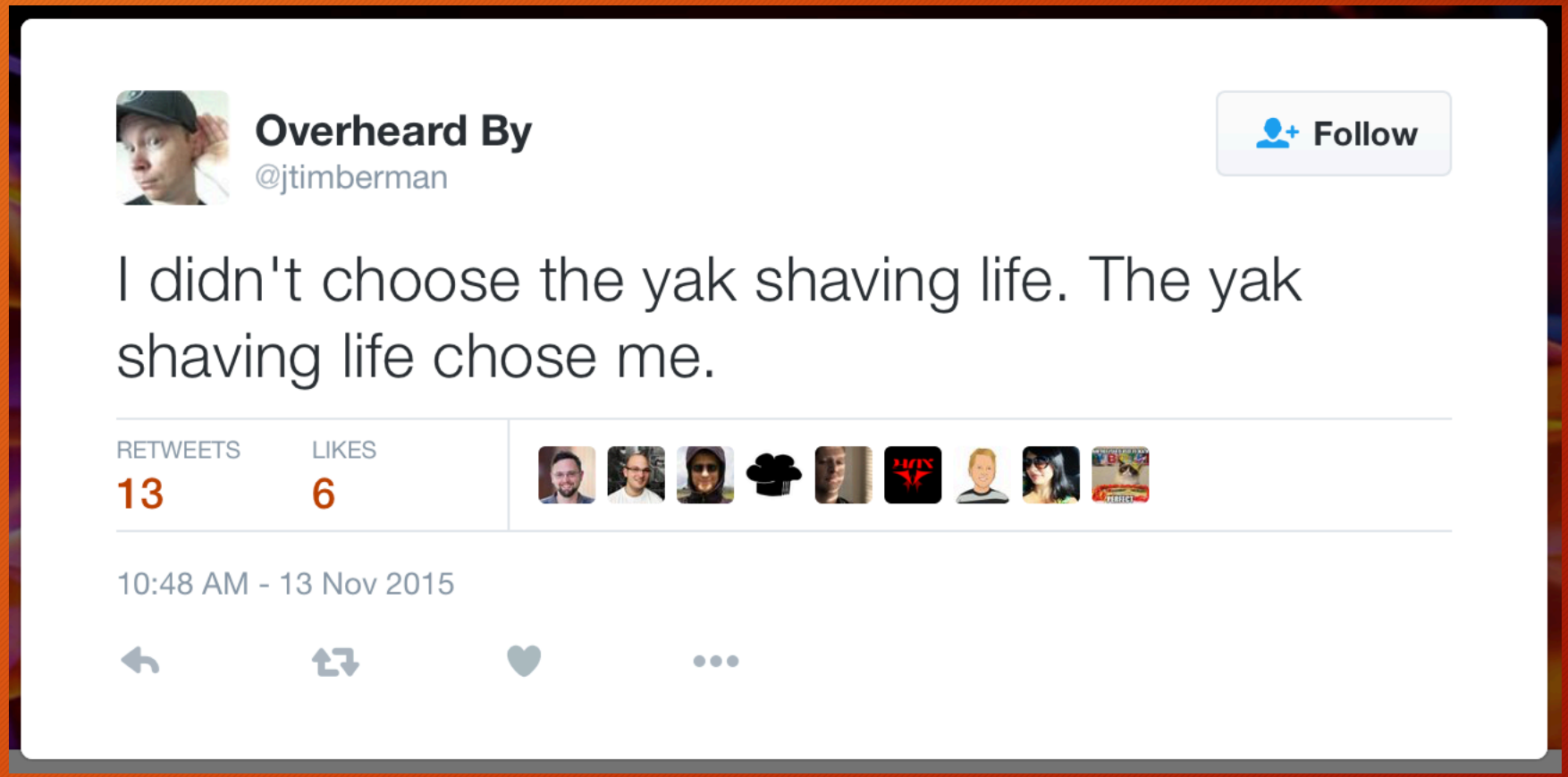
When ETS is too slow

Mark Allen
Basho Technologies
@bytemeorg
mallen@basho.com

Warning



https://twitter.com/jtimberman/status/665239748458381312

What even is “too slow?”

- Six months ago, working on speeding up Riak’s put path
- When we artificially disabled capability lookups, we saw an immediate 10% performance boost.
- We have ETS caches all over the application – many of them are essentially read-only:
  - capabilities,
  - bucket properties, and,
  - ring state

Hypothesis

- If we can speed up cache lookups, we will see big wins across the entire application.
- Idea: Let's adapt a ~~dirty hack~~ technique we use to share Riak's ring state globally on a node.
- This talk is about our journey to (sort of) understanding the limits of ETS performance and caching in Erlang.

Quick Review

ETS = Erlang Term Storage

- Super fast off-heap key-value storage.
- Tuple storage
- Idiom: store global/shared state
- A bunch of configuration options and storage engines
- Offers persistence by serializing table state to/from disk

The mochiglobal technique

Take an arbitrary Erlang term and dynamically build a module around it.

```
-module('some$random$thing’).
-export([term/0]).
term() ->
   {arbitrary, foo, bar, true, "hoge”, 3.14159}.
```

fling

- Generalized this concept into fling
- https://github.com/basho-labs/fling
- Hybrid approach – take writes against an ETS table and after X quiescent ticks (default: 5 ticks of 5 seconds each)
- Promote the ETS table state into a dynamically compiled and loaded module in the Erlang run time.
- If we take new writes, store them in ETS, the fling module is flushed out of memory and we restart our tick counter.

How fling works

- Create ETS table as usual, then you tell fling to manage it.
- Fling spawns a gen_server and “takes over” an ETS table. (See `ets:give_away/3`)
- All reads circumvent gen_server – they go straight to ets or the module
- All writes after take over go through gen_server
- gen_server manages ticks and the dynamic module compilation, loading and flush

basho_bench

- Benchmarking tool for arbitrary Erlang applications/code
- https://github.com/basho/basho_bench
- Compiles to an escript
- Concurrent like a boss
- Lots of built in key and value generator functions
- User-defined generators
- Lots of drivers for different (mostly Basho) apps

A simple driver

```
-module(basho_bench_driver_foo).
-export([new/1, run/4]).
-record(state, {bar, baz}).
new(_Id) -> {ok, #state{bar=true}}.
run(some_op, KeyGen, ValueGen, State) ->
    foo:do_something(KeyGen(), ValueGen()),
    {ok, State};
run(another_op, KeyGen, _ValueGen, State) ->
    true = foo:some_other_thing(KeyGen()),
    {ok, State}.
```

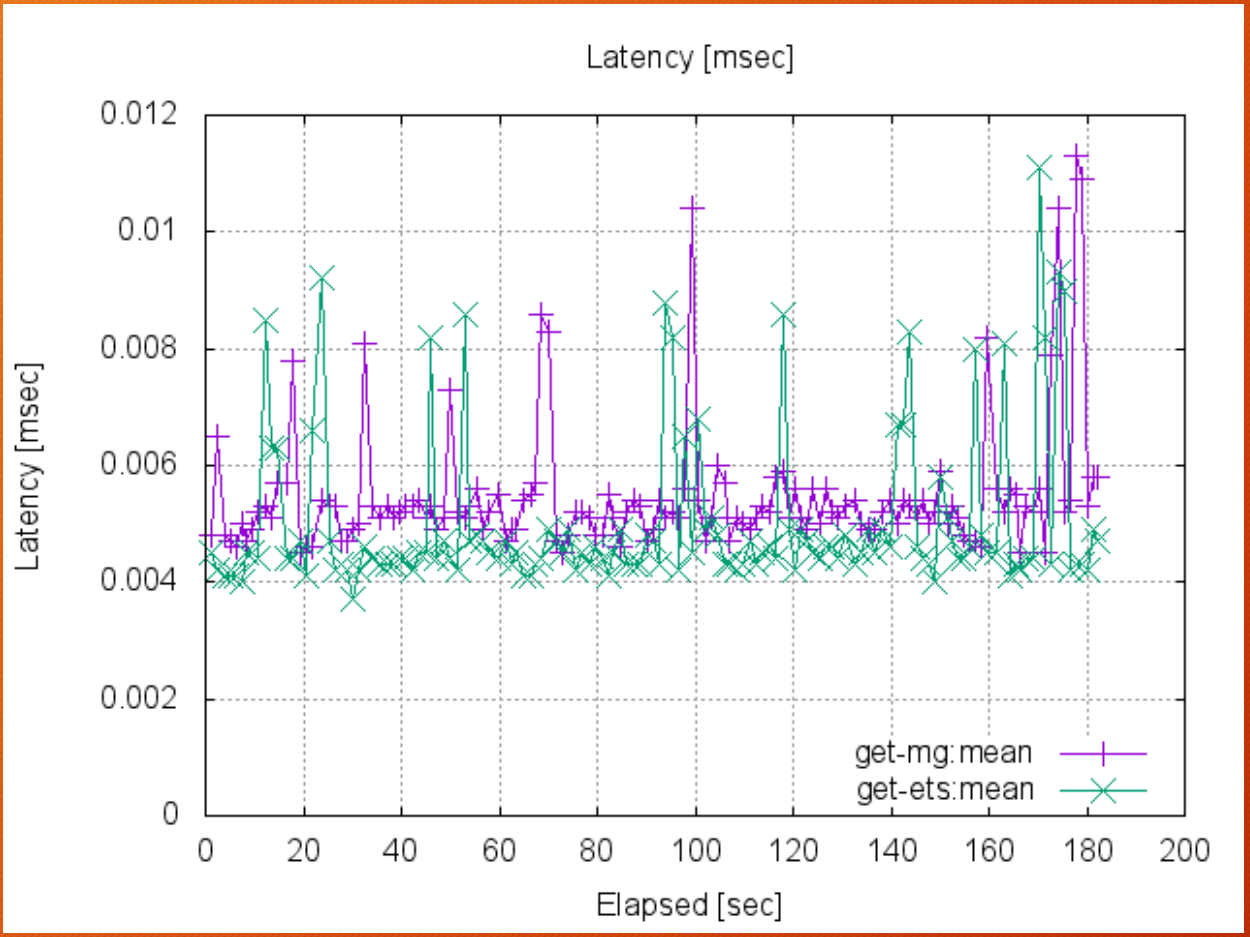
What we expected to see

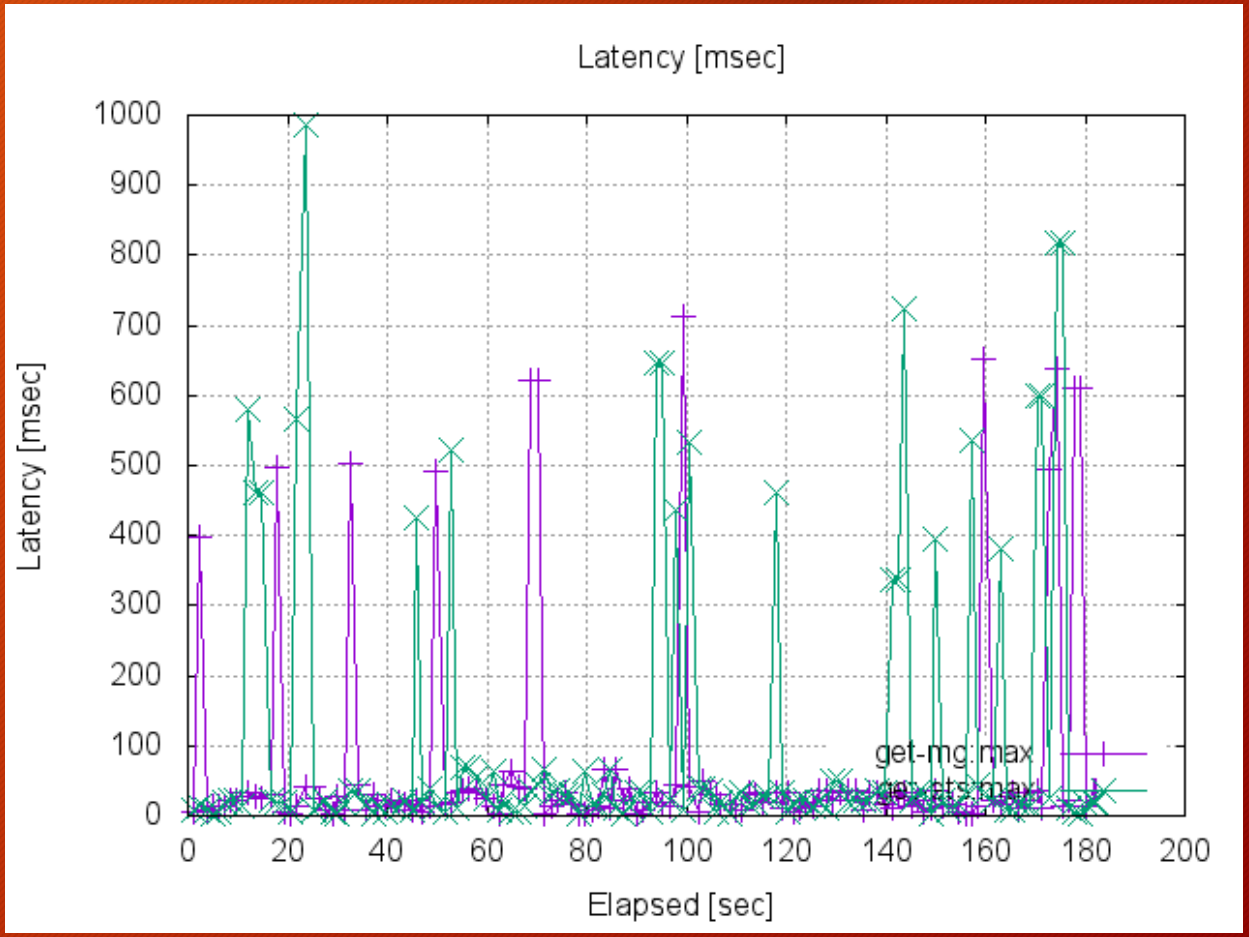
- Better lookup speed vs. ETS
- Better tail latencies vs. ETS

Spoiler Alert

Your intuition is usually wrong about optimizations.

What we saw


Latency [msec]
0.012
0.01
0.008
0.006
0.004
0.002
0
Latency [msec]
0
20
40
60
80
100
120
140
160
180
200
Elapsed [sec]
get-mg:mean
get-ets:mean


Latency [msec]
1000
900
800
700
600
500
400
300
200
100
0
Latency [msec]
0
20
40
60
80
100
120
140
160
180
200
Elapsed [sec]
get-mg:max
get-ets:max

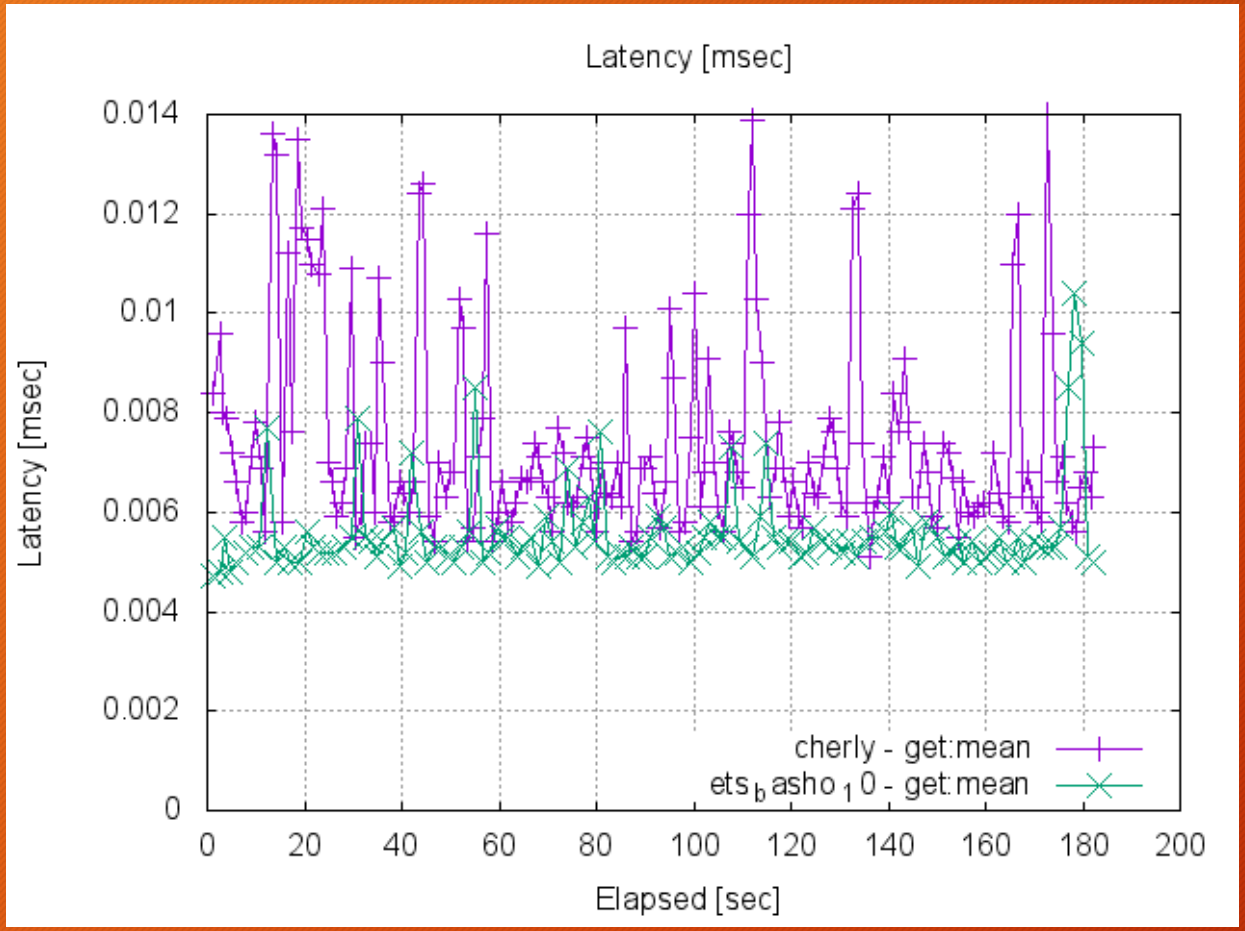
Hmm, what now?

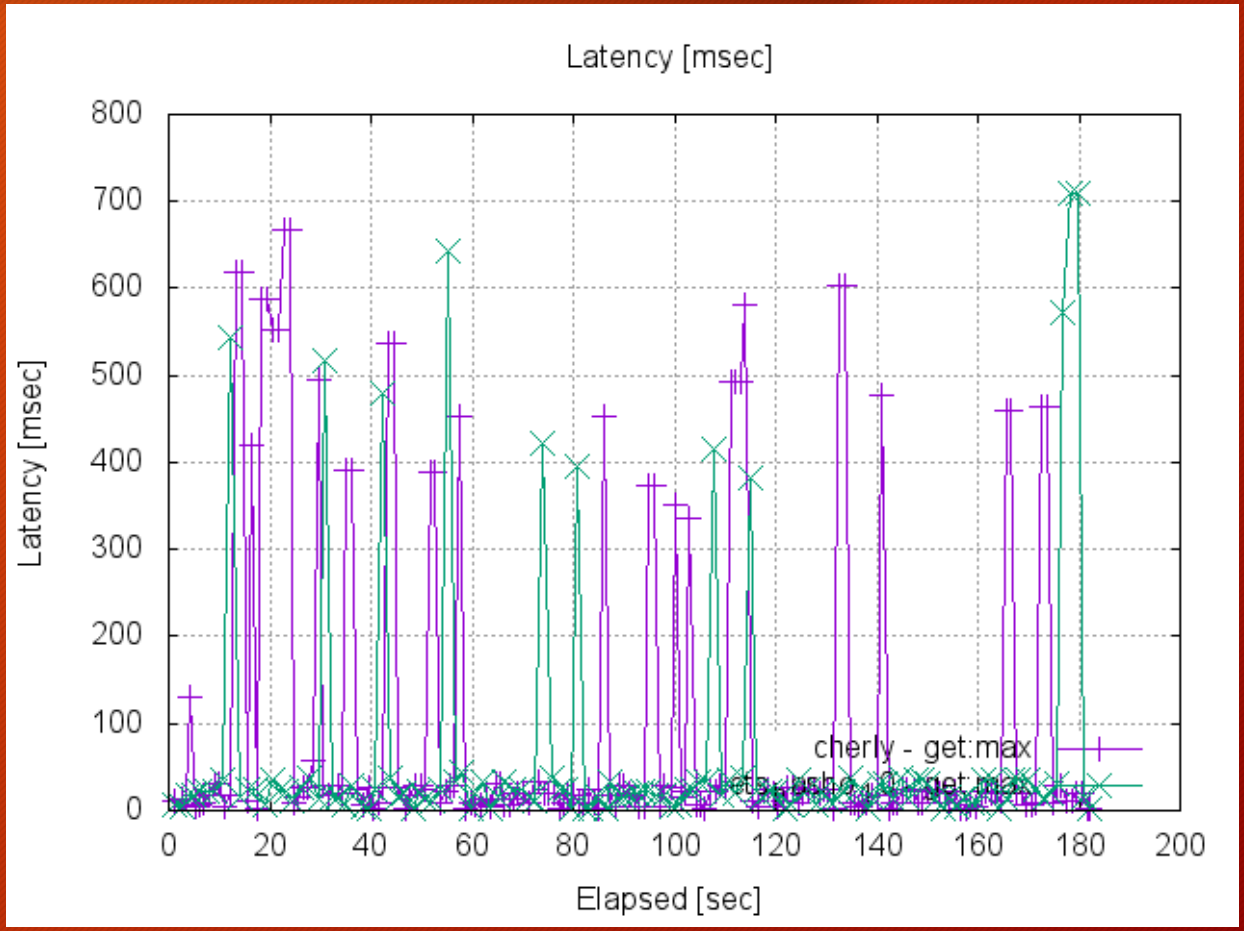
- Variations on fling
  - Store gb_trees
  - Make keys that are atom into function names
- How about a NIF?
- Evaluated two NIF caching libraries
- cherly – https://github.com/leo-project/cherly
- e2qc – https://github.com/arekinath/e2qc

cherly

- LRU cache implemented in C
- Originally written by Cliff Moon in 2009 via port driver mechanism
- Adopted, rebar'ized and refactored into a NIF by Rakuten and Leo Project in 2012

cherly results

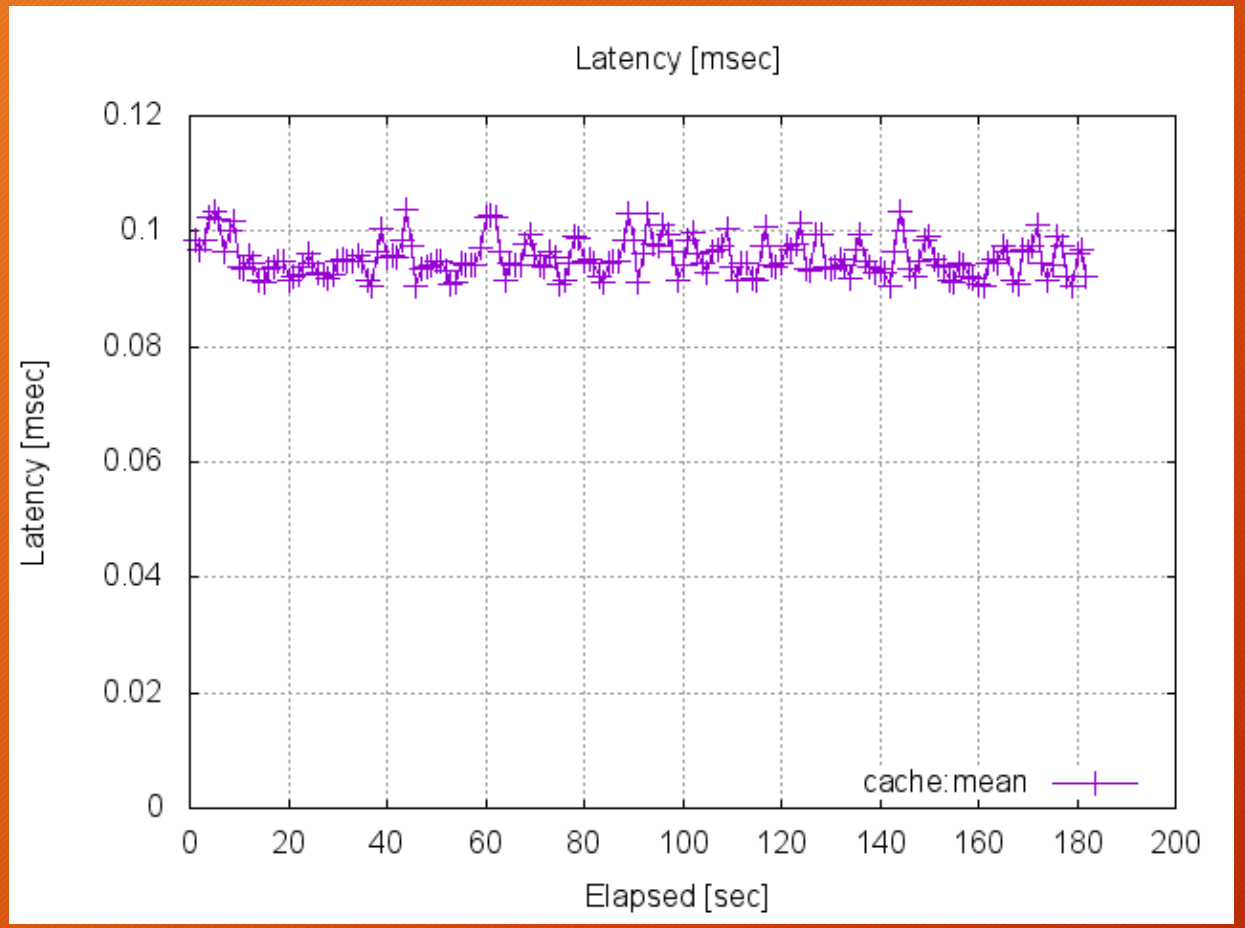


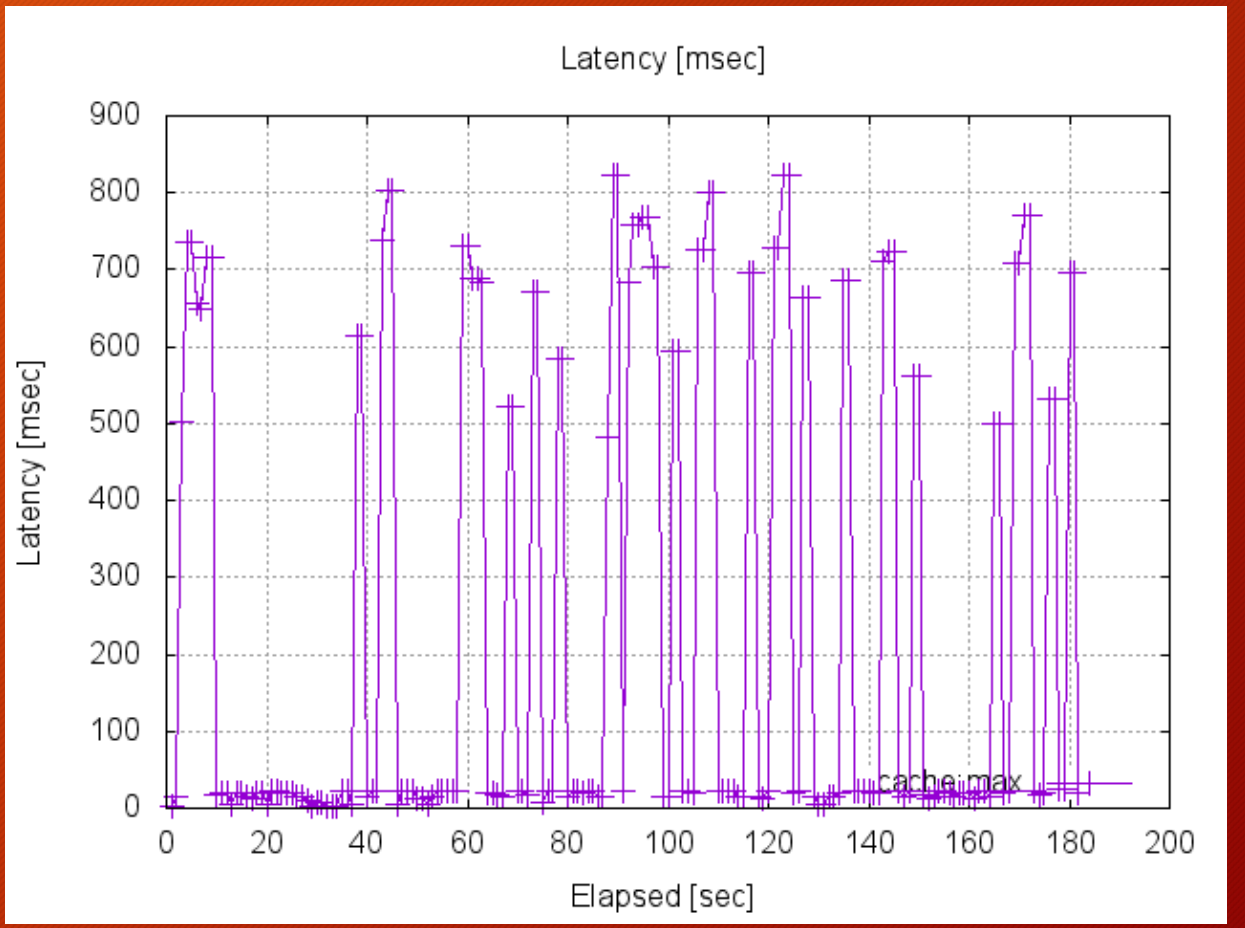


e2qc

- Implementation of the 2Q cache eviction paper
- http://www.vldb.org/conf/1994/P439.PDF
- Beautifully simple interface

e2qc results

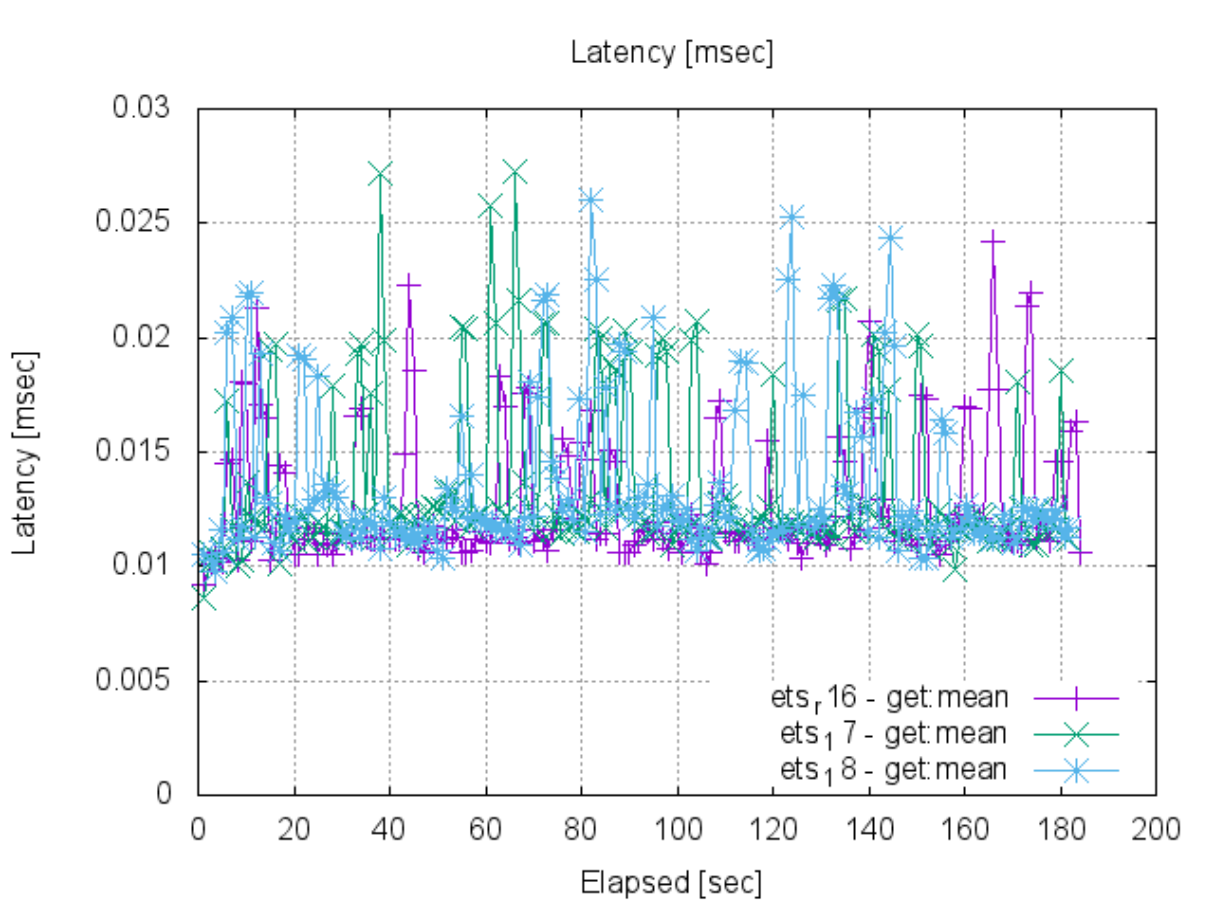


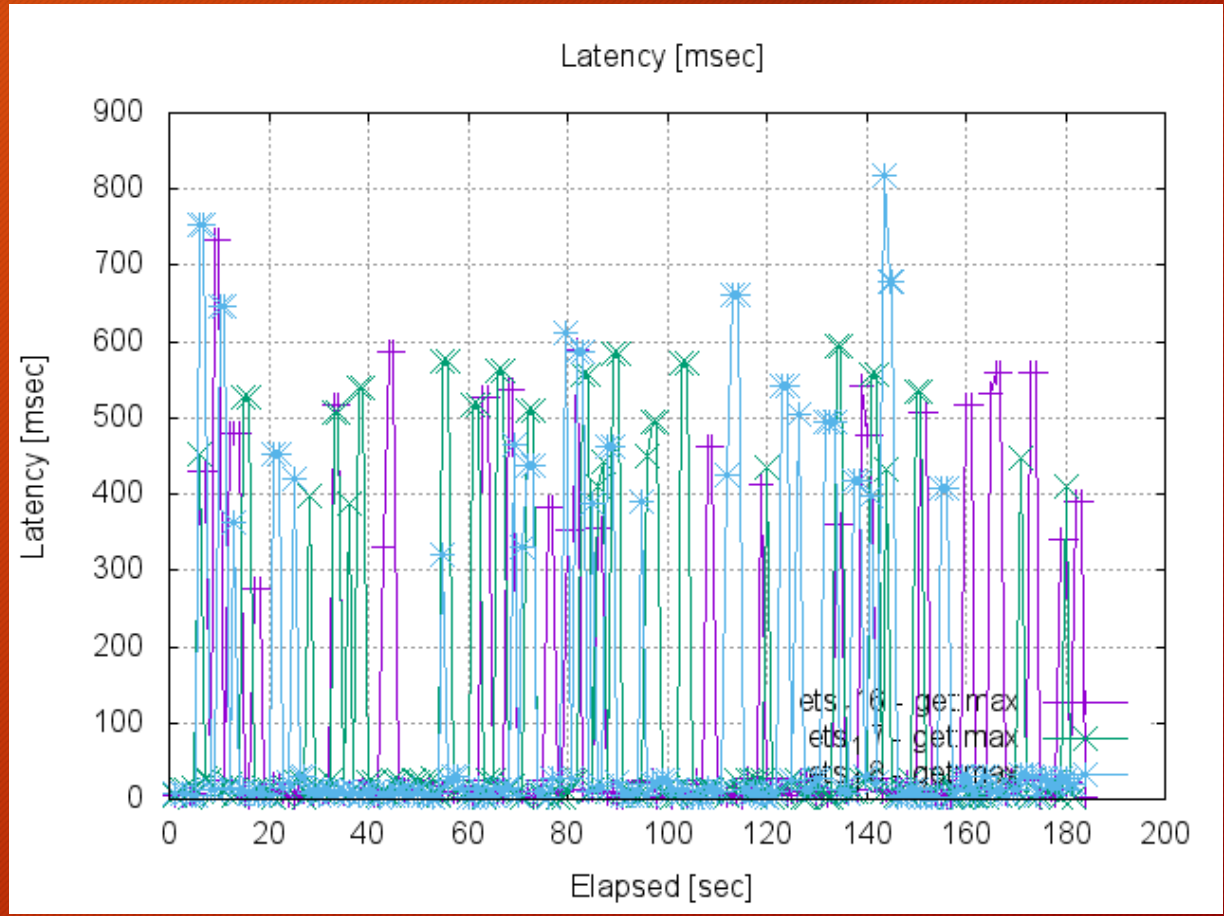


What about…

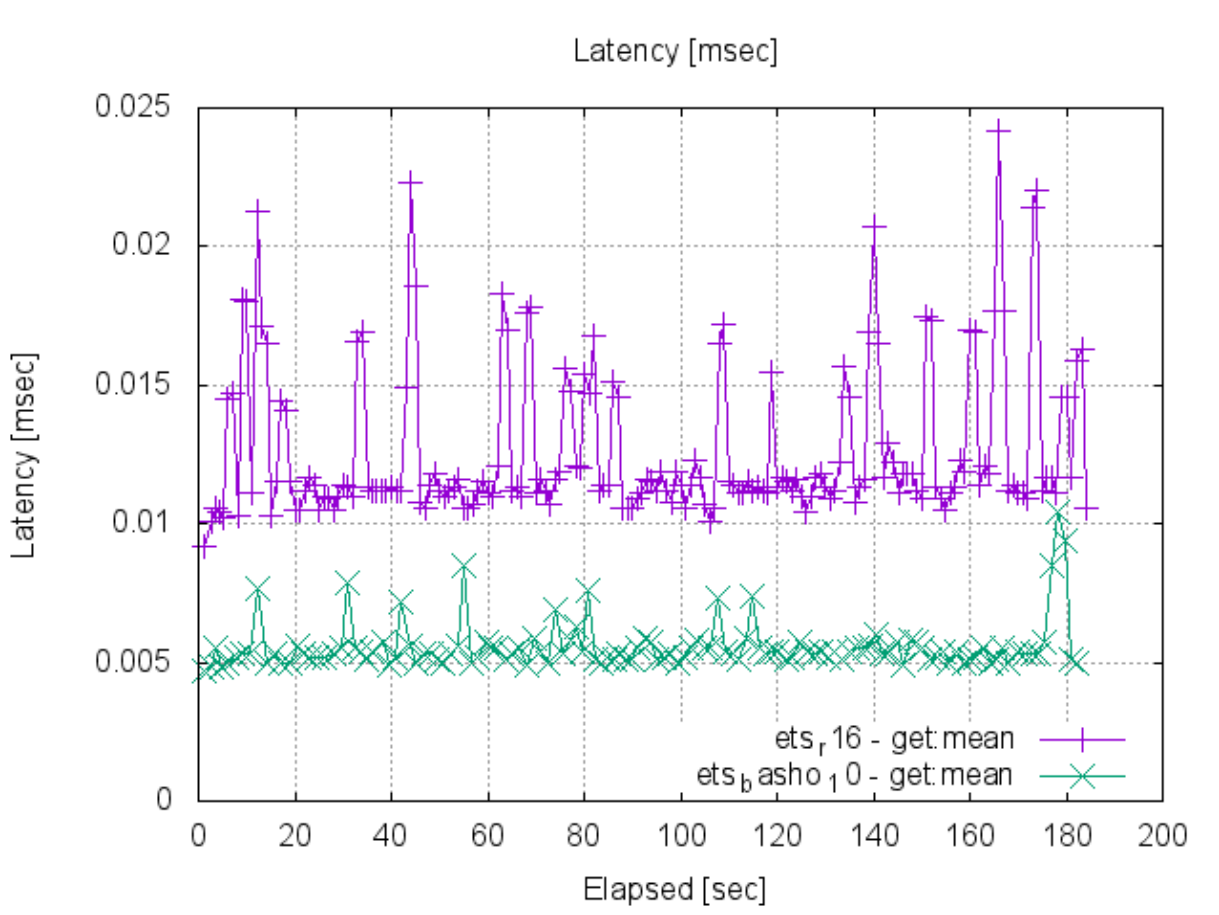
- R16 ETS vs. 17 ETS vs 18 ETS?
- With and without fine locking (on 17)

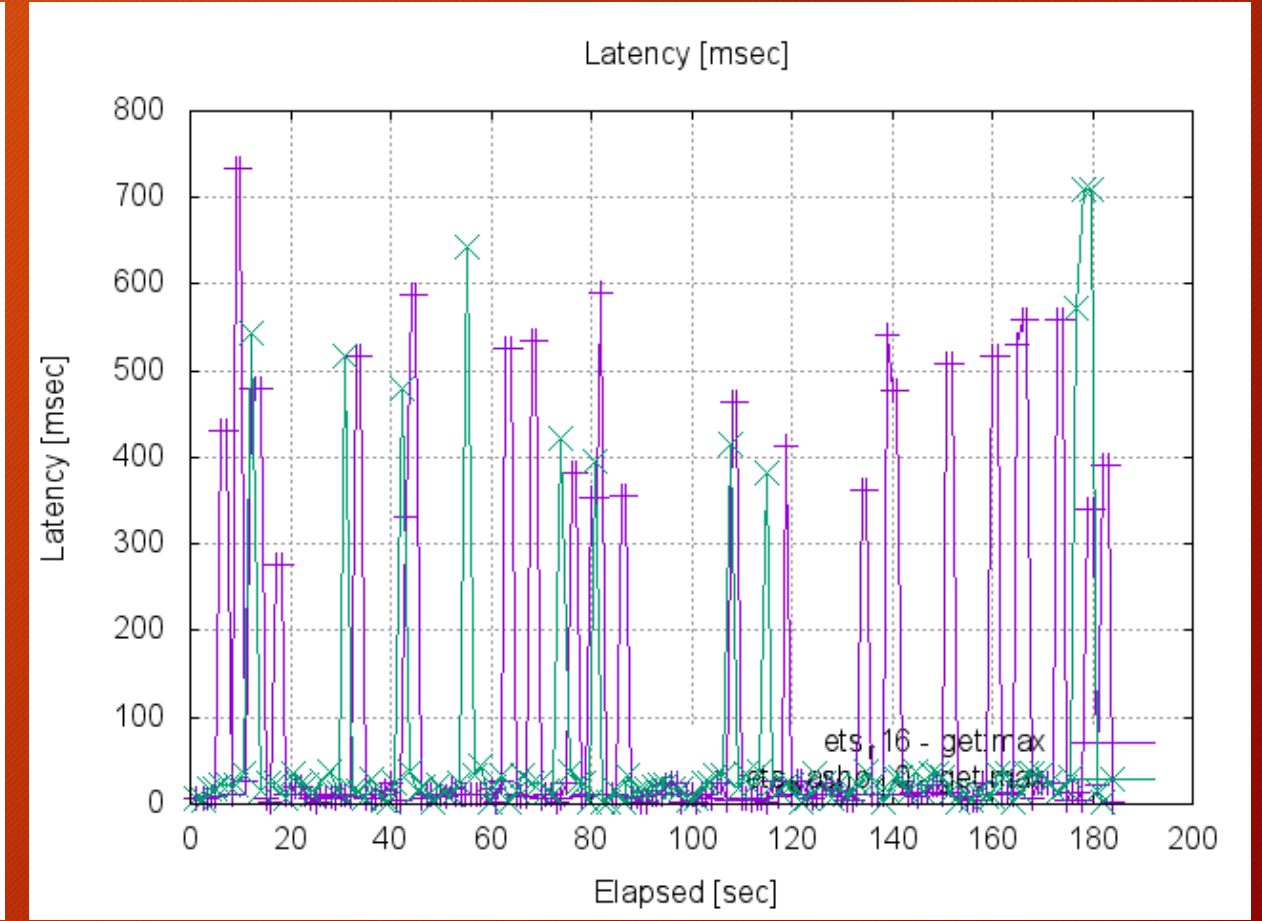
ETS Battle Royale Results



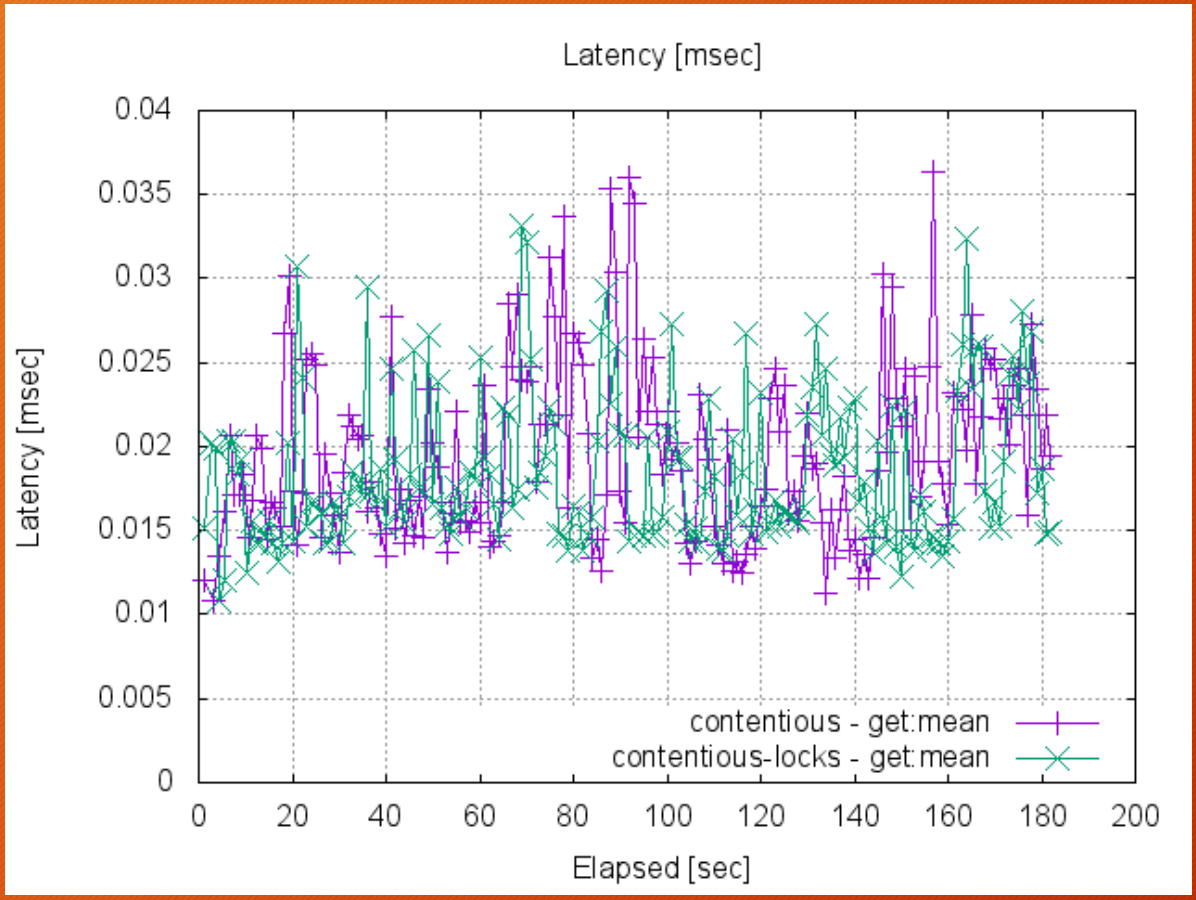


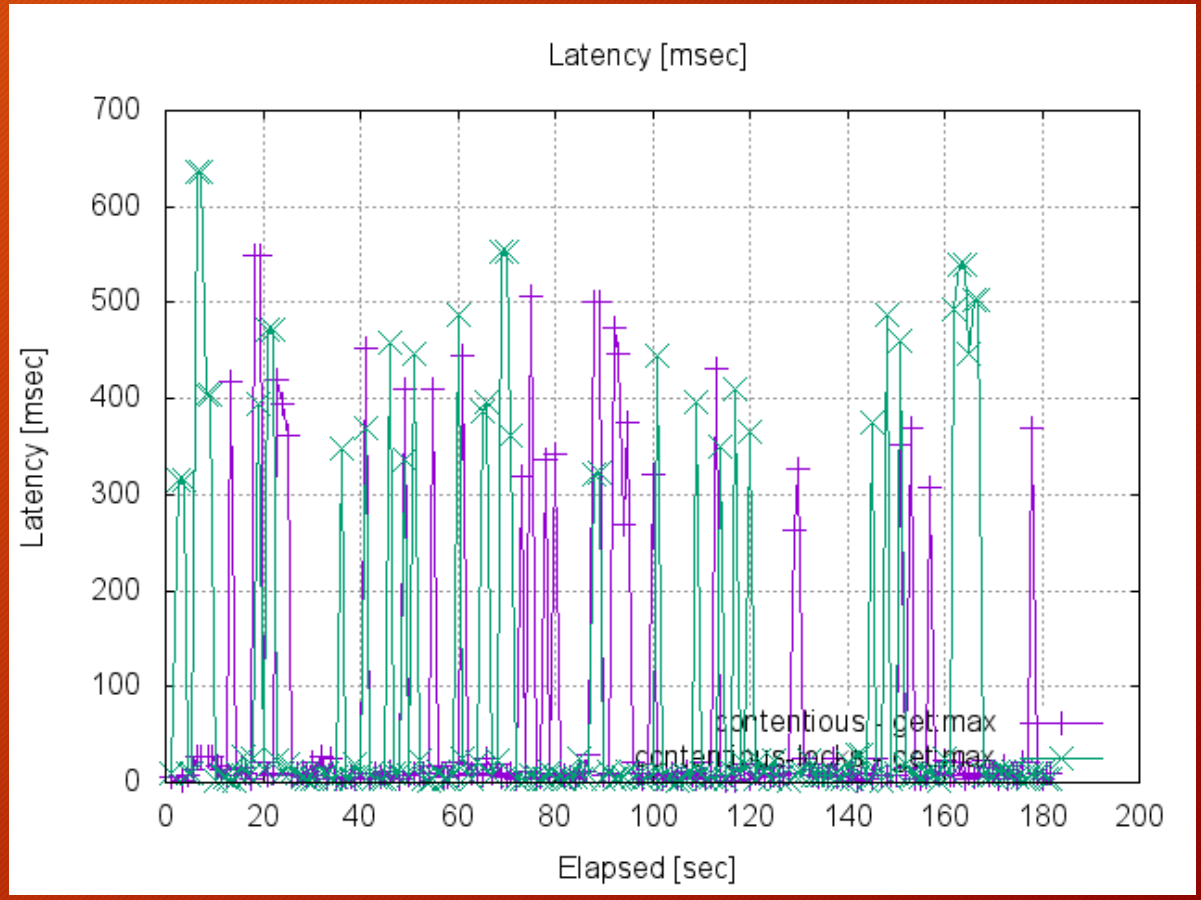
Stock OTP 16 vs Basho OTP 16





With and without fine grained locks (OTP 17)





ETS is hard to beat, but…

Remember what we are measuring versus what we observed in production and want to minimize.

How can we approach measuring our changes in a “real world” scenario?

Measuring Erlang performance “in situ”

- https://github.com/slfritchie/eflame
- https://github.com/engelsanchez/riak_perf_analysis
- https://www.youtube.com/watch?v=4Si-7nAic2c (Scott Fritchie dtrace.conf 2012)
- https://www.youtube.com/watch?v=rECtTWbi2Zc (Erlang Factory 2014)
- Limitations on observability in the Erlang Run Time system
- Most of the time, these tools help you **rule out** root causes rather than identify them.

Getting the most out of ETS

- Always use `{read_concurrency, true}, {write_concurrency, true}` when creating your ETS tables unless you shouldn't.
- More recent ETS releases are faster than older ETS releases, generally. (If you **can** upgrade OTP releases, you probably should.)
- Never do something because it was in a talk or on Hacker News /reddit/teh Internetz.

You need to benchmark your own particular situation to make the best decisions!

Resources / links

- https://github.com/mrallen1/ef2016
- https://github.com/basho-labs/fling
- https://github.com/basho/basho_bench
- https://github.com/leo-project/cherly
- https://github.com/arekinath/e2qc
- http://www.vldb.org/conf/1994/P439.PDF
- Eliminating Single Process Bottlenecks with ETS Concurrency Patterns (Jay Nelson - Erlang Factory SF 2014)

Thank you!

Questions?