O'REILLY®

Optimizing Java

PRACTICAL TECHNIQUES FOR IMPROVED PERFORMANCE TUNING

Benjamin J. Evans & James Gough

Optimizing Java

Author Name

Optimizing Java

by Benjamin J Evans and James Gough

Copyright © 2016 Benjamin Evans, James Gough. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (**http://safaribooksonline.com**). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

- Editor: Brian Foster
- Production Editor: FILL IN PRODUCTION EDITOR
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest

- January -4712: First Edition

Revision History for the First Edition

- 2016-02-23: First Early Release

See **http://oreilly.com/catalog/errata.csp?isbn=9781491933251** for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Optimizing Java, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-93325-1

[FILL IN]

Table of Contents

Preface v

CHAPTER 1: Optimization and Performance Defined **7**

Java Performance - The Wrong Way 7

Performance as an Experimental Science 8

A Taxonomy for Performance 9

Throughput 10

Latency 10

Capacity 10

Utilisation 10

Efficiency 11

Scalability 11

Degradation 11

Connections between the observables 12

Reading performance graphs 13

CHAPTER 2: Overview of the JVM **19**

Overview 19

Code Compilation and Bytecode 19

Interpreting and Classloading 24

Introducing HotSpot 25

JVM Memory Management 27

Threading and the Java Memory Model 28

The JVM and the operating system 29

CHAPTER 3: Hardware & Operating Systems **31**

Introduction to Modern Hardware 32

Memory 32

Memory Caches 34

Modern Processor Features 38

Translation Lookaside Buffer 38

Branch Prediction and Speculative Execution 38

Hardware Memory Models 39

Operating systems 40

The Scheduler 41

A Question of Time 42

Context Switches 43

A simple system model 45

Basic Detection Strategies 46

Context switching 47

Garbage Collection 48

I/O 49

Kernel Bypass I/O 49

Virtualisation 51

Preface

Java

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 Googles 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. 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.

> The execution speed of Java code is highly dynamic and fundamentally depends on the underlying Java Virtual Machine (JVM). The same 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'll 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:

- Methodology and performance testing 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.

> 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 more than 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
- "Tips and tricks"
- Secret algorithms that have been hidden from the uninitiated

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.

Performance as an Experimental Science

Performance tuning is a synthesis between technology, methodology, measurable quantities and tools. Its aim is to affect measurable outputs in a manner de-

sired 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 to implement
- 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 the so-called dismal methods of statistics. 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 user is encouraged to view the statistical techniques found in this book as a starting point, rather than a final 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
- Degradation
- Utilization
- Efficiency
- Scalability

We will briefly discuss each in turn. Note that for most performance projects, not every metric will be optimised simultaneously. The case of only 2-4 metrics being improved in a single performance iteration is far more common, and may be as many as can be tuned at once.

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.

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.

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) as 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 underutilised, then there should be some slack before observables change, but if re-

sources 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 underutilised, then increasing load may not apprciably 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.

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 9** 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. This causes 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 accept-

ance 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 success and failure 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.

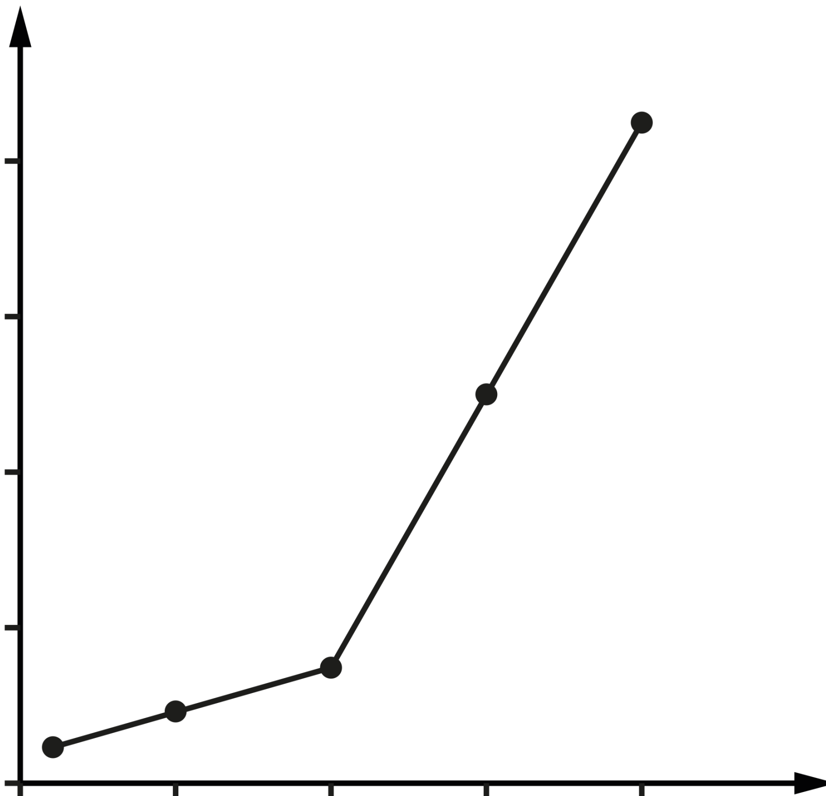


FIGURE 1-1

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.

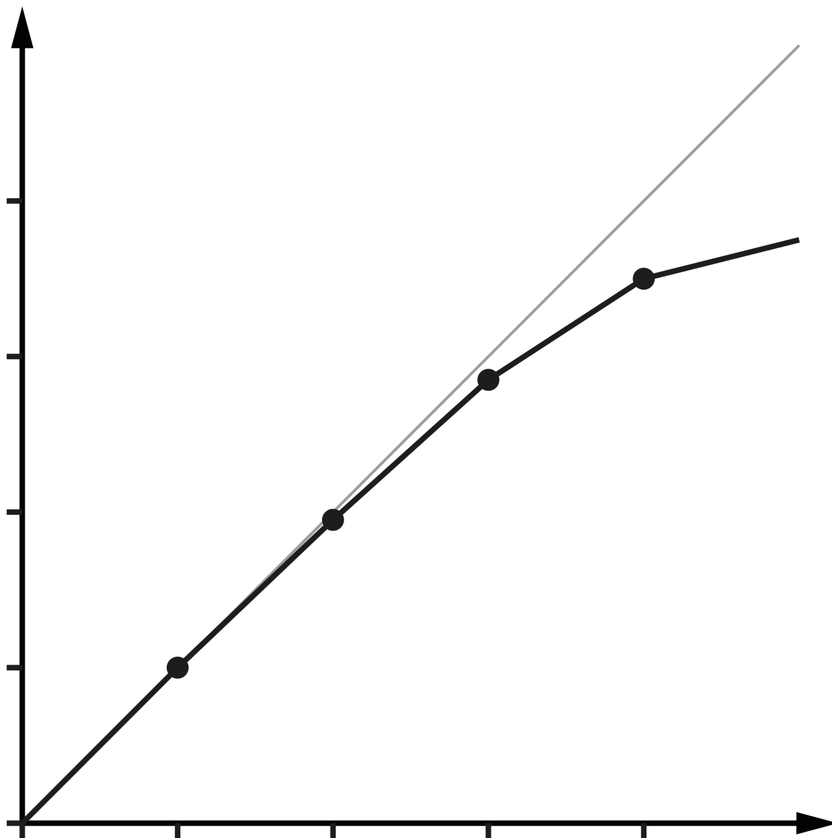


FIGURE 1-2

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.

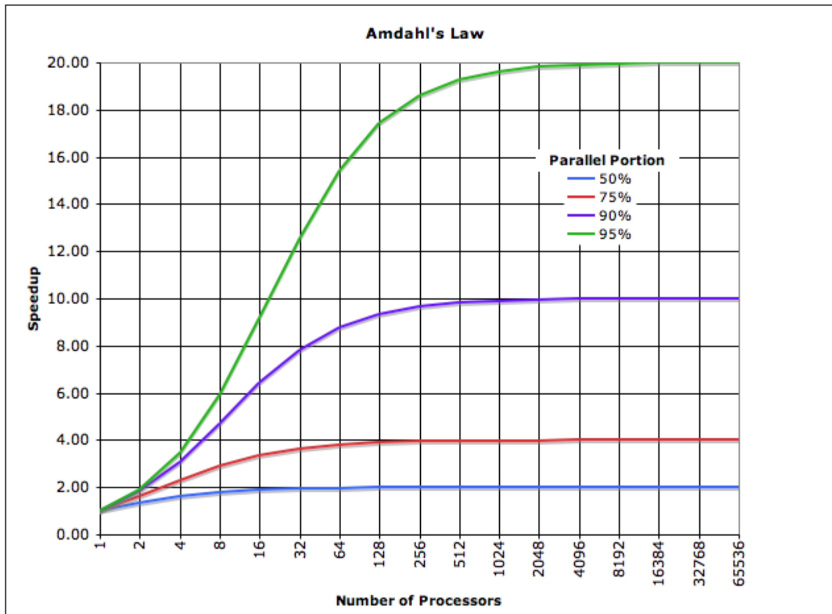


FIGURE 1-3

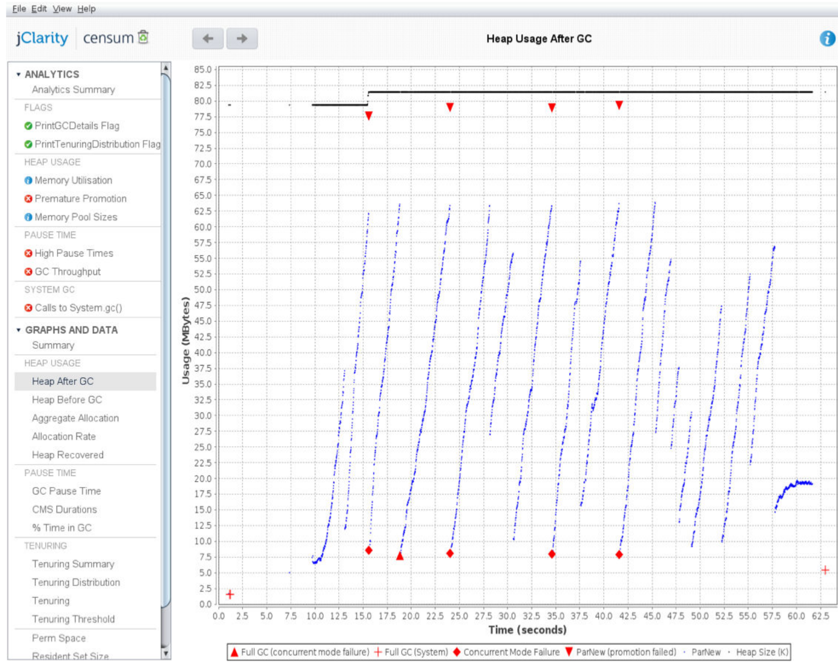
Amdahl's Law

In **Chapter 12** 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 in serial, 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.

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**.

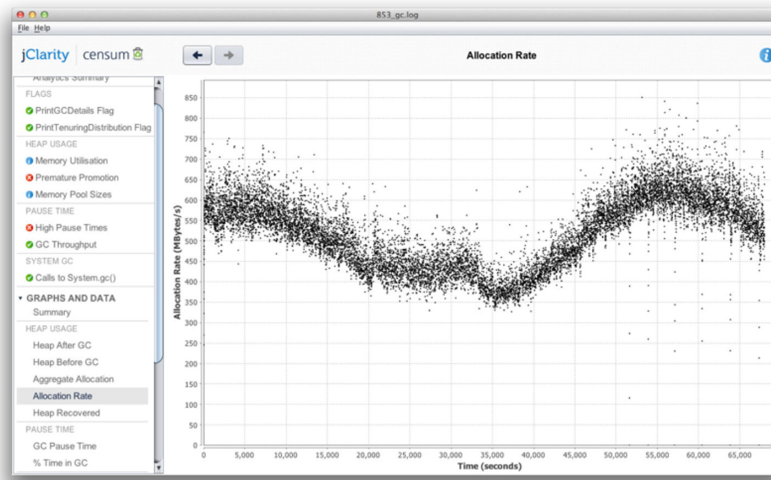


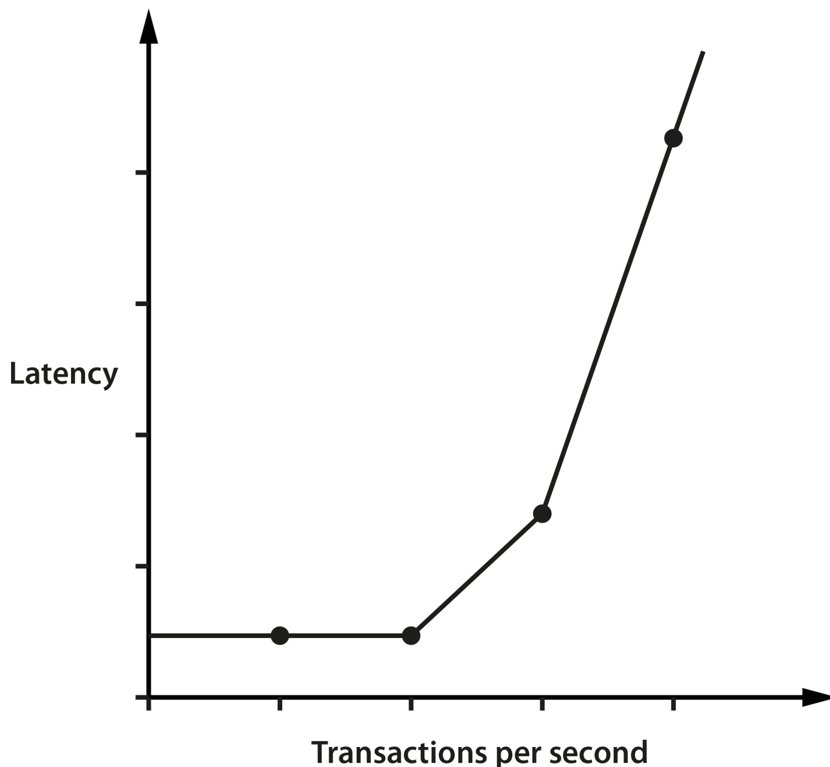
FIGURE 1-5

Stable allocation rate

In **Figure 1-5**, we show another memory graph that is very typical for a healthy application. The 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.

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.

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 what makes JVM-based performance optimization a particularly complex problem.

Overview of the JVM 2

Overview

There is no doubt that Java is one of the largest technology platforms on the planet, boasting roughly 9-10 million Java developers. 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 client 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.

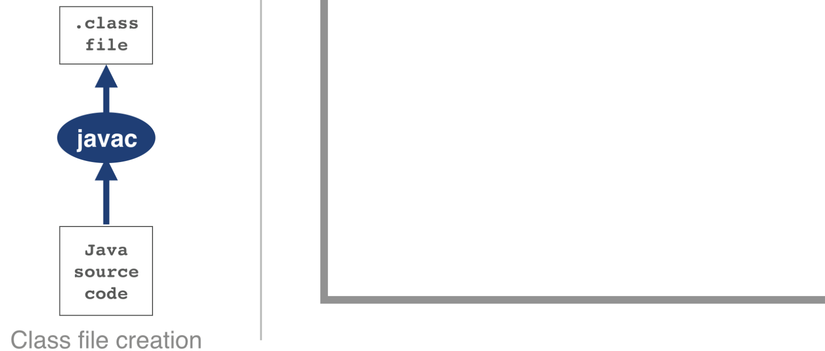
This chapter introduces how the JVM executes Java in order to form the basis for deeper exploration of these topics later in the book.

Code Compilation and Bytecode

It is important to appreciate that Java 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 developed 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.

> 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.

Component	Description
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.

> 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.

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 the class file at runtime.

Access flags are used to determine the modifiers applied to the class. The first part of the flag identifies whether a class is public followed by if 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.

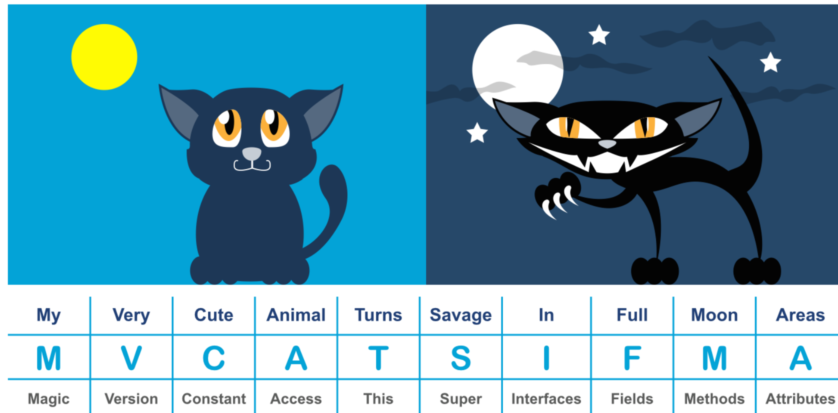
The `this` class, super class and interface entries are indexes into the constant pool to identify the type hierarchy belonging to the class. Fields and methods define a signature like structure including the modifiers that apply to the field or method. A set of attributes are then used to represent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

ods 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.

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
}
```

The overall layout describes the bytecode for HelloWorld class file, `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.

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

`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.

Interpreting and Classloading

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 action of the JVM interpreter can be simply thought of a switch inside a while loop - processing each opcode independently of the last using the stack positions to hold intermediate values.

As we will see when we delve into the internals of the Oracle / OpenJDK VM (HotSpot), the situation for real production-grade Java interpreters is a little 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 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 operating system starts the virtual machine process and almost immediately, the first in a chain of class loaders is initialised. This 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 13**.

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 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 tht it doesn't load the system (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 behaviour 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).

Introducing HotSpot

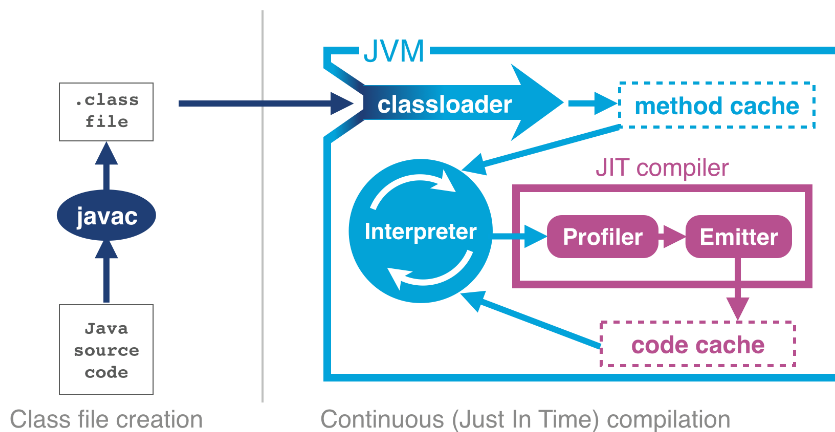


FIGURE 2-3

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

Java has never subscribed to the zero-overhead abstraction philosophy. Instead, it focused on practicality and developer productivity, even at the expense of raw performance. 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.

HotSpot 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 section of code.

The method of taking interpreted code and compiling once it passes a threshold is known as just-in-time compilation or JIT. One of the key benefits of JIT is having trace information that is collected during the interpreted phase, enabling HotSpot to make more informed optimisations. HotSpot has had hundreds of years (or more) of development attributed to it and new optimisations and benefits are added with almost every new release.

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 eliding virtual calls. HotSpot can also detect the precise CPU type it is running on at VM startup, and can use specific processor features if available, for even higher performance. These are known as JVM "intrinsics" and are discussed in detail in **Chapter 9**.

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.

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 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 core language. 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** and **8** 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 direct 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();
```

This means that all Java programs are 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.

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 12** contains a detailed look at how the JMM works, and the practicalities of working with threads and locks.

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;
```

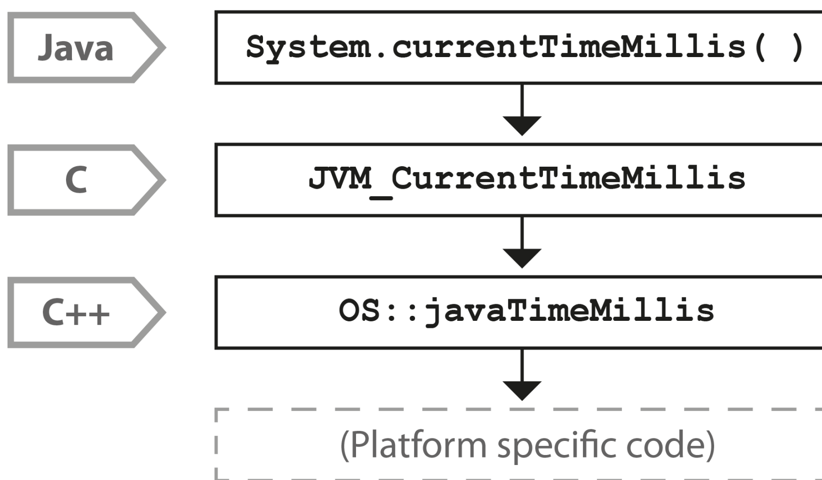
As all these methods deal with relatively low-level platform concerns, let's look at a more straightforward and familiar example - getting the system time.

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 2-4

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 demonstrates how the platform-independent parts of Java can call into services that are provided by the underlying operating system and hardware. 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.

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.

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.

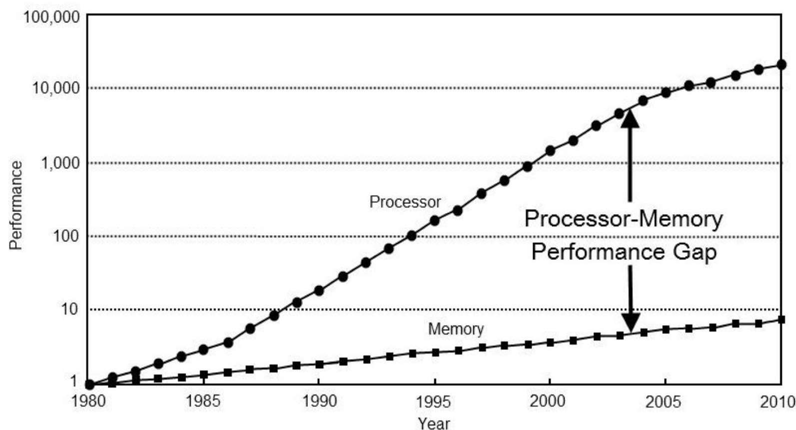


FIGURE 3-1

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.

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.

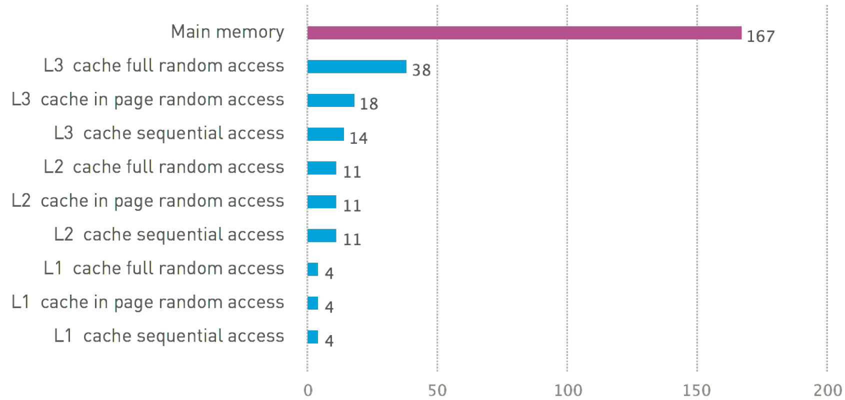
Modern CPUs have several layers of cache, with the most-often-accessed caches being located close to the processing core. The cache closest to the CPU is usually called L1, with the next being referred to as L2, and so on. Different processor architectures have a varying number, and configuration, of 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].

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

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 when applied in a parallel processing environment, as we will see later in this chapter.

Memory Caches

Modern hardware does introduce some new problems in terms of determining how memory is fetched into and written back from cache. This is known as a "cache consistency protocol".

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 here, 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. *The MESI protocol*

	M	E	S	I
M	-	-	-	Y
E	-	-	-	Y
S	-	-	Y	Y

	M	E	S	I
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**

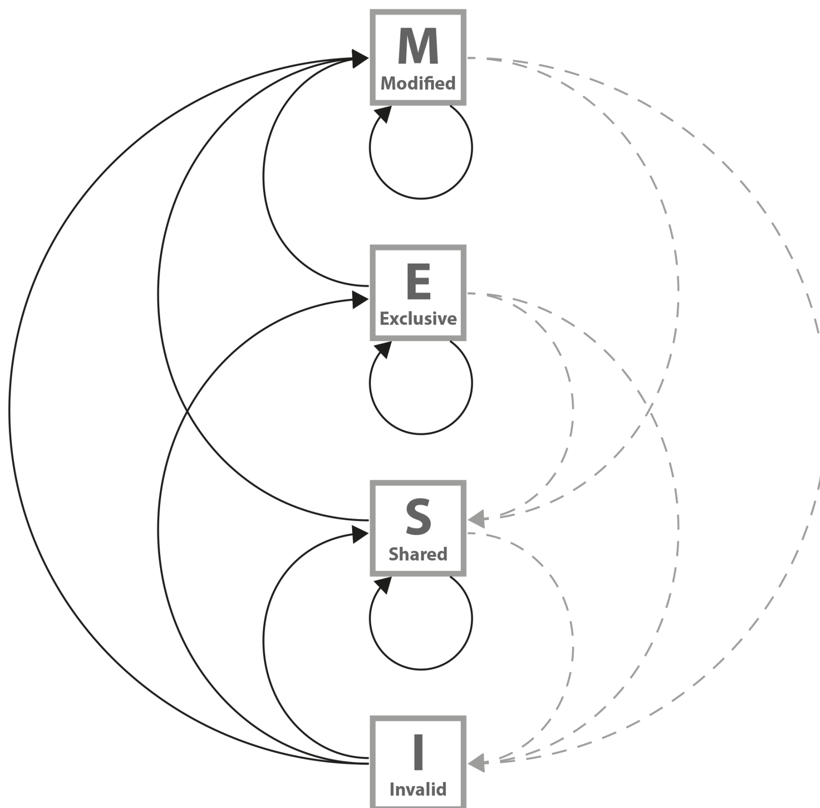


FIGURE 3-3

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.

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"). 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 = 10 * 1024 * 1024;
    private final int[] testData = new int[ARR_SIZE];

    private void run() {
        for (int i = 0; i < 10_000; i++) {
            touchEveryLine();
            touchEveryItem();
        }
        System.out.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;
            System.out.println(elEvery +" "+ 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**:

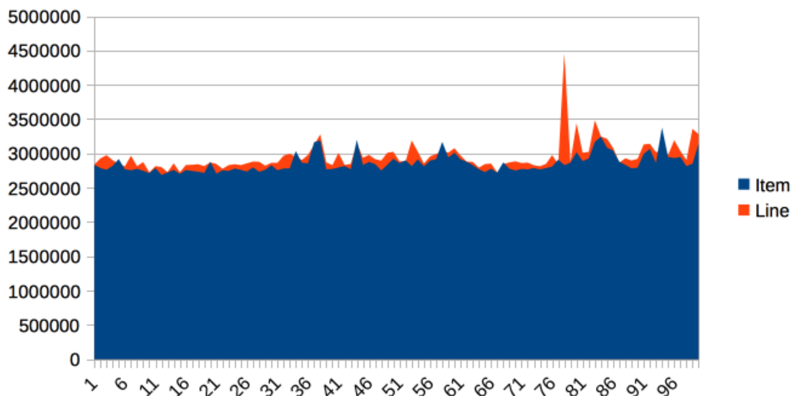


FIGURE 3-4

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.

> 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.

> 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 evalu-

ate a value needed for a conditional branch. Modern processors have multi-stage 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 up to 20 cycles, 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	-	-	-	-

	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 12**.

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 thread is shown in **Figure 3-5**.

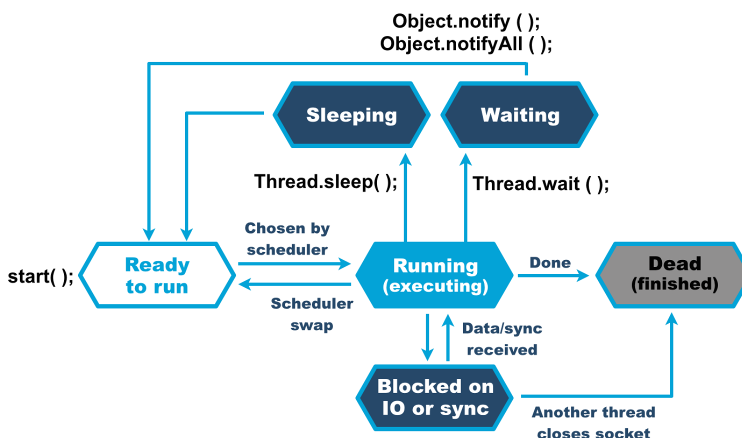


FIGURE 3-5

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 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.

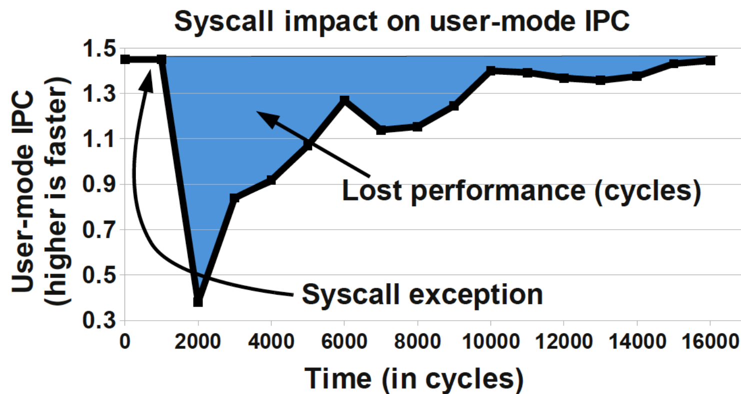
Context Switches

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 "wall-clock 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.

3 Image reproduced from "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls" by Soares & Stumm

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.

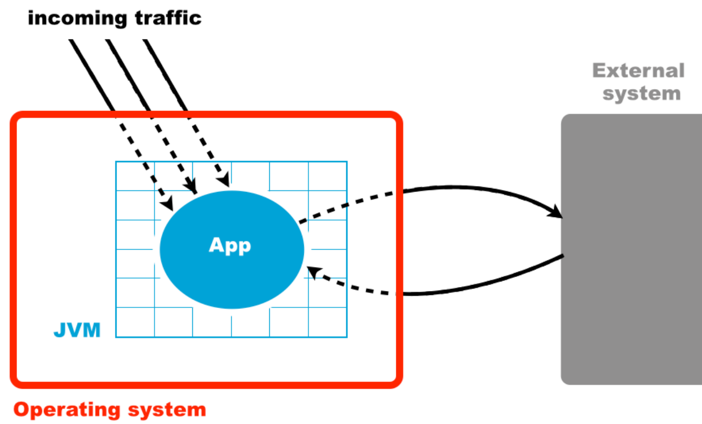


FIGURE 3-7

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.

> 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. In the memory section, the amount of swapped and free memory is shown, followed by the memory used as buffer and as cache.
3. The swap section shows the memory swapped from and to disk. Modern server class machines should not normally experience very much swap activity.
4. 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.
5. In the system section, the number of interrupts and the number of context switches per second are displayed.
6. 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).

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 & 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.

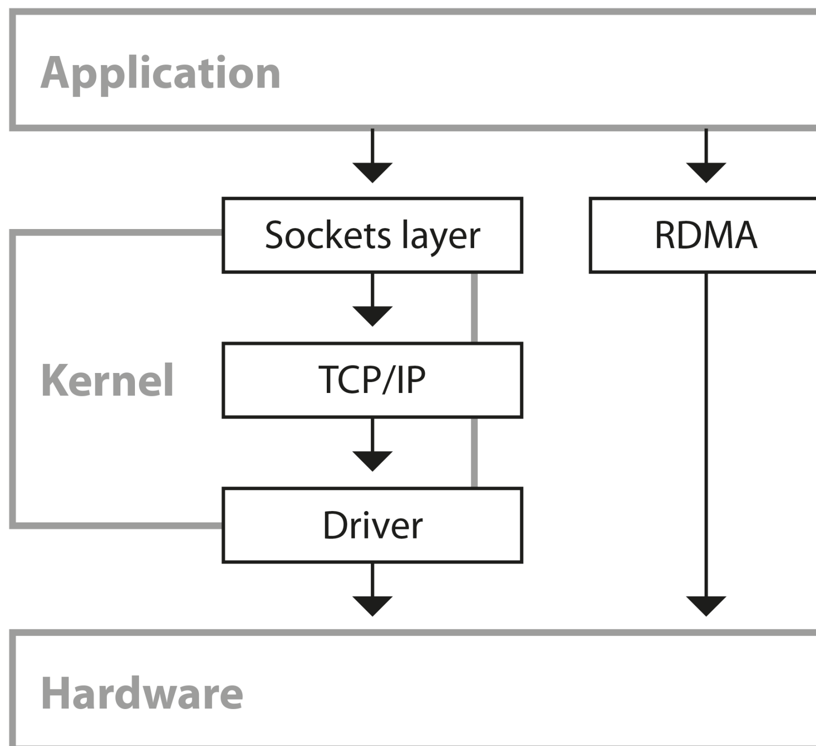
Finally, it's worth mentioning one aspect of I/O that is becoming more widely used across a class of Java applications that have a dependency on I/O but also stringent performance applications.

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



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.

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.

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.

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.

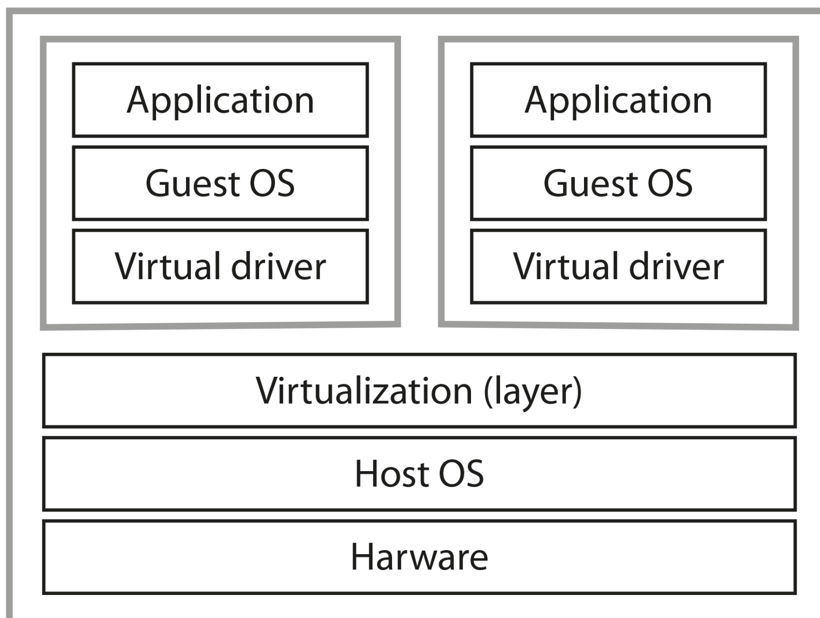


FIGURE 3-9

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.

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.