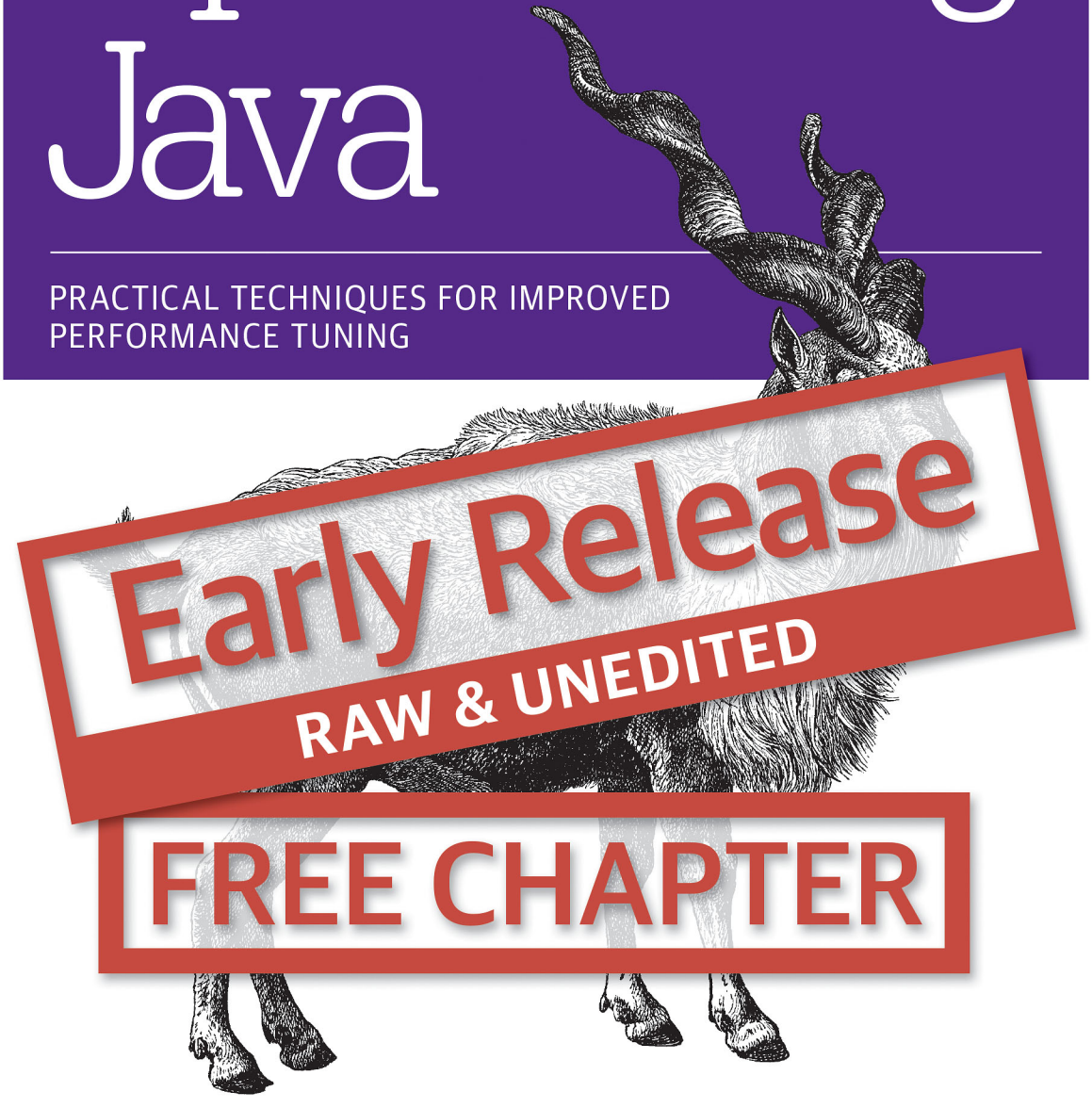
O'REILLY®

Optimizing Java

PRACTICAL TECHNIQUES FOR IMPROVED PERFORMANCE TUNING



Early Release

RAW & UNEDITED

FREE CHAPTER

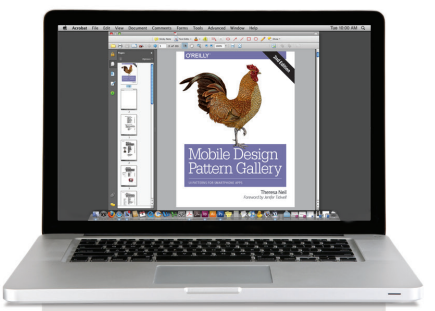
Benjamin J. Evans & James Gough

www.itbook.store/books/9781491933329

O'Reilly ebooks.

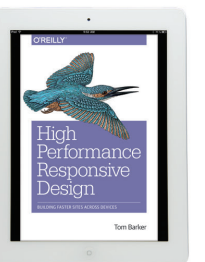
Your bookshelf on your devices.

PDF



Mobi

ePub



DAISY

When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O'Reilly ebooks through the iBookstore, the Android Marketplace, and Amazon.com.

O'REILLY®

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055

www.itbook.store/books/9781491933329

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*.

Editors: Brian Foster and Nan Barber
Production Editor:
Copyeditor:
Proofreader:
Indexer:
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2016: First Edition

Revision History for the First Edition
2016-MM-YY First Release

See *http://oreilly.com/catalog/errata.csp?isbn=0636920042983* 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 authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

978-1-491-93332-9

[LSI]

www.itbook.store/books/9781491933329

Table of Contents

Preface ix

1. Optimization and Performance Defined 11
Java Performance - The Wrong Way 11
Performance as an Experimental Science 12
A Taxonomy for Performance 13
Throughput 14
Latency 14
Capacity 14
Utilisation 14
Efficiency 15
Scalability 15
Degradation 15
Connections between the observables 16
Reading performance graphs 17

2. Overview of the JVM 23
Overview 23
Code Compilation and Bytecode 23
Interpreting and Classloading 28
Introducing HotSpot 29
JVM Memory Management 31
Threading and the Java Memory Model 32
The JVM and the operating system 33

3. Hardware & Operating Systems 35
Introduction to Modern Hardware 36
Memory 36

www.itbook.store/books/9781491933329

Memory Caches 38
Modern Processor Features 43
Translation Lookaside Buffer 43
Branch Prediction and Speculative Execution 43
Hardware Memory Models 44
Operating systems 45
The Scheduler 45
A Question of Time 47
Context Switches 48
A simple system model 49
Basic Detection Strategies 50
Context switching 52
Garbage Collection 53
I/O 53
Kernel Bypass I/O 54
Mechanical Sympathy 56
Virtualisation 56

4. Performance Testing Patterns and Antipatterns. 59
Types of Performance Test 59
Latency Test 60
Throughput Test 61
Load Test 61
Stress Test 61
Endurance Test 61
Capacity Planning Test 62
Degradation Test 62
Best Practices Primer 63
Top-Down Performance 63
Creating a test environment 63
Identifying performance requirements 64
Java-specific issues 65
Performance testing as part of the SDLC 66
Performance Antipatterns 66
Boredom 67
Resume Padding 67
Peer Pressure 67
Lack of Understanding 68
Misunderstood / Non-Existent Problem 68
Distracted By Simple 68
Description 68
Example Comments 69

www.itbook.store/books/9781491933329

Reality 69
Discussion 69
Resolutions 69
Distracted By Shiny 70
Description 70
Example Comment 70
Reality 70
Discussion 70
Resolutions 70
Performance Tuning Wizard 71
Description 71
Example Comment 71
Reality 71
Discussion 71
Resolutions 71
Tuning By Folklore 72
Description 72
Example Comment 72
Reality 72
Discussion 72
Resolutions 73
Blame Donkey 73
Description 73
Example Comment 73
Reality 74
Discussion 74
Resolutions 74
Missing the Bigger Picture 74
Description 74
Example Comments 75
Reality 75
Discussion 75
Resolutions 76
UAT Is My Desktop 76
Description 76
Example Comment 76
Reality 77
Discussion 77
Resolutions 77
PROD-like Data Is Hard 77
Description 77
Example Comment 78

www.itbook.store/books/9781491933329

Reality 78
Discussion 78
Resolutions 79
Cognitive Biases and Performance Testing 79
Reductionist Thinking 80
Confirmation Bias 80
Fog of war (Action Bias) 80
Risk bias 81
Ellsberg's Paradox 81

5. Microbenchmarking and statistics. 83
Overview 84
Don't microbenchmark if you can help it - a true story 84
Heuristics for when to microbenchmark 85
Introduction to measuring Java performance 86
Introduction to JMH 90
Selecting and Executing Benchmarks 90
Statistics for JVM performance 95
Systematic Error 96
Random Error 97
Spurious Correlation 97
Non-normal statistics 97
Common Problems for homemade benchmarks 98
Interpretation of statistics 98
Summary 99

6. Monitoring and Analysis. 101
VisualVM 101
Thermostat 101
Illuminate 101
New Relic 101
Java Flight Recorder 101

7. Understanding Garbage Collection. 103
Allocation and lifetime 103
Introducing Mark & Sweep 104
Garbage Collection Glossary 104
Garbage Collection in Hotspot 105
Thread-local allocation 105
Hemispheric Collection 106
The parallel collectors 106
The role of allocation 104

www.itbook.store/books/9781491933329

Summary 107

8. Garbage Collection Monitoring and Tuning. 109
Tuning 109
Introduction to the collectors 109
Parallel 109
CMS 110
G1 110
Other collectors 110
Tools 110
Censum 110
GCViewer 110
Heap Dump Analysis 110
jHiccup 110

9. HotSpot JIT Compilation. 111
Overview of Bytecode Interpretation 112
Profile Guided Optimization 112
The Code Cache 112
JIT Compilation Strategies 112
JITwatch 113

10. Java language performance techniques. 115

11. Profiling. 117
When to profile (and when not to) 117
JProfiler 117
VisualVM Profiler 117
Honest Profiler 117
Mission Control 117

12. Concurrent Performance Techniques. 119
Understanding The JMM 119
Analysing For Concurrency 119

13. The Future. 121
Changes coming in Java 9 121
Looking Ahead - Java 10? 121
Other trends 121
Conclusion 122

Index. 123

www.itbook.store/books/9781491933329

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.

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.

www.itbook.store/books/9781491933329

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 (& 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.

The goal of a latency tuning exercise is usually to directly improve user experience.

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.

www.itbook.store/books/9781491933329

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.

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" of the system. This is sometimes observed during a throughput test, but is more properly the point of a stress test.

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.

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 is 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 CMS garbage collector, which may eventually suffer "Concurrent Mode Failure" - see Section 8.3).

www.itbook.store/books/9781491933329

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 a full GC leading to a stop-the-world event (see Chapter 8 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.

www.itbook.store/books/9781491933329

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.

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. This includes not only application servers, but web servers, databases, load balancers, network firewalls, etc. Any services, e.g.

www.itbook.store/books/9781491933329

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.

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 for the risks introduced.

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 more widespread.

This makes the construction of a performance testing environment that looks like production potentially mopre 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.

www.itbook.store/books/9781491933329

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 100 ms
- 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 the goals to be achieved are, is essential and 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) The method is too large or complex to be analysed for compilation.

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.

www.itbook.store/books/9781491933329

In Chapter 9 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.

Performance Antipatterns

An antipattern is an undesired behavior of a software project or team, that is observed across a large number of projects. The commonality of occurance leads to the conclusion (or the suspicion) that some underlying factor is responsible for creating the unwanted behavior.

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,

www.itbook.store/books/9781491933329

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 [1]. 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 "Resume Padding").

Resume Padding

Occasionally that overuse of technology is not tied to boredom, but in fact an opportunity to boost experience with a technology on a CV. This can be related to boredom, and 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 resume 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 equally 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

1 *https://dzone.com/articles/why-developers-keep-making-bad*

www.itbook.store/books/9781491933329

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 what the current tools are actually capable of. 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.

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

www.itbook.store/books/9781491933329

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

Dev 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 / 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 & 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

www.itbook.store/books/9781491933329

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

www.itbook.store/books/9781491933329

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.

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

"It's Unix - I know this"

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

www.itbook.store/books/9781491933329

- 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 & 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.

www.itbook.store/books/9781491933329

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 & 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 / 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"

www.itbook.store/books/9781491933329

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.

www.itbook.store/books/9781491933329

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 engi-

www.itbook.store/books/9781491933329

neer 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:

- Measure in PROD
- Change one switch at a time in UAT
- Ensure that your UAT environment has the same stress points as production
- Also the same test data for the normal load condition.

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.

- Test change in UAT
- Retest in UAT
- Have someone recheck your reasoning
- Pair with them to discuss your conclusions

UAT Is My Desktop

Description

UAT environments often differ significantly from PROD, 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"

www.itbook.store/books/9781491933329

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 & opportunity cost related to lost customers
- Buy a UAT environment that is identical to PROD
- In most cases, the cost of the first, far outweighs the second and sometimes the right case needs to be made to managers

PROD-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 that processes all booked futures and options. The order of messages would be in the millions that the system would be required to process each 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.

www.itbook.store/books/9781491933329

> 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 PROD 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 PROD-like for accurate results. If data is not available for security reasons then it should be 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 PROD 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.

www.itbook.store/books/9781491933329

Resolutions

- Consult data domain experts & invest in a process to migrate PROD data back into UAT, 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.

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.

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.

www.itbook.store/books/9781491933329

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.

Tuning by Folklore, missing the bigger picture and abuse of microbenchmarks are all examples of antipatterns that are caused at least in part by the reductionism bias.

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 anti-pattern 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 anti-patterns at work, all leading to new uproved assumptions in the new library stack.

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.

www.itbook.store/books/9781491933329

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".[2]

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 thoroughout.

2 To reuse the phrase made famous by Donald Rumsfeld

www.itbook.store/books/9781491933329

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.

www.itbook.store/books/9781491933329

Want to read more?

You can buy this book at oreilly.com
in print and ebook format.

Buy 2 books, get the 3rd FREE!

Use discount code OPC10

All orders over $29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including
the iBookstore, the Android Marketplace, and Amazon.com.

O'REILLY®

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055

www.itbook.store/books/9781491933329