

Performance Methodology





Aleksey Shipilev
Java Performance
Oracle
@shipilev

Kirk Pepperdine
Java Performance
Kodewerk
@kcpeppe

Aleksey Shipilev

Speaker Bio

- 7+ years of (Java) Performance
 - 3 years at Intel
 - 4 years at Sun/Oracle

Projects

- Apache Harmony
- Oracle/OpenJDK
- SPECjbb201x
- https://github.com/shipilev/

Kirk Pepperdine



Speaker Bio

- I5 year Performance tuning across many industries
 - Background in super and exotic computing platforms
- Helped found <u>www.javaperformancetuning.com</u>
- Developed Java performance seminar (www.kodewerk.com)
- Member of Java Champion program, Netbeans Dream Team
- Recently founded JClarity,
 - a company who's purpose is to redefine performance tooling
 - Invite you to join Friends of JClarity (www.jclarity.com)



#include <disclaimer.h>

The resemblance of any opinion, recommendation or comment made during this presentation to performance tuning advice is merely coincidental.

Measure Don't Guess

- Hypothesis free investigations
- Progress through a series of steps to arrive at a conclusion

Introduction

Computer Science → **Software Engineering**

- Way to construct software to meet functional requirements
- Abstract machines
- Abstract and composable, "formal science"

Software Performance Engineeering

- "Real world strikes back!"
- Researching complex interactions between hardware, software, and data
- Based on empirical evidence

Benchmarking



Experimental Setup

You can't go any further without the proper test environment

- **Relevant:** reproduces the phenomena
- Isolated: leaves out unwanted effects
- **Measurable:** provides the metrics
- **Reliable:** produces consistent result



WWW.PHDCOMICS.COM

"Piled Higher and Deeper" by Jorge Cham www.phdcomics.com

Relevant and Isolated

- Hardware
 - Production like
 - Phantom bottlenecks
 - Quiet
 - Software
 - Test harness
 - Load injector and acceptor
 - Data
 - Production like in volumes and veracity

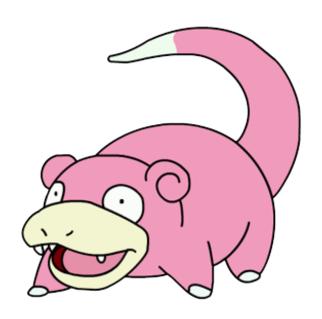
Measurable and Reliable

- Usage Patterns
 - Describes load
 - Use case + number of users and transactional rates, velocity
 - Performance requirements
 - Trigger metric is most likely average response time
- Validation
 - Test the test!
 - Make the sure your bottleneck isn't in the test harness!

Performance Testing Steps

- Script usage patterns into a load test
- Install/configure application to the same specs as production
- Setup monitoring
 - Performance requirements
 - OS performance counters and garbage collection
- Kill everything on your system
- Spike test to ensure correctness
- Load test
- Validate results
- Repeat as necessary

Demo 1



Introducing the test

Metrics

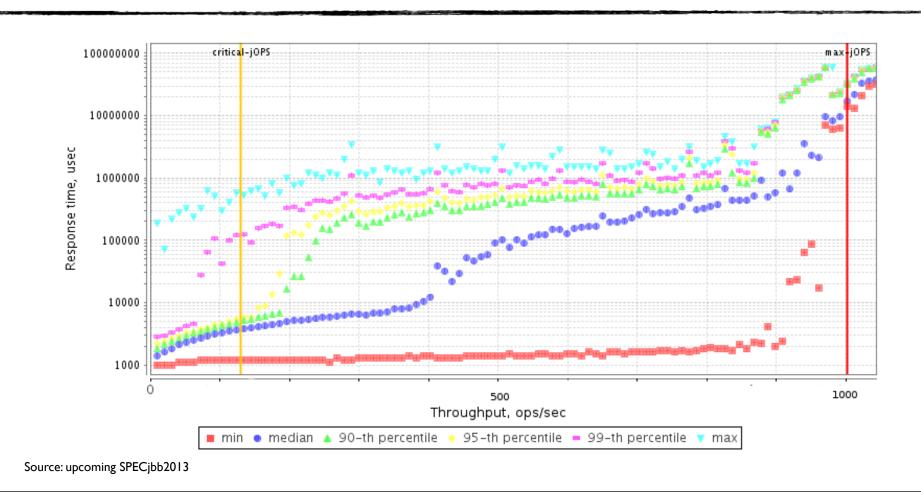
Throughput (Bandwidth)

- How many operations are done per time unit?
- Have many forms: ops/sec, MB/ sec, frags/sec
- Easiest to measure
- Easiest to interpret

Time (Latency)

- How much time one operation took?
- Targets many things: latency, response time, startup time
- Generally hard to measure (reliably)

Bandwidth vs. Latency



Little's Law

The nice artifact of the queuing theory

$$L = \lambda T$$

L: number of outstanding requests, concurrency level

λ: throughput

τ: service time

Implications:

- Under the same L, λ is inversely proportional to τ
- Under known λ and τ , you can infer the L

Pop Quiz

Imagine the application with two distinct phases

- Part A takes 70% of time, potential speedup = 2x
- Part B takes 30% of time, potential speedup = 6x
- Which one to invest in?

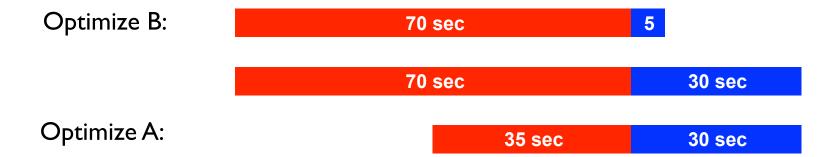
70 sec

30 sec

Pop Quiz

Imagine the application with two distinct phases

- Part A takes 70% of time, potential speedup = 2x
- Part B takes 30% of time, potential speedup = 6x
- Which one to invest in?



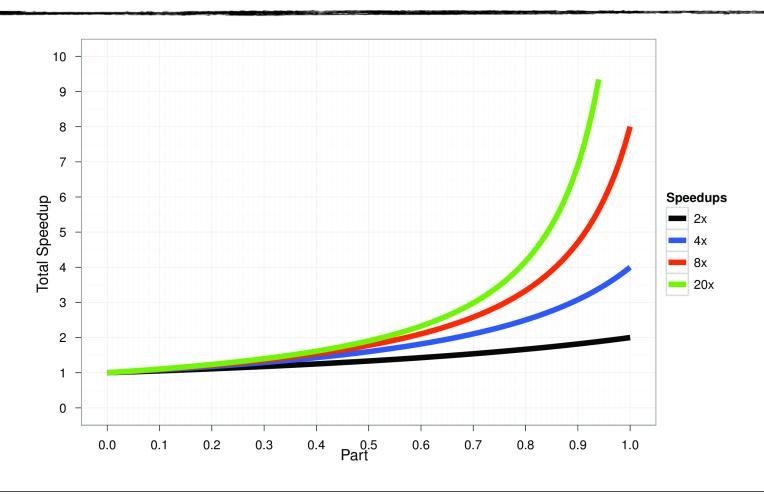
Ahmdal's Law

We can generalize this observation as:

$$Part(A) = \frac{A}{A+B}$$

$$SpeedUp = \frac{1}{(1 - Part(A)) + \frac{Part(A)}{SpeedUp(A)}}$$

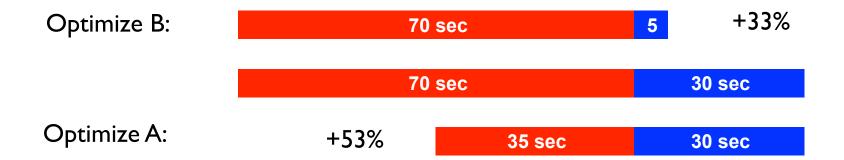
Ahmdal's Law Limits Speedups



Applying Ahmdal's Law

Imagine the application with two distinct phases

- Part A takes 70% of time, potential speedup = 2x
- Part B takes 30% of time, potential speedup = 6x
- Which one to invest in?



Where Ahmdal's Law Breaks Down

Composability

- Given two functional blocks, A and B
- The difference with executing (A seq B) or (A par B)?

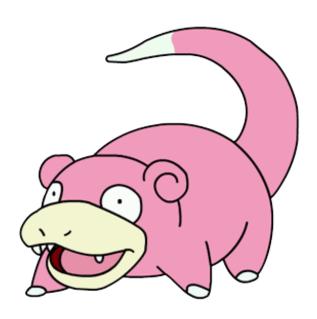
Functional-wise:

- Result(A seq B) == Result(A par B)
- "Black box abstraction"

Performance-wise:

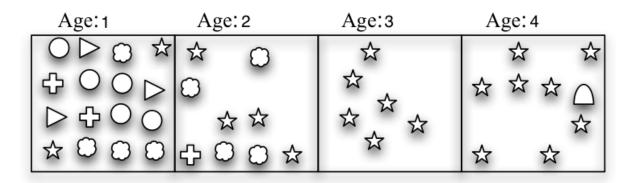
- Performance(A seq B) ??? Performance(A par B)
- No one really knows!

Demo 2



Ensure test is reliable

Generational Counts



Generational Counts (2)

Age:1	Age: 2	Age: 3	Age:4	
000 \$	☆ ○	☆	☆	☆
⊕ O O ⊳	0	☆	☆ ☆ ·	☆ △
▷⇔⊙⊙	☆☆	☆ ☆ ☆		- ☆
\$000	⊕00☆	☆ ~	☆ 5	☆~

Object	Generations	Count	Classify
0	1	1	Normal
4	1, 2	2	Normal
	4	1	Cached
	1	1	Normal
	1, 2	2	Normal
☆	1, 2, 3, 4	4	Leak

Change something somewhere in some specific way!

Change something somewhere in some specific way!



Change something somewhere in some specific way!

■ What?

■ Where?

■ How?

Change something somewhere in some specific way!

■ What prevents the application to work faster?

■ Where it resides?

■ How to change it to stop messing with performance?

Change something somewhere in some specific way!

- What prevents the application to work faster? Courage, experience, and monitoring tools
- Where it resides?
 Courage, experience, and profiling tools
- How to change it to stop messing with performance?
 Courage, experience, your brain, and your favorite IDE

Top-Down Approach (classic)

System Level

■ Network, Disk, CPU/Memory, OS

Application Level

Algorithms, Synchronization, Threading, API

Microarchitecture Level

■ Code/data alignment, Caches, Pipeline stalls

Top-Down Approach (Java)

System Level

Network, Disk, CPU/Memory, OS

JVM Level

■ GC, JIT, Classloading

Application Level

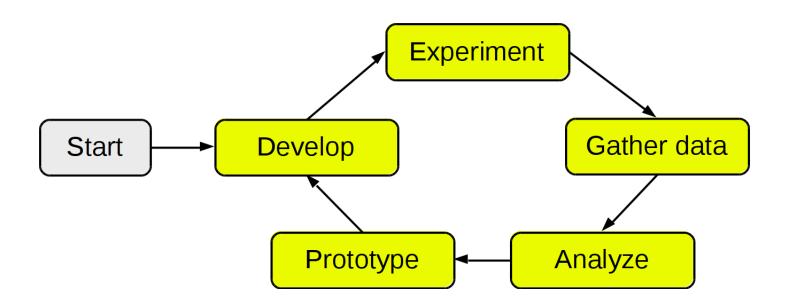
Algorithms, Synchronization, Threading, API

Microarchitecture Level

■ Code/data alignment, Caches, Pipeline stalls



+ Iterative Approach

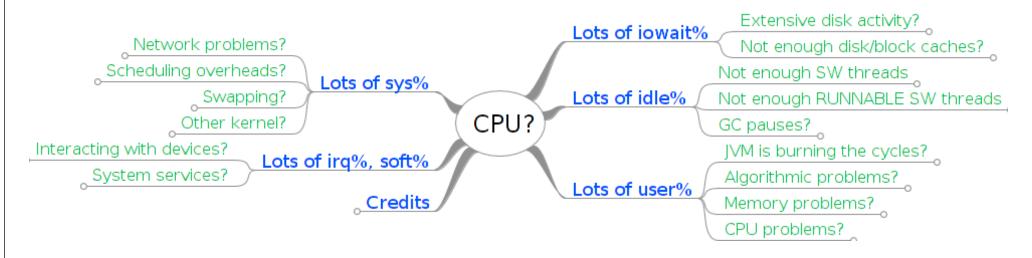


- Start new phase when functional tests are passed
- Single change per cycle
- Document the changes

System Level



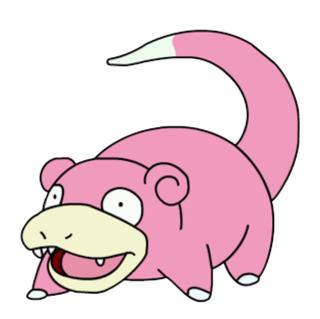
System Level (CPU)



The entry point is CPU utilization!

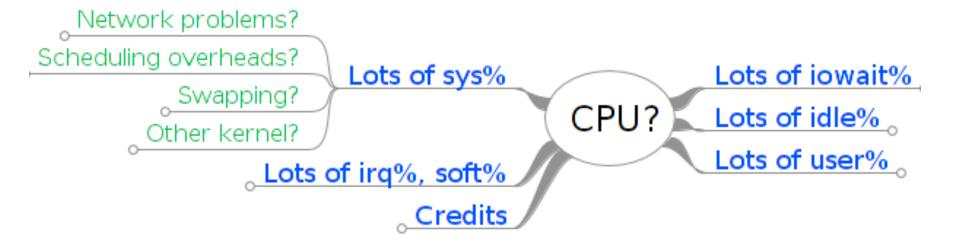
- Then, you have multiple things to test for
- Depending on sys%, irq%, iowait%, idle%, user%
- Need tools to examine each particular branch

Demo 3



First dive into the monitoring

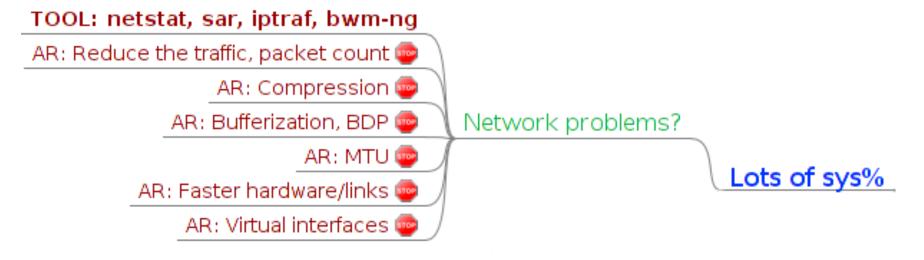
System Level (sys%)



Not particularly the application code fault

- Most obvious contender is network I/O
- Then, scheduling overheads
- Then, swapping
- Then, in minor cases, other kernel

System Level (sys%, network)



One of the major contributors to sys%

- In many cases, hardware/OS configuration is enough
- In other cases, application changes might be necessary

System Level (sys%, scheduling)

TOOL: vmstat, mpstat, sar

AR: Reduce the amount of worker threads 🐵

AR: Less context switches

AR: Scheduling groups, quanta adjustments, priority 🚭

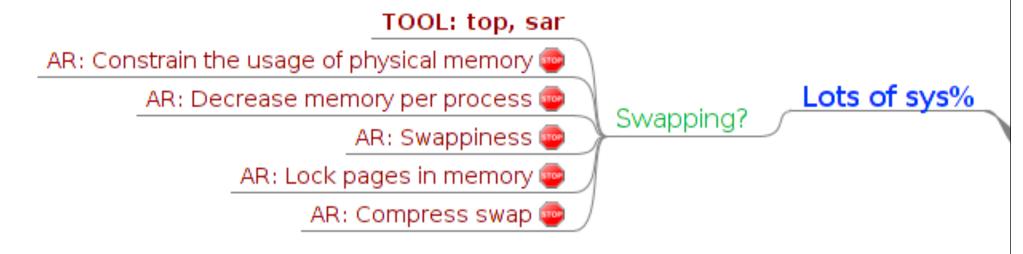
Scheduling overheads?

Lots of sys%

The symptom of the unbalanced threading

- Lots of voluntary context switches (thread thrashing)
- Lots of involuntary context switches (over-saturation)

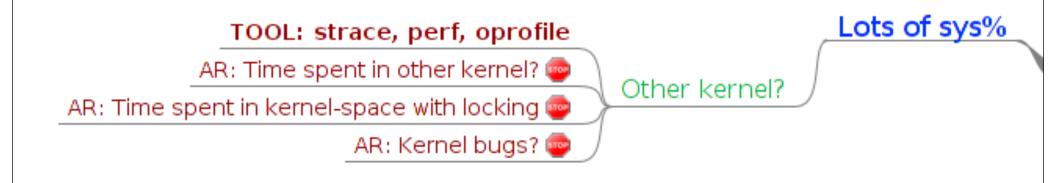
System Level (sys%, swapping)



Swapping is the killer for Java performance

- The target is to avoid swapping at all costs
- Swapping out other processes to save the memory is good

System Level (sys%, other)



Sometimes kernel is your enemy

- Unusual API choices from the JVM and/or application
- (Un)known bugs

System Level (irq%, soft%)

TOOL: mpstat, sar

AR: Interrupt offload 😳

Interacting with devices?

Lots of irq%, soft%

AR: IRQ balancing 🐵

Usual thing when interacting with the devices

- Sometimes IRQ balancing is required
- Sometimes IRQ balancing is expensive

System Level (iowait%)

Extensive disk activity?

Not enough disk/block caches?

Lots of iowait%



Expected contributor with disk I/O

- Watch for disk activity
- Watch for disk throughput
- Watch for disk IOPS

System Level (iowait%, disk)

TOOL: iostat, sar

AR: Reduce the disk activity 😳

AR: HW caching/bufferization 🐵

AR: SW caching/bufferization

AR: More disks always help (but not your budget)

Extensive disk activity?

Lots of iowait%

Is that amount of I/O really required?

- Caching, bufferization are your friends
- More (faster) disks can solve throughput/IOPS problems

System Level (iowait%, caches)

TOOL: top, sar

AR: Increase cache memory (reduce other usages) 👄

AR: Get easy on flush()-es and cache invalidations 👄

AR: More disks always help (but not your budget)

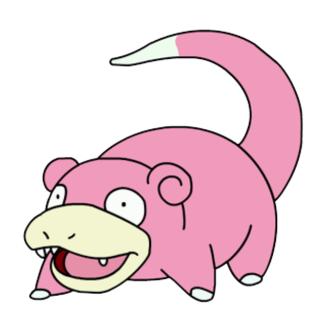
Not enough disk/block caches?

Lots of iowait%

More caching helps?

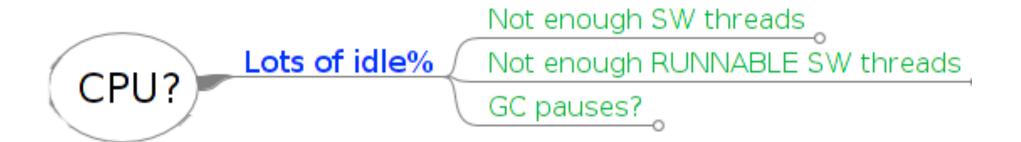
- Reduce other physical memory usages, free up for caches
- Trade in performance over consistency

Demo 4



Fixing the iowait problem \rightarrow next step

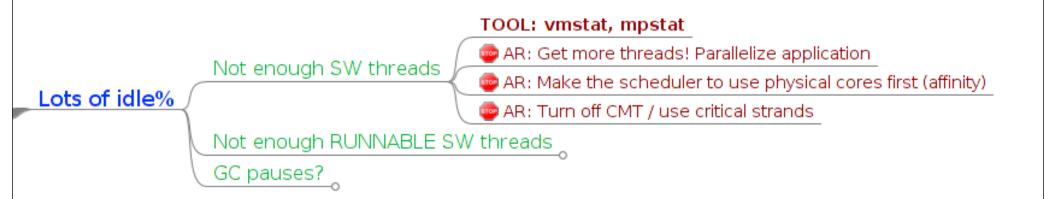
System Level (idle%)



There are resources, but nobody uses them?

- This is admittedly easy to diagnose
- ...and very easy to miss

System Level (idle%, threads)



Running low-threaded applications on manycore hosts

- The signal for you to start parallelizing
- Or, reduce the number of available HW strands

System Level (idle%, threads)



There are not enough threads ready to run

- Locking?
- Waiting for something else?

System Level (idle%, GC)

Not enough SW threads

Not enough RUNNABLE SW threads

TOOL: -verbose:gc, etc

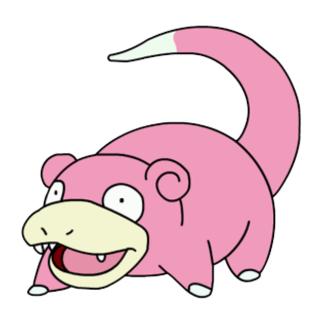
GC pauses? AR: More threads for GC

AR: Pause-targeted GC-specific tuning

Very rare, and surprising case

- Application is highly threaded
- GC is frequently running with low thread count
- The average CPU utilization is low

Demo 5

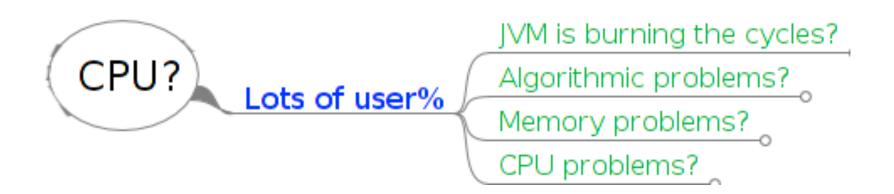


Fixing the idle problem \rightarrow next step

Application/ JVM Level



Application Level (user%)



Application/JVM is finally busy

- This is where most people start
- This is where profilers start to be actually useful

Application Level (Memory)

Lots of user%

Memory problems?

Caches

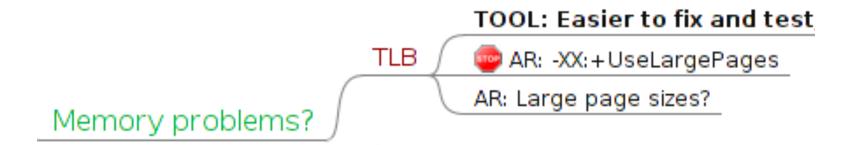
NUMA (NUCA)

Memory bandwidth

Memory

- The gem and the curse of von-Neumann architectures
- Dominates most of the applications (in different forms)

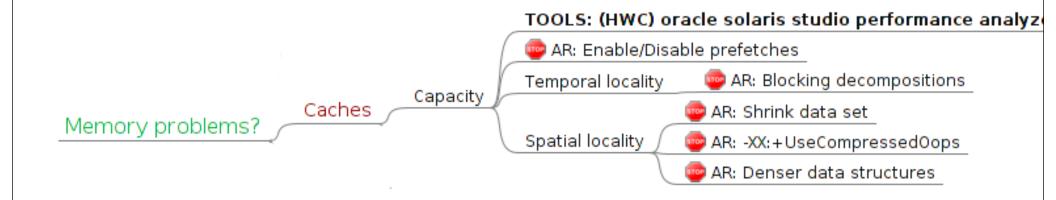
Application Level (TLB)



TLB

- Very important for memory-bound workloads
- "Invisible" artifact of virtual memory system

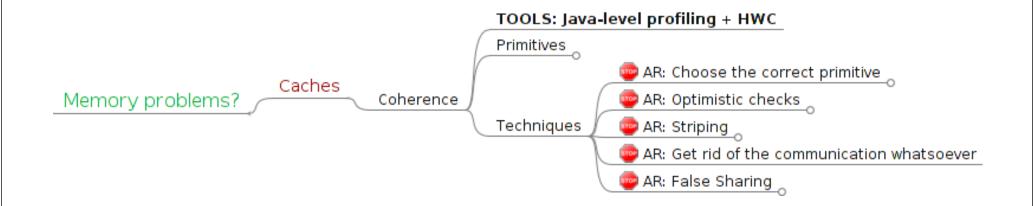
Application Level (Caches)



CPU caches: capacity

- Important to hide memory latency (and bandwidth) issues
- Virtually all applications today are memory/cache-bounded

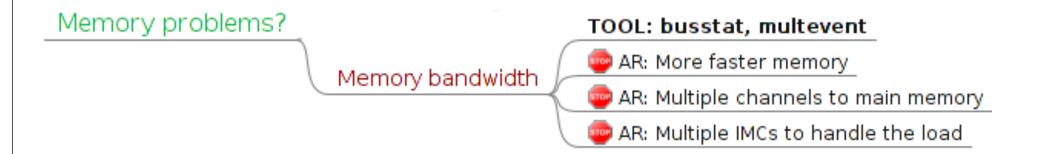
Application Level (Caches)



CPU Caches: coherence

- Inter-CPU communication is managed via cache coherence
- Understanding this is the road to master the communication

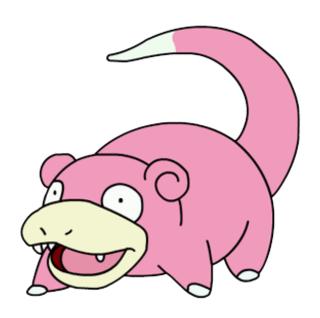
Application Level (Bandwidth)



Memory Bandwidth

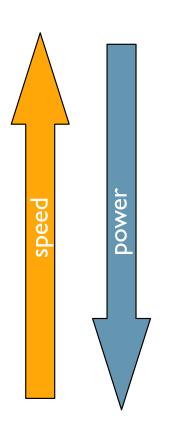
- Once caches run out, you face the memory
- Dominates the cache miss performance
- Faster memory, multiple channels help

Demo 6



Solving the concurrency problem \rightarrow next step

Coherence: Primitives



Plain unshared memory Plain shared memory

■ Provide communication

Volatile

■ All above, plus visibility

Atomics

■ All above, plus atomicity

Atomic sections

■ All above, plus group atomicity

Spin-locks

■ All above, plus mutual exclusion

Coherence: Optimistic Checks

It is possible at times to make an optimistic check

- Fallback to pessimistic version on failure
- The optimistic check has less power, but more performant

Coherence: Optimistic Checks

It is possible at times to make an optimistic check

- Fallback to pessimistic version on failure
- The optimistic check has less power, but more performant

```
ReentrantLock lock = ...;
int count = -LIMIT;
while (!lock.tryLock()) {
   if (count++ > 0) {
      lock.lock();
      break;
   }
}
```

Coherence: Striping

It is possible at times to split the shared state

- Much less contention on modifying the local state
- The total state is the superposition of local states

Example: thread-safe counter

- synchronized { i++; }
- AtomicInteger.inc();
- ThreadLocal.set(ThreadLocal.get() + 1);
- AtomicInteger[random.nextInt(count)].inc();

Coherence: No-coherence zone

If you can remove the communication, do that!

- Immutability to enforce
- Thread local states

Example: ThreadLocalRandom @ JDK7

- Random: use CAS to maintain the state
- ThreadLocalRandom: essentially, ThreadLocal<Random>
 - Can use plain memory ops to maintain the state

Coherence: (False) Sharing

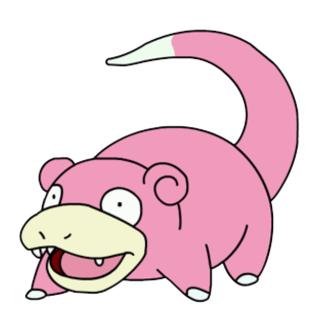
Communication quanta = cache line

- 32 128 bytes long
- Helps with bulk memory transfers, cache architecture
- Coherence protocols working on cache lines

False Sharing

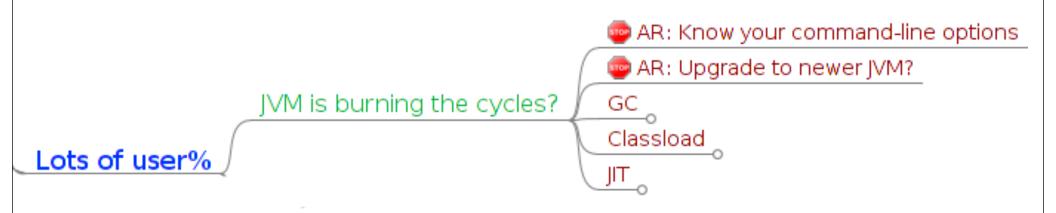
- CPUs updating the adjacent fields?
- Cache line ping-pong!

Demo 7



Diagnosing with allocation profiles

JVM Level



JVM is the new abstraction level

- Interacts with the application, mangles into application
- JVM performance affects application performance

JVM Level (GC)

JVM is burning the cycles?

GC

TOOL: -verbose:gc, -XX:+PrintGCDetails, VisualGC

AR: Tune Java heap, generations, and regions

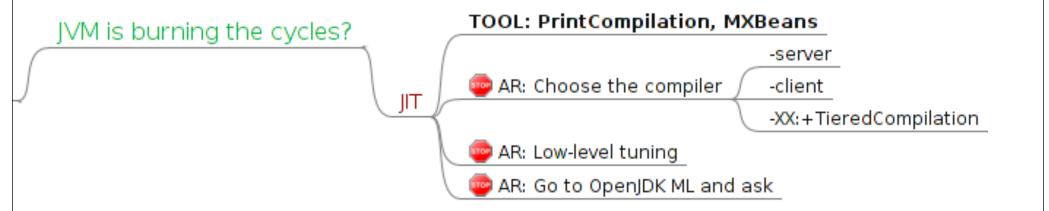
AR: Thread stack size

AR: (Un)usual tuning

GC

- Most usual contender in JVM layer
- Lots of things to try fixing (not covered here, see elsewhere)

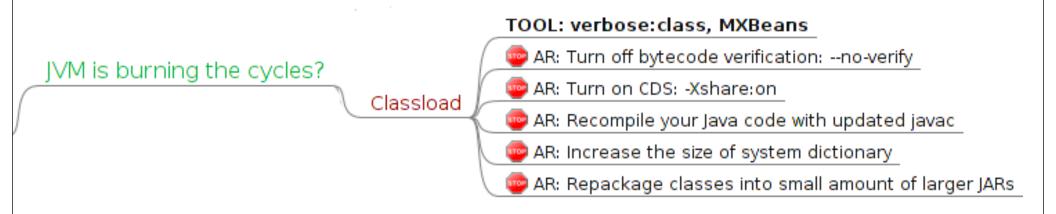
JVM Level (JIT)



JIT

- Very cool to have your code compiled
- Sometimes it's even cooler to get the code compiled better

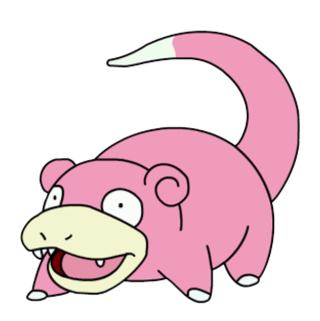
JVM Level (Classload)



Classload

- Important for startup metrics; not really relevant for others
- Removing the loading obstacles is the road to awe

Demo 8



Fixing the allocation problem

Application Level

Lots of user%

Algorithmic problems?

Algorithmic complexity
Caching/Memoizing

Busy-waiting

Batching and work scheduling

Application level

- In many, many cases, silly oversights in algorithms use
- Cargo cult of approaches, patterns, code reuse

Application Level (Algos)



Algorithmic Complexity

- Figuring out the straight-forward code has huge complexity
- Sometimes, the low-O code is slower than high-O code

Application Level (Caching)

Algorithmic problems?

Caching/Memoizing

TOOL: Profilers + Brain

AR: Memoize the results where appropriate

AR: Use new objects where appropriate

AR: For (distributed) caching the record size should be smaller

Application Caching

- Seems to be the answer to most performance problems?
- In fact, blows up the footprint, heap occupancy, etc.

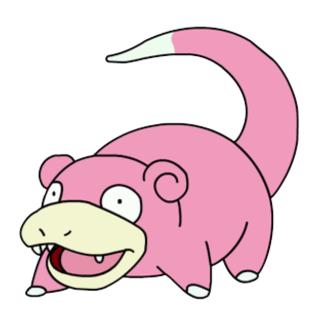
Application Level (Busy-waits)



Application Busy-Waits

- The natural instinct: blocked waits (with helping)
- For latency-oriented: busy-waits are profitable

Demo 9



Analyzing with execution profiles

uArch Level (CPU)

Lots of user%

Not enough CPU frequency?

CPU problems?

Not enough Execution Units?

ILP depleted?

CPU

- Most applications are not getting here
- A very simple capacity problem

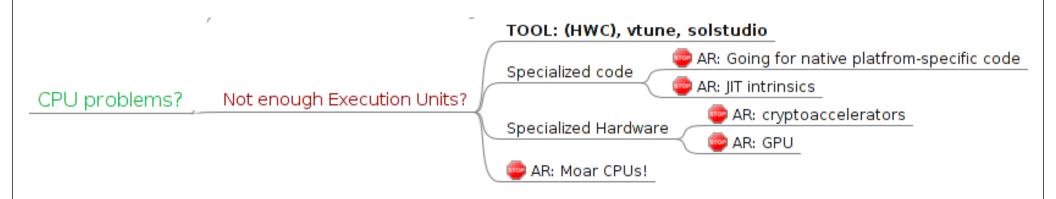
uArch Level (CPU, frequency)



CPU Frequency

- Exception: affects the memory/speculating performance
- How many servers out there are running with "ondemand"?

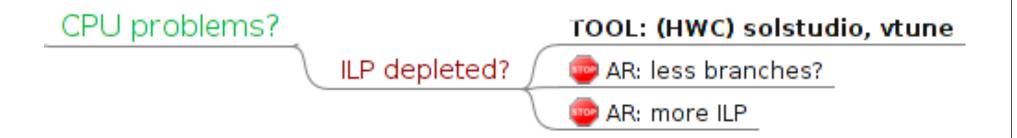
uArch Level (CPU, EU)



CPU, Execution Units

- Heavily-threaded hardware shares the CPU blocks
- Easy to run out of specific units with the homogeneous work

uArch Level (CPU, ILP)



Instruction Level Parallelism

- CPUs speculate aggressively
- Exposing less dependencies in the code help to speculate

Closing Thoughts



Q&A



Utilization = how busy the resource is?

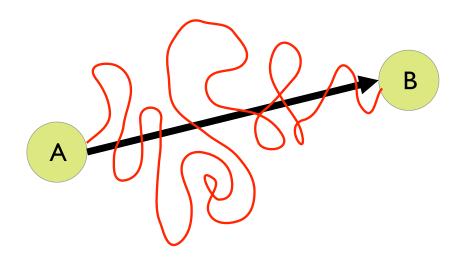
$$Utilization = \frac{ResourceBusyTime}{TotalTime}$$

Idle = how free the resource is?

$$IdleTime = 1 - Utilization$$

Efficiency = How much time is spent doing useful work?

- Not really possible to measure
- High Utilization != High Efficiency



SpeedUp = A is N times faster than B means:

$$SpeedUp = \frac{time(B)}{time(A)} = \frac{throughput(A)}{throughput(B)}$$

%Boost = A is P% faster than B means:

$$SpeedUp = 1 + \frac{n}{100\%}$$

$$Boost\% = (SpeedUp - 1) * 100\%$$

$$Boost\% = \frac{time(B) - time(A)}{time(A)}$$

$$Boost\% = \frac{throughput(A) - throughput(B)}{throughput(B)}$$

Performance

= Scalar Field in Config Space

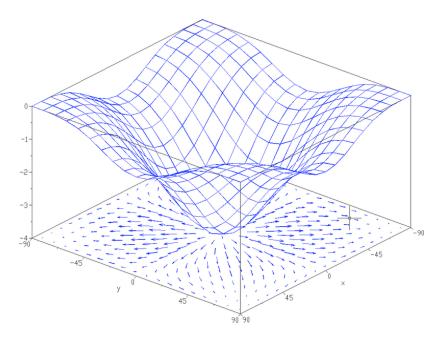
$$P:K^n \to \mathbb{R}$$

Scalability

= Gradient of PSF
$$S = \nabla P$$

Resource Scalability

= specific ∂P nponent in SC vector $S_i = \frac{\partial P}{\partial R_i}$



source: http://en.wikipedia.org/wiki/Gradient

Optimization Task

The configuration space can be humongous

- You don't want to traverse it all
- Or, you do want to exhaustive search if space is small

Random walks are inefficient

- Need to estimate the gradient in all N dimensions
- Means 2*N experiments per each step

Local estimates to rescue!

- Can predict if P would grow, should we add specific resource
- This is where the bottleneck analysis steps in

First step (mistakes)

We frequently hear:

- "I see the method foo() is terribly inefficient, let's rewrite it"
- "I see the profile for bar() is terribly high, at 5%, let's remove it"
- "I think our DBMS is a slowpoke, we need to migrate to [buzzword]"

Correct answer:

- Choose the metric!
- Make sure the metric is relevant!
- Your target at this point is improving the metric

Second step mistakes

"I can see the method foo() is terribly inefficient, let's rewrite!"

- ...what if the method is not used at all
- ...what if it accounts for just a few microseconds of time
- ...what if it does account for significant time, but...

Actually, not a bad idea

- ...as the part of controlled experiment
- ...if the changes are small, isolated, and painless to make

Second step mistakes

- "I can see the method bar() accounts for 5% of time, let's remove it!"
- ...what if the CPU utilization is just 6.25%?
- ...what if this method pre-computes something reused later?
- ...what if this method is indeed problematic, but...

Second step mistakes

"I think our database is the problem! Let's migrate to [buzzword]!"

- ...what if the you just depleted the disk bandwidth?
- ...what if your IT had shaped the network connection?
- ...what if your poor database just needs a cleanup?
- ...what if the database is indeed the bottleneck, but...

TLBs Detailed

Virtual memory operates on virtual addresses

- But hardware needs physical addresses to access memory
- Needs virtual → physical translation
- Tightly cooperates with OS (walks through page tables)

Extreme cost to do a single translation

- Happens on each memory access
- Let's cache the translated addresses!
- TLB = Translation Look-aside Buffer
- Granularity: single memory page