# CS764 Project: Adventures in Moodle Performance Analysis

Brian Kroth

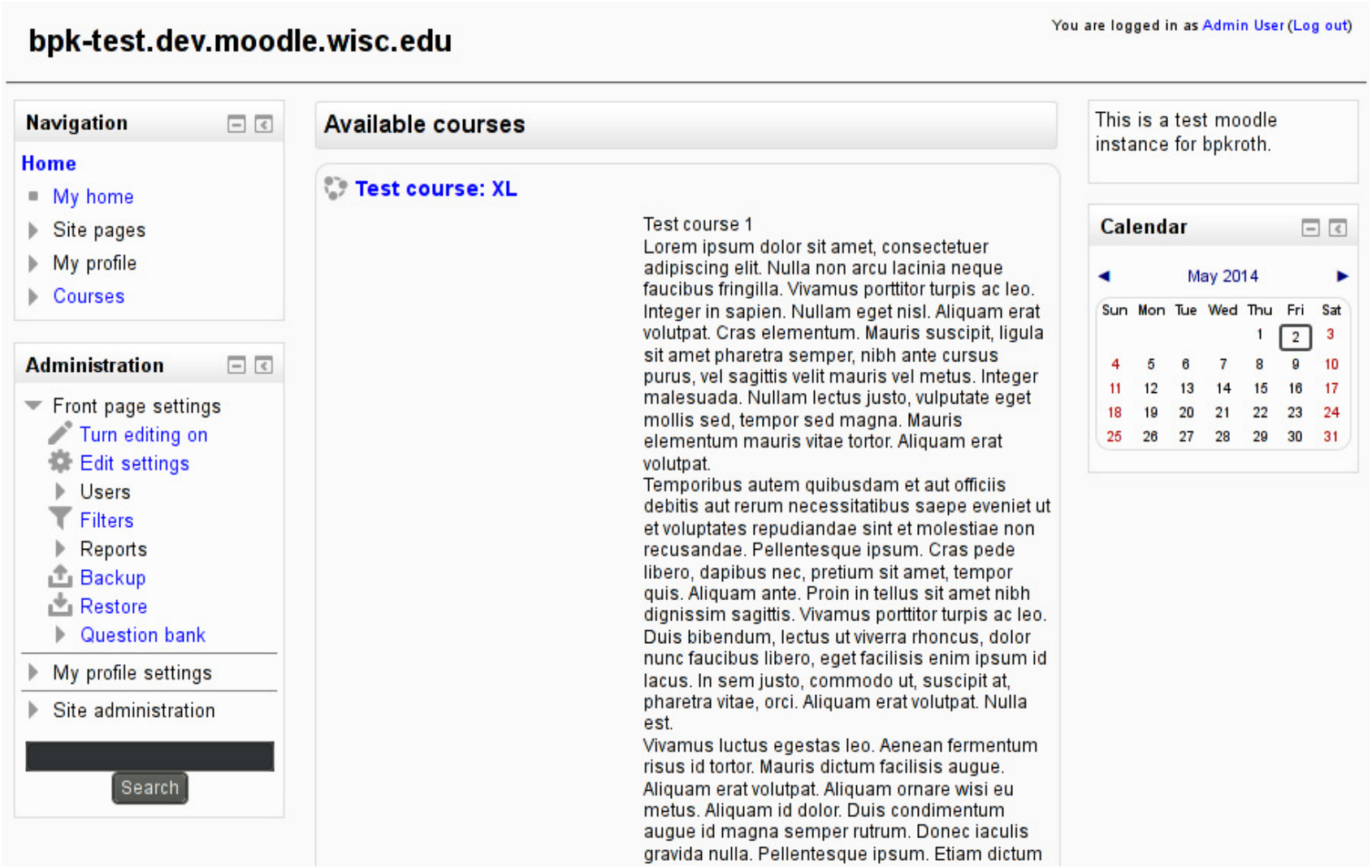
2014-05-07

# Background

## Basic Moodle Description

- Moodle is an open source learning management system (LMS) that's in use as a UW-Madison campus wide service.
- Basically a LAMP webapp.
- Hosted at College of Engineering using a form of 3-tier web setup (part of a much larger general vhosting system).
  http(s) proxy $\rightarrow$ backend application server(s) $\rightarrow$ DB/storage

# Basic Moodle Screenshot



# The Problems

## Cache Data Problem Motivation

- Fairly CPU intensive, so needs to cache processed data.
- Not originally built with multiple backends in mind.
- Expects a shared cache for certain things (probably fixable), but right now local cache has consistency issues.
- The default shared NFS cache is almost as slow as not caching!
  Example logged in front page `xhprof` profile results:

| Cache Mode | Queries | Query Time | Total Time |
|---|---|---|---|
| "Local Global" Cache | 42 | 31 ms | 219 ms |
| `NFS (flock cto)` | 43 | 32 ms | 504 ms |
| `CACHE_DISABLE_ALL` | 85 | 80 ms | 610 ms |

- Would like this cache to survive a restart to avoid the need to rebuild it.
  For some cached data (eg: filtered text) it can take 9.8 seconds to rebuild on demand, only 1.5 seconds of which is spent in database or file IO calls (15%). 1.9 seconds spent just in `preg_match` calls. That's a very long page load for a user!

# The Problems

## Session Data Problem Motivation

- Logged in user's session data exceeds standard cookie size (abused for cache as well), so must be stored in a shared location on the servers.
  eg: DB (shared NFS is too slow due to cache coherency overhead)
- Session data is very write intensive.
  Read and written back on every page load.
  Means waiting on a flush to disk before returning page content to the user.
  1st among all tables for total write waits time (2nd and 3rd place wait 3.5x and 7.5x less respectively).
- But does not have *very* strict persistence requirements.
  Would like it to survive a restart so that users who were logged in don't get logged out, but if we miss a few seconds of updates, it's probably ok.

## The Solutions?

Moodle has some recent (v2.6) support for alternative storage for cache (eg: Memcached, MongoDB) and session data (eg: Memcached).

## Qualitative Constraints

- Shared hosting environment.
  Firewalls insufficient. What about authentication?
- How do things react when services restart?
  Some degree of data persistence for certain data?

## The Goals

Two basic issues we want to explore:

1. What is the "best" cache storage mechanism to use?
2. What is the "best" session storage mechanism to use?

We'll use quantitative response time as our measure of "best" in this talk.

## Test Setup (Part 1)

To test each alternative data storage system, we setup an independent basic Moodle install (v2.6) to mimic the smaller campus instance (`innovate.moodle.wisc.edu`):

- Separate MySQL 5.6 Master w/ 12G of RAM, 4 vCPUs, 4 virtual disks (OS, db data, InnoDB logs, binary logs).
- MySQL replication slave (for backups)
- Separate Apache 2.2.22 Proxy w/ 4 vCPUs and 4G of RAM.
- 2 PHP 5.4 Apache backends w/ 6G of RAM and 4 vCPUs.
- Created 3 Memcached servers w/ 2G and 2 vCPUs (only 1 used for some tests)
- Created MongoDB 2.4 server with 4G RAM, 2 vCPUs, 3 virtual disks (OS, db data, journal)

In all cases, the active data fits in memory (minimal read IO).

# Brief Technology Intro

## Memcached

- By default an in memory Key Value store.
- Supposed to be able to support SASL authentication ...
- But no access control.
  Any client can `FLUSH` the cache, so need many separate instances.
- There are implementations that can make it semi-persistent.
  eg: MySQL, MemcacheDB, Couchbase, etc.

# Brief Technology Intro continued

## MongoDB

- JSON Document Store.
- Persistent to disk.
- Depends on OS for in memory cache.
- Per-transaction "Write Concerns".
  Like having "Degrees of Durability" ...
- Supports authentication and access control across separate DBs (collections).

  Possibly nice for shared hosting environments.

## Test Method

- Run a variation of `moodle-performance-comparison` at different concurrency levels ($XS = 1\,\text{user}$, $S = 30\,\text{users}$, $M = 100\,\text{users}$) and a target throughput of $20\,\text{requests/sec}$.
  With higher levels of concurrency, Java encounters OOM issues.
- Mostly just a wrapper around `JMeter`.
- Parses Moodle debug output to report on things like # of DB queries, session size, etc. in addition to request latency.
- It also assembles output for comparison via a PHP webapp and has scripts meant to help reset DB state for accurate comparisons.
- Test involves simulating a user logging into the site, accessing a course, viewing and posting to a forum, and logging out.
- There's a warm up period prior to results collection.
- Test runs several iterations.

# Outline

# Outline

# Outline

# Cache Data Test Environments

Tested several different configurations of

- NFS - flock cto, noflock cto, noflock nocto
  - Moodle by default uses shared file locks on file based caches, but that flushes the Linux NFS page cache!
  - Close-to-open affects whether the client checks back with the server.
  - Disabling CTO can lead to stale results returned (limited by NFS stat cache TTL).
- Memcached - standard, cluster, compression, igbinary serializer, both
  - igbinary is an alternative PHP data structure serializer.
  - In simple tests on the Moodle cache files it results in 50% less space (important for network storage) and 37% less CPU time in unserializing.

Also

- MongoDB - single node, with authentication
- Local FS

# Bugs Filed

## Missing Memcached SASL Support

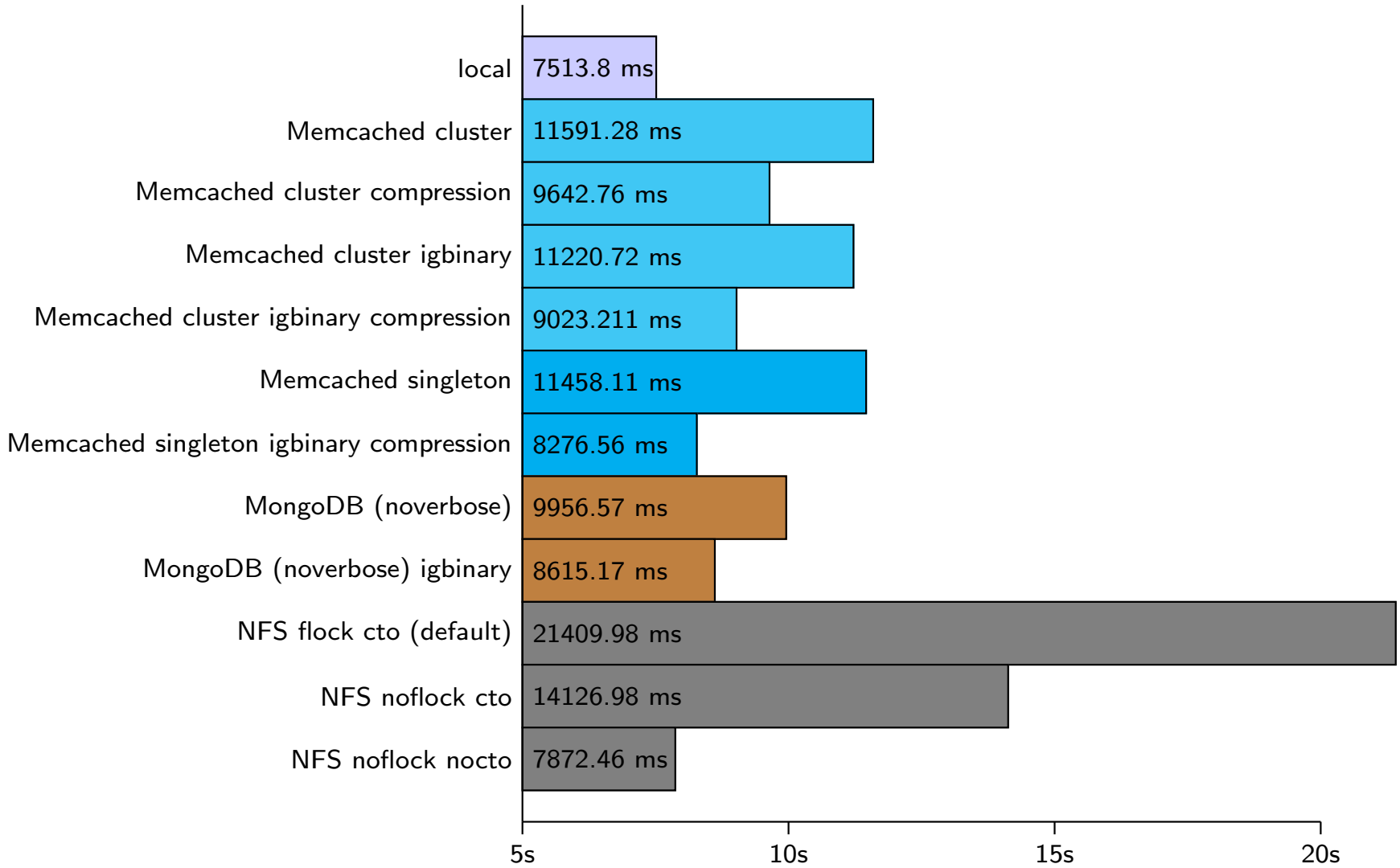
Debian Bug #724872: 2 line Makefile variable misspelling prevents SASL authentication for Memcached, affects all dependent libraries! (eg: PHP driver)

Seems to indicate that very few people actually use SASL authentication with Memcached.

## MySQL Memcached Broken

MySQL Bug #72435: assertion in MySQL InnoDB Memcached encountered with Moodle and `memccapable`, but not `memcslap`.

Attempted to compare MySQL Memcached and standard Memcached performance using various "delayed commit" settings at a number of different concurrency levels, but found very little difference.




Cache Stores - Average Total User Test Latency (ms) - M Course

# Results comments

- Some items left off to keep graph semi-legible.
- Smaller concurrency sizes have similar patterns.
- Memcached singleton variations shows similar patterns to Memcached cluster variations.

# Cache Data Observations

- P90 values are much higher than averages indicating there's still significant variability we haven't found/fixed.
- Local FS cache and NFS noflock nocto cache request latency averages are nearly identical ($< 5\%$ difference) due to very little network accesses.
- NFS cto noflock still requires checking in with the NFS server, which increases latency by 79%.
  Only partial (53%) savings in just not flushing local page cache (noflock).
- Using igbinary with NFS noflock nocto is actually worse! CPU overhead no longer worth it when there's no network in the way.

# Cache Data Observations continued

- Disable verbose mode for MongoDB! Resulted in increase in syslog traffic (47%) and slowed down responses (23%).
- MongoDB (noverbose) had nearly the same request latency averages ($< 1\%$ difference) as Memcached w/ compression.
- Memcached (cluster or singleton) with igbinary reduces latency slightly due to decreased network at the cost of higher CPU overhead.
- Memcached (cluster or singleton) shows greater latency reduction (due to network reduection) for compression.
- Memcached using both decreased request latency averages by roughly 20% due to decreased network usage (roughly 50%).
- Memcached cluster slightly worse than singleton, perhaps due to extra network connections.

# Possible Improvements

- MongoDB (noverbose) used the same network traffic as standard Memcached.
- No network compression for MongoDB. Just BSON encoding.
- Tried to have it use igbinary, but our PHP extension wouldn't support non-UTF-8 strings.
- FIXED: Results in performance somewhere in between singleton Memcached and cluster Memcached with data persistence!

# Outline

# Session Data Problem Recap

- Logged in user session data needs to be stored in a shared location (eg: DB).
- Lots of activity. Read and written on every page load.
- Doesn't necessarily need strict persistence requirements. We could lose a few seconds of updates and it'd probably be okay.

# Session Writes Problem

For tables that are accessed frequently, `mdl_sessions` table is:

(per `performance_schema.table_io_waits_summary_by_table` analysis)

- 1st among all tables for total write waits time (2nd and 3rd place wait 3.5x and 7.5x less respectively).
- 11th for average write waits times (not *that* slow on average).
- 4th for total read waits time.
- 11th for average read wait times (reads are generally fairly quick, though don't get to make use of query cache due to writes invalidating it).
- 4th for total overall waits time.
- 2nd for average overall wait times.
- Aside: the `mdl_cache_text` table, which is handled outside of the regular MUC caching systems described earlier, is another top contender that doesn't require strict persistence.

# Session Writes Problem continued

- Databases are replicated for backup purposes.
- Two aspects to writes in a MySQL database: InnoDB logs, Binary (replication) logs.
- Both flushed at commit for consistency.
- Server wide, not per transaction, setting (`binlog_sync=1`, `innodb_flush_log_at_trx_commit=1`).
- In one sample window `mdl_session` table data made up 46% of all binary log data at and 27% of all binary log events (per `mysqlbinlog` analysis).
- 55% of all time spent waiting for a production Moodle instance using a DB as the session data store are for InnoDB logs and binary logs (per `performance_schema` analysis).
- So, would expect an improvement of at least 15-25% by diverting that activity somewhere that didn't have to wait on a disk flush immediately. Perhaps more simply due to reduced resource contention.
- Aside: the `mdl_cache_text` table makes up 45% of all binlog events, but only 1% of the binlog data. It takes second place for write waits.

# Session Data Thoughts

## Session Data Thoughts

- Idea: Write back the session in the background.

  (ie: don't block page response)

  Already being done?

  (PHP session `write()` handler is called after output stream is closed)

  Problem: Hides errors? Yes.
- Since session data is important to user experience, would like the data to at least be semi-persistent.

# General Thoughts

## Degrees of Durability

- It is convenient for developers to have a single place to manage data for an application (eg: SQL DB).
- But not all data has the same requirements.
- Need a way to declare those different requirements to the system.

## Degrees of Durability Proposal

- `SET TRANSACTION DURABILITY = [STRICT|BACKGROUND|LAZY]`
- Still use WAL protocol.
- Basically don't wait on or even skip WAL log *flush* at `COMMIT`.
- If a background thread or another transaction forces log (eg: due to `COMMIT`), then our transaction will have been made durable.
- Allows larger Group Commit.

## Potential Problems

- What about distributed transactions or replication?

  eg: slave reads from in memory replication log instead of what's actually on disk and then master crashes

- Do we need a "prepared, but not committed" state?
- MSSQL 2014 has already implemented this feature!

  `COMMIT TRANSACTION WITH DELAYED_DURABILITY=ON`

# Session Data Tests

## Session Data Drivers Explored

- Memcached (for comparison, but has access control and persistence issues)
- NFS (for comparison) with flock cto (since it needs consistency)
- Wrote a MongoDB session driver (optionally with reduced Write Concern level)
- Wrote a separate-db session driver (extra network connection)
- Used with separate MySQL server with reduced durability levels, no replication.
- ~~Same MySQL server, but with replication disabled for the `mdl_sessions` table transactions (potential slave sync issues)~~
  Not practical since `SET sql_log_bin=1` requires `SUPER` privileges to the DB server.

# Session Data Driver Comments

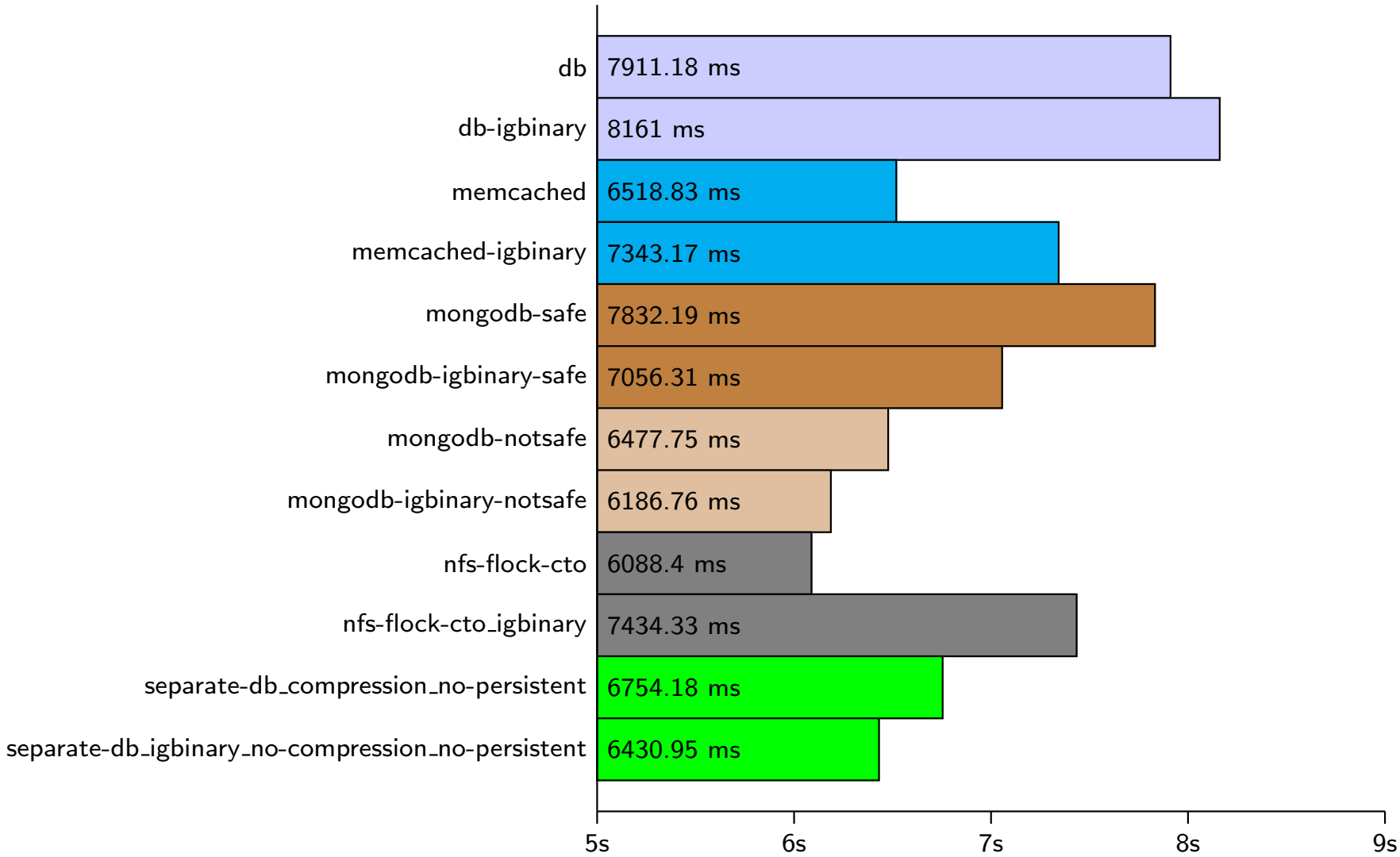
## File and NoSQL Locking

- We use NFSv3 in this case, which has explicit locking protocol.
- But no way in PHP file session handler to specify a timeout.
- NFSv4 uses polling with exponential backoff. Slow!
- Memcached PHP session driver also uses polling to "lock" a key via CAS.
- MongoDB session driver wrote uses polling of separate lock document which depends on atomic insert operation.
- Requires using `safe=true, w=1` writes for those operations, else no response code to check!
- For all of these, lock "canary" may remain in case of client failure.
- Need to periodically remove old locks.
- May lead to bad client behavior.

# Session Data Driver Comments continued

## MySQL Locking

- For MySQL, named locks are released automatically at client connection close (or change user in the case of persistent connections).
- Natively supports timeout.
- Can be made to support early backout when the number of waiters is too high.
- Helps to prevent DoS problems when redirect loop causes too many PHP processes to wait on the same MySQL session lock.




Session Stores - Average Total User Test Latency (ms) - M Course

# Session Data Comments

- Default db tests include network compression.
- Memcached PHP driver also does dynamic compression (above a threshold).
- For session data, igbinary + compression is too much CPU overhead for little network traffic reduction gains.
- Smaller concurrency sizes have similar patterns.

# Session Data Observations

- For all "db"-like systems, relaxing durability (eg: safe $\rightarrow$ notsafe, db $\rightarrow$ separate-db) reduces latency.
- Similarly, using no-durability (Memcached) also improves performance, though not much better.
- Cost becomes more in the network and CPU.
- Surprisingly, NFS (v3), even with flock and cto, performs very well.
- Guessing due to proxy matching clients to the same backend so nodes don't fight over locks.
- MongoDB does no network compression, so it benefits from igbinary serialization improvements.
- MySQL 5.6 `ROW_FORMAT=DYNAMIC` may have better storage of igbinary BLOBs (1073 bytes on average) than standard PHP serialized TEXT (1432 bytes on average) (hard to tell) .

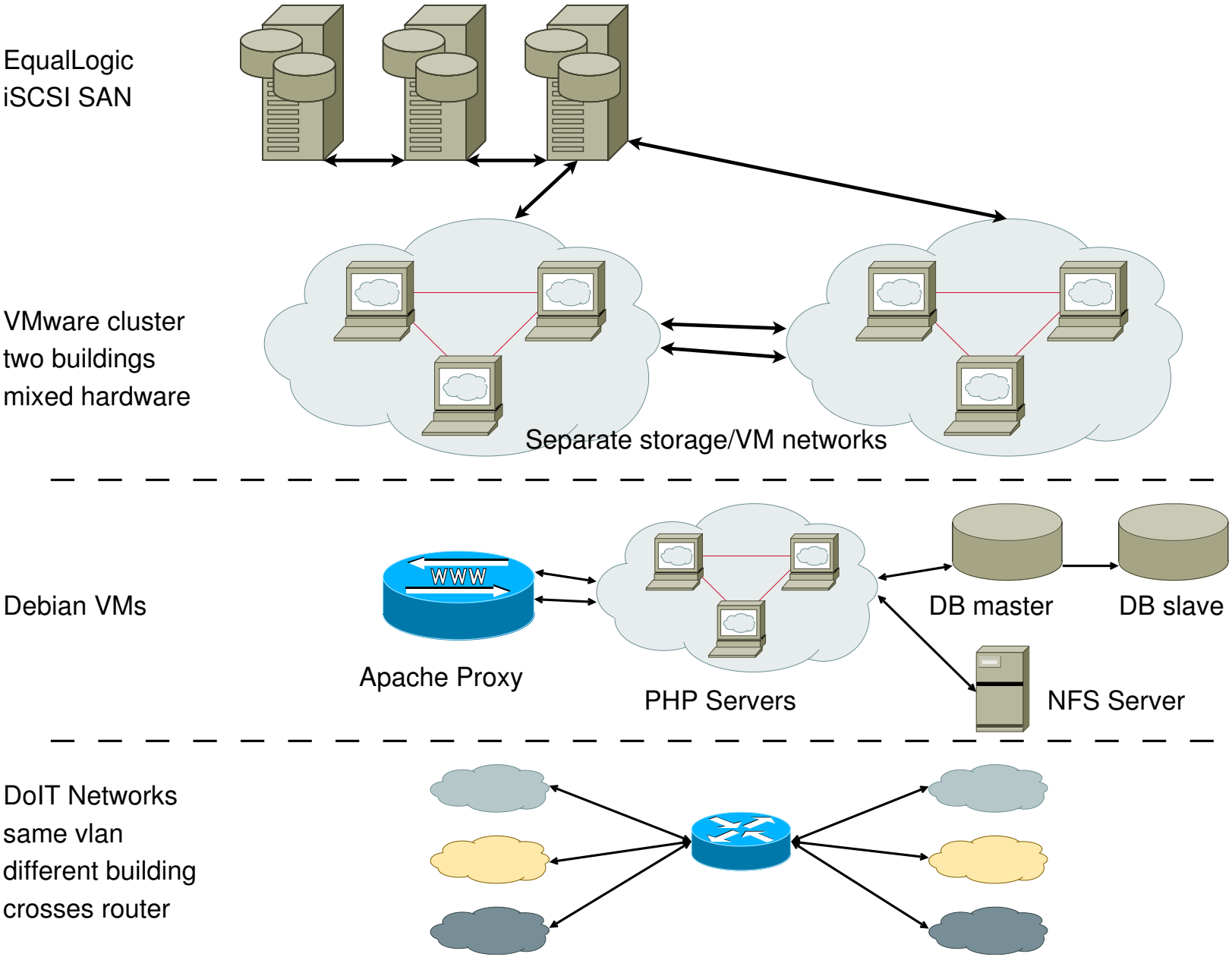
# Outline

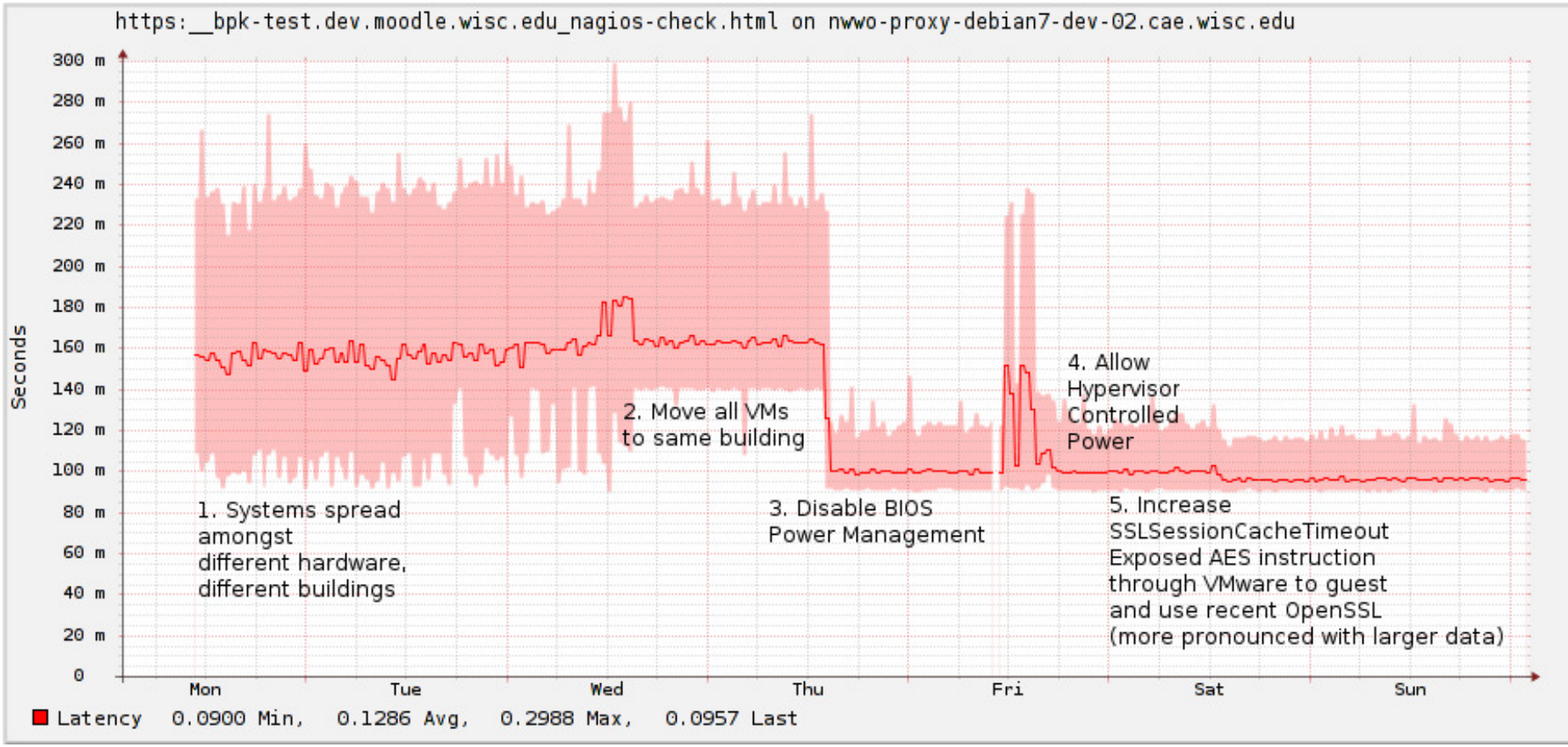
# Systems Discoveries

## Troubles with reliable measurements

- Variability wreaks havoc on measurements.
- Shows up in a number of different and interesting ways.
  Network, Power Management, Load Balancing, Cache Management, HW exposure, etc.

Next, a brief systems overview.


EqualLogic
iSCSI SAN
VMware cluster
two buildings
mixed hardware
Separate storage/VM networks
Debian VMs
WWW
Apache Proxy
PHP Servers
DB master
DB slave
NFS Server
DoIT Networks
same vlan
different building
crosses router


https:__bpk-test.dev.moodle.wisc.edu_nagios-check.html on nwwo-proxy-debian7-dev-02.cae.wisc.edu
Seconds
300 m
280 m
260 m
240 m
220 m
200 m
180 m
160 m
140 m
120 m
100 m
80 m
60 m
40 m
20 m
0
Mon
Tue
Wed
Thu
Fri
Sat
Sun
1. Systems spread amongst different hardware, different buildings
2. Move all VMs to same building
3. Disable BIOS Power Management
4. Allow Hypervisor Controlled Power
5. Increase SSLSessionCacheTimeout Exposed AES instruction through VMware to guest and use recent OpenSSL (more pronounced with larger data)
Latency 0.0900 Min, 0.1286 Avg, 0.2988 Max, 0.0957 Last

# Systems Improvement Lessons

- Crossing a router (a shared resource) for same vlan (across buildings) traffic makes network much more unpredictable.
  Roughly 40 ms drop in variability when we move all VMs to two hosts in the same room.
- Dynamic migration to balance VM load during load testing of typically idle machines makes things unpredictable. Disable it or segregate systems for testing.
- Disable power management, or at least place it in OS control (it has more insight into upcoming workload).
  Resulted in roughly 50 ms drop in variability and average latency.

Response times for a simple (37 bytes) static HTML (no PHP) page request over SSL at different BIOS power modes:

| PM Mode | Avg Watts | Min Latency | Max Latency | Avg Latency |
|---|---|---|---|---|
| BIOS managed | 185 W | 140 ms | 220 ms | 162 ms |
| No throttling | 232 W | 90 ms | 120 ms | 99 ms |
| OS Managed | 190 W | 99 ms | 132 ms | 109 ms |

# Systems Improvement Lessons continued

- For iSCSI SANs use MPIO instead of LACP. It allows multiple TCP endpoints (eg: different arrays), so you get higher cache, buffer, and spindle utilization.
  Resulted in an average of 77.6% improvement in IOPS `iozone` results and 54.8% improvement in throughput `iozone` results.
- Need to expose the extra I/O paths to VMs by placing disks on separate vSCSI controllers.
  Doing so cut the combined per operation average read latency from 15.1 ms to 4.4 ms to on MySQL servers log volumes.
- Even on local networks it often makes sense to trade CPU for decreased network traffic.
  Enabling MySQL connection compression reduces page latency by 27.4%.

# Systems Improvement Lessons continued

- Need to expose underlying hardware capabilities through VM layer (eg: VMware EVC Mode).
  Exposing the `AES-NI` instruction and using a recent version of `OpenSSL` resulted in over 5x faster SSL operations as reported by `openssl speed`.
- Increase `SSLSessionCacheTimeout` to allow greater hit rate on reconnecting (not `KeepAlive`) clients in order avoid SSL Renegotiation overhead ($30 - 80\,\mathrm{ms}$).
- Increase TTL on `mdl_cache_text` data in order to avoid semi-random and *very* lengthy ($6 - 10\,\mathrm{s}$) cache regen events.
  Alternatively, fix the Moodle code to use deterministic cache invalidation mechanism. See Also: MDL-43524

# Outline

# Related Work

## Related Work

- Not much in the way of serious Moodle performance analysis available.
- *J. Coelho and V. Rocio. A study on moodles performance. 2008.*
  Looked at MySQL vs. PostgreSQL and Windows vs. Linux for a much earlier version of Moodle (1.9)
- *K. A. Bakar, M. H. M. Shaharill, and M. Ahmed. Performance evaluation of a clustered memcache. 2010.*
  General analysis of the impact of clustered memcached on a typical webapp to alleviate DB load. Also shows an increase in overhead when using a cluster. No consideration of persistence or cold cache or failure analysis. Also, our DB load for reads typically isn't that high.
- Also little on MongoDB when used for a cache.
- Lots of other ways to load test.
  eg: `ab`, `tsung`, `httperf`, `faban`, etc.
- But different from benchmarks, which don't necessarily represent your application. Some possibly applicable ones: TCP-W, TCSB, NoBench, etc.
- Lots of others left off ...

# Outline

# Conclusions

## Summary

- Investigated different cache and session configurations for Moodle.
- And several sources of variability.

## Conclusions

- Hardware and network organization details matter, even in a virtual environment.
- As expected, network delay seems to be biggest source of performance differences.
- Compression and encoding really matter here.
- Different data has different needs. Informing the system about that probably useful.

# Questions?

Questions?