets has some consequences, particularly in JNI code. The JNI provides APIs called `GetPrimitiveArrayCritical`, `ReleasePrimitiveArrayCritical`, `GetStringCritical`, and `ReleaseStringCritical` that provide direct access to array data where possible. The Balanced collector does offer direct access to contiguous arrays but is required to copy the array data into a contiguous block of off-heap memory for discontiguous arraylets, as the representation of arraylets is inconsistent with the contiguous representation required by these APIs. You can check if this is happening via the `isCopy` parameter in the JNI API. If the API sets this to `JNI_TRUE`, then the array was discontiguous and the data was copied. Aside from rewriting the JNI code in Java, your only option would be to increase the size of the heap.

Balanced has three different types of collection cycle:

- **Partial garbage collection** is the most common, and collects a subset of regions known as the collection set. It is the equivalent of a young-generation collection. This is a stop-the-world operation.
- **The global mark phase** incrementally traces all live objects in the heap, but is not responsible for returning free memory to the heap or compacting fragmented areas of the heap. The phases are scheduled approximately halfway between each partial collection.
- **Global garbage collection** marks, sweeps, and compacts the whole heap in a stop-the-world fashion. Global collection is normally used only when the collector is explicitly invoked through a `System.gc()` call by the application. It can also be performed in very tight memory conditions as a last resort to free up memory before throwing an OutOfMemoryError.

Balanced uses a remembered set to track all inbound references on a per-region basis. It creates and discovers references during program execution through a write barrier. Off-heap memory is reserved for remembered-set storage and is typically no more than 4% of the current heap size.

When to use Balanced

The Balanced collector has about a 10% overhead in terms of throughput when compared to IBM's conventional Generational Concurrent Garbage Collector (`-Xgcpolicy:gencon`). It is best suited for a heap larger than 4 GB and requires a 64-bit platform.

Balanced is specifically designed for non-uniform memory access (NUMA), a hardware architecture in which the processors and memory are organised into groups called nodes. Processors can access memory local to their own node faster than memory associated with other nodes.

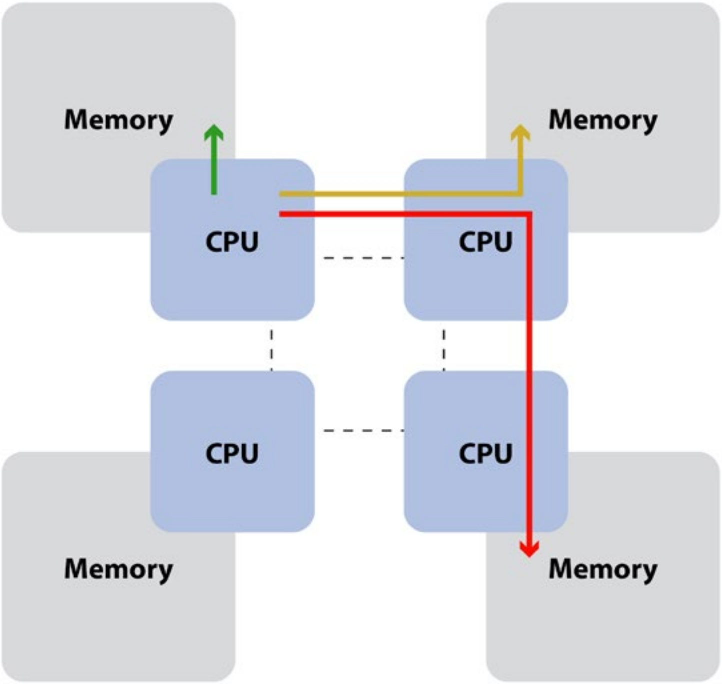


Figure 9: NUMA architecture

Java can perform very poorly if an application doesn't sit within one single NUMA region because the collector assumes it has random access to the memory, so if a GC goes across NUMA regions, it can suddenly slow down dramatically and will also perform erratically.

Balanced deals with this by binding each heap region to one of the system's NUMA nodes on startup. The heap is divided as evenly as possible so that all nodes have approximately the same number of regions.

Most threads in the JVM are similarly bound, or "affinitized", to NUMA nodes in a simple round-robin manner: the first thread is bound to the first node, the second thread to the second node, and so on. Some threads, such as the main thread and threads attached through JNI, might not be bound to a specific node.

As threads allocate objects, they attempt to place those objects in their local memory for faster access. If there is insufficient local memory, the thread can borrow memory from other nodes. Although this memory is slower, it is still preferable to an unscheduled GC or an OutOfMemoryError.

During garbage collection, each parallel GC thread gives preference to work associated with its own node. This results in shorter pauses. The collector also moves objects so that each one is stored in memory local to the thread using the object, whenever possible. This can result in improved application throughput.

Metronome

IBM's Metronome is an incremental mark-sweep collector with partial on-demand compaction to avoid fragmentation. It uses a deletion write-barrier, marking live any object whose reference is overwritten during a write. After sweeping to reclaim garbage, Metronome compacts if necessary to ensure that enough contiguous free space is available to satisfy allocation requests until the next collection. Like Shenandoah, Metronome uses Brooks-style forwarding pointers, imposing an indirection on every mutator access.

The original Metronome was non-generational but the current version does have a generational filter, using the nursery as a way to reduce the rate of allocation into the primary heap. The heap is region-based with fixed size regions divided by region size.

Metronome guarantees the mutator a predetermined percentage of execution time, with any remaining time given to the collector to use as it sees fit. By maintaining uniformly short collection pause times, Metronome is able to give finer-grained utilisation guarantees than traditional collectors. Metronome sets a default utilisation target for the mutator of 70%, but this can be further tuned to allow an application to meet its space constraints. Of course, when the collector is not active, the mutator can get 100% utilisation.

Metronome uses a number of techniques to achieve deterministic pause times. Like the Balanced collector, it uses arraylets to allow allocation of arrays in multiple chunks. It also uses a Brooks-style read barrier to ensure that the overhead for accessing objects has a uniform cost even if the collector has moved the objects. This can be expensive, so Metronome reduces the cost by using an eager read-barrier that forwards all references as they are loaded from the heap to make sure that they always refer to to-space, meaning that access via these references incurs no indirection overhead. This approach is combined with a number of specialist compiler optimisations to further reduce overhead.

Metronome's approach is based on dividing the time that consumes GC cycles into a series of increments IBM calls "quanta". Metronome uses a series of short, incremental pauses to complete each collection cycle, but these are still stop-the-world events and depend on all mutator threads being at a GC-safe point. It scans each complete stack thread within a single collection quantum to avoid losing pointers to objects. This means that whilst it generally requires no code changes to use, developers do need to take care not to use deep stacks in their real-time applications so as to permit each stack to be scanned in a single pass.

Metronome uses time-based scheduling with two different threads. The alarm thread is a very high priority thread (higher than any mutator thread). It wakes up every 500 milliseconds, acting as a heartbeat for deciding whether to schedule a collector quantum. A second collector thread performs that actual collection for each quantum. Metronome has consistent CPU utilisation, but the use of time-based scheduling is susceptible to variations in memory requirements if the mutator allocation rate varies over time.

Metronome has latency advantages in most cases over CMS and G1, but may see throughput limitations and still suffers significant stop-the-world events. It can achieve single-digit millisecond pause times via constrained allocation rates and live set sizes.

C4

Azul's C4 collector, included in their HotSpot-based Zing JVM[1], is both parallel and concurrent. It has been widely used in production systems for several years now and has been successful at removing, or significantly reducing, sensitivity to the factors that typically cause other concurrent collectors to pause. Zing is currently available on Linux only, and is commercially licensed.

C4 is a generational collector, but this is essentially an efficiency measure, not a pause-containment measure; C4 uses the same GC mechanism for both the new and old generations, working concurrently and compacting in both cases. Most importantly, C4 has no stop-the-world fallback. All compaction is performed concurrently with the running application.

Since a number of aspects of C4 are genuinely novel, I'll spend some time examining them.

The loaded value barrier

Core to C4's design, the collector uses what Azul CTO Gil Tene calls a "loaded value barrier", a type of read barrier that tests the values of object references as they are loaded from memory and enforces key collector invariants. The loaded value barrier (LVB) effectively ensures that all references are "sane" before the mutator ever sees them, and this strong guarantee is responsible for the algorithm's relative simplicity.

The LVB enforces two critical qualities for all application-visible references:

1. The reference state indicates that it has been "marked through" if a mark phase is in progress. (See mark phase description below.)

2. The reference is not pointing to an object that has been relocated.

If either of the above conditions is not met, the loaded value barrier will trigger a "trap condition", and the mutator will correct the reference to adhere to the required invariants before it becomes visible to application code. The use of the LVB test in combination with self-healing trapping (see below) ensures safe single-pass marking, preventing the application thread from causing live references to escape the collector's reach. The same barrier and trapping mechanism combination is also responsible for

[1] http://www.azulsystems.com/products/zing/

supporting lazy, on-demand, concurrent compaction (both object relocation and reference remapping), ensuring that application threads will always see the proper values for object references without ever needing to wait for the collector to complete either compaction or remapping.

Self-healing

Key to C4's concurrent collection is the self-healing nature of handling barrier trap conditions. This feature dramatically lowers the time cost of C4's LVB (by orders of magnitude) compared to other types of read barrier. C4's LVB is currently the only read barrier in use in a production JVM, and its self-healing qualities are one of the main reasons for its evident viability in a high-throughput, ultra-low-pause collector. When a LVB test indicates that a loaded reference value must be changed before the application code proceeds, the value of both the loaded reference and of the memory location from which it was loaded will be modified to adhere to the collector's current invariant requirements (e.g. to indicate that the reference has already been marked through or to remap the reference to a new object location). By correcting the cause of the trap in the source memory location (possible only with a read barrier, such as the LVB, that intercepts the source address), the GC trap has a self-healing effect: the same object references will not re-trigger additional GC traps for this or other application threads. This ensures a finite and predictable amount of work in a mark phase, as well as the relocate and remap phases. Azul coined the term "self-healing" in their first publication of the pauseless GC algorithm in 2005, and Tene believes this self-healing aspect is still unique to the Azul collector.

How the Azul algorithm works

The Azul algorithm is implemented in three logical phases as illustrated below:

1. **Mark:** responsible for periodically refreshing the mark bits.

2. **Relocate:** uses the most recently available mark bits to find pages with little live data, to relocate and compact those pages, and to free the backing physical memory.

3. **Remap:** updates (forwards) every relocated pointer in the heap.

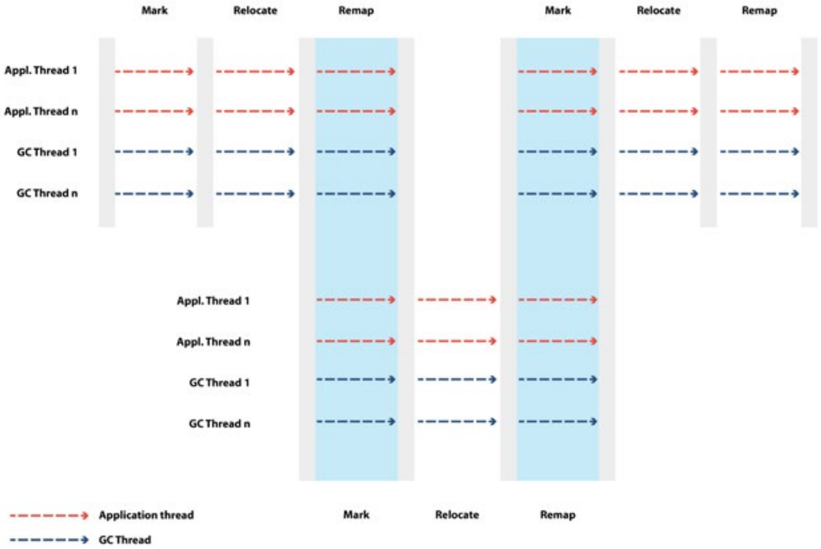


Figure 10: Azul C4 algorithm overview

The mark phase

The C4 mark phase has been characterised by David Bacon et al. as a "precise wavefront" marker,[2] not SATB, augmented with a read barrier. The mark phase introduces and tracks a "not marked through" (NMT) state on object references. The NMT state is tracked in metadata information in each reference, and the LVB verifies that each reference's state matches the current GC cycle's expected NMT state. The invariant the LVB imposes on the NMT state eliminates the possibility that the application threads would cause live references to escape the collector's reach, allowing the collector to guarantee robust and efficient single-pass marking.

At the start of the mark phase, the marker's work list is primed with a root set that includes all object references in application threads. As is common to all markers, the root set generally includes all refs in CPU registers and on the threads' stacks. Running threads collaborate by marking their own root set, while blocked (or stalled) threads get marked in parallel by the collector's mark-phase threads.

Rather than use a global, stop-the-world safe point (where all application threads are stopped at the same time), the marker algorithm uses a check-

[2] http://www.srl.inf.ethz.ch/papers/pldi06-cgc.pdf

point mechanism. Each thread can immediately proceed after its root set has been marked (and expected-NMT flipped) but the mark phase cannot proceed until all threads have crossed the checkpoint.

After all root sets are marked, the algorithm continues with a parallel and concurrent marking phase. It pulls live refs from the work lists, marks their target objects as live, and recursively works on their internal refs.

In addition to references discovered by the marker's own threads during the normal course of following the marking work lists, freely running mutator threads can discover and queue references they may encounter that do not have their NMT bit set to the expected "marked through" value when they are loaded from memory. Such reference loads trigger the loaded value barrier's trapping condition, at which point the offending reference is queued to ensure that it will be traversed and properly marked through by the collector. Its NMT value will therefore be immediately fixed and healed (in the original memory location) to indicate that it can be considered to be properly marked through, avoiding further LVB condition triggers.

The mark phase continues until all objects in the marker work list are exhausted, at which point all live objects have been traversed. At the end of the mark phase, only objects that are known to be dead are not marked "live" and all valid references have their NMT bit set to "marked through".

The relocate and remap phases

During the relocate phase, objects are relocated and pages are reclaimed. In the course of the phase, pages with some dead objects are made wholly unused by concurrently relocating their remaining live objects to other pages. References to these relocated objects are not immediately remapped to point to the new object locations. Instead, by relying on the loaded value barrier's imposed invariants, reference remapping can proceed lazily and concurrently after relocation, until the completion of the collector's next remap phase assures the collector that no references that require remapping exist.

During relocation, sets of pages (starting with the sparsest pages) are selected for relocation and compaction. Each page in the set is protected from mutator access, and all live objects in the page are copied out and relocated into contiguous, compacted pages. Forwarding information that tracks the location of relocated objects is maintained outside the original page, and is used by concurrent remapping.

During and after relocation, any attempt by the mutator to use references to relocated objects is intercepted and corrected. Attempts by the mutator to load such references will trigger the loaded value barrier's trapping condition, at which point the stale reference will be corrected to point to the object's proper location, and the original memory location from which the reference was loaded will be healed to avoid future triggering of the LVB condition.

Using a feature that Tene calls "quick release", the C4 collector immediately recycles memory page resources without waiting for remapping to complete. By keeping all forwarding information outside the original page, the collector is able to safely release physical memory immediately after the page contents have been relocated and before remapping has been completed. A compacted page's virtual-memory space cannot be freed until no more stale references to that page remain in the heap (which will only be reliably true at the end of the next remap phase) but the physical-memory resources backing that page are immediately released by the relocate phase and recycled at new virtual-memory locations as needed. The quickly released physical resources are used to satisfy new object allocations as well as the collector's own compaction pipeline. With "hand over hand" compaction along with quick release, the collector can use page resources released by the compaction of one page as compaction destinations for additional pages, and the collector is able to compact the entire heap in a single pass without requiring prior empty target memory to compact into.

During the remap phase, which follows the relocate phase, GC threads complete reference remapping by traversing the object graph and executing a LVB test on every live reference found in the heap. If a reference is found to be pointing to a relocated object, it is corrected to point to the object's proper location. Once the remap phase completes, no live heap reference can exist that would refer to pages protected by the previous relocate phase, and at that point the virtual memory for those pages is freed.

Since the remap phase traverses the same live object graph as a mark phase would, and because the collector is in no hurry to complete the remap phase, the two logical phases are rolled into one in actual implementation, known as the "combined mark and remap phase". In each combined mark/remap phase, the collector will complete the remapping of references affected by the previous relocate phase and at the same time perform the marking and discovery of live objects to be used by the next relocate phase.

C4 is designed to remain robust across a wide operating range. Through the use of a guaranteed single-pass marker, the collector is completely insensitive to mutation rates. By performing concurrent compaction of the entire heap, the collector is insensitive to fragmentation. By performing weak, soft, and final reference processing concurrently, the collector has been made insensitive to the use of such features by applications. By using quick release, and through the nature of the loaded value barrier's self-healing properties, the collector avoids the sensitivity of rushing to complete a phase of its operation for fear that its efficiency might drop, or that it may not be able to complete a phase or make forward progress without falling back to a stop-the-world pause.

Shenandoah

Red Hat's Shenandoah is, at time of writing, still under development. It is open source, and is expected to be incorporated into OpenJDK via JEP 189[3] at some point in the future — most likely Java 10 or later.

The collector uses a regional heap structure similar to that in G1 and C4, but with no generational filter applied, at least in the current version. Project lead Christine Flood told me that this might change in the future, although development on a generational version hasn't started at the time of writing:

"The initial thinking was that we didn't need to treat young-generation objects differently. The garbage collector would pick the regions with the most space to reclaim, regardless of whether the application fit the generational hypothesis or not. This was based on a survey of currently trendy applications that claimed that they had mostly medium-lived objects, which would not benefit from a generational collector. However, SpecJVM is still important, and it does behave in a generational manner, so we are reconsidering whether it makes sense to make a generational Shenandoah. It's basically added complexity for a payoff for some applications."

[3] http://openjdk.java.net/jeps/189

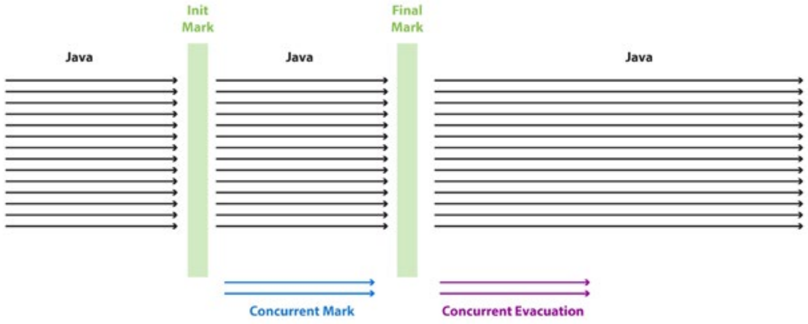


Figure 11: Shenandoah algorithm overview

Shenandoah is intended for large heaps, those greater than 20 GB. It is also expected to carry a considerable performance penalty (of the order of 10% or so) over current collectors on applications with no GC pauses and no perceivable performance penalty when running any of the current collectors. It isn't expected to be pauseless in version 1, though this is a stated aim of the project. It will, though, have a very low pause time. Speaking at StrangeLoop, Flood said that the target was for "less than 10-ms GC pause times for 100+ GB heaps," however if "the reality is you have a thousand threads and they all have really big stacks, you are not going to hit the 10-ms pause time; at least not in the current Shenandoah."[4]

It uses as many threads as are available to do concurrent phases.

The mark phase

Shenandoah is a mark/copy collector. In the first phase, it marks all the live objects in the heap. Starting from the GC roots, it follows the object graph and marks each object it encounters, updating the containing region's live counter accordingly. The primary purpose of the mark phase is to calculate the number of live objects in each region.

The mark phase starts with a brief stop-the-world event in order to scan the GC roots, then continues concurrently with the mutator. Like G1, it uses the snapshot-at-the-beginning technique. Like most modern collectors, it uses multiple threads during the mark phase, and it concurrently processes the SATB queues as it goes along. The final mark does another

[4] https://www.youtube.com/watch?v=QcwyKLlmXeY

pass over the root set to make sure everything currently in the root set is marked live. This may require some draining of SATB queues, but most of that work should have already been done. The collector only pushes unmarked objects onto a queue to be scanned; if an object is already marked, the collector doesn't need to push it.

Each thread maintains a work-stealing queue. Each item in the queue is a heap object. Initially, the queues get filled with objects referenced by the GC roots. Working in a loop, each GC thread then pops an object from the queue, marks it, increases the containing region's liveness counter, then checks all reference fields of that object and pushes each object that is referenced back to the thread's queue. The per-region liveness counters are thread-local and get combined during a stop-the-world pause.

Eventually, a thread will run out of objects. When that happens, the GC work thread attempts to steal from another thread's queue. When all threads are done, i.e. no thread can steal from any other thread any more, the mark phase is complete.

After the concurrent marking phase is done, Shenandoah stops the world and processes the SATB list by marking all objects in it, traversing their references where necessary.

After the whole marking phase is done, the collector knows for each region how much live data (and therefore garbage) each region contains, and all reachable objects are marked as such.

The evacuation phase

During the evacuation phase, Shenandoah selects the regions that it wants to evacuate ("from" regions) and corresponding regions that the objects are to be copied to ("to" regions). It then scans the "from" regions and copies all objects that are marked to "to" regions that have enough empty space. After evacuation, the "from" regions only contain garbage objects; once all references to those objects are removed (in a subsequent marking phase), these regions will be freed to satisfy future allocations. Like the concurrent marking phase, the evacuation phase uses work stealing to use all available threads.

The evacuation phase makes use of Brooks forwarding pointers.[5] The idea is that each object on the heap has one additional reference field. This field either points to the object itself or, as soon as the object gets

[5] http://www.memorymanagement.org/glossary/f.html#forwarding.pointer

copied to a new location, to its new location. This will enable it to evacuate objects concurrently with mutator threads.

Specifically, the collector reads the forwarding pointing to the "from" region, and allocates a temporary copy of the object in a "to" region. It then speculatively copies the data and compares and swaps (CASs) the forwarding pointer to the new copy. If the CAS operation succeeds, it just carries on. If it fails, another thread has already copied the object, so it rolls back the allocation in the thread-local allocation buffer and carries on.

When evacuation is done, the collector needs to update all references to objects to point to the "to" space locations. This is done by piggybacking on the concurrent marking phase. When the collector traverses the heap for marking, it sees all the live objects and references, and whenever it visits an object, it updates all its object references to point to the new locations of the referents (i.e. the objects themselves as opposed to the pointers to them).

Shenandoah can either do a third stop-the-world phase to initiate an update-references phase or can wait until the next concurrent marking to update references. In either case, it can't reuse the "from" regions until it updates all references to them.

Known trade-offs

Since Shenandoah is very much in development, not all the trade-offs are yet known. However, we can list some strengths and weaknesses as it currently stands.

The use of forwarding pointers has some significant advantages. Shenandoah does not need a remembered set. As we saw in our discussion on G1, remembered sets can become very large but, perhaps more significantly, updating a card table can hinder parallelism in some cases because independent threads may still need to touch the same region in the card table.

However, the use of forwarding pointers necessitates the use of more heap space (one word per object, which can get expensive if you have a lot of small objects), and it needs more instructions to read and write objects, with both read and write barriers thus adding a performance penalty. This is necessary because as soon as there are two copies of an object (one in "from" space, one in "to" space), you have to be careful to maintain consistency, and therefore any changes to objects need to happen in the "to" space copy. This is achieved by putting barriers in various places, specifically everything that writes to objects or runtime code that uses objects, which then resolve the object before writing into it or using it; if this isn't done you could end up with some threads using the old copy and some threads using the new copy. The barrier simply reads the Brooks pointer field and returns the forwarded object to any code that uses it. In terms of machine-code instructions, this means one additional read instruction for each write operation into the heap (and some read operations by the runtime).

The read barrier is straightforward and lightweight. It is an unconditional read barrier. All it does is read the address above the pointer before the pointer is read, and then uses that to read the field:

```
movq R10, [R10 + #-8 (8-bit)] # ptr
movq RBX, [R10 + #16 (8-bit)] # ptr ! Field: java/lang/
ClassLoader.parent
```

The collector also needs an SATB write barrier, which adds previous reference values to a queue to be scanned, and write barriers on all writes (even base types) to ensure objects are copied in targeted regions before they are written to.

Since CAS involves a write operation, which can only happen in "to" space, a CAS operation is expensive for Shenandoah. Three things have to be copied from "from" space to "to" space: the original object, the new object, and the compare value. Copies are, however, bounded by region size, with objects larger than a region handled separately.

At the time of writing, Shenandoah shows some promise. It has some demonstrable prototypes but these are not yet ready for comparative performance measurements. It will be interesting to see what the metrics look like with regard to sustainable max-pause times, allocation and mutation throughput, and mutator execution speed during collection for the working and fully correct collector implementation.

It is also worth saying that it generally takes considerable time to get a collector working correctly — both CMS and G1 took around 10 years.

PART FOUR

General monitoring and tuning advice

"Alice came to a fork in the road.
'Which road do I take?' she asked.

'Where do you want to go?'
responded the Cheshire Cat.

'I don't know,' Alice answered.

'Then,' said the Cat, 'it doesn't matter'."

— *Lewis Carroll,* Alice in Wonderland

Any tuning exercise is going to be specific to a particular application and scenario, so it's important to realise that whilst I can provide you with some ideas as to what flags to look at, tools to use, and so on, this isn't intended to be everything you need to know about tuning; I'm not convinced that such a thing is even possible.

Instead, this part of the book will examine the three major attributes of garbage-collection performance, how to select the best collector for your application, fundamental garbage-collection principles, and the information to collect when tuning a HotSpot garbage collector.

The major attributes

At the top level, any GC-tuning exercise involves a trade-off around three different attributes:

- **Throughput:** a measure of the garbage collector's ability to allow an application to run at peak performance, without regard for the duration of garbage-collection pause times or the amount of memory it requires.
- **Latency:** a measure of the garbage collector's ability to allow an application to run with little or no GC-induced pause time or jitter.
- **Footprint:** a measure of the amount of memory a garbage collector requires to operate efficiently.

A performance improvement in any one of these attributes almost always comes at the expense of one or both of the other attributes. For example, for all the concurrent collectors targeting latency, you have to give up some throughput and gain footprint. Depending on the efficiency of the concurrent collector, you may give up only a little throughput, but you are always adding significant footprint. For an operation that is truly concurrent with few stop-the-world events, you would need more CPU cores to maintain throughput during the concurrent operation.

However, for most applications, one or two of these are more important than the others. Therefore, as with any performance-tuning exercise, you need to decide which is the most important for you before you start, agree on targets for throughput and latency, and then tune the JVM's garbage collector for two of the three.

Getting started

Once you've got targets, start by developing a set of representative load tests that can be run repeatedly. A mistake that I've seen several organisations make is to have the development team run performance testing and tune the system before releasing it to production for the first time, and then never look at it again. Obtaining optimum performance is not a one-time task; as your application and its underlying data changes, you will almost certainly need to repeat the exercise regularly to keep the system running correctly.

> With a set of performance tests, you can start by benchmarking your application, and then resort to tuning if you need to.

Choosing a collector

The following is a simple approach to choosing a collector:

1. If you don't have specific non-functional requirements to do with latency, stick with the Parallel collector. As we noted before, it has the best throughput of any collector we've looked at, though we should also observe that it will, in practice, also impose a small heap size on you — of the order of 1-2 GB.

2. If you do have specific latency requirements or need a larger heap:

   A. Benchmark your application with the Parallel collector in case it's okay, and as a useful reference as you look at other collectors.

   B. Benchmark your application using CMS. If CMS meets your requirements, stop there.

   C. If CMS doesn't hit your performance goals, try benchmarking with G1.

   D. If G1 also doesn't hit your performance goals, you need to decide whether to spend time and effort tuning your application and collector to hit your requirements, or try a low-pause-time collector.

Low-pause-time and real-time options

Besides Azul C4 and IBM's real-time collectors, Oracle also has a real-time collector: JRockit Real Time, which is still available as part of Oracle Java SE Suite, but whose use is mostly limited to those who were JRockit users. Donald Smith, Oracle's Java product manager told me, *"We have no plans or roadmaps to announce at this time, but our intent is to bring RT features back into HotSpot and the converged Oracle JDK at some time in the future."*

The C4 collector used in Azul's Zing product is, as far as I know, the only Java collector currently available that can be truly concurrent for collection and compaction while maintaining high throughput for all generations. As we've seen, it does, though, need specific OS-level hooks in order to work. It's perhaps also worth pointing out that although the published C4 algorithm is properly pauseless (in that it requires no global stop-the-world safe point), the actual GC implementation of C4 in Zing does include actual pauses that deal with other JVM-related details and phase changes. While C4 will pause up to four times per GC cycle, these stop-the-world events are used for phase shifts in the collection cycle and do not involve any work that is related to live-set size, heap size, allocation rate, or mutation rate. The resulting phase-shift pauses are sufficiently small to be hard to detect among the stalls that are regularly caused by the operating system itself; it is common for applications running on a well-tuned Linux system running Zing for weeks to pause for 2 ms at worst. Measured stalls larger than that are generally dominated by OS-related causes (scheduling, swapping, file-system lockups, power management, etc.) that have not been tuned out, and which would affect any process on the system.

Tuning a collector

For GC tuning, the starting point should be the GC logs. Collecting these basically has no overhead and, assuming your application is already in production, will provide you with the best information you can get. If enabling the logs does affect application per-

> I recommend enabling GC logging while running your production applications. The overhead for this is minimal and it produces invaluable tuning data.

formance then this is worth investigating in and of itself, but it's unlikely to be related to garbage collection.

The JVM-tuning decisions made in the tuning process utilise the metrics observed from monitoring garbage collections.

To understand how your application and garbage collector are behaving, start your JVM with at least the following arguments:

- `-Xloggc:<filename>`: directs the GC information to the file named <filename>.
- `-XX:+PrintGCDetails`: provides GC-specific statistics and thus varies depending on the garbage collector in use.
- `-XX:+PrintGCDateStamps`: added in Java 6 update 4, this shows the year, month, day, and time of garbage collection. If you are using an earlier version of Java, `-XX:+PrintGCTimeStamps` prints a timestamp for the garbage collection as the number of seconds since the HotSpotVM was launched.
- `-XX:+PrintTenuringDistribution`: prints tenuring-age information.
- `-XX:+PrintGCApplicationConcurrentTime`: can be used to determine whether the application was executing, and for how long, during some period of interest when an observed response time exceeds application requirements.
- `-XX:+PrintGCApplicationStoppedTime`: how long a safe-point operation blocked application threads from executing.

If you have latency spikes outside your acceptable range, try to correlate these with the GC logs to determine if GC is the issue. It is possible that other issues may be causing the latency spike, such as a poorly performing database query, or a bottleneck on the network.

Understanding the GC logs

The important things that all GC logs have in common are:

- heap size after the collection,
- time taken to run the collection, and
- number of bytes reclaimed by the garbage collection.

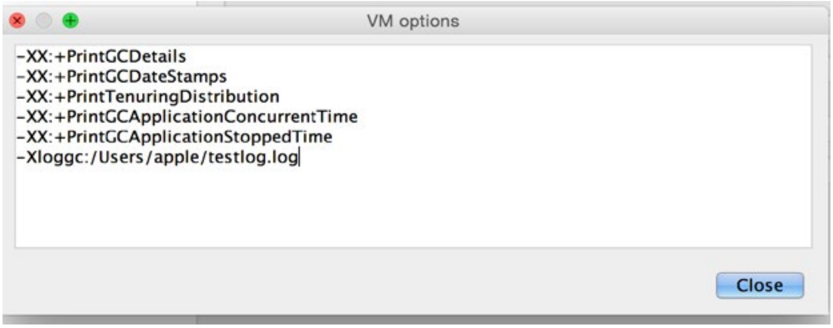
Let's work through a contrived example to get a sense of what a GC log file looks like. The following program will generate some garbage for us:

```
package com.conissaunce.gcbook;
import java.util.HashMap;
public class SimpleGarbageMaker {
  public static void main(String[] args) {
    System.out.println("InfoQ GC minibook test
    program");
    String stringDataPrefix = "InfoQ GC minibook test";

    {
      /**
      * Using HashMap
      */
      HashMap stringMap = new HashMap();
      for (int i = 0; i < 5000000; ++i) {
         String newStringData = stringDataPrefix +
         " index_" + i;
         stringMap.put(newStringData,
         String.valueOf(i));
      }
      System.out.println("MAP size: " + stringMap.
      size());
      for (int i = 0; i < 4000000; ++i) {
        String newStringData = stringDataPrefix +
        " index_" + i;
        stringMap.remove(newStringData);
      }
      System.out.println("MAP size: " +
      stringMap.size());
      System.gc();
    }
  }
}
```

Note: `System.gc()` does not necessarily immediately trigger a synchronous garbage collection. Instead, the GC call is really a hint to the runtime that now would be a fine time for it to go and run the collector. The runtime itself decides whether to run the garbage collection and what type to run, and the effect of calling `System.gc()` can be completely disabled using the flag `-XX:+DisableExplicitGC.`

Set the VM arguments as described above:



```
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+PrintTenuringDistribution
-XX:+PrintGCApplicationConcurrentTime
-XX:+PrintGCApplicationStoppedTime
-Xloggc:/Users/apple/testlog.log
```

Compile and run the program. The `system.out` log should show this:

```
InfoQ GC minibook test program
MAP size: 5000000
MAP size: 1000000
```

The Parallel collector

If you open the GC log file (`testlog.log` in my case) in a text editor, you'll see a series of messages similar to the following:

```
[PSYoungGen: 66048K->10736K(76800K)]
66048K->45920K(251392K), 0.0461740 secs] [Times:
user=0.11 sys=0.03, real=0.04 secs]
```

In my particular run, using Java 8 on OS X Yosemite, I got 12 GC pauses, five of which were full GC cycles.

PSYoungGen refers to the garbage collector in use for the minor collection. The name indicates the generation and collector in question. PS stands for "Parallel Scavenge".

The numbers before and after the first arrow (e.g. "66048K->10736K" from the first line) indicate the combined size of live objects before and after garbage collection, respectively. The next number, in parentheses, is the committed size of the heap: the amount of space usable for Java objects without requesting more memory from the operating system.

The second set of numbers indicates heap usage.

So in this case:

- 66,048 kB live objects used before GC, 10,736 kB after GC. The committed size of the heap is 76,800 kB.
- 66,048 kB heap used before GC, 45,920 kB after GC, and total heap size is 251,392 kB.

- The event took 0.0461740 seconds.

Here is a second example, this time showing a full GC event with ParOldGen highlighted in bold.

```
[PSYoungGen: 542688K->253792K(628224K)]
1290819K->1185723K(1629184K), 0.3456630 secs] [Times:
user=1.01 sys=0.08, real=0.35 secs]
2015-03-07T22:11:25.540+0000: 7.239: [Full GC
[PSYoungGen: 253792K->0K(628224K)] [ParOldGen:
931931K->994693K(1595392K)] 1185723K->994693K(2223616K)
[PSPermGen: 3015K->3015K(21504K)], 3.1400810 secs]
[Times: user=8.78 sys=0.15, real=3.14 secs]
```

The format is essentially the same, so for the old-generation collection:

- 931,931 kB used before GC, 994,693 kB after GC.
- 1,185,723 kB heap used before GC, 994,693 kB after GC.
- The event took 3.1400810 seconds.

CMS

As we noted in Part 2, CMS makes a full GC event less frequent at the expense of reduced throughput. For this test, switching to CMS reduced the number of full GC events to a single one, but with 22 pauses.

The log file includes details of the various phases of the collection process. It is similar but not identical to the Parallel collector.

This is the young-generation (ParNew) collection:

```
2015-03-07T22:36:27.143+0000: 0.785: [GC2015-03-
07T22:36:27.143+0000: 0.785: [ParNew
Desired survivor size 4456448 bytes, new threshold 1
(max 6)
- age 1: 8912256 bytes, 8912256 total
: 76951K->8704K(78656K), 0.1599280 secs]
183416K->182512K(253440K), 0.1599980 secs] [Times:
user=0.38 sys=0.03, real=0.16 secs]
```

Young-generation live-object use is 76,951 kB before collection and its occupancy drops down to 8,704 kB after collection. This collection took 0.1599980 seconds.

This is the beginning of tenured-generation collection:

```
2015-03-07T22:36:27.055+0000: 0.697: [GC [1 CMS-ini-
tial-mark: 106465K(174784K)] 116507K(253440K), 0.0085560
secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
2015-03-07T22:36:27.063+0000: 0.705
```

Capacity of the tenured-generation space is 174,784 kB and CMS was triggered at the occupancy of 106,465 kB.

This, fairly obviously, is the start of the concurrent-marking phase:

```
2015-03-07T22:36:27.063+0000: 0.705: [CMS-concurrent-
mark-start]
```

```
2015-03-07T22:36:27.143+0000: 0.785: Application time:
0.0797880 seconds
```

This is the end of concurrent marking:

```
2015-03-07T22:36:31.589+0000: 5.231: [CMS-concur-
rent-mark: 1.632/4.525 secs] [Times: user=9.80 sys=0.53,
real=4.52 secs]
```

Marking took a total 1.632 seconds of CPU time and 4.525 seconds of wall time (that is, the actual time that step took to execute its assigned tasks).

Next we have the pre-cleaning step, which is also a concurrent phase. This is where the collector looks for objects that got updated by promotions from young generation, along with new allocations and anything that got updated by mutators, whilst it was doing the concurrent marking.

Start of pre-cleaning:

```
2015-03-07T22:36:31.589+0000: 5.231: [CMS-concur-
rent-preclean-start]
```

End of pre-cleaning:

```
2015-03-07T22:36:32.166+0000: 5.808: [CMS-concur-
rent-preclean: 0.530/0.530 secs] [Times: user=1.11
sys=0.05, real=0.58 secs]
```

Concurrent pre-cleaning took 0.530 seconds of total CPU time and the same amount of wall time.

This is a stop-the-world phase:

```
2015-03-07T22:36:32.181+0000: 5.823: [Rescan (paral-
lel) , 0.0159590 secs]2015-03-07T22:36:32.197+0000:
5.839: [weak refs processing, 0.0000270 secs]2015-
03-07T22:36:32.197+0000: 5.839: [scrub string table,
0.0001410 secs] [1 CMS-remark: 1201997K(1203084K)]
1360769K(1509772K), 0.0161830 secs] [Times: user=0.07
sys=0.00, real=0.01 secs]
```

This phase rescans any residual updated objects in CMS heap, retraces from the roots, and also processes Reference objects. Here, the rescanning work took 0.0159590 seconds and weak reference-object processing took 0.0000270 seconds. This phase took a total of 0.0161830 seconds to complete.

```
2015-03-07T22:36:32.197+0000: 5.840: [CMS-concurrent-
sweep-start]
```

This marks the start of sweeping of dead/non-marked objects. Sweeping is a concurrent phase performed with all other threads running.

```
[CMS2015-03-07T22:36:32.271+0000: 5.913: [CMS-con-
current-sweep: 0.073/0.073 secs] [Times: user=0.10
sys=0.00, real=0.08 secs]
```

Sweeping took 0.073 seconds.

Finally, a reset event occurs — shown as CMS-concurrent-reset-start in the log — where the CMS data structures are reinitialised so that a new cycle may begin at a later time.

The CMS log file may also show a number of error conditions. Some of the most common are:

- **Concurrent mode failure.** If the concurrent collector cannot finish reclaiming the space occupied by dead objects before the tenured generation fills up, or if an allocation cannot be satisfied with the available free-space blocks in the tenured generation, then the application is paused and the collection is completed with all the application threads stopped. The concurrent mode failure can either be avoided by increasing the tenured generation size or by initiating the CMS collection at a lesser heap occupancy by setting `CMSInitiatingOccupancyFraction` to a lower value and setting `UseCMSInitiatingOccupancyOnly` to true. Sometimes, we see these promotion failures even when the logs show that there is enough free space in tenured generation. The reason is fragmentation: the free space available in tenured generation is not contiguous, and promotions from young generation require a contiguous free block to be available in tenured generation.
- **GC locker:** Trying a full collection because scavenge failed. Stop-the-world GC was happening when a JNI critical section was released. Young-generation collection failed due to “full promotion guarantee failure” and thus the full GC was invoked.

Garbage First

Repeating the test with G1 had 17 young collections, i.e. collections from Eden and survivor regions, and one full collection. The G1 log file starts like this:

```
2015-03-08T08:03:05.171+0000: 0.173: [GC pause (young)

Desired survivor size 1048576 bytes, new threshold 15
(max 15)

, 0.0090140 secs]
```

This is the highest-level information, telling us that an evacuation pause started 0.173 seconds after the start of the process. This collection took 0.0090140 seconds to finish. Evacuation pauses can be mixed as well — shown as GC pause (mixed) — in which case the set of regions selected includes all of the young regions plus some old regions.

As with the CMS log, the G1 log breaks out each step and provides a great deal of information.

Parallel time is the total elapsed time spent by all the parallel GC worker threads. The indented lines indicate the parallel tasks performed by these worker threads in the total parallel time, which in this case is 8.7 ms.

```
2015-03-08T08:03:05.171+0000: 0.173: Application time:
0.1091340 seconds
2015-03-08T08:03:05.171+0000: 0.173: [GC pause (young)
Desired survivor size 1048576 bytes, new threshold 15
(max 15)
, 0.0090140 secs]
 [Parallel Time: 8.7 ms, GC Workers: 4]
 [GC Worker Start (ms): Min: 172.9, Avg: 172.9, Max:
173.1, Diff: 0.2]
 [Ext Root Scanning (ms): Min: 0.6, Avg: 0.9, Max: 1.8,
Diff: 1.2, Sum: 3.8]
 [Update RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff:
0.0, Sum: 0.0]
 [Processed Buffers: Min: 0, Avg: 2.8, Max: 11, Diff:
11, Sum: 11]
 [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0,
Sum: 0.0]
 [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0,
Diff: 0.0, Sum: 0.0]
 [Object Copy (ms): Min: 6.8, Avg: 7.5, Max: 7.7, Diff:
0.9, Sum: 29.9]
```

```
 [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff:
0.1, Sum: 0.3]
 [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.1,
Diff: 0.1, Sum: 0.3]
 [GC Worker Total (ms): Min: 8.4, Avg: 8.6, Max: 8.6,
Diff: 0.2, Sum: 34.3]
 [GC Worker End (ms): Min: 181.5, Avg: 181.5, Max:
181.5, Diff: 0.1]
 [Code Root Fixup: 0.0 ms]
 [Code Root Migration: 0.0 ms]
 [Clear CT: 0.0 ms]
 [Other: 0.2 ms]
 [Choose CSet: 0.0 ms]
 [Ref Proc: 0.1 ms]
 [Ref Enq: 0.0 ms]
 [Free CSet: 0.0 ms]
 [Eden: 14.0M(14.0M)->0.0B(10.0M) Survivors:
0.0B->2048.0K Heap: 14.0M(256.0M)->5792.0K(256.0M)]
```

The GC Worker Start line tells us the minimum, average, and maximum of the start times of all of the worker threads. Diff is the difference between the minimum and maximum values.

Ext Root Scanning gives us the minimum, average, maximum, and diff of the times all the worker threads spent scanning the roots (globals, registers, thread stacks, and VM data structures).

Then:

- **Update RS:** The time each thread spent updating the remembered set. The collector keeps track of changes made by the mutator during this step, in buffers called “update buffers”. The Processed Buffers step is the Update RS subtask that processes the update buffers that were not able to be processed concurrently and updates the corresponding remembered sets of all regions.

- **Scan RS:** The time each worker thread had spent in scanning the remembered sets.

- **Code Root Scanning:** A region’s remembered set contains the reference from the compiled code into that region. These references are added by compilation, not directly by the mutator. This is the time each worker thread spent processing this region.

- **Object Copy:** The time spent by each worker thread copying live objects from the regions in the collection set to the other regions.

- **Termination:** Termination time is the time spent by the worker threads offering to terminate. Before terminating, each thread checks the work queues of other threads to ascertain whether there are still object references in other work queues. If there are, it attempts to steal object references, and if it succeeds in stealing a reference, it processes that and then offers to terminate again.
- **GC Worker Other:** The time that each worker thread spent in performing some other tasks that have not been accounted for above but fall within the total parallel time.
- **Clear CT:** This is the time spent in clearing the card table. This task is performed in serial mode.
- **Other:** Total time spent performing other tasks. These are then broken out as individual subtasks:
  - **Choose CSet:** Time spent in selecting the regions for the collection set.
  - **Ref Proc:** Time spent processing reference objects.
  - **Ref Enq:** Time spent "enqueuing", i.e. adding, references to the ReferenceQueues.
  - **Free CSet:** Time spent in freeing the collection-set data structure.

Finally, the Eden line details the heap size changes caused by the evacuation pause. This shows that Eden had an occupancy of 14 MB and its capacity was also 14 MB before the collection. After the collection, its occupancy got reduced to 0 MB (since everything is evacuated/promoted from Eden during a collection) and its target size shrank to 10 MB. The new Eden capacity of 10 MB is not reserved at this point; rather, this value is the target size of Eden. Regions are added to Eden as the demand is made and when it reaches the target size, it starts the next collection.

Similarly, survivors had an occupancy of 0 bytes and grew to 2,048 kB after the collection. The total heap occupancy and capacity were 14 MB and 256 MB respectively before the collection and became 5,792 kB and 256 MB after the collection.

Like CMS, G1 can fail to keep up with promotion rates, and will fall back to a stop-the-world, full GC. Like CMS and its "concurrent mode failure", G1 can suffer an evacuation failure, seen in the logs as "to-space overflow". This occurs when there are no free regions into which objects can be evacuated, which is similar to a promotion failure.

An evacuation failure is very expensive in G1:

1. For successfully copied objects, G1 needs to update the references and the regions have to be tenured.

2. For unsuccessfully copied objects, G1 will self-forward them and tenure the regions in place.

To help explain the cause of evacuation failure, the argument `-XX:+PrintAdaptiveSizePolicy` will provide details about adaptive generation sizing that are purposefully kept out of the `-XX:+PrintGCDetails` option.

An evacuation failure may be symptomatic of over-tuning. To test this, run a benchmark using only `-Xms` and `-Xmx` and with a pause-time goal (`-XX:MaxGCPauseMillis`). If the problem goes away then over-tuning is the likely cause.

If the problem persists, increasing the size of the heap may help. If you can't do this for some reason then you'll need to do some more analysis:

1. If you notice that the marking cycle is not starting early enough for G1 GC to be able to reclaim the old generation, drop your `-XX:InitiatingHeapOccupancyPercent`. The default for this is 45% of your total Java heap. Dropping the value will help start the marking cycle earlier.

2. Conversely, if the marking cycle is starting early and not reclaiming much, you should increase the threshold above the default value to make sure that you are accommodating the live data set for your application.

3. If you find that the concurrent-marking cycles are starting on time but take a long time to finish, try increasing the number of concurrent-marking threads, using the command line option `-XX:ConcGCThreads`. Long-running concurrent-marking cycles can cause evacuation failures because they delay the mixed garbage-collection cycles, resulting in the old generation not being reclaimed in a timely manner.

4. If the availability of "to" space survivor region is the issue, then increase the `-XX:G1ReservePercent`. The default is 10% of the Java

heap. A false ceiling is created by G1 GC to reserve memory, in case there is a need for more "to" space. Note that G1 won't respect a value greater than 50% to prevent an end user setting it to an impractically large value.

Balanced

IBM recommends that you restrict tuning Balanced to adjusting the size of Eden using `Xmnx<size>` to set its maximum size. As a general rule, for optimal performance the amount of data surviving from Eden space in each collection should be kept to approximately 20% or less. Some systems might be able to tolerate parameters outside these boundaries, based on total heap size or number of available GC threads in the system.

If you set the size of Eden too small, your system may pause for GC more frequently than it needs to, reducing performance. It will also have consequences if there is a spike in workload and the available Eden space is exceeded. Conversely, if Eden is too large, you reduce the amount of memory available to the general heap. This forces the Balanced collector to incrementally collect and defragment large portions of the heap in each partial collection cycle in an effort to keep up with demand, resulting in long GC pauses.

If you are moving to Balanced from the older Generational Concurrent Garbage Collector then any `-Xmn` setting that you used is generally applicable to Balanced as well.

Balanced outputs verbose GC logs in XML format. This is an example of what it looks like, here showing a typical partial collection cycle:

```
<exclusive-start id="137" timestamp="2015-03-
22T16:18:32.453" intervalms="3421.733">
  <response-info timems="0.146" idlems="0.104"
threads="4" lastid="0000000000D97A00"
  lastname="XYZ Thread Pool : 34" />
</exclusive-start>
<allocation-taxation id="138" taxation-thresh-
old="671088640"
  timestamp="2015-03-22T16:18:32.454" inter-
valms="3421.689" />
<cycle-start id="139" type="partial gc" contextid="0"
timestamp="2015-03-22T16:18:32.454"
  intervalms="3421.707" />
<gc-start id="140" type="partial gc" contextid="139"
timestamp="2015-03-22T16:18:32.454">
```

```
  <mem-info id="141" free="8749318144" total="10628366336" percent="82">
  <mem type="Eden" free="0" total="671088640" percent="0" />
  <numa common="10958264" local="1726060224" non-local="0" non-local-percent="0" />
 <remembered-set count="352640" freebytes="422080000" totalbytes="424901120"
 percent="99" regionsoverflowed="0" />
  </mem-info>
</gc-start>
<allocation-stats totalBytes="665373480" >
  <allocated-bytes non-tlh="2591104" tlh="662782376" arrayletleaf="0"/>
  <largest-consumer threadName="WXYConnection[192.168.1.1,port=1234]»
  threadId="0000000000C6ED00" bytes="148341176" />
</allocation-stats>
<gc-op id="142" type="copy forward" timems="71.024" contextid="139"
  timestamp="2015-03-22T16:18:32.527">
  <memory-copied type="Eden" objects="171444" bytes="103905272"
  bytesdiscarded="5289504" />
  <memory-copied type="other" objects="75450" bytes="96864448" bytesdiscarded="4600472" />
  <memory-cardclean objects="88738" bytes="5422432" />
  <remembered-set-cleared processed="315048" cleared="53760" durationms="3.108" />
  <finalization candidates="45390" enqueued="45125" />
  <references type="soft" candidates="2" cleared="0" enqueued="0" dynamicThreshold="28"
  maxThreshold="32" />
  <references type="weak" candidates="1" cleared="0" enqueued="0" />
</gc-op>
<gc-op id="143" type="classunload" timems="0.021" contextid="139"
  timestamp="2015-03-22T16:18:32.527">
  <classunload-info classloadercandidates="178" classloadersunloaded="0"
  classesunloaded="0" quiescems="0.000" setupms="0.018" scanms="0.000" postms="0.001" />
</gc-op>
<gc-end id="144" type="partial gc" contextid="139" durationms="72.804"
```

```
  timestamp="2015-03-22T16:18:32.527">
  <mem-info id="145" free="9311354880" total="10628366336" percent="87">
  <numa common="10958264" local="1151395432" non-local="0" non-local-percent="0" />
  <pending-finalizers system="45125" default="0" reference="0" classloader="0" />
  <remembered-set count="383264" freebytes="421835008"
totalbytes="424901120"
  percent="99" regionsoverflowed="0" />
  </mem-info>
</gc-end>
<cycle-end id="146" type="partial gc" contextid="139"
  timestamp="2015-03-22T16:18:32.530" />
<exclusive-end id="147" timestamp="2015-03-
22T16:18:32.531" durationms="77.064" />
```

Like CMS and G1, Balanced can encounter something similar to a "concurrent mode failure" during a partial garbage collection if there is not enough free space to copy all of the live objects.

In such cases, Balanced will switch from a copying mechanism to an in-place trace-and-compact approach. The compact pass uses sliding compaction, which is guaranteed to successfully complete the operation without requiring any free heap memory, unlike the preferred copying approach. This information is presented in the `verbose:gc` output with two additional lines in the log file for the copy forward garbage collection operation:

```
<memory-traced type="eden" objects="9447"
bytes="19746628" />
<memory-traced type="other" objects="51" bytes="1624548"
/>
```

Metronome

Metronome can struggle to keep up with allocation rate. If both the target utilisation and allocation rate are high, the application can run out of memory, forcing the GC to run continuously and dropping the utilisation to 0% in most cases. If this scenario is encountered, you must choose to decrease the target utilisation to allow for more GC time, increase the heap size to allow for more allocations, or a combination of both. The relationship between utilisation and heap size is highly application dependent, and striking an appropriate balance requires iterative experimentation with the application and VM parameters.

Here is an example of Metronome's verbose-gc output. As with Balanced, it is in XML format:

```
<verbosegc version="201502_15-Metronome">
<gc type="synchgc" id="1" timestamp="Mon April 13 15:17:18
2015" intervalms="0.000">
  <details reason="system garbage collect" />
  <duration timems="30.023" />
  <heap freebytesbefore="535265280" />
  <heap freebytesafter="535838720" />
  <immortal freebytesbefore="15591288" />
  <immortal freebytesafter="15591288" />
  <synchronousgcpriority value="11" />
</gc>

<gc type="trigger start" id="1" timestamp="Mon April 13
15:17:45 2015" intervalms="0.000" />
<gc type="heartbeat" id="1" timestamp="Mon April 13
15:17:46 2015" intervalms="1003.413">
  <summary quantumcount="477">
    <quantum minms="0.078" meanms="0.503" maxms="1.909"
    />
    <heap minfree="262144000" meanfree="265312260"
    maxfree="268386304" />
    <immortal minfree="14570208" meanfree="14570208"
    maxfree="14570208" />
    <gcthreadpriority max="11" min="11" />
  </summary>
</gc>

<gc type="heartbeat" id="2" timestamp="Mon April 13
15:17:47 2015" intervalms="677.316">
  <summary quantumcount="363">
    <quantum minms="0.024" meanms="0.474" maxms="1.473" />
    <heap minfree="261767168" meanfree="325154155"
    maxfree="433242112" />
    <immortal minfree="14570208" meanfree="14530069"
    maxfree="14570208" />
    <gcthreadpriority max="11" min="11" />
  </summary>
</gc>

<gc type="trigger end" id="1" timestamp="Mon April 13
15:17:47 2015" intervalms="1682.816"/>

</verbosegc>
```

A synchgc type represents a synchronous GC, which is a GC cycle that ran uninterrupted from beginning to end — that is, interleaving with the application didn't happen. These can occur for two reasons:

- `System.gc()` was invoked by the application.
- The heap filled up and the application failed to allocate memory.

The `trigger` GC events correspond to the GC cycle's start and end points. They're useful for delimiting batches of `heartbeat` GC events, which roll up the information of multiple GC quanta into one summarised verbose event. Note that this is unrelated to the alarm-thread heartbeat. The `quantumcount` attribute corresponds to the amount of GC quanta rolled up in the heartbeat GC. The `<quantum>` tag represents timing information about the GC quanta rolled up in the heartbeat GC. The `<heap>` and `<immortal>` tags contain information about the free memory at the end of the quanta rolled up in the heartbeat GC. The `<gcthreadpriority>` tag contains information about the priority of the GC thread when the quanta began.

The quantum time values correspond to the pause times seen by the application. Mean quantum time should be close to 500 microseconds, and the maximum quantum times must be monitored to ensure they fall within the acceptable pause times for the real-time application. Large pause times can occur when other processes in the system preempt the GC and prevent it from completing its quanta and allowing the application to resume, or when certain root structures in the system are abused and grow to unmanageable sizes.

Metronome also has some specific quirks. GC work in Metronome is time-based, and this means that any change to the hardware clock can cause hard-to-diagnose problems. An example is synchronising the system time to a Network Time Protocol (NTP) server and then synchronising the hardware clock to the system time. This appears as a sudden jump in time to the GC and can cause problems such as a failure in maintaining the utilisation target and even out-of-memory errors.

Running multiple JVMs on a single machine can introduce interference across the JVMs, skewing the utilisation figures. The alarm thread, being a high-priority real-time thread, preempts any other lower-priority thread, and the GC thread also runs at real-time priority. If sufficient GC and alarm threads are active at any time, a JVM without an active GC cycle might have its application threads preempted by another JVM's GC and alarm threads while time is actually taxed to the application because the GC for that VM is inactive.

Log-file analysis tools

As we've seen, the log-file format varies by collector. It is also subject to change in future versions. Thankfully, there are a number of tools that can help automate the process of analysing log files.

IBM offers a free Garbage Collection and Memory Visualizer (GCMV) tool as part of IBM Support Assistant[1] that can work with their XML format. GCMV supports graphing of pause times, heap sizes, and so on, and supports the comparison of multiple files. It also provides tuning recommendations based on the data, and flags errors.

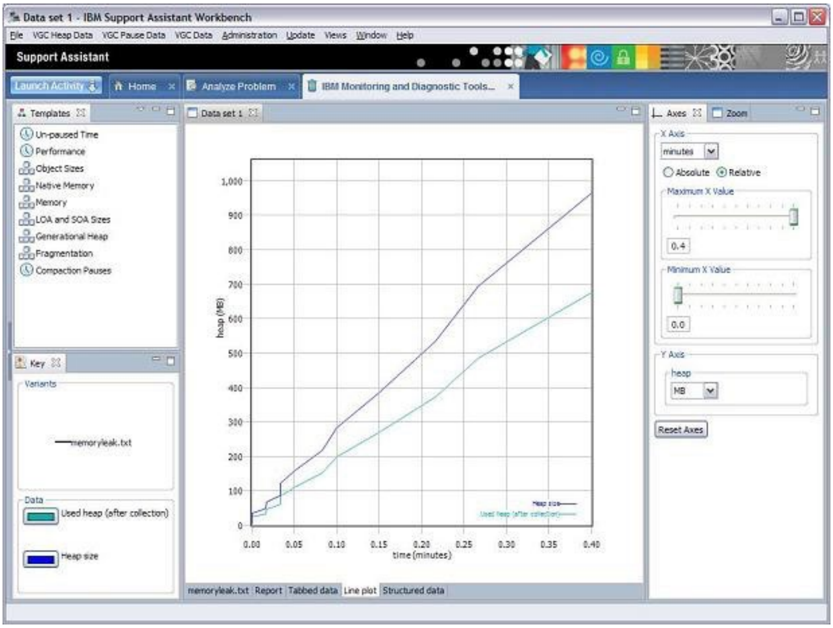


Figure 12: IBM's GCMV

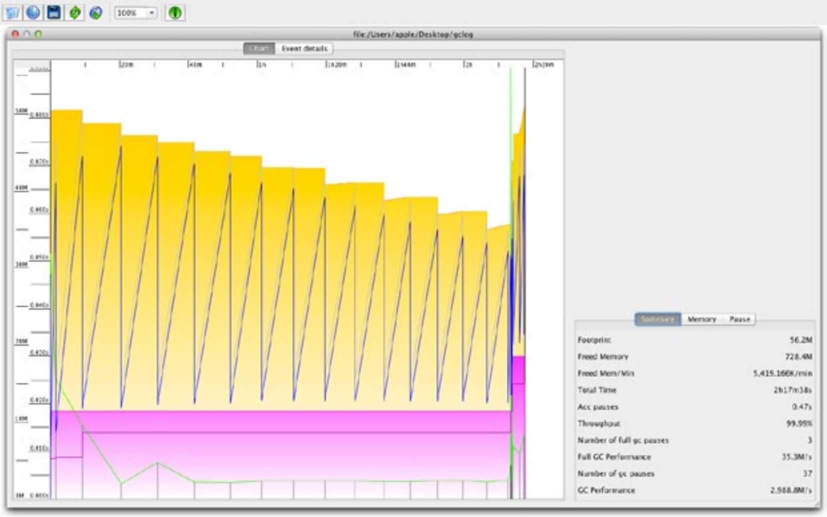
Tuning Fork is a separate tool for tuning Metronome to better suit the user application. Tuning Fork lets the user inspect many details of GC activity either after the fact through a trace log or during runtime through a socket. Metronome was built with Tuning Fork in mind and logs many events that can be inspected within the Tuning Fork applica-

[1] http://www-01.ibm.com/software/support/isa/

tion. For example, it displays the application utilisation over time and inspects the time taken for various GC phases.

For the OpenJDK collectors, a freely available tool is Chewiebug.[2]

The Chewiebug screenshot below is from a real-world application using the CMS collector. The application is fairly predictable, with the heap filling up over about a 10-minute interval, triggering a collection. However, there's a substantial spike of activity around 2 hours 15 minutes.



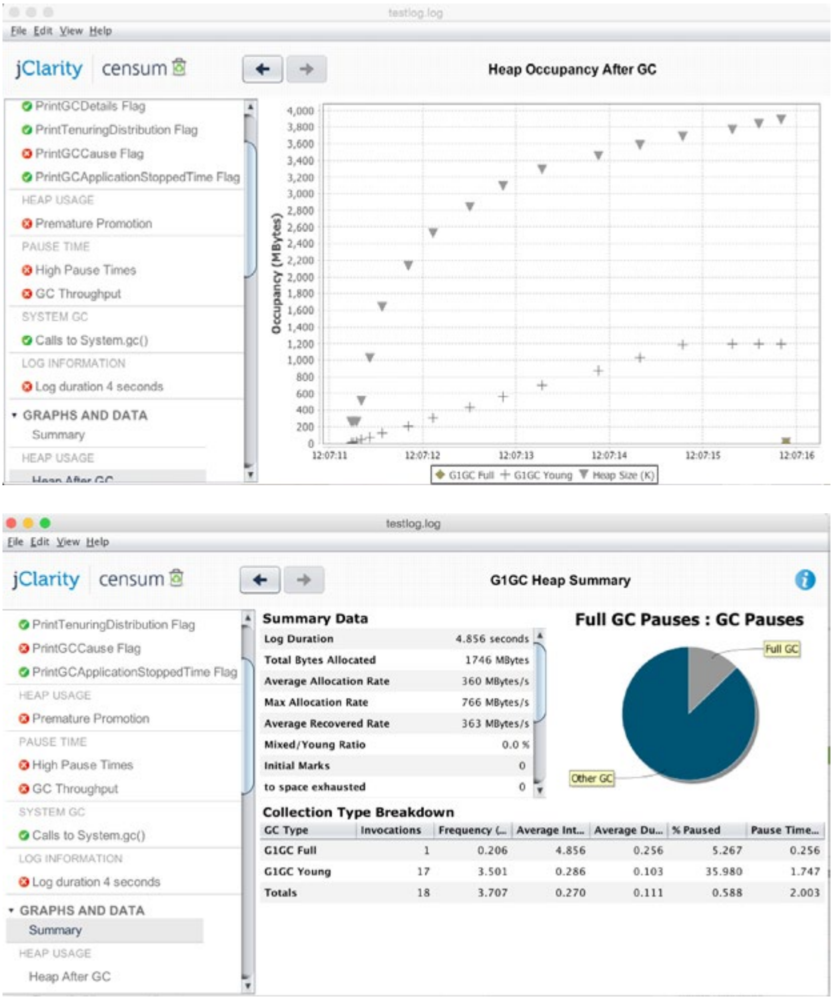
Footprint	56.2M
Freed Memory	728.4M
Freed Mem/Min	5,419.166K/min
Total Time	2h17m38s
Acc pauses	0.47s
Throughput	99.99%
Number of full gc pauses	3
Full GC Performance	35.3M/s
Number of gc pauses	37
GC Performance	2,988.8M/s

Figure 13: Chewiebug

jClarity's Censum[3] is a commercial product from a London-based startup founded by Martijn Verburg, Ben Evans, and Kirk Pepperdine. At the time of writing, it is the only tool I know of that can correctly analyse G1 logs and is, in my opinion, more user-friendly and comprehensive than Chewiebug.

[2] https://github.com/chewiebug/GCViewer

[3] http://www.jclarity.com/censum/



- ✓ PrintGCDetails Flag
- ✓ PrintTenuringDistribution Flag
- ✗ PrintGCCause Flag
- ✓ PrintGCApplicationStoppedTime Flag

HEAP USAGE

- ✗ Premature Promotion

PAUSE TIME

- ✗ High Pause Times
- ✗ GC Throughput

SYSTEM GC

- ✓ Calls to System.gc()

LOG INFORMATION

- ✗ Log duration 4 seconds

▾ GRAPHS AND DATA

Summary

HEAP USAGE

Heap After GC

Summary Data

Log Duration	4.856 seconds
Total Bytes Allocated	1746 MBytes
Average Allocation Rate	360 MBytes/s
Max Allocation Rate	766 MBytes/s
Average Recovered Rate	363 MBytes/s
Mixed/Young Ratio	0.0 %
Initial Marks	0
to space exhausted	0

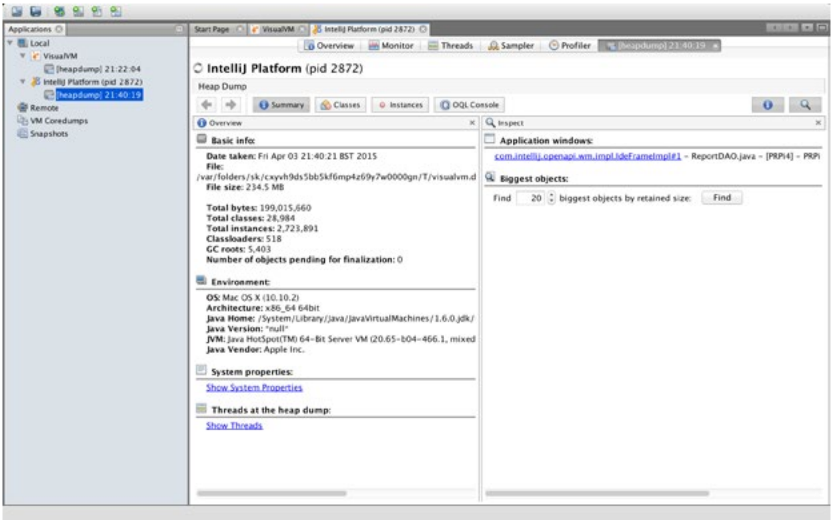
Collection Type Breakdown

GC Type	Invocations	Frequency (...	Average Int...	Average Du...	% Paused	Pause Time...
G1GC Full	1	0.206	4.856	0.256	5.267	0.256
G1GC Young	17	3.501	0.286	0.103	35.980	1.747
Totals	18	3.707	0.270	0.111	0.588	2.003

Figure 14: jClarity's Censum.

VisualVM[4] is available as a JDK tool in Oracle/Sun JDK distributions starting from JDK 6 Update 7 and Apple's Java for Mac OS X 10.5 Update 4. Amongst other things, VisualVM has a visualiser that lets you easily browse a heap dump. By examining the heap, you can locate where objects are created and find the references to those objects in the source. If the JVM is failing to remove unneeded objects from the heap, VisualVM can help you locate the nearest GC root for the object.

[4] http://visualvm.java.net



IntelliJ Platform (pid 2872)

Heap Dump

Basic info:

Date taken: Fri Apr 03 21:40:21 BST 2015
File:
/var/folders/sk/cxyvh9ds5bb5kf6mp4z69y7w0000gn/T/visualvm.d
File size: 234.5 MB

Total bytes: 199,015,660
Total classes: 28,984
Total instances: 2,723,891
Classloaders: 518
GC roots: 5,403
Number of objects pending for finalization: 0

Environment:

OS: Mac OS X (10.10.2)
Architecture: x86_64 64bit
Java Home: /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/
Java Version: "null"
JVM: Java HotSpot(TM) 64-Bit Server VM (20.65-b04-466.1, mixed
Java Vendor: Apple Inc.

System properties:

Show System Properties

Threads at the heap dump:

Show Threads

Application windows:

com.intellij.openapi.wm.impl.IdeFrameImpl#1 - ReportDAO.java - [PRPi4] - PRPi

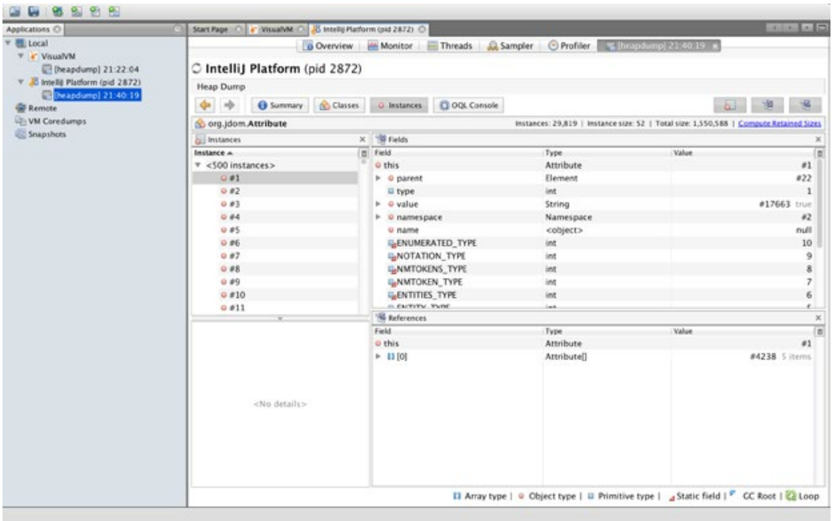
Biggest objects:

Find 20 biggest objects by retained size: Find

Figure 15: VisualVM heap Summary view

When you open a heap dump, VisualVM displays the Summary view, which shows the running environment where the heap dump was taken along with other system properties.

The Classes view displays a list of classes and the number and percentage of instances referenced by that class. You can view a list of the instances of a specific class by right-clicking the name and choosing "Show in Instances View".



IntelliJ Platform (pid 2872)

Heap Dump

org.jdom.**Attribute** Instances: 29,819 | Instance size: 52 | Total size: 1,550,588 | Compute Retained Sizes

Instance ▲
<500 instances>
#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
#11

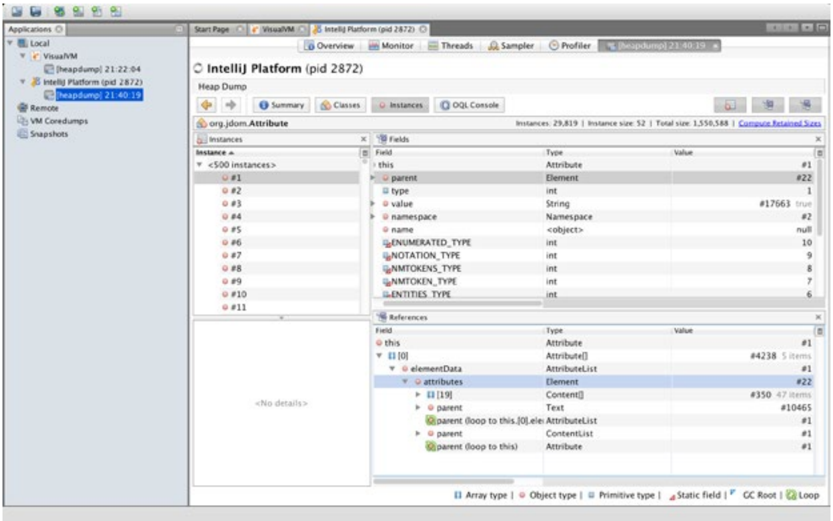
<No details>

Field	Type	Value
this	Attribute	#1
parent	Element	#22
type	int	1
value	String	#17663 true
namespace	Namespace	#2
name	<object>	null
ENUMERATED_TYPE	int	10
NOTATION_TYPE	int	9
NMTOKENS_TYPE	int	8
NMTOKEN_TYPE	int	7
ENTITIES_TYPE	int	6

Field	Type	Value
this	Attribute	#1
[0]	Attribute[]	#4238 5 items

Figure 16: VisualVM heap Instances view.

The Instances view displays object instances for a selected class. When you select an instance from the Instance pane, VisualVM displays the fields of that class and references to that class in the respective panes. In the References pane, you can right-click an item and choose "Show Nearest GC Root" to display the nearest GC root object.



Field	Type	Value
this	Attribute	#1
parent	Element	#22
type	int	1
value	String	#17663 true
namespace	Namespace	#2
name	<object>	null
ENUMERATED_TYPE	int	10
NOTATION_TYPE	int	9
NMTOKENS_TYPE	int	8
NMTOKEN_TYPE	int	7
ENTITIES_TYPE	int	6

Field	Type	Value
this	Attribute	#1
[0]	Attribute[]	#4238 5 items
elementData	AttributeList	#1
attributes	Element	#22
[19]	Content[]	#350 47 items
parent	Text	#10465
parent (loop to this.[0].ele	AttributeList	#1
parent	ContentList	#1
parent (loop to this)	Attribute	#1

Figure 17: VisualVM GC root information.

Another useful tool to consider is jHiccup,[5] which can be used to track pauses within the JVM and across a system as a whole. Figure 18 shows an idle app on a quiet system. The top part of the chart shows what the jHiccup system saw, with the percentiles for the overall run. Below, the tool provides more detail, based on a 10,000-point percentile sample. Were the system (as opposed to the application) busier, we'd see more stalls as the scheduler steals more of the application's time.

[5] http://www.azulsystems.com/jHiccup

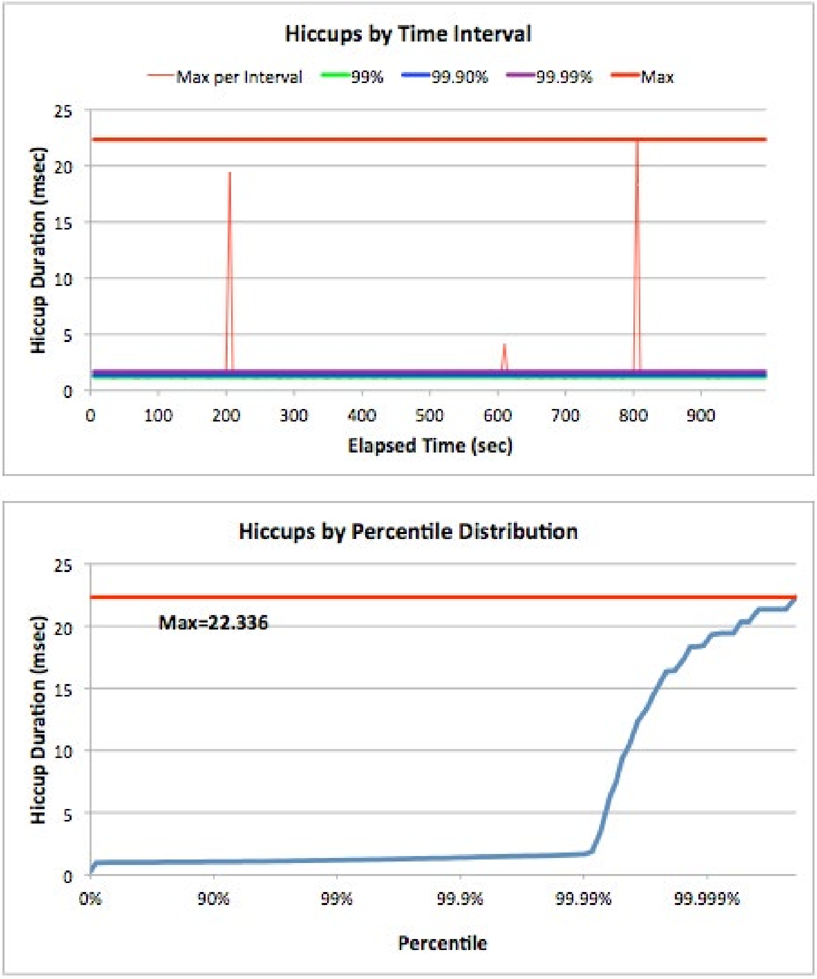


Figure 18: jHiccup on an idle system.

Figure 19 shows a real-world example, from a telco billing application running on HotSpot with the CMS collector. By correlating this with the GC logs, we can tell that what we're seeing here are the minor and major GC events.

As we've already discussed, CMS is a generational collector that segregates objects into young generation and old generation, and preferentially collects younger objects. The application, as is somewhat typical of these kind of telco apps, has a hard time in the young generation, since it tracks many sessions that last for just a few seconds and then die. As a result, the sessions tend to get collected in the young generation. The larger spike will be an old-generation pause.

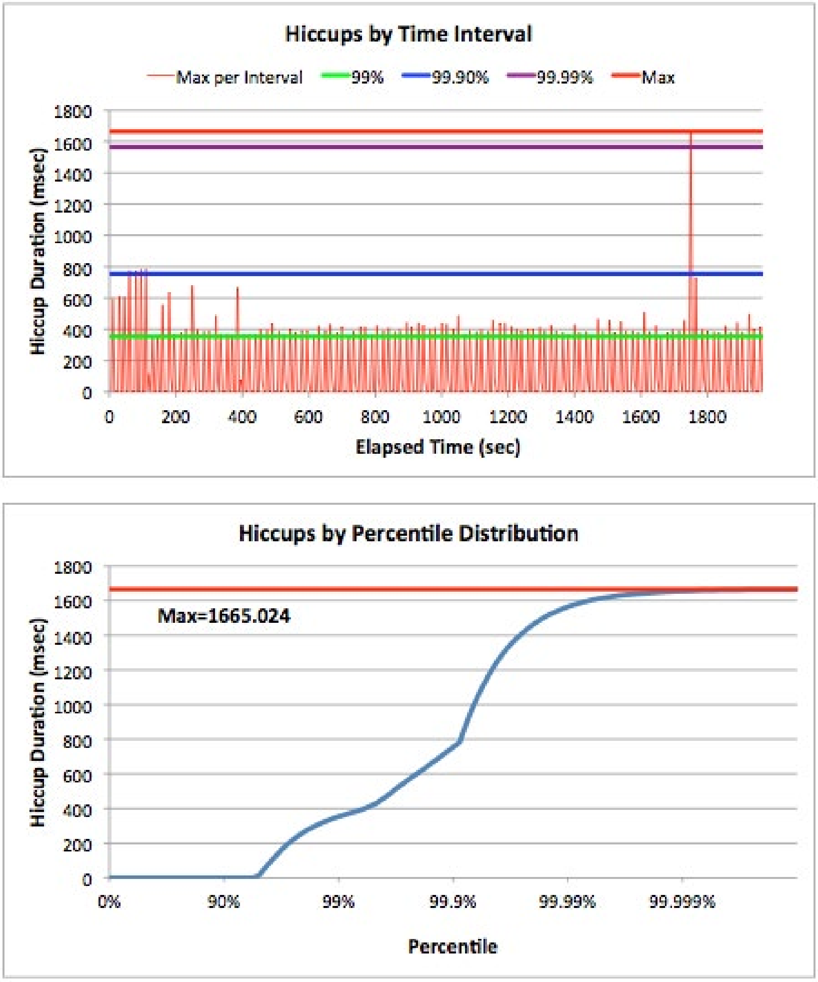


Figure 19: jHiccup with a telco billing application.

Memory footprint vs. CPU cycles

Collectors need empty memory. For both mark/compact and copying collectors, the amount of work per cycle increases linearly with the size of the live set, as we discussed in Part 1, and how often we need to do the cycle depends on the amount of empty memory. When the system runs out of memory, it triggers a GC event — and how much work is required doesn't depend on how much empty memory there is but on how much live is left.

To understand this, imagine that your collector has infinite memory — the system would never run out of memory and the JVM would never need to collect garbage. Now imagine a stable system in which whenever a mutator needs to allocate one byte another byte dies. Here, the collector would be spending all its time looking for that one byte every time you needed to allocate, doing a full GC cycle each time. In this scenario, the collector would hog the CPU and you'd never get any work done.

Between these two extremes, you'll more or less follow a 1/x curve. This typically means that doubling the empty memory halves the work the garbage collector has to do: if your collector is using 50% of the CPU to do its job, then doubling the empty memory drops that to 25%. Double it again and it will drop to 12.5%, and so on. This is the most powerful tool you have for controlling your collector's consumption of CPU cycles.

There is a quirk with a mark/sweep collector so that the efficiency doesn't double in quite the same way, but it does for a copying collector, and since all the standard HotSpot collectors use copying collection for the young-generation, up-front work, growing the empty memory is an efficient way of dealing with CPU resources.

Whilst empty memory controls the efficiency of the collector, it is worth pointing out that it doesn't control the length of the pauses. More empty memory means that stop-the-world events occur less frequently but the pauses won't be any smaller. This is a trap that people often fall into during performance testing; if you have a test that runs for 20 minutes, it's actually easy to tune the collector by growing empty memory so you never see a pause in that 20-minute window. What you can't do is make it go away — maybe it occurs at minute 21. Moreover, a mark/sweep collector has to do more work as you grow the heap; it will still improve in terms of efficiency with more memory, because some of its work is linear to the size of the heap rather than to the size of the live set, but the pause times will also grow if you grow the empty memory.

Setting the size of the heap

You can control the heap size using various parameters.

The `-Xms` and `-Xmx` parameters define the minimum and maximum heap sizes, respectively.

By default, the JVM grows or shrinks the heap at each collection to try to keep the proportion of free space available to the live objects at each collection within a specific range. This range is set as a percentage by the parameters `-XX:MinHeapFreeRatio=<minimum>` and `-XX:MaxHeapFreeRatio=<maximum>`; and the total size bounded by `-Xms` and `-Xmx`.

When the heap grows or shrinks, the JVM must recalculate the sizes of the old and new generations to maintain a predefined `NewRatio`; server-side applications can sometimes have the values of `-Xms` and `-Xmx` set equal to each other for a fixed heap size.

The `NewSize` and `MaxNewSize` parameters control the new generation's minimum and maximum size. You can regulate the new-generation size by setting these parameters equal. The bigger the younger generation, the less often minor collections occur. By default, the young generation is controlled by `NewRatio`. For example, setting `-XX:NewRatio=3` means that the ratio between the old and young generation is 1:3, so the combined size of Eden and the survivor spaces will be a quarter of the heap.

The following are important guidelines for sizing the Java heap:

- Decide the total amount of memory you can afford to allocate to the JVM. Accordingly, graph your own performance metric against young-generation sizes to find the best setting.
- Make plenty of memory available to the young generation. Since Java 1.4, the default has been calculated from `NewRatio` and the `-Xmx` setting.
- Larger Eden or young-generation spaces increase the time between full garbage collections, but young-space collections could take a proportionally longer time. In general, you can keep the Eden size between a quarter and a third of the maximum heap size.
- The old generation must typically be larger than the new generation.

All the concurrent collectors tend to function more efficiently when plenty of space is allocated. As a rule of thumb, you should set a heap size of at least two to three times the size of the live set for efficient operation. However, space requirements for maintaining concurrent operation grow with application throughput and the associated allocation and promotion rates. So, higher-throughput applications may warrant a larger heap-size to live-set ratio. Given memory's relatively low cost and the huge memory spaces available to today's systems, footprint is seldom an issue on the server side.

Survivor ratio

The `SurvivorRatio` parameter controls the size of the two survivor spaces. For example, `-XX:SurvivorRatio=6` sets the ratio between each survivor space and Eden to be 1:6, so each survivor space will occupy one eighth of the young generation.

If survivor spaces are too small, copying collection overflows directly into the old generation. If survivor spaces are too large, they will remain largely empty.

At each garbage collection, the JVM chooses a threshold number of times an object can be copied before it is tenured. This threshold is chosen to keep the survivors half full.

The option `-XX:+PrintTenuringDistribution` can be used to show this threshold and the ages of the objects in the new generation. It is useful for observing the lifetime distribution of an application.

Next, at each minor garbage collection, maximise the number of objects reclaimed. In their *Java Performance* book,[6] Charlie Hunt and Binu John call this the "minor GC reclamation principle". They write:

"Adhering to this principle helps reduce the number and frequency of full garbage collections experienced by the application. Full garbage collections typically have the longest duration and as a result are the number one reason for applications not meeting their latency or throughput requirements".

Conclusion

GC tuning can become a highly skilled exercise that often requires application changes to reduce object allocation rates or object lifetimes. If this is the case, then a commercial trade-off between time and resource spent on GC tuning and application changes versus purchasing one of the commercial concurrent-compacting JVMs such as WebSphere Real Time or Azul Zing may be required.

[6] http://www.amazon.co.uk/Java-Performance-Addison-Wesley-Charlie-Hunt/dp/0137142528

“This is the hell that the JVM developers created, giving you options so you could paint yourself into a corner. We left the Java developers alone. I’m sorry.”
— Eva Andreasson, “Garbage Collection Is Good”, QCon London 2014[7]

[7] http://www.infoq.com/presentations/garbage-collection-benefits

PART FIVE

Programming for less garbage

"We don't get to choose what is true. We only get to choose what we do about it."

— *Kami Garcia,* Beautiful Darkness

Application responsiveness and scalability limitations are generally sensitive to the type of processing that occurs during stop-the-world pauses and to the type or rate of operations that induce such pauses to occur more often. Examples of metrics that applications may be sensitive to include:

1. **Live-set size:** The amount of live data in the heap will generally determine the amount of work a collector must do in a given cycle. When a collector performs operations that depend on the amount of live data in the heap during stop-the-world pauses, the application's responsiveness becomes sensitive to the live-set size. Operations that tend to depend on live-set size are marking (in mark/sweep or mark/compact collectors), copying (in copying collectors), and the remapping of references during heap compaction (in any compacting collector).

2. **Heap size:** Depending on the algorithm and mechanisms used, the total size of the heap may determine the amount of work a collector must do in a given cycle. When a collector performs operations that depend on the amount of memory in the heap (regardless of whether the contents are alive or dead) and those operations are performed during a stop-the-world event, the application's responsiveness becomes sensitive to the heap size. In mark/sweep collectors, sweeping depends on heap size.

3. **Fragmentation and compaction:** Fragmentation is inevitable, and as a result so is compaction. As you allocate objects and some of them die, the heap develops holes that are large enough for some objects but not for others. As time goes by, you get more, smaller holes. Eventually, you get to a point where there is plenty of room in the heap but no hole large enough for an object, and a compaction is required in order to keep the application operating. Compaction will relocate at least some live objects in the heap in order to free up contiguous space for further allocation. When live objects are relocated to new locations during compaction, all references to the relocated objects must be tracked down and remapped to point to the new locations. All commercial JVMs contain code to compact the heap, without which the heap would become useless. All the collector modes (concurrent or not) of the HotSpot and IBM's J9 JVMs will perform compaction only during stop-the-world pauses. A compaction pause is generally the longest pause an application will experience during

normal operation, and these collectors are therefore sensitive to heap fragmentation and heap size.

4. **Mutation rate:** Mutation rate is defined as how quickly a program updates references in memory, i.e. how fast the heap, and specifically the pointers between objects in the heap, are changing. Mutation rate generally increases linearly with application load; the more operations per second you do, the more you will mutate references in the heap. Thus, if a collector can only mark mutated references in a pause or if a concurrent collector has to repeatedly revisit mutated references before completing a scanning phase, the collector is sensitive to workload and a high mutation rate can result in significant application pauses.

5. **Number of weak or soft references:** Some collectors, including the popular CMS and ParallelGC collectors in HotSpot, process weak or soft references only in a stop-the-world pause. As such, pause time is sensitive to the number of weak or soft references that the application uses.

6. **Object lifetime:** Most allocated objects die young, so collectors often make a generation distinction between young and old objects. A generational collector collects recently allocated objects separately from long-lived objects. Many collectors use different algorithms for handling the young and old generations — for example, they may be able to compact the entire young generation in a single pass but need more time to deal with the older generation. Many applications have long-lived datasets, however, and large parts of those datasets are not static (for example, caching is a common pattern in enterprise systems). The generational assumption is an effective filter, but if collecting old objects is more time-consuming then your collector becomes sensitive to object lifetime.

There are a number of techniques that can help reduce the frequency of GC events. Using a profiler can help identify the hot methods in your application that may be worth examining for code that can be optimised for this purpose. There are a few basic principles to keep in mind.

Reducing allocation rate

Reducing allocation rate is ultimately all about putting off GC pauses. There are two solutions for getting rid of GC pauses in Java:

- Don't generate any garbage.
- Use Azul Zing.

However, for projects that have latency performance goals that are less extreme, tuning at a code level, as opposed to adjusting the JVM GC values, can be worthwhile.

On every project I've worked on, the biggest performance improvements have often come from simply reducing the allocation rate. Many others in the field, including Martin Thompson, Kirk Pepperdine, and Todd Montgomery, have shared similar experiences.[1] Kirk Pepperdine has even offered a rule of thumb:

Reducing allocation rate will reduce the frequency of pauses, but it won't in and of itself have an impact on the length of a pause. If your problem is long pauses, allocation is not the cause.

> *"My current threshold for starting to look at memory efficiency as a way of improving performance is ~3-400 MB/sec.... If allocation rates exceed that then I know I can get significant gains in performance [by] working on memory efficiency."*

To do this, you'll need to work with a profiler. An effective technique is to target the top five to 10 methods and choose the one that is quickest and easiest to fix. The reason for this is that once you change one method, the profile is likely to be different next time, sometimes dramatically so. Using this approach therefore enables you to get the fastest result for the minimum effort.

Avoid creating unnecessary objects

Avoid creating objects when you don't need to. For example, there is no need to create an extra Integer to parse a String containing an int value:

```java
String meaningOfLife = "42";
```

[1] https://groups.google.com/forum/#!msg/mechanical-sympathy/jdIhW0TaZQ4/UyXPDGQVVngJ

```
int myInt = new Integer(meaningOfLife).intValue();
```

Instead, there is a static method available for parsing:

```
int myInt = Integer.parseInt(meaningOfLife);
```

Unfortunately, some classes do not provide static methods that avoid the spurious intermediate creation of objects. When a class does not provide a static method, you can sometimes use a dummy method to execute instance methods repeatedly, thus avoiding the need to create extra objects.

Another basic technique is to reduce the number of temporary objects being used, especially in loops. It is surprisingly easy (and common) to make a loop that has side effects such as making copies. The effect of this can be dramatic with a loop that is hit repeatedly; I once saw a project go from consistently missing its performance-testing targets to consistently passing by dint of a one-line change to remove a single object creation from a heavily utilised loop.

Using primitives

The primitive data types in Java use memory that does need to be reclaimed, but the overhead of doing so is smaller: it is reclaimed when holding the object and so has no additional impact. For example, an object with just one instance variable containing an int is reclaimed in one object reclaim. If the same object holds an Integer, the garbage collector needs to reclaim two objects. Moreover, temporary primitive data types exist only on the stack and do not need to be garbage collected at all.

Reducing garbage collection by using primitive data types also applies when you have a choice of formats in which to hold an object. As an example, if you had a large number of objects, each with a String instance variable holding a number, as in our “meaning of life” example above, it is better to make the instance variable an int data type and store the numbers as ints, providing the conversion overhead does not swamp the benefits of holding the values in this alternative format.

Similarly, you can hold a Date object as an `int` (or `long`), thus creating one less object and saving the associated GC overhead. Of course, this is another trade-off since those conversion calculations may take up more time.

Arrays are more problematic. Whilst current versions of the Java compiler support arrays or maps with a primitive key or value type through the use of “boxing” — wrapping the primitive value in a standard object which can be allocated and recycled by the GC — Java implements most collections using internal arrays. What this means is that for each key/value entry added to a HashMap, an internal object is allocated to hold both values. This is a necessary evil when dealing with maps, but means an extra allocation and possible deallocation made every time you put an item into a map. There is also the possible penalty of outgrowing capacity and having to reallocate a new internal array. When dealing with large maps containing thousands or more entries, these internal allocations can have significant cost in terms of GC.

The standard `Integer.valueOf` method caches the values between -128 and 127, but for each number outside that range a new object will be allocated, in addition to the internal key/value entry object. This can potentially more than triple GC overhead for the map.

This is being worked on for a future version of Java but until then, there are some libraries that provide primitive trees, maps, and lists for each of Java’s primitive types. Trove[2] is one example that I’ve used and found to be particularly good. The GS Collections[3] framework that Goldman Sachs open sourced in January 2012 also has support for primitive collections as well as optimised replacements for the standard JDK collections classes like `ArrayList`, `HashSet`, and `HashMap`.

Array-capacity planning

The convenience of Java’s dynamic collections, such as `ArrayLists`, make it easy to overuse them. `ArrayLists`, `HashMaps`, and `TreeMaps` are implemented using underlying `Object[]` arrays. Like **`Strings`** (which are wrappers over `char[]` arrays), array size is immutable.

Consider this piece of code:

```
List<Widget> items = new ArrayList<Widget>();

int x = 20;

for (int i = 0; i < x; i++) {
  Widget item = readNextItem();
  widget.add(item);
}
```

[2] http://trove.starlight-systems.com

[3] https://github.com/goldmansachs/gs-collections

The value of x determines the size of the ArrayList once the loop has finished, but this value is unknown to the ArrayList constructor which therefore allocates a new Object[] array of default size. Whenever the capacity of the internal array is exceeded, it is replaced with a new array of sufficient length, making the previous array garbage.

To avoid this, whenever possible allocate lists and maps with an initial capacity:

```
List<MyObject> items = new ArrayList<MyObject>(len);
```

This ensures that no allocations and deallocations of internal arrays occur at runtime as the list now has sufficient capacity to begin with. If you don't know the actual size then it is still generally better to go with an estimate with some buffer in it.

How long is a piece of string?

String handling can be a significant problem because it is easy to unintentionally create implicit strings. The wide use of XML and JSON encodings, as well as the tag-value-encoded FIX protocol in financial systems, means that we have to deal with text even on the most performance-sensitive systems.

Strings are immutable; they cannot be modified after allocation. Operators such as "+" for concatenation actually allocate a new string containing the contents of the strings being joined. In fact there is an implicit `StringBuilder` object that's allocated to do the work of combining them. So, for a piece of code like:

```
a = a + b; // a and b are Strings
```

The compiler generates this code behind the scenes:

```
a = a + b; // a and b are Strings StringBuilder temp =
new StringBuilder(a).
temp.append(b);
a = temp.toString(); // a new String is allocated here.
```

The previous "a" is now garbage.

Now have a look at this example:

```
String result = foo() + arg;
result += boo();
System.out.println("result = " + result);
```

In this case, three StringBuilder objects are allocated in the background — one for each "+" operation. There are also two additional strings being created — one to hold the result of the second assignment and another to hold the string passed into the print method. That's five additional objects in what would otherwise appear to be a fairly trivial statement.

If you are writing your own parser, writing your own string-handling classes could be worthwhile. At the very least, use StringBuffer in preference to a string-concatenation operator (+), and be aware of which methods alter objects directly as opposed to making copies and which ones return a copy of the object. For example, any String method that changes the string, such as String.trim(), returns a new string object, whilst a method like Vector.setSize() does not. If you do not need a copy, use (or create) methods that do not return a copy of the object.

Weak references

Back in Part 1, we introduced the heap and pointers with a fairly simplified model, and stated that garbage collection works through following chains of pointers. Things do get a little more complicated than this, however. One thing to keep in mind is that a reference may be either strong or weak. A strong reference is an ordinary Java reference. A line of code such as:

```
StringBuffer buffer = new StringBuffer();
```

creates a new StringBuffer() and stores a strong reference to it in the variable buffer. If an object is strongly reachable, i.e. it is reachable via a chain of strong references, it is not eligible for garbage collection.

A weak reference (java.lang.ref.WeakReference) is a reference that isn't strong enough to force an object to remain in memory. You create a weak reference like this:

```
WeakReference<Widget> weakWidget =
      new WeakReference<Widget>(widget);
```

Elsewhere in your code, you can use weakWidget.get() to get a strong reference to the actual widget object. Since the weak reference itself isn't strong enough to prevent garbage collection, you may find that the widget.get() suddenly starts returning null. Once a WeakReference starts returning null, the object it pointed to has become garbage and the `WeakReference` object is essentially useless. This generally means that some sort of cleanup is required; `WeakHashMap`, for example, has to remove such defunct entries to avoid holding on to an ever-increasing number of dead `WeakReferences`.

The `ReferenceQueue` class makes it easy to keep track of dead references. If you pass a `ReferenceQueue` into a weak reference's constructor, the reference object will be automatically inserted into the reference queue when the object to which it pointed becomes garbage. You can then, at some regular interval, process the `ReferenceQueue` and perform whatever cleanup is needed for dead references.

Weak references can be generalised to provide multiple levels of weak pointers in addition to strong references, and Java is, as far as I know, unique in doing this. In effect, Java has a hierarchy of pointer strengths. They are, from strongest to weakest: strong, soft, weak, finaliser, and phantom.

The collector can clear a soft reference at its discretion, based on current space usage. Soft references aren't required to behave any differently than weak references, but the OpenJDK server JRE will attempt to hold on to an object referenced by a soft reference if there isn't pressure on the available memory. Whilst this may appear to suggest that soft references make a good foundation for a cache, caches based on soft references tend to exhibit uncontrolled and semi-random effects ranging from thrashing on GC work to emptying the entire cache periodically, and everything in between. It is fundamentally a bad idea to hope that a collector will be able to "intelligently" manage the empty space amongst the various consumers of soft references in the heap without driving them (or itself) into thrashing conditions. As a result, we find that well-tuned JVMs that run code that does make use of soft references will tend to set the soft-reference reclamation policy to make soft references equivalent to weak references (e.g. by setting `-XX:SoftRefLRUPolicyMSPerMB=0`).

Depending on the specific algorithm that is used, extensive use of weak and/or soft references can increase the length of stop-the-world pauses you see during collection.

Finally, a phantom reference (`java.lang.ref.PhantomReference`) is the weakest kind of weak reference in Java. If you try and retrieve the associated object, it will return null, which means they can be instantly cleaned up.

Most collectors require a stop-the-world event to process soft and weak references, so overall application performance can be adversely affected by their use. Azul C4 is an exception since the C4 mark phase also performs concurrent processing of soft, weak, final, and phantom references. This quality makes the collector relatively insensitive to the number of soft or weak references used by the application.

Try-with-resources

Java 7 introduced the `try-with-resources` statement. This is somewhat analogous to C#'s "using" statement or the approach used in C++ where the class implementor would define a destructor function that performs the cleanup whenever an object of that class goes out of scope. The advantage of this approach is that the user of the object can't forget to clean it up — the destructor gets called automatically, even if an exception is thrown. This approach is known by the frankly terrible name of RAII, for "resource acquisition is initialisation".

A try-with-resources statement can have catch and finally blocks just like an ordinary try statement. In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

The following example uses a try-with-resources statement to automatically close a `FileInputStream` object.

```
import java.io.FileInputStream;
import java.io.IOException;

public class tryWithResources {

  private static void printFile() throws IOException {

    try (FileInputStream input = new FileInputStream
    ("file.txt")) {

      int data = input.read();
      while(data != -1){
        System.out.print((char) data);
        data = input.read();
      }
    }
  }
}
```

Distributing programs

A widely used technique amongst enterprise Java teams is to distribute programs. This can both keep the heap size smaller, making the pauses shorter, and allow some requests to continue whilst others are paused. In certain specific situations, this may be the correct thing to do from an engineering standpoint as well — the example I cited in the preface, of a web application that was also required to perform ad hoc batch-job-type functions, absolutely required breaking into parts and distributing in order for it to work.

Some organisations extend the idea further by implementing rolling restarts. This is a technique in which a cluster of machines have their heap sizes set to avoid a major GC event during an operating window, say a trading day, and individual machines are then restarted either during or at the end of the window. Implemented correctly, this approach can eliminate major pauses, though it won't get rid of minor GC events. It does also introduce some other problems, notably that the JVM needs to be warmed up again after a restart before it will run at optimum speed. Azul's ReadyNow product has been specifically designed with that problem in mind.

Other common techniques

Java teams also use approaches such as object pooling (though this should only be used for certain objects that are extremely expensive to create, such as database connections), using fixed-sized objects to prevent fragmentation, and using off-heap storage. In other words, it is possible to write code in Java that generates very little garbage, and that is certainly a way around the problems that collection pauses can cause. However, if you go down that path, you are also going to have to severely limit the number of third-party libraries you depend on since very few, if any, are written with the same constraints.

Suggestions for further reading

I hope you've found this book useful. If you want to learn more, here are some suggestions.

The definitive book on garbage collection is *The Garbage Collection Handbook: The Art of Automatic Memory Management* by Richard Jones, Antony Hosking, and Eliot Moss (Chapman & Hall/CRC Applied Algorithms and Data Structure Series): http://gchandbook.org

The Parallel collector is described in "Parallel Garbage Collection for Shared Memory Multiprocessors" by Christine H. Flood et al., available via USENIX here: https://www.usenix.org/legacy/event/jvm01/full_papers/flood/flood.pdf

CMS is described in "A Generational Mostly-concurrent Garbage Collector" by Tony Printezis and David Detlefs, available on CiteSeerx here: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8915

The Garbage First collector is described in more detail in the "Garbage-first garbage collection" paper by David Detlefs et al., available from ACM here: http://dl.acm.org/citation.cfm?id=1029879

IBM's Metronome collector is described in more detail in "Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for Java" by David Bacon, Perry Chang, and V. T. Rajan, available from ACM here: http://dl.acm.org/citation.cfm?id=780744

Donald Raab, creator of the GS Collections library, has published a good introduction to it on InfoQ:

http://www.infoq.com/articles/GS-Collections-by-Example-1

http://www.infoq.com/articles/GS-Collections-by-Example-2