IT IS SLOW

...

Performance analysis and optimisation on the JVM

Materials

Code

```
git clone https://github.com/srvaroa/jbcnconf
```

check README for build/exec instructions

Apps

JDK (Hotspot > OpenJDK, but both fine)

Vagrant + VirtualBox (optional)

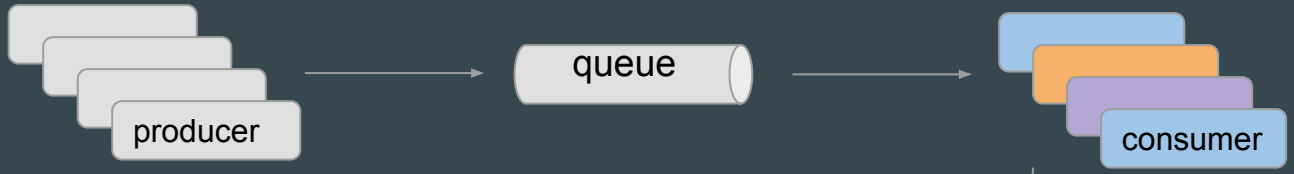
Slides from BcnJUG meetup

http://bit.ly/1UJSeTW

Agenda

- Learn to analyze aspects of an app’s performance
- Tools
- Some common pathologies
- Some JDK tools and patterns to address them

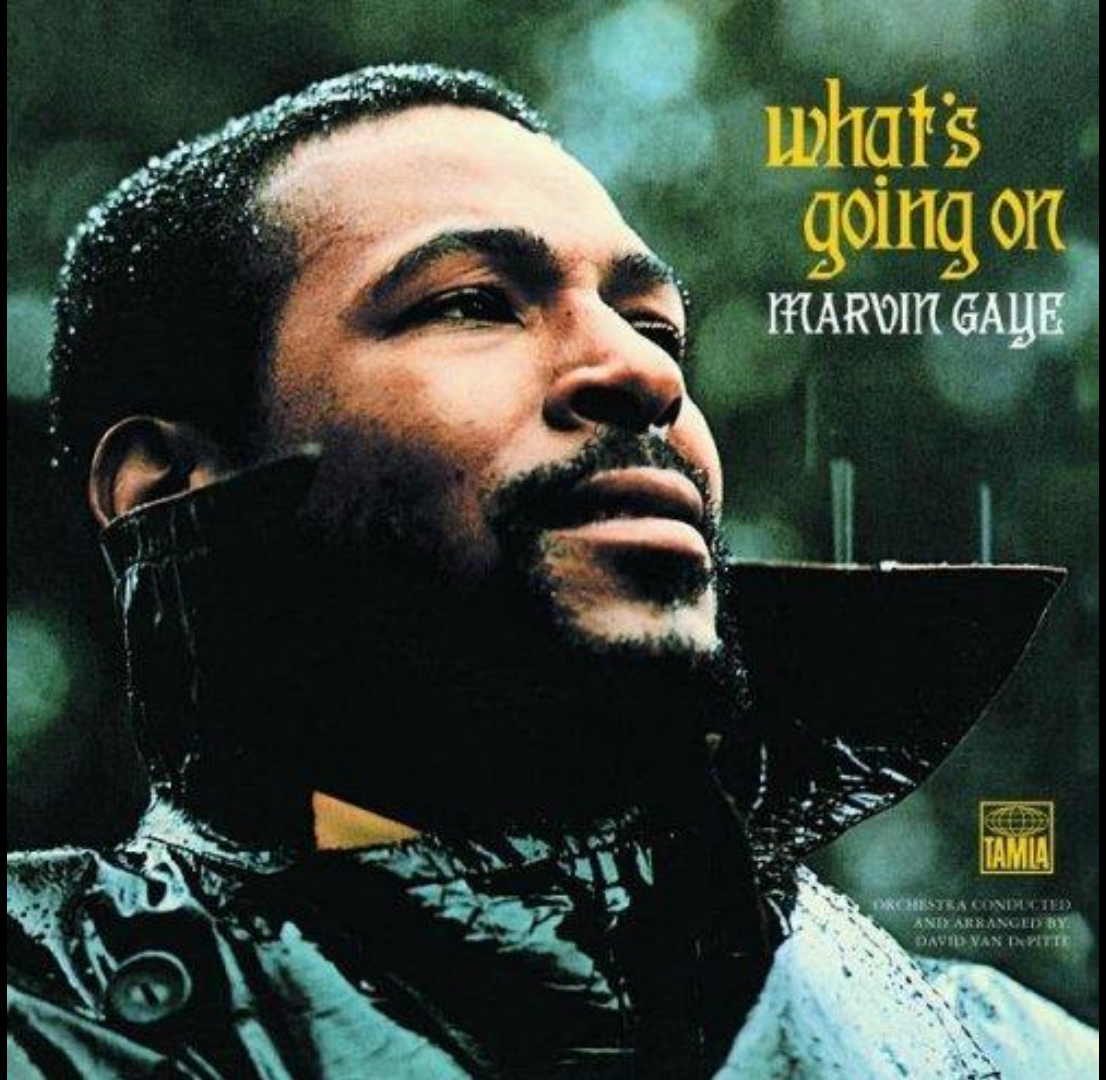
SimpleApp



Use a Supplier<Long>
to emit numbers

Use different
Consumer<Long> to
do stuff with them

Concurrency


what's going on
MARVIN GAYE
TAMLA
ORCHESTRA CONDUCTED AND ARRANGED BY DAVID VAN DePITTE

JDK tools

Why?

- Usually, only available tool (e.g.: remote / production / customer deployments)

JDK tools

CPU

```
jstack -l $PID 1000
```

JDK tools

Memory

```
jstat -gc $PID $INTERVAL

jmap -heap $PID

jmap -histo $PID

jmap -histo:live $PID
```

JDK tools

Performance counters

```
jcmd $PID PerfCounter.print
```

Flight Recorder + Mission Control

```
OSX: /Library/Java/Home/bin/jmc

Linux: $JAVA_HOME/bin/jmc
```

Run App

Attach MC

Flight Recorder + Mission Control

```
-XX:+UnlockCommercialFeatures
-XX:+FlightRecorder
-XX:StartFlightRecording=duration=60s,filename=rec.jfr,
settings=profile
```

Profiles (editable) `$JAVA_HOME/jre/lib/jfr/`

```
-XX:FlightRecorderOptions=loglevel=trace
```

Experiments

Change queue sizes

Change producer + consumer threads (symmetric, asymmetric)

Change supplier implementation

Comment out one of the Consumer implementations

Use different logging mechanisms

Benchmarking

JMH

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

http://openjdk.java.net/projects/code-tools/jmh/

Hello World:

```
com.github.srvaroa.jmh.IterableBenchmark
```

Documentation:

https://github.com/midonet/midonet/blob/master/docs/micro-benchmarks.md

Observations

Observations

Contention is expensive

Easy synchronization tools are too inefficient

- synchronized is borderline harmful

Better sync tools requires complex code

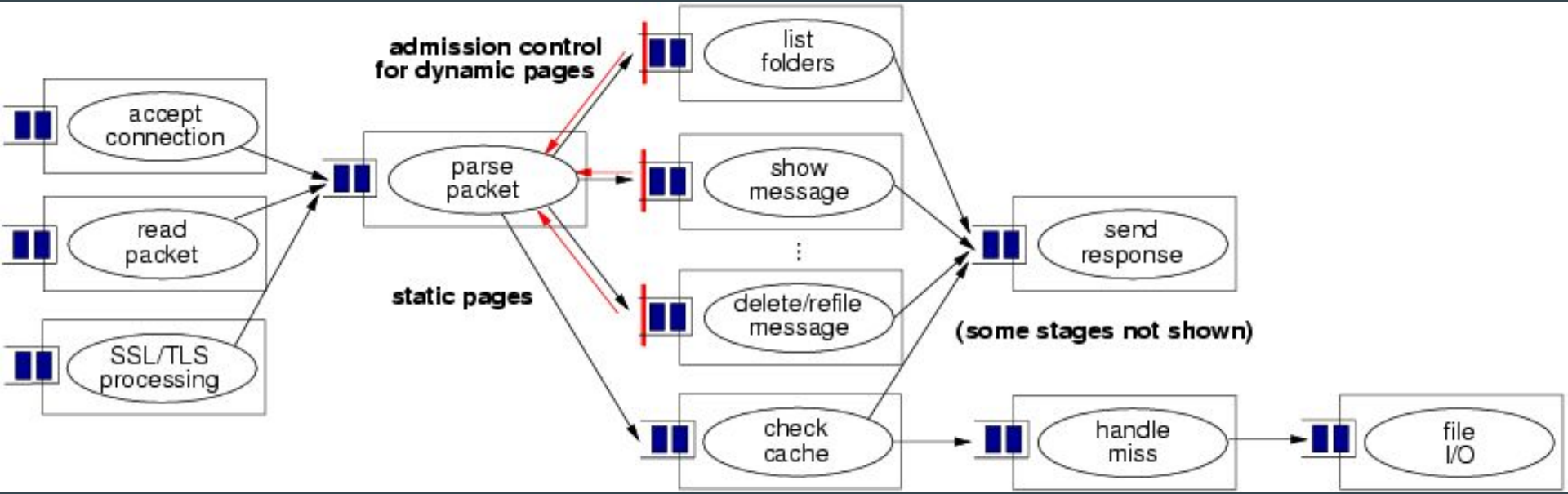
Optimisation requires hardware-specific knowledge

Lots of black boxes underneath (JDK, JVM, OS, CPUs, Network cards...)

Queues are troublesome at scale

Queues are key in SEDA architecture, but a major hotspot

https://issues.apache.org/jira/browse/CASSANDRA-10989: Move from SEDA to TPC (thread per core)



OneToOneApp

Paired producer - consumer

Use `JcToolsSpscBackedQueue` and check code inside

Frameworks

ReactiveX

https://github.com/ReactiveX/RxJava/ - in provided VM

Rx is push based. Paradigm change: work enters a pipeline, functions in a single thread until told otherwise.

Does this make sense in light of what we saw above?

> Mental model change: from stages communicated with queues to flows of data

Caveat: not all Rx operators are optimal.

Akka (Actors + Streams)

https://github.com/akka/akka - in provided VM

Good points

- Again, avoid contention in processing units (Actors)

Bad points

- Queues, CPU hopping
- Small units of work have large contention on Dispatcher queues

Notes on RX & Actors

They are both a means to model your system, let you define separately:

- Stages, units of work (Akka actors)
- Flows of computation (Rx, Akka streams)
- How they are scheduled (Akka: put a group of actors on a particular dispatcher)

Great in many applications, but some pitfalls, tradeoffs:

- Example: sending work to another dispatcher can be expensive (wake up thread)
- Always tune / profile

Perf & Flame Graphs

Perf (GNU/Linux)

Setup VM

```
$ cd vm
$ vagrant up
```

By hand: check end of `vm/VagrantFile` for packages

Build JAR locally, then copy into VM:

```
$ cp ./build/libs/jbcnconf-0.1-SNAPSHOT-all.jar ../vm/data # from code/

$ cd ../vm && vagrant ssh
```

File will be at `/vagrant_data/jbcnconf-0.1-SNAPSHOT-all.jar`

Perf stat (not available in VM)

```
perf stat -d -p $PID   # also: cat /proc/$PID/status
```

```
Performance counter stats for process id '46185':

     46802,773132 task-clock                #    0,367 CPUs utilized
            24521 context-switches          #    0,001 M/sec
             1056 CPU-migrations            #    0,000 M/sec
              450 page-faults               #    0,000 M/sec
     129724461306 cycles                    #    2,772 GHz                      [40,03%]
     120451823606 stalled-cycles-frontend   #   92,85% frontend cycles idle     [40,40%]
     101327998695 stalled-cycles-backend    #   78,11% backend  cycles idle     [40,56%]
       5710255927 instructions              #    0,04  insns per cycle
                                            #   21,09  stalled cycles per insn [50,69%]
       1334448420 branches                  #   28,512 M/sec                    [50,80%]
         34780770 branch-misses             #    2,61% of all branches          [50,69%]
       1540840733 L1-dcache-loads           #   32,922 M/sec                    [50,22%]
       4551850342 L1-dcache-load-misses     #  295,41% of all L1-dcache hits    [50,02%]
         71937736 LLC-loads                 #    1,537 M/sec                    [39,83%]
         42543914 LLC-load-misses           #   59,14% of all LL-cache hits     [39,78%]

    127,428135422 seconds time elapsed
```

Flame Graphs (available in VM)

Visualization for sampled stack frames. Great to see where your CPU is spending time.

http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

Java: stacks showed only application & JDK code paths, not OS.

Since 1.8 u60 b19: `-XX:+PreserveFramePointer` enables end-to-end stacks

http://techblog.netflix.com/2015/07/java-in-flames.html

http://www.brendangregg.com/FlameGraphs/cpu-mixedmode-flamegraph-java.svg

Flame Graphs

On Java, config instructions (already in VM):

https://github.com/brendangregg/FlameGraph
http://isuru-perera.blogspot.com.es/2015/07/java-cpu-flame-graphs.html

Flame Graphs

Preconfigured in provided VM, easy to generate:

```
$ java -cp $JAR -XX:+PreserveFramePointer com.github.srvaroa.queue.SimpleApp
$ jps -m
11633 Jps -m
11606 SimpleApp
$ cd perf-map-agent $$ bin/perf-java-flames 11606
Recording events for 15 seconds (adapt by setting PERF_RECORD_SECONDS)
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.214 MB /tmp/perf-11606.data (~9357 samples)
]
Flame graph SVG written to PERF_FLAME_OUTPUT='/home/vagrant/perf-map-
agent/flamegraph-11606.svg'.
$ cp /vagrant/perf-map-agent/flamegraph-11606.svg /vagrant_data
```

Use browser to open in host at: `jbcnconf/vm/data/flamegraph-11606.svg`

Q & A

...

Thanks!

Galo Navarro *@srvaroa*

Guillermo Ontañón *@gontanon*