Continuous Profiling in Production: What, Why and How

...

Richard Warburton (@richardwarburto)
Sadiq Jaffer (@sadiqj)
https://www.opsian.com

Opsian

Why Performance Tools Matter

Development isn’t Production

Profiling vs Monitoring

Continuous Profiling

Conclusion

Opsian

Known Knowns

Known Unknowns

Unknown Unknowns

Why Performance Tools Matter

Development isn’t Production

Profiling vs Monitoring

Continuous Profiling

Conclusion

Opsian

Development isn't Production

Performance testing in development can be easier

May not have access to production

Tooling often desktop-based

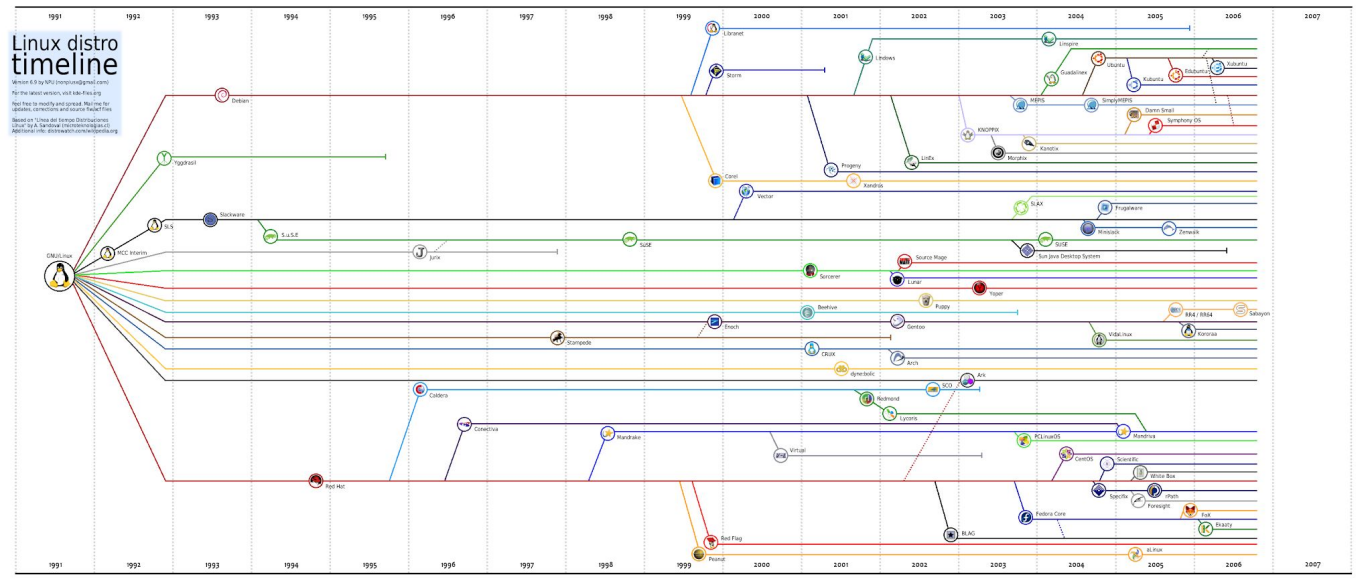
Not representative of production

Unrepresentative Hardware

VS

Opsian

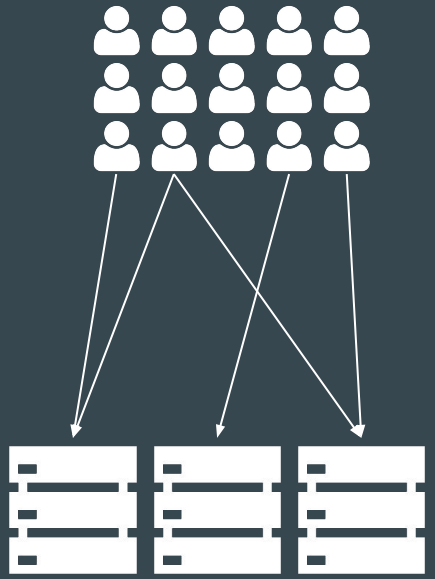
Unrepresentative Software



Opsian

Unrepresentative Workloads

VS



Opsian

The JVM may have very different behaviour in production

Hotspot does adaptive optimisation

Production may optimise differently

Opsian

Why Performance Tools Matter

Development isn't Production

Profiling vs Monitoring

Continuous Profiling

Conclusion

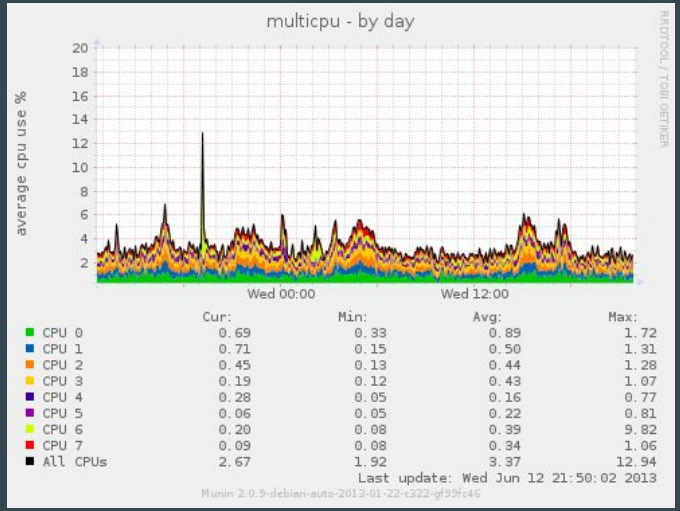
Opsian

Ambient/Passive/System Metrics

Preconfigured Numerical Measure

CPU Time Usage / Page-load Times

Cheap and sometimes effective



Opsian

Opsian

Logging

Records arbitrary events emitted by the system being monitored

log4j/slf4j/logback

Logs of GC events

Often manual, aids system understanding, expensive

Coarse Grained Instrumentation

Measures time within some instrumented section of the code

Time spent inside the controller layer of your web-app or performing SQL queries

More detailed and actionable though expensive

Production Profiling

What methods use up CPU time?

What lines of code allocate the most objects?

Where are your CPU Cache misses coming from?

Automatic, can be cheap but often isn’t

Opsian

Where Instrumentation can be blind in the Real World

Problem: every 5 seconds an HTTP endpoint would be really slow.

Instrumentation: on the servlet request, didn’t even show the pause!

Cause: Tomcat expired its resources cache every 5 seconds, on load one resource scanned the entire classpath

Opsian

Surely a better way?

Not just Metrics - Actionable Insights

Diagnostics aren’t Diagnosis

What about Profiling?

Opsian

Why Performance Tools Matter

Development isn’t Production

Profiling vs Monitoring

Continuous Profiling

Conclusion

Opsian

How to use Continuous Profilers

1) Extract relevant time period and apps/machines
2) Choose a type of profile: CPU Time/Wallclock Time/Memory
3) View results to tell you what the dominant consumer of a resource is
4) Fix biggest bottleneck
5) Deploy / Iterate

Opsian

CPU Time vs Wallclock Time

Opsian

You need both CPU Time and Wallclock Time

CPU - Diagnose expensive computational hotspots and inefficient algorithms

Spot code that should not be executing but is ...

Wallclock - Diagnose blocking that stops CPU usage

e.g blocking on external IO and lock contention issues

Opsian

Profiling Hotspots

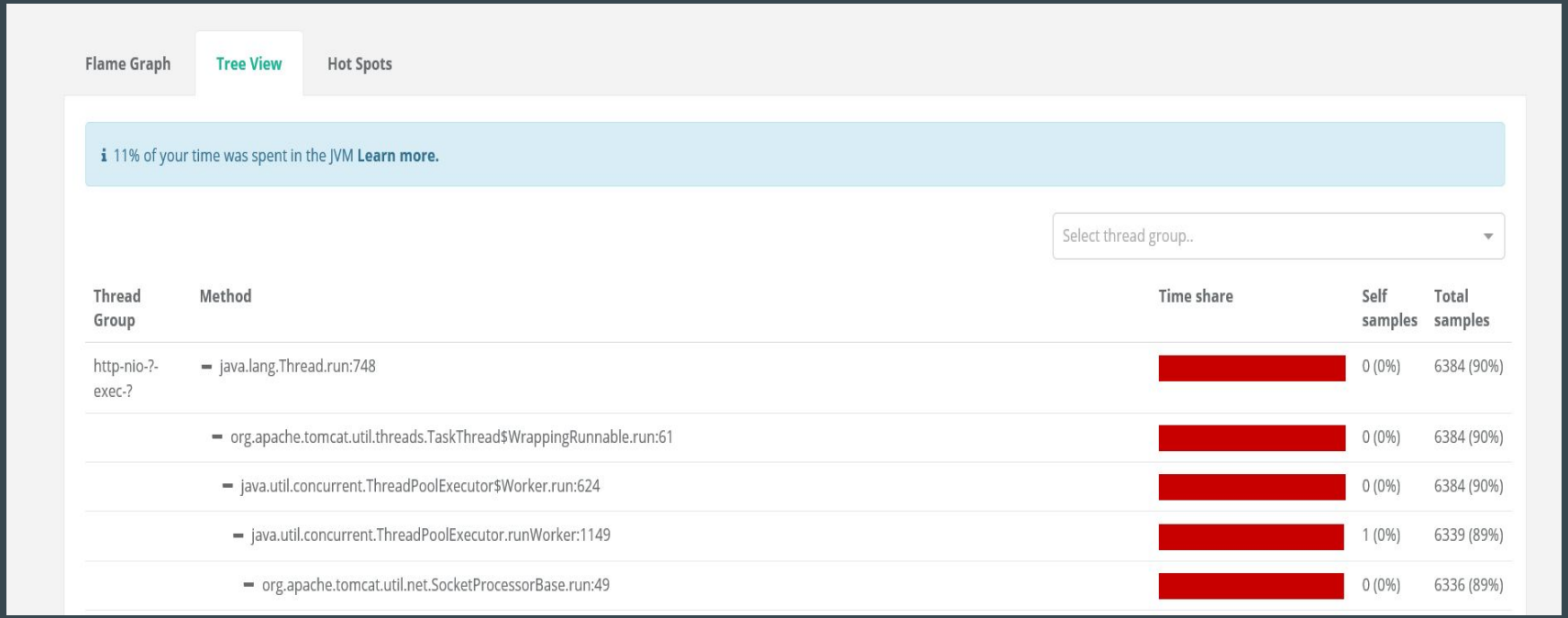
Packages to exclude: com.sun org.foo.bar

Method	Calls	Time
java.util.zip.ZipFile.getEntry	589 (8%)	84,227ms
java.io.UnixFileSystem.getBooleanAttributes0	419 (5%)	59,917ms
java.security.AccessController.doPrivileged	135 (1%)	19,305ms
sun.misc.Unsafe.park	119 (1%)	17,017ms
sun.nio.ch.EPollArrayWrapper.interrupt	108 (1%)	15,444ms
java.lang.String.charAt	103 (1%)	14,729ms

Hot Spot line information	Samples
String.java:**657**	64 (62%)
String.java:**660**	39 (38%)

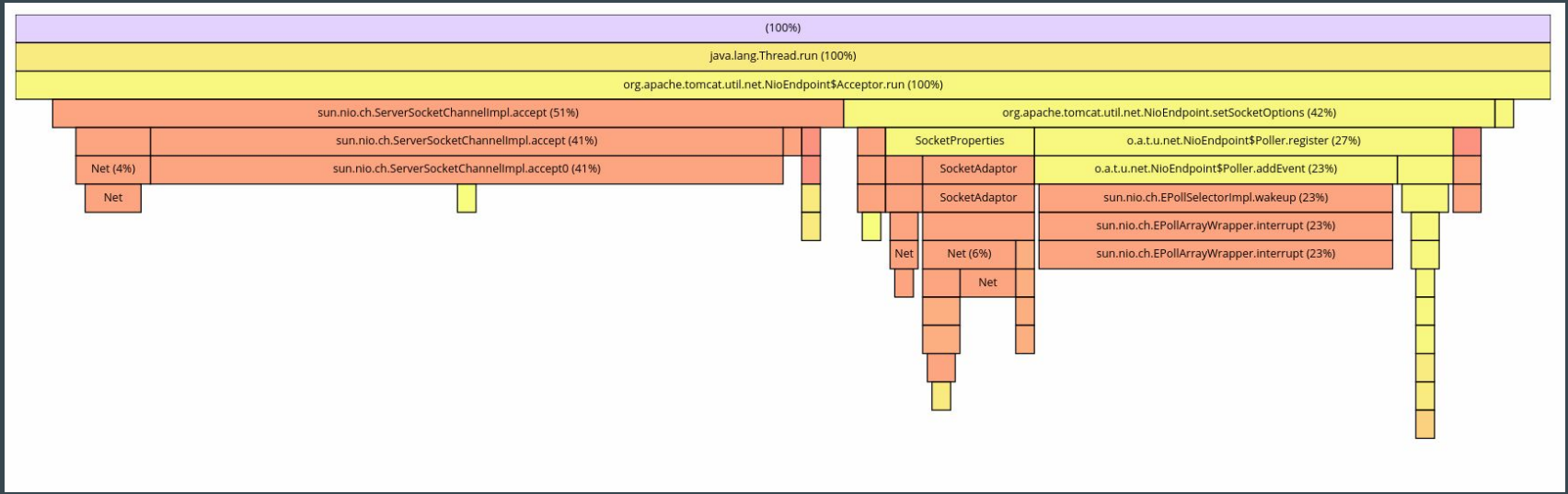
Opsian

Profiling Treeviews



Opsian

Profiling Flamegraphs



Opsian

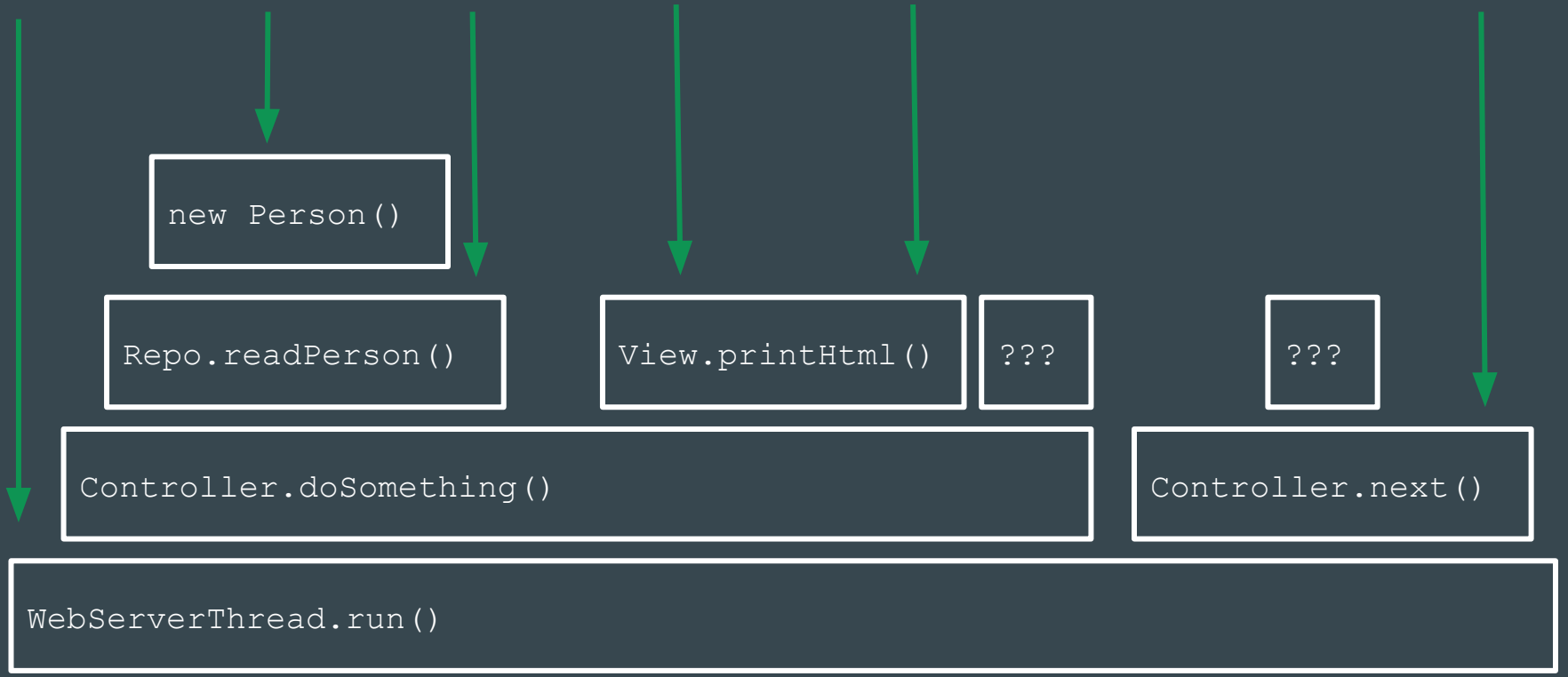
Instrumenting Profilers

Add instructions to collect timings (Eg: JVisualVM Profiler)

Inaccurate - modifies the behaviour of the program

High Overhead - > 2x slower

Sampling/Statistical Profilers



Opsian

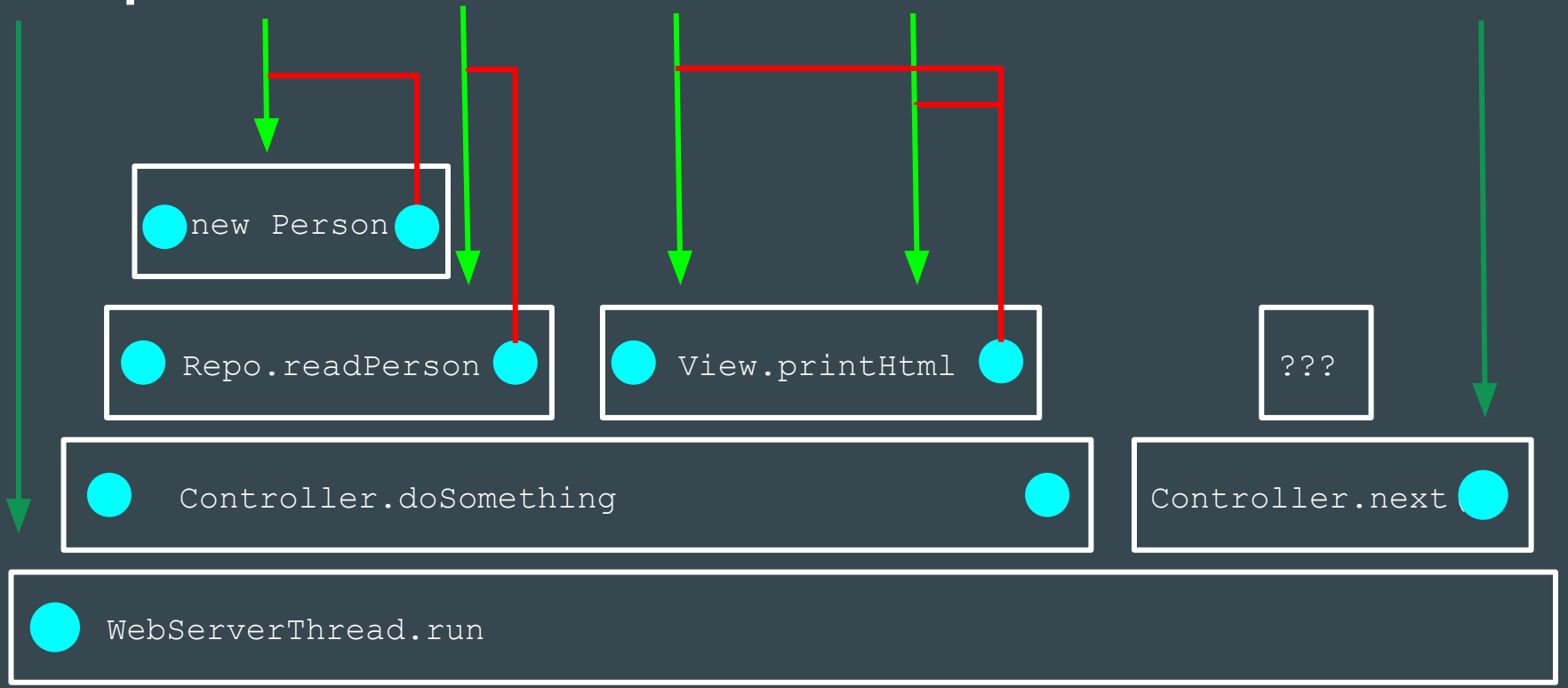
Safepoints

Mechanism for bringing Java application threads to a halt

Safepoint *polls* added to compiled code read known memory location

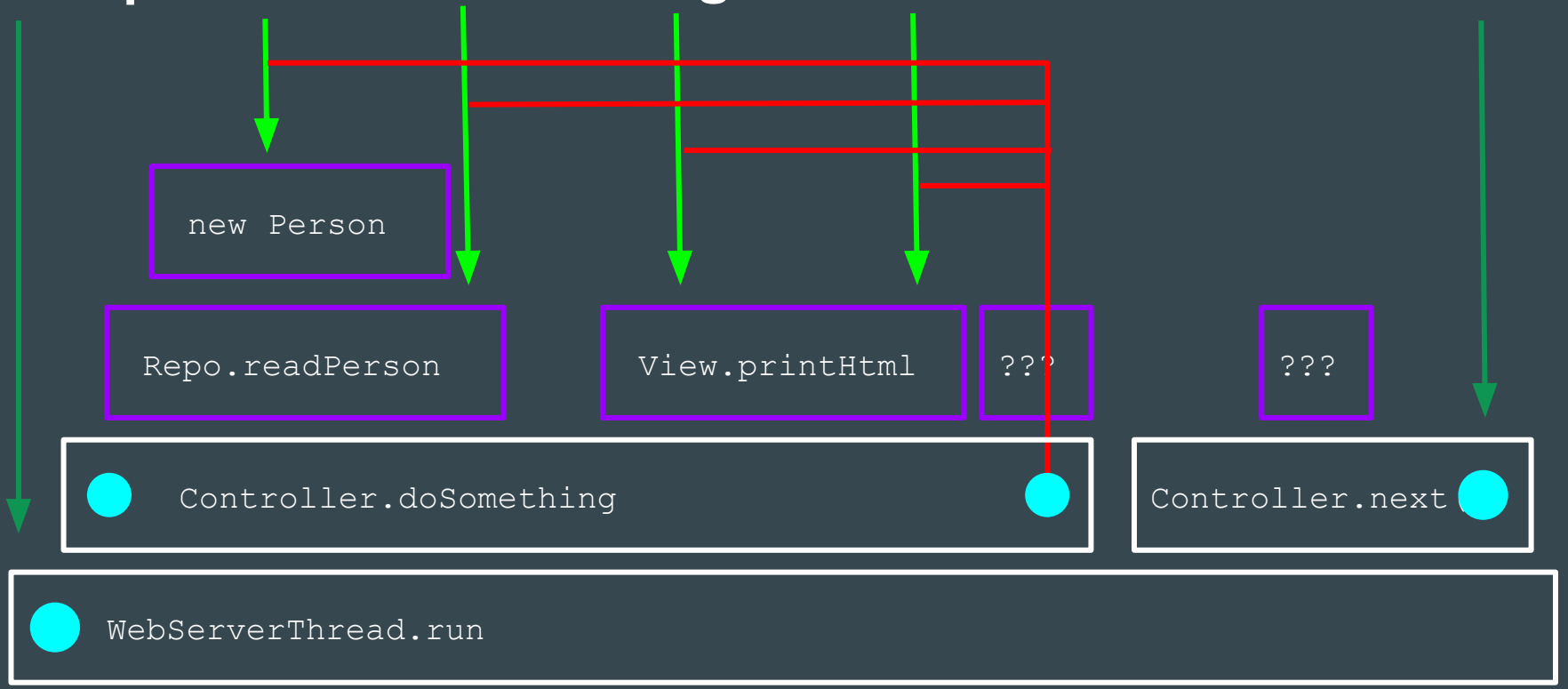
Protecting memory page triggers a segfault and suspends threads

Safepoint Bias



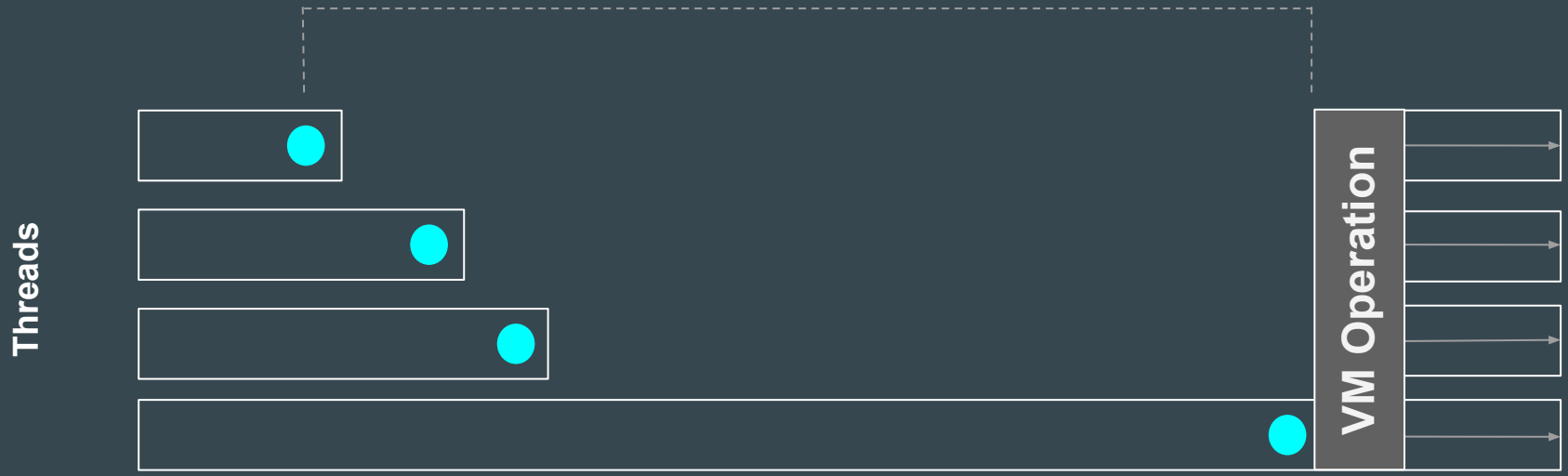
Opsian

Safepoint Bias after Inlining



Opsian

Time to Safepoint



-XX:+PrintSafepointStatistics

Safepoint poll

Opsian

Statistical Profiling in Java

Problem: *getAllStackTraces* is expensive to do frequently and inaccurate, also only gives us Wallclock time

Need ways to:

1. Interrupt application
2. Sample resource of interest

Advanced Statistical Profiling in Java

- Interrupt with OS signals
  - Delivered to handler on only one thread
  - Lightweight
- Sample resource of interest
  - Use *AsyncGetCallTrace* to sample stack
  - Examine JVM internals for other resources

Opsian

Advanced Statistical Profiling in Java

Approach not used by existing profilers (VisualVM and desktop commercial alternatives)

Can give very low overheads (<1%) for reasonable sampling rates

Opsian

People are put off by practical as much as technical issues

Barriers to Ad-Hoc Production Profiling

Generally requires access to production

Process involves manual work - hard to automate

Low-overhead open source profilers without commercial support

Opsian

What if we profiled all the time?

Historical Data

Allows for post-hoc incident analysis

Enables correlation with other data/metrics

Performance regression analysis

Opsian

Putting Samples in Context

Application version

Environment parameters (machine type, CPU, location, etc.)

Ad-hoc profiling we can’t do this

Opsian

How to implement Continuous Profiling

Opsian

Google-wide profiling

Article: *Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers*

Profiling data and binaries collected, processed and made available for browser-based reporting

“The system has been actively profiling nearly all machines at Google for several years”

https://ai.google/research/pubs/pub36575

Self-build

- Open source Java profilers suitable for production
  - Async-profiler
  - Honest profiler
  - Flight Recorder
- Need to collect and store profiles in a database
- Tools for retrieving and visualising stored profiling data
  - Browser-based
  - Command line

Opsian

Opsian - Continuous Profiling

JVM Agents

Opsian Aggregation service

Web Reports

Opsian

Summary

It’s possible to profile in production with low overhead

To overcome practical issues we can profile production all the time

We gain new capabilities by profiling all the time

Opsian

Why Performance Tools Matter

Development isn't Production

Profiling vs Monitoring

Continuous Profiling

Conclusion

Opsian

Performance Matters

Development isn’t Production

Metrics can be unactionable

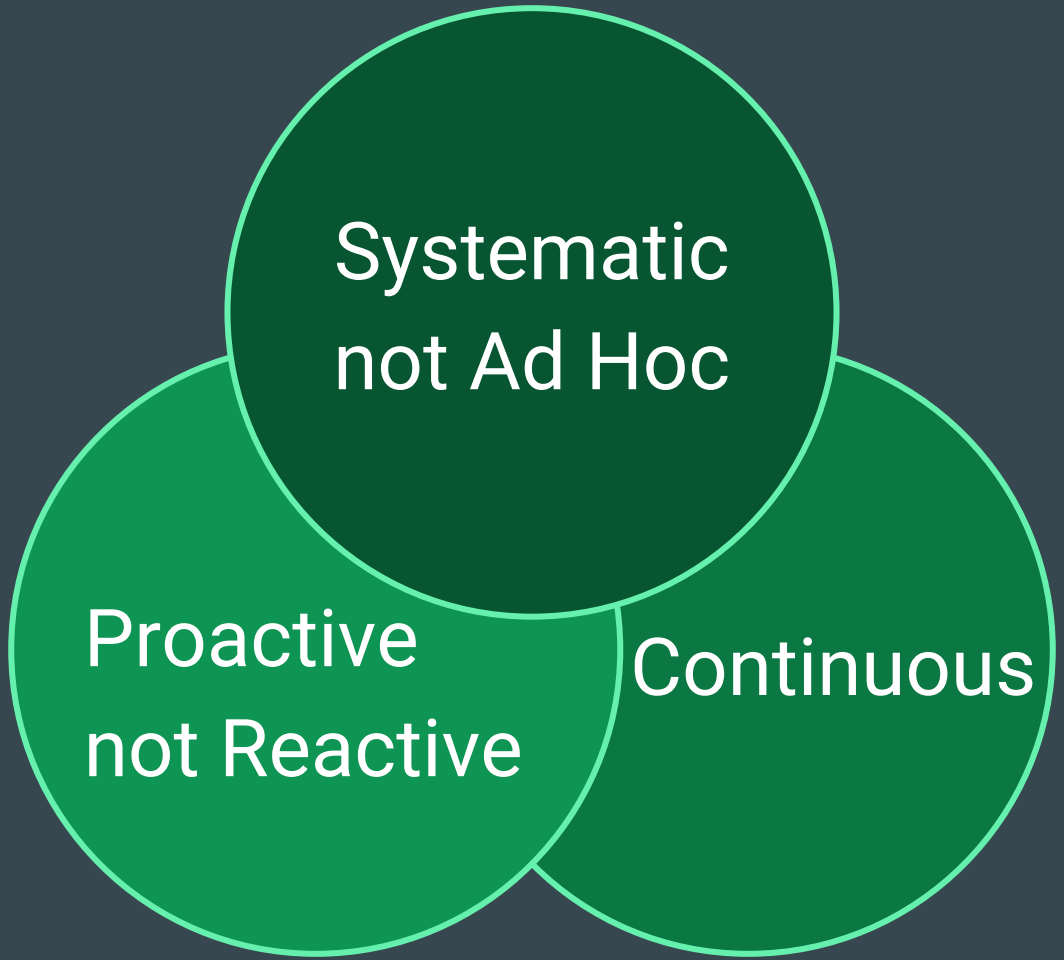
Instrumentation has high overhead

Continuous Profiling provides insight

Opsian

We need an attitude shift on profiling + monitoring

Opsian



Opsian

Please do Production Profiling.
All the time.

Opsian

Any Questions?

https://www.opsian.com/

Opsian

Live Demo?

Links

Collector - Flame Graph

Collector - Hot Spots

The End

Existing tools are blind

Traditional profilers don’t work in production

Metrics aren’t code level visibility

Instrumentation must be done ahead of time

How do we help?

Reduce the risk of change

Help you scale with happy customers

Cut the cost of infrastructure

Opsian

Production Visibility

Actionable reports for causes of latency and CPU usage

From high-level reports to line-level granularity

Very low overhead (<1%) and always-on

Opsian

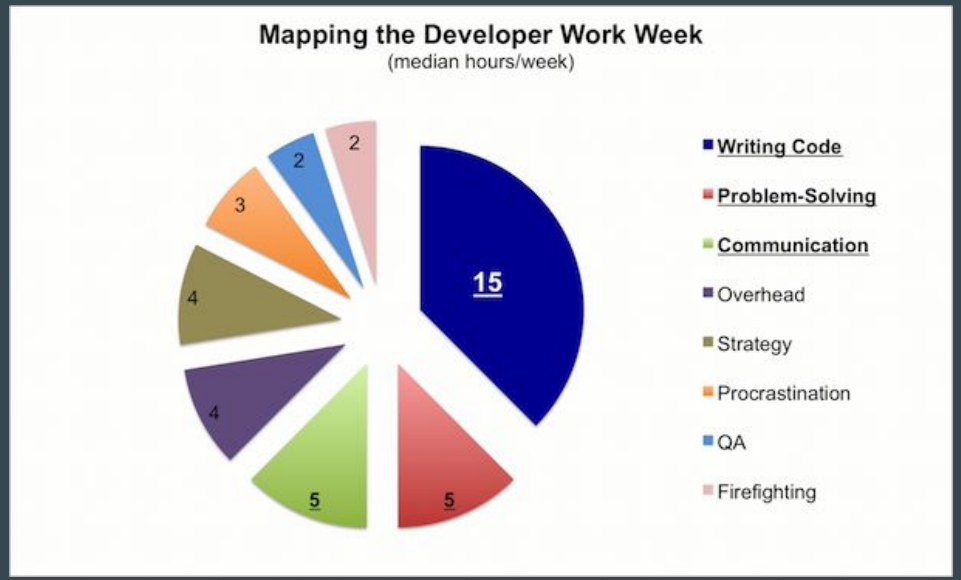
Reduce the risk of change

On-demand performance comparison between releases

Accelerate root-cause analysis for performance regressions

Opsian

Improve Developer Productivity



Source: ZT RebelLabs Developer Productivity Report 2017

Opsian

Understand don’t Overwhelm

Too Little

You can’t understand production problems

Too Much

Needle in a Haystack

You are the problem (overhead)

Opsian

Normalisation of Deviance

“Some of the tests always fail, so we just ignore them.”

“Some of the alerts get triggered regularly, so we just ignore them.”

Alert false positives have a cost

Opsian

Open Source Java Profilers

High Overhead

VisualVM

hprof

Twitter's CPUProfile

Anything *GetAllStackTraces* based

Low Overhead

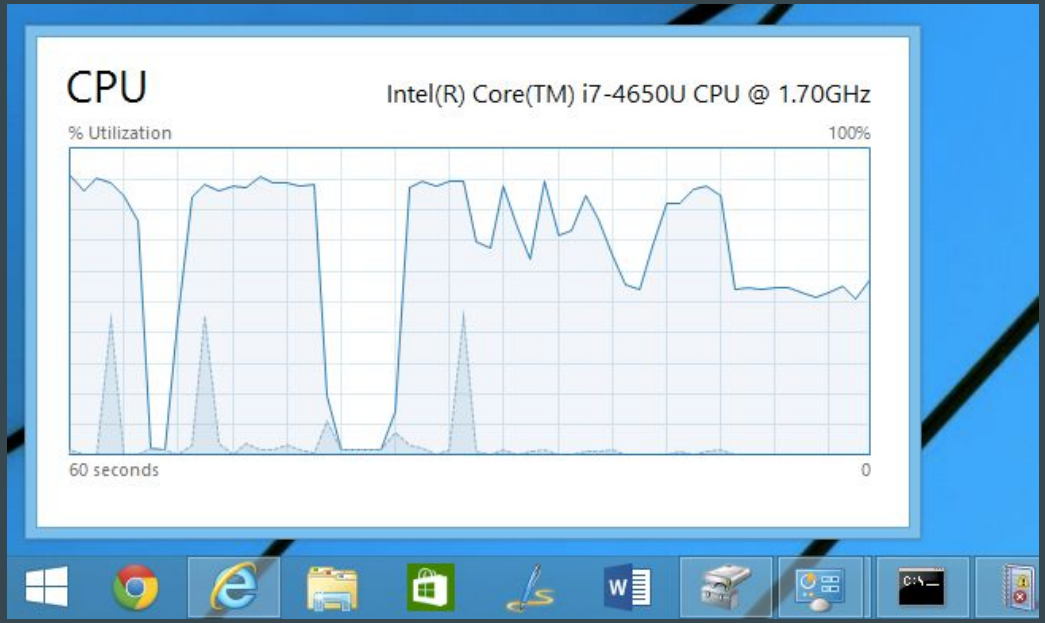
Async Profiler

Honest Profiler

Java Mission Control

Opsian

Unactionable Metrics



Many metrics provide pretty graphs but don't progress treatment

Opsian

Profiling Support in the Linux Kernel

perf and eBPF

perf-map-agent for the JVM

Hardware events (L1/L2/L3 cache misses, branch mispredictions, etc.)

Take heed: potential security issues

Customer Experience

Opsian

Responsive Applications make more Money

Amazon: 100ms of latency costs 1% of sales

Google: 500ms seconds in search page generation time drops traffic by 20%

Opsian

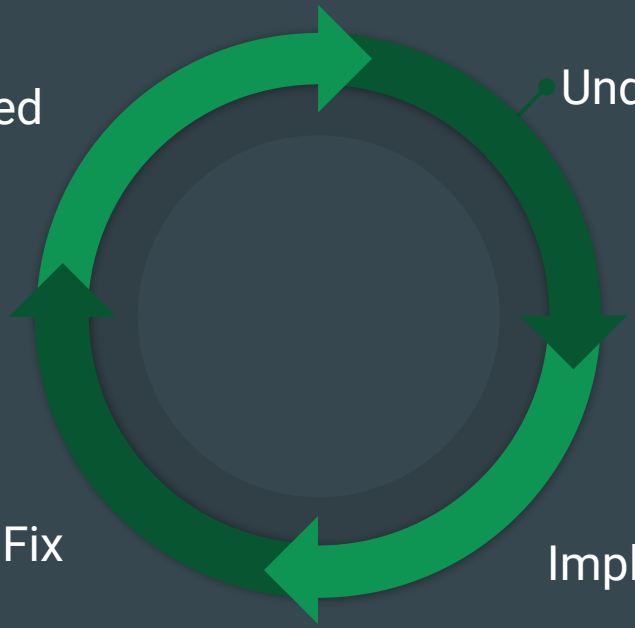
Stop Costly Downtime

Opsian

Reduce Costs

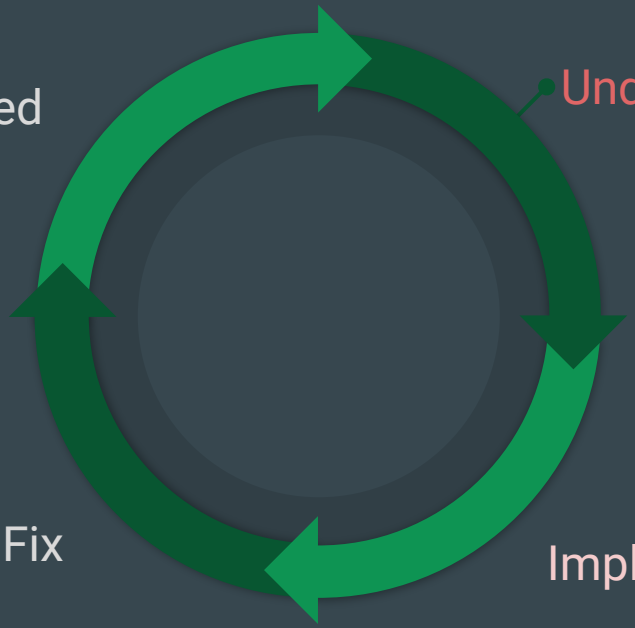
Opsian

Performance Optimisation Cycle



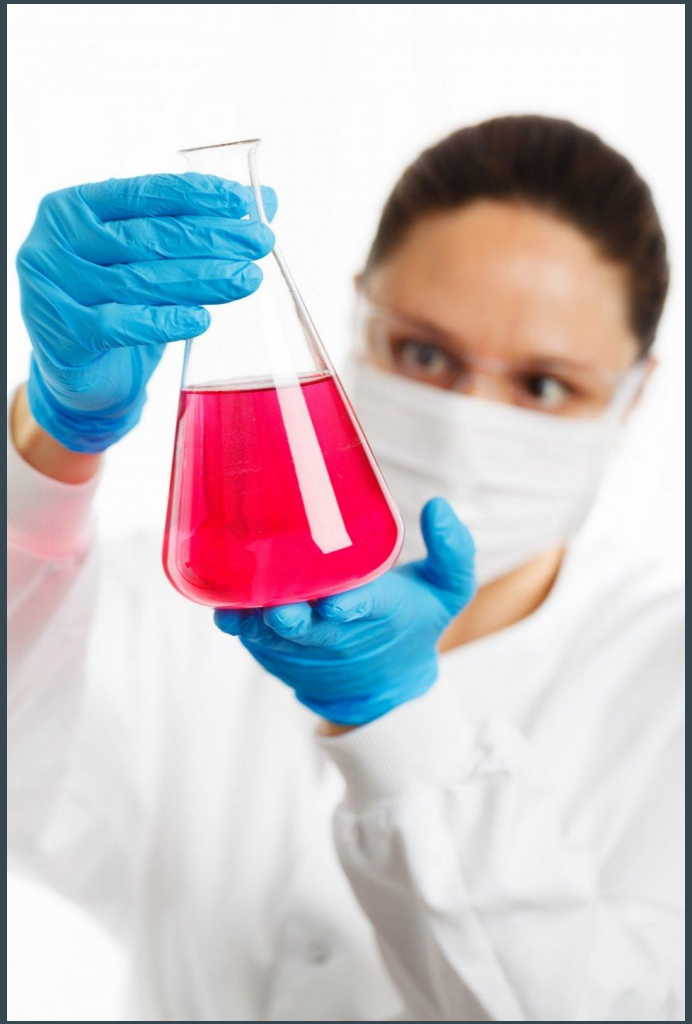
Opsian

What's Hard?



Opsian

How do you find performance bottlenecks?



Opsian