All Your IOPS Are Belong To Us

A Pinteresting Case Study in Performance Optimization

Ernie Souhrada
Database Engineer / Bit Wrangler, Pinterest
16 April 2015

Agenda

- Introductions
- A Giant Pile of Bits
- Where Have All the IOPS Gone?!
- Life is Short, Eat Dessert First.
- A Case of Kernel Panic?
- Show Me The DATA!
- Let Them Eat IRQs
- I Don’t Always Test, But When I Do, I Test In Production.
- Droppin’ Mo’ Science
- Questions? Answers!

Introductions

About You?

About Me?

- Joined Pinterest in 2015
  - Before that: Percona, Sun, and lots of assorted companies you've never heard of.
- Using MySQL since 3.23 (and mSQL before that).
- Technological Renaissance Man (sorry, I know it sounds pompous).

A Giant Pile of Bits

MySQL at Pinterest, early 2015

- Hosted inside AWS.
- Ubuntu 12.04, Linux kernel 3.2.
- Ephemeral storage for MySQL, EBS/S3 for backups.
- Hundreds of boxes, mix of Percona Server 5.5 and 5.6.

Primary MySQL footprint

- 2 types of data; N clusters of type 1, N/8 clusters of type 2.
- Several hundred TB of data.
- Several thousand individual databases.
- i2.4xlarge instance type[1]
  - 4 x 800GB SSD RAID-0, 122GB RAM, 16 cores
- We don’t fix the broken ones, we shoot them.

[1] - http://aws.amazon.com/ec2/instance-types/

A Giant Pile of Bits

QPS Volume (ca. January 2015)

- Masters only (slaves for DR/HA)
- Type 1: R/W ratio: 10:1
- Type 2: R/W ratio: 4:1
- Individual servers ~ 2000 QPS at peak
- Lots of caching

Performance of T1 boxes (January 2015):

- Average of all masters:
  - p90: < 1ms
  - p99: 5ms
  - p99.9: 15 – 45ms
- Worst-performing master:
  - p90: 1 – 3ms
  - p99: 10 – 100ms
  - p99.9: 50 – 1100ms

A Giant Pile of Bits

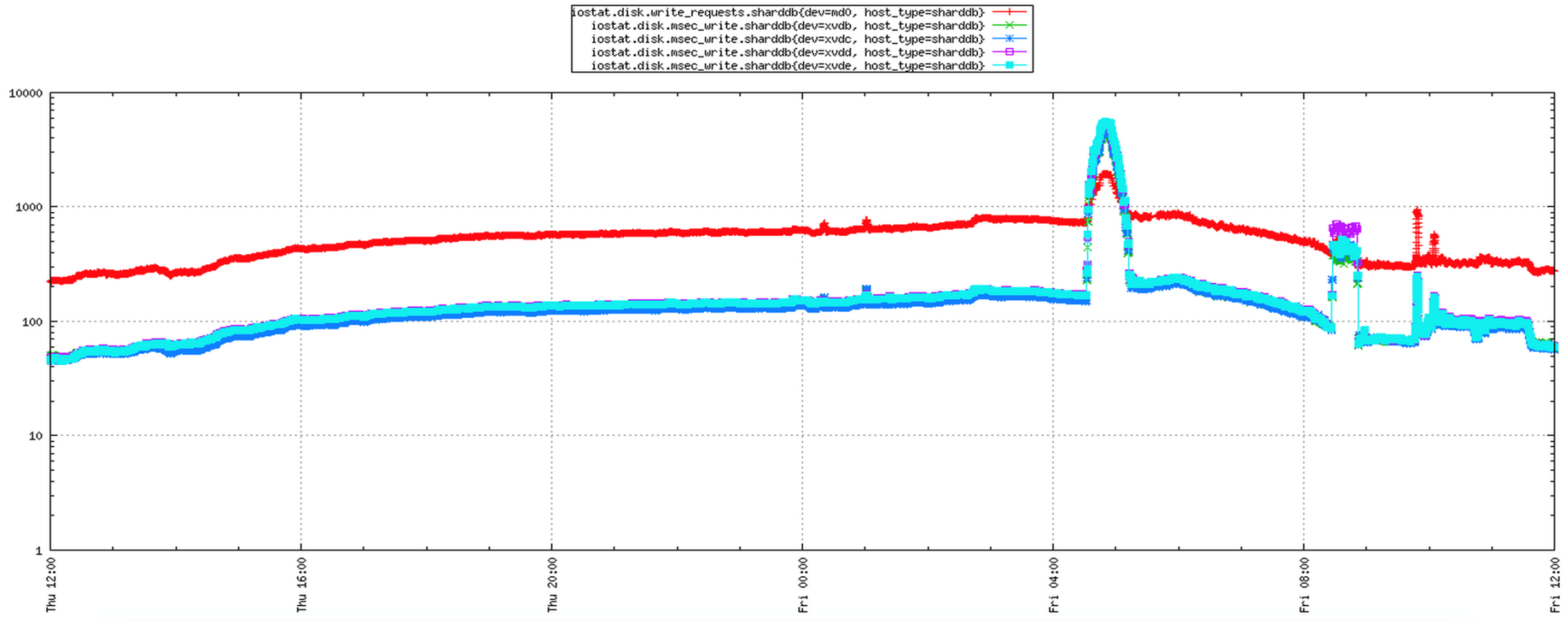
So What's The Problem?

Late 2014 / Early 2015

- Questions about hardware performance.
- Nightly ETL causing nontrivial replication lag.
- Problems for failover.
  - Can't promote lagged slaves.
- Problems for data team.
  - Jobs aren't running fast enough.
  - Internal consumers are blocked.
- Problems for data integrity.
  - Periodic checksum jobs disabled to provide more resources to ETL
- Cost / resource waste.
  - Using much bigger boxes than necessary because of apparently-insufficient IO capacity.

Where Have All The IOPS Gone?!

Sung to the tune of "Where Have All The Cowboys Gone?"



Where Have All The IOPS Gone?!

When we exceed 800 IOPS, performance goes to hell.

Side note: IOPS are sort of BS.

Think about what that means relative to the hardware...

- A RAID-0 of 4 SSDs can only do 800(3200) IOPS ?
- Only 200(800?) IOPS per SSD ??
- The SSD in my old laptop (Samsung 840 Pro) can do 40x what these drives are putting out ???

Where Have All The IOPS Gone?!

No, really, where are they?

This is a family-friendly conference, but…

WTF?!!

Where Have All The IOPS Gone?!

The docs claim they're not missing…

For reference[1]:

Instance Size	Read IOPS	First Write IOPS
i2.xlarge	35,000	35,000
i2.2xlarge	75,000	75,000
i2.4xlarge	175,000	155,000
i2.8xlarge	365,000	315,000

These are measured in 4K IOPS.

Even accounting for InnoDB page size the math doesn't work.

[1] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/i2-instances.html#i2-instances-diskperf

Where Have All The IOPS Gone?!

Is this just a case of RTFM?

According to the Amazon docs, Linux 3.8 or newer is required to get those numbers.

We were running 3.2 (Ubuntu 12.04 standard)

So that's it… Right?
Just upgrade to 3.8 and we're done?

Where Have All The IOPS Gone?!

Nothing is ever that easy.

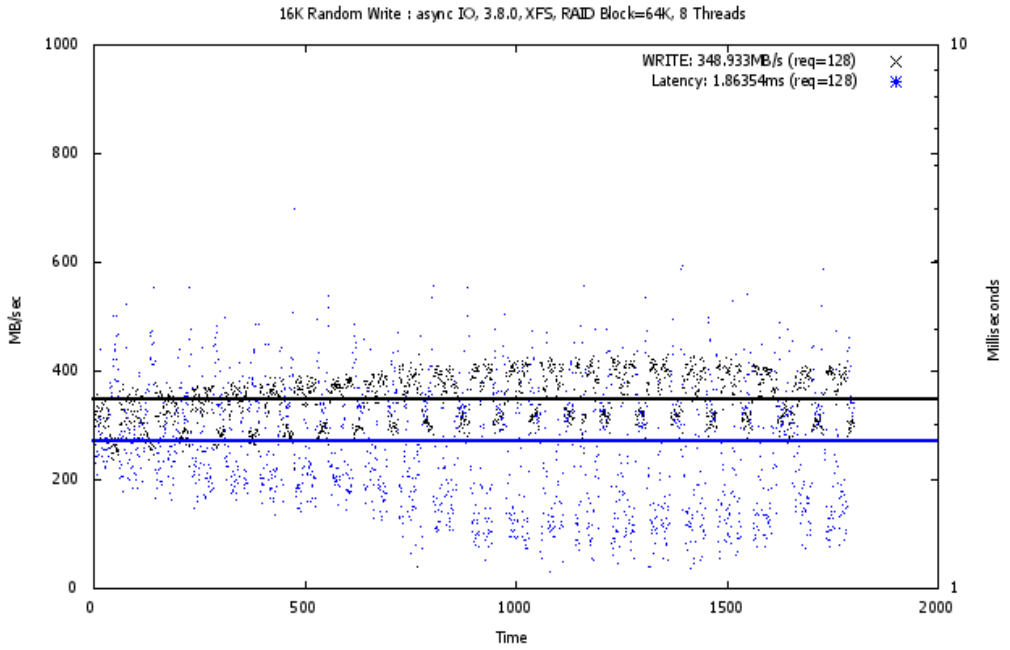
Of course not.

That was just the beginning.

Where Have All The IOPS Gone?!

3.8 doesn't have them.

- Kernel 3.8 is significantly better than 3.2 for single-drive installations.
- But we don't run that way. We use MDRAID.
- And the numbers still don't add up to what we would expect from a multiple-device RAID0.



Life Is Short, Eat Dessert First.

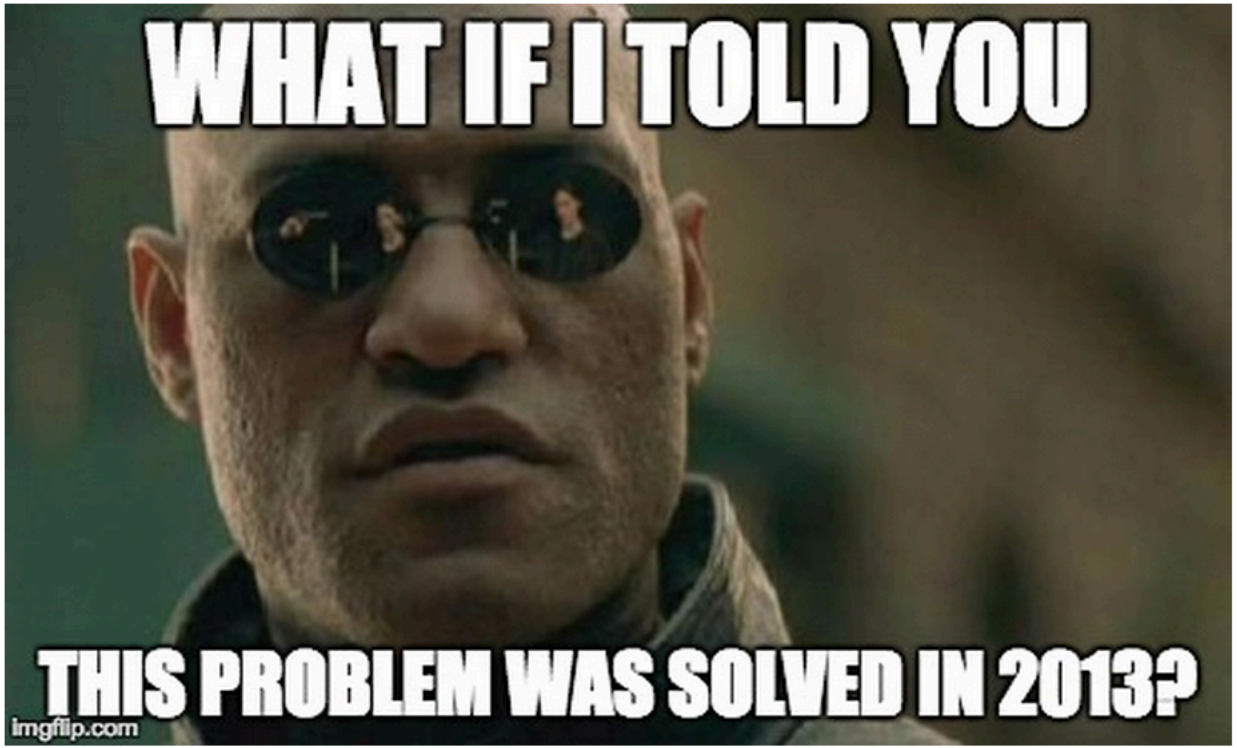
It's late in the day, you might already be checked out....

If you take nothing else away from this talk:

1. If you use Linux software RAID (MySQL or not) with anything faster than spinning rust, you should be running Linux kernel 3.14 or higher.
2. If you also use AWS or some other Xen-based cloud provider, you should be running with irqbalance 1.0.6+ and RPS (receive packet steering) enabled.

A Case of Kernel Panic

The Matrix has you, Neo.



A Case of Kernel Panic

A Brief History of the Linux Block Layer

Request Queue Interface

- Generally FIFO, but the block layer can reorder, coalesce, or apply bandwidth / fairness policies.
    1. Protected by a single lock – any action on the queue is synchronized.
    2. All hardware interrupts typically go to CPU core 0.
    3. Any remote memory accesses (e.g., if an IO completes on a different core other than the one it started on) lead to lots of cache invalidations because of #1.
- This is Not Good™ for multi-core boxes with fast storage.

“Make Request” Interface

- A way to intercept requests before they hit the queue and do “something else” with them – reroute them, preprocess them, etc.
- Designed for “stacked” block drivers like MDRAID, **but** never intended for high-performance devices / device drivers. (SSDs would qualify…)

A Case of Kernel Panic

There has to be a better way....

Multiqueue: blk-mq to the rescue

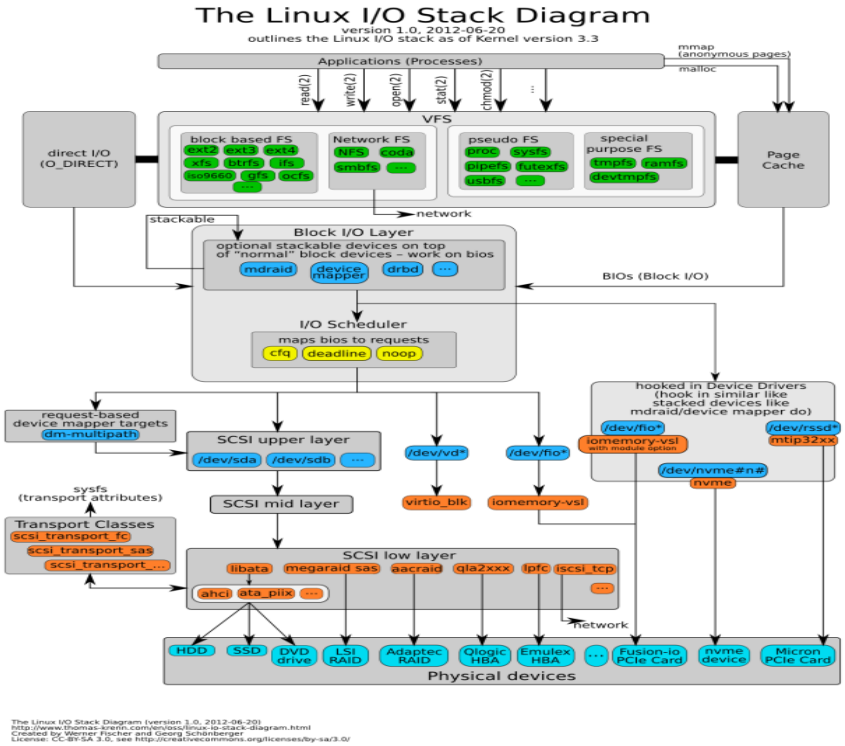
- 2013 paper[1] written by researchers from FusionIO and the IT University of Copenhagen.
- Each CPU/node has a “submission queue” for its own IO requests.
- No more (or greatly reduced) contention.
- One or more hardware dispatch queues handle buffering IO for the driver.
- Sort of analogous to splitting the InnoDB buffer pool.

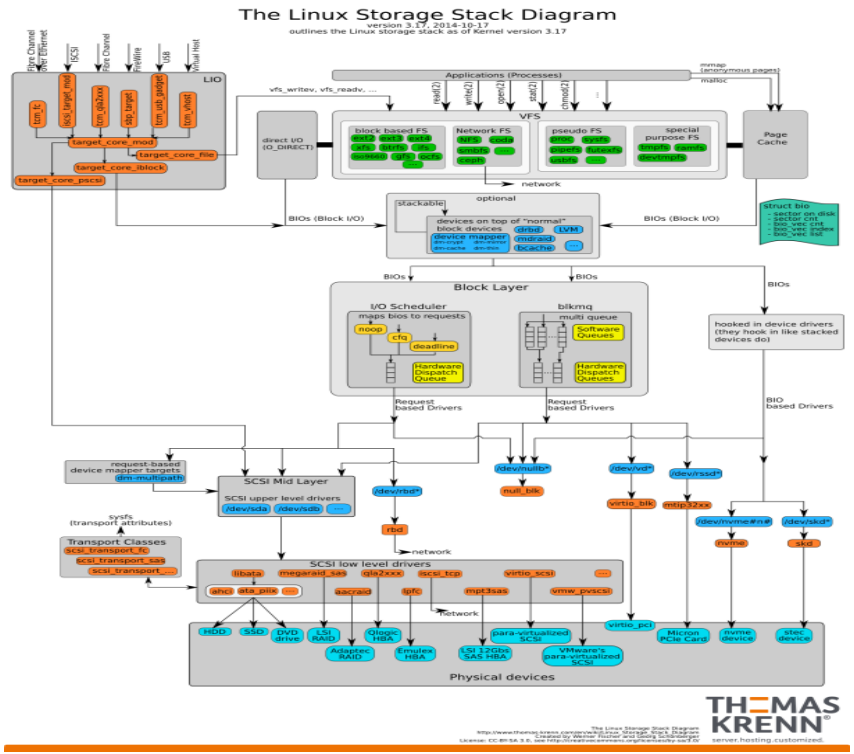
- Appeared first in **Linux 3.13** and has been subsequently refined.
- Linux 3.13.0 was released on 19 January 2014.

[1] http://kernel.dk/blk-mq.pdf

A Case of Kernel Panic

The Linux Storage Stack – That was then, this is now.





All Your IOPS Are Belong To Us: Ernie Souhrada, Database Engineer @ Pinterest - Percona Live MySQL Conference & Expo 2015 18

Credits: https://www.thomas-krenn.com/en/wiki/Linux_I/O_Stack_Diagram

A Case of Kernel Panic

Keep calm and benchmark on!

A new world of questions and possibilities:

- 3.8 vs. 3.13 vs. 3.X performance?
- Adjustments to my.cnf?
- Are there other kernel or OS-level tweaks we should be looking at?
- Ladies and gentlemen, start your sysbench!

Show Me The DATA!

With apologies to Tom Cruise and Cuba Gooding, Jr.

Sysbench

- Everybody knows it, everybody loves it.
- Initially, kernel 3.8 vs. 3.13 vs. 3.18.
- Investigation of 3.8 abandoned fairly early on.

Test Setup

- Multiple i2.2xlarge instances (2 x 800GB SSD RAID0).
- FileIO testing with XFS and EXT4.
- Varying RAID block sizes (4K, 16K, 64K, 256K).
- Different journaling options for EXT4 (ordered, journal).
- XFS mount options: noatime, nobarrier, discard, inode64, logbsize=256k.

Show Me The DATA!

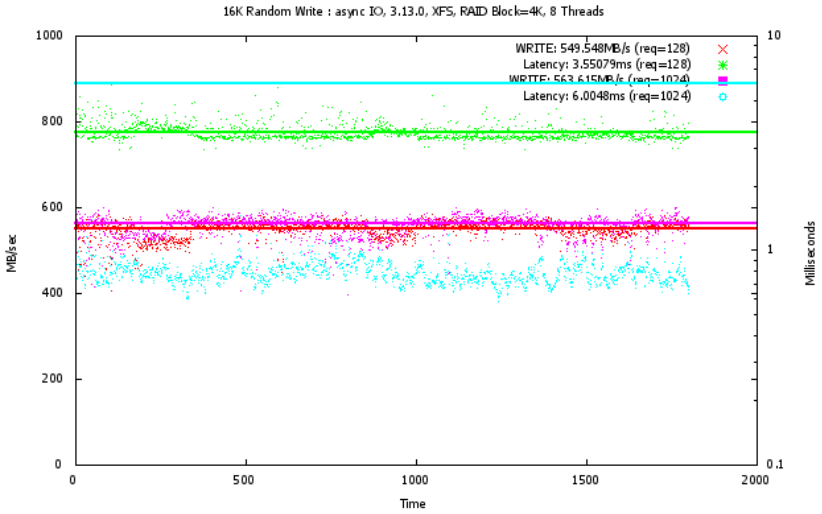
A little bash goes a long way.

- Test script ->
- Each test ran for 1 hour.
- Data collected at 1s intervals.
- First 600 seconds discarded to account for cache warmup.
- Data processed through a combination of Perl, Awk, and GNUPlot.
- A total of 57 different combinations were tested.

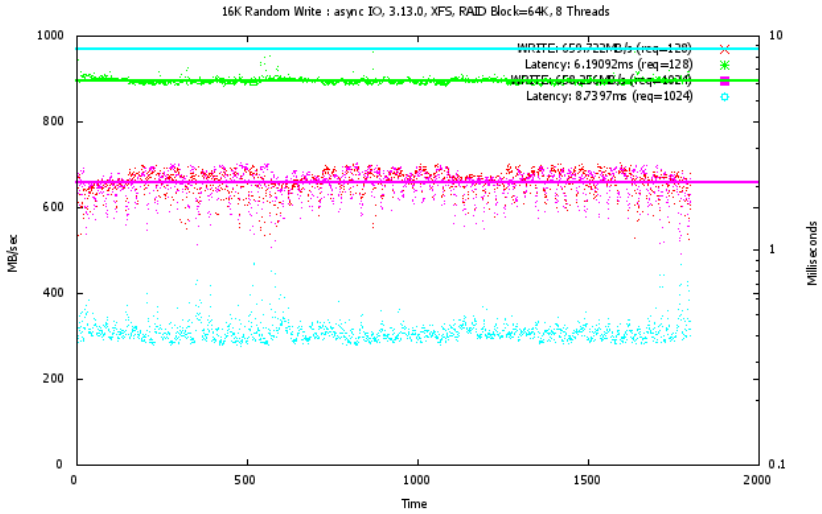
```
#!/bin/bash

cd /raid0/sysbench
outputdir=~esouhrada/sysbench
mkdir -p $outputdir

for threads in 8 16;
do
  for mode in async sync ;
  do
    for test in rndrw rndwr rndrd;
    do
      echo 3 > /proc/sys/vm/drop_caches
      echo "`date` Starting ${mode} ${test} test with ${threads} threads..."
      sysbench --test=fileio --file-total-size=1T --file-test-mode=${test} \
        --num-threads=${threads} --max-time=3600 --max-requests=0 \
        --rand-init=on --file-fsync-freq=0 --report-interval=1 \
        --rand-type=pareto --file-extra-flags=direct \
        --file-block-size=16384 --file-io-mode=${mode} --percentile=99 \
        run | tee ${outputdir}/${test}.${mode}.${threads}.log
      echo 'done.'
    done
  done
done
```

Show Me The DATA!

Kernel 3.13.0, XFS filesystem, sysbench async random write



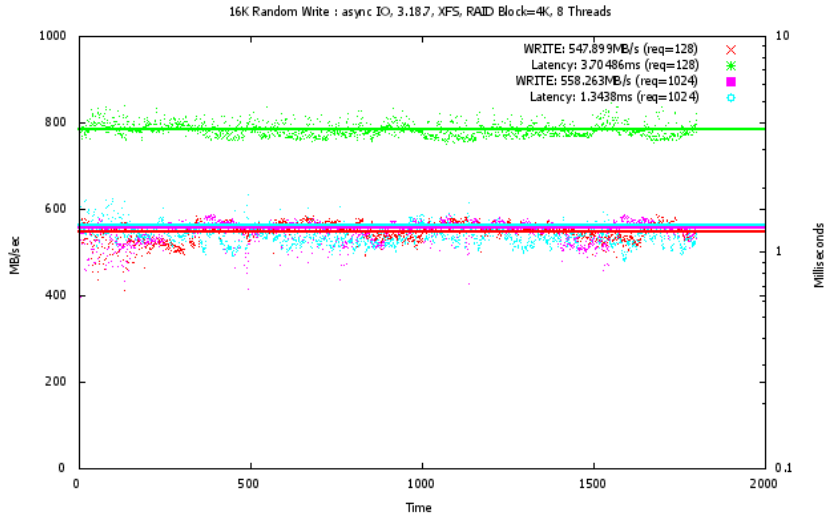
4K RAID block, XFS, kernel 3.13
Write throughput 550MB/sec
99th-percentile latency: 3.5ms



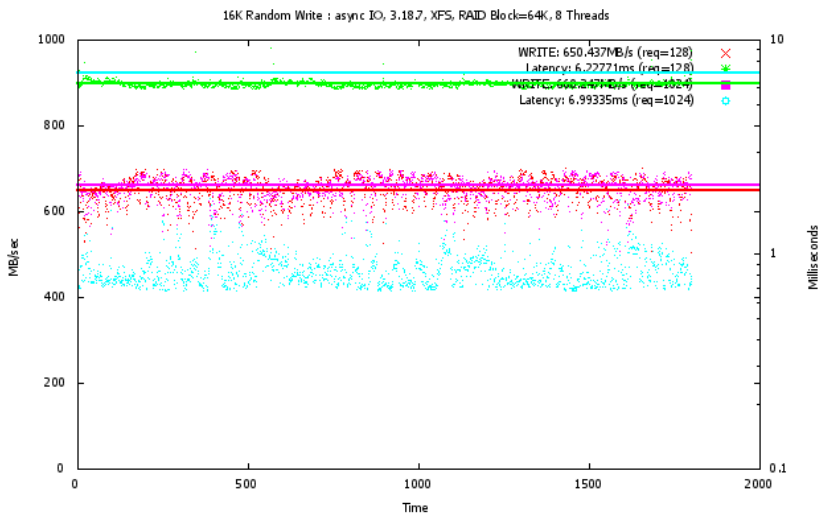
64K RAID block, XFS, kernel 3.13
Write throughput 650MB/sec
99th-percentile latency: 6.2ms

Show Me The DATA!

Kernel 3.18.7, XFS filesystem, sysbench async random write



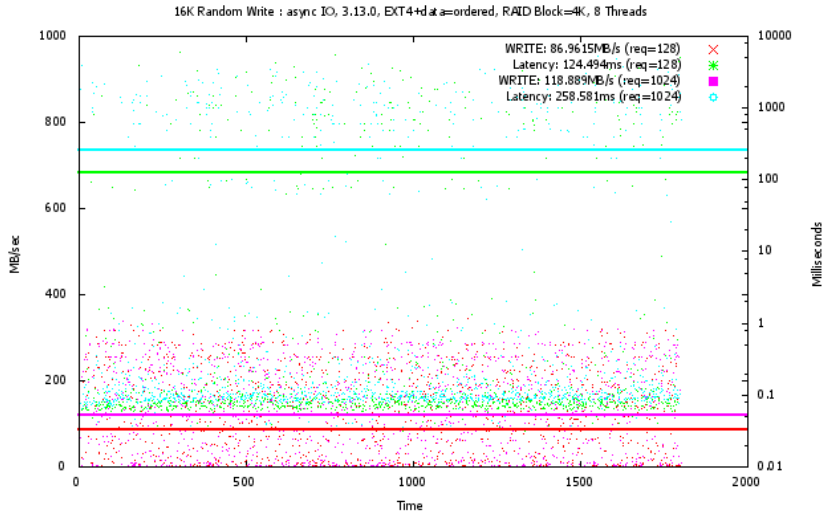
4K RAID block, XFS, kernel 3.18
Write throughput 550MB/sec
99th-percentile latency: 3.7ms



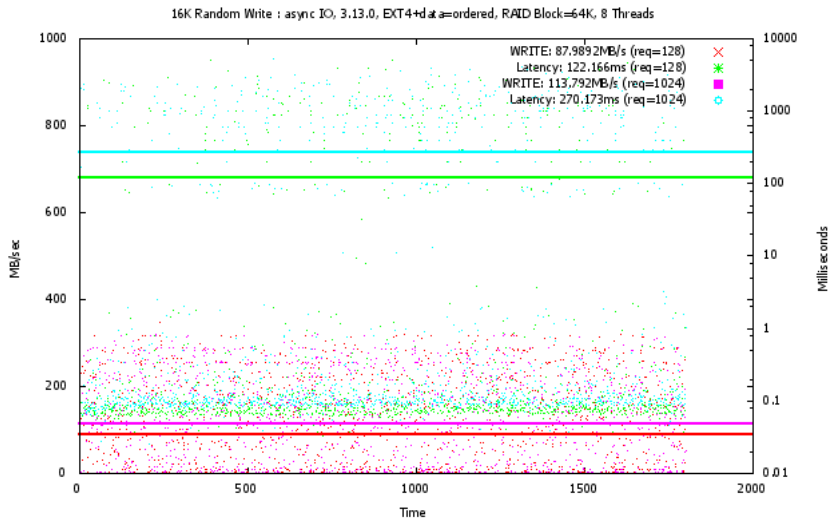
64K RAID block, XFS, kernel 3.18
Write throughput 650MB/sec
99th-percentile latency: 6.2ms

Show Me The DATA!

Kernel 3.13.0, EXT4 filesystem + data=ordered, sysbench async random write



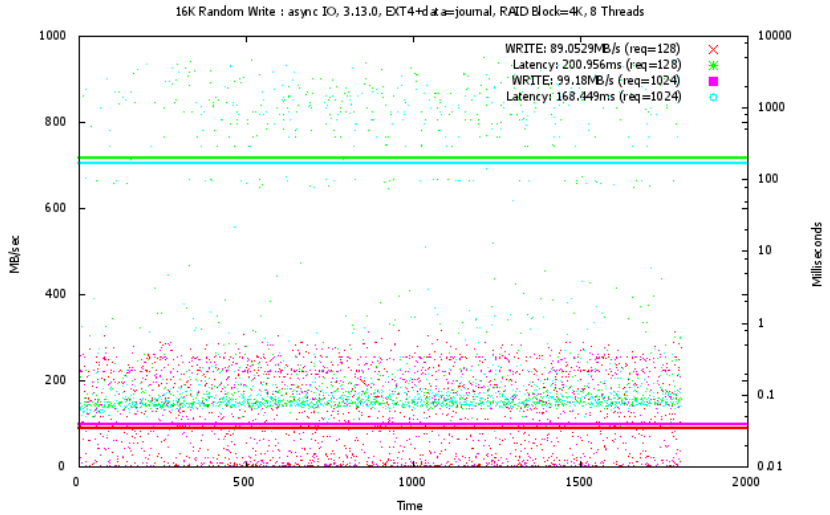
4K RAID block, EXT4, kernel 3.13
Write throughput **87MB**/sec
99th-percentile latency: 124ms



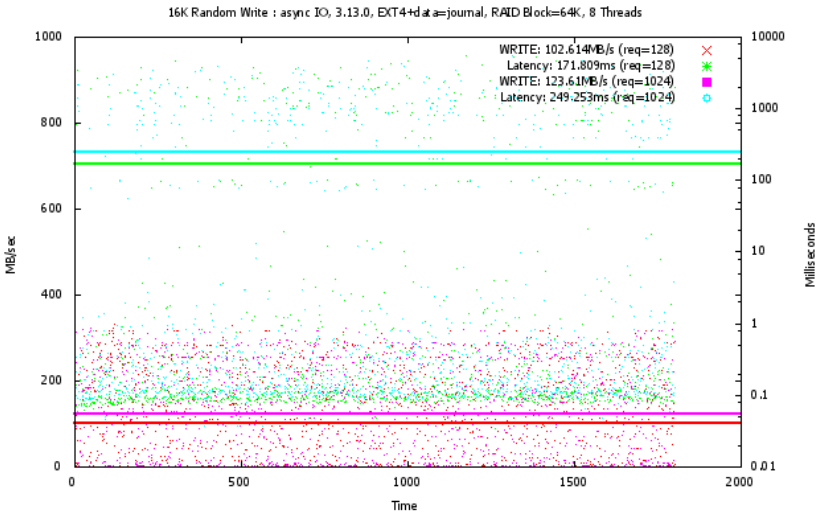
64K RAID block, EXT4, kernel 3.13
Write throughput **88MB**/sec
99th-percentile latency: 122ms

Show Me The DATA!

Kernel 3.13.0, EXT4 filesystem + data=journal, sysbench async random write



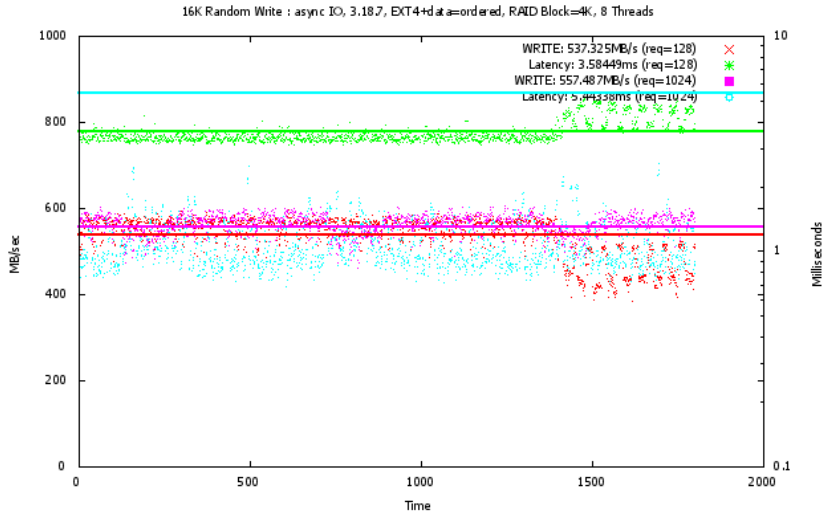
4K RAID block, EXT4, kernel 3.13
Write throughput **89MB**/sec
99th-percentile latency: 201ms



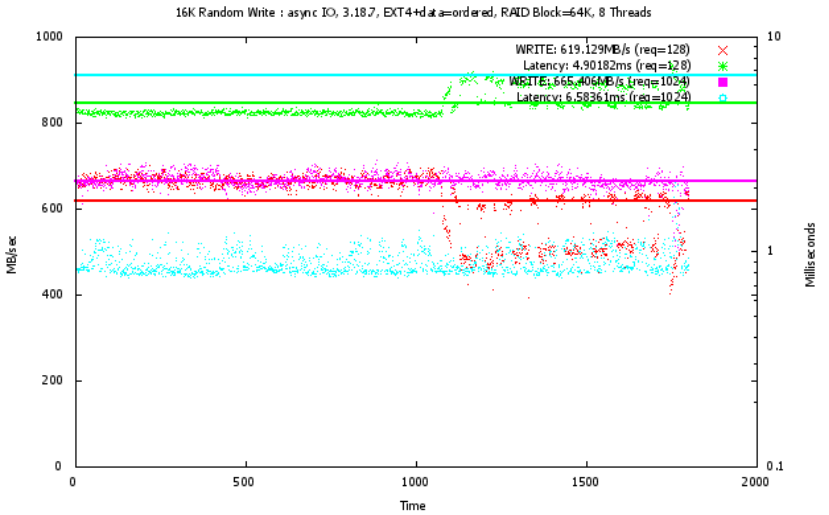
64K RAID block, EXT4, kernel 3.13
Write throughput **102MB**/sec
99th-percentile latency: 171ms

Show Me The DATA!

Kernel 3.18.7, EXT4 filesystem + data=ordered, sysbench async random write



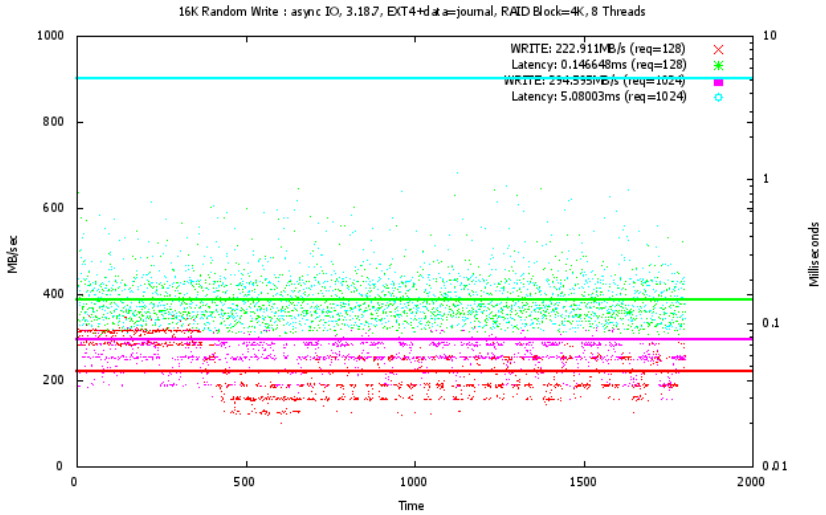
4K RAID block, EXT4, kernel 3.18
Write throughput 537MB/sec
99th-percentile latency: 3.58ms



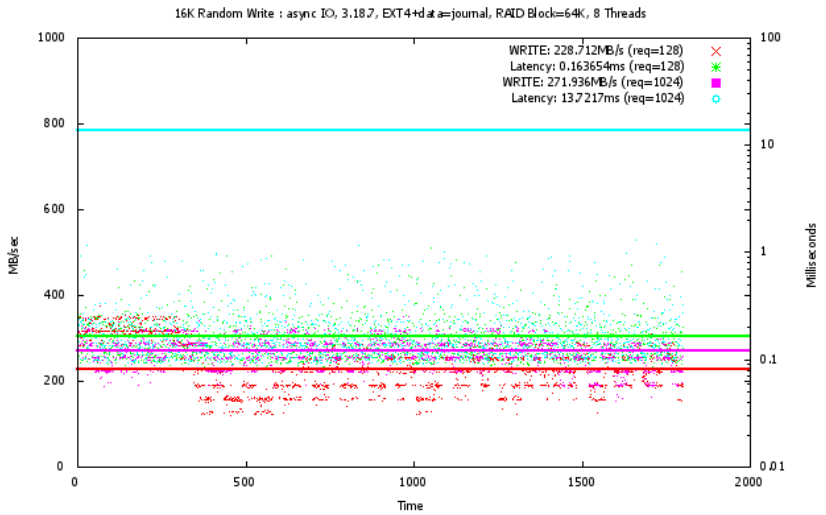
64K RAID block, EXT4, kernel 3.18
Write throughput 619MB/sec
99th-percentile latency: 4.90ms

Show Me The DATA!

Kernel 3.18.7, EXT4 filesystem + data=journal, sysbench async random write



4K RAID block, EXT4, kernel 3.18
Write throughput **223MB**/sec
99th-percentile latency: 0.15ms



64K RAID block, EXT4, kernel 3.18
Write throughput **228MB**/sec
99th-percentile latency: 0.16ms

Show Me The DATA!

Sysbench FileIO General Thoughts

EXT4 + journal throughput is often pretty bad (relative to the others).

EXT4 + journal latency with 3.18 often quite good.

3.18 usually better than its 3.13 counterpart.

Some results more surprising than others.

Not much difference between highest-performing configurations.

Show Me The DATA!

Sysbench async random read

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	4K	486.28	0.82
3.13	EXT4, ordered	4K	477.03	0.51
3.13	XFS	4K	629.43	2.13
3.18	EXT4, journal	4K	520.86	0.39
3.18	EXT4, ordered	4K	620.27	3.14
3.18	XFS	4K	583.78	3.96
3.13	EXT4, journal	16K	624.24	1.82
3.13	EXT4, ordered	16K	712.95	4.24
3.13	XFS	16K	720.68	5.90
3.18	EXT4, journal	16K	489.51	0.45
3.18	EXT4, ordered	16K	718.04	4.20
3.18	XFS	16K	719.09	4.39

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	64K	480.52	0.48
3.13	EXT4, ordered	64K	474.60	0.57
3.13	XFS	64K	696.53	5.22
3.18	EXT4, journal	64K	477.60	0.51
3.18	EXT4, ordered	64K	717.33	4.20
3.18	**XFS**	**64K**	**697.32**	**3.73**
3.13	EXT4, journal	256K	718.73	3.48
3.13	EXT4, ordered	256K	696.88	2.61
3.13	XFS	256K	698.53	3.81
3.18	EXT4, journal	256K	489.22	0.47
3.18	EXT4, ordered	256K	700.53	4.23
3.18	XFS	256K	718.00	4.38

Show Me The DATA!

Sysbench async random write

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	4K	89.05	**200.96**
3.13	EXT4, ordered	4K	**86.96**	124.49
3.13	XFS	4K	549.54	3.55
3.18	EXT4, journal	4K	222.91	0.15
3.18	EXT4, ordered	4K	537.33	3.58
3.18	XFS	4K	547.90	3.70
3.13	EXT4, journal	16K	530.46	3.22
3.13	EXT4, ordered	16K	643.41	4.65
3.13	XFS	16K	656.71	5.50
3.18	EXT4, journal	16K	233.54	0.15
3.18	EXT4, ordered	16K	617.95	4.92
3.18	XFS	16K	652.75	6.33

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	64K	102.61	171.81
3.13	EXT4, ordered	64K	87.99	122.17
3.13	XFS	64K	659.72	6.19
3.18	EXT4, journal	64K	228.71	0.16
3.18	EXT4, ordered	64K	619.13	4.90
3.18	**XFS**	**64K**	**650.44**	**6.22**
3.13	EXT4, journal	256K	644.68	3.28
3.13	EXT4, ordered	256K	652.33	3.70
3.13	XFS	256K	657.08	4.69
3.18	EXT4, journal	256K	233.39	**0.14**
3.18	EXT4, ordered	256K	627.89	4.77
3.18	XFS	256K	**661.27**	2.98

Show Me The DATA!

Sysbench async random read/write

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	4K	181 / 120	1.53
3.13	EXT4, ordered	4K	178 / 119	1.12
3.13	XFS	4K	347 / 232	3.76
3.18	EXT4, journal	4K	204 / 136	1.52
3.18	EXT4, ordered	4K	339 / 226	3.46
3.18	XFS	4K	346 / 230	3.66
3.13	EXT4, journal	16K	343 / 229	2.39
3.13	EXT4, ordered	16K	407 / 271	4.45
3.13	XFS	16K	411 / 274	4.59
3.18	EXT4, journal	16K	206 / 137	1.41
3.18	EXT4, ordered	16K	411 / 274	4.41
3.18	XFS	16K	413 / 275	4.58

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	64K	158 / 105	0.89
3.13	EXT4, ordered	64K	177 / 118	1.12
3.13	XFS	64K	238 / 159	12.06
3.18	EXT4, journal	64K	202 / 135	1.60
3.18	EXT4, ordered	64K	409 / 273	4.41
3.18	XFS	64K	402 / 268	8.12
3.13	EXT4, journal	256K	410 / 273	4.41
3.13	EXT4, ordered	256K	408 / 272	4.05
3.13	XFS	256K	406 / 270	4.65
3.18	EXT4, journal	256K	207 / 138	1.45
3.18	EXT4, ordered	256K	401 / 267	4.46
3.18	XFS	256K	412 / 275	6.15

Show Me The DATA!

Sysbench sync random write, 8 threads

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	4K	**84.22**	141.72
3.13	EXT4, ordered	4K	90.30	178.79
3.13	XFS	4K	536.63	0.47
3.18	EXT4, journal	4K	283.24	0.10
3.18	EXT4, ordered	4K	536.05	0.56
3.18	XFS	4K	511.60	0.57
3.13	EXT4, journal	16K	501.17	0.62
3.13	EXT4, ordered	16K	498.73	0.94
3.13	XFS	16K	574.31	0.54
3.18	EXT4, journal	16K	306.40	0.14
3.18	EXT4, ordered	16K	592.56	0.48
3.18	XFS	16K	583.59	0.53

Kernel	FS	RAID chunk	Throughput: MB/s	p99 latency: ms
3.13	EXT4, journal	64K	95.22	**185.69**
3.13	EXT4, ordered	64K	87.23	157.61
3.13	XFS	64K	568.23	0.60
3.18	EXT4, journal	64K	300.47	0.16
3.18	EXT4, ordered	64K	586.47	0.52
3.18	**XFS**	**64K**	**564.31**	**0.71**
3.13	EXT4, journal	256K	515.54	0.90
3.13	EXT4, ordered	256K	484.24	0.92
3.13	XFS	256K	575.36	0.54
3.18	EXT4, journal	256K	306.85	**0.10**
3.18	EXT4, ordered	256K	566.73	0.64
3.18	XFS	256K	**584.06**	0.55

Show Me The DATA!

What else can we mess with?

Other possible tweaks & experiments (/sys/block/${DEV}/queue)

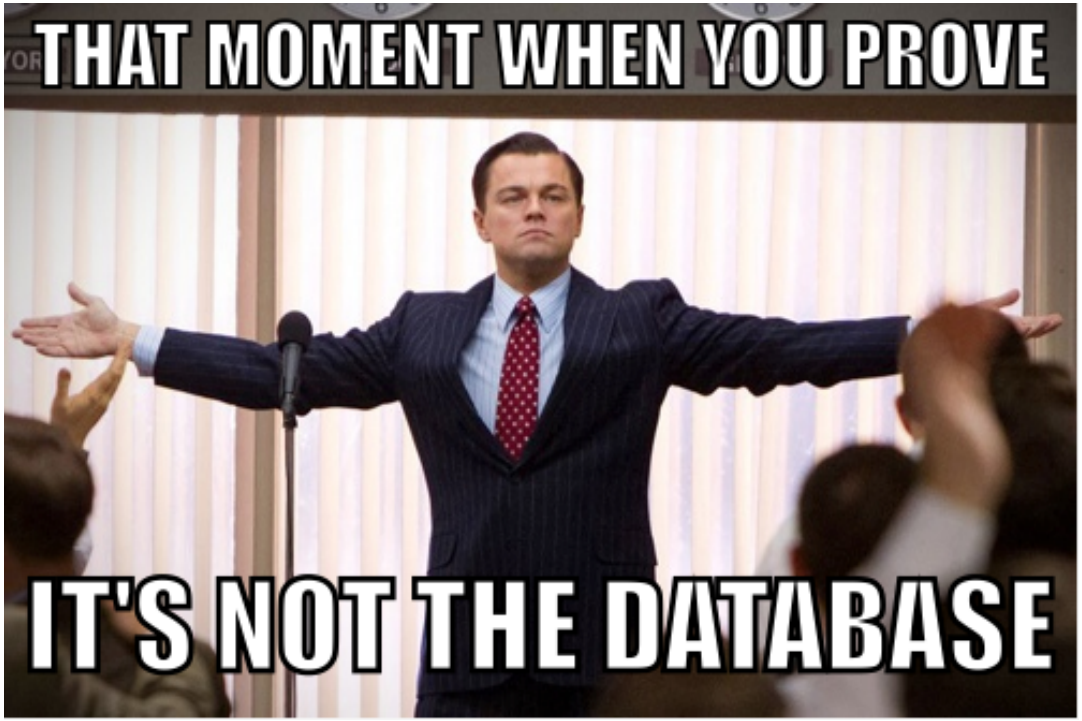
- nr_requests (default: 128 | try: 1024)
- rq_affinity (default: 1 | try: 2)
  - 1: migrate request completions to the cpu group that submitted the request.
  - 2: force the completion to run on the requesting cpu.
- read_ahead_kb
- These are workload dependent!

Final choice: kernel 3.18.7 + XFS + 64K RAID block size.

- Best **overall** performance for async random read.
- Very competitive performance everywhere else.
- Networking-related kernel bugs (Xen-specific) in 3.13 that aren't fixed until 3.16.

Show Me The DATA!

Sysbench OLTP



Show Me The DATA!

FileIO is great, but what about MySQL?

- **Sysbench OLTP**
  - 1024 tables of 1M rows.
  - Parameters modified from default to more closely mirror our query patterns.
  - Modified sysbench source to print p50, p90, p99.
- **Tested configurations:**
  - Current production 3.2 kernel & configuration.
  - 3.13.0 kernel, although as with the 3.8 kernel and the fileIO test, this was quickly abandoned.
  - 3.18.7 kernel with original my.cnf.
  - 3.18.7 kernel with optimized my.cnf.
  - 3.18.7 kernel with jemalloc.
  - 3.18.7 kernel with some scheduler and memory management tweaks.
  - And various permutations of the above.

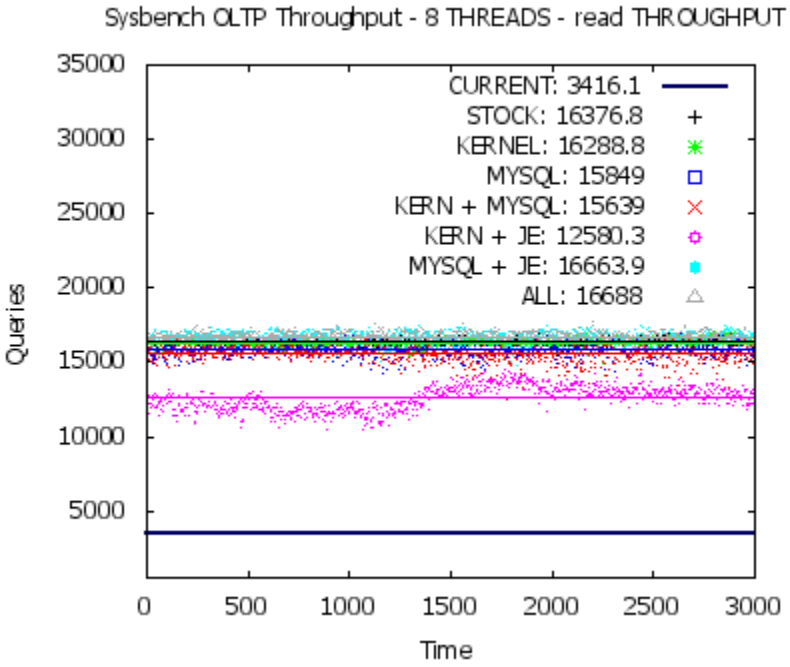
Show Me The DATA!

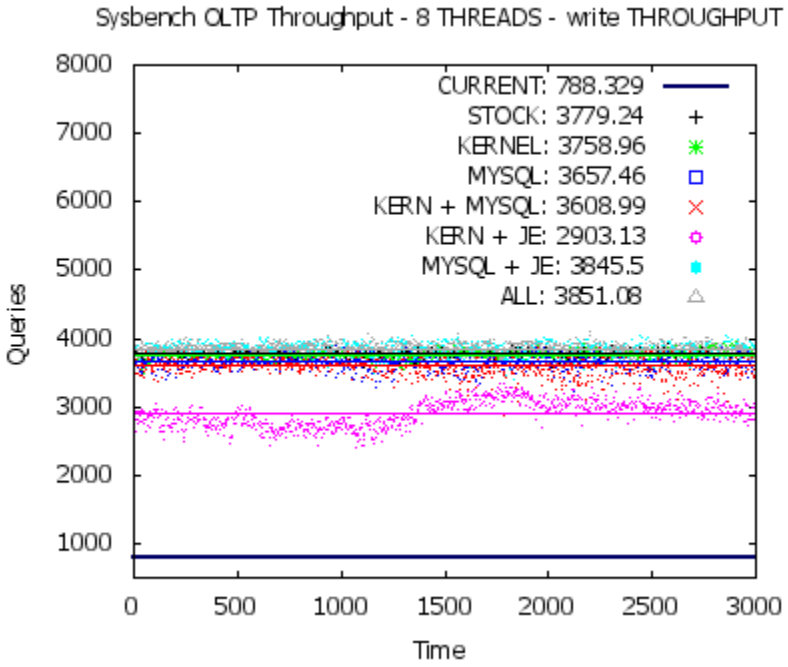
A little more bash-fu, please.

```
#!/bin/bash

outputdir=/home/esouhrada/sysbench
mkdir -p $outputdir
for threads in 8 16 32 64
do
  mysqld_multi stop 3306
  PSCOUNT=`ps aux | grep mysqld | grep -v grep | wc -l`
  while [[ $PSCOUNT -ne 0 ]];
  do
    echo "sleeping 60 seconds for mysql to stop..."
    sleep 60
    PSCOUNT=`ps aux | grep mysqld | grep -v grep | wc -l`
  done
  echo "mysql is not running.  starting it."
  mysqld_multi start 3306
  RETVAL=1
  while [[ $RETVAL -ne 0 ]] ;
  do
    echo "sleeping 60 seconds for mysql to start..."
    sleep 60
    mysqladmin ping > /dev/null
    RETVAL=$?
  done
  echo "mysql is running.  starting test at $(date)"

  mysqladmin ext -i1 -c3600 > ${outputdir}/mysqladmin.${threads}.log &

  sysbench.new --test=db/oltp.lua \
    --mysql-socket=/var/run/mysqld/mysqld_3306.sock \
    --mysql-user=root --mysql-db=sbtest --oltp-table-size=1000000 \
    --max-requests=0 --max-time=3600 --rand-init=on \
    --rand-type=pareto --oltp-tables-count=1024 \
    --num-threads=${threads} --report-interval=1 \
    --oltp-reconnect=on --oltp-read-only=off \
    --oltp-sum-ranges=0 --oltp-non-index-updates=0 \
    --oltp-dist-type=pareto --oltp-test-mode=notrx \
    --oltp-skip-trx=on --mysql-ignore-errors=all \
    --oltp-non-trx-mode=insert,select,update_key \
    --percentile=99 run | tee ${outputdir}/oltp.${threads}.log

  echo "done at $(date)"
done
```

Show Me The DATA!

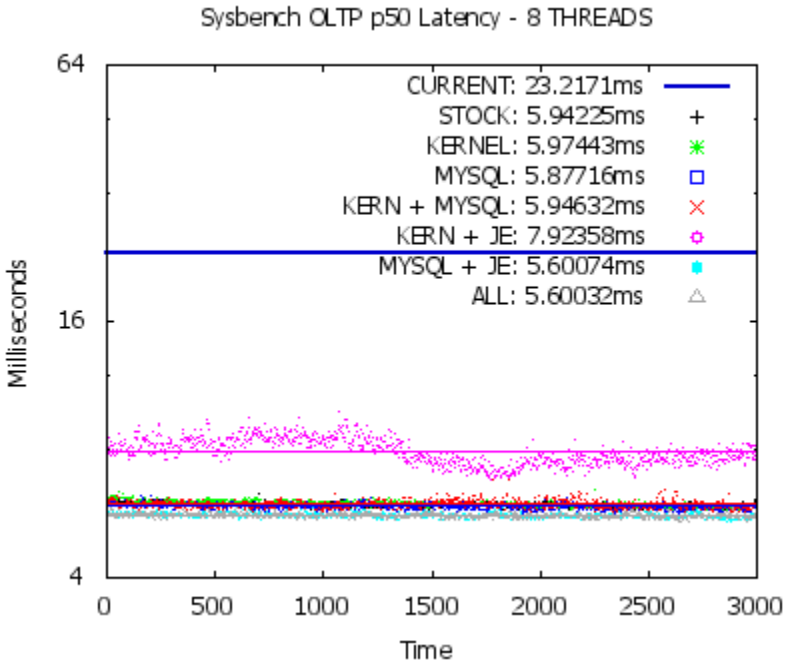
Sysbench OLTP throughput at 8 threads

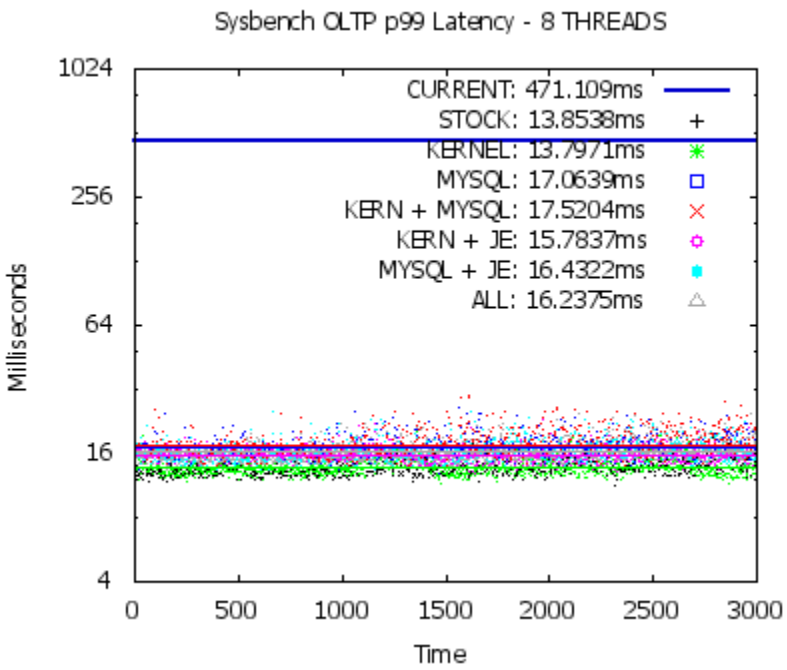




Show Me The DATA!

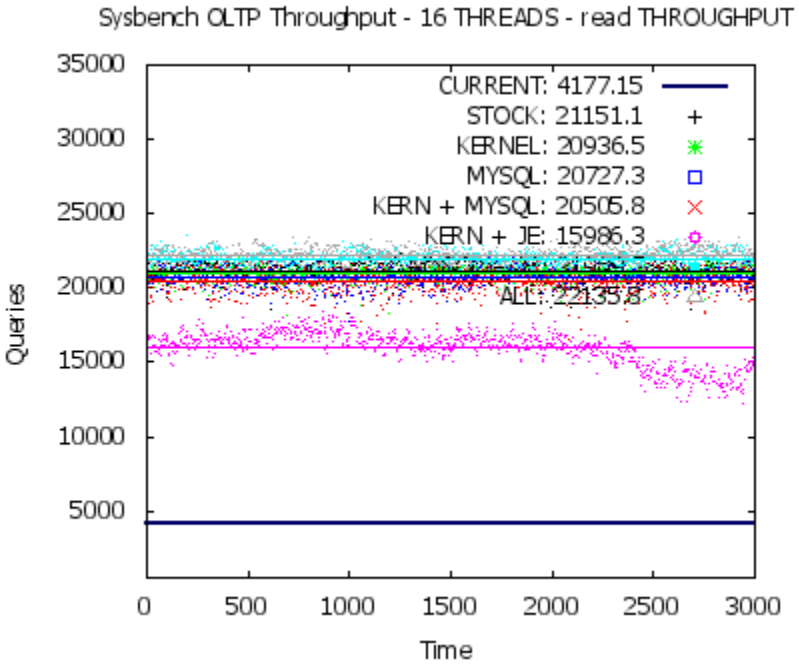
Sysbench OLTP latency at 8 threads

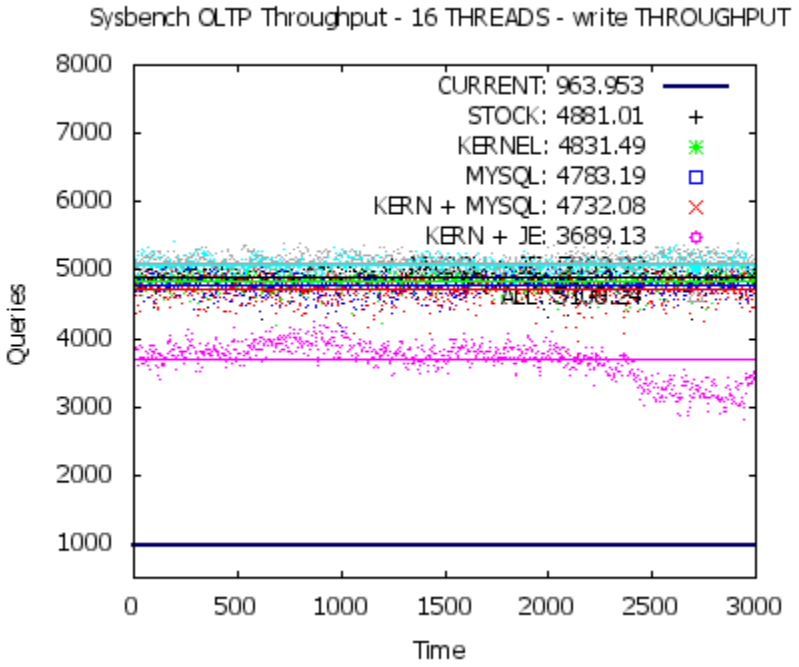




Show Me The DATA!

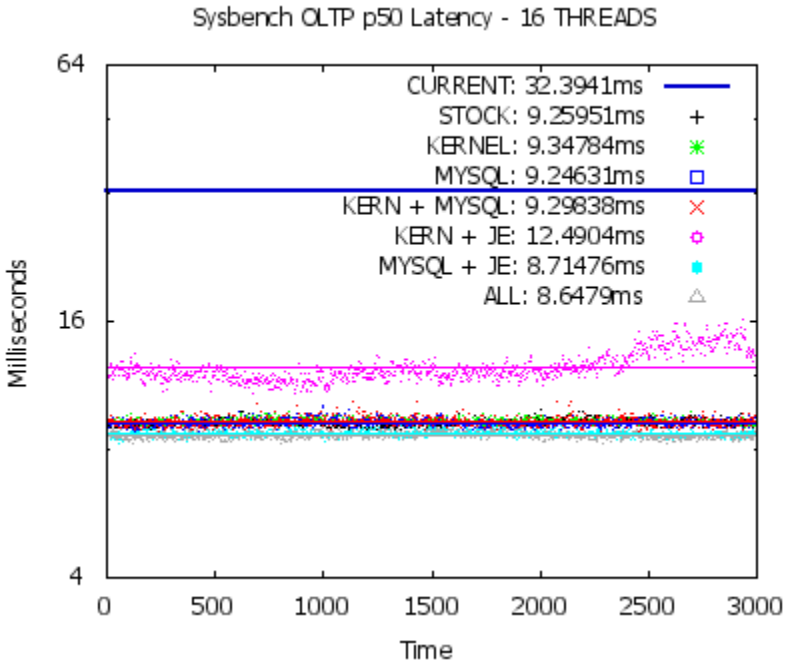
Sysbench OLTP throughput at 16 threads

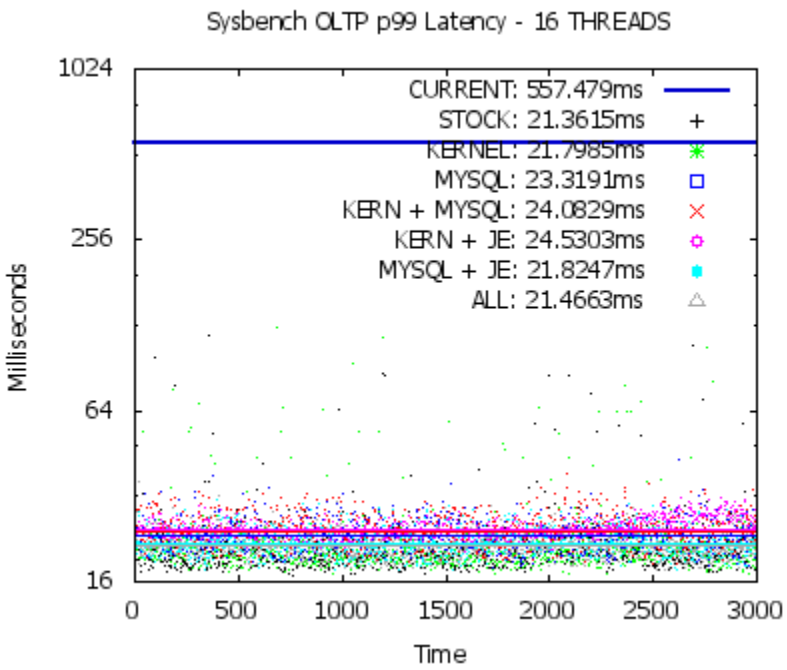




Show Me The DATA!

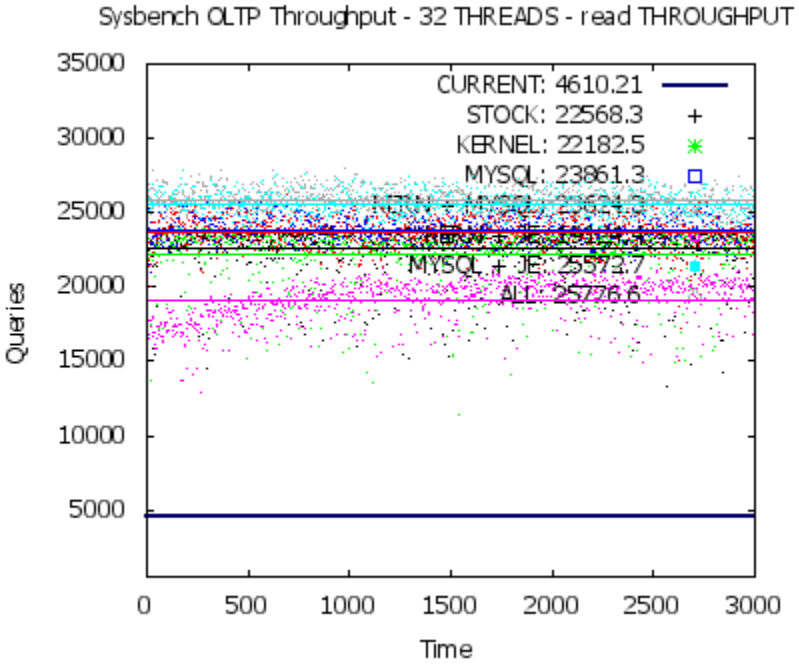
Sysbench OLTP latency at 16 threads

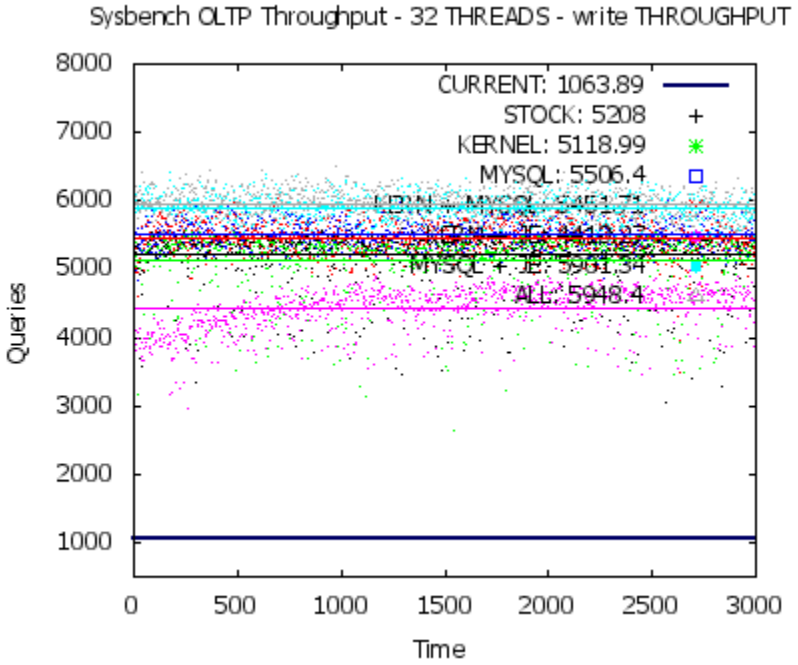




Show Me The DATA!

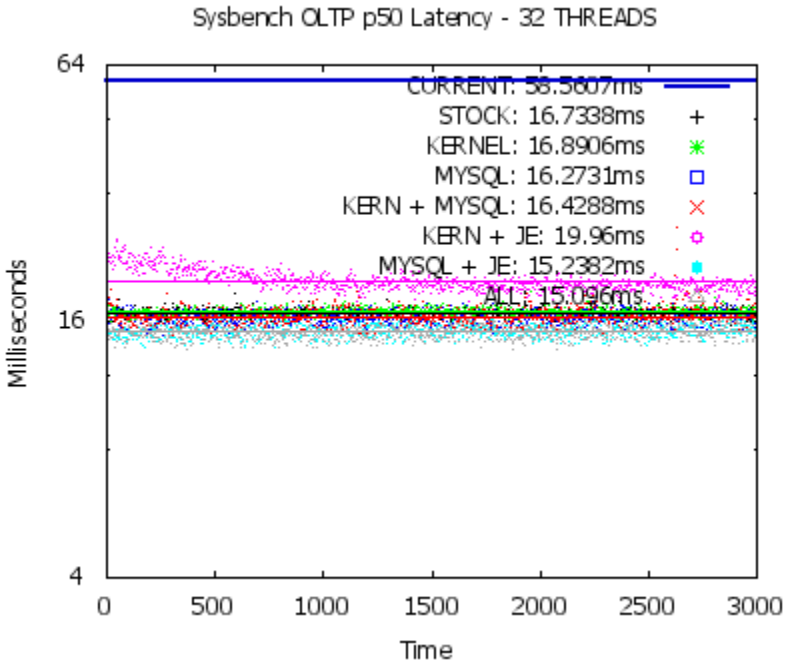
Sysbench OLTP throughput at 32 threads

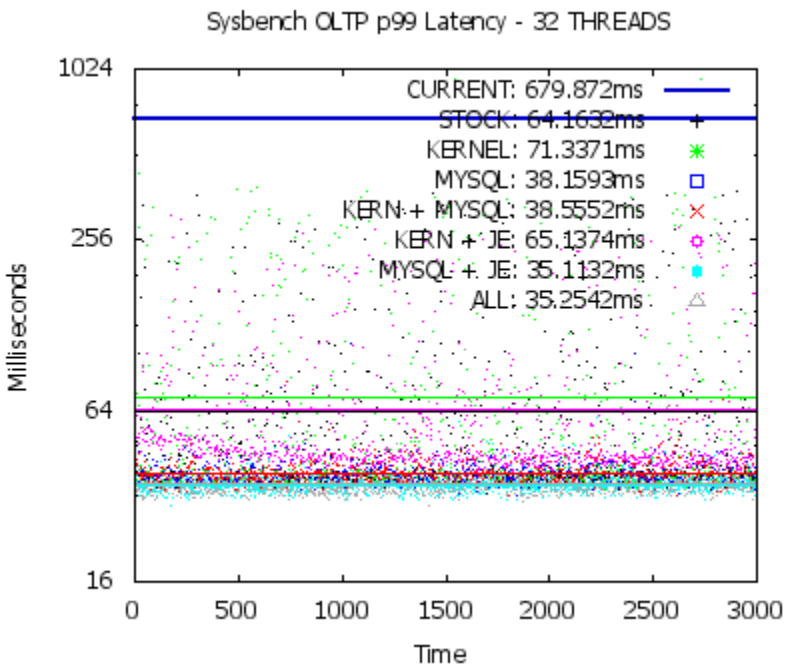




Show Me The DATA!

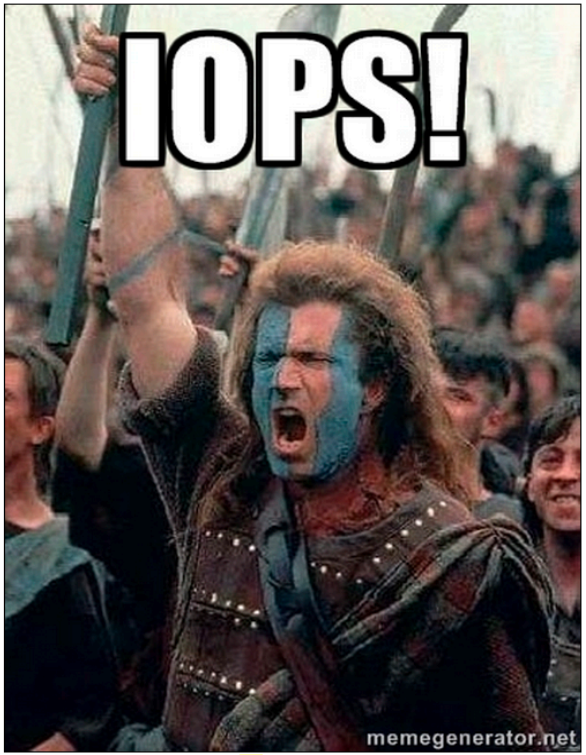
Sysbench OLTP latency at 32 threads





Show Me The DATA!

WWWWD (What Would William Wallace Do) ?



Let Them Eat IRQs!

Other (P)interesting Discoveries

- irqbalance < 1.0.6 (Ubuntu 12.04 has 0.56) does not work with AWS block devices (EBS or otherwise).
- This is a Xen incompatibility, not just an AWS issue.
- Example:

```
$ cat /proc/interrupts | grep xen-dyn-event
 113: 1162533196    0    0    0    0    0    0    0   xen-dyn-event     blkif
 114: 1106705502    0    0    0    0    0    0    0   xen-dyn-event     blkif
 115: 3761537131    0    0    0    0    0    0    0   xen-dyn-event     eth0
 116:    2369794    0    0    0    0    0    0    0   xen-dyn-event     blkif
 117:    2368199    0    0    0    0    0    0    0   xen-dyn-event     blkif
 118:    2368535    0    0    0    0    0    0    0   xen-dyn-event     blkif
 119:    2368330    0    0    0    0    0    0    0   xen-dyn-event     blkif
```

Let Them Eat IRQs!

Because, well, they're cheaper than cake.

- All the IRQs are going to core 0.
- If core 0 gets overwhelmed, the other cores will starve.
- This manifests as one core very busy and the rest basically idle.
- Disk-related IRQs can be properly balanced with irqbalance >= 1.0.6
  - Go to github, compile it yourself if you need to.
  - This is only really a problem if you use multiple block devices and have nontrivial disk IO.
- Network-related IRQs can't be handled this way (in classic AWS)
  - There's only one send/receive queue.
  - Fix it in the kernel: enable RPS (Receive Packet Steering)[1]

```
echo ffff > /sys/class/net/eth0/queues/rx-0/rx_cpus
```

[1] - http://engineering.pinterest.com/post/53467339970/building-pinterest-in-the-cloud

I Don’t Always Test…

But when I do, I test in production

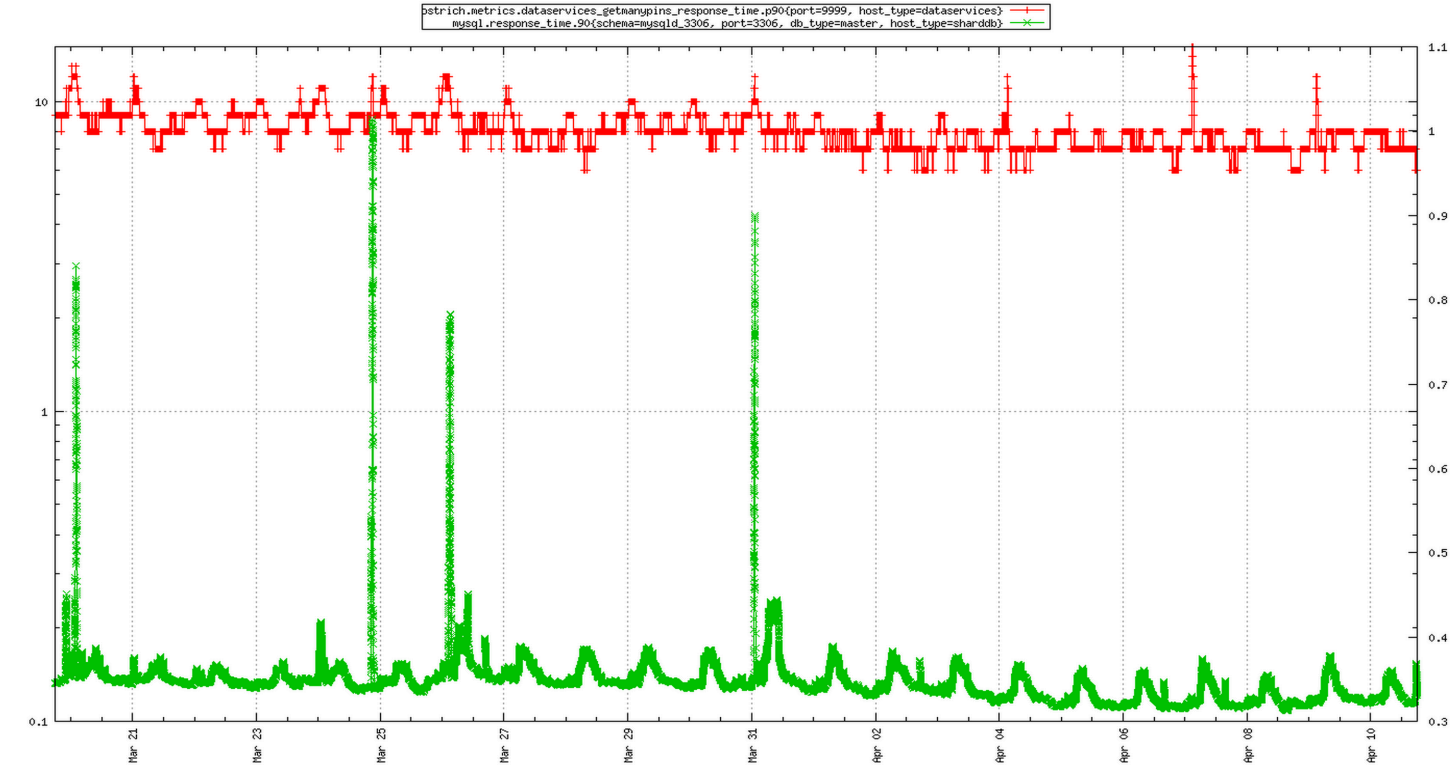
I Don’t Always Test…

But when I do, I test in production

- We rolled out our updated configuration to all primary (T1 and T2) servers at the end of March / early April 2015.
- We’ve also had roughly a 50% increase in MySQL activity.
  - On 01 January 2015, peak load for an individual server ~ 2000 QPS
  - On 31 January 2015, peak load for an individual server ~ 2500 QPS
  - On 31 March 2015, peak load for an individual server ~ 3000 QPS
- We measure both server latency (via the query response time plugin) and client-perceived latency.
- Graphs to follow were created by Rob Wultsch, covering the time just before the upgrades were rolled out until last Friday (10 April)

I Don’t Always Test…

But when I do, I test in production

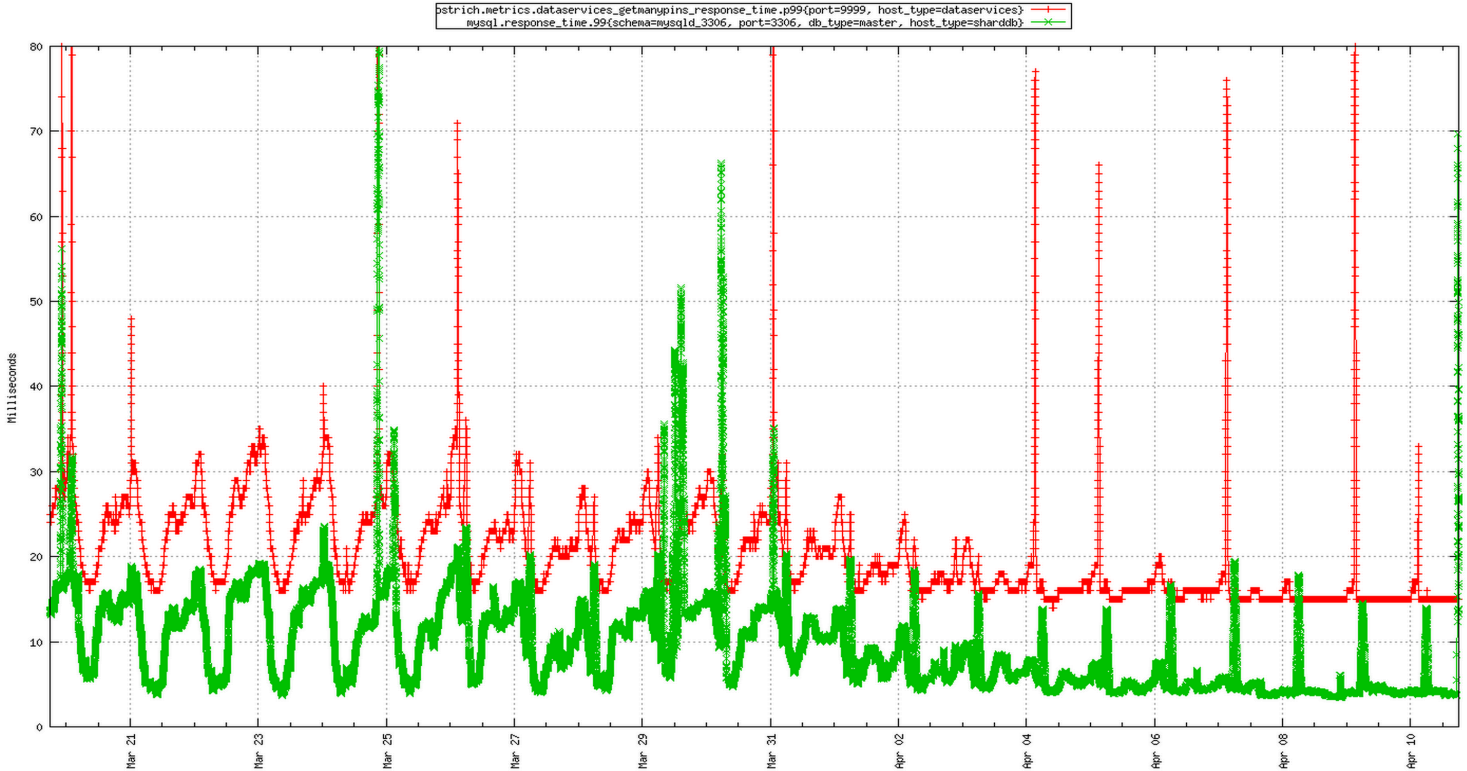


Client-measured p90 dropped from 8-15ms to 7-8ms

Server-measured p90 dropped from 0.37ms to 0.32ms and outliers disappeared

I Don’t Always Test…

But when I do, I test in production

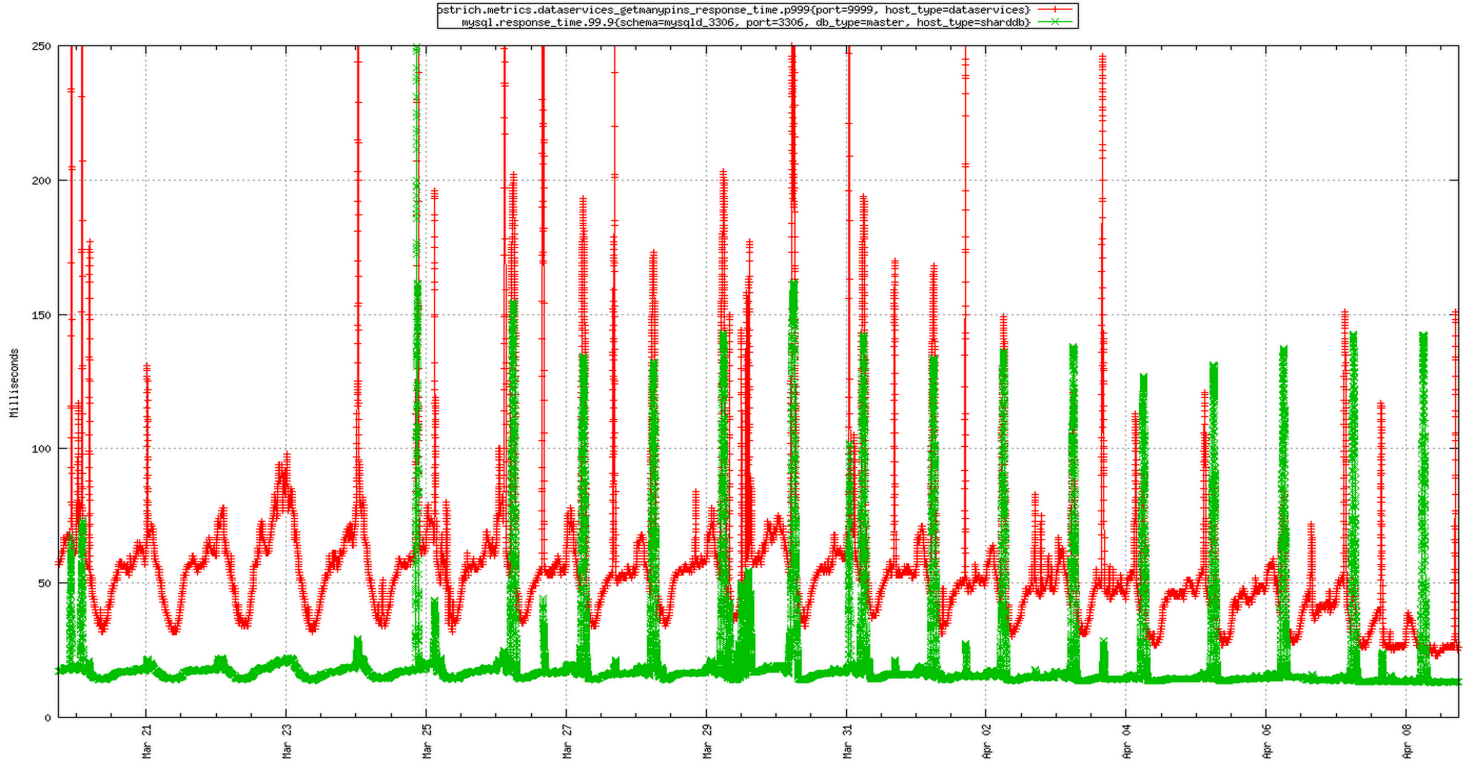


Client-measured p99 dropped from a highly variable 15-35ms to a generally-flat 15ms, and outlier magnitude dropped under 80ms.

Server-measured p99 dropped from a wavy 5-15ms to a flat 5ms with a daily spike to 18ms due to maintenance.

I Don’t Always Test…

But when I do, I test in production

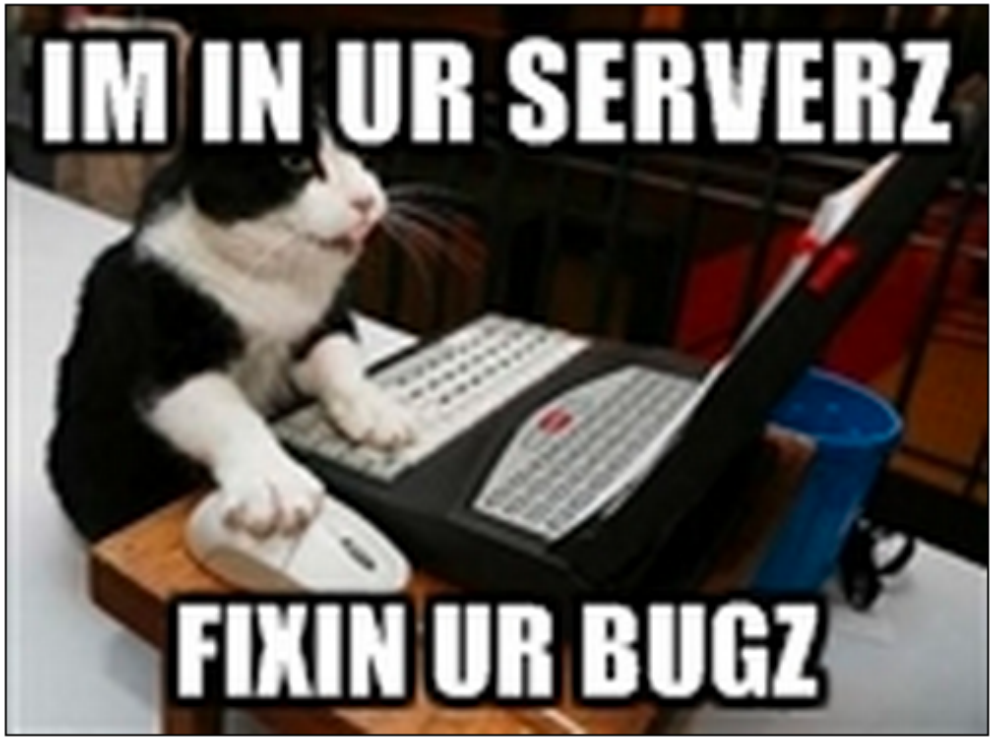


Client-measured p999 dropped from a periodic 30-90ms to a periodic 20-40ms; outliers dropped from 250ms to 150ms

Server-measured p999 flattened out from 15-20ms to 15ms with a daily spike to 140ms due to maintenance (a more aggressive / thorough checksum process).

I Don't Always Test…

But when I do, I test in production



- 50% more load handled.
- Response time down.
- Response time variability way down.
- Server headroom way up.

Droppin' Mo' Science

All the changes that are fit to print.

Full list of customizations / changes deployed:

- Linux kernel 3.18.7
- irqbalance 1.0.8
- RPS enabled
- Jemalloc instead of Glibc
- Disk IO scheduler = noop
- XFS + 64K RAID block size
- Mount options:
  - noatime,nobarrier,discard,inode64,logbsize=256k
- my.cnf changes:
  - innodb_max_dirty_pages_pct = 75 (was 12)
  - innodb_checksum_algorithm = CRC32 (was NONE)
  - innodb_io_capacity = 10000 (was 500)
  - innodb_io_capacity_max = 16000 (was 1000)
  - innodb_lru_scan_depth = 2000 (was 1024)
  - innodb_log_buffer_size = 32M (was 8M)
  - innodb_read_io_threads = 8 (was 4)
  - innodb_write_io_threads = 8 (was 4)
  - relay_log_info_repository = TABLE
  - relay_log_recovery = ON
  - table_open_cache_instances = 8
  - metadata_locks_hash_instances = 256

The above is unlikely to be 100% optimal, but improvements are likely to be marginal.
Some of what works well for us might not work for you.
Don't take my word for it. Test for yourself.

Questions? Answers!

email: esouhrada@pinterest.com | twitter: @denshikarasu | pinterest engineering blog: http://engineering.pinterest.com