When the OS gets in the way

(and what you can do about it)

Mark Price
LMAX Exchange

Linux When the OS gets in the way

(and what you can do about it)

Mark Price
LMAX Exchange

It’s not the OS’s fault

- Linux is an excellent general-purpose OS
- Many target platforms
- Low-latency is a special use-case
- We need to provide some hints

Why should I care?

Useful in some scenarios

- Low latency applications
- Response times < 1ms
- Compute-intensive workloads
- Long-running jobs

Jitter

- *“slight irregular movement, variation, or unsteadiness, especially in an electrical signal or electronic device”*
- Variation in response time latency
- Long-tail in response time

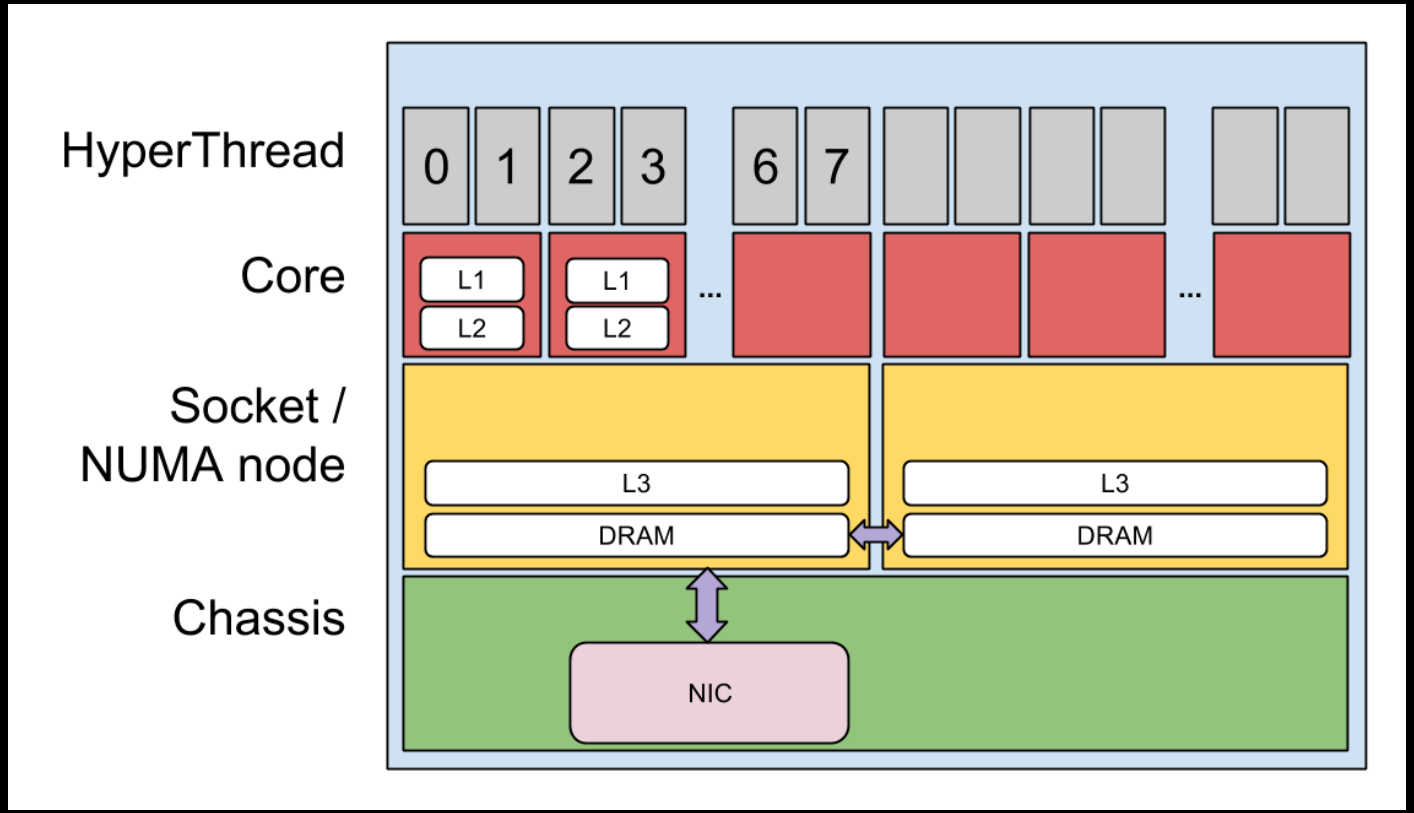
Dealing with it

- First take care of the low-hanging fruit
  - e.g. Garbage collection (gc-free / Zing)
  - e.g. Slow I/O
- Once response times are < 10ms the fun begins
- Make sure your code is running!

Measure first

- Need to validate changes are good
- End-to-end tests
- Using realistic load
- Change one thing and observe
- A refresher...

Modern hardware layout



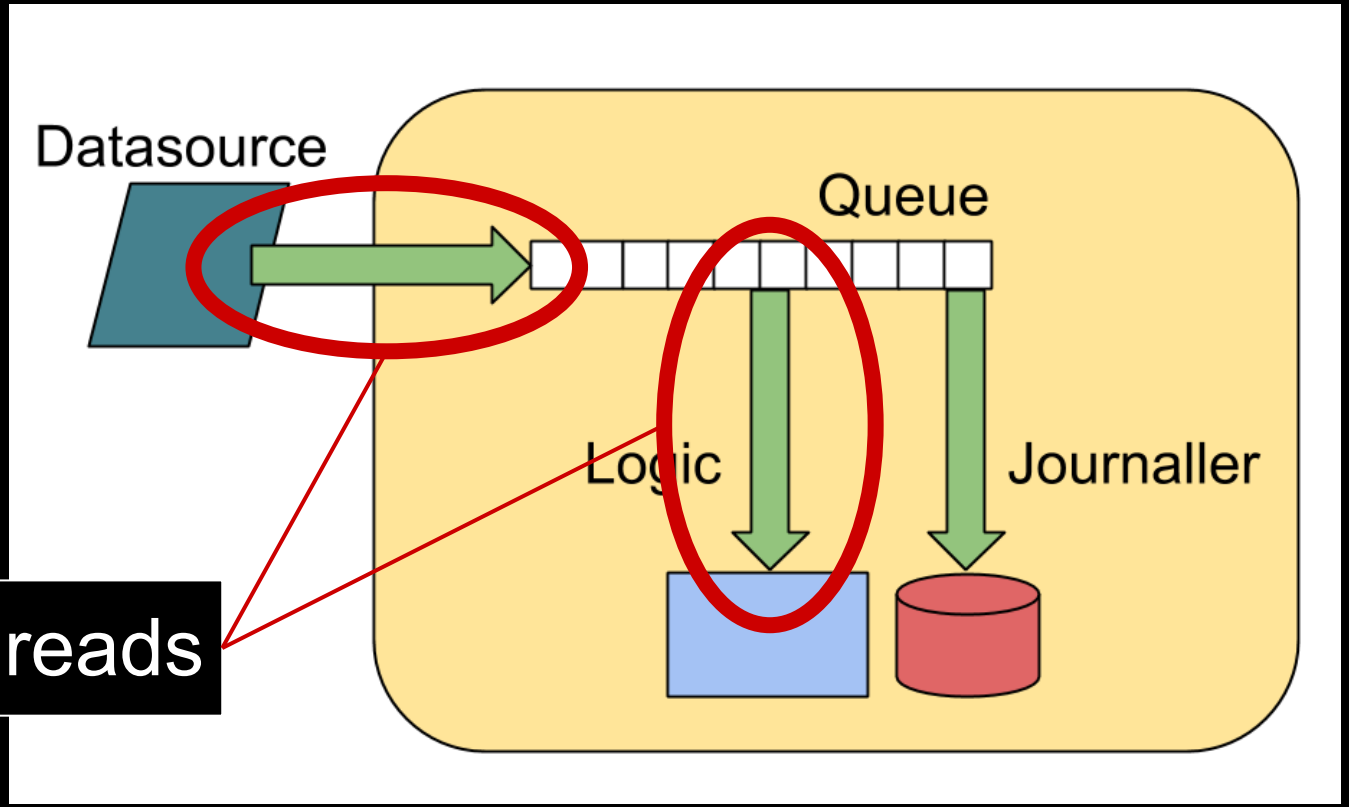
Multi-tasking

- num(tasks) > num(HyperThreads)
- OS must share out hardware resources
- Clever? Dumb? Fast? Slow?
- Fair...

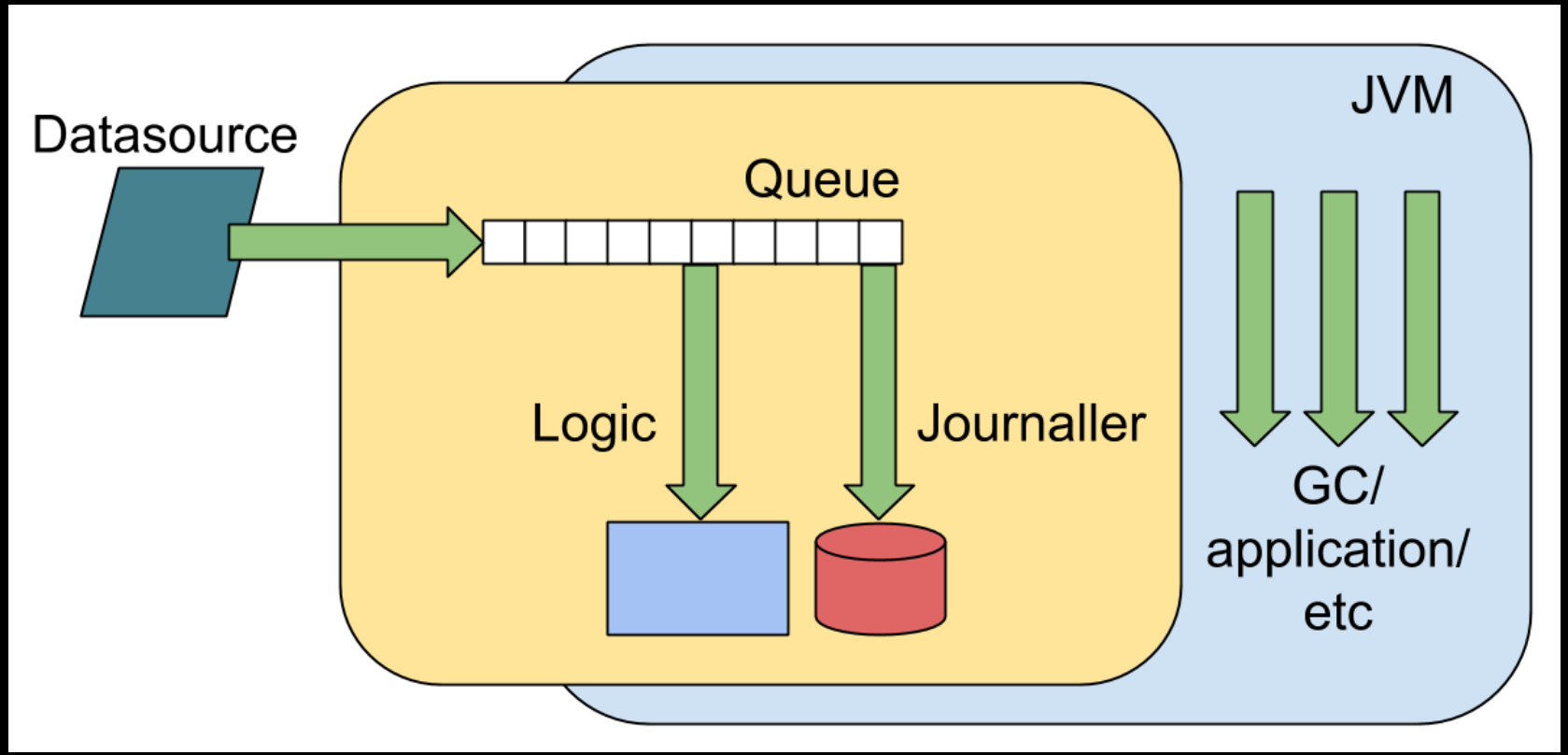
Linux CFS

- Completely Fair Scheduler
- Maintains a task ‘queue’ per HT
- Runs the task with the lowest runtime
- Updates task runtime after execution
- Higher priority implies longer execution time
- Tasks are load-balanced across HTs

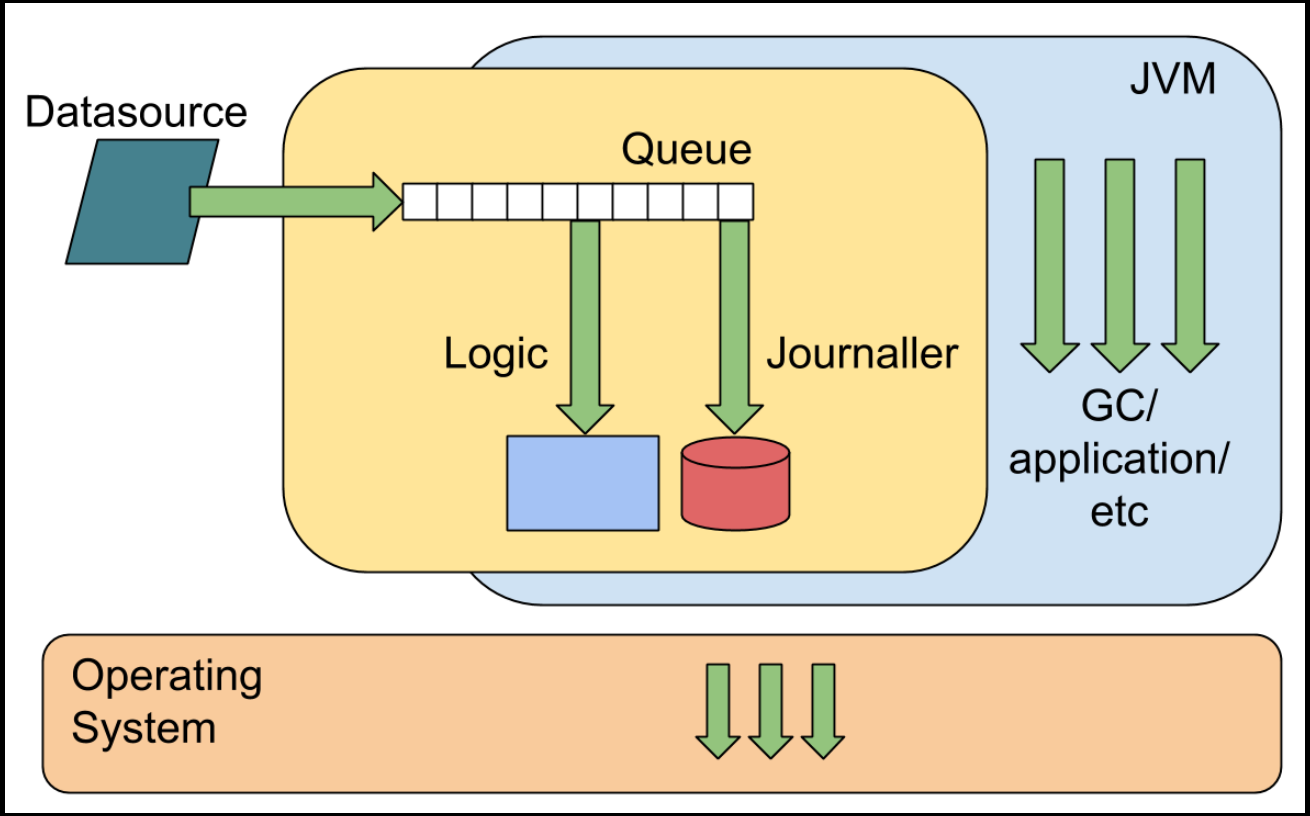
An example application ...



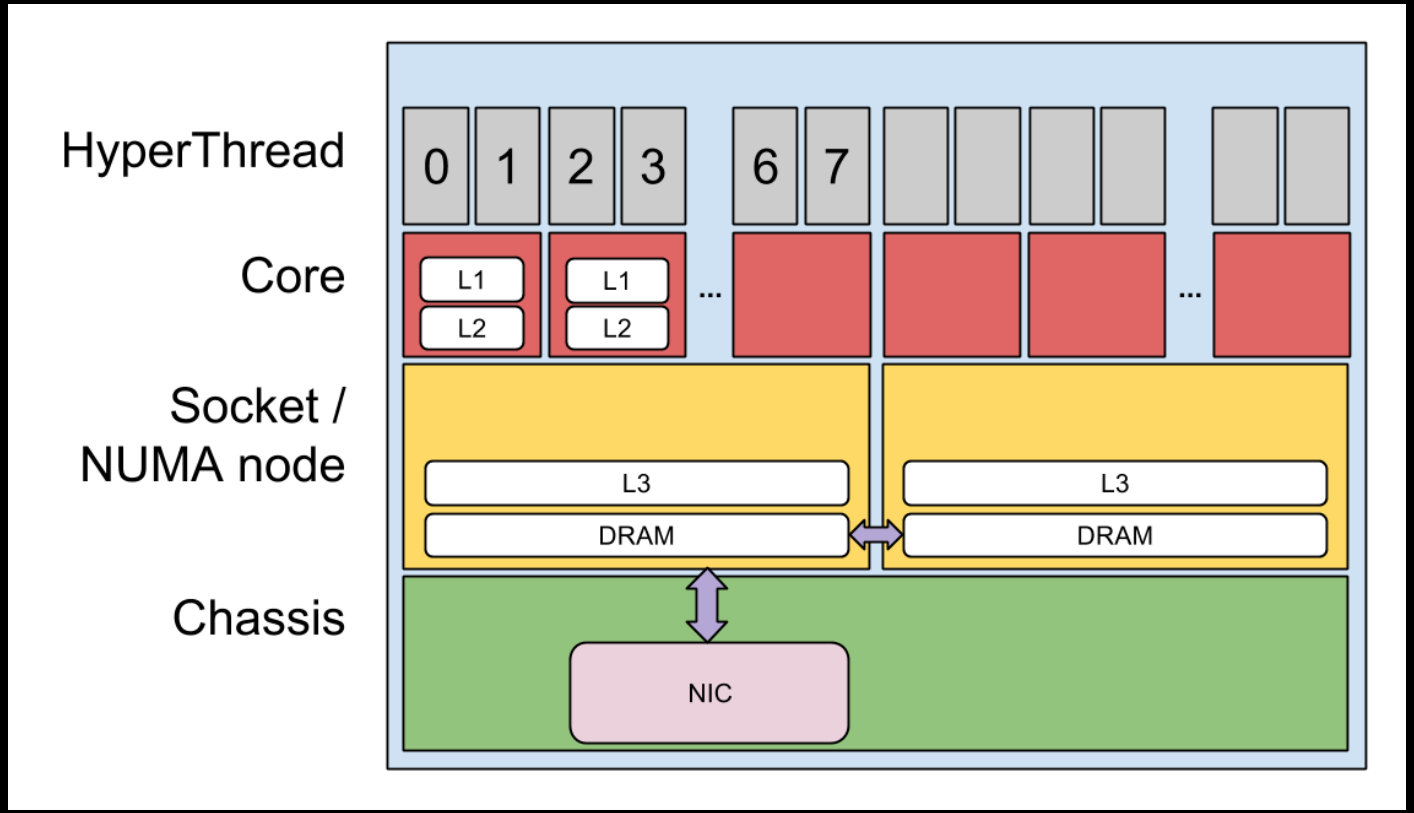
… running on a language runtime



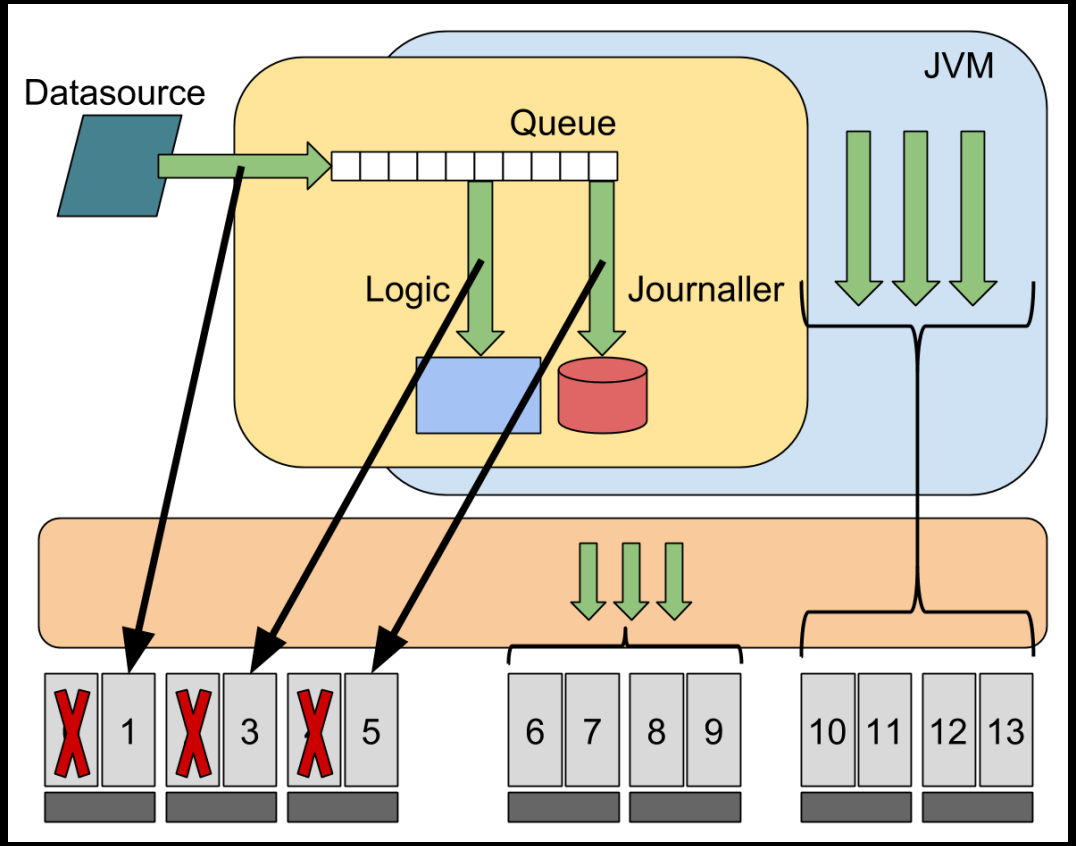
… running on an operating system



Optimise for locality - PCI/memory



Target deployment



How do I start?

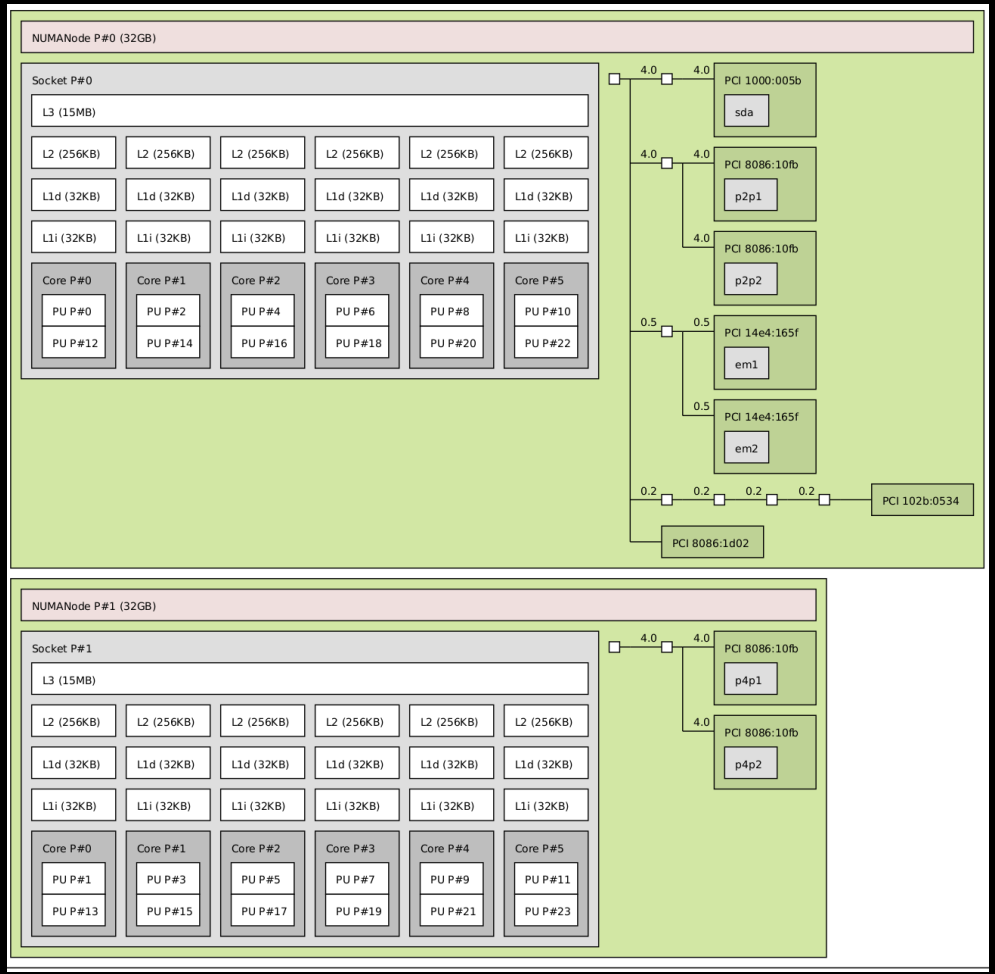
Start with the metal

- BIOS settings for maximum performance
- That’s a whole other talk...

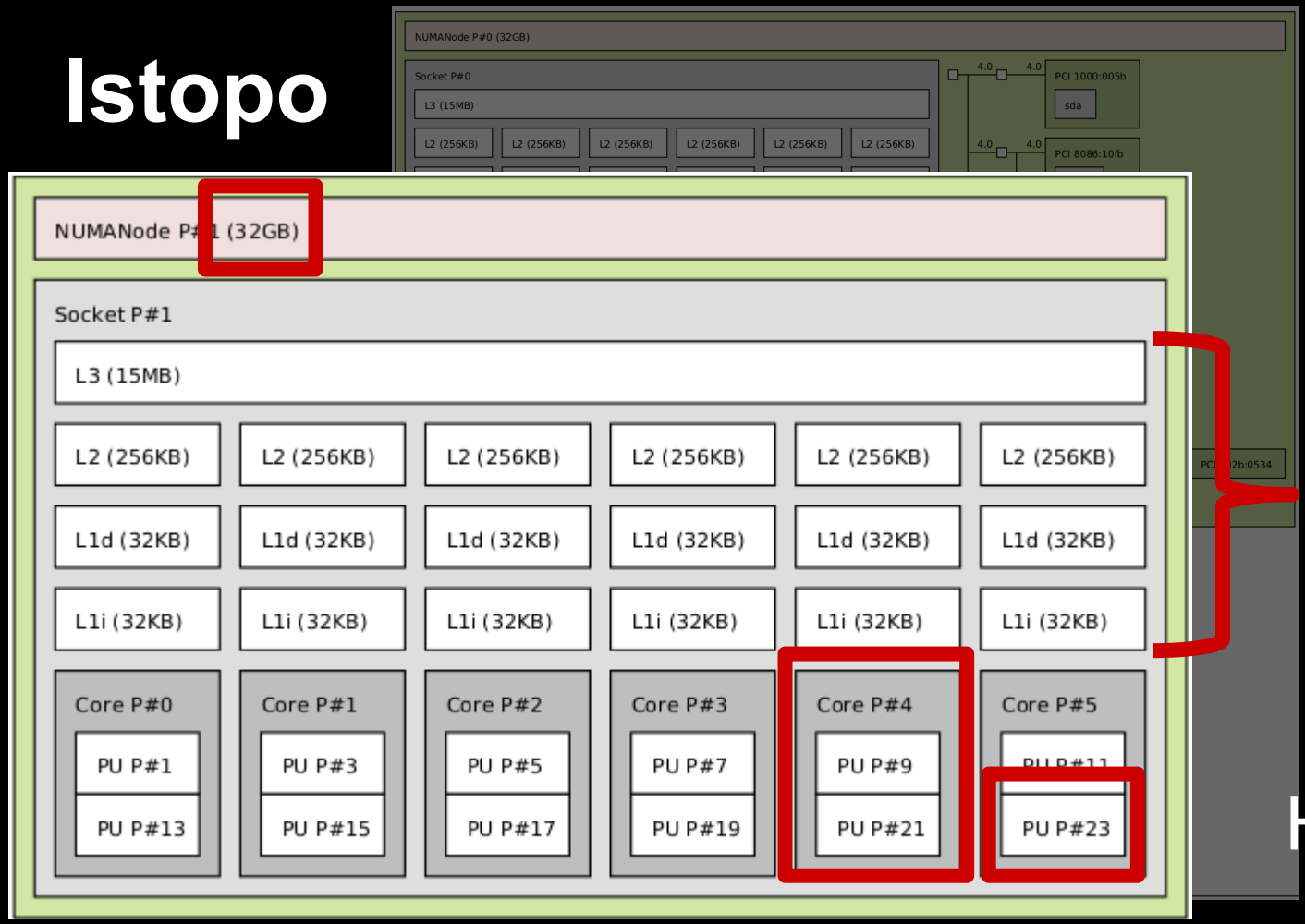
Discover what's available

- *lstopo* is a useful tool for looking at hardware
- Provided by the hwloc package
- Displays:
  - HyperThreads
  - Physical cores
  - NUMA nodes
  - PCI locality

lstopo


NUMANode P#0 (32GB)
Socket P#0
L3 (15MB)
L2 (256KB)
L1d (32KB)
L1i (32KB)
Core P#0
PU P#0
PU P#12
Core P#1
PU P#2
PU P#14
Core P#2
PU P#4
PU P#16
Core P#3
PU P#6
PU P#18
Core P#4
PU P#8
PU P#20
Core P#5
PU P#10
PU P#22
4.0
PCI 1000:005b
sda
PCI 8086:10fb
p2p1
p2p2
0.5
PCI 14e4:165f
em1
em2
0.2
PCI 102b:0534
PCI 8086:1d02
NUMANode P#1 (32GB)
Socket P#1
PU P#1
PU P#13
PU P#3
PU P#15
PU P#5
PU P#17
PU P#7
PU P#19
PU P#9
PU P#21
PU P#11
PU P#23
p4p1
p4p2

lstopo



NUMA-local RAM

Caches

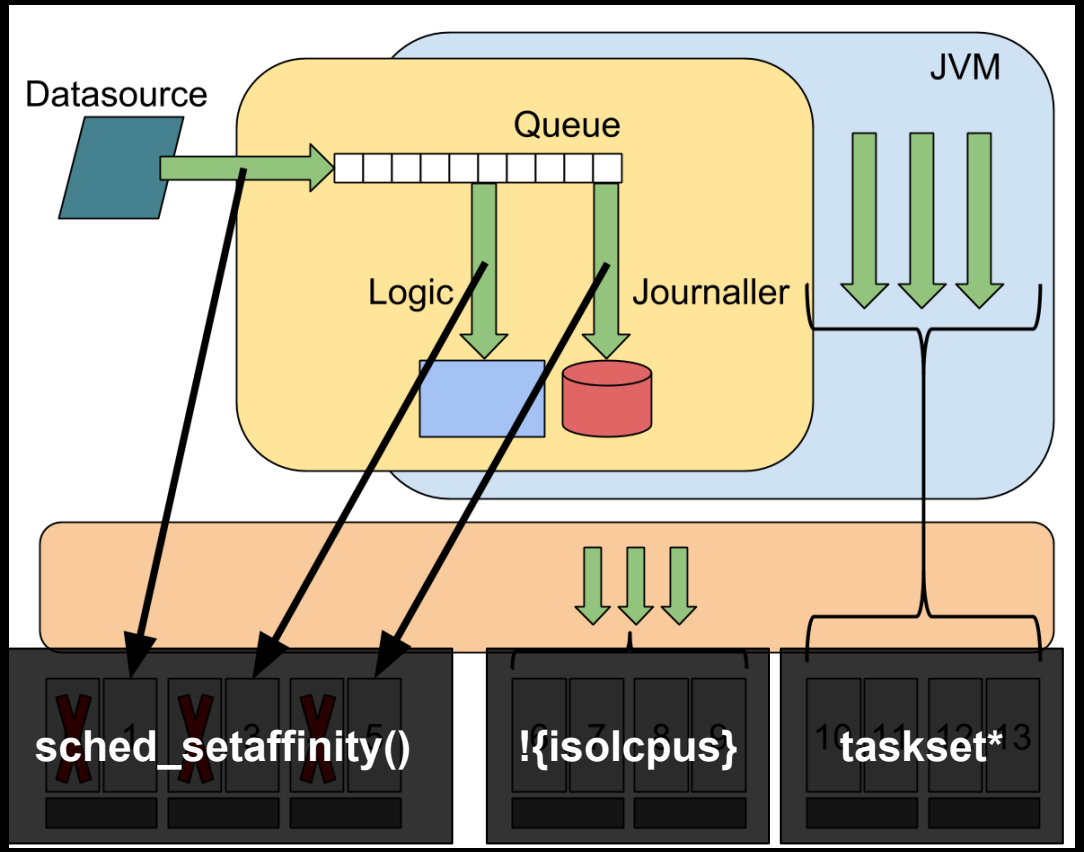
Core

HyperThread

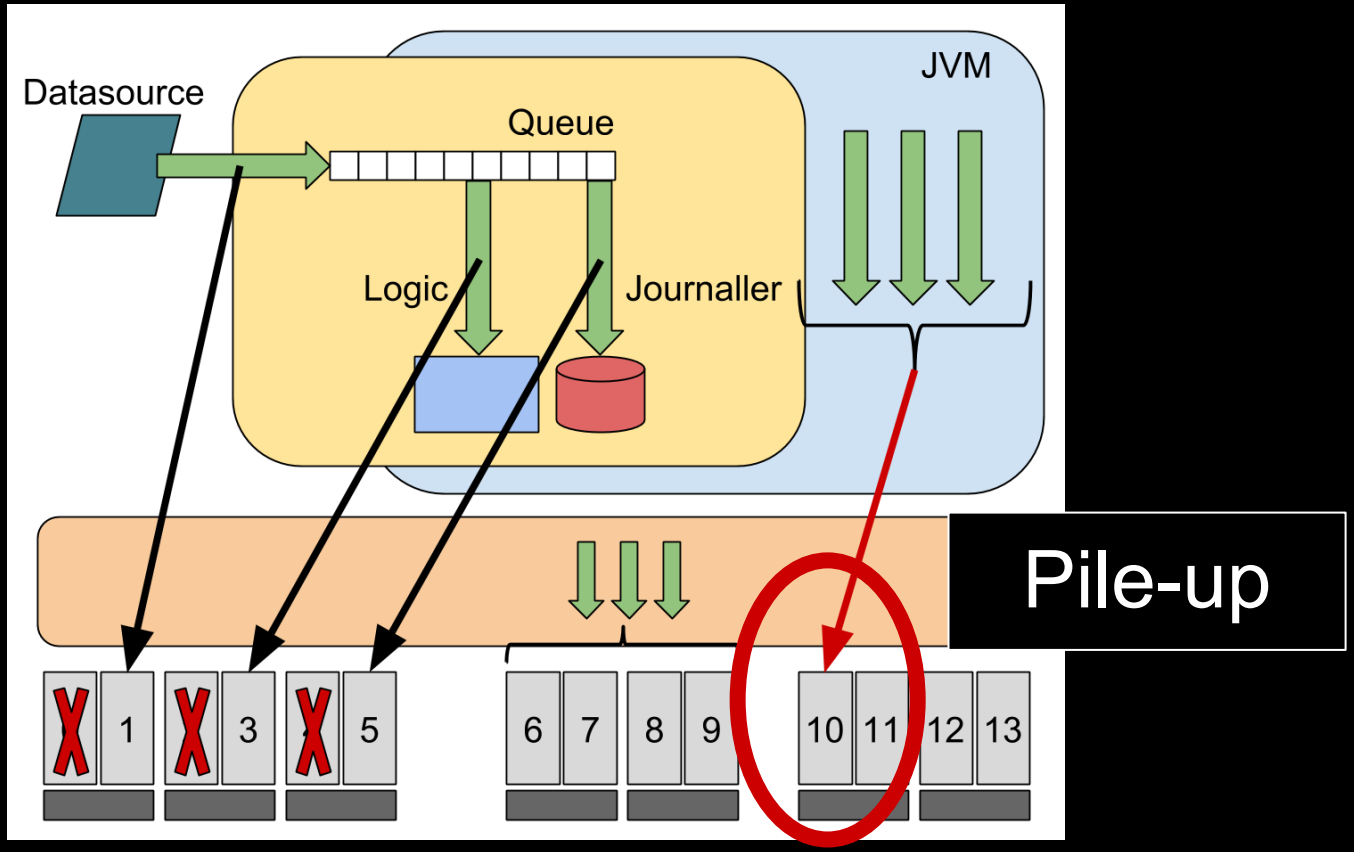
Reserve & use specific resource

- Use *isolcpus* to reserve cpu resource
- kernel boot parameter
- isolcpus=0-5,10-13
- Use *taskset* to pin your application to cpus:
- taskset -c 10-13 *java …*
- Set affinity of hot threads:
- *sched_setaffinity(...)*

Deploy the application



You have no load-balancer



A solution: cpusets

- Create hierarchical sets of reserved resource
- CPU, memory
- Userland tools: cset (SUSE)

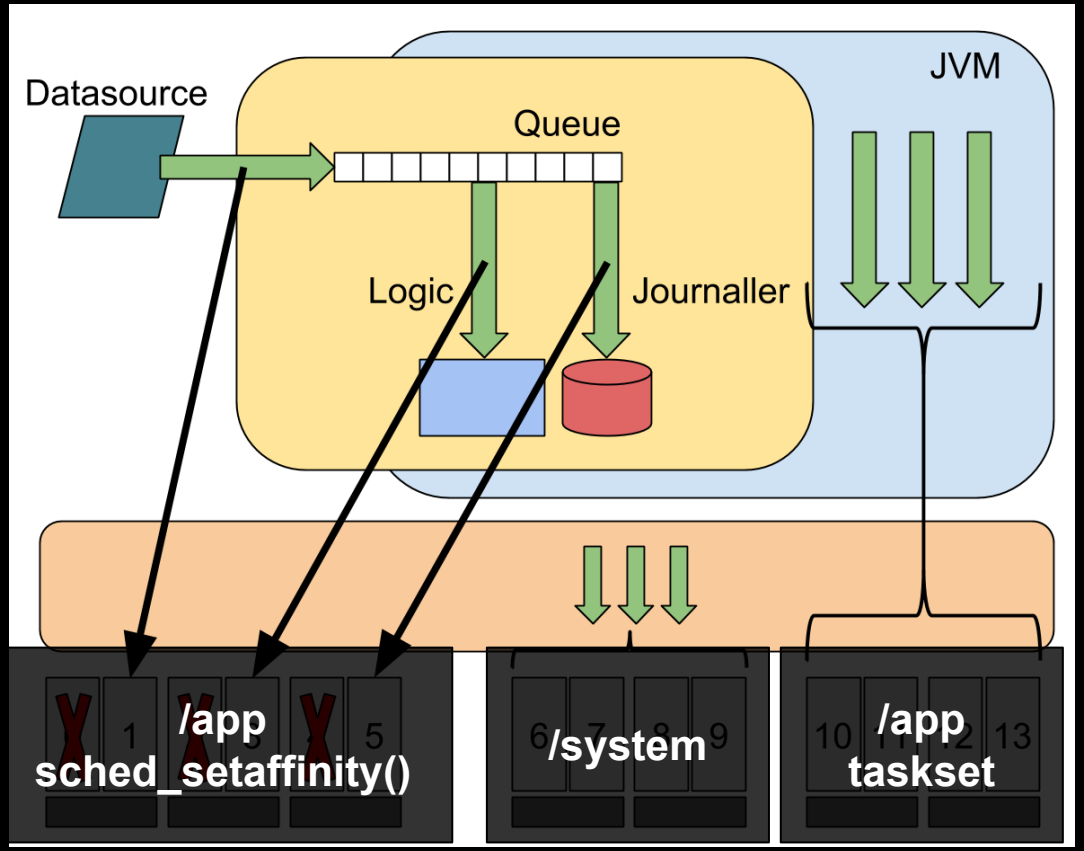
Isolate OS processes

- cset set --set=/system --cpu=6-9
  - *create a cpuset with cpus 6-9*
  - *create it at the path /system*
- cset proc --move --from-set=/ --to-set=/system
  - *move all processes from / to /system*
  - *-k => move unbound kernel threads*
  - *--threads => move child threads*
  - *--force => erm... force*

Run the application

- cset set --cpu=0-5,10-13 --set=/app
- cset proc --exec /app *taskset -cp 10-13 java …*
  - *start a process in the /app cpuset*
  - *run the program on cpus 10-13*
- sched_setaffinity() to pin the hot threads to cpus 1,3,5

Isolated threads



No more jitter?

perf_events

- Sampling tracer
- Static/dynamic trace points
- Very low overhead
- A good starting point for digging deeper
- *perf list* to view available trace points
- network, file-system, scheduler, etc

What’s happening CPU?

- perf record -e "sched:sched_switch" -C 3
  - Sample task switches on CPU 3
- perf report (best for multiple events)
- perf script (best for single events)

Rogue process

java

36049 [003] 3011858.780856: sched:sched_switch: java:
36049 [110] R ==> kworker/3:1:13991 [120]

kworker/3:1

13991 [003] 3011858.780861: sched:sched_switch:
kworker/3:1:13991 [120] S ==> java:36049 [110]

ftrace

- Function tracer
- Static/dynamic trace points
- Higher overhead
- But captures everything
- Can provide function graphs
- trace-cmd is the usable front-end

So what is that kernel thread doing?

- trace-cmd record -P <pid> -p function_graph
  - Trace functions called by process <pid>
- trace-cmd report
  - Display captured trace data

Some things can’t be deferred

```
kworker/3:1-13991 [003] 3013287.180771: funcgraph_entry:           |  process_one_work() {
kworker/3:1-13991 [003] 3013287.180772: funcgraph_entry:           |    cache_reap() {
kworker/3:1-13991 [003] 3013287.180772: funcgraph_entry: 0.137 us  |      mutex_trylock();
kworker/3:1-13991 [003] 3013287.180772: funcgraph_entry: 0.289 us  |      drain_array();
kworker/3:1-13991 [003] 3013287.180773: funcgraph_entry: 0.040 us  |      _cond_resched();
.........
.........
kworker/3:1-13991 [003] 3013287.180859: funcgraph_exit: +86.735 us |  }
```

Things to look out for

- cache_reap() - SLAB allocator
- vmstat_update() - kernel stats
- other workqueue events
  - perf record -e “workqueue:*” -C 3
- Interrupts - set affinity in /proc/irq
- Timer ticks - tickless mode
- CPU governor - set to performance
  - /sys/devices/system/cpu/cpuN/cpufreq/scaling_governor

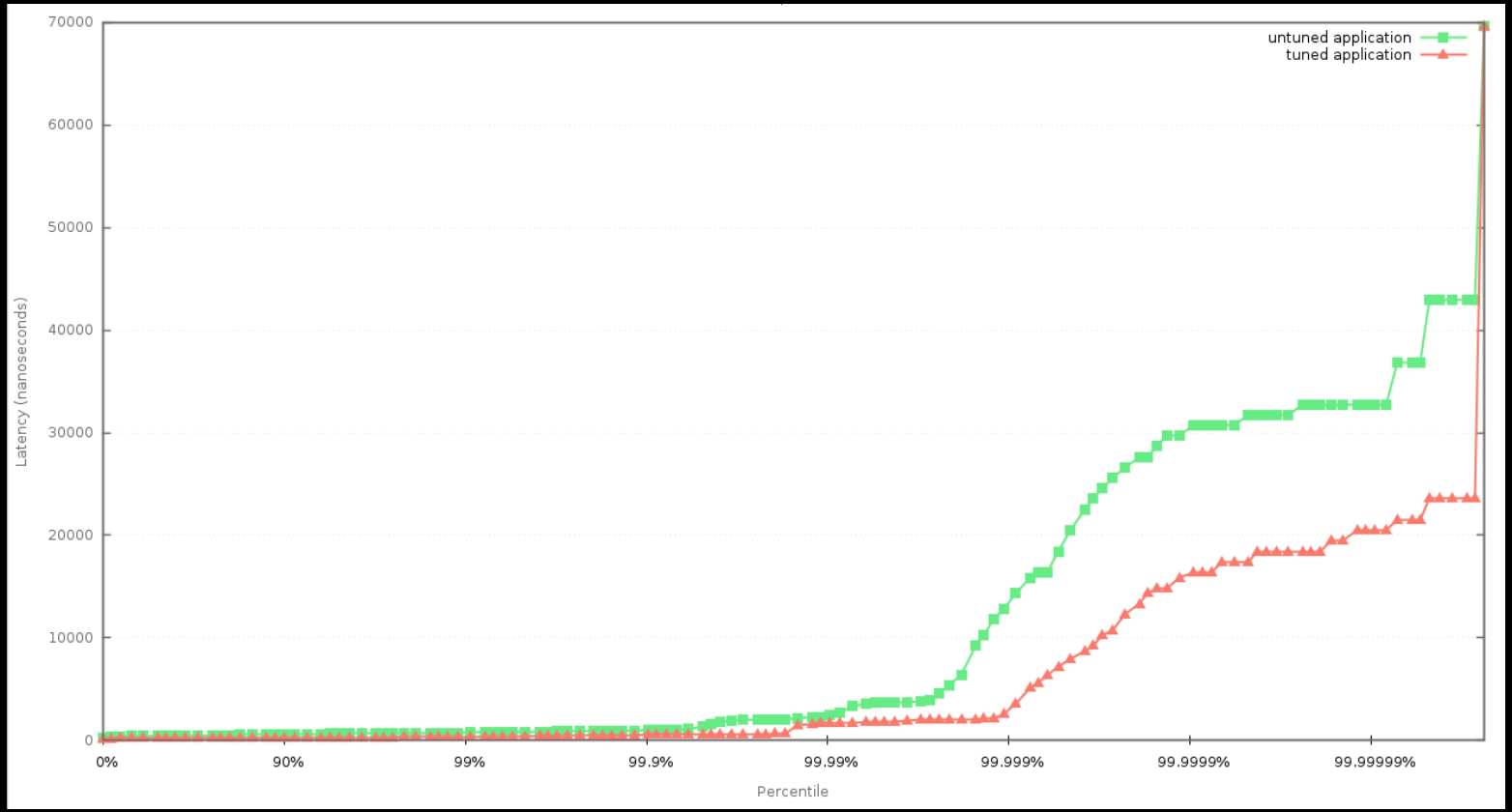
Some numbers

- Inter-thread latency is a good proxy
- 2 busy-spinning threads passing a message
- Time taken between producer & consumer
- Record times over several seconds
- Compare tuned/untuned

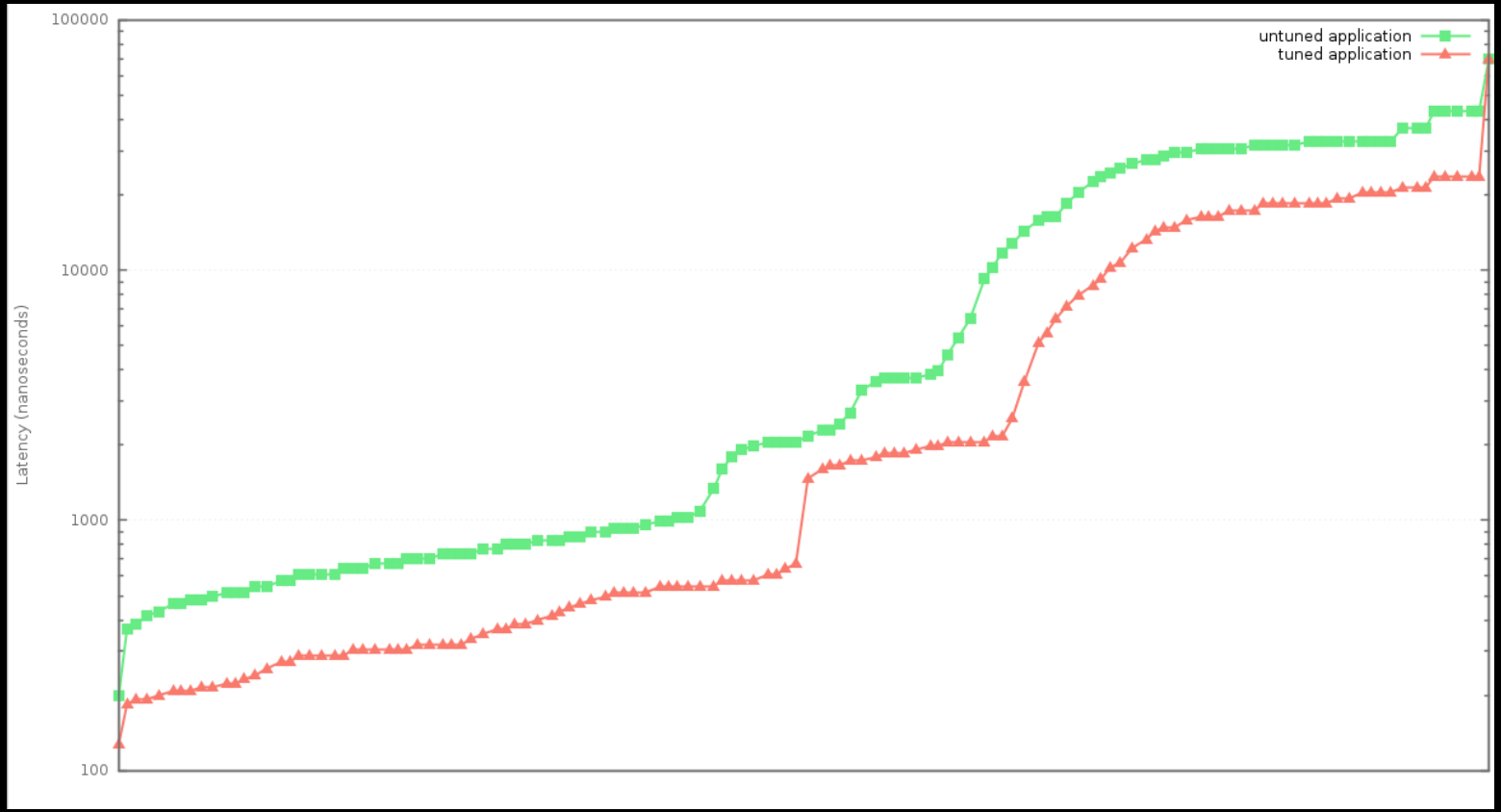
Results

== Latency (ns) ==	untuned	tuned
mean	466	216
min	200	128
50.00%	464	208
90.00%	608	288
99.00%	768	336
99.90%	992	544
99.99%	2432	1664
max	69632	69632

tuned vs untuned



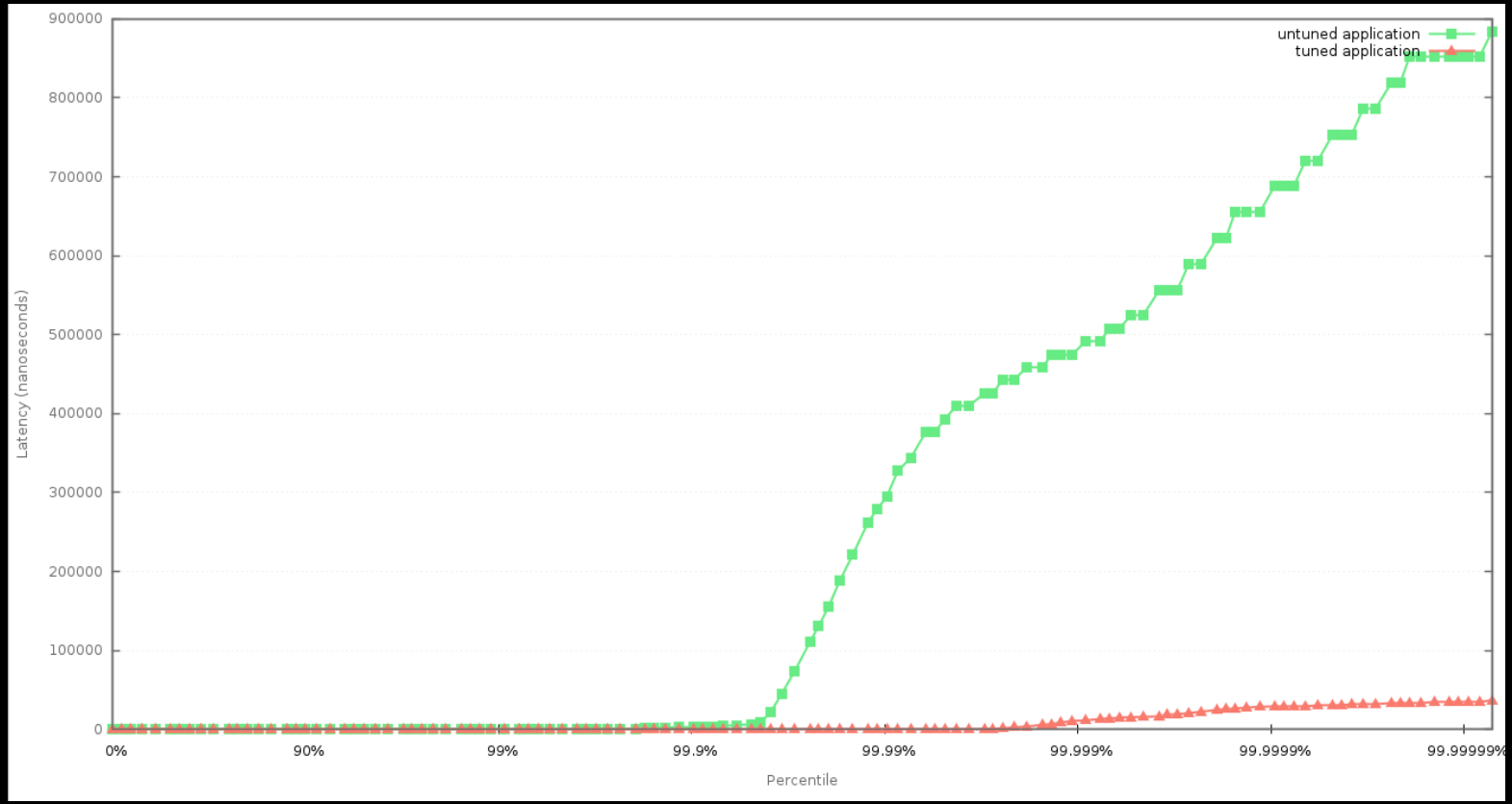
tuned vs untuned (log scale)



Results (loaded system)

== Latency (ns) ==	untuned	tuned
mean	545	332
min	144	216
50.00%	464	336
90.00%	544	352
99.00%	736	448
99.90%	2944	544
99.99%	294913	704
max	884739	36864

tuned vs untuned (loaded system)



Summary

- Select threads that need access to CPU
- Isolate CPUs from the OS
- Pin important threads to isolated CPUs
- Don’t forget interrupts
- There will be more things…
- Always test assumptions!

Thank you

- lmax.com/blog/staff-blogs
- github.com/epickrram/perf-workshop
- @epickrram