The dark art of performance tuning

or how to become a perf hero

AVSystem 2016

https://github.com/AVSystem/Wyklady_2016

Agenda

- Introduction to benchmarking and profiling
- Profiling metrics and tools - O/S level
- JVM primer
- Profiling metrics and tools - JVM level
- JVM Benchmarking
  - pitfalls
  - JMH

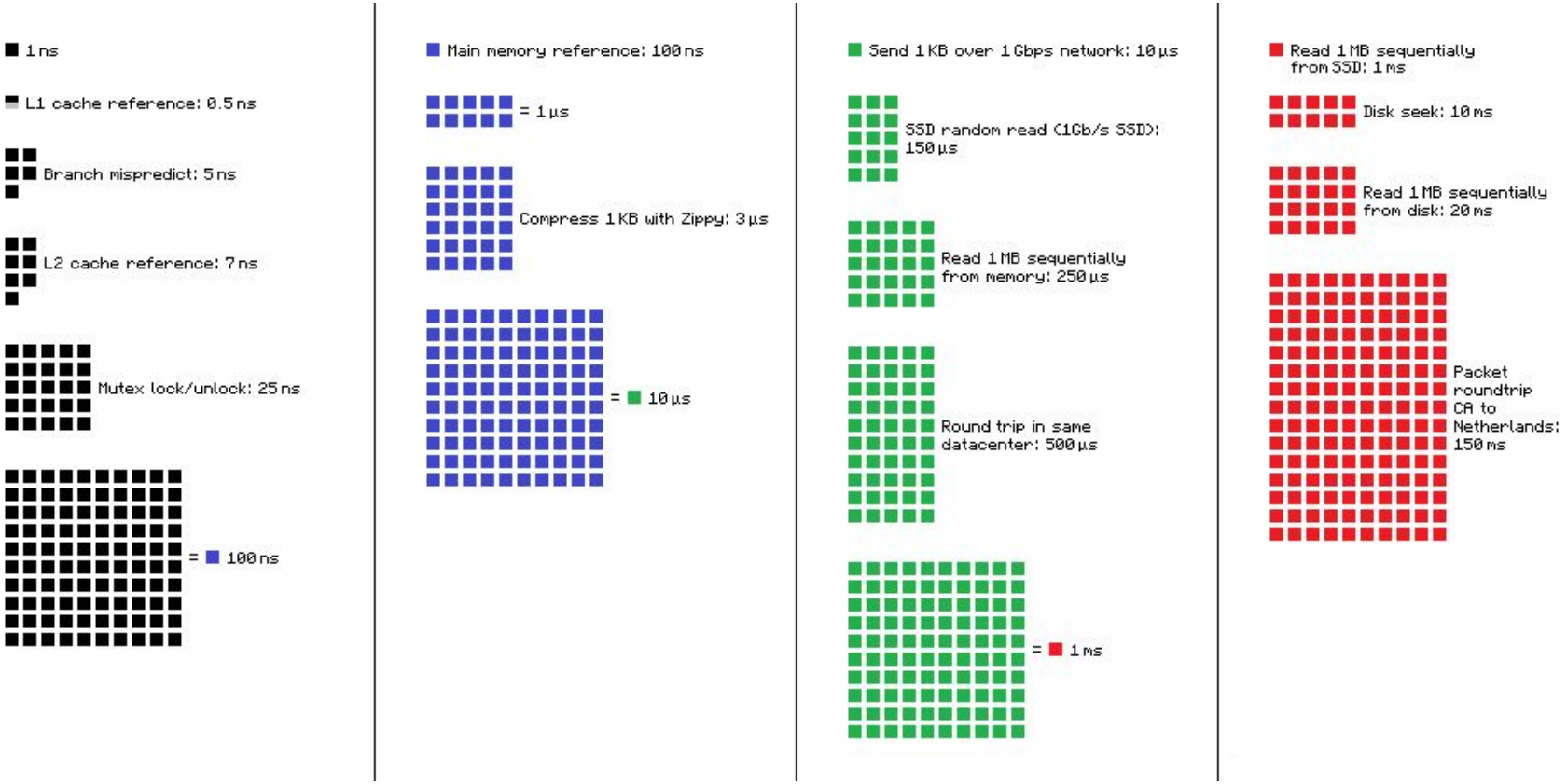
Benchmarking vs profiling

- Benchmarking
  - measuring different solutions to the same problem, e.g. sorting
- Profiling
  - runtime program analysis that yields execution information, e.g.
    - frequency and duration of particular function calls
    - memory allocation rate
- Usually go hand in hand
  - benchmark -> profile -> make a change -> benchmark

Performance analysis - why

- expectations vs reality
- new client requirements
  - SLAs to meet, e.g. 10k ops/s
  - new features
- marketing
- gathering knowledge to make conscious decisions
- The First Rule of Program Optimization: *Don't do it.* The Second Rule of Program Optimization (for experts only!): *Don't do it yet*
  - prefer readability to complicated but more performant code in MOST of the cases
  - optimize hotspots and bottlenecks
- fun & profit

Latency numbers every programmer should know



Benchmarks amplify all the effects visible at the same scale

- kilo: > 1000 s, Linpack
- _____: 1...1000 s, SPECjvm2008, SPECjbb2013
- milli: 1...1000 ms, SPECjvm98, SPECjbb2005
- micro: 1...1000 us, single webapp request
- nano: 1...1000 ns, single operations
- pico: 1...1000 ps, pipelining

not really hard

challenging

damned beasts

...

Optimization quiz

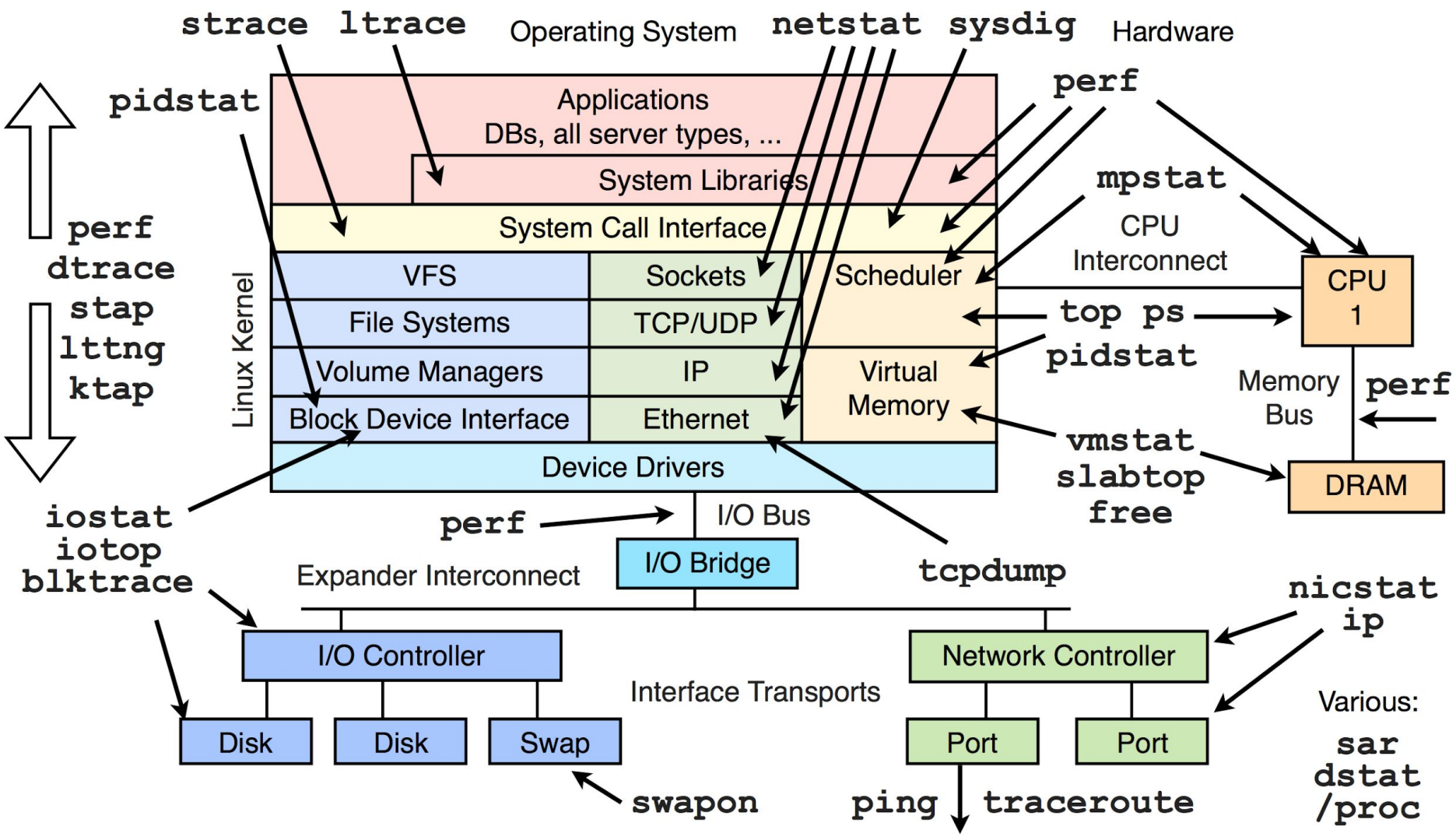
Empty benchmark, system reports 4 online CPUs

Threads	Ops/nsec	Scale
1	3.06 ± 0.10	
2	5.72 ± 0.10	1.87 ± 0.03
4	5.87 ± 0.02	1.91 ± 0.03

Software is *abstract*

hardware is **real**

O/S level



http://www.brendangregg.com/Perf/linuxperftools.png

O/S level (Linux)

- top/htop/ps
  - overall state of the system:
    - list of processes and threads
    - cpu usage
    - memory/swap usage
    - ...
- vmstat
  - overall state of the system:
    - memory/swap usage
    - basic i/o information
    - interrupts/context switches
    - cpu usage - system, user, idle, waiting for I/O,

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  0      3   1257    137   1390    0    0     0     0  906 2064  6  1 94  0  0
 1  0      3   1257    137   1390    0    0     0     0 1074 2537  3  1 96  0  0
```

- vmstat -S m
  - r - processes waiting for run time
  - b - processes blocked, waiting on resources
  - swpd - swapped memory
  - free - free memory
  - buff/cache - memory used as buffers/cache
  - si/so - swap in/swap out in pages
  - bi/bo - blocks received/sent from/to block device
  - in - interrupts
  - cs - context switches
  - us/sy/id/wa/st - CPU time

O/S level (Linux)

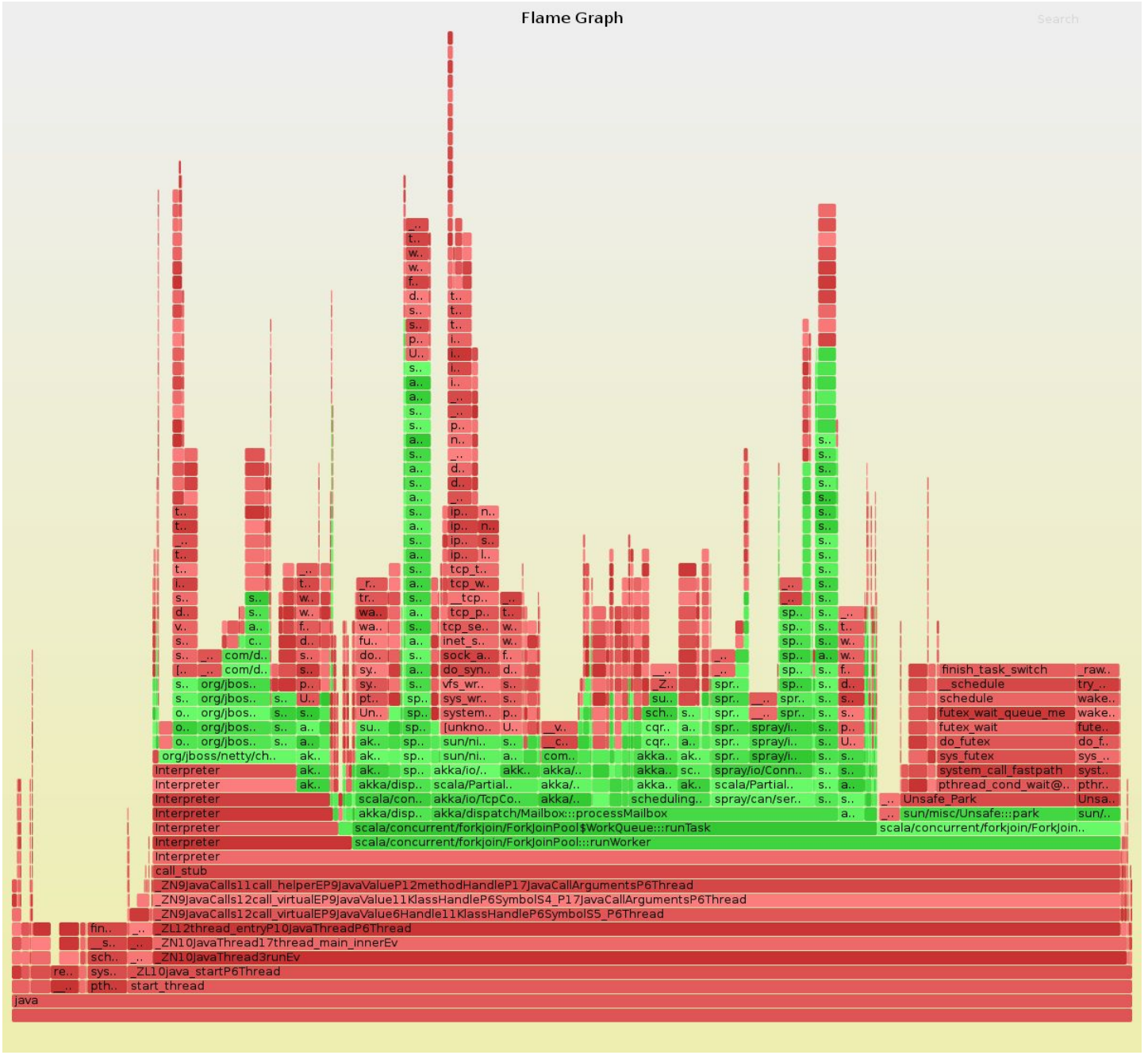
- iostat/iotop
  - I/O requests per second
  - I/O requests completed
  - average size of the requests
  - average wait time of requests to be completed (queue + service)
  - …
- pidstat
  - various statistics for linux tasks
  - I/O, page faults and memory utilization, context switches
- nload/iftop
  - networking statistics

```
Total DISK READ :       0.00 B/s | Total DISK WRITE :       2.46 M/s
Actual DISK READ:       0.00 B/s | Actual DISK WRITE:       0.00 B/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 4168 be/4 cassandr    0.00 B/s  143.61 K/s  0.00 %  0.24 % java -ea -javaagent:/usr/share/cassandra/l
 3973 be/4 cassandr    0.00 B/s  191.48 K/s  0.00 %  0.17 % java -ea -javaagent:/usr/share/cassandra/l
 5859 be/4 cassandr    0.00 B/s   62.60 K/s  0.00 %  0.09 % java -ea -javaagent:/usr/share/cassandra/l
 3972 be/4 cassandr    0.00 B/s  228.31 K/s  0.00 %  0.04 % java -ea -javaagent:/usr/share/cassandra/l
 3970 be/4 cassandr    0.00 B/s  301.96 K/s  0.00 %  0.00 % java -ea -javaagent:/usr/share/cassandra/l
 4165 be/4 cassandr    0.00 B/s  228.31 K/s  0.00 %  0.00 % java -ea -javaagent:/usr/share/cassandra/l
 4166 be/4 cassandr    0.00 B/s  180.44 K/s  0.00 %  0.00 % java -ea -javaagent:/usr/share/cassandra/l
 4167 be/4 cassandr    0.00 B/s  235.67 K/s  0.00 %  0.00 % java -ea -javaagent:/usr/share/cassandra/l
 5778 be/4 adebski     0.00 B/s   29.46 K/s  0.00 %  0.00 % java -XX:+PrintGCDetails -Xloggc:gc-14599
 5779 be/4 adebski     0.00 B/s   33.14 K/s  0.00 %  0.00 % java -XX:+PrintGCDetails -Xloggc:gc-14599
 5780 be/4 adebski     0.00 B/s   29.46 K/s  0.00 %  0.00 % java -XX:+PrintGCDetails -Xloggc:gc-14599
 5791 be/4 adebski     0.00 B/s   29.46 K/s  0.00 %  0.00 % java -XX:+PrintGCDetails -Xloggc:gc-14599
 5793 be/4 adebski     0.00 B/s    7.36 K/s  0.00 %  0.00 % java -XX:+PrintGCDetails -Xloggc:gc-14599
```

- iotop --only
  - SWAPIN - percentage of time the thread spent while swapping in
  - IO - percentage of time the thread spent while waiting on I/O

Perf events - Linux

- lightweight profiling solution for Linux systems
- included in the kernel
- based on notion of events
  - cache misses
  - branch mispredictions
  - page faults
  - context switches
  - cpu time
  - system calls
  - ...
- disadvantage: hard to find documentation for specific events

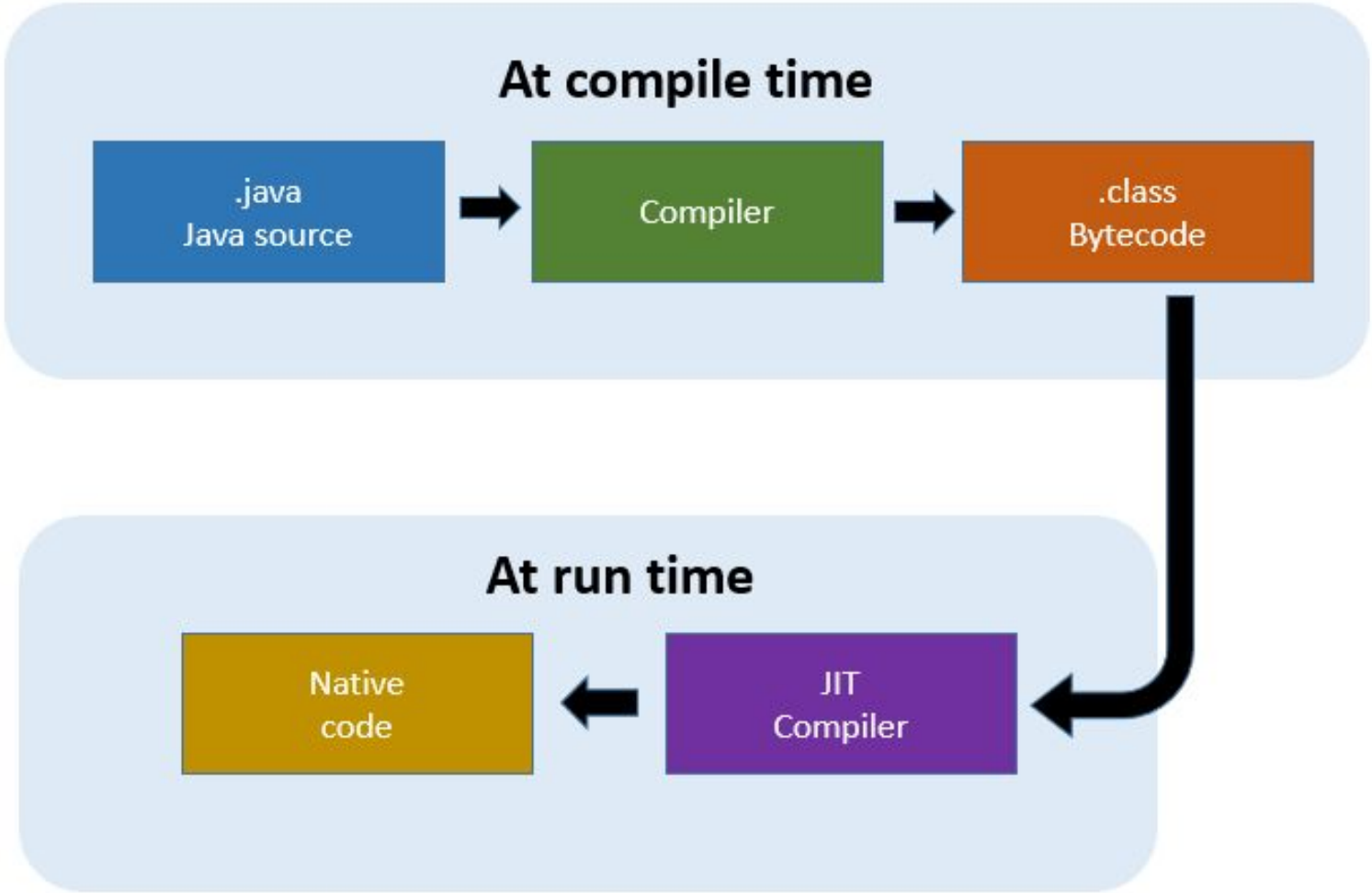

Flame Graph
Search
finish_task_switch
__schedule
schedule
futex_wait_queue_me
futex_wait
do_futex
sys_futex
system_call_fastpath
pthread_cond_wait@..
Unsafe_Park
sun/misc/Unsafe:::park
org/jboss/netty/ch..
Interpreter
akka/dispatch/Mailbox:::processMailbox
scala/concurrent/forkjoin/ForkJoinPool$WorkQueue:::runTask
scala/concurrent/forkjoin/ForkJoinPool:::runWorker
call_stub
_ZN9JavaCalls11call_helperEP9JavaValueP12methodHandleP17JavaCallArgumentsP6Thread
_ZN9JavaCalls12call_virtualEP9JavaValue11KlassHandleP6SymbolS4_P17JavaCallArgumentsP6Thread
_ZN9JavaCalls12call_virtualEP9JavaValue6Handle11KlassHandleP6SymbolS5_P6Thread
_ZL12thread_entryP10JavaThreadP6Thread
_ZN10JavaThread17thread_main_innerEv
_ZN10JavaThread3runEv
_ZL10java_startP6Thread
start_thread
java

Flamegraphs

- http://www.brendangregg.com/flamegraphs.html
- a way to visualize profiling data, not tied to perf tool
- perf record + perf script + perl script = svg image
- problem: for stacktraces to be readable requires debug symbols
- what about languages that run in VMs:
  - java - https://bugs.openjdk.java.net/browse/JDK-8068945
  - ruby
  - javascript
  - ...

JVM internals - primer

JIT compiler



http://www.cs.sit.kmutt.ac.th/blog/?p=403

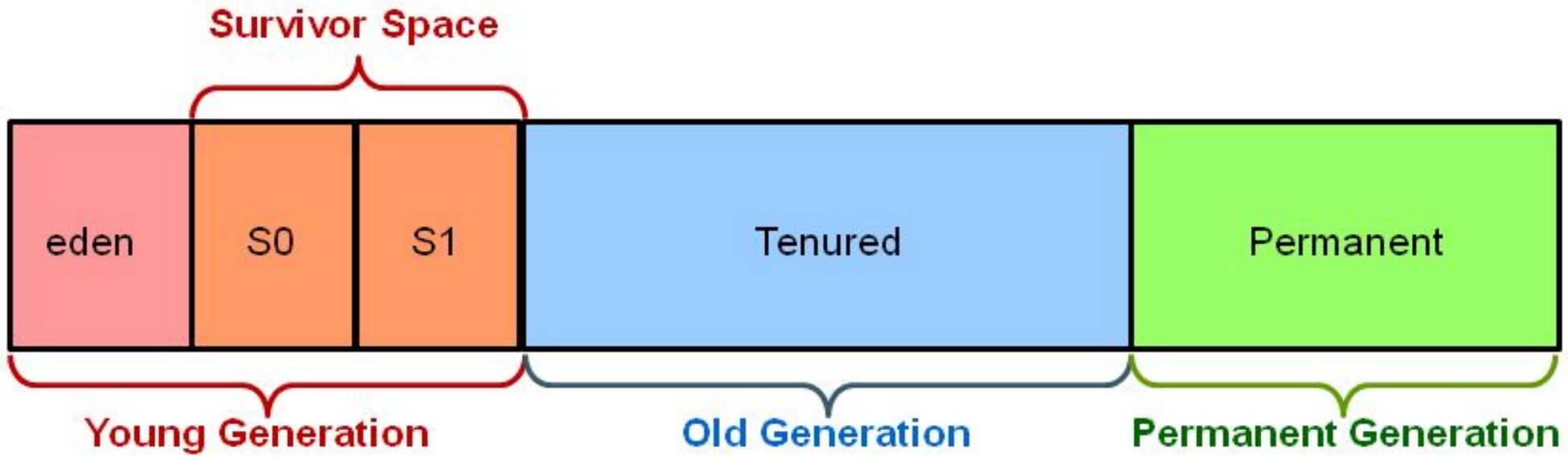
JIT compiler

- introduced to Sun JVM around java 1.2/1.3
- almost no optimizations during initial compilation (javac), generated bytecode mostly reflects .java code
- at the beginning bytecode is interpreted, only hot methods are compiled to native code
- advantages:
  - runtime information can lead to interesting optimizations, e.g. inlining virtual method call because only single class with given interface is currently loaded, removing null checks
  - leveraging platform specific optimizations during runtime
  - code written and compiled for java X can benefit from future performance improvements in java X + 1

JIT compiler

- disadvantages
  - initial performance is very low (JVM warmup)
  - compilation and optimization costs
  - unpredictable
  - optimistic assumptions can fail - code returns to being interpreted
    and awaits second JIT compilation

Memory layout

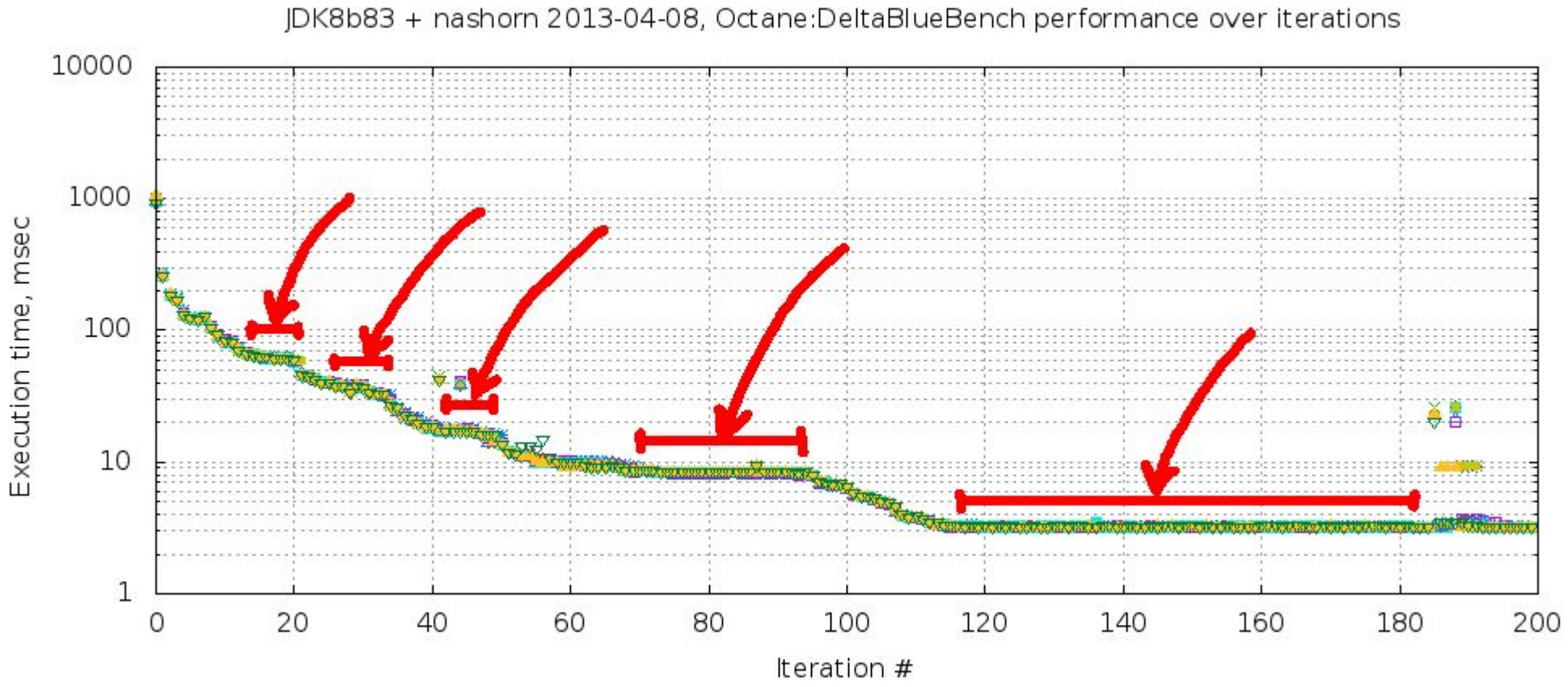


http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Garbage collection

- cheap allocation, paying the price later during GC
- STW (stop the world) pause
- more than single algorithm available
  - single threaded STW
  - multi threaded STW
  - CMS - only old generation, “mostly concurrent”
  - G1 - both young and old generation, “mostly concurrent”
- many tuning parameters:
  - usually best to use the defaults and rely on JVM runtime analysis
  - using arcane options can lead to very fragile performance
  - single JVM - single GC - multiple allocation profiles and requirements

Warmup



http://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf

Profiling metrics and tools - JVM level

Command line tools

- bin directory has 49 executables, some of them are:
  - **jstat** - GC and JIT statistics
  - **jstack** - stack trace of all threads, deadlock detection
  - **jps** - list of java processes, prints JVM options
  - **jinfo** - more detailed information about java process
  - **jmap** - heap histogram, heap dump
- available on each system with JDK

Command line options

- GC logs
  - -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -Xloggc:<file>
  - very useful when investigating GC issues
- Peeking under the JIT hood
  - add -XX:+UnlockExperimentalVMOptions just to be on the safe side
  - -XX:+PrintCompilation -XX:+CITime - basic information about compilation
  - -XX:+PrintInlining - outputs information what methods were inlined and what weren't and why
  - -XX:+PrintAssembly - shows what assembly code was generated by the JIT compiler

Profilers - VisualVM

- bundled with JDK, free of charge
- CPU information
- detailed memory information with VisualGC
- sampling (CPU/memory)
- profiling (CPU/memory)
- JMX browser
- thread information
- with a little effort can be used with remote JVMs

Profilers - Oracle Mission Control

- Everything available in VisualVM and more
- Free for development, have to pay when used in production
- Very low overhead (according to the Oracle) due to using internal and undocumented APIs
- Available since JDK 7u40

Eclipse memory analyzer

- analyzes full heap dump that can be obtained using jmap tool
- it can be used to obtain information like:
  - finding biggest objects on the heap
  - learning why specific objects are not removed by GC (GC roots)
  - listing dominating objects on the heap
- allows to inspect field values of specific objects
- may require significant memory during first pass through the heap dump but subsequent openings will be a lot faster

Microbenchmarking

What?

Micro + benchmark

performance measurement of a **very small** piece of code, something that might take µs or ns to run

How?

Java Microbenchmarking Harness

- benchmark code generation, driven by annotations.
- generated classes and all their dependencies get packaged in an all-in-one runnable jar
- benchmark runner supporting single threaded, multi-threaded and thread groups
- pluggable profilers
- multi-language support
- reporting formats - JSON, CSV, etc.

JMH examples

```
package jmh

import org.openjdk.jmh.annotations._

/**
  * @author nuk
  */
@Warmup(iterations = 5)
@Measurement(iterations = 20)
@Fork(1)
@Threads(8)
class HelloWorld {

  @Benchmark
  def yo(): Unit = {
    // this method was intentionally left blank.
  }

}
```

JMH examples

AVSYSTEM

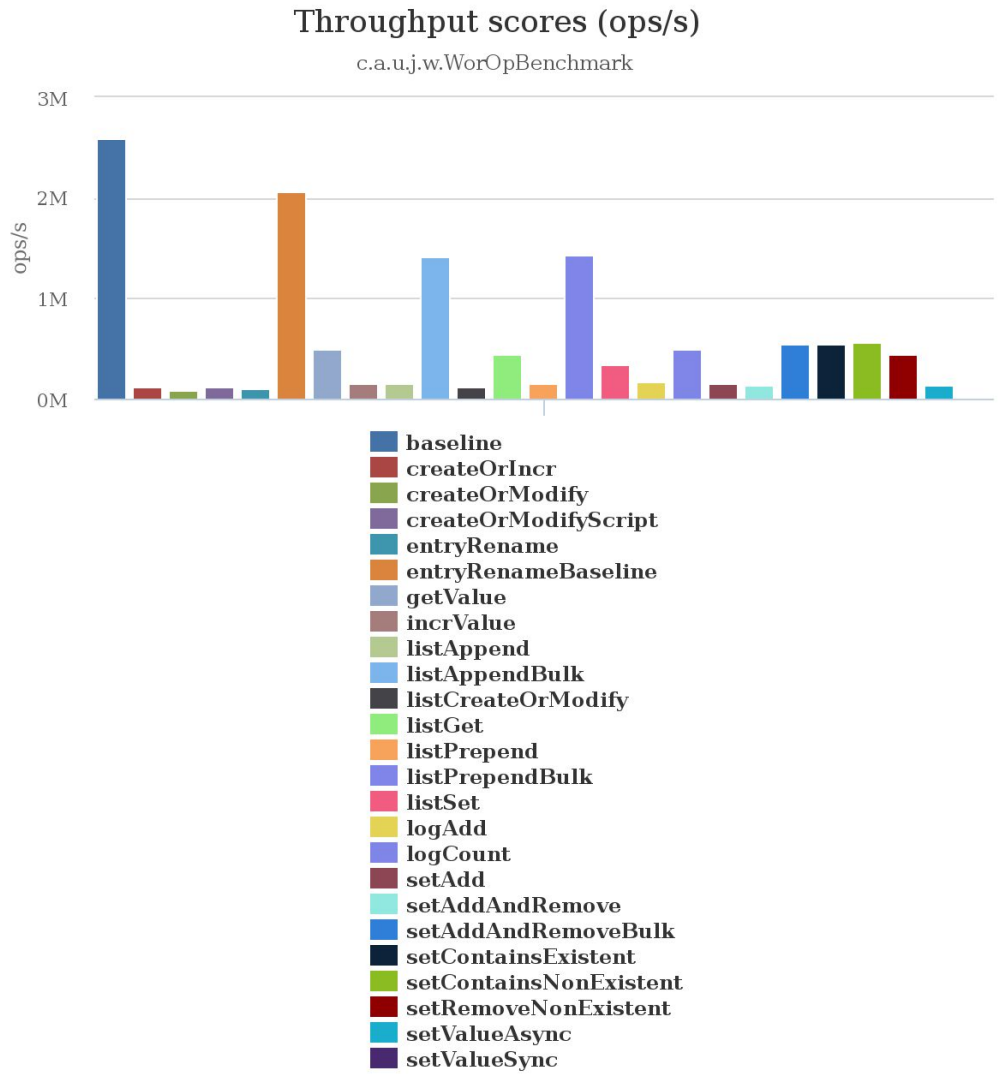
```scala
@Benchmark
@OperationsPerInvocation(OpBatch)
def setAddAndRemove(bh: Blackhole, state: BenchmarkState): Unit = {
  import state.manager.executor
  val strings = (1 to OpBatch).map(_ => nextString())
  val f1 = strings.map(state.set.add(_))
  val f2 = strings.map(state.set.remove(_))
  bh.consume(Await.result(Future.sequence(f1), timeout))
  bh.consume(Await.result(Future.sequence(f2), timeout))
}
```

JMH examples

```
@State(Scope.Benchmark)
class BenchmarkState {

  val manager = new WorManager(
    new Config("127.0.0.1", 6379, 1, "jmh"),
    new ActorSystemProvider {lazy val system = ActorSystem("WorOpBenchmark")}
  )
  var wor: Wor = _
  var value: WorValue = _
  var set: WorSet = _
  var log: WorHyperLogLog = _
  var list: WorList = _


  @Setup(Level.Iteration)
  def setup(): Unit = {
    Await.result(manager.executor.flushdb(), timeout)
    import manager.executor
    wor = Await.result(manager.getOrCreateWor("values"), timeout)
    val f = Future.sequence(
      Seq(
        wor.createValue("value", "1").map(value = _),
        wor.createSet("set", "a").map(set = _),
        wor.createHyperLogLog("log", "a").map(log = _),
        wor.createList("list", "a").map(list = _)
      )
    )
    Await.result(f, timeout)
  }


  @TearDown(Level.Iteration)
  def flush(): Unit = {
    Await.result(manager.executor.flushdb(), timeout)
  }

  @TearDown
  def shutdown(): Unit = {
    manager.system.terminate()
    Await.result(manager.system.whenTerminated, timeout)
  }
}
```

Result analysis

- Simple charts for JMH benchmarks http://nilskp.github.io/jmh-charts/
- GUI for comparing JMH results https://github.com/akarnokd/jmh-compare-gui



Java vs Scala

divided we fail

Resources

- GC
  - https://plumbr.eu/java-garbage-collection-handbook
  - http://mechanical-sympathy.blogspot.com/2013/07/java-garbage-collection-distilled.html
- Tools
  - https://github.com/AdoptOpenJDK/jitwatch
  - https://github.com/giltene/jHiccup
  - https://github.com/chewiebug/GCViewer
  - http://gceasy.io/
- Microbenchmarking
  - Nanotrusting the Nanotime
  - Scala JMH Samples
  - Java JMH Samples
  - https://www.youtube.com/watch?v=VaWgOCDBxYw
- Performance analysis
  - https://www.youtube.com/watch?v=dqg0R3gYGac
  - https://www.youtube.com/watch?v=nZfNehCzGdw
  - http://shipilev.net/
  - http://mechanical-sympathy.blogspot.com/

Conclusions

- intuition is almost always wrong (unless you rock)
- never trust anything (unless checked before… and after)
- challenge everything (especially these slides)
- embrace failure (especially your failures)
- grind your teeth, and redo the tests (especially yours)


UMP
Thank you!
You can go back to coding now
unless you don’t know what to code,
in that case visit: rozkmina.com