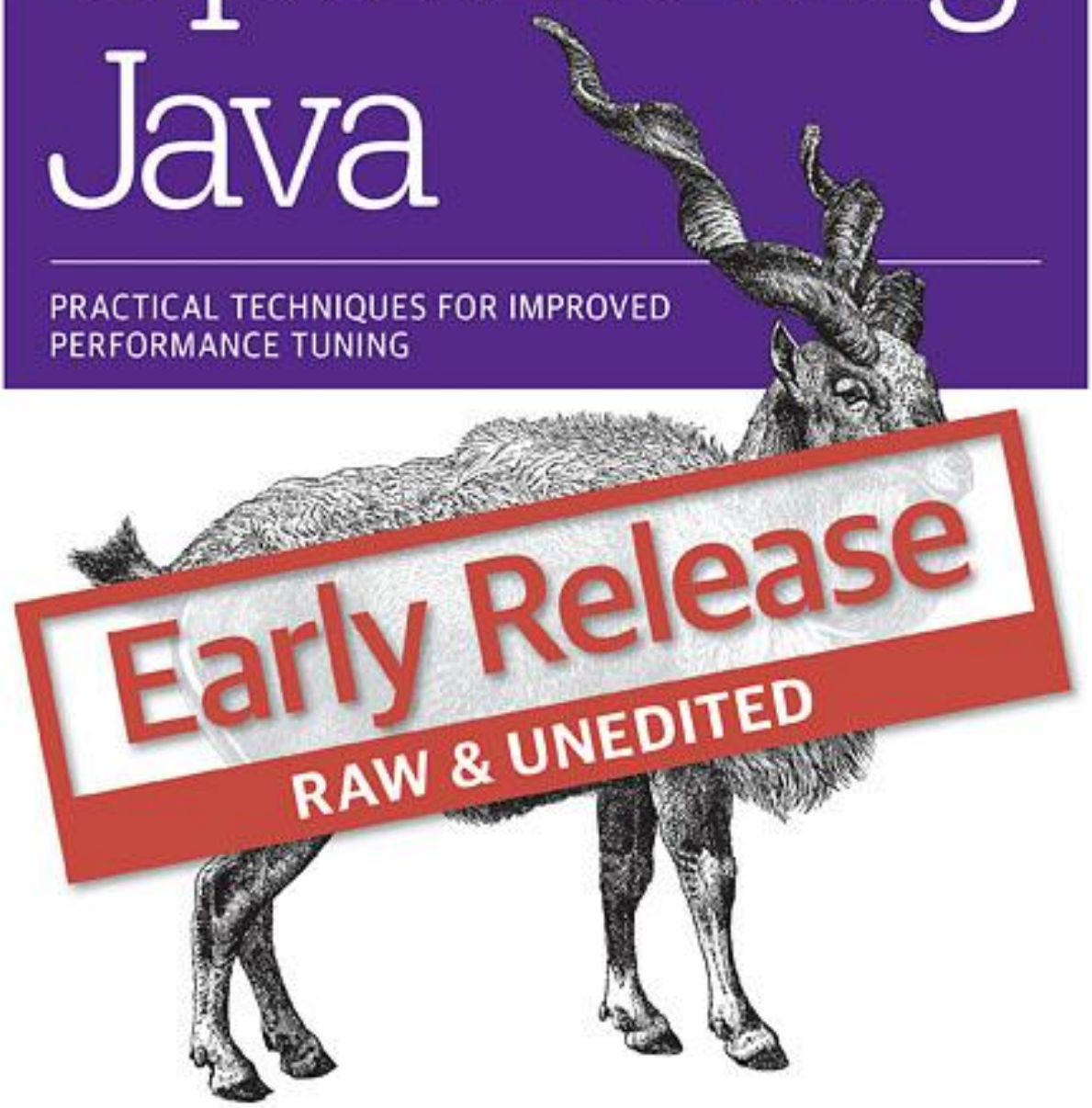
O'REILLY®

Optimizing Java

PRACTICAL TECHNIQUES FOR IMPROVED PERFORMANCE TUNING



Benjamin J. Evans & James Gough

Optimizing Java

by Benjamin J Evans, James Gough
Publisher: O'Reilly Media, Inc.
Release Date: January 2018
ISBN: 9781491933329
Topic: Java

Book Description

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. You'll also receive updates when significant changes are made, new chapters are available, and the final ebook bundle is released.

Currently, no books exist that focus on the practicalities of Java application performance tuning, as opposed to the theory and internals of Java virtual machines. This practical guide is the "missing link" that aims to move Java performance tuning from the realm of guesswork and folklore to an experimental science.

- Learn how to approach performance problems in an consistent and systematic way
- Resolve production performance issues by learning core Java performance topics
- Identify and resolve performance issues before encountering them in production
- Understand the performance problems you encounter by learning the Java platform's internals

Chapter 1. Optimization and Performance Defined

Optimizing the performance of Java (or any other sort of code) is often seen as a Dark Art. There's a mystique about performance analysis - it's often seen as a craft practiced by the "lone hacker, who is tortured and deep thinking" (one of Hollywood's favourite tropes about computers and the people who operate them). The image is one of a single individual who can see deeply into a system and come up with a magic solution that makes the system work faster.

This image is often coupled with the unfortunate (but all-too common) situation where performance is a second-class concern of the software teams. This sets up a scenario where analysis is only done once the system is already in trouble, and so needs a performance "hero" to save it. The reality, however, is a little different…

Java Performance - The Wrong Way

For many years, one of the top 3 hits on Google for "Java Performance Tuning" was an article from 1997-8, which had been ingested into the index very early in Google's history. The page had presumably stayed close to the top because its initial ranking served to actively drive traffic to it, creating a feedback loop.

The page housed advice that was completely out of date, no longer true, and in many cases detrimental to applications. However, the favoured position in the search engine results caused many, many developers to be exposed to terrible advice.

For example, very early versions of Java had terrible method dispatch performance. As a workaround, some Java developers advocated avoiding writing small methods and instead writing monolithic methods. Of course, over time, the performance of virtual dispatch greatly improved. Not only that, but with modern JVMs and especially automatic managed inlining, virtual dispatch is now eliminated at the majority of call sites. Code that followed the "lump everything into one method" advice would now be at a substantial disadvantage, as it would be very unfriendly to modern JIT compilers.

There's no way of knowing how much damage was done to the performance of applications that were subjected to the bad advice, but it neatly demonstrates the dangers of not using a quantitative and verifiable approach to performance. It also provides another excellent example of not believing everything you read on the Internet.

Note

The execution speed of Java code is highly dynamic and fundamentally depends on the underlying Java Virtual Machine (JVM). An old piece of Java code may well execute faster on a more recent JVM, even without recompiling the Java source code.

As you might imagine, for this reason (and others we will discuss later) this book does not consist of a cookbook of performance tips to apply to your code. Instead, we focus on a range of aspects that come together to produce good performance engineering:

- Performance methodology within the overall software lifecycle
- Theory of testing as applied to performance
- Measurement, statistics and tooling
- Analysis skills (both systems and data)
- Underlying technology and mechanisms

Later in the book, we will introduce some heuristics and code-level techniques for optimization, but these all come with caveats and tradeoffs that the developer should be aware of before using them.

Note

Please do not skip ahead to those sections and start applying the techniques detailed without properly understanding the context in which the advice is given. All of these techniques are capable of doing more harm than good without a proper understanding of how they should be applied.

In general, there are:

- No magic “go-faster” switches for the JVM
- No “Tips and tricks” to make Java run faster
- No secret algorithms that have been hidden from you

As we explore our subject, we will discuss these misconceptions in more detail, along with some other common mistakes that developers often make when approaching Java performance analysis and related issues. Still here? Good. Then let’s talk about performance.

Java Performance Overview

To understand why Java performance is the way that it is, let’s start by considering a classic quote from James Gosling, the creator of Java.

> Java is a blue collar language. It’s not PhD thesis material but a language for a job.
>
> James Gosling

That is, Java has always been an extremely practical language. Its attitude to performance was initially that as long as the environment was *fast enough*, then raw performance could be sacrificed if developer productivity, benefited. It was therefore not until relatively recently, with the increasing maturity and sophistication of JVMs

such as HotSpot that the Java environment became suitable for high-performance computing applications.

This practicality manifests itself in many ways in the Java platform, but one of the most obvious is the used of *managed subsystems*. The idea is that the developer gives up some aspects of low-level control in exchange for not having to worry about some of the details of the capability under management.

The most obvious example of this is, of course, memory management. The JVM provides automatic memory management in the form of a pluggable garbage collection subsystem, so that memory does not have to be manually tracked by the programmer.

Note

Managed subsystems occur throughout the JVM and their existence introduces extra complexity into the runtime behavior of JVM applications.

As we will discuss in the next section, the complex runtime behavior of JVM applications requires us to treat our applications as experiments under test. This leads us to think about the statistics of observed measurements, and here we make an unfortunate discovery.

The observed performance measurements of JVM applications are very often not normally-distributed. This means that elementary statistical techniques (e.g. *standard deviation* and *variance*) are ill-suited for handling results from JVM applications. This is because many basic statistics methods contain an implicit assumption about the normality of results distributions.

One way to understand this is that for JVM applications outliers can be very significant - for a low-latency trading application for example. This means that sampling of measurements is also problematic, as it can easily miss the exact events that have the most importance.

Finally, a word of caution. It is very easy to be misled by Java performance measurements. The complexity of the environment means that it is very hard to isolate individual aspects of the system.

Measurement also has an overhead, and frequent sampling (or recording every result) can have an observable impact on the performance numbers being recorded. The nature of Java performance numbers requires a certain amount of statistical sophistication, and naive techniques famously produce incorrect results when applied to Java / JVM applications.

Performance as an Experimental Science

Java / JVM software stacks are, like most modern software systems, very complex. In fact, due to the highly optimizing and adaptive nature of the JVM, production systems built on top of the JVM can have some incredibly subtle and intricate performance behavior. This complexity has been made possible by Moore's Law and the unprecedented growth in hardware capability that it represents.

> The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.
>
> Henry Petroski

Whilst some systems have squandered the gains of Moore's Law, the JVM represents something of an engineering triumph, and has produced a very high performance, general purpose execution environment that puts those gains to very good use. The tradeoff, however, is that like any complex, high performance system, the JVM requires a measure of skill and experience to truly get the best out of it.

JVM Performance tuning is therefore a synthesis between technology, methodology, measurable quantities and tools. Its aim is to affect measurable outputs in a manner desired by the owners or users of a system. In other words, performance is an experimental science - it achieves a desired result by:

-

Defining the desired outcome

-
-

Measuring the existing system

-
-

Determining what is to be done to achieve the requirement

-
-

Undertaking an improvement exercise

-

-

Retesting

-
-

Determining whether the goal has been achieved

-

The process of defining and determining desired performance outcomes builds a set of quantative objectives. It is important to establish what should be measured and record the objectives, which then forms part of the project artefacts and deliverables. From this, we can see that performance analysis is based upon defining, and then achieving non-functional requirements.

This process is, as has been previewed, not one of reading chicken entrails or other divination method. Instead, this relies upon statistics and an appropriate handling of results. In Chapter 5 we will introduce a primer on the basic statistical techniques that are required for accurate handling of data generated from a JVM performance analysis project.

For many real-world projects, a more sophisticated understanding of data and statistics will undoubtedly be required. The advanced reader is encouraged to view the statistical techniques found in this book as a starting point, rather than a definitive statement.

A Taxonomy for Performance

In this section, we introduce some basic performance metrics. These provide a vocabulary for performance analysis and allow us to frame the objectives of a tuning project in quantitative terms. These objectives are the non-functional requirements that define our performance goals. One common basic set of performance metrics is:

-

Throughput

-
-

Latency

-

-

Capacity

-
-

Utilization

-
-

Efficiency

-
-

Scalability

-
-

Degradation

-

We will briefly discuss each in turn. Note that for most performance projects, not every metric will be optimised simultaneously. The case of only a few metrics being improved in a single performance iteration is far more common, and may be as many as can be tuned at once. In real-world projects, it may well be the case that optimizing one metric comes at the detriment of another metric or group of metrics.

Throughput

Throughput is a metric that represents the rate of work a system or subsystem can perform. This is usually expressed as number of units of work in some time period. For example, we might be interested in how many transactions per second a system can execute.

For the throughput number to be meaningful in a real performance exercise, it should include a description of the reference platform it was obtained on. For example, the hardware spec, OS and software stack are all relevant to throughput, as is whether the system under test is a single server or a cluster. In addition, transactions (or units of work) should be the same between tests. Essentially, we should seek to ensure that the workload for throughput tests is kept consistent between runs.

Latency

Performance metrics are sometimes explained via metaphors that evokes plumbing. If a water pipe can produce 100l per second, then the volume produced in 1 second (100 litres) is the throughput. In this metaphor, the latency is effectively the length of the pipe. That is, it's the time taken to process a single transaction and see a result at the other end of the pipe.

It is normally quoted as an end-to-end time. It is dependent on workload, so a common approach is to produce a graph showing latency as a function of increasing workload. We will see an example of this type of graph in Section 1.4

Capacity

The capacity is the amount of work parallelism a system possesses. That is, the number units of work (e.g. transactions) that can be simultaneously ongoing in the system.

Capacity is obviously related to throughput, and we should expect that as the concurrent load on a system increases, that throughput (and latency) will be affected. For this reason, capacity is usually quoted as the processing available at a given value of latency or throughput.

Utilisation

One of the most common performance analysis tasks is to achieve efficient use of a systems resources. Ideally, CPUs should be used for handling units of work, rather than being idle (or spending time handling OS or other housekeeping tasks).

Depending on the workload, there can be a huge difference between the utilisation levels of different resources. For example, a computation-intensive workload (such as graphics processing or encryption) may be running at close to 100% CPU but only be using a small percentage of available memory.

Efficiency

Dividing the throughput of a system by the utilised resources gives a measure of the overall efficiency of the system. Intuitively, this makes sense, as requiring more resources to produce the same throughput, is one useful definition of being less efficient.

It is also possible, when dealing with larger systems, to use a form of cost accounting to measure efficiency. If solution A has a total dollar cost of ownership (TCO) twice that of solution B for the same throughput then it is, clearly, half as efficient.

Scalability

The throughout or capacity of a system depends upon the resources available for processing. The change in throughput as resources are added is one measure of the scalability of a system or application. The holy grail of system scalability is to have throughput change exactly in step with resources.

Consider a system based on a cluster of servers. If the cluster is expanded, for example, by doubling in size, then what throughput can be achieved? If the new cluster can handle twice the volume of transactions, then the system is exhibiting “perfect linear scaling”. This is very difficult to achieve in practice, especially over a wide range of posible loads.

System scalability is dependent upon a number of factors, and is not normally a simple constant factor. It is very common for a system to scale close to linearly for some range of resources, but then at higher loads, to encounter some limitation in the system that prevents perfect scaling.

Degradation

If we increase the load on a system, either by increasing the number of requests (or clients) or by increasing the speed requests arrive at, then we may see a change in the observed latency and/or throughput.

Note that this change is dependent on utilisation. If the system is under-utilised, then there should be some slack before observables change, but if resources are fully utilised then we would expect to see throughput stop increasing, or latency increase. These changes are usually called the degradation of the system under additional load.

Connections between the observables

The behaviour of the various performance observables is usually connected in some manner. The details of this connection will depend upon whether the system is running at peak utility. For example, in general, the utilisation will change as the load on a system increases. However, if the system is under-utilised, then increasing load may not appreciably increase utilisation. Conversely, if the system is already stressed, then the effect of increasing load may be felt in another observable.

As another example, scalability and degradation both represent the change in behaviour of a system as more load is added. For scalability, as the load is increased, so are available resources, and the central question is whether the system can make use of them. On the other hand, if load is added but additional resources are not provided, degradation of some performance observable (e.g. latency) is the expected outcome.

Note

In rare cases, additional load can cause counter-intuitive results. For example, if the change in load causes some part of the system to switch to a more resource intensive, but higher performance mode, then the overall effect can be to reduce latency, even though more requests are being received.

To take one example, in Chapter 10 we will discuss HotSpot's JIT compiler in detail. To be considered eligible for JIT compilation, a method has to be executed in interpreted mode "sufficiently frequently". So it is possible, at low load to have key methods stuck in interpreted mode, but to become eligible for compilation at higher loads, due to increased calling frequency on the methods. Thiscauses later calls to the same method to run much, much faster than earlier executions.

Different workloads can have very different characteristics. For example, a trade on the financial markets, viewed end to end, may have an execution time (i.e. latency) of hours or even days. However, millions of them may be in progress at a major bank at any given time. Thus the capacity of the system is very large, but the latency is also large.

However, let's consider only a single subsystem within the bank. The matching of a buyer and a seller (which is essentially the parties agreeing on a price) is known as "order matching". This individual subsystem may have only hundreds of pending order at any given time, but the latency from order acceptance to completed match may be as little as 1 millisecond (or even less in the case of "low latency" trading).

In this section we have met the most frequently encountered performance observables. Occasionally slightly different defintions, or even different metrics are used, but in most cases these will be the basic system numbers that will normally be used to guide performance tuning, and act as a taxonomy for discussing the performance of systems of interest.

Reading performance graphs

To conclude this chapter, let's look at some common patterns of behavior that occur in performance tests. We will explore these by looking at graphs of real observables, and we will encounter many other examples of graphs of our data as we proceed.

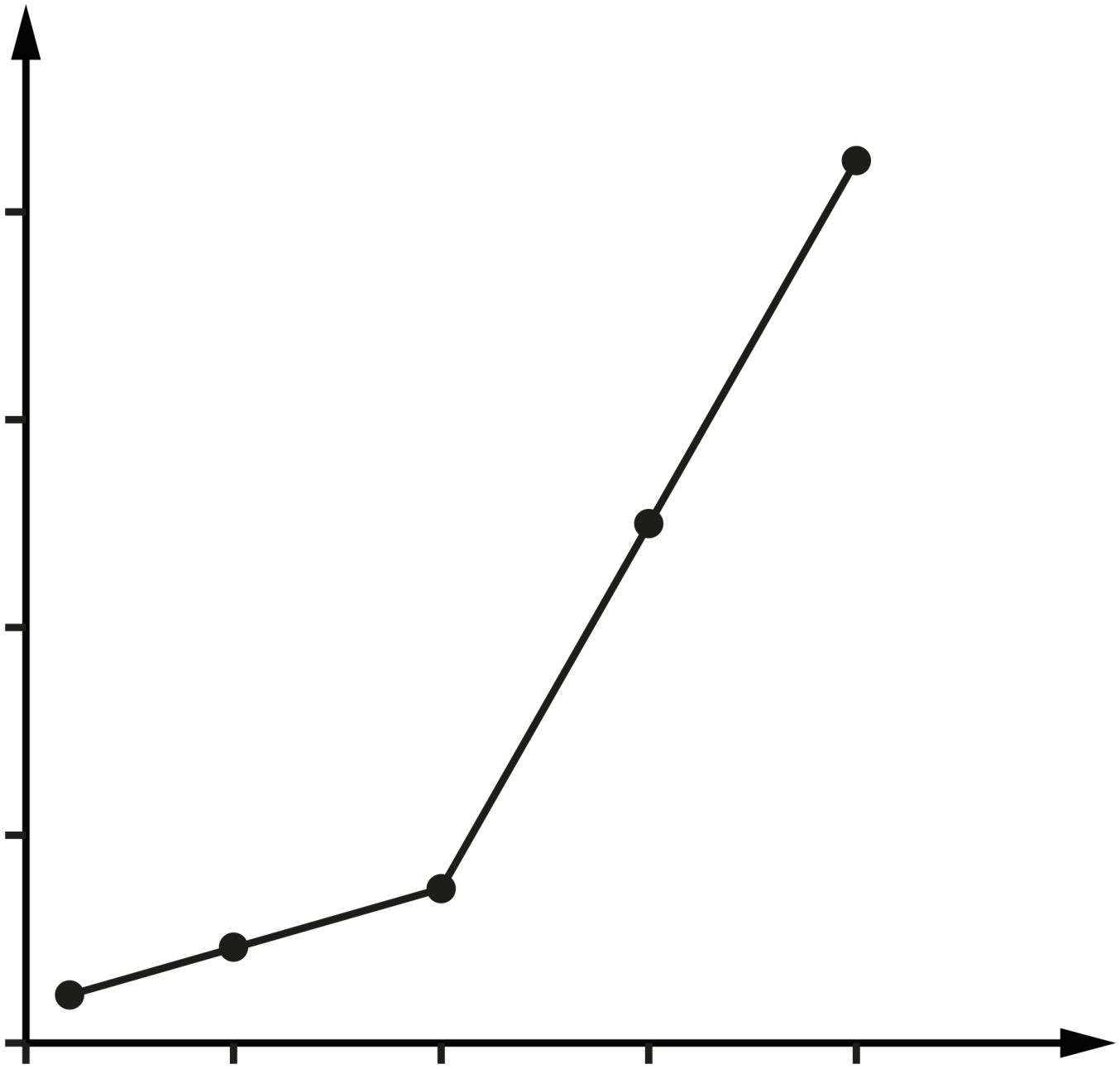


Figure 1-1. A performance “elbow”

The graph in Figure 1-1 shows sudden, unexpected degradation of performance (in this case, latency) under increasing load - commonly called a performance elbow.

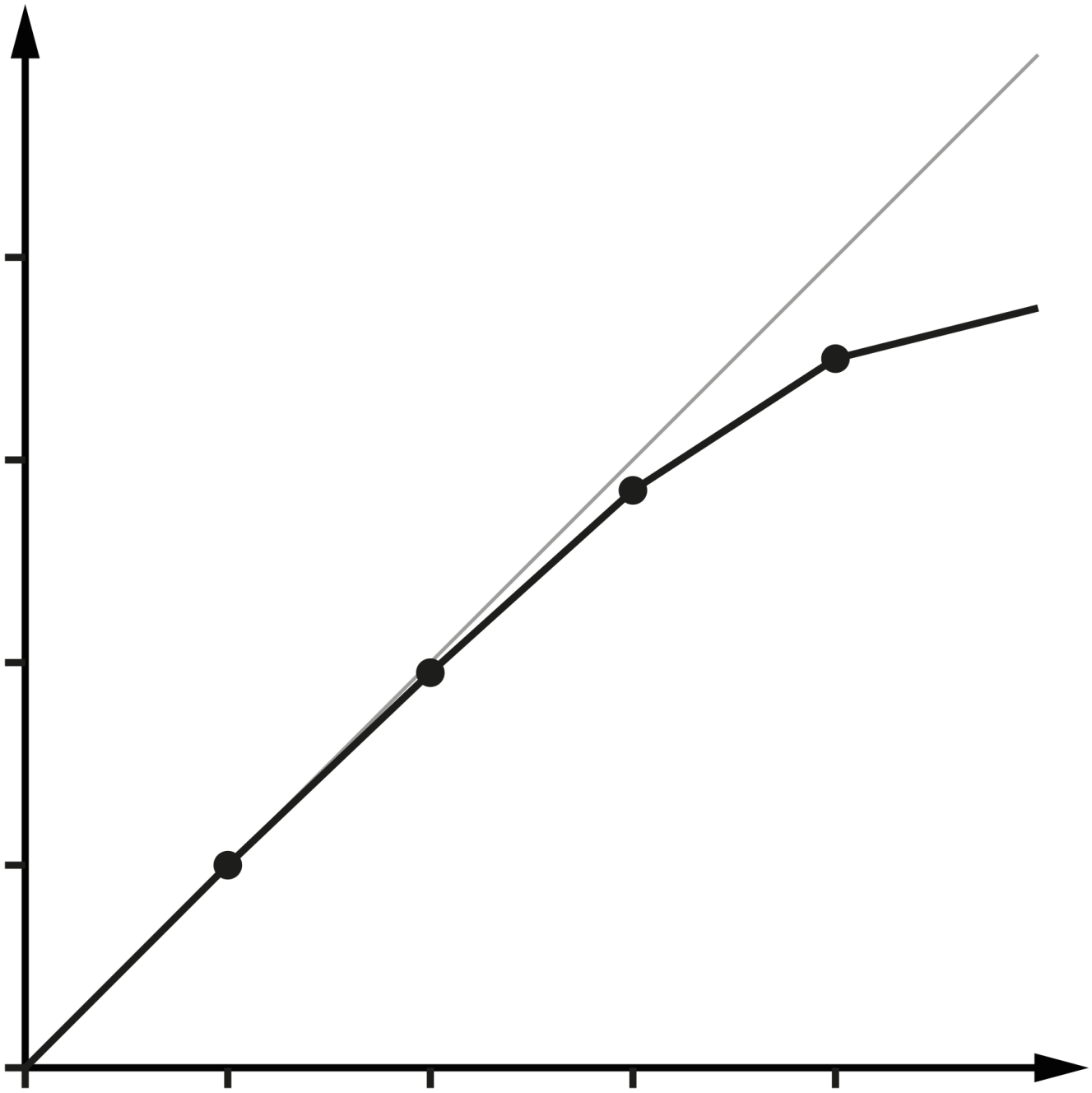


Figure 1-2. Near linear scaling

By contrast, Figure 1-2 shows the much happier case of throughput scaling almost linearly as machines are added to a cluster. This is close to ideal behaviour, and is only likely to be achieved in extremely favourable circumstances - e.g. scaling a stateless protocol with no need for session affinity with a single server.

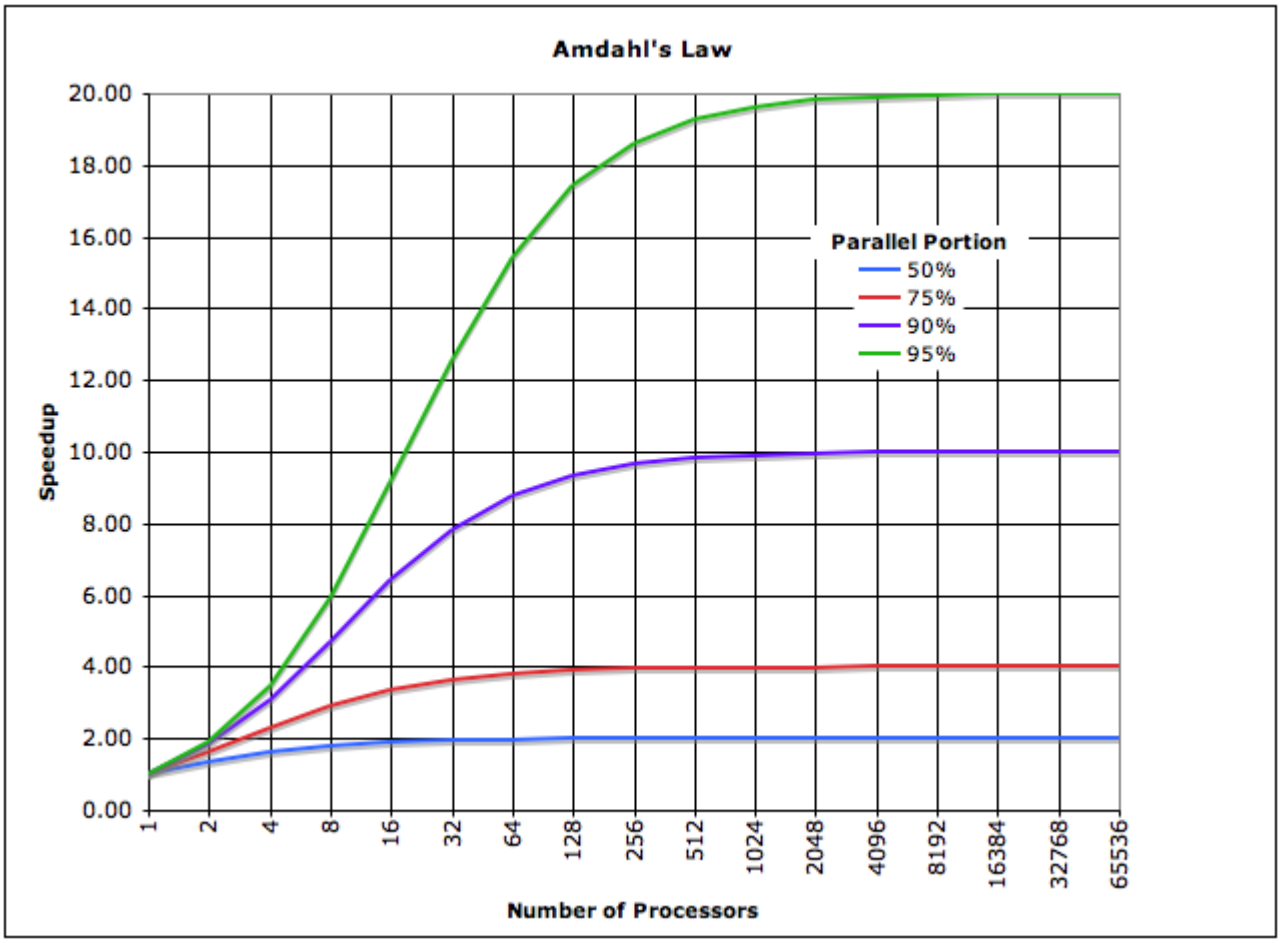


Figure 1-3. Amdahl's Law

In Chapter 15 we will meet Amdahl's Law, named for the famous computer scientist (and "father of the mainframe") Gene Amdahl of IBM. Figure 1-3 shows a graphical representation of this fundamental constraint on scalability. It shows that whenever the workload has any piece at all that must be performed serially, linear scalability is impossible, and there are strict limits on how much scalability can be achieved. This justifies the commentary around Figure 1-2 - even in the best cases linear scalability is all but impossible to achieve.

The limits imposed by Amdahl's Law are surprisingly restrictive - note in particular that the x-axis of the graph is logarithmic, and so even with an algorithm that is (only) 5% serial, 32 processors are needed for a factor-of-12 speedup, and that no matter how many cores are used, the maximum speedup is only a factor-of-20 for that algorithm. In practice, many algorithms are far more than 5% serial, and so have a more constrained maximum possible speedup.

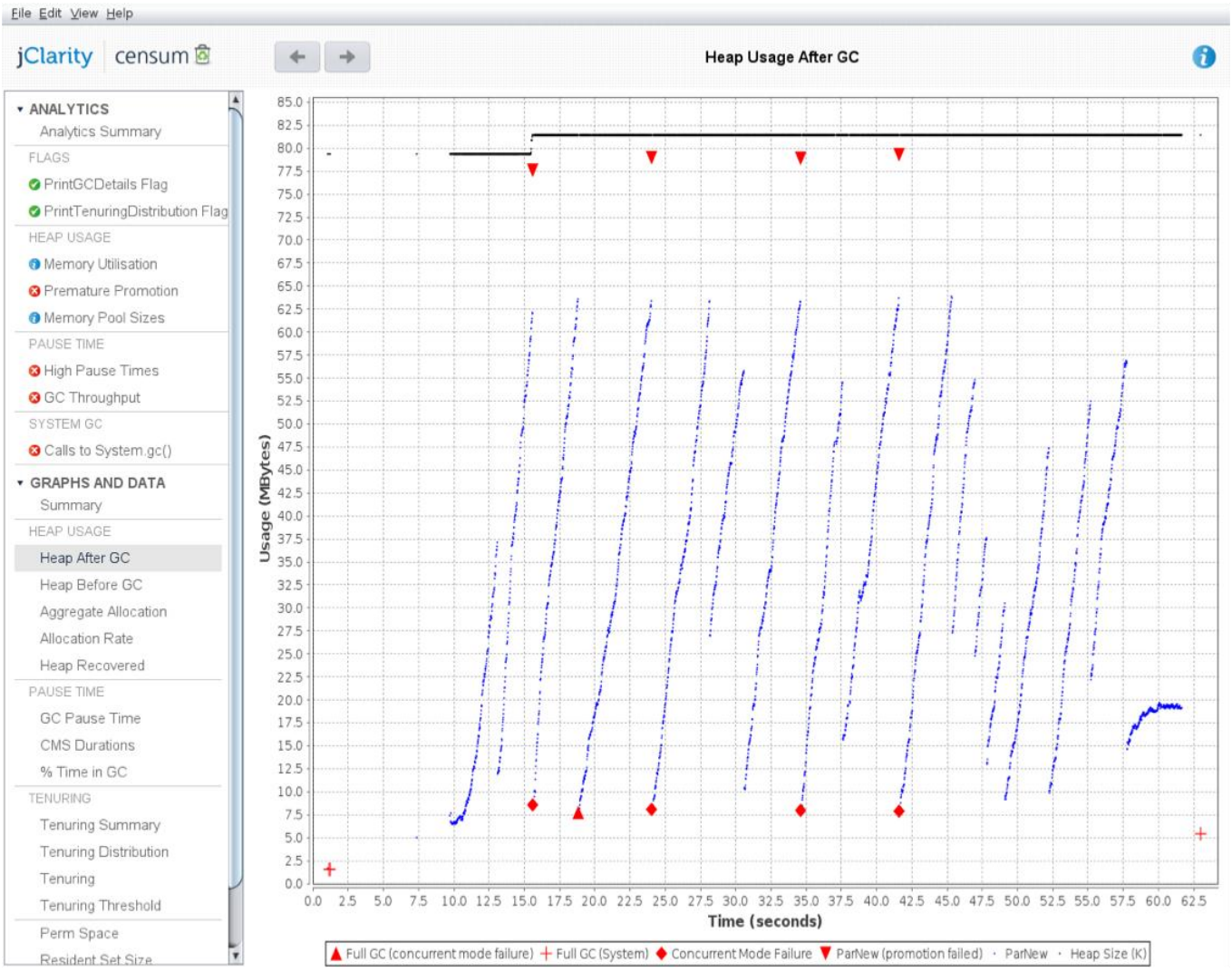


Figure 1-4. Healthy memory usage

As we will see in Chapter 7, the underlying technology in the JVMs garbage collection subsystem naturally gives rise to a "sawtooth" pattern of memory used for healthy applications that aren't under stress. We can see an example in Figure 1-4.

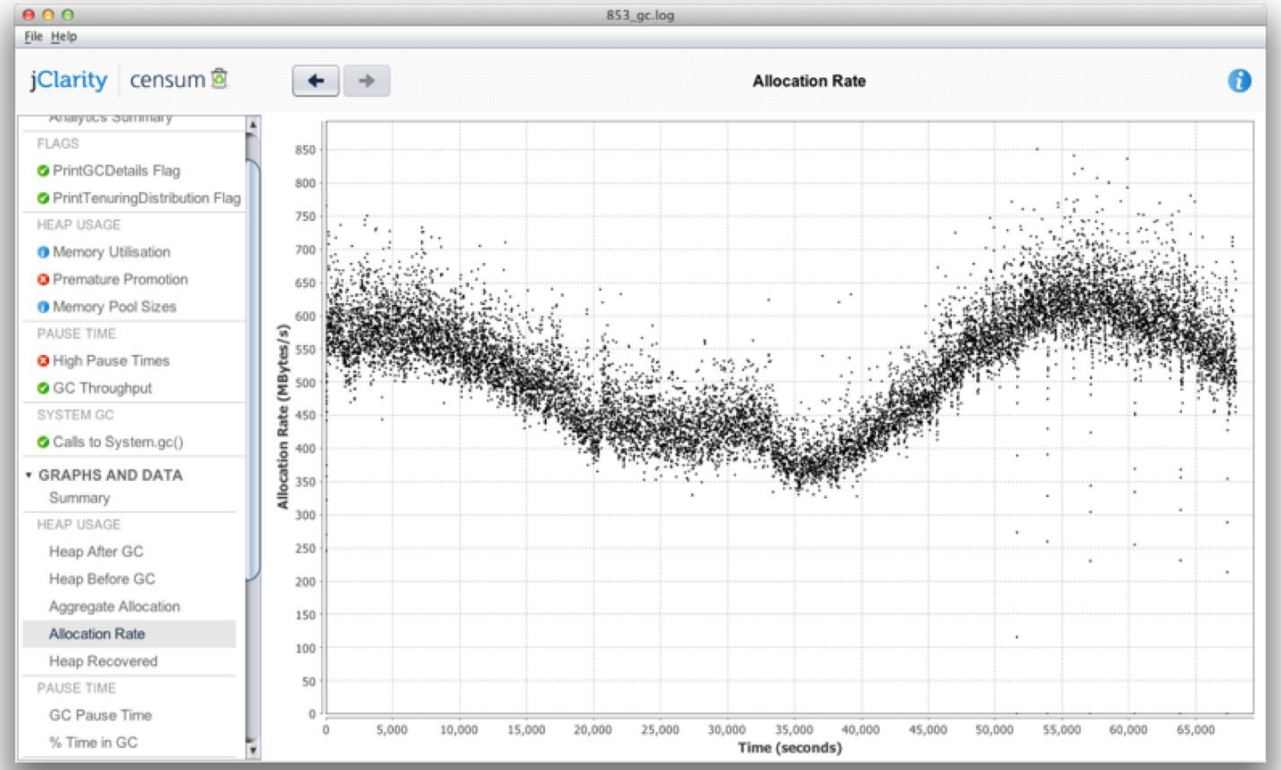


Figure 1-5. Stable allocation rate

In Figure 1-5, we show another memory graph that is very typical for a healthy application. The memory allocation rate can be of great importance when performance tuning an application. This example shows a clear signal of allocation varying gently over the course of a business day, but within well-defined limits (between roughly 300 and 750 MB/s) that lie well within the capability of modern server class hardware. We will have much more to say about this subject when we discuss garbage collection in Chapter 7.

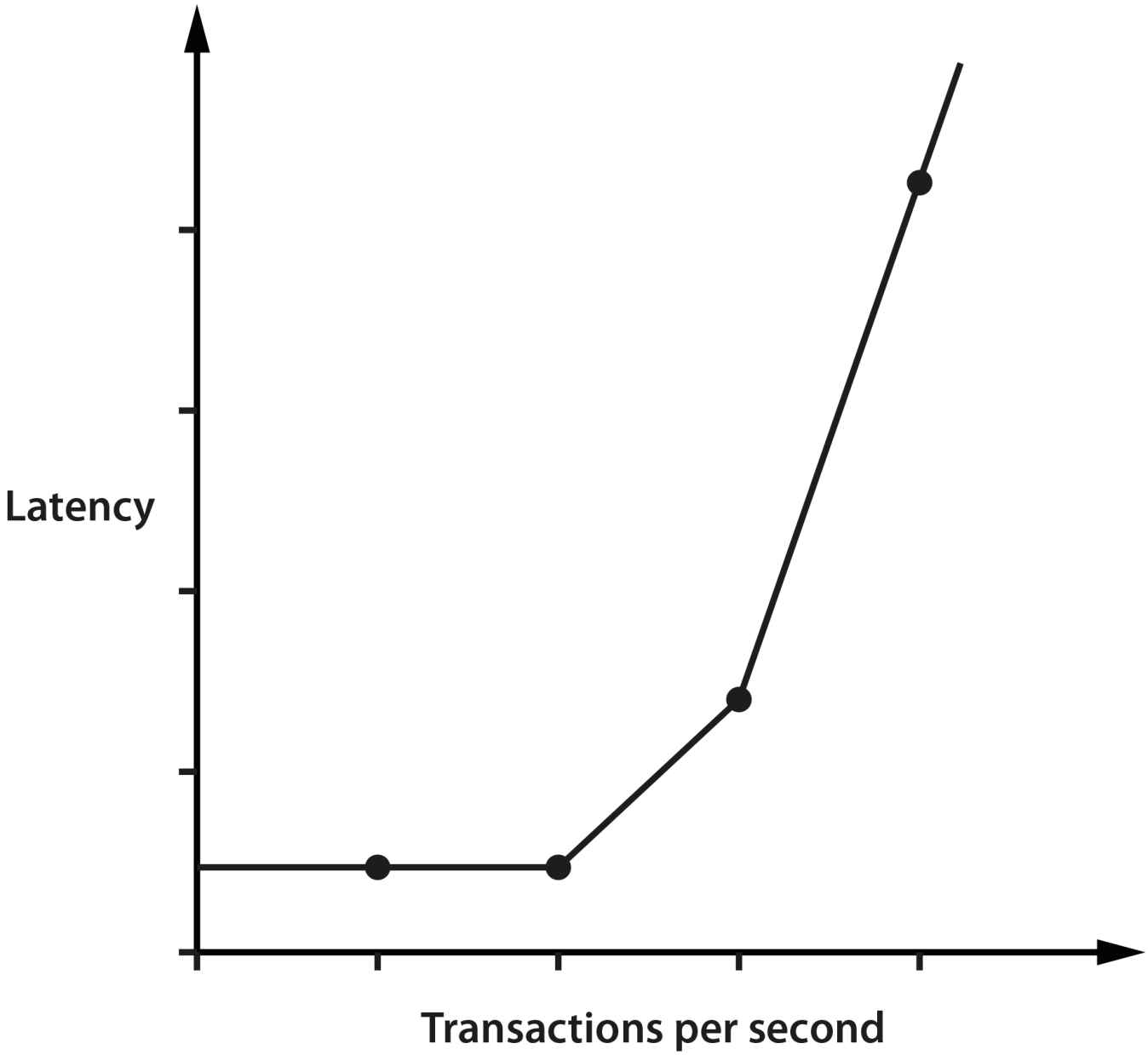


Figure 1-6. Degrading latency under higher load

In the case where a system has a resource leak, it is far more common for it to manifest in a manner like that shown in Figure 1-6, where an observable (in this case latency) to slowly degrade as the load is ramped up, before hitting an inflection point where the system rapidly degrades.

In this chapter we have started to discuss what Java performance is and is not. We have introduced the fundamental topics of empirical science and measurement, and introduced the basic vocabulary and observables that a good performance exercise will use. Finally, we have introduced some common cases that are often seen within the results obtained from performance tests. Let's move on and begin our discussion of

some of the major aspects of the JVM and set the scene for understanding the detail of what makes JVM-based performance optimization a particularly complex problem.

Chapter 2. Overview of the JVM

Overview

There is no doubt that Java is one of the largest technology platforms on the planet, boasting roughly 9-10 million Java developers (according to Oracle). By design, many developers do not need to know about the low level intricacies of the platform they work with. This leads to a situation where developers only meet these aspects when a customer complains about performance for the first time.

As a developer interested in performance, however, it is important to understand the basics of the JVM technology stack. Understanding JVM technology enables developers to write better software and provides the theoretical background required for investigating performance related issues.

This chapter introduces how the JVM executes Java in order to form the basis for deeper exploration of these topics later in the book. In particular, Chapter 10 has an in-depth treatment of bytecode. One strategy for the reader could be to read this chapter now, and then re-read it in conjunction with Chapter 10, once some of the other topics have been understood.

Interpreting and Classloading

According to the specification that defines the Java virtual machine (usually called the VM Spec), the JVM is a stack based interpreted machine. This means that rather than having registers (like a physical hardware CPU), it uses an execution stack of partial results, and performs calculations by operating on the top value (or values) of that stack.

The basic behavior of the JVM interpreter can be thought of as essentially a "switch inside a while loop" - processing each opcode of the program independently of the last using the evaluation stack to hold intermediate values.

Note

As we will see when we delve into the internals of the Oracle / OpenJDK VM (Hotspot), the situation for real production-grade Java interpreters can be more complex, but *switch-inside-while* is an acceptable mental model for the moment.

When our application is launched using the `java HelloWorld` command, the operating system starts the virtual machine process (the `java` binary). This sets up the Java

virtual environment and initialises the stack machine that will actually execute the user code in the `HelloWorld` class file.

The entry point into the application will be the `main()` method of `HelloWorld.class`. In order to hand over control to this class, it must be loaded by the virtual machine before execution can begin.

To achieve this, the Java classloading mechanism is used. When a new Java processes is initialising, a chain of class loaders is used. The initial loader is known as the Bootstrap classloader, and contains classes in the core Java runtime. In current versions these are loaded from `rt.jar`, although this is changing in Java 9, as we will see in Chapter 16.

The main point of the Bootstrap classloader is to get a minimal set of classes (which includes essentials such as `java.lang.Object`, `Class` and `Classloader`) loaded to allow other classloaders to bring up the rest of the system.

Note

Java models classloaders as objects within its own runtime and type system, so there needs to be some way to bring an initial set of classes into existence or otherwise there would be a circularity problem in defining what a classloader is.

The Extension classloader is created next, it defines its parent to be the Bootstrap classloader, and will delegate to parent if needed. Extensions are not widely used, but can supply overrides and native code for specific operating systems and platforms. Notably, the Nashorn Javascript runtime introduced in Java 8 is loaded by the Extension loader.

Finally the Application classloader is created, which is responsible for loading in user classes from the defined classpath. Some texts unfortunately refer to this as the "System" classloader. This term should be avoided, for the simple reason that it doesn't load the system classes (the Bootstrap classloader does). The Application classloader is encountered extremely frequently, and it has the Extension loader as its parent.

Java loads in dependencies on new classes when they are first encountered during the execution of the program. If a class loader fails to find a class the behavior is usually to delegate the lookup to the parent. If the chain of lookups reaches the bootstrap class loader and isn't found a `ClassNotFoundException` will be thrown. It is important that developers use a build process that effectively compiles with the exact same classpath that will be used in Production, as this helps to mitigate this potential issue.

Under normal circumstances Java only loads a class once and a `Class` object is created to represent the class in the runtime environment. However, it is important to realise that the same class can potentially be loaded twice by different classloaders. As a result class in the system is identified by the classloader used to load it as well as the fully qualified classname (which includes the package name).

Executing bytecode

It is important to appreciate that Java source code goes through a significant number of transformations before execution. The first is the compilation step using the Java Compiler `javac`, often invoked as part of a larger build process.

The job of `javac` is to convert Java code into `.class` files that contain bytecode. It achieves this by doing a fairly straightforward translation of the Java source code, as shown in Figure 2-1. Very few optimizations are done during compilation by `javac` and the resulting bytecode is still quite readable and recognisable as Java code when viewed in a disassembly tool, such as the standard `javap`.

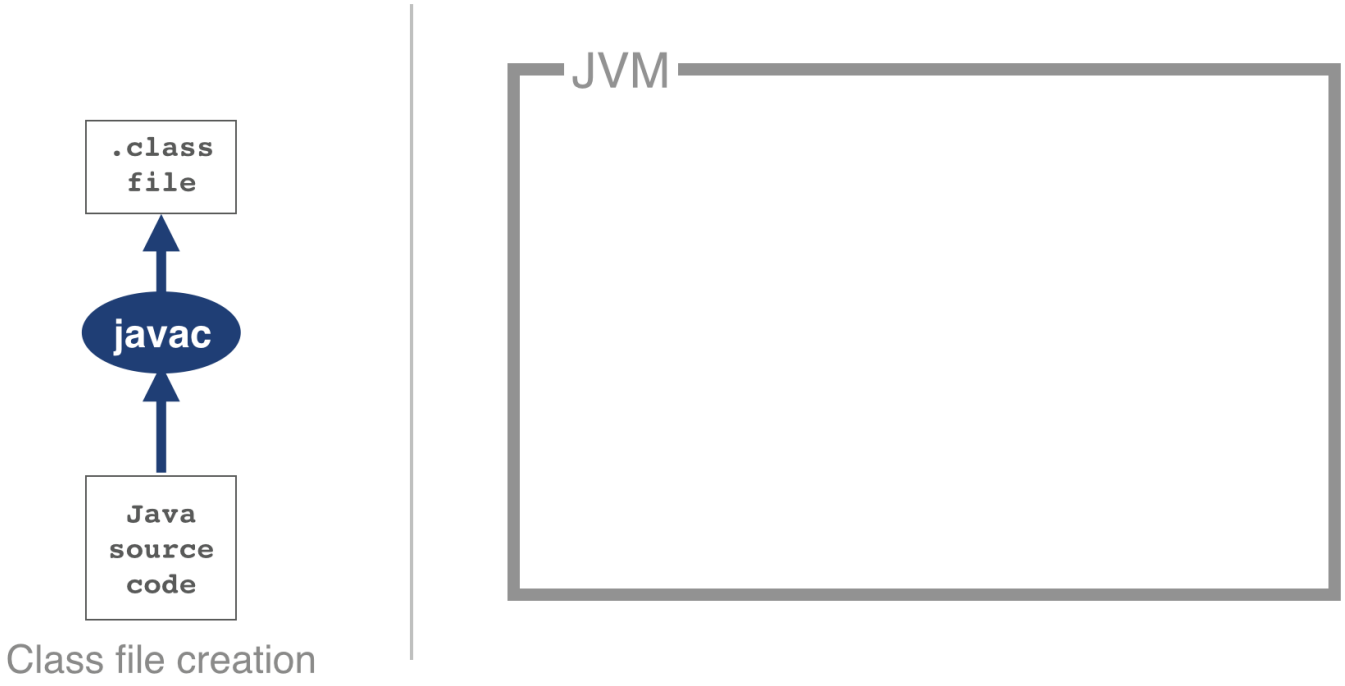


Figure 2-1. Java class file compilation

Bytecode is an intermediate representation that is not tied to a specific machine architecture. Decoupling from the machine architecture provides portability, meaning already-developed (or compiled) software can run on any platform supported by the JVM and provides an abstraction from the Java language. This provides our first important insight into the way the JVM executes code.

Note

The Java language and the Java Virtual Machine are now to a degree independent, and so the J in JVM is potentially a little misleading, as the JVM can execute any JVM language that can produce a valid class file. In fact, Figure 2-1 could just as easily show the Scala compiler `scalac` generating bytecode for execution on the JVM.

Regardless of the source code compiler used, the resulting class file has a very well defined structure specified by the VM specification. Any class that is loaded by the JVM will be verified to conform to the expected format before being allowed to run.

Table 2-1. Anatomy of a Class File

Component	Description
Magic Number	0xCAFEBABE
Version of Class File Format	The minor and major versions of the class file
Constant Pool	Pool of constants for the class
Access Flags	For example whether the class is abstract, static, etc.
This Class	The name of the current class
Super Class	The name of the super class
Interfaces	Any interfaces in the class
Fields	Any fields in the class
Methods	Any methods in the class
Attributes	Any attributes of the class (e.g. name of the sourcefile, etc.)

Every class file starts with the magic number `0xCAFEBABE`, the first 4 bytes in hexidecimal serving to denote conformance to the class file format. The following 4 bytes represent the minor and major versions used to compile the class file, these are checked to ensure that the target JVM is not higher than the version used to compile the class file. The major minor version is checked by the classloader to ensure compatibility, if these are not compatible an `UnsupportedClassVersionError` will be thrown at runtime indicating the runtime is a lower version than the compiled class file.

Note

Magic numbers provide a way for Unix environments to identify the type of a file (whereas Windows will typically use the file extension). For this reason, they are

difficult to change once decided upon. Unfortunately, this means that Java is stuck using the rather embarassing and sexist `0xCAFEBABE` for the forseeable future, although Java 9 introduces the magic number `0xCAFEDADA` for module files.

The constant pool holds constant values in code for example names of classes, interfaces and field names. When the JVM executes code the constant pool table is used to refer to values rather than having to rely on the layout of memory at runtime.

Access flags are used to determine the modifiers applied to the class. The first part of the flag identifies general properties - such as whether a class is public followed by whether it is final and cannot be subclassed. The flag also determines whether the class file represents an interface or an abstract class. The final part of the flag represents whether the class file represents a synthetic class that is not present in source code, an annotation type or an enum.

The `this` class, super class and interface entries are indexes into the constant pool to identify the type hierarchy belonging to the class. Fields and methods define a signature like structure including the modifiers that apply to the field or method. A set of attributes are then used to represent structured items for more complicated and non fixed size structures. For example, methods make use of the code attribute to represent the bytecode associated with that particular method.

Figure 2-2 provides a mnemonic for remembering the structure.

My	Very	Cute	Animal	Turns	Savage	In	Full	Moon	Areas
M	V	C	A	T	S	I	F	M	A
Magic	Version	Constant	Access	This	Super	Interfaces	Fields	Methods	Attributes

Figure 2-2. mnemonic for class file structure

Taking a very simple code example it is possible to observe the effect of running `javac`:

```
public class HelloWorld {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello World");
        }
    }}
```

Java ships with a class file disassembler called `javap`, allowing inspection of .class files. Taking the HelloWorld class file and running `javap -c HelloWorld` gives the following output:

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush        10
       5: if_icmpge     22
       8: getstatic     #2    // Field java/lang/System.out ...
      11: ldc           #3    // String Hello World
      13: invokevirtual #4    // Method java/io/PrintStream.println ...
      16: iinc          1, 1
      19: goto          2
      22: return}
```

Theis layout describes the bytecode for the file `HelloWorld.class`. For more detail `javap` also has a `-v` option that provides the full classfile header information and constant pool details. The class file contains two methods, although only the single main method was supplied in the source file - this is the result of javac automatically adding a default constructor to the class.

The first instruction executed in the constructor is `aload_0`, which places the `this` reference onto the first position in the stack. The `invokespecial` command is then called, which invokes an instance method that has specific handling for calling super constructors and the creation of objects. In the default constructor the invoke matches to the default constructor for `Object` as an override was not supplied.

Note

Opcodes in the JVM are concise and represent the type, the operation and the interaction between local variables, the constant pool and the stack.

Moving on to the main method, `iconst_0` pushes the int constant 0 onto the evaluation stack. `istore_1` stores this constant value into the local variable at offset 1 (we represented as `i` in the loop). Local variable offsets start at 0, but for instance methods, the 0th entry is always `this`. The variable at offset 1 is then loaded back onto the stack and the constant 10 is pushed for comparison using `if_icmpge` ("if integer compare greater or equal"). The test only succeeds if the current integer is >= 10.

For the first few iterations, this comparison test fails and so we continue to instruction 8. Here the static method from `System.out` is resolved, followed by the loading of the Hello World String from the constant pool. The next invoke, `invokevirtual` is encountered, which invokes an instance method based on the class. The integer is then incremented and goto is called to loop back to instruction 2.

This process continues until the `if_icmpge` comparison eventually succeeds (when the loop variable is >= 10), and on that iteration of the loop control passes to instruction 22 and the method returns.

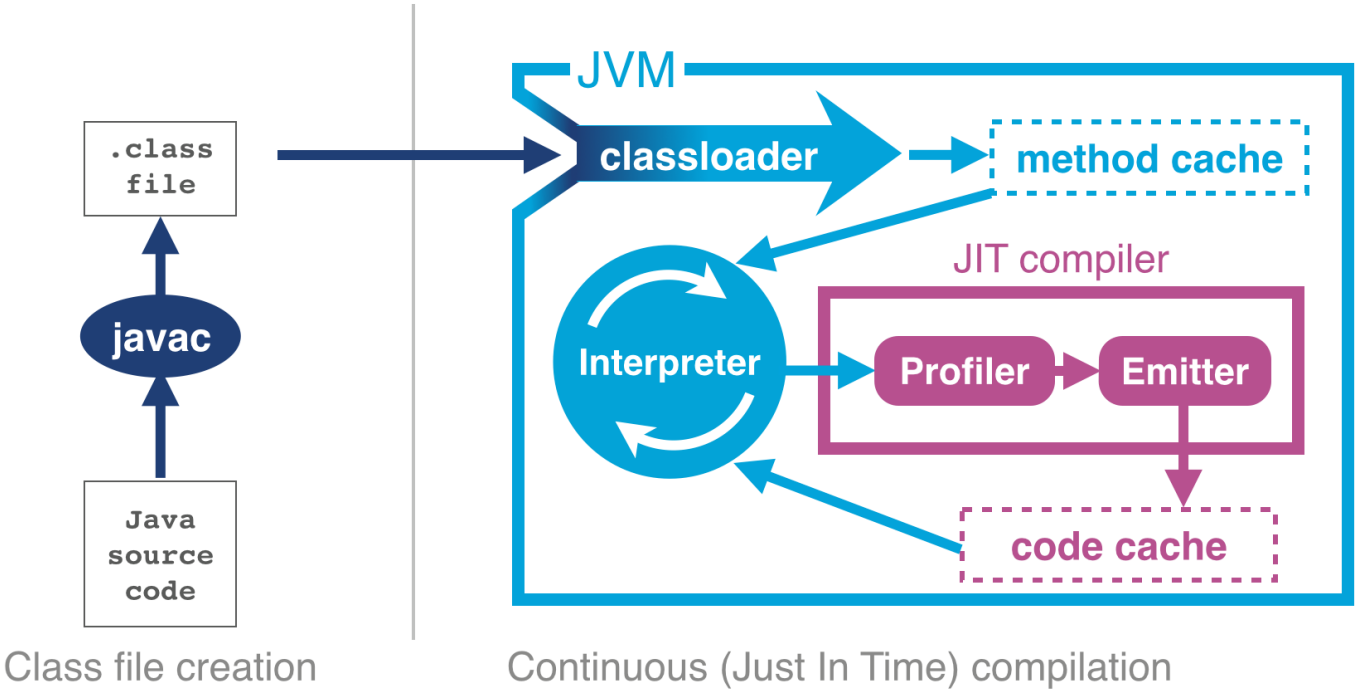


Figure 2-3. The HotSpot JVM

In April 1999 Sun introduced one of the biggest changes to Java in terms of performance. The HotSpot virtual machine is a key feature of Java that has evolved to enable performance that is comparative to (or better than) languages such as C and C++. To explain how this is possible, let's delve a little deeper into the design of languages intended for application development.

Language and platform design frequently involves making decisions and tradeoffs between desired capabilities. In this case, the division is between languages that stay "close to the metal" and rely on ideas such as "zero cost abstractions", and languages that favour developer productivity and "getting things done" over strict low-level control.

> C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further, what you do use, you couldn't hand code any better.
>
> Bjarne Stroustrup

The zero-overhead principle sounds great in theory, but it requires all users of the language to deal with the low-level reality of how operating systems and computers

actually work. This is a significant extra cognitive burden that is placed upon developers that may not care about raw performance as a primary goal.

Not only that, but it also requires the source code to be compiled to platform-specific machine code at build time - usually called Ahead of Time (AOT) compilation. This is because alternative execution models such as interpreters, virtual machines and portablity layers all are most definitely not zero overhead.

The principle also hides a can of worms in the phrase “what you do use, you couldn’t hand code any better”. This presupposes a number of things, not least that the developer is able to produce better code than an automated system. This is not a safe assumption at all. Very few people want to code in assembly language any more, so the use of automated systems (such as compilers) to produce code is clearly of some benefit to most programmers.

Java has never subscribed to the zero-overhead abstraction philosophy. Instead, the approach is that taken by the HotSpot virtual machine has been designed to analyse the runtime behaviour of your program and intelligently apply optimisations where they will benefit performance the most. The goal of the Hotspot VM is to allow you to write idiomatic Java and follow good design principles rather then contort your program to fit the VM.

Introducing Just-In-Time Compilation

Java programs begin their execution in the bytecode interpreter where instructions are performed on a virtualised stack machine. This abstraction from the CPU gives the benefit of class file portability but to get maximum performance your program must execute directly on the CPU, making use of its native features.

Hotspot achieves this by compiling units of your program from interpreted bytecode into native code. The units of compilation in the HotSpot VM are the method and the loop. This is known as *Just-In-Time* (JIT) compilation.

JIT compilation works by monitoring the application whilst it is running in interpreted mode and observing parts of code that are most frequently executed. During this analysis process programatic trace information is captured that allows for more sophisticated optimisation. Once execution of a particular method passes a threshold the profiler will look to compile and optimize that particular section of code.

There are many advantages to the JIT approach to compilation, but one of the main ones is that it allows compiler optimisation decisions to be based on trace information that is collected during the interpreted phase, enabling HotSpot to make more informed optimisations.

Not only that, but HotSpot has had hundreds of engineering years (or more) of development attributed to it and new optimisations and benefits are added with almost every new release. This means that any Java application that runs on top of a new release of Hotspot will be able to take advantage of new performance optimisations present in the VM, without even needing to be recompiled.

Note

After translation from Java source to bytecode and now another step of (JIT) compilation the code actually being executed has changed very significantly from the source code as written. This is a key insight and it will drive our approach to dealing with performance related investigations. JIT-compiled code executing on the JVM may well look nothing like the original Java source code.

The general picture is that languages like C++ (and the up-and-coming Rust) tend to have more predictable performance, but at the cost of forcing a lot of low-level complexity onto the user.

Note that "more predictable" does not necessarily mean "better". AOT compilers produce code that may have to run across a broad class of processors, and may not be able to assume that specific processor features are available.

Environments that use profile-guided optimization, such as Java, have the potential to use runtime information in ways that are simply impossible to most AOT platforms. This can offer improvements to performance, such as dynamic inlining and optimizing away virtual calls. HotSpot can even detect the precise CPU type it is running on at VM startup, and can use specific processor features if available, for even higher performance (a technique known as JVM "intrinsics"). There is a full discussion of PGO and just-in-time compilation can be found in Chapter 10.

The sophisticated approach that HotSpot takes is a great benefit to the majority of ordinary developers, but this tradeoff (to abandon zero overhead abstractions) means that in the specific case of high performance Java applications, the developer must be very careful to avoid "common sense" reasoning and overly simplistic mental models of how Java applications actually execute.

Note

Analysing the performance of small sections of Java code ("microbenchmarks") is usually actually harder than analysing entire applications, and is a very specialist task that the majority of developers should not undertake. We will return to this subject in Chapter 5.

HotSpot's compilation subsystem is one of the two most important subsystems that the virtual machine provides. The other is automatic memory management, which was originally one of the major selling points of Java in the early years.

JVM Memory Management

In languages such as C, C++ and Objective-C the programmer is responsible for managing the allocation and releasing of memory. The benefits of managing your own memory and lifetime of objects are more deterministic performance and the ability to tie resource lifetime to the creation and deletion of objects. This benefit comes at a huge cost - for correctness developers must be able to accurately account for memory.

Unfortunately, decades of practical experience showed that many developers have a poor understanding of idioms and patterns for management of memory. Later versions of C++ and Objective-C have improved this using smart pointer idioms in the standard library. However at the time Java was created poor memory management was a major cause of application errors. This led to concern among developers and managers about the amount of time spent dealing with language features rather than delivering value for the business.

Java looked to help resolve the problem by introducing automatically managed heap memory using a process known as Garbage Collection (GC). Simply put, Garbage Collection is a non-deterministic process that triggers to recover and resue no-longer-needed memory when the JVM requires more memory for allocation.

However the story behind GC is not quite so simple or glib, and various algorithms for garbage collection have been developed and applied over the course of Java's history. GC comes at a cost, when GC runs it often *stops the world*, which means whilst GC is in progress the application pauses. Usually these pause times are designed to be incredibly small, however as an application is put under pressure these pause times can increase.

Garbage collection is a major topic within Java performance optimization, and we will devote Chapter 7, 8 and 9 to the details of Java GC.

Threading and the Java Memory Model

One of the major advances that Java brought in with its first version is inbuilt support for multithreaded programming. The Java platform allows the developer to create new threads of execution. For example, in Java 8 syntax:

```
Thread t = new Thread(() -> {System.out.println("Hello World!");});
t.start();
```

Not only that, but the Java environment is inherently multithreaded, as is the JVM. This produces additional, irreducible complexity in the behaviour of Java programs, and makes the work of the performance analyst even harder.

In most mainstream JVM implementations, each Java application thread corresponds precisely to a dedicated operating system thread. The alternative, using a shared pool of threads to execute all Java application threads (an approach known as *green threads*), proved not to provide an acceptable perfromance profile and added needless complexity.

Note

It is safe to assume that every JVM application thread is backed a unique OS thread that is created when the `start()` method is called on the corresponding `Thread` object.

Java's approach to multithreading dates from the late 1990s and has these fundamental design principles:

- All threads in a Java process share a single, common garbage-collected heap
- Any object created by one thread can be accessed by any other thread that has a reference to the object
- Objects are mutable by default - the values held in object fields can be changed unless the programmer explicitly uses the `final` keyword to mark them as immutable.

The Java Memory Model (JMM) is a formal model of memory that explains how different threads of execution see the changing values held in objects. That is, if threads A and B both have references to object `obj`, and thread A alters it, what happens to the value observed in thread `B`.

This seemingly simple question is actually more complicated than it seems, because the operating system scheduler (which we will meet in Chapter 3) can forcibly evict threads from CPU cores. This can lead to another thread starting to execute and

accessing an object before the original thread had finished processing it, potentially seeing the object in a damaged or invalid state.

The only defence the core of Java provides against this potential object damage during concurrent code execution is the mutual exclusion lock, and this can be very complex to use in real applications. Chapter 15 contains a detailed look at how the JMM works, and the practicalities of working with threads and locks.

Meet the JVMs

Many developers may only be immediately familiar with the Java implementation produced by Oracle. We have already met the virtual machine that comes from the Oracle implementation, *Hotspot*. However, there are several other implementations that we will discuss in this book, to varying degrees of depth.

OpenJDK

OpenJDK is an interesting special case. It is an open-source (GPL) project that provides the reference implementation of Java. The project is led and supported by Oracle and provides the basis of their Java releases.

Oracle

Oracle's Java is the most widely-known implementation. It is based on OpenJDK, but relicensed under Oracle's proprietary license. Almost all changes to Oracle Java start off as commits to an OpenJDK public repository (with the exception of security fixes that have not yet been publicly disclosed).

Zulu

Zulu is a free (GPL-licensed) OpenJDK implementation that is fully Java-certified and provided by Azul Systems. It is unencumbered by proprietary licenses and is freely redistributable. Azul are one of the few vendors to provide paid support for OpenJDK.

IcedTea

Red Hat were the first non-Oracle vendor to produce a fully certified Java implementation based on OpenJDK. IcedTea is fully certified and redistributable.

Zing

Zing is a high-performance proprietary JVM. It is a fully certified implementation of Java and is produced by Azul Systems. It is 64-bit Linux only, and is designed for systems with large heaps and a lot of CPU.

J9

IBM's J9 started life as a proprietary JVM but was open-sourced partway through its life (just like Hotspot). It is now built on top of an Eclipse open runtime project (OMR), and forms the basis of IBM's proprietary product. It is fully compliant with Java certification.

Avian

The Avian implementation is not 100% Java conformant in terms of certification. It is included in this list as it is an interesting open-source project and a great learning tool for developers interested in understanding the details of how a JVM works, rather than as a 100% production ready solution.

Android

Google's Android project is sometimes thought of as being "based on Java". However, the picture is actually a little more complicated. Android originally used a different implementation of Java's class libraries (from the clean-room Harmony project) and a cross compiler to convert to a different (.dex) file format for a non-JVM virtual machine.

Of these implementations, the great majority of the book focuses on Hotspot. This material applies equally to Oracle Java, Azul Zulu, Red Hat IcedTea and all other OpenJDK derived JVMs.

Note

There are essentially no performance-related differences between the various Hotspot-based implementations, when comparing like-for-like versions.

We also include some material related to IBM J9 and Azul Zing. This is intended to provide an awareness of these alternatives rather than a definitive guide. Some readers may wish to explore these technologies more deeply, and they are encouraged to proceed by setting performance goals, and then measuring and comparing, in the usual manner.

Android is moving to use the OpenJDK 8 class libraries with direct support in the Android Runtime. As this technology stack is so far from the other examples, we won't consider Android any further in this book.

A note on licenses

Almost all of the JVMs we will discuss are open-source, and in fact, most of them are derived from the GPL-licensed Hotspot. The exceptions are IBM's J9, which is Eclipse-licensed and Azul Zing, which is commercial (although their Zulu product is GPL).

The situation with Oracle Java is slightly more complex. Despite being derived from OpenJDK, it is proprietary, and is *not* open-source software. Oracle achieve this by having all contributors to OpenJDK sign a license agreement that permits dual-licensing of their contribution to both the GPL of OpenJDK and Oracle's proprietary license.

Each update release to Oracle Java is taken as a branch off OpenJDK mainline, which is not then patched on-branch for future releases. This prevents divergence of Oracle and OpenJDK, and accounts for the lack of meaningful difference between Oracle JDK and an OpenJDK binary based off the same source.

This means that the only real difference between Oracle JDK and OpenJDK is the license. This may seem an irrelevance, but the detail of the Oracle license contains several clauses that developers should be aware of:

- Oracle do not grant the right to redistribute their binaries outside of your own organisation (e.g. as a Docker image)
-
-
- You are not permitted to apply a binary patch to an Oracle binary without their agreement (which will usually mean a support contract)
-

As we will see in Chapter 6, there are also several other commercial features and tools that Oracle makes available which will only work with Oracle's JDK, and within the terms of their license.

When planning a new deployment, developers and architects should consider carefully their choice of JVM vendor. Some large organisations, notably Twitter and Alibaba, even maintain their own private builds of OpenJDK, although the engineering effort required for this is beyond the reach of many companies.

In Chapter 3 we will discuss some details of how operating systems and hardware work. This is to provide necessary background for the Java peformance analyst to understand observed results. We will also look at the timing subsystem in more detail, as a complete example of how the VM and native subsystems interact.

Chapter 3. Hardware & Operating Systems

Why should Java developers care about Hardware?

For many years the computer industry has been driven by Moore's Law, a hypothesis made by Intel founder, Gordon Moore, about long-term trends in processor capability. The law (really an observation or extrapolation) can be framed in a variety of ways, but one of the most usual is:

> The number of transistors on a mass-produced chip roughly doubles every 18 months
>
> Gordon Moore

This phenomenon represents an exponential increase in computer power over time. It was originally cited in 1965, so represents an incredible long-term trend, almost unparalleled in the history of human development. The effects of Moore's Law have been transformative in many (if not most) areas of the modern world.

Note

The death of Moore's Law has been repeatedly proclaimed for decades now. However, there are very good reasons to suppose that, for all practical purposes, this incredible progress in chip technology has (finally) come to an end.

However, hardware has become increasingly complex in order to make good use of the "transistor budget" available in modern computers. The software platforms that run on that hardware has also increased in complexity to exploit the new capabilities, so whilst software has far more power at its disposal it has come to rely on complex underpinnings to access that performance increase.

The net result of this huge increase in the performance available to the ordinary application developer has been the blossoming of complex software. Software applications now pervade every aspect of global society. Or, to put it another way:

> Software is eating the world.
>
> Marc Andreessen

As we will see, Java has been a beneficiary of the increasing amount of computer power. The design of the language and runtime has been well-suited (or lucky) to make use of this trend in processor capability. However, the truly performance-conscious

Java programmer needs to understand the principles and technology that underpins the platform in order to make best use of the available resources.

In later chapters, we will explore the software architecture of modern JVMs and techniques for optimizing Java applications at the platform and code levels. Before turning to those subjects, let's take a quick look at modern hardware and operating systems, as an understanding of those subjects will help with everything that follows.

Introduction to Modern Hardware

Many universty courses on hardware architectures still teach a simple-to-understand, "classical" view of hardware. This "motherhood and apple pie" view of hardware focuses on a simple view of a register-based machine, with arithmetic and logic operations, load and store operations.

Since then, however, the world of the application developer has, to a large extent, revolved around the Intel x86 / x64 architecture. This is an area of technology that has undergone radical change and many advanced features now form important parts of the landscape. The simple mental model of a processor's operation is now completely incorrect, and intuitive reasoning based on it is extremely liable to lead to utterly wrong conclusions.

To help address this, in this chapter, we will discuss several of these advances in CPU technology. We will start with the behavior of memory, as this is by far the most important to a modern Java developer.

Memory

As Moore's Law advanced, the exponentially increasing number of transistors was initially used for faster and faster clock speed. The reasons for this are obvious - faster clock speed means more instructions completed per second. Accordingly, the speed of processors has advanced hugely, and the 2+ GHz processors that we have today are hundreds of times faster than the original 4.77 MHz chips found in the first IBM PC.

However, the increasing clock speeds uncovered another problem. Faster chips require a faster stream of data to act upon. As Figure 3-1 [1] shows, over time main memory could not keep up with the demands of the processor core for fresh data.

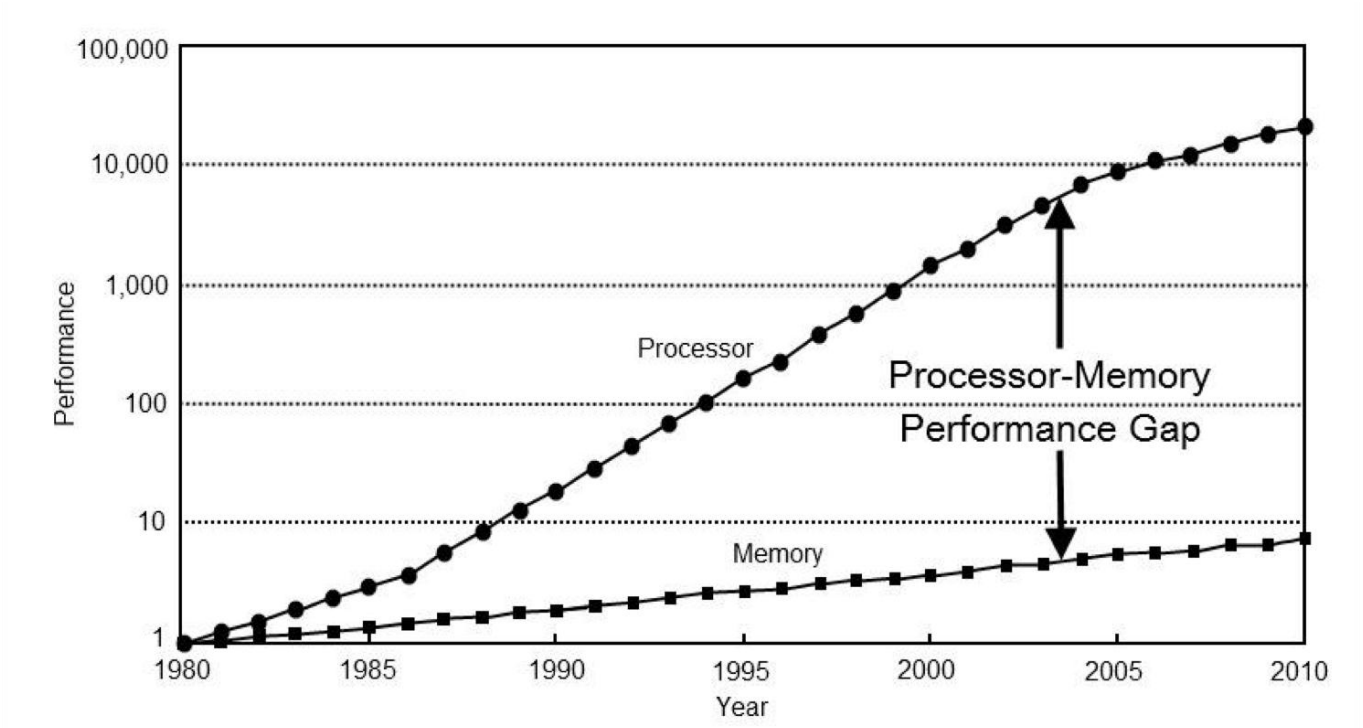


Figure 3-1. Speed of memory and transistor counts

This results in a problem - if the CPU is waiting for data, then faster cycles don't help, as the CPU will just have to idle until the required data arrives.

Memory Caches

To solve this problem, CPU caches were introduced. These are memory areas on the CPU that are slower than CPU registers, but faster than main memory. The idea is for the CPU to fill the cache with copies of often-accessed memory locations rather than constantly having to re-address main memory.

Modern CPUs have several layers of cache, with the most-often-accessed caches being located close to the processing core. The cache closest to the CPU is usually called L1 (for "level 1 cache"), with the next being referred to as L2, and so on. Different processor architectures have a varying number, and configuration, of caches, but a common choice is for each execution core to have a dedicated, private L1 and L2 cache, and an L3 chache that is shared across some or all of the cores. The effect of these caches in speeding up access times is shown in Figure 3-2 [2].

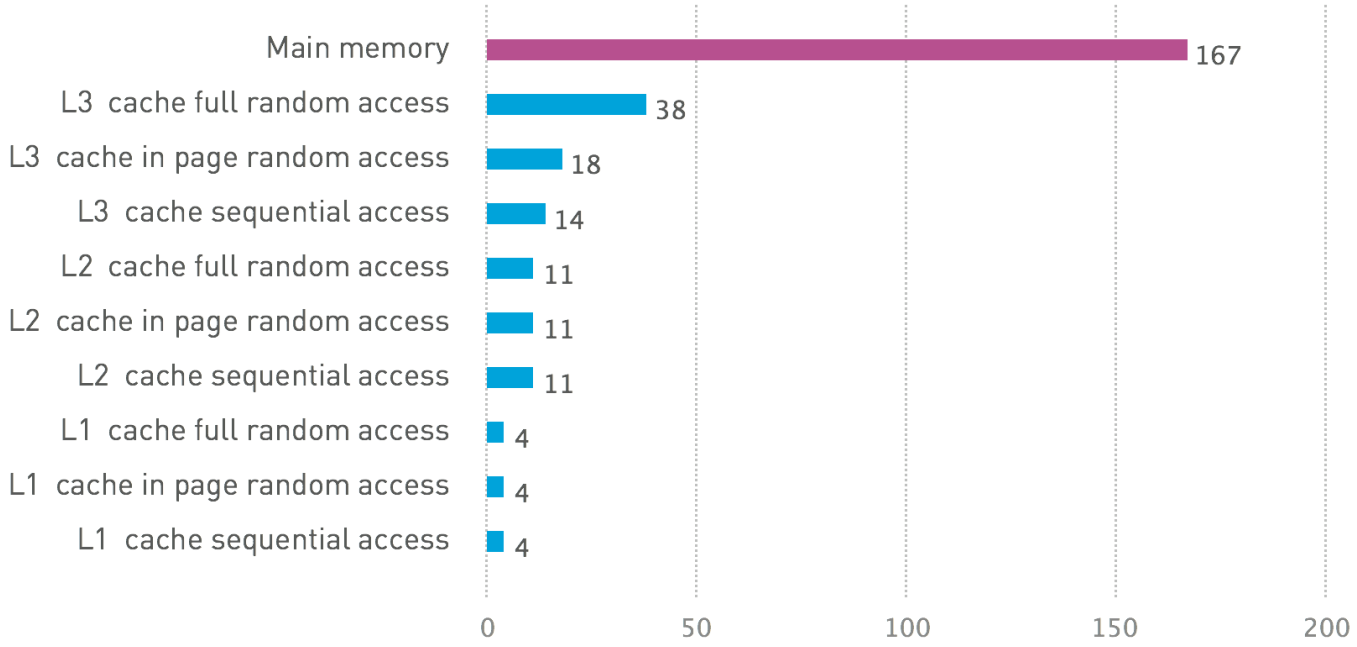


Figure 3-2. Access times for various types of memory

This approach to cache architecture improves access times and helps keep the core fully stocked with data to operate on. However, it introduces a new set of problems such as determining how memory is fetched into and written back from cache. The solutions to this problem are usually referred to as “cache consistency protocols”.

Note

There are other problems that crop up when this type of caching is applied in a parallel processing environment, as we will see later in this chapter.

At the lowest level, a protocol called MESI (and its variants) is commonly found on a wide range of processors. It defines four states for any line in a cache. Each line (usually 64 bytes) is either:

-

Modified (but not yet flushed to main memory)

-
-

Exclusive (only present in this cache, but does match main memory)

-

-

Shared (may also be present in other caches, matches main memory)

-
-

Invalid (may not be used, will be dropped as soon as practical)

-

Table 3-1. MESI allowable states between processors

	M	E	S	I
M	–	–	–	Y
E	–	–	–	Y
S	–	–	Y	Y
I	Y	Y	Y	Y

The idea of the protocol is that multiple processors can simultenously be in the Shared state. However, if a processor transitions to any of the other valid states (Exclusive or Modified), then this will force all the other processors into the Invalid state. This is shown in Table 3-1

The protocol works by broadcasting the intentions of a processor that is intending to change state. An electrical signal is sent across the shared memory bus, and the other processors are made aware. The full logic for the state transitions is show in Figure 3-3

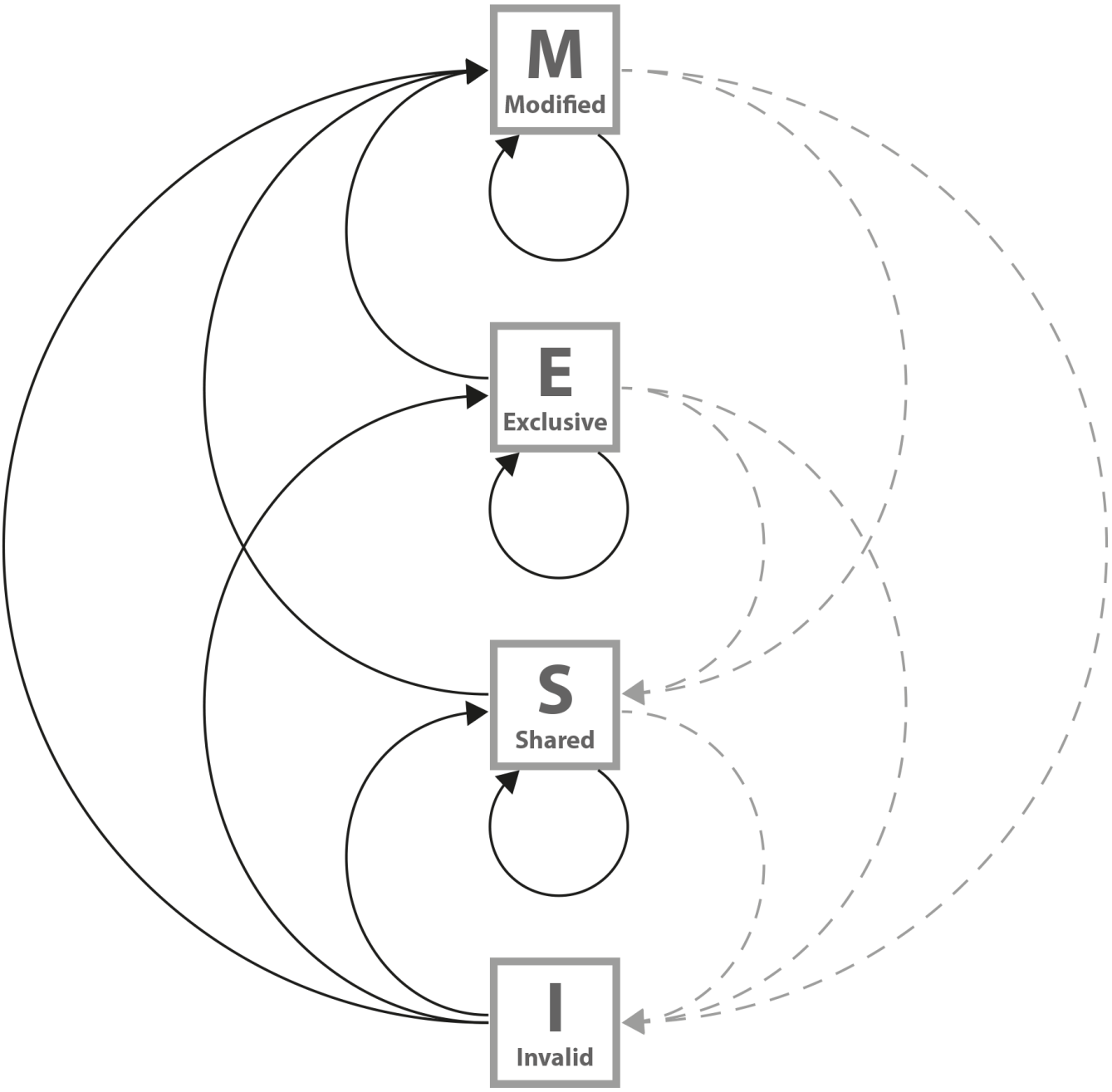


Figure 3-3. MESI state transition diagram

Originally, processors wrote every cache operation directly into main memory. This was called “write-through” behavior, but it was and is very inefficient, and required a large amount of bandwidth to memory. More recent processors also implement “write-back” behavior, where traffic back to main memory is significantly reduced by processors only writing modified (dirty) cache blocks to memory when the cache blocks are replaced.

The overall effect of caching technology is to greatly increase the speed at which data can be written to, or read from, memory. This is expressed in terms of the

bandwidth to memory. The “burst rate”, or theoretical maximum is based on several factors:

- Clock frequency of memory
- The width of the memory bus (usually 64 bits)
- Number of interfaces (usually 2 in modern machines)

This is multiplied by 2 in the case of DDR RAM (DDR stands for “double data rate” as it communicates on both edges of a clock signal). Applying the formula to 2015 commodity hardware gives a theoretical maximum write speed of 8-12GB/s. In practice, of course, this could be limited by many other factors in the system. As it stands, this gives a modestly useful value to allow us to see how close the hardware and software can get.

Let’s write some simple code to exercise the cache hardware:

```
public class Caching {

    private final int ARR_SIZE = 2 * 1024 * 1024;
    private final int[] testData = new int[ARR_SIZE];

    private void run() {
        System.err.println("Start: "+ System.currentTimeMillis());
        for (int i = 0; i < 15_000; i++) {
            touchEveryLine();
            touchEveryItem();
        }
        System.err.println("Warmup finished: "+ System.currentTimeMillis());
        System.err.println("Item    Line");
        for (int i = 0; i < 100; i++) {
            long t0 = System.nanoTime();
            touchEveryLine();
            long t1 = System.nanoTime();
```

```
            touchEveryItem();
            long t2 = System.nanoTime();
            long elEvery = t1 - t0;
            long elLine = t2 - t1;
            double diff = elEvery - elLine;
            System.err.println(elEvery + " " + elLine +" "+  (100 * diff / elLine));
        }
    }

    private void touchEveryItem() {
        for (int i = 0; i < testData.length; i++)
            testData[i]++;
    }

    private void touchEveryLine() {
        for (int i = 0; i < testData.length; i += 16)
            testData[i]++;
    }

    public static void main(String[] args) {
        Caching c = new Caching();
        c.run();
    }}
```

Intuitively, `touchEveryItem()` "does 16 times as much work" as `touchEveryLine()`, as 16 times as many data items must be updated. However, the point of this simple example is to show how badly intuition can lead us astray when dealing with JVM performance. Let's look at some sample output from the Caching class, as shown in Figure 3-4:

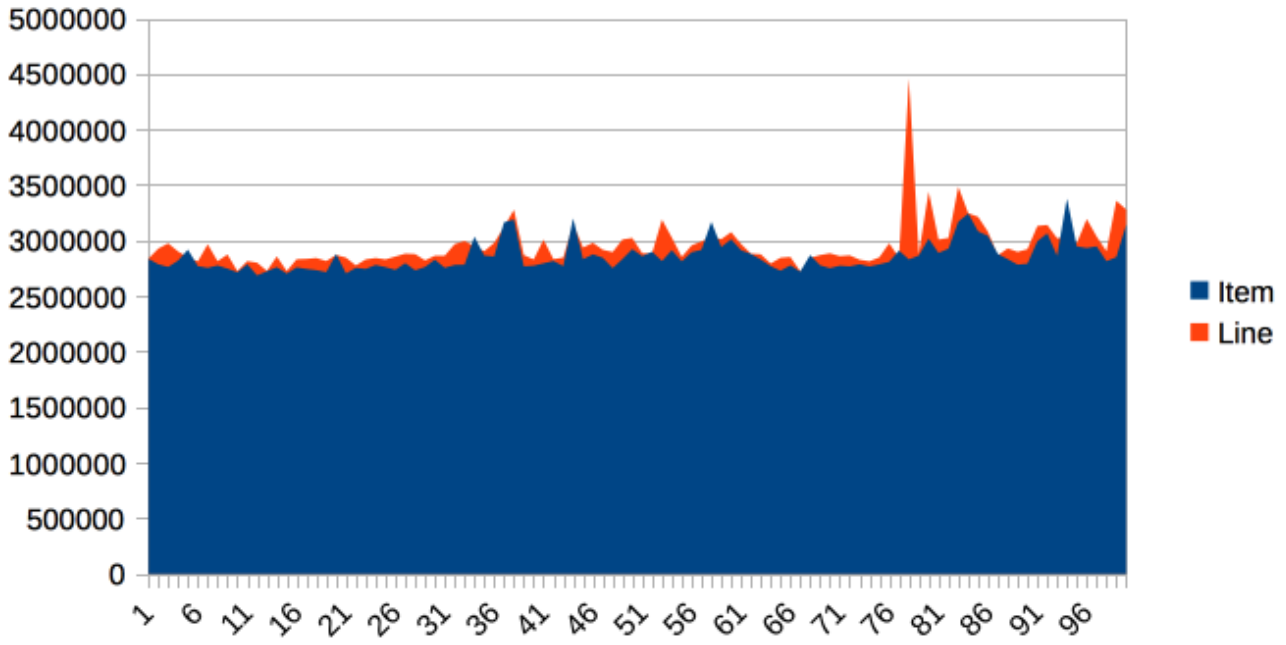


Figure 3-4. Time elapsed for Caching example

The graph shows 100 runs of each function, and is intended to show several different effects. Firstly, notice that the results for both functions are remarkable similar to each other in terms of time taken, so the intuitive expectation of "16 times as much work" is clearly false.

Instead, the dominant effect of this code is to exercise the memory bus, by transferring the contents of the array from main memory, into the cache to be operated on by `touchEveryItem()` and `touchEveryLine()`.

In terms of the statistics of the numbers, although the results are reasonably consistent, there are individual outliers that are 30-35% different from the median value.

Overall, we can see that each iteration of the simple memory function takes around 3 milliseconds (2.86ms on average) to traverse a 100M chunk of memory, giving an effective memory bandwidth of just under 3.5GB per second. This is less than the theoretical maximum, but is still a reasonable number.

Note

Modern CPUs have a hardware prefetcher, that can detect predictable patterns in data access (usually just a regular “stride” through the data). In this example, we’re taking advantage of that fact in order to get closer to a realistic maximum for memory access bandwidth.

One of the key themes in Java performance is the sensitivity of applications to object allocation rates. We will return to this point several times, but this simple example gives us a basic yardstick for how high allocation rates could rise.

Modern Processor Features

Hardware engineers sometimes refer to the new features that have become possible as a result of Moore’s Law as “spending the transistor budget”. Memory caches are the most obvious use of the growing number of transistors, but other techniques have also appeared over the years.

Translation Lookaside Buffer

One very important use is in a different sort of cache - the Translation Lookaside Buffer (TLB). This acts as a cache for the page tables that map virtual memory addresses to physical addresses. This greatly speeds up a very frequent operation - access to the physical address underlying a virtual address.

Note

There’s a memory-related software feature of the JVM that also has the acronym TLB (as we’ll see later). Always check which features is being discussed when you see TLB mentioned.

Without the TLB cache, all virtual address lookups would take 16 cycles, even if the page table was held in the L1 cache. Performance would be unacceptable, so the TLB is basically essential for all modern chips.

Branch Prediction and Speculative Execution

One of the advanced processor tricks that appears on modern processors is branch prediction. This is used to prevent the processor having to wait to evaluate a value needed for a conditional branch. Modern processors have multistage instruction

pipelines. This means that the execution of a single CPU cycle is broken down into a number of separate stages. There can be several instructions in-flight (at different stages of execution) at once.

In this model, a conditional branch is problematic, because until the condition is evaluated, it isn't known what the next instruction after the branch will be. This can cause the processor to stall for a number of cycles (in practice, up to 20), as it effectively empties the multi-stage pipeline behind the branch.

To avoid this, the processor can dedicate transistors to building up a heuristic to decide which branch is more likely to be taken. Using this guess, the CPU fills the pipeline based on a gamble - if it works, then the CPU carries on as though nothing had happened. If it's wrong, then the partially executed instructions are dumped, and the CPU has to pay the penalty of emptying the pipeline.

Hardware Memory Models

The core question about memory that must be answered in a multicore system is "How can multiple different CPUs access the same memory location safely?".

The answer to this question is highly hardware dependent, but in general, `javac`, the JIT compiler and the CPU are all allowed to make changes to the order in which code executes, provided that it doesn't affect the outcome as observed by the current thread.

For example, let's suppose we have a piece of code like this:

```
myInt = otherInt;intChanged = true;
```

There is no code between the two assignments, so the executing thread doesn't need to care about what order they happen in, and so the environment is at liberty to change the order of instructions.

However, this could mean that in another thread that has visibility of these data items, the order could change, and the value of `myInt` read by the other thread could be the old value, despite `intChanged` being seen to be `true`.

This type of reordering (store moved after store) is not possible on x86 chips, but as Table 3-2 shows, there are other architectures where it can, and does, happen.

Table 3-2. Hardware memory support

	ARMv7	POWER	SPARC	x86	AMD64	zSeries
Loads moved after loads	Y	Y	-	-	-	-

Table 3-2. Hardware memory support

	ARMv7	POWER	SPARC	x86	AMD64	zSeries
Loads moved after stores	Y	Y	-	-	-	-
Stores moved after stores	Y	Y	-	-	-	-
Stores moved after loads	Y	Y	Y	Y	Y	Y
Atomic moved with loads	Y	Y	-	-	-	-
Atomic moved with stores	Y	Y	-	-	-	-
Incoherent instructions	Y	Y	Y	Y	-	Y

In the Java environment, the Java memory model (JMM) is explicitly designed to be a weak model to take into account the differences in consistency of memory access between processor types. Correct use of locks and volatile access is a major part of ensuring that multithreaded code works properly. This is a very important topic that we will return to later in the book, in Chapter 14.

There has been a trend in recent years for software developers to seek greater understanding of the workings of hardware in order to derive better performance. The term “Mechanical Sympathy” has been coined by Martin Thompson and others to describe this approach, especially as applied to the low-latency and high-performance spaces. It can be seen in recent research into lock-free algorithms and data structures, which we will meet towards the end of the book.

Operating systems

The point of an operating system is to control access to resources that must be shared between multiple executing processes. All resources are finite, and all processes are greedy, so the need for a central system to arbitrate and meter access is essential. Among these scarce resources, the two most important are usually memory and CPU time.

Virtual addressing via the memory management unit and its page tables are the key feature that enables access control of memory, and prevents one process from damaging the memory areas owned by another.

The TLBs that we met earlier in the chapter are a hardware feature that improve lookup times to physical memory. The use of the buffers improves performance for software’s access time to memory. However, the MMU is usually too low-level for developers to directly influence or be aware of. Instead, let’s take a closer look at the OS process scheduler, as this controls access to the CPU and is a far more user-visible piece of the operating system kernel.

The Scheduler

Access to the CPU is controlled by the process scheduler. This uses a queue, known as the "run queue" as a waiting area for threads or processes that are eligible to run but which must wait their turn for the CPU. On a modern system there are effectively always more threads / processes that want to run than can, and so this CPU contention requires a mechanism to resolve it.

The job of the scheduler is to respond to interrupts, and to manage access to the CPU cores. The lifecycle of a Java thread is shown in Figure 3-5. In theory, the Java specification permits threading models whereby Java threads do not necessarily correspond to operating system threads. However, in practice, such "green threads" approaches have not proved to be useful, and have been abandoned in mainstream operating environments.

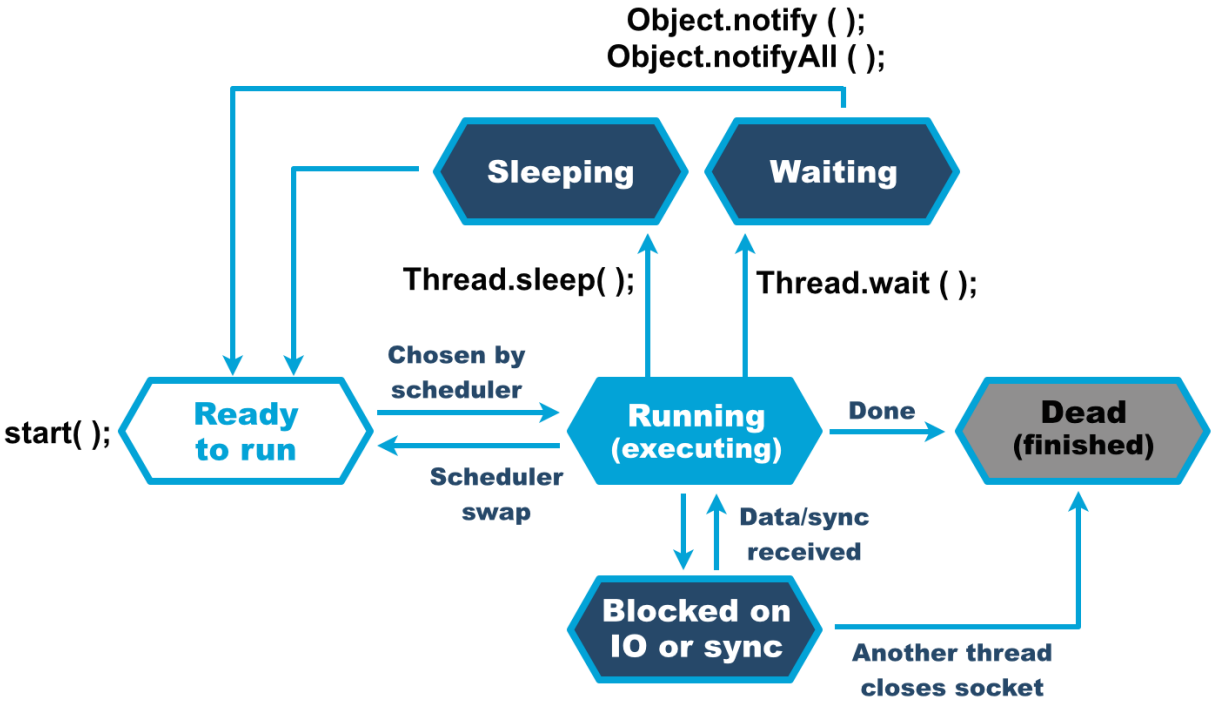


Figure 3-5. Thread Lifecycle

In this relatively simple view, the OS scheduler moves threads on and off the single core in the system. At the end of the time quantum (often 10ms or 100ms in older operating systems), the scheduler moves the thread to the back of the run queue to wait until it reaches the front of the queue and is eligible to run again.

If a thread wants to voluntarily give up its time quantum it can do so either for a fixed amount of time (via `sleep()`), or until a condition is met (using `wait()`). Finally, a thread can also block on I/O or a software lock.

When meeting this model for the first time, it may help to think about a machine that has only a single execution core. Real hardware is, of course, more complex and virtually any modern machine will have multiple cores, and this allows for true simultaneous execution of multiple execution paths. This means that reasoning about execution in a true multiprocessing environment is very complex and counter-intuitive.

An often-overlooked feature of operating systems is that by their nature, they introduce periods of time when code is not running on the CPU. A process that has completed its time quantum will not get back on the CPU until it comes to the front of the run queue again. This combines with the fact that CPU is a scarce resource to give us the fact that "code is waiting more often than it is running".

This means that the statistics we want to generate from processes that we actually want to observe are affected by the behavior of other processes on the systems. This "jitter" and the overhead of scheduling is a primary cause of noise in observed results. We will discuss the statistical properties and handling of real results in Chapter 5.

One of the easiest ways to see the action and behavior of a scheduler is to try to observe the overhead imposed by the OS to achieve scheduling. The following code executes 1000 separate 1 ms sleeps. Each of these sleeps will involve the thread being sent to the back of the run queue, and having to wait for a new time quantum. So, the total elapsed time of the code gives us some idea of the overhead of scheduling for a typical process.

```
long start = System.currentTimeMillis();
for (int i = 0; i < 1_000; i++) {
    Thread.sleep(1);
}
long end = System.currentTimeMillis();
System.out.println("Millis elapsed: " + (end - start));
```

Running this code will cause wildly divergent results, depending on operating system. Most Unixes will report roughly 10-20% overhead. Earlier versions of Windows had notoriously bad schedulers - with some versions of Windows XP reporting up to 180% overhead for scheduling (so that a 1000 sleeps of 1 ms would take 2.8s). There are even reports that some proprietary OS vendors have inserted code into their releases in order to detect benchmarking runs and cheat the metrics.

Timing is of critical importance to performance measurements, to process scheduling and to many other parts of the application stack, so let's take a quick look at how timing is handled by the Java platform (and a deeper dive into how it is supported by the JVM and the underlying OS).

A Question of Time

Despite the existence of industry standards such as POSIX, different operating systems can have very different behaviors. For example, consider the `os::javaTimeMillis()` function. In OpenJDK this contains the OS-specific calls that actually do the work and ultimately supply the value to be eventually returned by Java's `System.currentTimeMillis()` method.

As we discussed in Section 2.6, as this relies on functionality that has to be provided by the host operating system, this has to be implemented as a native method. Here is the function as implemented on BSD Unix (e.g. for Apple's OS X operating system):

```
jlong os::javaTimeMillis() {
  timeval time;
  int status = gettimeofday(&time, NULL);
  assert(status != -1, "bsd error");
  return jlong(time.tv_sec) * 1000  +  jlong(time.tv_usec / 1000);}
```

The versions for Solaris, Linux and even AIX are all incredibly similar to the BSD case, but the code for Microsoft Windows is completely different:

```
jlong os::javaTimeMillis() {
  if (UseFakeTimers) {
    return fake_time++;
  } else {
    FILETIME wt;
    GetSystemTimeAsFileTime(&wt);
    return windows_to_java_time(wt);
  }}
```

Windows uses a 64-bit FILETIME type to store the time in units of 100ns elapsed since the start of 1601, rather than the Unix `timeval` structure. Windows also has a notion of the "real accuracy" of the system clock, depending on which physical timing hardware is available. So the behavior of timing calls from Java can be highly variable on different Windows machines.

The differences between the operating systems do not end with just timing, as we shall see in the next section.

Context Switches

Context switches can be a very costly operation, whether between user threads or from user mode into kernel mode. The latter case is particularly important, because a user thread may need to swap into kernel mode in order to perform some function partway through its time slice. However, this switch will force instruction and other caches to be emptied, as the memory areas accessed by the user space code will not normally have anything in common with the kernel.

A context switch into kernel mode will invalidate the TLBs and potentially other caches. When the call returns, these cashes will have to be refilled and so the effect of a kernel mode switch persists even after control has returned to user space. This causes the true cost of a system call to be masked, as can be seen in Figure 3-6 [3].

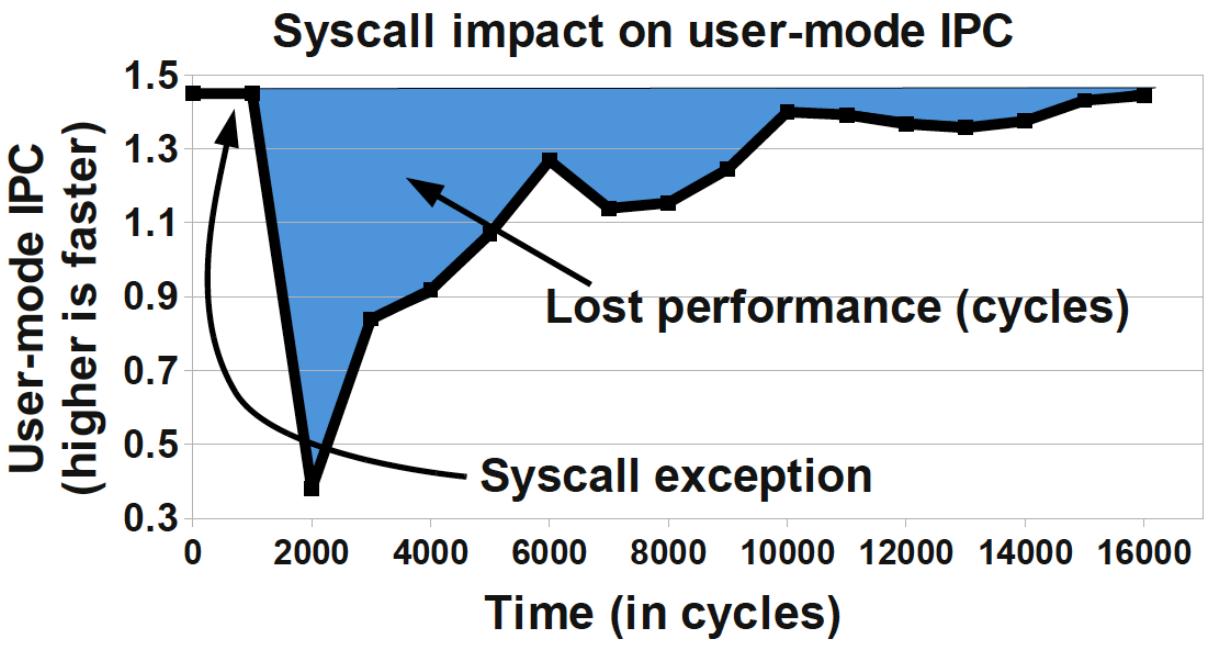


Figure 3-6. Impact of a system call

To mitigate this when possible, Linux provides a mechanism known as vDSO (Virtual Dynamically Shared Objects). This is a memory area in user space that is used to speed up syscalls that do not really require kernel privileges. It achieves this speed increase by not actually performing a context switch into kernel mode. Let's look at an example to see how this works with a real syscall.

A very common Unix system call is `gettimeofday()`. This returns the "wallclock time" as understood by the operating system. Behind the scenes, it is actually just reading a kernel data structure to obtain the system clock time. As this is side-effect free, it does not actually need privileged access.

If we can use vDSO to arrange for this data structure to be mapped into the address space of the user process, then there's no need to perform the switch to kernel mode, and the refill penalty shown in Figure 3-6 does not have to be paid.

Given how often most Java applications need to access timing data, this is a welcome performance boost. The vDSO mechanism generalises this example slightly and can be a useful technique, even if it is only available on Linux.

A simple system model

In this section we describe a simple model for describing basic sources of possible performance problems. The model is expressed in terms of operating system observables of fundamnetal subsystems and can be directly related back to the outputs of standard Unix command line tools.

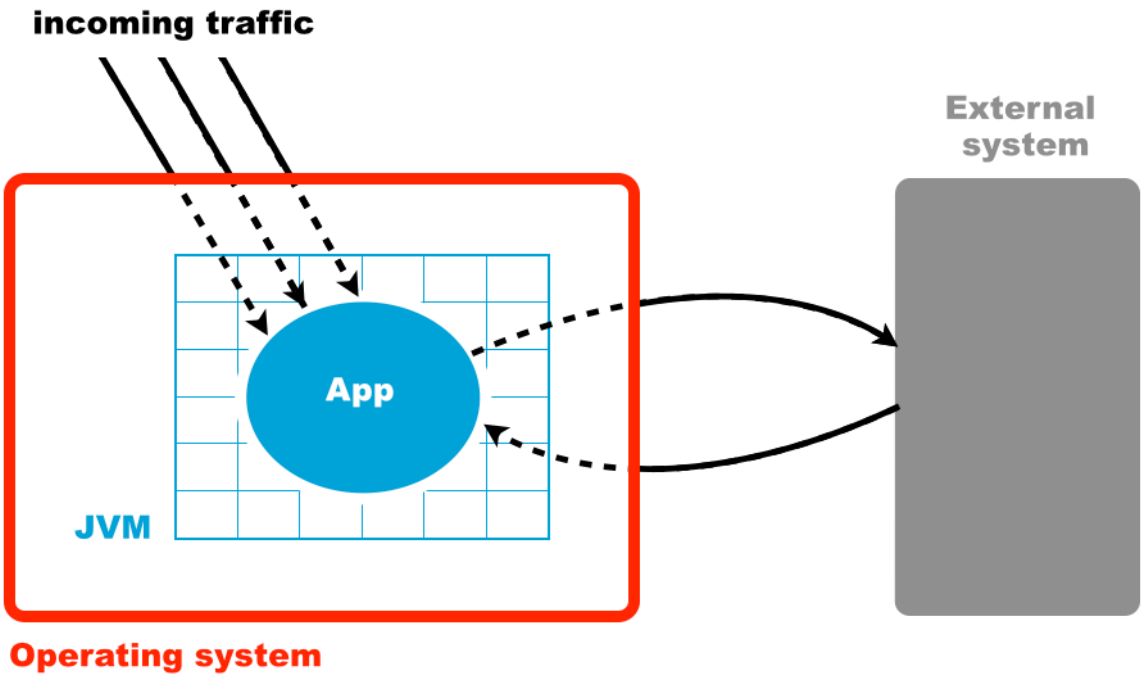


Figure 3-7. Simple system model

The model is based around a simple conception of a Java application running on a Unix or Unix-like operating system. Figure 3-7 shows the basic components of the model, which consist of:

-

The hardware and operating system the application runs on

-
-

The JVM (or container) the application runs in

-
-

The application code itself

-
-

Any external systems the application calls

-
-

The incoming request traffic that is hitting the application

-

Any of these aspects of a system can be responsible for a performance problem. There are some simple diagnostic techniques that can be used to narrow down or isolate particular parts of the system as potential culprits for performance problems, as we will see in the next section.

Basic Detection Strategies

One definition for a well-performing application is that efficient use is being made of system resources. This includes CPU usage, memory and network or I/O bandwidth. If an application is causing one or more resource limits to be hit, then the result will be a performance problem.

It is also worth noting that the operating system itself should not normally be a major contributing factor to system utilisation. The role of an operating system is to manage resources on behalf of user processes, not to consume them itself. The only real exception to this rule is when resources are so scarce that the OS is having difficulty allocating anywhere near enough to satisfy user requirements. For most modern server-class hardware, the only time this should occur is when I/O (or occassionally memory) requirements greatly exceed capability.

A key metric for application performance is CPU utilisation. CPU cycles are quite often the most critical resource needed by an application, and so efficient use of them is essential for good performance. Applications should be aiming for as close to 100% usage as possible during periods of high load.

Note

When analysing application performance, the system must be under enough load to exercise it. The behavior of an idle application is usually meaningless for performance work.

Two basic tools that every performance engineer should be aware of are `vmstat` and `iostat`. On Linux and other Unixes, these command-line tools provide immediate and often very useful insight into the current state of the virtual memory and I/O subsystems, respectively. The tools only provide numbers at the level of the entire host, but this is frequently enough to point the way to more detailed diagnostic approaches. Let's take a look at how to use `vmstat` as an example:

```
ben@janus:~$ vmstat 1
 r  b swpd  free    buff  cache   si   so  bi  bo   in   cs us sy  id wa st
 2  0    0 759860 248412 2572248    0    0   0  80   63  127  8  0  92  0  0
 2  0    0 759002 248412 2572248    0    0   0   0   55  103 12  0  88  0  0
 1  0    0 758854 248412 2572248    0    0   0  80   57  116  5  1  94  0  0
 3  0    0 758604 248412 2572248    0    0   0  14   65  142 10  0  90  0  0
 2  0    0 758932 248412 2572248    0    0   0  96   52  100  8  0  92  0  0
 2  0    0 759860 248412 2572248    0    0   0   0   60  112  3  0  97  0  0
```

The parameter 1 following vmstat indicates that we want vmstat to provide ongoing output (until interrupted via Ctrl-C) rather than a single snapshot. New output lines are printed, every second, which enables a performance engineer to leave this output running (or capturing it into a log) whilst an initial performance test is performed.

The output of vmstat is relatively easy to understand, and contains a large amount of useful information, divided into sections.

1.

The first two columns show the number of runnable and blocked processes.

2.

3.

In the memory section, the amount of swapped and free memory is shown, followed by the memory used as buffer and as cache.

4.

5.

The swap section shows the memory swapped from and to disk. Modern server class machines should not normally experience very much swap activity.

6.

7.

The block in and block out counts (bi and bo) show the number of 512-byte blocks that have been received from, and sent to a block (I/O) device.

8.

9.

In the system section, the number of interrupts and the number of context switches per second are displayed.

10.

11.

The CPU section contains a number of directly relevant metrics, expressed as percentages of CPU time. In order, they are user time (us), kernel time (sy, for “system time”), idle time (id), waiting time (wa) and the “stolen time” (st, for virtual machines).

12.

Over the course of the remainder of this book, we will meet many other, more sophisticated tools. However, it is important not to neglect the basic tools at our disposal. Complex tools often have behaviors that can mislead us, whereas the simple tools that operate close to processes and the operating system can convey simple and uncluttered views of how our systems are actually behaving.

In the rest of this section, let's consider some common scenarios and how even very simple tools such as `vmstat` can help us spot issues.

Context switching

In Section 3.4.3, we discussed the impact of a context switch, and saw the potential impact of a full context switch to kernel space in Figure 3-6. However, whether between user threads or into kernel space, context switches introduce unavoidable wastage of CPU resources.

A well-tuned program should be making maximum possible use of its resources, especially CPU. For workloads which are primarily dependent on computation ("CPU-bound" problems), the aim is to achieve close to 100% utilisation of CPU for userland work.

To put it another way, if we observe that the CPU utilisation is not approaching 100% user time, then the next obvious question is to ask why not? What is causing the program to fail to achieve that? Are involuntary context switches caused by locks the problem? Is it due to blocking caused by I/O contention?

The `vmstat` tool can, on most operating systems (especially Linux), show the number of context switches occurring, so on a `vmstat 1` run, the analyst will be able to see the real-time effect of context switching. A process that is failing to achieve 100% userland CPU usage and is also displaying high context-switch rate is likely to be either blocked on I/O or thread lock contention.

However, `vmstat` is not enough to fully disambiguate these cases on its own. I/O problems can be seen from `vmstat`, as it provides a crude view of I/O operations as well. However, to detect thread lock contention in real time, tools like VisualVM that can show the states of threads in a running process should be used. One additional common tool is the statistical thread profiler that samples stacks to provide a view of blocking code.

Garbage Collection

As we will see in Chapter 7, in the HotSpot JVM (by far the most commonly used JVM), memory is allocated at startup and managed from within user space. That means, that system calls (such as `sbrk()`) are not needed to allocate memory. In turn, this means that kernel switching activity for garbage collection is quite minimal.

Thus, if a system is exhibiting high levels of system CPU usage, then it is definitely not spending a significant amount of its time in GC, as GC activity burns user space CPU cycles and does not impact kernel space utilization.

On the other hand, if a JVM process is using 100% (or close to) of CPU in user space, then garbage collection is often the culprit. When analysing a performance problem, if simple tools (such as vmstat) show consistent 100% CPU usage, but with almost all cycles being consumed by userspace, then a key question that should be asked next is: "Is it the JVM or user code that is responsible for this utilization?". In almost all cases, high userspace utilization by the JVM is caused by the GC subsystem, so a useful rule of thumb is to check the GC log and see how often new entries are being added to it.

Garbage collection logging in the JVM is incredibly cheap, to the point that even the most accurate measurements of the overall cost cannot reliably distinguish it from random background noise. GC logging is also incredibly useful as a source of data for analytics. It is therefore imperative that GC logs be enabled for all JVM processes, especially in production.

We will have a great deal to say about GC and the resulting logs, later in the book. However, at this point, we would encourage the reader to consult with their operations staff and confirm whether GC logging is on in production. If not, then one of the key points of Chapter 8 is to build a strategy to enable this.

I/O

File I/O has traditionally been one of the murkier aspects of overall system performance. Partly this comes from its closer relationship with messy physical hardware, with engineers making quips about "spinning rust" and similar, but it is also because I/O lacks as clean abstractions as we see elsewhere in operating systems.

In the case of memory, the elegance of virtual memory as a separation mechanism works well. However, I/O has no comparable abstraction that provides suitable isolation for the application developer.

Fortunately, whilst most Java programs involve some simple I/O, the class of applications that make heavy use of the I/O subsystems is relatively small, and in particular, most applications do not simultenously try to saturate I/O at the same time as either CPU or memory.

Not only that, but established operational practice has led to a culture in which production engineers are already aware of the limitations of I/O, and actively monitor processes for heavy I/O usage.

For the performance analyst / engineer, it suffices to have an awareness of the I/O behavior of our applications. Tools such as `iostat` (and even `vmstat`) have the basic counters (e.g. blocks in or out) that are often all we need for basic diagnosis,

especially uf we make the assumption that only one I/O-intensive application is present per host.

Finally, it's worth mentioning one aspect of I/O that is becoming more widely used across a class of Java applications that have a dependency on I/O but also stringent performance applications.

Kernel Bypass I/O

For some high-performance applications, the cost of using the kernel to copy data from, for example, a buffer on a network card, and place it into a user space region is prohibitively high. Instead, specialised hardware and software is used to map data directly from a network card into a user-accessible area. This approach avoids a "double-copy" as well as crossing the boundary between user space and kernel, as we can see in Figure 3-8.

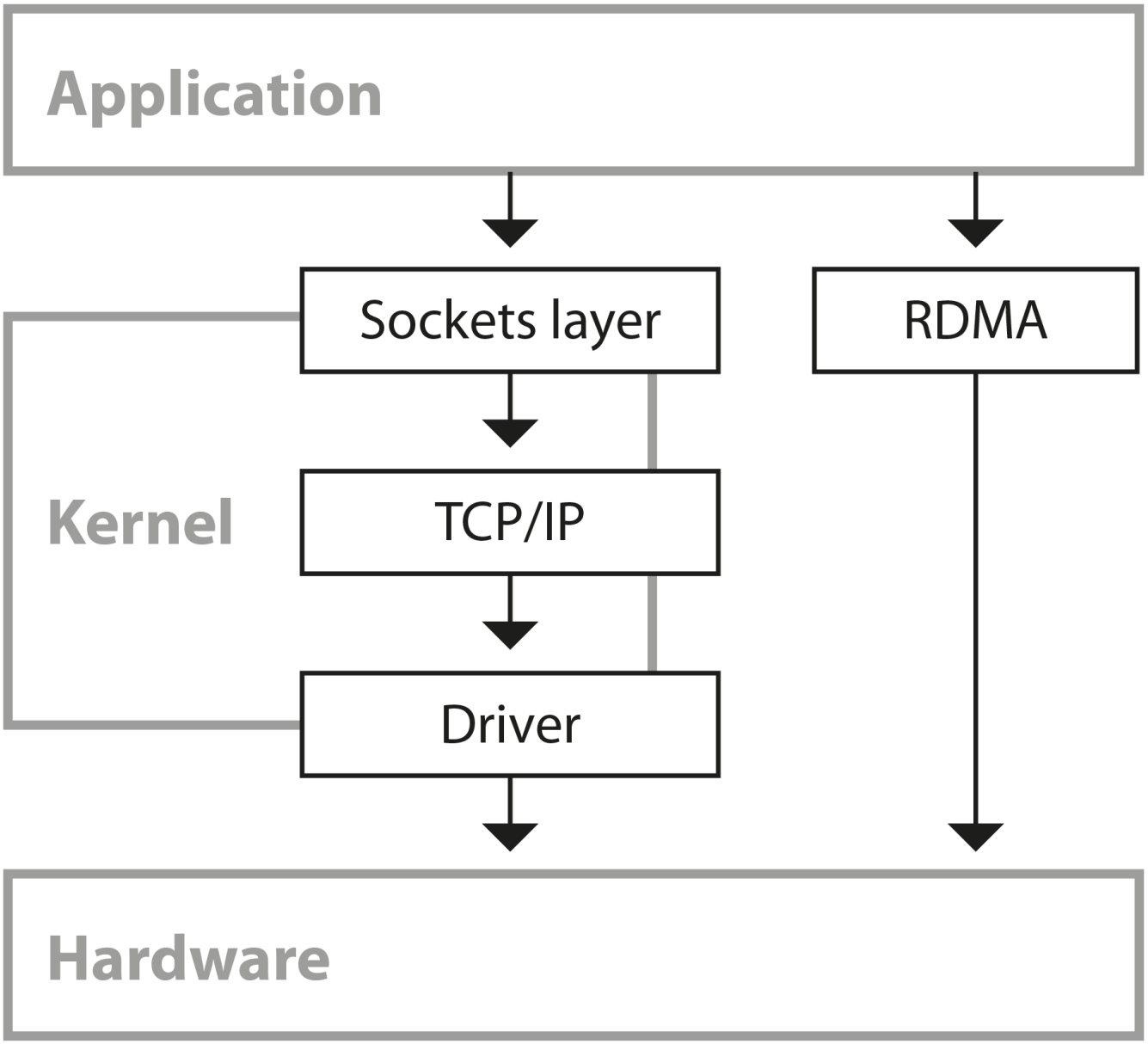


Figure 3-8. Kernel Bypass I/O

However, Java does not provide specific support for this model, and instead applications that wish to make use of it rely upon custom (native) libraries to implement the required semantics. It can be a very useful pattern and is increasingly commonly implemented in systems that require very high-performance I/O.

Note

In some ways, this is reminiscent of Java's New I/O (NIO) API that was introduced to allow Java I/O to bypass the Java heap and work directly with native memory and underlying I/O.

In this chapter so far we have discussed operating systems running on top of “bare metal”. However, increasingly, systems run in virtualised environments, so to conclude this chapter, let’s take a brief look at how virtualisation can fundamentally change our view of Java application performance.

Mechanical Sympathy

“What is the best way to measure the impact of a context switch in a live system? I’d like to say the number of threads is the problem and here is the hard evidence to prove it.”

Virtualisation

Virtualisation comes in many forms, but one of the most common is to run a copy of an operating system as a single process on top of an already-running OS. This leads to a situation represented in Figure 3-9 where the virtual environment runs as a process inside the unvirtualized (or “real”) operating system that is executing on bare metal.

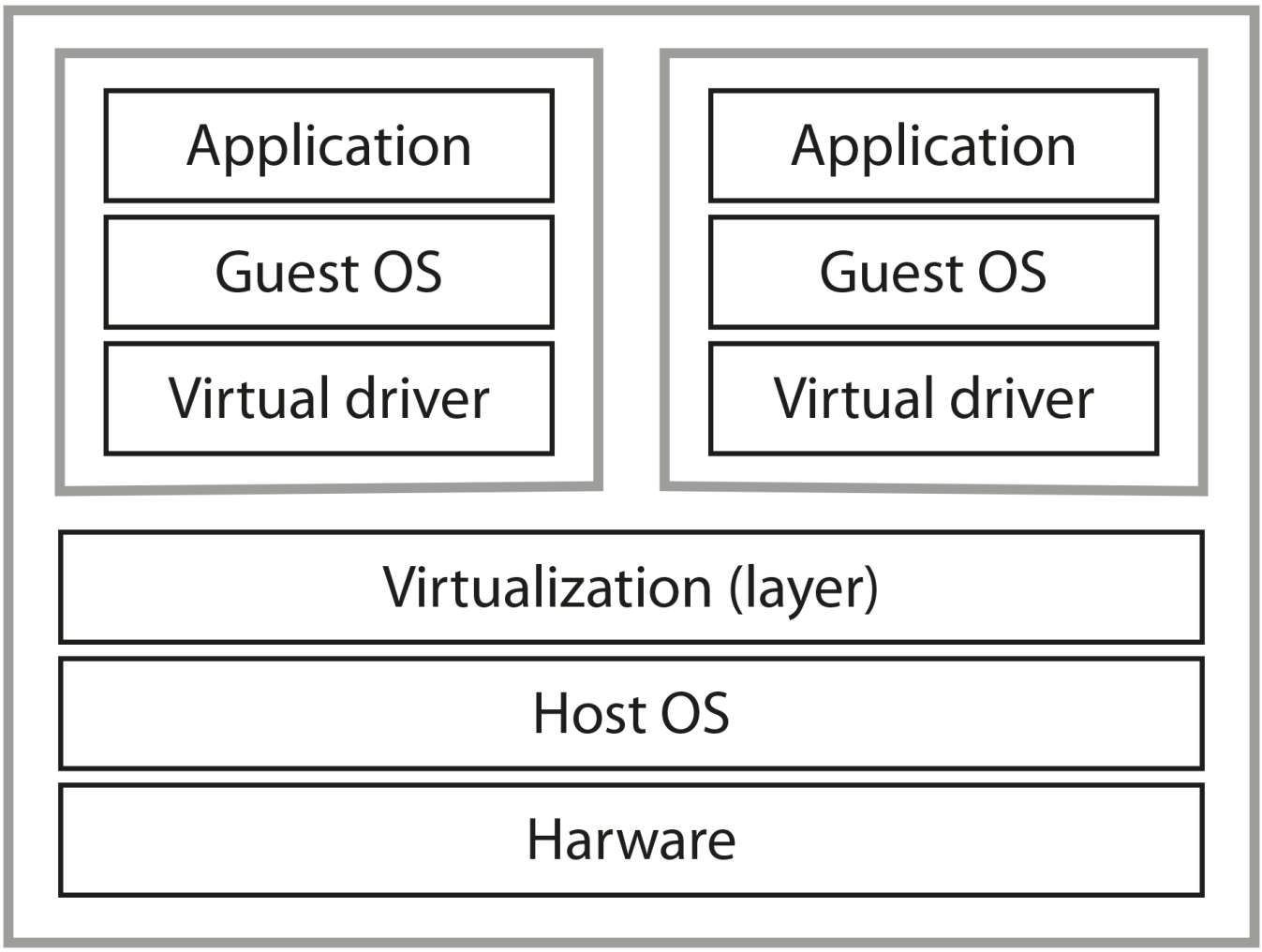


Figure 3-9. Virtualisation of operating systems

A full discussion of virtualisation, the relevant theory and its implications for application performance tuning would take us too far afield. However, some mention of the differences that virtualisation causes seems approriate, especially given the increasing amount of applications running in virtual, or cloud environments.

Although virtualisation was originally developed in IBM mainframe environments as early as the 1970s, it was not until recently that x86 architectures were capable of supporting “true” virtualisation. This is usually characterized by these three conditions:

- Programs running on a virtualized OS should behave essentially the same as when running on “bare metal” (i.e. unvirtualized)
-

-

The hypervisor must mediate all accesses to hardware resources

-
-

The overhead of the virtualization must be as small as possible, and not a significant fraction of execution time.

-

In a normal, unvirtualized system, the OS kernel runs in a special, privileged mode (hence the need to switch into kernel mode). This gives the OS direct access to hardware. However, in a virtualized system, direct access to hardware by a guest OS is disallowed.

One common approach is to rewrite the privileged instructions in terms of unprivileged instructions. In addition, some of the OS kernel's data structures need to be "shadowed" to prevent excessive cache flushing (e.g. of TLBs) during context switches.

Some modern Intel-compatible CPUs have hardware features designed to improve the performance of virtualized OSs. However, it is apparent that even with hardware assists, running inside a virtual environment presents an additional level of complexity for performance analysis and tuning.

The JVM and the operating system

Java and the JVM provide a portable execution environent that is independent of the operating system. This is implemented by providing a common interface to Java code. However, for some fundamental services, such as thread scheduling (or even something as mundane as getting the time from the system clock), the underlying operating system must be accessed.

This capability is provided by native methods, which are denoted by the keyword `native`. They are written in C, but are accessible as ordinary Java methods. This interface is referred to as the Java Native Interface (JNI). For example, `java.lang.Object` declares these non-private native methods:

```
public final native Class<?> getClass();
public native int hashCode();
protected native Object clone() throws CloneNotSupportedException;
public final native void notify();
```

```
public final native void notifyAll();
public final native void wait(long timeout) throws InterruptedException;
```

As all these methods deal with relatively low-level platform concerns, let's look at a more straightforward and familiar example - getting the system time.

Consider the `os::javaTimeMillis()` function. This is the (system-specific) code responsible for implementing the Java `System.currentTimeMillis()` static method. The code that does the actual work is implemented in C++, but is accessed from Java via a "bridge" of C code. let's look at how this code is actually called in HotSpot:

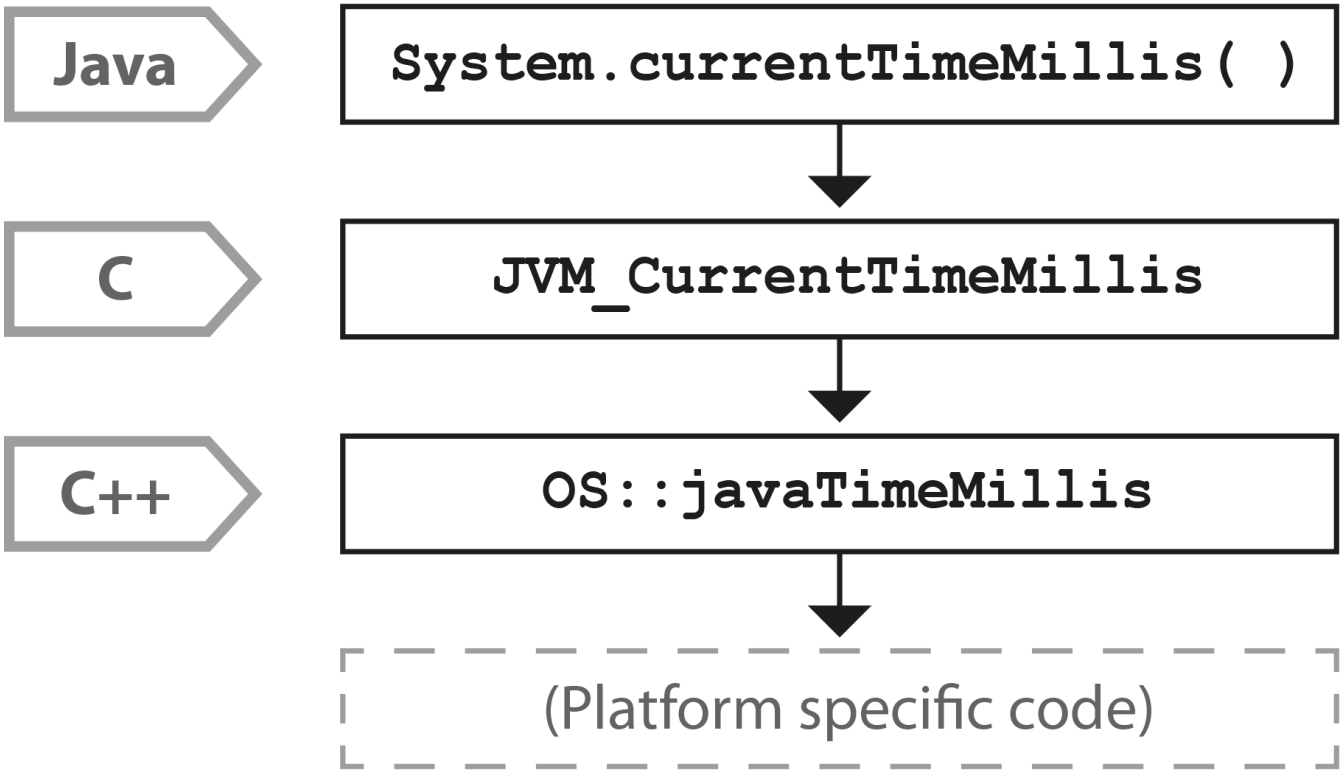


Figure 3-10. The Hotspot Calling stack

As you can see in Figure 2-4, the native `System.currentTimeMillis()` method is mapped to the JVM entry point method `JVM_CurrentTimeMillis`. This mapping is achieved via the JNI `Java_java_lang_System_registerNatives` mechanism contained in `java/lang/System.c`.

`JVM_CurrentTimeMillis` is essentially a call to the C++ method `os::javaTimeMillis()` wrapped in a couple of OpenJDK macros. This method is defined in the `os` namespace, and is unsurprisingly operating system-dependent. Definitions for this method are provided by the OS-specific subdirectories of source code within OpenJDK. This provides a simple demonstration of how the platform-independent parts of Java can call into services that are provided by the underlying operating system and hardware.

In the next chapter we will introduce the core methodology of performance tests. We will discuss the primary types of performance tests, the tasks that need to be undertaken and the overall lifecycle of performance work. We will also catalogue some common best practices (and antipatterns) in the performance space.

[1] From Computer Architecture: A Quantitative Approach by Hennessy, et al.

[2] Access times shown in terms of number of clock cycles per operation, provided by Google

[3] Image reproduced from “FlexSC: Flexible System Call Scheduling with Exception-Less System Calls” by Soares & Stumm

Chapter 4. Performance Testing Patterns and Antipatterns

Performance testing is undertaken for a variety of different reasons. In this chapter we will introduce the different types of test that a team may wish to execute, and discuss best practices for each type.

In the second half of the chapter, we will outline some of the more common antipatterns that can plague a performance test or team, and explain refactored solutions to help prevent them becoming a problem for teams.

Types of Performance Test

Performance tests are frequently conducted for the wrong reasons, or conducted badly. The reasons for this vary widely, but are frequently caused by a failure to understand the nature of performance analysis, and a belief that "something is better than nothing". As we will see repeatedly, this belief is often a dangerous half-truth at best.

One of the more common mistakes is to speak generally of "performance testing" without engaging with the specifics. In fact, there are a fairly large number of different types of large-scale performance tests that can be performed on a system.

Note

Good performance tests are quantitative. They ask questions that have a numeric answer that can be handled as an experimental output, and subjected to statistical analysis.

The types of performance tests we will discuss in this book usually have mostly independent (but somewhat overlapping) goals, so care should be taken when thinking about the domain of any given single test. A good rule-of-thumb when planning a performance test is simply to write down (and confirm to management / the customer) the quantitative questions that the test is intended to answer, and why they are important for the application under test.

Some of the most common test types, and an example question for each are given below.

- Latency Test - what is the end-to-end transaction time?
-

-

Throughput Test - how many concurrent transactions can the current system capacity deal with?

-
-

Load Test - can the system handle a specific load?

-
-

Stress Test - what is the breaking point of the system?

-
-

Endurance Test - what performance anomalies are discovered when the system is run for an extended period?

-
-

Capacity Planning Test - Does the system scale as expected when additional resources are added?

-
-

Degradation - What happens when the system is partially failed?

-

Let’s look at each of these test types in turn, in more detail.

Latency Test

This is one of the most common types of performance test, usually because it can be closely related to a system observable that is of direct interest to management - how long are our customers waiting for a transaction (or a page load). This is a double-edged sword, as because the quantitative question that a latency test seeks to answer seems so obvious, that it can obscure the necessity of identifying quantitative questions for other types of performance tests.

Note

The goal of a latency tuning exercise is usually to directly improve user experience, or to meet a service level agreement.

However, even in the simplest of cases, a latency test has some subtlties that must be treated carefully. One of the most noticable of these is that (as we will discuss fully in Section 5.3) a simple mean (average) is not very useful as a measure of how well an application is reacting to requests.

Throughput Test

Throughput is probably the second most common quantity to be the subject of a performance exercise. It can even be thought of as dual to latency, in some senses.

For example, when conducting a latency test, it is important to state (and control) the concurrent transactions ongoing when producing a distribution of latency results.

Note

The observed latency of a system should be stated at known and controlled throughput levels.

Equally, a throughput test is usually conducted whilst monitoring latency. The "maximum throughput" is determined by noticing when the latency distribution suddenly changes - effectively a "breaking point" (also called an inflection point) of the system. The point of a stress test, as we will see, is to locate such points and the load levels at which they occur.

A throughput test, on the other hand, is about measuring the observed maximum throughput before the system starts to degrade.

Load Test

A load test differs from a throughput test (or a stress test) in that it is usually framed as a binary test - "Can the system handle this projected load or not?". Load tests are sometimes conducted in advanced of expected business events, e.g. the onboarding of a new customer or market that is expected to drive greatly increased traffic to the application. Other examples of possible events that could warrant performing this tpye of test include advertising campaigns, social media events and "viral content".

Stress Test

One way to think about a stress test is as a way to determine how much spare headroom the system has. The test typically proceeds by placing the system into a steady state of transactions - that is a specified throughput level (often current peak). The test then ramps up the concurrent transactions slowly, until the system observables start to degrade.

The maximum throughput achieved in a throughput test is then determined as being the value just before the observables started to degrade.

Endurance Test

Some problems only manifest over much longer periods (often measured in days). These include slow memory leaks, cache pollution and memory fragmentation (especially for applications that use the Concurrent Mark and Sweep garbage collector, which may eventually suffer "Concurrent Mode Failure" - see Section 8.3 for more details).

To detect these types of issue, an Endurance Test (also known as a Soak Test) is the usual approach. These are run at average (or high) utilisation, but within observed loads for the system. During the test resource levels are closely monitored and breakdowns or exhaustions of resources are looked for.

This type of test is very common in fast response (or low-latency) systems, as it is very common that they will not be able to tolerate the length of a stop-the-world event caused by a full GC cycle (see Chapter 7 and subsequent chapters for more on stop-the-world and related GC concepts).

Capacity Planning Test

Capacity planning tests bear many similarities to stress tests, but they are a distinct type of test. The role of a stress test is to find out what the current system will cope with, whereas a capacity planning test is more forward-looking and seeks to find out what load an upgraded system could handle.

For this reason, they are often carried out as part of a scheduled planning exercise, rather in response to a specific event or threat.

Degradation Test

This type of test is also known as a Partial Failure Test. The general discussion of resilience and fail-over testing is outside the scope of this book, but suffice it to say that in the most highly regulated and scrutinized enivronments (including banks and financial institutions), fail-over and recovery testing is taken extremely seriously and is usually planned in meticulous depth.

For our purposes, the only type of resilience test we consider is the degradation test. The basic approach to this test is to see how the system behaves when a component or entire subsystem suddenly loses capacity whilst the system is running at simulated loads equivalent to usual production volumes. Examples could be application server clusters that suddenly lose members, databases that suddenly lose RAID disks or network bandwidth that suddenly drops.

Key observables during a degradation test include the transaction latency distribution and throughput.

One particularly interesting subtype of Partial Failure Test is known as the Chaos Monkey. This is named after a project at Netflix that was undertaken to verify the robustness of their infrastructure.

The idea behind Chaos Monkey is that in a truly resilient architecture, the failure of a single component should not have the ability to cause a cascading failure or to cause meaningful impact on the overall system.

Chaos Monkey attempts to demonstrate this by randomly killing off live processes that are actually in use in the production environment.

In order to successfully implement Chaos Monkey type systems, an organisation must have the highest levels of system hygiene, service design and operational excellence. Nevertheless, it is an area of interest and aspiration for an increasing number of companies and teams.

Best Practices Primer

When deciding where to focus your effort in a performance tuning exercise, there are three golden rules that can provide useful guidance:

-

Identify what you care about and figure out how to measure it.

-
-

Optimize what matters, not what is easy to optimize.

-
-

Play the big points first.

-

The second point has a converse, which is to remind yourself not to fall into the trap of attaching too much significance to whatever quantity you can easily measure. Not every observable is significant to a business, but it is sometimes tempting to report on an easy measure, rather than the right measure.

Top-Down Performance

One of the aspects of Java performance that many engineers miss at first encounter is that large-scale benchmarking of Java applications is usually actually easier than trying to get accurate numbers for small sections of code. We will discuss this in detail in Chapter 5.

Note

The approach of starting with the performance behavior of an entire application is usually called “Top-Down Performance”.

To make the most of the top-down approach, a testing team needs a test environment, a clear understanding of what they need to measure and optimize, and an understanding of how the performance exercise will fit into the overall software development lifecycle.

Creating a test environment

Setting up a test environment is one of the first tasks most performance testing teams will need to undertake. Wherever possible, this should be an exact duplicate of the production environment - in all aspects (the servers should have the same number of CPUs, same version of the OS and Java runtime, etc). This includes not only application servers, but web servers, databases, load balancers, network firewalls, etc. Any services, e.g. 3rd party network services that are not easy to replicate, or do not have sufficient QA capacity to handle a production-equivalent load, will need to be mocked for a representative performance testing environment.

Sometimes teams try to reuse or time-share an existing QA environment for performance testing. This can be possible for smaller environments or for one-off testing but the management overhead and scheduling and logistical problems that it can cause should not be underestimated.

Note

Performance testing environments that are significantly different from the production environments that they attempt to represent often fail to achieve results that have any usefulness or predictive power in the live environment.

For traditional (i.e. non cloud-based) environments, a production-like performance testing environment is relatively straightforward to achieve - the team simply buys as many physical machines as are in use in the production environment and then configures them in exactly the same way as production is configured.

As we will see in Section 4.3.7, however, management is sometimes resistent to the additional infrastructure cost that this represents. This is almost always a false economy, but sadly many organisations fail to account correctly for the cost of outages. This can lead to a belief that the savings made by not having an accurate performance testing environment are meaningful, as it fails to properly account for the risks introduced by having a QA environment that does not mirror production.

Recent developments, notably the advent of cloud technologies, have changed this rather traditional picture. On-demand and autoscaling infrastructure means that an increasing number of modern architectures no longer fit the model of “buy servers, draw network diagram, deploy software on hardware”. The devops approach of treating server infrastructure as “livestock, not pets” means that much more dynamic approaches to infrastructure management are becoming widespread.

This makes the construction of a performance testing environment that looks like production potentially more challenging. However, it raises the possibility of setting up a performance environment that can be turned off when not actually being used. This can be a very significant cost saving to the project, but it requires a proper process for starting up and shutting down the environment as scheduled.

Identifying performance requirements

Let’s recall the simple system model that we met in Section 3.5. This clearly shows that the overall performance of a system is not solely determined by your application code. The container, operating system and hardware all have a role to play.

Therefore, the metrics that we will use to evaluate performance should not be thought about solely in terms of the code. Instead, we must consider systems as a whole and the observable quantities that are important to customers and management. These are usually referred to as performance non-functional requirements (NFRs), and are the key indicators that we want to optimize.

Some goals are obvious:

- Reduce 95% percentile transaction time by 100ms
- Improve system so that 5x throughput on existing hardware is possible
- Improve average response time by 30%

Others may be less apparent:

- Reduce resources cost to serve the average customer by 50%
- Ensure system is still within 25% of response targets, even when application clusters are degraded by 50%
- Reduce customer “drop-off” rate by 25% per 25ms of latency

An open discussion with the stakeholders as to exactly what should be measured and what goals are to be achieved is essential. Ideally, this discussion should form part of the first kick-off meeting for the performance exercise.

Java-specific issues

Much of the science of performance analysis is applicable to any modern software system. However, the nature of the JVM is such that there are certain additional complications that the performance engineer should be aware of and consider carefully. These largely stem from the dynamic self-management capabilities of the JVM, such as the dynamic tuning of memory areas.

One particularly important Java-specific insight is related to JIT compilation. Modern JVMs analyse which methods are being run to identify candidates for Just-In-Time (JIT) compilation to optimized machine code. This means that if a method is not being JIT-compiled, then one of two things is true about the method:

1.

It is not being run frequently enough to warrant being compiled.

2.

3.

The method is too large or complex to be analysed for compilation.

4.

The second option is much rarer than the first. However, one early performance exercise for JVM-based applications is to switch on simple logging of which methods are being compiled and ensure that the important methods for the application's key code paths are being compiled.

In Chapter 10 we will discuss JIT compilation in detail, and show some simple techniques for ensuring that the important methods of applications are targeted for JIT compilation by the JVM.

Performance testing as part of the SDLC

Some companies and teams prefer to think of performance testing as an occasional, one-off activity. However, more sophisticated teams tend to build ongoing performance tests, and in particular performance regression testing as an integral part of their Software Development LifeCycle (SDLC).

This requires collaboration between developers and infrastructure teams for controlling which versions of code are present in the performance environment at any given time. It is also virtually impossible to implement without a dedicated performance testing environment.

Having discussed some of the most common best practices for performance, it is also important to discuss the pitfalls and antipatterns that teams can fall prey to.

Introducing Performance Antipatterns

An antipattern is an undesired behavior of a software project or team, that is observed across a large number of projects. [1] The commonality of occurence leads to the conclusion (or suspicion) that some underlying factor is responsible for creating the unwanted behavior. Some antipatterns may at first sight seem to be justified, with their non-ideal aspects not immediately obvious. Others are the result of negative project practices slowly accreting over time.

In some cases, the behavior may be driven by social or team constraints, by common misapplied management techniques or by simple human (and developer) nature. By classifying and categorising these unwanted features, we develop a "pattern language" for discussing, and hopefully eliminating them from our projects.

Performance should always be treated as a very objective process, with precise goals set early in the planning phase. This is an easy statement to make, but when a team is under pressure or not operating under reasonable circumstances this can simply fall by the wayside.

Many readers will have seen the situation where either a new client is going live, or a new feature is being launched and an unexpected outage occurs, in UAT if you are lucky, but often in production. The team is then left scrambling to fix and find what has caused the bottleneck. This usually means performance testing has not been carried out, or the team "ninja" had made an assumption and has now disappeared - ninjas are good at this.

Teams that work in this way will likely fall victim to antipatterns more often than a team that follows good performance testing practices and have open and reasoned conversations. As with many development issues, it is often the human elements, such as communication problems, rather than any technical aspect that leads to an application having problems.

One interesting possibility for classification was provided in a blogpost by Carey Flichel called "Why Developers Keep Making Bad Technology Choices"[2]. The post specifically calls out five main reasons which cause developers to make bad choices:

Boredom

Most developers have experienced boredom in a role, for some this doesn't have to last very long before the developer is seeking a new challenge or role. However, in some cases either the opportunities are not present in the organisation or moving somewhere else is not possible.

It is likely many of readers have come across a developer who is simply riding it out, perhaps even actively seeking an easier life. However, bored developers can harm a project in a number of ways. For example, technology or code complexity is introduced that is not required, such as writing a sorting algorithm directly in code when a simple `Collections.sort` would be sufficient. Boredom could also manifest as looking to build components with technologies that are unknown or perhaps don't fit the use case just as an opportunity to use them (see "Résumé Padding").

Résumé Padding

Occasionally the overuse of technology is not tied to boredom, but in fact represents the developer exploiting an opportunity to boost experience with a particular technology on their résumé (or CV). This can be related to boredom, or can be an active attempt by a developer to increase their potential salary and marketability when about to re-enter the job market. It's not usual that many people would get away with this inside a well functioning team, but it can still be the root of a choice that takes a project down an unnecessary path.

The consequences of an unnecessary technology being added due to a developer's boredom or résumé padding can be far-reaching and very long-lived, lasting for many years after the original developer has left for greener pastures.

Peer Pressure

Technical decisions are often at their worse when something is not voiced or discussed at the time. This can manifest in a few ways; perhaps a junior developer not wanting to make a mistake in front of more senior members of their team (or "imposter syndrome"). Equally, it can be caused by a developer not wanting to come across as uninformed in a particular topic. Another particularly toxic type of peer pressure is for competitive teams to want to be seen to have high development velocity and in doing so, rush key decisions without fully exploring all of the consequences.

Lack of Understanding

Developers may look to introduce new tools to help solve a problem because they are not aware of of the full capability of their current tools. It is often tempting to turn to a new and exciting technology component as it is great at performing one specific feature. However introducing more technical complexity must be taken on balance with what the current tools can actually do.

For example, Hibernate is sometimes seen as the answer to simplifying translation between domain objects and databases. If there is only limited understanding of Hibernate on the team, developers can make assumptions about its suitablity based on having seen it used in another project.

This lack of understanding can cause over-complicated usage of Hibernate and unrecoverable production outages. By contrast, rewriting the entire data layer using simple JDBC calls allows the developer to stay on familiar territory. One of the authors has taught a Hibernate course that contained a delegate in exactly this position - he was trying to learn enough Hibernate to see if the application could be recovered, but ended up having to rip out Hibernate over the course of a weekend - definitely not an enviable position.

Misunderstood / Non-Existent Problem

Developers may often use a technology to solve a particular issue where the problem space itself has not been adequately investigated. Without having measured performance values it is almost impossible to understand the success of a particular solution. Often collating these performance metrics enables better understanding of the problem.

To avoid antipatterns it is important to ensure that the team communication about technical issues is open from all participants in the team, and actively encouraged. Where things are unclear gathering factual evidence and working on prototypes can help to steer team decisions. A technology may look attractive, however if the prototype does not measure up a more informed decision can be made.

Performance Antipatterns Catalogue

In this section we will present a short catalogue of performance antipatterns. The list is by no means exhaustive, and there are doubtless many more still to be discovered.

Distracted By Shiny

Description

Newest or coolest tech is often first tuning target, as it can be more exciting to understand how newer technology works rather than digging around in legacy code. It may also be that the code accompanying the newer technology is better written and easier to maintain. Both of these facts push developers towards looking at the newer components of the application.

Example Comment

> “It’s teething trouble - we need to get to the bottom of it”

Reality

This is often just a shot in the dark rather than an effort at targeted tuning or measuring the application. The developer may not fully understand the new technology yet, and will tinker around rather than understand the documentation - often in reality causing other problems. In the cases of new technology examples online are for small or sample datasets and don’t discuss good practice about scaling to an enterprise size.

Discussion

This antipattern is most often seen with younger teams. Eager to prove themselves, or to avoid becoming tied to what they see as *legacy* systems, they are often advocates for newer, “hotter” technologies - which may, coincidentally, be exactly the sort of technologies which would confer a salary uptick in any new role.

Therefore, the logical subconcious conclusion to any performance issue is to first take a look at the new tech - after all, it’s not properly understood, so a fresh pair of eyes would be helpful, right?

Resolutions

-

Measure to determine real location of bottleneck

-
-

Ensure adequate logging around new component

-

-

Look at best practices as well as simplified demos

-
-

Ensure the team understand the new technology and establish a level of best practice across the team

-

Distracted By Simple

Description

The simplest parts of the system are targeted first rather than profiling the application overall and objectively looking for pain points in the application. There may be parts of the system that are deemed “specialist” that only the original wizard that wrote it can edit that part.

Example Comments

> “Let’s get into this by starting with the parts we understand.”
>
> “John wrote that part of the system, and he’s on holiday. Let’s wait until he’s back to look at the performance”

Reality

The developer understands how to tune (only?) that part of the system. There has been no knowledge sharing or pair programming on the various system components, creating single experts.

Discussion

The dual of “Distracted by Shiny”, this antipattern is often seen in an older, more established team, which may be more used to a maintanence or keep-the-lights-on role. If their application has recently been merged or paired with newer technology, the team may feel intimidated or not want to engage with the new systems.

Under these circumstances, developers may feel more comfortable by only profiling those parts of the system that are familiar, hoping that they will be able to achieve the desired goals without going outside of their comfort zone.

Of particular note is that both of these first two antipatterns are driven by a reaction to the unknown - in “Distracted by Shiny” this manifests as a desire by the developer (or team) to learn more and gain advantage - essentially an offensive play. By contrast, “Distracted by Simple” is a defensive reaction - to play to the familiar rather than engage with a potentially threathening new technology.

Resolutions

-

Measure to determine real location of bottleneck

-
-

Ask for help from domain experts if problem is in an unfamiliar component

-
-

Ensure that developers understand all components of the system

-

Performance Tuning Wizard

Description

Management have bought into the Hollywood image of a “lone genius” hacker and have hired someone who fits the stereotype, to move around the company and fix all performance issues in the team, by using their perceived superior performance tuning skills.

Note

There are genuine performance tuning experts and companies out there, most would agree that you have to measure and investigate any problem. It’s unlikely the same solution will apply to all uses of a particular technology in all situations.

Example Comment

> “I’m sure I know just where the problem is…”

Reality

-

The only thing a perceived wizard or superhero is likely to do is challenge the dress code.

-

Discussion

This antipattern can be a general alienation of developers in the team who perceive themselves to not be good enough to address performance issues. It’s concerning as in many cases a small amount of profile lead optimisation can lead to good performance increases.

That is not say that there aren’t specialists that can help with specific technologies, but the thought that there is a stereotype that will understand all performance issues from the beginning is absurd. Many technologists that are performance experts are specialists at measuring and problem solving based on those measurements.

Superheroes types in teams can be very counterproductive if they are not willing to share knowledge or the approaches that they took to resolving a particular issue.

Resolutions

-

Measure to determine real location of bottleneck

-
-

Ensure that when hiring experts onto a team they are willing to share and act as part of the team

-

Tuning By Folklore

Description

Whilst desperate to try and find a solution to a performance problem in production a team member finds a “magic” configuration parameter on a website. Without testing the parameter it is applied to production, because it must improve things exactly as it has for the person on the internet….

Example Comment

> “I found these great tips on Stack Overflow. This changes **everything.**”

Reality

-

Developer does not understand the context or basis of performance tip and true impact is unknown

-
-

It may have worked for that specific system, but it doesn’t mean the change will even have a benefit in another. In reality it could make things worse.

-

Discussion

A performance tip is a workaround for a known problem - essentially a solution looking for a problem. They have a shelf life and usually date badly - someone will come up with a solution which will render the tip useless (at best) in a later release of the software or platform.

One source of performance advice that is usually particularly bad are admin manuals. They contain general advice devoid of context - this advice and “recommended configurations” is often insisted on by lawyers, as an additional line of defence if the vendor is sued.

Java performance happens in a specific context - with a large number of contributing factors. If we strip away this context, then what is left is almost impossible to reason about, due to the complexity of the execution environment.

Note

The Java platform is also constantly evolving, which means a parameter that provided a performance workaround in one version of Java may not work in another.

For example, the switches used to control garbage collection algorithms frequently change between releases. What works in an older VM (version 7 or 6) may not even be applied in the current version (Java 8). There are even switches that are valid and useful in version 7 that will cause the VM not to start up in the forthcoming version 9.

Resolutions

-

  Only apply well-tested and well-understood techniques which directly affect the most important aspects of system.

-
-

  Look for and try out parameters in UAT, but as with any change it is important to prove and profile the benefit.

-
-

  Review and discuss configuration with other developers and operations staff or devops.

-

Configuration can be a one or two character change, but have significant impact in a production environment if not carefully managed.

Blame Donkey

Description

Certain components are always identified as the issue, even if they had nothing to do with the problem.

For example, one of the authors saw a massive outage in UAT the day before go-live. A certain path through the code caused a table lock on one of the central database tables. An error occurred in the code and the lock was retained, rendering the rest of the application unusable until a full restart was performed. Hibernate was used as the data access layer and immediately blamed for the issue. However, in this case, the culprit wasn't Hibernate but an empty catch block for the timeout exception - that did not clean up the database connection. It took a full day for developers to stop blaming Hibernate and to actually look at their code to find the real bug.

Example Comment

> "It's always JMS / Hibernate / A_N_OTHER_LIB"

Reality

-

Insufficient analysis has been done to reach this conclusion

-
-

The usual suspect is the only suspect in the investigation

-
-

The team is unwilling to look wider to establish a true cause

-

Discussion

This antipattern is often displayed by management or the business, as in many cases they do not have a full understanding of the technical stack and so are proceeding by pattern matching and have unacknowledged cognitive biases. However, technologists are far from immune from this antipattern.

Technologists often fall victim to this antipattern when they have little understanding about the code base or libraries outside of the ones usually blamed. It

is often easier to name a part of the application that is often the problem, rather than perform a new investigation. It can be the sign of a tired team, with many production issues at hand. Hibernate is the perfect example of this, in many situations Hibernate grows to the point where it is not setup or used correctly.

The team has a tendency to bash the technology, as they have seen it fail or not perform in the past. The problem could just as easily be the underlying query, use of an inappropriate index, the physical connection to the database, the object mapping layer, etc. Profiling to isolate the exact cause is essential.

Resolutions

-

Resist pressure to rush to conclusions

-
-

Perform analysis as normal

-
-

Communicate the results of analysis to all stakeholders (to encourage a more accurate picture of the causes of problems).

-

Missing the Bigger Picture

Description

The team becomes obsessed with trying out changes or profiling smaller parts of the application without fully appreciating the full impact of the changes. The team tweaks JVM switches to look to gain better performance, perhaps based on an example or a different application in the same company.

The team may also look to profile smaller parts of the application using Microbenchmarking (which is notoriously difficult to get right, as we will explore in Chapter 5).

Example Comments

> “If I just change these settings, we’ll get better performance”
>
> “If we can just speed up method dispatch time…”

Reality

-

Team does not fully understand the impact of changes

-
-

Teams has not profiled the application fully under the new JVM settings

-
-

Overall system impact from a Microbenchmark has not been determined

-

Discussion

The JVM has literally hundreds of switches - this gives a very highly configurable runtime, but gives rise to a great temptation to make use of all of this configurability. This is usually a mistake - the defaults and self-management capabilities are usually sufficient. Some of the switches also combine with each other in unexpected ways - which makes blind changes even more dangerous. Applications even in the same company are likely to operate and profile in a completely different way, so it’s important to spend time trying out settings that are recommended.

Performance tuning is a statistical activity, which relies on a highly specific context for execution. This implies that larger systems are usually easier to benchmark than smaller ones - because with larger systems, the law of large numbers works in the engineers favour, helping to correct for effects in the platform that distort individual events.

By contrast, the more we try to focus on a single aspect of the system, the harder we have to work to unweave the separate subsystems (e.g. threading, GC, scheduling, JIT compilation, etc) of the complex environment that makes up the platform (at least in

the Java / C# case). This is extremely hard to do, and the handling of the statistics is sensitive, and is not often a skillset that software engineers have acquired along the way. This makes it very easy to produce numbers that do not accurately represent the behaviour of the system aspect that the engineer believed they were benchmarking.

This has an unfortunate tendency to combine with the human bias to see patterns, even when none exist. Together, these effects lead us to the spectacle of a perfomance engineer who has been deeply seduced by bad statistics or a poor control - an engineer arguing passionately for a performance benchmark or effect that their peers are simply not able to replicate.

Resolutions

Before putting any change to switches live:

1.

Measure in production

2.

3.

Change one switch at a time in UAT

4.

5.

Ensure that your UAT environment has the same stress points as production

6.

7.

Also the same test data for the normal load condition.

8.

9.

Test change in UAT

10.

11.

Retest in UAT

12.

13.

Have someone recheck your reasoning

14.

15.

Pair with them to discuss your conclusions

16.

There’s no point in having an optimization that helps your application only in high stress situation and kills performance in the general case. Obtaining sets of data like this which can be used for accurate emulation can be difficult.

UAT Is My Desktop

Description

UAT environments often differ significantly from production, although not always in a way that’s expected or fully understood. Many developers will have worked in situations where a low powered desktop is used to write code for high powered production severs. However it’s also becoming more common that a developers machine is massively more powerful than the small services deployed in production. Low powered micro-environments are usually not a problem, as they can often be virtualised for a developer to have one each. This is not true of high powered production machines, which will often have significantly more cores, RAM and efficient IO than a developer’s machine.

Example Comment

> “A full-size UAT environment would be too expensive”

Reality

-

Outages caused by differences in environments are almost always more expensive than a few more boxes

-

Discussion

UAT is my Desktop stems from a different kind of cognitive bias than we have previously seen. This bias insists that doing some sort of UAT must be better than doing none at all. Unfortunately, this hopefulness fundamentally misunderstands the complex nature of enterprise environments. For any kind of meaningful extrapolation to be possible, the UAT environment must be production like.

In modern adaptive environments, the runtime subsystems will make best use of the available resources. If these differ radically from the target deployment, they will make different decisions under the differing circumstances - rendering our hopeful extrapolation useless at best.

Resolutions

-

Track the cost of outages and opportunity cost related to lost customers

-
-

Buy a UAT environment that is identical to production

-
-

In most cases, the cost of the first, far outweighs the second and sometimes the right case needs to be made to managers

•

Production-like Data Is Hard

Description

Also known as the DataLite antipattern, this antipattern relates to a few common pitfalls that people encounter whilst trying to represent production like data. Consider a trade processing plant at a large bank that processes futures and options trades that have been booked but need to be settled. Such a system would typically handle millions of messages a day. Consider the following UAT strategies and their potential issues.

To make things easy to test the mechanism was to capture a small selection of these messages during the course of the day. The messages are then all run through the UAT system. This approach fails to capture burst like behaviour of messages that system could see. It may also not capture the warm up caused by more futures trading on a particular market before another market opens that trades options.

To make things easier to assert the values on the trades and options are updated to only use simple values for assertion. This does not give us the “realness” of production data. Consider we are using an external library or system for options pricing, it would be impossible for us ot evaluate with our UAT dataset that this production dependency has not now caused a performance issue - as the range of calculations we are performing is a simplified subset of production data.

To make things easier all values are pushed through the system at once. This is often done in UAT, but misses out key warm up and optimisations that may happen when then data is fed at a different rate.

Most of the time in UAT the test data set is simplified **to make things easier**. However it rarely makes results useful.

Example Comment

> “It’s too hard to keep production and UAT in synch”
>
> “It’s too hard to manipulate data to match what the system expects”
>
> “Production data is protected by security considerations. Developers should not have access to it.”

Reality

-

Data in UAT must be production-like for accurate results. If data is not available for security reasons then it should be scrambled (aka masked or obfuscated) so it can still be used for a meaningful test. Another option is to partition UAT so developers still don’t see the data, but can see the output of the performance tests to be able to identify problems.

-

Discussion

This antipattern also falls into the trap of “something must be better than nothing”. The idea is that testing against even out of date and unrepresentative data is better than not testing.

As before, this is an extremely dangerous line of reasoning. Whilst testing against **something** (even if it is nothing like production data) at scale can reveal flaws and omissions in the system testing, it provides a false sense of security.

When the system goes live, and the usage patterns fail to conform to the expected norms that have been anchored by UAT data, the development and ops teams may well find that they have become complacent due to the warm glow that UAT has provided, and are unprepared for the sheer terror that can quickly follow an at-scale production release.

Resolutions

-

Consult data domain experts and invest in a process to migrate production data back into UAT, scrambling or obfuscating data if necessary.

-
-

Over-prepare for releases with expectation of high volumes of customers or transactions

-

Cognitive Biases and Performance Testing

Humans can be bad at forming accurate opinions quickly, even when faced with a problem that can draw upon past experiences and similar situations.

Note

A cognitive bias is a psychological effect that cause the human brain to draw incorrect conclusions. They are especially problematic because the person exhibiting the bias is usually unaware of it and may believe they are being rational.

Many of the antipatterns that have been explored in this chapter are caused, in whole or in part, by one or more cognitive biases that are in turn based on an unconscious assumption.

For example, Blame Donkey is caused by a cognitive bias, because if a component has caused several recent outages then the team could be expecting the same component to be the cause of any new performance problem. Any data that's analysed may be more likely to be considered credible if it confirms the idea that the Blame Donkey is responsible. The antipattern combines aspects of the biases known as Confirmation Bias and Recency Bias.

Note

A single component in Java can behave differently from application to application depending on how it is optimised at runtime. In order to remove any pre-existing bias it is important to check the application is looked at as a whole.

Before we move on, let's consider a pair of biases that are dual to each other. These are the biases that assume that the problem is not software related at all and it must be the infrastructure the software is running on. "This worked fine in UAT so there must be a problem with the production kit", or the Works For Me antipattern. The converse is to assume that every problem must be caused by software, because that's the part of the system the developer knows about and can directly affect.

Reductionist Thinking

This cognitive bias is based on an analytical approach that insists that if broken into small enough pieces, a system can be understood by understanding the constituent parts. Understanding each part means a reduction in the chance an assumption is made. The problem with this view is that in complex systems it just isn't true. Non-trivial

software (or physical) systems almost always display emergent behaviour, where the whole is greater than simply aggregating the parts would indicate.

Confirmation Bias

Confirmation bias can lead to significant problems when it comes to performance testing or attempting to look at application subjectively. A confirmation bias is introduced, usually not intentionally, when a poor test set is selected or results from the test are not analysed in a statistically sound way. Confirmation bias is quite hard to counter, because it can be lead by emotions or someone in the team trying to prove a point.

Consider an antipattern such as Distracted by Shiny, where a team member is looking to bring in the latest and greatest NoSQL database. They run some tests against data that isn't like production, because representing the full schema is too complicated for evaluation purposes. They quickly prove that on a test set the NoSQL database produces superior access times on their local machine. The developer had told everyone this would be the case, and on seeing the results they proceed with a full implementation. There are several antipatterns at work, all leading to new uproved assumptions in the new library stack.

Fiddle With Switches

Tuning by Folklore, missing the bigger picture and abuse of microbenchmarks are all examples of antipatterns that are caused at least in part by a combination of the reductionism and confirmation biases. One particular egregious example is a subtype of Tuning by Folklore - known as *Fiddle With Switches*.

This antipattern arises because although the VM attempts to choose settings appropriate for the detected hardware, there are some cirucmstances where the engineer will need to manually set flags to tune the performance of code. This is not harmful in itself, but there is a hidden cognitive trap here, in the extremely configurable nature of the JVM with command-line switches.

To see a list of the VM flags use the following switch:

```
-XX:+PrintFlagsFinal
```

As of Java 8u131, this produces over 700 posible switches. Not only that, but there are also additional tuning options available only when the VM is running in diagnostic mode. To see these add the additional switch:

```
-XX:+UnlockDiagnosticVMOptions
```

This unlocks around another 100 switches. There is no way that any human can correctly reason about the aggregate effect of applying the possible combinations of these switches. Moreover, in most cases, experimental observations will show that the effect of changing switch values is small - often much smaller than developers expect.

Fog of war (Action Bias)

This bias usually manifests itself during outages or situations where the system is not performing as expected. The most common situations include:

-

  Changes to infrastructure that the system runs on, perhaps without notification or realising there would be an impact

-
-

  Changes to libraries that our system is dependant on

-
-

  A strange bug or race condition the manifests itself on the busiest day of the year

-

In a well maintained applciation with sufficient logging and monitoring, this should lead to a clear error message being generated that will lead the support team to the cause of the problem.

However, too many applications have not tested failure scenarios and lack appropriate logging. Under these circumstances even experienced engineers can fall into the trap of needing to feel that they’re doing something to resolve the outage, and mistake motion for velocity, and the “fog of war” descends.

At this time, many of the human elements discussed in this chapter can come into effect if participants are not systematic about their approach to the problem. For example, an antipattern such as Blame Donkey may shortcut a full investigation and lead the production team down a particular path of investigation - often missing the bigger picture. A similar example may include trying to break the system down into its constituent parts and look through the code at a low level without first establishing in which subsystem the problem truly resides.

In the past when dealing with outage scenarios it always pays to use a systematic approach, leaving anything that did not require a patch to a post mortem. However, this is the realm of human emotion and it can be very difficult to take the tension out of the situation, especially during an outage.

Risk bias

Humans are naturally risk averse and resistant to change. Mostly this is because people have seen change and how it can go wrong. This leads a natural stance ot attempt to avoid that risk. This can be incredibly frustrating when taking small calculated risks can move the product forward. Risk bias is reduced significantly by having a robust set of tests, both in terms of unit tests and production regression tests. If either of these are not trusted change becomes extremely difficult and without control of the risk factor.

This even manifests by a failure to learn from application problems (even service outages) and implement appropriate mitigation.

Ellsberg's Paradox

As an example of how bad humans are at understanding probability, let us consider Ellsberg's Paradox. Named for the famous US investigative journalist and whistleblower, Daniel Ellsberg, the paradox deals with the human desire for "known unknowns" over "unknown unknowns".[3]

The usual formulation of Ellsberg's Paradox is as a simple probablity thought experiment. Consider a barrel, containing 90 colored balls - 30 are known to be blue, and the rest are either red or green. The exact distribution of red and green balls is not known, but the barrel, the balls and therefore the odds are fixed throughout.

The first step of the paradox is expressed as a choice of wagers. The player can choose either to take either of two bets:

A) The player will win $100 if a ball drawn at random is blue

B) The player will win $100 if a ball drawn at random is red

Most people choose A) as it represents known odds - the likelihood of winning is exactly 1/3. However, when presented with a second bet (assuming that when a ball is removed it is placed back in the same barrel and then re-randomized), something surprising happens:

C) The player will win $100 if a ball drawn at random is blue or green

D) The player will win $100 if a ball drawn at random is red or green

In this situation, bet D corresponds to known odds (2/3 chance of winning), so virtually everyone picks this option.

The paradox is that the set of choices A and D is irrational. Choosing A implicitly expresses an opinion about the distribution of red and green balls - effectively that "there are more green balls than red balls". Therefore, if A is chosen, then the logical strategy is to pair it with C, as this would provide better odds than the safe choice of D.

When evaluating performance results it is essential to handle the data in an appropriate manner and avoid falling into unscientific and subjective thinking. In this chapter, we have met some of the types of test, testing best practices and antipatterns that are native to performance analysis.

In the next chapter, we're going to move on to looking at low-level performance measurements and the pitfalls of microbenchmarks and some statistical techniques for handling raw results obtained from JVM measurements.

[1] The term was popularized by the book "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis" by Brown et al.

[2] http://www.carfey.com/blog/why-developers-keep-making-bad-technology-choices/

[3] To reuse the phrase made famous by Donald Rumsfeld

Chapter 5. Microbenchmarking and statistics

In this chapter, we will consider the specifics of measuring Java performance numbers directly. The dynamic nature of the JVM means that performance numbers are often harder to handle than many developers expect. As a result, there are a lot of inaccurate or misleading performance numbers floating around the Internet.

A primary goal of this chapter is to ensure that the reader is aware of these possible pitfalls and only produces performance numbers that can be relied upon by themselves and others. In particular, the measurement of small pieces of Java code ("microbenchmarking") is notoriously subtle and difficult to do correctly, and this subject and its proper usage by performance engineers is a major theme throughout the chapter.

> The first principle is that you must not fool yourself – and you are the easiest person to fool.
>
> Richard Feynman

The second portion of the chapter describes how to use the gold standard of microbenchmarking tools - JMH. If, even after all the warnings and caveats, you really feel that your application and use cases warrant the use of microbenchmarks, then you will avoid numerous well-known pitfalls and "bear traps" by starting with the most reliable and advanced of the available tools.

Finally, we turn to the subject of statistics. The JVM routinely produces performance numbers that require somewhat careful handling. The numbers produced by microbenchmarks are usually especially sensitive, and so it is incumbent upon the performance engineer to treat the observed results with a degree of statistical sophistication. The last sections of the chapter explain some of the techniques for working with JVM performance data, and problems of interpreting data.

Introduction to measuring Java performance

In Section 1.2, we described performance analysis as a synthesis between different aspects of the craft, that resulted in a discipline that is fundamentally an experimental science.

That is, if we want to write a good benchmark (or microbenchmark) then it can be very helpful to consider it as though it were a science experiment.

This approach leads us to view the benchmark as a “black box” - it has inputs and outputs, and we want to collect data, from which we can conjecture, or infer results. However, we must be cautious - it is not enough to simply collect data. We need to ensure that we are not deceived by our data.

> Benchmark numbers don’t matter on their own. It’s important what models you derive from those numbers.
>
> Aleksey Shipilëv

Our ideal goal is therefore to make our benchmark a fair test - meaning that as far as possible we only want to change a single aspect of the system and ensure any other external factors in our benchmark are controlled. In an ideal world, the other possibly changeable aspects of the system would be completely invariant between tests, but we are rarely so fortunate in practice.

Note

Even if the goal of a scientifically pure fair test is unachievable in practice, it is essential that our benchmarks are at least repeatable, as this is the basis of any empirical result.

One central problem with writing a benchmark for the Java platform is the sophistication of the Java runtime. A considerable portion of this book is devoted to illuminating the automatic optimisations that are applied to a developer’s code by the JVM. When we think of our benchmark as a scientific test in the context of these optimisations then our options become limited.

That is, to fully understand and account for the precise impact that these optimisations have is all but impossible. Accurate models of the “real” performance of our application code are difficult to create and tend to be limited in their applicability.

Put another way, we cannot truly divorce the executing Java code from the JIT compiler, memory mangement and other subsystems provided by the Java runtime. Neither can we ignore the effects of operating system, hardware or runtime conditions (load, etc.) that are current when our tests are run.

> No man is an island, Entire of itself
>
> John Donne

It is easier to smooth out these effects by dealing with a larger aggregate (a whole system or subsystem). Conversely, when dealing with small-scale or microbenchmarks, it

is much more difficult to truly isolate application code from the background behavior of the runtime. This is the fundamental reason why microbenchmarking is so hard, as we will discuss.

Let's consider what appears to be a very simple example - a benchmark of code that sorts a list of 100 thousand numbers. We want to examine it with the point of view of trying to create a truly fair test.

The benchmark creates an array of random integers and once this is complete logs the start time of the benchmark. The benchmark then loops round copying the template array and then runs a sort over the data. Once this has run for `I` times the duration is converted to seconds and divided by the number of iterations to give us the time taken per operation.

The first concern with the benchmark is that it goes straight into testing the code, without any consideration for warming up the JVM. Consider the case where the sort is running in a server application in production. It is likely to have been running for hours, maybe even days. However, we know that the JVM includes a just-in-time compiler, that will convert interpreted bytecode to highly optimized machine code. This compiler only kicks in after the method has been run a certain number of times.

The test we are therefore conducting is not representative of how it will behave in production. Time will be spent by the JVM optimising the call whilst we are trying to benchmark. We can see this effect by running with a few JVM flags in this fragment of the output:

```
java -Xms2048m -Xmx2048m -XX:+PrintCompilation ClassicSort
```

Recall that the `-Xms` and `-Xmx` options control the size of the heap, in this case pinning the heap size to 2G. The `PrintCompilation` flag outputs a logline whenever a method is compiled (or other compilation event happens).

```
Testing Sort Algorithm
  73    29    3       java.util.ArrayList::ensureExplicitCapacity (26 bytes)
  73    31    3       java.lang.Integer::valueOf (32 bytes)
  74    32    3       java.util.concurrent.atomic.AtomicLong::get (5 bytes)
  74    33    3       java.util.concurrent.atomic.AtomicLong::compareAndSet (13 bytes)
  74    35    3       java.util.Random::next (47 bytes)
  74    36    3       java.lang.Integer::compareTo (9 bytes)
  74    38    3       java.lang.Integer::compare (20 bytes)
  74    37    3       java.lang.Integer::compareTo (12 bytes)
  74    39    4       java.lang.Integer::compareTo (9 bytes)
  75    36    3       java.lang.Integer::compareTo (9 bytes) made not entrant
  76    40    3       java.util.ComparableTimSort::binarySort (223 bytes)
  77    41    3       java.util.ComparableTimSort::mergeLo (656 bytes)
```

```
  79    42    3       java.util.ComparableTimSort::countRunAndMakeAscending (123
bytes)
  79    45    3       java.util.ComparableTimSort::gallopRight (327 bytes)
  80    43    3       java.util.ComparableTimSort::pushRun (31 bytes)
```

The JIT compiler is working overtime to optimise parts of the call hierarchy to make our code more efficient. This means the performance of the benchmark changes over the duration of our timing capture, we have inadvertently left a variable uncontrolled in our experiment. A technique known as “warmup” is desirable, which will allow the JVM to settle down before we capture our timings. Usually this involves running the code we are about to benchmark for a number of iterations without capturing the timing details.

Another external factor that we need to consider is garbage collection. Ideally we want to try and surpress GC from running during our time capturing and also to be normalised after setup. Due to the non-deterministic nature of garbage collection this is incredibly difficult to control.

One improvement that could definitely be made is to ensure that we are not capturing timings whilst GC is likely to be running. We could potentially look to ask the system for a GC to be run and wait a short time, but the system could decide to ignore this call. As it stands, the timing in this benchmark is far too broad, so we need more detail about the garbage collection events that could be occuring.

Not only that, but as well as selecting our timing points we also want to select a reasonable number of iterations, which can be tricky to figure out by trial and improvement. The effects of garbage collection can be seen with another VM flag (for the detail of the log format, see Chapter 8):

```
java -Xms2048m -Xmx2048m -verbose:gc ClassicSort
```

This will produce GC log entries similar to the following:

```
Testing Sort Algorithm[GC (Allocation Failure)  524800K->632K(2010112K), 0.0009038
secs][GC (Allocation Failure)  525432K->672K(2010112K), 0.0008671 secs]Result:
9838.556465303362 op/s
```

Another common mistake that is made in benchmarks is to not actually use the result generated from the code we are testing. In the benchmark `copy` is effectively dead code, it is therefore possible for the JIT to identify it is a dead code path and optimize away what we are actually trying to benchmark.

Looking at a single timed result, even though averaged does not give us the full story of how our benchmark performed. Ideally we want to capture the margin of error to understand the reliability of the collected value. If the error margin is high it may

point to an uncontrolled variable or indeed that the code we have written is not performant. Either way without capturing the margin of error there is no way to identify there is even an issue.

Benchmarking even a very simple sort can have pitfalls that mean the benchmark is wildly thrown out, however things rapidly get much, much worse. Consider a benchmark that looks to assess multithreaded code. Multithreaded code is extremely difficult to benchmark as it requires ensuring that all the threads are held until each has fully started up from the beginning of the benchmark to ensure accurate results. If this is not the case the margin of error will be high.

There are also hardware considerations when it comes to benchmarking concurrent code, and this is beyond simply the hardware configuration. Consider if power management was to kick in or there are other contentions on the machine.

Getting the benchmark code correct is complicated and involves considering a lot of factors. As developers the primary concern is the code we are looking to profile - rather than all the issues highlighted above. All the above concerns combine to create a situation in which unless you are a JVM expert it is extremely easy to miss something and get an erroneous benchmark result.

There are two ways to deal with this problem. The first is to only benchmark systems as a whole. In this case, the low-level numbers are simply ignored and not collected. The overall outcome of so many copies of separate effects is to average out and allow meaningful large-scale results to be obtained. This approach is the one that is needed in most situations and by most developers.

The second approach is to try to address many of the concerns above by using a common framework, to allow meaningful comparison of related low-level results. The ideal framework would take away some of the pressures discussed above. Such a tool would have to follow the mainline development of OpenJDK to ensure new optimisations and other external control variables were managed.

Fortunately, such a tool does actually exist, and it is the subject of our next section. For most developers it should be regarded purely as reference material, and it can be safely skipped in favour of Section 5.3.

Introduction to JMH

We open with an example (and a cautionary tale) of how and why microbenchmarking can easily go wrong if it is approached naïvely. From there, we introduce a set of heuristics that indicate whether your use case is one where microbenchmarking is appropriate - for the vast majority of applications, the outcome will be that the technique is not suitable.

Don’t microbenchmark if you can help it - a true story

After a very long day in the office one of the authors was leaving the building when they passed a colleague still working at her desk - staring intensely at a single Java method. Thinking nothing of it, they left to catch a train home. However, two days later a very similar scenario played out - with a very similar method on her screen and a tired, annoyed look on her face. Clearly some deeper investigation was required.

The application that she was renovating had an easily observed performance problem. The new version was not performing as well as the version that the team was looking to replace, despite using newer versions of well-known libraries. She had been spending some of her time removing parts of the code and writing small benchmarks in an attempt to find where the problem was hiding.

The approach somehow felt wrong, looking for a needle in a haystack. Instead, we worked together on another approach, and quickly confirmed that the application was maxing out CPU utilisation. As this is a known good use case for execution profilers (see Chapter 14 for the full details of when to use profilers), ten minutes profiling the application found the true cause. Sure enough, the problem wasn’t in our application code at all, but in a new infrastructure library we were using.

This war story illustrates an approach to Java performance that is unfortunately all too common. Quite often, this approach is caused by the very problem-solving and reductionist approach mentioned in Section 4.4.1. Developers can become obsessed with the idea that their own code must be to blame, and miss the bigger picture.

Tip

Looking closely at small-scale code constructs is often where developers want to start hunting for problems, however benchmarking at this level is extremely difficult and has some dangerous “bear traps”.

As we discussed briefly in Chapter 2, the dynamic nature of the Java platform, and features like garbage collection and aggressive JIT optimization lead to performance that is hard to reason about directly. Worse still, performance numbers are frequently dependent on the exact runtime circumstances in play when the application is being measured.

Note

It is almost always easier to analyse the true performance of an entire Java application than a small Java code fragment.

However, there are occasionally times when we need to directly analyse the performance of an individual method or even a single code fragment. This analysis should not be undertaken lightly, and in general there are three main use cases for low-level analysis or microbenchmarking:

- You're developing general-purpose library code with broad use cases
- You're a developer on OpenJDK or another Java platform implementation
- You're developing extremely latency sensitive code (e.g. low-latency trading)

The rationale for the three use cases is slightly different in each case:

General purpose libraries (by definition) have limited knowledge about the contexts in which they will be used. Examples of these types of libraries include Google Guava or the Eclipse Collections (originally contributed by Goldman Sachs). They need to provide acceptable-or-better performance across a very wide range of use cases - from data sets containing a few dozen elements up to 100s of millions of elements.

Due to the broad nature of how they will be used, general purpose libraries are sometimes forced to use microbenchmarking as a proxy for more conventional performance and capacity testing techniques.

Platform developers are a key user community for microbenchmarks, and the JMH tool was created by the OpenJDK team primarily for their own use. The tool has proved to be useful to the wider community of performance experts, though.

Finally, there are some developers who are working at the cutting edge of Java performance, who may wish to use microbenchmarks for the purpose of selecting algorithms and techniques that best suits their applications and extreme use cases. This would include low-latency financial trading but relatively few other cases.

Heuristics for when to microbenchmark

Whilst it should be apparent if you are a developer who is working on OpenJDK or a general purpose library, there may be developers who are confused about whether their performance requirements are such that they should consider microbenchmarks.

> The scary thing about microbenchmarks is that they always produce a number, even if that number is meaningless. They measure something, we're just not sure what.
>
> Brian Goetz

Generally, only the most extreme applications should use microbenchmarks. There are no definitive rules, but unless your application meets most or all of these criteria, you are unlikely to derive genuine benefit from microbenchmarking your application:

-

Your total code path execution time should certainly be less than 1ms, and probably less than 100us.

-
-

You should have measured your memory (object) allocation rate (see Chapter 7 and Chapter 8 for details) and it should be <1 MB/s and ideally very close to zero-allocation.

-
-

You should be using close to 100% of available CPU, and the system utilisation rate should be remaining low (under 10%)

-
-

You should have already used an execution profiler (see Chapter 14) to understand the distribution of which methods are consuming CPU. There should be at most 2 or 3 dominant methods in the distribution.

-

With all of this said, it is hopefully obvious that microbenchmarking is an advanced, and rarely undertaken, technique. However, it is useful to understand some of the basic theory and complexity that it reflects, as it leads to a better understanding of the difficulties of performance work in less extreme applications on the Java platform.

> Any nanobenchmark test that does not feature disassembly/codegen analysis is not to be trusted. Period.
>
> Aleksey Shipilëv

The rest of this section explores microbenchmarking more thoroughly and introduces some of the tools and considerations developers must take into account in order to produce results that are reliable and don't lead to an incorrect conclusion. It should be useful background for all performance analysts, regardless of whether it is directly relevant to your current projects.

The JMH Framework

JMH is designed to be the framework that resolves the issues that we have just discussed in this chapter.

> JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targetting the JVM.
>
> OpenJDK

There have been several attempts at simple benchmarking libraries in the past with, for example Google Caliper being one of the most well regarded amongst developers. However all these frameworks have had many challenges to contend with, and often what seems like a rational way of setting up or measuring code performance can have some subtle bear traps to contend with. This is especially true with the continually evolving nature of the JVM as new optimisations are applied.

JMH is very different in that regard, and has been worked on by the same engineers that build the JVM. Therefore, the JMH authors know how to avoid the gotchas and optimisation bear traps that exist within each version of the JVM. As a result, JMH evolves as a benchmarking harness with each release of the JVM, allowing developers to simply focus on using the tool and on the benchmark code itself.

JMH takes into account some key benchmark harness design issues, in addition to some of the problems already highlighted.

A benchmark framework has to be dynamic, as it does not know the contents of the benchmark at compile time. One obvious choice to get around this would be to execute benchmarks the user has written using reflection. However, this then involves another complex JVM subsystem in the benchmark execution path. Instead, JMH operates by generating additional Java source from the benchmark, via annotation processing.

Note

Many common annotation-based Java frameworks (e.g. JUnit) use reflection to achieve their goals, so the use of a processor that generates additional source may be somewhat unexpected to some Java developers.

If the benchmark framework was to call the users code for a large number of iterations there is an issue that loop optimisations may be triggered. This means the actual process of running the benchmark can cause issues with reliable results.

In order to avoid hitting loop optimisation constraints JMH generates code for the benchmark wrapping the benchmark code in a loop, with the iteration count carefully set to a value which avoids optimisation.

Executing Benchmarks

The complexities involved in JMH execution are mostly hidden from the user and setting up a simple benchmark using Maven is straightforward. A new JMH project can be setup by executing the following command:

```
$ mvn archetype:generate \
          -DinteractiveMode=false \
          -DarchetypeGroupId=org.openjdk.jmh \
          -DarchetypeArtifactId=jmh-java-benchmark-archetype \
          -DgroupId=org.sample \
          -DartifactId=test \
          -Dversion=1.0
```

This downloads the required artefacts and creates a single benchmark stub to house your code. The benchmark is annotated with `@Benchmark`, indicating that the harness will execute the method to benchmark it (after the framework has performed various setup tasks).

```
public class MyBenchmark {
    @Benchmark
    public void testMethod() {
        //Stub for code
    }}
```

As the author of the benchmark there are parameters that can be configured to setup the benchmark execution. The parameters can be set on either the command line or in the main method of the benchmark. The parameters on the command line override any parameters that have been set in the main method.

```
public class MyBenchmark {

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(SortBenchmark.class.getSimpleName())
                .warmupIterations(100)
                .measurementIterations(5).forks(1)
                .jvmArgs("-server", "-Xms2048m", "-Xmx2048m").build();

        new Runner(opt).run();
    }}
```

Usually a benchmark requires some setup, for example creating a dataset or setting up the conditions required for a orthogonal set of benchmarks to compare performance.

State and controlling state is another feature that is baked into the JMH framework. The `@State` annotation can be used to define that state, and accepts the `Scope` enum to define where the state is visible `Benchmark`, `Group` or `Thread`. Objects that are annotated with `@State` are reachable for the lifetime of the benchmark: it may be necessary to perform some setup.

Multithreaded code also requires careful handling in order to ensure that benchmarks are not skewed by state that is not well managed.

In general, if the code executed in a method has no side-effects and the result is not used, then the method is a candidate for removal by the JVM. JMH needs to prevent this effect from occuring and in fact makes this extremely straightforward for the benchmark author. Single results can be returned from the benchmark method, and the framework ensures that the value is implicity assigned to a blackhole, a mechanism developed by the framework authors to have negligible performance overhead.

If a benchmark performs multiple calculations it may be costly to combine and return the results from the benchmark method. In that scenario it may be necessary to use an explicit blackhole, which is achieved by creating a benchmark that takes a `Blackhole` as a parameter, that the benchmark will inject for the author.

Blackholes provide four protections related to optimisations that could potentially impact the benchmark. Some protections are about preventing the benchmark over-optimising due to the limited scope of the benchmark, and the others are about avoiding predictable runtime patterns of data which would not happen in a typical run of the system. The protections are:

- Remove the potential for dead code to be removed as an optimisation at runtime

-
-

Prevent repeated calculations from being folded into constants.

-
-

Prevent false sharing, where the reading or writing of a value can cause the current cache line to be impacted

-
-

Protect against “write walls”

-

The term “wall” in performance generally refers to a point at which your resources become saturated and the impact to the application is effectively a bottleneck. Hitting the write wall can impact caches and pollute buffers that are being used for writing. If you do this within your benchmark you are potentially impacting your benchmark in a big way.

As documented in the `BlackHole` JavaDoc (and as noted earlier) in order to provide these protections intimate knowledge of the JIT is required to build a benchmark that avoid optimizations.

Let’s take a quick look at the two consume methods used by Blackholes to give us an insight into some of the tricks JMH uses (feel free to skip this bit if you’re not interested in how JMH is implemented):

```
public volatile int i1 = 1, i2 = 2;
/** * Consume object. This call provides a side effect preventing JIT to eliminate * dependent computations. * * @param i int to consume. */public final void consume(int i) {
    if (i == i1 & i == i2) {
        // SHOULD NEVER HAPPEN
        nullBait.i1 = i; // implicit null pointer exception
    }}
```

This code is repeated for consuming all primitives (changing int for the corresponding primitive type. The variables i1 and i2 are declared as volatile, which means the runtime must re-evaluate. The if statement can never be true, but the compiler must allow the code to run. Also note the use of the “bitwise AND” inside the if

statement. This avoids additional branch logic being a problem and results in a more uniform performance.

```
public int tlr = (int) System.nanoTime();
/** * Consume object. This call provides a side effect preventing JIT to eliminate *
dependent computations. * * @param obj object to consume. */public final void
consume(Object obj) {
        int tlr = (this.tlr = (this.tlr * 1664525 + 1013904223));
    if ((tlr & tlrMask) == 0) {
        // SHOULD ALMOST NEVER HAPPEN IN MEASUREMENT
        this.obj1 = obj;
        this.tlrMask = (this.tlrMask << 1) + 1;
    }}
```

When it comes to objects it would seem at first the same logic could be applied, as nothing the user has could be equal to objects that the `BlackHole` holds. However, the compiler is also trying to be smart about this too. If the compiler asserts that the object is never equal to something else due to escape analysis it is possible that comparison itself could be optimized to return false.

Instead, objects are consumed under a condition that only executes in rare scenarios. The value for `tlr` is computed and put against a mask to reduce the changes of a 0 value, but doesn't outright eliminate it. This ensures objects are consumed largely without the requirement to assign the object. Benchmark framework code is extremely fun to review, as it is so different from real-world Java applications. In fact, if code like that was found anywhere in a production Java applications then the developer responsible should probably be fired.

As well as writing an extremely accurate micro benchmarking tool the authors have also managed to create impressive documentation on the classes. If you're interested in the magic that is going on behind the scenes the comments explain it well.

It doesn't take much with the above information to get a simple benchmark up and running, however JMH also has some fairly advanced features. The official documentation has examples of each, all of which are worth reviewing.

Interesting features that demonstrate the power of JMH and its relative closeness to the JVM include:

- Being able to control the compiler
-
-

Simulating CPU usage levels during a benchmark

-

The `@CompilerControl` annotation can be used to ask the compiler not to inline, explicitly inline or exclude the method from compilation. This is extremely useful if you come across a performance issue where it is suspected that the JVM is causing specific problems due to inlining or compilation. Another cool feature is using blackholes to actually consume CPU cycles to allow you to simulate a benchmark under various CPU loads.

```
@State(Scope.Benchmark)@BenchmarkMode(Mode.Throughput)@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)@OutputTimeUnit(TimeUnit.SECONDS)@Fork(1)public class SortBenchmark {

    private static final int N = 1_000;
    private static final List<Integer> testData = new ArrayList<>();

    @Setup
    public static final void setup() {
        Random randomGenerator = new Random();
        for (int i = 0; i < N; i++) {
            testData.add(randomGenerator.nextInt(Integer.MAX_VALUE));
        }
        System.out.println("Setup Complete");
    }

    @Benchmark
    public List<Integer> classicSort() {
        List<Integer> copy = new ArrayList<Integer>(testData);
        Collections.sort(copy);
        return copy;
    }

    @Benchmark
    public List<Integer> standardSort() {
        return testData.stream().sorted().collect(Collectors.toList());
    }

    @Benchmark
    public List<Integer> parallelSort() {
        return testData.parallelStream().sorted().collect(Collectors.toList());
    }

    public static void main(String[] args) throws RunnerException {
```

```
        Options opt = new OptionsBuilder()
                .include(SortBenchmark.class.getSimpleName()).warmupIterations(100)
                .measurementIterations(5).forks(1)
                .jvmArgs("-server", "-Xms2048m", "-Xmx2048m")
                .addProfiler(GCProfiler.class)
                .addProfiler(StackProfiler.class)
                .build();

        new Runner(opt).run();
    }}
```

Running the benchmark produces the following output:

```
Benchmark                                Mode  Cnt      Score      Error
Unitsoptjava.SortBenchmark.classicSort   thrpt  200  14373.039 ±  111.586
ops/soptjava.SortBenchmark.parallelSort  thrpt  200   7917.702 ±   87.757
ops/soptjava.SortBenchmark.standardSort  thrpt  200  12656.107 ±   84.849  ops/s
```

Looking at this benchmark it would be easy to jump to the quick conclusion that a classic method of sorting is more effective than using streams. Both code runs use one array copy and one sort, so it should be OK. Developers may look at the low error rate and high throughput and conclude thet the benchmark must be correct.

Lets consider some reasons why our benchmark might not be giving an accurate picture of performance - basically trying to answer the question “Is this a controlled test?”. To begin with, lets look at the impact of garbage collection on the classicSort test.

```
Iteration   1:[GC (Allocation Failure)  65496K->1480K(239104K), 0.0012473 secs][GC
(Allocation Failure)  63944K->1496K(237056K), 0.0013170 secs]10830.105 ops/sIteration
2:[GC (Allocation Failure)  62936K->1680K(236032K), 0.0004776 secs]10951.704 ops/s
```

In this snapshot it is clear that there is one GC cycle running per iteration (approximately). Comparing this to parallel sort is interesting:

```
Iteration   1:[GC (Allocation Failure)  52952K->1848K(225792K), 0.0005354 secs][GC
(Allocation Failure)  52024K->1848K(226816K), 0.0005341 secs][GC (Allocation Failure)
51000K->1784K(223744K), 0.0005509 secs][GC (Allocation Failure)
49912K->1784K(225280K), 0.0003952 secs]9526.212 ops/sIteration   2:[GC (Allocation
Failure)  49400K->1912K(222720K), 0.0005589 secs][GC (Allocation Failure)
49016K->1832K(223744K), 0.0004594 secs][GC (Allocation Failure)
48424K->1864K(221696K), 0.0005370 secs][GC (Allocation Failure)
47944K->1832K(222720K), 0.0004966 secs][GC (Allocation Failure)
47400K->1864K(220672K), 0.0005004 secs]
```

So, by adding in flags to see what is causing this unexpected disparity, we can see that something else in the benchmark is causing noise - in this case garbage collection.

The takeaway is that it is easy to assume that the benchmark represents a controlled environment, but the truth can be far more slippery. Often the uncontrolled variables are hard to spot, so even with a harness like JMH, caution is still required. We also need to take care to correct for our confirmation biases and ensure we are measuring the observables that truly reflect the behavior of our system.

In Chapter 10 we will meet JITWatch, which will also give us another view into what is happening with our bytecode by the JIT compiler. This can often help lend an insight into why bytecode generated for a particular method may be causing the benchmark to not perform as expected.

Statistics for JVM performance

If performance analysis is truly an experimental science, then we will inevitably find ourselves dealing with distributions of results data. Statisticians and scientists know that results that stem from the real world are virtually never represented by clean, stand-out signals. Instead, we must deal with the world as we find it, rather than the over-idealized state in which we would like to find it.

> In God we trust. Everyone else, bring data.
>
> Michael Bloomberg

All measurements contain some amount of error. In the next section we'll describe the two main types of error that a Java developer may expect to encounter when doing performance analysis.

Types of Error

There are two main sources of error that an engineer may encounter. These are:

-

Random Error - measurement error or an unconnected factor is affecting results in an uncorrelated manner

-
-

Systematic Error - an unaccounted factor is affecting measurement of the observable in a correlated way

-

There are specific words associated with each type of error. For example, accuracy is used to describe the level of systematic error in a measurement - high accuracy corresponds to low systematic error. Similarly, precision is the term corresponding to random error - high precision is low random error.

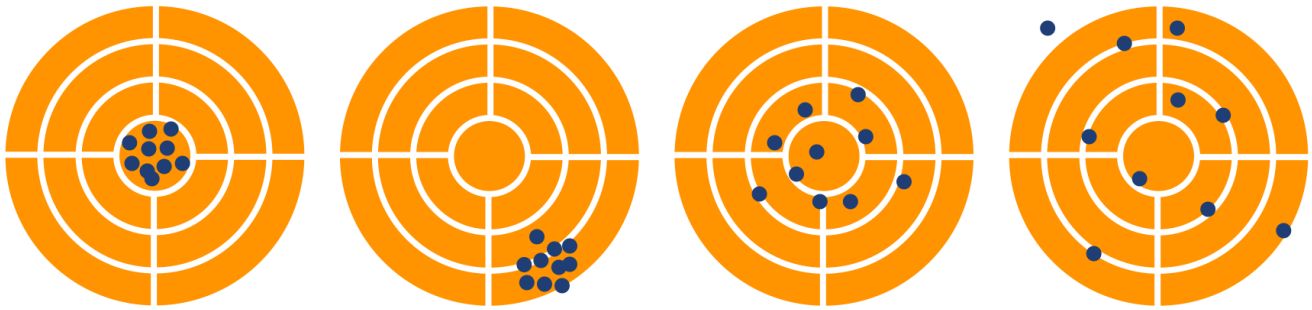


Figure 5-1. Different types of error

The graphics in Figure 5-1 show the effect of these two types of error on a measurement. The extreme left image shows a clustering of measurements around the true result (the “centre of the target”). These measurements have both high precision and high accuracy. The second image has a systematic effect (miscalibrated sights perhaps?) that is causing all the shots to be off-target - so these measurements have high precision, but low accuracy. The third images shows shots basically on target but loosely clustered on the target - so low precision but high accuracy. The final image shows no clear signal, as a result of having both low precision and low accuracy.

Systematic Error

As an example, consider a performance test running against a group of back-end Java web services that send and receive JSON. This type of test is very common when it is problematic to directly use the application front-end for load-testing.

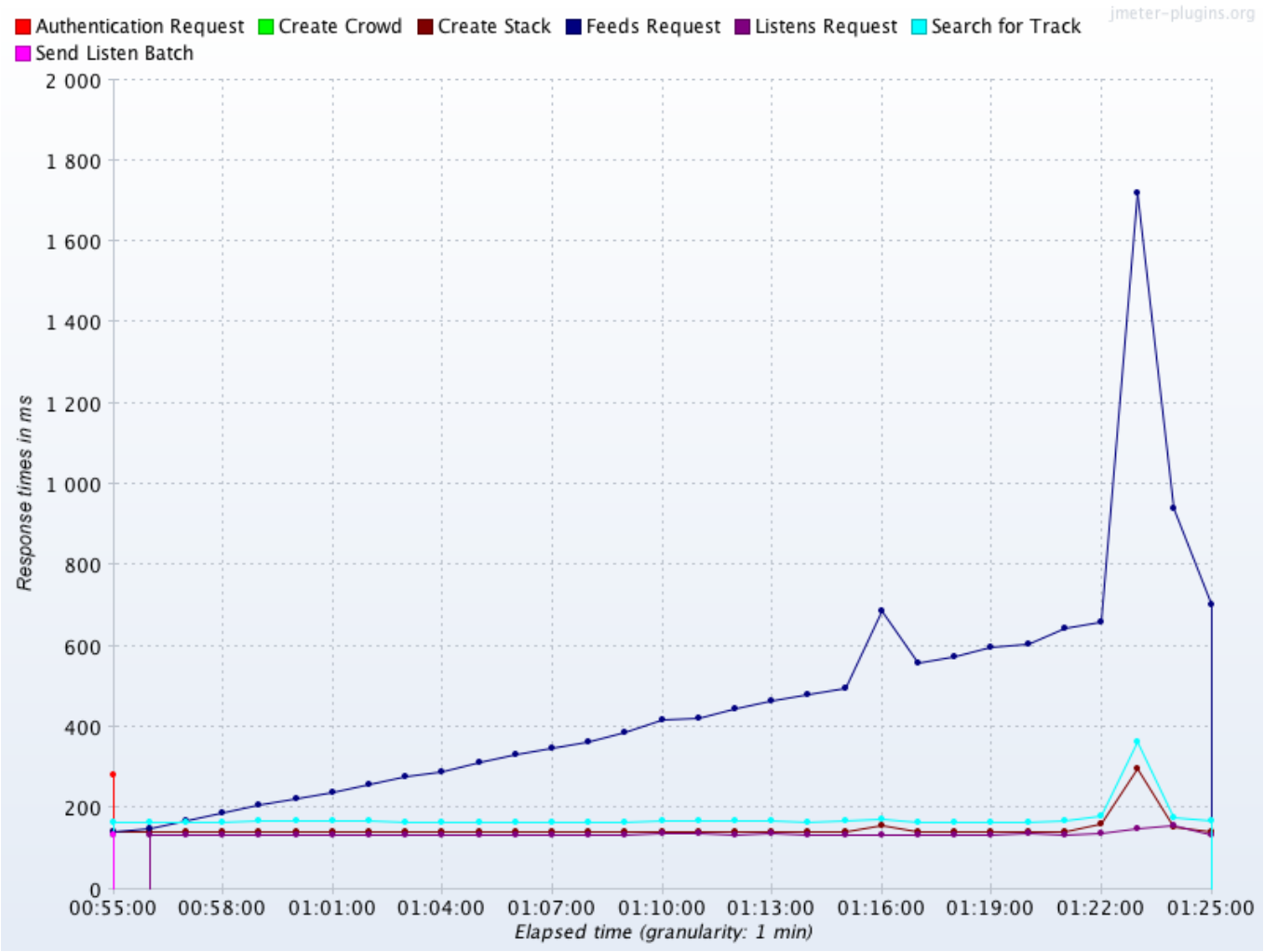


Figure 5-2. Systematic Error

Figure 5-2 was generated from the JP GC extension pack for the Apache JMeter load generation tool). In it, there are actually two systematic effects at work. The first is the linear pattern observed in the top line (the outlier service). This pattern represents slow exhaustion of some limited server resource. This type of pattern is often associated with a memory leak, or some other resource being used and not released by a thread during request handling, and represents a candidate for investigation - it looks like it could be a genuine problem.

Note

Further analysis would be needed to confirm the type of resource that was being affected - we can't just conclude that it's a memory leak.

The second effect that should be noticed is the consistency of the majority of the other services at around the 180ms level. This is suspicious, as the services are

doing very different amounts of work in response to a request. So, why are the results so consistent?

The answer is that whilst the services under test are located in London, this load test was conducted from Mumbai, India. The observed response time includes the intrinsic round-trip network latency from Mumbai to London. This is in the range 120-150ms, and so accounts for the vast majority of the observed time for the services other than the outlier.

This large, systematic effect is drowning out the differences in the actual response time (as the services are actually responding in much less than 120ms). This is an example of a systematic error that does not represent a problem with our application. Instead, this error stems from a problem in our test setup, and the good news is that this artefact completely disappeared (as expected) when the test was rerun from London.

Random Error

The case of random error deserves a mention here, although it is a very well-trodden path.

Note

The reader is assumed to be familiar with basic statistical handling of normally-distributed measurements (mean, mode, standard deviation, etc.) - readers who aren't should consult a basic textbook, such as "The Handbook of Biological Statistics"[1]

Random errors are caused by unknown or unpredictable changes in the environment. In general scientific usage, these changes may occur in either the measuring instrument or in the environment, but for software we assume that our measuring harness is reliable, and so the source of random error can only be the operating environment.

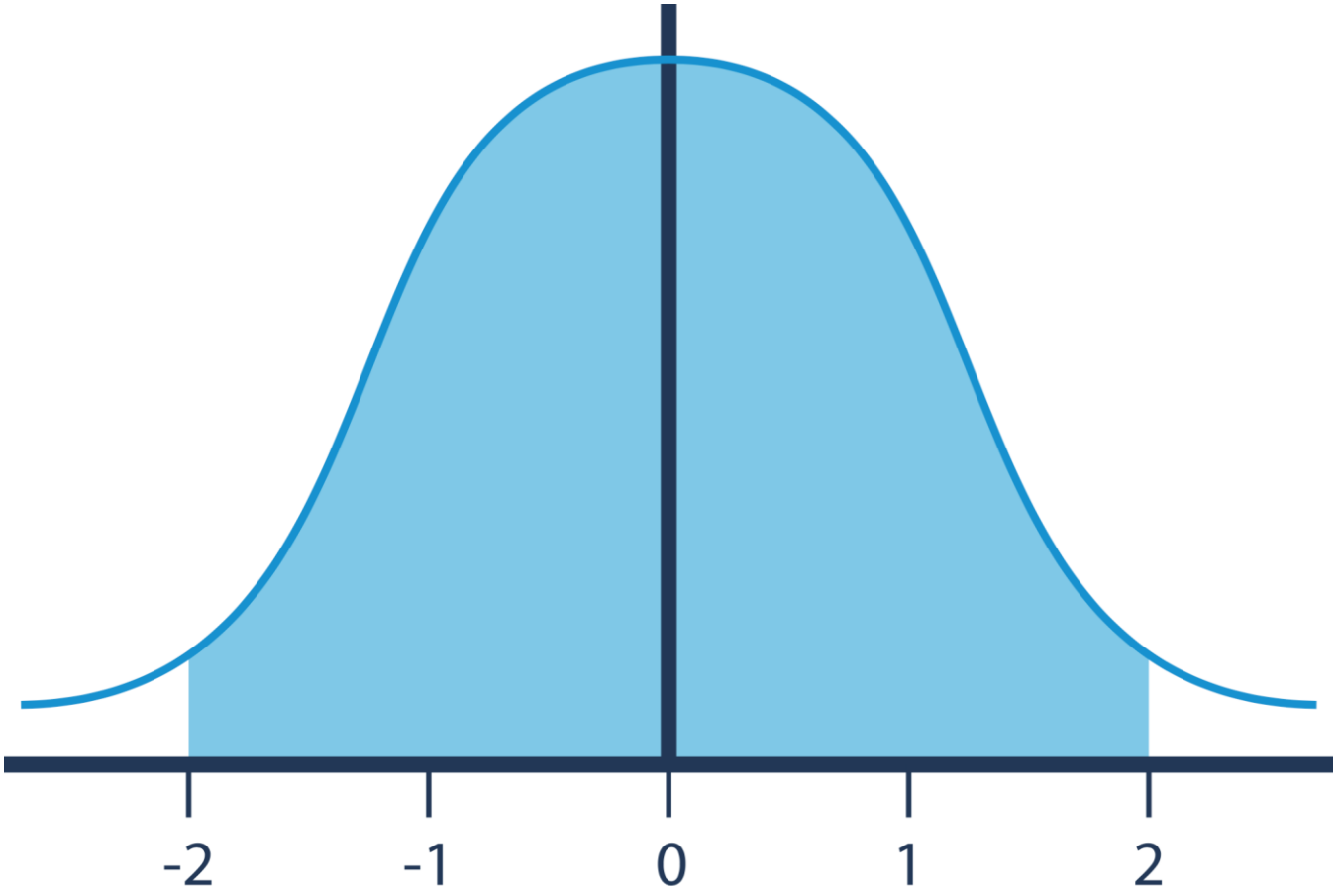


Figure 5-3. A Gaussian distribution (aka normal distribution or bell curve)

Random error is usually considered to obey a Gaussian (aka normal) distribution. A typical one is shown in Figure 5-3. The distribution is a good model for the case where an error is equally likely to make a positive or negative contribution to an observable, but as we will see, this is not a good fit for the JVM.

Spurious Correlation

One of the most famous aphorisms about statistics is "correlation does not imply causation" - that just because two variables appear to behave similarly, does not imply that there is an underlying connection between them.

In the most extreme examples, if a practictioner looks hard enough, then a correlation can be found between entirely unrelated measurements. For example, in Figure 5-4 we can see that consumption of chicken in the US is well correlated with total import of crude oil.[2]

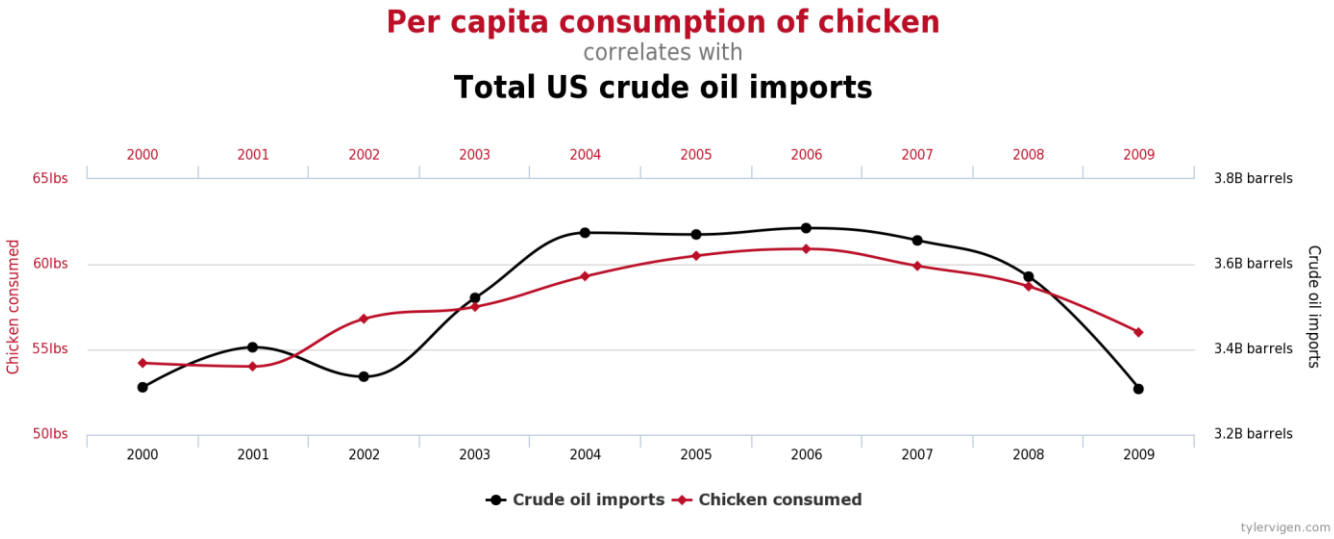


Figure 5-4. A completely spurious correlation

These numbers are clearly not causally related - there is no factor that drives both the import of crude oil and the eating of chicken. However, it isn't the absurd and ridiculous correlations that the practitioner needs to beware.

In Figure 5-5, we see the revenue generated by video arcades correlated to the number of computer science PhDs awarded. It isn't too much of a stretch to imagine a sociological study that claimed a link between these observables, perhaps claiming that "stressed doctoral students were finding relaxation with a few hours of video games". These type of claims are depressingly common, despite no such common factor actually existing.

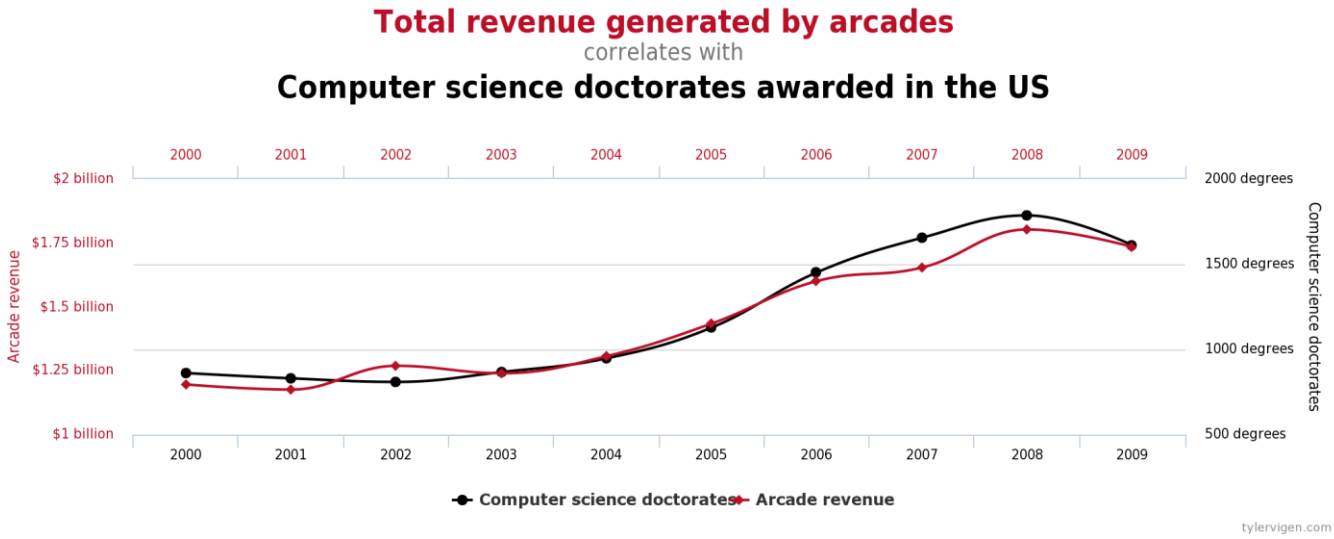


Figure 5-5. A less spurious correlation?

In the realm of the JVM and performance analysis, we need to be especially careful not to attribute a causal relationship between measurements based solely on correlation, and that the connection "seems plausible". This can be seen as one aspect of Feynman's "don't fool yourself" maxim.

We've met some examples of sources of error and mentioned the notorious bear trap of spurious correlation, so let's move on to discuss an aspect of JVM performance measurement that requires some special care and attention to detail.

Non-normal statistics

Statistics that is based on the normal distribution is relatively easy to teach and does not require much mathematical sophistication. For this reason, the standard approach to statistics that is typically taught at pre-college or undergraduate level focuses heavily on the analysis of normally distributed data.

Students are taught to calculate the mean, and the standard deviation (or variance) and sometimes higher moments, such as skew and kurtosis. However, these techniques have a serious drawback, in that they are easy to distort if the distribution has even a few far-flung outlying points.

Note

In Java performance, the outliers represent slow transactions and unhappy customers. We need to pay special attention to these points, and avoid techniques that dilute the importance of outliers.

To consider it from another viewpoint: unless a large number of customers are already complaining, it is unlikely that improving the average response time is the goal. For sure, doing so will improve the experience for everyone, but it is far more usual for a few disgruntled customers to be the cause of a latency tuning exercise. This implies that the outlier events are likely to be of more interest than the experience of the majority who are receiving satisfactory service.

In Figure 5-6 we can see a more realistic curve for the likely distribution of method (or transaction) times. It is clearly not normally distributed.

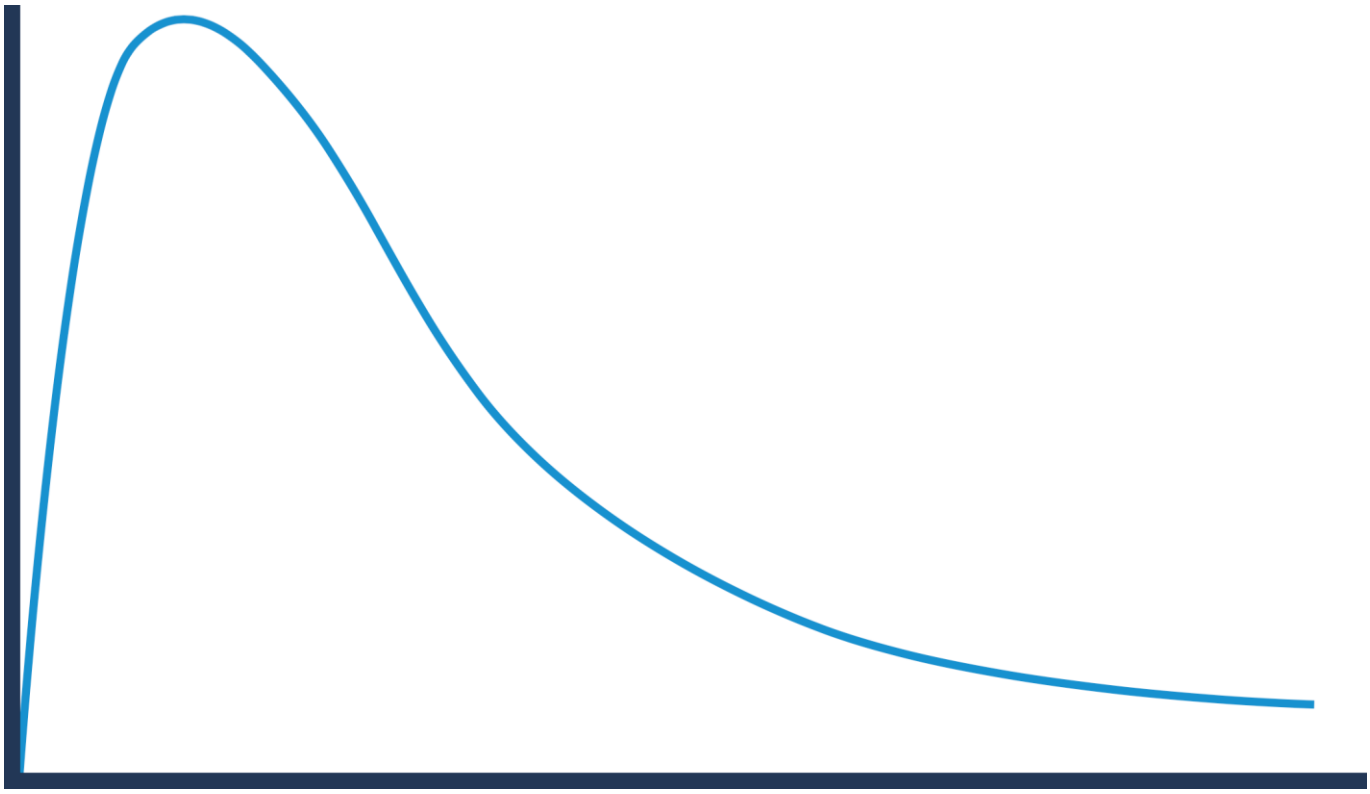


Figure 5-6. A more realistic view of the distribution of transaction times

The shape of the distribution in Figure 5-6 shows something that we know intuitively about the JVM. It has "hot paths" where all the relevant code is already JIT-compiled, there are no GCs, etc. These represent a best-case scenario (albeit a common one) - there simply are no calls that are randomly a bit faster.

This violates a fundamental assumption of Gaussian statistics and forces us to consider distributions that are non-normal.

Tip

For distributions that are non-normal, many "basic rules" of normally distributed statistics are violated. In particular, standard deviation / variance and other higher moments are basically useless.

One technique that is very useful for handling the non-normal, "long tail" distributions that the JVM produces is to use a modified scheme of percentiles. Remember that a distribution is a whole graph - it is a shape of data, and is not represented by a single number.

Instead of computing just the mean, which tries to express the whole distribution in a single result, we can use a sampling of the distribution at intervals. When used for

normally distributed data the samples are usually taken at regular intervals. However, a small adaptation means that the technique can be used for JVM statistics.

50.0% level was 23 ns
90.0% level was 30 ns
99.0% level was 43 ns
99.9% level was 164 ns
99.99% level was 248 ns
99.999% level was 3,458 ns
99.9999% level was 17,463 ns

The modification is to use a sampling that takes into account the long tail distribution, by starting from the mean, then the 90th percentile, and then moving out logarithmically, as shown in Table 5-1. This means that we're sampling according to a pattern that better corresponds to the shape of the data.

The samples show us that whilst the average time was 23ns to execute a getter method, the time, 1 request in 1000 was an order of magnitude worse, and 1 request in 100000 was *two* orders of magnitude worse than average.

Long tail distributions can also referred to by the term "high dynamic range" distributions. The dynamic range of an observable is usually defined as the max recorded value divided by the min (assuming non-zero).

Logarithmic percentiles are a useful simple tool for understanding the long tail. However, for more sophisticated analysis, we can use a public domain library for handling datasets with high dynamic range. The library is called HdrHistogram and is available from: https://github.com/HdrHistogram/HdrHistogram - it was originally created by Gil Tene (Azul Systems) with additional work by Mike Barker, Darach Ennis and Coda Hale.

Note

A histogram is a way of summarizing data by using a finite set of ranges (called "buckets") and displaying how often data falls into each bucket.

HdrHistogram is also available on Maven Central. At time of writing, the current version is 2.1.9, and it can be added to your projects by adding this dependency stanza to pom.xml:

```
<dependency>
    <groupId>org.hdrhistogram</groupId>
    <artifactId>HdrHistogram</artifactId>
    <version>2.1.9</version></dependency>
```

Let's look at a simple example for HdrHistogram. This example takes in a file of numbers and computes the HdrHistogram for the difference between successive results.

```
public class BenchmarkWithHdrHistogram {
    private static final long NORMALIZER = 1_000_000;

    private static final Histogram HISTOGRAM
            = new Histogram(TimeUnit.MINUTES.toMicros(1), 2);

    public static void main(String[] args) throws Exception {
        final List<String> values = Files.readAllLines(Paths.get(args[0]));
        double last = 0;
        for (final String tVal : values) {
            double parsed = Double.parseDouble(tVal);
            double gcInterval = parsed - last;
            last = parsed;
            HISTOGRAM.recordValue((long)(gcInterval * NORMALIZER));
        }
        HISTOGRAM.outputPercentileDistribution(System.out, 1000.0);
    }}
```

Here's what the histogram plotter produces for a sample GC log - it shows the times between successive garbage collections. As we'll see in Chapter 7 and 8, garbage collections do not occur at regular intervals, and understanding how the distribution of how close together GCs occur could be useful.

```
       Value     Percentile TotalCount 1/(1-Percentile)

       14.02 0.000000000000          1           1.00
     1245.18 0.100000000000         37           1.11
     1949.70 0.200000000000         82           1.25
     1966.08 0.300000000000        126           1.43
     1982.46 0.400000000000        157           1.67

...

    28180.48 0.996484375000        368         284.44
    28180.48 0.996875000000        368         320.00
    28180.48 0.997265625000        368         365.71
    36438.02 0.997656250000        369         426.67
    36438.02 1.000000000000        369
#[Mean    =      2715.12, StdDeviation   =      2875.87]
#[Max     =     36438.02, Total count    =          369]
#[Buckets =           19, SubBuckets     =          256]
```

This is rather hard to analyse from the raw output of the formatter, but fortunately, the HdrHistogram project includes an online formatter that can be used to generate visual histograms from the raw output. It can be found at: http://hdrhistogram.github.io/HdrHistogram/plotFiles.html

For this example, it produces output like that shown in Figure 5-7.

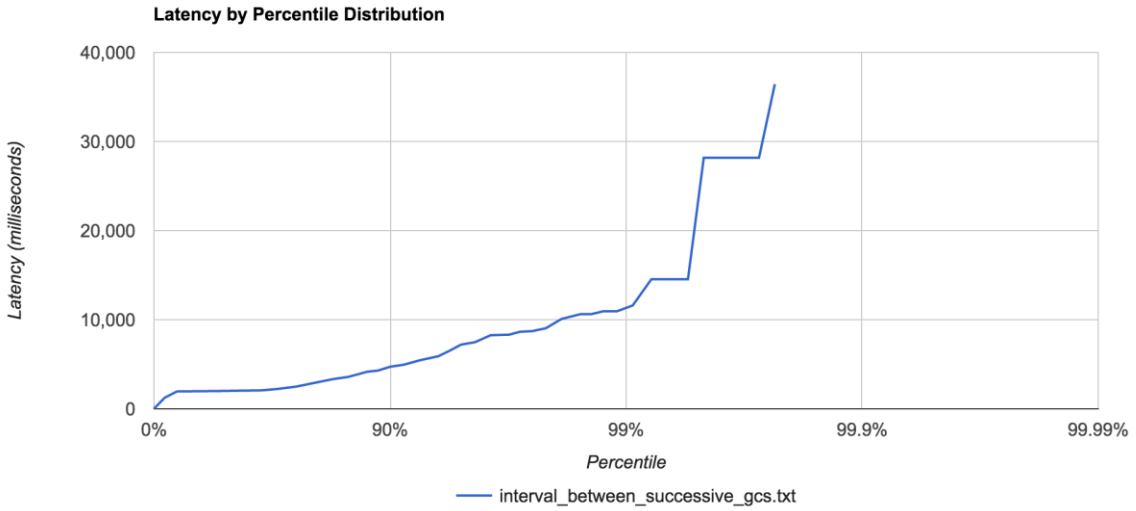


Figure 5-7. Example HdrHistogram visualisation

For many observables that we wish to measure in Java performance tuning, the statistics of them is often highly non-normal, and the HdrHistogram can be a very useful tool in helping to understand and visualise the shape of the data.

Interpretation of statistics

Empirical data and observed results do not exist in a vacuum and it is quite common that one of the hardest jobs lies in interpreting the results that we obtain from measuring our applications.

> No matter what they tell you, it's always a people problem.
>
> Gerald Weinberg

In Figure 5-8 we show an example memory allocation rate for a real Java application. This example is for a reasonably well-performing application. The screenshot comes from the Censum garbage collection analyser, which we will meet in Chapter 8.

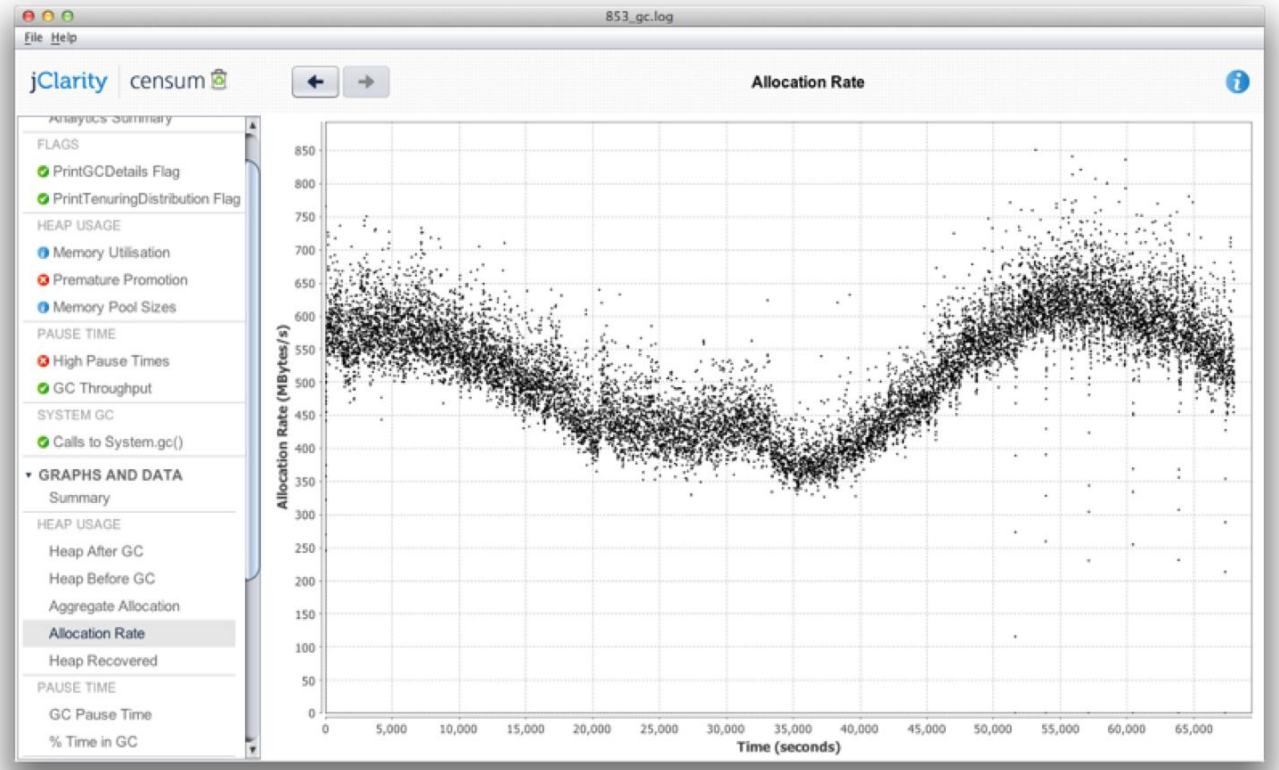


Figure 5-8. Example allocation rate

The interpretation of the allocation data is relatively straightforward, as there is a clear signal present - over the time period covered (almost a day), allocation rates were basically stable between 350 and 700 MB being allocated per second. There is a downward trend starting approximately 5 hours after the JVM started up, and a clear minimum between 9 and 10 hours, after which the allocation rate starts to rise again.

These type of trends in observables are very common, as the allocation rate will usually reflect the amount of work an application is actually doing, and this will vary widely depending on the time of day. However, when interpreting real observables, the picture can rapidly become more complicated. This can lead to what is sometimes called the "Hat-Elephant" problem, after a passage in "The Little Prince" by Antoine de Saint-Exupéry.

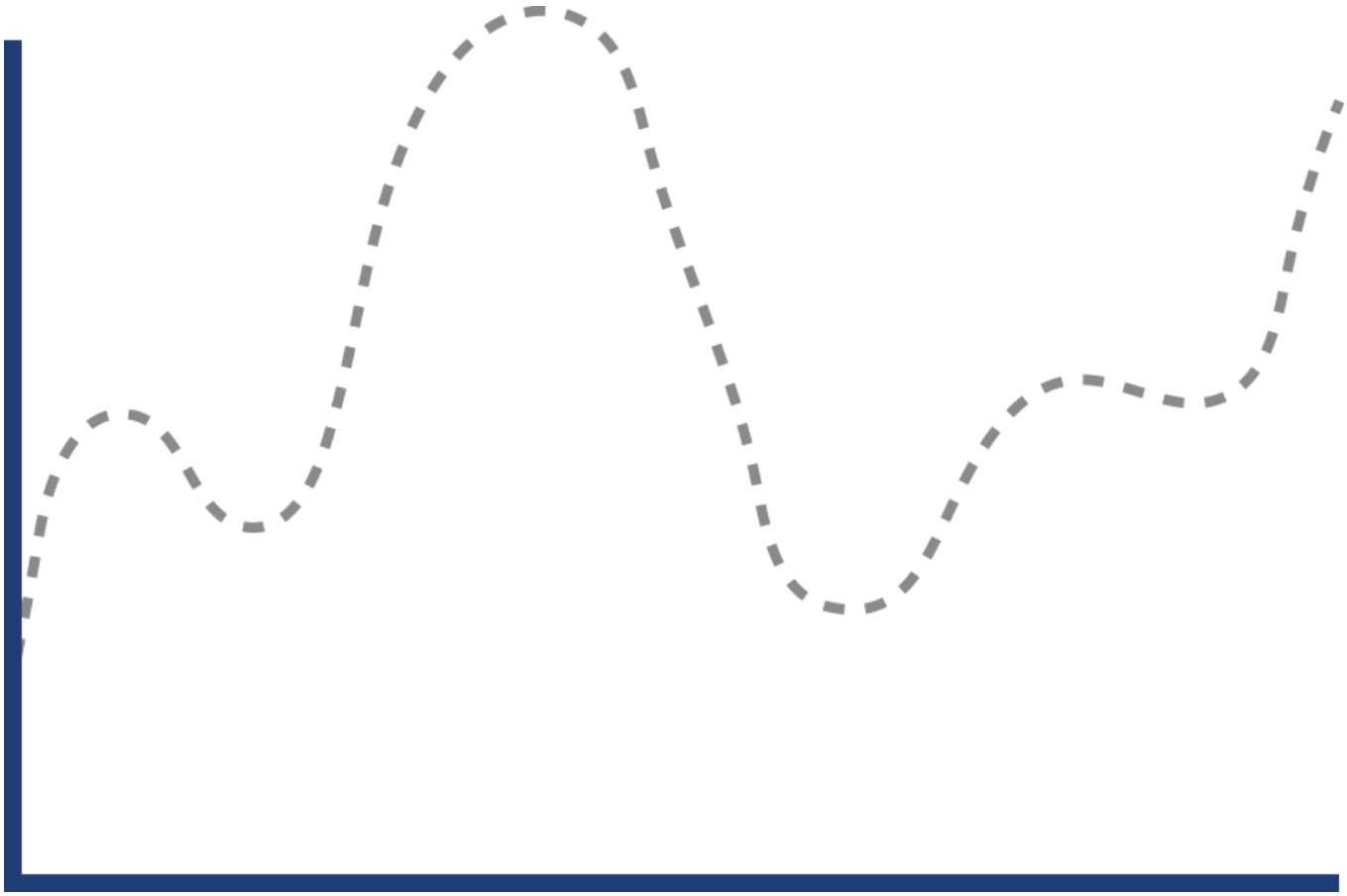


Figure 5-9. Hat or elephant eaten by a boa?

The problem is illustrated by Figure 5-9 - all we can initially see is a complex histogram of HTTP request-response times. However, just like the narrator of the book, if we can imagine or analyse a bit more, we can see that the complex picture is actually made up of several fairly simple pieces.

The key to decoding the response histogram is to realise that "web application responses" is a very general category - including sucessful requests (so-called 2xx responses), client error (4xx - including the ubiquitous 404 error) and server errors (5xx, especially 500 Internal Server Error).

Each type of response has a different characteristic distribution for response times - if a client makes a request for a URL that has no mapping (a 404), then the web server can immediately reply with a response - meaning that the histogram for only client error responses looks more like Figure 5-10.

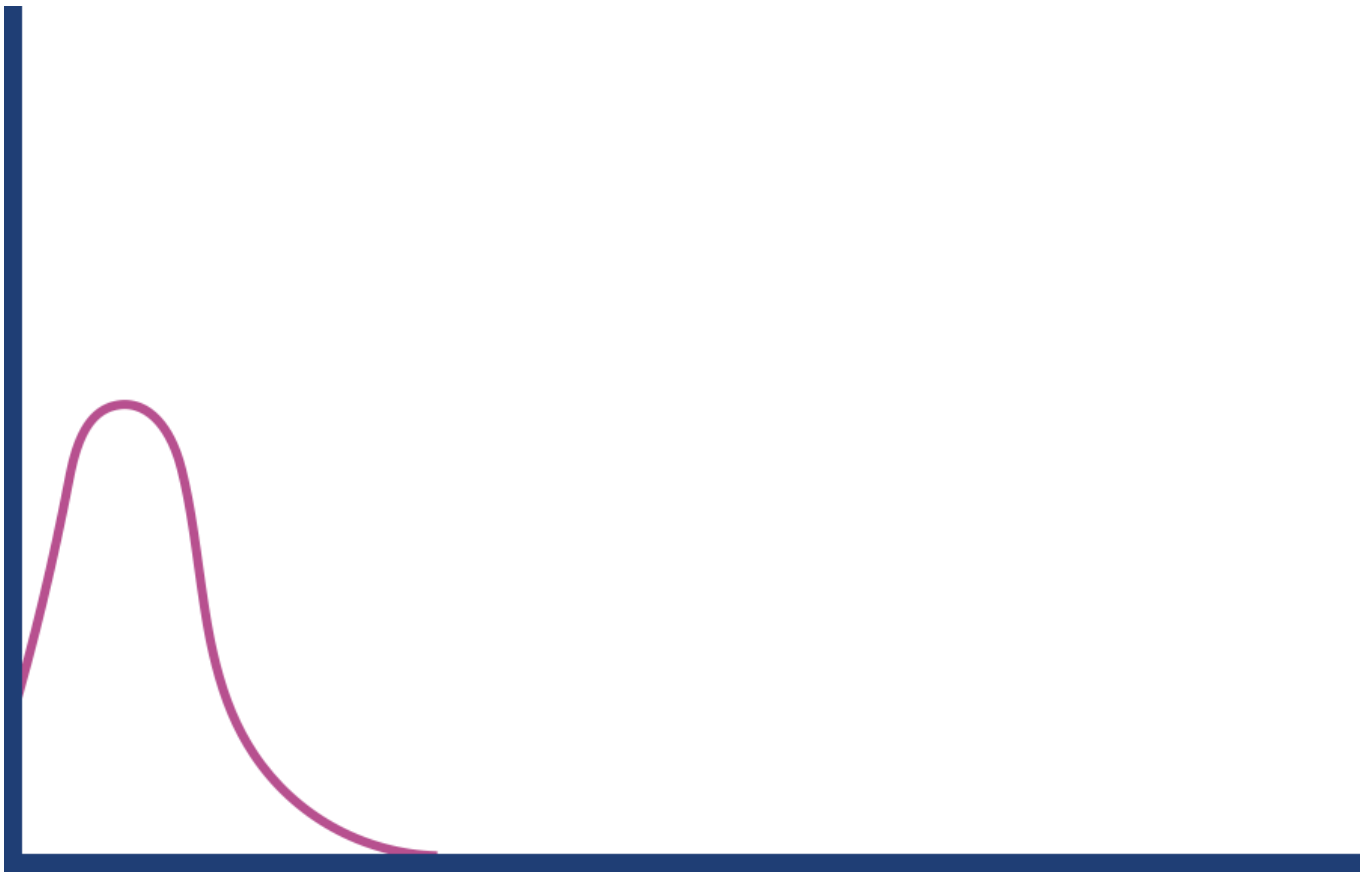


Figure 5-10. Client Errors

By contrast, server errors often occur after a large amount of processing time has been expended (for example due to backend resources being under stress, or timing out). So, the histogram for server error responses might look like Figure 5-11.

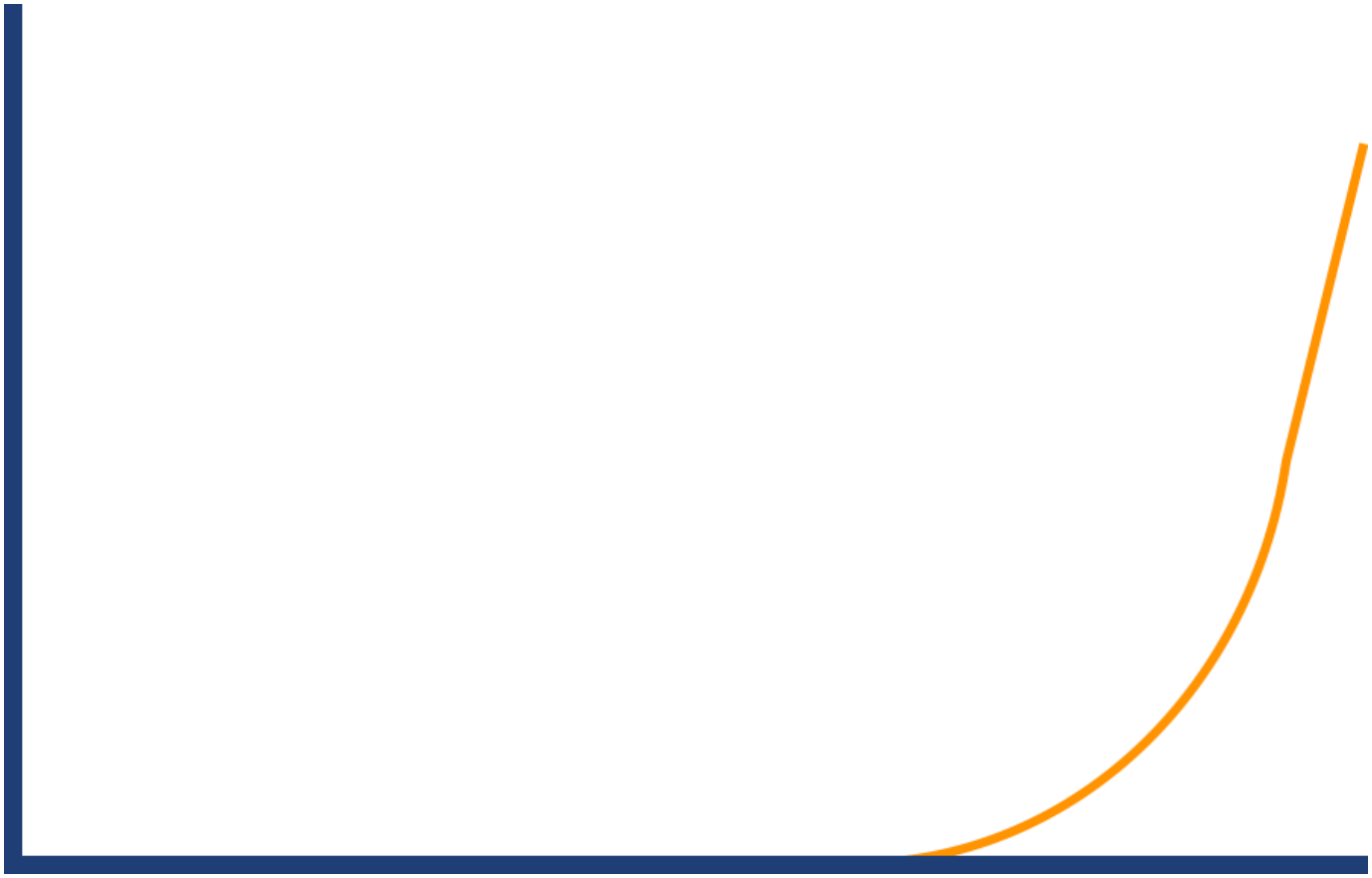


Figure 5-11. Server Errors

The successful requests will have a long tail distribution, but in reality we may expect the response distribution to be “multimodal” - and have several local maxima. An example is shown in Figure 5-12, and represents the possibility that there could be two common execution paths through the application with quite different response times.

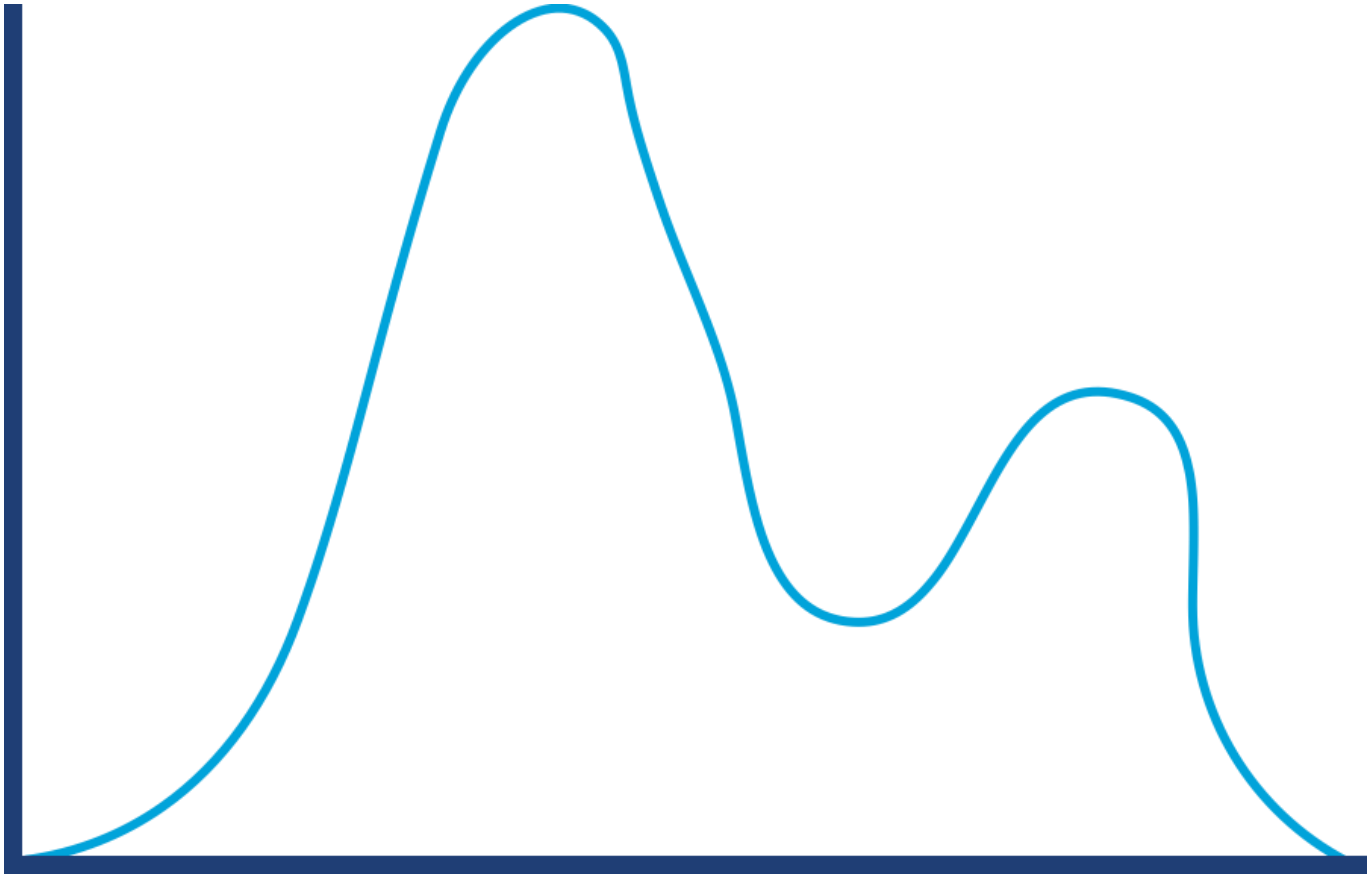


Figure 5-12. Successful Requests

The result of combining these different types of responses into a single graph results in the structure shown in Figure 5-13. We have rederived our original “hat” shape from the separate histograms.

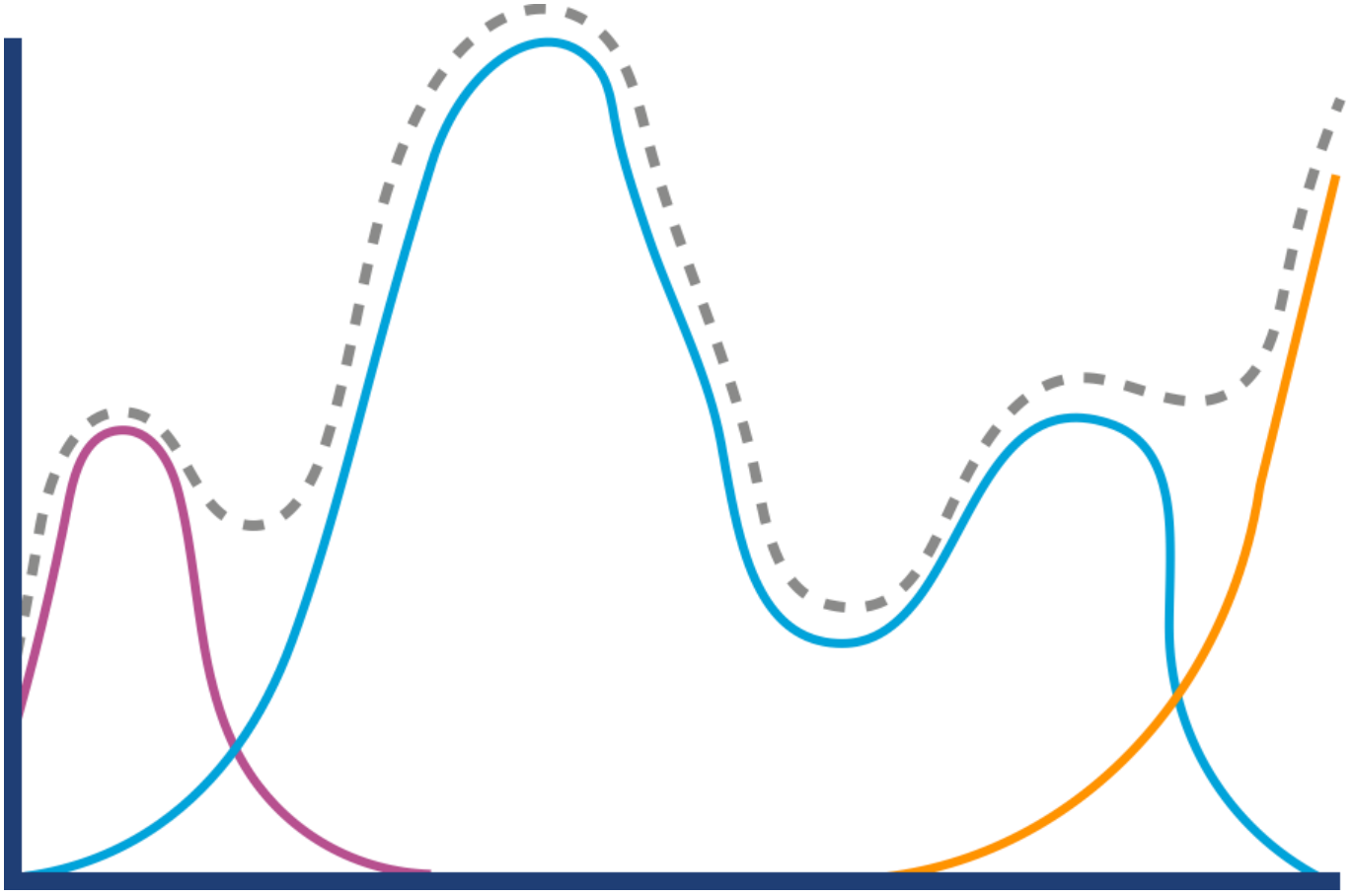


Figure 5-13. Hat or elephant revisited

The concept of breaking down a general observable into more meaningful sub-populations is a very useful one, and shows that we need to make sure that we understand our data and domain well enough before trying to infer conclusions from our results. We may well want to further break down our data into smaller sets - for example the successful requests may have very different distributions for requests that are predominantly read, as opposed to requests that are updates or uploads.

The engineering team at Paypal have written extensively about their use of statistics and analysis - they have a blog at https://www.paypal-engineering.com/ that contains excellent resources - the piece "Statistics for Software"[3] by Mahmoud Hashemi is a great introduction to their methodologies, and includes a version of the Hat-Elephant problem as we discussed above.

Summary

The discussion of microbenchmarks is the closest that Java performance comes to a "Dark Art". Whilst this characterisation is evocative, it is not wholly deserved. It

is still an engineering discipline undertaken by working developers. However, microbenchmarks should be used with caution:

- Do not microbenchmark unless you know you are a known use case for it.
- If you must microbenchmark, use JMH.
- Discuss your results as publicly as you can, and in the company of your peers.
- Be prepared to be wrong a lot, and have your thinking challenged repeatedly.

One of the positive aspects of working with microbenchmarks is that it exposes the highly dynamic behavior and non-normal distributions that are produced by low-level subsystems. This, in turn, leads to a better understanding and mental models of the complexities of the JVM.

In the next chapter, we will move from the low-level approach to the far more tractable case of top-down analysis, and look at the available tools for high-level performance monitoring and analysis.

[1] McDonald, J.H. 2014. Handbook of Biological Statistics, 3rd ed. Sparky House Publishing, http://biostathandbook.com/

[2] The spurious correlations in this section come from Tyler Viglen's site: http://tylervigen.com/spurious-correlations and are reused here with permission under a Creative Commons License. If you enjoyed them, Tyler has a book with many more amusing examples, available from the link.

[3] https://www.paypal-engineering.com/2016/04/11/statistics-for-software/

Chapter 6. Understanding Garbage Collection

The Java environment has several iconic or defining features, and garbage collection is one of the most immediately recognisable. However, when the platform was first released, there was considerable hostility to GC. This was fueled by the fact that Java deliberately provided no language-level way to control the behavior of the collector (and continues not to, even in modern versions).[1]

This meant that in the early days, there was a certain amount of frustration over the performance of Java's GC, and this fed into perceptions of the platform as a whole.

However, the early vision of mandatory, non-user-controllable GC has been more than vindicated, and these days very few application developers would attempt to defend the opinion that memory should be managed by hand. Even modern takes on systems programming languages (e.g. Go and Rust) regard memory management as the proper domain of the compiler and runtime rather than the programmer (in anything other than exceptional circumstances).

The essence of Java's garbage collection is that rather than requiring the programmer to understand the precise lifetime of every object in the system, that the runtime should keep track of objects on the programmers behalf and automatically get rid of objects that are no longer required. The automatically reclaimed memory can then be wiped and reused.

There are two fundamental rules of garbage collection that all implementations are subject to:

1.

Algorithms must collect all garbage

2.

3.

No live object must ever be collected

4.

Of these two rules, the second is by far the most important. Collecting a live object could lead to segmentation faults, or (even worse) silent corruption of program data. Java's GC algorithms need to be sure that they will never collect an object the program is still using.

The idea of the programmer surrendering some low-level control in exchange for not having to account for every low-level detail by hand is the essence of Java's managed approach and expresses James Gosling's conception of Java as a blue-collar language for getting things done.

In this chapter, we will meet some of the basic theory that underpins Java garbage collection, and explain why it is one of the hardest parts of the platform to fully understand and to control. We will also introduce the basic features of Hotspot's runtime system, including such details as how Hotspot represents objects in the heap at runtime.

Towards the end of the chapter, we will introduce the simplest of Hotspot's production collectors, the parallel collectors, and explain some of the details that make them so useful for many workloads.

Introducing Mark & Sweep

Most Java programmers, if pressed, can recall that Java's GC relies on an algorithm called *mark and sweep*, but most also struggle to recall any details as to how the process actually operates.

In this section we will introduce a basic form of the algorithm and show how it can be used to reclaim memory automatically. This is a deliberately simplified form of the algorithm and is only intended to introduce a few basic concepts - it is not representative of how production JVMs actually carry out GC.

This introductory form of the mark-and-sweep algorithm uses an allocated object list to hold a pointer to each object that has been allocated, but not yet reclaimed. The overall GC algorithm can then be expressed as:

1.

Loop through the allocated list clearing the mark bit

2.

3.

Starting from the GC roots, find the live objects

4.

5.

Set mark bit on each object reached

6.

7.

Loop through allocated list

8.

1.

For each object whose mark bit hasn’t been set

2.

3.

Reclaim the memory in heap and place it back on the free list

4.

5.

Remove object from allocated list

6.

The live objects are usually located depth-first, and the resulting graph of objects is called “the live object graph”. It is sometimes also called the the “transitive closure of reachable objects”, and an example can be seen in Figure 7-1

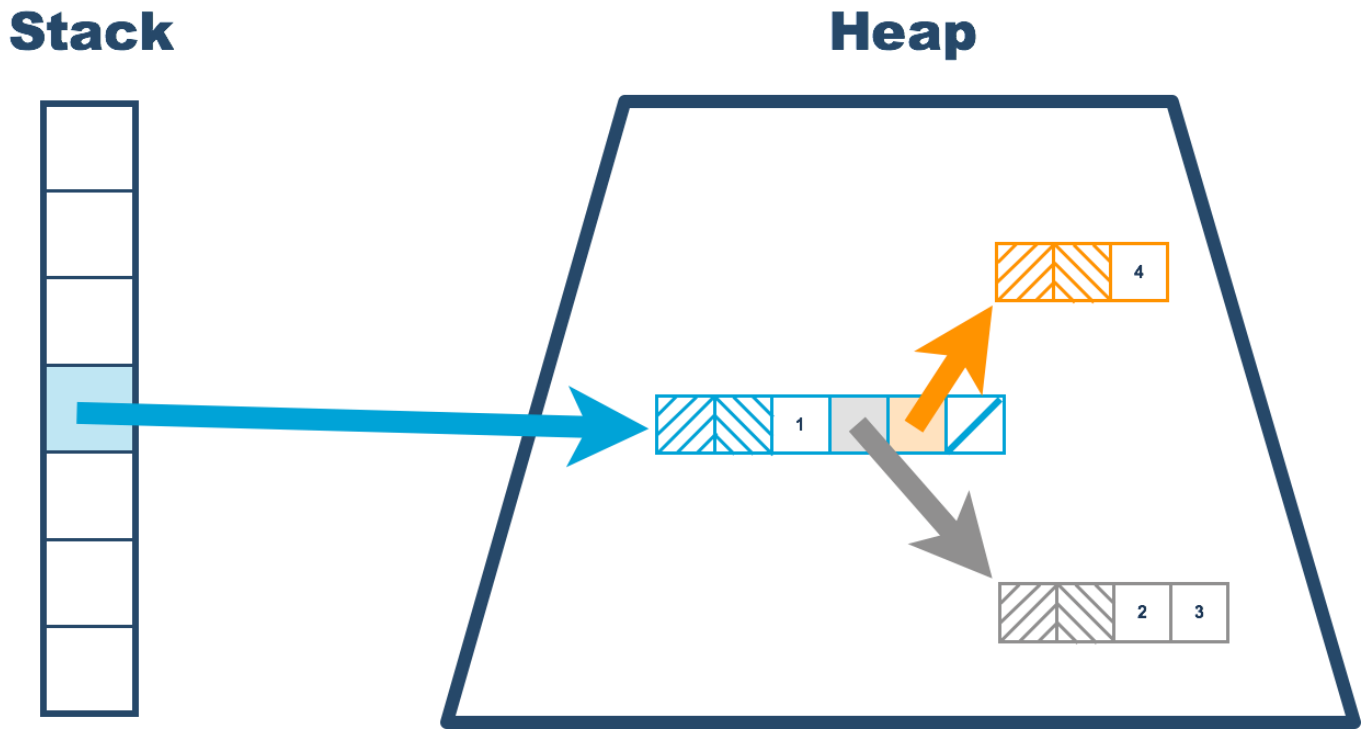


Figure 6-1. Simple view of memory layout

The state of the heap can be hard to visualise, but fortunately there are some simple tools to help us. One of the simplest is the `jmap -histo` command-line tool. This shows the number of bytes allocated per type, and the number of instances that are collectively responsible for that memory usage.

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         20839       14983608  [B
   2:        118743       12370760  [C
   3:         14528        9385360  [I
   4:           282        6461584  [D
   5:        115231        3687392  java.util.HashMap$Node
   6:        102237        2453688  java.lang.String
   7:         68388        2188416  java.util.Hashtable$Entry
   8:          8708        1764328  [Ljava.util.HashMap$Node;
   9:         39047        1561880  jdk.nashorn.internal.runtime.CompiledFunction
  10:         23688        1516032
com.mysql.jdbc.ConnectionPropertiesImpl$BooleanConnectionProperty
  11:         24217        1356152  jdk.nashorn.internal.runtime.ScriptFunction
  12:         27344        1301896  [Ljava.lang.Object;
  13:         10040        1107896  java.lang.Class
  14:         44090        1058160  java.util.LinkedList$Node
  15:         29375         940000  java.util.LinkedList
```

```
  16:         25944         830208
jdk.nashorn.internal.runtime.FinalScriptFunctionData
  17:            20         655680  [Lscala.concurrent.forkjoin.ForkJoinTask;
  18:         19943         638176  java.util.concurrent.ConcurrentHashMap$Node
  19:           730         614744  [Ljava.util.Hashtable$Entry;
  20:         24022         578560  [Ljava.lang.Class;
```

There is also a GUI tool available, the memory sampling tab of VisualVM as seen in Chapter 6. The VisualGC plugin to VisualVM also provides a real-time view of how the heap is changing. In general the moment-to-moment view of the heap is not sufficient for accurate analysis however and instead we should use GC logs for better insight - this will be a major theme of Chapter 9.

Garbage Collection Glossary

The jargon used to describe GC algorithms is sometimes a bit confusing (and the meaning of some of the terms has changed over time). For the sake of definiteness, we include a basic glossary of how we use specific terms:

Stop-The-World (STW)

The GC cycle requires all application threads to be paused whilst garbage is collected. This prevents application code from invalidating the GC threads view of the state of the heap. This is the usual case for most simple GC algorithms.

Concurrent

GC threads can run whilst application threads are running. This is very, very difficult to achieve, and very expensive in terms of computation expended. Virtually no algorithms are truly concurrent. Instead, complex tricks are used to give most of the benefits of concurrent collection. In Section 8.3 we'll meet Hotspot's "Concurrent Mark and Sweep" collector (CMS) that is usually thought of as concurrent, but is perhaps best described as a "mostly-concurrent" collector.

Parallel

Multiple threads are used to execute garbage collection.

Exact

An exact GC scheme has enough type information about the state of the heap to ensure that all garbage can be collected on a single cycle. More loosely, an exact scheme has the property that it can always tell the difference between an int and a pointer.

Conservative

A conservative scheme lacks the information of an exact scheme. As a result, conservative schemes frequently fritter away resources and are typically far less efficient as a result of their fundamental ignorance of the type system they purport to represent.

Moving

In a moving collector, objects can be relocated in memory. This means that they do not have stable addresses. Environments that provide access to raw pointers (e.g. C++) are not a natural fit for moving collectors.

Compacting

At the end of the collection cycle, allocated memory (i.e. surviving objects) is arranged as a single contiguous region (usually at the start of the region) and there is a pointer indicating the start of empty space that is available for objects to be written into. A compacting collector will avoid memory fragmentation.

Evacuating

At the end of the collection cycle, the collected region is totally empty, and all live objects have been moved (evacuated) to another region of memory

In most other languages and environments the same terms are used. For example, the JavaScript runtime of the Firefox web browser (SpiderMonkey) also makes use of garbage collection, and in recent years has been adding features (e.g. exactness and compaction) that are already present in GC implementations in Java.

Introducing the HotSpot runtime

In addition to the general GC terminology, HotSpot introduces terms that are more specific to the implementation. To obtain a full understanding of how garbage collection works on this JVM, we need to get to grips with some of the details of HotSpot's internals.

For what follows it will be very helpful to remember that Java has only two sorts of value:

- Primitive types (`byte`, `int`, etc.)
-

- Object references

-

Many Java programmers loosely talk about *objects*, but for our purposes it is important to remember that, unlike C++, Java has no general address dereference mechanism and can only use an *offset operator* (the . operator) to access fields and call methods on *object references*. It should also be kept in mind that Java's method call semantics are purely call-by-value, although for object references this means that the value that is copied is the address of the object in the heap.

Representing objects at runtime

HotSpot represents Java objects at runtime via a structure called an oop. This is short for *Ordinary Object Pointer*, and is a genuine pointer in the C sense. These pointers can be placed in local variables of reference type where they point from the stack frame of the Java method into the memory area comprising the Java heap.

Note

There are several different data structures that comprise the family of oops, and the sort that represent instances of a Java class are called *instanceOops*.

The memory layout of an instanceOop is as shown in Figure 7-2. The layout starts with two machine words of header present on every object. The *mark word* is the first of these, and is a pointer that points at instance-specific metadata. Next is the *klass word*, which points at class-wide metadata, stored in a structure called a *klassOop*.

= FIXME

Figure 6-2. Simple view of memory layout

The klassOops are held outside of the main part of the Java heap (but not outside the C heap of the JVM process). The "k" at the start is used to help disambiguate the klassOop from an instanceOop representing the Java `Class<?>` object - they are not the same thing.

In Figure 7-3 we can see the difference - fundamentally the klassOop contains the virtual function table (vtable) for the class, whereas the Class object contains an array of references to Method objects for use in reflective invocation. We will have more to say on this subject in Chapter 10 when we discuss JIT compilation.

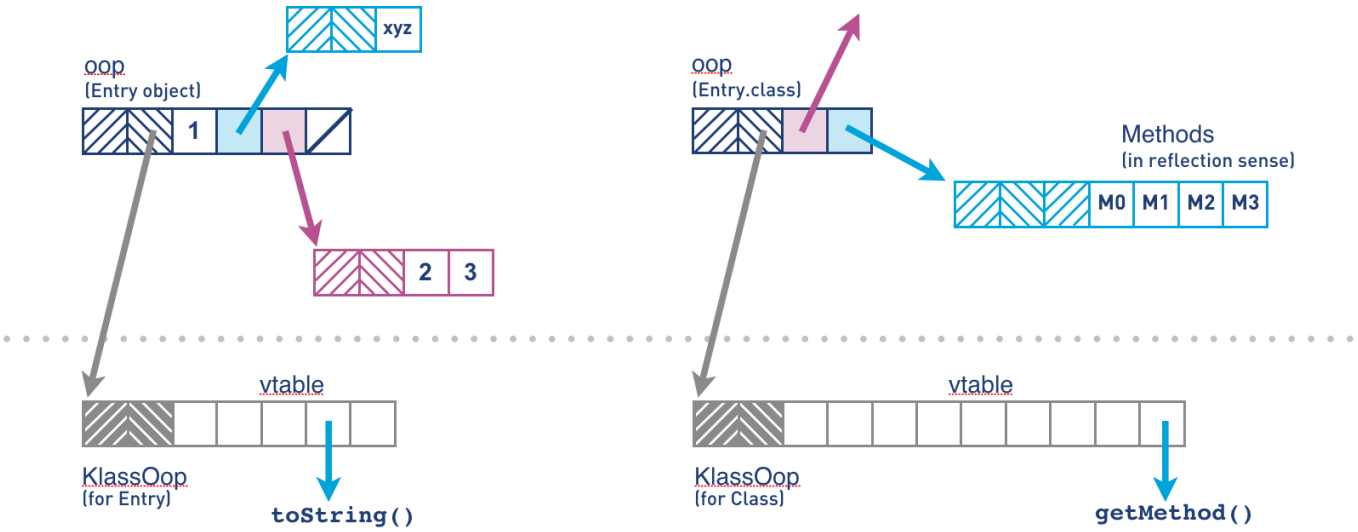


Figure 6-3. klassOops and Class objects

Oops are usually a machine word - so 32 bits on a legacy 32-bit machine, and 64 bits on a modern processor. However, this has the potential to waste a possibly significant amount of memory. To help mitigate this, Hotspot provides a technique called compressed oops. If the option:

-XX:+UseCompressedOops

is set (and it is the default for 64-bit heaps from Java 7 and upwards), then the following oops in the heap will be compressed:

- Klass word of every object in the heap
- Instance fields of reference type
- Every element of an array of objects

This means that, in general, a Hotspot object header consists of:

- Mark word at full native size
- Klass word (possibly compressed)
- Length word if the object is an array - always 32 bits

-

A 32-bit gap if required by alignment rules)

-

The instance fields of the object then follow immediately after the header. For klassOops, the vtable of methods follows directly after the klass word.

In the past, some extremely latency sensitive applications could occasionally see improvements by switching off the compressed oops feature - at the cost of increased heap size (typically an increase of 10 - 50%). However the class of applications for which this would yield measurable performance benefits is very small, and for almost modern applications this would be a classic example of the Fiddle With Switches antipattern.

Note

The klass word of an instanceOop points into the area of memory called *metaspace* (in Java 7 and before the metadata was stored in an area called *PermGen* which was a part of the Java heap).

As we remember from basic Java, arrays are objects. This means that the JVM's arrays are represented as oops as well. This is why arrays have have a third word of metadata as well as the usual mark and klass words. This third word is the array's length - which also explains why array indices in Java are limited to 32-bit values.

Note

The use of additional metadata to carry an array's length alleviates a whole class of problems present in C and C++ where not knowing the length of the array means that additional parameters must be passed to functions.

The managed environment of the JVM does not allow a Java reference to point anywhere but at an instanceOop (or `null`). This means that at a low level:

-

A Java value is a bit pattern corresponding to either a primitive value or to the address of an instanceOop (a reference).

-
-

Any Java reference, considered as a pointer, refers to an address in the main part of the Java heap.

-
-

Addresses that are the targets of Java references contain a mark word followed by a klass word as the next machine word.

-
-

A klassOop and an instance of `Class<?>` are different (as the former lives in the metadata area of the heap), and a klassOop cannot be placed into a Java variable.

-

HotSpot defines a hierarchy of oops in .hpp files that are kept in: `hotspot/src/share/vm/oops` in the OpenJDK 8 source tree. The overall inheritance hierarchy for oops looks like this:

```
oop (abstract base)
 instanceOop (instance objects)
 methodOop (representations of methods)
 arrayOop (array abstract base)
 symbolOop (internal symbol / string class)
 klassOop (klass Header)
 markOop
```

This use of oop structures to represent objects at runtime, with one pointer to house class-level metadata and another to house instance metadata is not particularly unusual. Many other JVMs and other execution environments use a related mechanism. For example, Apple’s iOS uses a very similar scheme for representing objects.

GC Roots and Arenas

Articles and blog posts about HotSpot frequently refer to *GC roots*. These are “anchor points” for memory, essentially known pointers that originate from outside a memory pool of interest and point into it. They are *external* pointers as apposed to *internal* pointers, which originate inside the memory pool and point to another memory location within the memory pool.

We saw an example of a GC root in Figure 7-1. However, as we will see there are other sorts of GC root, including:

-

Stack frames

-
-

JNI

-
-

Registers (for the hoisted variable case)

-
-

Code roots (from the JVM code cache)

-
-

Globals

-
-

Class metadata from loaded classes

-

If this definition seems rather complex, then the simplest example of a GC root is a local variable of reference type which will always point to an object in the heap (provided it is not null).

The HotSpot garbage collector works in terms of areas of memory called *arenas*. This is a very low-level mechanism, and Java developers usually don't need to consider the operation of the memory system in such detail. However, performance specialists sonetimes need to delve deeper into the internals of the JVM, and so a familiarity with the concepts and terms used in the literature is helpful.

One important fact to remember is that HotSpot does not use system calls to manage the Java heap. Instead, as we discussed in Section 3.6, HotSpot manages the heap size from

userspace code, and so we can use simple observables to determine whether the GC subsystem is causing some types of performance problems.

In the next section, we'll take a closer look at two of the most important characteristics that drive the garbage collection behavior of any Java or JVM workload. A good understanding of these characteristics is essential for any developer who wants to really grasp the factors that drive Java GC (which is one of the key overall drivers for Java performance).

Allocation and lifetime

There are two primary drivers of the garbage collection behavior of a Java application:

- Allocation rate
- Object lifetime

The allocation rate is the amount of memory used by newly created objects over some time period (usually measured in MB/s). This is not directly recorded by the JVM, but is a relatively easy observable to estimate, and tools such as Censum can determine it precisely.

By contrast, the object lifetime is normally a lot harder to measure (or even estimate). In fact, one of the major arguments against using manual memory management is the complexity involved in truly understanding object lifetimes for a real application. As a result, object lifetime is if anything even more fundamental than allocation rate.

Tip

Garbage collection can also be thought of as "memory reclamation and reuse". The ability to use the same physical piece of memory over and over again, because objects are short-lived is a key assumption of garbage collection techniques.

The idea that objects are created, exist for a time, and then the memory used to store their state can be reclaimed is essential - without it garbage collection would not

work at all. As we will see in Chapter 8, there are a number of different tradeoffs that garbage collectors must balance - and some of the most important of these tradeoffs are driven by lifetime and allocation concerns.

Weak Generational Hypothesis

One key part of the JVM's memory management relies upon an observed runtime effect of software systems - the Weak Generational Hypothesis.

> The distribution of object lifetimes on the JVM and similar software systems is bimodal - with the vast majority of objects being very short-lived and a secondary population having a much longer life expectancy
>
> Weak Generational Hypothesis

This hypothesis, which is really an experimentally observed rule of thumb about the behavior of object oriented workloads, leads to an obvious conclusion. The conclusion is that garbage collected heaps should be structured in such a way to allow short-lived objects to be easily and quickly collected, and ideally for long-lived objects to be separated from short-lived objects.

HotSpot uses several mechanisms to try to take advantage of the Weak Generational Hypothesis.

-

It tracks the "generational count" of each object (the number of garbage collections that the object has survived so far)

-
-

With the exception of large objects, it creates new object in the "Eden" space (also called the "Nursery") and expects to move surviving objects

-
-

It maintains a separate area of memory (the "Old Generation") to hold objects that are deemed to have survived long enough that they are likely to be long-lived

-

This approach leads to a view shown in simplified form in Figure 7-4 whereby objects that have survived a certain number of garbage collection cycles are promoted to the old (or Tenured) generation. Note the continuous nature of the regions as shown on the diagram.

-

FIXME Placeholder image

-

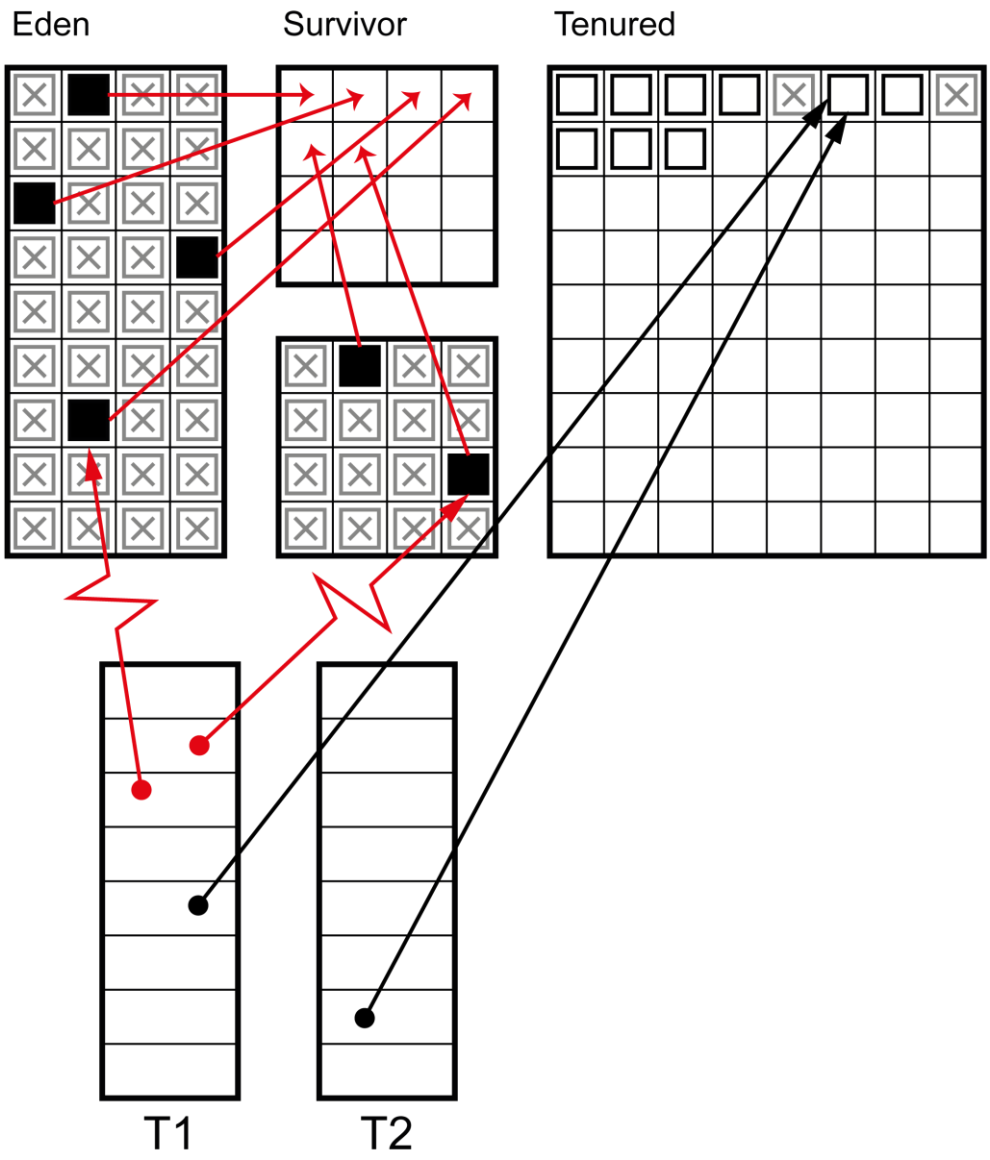


Figure 6-4. Generational collection

Divding up memory into different regions for purposes of generational collection has some additional consequences in terms of how HotSpot implements a mark-and-sweep

collection. One important technique involves keeping track of pointers that point into the young generation from outside. This saves the GC cycle from having to traverse the entire object graph in order to determine the young objects that are still live.

Note

"There are few references from old to young objects" is sometimes cited as a secondary part of the Weak Generational Hypothesis

To facilitate this process, HotSpot maintains a structure called a *card table* to help record which old generation objects could potentially point at young objects. The card table is essentially an array of bytes managed by the JVM. Each element of the array corresponds to a 512-byte area of old generation space.

The central idea is that when a field of reference type on an old object o is modified, then the card table entry for the card containing the instanceOop corresponding to o is marked as dirty. In HotSpot, this is achieved with a simple *write barrier* when updating reference fields. It essentially boils down to this bit of code being executed after the field store:

```
cards[*instanceOop >> 9] = 0;
```

Note that the dirty value for the card is 0, and the right-shift by 9 bits gives the size of the card table as 512 bytes.

Finally, we should note that the description of the heap in terms of old and young areas is historically the way that Java's collectors have managed memory. With the arrival of Java 8u40, a new collector ("Garbage First", or G1) reached production quality. G1 has a somewhat different view of how to approach heap layout, as we will see in Section 8.3. This new way of thinking about heap managament will be increasingly important as Oracle's intention is for G1 to become the default collector from Java 9 onwards.

Garbage Collection in HotSpot

Recall that unlike C / C++ and similar environments, Java does not use the operating system to manage dynamic memory. Instead, the JVM allocates (or reserves) memory up-front, when the JVM process starts, and manages a single, contiguous memory pool from user space.

As we have seen, this pool of memory is made up of different regions with dedicated purposes, and the address that an object resides at will very often change over time, as the collector relocates objects, which are normally created in Eden. Collectors

that perform relocation are known as an “evacuating” collector, as mentioned in the glossary in Section 7.1.1. Many of the collectors that HotSpot can use are evacuating.

Thread-local allocation

The JVM uses a performance enhancement to manage Eden. This is a critical region to manage efficiently, as it is where most objects are created, and very short-lived objects (those with lifetimes less than the remaining time to the next GC cycle) will never be located anywhere else.

For efficiency, the JVM partitions Eden into buffers, and hands out individual regions of Eden for application threads to use as allocation regions for new objects. The advantage of this approach is that each thread knows that it does not have to consider the possibility that other threads are allocating within the buffer. These regions are called *thread local allocation buffers* or TLABs.

Note

HotSpot dynamically sizes the TLABs that it gives to application threads, so if a thread is burning through memory, it can be given larger TLABs to reduce the overhead in providing buffers to the thread.

The exclusive control that an application thread has over its TLABs means that allocation is O(1) for JVM threads. This is because when a thread creates a new object, storage is allocated for the object, and the thread-local pointer is updated to the next free memory address. In terms of the C runtime, this is a simple pointer bump, i.e. one addition instruction to move the “next free” pointer onward.

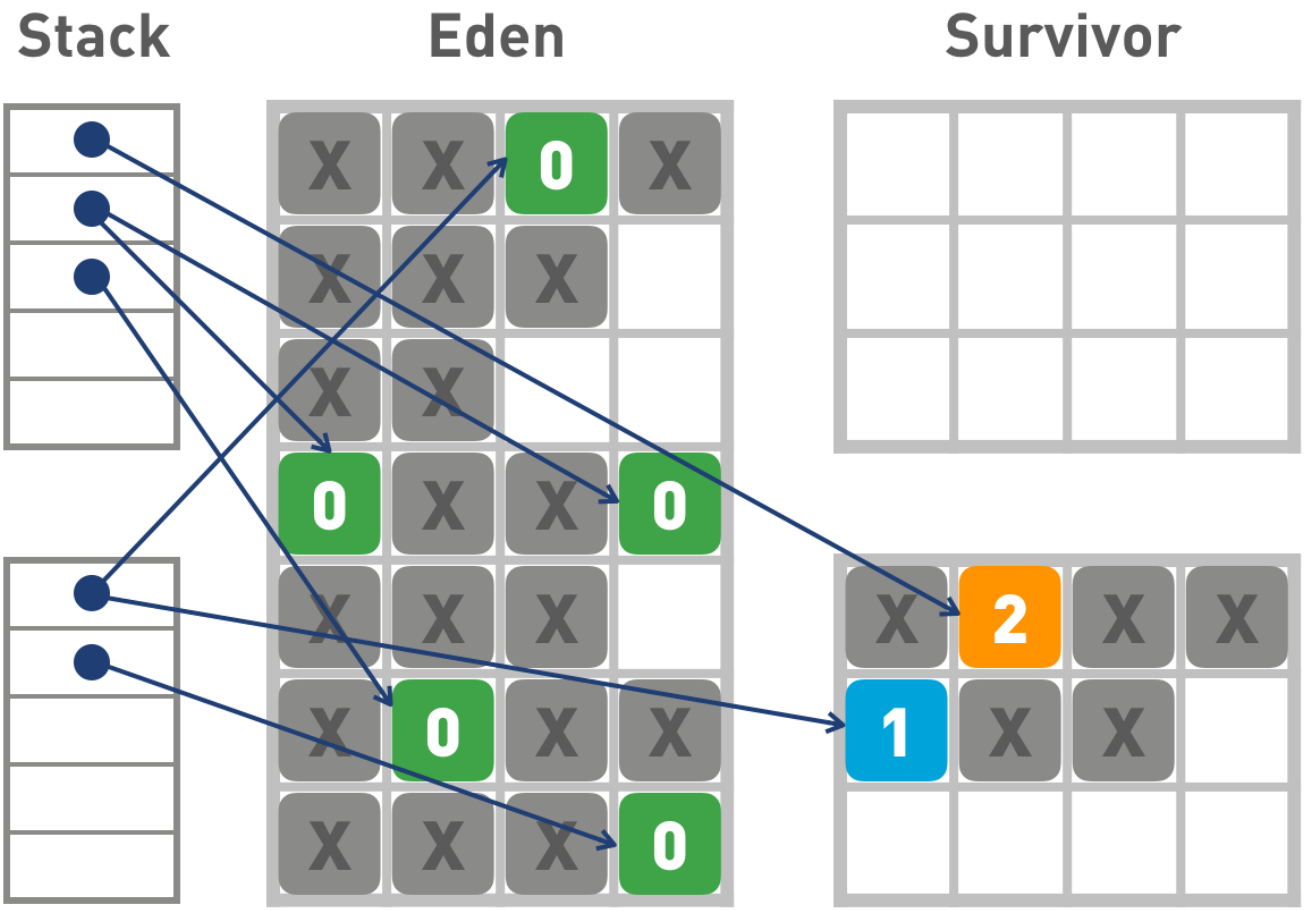


Figure 6-5. Thread-local allocation

This behavior can be seen in Figure 7-5, where each application thread holds a buffer to allocate new objects. If the application thread fills the buffer, then the JVM provides a pointer to a new area of Eden.

Hemispheric Collection

One particular special case of the evacuating collector is worth noting. Sometimes referred to as a hemispheric evacuating collector, this type of collector uses 2 (usually equal-sized) spaces. The central idea is to use the spaces as a temporary holding area for objects that are not actually long-lived. This prevents short-lived objects from cluttering up the Tenured generation and reduces the frequency of full GCs. The spaces have a couple of basic properties:

1.

When collecting the currently live hemisphere, objects are moved in a compacting fashion to the other hemisphere and the collected hemisphere is emptied for reuse.

2.

3.

One half of the space is kept completely empty at all times

4.

This approach does, of course, use twice as much memory as can actually be held within the hemispheric part of the collector. This is somewhat wasteful, but it is often a useful technique if the size of the spaces is not excessive. HotSpot uses this hemispheric approach in combination with the Eden space to provide a collector for the young generation.

The hemispheric part of HotSpot's young heap is referred to as the *survivor spaces*. As we can see from the view of VisualGC shown in Figure 7-6, the survivor spaces are normally relatively small as compared to Eden, and the role of the survivor spaces swaps with each young generational collection. We will discuss how to tune the size of the survivor spaces in Chapter 8.

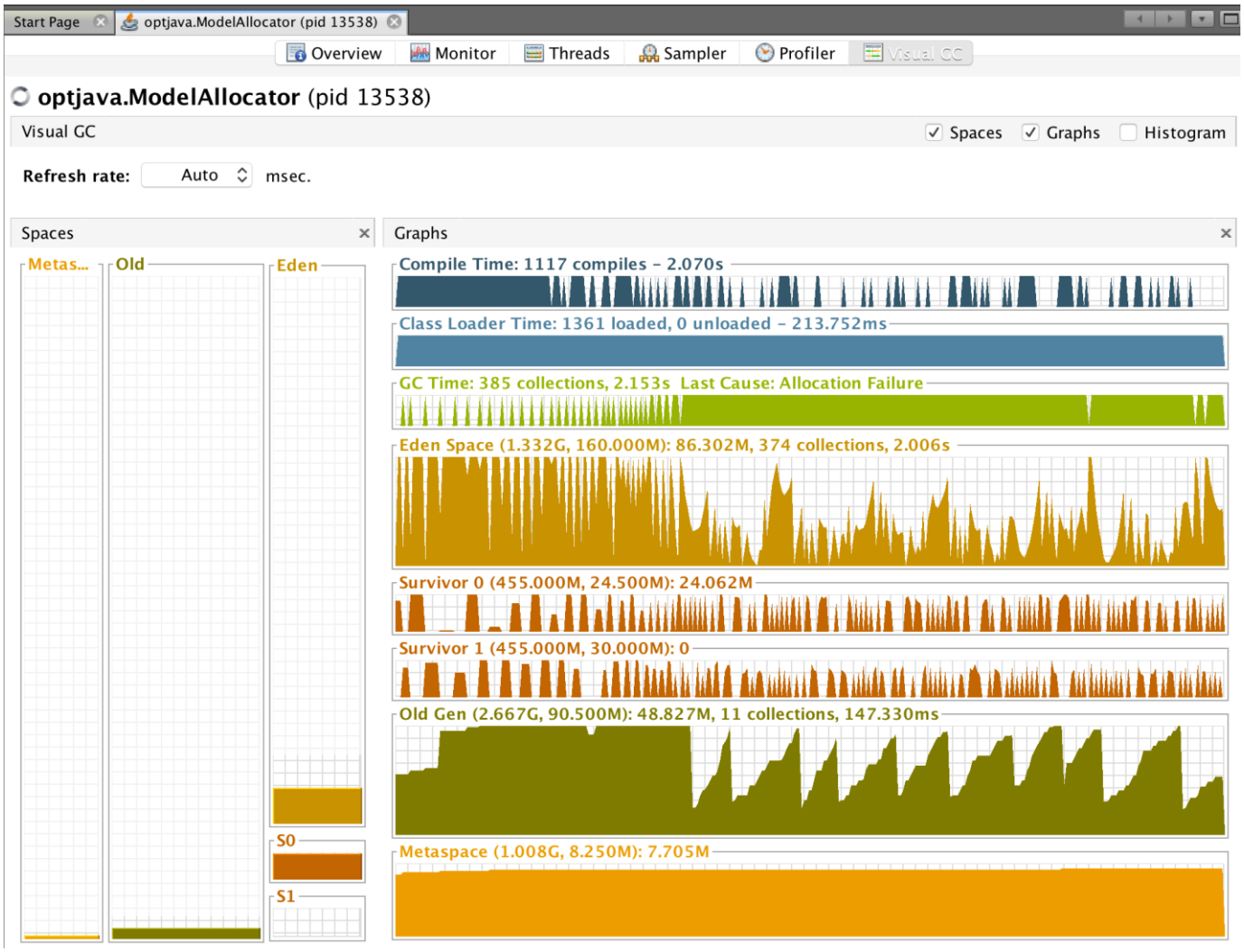


Figure 6-6. The VisualGC plugin

The VisualGC plugin for VisualVM, introduced in section XXX is a very useful initial GC debugging tool. As we will discuss in Chapter 8, the GC logs contain far more useful information and allow a much deeper analysis of GC than is possible from the moment-to-moment JMX data that VisualGC uses. However, when starting a new analysis, it is often helpful to simply eyeball the application's memory usage.

Using VisualGC it is possible to see some aggregate effects of garbage collection - such as objects being relocated in the heap, and the cycling between survivor spaces that happens at each young collection.

The parallel collectors

In Java 8 and earlier versions, the default collectors for the JVM are the parallel collectors. These are fully STW for both young and full collections, and they are

optimized for throughput. After stopping all application threads the parallel collectors use all available CPU cores to collect memory as quickly as possible. The available parallel collectors are:

- ParallelGC - the simplest collector for the young generation
-
- ParNew - a slight variation of ParallelGC that is used with the CMS collector
-
- ParallelOld - the parallel collector for the old (aka Tenured) generation
-

The parallel collectors are in some ways similar to each other - both are designed to use multiple threads to identify live objects as quickly as possible and to do minimal bookkeeping. However, there are some differences between them, so let's take a look at each in turn.

Young parallel collections

The most common type of collection is the young generational collection. This usually occurs when a thread tries to allocate an object into Eden but doesn't have enough space in its TLAB, and the JVM can't allocate a fresh TLAB for the thread. When this occurs, the JVM has no choice other than to stop all the application threads - because if one thread is unable to allocate, then very soon every thread will be unable.

Note

Threads can also allocate outside of TLABs, e.g. for large blocks of memory. The desired case is when the rate of non-TLAB allocation is low.

Once all application (or user) threads are stopped, HotSpot looks at the young generation (which is defined as Eden and the currently non-empty survivor space) and identifies all objects that are not garbage. This will utilize the GC roots (and the card table to identify GC roots coming from the old generation) as starting points for a parallel marking scan.

The `ParallelGC` collector then evacuates all of the surviving objects into the currently empty survivor space (and increments their generational count as they are relocated). Finally, Eden and the just-evacuated survivor space are marked as empty, reusable space and the application threads are started so that the process of handing out TLABs to application threads can begin again. This process can be seen starting in Figure 7-5 and continuing in Figure 7-7 and Figure 7-8.

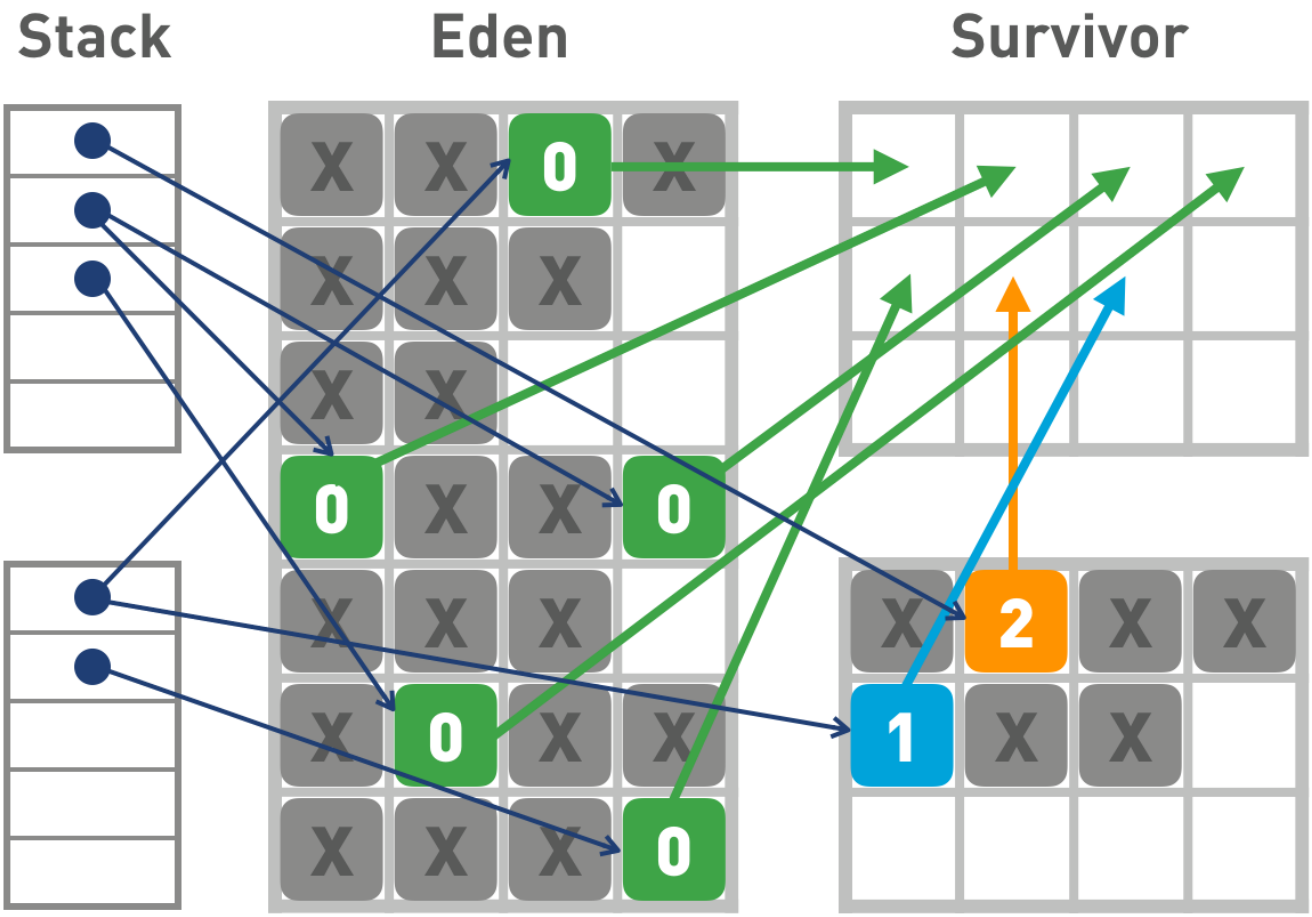


Figure 6-7. Collecting the young generation

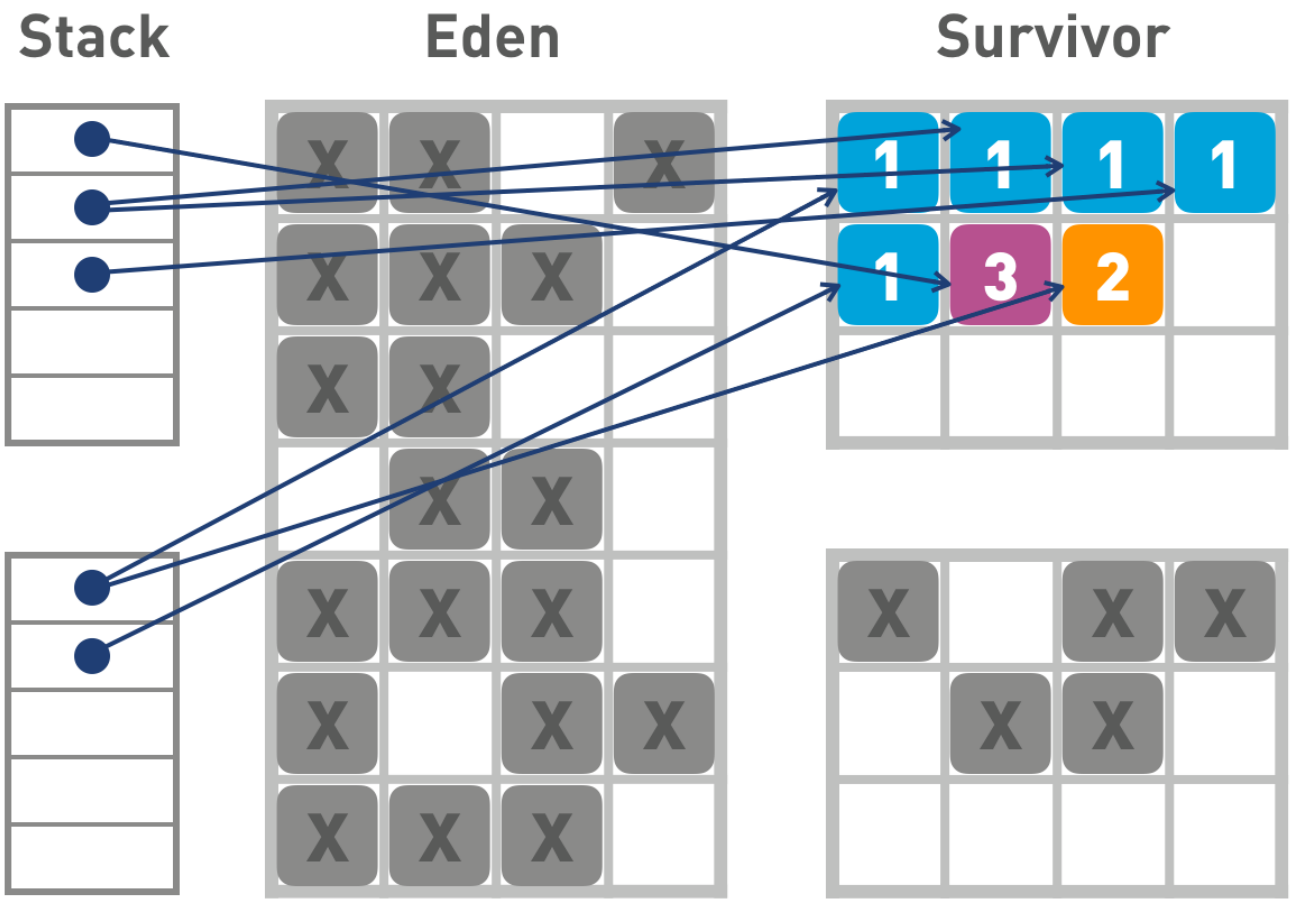


Figure 6-8. Evacuating the young generation

This approach attempts to take full advantage of the Weak Generational Hypothesis, by only touching live objects. It also wants to be as efficient as possible, and run using all cores as much as possible to shorten the STW pause time.

Old parallel collections

The `ParallelOld` collector is currently (as of Java 8) the default collector for the old generation. It has some strong similarities to `ParallelGC` but also some fundamental differences. In particular, `ParallelGC` is a hemispheric, evacuating collector, whereas `ParallelOld` is a compacting collector with only a single continuous memory space.

This means that as the old generation has no other space to be evacuated to, the parallel collector attempts to relocate objects within the old generation to reclaim space that may have been left by old obejcts dying. This means that the collector can

potentially be very efficient in its use of memory, and will not suffer from memory fragmentation.

-

FIXME Placeholder image

-

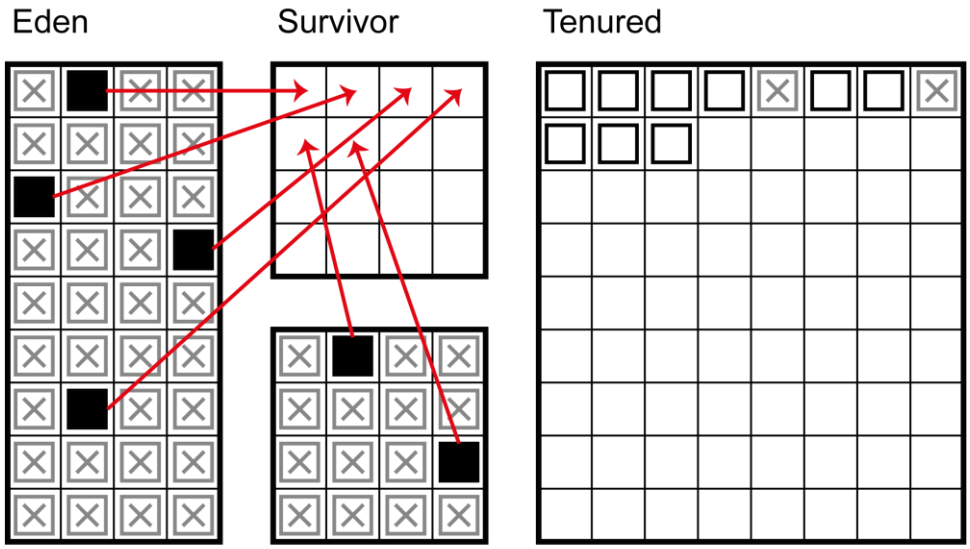


Figure 6-9. Evacuating vs compacting

This results in a very efficient memory layout at the cost of using a potentially large amount of CPU during full GC cycles. The difference between the two approaches can be seen in Figure 7-9.

The behavior of the two memory spaces is quite radically different, as they are serving different purposes. The purpose of the young collections is to deal with the short-lived objects, so the occupancy of the young space is changing radically with allocation and clearance at GC events.

By contrast, the old space does not change as obviously. Occasional large objects will be created directly in Tenured, but apart from that, the space will only change at collections - either by having objects promoted from young generation, or by a full rescan and rearrangement at an old or full collection.

Limitations of parallel collectors

The parallel collectors deal with the entire contents of a generation at once, and try to collect as efficiently as possible. However, this design has some drawbacks. Firstly, they are fully stop-the-world. This is not usually an issue for young collections, as the Weak Generational Hypothesis means that very few objects should survive.

Note

The design of the young parallel collectors is such that dead objects are never touched, so the length of the marking phase is proprortional to the (small) number of surviving objects

This basic design, coupled with the usually small size of the young regions of the heap, means that the pause time of young collections is very short for most workloads. A typical pause time for a young collection on a 2G heap (with default sizing) on modern heap might well be just a few milliseconds, very frequently under 10ms.

However, collecting the old generation is often a very different story. For one thing, the old generation is by default 7 times the size of the young generation. This fact alone will make the expected STW length of a full collection much longer than for a young collection.

Another key fact is that the marking time is proportional to the number of live objects in a region. Old objects may be long-lived, so a potentially larger number of old objects may survive a full collection.

This behavior also explains a key weakness of the parallel old collection - the STW time will scale roughly linearly with the size of the heap. As heap sizes continue to increase, `ParallelOld` starts to scale badly in terms of pause time.

Newcomers to GC theory sometimes entertain private theories that minor modifications to mark-and-sweep algorithms might help to alleviate STW pauses. However, this is not the case - garbage collection has been a very well studied research area of Computer Science for over 40 years and no such "can't you just…" improvement has ever been found.

As we will see in Chapter 8, mostly-concurrent collectors do exist and they can run with greatly reduced pause times. However, they are not a panacea and several fundamental difficulties with garbage collection remain.

As an example of one of the central difficulties with the naive approach to GC, let's consider TLAB allocation. This provides a great boost to allocation performance but is of no help to collection cycles. To see why, consider this code:

```
public static void main(String[] args) {
    int[] anInt = new int[1];
    anInt[0] = 42;
    Runnable r = () -> {
        anInt[0]++;
        System.out.println("Changed: "+ anInt[0]);
    };
    new Thread(r).start();
}
```

The variable `anInt` is an array object containing a single `int`. It is allocated from a TLAB held by the main thread, but immediately afterwards is passed to a new thread. To put it another way, the key property of TLABs - that they are private to a single thread - is true only at the point of allocation. This property can be violated basically as soon as objects have been allocated.

The ability of the Java environment to trivially create new threads is a fundamental, and extremely powerful, part of the platform. However, it complicates the picture for garbage collection considerably, as new threads imply execution stacks, each frame of which is a source of GC roots.

The role of allocation

Java's garbage collection process is most commonly triggered when memory allocation is requested but there is not enough free memory on hand to provide the required amount. This means that GC cycles do not occur on a fixed or predictable schedule but purely on an as-need basis.

This is one of the most critical aspects of garbage collection - it is not deterministic and does not occur at a regular cadence. Instead, a GC cycle is

triggered when one or more of the heap's memory spaces are essentially full, and further object creation would not be possible.

Tip

This as-needed nature makes garbage collection logs hard to process using traditional time series analysis methods. The lack of regularity between GC events is an aspect that most time series libraries cannot easily accomodate.

When GC occurs, all application threads are paused (as they cannot create any more objects, and no substantial piece of Java code can run for very long without producing new objects). The JVM takes over all of the cores to peform GC, and reclaims memory before restarting the application threads.

To better understand why allocation is so critical, let's consider the following highly simplified case study. The heap parameters are set up as shown, and we assume that they do not change over time. Of course a real application would normally have a dynamically resizing heap, but this example is to illustrate a simple case study.

Heap area	Size
Overall	2G
Old generation	1.5G
Young generation	500M
Eden	400M
S1	50M
S2	50M

After the application has reached its steady state, the following GC metrics are observed:

Allocation rate	100M/s
Young GC time	2ms
Full GC time	100ms
Object lifetime	200ms

This shows that Eden will fill in 4 seconds. So at steady state, a young GC will occur every 4 seconds. Eden has filled, and so GC is triggered. Most of the objects in Eden are dead, but any object that is still alive will be evacuated to a survivor space (SS1, for the sake of argument). In this simple model, any objects that were created in the last 200ms have not had time to die, so they will survive.

GC0 @ 4s 20M Eden → SS1 (20M)

After another 4 seconds, Eden refills and will need to be evacuated (to SS2 this time). However, in this simplified model, no objects that were promoted into SS1 by GC0 survive, because their lifetime is only 200ms and another 4s has elapsed, so all of the objects allocated prior to GC0 are now dead.

GC1 @ 8.002s 20M Eden → SS2 (20M)

Another way of saying this is that after GC1, the contents of SS2 consist solely of objects newly arrived from Eden and no object in SS2 has a generational age > 1. Continuing for one more collection and the pattern should become clear.

GC2 @ 12.004s 20M Eden → SS1 (20M)

This idealized, simple model leads to a situation where no objects ever become eligible for promotion to the Tenured generation, and the space remains empty throughout the run. This is, of course, very unrealistic.

Instead, the weak generational hypothesis indicates that object lifetimes will be a distribution, and due to the uncertainty of this distribution, some objects will end up surviving to reach Tenured.

Let's look at a very simple simulator for this allocation scenario. It allocates objects, most of which are very short-lived, but some of which have a considerably longer life span. It has a couple of parameters that define the allocation - `x` and `y` that between them define the size of each object, the allocation rate (`mbPerSec`), the lifetime of a short-lived object and the number of threads that the application should simulate (`nThreads`). The default values are shown in the next listing:

```
public class ModelAllocator implements Runnable {
    private volatile boolean shutdown = false;

    private double chanceOfLongLived = 0.02;
    private int multiplierForLongLived = 20;
    private int x = 1024;
    private int y = 1024;
    private int mbPerSec = 50;
    private int shortLivedMs = 100;
    private int nThreads = 8;
    private Executor exec = Executors.newFixedThreadPool(nThreads);
```

Omitting `main()` and any other startup / parameter setting code, the rest of the `ModelAllocator` looks like this:

```
    public void run() {
        final int mainSleep = (int) (1000.0 / mbPerSec);
```

```
        while (!shutdown) {
            for (int i = 0; i < mbPerSec; i++) {
                ModelObjectAllocation to = new ModelObjectAllocation(x, y,
lifetime());
                exec.execute(to);
                try {
                    Thread.sleep(mainSleep);
                } catch (InterruptedException ex) {
                    shutdown = true;
                }
            }
        }
    }

    // Simple function to model Weak Generational Hypothesis
    // Returns the expected lifetime of an object - usually this
    // is very short, but there is a small chance of an object
    // being "long-lived"
    public int lifetime() {
        if (Math.random() < chanceOfLongLived) {
            return multiplierForLongLived * shortLivedMs;
        }

        return shortLivedMs;
    }}
```

The allocator main runner is combined with a simple mock object used to represent the object allocation performed by the application.

```
public class ModelObjectAllocation implements Runnable {
    private final int[][] allocated;
    private final int lifeTime;

    public ModelObjectAllocation(final int x, final int y, final int liveFor) {
        allocated = new int[x][y];
        lifeTime = liveFor;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(lifeTime);
            System.err.println(System.currentTimeMillis() +": "+ allocated.length);
        } catch (InterruptedException ex) {
```

```
        }
    }}
```

When seen in VisualVM, this will display the *simple sawtooth* pattern that is often observed when looking at the memory behavior of Java applications that are making efficient use of the heap. This can be seen in Figure 7-10.

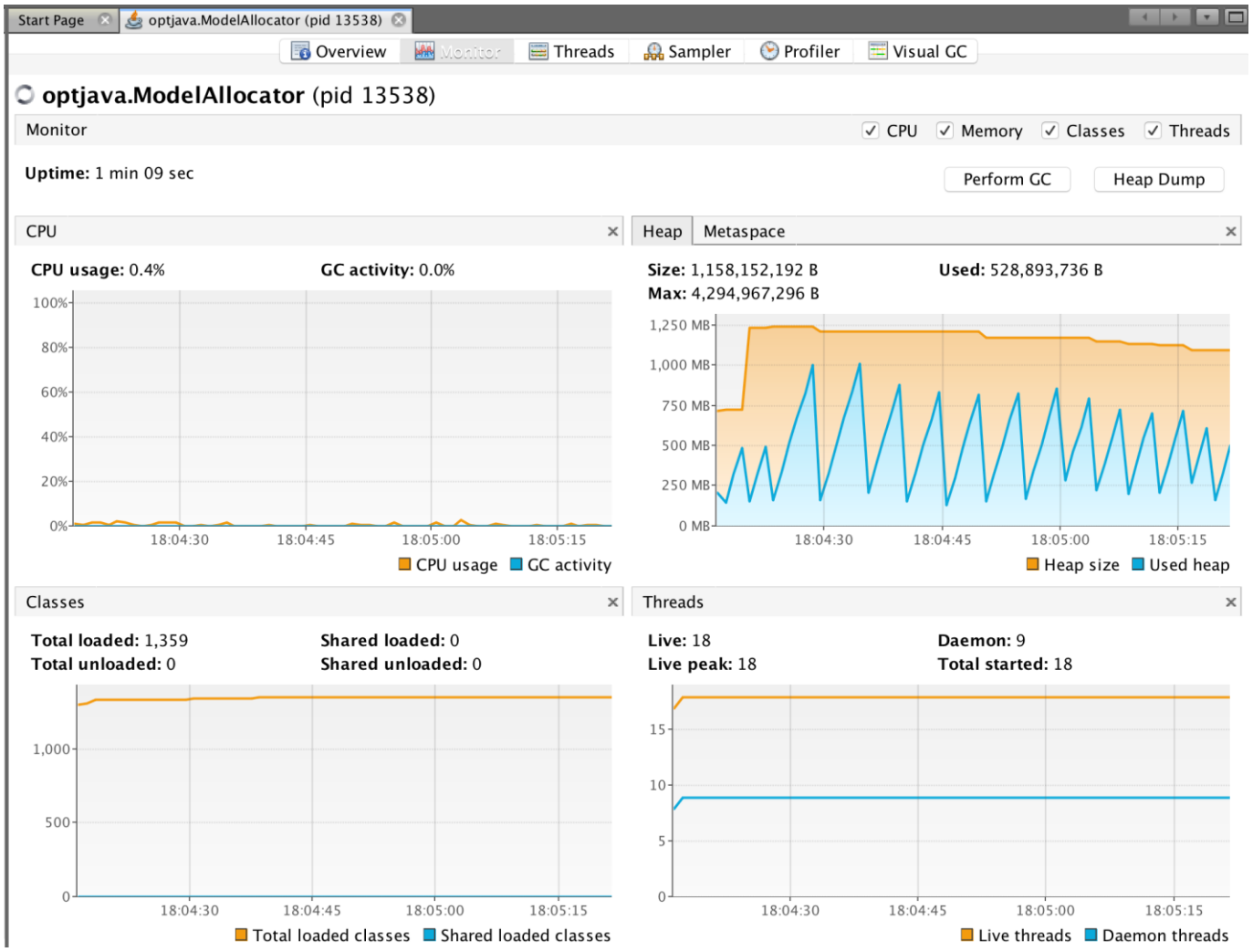


Figure 6-10. Simple sawtooth

We will have a great deal more to say about tooling and visualization of memory effects in Chapter 8. The interested reader can steal a march on the tuning discussion by downloading the allocation and lifetime simulator referenced in this chapter, and setting parameters to see the effects of allocation rates and percentages of long-lived objects.

To finish this discussion of allocation, we want to draw the reader's attention to a very common aspect of allocation behavior. In the real world, allocation rates can be

highly changeable and “bursty”. Consider the following scenario for an application with a steady-state behavior as previously described.

-

2s: Steady-state allocation - 100Mb/s

-
-

1s: Burst / spike allocation - 1Gb/s

-
-

100s: Back to steady-state - 100Mb/s

-

The initial steady state execution has allocated 200Mb in Eden. In the absence of long-lived objects, all of this memory has a lifetime of 100ms. Next the allocation spike kicks in. This allocates the other 200Mb of Eden space in just 0.2s and of this, 100Mb is under the 100ms age threshold. The size of the surviving cohort is larger than the survivor space, and so the JVM has no option but to promote these objects directly into Tenured.

GC0 @ 2.2s 100M Eden → Tenured (100M)

The sharp increase in allocation rate has produced 100M of surviving objects, although the reader should note that in this model all of the “survivors” are in fact short-lived and will very quickly become dead objects cluttering up the Tenured generation. They will not be reclaimed until a full collection occurs.

Continuing for a few more collections, the pattern becomes clear:

GC1 @ 2.602s 200M Eden → Tenured (300M)
GC2 @ 3.004s 200M Eden → Tenured (500M)
GC2 @ 7.006s 20M Eden → SS1 (20M) [+ Tenured (500M)]

Notice that, as discussed, the garbage collector runs as needed, and not at regular intervals. The bigger the allocation rate, the more frequent GCs are. If the allocation rate is too high, then objects will end up being forced to be promoted early.

This phenomenon is called *Premature Promotion*, and it is one of the most important indirect effects of garbage collection, and a starting point for many tuning exercises, as we will see in the next chapter.

Summary

The subject of garbage collection has been a live topic of discussion within the Java community since the platform's inception. In this chapter, we have introduced the key concepts that performance engineers need to understand to work effectively with the JVM's GC subsystem. These have included:

- Mark and sweep collection
- HotSpot's internal runtime representation for objects
- The weak generational hypothesis
- Practicalities of HotSpot's memory subsystems
- The parallel collectors
- Allocation and the central role it plays

In the next chapter, we will discuss GC tuning, monitoring and analysis. Several of the topics we met in this chapter, especially allocation (and specific effects such as

premature promotion) will be of particular significance to the goals and topics, and it may be helpful to refer back to this chapter frequently.

[1] The `System.gc()` method exists, but is basically useless for virtually any practical purpose

Chapter 7. Advanced garbage collection

In the last chapter we introduced the basic theory of Java garbage collection. From that starting point, we will move forward to introduce the theory of modern Java garbage collectors. This is an area that has unavoidable tradeoffs that guide the engineers choice of collector.

To begin with we'll introduce and delve into the other collectors the Hotspot JVM provides. This includes the ultra low-pause, mostly concurrent collector (CMS) and the modern general purpose collector (G1).

We will also consider some more rarely seen collectors. These are:

- Shenandoah
- C4
- Balanced
- Legacy Hotspot collectors

Not all of these collectors are used in the Hotspot virtual machine. So we will also be discussing the collectors of two other virtual machines - IBM J9 (a formerly closed-source JVM that IBM are in the process of opening) and Azul Zing (a proprietary JVM). We have previously introduced both of these VMs in Section 2.7 in the survey of VM implementations.

Tradeoffs and pluggable collectors

One aspect of the Java platform that beginners don't always immediately recognise is that while Java has a garbage collector, the language and VM specifications do not say

how GC should be implemented. In fact, there have been Java implementations (e.g. Lego Mindstorms) that didn't implement any kind of GC at all! [1]

Within the Sun (now Oracle) environment, the GC subsystem is treated as a pluggable subsystem - that means that the same Java program can execute with different garbage collectors without changing the semantics of the program, although the performance of the program may vary considerably based on which collector is in use.

The primary reason for having pluggable collectors is that GC is a very general computing technique and the same algorithm may not be appropriate for every workload. As a result, GC algorithms represent a compromise or tradeoff between competing concerns.

Note

There is no single, general purpose GC algorithm that can optimize for all GC concerns simultaneously.

The main concerns that developers often need to consider when choosing a garbage collector include:

- Pause time (aka pause length or duration)
- Throughput (as a percentage of GC time to application run time)
- Pause frequency (how often the collector needs to stop the application)
- Reclamation efficiency (how much garbage can be collected on a single GC duty cycle)
- Pause consistency (are all pauses roughly the same length)

-

Of these, pause time often attracts a disproportionate amount of attention. Whilst important for many applications, it should not be taken in isolation.

Note

For many workloads pause time is not an effective or useful performance characteristic.

For example, a highly parallel batch processing or Big Data application is likely to be much more concerned with throughput rather than pause length. For many batch jobs, pause times of even 10s of seconds are not really relevant, so a GC algorithm that favours CPU efficiency of GC and throughput is greatly to be preferred to an algorithm that is low-pause at any cost.

The performance engineer should also note that there are a number of other tradeoffs and concerns that are sometimes important when considering the choice of collector. However, in the case of Hotspot, the choice is constrained by the available collectors.

Within Oracle / Open JDK, as of version 9, there are three mainstream collectors for general production use. We have already met the parallel (aka throughput) collectors, and they are the easiest to understand from a theoretical and algorithmic point of view. In this chapter, we will meet the two other mainstream collectors and explain how they differ from parallel.

Towards the end of the chapter, in Section 8.4 and beyond we will also meet some other collectors that are available, but please note that not all of them are recommended for production use, or are now deprecated. We also provide a short discussion of collectors available in non-Hotspot JVMs.

Concurrent GC Theory

In specialised systems, such as graphics or animation display systems, there is often a fixed frame rate, which provides a regular, fixed opportunity for GC to be performed.

However, garbage collectors intended for general purpose use have no such domain knowledge with which to improve the determinism of their pauses. Worse still, non-determinism is directly caused by allocation behavior, and many of the systems that Java is used for exhibit highly variable allocation.

> The minor disadvantage of this arrangement is the delay of the computation proper; its major disadvantage is the unpredictability of these garbage collecting interludes
>
> Edgser Dijkstra

The starting point for modern GC theory is to try to address Dijkstra's insight that the non-deterministic nature of STW pauses (both in duration and in frequency) is the major annoyance when using GC techniques.

One approach is to use a collector that is concurrent (or at least partially or mostly concurrent) in order to reduce pause time by doing some of the work needed for collection whilst the application threads are running. This inevitably reduces the processing power available for the actual work of the application, as well as complicating the code needed to perform collection.

Before discussing concurrent collectors, though, there is an important piece of GC terminology and technology that we need to address, as it is essential to understanding the nature and behavior of modern garbage collectors.

JVM safepoints

In order to carry out a STW garbage collection, such as those performed by Hotspot's parallel collectors, all application threads must be stopped. This seems almost a tautology, but until now we have not discussed exactly *how* the JVM achieves this.

> The JVM is not actually a fully pre-emptive multithreading environment
>
> A Secret

This does not mean that it is purely a co-operative environment - quite the opposite. The operating system can still pre-empt (remove a thread from a core) at any time. This is done, for example, when a thread has exhausted its timeslice, or put itself into a `wait()`.

As well as this core OS functionality, the JVM also need to perform co-ordinated action. In order to facilitate this, the runtime requires each application thread to have special execution points, called *safepoints*, where the thread's internal data structures are in a known-good state. At these times, the thread is able to be suspended for coordinated actions.

Note

We can see the effects of safepoints in STW GC (the classic example) and thread synchronization, but there are others as well.

To understand the point of safepoints, consider the case of a fully STW garbage collector. For this to run, it requires a stable object graph. This means that all application threads must be paused. There is no way for a GC thread to demand that the OS enforces this demand on an application thread, so the application threads (which are executing as part of the JVM process) must co-operate to achieve this. There are two primary rules that govern the JVM's approach to safepointing:

-

JVM cannot force a thread to safepoint

-
-

JVM can prevent a thread leaving safepoint

-

This means that the implementation of the JVM interpreter must contain code to yield at a barrier if safepointing is required. For JIT-compiled methods, equivalent barriers must be inserted into the generated machine code. The general case for reaching safepoints then looks like this:

1.

JVM sets a global "time to safepoint" flag

2.

3.

Individual application threads will poll and see the flag has been set

4.

5.

They pause and wait to be woken up again

6.

When a safepoint occurs all app threads must stop. Threads that stop quickly must wait for slower stoppers (and this time may not be fully accounted for in the pause time statistics).

Normal app threads use this polling mechanism. They will always check in-between executing any 2 bytecodes in the interpreter. In compiled code, the most common cases where the JIT compiler has inserted a poll for safepoints are exiting a compiled method and when a loop branches backwards (e.g. to top of loop).

It is possible for a thread to take a long time to safepoint, and even theoretically to never stop (but this is a pathological case that must be deliberately provoked).

Tip

The idea that all threads must be fully stopped before the STW phase can commence is similar to latches, such as that implemene by `CountDownLatch` in the `java.util.concurrent` library.

Some specific cases of safepoint conditions are worth mentioning here.

A thread is automatically at a safepoint if it:

-

is blocked on a monitor

-
-

is executing JNI code

-

A thread is *NOT* necessarily at a safepoint if it:

-

is partway through executing a bytecode (interpreted mode)

-

-

has been interrupted by the OS

-

We will meet the safepointing mechanism again later on, as it is a critical piece of the internal workings of the JVM.

Tri-color marking

A key paper in the development of concurrent GC theory was “On-the-Fly Garbage Collection: An exercise in Cooperation” (1978) by Dijkstra and Lamport. This paper decribes their *tri-color marking* algorithm. This was a landmark for both correctness proofs of concurrent algorithms and GC, and the basic algorithm it describes remains an important part of garbage collection theory.

The algorithm works like this:

-

GC Roots are colored grey

-
-

All other objects are colored white

-
-

A marking thread moves from a grey node to a white child, and colors it grey

-
-

If a grey node has no white children, color the grey node black

-
-

Terminate once there are no grey nodes left

-
-

White nodes are eligible for collection

-

There are some complications but this is the basic form of the algorithm. An example is shown in Figure 8-1.

= FIXME

Figure 7-1. Tri-color marking

Concurrent collection also frequently makes use of a technique called *snapshot at the begining* (SATB). This means that the collector regards objects as live if they were reachable at the start of the collection cycle or have been allocated since. This add some minor wrinkles to the algorithm, such as mutator threads needing to create new objects in black state if a collection is running, and in white state if no collection is in progress.

The tri-color marking algorithm needs to be combined with a small amount of additional work, in order to ensure that the changes introduced by the running application threads do not cause live objects to be collected. This is because in a concurrent collector, application (mutator) threads are changing the object graph whilst marking threads are executing the tri-color algorithm.

Consider the situation where an object has already been colored black by a marking thread, and then is updated to point at a white object by a mutator thread. This is the situation shown in Figure 8-2.

= FIXME

Figure 7-2. Mutator thread could invalidate tri-color marking

If all references from grey objects to the new white object are now deleted, we have a situation where the white object should still be reachable, but will be deleted, as it will not be found, according to the rules of the algorithm.

This problem can be solved in several different ways. We could, for example, change the color of the black object back to grey, and add it back to the set of grey nodes that need processing as the mutator thread processes the update.

That approach, using a "write barrier" for the update would have the nice algorithmic property that it would maintain the *tri-color invariant* throughout the whole of the marking cycle.

> No black object node may hold a reference to a white object node during concurrent marking
>
> Tri-color invariant

An alternative approach would be to keep a queue of all changes that could potentially violate the invariant, and then have a secondary "fixup" phase that runs after the main phase has finished. Different collectors can resolve this problem with tri-color marking in different ways, based on criteria such as performance, or the amount of locking required.

In the next section we will meet the low-pause collector, CMS. We are introducing this collector before the others despite it being a collector with only a limited range of

applicability. This is because developers are frequently unaware of the extent to which GC tuning requires trade-offs and compromises.

By considering CMS first, we can showcase some of the practical issues that performance engineers should be aware of when thinking about garbage collection. The hope is that this will lead to a more evidence-based approach to tuning and the inherent trade-offs in the choice of collector, and a little less Tuning By Folklore.

CMS

The Concurrent Mark and Sweep (CMS) collector is designed to be a extremely low pause collector for the Tenured (aka old gen) space only. It is usually paired with a slightly modified parallel collector for collecting the young generation (called *ParNew* rather than *ParallelGC*).

CMS does as much work as possible whilst application threads are still running, so as to minimise pause time. The marking algorithm used is a form of tri-color marking and this means, of course, that the object graph may mutating whilst the collector is scanning the heap. As a result, CMS must fix up its records to avoid violating the second rule of garbage collectors, and collecting an object that is still alive.

This leads to a more complex set of phases in CMS than are seen in the parallel collectors. These phases are usually referred to as follows:

1.

Initial Mark (STW)

2.

3.

Concurrent Mark

4.

5.

Concurrent Preclean

6.

7.

Remark (STW)

8.

9.

Concurrent Sweep

10.

11.

Concurrent Reset

12.

For most of the phases, GC runs alongside application threads. However, for two phases (Initial Mark and Remark), all application threads must be stopped. The overall effect should be to replace a single long STW pause with 2 STW pauses which are usually very short.

The purpose of the Initial Mark phase is to provide a stable set of starting points for GC that are within the region - these are known as the *internal pointers* and provide an equivalent set to the GC roots for the purposes of the collection cycle. The advantage of this approach is that it allows the marking phase to focus on a single GC pool without having to consider other memory regions.

After the Initial Mark has concluded, the concurrent marking phase commences. This essentially runs the tri-color marking algorithm on the heap, keeping track of any changes that might later require fixup.

The Concurrent Preclean phase appears to try to reduce the length of the STW Remark phase as much as possible. The Remark phase uses the card tables to fix up the marking that might have been affected by mutator threads during the concurrent marking phases.

The observable effects of using CMS are as follows, for most workloads:

•

Application threads don’t stop for as long

-
-

A single full GC cycle takes longer (in wallclock time)

-
-

Application throughput is reduced whilst a CMS GC cycle is running

-
-

GC uses more memory for keeping track of objects

-
-

Considerably more CPU time is needed overall to perform GC

-
-

CMS does not compact the heap, so Tenured can become fragmented

-

The careful reader will note that not all of the above characteristics are positive. Remember that with GC there is no silver bullet - merely a set of choices that are appropriate (or acceptable) to the specific workload an engineer is tuning.

How CMS works

One of the most often overlooked aspects of CMS is, bizarrely, its great strength. CMS mostly runs concurrently with application threads. By default, CMS will use half of the available threads to perform the concurrent phases of GC, and leave the other half for application threads to execute Java code *and this inevitably involves allocating new objects*. This sounds like a flash of the blindingly obvious, but it has an immediate consequence:

Note

What happens if Eden fills up while CMS is running?

The answer is, unsurprisingly, that because application threads cannot continue, they pause and a (STW) young GC runs whilst CMS is running. This young GC run will usually take longer than in the case of the parallel collectors, because it only has half the cores available for young gen GC (as the other half of the cores are running CMS).

At the end of this young collection, some objects will usually be eligible for promotion to Tenured. These promoted objects need to be moved into Tenured whilst CMS is still running, which requires some co-ordination between the two collectors. This is why CMS requires a slightly different young collector.

Under normal circumstances, the young collection promotes only a small amount of objects to Tenured, and the CMS old collection completes normally, freeing up space in Tenured. The application then returns to normal processing, with all cores released for application threads.

However, consider the case of very high allocation, perhaps causing premature promotion (discussed at the end of Section 7.6) in the young collection. This can cause a situation in which the young collection has too many objects to promote for the available space in Tenured. This can be seen in Figure 8-3.

= FIXME

Figure 7-3. Concurrent mode failure from allocation pressure

This is known as a *Concurrent Mode Failure* (CMF) and the JVM has no choice at this point but to fall back to a collection using ParallelOld - which is fully STW. Effectively, the allocation pressure has been so high that CMS did not have time to finish processing the old generation before all the “headroom” space to accomodate newly-promoted objects filled up.

To avoid frequent concurrent mode failures, CMS needs to start a collection cycle before Tenured is completely full. The heap occupancy level of Tenured at which CMS will start to collect is controlled by the observed behavior of the heap. It can be affected by switches, and starts off defaulted to 75% of Tenured.

There is one other situation that can lead to a concurrent mode failure, and this is heap fragmentation. Unlike ParallelOld, CMS does not compact Tenured as it runs. This means that after a completed run of CMS, the free space in Tenured is not a single contiguous block, and objects that are promoted have to be filled into the gaps between existing objects.

Figure 7-4. Concurrent mode failure from fragmentation

At some point, a young collection may encounter a situation where an object can’t be promoted into Tenured due to a lack of sufficient contiguous space to copy the object into. This can be seen in Figure 8-4.

This is a concurrent mode failure caused by heap fragmentation, and as before, the only solution is to fall back to a full collection using ParallelOld (which *is* compacting) as this should free up enough contiguous space to allow the object to be promoted.

In both the heap fragmentation case and the case where young collections outpace CMS, the need to fall back to a fully STW ParallelOld collection can be a major event for an application. In fact, the tuning of low latency applications that use CMS in order to avoid ever suffering a CMF is a major topic in its own right.

Internally, CMS uses a free list of *chunks* of memory to manage the available free space. During the final, Concurrent Sweep, phase contiguous free blocks will be coalesced by the sweeper threads. This is to provide larger blocks of free space and try to avoid CMFs caused by fragmentation.

However, the sweeper runs concurrently with mutators. Thus, unless the sweeper and the allocator threads synchronize properly, a freshly allocated block might get incorrectly swept up. To prevent this, the sweeper locks the free lists while it is sweeping.

Basic JVM flags for CMS

The CMS collector is activated with this flag:

```
-XX:+UseConcMarkSweepGC
```

This flag will activate CMS, and on modern versions of Hotspot will also activate `ParNewGC` (a slight variation of the parallel young collector) as well.

In general, CMS provides a very large number of flags (over 60) that can be adjusted. It can sometimes be tempting to engage in benchmarking exercises that attempt to optimize performance by carefully adjusting the various options that CMS provides. This should be resisted, as in most cases this is actually the *Missing the Bigger Picture* (Section 4.6) or *Tuning by Folklore* (Section 4.4) antipattern in disguise.

We will cover CMS tuning in more detail in Section 9.4.

G1

G1 (the "Garbage First" collector) is a very different style of collector than either Parallel or CMS. It was first introduced in a highly experimental and unstable form in Java 6, but was extensively rewritten throughout the lifetime of Java 7, and only really became stable and production ready with the release of Java 8 Update 40.

Tip

We do not recommend using G1 with any version of Java prior to 8u40, regardless of the type of workload being considered.

G1 was orginally intended to be a replacement low-pause collector, that was:

-

Much easier to tune than CMS

-
-

Less susceptible to premature promotion

-
-

Capable of better scaling behavior (especially pause time) on big heaps

-
-

Able to eliminate (or greatly reduce the need to fall back to) full STW collections

-

However, over time, G1 evolved into being thought of as more of a general purpose collector, that had better pause time on larger heaps (which are increasingly thought of as “the new normal”).

Note

G1 is intended by Oracle to become the default collector in Java 9, taking over from the Parallel collectors. It is therefore very important that performance analysts have a good understanding of G1.

The G1 collector has a design that rethinks the notion of generations, as we have met them so far. Unlike the parallel or CMS collectors, G1 does not have dedicated, contiguous memory spaces per generation. In addition, it does not follow the hemispherical heap layout, as we will see.

G1 Heap layout and regions

The G1 heaps is based upon the concept of regions. These are areas which are by default 1Mb in size (but are larger on bigger heaps). The use of regions allows for non-contiguous generations and makes it possible to have a collector that does not need to collect all garbage on each run.

Note

The overall G1 heap is still contiguous in memory - it's just that the memory that makes up each generation no longer has to to be.

The region-based layout of the G1 heap can be seen in Figure 8-5. This view was shown by the *regions* JavaFX application, which we will meet properly in Chapter 9.

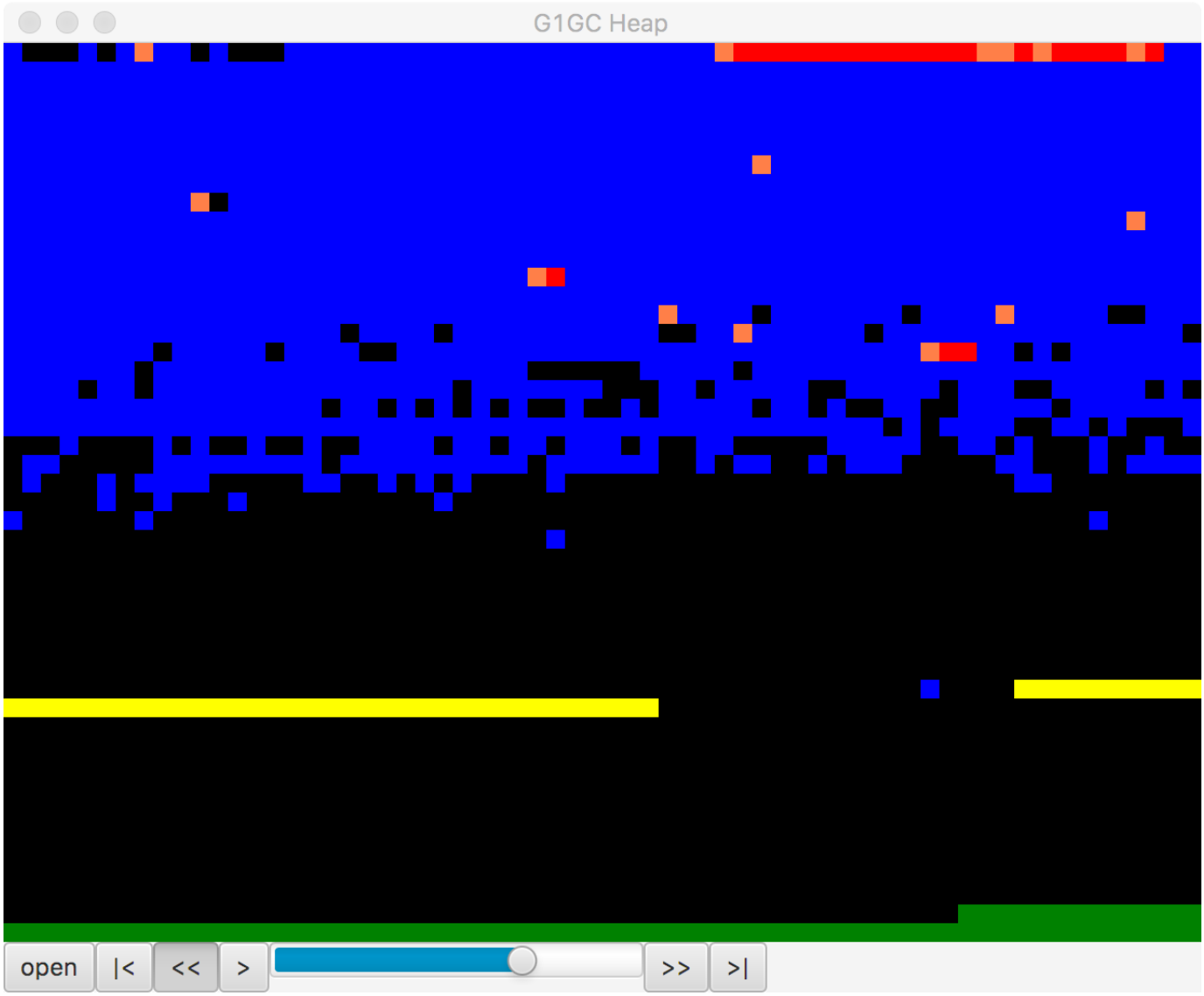


Figure 7-5. Visualising G1's heap

G1's algorithm allows regions of either 1, 2, 4, 8, 16, 32 or 64M. By default, it expects between 2048 and 4095 regions in the heap and will adjust the region size to achieve this.

To calculate the region size, we compute this value:

<Heap size> / 2048

and then round it down to the nearest permitted region size value. Then the number of regions can be calculated:

Number of regions = <Heap size> / <region size>

This value can, as usual, be changed by applying a runtime switch.

G1 algorithm design

The high-level picture of the collector is that G1:

- Uses a concurrent marking phase
- Is an evacuating collector
- Provides “statistical compaction”

Whilst warming up, the collector keeps track of the statistics of how many “typical” regions can be collected per GC duty cycle. If enough memory can be collected to balance the new objects that have been allocated since the last GC, then G1 is not losing ground to allocation.

The concepts of TLAB allocation, evacuation to survivor space and promoting to Tenured are broadly similar to the other Hotspot GCs that we’ve already met.

Note

Objects that occupy more space than half a region size are considered humongous objects. They are directly allocated in special *humongous regions* which are free,

contiguous regions that are immediately made part of the Tenured generation (rather than Eden).

G1 still has a concept of a Young generation made up of Eden and Survivor regions, but of course the regions that make up a generation are not contiguous in G1. The size of the Young generation is adaptive and is based on the overall pause time goal.

Recall that when we met the ParallelOld collector, the heuristic "few references from Old to Young generation" was discussed, in Section 7.3.1. Hotspot uses a mechanism called card tables to help take advantage of this phenomenon in the Parallel and CMS collectors.

The G1 collector has a related feature to help with region tracking. The *Remembered Sets* (usually just called *RSets*) are per-region entries that track outside references that point into a heap region. This means that instead of tracing through the entire heap for references that point into a region G1 just needs to examine RSet and then scan those regions for references.

= FIXME

Figure 7-6. Remembered Sets

The RSets are shown in Figure 8-6 and are used to implement G1's approach to dividing up the work of GC between allocator and collector.

Both RSets and card tables are approaches that can help with a GC problem called *Floating Garbage*. This is an issue that is caused when objects that are otherwise dead are kept alive by references from dead objects outside the current collection set.

That is, on a global mark they can be seen to be dead, but a more limited local marking may incorrectly report them as alive, depending on the root set used.

G1 phases

G1 has a collection of phases that are somewhat similar to the phases we've already met, especially in CMS.

1.

Initial Mark (STW)

2.

3.

Concurrent Root Scan

4.

5.

Concurrent Mark

6.

7.

Remark (STW)

8.

9.

Cleanup (STW)

10.

The Concurrent Root Scan is a concurrent phase that scans survivor regions of the initial mark for references to the old generation. This phase must complete before the next Young GC can start. In the Remark phase the marking cycle is completed. This phase also performs reference processing (including weak and soft references), and handles cleanup relating to implementing the SATB approach.

Cleanup is mostly STW, and comprises accounting and RSet "scrubbing". The accounting task identifies regions that are now completely free, and ready to be reused (e.g. as Eden regions).

Basic JVM flags for G1

The switch that you need to enable G1 (in Java 8 and before) is:

+XX:UseG1GC

Recall that G1 is based around *pause goals*. These allow the developer to specify a goal for the maximum amount of time that the application should pause on each garbage collection cycle. This is expressed as a goal and there is no guarantee that the application will be able to meet it. If this value is set too low, then the GC subsystem will be unable to meet it.

Note

Garbage collection is driven by allocation, which can be highly unpredictable for many Java applications. This can limit or destroy G1's ability to meet pause goals. The switch that controls the core behavior of the collector is:

-XX:MaxGCPauseMillis=200

This means that the default pause time goal is 200ms. In practice, pause times under <100ms are very hard to achieve reliably and the goal may not be met by the collector. One other option that may also be of use is the option of changing the region size, overriding the default algorithm:

-XX:G1HeapRegionSize=<n>

Note that n must be a power of 2, between 1 and 64, as MB. We will meet other G1 flags when we discuss tuning G1 in Chapter 9.

Overall, G1 is stable as an algorithm and is fully supported by Oracle (and recommended as of 8u40). However, for truly low latency workloads it is still not as

low pause as CMS for most workloads. It is also unclear that it will ever be able to challenge a concurrent collector like CMS on pure pause time.

However, the collector is still improving and is the focus of Oracles' GC engineering efforts within the JVM team. At time of writing, it seems highly likely that the default garbage collector for Java 9 will indeed change to becoming G1. It is therefore very important to test your applications with the G1 collector as soon as possible.

Shenandoah

As well as the Oracle-led effort to produce a next-generation, general purpose collector, Red Hat have been working on their own collector, called *Shenandoah*, within the OpenJDK project. This is still an experimental collector and not ready for production use at time of writing. However, it shows some promising characteristics and deserves at least an introduction.

Like G1, the aim of Shenandoah is to bring down pause times (especially on large heaps). Shenandoah's approach to this is to perform concurrent compaction. The resulting phases of collection in the Shenandoah are:

1.

Initial Mark (STW)

2.

3.

Concurrent Marking

4.

5.

Final Marking (STW)

6.

7.

Concurrent Compaction

8.

These phases may initially seem similar to those seen in CMS and G1, and some similar approaches (e.g. SATB) are used by Shenandoah. However, there are some important fundamental differences.

One of the most striking and important aspects of Shenandoah is its use of a *Brooks Barrier*. This technique uses an additional word of memory per object, and was first introduced in Brooks' paper "Trading data space for reduced time and code space in real-time garbage collection on stock hardware" (1984). This word is used to indicate that the object has been relocated during a previous phase of garbage collection, and to give the location of the new version of the object contents.

Figure 7-7. The Brooks pointer

The resulting heap layout used by Shenandoah for its oops can be seen in Figure 8-7. This mechanism is sometimes called a "forwarding pointer" approach.

Note

The Brooks pointer mechanism relies upon the availability of hardware compare-and-swap (CAS) operations to provide atomic updates of the forwarding address.

The Concurrent Marking phase traces through the heap and marks any live objects. If an object reference points to an oop that has a forwarding pointer, then the reference is updated to point directly at the new oop location. This can be seen in Figure 8-8

= FIXME

Figure 7-8. Updating the forwarding pointer

In the Final Marking phase, Shenandoah stops the world to re-scan the root set, copy and update roots to point to the evacuated copies.

Concurrent Compaction

The GC threads (which are running concurrently with app threads) now perform evacuation as follows:

- Copy the object into a TLAB (speculatively)
- Use a CAS operation to update the Brooks pointer to point at speculative copy

If this succeeds, then the compaction thread won the race, and all future accesses to this version of the object will be via the Brooks pointer

-
-

If it fails, the the compaction thread lost. It undoes the speculative copy, and follows the Brooks pointer left by the winning thread

-

As Shenandoah is a concurrent collector, then whilst a collection cycle is running, more garbage is being created by the application threads. Therefore, over an application run, collection has to keep up with allocation.

Obtaining Shenandoah

The Shenandoah collector is not currently available as part of a shipping Oracle Java build, or in most OpenJDK distributions. It is shipped as part of the IcedTea binaries in some Linux distros, including Red Hat Fedora.

For other users, compilation from source will be necessary (at time of writing). This is straightforward on Linux, but may be less so on other operating systems, due to differences in compilers (macOS uses `clang` rather than `gcc`, for example) and other aspects of the operating environment. The details can be found in XXX Appendix A?

Once a working build has been obtained, Shenandoah can be activated with this switch:

-XX:+UseShenandoahGC

A comparison of Shenandoah's pause times as compared to other collectors can be seen in Figure 8-9.

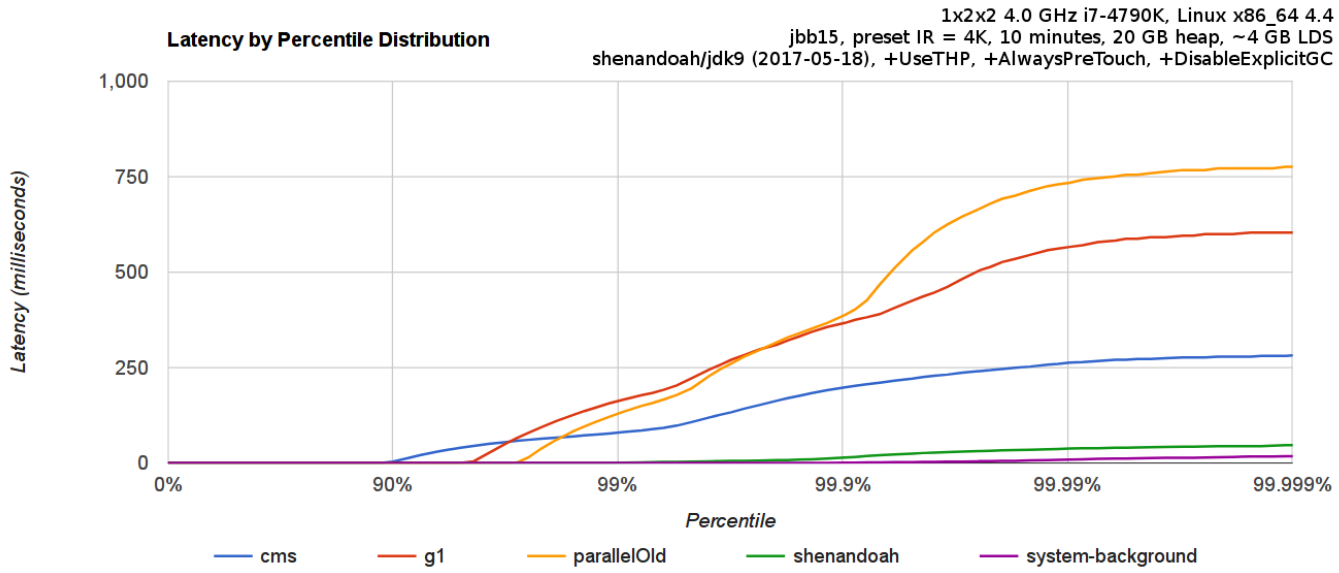


Figure 7-9. Shenandoah compared to other collectors

One aspect of Shenandoah that is underappreciated is that it is not a generational collector. As it moves closer to being a production-capable collector, the consequences of this design decision will become clearer, but this is a potential source of concern for performance sensitive applications.

C4 (Azul Zing)

Azul Systems produce two different Java platform offerings. One of them, Zulu, is an OpenJDK-based FOSS solution available for multiple platforms. The other is Zing, which is a commercial and proprietary platform, which is available for Linux only. It uses a version of the Java class libraries that derive from OpenJDK (albeit under an Oracle proprietary license), but a completely different virtual machine.

Note

One important aspect of Zings design is that it was designed for 64 bit machines from the start, and never had any intention of supporting 32 bit architectures.

The Zing VM contains several novel software technologies, including the C4 (Continuously Concurrent Compacting Collector) garbage collector and some novel JIT technology, including ReadyNow and a compiler called Falcon which we will meet in Chapter 10.

Like Shenandoah, Zing uses a concurrent compaction algorithm, but it does not utilise a Brooks pointer. Instead, Zings object header is a single 64-bit word (rather than

the 2 word header that Hotspot uses). The single header word contains a *kid* which is a numeric klass ID (around 25 bits long) rather than a klass pointer.

In Figure 8-10 we can see the Zing object header, including the use of some of the oop reference bits for the LVB rather than as address bits.

= FIXME

Figure 7-10. Object header layout in Zing

The bottom 32 bits of the header word are used for lock information. This includes the lock state but also additional information depending on the state of the lock. For example, in the case of a *thin lock* this will be the owning thread ID for thin locks. See section Section 15.3 for more details about thin locks.

Note

Zing does not support compressed oops or an equivalent technology, so for heaps smaller than ~30GB the object headers are larger and will take up more space in the heap than the equivalent Hotspot heap.

The design choice of choosing to implement only on 64-bit architectures meant that Zings metadata never needed to fit into 32 bits (or to allocate extended structures to be addressed via pointer indirection), which meant that some of the pointer gymnastics present in 32-bit Hotspot could be avoided.

Loaded value barrier (LVB)

In the Shenandoah collector application threads can load references to objects that might have been relocated, and the Brooks pointer is used to keep track of their new location. The central idea of the LVB is to avoid this pattern, and instead provide a solution where every loaded reference is pointing directly at the current location of the object as soon as the load has completed. Azul refers to this as a “self-healing barrier”.

A comparison between the two techniques can be seen in Figure 8-11

= FIXME

Figure 7-11. LVB vs Brooks barrier

If Zing follows a reference to an object that has been relocated by the collector, then before doing anything else, the application thread updates the reference to the new location of the object, thus “healing” the cause of the relocation problem. This means that each reference is updated at most once, and if the reference is never used again, no work is done to keep it up to date.

As well as the header word, Zings object references (e.g. from a local variable located on the stack to the object stored on the heap) use some of the bits of the reference to indicate metadata about the GC state of the object. This saves some space by using bits of the reference itself, rather than bits of the single header word.

Zing defines a Reference structure, which is laid out like this:

```
struct Reference {
    unsigned inPageVA : 21;     // bits 0-20
    unsigned PageNumber: 21;    // bits 21-41
```

```
    unsigned NMT : 1;           // bit 42
    unsigned SpaceID : 2;       // bits 43-44
    unsigned unused : 19;       // bits 45-63
};

int Expected_NMT_Value[4] = {0, 0, 0, 0};

// Space ID values:
// 00 NULL and non-heap pointers
// 01 Old Generation references
// 10 New Generation references
// 11 Unused
```

The NMT bit is a piece of metadata which stands for *Not Marked Through*. This bit is used to indicate that the object has already been marked in the current collection cycle. C4 maintains a target state that live objects should be marked as, and when an object is located during the mark, the NMT bit is set so that it is equal to the target state. At the end of the collection cycle, C4 flips the target state bit, so now all surviving objects are ready for the next cycle.

The overall phases of the GC cycle in C4 are:

1.

Mark

2.

3.

Relocate

4.

5.

Remap

6.

The relocation phase, like G1, will focus on the sparsest from-pages. This is to be expected, given that C4 is an evacuating collector.

C4 uses a technique called *hand over hand* compaction to provide continuous compaction. This relies upon a feature of the virtual memory system - the disconnect between physical and virtual addresses. In normal operation, the virtual memory subsystem maintains a mapping between a virtual page in the process address space, and an underlying physical page.

Note

Unlike Hotspot, which does not use system calls to manage Java heap memory, Zing does make use of calls into the kernel as part of its GC cycle.

With Zing's evacuation technique, objects are relocated by copying them to a different page, which naturally corresponds to a different physical address. Once all objects from a page have been copied, the physical page can be freed returned to the operating system. There will still be application references that point into this now-unmapped virtual page. However, the LVB will take care of such references and will fix them before a memory fault occurs.

Zing's C4 collector runs two collection algorithms at all times - one for young and one for old objects. This obviously has an overhead, but as we we will examine in Chapter 9, when tuning a concurrent collector (such as CMS), it is useful for overhead and capacity planning reasons to assume that the collector may be in *back-to-back* mode when running. This is not so different to the continuously running mode that C4 exhibits.

Ultimately, the performance engineer should examine the benefits and tradeoffs of moving to Zing and the C4 collector carefully. The "measure, don't guess" philosophy applies to the choice of VM as much as it does anywhere else.

Balanced (IBM J9)

IBM produce a JVM called J9. This has historically been a proprietary JVM, but IBM are now in the process of open-sourcing it, and renaming it Open J9. The VM has several different collectors that can be switched on, including a high-throughput collector similar to the Parallel collector Hotspot defaults to.

In this section, however, we will discuss the Balanced GC collector. It is a region-based collector available on 64-bit J9 JVMs and designed for heaps in excess of 4gb. Its primary design goals are:

1.

Improve scaling of pause times on large Java heaps

2.

3.

Minimize worst-case pause times

4.

5.

NUMA awareness

6.

To achieve the first goal, the heap is split into a number of regions, which are managed and collected independently. Like G1, the Balanced collector wants to manage at most 2048 regions, and so will choose a region size to achieve this. The region size is a power of two, as for G1, but Balanced will permit regions as small as 512KB.

As we would expect from a generational region-based collector, each region has an associated age, with age zero regions (Eden) used for allocation of new objects. When the eden space is full, a collection must be performed. The IBM term for this is a *Partial Garbage Collection (PGC)*.

A PGC is a STW operation that collects all eden regions, and may additionally choose to collect regions with a higher age, if the collector determines that they are worth collecting. In this manner, PGCs are similar to G1's Mixed collections.

Note

Once a PGC is complete, the age of regions containing the surviving objects is increased by 1. These are sometimes referred to as *generational regions*.

Another benefit, as compared to other J9 GC policies is that class unloading can be performed incrementally. Balanced can collect classloaders that are part of the

current collection set during a PGC. This is in contrast to other J9 collectors, where classloaders could only be collected during a global collection.

One downside is that because a PGC only has visibility of the regions it has chosen to collect, this type of collection can suffer from Floating Garbage. To resolve this problem, Balanced employs a *Global Mark Phase (GMP)*. This is a partially concurrent operation that scans the entire Java heap, marking dead objects for collection. Once a GMP completes, the following PGC acts on this data. Thus, the amount of floating garbage in the heap is bounded by the number of objects that died since the last GMP started.

The final type of GC operation that Balanced carried out is *Global Garbage Collection (GGC)*. This is a full, STW collection that compacts the heap. It is similar to the full collections that would be triggered in Hotspot by a Concurrent Mode Failure.

J9 Object headers

The basic J9 object header is a *class slot*, the size of which is 64 bits or 32 bits when Compressed References is enabled.

Note

Compressed References is the default for heaps smaller than 57gb and is similar to Hotspot's `CompressedOops` technique.

However, the header may have additional slots depending on the type of object:

- arrays have either 1 or 2 extra slots for the array length
- monitor slots for synchronized objects
- hashed slots for objects that are put into class library hash structures or internal JVM hash structures

In addition, the monitor and hashed slots are not necessarily adjacent to the object header - they may be stored anywhere in the object taking advantage of otherwise wasted space due to alignment. J9's object layout can be seen in Figure 8-12

= FIXME

Figure 7-12. J9 Object Layout

The highest 24 (or 56) bits of the class slot is a pointer to the class structure, which is off-heap, similarly to Java 8's Metaspace. The lower 8 bits are flags which are used for various purposes depending on the GC policy in use.

Large arrays in Balanced

Allocating large arrays in Java is a common trigger for compacting collections, as enough contiguous space must be found to satisfy the allocation. We saw one aspect of this in the discussion of CMS, where free list coalescing is sometimes insufficient to free up enough space for a large allocation, and a concurrent mode failure results.

For a region-based collector, it is entirely possible to allocate an array object in Java that exceeds the size of a single region. To address this, Balanced uses an alternate representation for large arrays which allows them to be allocated in discontiguous chunks. This representation is known as *arraylets*, and this is the only circumstance under which heap objects can span regions.

Note

The arraylet representation is potentially visible through JNI APIs (although not from regular Java), so the programmer should be aware that when porting JNI code from another JVM, then the spine and leaf representation may need to be taken into account.

The arraylet representation is invisible to user Java code, and instead is handled transparently by the JVM. The allocator will represent a large array as a central object, called the *spine* and a set of array *leaves* that contain the actual entries of the array and which are pointed to by entries of the spine. This enables entries to be read with only the additional overhead of a single indirection. An example can be seen in Figure 8-13

= FIXME

Figure 7-13. Arrays in the Balanced collector

By performing partial GCs on regions, average pause time is reduced, although overall time spent performing GC operations may be higher, due to the overhead of maintaining information about the regions of a referrer and referent.

Crucually, the likelihood of requiring a global STW collection or compaction (the worst case for pause times) is greatly reduced, typically occurring as a last resort when the heap is full.

There is an overhead to managing regions and discontiguous large arrays, and as such, Balanced is suited to applications where avoiding large pauses is more important than outright throughput.

NUMA and Balanced

NUMA (Non-Uniform Memory Architecture) is a memory architecture used in multi-processor systems, typically medium to large servers. In such a system, there is a concept of distance between memory and a processor, with processors and memory arranged into nodes. A processor on a given node may access memory from any node, however access time is significantly faster for local memory (belonging to the same node).

For JVMs that are executing across multiple NUMA nodes, the Balanced collector can split the Java heap across them. Application threads are arranged such that they favour execution on a particular node, and object allocations favour regions in memory local to that node. A schematic view of this arrangement can be seen in Figure 8-14.

= FIXME

Figure 7-14. NUMA and the Balanced collector

In addition, a Partial Garbage Collection (PGC) will attempt to move objects closer (in terms of memory distance) to the objects and threads that refer to them. By doing so, the memory referenced by a thread is more likely to be local, improving performance. This process is invisible to the application.

Legacy Hotspot collectors

In earlier versions of Hotspot, various other collectors were available. We mention them for completeness here, but none of them are recommended for production use any

more, and several combinations have been deprecated as of Java 8, and will be disallowed (or removed) as of Java 9.

Serial and SerialOld

These collectors operated in a similar fashion to the Parallel and ParallelOld collectors, with one important difference - they only used a single CPU core to perform GC. Despite this, they were not concurrent collectors and were still fully STW. On modern multicore systems there is, of course, no possible performance benefit from using these collectors, and so they should not be used.

As a performance engineer, you should be aware of these collectors and their activation switches simply to ensure that if you ever encounter an application that sets them, that you can recognise and remove them.

These collectors were deprecated as part of Java 8, so they will only ever be encountered on earlier, legacy applications that are still using very old versions of Java.

Incremental CMS (iCMS)

The incremental CMS collector, normally referred to as iCMS was an older attempt to do concurrent collection that tried to introduce into CMS some of the ideas that would later result in G1. The switch to enable this mode of CMS was:

```
-XX:+CMSIncrementalMode
```

Some experts still maintain that corner cases exist (for applications deployed on very old hardware with only 1-2 cores) where iCMS can be a valid choice from a performance point of view. However, virtually all modern server-class applications should not use iCMS.

Note

Unless extraordinary evidence exists that your workload will benefit from incremental mode, you should not use it, due to the safepointing behavior and other negatives.

iCMS is anticipated to be removed from Java 9, at time of writing.

Full table of deprecated and removed GC combinations

The current usual cadence of deprecation and removal is that features are marked as deprecated in one Java version and then removed in the next version, or subsequently. Accordingly, in Java 8, these GC flag combinations were marked as deprecated, and will be removed in Java 9

Table 7-1. Deprecated GC combinations

Combination	Flags
DefNew + CMS	-XX:-UseParNewGC -XX:+UseConcMarkSweepGC
ParNew + SerialOld	-XX:+UseParNewGC
ParNew + iCMS	-Xincgc
ParNew + iCMS	-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC
DefNew + iCMS	-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC -XX:-UseParNewGC
CMS foreground	-XX:+UseCMSCompactAtFullCollection
CMS foreground	-XX:+CMSFullGCsBeforeCompaction
CMS foreground	-XX:+UseCMSCollectionPassing

This table should be consulted when starting a new engagement, in order to confirm at the start that the application to be tuned is *not* using a deprecated configuration.

Epsilon

The Epsilon collector is not really a "legacy" collector. However, it is included here because it *must not be used in production under any circumstances*. Whilst the other collectors, if encountered within your environment, should be immediately flagged as extremely high-risk and marked for immediate removal, Epsilon is slightly different.

Epsilon is an experimental collector designed for testing purposes only. It is a *zero-effort* collector. This means it makes no effort to actually collect any garbage.

> Develop a GC that only handles memory allocation, but does not implement any actual memory reclamation mechanism. Once available Java heap is exhausted, perform the orderly JVM shutdown.
>
> Epsilon JEP

Such a "collector" would be very useful for the following purposes:

- Performance testing and microbenchmarks

-
-

Regression testing

-
-

Testing low / zero allocation Java application or library code

-

In particular, JMH tests would benefit from the ability to confidently exclude any GC events from disrupting performance numbers. Memory allocation regression tests - ensuring that changed code does not greatly alter allocation behavior would also become easy to do. Developers could write tests that run with an Epsilon configuration that accepts only a bounded number of allocations, and then fails on any further allocation due to heap exhaustion.

Finally, the proposed VM GC interface would also benefit from having Epsilon as a minimal test case for the interface itself.

Summary

Garbage collection is a truly fundamental aspect of Java performance analysis and tuning. Java's rich landscape of garbage collectors is a great strength of the platform, but it can be intimidating for the newcomer, especially given the scarcity of documentation that considers tradeoffs and performance consequences of each choice.

In this chapter, we have outlined the decisions that face the performance engineer, and the trade-offs that they must face when deciding an appropriate collector to use for their application. We have discussed some of the underlying theory and met a range of modern GC algorithms that implement these ideas.

In the next chapter, we will put some of this theory to work, and introduce logging, monitoring and tooling as a way to bring some scientific rigor to our discussion of performance tuning garbage collection.

[1] Such a system is very difficult to program in, as every object that's created has to be reused, and any object that goes out of scope effectively leaks memory

Chapter 8. GC logging, monitoring, tuning and tools

In this chapter, we will introduce the huge subject of GC logging and monitoring. This is one of the most important and visible aspects of Java performance tuning, and also one of the most often misunderstood.

Introduction to GC logging

The GC log is a great source of information. It is especially useful for “cold case” analysis of performance problems, such as providing some insight into why a crash occured. It is especially useful as it will allow the analyst to work, even with no live application process to diagnose.

Every serious application should always:

- Generate a GC log
- Keep it in a separate file from application output

This is especially true for production applications. As we’ll see, GC logging has no real observable overhead, so it should always be on for any important JVM process.

Switching on GC logging

The first thing to do is to add some switches to the application startup. These are best thought of as the “mandatory GC logging flags”, which should be on for any Java / JVM application (except, perhaps, desktop apps). The flags are:

```
-Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintTenuringDistribution
-XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps
```

Let’s look at each of these flags in more detail. The usage and meaning of each flag can be found in Table 9-1.

Table 8-1. Madatory GC flags

Effect	Flags
Controls which file to log GC events to	-Xloggc:gc.log
Logs GC event details	-XX:+PrintGCDetails
Prints the wallclock time that GC events occured at.	-XX:+PrintGCDateStamps
Prints the time (in secs since VM start) that GC events occured at.	-XX:+PrintGCTimeStamps
Adds extra GC event detail that is vital for tooling	-XX:+PrintTenuringDistribution

The performance engineer should take note of the following detail about some of these flags:

- The `PrintGCDetails` flag replaces the older `verbose:gc` - applications should remove the older flag.
-
- The `PrintTenuringDistribution` flag is different from the others, as the information it provides is not easily usable directly by humans. The flag provides the raw data needed to compute key memory pressure effects and events such as premature promotion.
-
- Both `PrintGCDateStamps` and `PrintGCTimeStamps` are required, as the former is used to correlate GC events to application events (in application log files) and the latter is used to correlate between GC and other internal JVM events.
-

This level of logging detail will not have a measurable effect on the JVM's performance. The amount of logs generated will, of course, depend on many factors, including the allocation rate, collector in use and the heap size - as smaller heaps will need to GC more frequently and so will produce logs more quickly.

To give some idea, a 30 minute run of the model allocator example (as seen in Chapter 7) produces ~600K of logs for a 30 minute run, allocating 50Mb per second.

In addition to the mandatory flags, there are also some flags (shown in Table 9-2) that control GC log rotation, which many application support teams find useful in production environments.

Table 8-2. GC log rotation flags

Effect	Flags
Switches on log file rotation	-XX:+UseGCLogFileRotation
Set the maximum number of log files to keep	-XX:+NumberOfGCLogFiles=<n>
Set the maximum size of each file before rotation	-XX:+GCLogFileSize=<size>

The setup of a sensible log rotation strategy should be done in conjunction with operations staff (including devops). Options for such a strategy, and a discussion of appopriate logging and tools is outside the scope of this book.

GC Logs vs JMX

We have already met the VisualGC tool in Section 6.1.1 - which is capable of displaying a realtime view of the state of the JVM's heap. This tool actually relies on Java's Managment eXtensions (JMX) interface to collect the data from the JVM. A full discussion of JMX is outside the scope of this book, but as far as JMX impacts GC, the performance engineer should be aware of the following:

1.

GC log data is driven by actual garbage collection events, whereas JMX sourced data is obtained by sampling

2.

3.

GC log data is extremely low impact to capture, whereas JMX has implicit proxying and RMI costs.

4.

5.

GC log data contains over 50 aspects of performance data relating to Java's memory management, whereas JMX has less than 10.

6.

Traditionally, one advantage that JMX has had over logs as a performance data source is that JMX can provide streamed data out of the box. However, modern tools, such as jClarity Censum (see Section 9.2) provide APIs to stream GC log data, closing this gap.

Warning

For a rough trend analysis of basic heap usage, then JMX is a fairly quick and easy solution, however for deeper diagnosis of problems it quickly becomes underpowered.

The beans made available via JMX are a standard and are readily accessible. There are some good free tools out there which can already visualise the data for you, e.g. VisualVM / VisualGC as discussed in Section 6.1.1.

Drawbacks of JMX

Clients monitoring an application using JMX will usually rely on sampling the runtime to get an update on the current state. To get a continuous feed of data the client needs to poll the JMX beans in that runtime.

In the case of garbage collection, this causes a problem - the client has no way of knowing when the collector ran. This also means that the state of memory before and after each collection cycle is unknown. This rules out performing a range of deeper and more accurate analysis techniques on the GC data.

Analysis based on data from JMX is still useful, but it is limited to determining long term trends. However, if we want to accurately tune a garbage collector we need to do better. In particular, being able to understand the state of the heap before and after each collection is extremely useful.

Additionally there are a set of extremely important analyses around memory pressure (i.e. allocation rates) that are simply not possible due to the way data is gathered from JMX.

Not only that, but the current implementation of the `JMXConnector` specification relies on RMI. Thus use of JMX is subject to the issues caused when using any RMI based communication channel. These include:

-

Opening ports in firewalls so secondary socket connections can be established

-
-

Using proxy objects to facilitate `remove()` method calls

-
-

Dependency on Java finalization.

-

For a few RMI connections the amount of work involved to close down a connection is minuscule. However, the teardown relies upon finalization. This means that the garbage collector has to run to reclaim the object.

The nature of the lifecycle of the JMX connection means that most often this will result in the RMI object not being collected until a Full GC. See Section 12.5 for more details about the impact of finalization and why it should always be avoided.

By default, any application using RMI will cause Full GC's to be triggered once an hour. For applications that are already using RMI, using JMX will not add to the costs. However, for applications not already using RMI, they necessarily take an additional hit if they decide to use JMX.

Benefits of GC Log Data

Modern day garbage collectors contain many different moving parts which put together result in an extremely complex implementation. So complex is the implementation that the performance of a collector is seemingly difficult if not impossible to predict. These types of software systems are known as emergent, in that their final behavior and performance is a consequence of how all of the systems work and perform together. Different pressures stress different components in different ways resulting a very fluid cost model.

Initially Java's GC developers added GC logging to help debug their implementations and consequently a significant portion of the data produced by the almost than 60 flags that add information to the GC is for performance debugging purposes.

Over time those tasked with tuning the garbage collection process in their applications came to recognized that given the complexities of tuning GC, they also

benefitted from having this exact picture of what was happening in the runtime. Thus being able to collect and read GC logs is now an instrumental part of any tuning regime.

Note

GC Logging is done from within the Hotspot JVM using a non blocking writing mechanism. It has ~0% impact on application performance and should be switched on for all production applications.

Because the raw data in the GC logs can be pinned to specific GC events we can perform all kinds of useful analysis that can give us insights into the costs of the collection and consequently which tuning actions are more likely to produce positive results.

Log Parsing Tools

There is no language or VM spec standard format for GC log messages. This leaves the content of any individual message to the whim of the HotSpot GC development team. Formats can and do change from minor release to minor release.

This is further complicated by the fact that, whilst the simplest log formats are easy to parse, as GC log flags are added the resulting log output becomes massively more complicated. This is especially true of the logs generated by the concurrent collectors.

All too often, systems with hand-rolled GC log parsers experience an outage at some point after making changes to the GC configuration which alters the log output format. When the outage investigation turns to examining the GC logs, the team finds that the homebrew parser cannot cope with the changed log format - failing at the exact point when the log information is most valuable.

Warning

It is not recommended that developers attempt to parse GC logs themselves. Instead, a tool should be used.

In this section, we will examine two actively maintained tools (1 commerical and 1 OSS) that are available for this purpose - there are others, such as GarbageCat that are maintained only sporadically, or not at all.

Censum

The Censum memory analyser is a commercial tool developed by jClarity. It is available both as a desktop tool (for hands-on analysis of a single JVM) and as a monitoring service (for large groups of JVMs). The aim of the tool is to provide best-available GC log parsing, information extraction and also automated analytics.

In Figure 9-1, we can see the Censum desktop view showing allocation rates for a financial trading application running the G1 garbage collector. Even from this simple view we can see that the trading application has periods of very low allocation - corresponding to quiet periods in the market.

Figure 8-1. Censum Allocation View

Other views available from Censum include the pause time graph, shown in Figure 9-2 from the SAAS service version.

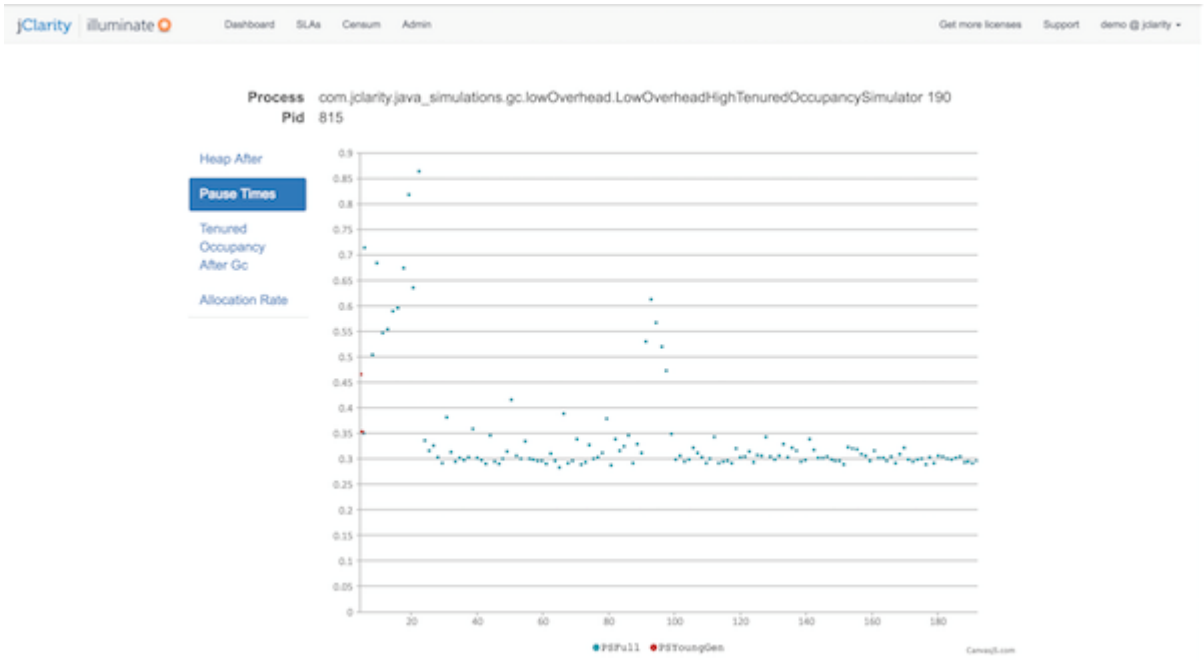


Figure 8-2. Censum Pause Time View

One of the advantages of using the Censum SAAS monitoring service is being able to view the health of an entire cluster simultaneously, as in Figure 9-3. For ongoing monitoring, this is usually easier than dealing with a single JVM at a time.

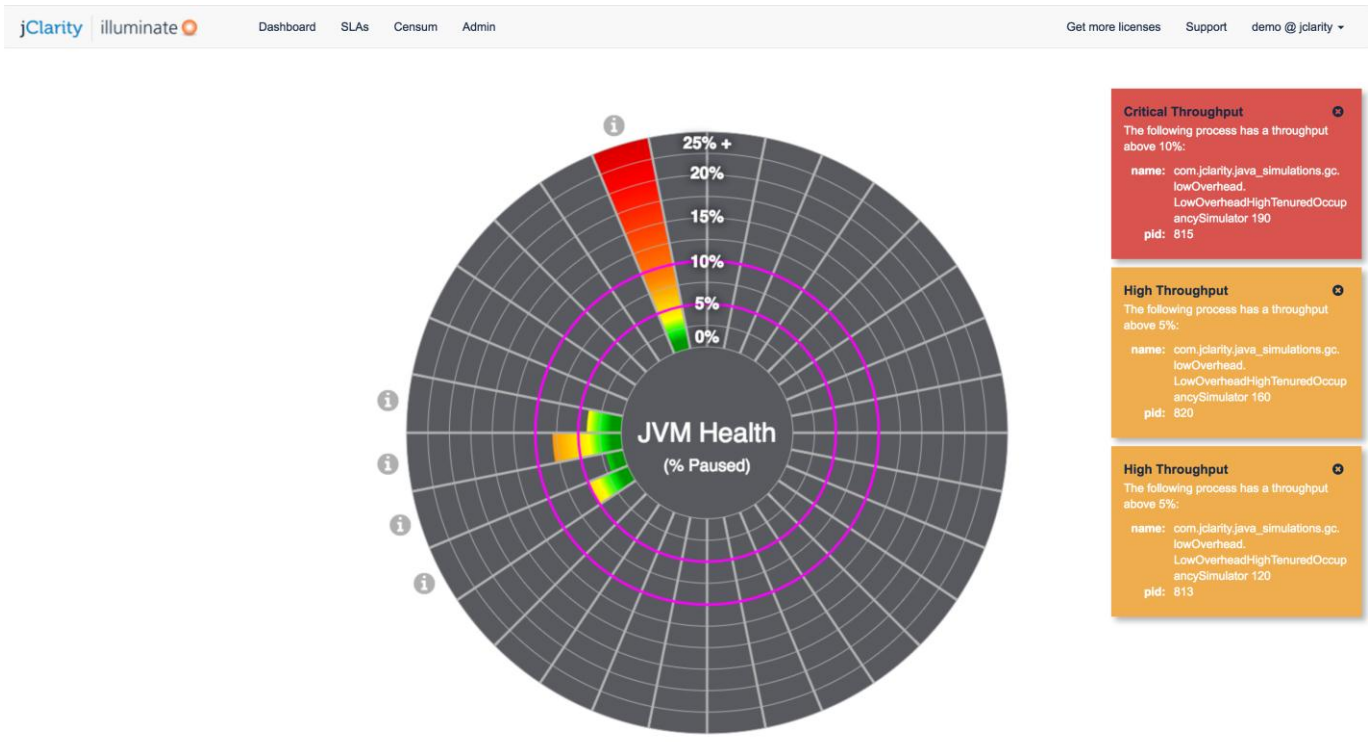


Figure 8-3. Censum Cluster Health Overview

Censum has been developed with close attention to the log file formats and all checkins to OpenJDK that could affect logging are monitored. Censum supports all versions of Sun / Oracle Java from 1.4.2 to 8, all collectors and the largest number of GC log configurations of any tool in the market.

As of the current version, Censum supports these automatic analytics:

- Accurate allocation rates
- Spotting Premature Promotion
- Aggressive or “spiky” allocation
- Idle Churn
- Memory leak detection
- Heap sizing & capacity planning
- OS interference with the VM

Poorly sized memory pools

-

For more details about Censum, and to obtain a trial, go to: https://www.jclarity.com/

GCViewer

GCViewer is a desktop tool that provides a limited subset of the capabilities of Censum. Its big positives are that it is open-source software and is free to use.

The source can be found at: https://github.com/chewiebug/GCViewer - after compiling and building it will package into an executable jar file.

After opening GCViewer, GC log files can then be opened in the main UI. An example is shown in Figure 9-4.

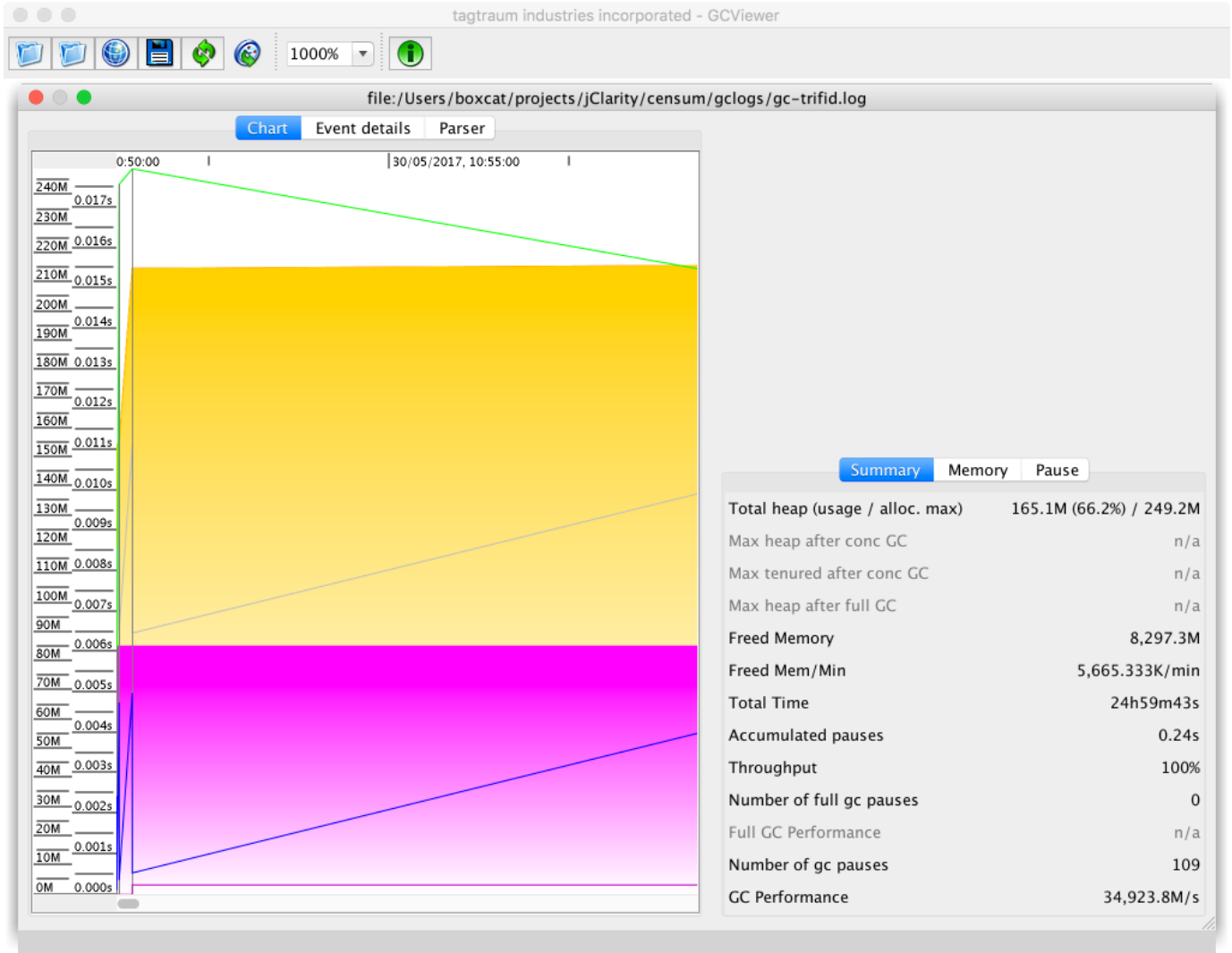


Figure 8-4. GCViewer

GCViewer lacks analytics and can only parse some of the possible GC log formats that Hotspot can produce.

It is possible to use GCViewer as a parsing library, and export the data points into a visualizer, but this requires additional development on top of the existing open-source codebase.

Different visualisations of the same data

You should be aware that different tools may produce different visualisations from each other. The double sawtooth pattern that we met in Section 7.6 was based on a sampled view of the overall size of the heap as the observable.

When the GC log that produced this pattern is plotted using the "Heap Occupancy after GC" view of GCViewer, we end up with a visualisation as shown in Figure 9-5.

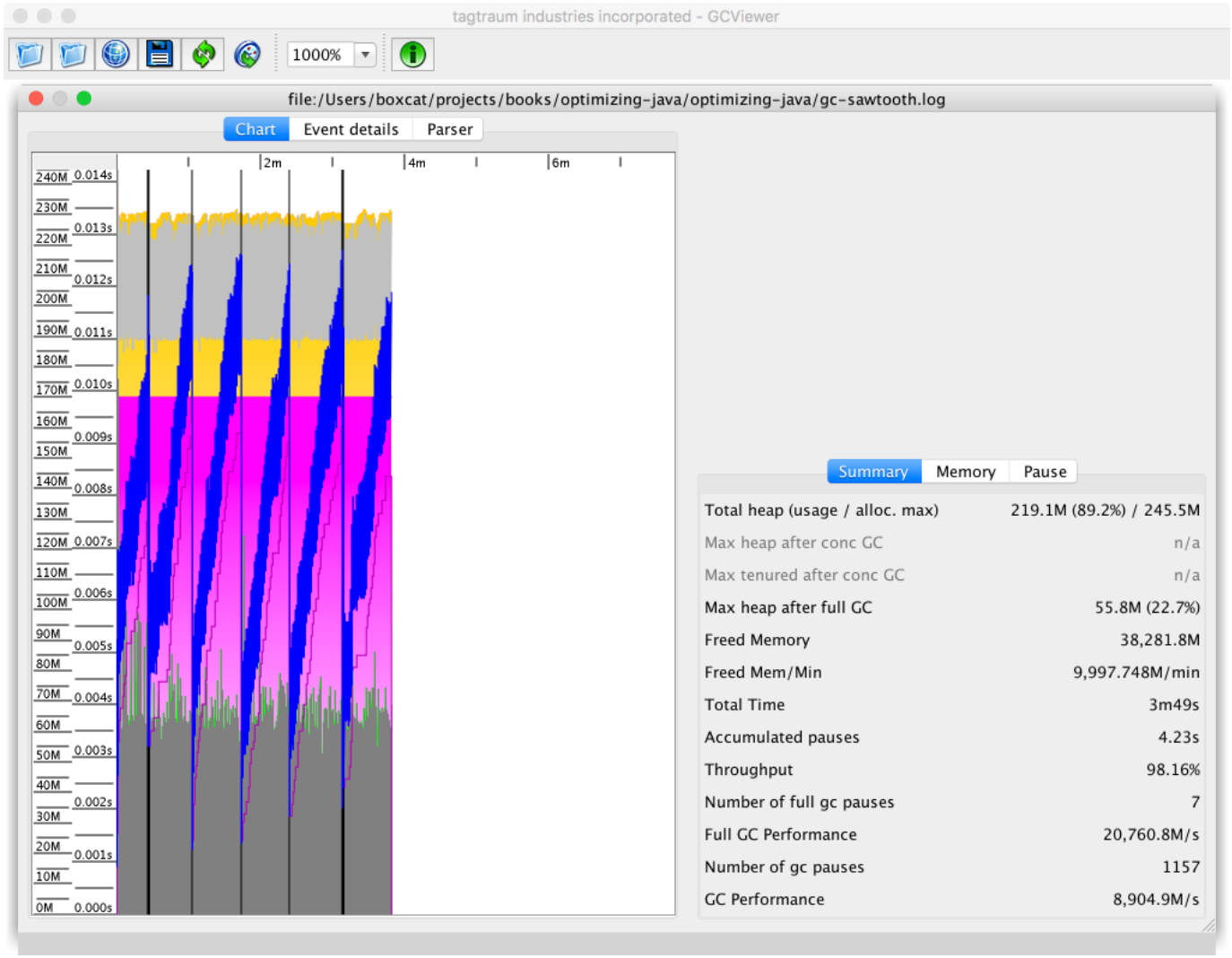


Figure 8-5. Simple sawtooth in GCViewer

Let's look at the same simple sawtooth pattern displayed in Censum. Figure 9-6 shows the result.

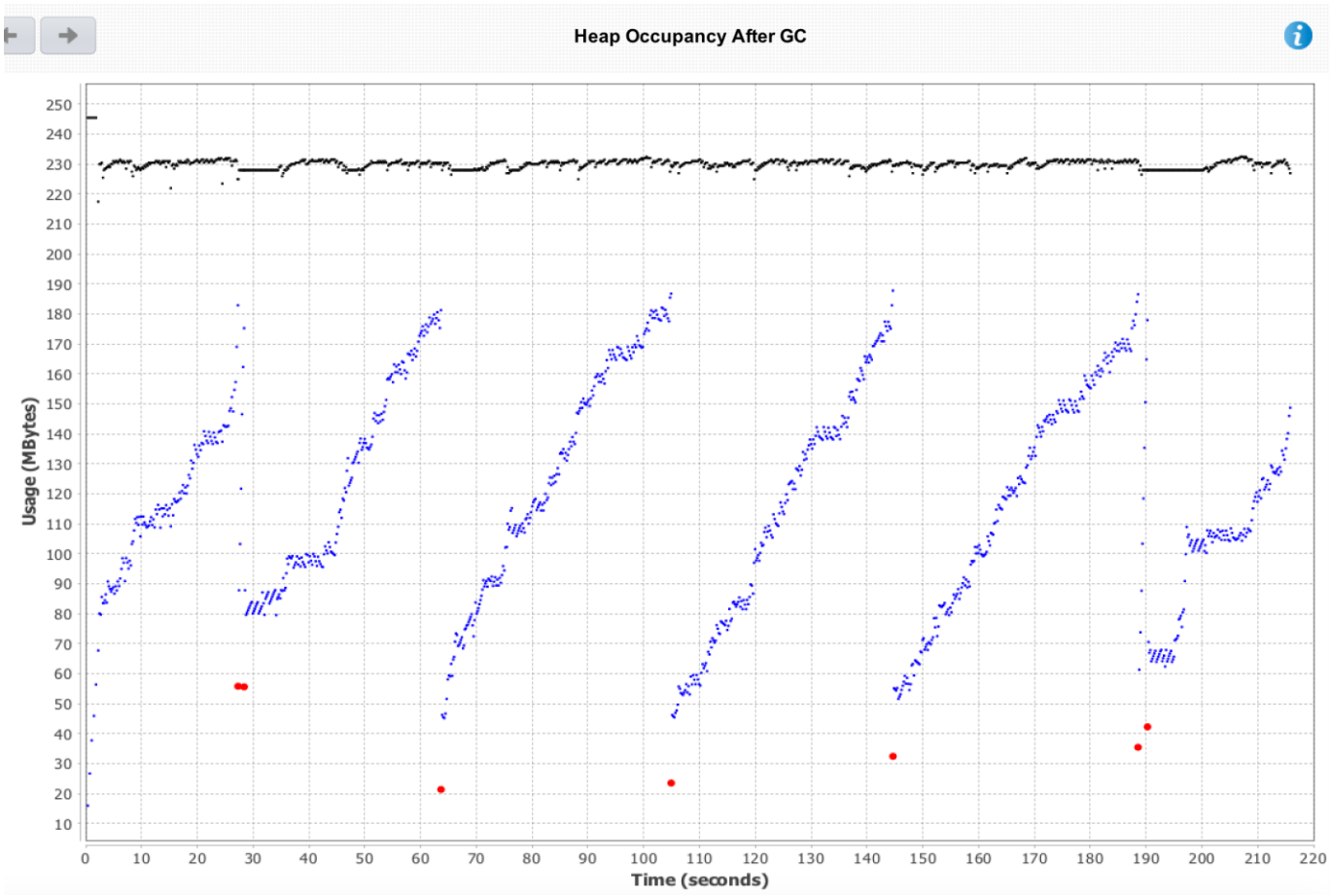


Figure 8-6. Simple sawtooth in Censum

Whilst the images and visualisations are different, the message from the tool is the same in both cases – that the heap is operating normally.

Basic GC Tuning

When considering strategies for tuning the JVM, the question "When should I tune GC?" often arises. As with any other tuning technique, GC tuning should form part of an overall diagnostic process. The following facts about GC tuning are very useful to remember:

1.

It's cheap to eliminate or confirm GC as the cause of a performance problem

2.

3.

It’s cheap to switch on GC flags in UAT

4.

5.

It’s not cheap to setup memory or execution profilers

6.

The engineer should also know that there are four main factors should be studied and measured during a tuning exercise:

-

Allocation

-
-

Pause sensitivity

-
-

Throughput behavior

-
-

Object Lifetime

-

Of these, allocation is very often the most important.

Note

Throughput can be affected by a number of factors, such as the fact that concurrent collectors take up cores when they’re running.

Let's take a look at some of the basic heap sizing flags, as shown in Table 9-3:

Table 8-3. GC heap sizing flags

Effect	Flag
Set the minimum size reserved for the heap	-Xms<size>
Set the maximum size reserved for the heap	-Xmx<size>
Set the maximum size permitted for PermGen (Java 7)	-XX:MaxPermSize=<size>
Set the maximum size permitted for Metaspace (Java 8)	-XX:MaxMetaspaceSize=<size>

The `MaxPermSize` flag is legacy and only applies to Java 7 and before. In Java 8 and up, the PermGen has been removed, and replaced by Metaspace.

Note

If you are setting MaxPermSize on a Java 8 application, then you should just remove the flag. It is being ignored by the JVM anyway, so it clearly is having no impact on your application.

On the subject of additional GC flags for tuning:

-

Only add flags one at a time

-
-

Make sure that you fully understand the effect of each flag

-
-

Recall that some combinations produce side effects

-

To check whether GC is the cause of a performance issue is relatively easy, assuming the event is happening live. The first step is to look at the machine's high-level metrics using vmstat or a similar tool, as discussed in Section 3.6. First, log on to the box having the performance issue and check that:

1.

CPU utilisation is close to 100%.

2.

3.

Vast majority of this time (90+%) is being spent in user space

4.

5.

The GC log is showing activity, indicating that GC is currently running

6.

This assumes that the issue is happening right now and can be observed by the engineer in real time. For past events, sufficient historical monitoring data must be available (including CPU utilisation and GC logs that have timestamps).

If all three of these conditions are met, GC should be investigated and tuned, as the most likely cause of the current performance issue. The test is very simple, and has a nice clear result - either that “GC is OK” or “GC is not OK”.

If GC is indicated as a source of performance problems, then the next step is to understand the allocation and pause time behavior, then tune the GC and potentially, bring out the memory profiler if required.

Understanding Allocation

Allocation rate analysis is extremely important for determining not just how to tune the garbage collector, but whether or not you can actually tune the garbage collector to help improve performance at all.

We can use the data from both the young generational collection events to calculate the amount of data allocated and the time between the two collections. This information can then be used to calculate the average allocation rate during that time interval.

Note

Rather than spending time and effort manually calculating allocation rates, it is usually better to use tooling to provide this number.

Experience shows that sustained allocation rates of greater than 1GB/sec are almost always indicative of performance problems that cannot be corrected by tuning the garbage collector. The only way to improve performance in these cases is to improve the memory efficiency of the application by refactoring to eliminate allocation in critical parts of the application.

A simple memory histogram, such as that shown by VisualVM (as seen in Section 6.1) or even `jmap` (Section 7.1) can serve as a starting point for understanding memory allocation behavior. A useful initial allocation strategy is to concentrate on four simple areas:

-

Trivial, avoidable object allocation (e.g. log debug messages)

-
-

Boxing costs

-
-

Domain objects

-
-

Large numbers of non-JDK framework objects

-

For the first of these, it is simply a matter of spotting, and removing unnecessary object creation. Excessive boxing can be a form of this, but other examples (such as autogenerated code for serialising / deserialising to JSON, or ORM code) are also possible sources of wasteful boxing object creation.

It is unusual for domain objects to be a major contributor to the memory utilisation of an application. Far more common are types such as:

-

`char[]` - the characters that comprise Strings

-
-

`byte[]` - raw binary data

-
-

`double[]` - calculation data

-
-

Map entries

-
-

`Object[]`

-
-

Internal data structures (such as methodOops and klassOops)

-

A simple histogram can often show up leaking or excessive creation of unecessary domain objects, simply by their presence among the top elements of a heap histogram. Often, all that is necessary is a quick calculation to figure out the expected data volume of domain objects, to see whether the observed volume is in line with expectations or not.

In Section 7.4.1 we met thread-local allocation. The aim of this technique is to provide a private area for each thread to allocate new objects, and thus achieve `O(1)` allocation.

The TLABs are sized dynamically on a per-thread basis, and regular objects are allocated in the TLAB, if there's space. If not, the thread requests a new TLAB from the VM and tries again.

If the object will not fit in an empty TLAB, the VM will next try to allocate the object directly in Eden, in an area outside of any TLAB. If this fails, the next step is to perform a Young GC (which might resize the heap). Finally, if this fails and there's still not enough space, the last resort is to allocate the object directly in Tenured.

From this we can see that the only objects that are really likely to end up being directly allocated in Tenured are large arrays (especially byte and char arrays).

Hotspot has a couple of tuning flags that are relevant to the TLABs and the pretenuring of large objects:

-XX:PretenureSizeThreshold=<N>
-XX:MinTLABSize=<N>

As with all switches, these should not be used without a benchmark and solid evidence that they will have an effect. In most cases, the builtin dynamic behavior will produce great results, and any changes will have little to no real observable impact.

Allocation rates also affect the number of objects that are promoted to Tenured. If we assume that short-lived Java objects have a fixed lifetime (when expressed in wallclock time), then higher allocation rates will result in young GCs that are closer together. If the collections become too frequent then short-lived objects may not have had time to die and will be erroneously promoted to Tenured.

To put it another way, spikes in allocation can lead to the premature promotion problem that we met in Section 8.4.2. To guard against this, the JVM will resize survivor spaces dynamically to accomodate greater amounts of surviving data without promoting it to Tenured.

One JVM switch that can sometimes be helpful when dealing with tenuring problems and premature promotion is:

-XX:MaxTenuringThreshold=N

This controls the number of garbage collections that an object must survive to be promoted into Tenured. It defaults to 4 but can be set anywhere from 1 to 15. Changing this value represents a tradeoff between two concerns:

-

The higher the threshold, the more copying of genuinely long-lived objects will occur

-
-

If the threshold is too low, some short-lived objects will be promoted increasing memory pressure on Tenured.

-

One consequence of too low a threshold could be that full collections occur more often, due to the greater volume of objects being promoted to Tenured, causing it to fill up more quickly. As always, the switch should not be altered without a clear benchmark indicating that performance is improved by setting it to a non-default value.

Understanding Pause Time

Developers frequently suffer from cognitive bias about pause time. Many applications can easily tolerate 100+ms pause times. The human eye can only process 5 updates per second of a single data item, so a 100-200ms pause is below the threshold of visibility for most human-facing applications (such as web apps).

One useful heuristic for pause time tuning is to divide applications into three broad bands. These bands are based on the application's need for responsiveness, expressed as the pause time that the application can tolerate. They are:

1.

>1s: Can tolerate over 1s of pause

2.

3.

1s - 100ms: Can tolerate more that 200ms but less than 1s of pause

4.

5.

< 100ms: Cannot tolerate 100ms of pause.

6.

If we combine the pause sensitivity with the expected heap size of the application, we can construct a simple table of best guesses at a suitable collector. The result is shown in Table 9-4:

Table 8-4. Initial collector choice

	Pause Time Tolerance		
	>1s	1s - 100ms	<100ms
< 2G	Parallel	Parallel	CMS
< 4G	Parallel	Parallel / G1	CMS
< 4G	Parallel	Parallel / G1	CMS
< 10G	Parallel / G1	Parallel / G1	CMS
< 20G	Parallel / G1	G1	CMS
> 20G	G1	G1	CMS / G1

These are guidelines and rules of thumb intended for a starting point for tuning, not 100% unambiguous rules.

Looking into the future, as G1 matures as a collector, then it is reasonable to expect that it will expand to cover more of the use cases currently covered by Parallel. It is also possible, but perhaps less likely, that it will also expand to CMS's use cases as well.

Tip

When a concurrent collector is in use, allocation should still be reduced before attempting to tune pause time.

Reduced allocation means that there is less memory pressure on a concurrent collector - the collection cycles will find it easier to keep up with the allocating threads. This will reduce the possibility of a concurrent mode failure, which is usually an event that pause-sensitive applications need to avoid if at all possible.

Collector Threads and GC Roots

One useful mental exercise is to "think like a GC thread". This can provide insight into how the collector behaves under various circumstances. However, as with so many other aspects of GC, there are fundamental tradeoffs in play. These tradeoffs include that the scanning time needed to locate GC roots is affected by factors including:

-

Number of application threads

-
-

Amount of compiled code in the code cache

-
-

Size of heap

-

Even for this single aspect of GC, which one of these events dominates the GC root scanning will always depends on run time conditions and the amount of parallelization that can be applied.

For example, consider the case of a large `Object[]` discovered in the Mark phase. This will be scanned by a single thread - no work stealing is possible. In the extreme case, this single-threaded scanning time will dominate the overall mark time.

In fact, the more complex the object graph, the more pronounced this effect becomes - meaning that marking time will get even worse the more "long chains" of objects exist within the graph.

High numbers of application threads will also have an impact on GC times - as they represent more stack frames to scan and more time needed to reach a safepoint. They also exert more pressure on thread scheduler in both bare metal and virtualised environments.

As well as these traditional examples of GC roots, there are also other sources of GC roots - including JNI frames and the code cache for JIT-compiled code (which we will meet properly in Section 10.4)

Warning

GC root scanning in the code cache is single threaded (at least for Java 8).

Of the three effects, stack and heap scanning is reasonably well paralyzed. Generational collectors also keep track of roots that originate in other memory pools. These include the RSets in G1 and the card tables in Parallel and CMS.

For example, let's consider the card tables, introduced in Section 7.3.1. These are used to indicate a block of memory that has a reference back into YoungGen from

OldGen. As each byte represents 512 bytes of OldGen, it's clear that for each 1G of OldGen memory, 2M of card table must be scanned.

To get a feel for how long a card table scan will take, consider a simple benchmark that simulates scanning a card table for a 20G heap:

```
@State(Scope.Benchmark)@BenchmarkMode(Mode.Throughput)@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)@OutputTimeUnit(TimeUnit.SECONDS)@Fork(1)public class SimulateCardTable {

    // OldGen is 3/4 of heap, 2M of card table is required for 1G of old gen
    private static final int SIZE_FOR_20_GIG_HEAP = 15 * 2 * 1024 * 1024;

    private static final byte[] cards = new byte[SIZE_FOR_20_GIG_HEAP];

    @Setup
    public static final void setup() {
        final Random r = new Random(System.nanoTime());
        for (int i=0; i<100_000; i++) {
            cards[r.nextInt(SIZE_FOR_20_GIG_HEAP)] = 1;
        }
    }


    @Benchmark
    public int scanCardTable() {
        int found = 0;
        for (int i=0; i<SIZE_FOR_20_GIG_HEAP; i++) {
            if (cards[i] > 0)
                found++;
        }
        return found;
    }
}
```

The results for running this benchmark show an output similar to:

```
Result "scanCardTable":
  108.904 ±(99.9%) 16.147 ops/s [Average]
  (min, avg, max) = (102.915, 108.904, 114.266), stdev = 4.193
  CI (99.9%): [92.757, 125.051] (assumes normal distribution)


# Run complete. Total time: 00:01:46
```

```
Benchmark                              Mode  Cnt    Score    Error  Units
SimulateCardTable.scanCardTable        thrpt    5  108.904 ± 16.147  ops/s
```

This gives that it takes about 10ms to scan the card table for a 20G heap - this is, of course, the result for a single-threaded scan. However, it does provide a useful rough lower bound for the pause time for young collections.

We've examined some general techniques that should be applicable to tuning most collectors, so now let's move on to look at some collector-specific approaches.

Tuning Parallel

Parallel is the simplest of the collectors, and so it should be no surprise that it is also the easiest to tune. However, it usually requires minimal tuning. The goals and the tradeoffs of Parallel are clear:

- Fully STW
- High GC throughput / computationally cheap
- No possibility of a partial collection
- Linearly growing pause time in the size of the heap

If your application can tolerate the characteristics of Parallel, then it can be a very effective choice - especially on small heaps, such as those under ~4G.

Older applications may have made use of explicit sizing flags to control the relative size of various memory pools. These flags are summarized in Table 9-5:

Table 8-5. Older GC heap sizing flags

Effect	Flag
(Old flag) Set ratio of YoungGen to Heap	-XX:NewRatio=N
(Old flag) Set ratio of Survivor spaces to YoungGen	-XX:SurvivorRatio=N
(Old flag) Set min size of YoungGen	-XX:NewSize=N
(Old flag) Set max size of YoungGen	-XX:MaxNewSize=N
(Old flag) Set min % of heap free after GC to avoid expanding	-XX:MinHeapFreeRatio
(Old flag) Set max % of heap free after GC to avoid shrinking	-XX:MaxHeapFreeRatio

The survivor ratio, new ratio and overall heap size are connected by the following formula:

```
Flags set:
-XX:NewRatio=N-XX:SurvivorRatio=K
YoungGen = 1 / (N+1) of heapOldGen = N / (N+1) of heap
Eden = (K - 2) / K of YoungGenSurvivor1 = 1 / K of YoungGenSurvivor2 = 1 / K of
YoungGen
```

For most modern applications, these types of explicit sizing should not be used, as the ergonomic sizing will do a better job than humans in almost all cases. For Parallel, resorting to these switches is something close to a last resort.

Tuning CMS

The CMS collector has a reputation of being difficult to tune. This is not wholly undeserved - the complexities and trade-offs involved in getting the best performance out of CMS are not to be underestimated.

The simplistic position that “pause time bad, therefore concurrent marking collector good” is unfortunately common among many developers. A low-pause collector like CMS should really be seen as a last resort - only to be used if the use case genuinely requires low STW pause times. To do otherwise may mean that a team is lumbered with a collector that is difficult to tune and provides no real tangible benefit to the application performance.

CMS has a very large number (almost 100 as of Java 8u131) of flags, and some developers may be tempted to change the values of these flags in an attempt to improve performance. However, this can easily lead to several of the antipatterns that we met in Section 4.4 including:

-

Fiddle With Switches

-
-

Tuning By Folklore

-
-

Missing the Bigger Picture

-

The conscientous performance engineer should resist the temptation to fall prey to any of these cognitive traps.

Warning

The majority of applications that use CMS will probably not see any real observable improvement by changing the values of CMS command-line flags.

Despite this danger, there are some circumstances where tuning is necessary to improve (or obtain acceptable) CMS performance. Let’ s start by considering the throughput behavior.

Whilst a CMS collection is running, by default half the cores are doing GC. This inevitably causes application throughoput to be reduced. One useful rule of thumb can be to consider the situation of the collector just before a CMF occurs.

In this case, as soon as a CMS collection has finished, a new CMS collection immediately starts. This is known as a *back-to-back* collection, and in the case of a concurrent collector, it indicates the point at which concurrent collection is about to break down. If the application allocates any faster, then reclamation will be unable to keep up and the result will be a CMF.

In the back-to-back case, throughput will be reduced by 50% for essentially the entire run of the application. When undertaking a tuning exercise, the engineer should consider whether the application can tolerate this worst-case behavior. If not, then the solution may be to run on a host with more cores available.

An alternative is to reduce the number of cores assigned to GC during a CMS cycle. Of course, this is dangerous as it reduces the amount of CPU time available to perform collection, and so will make the application less resilient to allocation spikes (and in turn may make it more vulnerable to a CMF). The switch that controls this is:

-XX:ConcGCThreads=<n>

It should be obvious that if an application cannot reclaim fast enough with the default setting, then reducing the number of GC threads will only make matters worse.

CMS has two separate STW phases:

- Initial Mark - marks the immediate interior nodes - those that are directly pointed at by GC roots
- Remark - use the card tables to identify objects that may require fixup work

This means that all app threads must stop, and hence safepoint, twice per CMS cycle. This effect can become important for some low-latency apps that are sensitive to safepointing behavior.

Two flags that are sometimes seen together are:

-XX:CMSInitiatingOccupancyFraction=<n>
-XX:+UseCMSInitiatingOccupancyOnly

These are flags that can be very useful. They also illustrates, once again, the importance and dilemma presented by unstable allocation rates.

The initiating occupancy flag is used to determine at what point CMS should begin collecting. Some heap headroom is required so that there's spare space for objects to be promoted into from young collections that may occur whilst CMS is running.

Like many other aspects of Hotspot's approach to GC, the size of this spare space is controlled by the statistics collected by the JVM itself. However, an estimate for the first run of CMS is still required. The headroom size for this initial guess is controlled by the flag `CMSInitiatingOccupancyFraction`. The default for this flag means that the first CMS full GC will begin when the heap is 75% full.

If the `UseCMSInitiatingOccupancyOnly` flag is also set, then the dynamic sizing for the initiating occupancy is turned off. This should not be done lightly, and it is rare in practice to decrease the headroom (increase the parameter value above 75%).

However, for some CMS applications that have very bursty allocation rates, one strategy would be to increase the headroom (decrease the parameter) whilst turning off the adaptive sizing. The aim here would be to try to reduce concurrent mode failures, at the expense of CMS concurrent GCs occuring more often.

Concurrent Mode Failures due to fragmentation

Let's look at another case where the data we need to perform a tuning analysis is only available in the GC logs. In this case we want to use the *Free List Statistics* to predict when the JVM might suffer from a CMF due to heap fragmentation. This type of CMF is caused by the free list that CMS maintains which we met in Section 8.2.1.

We will need to turn on an additional JVM switch to see the additional output:

-XX:PrintFLSStatistics=1

The output that's produced in the GC log looks like this (detail of output statistics shown for a BinaryTreeDictionary benchmark):

Total Free Space: 40115394

Max Chunk Size: 38808526

Number of Blocks: 1360

Av. Block Size: 29496

Tree Height: 22

In this case we can get a sense of size distributions of chunks of memory, from the average size and the max chunk size. If we run out of chunks large enough to support moving a large live object into Tenured then a GC promotion will degrade into this concurrent failure mode.

In order to compact heap and coalesce the free space list, the JVM will fall back to the Parallel collector, probably causing a long STW pause. This analysis is useful when performed in real time as it can signal that a long pause is imminent. It can observed by parsing the logs, or using a tool like Censum that can automatically detect the approaching CMF.

Tuning G1

The overall goal of tuning G1 is to allow the end user to simply set maximum heap size and `MaxGCPauseMillis`, and have the collector take care of everything else. However, the current reality is still some way from this.

Like CMS, G1 comes with a large number of configuration options, some of which are still experimental and not fully surfaced within the VM (in terms of visible metrics for tuning). This makes it difficult to understand the impact of any tuning changes. If these options are needed for tuning (and currently they are for some tuning scenarios), then this switch must be specified:

```
-XX:+UnlockExperimentalVMOptions
```

In particular, this option must be specified if the options `-XX:G1NewSizePercent=<n>` or `-XX:G1MaxNewSizePercent=<y>` are to be used. At some point in the future, some of these options may become more mainstream, and not require the experimental options flag, but there is no roadmap for this yet.

In Section 8.4.1, we met the Regions app. This is a small, open-source Java FX application that parses a GC log and provides a way to visualise the region layout of a G1 heap over the lifetime of a GC log. The tool can be obtained from https://github.com/kcpeppe/regions and at time of writing, is still under very active development. A sample view can be seen in Figure 9-7.

= FIXME

Figure 8-7. G1 regions

One of the major problems with G1 tuning is that the internals have changed considerably over the early life of the collector. This has led to a significant problem with Tuning By Folklore, as many of the earlier articles that were written about G1 are now of limited validity.

As G1 is intended to become the default collector with the arrival of Java 9, performance engineers will surely be forced to address the topic of how to tune G1, but as of writing this chapter it remains an often frustrating task where best practice is still emerging.

However, let's finish this section where some progress has been made, and where G1 does offer the promise of exceeding CMS. Recall that CMS does not compact, so over time the heap can fragment. This will lead to a concurrent mode failure eventually and the JVM will need to perform a full, parallel collection (with the possibility of a significant STW pause).

In the case of G1, provided the collector can keep up with the allocation rate, then the incremental compaction offers the possibility of avoiding a CMF at all. An application that has a high, stable allocation rate and which creates mostly short-lived options should therefore:

-

Set a large young generation

-
-

Increase the Tenuring Threshold, probably to the maximum (15)

-
-

Set the longest pause time goal that the app can tolerate

-

In this configuration, Eden and survivor spaces have been configured to give the best chance that genuinely short-lived objects are not promoted. This reduces pressure on the old generation and reduces the need to clean old regions. There are rarely any guarantees in GC tuning, but this is an example workload where G1 may do significantly better than CMS, albeit at the cost of some effort to tune the heap.

jHiccup

We have already met HdrHistogram in Section 5.3.2. A related tool is *jHiccup*, available from https://github.com/giltene/jHiccup. This is an instrumentation tool that is designed to show "hiccups" where the JVM is not able to run continuously. One very common cause of this is GC STW pauses, but other OS and platform related effects can also cause a hiccup. It is therefore useful not only for GC tuning but for ultra-low-latency work.

In fact, we have also already seen an example of how jHiccup. In Section 8.5 we introduced the Shenandoah collector, and showcased a graph of the collectors performance as compared to others. The graph of comparative performance in Figure 8-9 was produced by JHiccup.

Note

jHiccup is an excellent tool for use when tuning Hotspot, even though the original author (Gil Tene) cheerfully admits that it was written to show up a perceived shortcoming of Hotspot as compared to Azul's Zing JVM.

jHiccup is usually used as a Java agent, by adding the `-javaagent:jHiccup.jar` switch to a Java command line. It can also be used via the Attach API (like some of the other command line tools) and the form taken for this is:

```
jHiccup -p <pid>
```

This effectively injects jHiccup into a running application.

jHiccup produces output in the form of a histogram log, that can be ingested by HdrHistogram. Let's take a look at this in action, starting by recalling the model allocation application introduced in Section 7.6.

To run this with a set of decent GC logging flags as well as jHiccup running as an agent, let's look at a small piece of shell scripting to set up the run:

```
#!/bin/bash

# Simple script for running jHiccup against a run of the model toy allocator

CP=./target/optimizing-java-1.0.0-SNAPSHOT.jar

JHICCUP_OPTS=-javaagent:~/.m2/repository/org/jhiccup/jHiccup/2.0.7/jHiccup-2.0.7.jar
```

```
GC_LOG_OPTS="-Xloggc:gc-jHiccup.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps
-XX:+PrintGCTimeStamps -XX:+PrintTenuringDistribution"

MEM_OPTS="-Xmx1G"

JAVA_BIN=`which java`

if [ $JAVA_HOME ]; then
    JAVA_CMD=$JAVA_HOME/bin/java
elif [ $JAVA_BIN ]; then
    JAVA_CMD=$JAVA_BIN
else
    echo "For this command to run, either $JAVA_HOME must be set, or java must be
    in the path."
    exit 1
fi

exec $JAVA_CMD -cp $CP $JHICCUP_OPTS $GC_LOG_OPTS $MEM_OPTS optjava.ModelAllocator
```

This will produce both a GC log and a `.hlog` file suitable to be fed to the `jHiccupLogProcessor` tool that ships as part of jHiccup. A simple, out-of-the-box jHiccup view is shown in Figure 9-8.

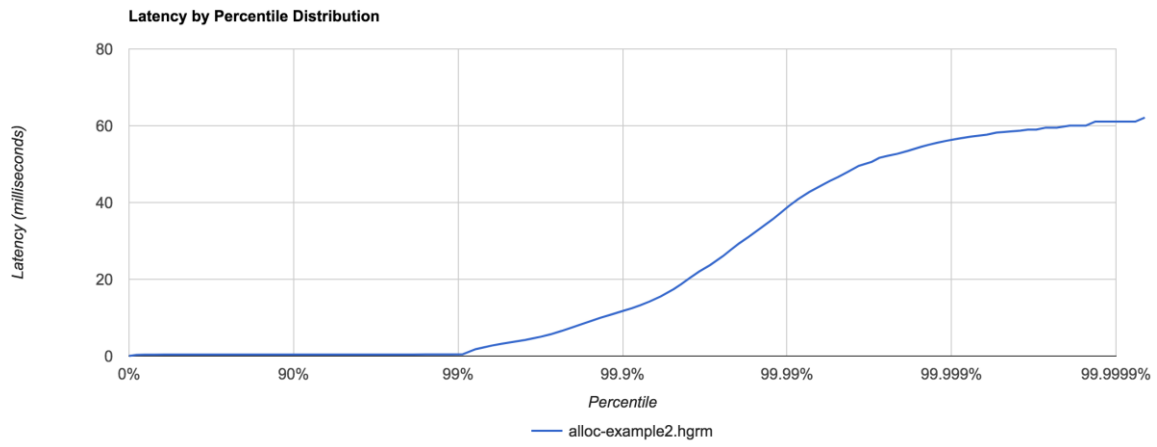


Figure 8-8. jHiccup view of ModelAllocator

This was obtained be a very simple invocation of jHiccup:

```
jHiccupLogProcessor -i hiccup-example2.hlog -o alloc-example2
```

There are some other useful switches - to see all available options just use:

jHiccupLogProcessor -h

It is very often the case that performance engineers need to comprehend multiple views of the same application behaviour, so for completeness, let's see the same run as seen from Censum, in Figure 9-9.

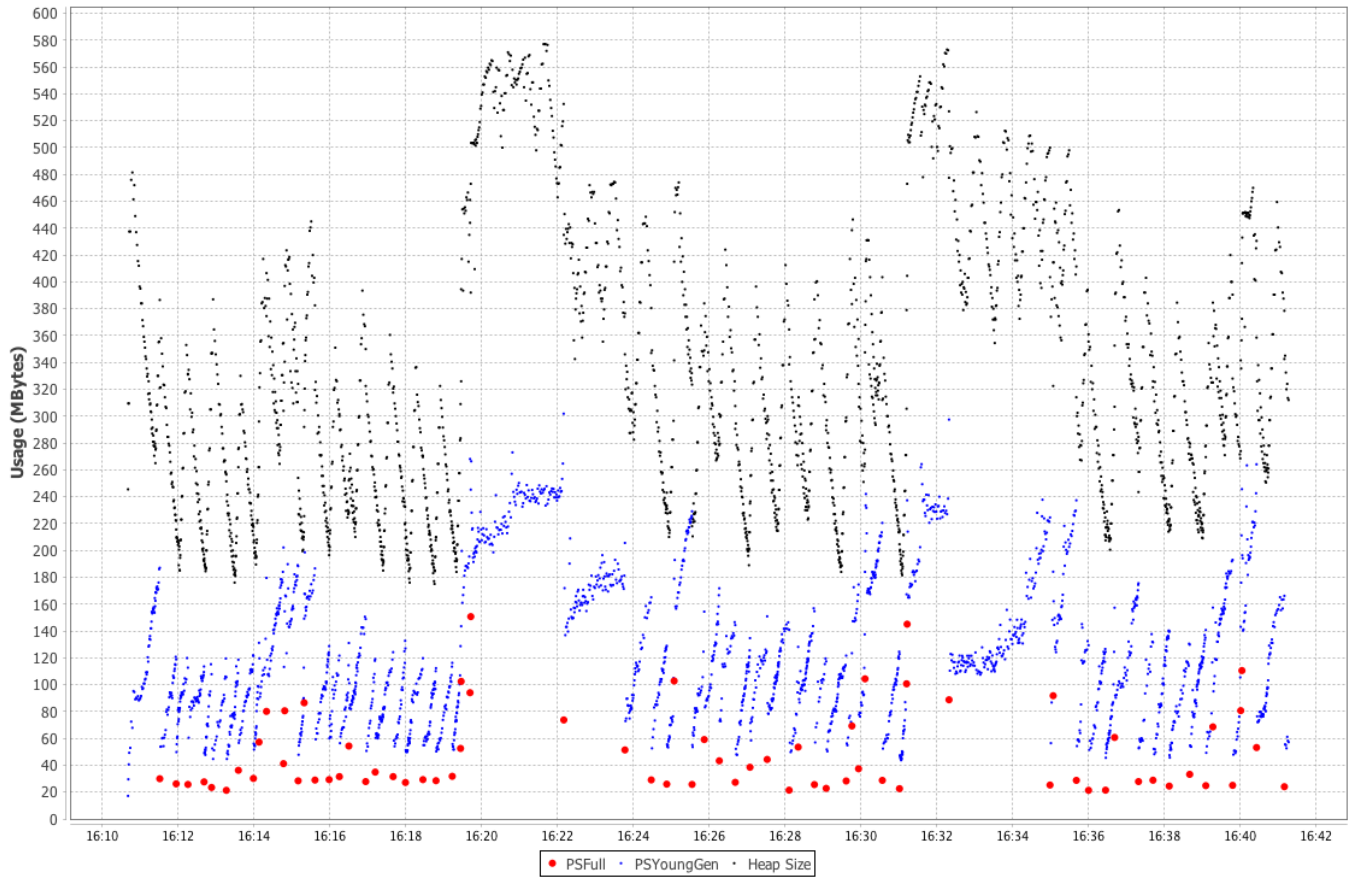


Figure 8-9. Censum view of ModelAllocator

This graph of heap size after GC reclamation shows Hotspot trying to resize the heap and not being able to find a stable state. This is a frequent circumstance, even for applications as simple as ModelAllocator. The JVM is a very dynamic environment and for the most part developers should avoid being overly concerned with the low-level detal of GC ergonomics.

Finally, some very interesting technical detail about both HdrHistogram and jHiccup can be found in a great blog post by Nitsan Wakart at: http://psy-lob-saw.blogspot.in/2015/02/hdrhistogram-better-latency-capture.html

Summary

In this chapter we have scratched the surface of the art of GC tuning. The techniques we have shown are mostly highly specific to individual collectors, but there are some underlying techniques that are generally applicable. We have also covered some basic principles for handling GC logs, and met some useful tools.

Moving onwards, in the next chapter we will turn to one of the other major subsystems of the JVM - execution of application code. We will begin with an overview of the interpreter and then proceed from there to a discussion of JIT compilation, including how it relates to standard (or "ahead of time") compilation. Towards the end of the next chapter, we will take a deep dive into the mechanisms used by Hotspot and how to use tooling to analyse and the compilation behavior.