

CS764 Project Report: Adventures in Moodle Performance Analysis

Brian Kroth
bpkroth@cs.wisc.edu

2014-05-16

Abstract

The Computer Aided Engineering center at the College of Engineering of UW-Madison manages a platform of web, database, storage, and other related servers for a campus wide Moodle service, an online learning management system. Due to the large scope and supported user base of this web application, it encounters some interesting systems and database related problems. The main focus of this project was to evaluate and analyze some of the different database and related (e.g. NoSQL) configurations for application cache and user session data that Moodle currently supports (or can be made to) with respect to their impact on scalability, reliability, security, and performance. We found that systems with relaxed durability, something we call “degrees of durability”, can perform as well as systems with no durability as the cost of network communication tends to outweigh the possible cost of background disk activity. Along the way we found some ways to help reduce the network costs and also found a number of general systems performance anomalies and optimizations which we discuss.

1 Introduction

The use and diversity of online learning management systems (LMS) at schools at all levels has increased dramatically in the last decade [28, 44]. On the one hand, students are increasingly demanding material on demand and on the go or from otherwise remote locations to meet their convenience and needs. Additionally, they may come from different backgrounds with different language requirements, necessitating content localization. On the other hand, educators are looking for more innovative ways to engage their students in a variety of disciplines. For instance, instructors may wish to include formula and other sorts of possibly interactive markup in their course content, all of which needs processing prior to delivery to the end user. This in combination with widely distributed open source nature of LMS projects such as Moodle, has led to developers constructing some fairly resource intensive solutions.

As such, Moodle, like many web applications, has to make extensive use of caching in order to achieve adequate performance. To do this, a core caching infrastructure layer called Moodle Universal Cache (MUC) was introduced in version 2.4 so that individual plugin developers needn’t individually implement it themselves. Unfortunately, the system didn’t originally have web clustered environments in mind and guaranteed a shared cache to developers.

By default, this cache is directed at a shared file system, such as NFS. However, profiling results of a simple front page request show that this is nearly as bad as not caching at all. As of Moodle 2.6, there are several other alternative caching stores such as Memcached and MongoDB that we explore in addition to modifications on the simple NFS caching scheme.

Separately, users expect an interactive experience from web applications like Moodle. However, the HTTP protocol is stateless, therefore Moodle needs to save session data in shared location, such as a database. As this data needs to be read in on each page load, and changes written out on each page unload, the data becomes a hotspot in the system. Instrumentation analysis of our large Moodle site’s database showed the session data table to be the number one reason for waits in the system, by factors of 3.5x and 7x when respectively compared with the second and third most commonly waited for resources. However, this data does not have the same durability requirements that the rest of the database has. Therefore, we would like

to find a way to offload the data to another system such as Memcached, MongoDB, or a separate database tuned for relaxed durability.

In the remainder of this paper we give some background on the Moodle system we have implemented and further details on the cache and session data problems in Section 2. In Section 3 we describe our evaluation strategy. Section 4 discusses our results as well as a number of general systems discoveries we made along the way. Finally, we discuss some related works and our thoughts on the need of systems with “degrees of durability” in Section 5 and conclude in Section 6.

2 Background

In this section we begin with a brief overview of our system’s architecture and provide some more details motivating our investigation into the cache and session data problems.

2.1 System Overview

The current UW-Madison campus Moodle service server platform, as hosted at the Computer Aided Engineering center in the College of Engineering, is based off of a three-tier distributed LAMP stack, and is a subtype of a larger general purpose website server platform that currently handles over 900 vhosts of various flavors (e.g. HTML only, PHP5, prepackaged Wordpress, etc.). Our CS787 project proposal document [38] has more details on the general system. For the purposes of this paper, it largely resembles the architecture of a scalable MOOC as described in [50], except that it is currently all homegrown and running on a local VMware virtualization infrastructure¹ rather than using any public cloud services.

The Moodle portion of the service currently consists of two production virtual web sites (vhosts), both currently running a locally patched Moodle 2.4, as well as a number of development and testing vhosts. Clients communicate with the vhosts through a pair of frontend Apache servers running `mod_proxy_balance` to distribute client requests to a pool of backend Apache `mod_php5` application servers that are specifically tuned for Moodle vhosts.² The frontend servers also optionally run `mod_disk_cache` in order to save and later serve certain static responses directly to clients without contacting the backends.

Each backend server has a local copy of the Moodle PHP code base, synchronized with an authoritative NFS server store using a homegrown two-phase commit `rsync` based protocol, in order to avoid the overhead of `stat` calls for code that doesn’t often change during runtime as well as provide relatively atomic code updates between all of the backends when code does need to change so that clients don’t receive different and inconsistent responses.³

For each site, dynamic user data such as PDF documents and quiz question banks are split between an NFS

¹ We like to refer to it as a “fog”, since it’s like a cloud, just closer. It currently consists of 15 physical hosts of slightly different Intel hardware epochs (e.g. Westmere vs. Sandybridge) and configurations. Roughly speaking they each have a pair of 10 Gb network interfaces for iSCSI storage and VMotion, a pair of 1 Gb network interfaces for public network trunks, at least 128 – 256 GB of RAM, and 12 – 16 cores between 2 CPU sockets running at 2.4 – 2.8 GHz. The hosts are spread between two buildings which have a private 20 Gbit link between them for storage and VMotion traffic, though primary storage is only in one building.

² At the time of this writing the main campus Moodle site, `courses.moodle.wisc.edu`, has five backend VMs assigned to it, each with 4 vCPUs and 6 GB of RAM, though that number may change throughout the year according to load demands in a manner originally developed in our CS787 project [37] though also resembles that of [35]. The other production site, `innovate.moodle.wisc.edu`, currently has two backend VMs assigned to it. The backend assignment systems attempts to avoid overlapping “large” vhosts whenever possible to avoid vhosts competing for VM resources. We depend upon VMware DRS to try and isolate individual VMs from competing for physical resources, though some of that is unfortunately unavoidable, since, for instance, we do not have entirely separate storage systems.

³ The `stat` calls are a function of the PHP opcode cache subsystem (APC) and can be disabled at the price of potentially inconsistent responses due to lack of awareness of code base changes, so it is therefore usually not done. Instead, the two-phased *nearly* atomic update of local code we developed alleviates the inconsistent responses problem. Additionally, once the code files are local, the APC `stat` calls tend to hit the OS inode cache, which has a much lower overhead when compared with an NFS call since there’s no network requests involved.

store and a MySQL 5.6 database server, respectively. There is a separate ⁴ database server VM for each of the two production sites, as well as a third for the development sites. ⁵ The MySQL servers each have binary logging enabled in order to asynchronously replicate changes to their own slave server so that consistent SQL level backups (i.e. not block level) can be taken of the database without locking all tables or interrupting service. ⁶ No reads are currently directed at the slave database server. Data is stored in InnoDB storage engine tables for ACID compliance. The systems are tuned and sized such that the typical active data set fits entirely in memory, so that actual disk read operations are only involved when long reports or application level (i.e. higher than SQL level) backups are performed.

We note that although the exact mechanisms we employ may differ from other implementations, our system shares the general design features of most other large installs that can be found on sites like the Moodle forums. ⁷

2.2 Cache Data

Like many web applications ⁸, Moodle makes heavy use of caching to attempt to improve performance. Moodle’s Universal Cache (MUC) [31] system originally separated this cache into two distinct pieces:

- User specific session cache, and
- Module specific application cache that applies to all users (e.g. JavaScript, unified CSS, language localized strings, L^AT_EX rendered equation images, etc.). ⁹

In earlier versions of Moodle, by default, these caches were stored in a file system hierarchy local to the given backend server. We would expect this to perform well, even optimally, given it involves no network I/O and typically the cache file content would remain within the OS page cache, so it would also suffer no disk I/O. The MUC attempts to make sure that cache files, generally identified by a hash of the key, that are missing are regenerated as necessary, so that inconsistencies do not arise.

However, as Moodle is a distributed development effort, not all of it is necessarily built with multiple backend servers in mind, so inconsistencies and program errors ¹⁰ do still occur. For instance, the use of a local file system cache has poor cache coherency since the MUC does not currently handle updating cache files on a local backend when another backend has updated the original source content that the cache file was derived from. ¹¹ Thus, clients may receive stale content until the out of sync backend’s local cache is manually purged and rebuilt.

⁴ We originally used a single database server for both production sites, but found that it was easier, at least at the VM level, to determine and isolate performance characteristics between the two sites when they used separate databases, similar to what we do for the vhost to backend VM assignment.

⁵ Each database server VM’s resources are tuned to the dataset size of the particular site that it serves. Currently the two production servers have 4 vCPUs, 16 GB of RAM each with three virtual disks for OS, data, and logging volumes respectively. The virtual disks are housed on separate VMware VMFS iSCSI LUNs backed by 6 EqualLogic iSCSI storage arrays. Since all of the arrays have large battery backed NVRAM, average write operation latency for the whole systems is generally 1 – 2 ms.

⁶ Mixed student and instructor usage means that the service does not follow a typical diurnal pattern.

⁷ <https://moodle.org/forums/>

⁸ Indeed most things in computer science.

⁹ It should be noted that as each module can register its own caching needs with the MUC, we can actually specify different cache stores for each of the components now. For instance, one might be able to direct language localization string caches to the local filesystem while directing question definitions at a Memcached definition. However, aside from performance implications, there may be correctness issues in doing so, and due to time constraints, in our work we only explored changing the default global application cache store used by all modules.

¹⁰ MDL-40569 [17] occurred this past semester, for instance.

¹¹ For previous versions of Moodle we had simple local hacks in place to detect such occurrences and purge the caches on all backends simultaneously. We believe the underlying deficiency could be fixed to natively support local filesystem caches, for instance by comparing the cache file’s timestamp against a `timemodified` field on the original source data tuple. This might result in additional load on the database server, but should be minimal, particularly if a single batch request for all relevant cache items can be made since only a single attribute is being retrieved. After all, at least the key of the cached content is already being retrieved from the database. Indeed, upon investigation we found that in some cases the key of the cached content includes a “version” component (usually just a timestamp) to accomplish exactly this. However the amount of development work to expand that to all modules or the underlying caching layer in general was beyond the scope of this project, so we did not explore it further.

Cache Mode	Queries	Query Time	Total CPU Time	Local/Global/Session Cache Hits
“Local Global” Cache	42	31 ms	219 ms	153/72/1
NFS (<code>flock cto</code>)	43	32 ms	504 ms	154/72/3
CACHE_DISABLE_ALL	85	80 ms	610 ms	0/0/0

Table 1: `xhprof` profiling cache results for example logged in front page request.

Therefore, in Moodle 2.6, the application cache described above was forced to a shared cache so that developers need not be concerned with such issues, and a third level of local machine cache was added for content that was deemed safe enough to handle coherency issues like those described.

To get a sense of the amount of time Moodle spends in the caching subsystem we built and used the PHP `xhprof` extension to profile a simple front page load when Moodle was configured with a few different global application cache configurations.

The results, as seen in Table 1, show that if we, for the purposes of comparison, temporarily accept possible cache coherency issues and configure a local filesystem as the “global” cache the system responds over twice as fast as when the system is configured with the default NFS shared cache, despite the global cache only handling about a third of the cache requests.

Moreover, if we disable the MUC cache subsystem entirely, we find that the response latency is nearly three times as bad as when a local filesystem cache is used, despite using only twice as many queries. This should illustrate the amount of CPU overhead involved in page processing as opposed to merely network and database activity.¹² Additionally, note that using a default NFS backed cache is only 17% faster than not caching at all. We explore the reasons for this in Sections 3.3 and 4.1. We also briefly note here, and discuss further in Section 4.3, that for the moment Moodle also does some cache materialization in a database table (`mdl_cache_text`) that’s handled outside of the MUC system for things like filtered text. Some aspects of that system can randomly increase certain page load times to as much as 10s, only 1.5s of which are spent in database calls. Clearly, in both cases, some attention needs to be paid to the cache storage mechanism for the system to perform well.

Note that in both cases, the cache does not require persistence as it can be rebuilt from its original sources as necessary.

2.3 Session Data

The other issue we investigated is that of how Moodle stores and manages session data.

Since HTTP is a stateless protocol, to provide an interactive user experience, and overcome limitations in browser cookie size, the Moodle code currently stores per-user session data as a serialized data structure¹³ in some shared location. The traditional options were an NFS file system directory (one file per session) or in a table (`mdl_sessions`, one row per session) on the main database server. We currently use the database option.

This data, identified by a unique `sid` that’s stored in a cookie in the user’s browser, is read from the session store on every page request receipt, and, if changes are found, written back to the store at page response transmission. As such, it gets a lot of activity.

Unfortunately, things like `lastaccess` timestamps are included in the session data, which results in it getting written out to the session store at the end of every page response.¹⁴ Since our MySQL database system has binary logging for replication and full ACID transactional semantics enabled¹⁵ this means that every

¹² The Moodle code base currently has 6626 PHP pages, 467 of which are included for the simple front page load test.

¹³ Maximum size for a record is currently 4M, though it is usually less than 50K.

¹⁴ We have not yet explored whether or not this can be prevented.

¹⁵ `binlog_sync=1, innodb_flush_log_at_txn_commit=1`

page response incurs at least two disk I/O waits while the InnoDB and then the binary replication logs are each flushed to disk.¹⁶

It is valuable for this data to survive a system restart (e.g. due to security patching or configuration changes) so that users are not all logged out and those in the middle of a task (e.g. a quiz essay question) need not start over from scratch, however the durability requirements on it are not as stringent as other parts of the database. For example, the session data need not even be backed up, let alone replicated. Moreover, losing a few seconds of updates to session data is probably tolerable, unlike submitted quiz data, for example, where we must have strict durability guarantees.

Additionally, since MySQL binary logging cannot currently be disabled for a single table [15], during periods of high activity, the amount of churn on the `mdl_sessions` alone can result in upwards of 5x as much binary logging traffic as there is total database data.¹⁷

By using the `performance_schema` instrumentation available for MySQL 5.6, we’ve found that over the past semester of activity on the main Moodle site, the `mdl_sessions` table is the number one reason for total write waits in the database server by factors of 3.5x and 7x when respectively compared to the number two and three reasons. However, it is not even in the top ten for average write wait times. Similarly, the table is fourth in total read wait times while average read wait times are relatively low. We believe they could still lower, except that the constant write activity to the table invalidates the MySQL query cache results, making that mechanism actually incur extra overhead for each query.

We additionally find that 55% of all time spent waiting in the database system is spent in InnoDB logs and binary replication logs. Of the latter, the `mdl_sessions` table accounts for 46% of all binary logged data in one sample window and 27% of the events logged.¹⁸ Therefore we would expect an improvement of at least 15 – 25% by diverting that data and workload to a system with no or reduced durability guarantees such as Memcached, MongoDB, or a separate specially tuned MySQL database server. Perhaps more, since the main database would experience decreased resource contention and disk activity and better query cache behavior as well.

3 Evaluation

As mentioned previously, our goals in this work were to find the “best” storage mechanisms for cache and session data respectively.

In particular, for cache data stores we evaluated several different configurations of Memcached, MongoDB, and NFS. Given that we expect a local filesystem cache to perform the best, we compared all of them against that, once again temporarily accepting potential cache coherency issues.¹⁹

For session data stores, we compared different configurations of Memcached, MongoDB, NFS, and a separate specially configured MySQL database server against two variations of the standard main database server.

In the rest of this section, we describe our test configurations in more detail and give some background on some of the alternative technologies involved as well as a qualitative assessment of some of the features of each configuration.

¹⁶ At least one other database write occurs for an insert to an `mdl_log` table that holds an application level record (i.e. not just web access logs) of user interactions with the site and the user’s course content (e.g. quizzes). Since this data is valuable both for forensics and educational research purposes, we did not investigate it further during the course of this work. However, we’ll note here that it too might benefit from some notion of “degrees of durability”.

¹⁷ The largest we have seen the `mdl_sessions` table, including its indexes, is roughly 500M. The entire database is roughly 20G, whereas there is currently over 75G of log data for the past day.

¹⁸ We also note that that the `mdl_cache_text` table used to cache filtered text makes up 45% of all binlog events, but only 1% of the binlog data. It also takes second place for write waits in the system.

¹⁹ None of our tests involved changing cached content at all, so we don’t believe this issue affects our comparison results, especially since they were merely meant to represent our expected “optimal” solution, from a performance standpoint.

3.1 Test Setup

To test each configuration we initially created a separate set of VMs that mimicked the smaller of the two production Moodle instances (`innovate.moodle.wisc.edu`). The setup included

- A separate MySQL 5.6 master with 12 GB of RAM, 4 vCPUs with four virtual disks for OS, DB data, InnoDB logs, and binary replication logs respectively. Each virtual disk was attached to its own virtual disk controller, for reasons explained in Section 4.3.
- A MySQL replication slave read the master’s binary logs for backup purposes.
- A separate Apache 2.2 proxy frontend configured with `mod_disk_cache`²⁰, 4 vCPUs, and 4 GB of RAM.
- Two separate Apache `mod_php` 5.4 backends with 6 GB of RAM and 4 vCPUs.
- All systems currently share the same NFS server, itself a VM with 16 GB of RAM and 4 vCPUs.
- All of the machines currently run Debian Squeeze or Wheezy with both Linux Kernel 3.2.

Relative to the template system, the new aspects to the system included

- Three Memcached 1.4 VMs, each with 2 GB of RAM and 2 vCPUs. Only one of which was used for some of the tests.
- A MongoDB 2.4 server with 4 GB of RAM, 2 vCPUs and three virtual disks on separate virtual disk controllers for the OS, DB data, and journal respectively. We did not explore MongoDB replication, failover, or read preferences for these tests.

Note that in all cases the active data set fits in memory.

For reasons explained in Section 4.3, all VMs except for the NFS server were placed on the same network and two dedicated physical VMware ESX machines that were located in the same data center.²¹

3.2 Test Methods

There are a number of tools available for performing web site testing.

PhantomJS, Selenium, Tsung, Faban, `curl-loader`, `ab`, `httperf`, `tcpcopy`, and so on.

Of the few that we’ve listed, PhantomJS and Selenium are closer to the unit testing for correctness end of the spectrum. They operate by attempting to simulate a browser’s rendering of a web page’s content and allowing one to specify DOM selectors to script interactions with the results and match against expected return data. This can be useful in replicating a browser’s tendency to open multiple connections to fetch extra page resources (e.g. CSS, JS, etc.) as necessary and for timing the expected time for a user to perceive a page as ready, which incorporates aspects such as the order in which the extra page resources were requested in the original page, rather than simply a network client’s ability to receive the bytes. However, they are generally meant for single client simulations.

On the other end of the spectrum, Apache Benchmark (`ab`) and `httperf` can simulate many concurrent connections, complete with keepalives and POST data, but only to a single URL. This can be useful to stress test a particularly popular page, however it is generally emblematic of typical user traffic.

`tcpcopy` is a network layer TCP replay tool that is capable of reusing actual client traffic to test development systems.

²⁰ Since it turned out that the JMeter test plans we used don’t currently request embedded resources such as JS and CSS, the `mod_disk_cache` served little purpose in this case. However, in general it alleviates the need for browsers to tie up a backend PHP process with serving that content.

²¹ In hindsight we probably should have setup a separate NFS server as well, though we don’t believe it affected our results.

Tsung, JMeter, Faban, and `curl-loader` are somewhere in between. They all have a limited amount of document rendering capabilities, but are more focused on replaying a stream of requests for a requested protocol, in this case HTTP(S). They each also have the ability to handle cookies and POST form data to simulate some limited user interactivity. They are also especially meant to scale these sorts of operations in order to simulate different users concurrently accessing different parts of the system. Included in each are useful graph and summary statistics generation tools as well as warm up period specification options.

We used Moodle’s built in scripts [11] to generate XS, S, M, L, and XL test courses with 1, 100, 1000, 10000, and 50000 users respectively as well as test JMeter plans with 1, 30, 100, 1000, and 5000 users respectively. The generated test plans handle things like cookie and simulated browser cache management, HTTP keepalives, warmup periods, and several test loop iterations to make sure that the results are somewhat reliable. The main test loop consists of simulate a user visiting the site and doing things like logging in, navigating to the course home page, viewing the list of participants, viewing the course home page again, posting to the forums, viewing the response, and logging out. In all, each user simulated makes 13 page requests, each with some wait time in between.

The entire test suite took about 30 minutes to complete for each different configuration we tried. As such, the results took place over several days at different times of the day. Since there are some shared resources with the rest of our systems at large that we were not completely able to isolate due to lack of resources (e.g. shared disks, shared network segments and equipment), there may have been some variability inherent in our results. We therefore ran most of the tests several times over a period of weeks in order to obtain what we felt were accurate results.

We increased the test plans’ target throughput to 20 requests/sec in order to match the peak usage seen on the `innovate.moodle.wisc.edu` site and ran tests at several different concurrency levels, including XS, S, and M from a single machine placed on the same network as the test rig in order to keep client network delays out of consideration as much as possible. Beyond those levels, the Java JVM that JMeter required encountered threading issues and would have required multiple cooperating JMeter clients to make work. However, we did not have time to explore that option as it likely would have required altering the test plans substantially and the `innovate.moodle.wisc.edu` site currently tops out at roughly 100 users anyways, so a M sized course test plan at our target throughput is already representative and sufficient enough to cause high load on the two Apache backend servers.

We used the `moodle-performance-comparison` GitHub project [12] as a starting point for JMeter wrapper scripts to handle running multiple tests, resetting the database state and services after each test. The project also includes a PHP web application as a simple way to view the results in the form of Google charts. We adapted the comparison web app to report 90th percentile (P90) values in addition to averages. It should be noted that the comparison web app implementation involves loading many multi-megabyte JMeter results files rendered as PHP into memory. This process severely fragments the APC cache and degrades future PHP requests to the server, so an Apache restart on the backends is required before further tests are run.

Although the server infrastructure we used matches that of `innovate.moodle.wisc.edu`, we chose a slightly different Moodle software configuration. For instance, instead of exactly mimicking that site’s Moodle version, local module patches, custom UW theme, and Moodle configuration settings, we instead opted for a relatively barebones, simplistic, out of the box Moodle 2.6 installation. Doing so made tracking our changes to upstream code via `git` more natural and we believe it also makes our results more widely applicable, which is useful when considering our changes for upstream inclusion requests. It was also more compatible with the generated JMeter test plans, which made testing easier.²²

One nice feature of doing this, is that the JMeter test plans produced also include logic to parse a simple set of performance statistics that can be output in the default Moodle theme’s footer. This data allows the `moodle-performance-comparison` web app to report on additional metrics like the number and type (e.g. read vs. write) of queries made against the main database, session size, server side CPU processing time expended by the request, average and peak RAM usage, number of strings filtered, etc., all of which can be useful when comparing different system configurations.

²² Our changes for both Moodle and `moodle-performance-comparison` are available in various “patches” branches on our GitHub account: <https://github.com/bpkroth/{moodle,moodle-performance-comparison}/>

3.3 Cache Data Storage Evaluation

In this section we provide some more background and details on our evaluation of the different cache data storage options that we explored. We include some thoughts on our qualitative assessment of each option as well.

In each case we reset the database and configuration to the same state prior to our tests. For consistency, we used the standard database session store in each of these tests, and evaluated the different options for that layer independently in the next set of tests.

3.3.1 Local Filesystem

As mentioned previously, for this test we simply configured a file cache store pointed at a local filesystem, accepting the possibility of cache coherency issues for comparison reasons. Since we were not changing the cached content, we do not believe this invalidates the essence of our results.

We expected this mechanism to perform the best given it doesn't need to access the network to lookup cache data upon every page request. This has the additional benefit of not depending upon yet another remote service to service requests. Instead the data can be accessed locally, and will most often be in the local OS page cache.

There is the potential that under extremely heavy load that memory is reclaimed by the OS in order to be used by Apache processes. However, in such a scenario we feel that the system is already thrashing, so the performance difference between a local cache and a remote cache due to network costs at that point becomes moot.

3.3.2 NFS Filesystem

The traditional extension for a global cache data store, is to use the same file cache store driver as we used for the local filesystem cache, but pointed at a shared filesystem. Since we already use NFS as the storage mechanism for large data files and as the authoritative repository for the Moodle code base, it became the obvious target to use. However, as previously noted, the default NFS cache implementation is nearly as slow as not caching at all.

Upon detailed profiling analysis of a simple logged in front page view we found that of the 504ms of CPU time, 231ms of which is spent in NFS calls: 89.6ms for `fread`, 72.3ms in `flock`, and 69.2ms in `fopen`.

After analyzing the Moodle `cache/stores/file/lib.php` driver, we found that each `get()` cache call acquires a shared (read) lock on the file prior to issuing the `fread`. It turns out, that the Linux NFS client flushes the local page cache whenever a lock is requested, so the client's `fread` request must fetch the content from the server again.²³ This seems excessive and unnecessary when we consider that first, the cache should seldom change, and second, the file cache driver's `set()` implementation uses the standard write to a side file and rename into place trick to make sure that partial file reads don't obtain inconsistent data. Indeed, the `set()` portion the code doesn't even bother to acquire a lock, instead depending upon the POSIX `rename()` operation to be atomic within a filesystem.

To work around this issue, we adapted the Moodle file cache driver to make use of a previously defined, but unused `$CFG->preventfilelocking` configuration option within Moodle to simply skip the `flock` call. This reduced the time spent in NFS calls to only the `fopen` call time.

That time was higher than expected due to the default NFS Close-To-Open cache semantics²⁴ which cause the client to double check that it's file attribute (i.e. `stat`) cache is still up to date prior to opening the file. If we instead define another NFS export mounted with the `nocto` option, the client will trust its local `stat` cache data up to the TTL, which has a default range of between 3 – 60s.

²³ http://nfs.sourceforge.net/#faq_d10 and kernel source code function `fs/nfs/file.c:do_setlk()` have further details.

²⁴ http://nfs.sourceforge.net/#faq_a8

Under the assumption that the cache data we were dealing with could tolerate a few seconds of inconsistency due to differing `stat` TTLs for different files between backends, we tested NFS cache stores with the default `flock` and `cto` enabled, only `flock` disabled, and with both disabled.

Note that unlike the local filesystem cache, which would never repair itself in the face of inconsistencies across different backends, the NFS `nocto` cache inconsistencies that might arise due to differing TTLs has a built in limit and automatic repair mechanism.

With both options disabled, one would imagine that the amount of network traffic involved to obtain cache data is relatively minimal and infrequent for commonly accessed data, and that it would perform nearly as well as a local filesystem cache.

We also note that for both file based caches we can use standard file system access controls and separate hierarchies to provide different independent caches for multiple Moodle instances. However, in that case, they would compete for local system page cache RAM instead of network bandwidth to a shared cache store.

3.3.3 Memcached

Memcached is really more of Key-Value a protocol these days than anything. It provides a simple `GET`, `SET`, `FLUSH`, etc. set of commands. In its traditional implementation it's an in memory storage mechanism only, though there are several implementation variations that provide some form of data persistence as well. Some examples include Couchbase [2] and MemcacheDB [16], though they may add (e.g. replication) or lack (e.g. expiration) features when compared with the traditional implementation.

The version of MySQL we run is another such example, which uses an InnoDB storage engine backed table as the persistent store for the Memcached key-value pairs, perhaps at some reduced durability levels by batching updates up and committing them in the background after some timeout. We attempted to implement such a service for testing, however we encountered an assertion in the database once we applied the Moodle workload on the system, which prevented us from making further progress.²⁵ This would appear to indicate that few people actually use a MySQL backed Memcached. Prior to that discovery, we ran some `memcslap` microbenchmark tests in order to compare the effect of different durability reduction settings on the performance of a MySQL backed Memcached when compared against a standard in memory implementation, however we were unable to find a significant difference at various levels of concurrency.

Persistence in a Memcached cache implementation might be important to avoid cold cache or thundering herd problems when the service must be restarted for security patching or configuration changes, since the in memory data would be entirely lost whenever the process ends. To test this aspect of things this we ran several JMeter tests for MongoDB and Memcached cache stores, restarting the cache store part way through the test. Alternatively, we could have crashed the machine entirely, but we felt that then the majority of the time is simply spent in the OS rebooting and recovering that this was a more useful comparison. In practice we found that since an in memory Memcached implementation is so light weight, it takes almost time to restart the service when compared with a “real” database-like system such as MongoDB or MySQL which have to analyze and possibly redo journal or log activity before accepting new connections. Thus, in our tests, the cost to rebuild the Memcached data from the original source is dwarfed by the other services’ restart times and the full uncached page requests the system needs to process in the meantime.

However, other systems do have an advantage over Memcached when it comes to access control. This is an important problem for us as we operate in a shared hosting environment. Thus, simple firewall protections that are used to protect Memcached in most settings are insufficient for ours.²⁶

One potential option that Memcached now provides is SASL authentication which acts a lightweight barrier in the form of a shared secret that both client and server must know to be able to access the Memcached instance. Unfortunately, again our attempts to test out this variation were met with problems. This time, we encountered a two line variable misspelling in the `libmemcached` package’s Makefile which resulted in all

²⁵ The bug filed was available at the following URL, however Oracle apparently now has the policy of marking all bugs as invisible once they’ve been confirmed: <http://bugs.mysql.com/?id=72435>

²⁶ For example, vhosts may move between VMs from time to time, and some VMs may host vhosts for both “privileged” and “unprivileged” clients to a given Memcached instance. This is usually safe to do in our environment due to other protection mechanisms that we have built into the system, such as separate Apache users and process spaces.

dependent packages, such as `php5-memcached`, from being built without SASL support, despite requesting it.²⁷ We deemed the amount of work required to fix the problem outside of the scope of this project, though did submit several patches upstream. Here again it would appear that few people are actually using SASL authentication with Memcached.

However, we note that even with authentication, Memcached does not perform any access control. Thus, any client that is able to connect can not only dirty the cache space, but also **FLUSH** it entirely. This makes using a single Memcached instance for multiple Moodle instances undesirable, since a key feature of the MUC system is the ability to purge all caches for events like code upgrades or theme changes. Doing so would invalidate the cache of all Moodle instances using that shared cache, leading to unnecessary cold cache periods on all sites. Additionally, if the same Memcached instance were used for both cache and session data, then purging the cache of even a single site, would log all users out of all sites, which is clearly undesirable. Thus, the use of Memcached in such an environment leads to a proliferation of extra service VMs and dependencies. Also note that the persistent forms of Memcached services also share this disadvantage since it is a deficiency in the protocol.

Left only with a standard in memory Memcached implementation, we still had several configuration options to explore. Among them was whether to use compression on the network communication or not, whether to use a single Memcached server, or have the client pre-hash the key to choose from among a cluster of servers to issue their request to, and whether to use the standard PHP serializer of data structures or an alternative serializer called `igbinary`.

The default PHP serializer's output resembles something like JSON in that it is a textual representation of the data structures. The `igbinary` serializer simply stores numerical values in their binary format instead of as ASCII strings, and by default also can compact repeated strings as references to previous instances. The goal in both cases is to save serialization space, possibly at the expense of some additional CPU overhead at the time of serialization, but a reduced amount at unserialization.

In some simple comparative tests we ran on the NFS file cache data, we found that the use of `igbinary` serialization resulted in 37% less unserialize CPU time and 50% less space. The latter of which one can imagine would be important for remote cache systems as it translates into less data transfer delays.

Although Memcached natively supports `igbinary` if it's available²⁸, we were impressed by these small scale results, so we attempted to provide some support for it in some of the other cache and session systems we developed.

It should be noted that in modern Memcached clients, failover does not usually occur since it would cause invalid cache consistency between nodes. Instead, a form of consistent hashing of keys is used to distribute values to particular server, and clients either block or timeout if that server is unavailable. For cache systems this is okay since the data can ultimately be reconstructed from the original source, but for sessions this may not be acceptable [4].

3.3.4 MongoDB

MongoDB is a schemaless JSON document store that is disk resident and primarily relies on the OS to handle buffer management. It supports replication complete with client specified read preferences, automatic data sharding, and authentication and access control, making it very attractive for use in a shared hosting environment. For instance, we could define several different databases for each separate Moodle instance, assigning separate credentials for each one to manage its own document collections independently. It also has a very nice feature that they call "write concerns".

MySQL's InnoDB storage engine and Memcache API, for instance, provides a way to get tunable persistence of data, thereby limiting the amount of cold start and session availability issues that a service restart or outage might entail. However, they are server wide settings rather than based on an individual client's requested level of durability.

²⁷ <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=724872#10>

²⁸ We prepared a Debian package for it since non-existed.

MongoDB on the other hand allows for the application to specify a given write’s durability requirements through the use of its “write concern” levels [10].

For example in the case of cache writes, we may be perfectly happy with a write concern of “acknowledged” but not necessarily journaled (this is the current Moodle default for MongoDB caches), since in the case of a failure, the cache should simply not contain the data (in MongoDB all single document writes are in theory atomic), and the system can go about repopulating it from the original source.

On the other hand, if replication is being used, the above situation may get us into a discrepancy when a write to is acknowledged at the primary, but fails to reach the secondary, and then the primary dies leading to that secondary being elected as the new primary. Now, with automatic client failover, or client specified read preferences which allow them to read from the secondary, the secondary (now primary) may return stale cache rather than up to date or no cache data, which can lead to display or logic errors for the client. Moreover, unlike the NFS `nocto` situation, this coherency issue may not resolve itself.

MongoDB’s replication feature is also an attractive feature for its potential to reduce system stalls, cold cache degradation, session loss, or other outages through the use of automatic client failover. However, due to time constraints we have not experimented with this feature at all.

Automatic sharding is another feature that MongoDB provides which splits a namespace up amongst participating servers in order to distribute the load. However, unlike the Memcached implementation which depends on the client to perform a hash to determine the server to communicate with, the MongoDB solution requires several more servers (`mongos` and `config` servers) to manage the shards. Given we think our current cache and session sizes can fit within 2G, we thought this was an unnecessary extra set of overhead for our current environment and did not explore it further. Clearly if we were to scale Moodle another order of magnitude or two to the MOOC scale, it might need reconsideration.

MongoDB unfortunately does not currently provide network communication compression. Instead, it encodes the JSON documents into BSON [6] format for network transmission and eventual storage. In our assessment, the JSON to BSON encoding is akin to the comparison between the standard PHP serialization and `igbinary` serialization. There can certainly be some savings in large numeric data, but string data, which appears to be a large portion of the Moodle cache and session data workload, is largely unaffected. Indeed, in certain cases BSON encoding can actually be larger than JSON encoding [1]. We believe MongoDB could also benefit from client-server network communication compression.

Since Moodle’s treatment of MongoDB is currently only as a Key-Value store, it makes little use of its ability to intelligently encode more complex data structures. Thus, lacking any other form of transparent network compression, we also explored the use of `igbinary` serialization as a replacement for the standard PHP serializer with the MongoDB cache driver.

3.4 Session Data Storage Evaluation

In this section we provide some more background and details on our evaluation of the different session data storage options that we explored.

For each of these session stores we also tested a variation with `igbinary` in use as the `session.serialize_handler`. We ran each of our tests with an NFS `noflock nocto` based cache store.

One important thing to note about all of these session stores is that they are really just wrappers around PHP’s built in session handler.²⁹ As part of that process, PHP automatically serializes the `$_SESSION` data structure according to the `session.serialize_handler` setting prior to handing it to the session save handler.

To make sure that the serialized session data remains consistent, session stores must acquire a lock on the data that isn’t released until just before the page is returned to the user in order to prevent parallel page requests for the same session³⁰ (excepting certain mostly static theme resources like images and CSS) from overwriting each other’s changes.

²⁹ e.g. `session_set_save_handler()`

³⁰ Most browsers issue anywhere from five to nine parallel requests over separate HTTP connections to the same server in order to improve page rendering time.

Each of the different systems handles this in a slightly different way according to its capabilities, but in every case it has the unfortunate effect of serializing a single user's requests and potentially tying up a large number of Apache and other resources in the process, especially in the case of errant client browser behavior (e.g. redirect loop). This can effectively amount to a Denial of Service attack.

3.4.1 Standard Database

In the case of a MySQL database backed session store, it requests a named lock (i.e. `GET_LOCK()`) on the row object.

There are two nice features about this mechanism. One is that the lock is automatically released when the MySQL client disconnects. As mentioned in Section 4.3 we do not use persistent connections, so this happens at the end of the page even if an error occurs. As we'll see later, all of the other session stores either require some background process to look for stale locks or exhibit poor error recovery behavior.

Another nice feature is that we can use either the MySQL `performance.schema.threads` or `information.schema.processlist` tables to introspect the current state of the database and get a count on the number of waiters for a given lock.³¹ This allows us to perform a quick check and exit early in the case that errant client behavior has caused an inordinate number of Apache processes to pile up waiting on a given lock.

However, the standard Moodle session database driver stores the serialized session data back in the main database. As mentioned previously, there is currently no way to specify per-transaction durability requirements in MySQL, and session data is always changing, so each page load results in a commit to the database. This in turn results in two forced write to the InnoDB logs and binary replication logs respectively.

It is unfortunately also not practical to disable binary replication logging for those transactions since the `SET sql_log_bin` statement requires `SUPER` user privileges to the database server.

3.4.2 Separate Database

As an alternative, we implemented a separate sessions database driver for Moodle and setup a separate MySQL database server that did no binary logging and was tuned for reduced durability³², allowing committed data to be buffered in memory for a time before actually flushing it to disk, and the `COMMIT` request to be acknowledged immediately. However, this mechanism also requires setting up and maintaining an extra network connection for each page request.

Since this database was entirely new, and since PHP expects data to be serialized out anyways, we were able to define a new table schema that used a simple `sid` `VARCHAR` key, `sessdata` `BLOB` value, and `timemodified` timestamp. The use of a `BLOB` type allowed us to directly make use of the `igbinary` serializer without `base64` encoding the data, which would have increased the serialized string size defeated the purpose.

We also explored the use of MySQL's `ROW_FORMAT=DYNAMIC` in order to save the `BLOB` values in separate overflow pages from the typical clustered B-Tree that table data goes in in order to keep the B-Tree index lookups efficient, since the first couple of queries for the data are only for the `sid` and `timemodified` values. However, it was difficult to get an insight into the MySQL internals to determine whether or not this had any measurable effect on our workload given everything fit in memory.

3.4.3 Memcached

Memcached has no native ability to lock a given record. Instead, the PHP session implementation provided by the Memcached extension uses the compare-and-swap mechanism to guarantee an atomic key insertion. By polling such operations for a known key, it can provide the illusion of locks on a given session key's data.

³¹ The `performance.schema.threads` method has less latching overhead, however it requires a view to be defined in order to delegate `SELECT` access to general clients, and is not available in all MySQL installations, so our patch includes support for both.

³² `innodb.flush_log_at_txn.commit=0, innodb.flush_log_at.timeout=1`

Unfortunately, it is up to the client who acquired the lock key to remove it when they are done in order to allow subsequent requests to proceed. If the client fails for any reason before that happens the lock will remain and subsequent client requests will timeout while waiting to acquire the lock. Memcached's data expiration feature can in part be used to automatically recover from such incidences, however the expiration period must be set to extremely long time periods (the default is currently 2 hours) in order to allow long running operations like user initiated course archive and restore operations to safely complete.

By default the Memcached session driver compresses data exceeding a particular size (e.g. 2000 bytes). Since, unlike with the cache data, the `igbinary` serialized session data did not appear to behave well with this feature, we experimented with both options separately.

As mentioned previously, an in memory Memcached when used for session data has the very undesirable behavior of effectively logging all users out whenever the process is restarted. We did not explore the use of a cluster of Memcacheds for session data, but one might expect that such a system would reduce the number of affected users to only some fraction.

3.4.4 MongoDB

Given its nice write concerns feature, we decided to try and implement a session driver targeted at MongoDB that would allow us to optionally require or not “safe” writes. Unfortunately MongoDB also lacks the ability to obtain an explicit record lock, so we implemented the same trick as Memcached and simulated it by performing atomic `insert` operations for lock documents. Since we needed to be able to check the response code to determine whether or not the `insert` succeeded (i.e. we obtained the lock), we must use “safe” writes for the lock operations. Thus, the ability to use decreased durability was limited only to the session data itself.

Additionally, these lock documents have the same client failure issues as Memcached, and we use the same “solution”, namely an `expireAfterSecs` index on the `timemodified` field of the lock document.

As mentioned previously, MongoDB does no network compression, instead merely relying on BSON encoding of the document, which is very similar to that of the PHP `igbinary` serializer.

If the PHP session handler didn't serialize the data ahead of time, we could potentially consider handing the entire `$_SESSION` data structure to MongoDB to be encoded as a BSON document rather than being stored as a single flat string. However, as it is, that would require multiple serialize and unserialize steps to accomplish and the extra CPU overhead didn't seem to be worth the limited network traffic savings considering we wouldn't be able to do something like only send the updated fields unless we also kept a full copy of the original session data and were able to efficiently perform a diff of the data. We merely mention it here as another case of impedance mismatch.

3.4.5 NFS Filesystem

In the case of an NFS backed session store, `flock` and `cto` options must be used in order to assure session consistency in the case that a single client session's request moves between different backends. In actuality, our frontend Apache proxies use a `ROUTEID` cookie to maintain “sticky sessions” to tie a client's session to a given backend as much as possible, so this may not be as much of an issue.³³ However, Moodle simply uses the internal PHP file session handler for this case, so to keep our amount of development work down, we did not explore the possibility of skipping the `flocks`.

It may in fact turn out that tying client sessions to the same backend actually optimizes the `flock` behavior since, unlike cache data which all backends must access for all client sessions, session data then becomes uniquely accessed by a single backend, so there is no lock contention between nodes - only between requests to the same node. We did not explore the NFS client/server code enough to determine whether or not this is indeed optimal behavior for this use case.

³³ We probably could use the same mechanism to keep session data within a local filesystem store, but that would prevent clients' logins from transparently failing over to another backend in case of failure, so we did not test it.

We will also note that we are currently using NFSv3 which has an explicit NLM lock and callback mechanism. NFSv4 added file delegations, however locking is done by polling, as it is with MongoDB and Memcached, only in this case with an exponential backoff, which can lead to very slow results between backends competing for the same lock. Hence, we do not recommend the use of NFSv4 for this session store.

Additionally, although local file locks are released when the process releases or closes the open file handle, there are cases where such events may not be registered with the NFS server. If the backend server crashes, for instance, it may take some time before other backends can take over the lock for that server's clients' session files.

4 Results

4.1 Cache Data Store Results

In this section we review our results for S and M sized course concurrencies on the different cache stores we tried. We omit the XS results since we felt that single threaded tests showed too much variability and were not reflective of a true workload. Additionally, although the `JMeter` and `moodle-performance-comparison` tool we used can export data for individual pages, including other metrics, we concentrated on the combined latency results for an entire simulated user's test run (i.e. all 13 page requests) to complete.

Figures 1 and 2 show the average and 90th percentile (P90) results for a small course (30 out of 100 users), respectively. Figures 3 and 4 show the same for a medium sized course (100 out of 1000 users). We focus primarily on the higher concurrency medium sized course results, though the general observations hold for the lower concurrency results as well.

Note that in order to keep the charts legible so that the relative difference between various configurations could easily be appreciated, we have adjusted the minimum and maximum values for each charts x-axis appropriately.

The first thing to note is that despite our best efforts, we were unable to remove all sources of variability. As such, the P90 values are still considerably higher, in one case almost by a factor of two though in most roughly 50% more, than the averages.

Next, note that as expected, the local filesystem cache and the default NFS cache give the best and worst performance, respectively. As discussed in Section 3.3, removing the `lock` from the NFS cache only partially solves the problem, resulting in a 53% improvement over the standard NFS cache for average latencies. As expected, removing the `cto` mount option as well, reduces the network traffic to almost none, allowing the NFS `no flock nocto` cache to nearly match the performance of a local cache (< 5% difference), though of course with the potential cache coherency problems mentioned previously.

Surprisingly we found that use of the `igbinary` serializer with the NFS `no flock nocto` cache, actually increased latency. Using the extra metrics that the `moodle-performance-comparison` tool provides, we were able to see that in this case the `igbinary` serializer actually results in extra CPU time, and since the extra work does not result in a savings in network traffic (once the local page cache is warm), the effort is wasted. What's counterintuitive about this is that we used the same dataset in our small scale PHP vs. `igbinary` serializers comparison and found that the `igbinary` unserializer used less CPU time than the standard PHP one. We've repeated both tests multiple times with the same results and cannot yet explain the discrepancy.³⁴

However, in the case of Memcached and MongoDB, the use of the `igbinary` serializer reduces the network traffic due to the reduced cache string size and so latency drops accordingly. Memcached can additionally benefit from compression in this case as well. We thought it was interesting that MongoDB, despite a lack of compression, seems to gain some benefit from its BSON encoding for wire transport. When that's combined with `igbinary` serialization, MongoDB is positioned somewhere between Memcached with compression and

³⁴ `igbinary` does have an option to disable its repeated string compaction routines and only encode numerical data in order to expend fewer CPU resources, but this comes at the cost of not being able to save as much storage and network, which defeats the point of the alternative serializer in general.

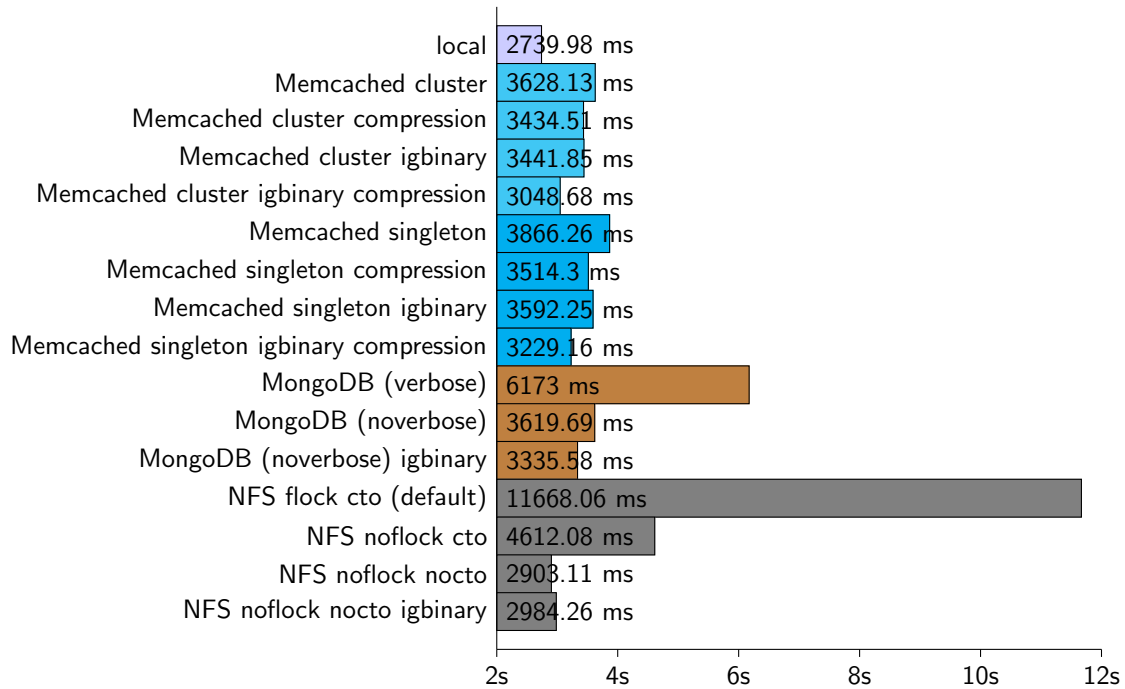


Figure 1: Cache Stores - Average Total User Test Latency (ms) - S Course

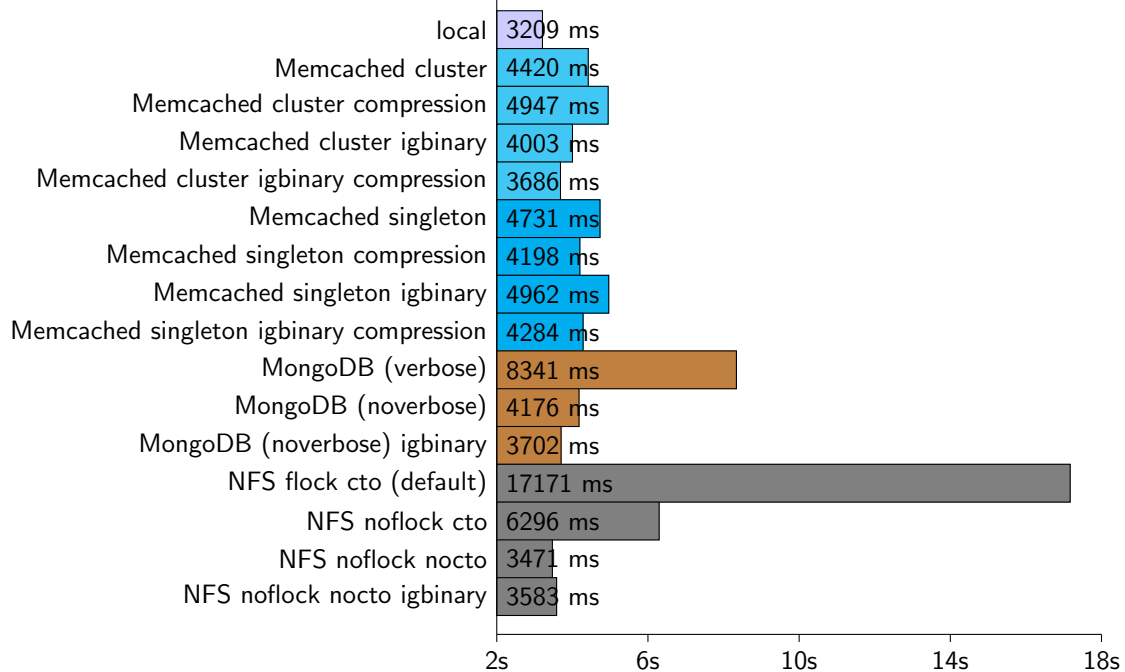


Figure 2: Cache Stores - P90 Total User Test Latency (ms) - S Course

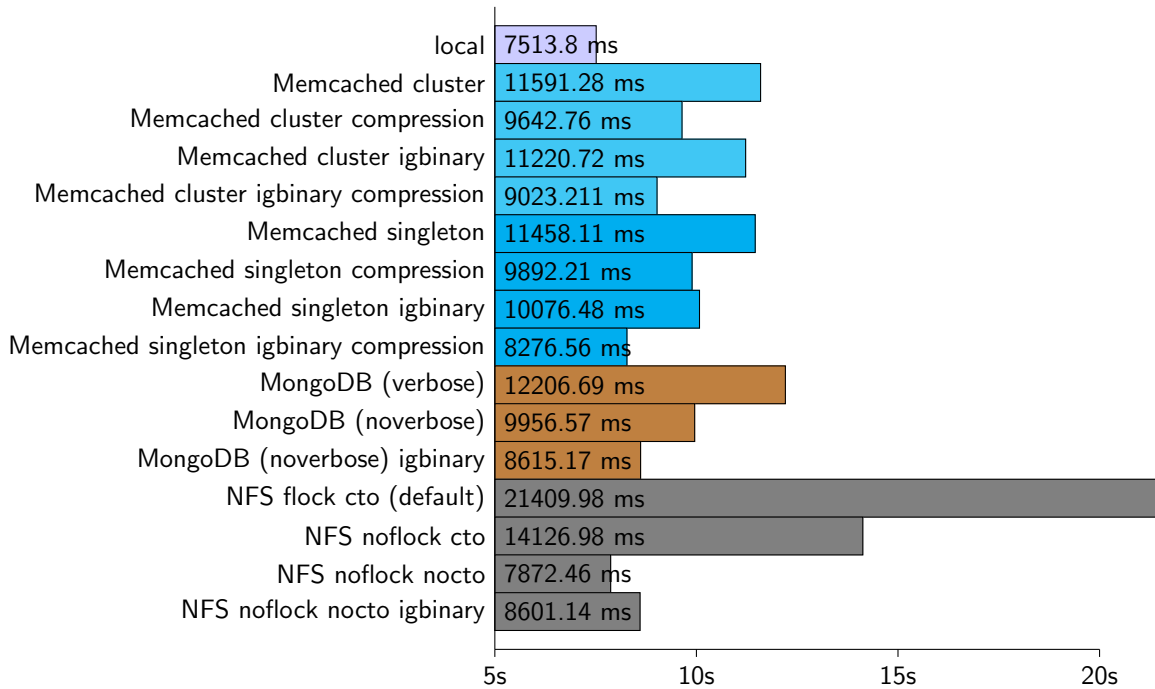


Figure 3: Cache Stores - Average Total User Test Latency (ms) - M Course

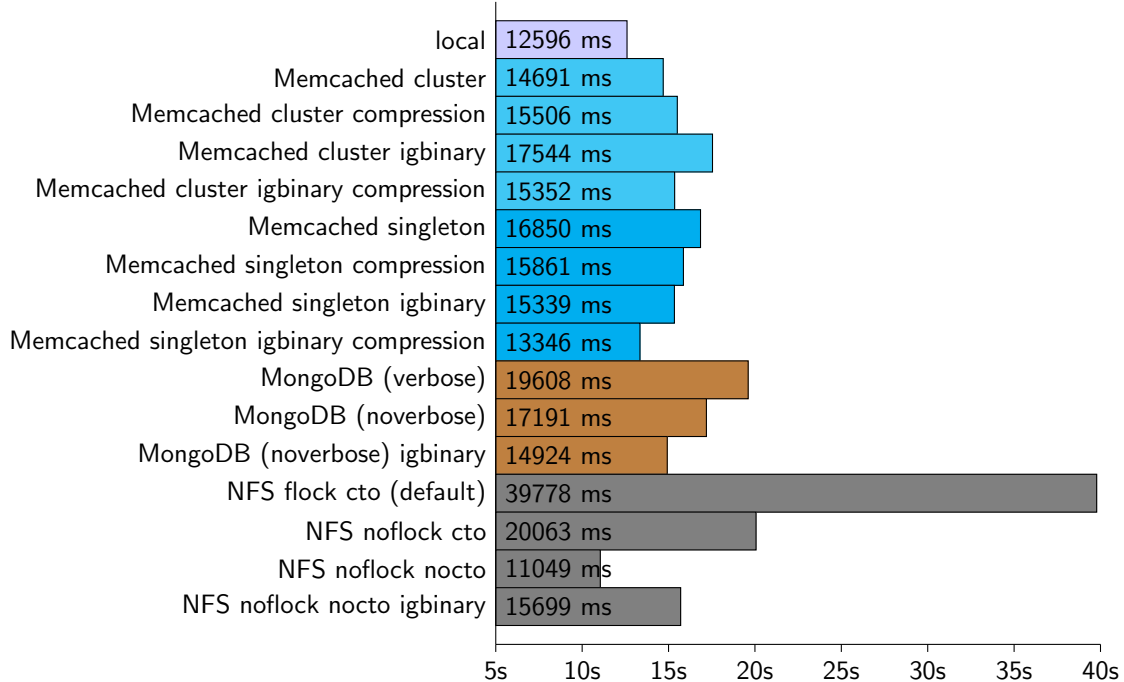


Figure 4: Cache Stores - P90 Total User Test Latency (ms) - M Course

Memcached with both compression and **igbinary** serialization. The remaining difference appears to be in terms up network usage.

The use of a cluster of Memcached servers had only a minor effect, but seemed to actually increase latency at higher levels of concurrency. We imagine this was due to the overhead of having to manage extra network connections.

Finally, we note that the use of verbose mode on MongoDB, which we had enabled during our initial deployment for troubleshooting purposes, significantly increased the latency of our tests (22.6%), although even then it was only about 6% off of what the standard Memcached results were.

4.2 Session Data Store Results

In this section we review our results for S and M sized course concurrencies on the different session stores we tried, with similar caveats to those mentioned in Section 4.1 on our cache results.

Figures 5 and 6 show the average and 90th percentile (P90) results for a small course (30 out of 100 users), respectively. Figures 7 and 8 show the same for a medium sized course (100 out of 1000 users). Once again, we focus primarily on the higher concurrency medium sized course results, with the general observations holding for the lower concurrency results as well.

First, though it can be better seen in the case of the small course test results, as expected, use of the main database to store session data has the worst overall latency in our tests. Additionally, as is explained in Section 4.3, we used MySQL's client compression for our primary database connection.

From our tests it appears that in this case, unlike the cache data case which is primarily an unserialize dominant set of operations, the **igbinary** serialized session data does not compress well, so the additional CPU overhead of needing to unserialize and reserialize it for every page load in addition to compressing that data between the backend Apache server and MySQL server simply increases the latency rather than helping at all.

We find similar results in the case of Memcached, whose PHP session handler automatically compresses data over 2000 bytes (which nearly all of our sessions are).

On the other hand, MongoDB, which lacks network compression, actually sees a benefit in the **igbinary** serialization CPU overhead, since it has a corresponding network reduction that cannot otherwise be reached.

Similarly, when we use a separate database to store session data only with **igbinary** serialization (no compression) we can achieve very good results. In fact in both cases, the separate database driver actually uses roughly 80% of the network traffic of the MongoDB or Memcached systems. We suspect some of this is due to its native locking support in MySQL which doesn't require polling.

Note also that we achieve a 12 – 19% improvement by reducing our durability levels from MongoDB safe writes to MongoDB unsafe writes and from a shared database with full ACID compliance to a separate database with relaxed durability. In both cases we have achieved performance equal to or even better than Memcached without sacrificing data persistence entirely. Though of course we have fewer guarantees on how up to date our persistent data is, it at least means we can restart services without logging out all users.

We also had the very surprising result of the average NFS session store latencies actually being the least at high latencies, despite much higher P90 results. As mentioned in Section 3.4 we think this may have something to do with an **flock** affinity to the previous backend to acquire the lock, but have not investigated it. Still, with the high variability and poor lock controls and behavior when compared with the database approach, we don't feel that this is a good method to use.

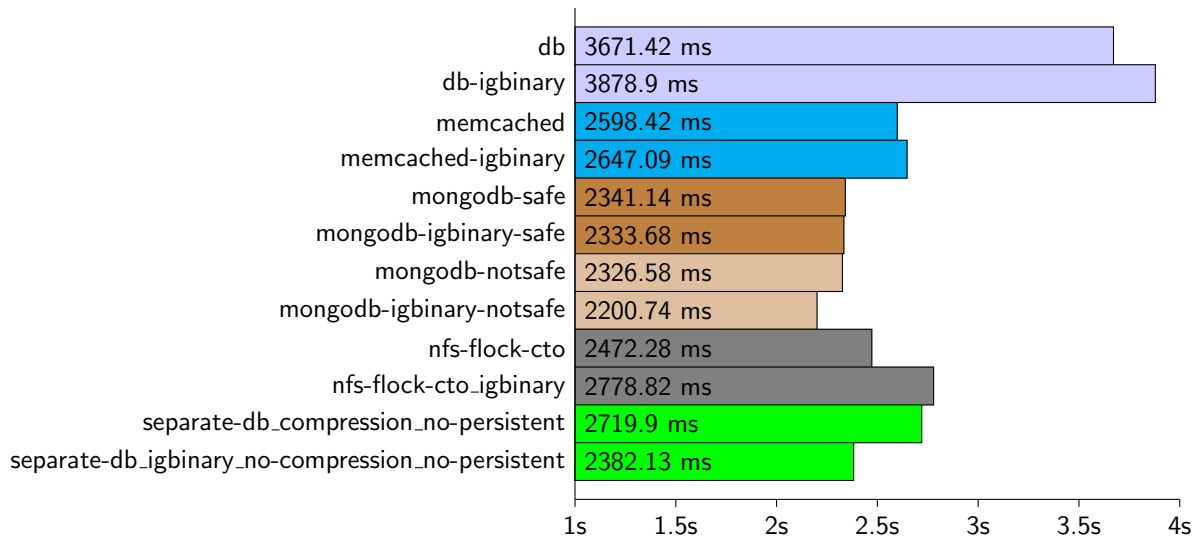


Figure 5: Session Stores - Average Total User Test Latency (ms) - S Course

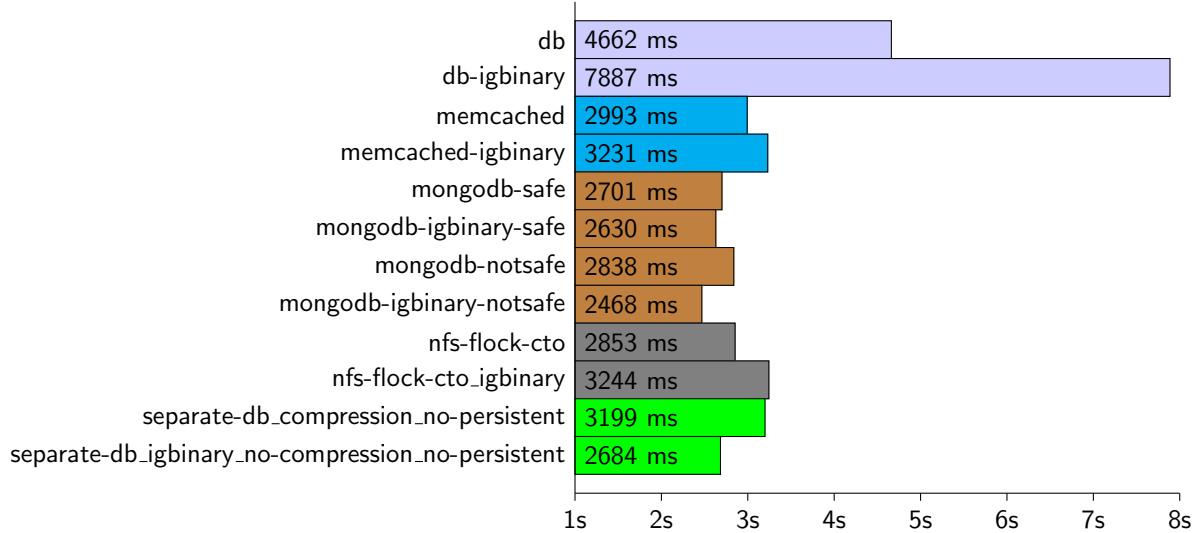


Figure 6: Session Stores - P90 Total User Test Latency (ms) - S Course

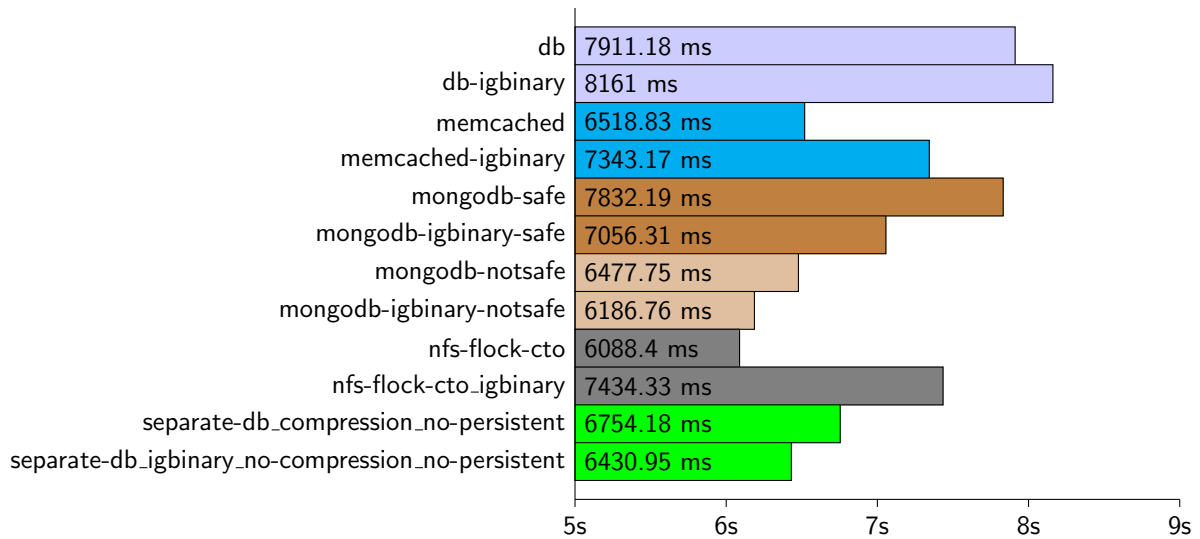


Figure 7: Session Stores - Average Total User Test Latency (ms) - M Course

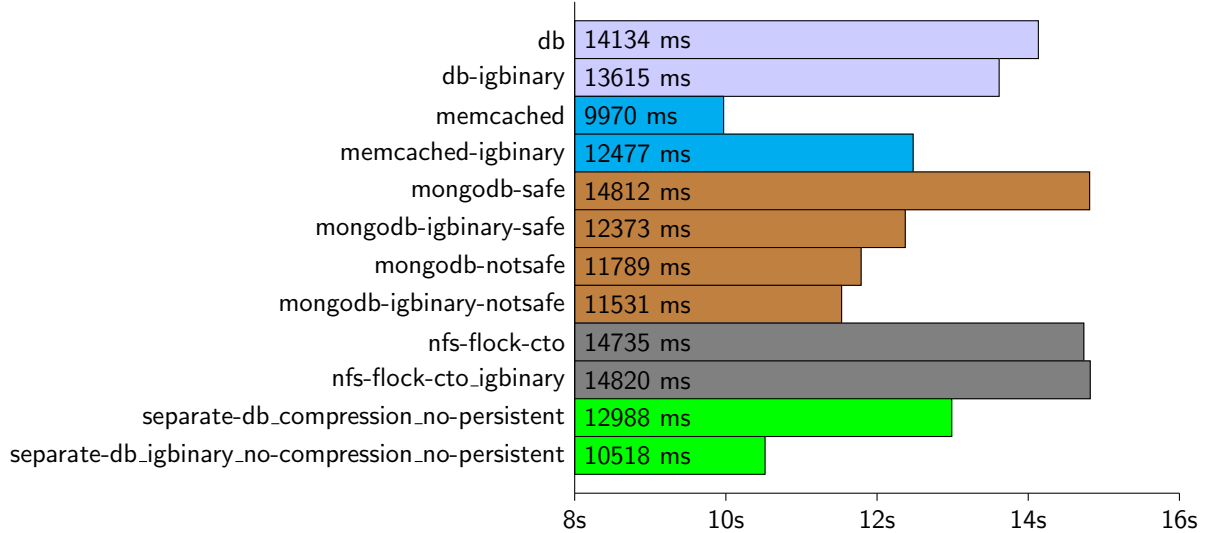


Figure 8: Session Stores - P90 Total User Test Latency (ms) - M Course

4.3 System Discoveries

As mentioned previously, we originally encountered difficulty in obtaining reliable and repeatable measurements in many cases. In this section, we briefly list a number of the general systems discoveries and improvements we made.

As a means of guiding our story, we present Figure 4.3, which shows an annotated RRD graph³⁵ of an HTTPS request of a simple (37 bytes) HTML page (i.e. no Moodle, PHP, or database calls involved) to our test site performed by its frontend proxy node (in order to rule as much network delay out of the test as possible). The dark line in the graph shows the average response time during a period, while the shaded region shows the minimum and maximum response times during the period.

At point one, when the graph starts, all of the VMs involved are placed according to whatever VMware DRS rules have determined are “best”. In particular, they were spread amongst different physical machines *and* physical datacenter buildings.

During an unrelated investigation into timeouts reported during a brief period on the smaller production site `innovate.moodle.wisc.edu` we learned that the public network connection between the two buildings is backhauled to another area of campus and crosses older routing hardware, despite the fact that the traffic in question was intra-VLAN traffic and therefore required no routing.³⁶ As the router is a core shared resource to many other groups around campus, our application was inadvertently hit with packet drops when an unrelated group’s traffic spikes overran the router’s resources.

We obtained permission to take two of the physical hosts in one of the datacenters out of the VMware cluster to use for our tests. As a result, our VMs no longer crossed the router for the intra-VM communication required to serve our test site. This had the additional benefit of isolating our tests from other unrelated VMs’ CPU needs, though not their disk needs as the storage was still shared. At the same time, we disabled automatic load-balancing VM migrations within our two-node cluster, preventing VMware from expending unnecessary network resources to try and migrate VMs to balance the load while we did our tests. As can be seen from point two in the graph, these simple changes resulted in a drop of roughly 40 ms (29%) in request variability.

Now, having some physical machines to experiment with we took the opportunity to review the recommended VMware hardware configuration settings. As can be seen from points three and four in our graph, we found that adjustments to how the BIOS handles power management can have a very large effect on general system response. We registered drops in average response latency of 60 ms (37.5%) and 70 ms (70%) in variability.

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

Table 2: Simple HTML request response times vs. wattage under different BIOS power management modes.

Table 2 shows that when we disable the Dell factory shipped default of BIOS managed power management in favor of either no power management or OS (hypervisor) control power management, we achieve significant improvements in average response latency for relatively minor overall power usage increases. This seems to be because the OS has a better idea of what to expect of future workloads than the hardware does.

The final point in our graph represents a drop due to two SSL optimizations. The first change, discovered during the course of the Heartbleed bug, was to instruct VMware to expose the `aes` instruction to VMs by increasing our test cluster’s EVC mode.³⁷ This change, combined with a recent enough OpenSSL library to

³⁵ We thankfully setup a `collectd` infrastructure just prior to this work which allowed us to see the effect on long term trends of our changes.

³⁶ Our group unfortunately no longer manages their own network equipment, but we’re told that the hardware and architecture involved in this issue is due to be replaced this summer.

³⁷ VMware allows a method of masking out the instructions exposed to guest OS in order to allow migrating VMs between machines of slightly different processor versions.

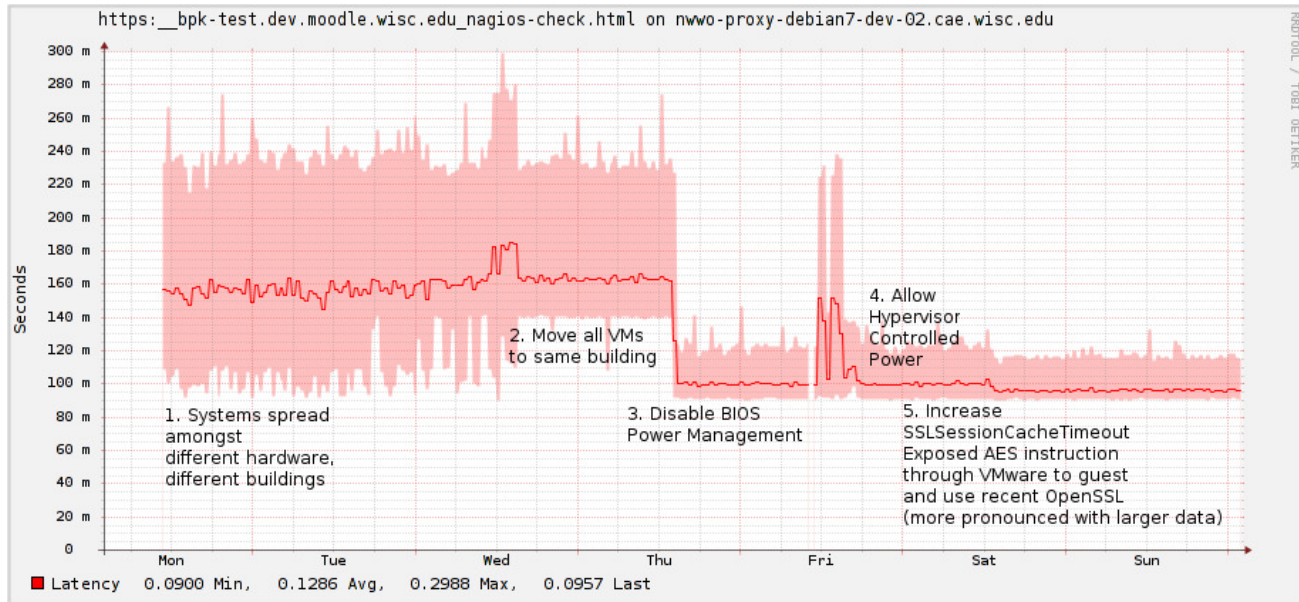


Figure 9: Simple HTTPS page fetch performance results due to various system changes.

make use of it, resulted in an increase in `aes-128-cbc` operations per second of 426%, according to `openssl speed` tests. However, this improvement only really appears in larger file transfers.

The second was an increase in Apache’s `SSLSessionCacheTimeout` which decreases how frequently non-keepalive clients need to go through the expensive SSL renegotiation process to establish symmetric keys for the session.³⁸ Unfortunately, that does not alleviate the need to exchange and verify the public key material³⁹, which accounts for the remainder of our latency in this simply test.

A similar cache timeout issue we found was related to the Moodle filtered text layer. Here we found that an “auto linking” filter was the source of cache regeneration events that could take as long as 10 seconds to complete. As mentioned in Section 2.2, this set of data is handled in a database table outside of the MUC. Moreover, the decision to regenerate the cache is based solely on a time-to-live value, rather than taking any `timemodified` values from the original source’s record into account, even though the data is available within the same system and could easily be obtained. We decided to increase the `cachetext` TTL to 1 hour in the hopes that it would decrease the frequency of such delays. As mentioned, we think it should be possible to fix the system to use more informed cache expiration mechanisms, however we recently discovered that Moodle developers intend to dispose of the text cache entirely [19].

As mentioned in Section 2.1, we now use multiple virtual disks each attached to separate virtual controllers in our database VMs. During our analysis, we found that when combined with iSCSI MPPIO at the hypervisor instead of LACP, we reduced our aggregate log disks’⁴⁰ read latency from an average of 15.1 ms to 4.4 ms.

The use of MPPIO instead of LACP (i.e. bonded network interfaces) with iSCSI allowed the hypervisor two 10 Gb TCP paths instead of a single 20 Gb path. This allows each TCP connection’s endpoint to target a separate array, which makes better use of the array’s write buffers, cache, and disk spindles. `iozone` benchmarks showed that this change resulted in an average of 77.6% improvement in IOPS and 54.8% improvement in BPS throughput.

³⁸ We use Entropy Keys and associated daemons to distribute entropy throughout our VMs so this isn’t as expensive in our systems.

³⁹ Public keys, including intermediaries, are roughly 6 KB in our case, compared with just over 4 KB in Google’s case, for instance. Their `app.connect` times as reported by `curl` are accordingly about two-thirds of ours.

⁴⁰ Since InnoDB uses MVCC in which case it may need to read the undo logs in the course of normal operations, whereas the binary logs are typically read by the slave out of the master’s memory, keeping the different disk workloads separate is also important.

Similarly, the use of multiple virtual disk controllers allows the guest OS to create multiple independent queues for disks, which, even if they disks are ultimately on shared storage, improves the IO scheduling.⁴¹

Finally, we note that we discovered through our initial JMeter tests two MySQL network connection settings that affected the combined user test latency. First, latencies increased by 30.4% when we enabled persistent connections so that each page request doesn't cause the backend Apache PHP process to reconnect with the database again. This is somewhat counterintuitive until we noted that there is an overhead of automatic transaction cleanup associated with the feature that the PHP MySQL driver performs [13]. Second, simulated user test run times were reduced by 27.4% when we patched Moodle's MySQL database driver to support client network connection compression, further illustrating the impact network latency has on performance, even in our modern fast networks. Given the improvement, we left the option enabled for all of our reported results and intend to submit the patch upstream.

5 Related and Future Work

5.1 Previous Performance Studies

5.1.1 Moodle

There isn't a great wealth of formal Moodle application specific performance analyses that we were able to find.

The Moodle developers post microbenchmark regression counts of the number of function calls, regex, db load/stores, etc. between major versions in the release notes, but they are not necessarily indicative of overall performance, nor do they include any cache or session subsystem details, so limited insight on different configurations can be gleaned from them. The Moodle forums have no shortage of comments and advice for performance recommendations, but very little with respect to MongoDB, nor is the suggestion of Memcached often justified beyond anecdotes.

Work by Coelho et al. [24] is the only paper on "Moodle performance" we were able to find that wasn't a study of student learning or instructor teaching achievements. In it, they simulate different load levels for different hardware, OS (Windows vs. Linux) and RDBMS (MySQL vs PostgreSQL) configurations based on some samples from a running system. However, the study was conducted with significantly smaller databases sizes than we have (400M was their "large" database, though admittedly our "large" install is not huge, it is at least 50x larger than that), and on a much older version of Moodle (1.9 in 2008) prior to Moodle including the MUC, Memcached, or MongoDB support.

For these reasons, we believe our work was still valuable.

5.1.2 Memcached

Numerous Memcached specific benchmarks and comparisons have been done that focus on making Memcached faster [52, 34].

However, while nearly every current major web presence uses some form of an in memory key-value cache like Memcached, there are relatively few studies that examine the general effect of introducing Memcached to a typical web application platform consisting of one or more web/application servers and a database server or cluster.

Bakar et al. in their evaluation of a clustered memcache [21] offers one such analysis. As expected it finds that Memcached can certainly reduce the load on the database server and thereby improve the number of requests per second the web/application servers are able to handle. However, they do not address concerns such as the use of Memcached for data that should be, at least in part, persistent, nor the effect of cold

⁴¹ We use the `noop` elevator on all of our VMs under the assumption that the arrays and/or hypervisors, which have more global knowledge, can do a better job of prioritizing disk IO themselves, so it's unnecessary for the guest, which is nearly always single purpose and has no IO competition between processes so that CFQ is unnecessary, to waste CPU scheduling IO.

cache or failures. It is interesting to note that, although they do not take failure scenarios into account, their experiments actually show a slightly higher overhead to using a clustered Memcached architecture. We guess that this is due to the extra connection management and hash computations required by the web/application servers.

Another such analysis comes from Facebook. In their work on scaling memcache [45] they introduce a number of changes to the standard Memcached to accommodate the extra load and scale that Facebook has as well as handle their concerns with failures, consistency, and at least probabilistic persistence. For instance, a client (mcrouter) handles scheduling and batching requests from web servers to Memcached servers helping to reduce the number of network connections and round trips required. They add replication to account for distributing the load for hot key requests as well as failures, and add an invalidation daemon (mcsqueal) on all database servers to handle authoritative updates to the Memcached data in order to maintain consistency, again using batching to alleviate network constraints. The replication feature also allows for a simple cache warmup mechanism. In addition to a number of other daemon modifications, they also reworked the memory allocation strategy so that data in the cache could survive a software upgrade, though not a complete systems reboot, also reducing the possibility of a cold cache scenario. However, several of their extensions, not all of which have been open sourced, required modifications to their application code. For instance, for mcsqueal to be able to invalidate upstream caches upon writes to the database layer, queries needed to include the memcache lookup key. Although their work certainly advances the Memcached technology, it does not seem directly applicable to such a comparatively small platform as ours.

5.1.3 MongoDB

There have been a number of studies on performance that have included MongoDB [47, 29, 55, 40, 53]. However, they tend to focus on whether or not MongoDB is a suitable replacement for a traditional RDBMS system, particularly for workloads such as document management, analytics, GIS data, etc. There appears to be very little data available on MongoDB when used as a cache system in front of an RDBMS.

5.2 Performance Analysis Techniques

5.2.1 Benchmarks

The NoBench [23] microbenchmark provides a means for directly measuring specific query types for a given NoSQL system. However, it currently lacks the ability to measure concurrency or scale out. Lungu et al. [41] also propose a combined NoSQL RDBMS system benchmark, currently developed for Windows, however the Yahoo! Cloud Serving Benchmark (YCSB) [25] is probably the current standard for such comparisons and supports scalability as well as simulated concurrency through the use of multiple threads on a single YCSB client.

Each of these tools is beneficial in comparing two or more products for serving the same task, but does not necessarily help us when determining their effect when used in combination with one another, each for a different subtask. Furthermore, as we did not yet know the exact workload types that Moodle used on each of the candidate systems, single system benchmark results would not have offered us very much insight, though now we could consider taking traces of some of our tests in order to help understand that relationship better.

Perhaps the closest benchmark for testing such an environment that was not Moodle application specific could be the now discontinued TCP-W benchmark [18]. However, the specification limits the tests to single systems since the focus is not on scalability. Moreover, its model is based on that of an e-commerce site that requires higher ACID compliance than the cache and session portions of our dataset that we are trying to alleviate load from.

5.2.2 Load Testing

Unlike benchmarking, which tends to stress test a particular component, load testing is focused on stress testing an actual end-to-end system. In our case a web application.

Since there are more variables at play there are many questions to consider when devising such a test. Work by Lucca et al. [26] provides a nice overview of a number of options relating to web application testing.

As mentioned in Section 3.2, tools like `ab` and `httperf` that only fetch a single URL, perhaps according to some probability distribution, do not adequately capture most client interactions with modern web applications, and so do not serve as useful load testing solutions.

Menascé [42] and Schroeder et al. [48] each offer an argument for closed or partly-open loop models that include think time for user sessions in order to accurately reflect their interactions within a web application. As Moodle interactions are generally gated by a login step, this partly-open-loop model is a much better representation of our workload, and one which we believe our JMeter tests implement well.

Menascé [42], whose work was web site load testing specific, also argues for reporting results in terms of number of abandoned user sessions due to long response times and lost revenue in addition to the typical page load response times as it varies over number of clients. Although the lost revenue figures do not directly apply to our scenario, we like the idea of incorporating the fact that users give up if a site seems to overwhelm. At best we were able to examine the error response rate in the JMeter logs we acquired, but users didn't actually drop out of the system when that happened.

In their work, Draheim et al. [27] also include the idea that multiple different classes of clients, that interact with the site in different ways, may be interacting with the system concurrently. This is certainly true in our environment as some students may be merely browsing course content (read dominant) while others could be taking a quiz (write and CPU dominant) or interacting with a forum (a mixture of read and writes). In the future we would like to be able to explore mixing different JMeter test plans to account for this.

Characterizing each of those interactions can be a difficult task in any load testing framework. Work by Jiang et al. in [32] aims to help automate that task through log mining of previously recorded load tests, though there is no reason they couldn't also be applied to production logs as well, in order to weed out anomalies that may require manual intervention as well as allowing an operator to focus on dominant behavior patterns.

Cloudstone [49] includes a nice summary of the arguments for needing to take the entire web application stack and different user interaction streams into account when testing. Though their model is based around computing the optimal platform setup for a given cost per user and SLA (e.g. 99th percentile response time) within a cloud hosted environment such as Amazon's, the techniques they employ and specific test cases they examine are very similar to those we have.

While the above focus on techniques for developing an general testing strategy that can be applied to specific instances, WikiBench [51] is an example of a specific implementation of such a test strategy that employs real data set and traffic replicas. Unfortunately, it is specific to the MediaWiki application, though can be used to test different platform configurations for it.

A subject related to load testing is correctness testing through the use of unit tests for instance, however we do not address this issue in our work.

5.3 Failure Models

As mentioned in in Section 3.3, we also tried to understand the impact of both brief component outages due to software maintenance vs. machine failures that may lead to full system outages or cold cache performance issues.

There are several other failure models we could have considered, such as network partitions or the Netflix Chaos Monkey.

However, given the comparatively small size of our system we think our simple tests involving service restarts due to system maintenance or single machine crashes are the far more likely scenario. Beyond that, the entire system is likely to be affected anyways.

Usually service outages are mitigated through replication and automatic failover and partitioning. MongoDB for instance includes this functionality natively. However, in a typical Memcached cluster, there is no replication, only key space partitioning, so failures may reduce the scope of an outage partially.

Work by Xiang et al. [54] offers a brief analysis of cache consistency and failures when using Memcached in a clustered environment. In it, they propose the use of consistent hashing combined with replication in order to tolerate failures, as well as Paxos [39] to achieve consistency between replicas. However, they do not address the performance implications of such an addition, which we expect could be substantial.

5.4 Security

In general, most NoSQL systems, including MongoDB and Memcached are designed to be operated in a trusted environment. Authentication is usually not required by default, so granular authorization is not possible. Moreover, just as their SQL predecessors, they are often vulnerable to injection attacks from clients [46].

In our general shared hosting platform we employ ModSecurity [5] to attempt to filter input for things like injection and DoS attacks before it is received by the application from the web server. However, due to the high expectation of false positives for computer science related coursework in particular, the precaution was requested to be disabled for the Moodle service.

Moreover, the filter does nothing to prevent an attack or abuse on the backend services, malicious or not, once a client is operating within the web service platform.

5.5 Additional Configurations and Technologies

In addition to MySQL, Moodle also supports PostgreSQL as a primary database. It is possible that it offers different performance characteristics than MySQL which may worth be exploring at some point. Unfortunately, there is little more than conjecture to this point on much of the `moodle.org` related sites mostly coming down to very old comments of what constitutes “properly tuned” and whether or not MySQL’s InnoDB was available for production use. Our initial impression from the literature is that PostgreSQL has traditionally been better at ensuring proper data durability and full ACID compliance, which might lead to more overhead in write intensive scenarios such as ours.

There are also other persistent NoSQL choices such as Cassandra or CouchDB that Moodle could likely be adapted to support. Both of which list some compelling resiliency features similar to MongoDB’s such as multi-master replication, no single point of failure, and per operation tunable durability.

As mentioned in Section 3.3, Couchbase is another alternative to Memcached which provides protocol compatibility but adds replication and persistence capabilities to avoid cold cache restart and repartitioning issues, though comes at a licensing cost. Memcachedb is an opensource alternative with similar advantages, but does not currently support expiration of key-values, which is important, particularly for handling session data “locks”. Yet another option we encountered was Redis, though it has similar access control issues to Memcached.

In general we note that not all data is equal and applications’ need for a combination of technologies like fully ACID RDBMS, no durability Memcached, and somewhere in the middle NoSQL solutions like MongoDB’s write concern levels represent a need for “degrees of durability”, similar in nature to Gray’s degress of consistency work [30].

The term “degrees of durability” also appears in work on the Oracle NoSQL Database [33] in which applications can specify per operation what level of durability and consistency they would like the system to provide. For instance, a write may be considered committed as soon as either a network acknowledgment from a replica or the local log disk write has been received, reducing the overall write operation’s latency to whichever of the two systems has the lower latency. This is very similar to MongoDB’s notion of write concerns.

Being able to declare such requirements for individual data transactions would make developers work easier by allowing them to maintain their data in a single system, alleviating the somewhat messy and confusing task of managing essentially the same data across multiple different storage systems and coding interfaces. As such, we were going to propose the need for a form of per-transaction relaxed durability for SQL as well. For instance, something like `SET TRANSACTION DURABILITY = [STRICT|BACKGROUND|LAZY]` that would simply change whether or not a transaction waited on the force to disk for the WAL protocol at commit time. However, we recently discovered that Microsoft's SQL Server has implemented just that feature in their 2014 release using syntax like `COMMIT TRANSACTION WITH DELAYED_DURABILITY=ON` [20]. It turns out that it was actually proposed long before that in 1986 [22].

Contrary to our previous observation, in [43] Nance et al. also offer a nice summary of the different arguments for a mix of both traditional RDBMS and NoSQL systems, as they are often designed for and solve different problems relating to both the data organization required as well as their consistency and durability requirements.

6 Conclusions

In this paper we evaluated a number of alternative SQL, NoSQL, and filesystem cache and session stores for the Moodle LMS. In both cases we found that one can achieve good performance without giving up at least partial data persistence, which can be an important property for certain types of data such as user sessions, with the differentiating factor most often being the amount of network traffic that one system or another uses. In our case, I believe we'll opt to use either the NFS `no flock nocto` or MongoDB `igbinary` cache options since they fit well into our shared environment while still providing good performance. For sessions, I believe we'll use the separately tuned session database primarily due to the graceful locking behavior it provides. Along the way we also detailed a number of simple systems improvements we found, the most interesting of which, we think was the effect that BIOS power management settings can have on system response time.

References

- [1] Bson spec faq. <http://bsonspec.org/faq.html>.
- [2] Couchbase memcached. <http://www.couchbase.com/memcached>.
- [3] Innodb integration with memcached. <https://dev.mysql.com/doc/refman/5.6/en/innodb-memcached.html>.
- [4] Memcached: Client configuration. <http://code.google.com/p/memcached/wiki/NewConfiguringClient>.
- [5] Modsecurity: Open source web application firewall. <http://www.modsecurity.org/>.
- [6] MongoDB bson. <http://docs.mongodb.org/meta-driver/latest/legacy/bson/>.
- [7] MongoDB: Journaling mechanics. <http://docs.mongodb.org/manual/core/journaling/>.
- [8] MongoDB: Replica set oplog. <http://docs.mongodb.org/manual/core/replica-set-oplog/>.
- [9] MongoDB: User privilege roles. <http://docs.mongodb.org/manual/reference/user-privileges/>.
- [10] MongoDB: Write concerns. <http://docs.mongodb.org/manual/reference/write-concern/>.
- [11] Moodle: Jmeter test plan generator. http://docs.moodle.org/26/en/JMeter_test_plan_generator.
- [12] Moodle performance comparison tool. <https://github.com/moodlehq/moodle-performance-comparison>.
- [13] Php: The mysqli extension and persistent connections - manual. <http://www.php.net/manual/en/mysqli.persistconns.php>.
- [14] Thundering herd problem. http://en.wikipedia.org/wiki/Thundering_herd_problem.
- [15] Mysql bug 2917. <http://bugs.mysql.com/bug.php?id=2917>, 2010.
- [16] Memcache benchmarks. <http://symas.com/mdb/memcache/>, 2013.
- [17] Moodle issue 40569. <https://tracker.moodle.org/browse/MDL-40569>, 2013.
- [18] Tpc-w. <http://www.tpc.org/tpcw/>, 2013.
- [19] Moodle issue 43524 - drop problematic global text caching. <https://tracker.moodle.org/browse/MDL-43524>, 2014.
- [20] What's new (database engine). [http://msdn.microsoft.com/en-us/library/bb510411\(v=sql.120\).aspx#Durability](http://msdn.microsoft.com/en-us/library/bb510411(v=sql.120).aspx#Durability), 2014.

- [21] K. A. Bakar, M. H. M. Shaharill, and M. Ahmed. Performance evaluation of a clustered memcache. In *Information and Communication Technology for the Muslim World (ICT4M), 2010 International Conference on*, pages E54–E60. IEEE, 2010.
- [22] H. Berenson. Sql server 2014 delayed durability/lazy commit. <http://hal2020.com/2014/03/06/sql-server-2014-delayed-durabilitylazy-commit/>, 2014.
- [23] C. Chasseur, Y. Li, and J. M. Patel. Enabling json document stores in relational systems.
- [24] J. Coelho and V. Rocio. A study on moodle’s performance. 2008.
- [25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.
- [26] G. A. Di Lucca and A. R. Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.
- [27] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 11–pp. IEEE, 2006.
- [28] D. A. Falvo and B. F. Johnson. The use of learning management systems in the united states. *TechTrends*, 51(2):40–45, 2007.
- [29] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the elephants handle the nosql onslaught? *Proceedings of the VLDB Endowment*, 5(12):1712–1723, 2012.
- [30] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.
- [31] S. Hemelryk. The moodle universal cache (muc). [http://docs.moodle.org/dev/The_Moodle_Universal_Cache_\(MUC\)](http://docs.moodle.org/dev/The_Moodle_Universal_Cache_(MUC)), 2013.
- [32] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 307–316. IEEE, 2008.
- [33] A. Joshi, S. Haradhvala, and C. Lamb. Oracle nosql database-scalable, transactional key-value store. In *IMMM 2012, The Second International Conference on Advances in Information Mining and Management*, pages 75–78, 2012.
- [34] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.
- [35] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *Proceedings of the 15th international conference on World Wide Web*, pages 595–604. ACM, 2006.
- [36] J. Krizanac, A. Grguric, M. Mosmondor, and P. Lazarevski. Load testing and performance monitoring tools in use with ajax based web applications. In *MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 428–434. IEEE, 2010.
- [37] B. Kroth. Cs787 project: Bin packing: A survey and its applications to job assignment and machine allocation. 2013.
- [38] B. Kroth. Cs787 project proposal. 2013.
- [39] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- [40] Y. Li and S. Manoharan. A performance comparison of sql and nosql databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19. IEEE, 2013.
- [41] I. LUNGU and B. G. TUDORICA. The development of a benchmark tool for nosql databases. *Database Systems Journal BOARD*, page 13.
- [42] D. Menascé. Load testing of web sites. *Internet Computing, IEEE*, 6(4):70–74, 2002.
- [43] C. Nance, T. Lossner, R. Iype, and G. Harmon. Nosql vs rdbms-why there is room for both. In *Proceedings of the Southern Association for Information Systems Conference*, pages 111–116, 2013.
- [44] G. Naveh, D. Tubin, and N. Pliskin. Student lms use and satisfaction in academic institutions: The organizational perspective. *The Internet and Higher Education*, 13(3):127–133, 2010.
- [45] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 385–398. USENIX Association, 2013.
- [46] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov. Security issues in nosql databases. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 541–547. IEEE, 2011.
- [47] Z. Parker, S. Poe, and S. V. Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, page 5. ACM, 2013.
- [48] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, volume 6, pages 18–18, 2006.

- [49] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.
- [50] N. Sonwalkar. The first adaptive mooc: A case study on pedagogy framework and scalable cloud architecturepart i. In *MOOCs Forum*, volume 1, pages 22–29. Mary Ann Liebert, Inc. 140 Huguenot Street, 3rd Floor New Rochelle, NY 10801 USA, 2013.
- [51] E.-J. van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. *Master’s thesis, VU University Amsterdam*, 2009.
- [52] I. Voras, D. Basch, and M. Zagar. A high performance memory database for web application caches. In *Electrotechnical Conference, 2008. MELECON 2008. The 14th IEEE Mediterranean*, pages 163–168. IEEE, 2008.
- [53] Z. Wei-ping, L. Ming-Xin, and C. Huan. Using mongodb to implement textbook management system instead of mysql. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 303–305. IEEE, 2011.
- [54] P. Xiang, R. Hou, and Z. Zhou. Cache and consistency in nosql. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 6, pages 117–120. IEEE, 2010.
- [55] J. Yang, W. Ping, L. Liu, and Q. Hu. Memcache and mongodb based gis web service. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 126–129. IEEE, 2012.