# Lessons Learnt from Software Tuning of a Memcached-Backed, Multi-Tier, Web Cloud Application

Muhammad Wajahat, Salman Masood, Abhinav Sau, Anshul Gandhi
*Stony Brook University*
{*mwajahat, smasood, asau, anshul*}*@cs.stonybrook.edu*

***Abstract*—Cloud computing has largely replaced dedicated and physical computing systems by providing critical features such as elasticity and on-demand access to resources. However, despite its many benefits, the cloud does have its limitations, such as limited or no control over the hardware and limited customization options. Users who deploy applications on the cloud only have control over software tuning and optimizations since the infrastructure is managed by the provider.**

**In this paper, we analyze cloud-deployed Web applications that are multi-tiered and employ Memcached as the object caching layer. Memcached is a high performance memory caching system and, if there are no other bottlenecks in the system, the overall application performance should be dictated by Memcached. However, we show that other components of the system such as web servers, load balancers, and some underlying system configurations, severely impact application performance. We analyze these components and provide guidelines on their implementation and parameter tuning to minimize resource waste in the cloud.**

## 1. Introduction

In the past decade, cloud computing has replaced conventional dedicated computing systems by providing on demand access to economical resources and services, including Virtual Machines (VMs). This has allowed companies, especially online service providers, to avoid upfront investment in infrastructure and instead focus on their applications.

Despite its many benefits, however, cloud computing does have its limitations. Tenants have limited or no control over the underlying hardware and its customization options, which are managed by the cloud service provider. Consequently, users who deploy applications on the cloud have to rely on *software tuning* options to maximize performance. Further, since VM placement is handled by the provider, users have to carefully deploy multi-tier applications to *avoid bottlenecks* at individual tiers that might impact the entire service chain. Without proper software tuning and bottleneck mitigation, tenants have to resort to capacity overprovisioning to meet their performance needs; such overprovisioning lowers resource usage efficiency.

Consider the popular object caching service, Memcached [1], which is widely used by online service providers (e.g., Facebook [2], [3] and Wikipedia [4]) to cache the results of database queries or API calls in memory to increase application throughput. Prior work on improving the throughput of Memcached has focused on leveraging hardware solutions (e.g., RDMA [5], GPUs [6], etc.), which are infeasible for cloud users. Further, Memcached is usually deployed as part of a service chain, such as a multi-tiered web application. We know that "a chain is only as strong as the weakest link"; while there has been a lot of research on optimizing Memcached performance (see Section 2.3), it is also critical to ensure that other links in the chain, such as web server-Memcached and load balancer-web server, are not a bottleneck.

The primary goal of this work is to investigate techniques for optimal design and configuration of a Memcached-backed multi-tiered web application deployed in the cloud. Specifically, we ask "*How can we maximize the throughput of a Memcached-backed cloud application?*"

To address this question, we explore software tuning and programming models that can be easily leveraged by cloud users. We first set up a customizable multi-tier web application complete with a load generator, load balancer, web servers, Memcached servers, and a database (Section 3). Next, we study the application performance and investigate software tuning and communication models at relevant components to mitigate bottlenecks and maximize throughput.

We use extensive experimental evaluation to find the best possible configuration of the components. We also switch from synchronous to asynchronous components one by one from upstream to downstream tiers and evaluate performance improvements. To emulate real world scenarios as closely as possible, we use the value size and popularity distributions as reported by Facebook [3]. We evaluate Apache (synchronous) vs Nginx (asynchronous) for load balancer and web server tiers, and optimize their respective configuration options.

Our experiments reveal that, with default configuration values, the other components in a Memcached service chain can significantly limit end-to-end throughput. Even after optimization, our peak application throughput is only about half of what the Memcached can achieve by itself, without other tiers in the service chain. Next, to optimize the service chain, we find that it is typically enough to consider a couple of tuning parameters at each component. Finally, when properly optimized, asynchronous components are not always superior to synchronous components; the choice

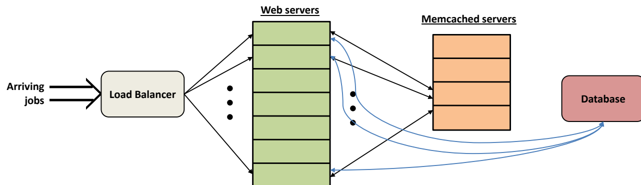



Figure 1. Illustration of a multi-tier Memcached-backed application.

between them is non-trivial and depends on the number of servers in each tier of the chain.

The rest of the paper is organized as follows. We discuss relevant background and prior work on Memcached and multi-tier Memcached applications in Section 2. We then describe our multi-tier Memcached-backed cloud web application in Section 3. Experimental results detailing our tuning and implementation efforts are described in Section 4. Finally, we conclude in Section 5.

## 2. Background and Prior Work

### 2.1. Memcached overview

Memcached [1] is a distributed in-memory caching system that efficiently scales to large memory capacities via multiple nodes due to its simple design. Memcached is a key-value (KV) store that typically sits in front of the database tier. Clients request data from the Memcached tier via a client-side library, such as libMemcached. Typical library functions include KV read (`get`) and write (`set`). The library hashes the requested key and determines which Memcached node is responsible for caching the associated KV pair. In case of a read request, the KV pair is fetched from the faster (memory access) Memcached node, if cached. Else, the client library can decide to request the KV pair from the slower (disk access) database, and optionally insert the retrieved pair into Memcached. Write requests proceed similarly; the client can choose to additionally write the KV pair to the database. Note that the client library, and not Memcached, determines which node to contact.

A single Memcached node can provide substantial throughput. In our experimental setup, a Memcached node deployed on a modestly-sized VM in OpenStack can provide in excess of a million operations/second (see Section 3).

### 2.2. Memcached-backed applications

In production systems, including Facebook [2], [3], Memcached is typically deployed as part of a multi-tier system where it sits in front of the data tier. Figure 1 shows an example multi-tier Memcached-backed application consisting of a load balancer (LB), a tier of web servers, a tier of Memcached servers, and a database. The LB distributes incoming requests to the web servers. The web servers typically parse the incoming request and determine the data needed, such as customer profiles or account information, to satisfy the request; this data is then fetched from the back-end data tier servers. Traditionally, the back-end servers store persistent data on (slow) hard drives/disks, usually in the form of databases. Memcached are servers are employed to reduce data access latency and increase throughput by caching popular data in memory (DRAM). In a cloud-deployed setting, each server would be hosted on a VM. Note that the database is connected to the web servers and not the Memcached; this is because data is requested via the client library at the web servers.

In this distributed setting, the end-to-end throughput of the application could be limited by the throughput at each of the components, and not just Memcached. For example, if the LB cannot sustain high request rates, then it becomes the bottleneck. Since Memcached can typically provide high throughput, it is important to carefully eliminate performance bottlenecks at other components to realize high end-to-end application throughput. If not, application owners may resort to expensive overprovisioning of VMs at non-Memcached tiers to increase throughput, leading to lower resource and energy efficiency at the data center level.

### 2.3. Prior work on Memcached optimization

There is much prior work on improving the performance of a single Memcached node. Most of the improvements have been realized by mitigating the network bottlenecks or addressing the shared lock in the caching system, e.g., CPHash [7] (concurrent hash table and message passing) and MemC3 [8] (smarter hashing and locking mechanisms).

Recently, efforts have been made to analyze and optimize the end-to-end performance of Memcached-backed application. Atikoglu et al. [3] analyze Facebook's Memcached deployment; their analysis reveals that Facebook's Memcached workload is read-heavy (30:1 read/write ratio) and has a moderate hit-rate of 81.4% despite power-law distributed request popularity. Hart et al. [9] study the impact of Memcached on overall site performance of Facebook by creating a Memcached performance model; the model is then used to predict throughput on sequential and parallel architectures with high accuracy. Li et al. [10] analyze the underlying causes of high tail latency in unoptimized server systems running Nginx and Memcached. They first find a theoretical baseline for performance using queueing theoretic models, and then upon finding the actual latency distributions to be much higher than the theoretical baseline, systematically identify and quantify the problem sources.

Several of the above efforts require hardware changes that are infeasible for cloud tenants. As such, it becomes critical to explore software techniques to improve end-to-end performance by focusing on all components of the Memcached-backed application.

## 3. Experimental Setup

Our experimental setup for a multi-tier Memcached-backed application closely resembles that in Figure 1. To generate load for our application, we employ httperf [11], and optimize it for high throughput. Specifically, we apply a patch for buffer overflows due to $FD_SETSIZE$ checks in $glibc$ and tune network related system parameters of the load generator VM such as $net.core.somaxconn$, $net.ipv4.ip_local_port_range$, $net.ipv4.tcp_max_syn_backlog$, etc. For the load balancer

(LB), we experiment with the popular Apache (version 2.4.7) LB running `mod_proxy` with `mpm_event` and the asynchronous event-driven Nginx LB (version 1.4.6). For the web server, we again experiment with the Apache web server running `mpm_prefork` with `mod_php` to handle PHP content and Nginx with PHP-FPM [12] to serve PHP content using FastCGI protocol; we use PHP version 5.5.9 for both cases. For the Memcached tier, we use Memcache version 1.4.31. To communicate with Memcached, the web servers employ the libmemcached library. Finally, for the database, we employ ardb [13] (version 0.9.3) which uses the Redis protocol for communication and leverages RocksDB [14] as the backend.

In terms of the KV data set, the key size is fixed at 11 bytes and the value sizes range from 1 byte to 100 bytes. The value sizes are distributed geometrically, meaning that smaller KV pairs are more popular, as observed by Facebook [3]. The data set contains a 100 Million KV pairs.

Our application is a simplified web service that serves PHP jobs. Each job arrival triggers a request for one hundred KV pairs (`get` requests). After parsing the request, the web server issues a multi-get to the Memcached tier for the KV pairs; note that several Memcached nodes might have to be contacted to serve all KV pairs. In case of a miss, the web server requests the KV pair from the database and then inserts the KV pair back into Memcached, possibly leading to evictions. We define *response time* to be the time elapsed from when a request arrives at the LB to the time that it is successfully served after fetching all requested KV pairs.

We deploy our application on modestly-sized VMs hosted by an OpenStack cluster to mimic a cost-efficient deployment. All of our VMs run Ubuntu 14.04.3 OS with Linux kernel version 3.13. The load generator and three web servers are deployed on 4-vCPU, 8GB (m.standard) VMs. Three Memcached servers are deployed on 2-vCPU, 4GB (m.milli) VMs, together providing a hit rate in excess of 90% because of the skewed popularity distribution exhibited by real web applications [3], [15]. The LB that handles all incoming requests is deployed on a 8-vCPU, 16GB (m.kilo) VM. To limit our focus to components upstream of Memcached, we deploy the database on a dedicated physical machine with 8 cores and 32GB memory; the database is tuned to avoid any bottlenecks. In particular, we set an appropriately sized block cache for RocksDB, maximize the $ulimit$ setting, and set the $fs.file - max$ to a high value. We also use persistent connections between the web servers and database to reduce connection overheads.

# 4. Evaluation Results

We now discuss our software tuning and implementation efforts to improve the throughput of our application. Most of our efforts are directed at the LB and web server tiers. We start with OS-level tuning in Section 4.1, followed by process-level tuning in Section 4.2. Finally, we discuss the impact of the communication model (sync versus async) and our key results in Section 4.3. For all experiments in this section, we report the average values across three runs.

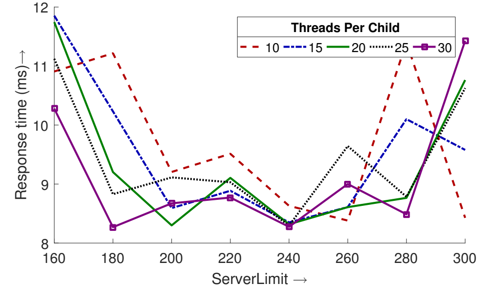


Figure 2. Tuning the Apache LB *ThreadsPerChild* and *MaxRequestWorkers*.

## 4.1. OS-level tuning

For the LB and web server, we find that the most important parameters are network related, since they have to handle several connections simultaneously. As such, we set net.ipv4.tcp_tw_recycle and net.ipv4.tcp_tw_reuse to 1 to allow sockets to be recycled and reused. Without these settings, several sockets may linger in the TIME-WAIT state. We also maximize the net.ipv4.ip_local_port_range (1024–65535) to allow several simultaneous network connections. For the same reason, we also maximize ulimit to 65535.

## 4.2. Process-level tuning

**Apache load balancer:** Our Apache LB uses the `mpm_event` module for processing incoming connections. In particular, under this configuration, Apache acts as a multi-process multi-threaded LB. The number of processes is specified via the *ServerLimit* directive. Each process in turn creates a number of server threads that handle requests; this number is specified via the *ThreadsPerChild* directive. The product of *ServerLimit* and *ThreadsPerChild* is limited by the *MaxRequestWorkers* directive which represents the maximum number of simultaneous requests that can be served by the LB. If a new request arrives while all threads are busy, it will be queued. While there are other configuration parameters to tune, we find that *ServerLimit* and *ThreadsPerChild* suffice for process tuning; *MaxRequestWorkers* can be set as the product of these two parameters.

In general, a high *ServerLimit* implies higher parallelization, but also higher overhead. Specifically, as *ServerLimit* increases, the web server can handle more requests simultaneously. However, as *ServerLimit* increases, the memory and context switching overhead also increases. Likewise, a high *ThreadsPerChild* implies higher parallelization and higher overhead, though the memory overhead should not increase since threads share the same address space.

We vary the *ThreadsPerChild* from 5 to 50, and vary the *ServerLimit* from 100 to 500. Figure 2 shows a subset of our results for a throughput of 100K ops/sec. We see that, as expected, a very low or very high *ServerLimit* can negatively impact performance. The impact of *ThreadsPerChild* is not that clear. While in this figure it might seem that a higher *ThreadsPerChild* is more beneficial, we found this not to be the case for higher *ServerLimit* and higher throughput. In general, we find that the best setting for *ThreadsPerChild* is 15 and that for *ServerLimit* is 240.

**Apache web server:** The Apache web server uses the

Figure 3. Tuning the *MaxRequestWorkers* for Apache web server.

`mpm_prefork` module to handle PHP requests (PHP is not thread safe, so we cannot use `mpm_event`). Under this module, the *MaxRequestWorkers* controls the number of simultaneous requests that can be handled; other parameters do not affect performance significantly (*ThreadsPerChild* and *ServerLimit* are not relevant for `mpm_prefork`). Figure 3 shows our results for *MaxRequestWorkers* under 1 Apache web server and 1 Memcached server. We see that as long as *MaxRequestWorkers* is not too small, response time is low; we set *MaxRequestWorkers* to 500.

**Nginx load balancer:** Nginx also uses multiple processes, like Apache, but each process can simultaneously work on many connections in parallel, with the limit per process set by *worker_connections*. Each Nginx process asynchronously handles multiple connections by processing events across connections using an event loop.

We set the number of Nginx processes to be equal to the number of vCPUs at the LB VM, that is, 8. We vary the *worker_connections* parameter to determine its best value. Figure 4 shows our results for response time as a function of *worker_connections* for our application at different throughputs; here, we use 1 web server and 1 Memcached server. We see that performance initially improves as *worker_connections* increases, due to increase capacity for handling multiple requests simultaneously. However, beyond a point, performance worsens, likely due to the overhead of too many *worker_connections*. Based on this result, we set *worker_connections* to 1500.

In the above results, we use epoll (for efficient I/O utilization) only on the Nginx LB. Figure 5 compares the performance as a function of *worker_connections* for a throughput of 280K ops/sec under epoll at LB (epollLB) and epoll at LB and web servers (epollEV). We see that under epollEV, the optimal *worker_connections* is now around 2100. Thus, the optimal value for a parameter can depend on other parameter and/or configuration values.

**Nginx web server:** The Nginx web server process can handle http content but not PHP requests. Instead, we install the PHP-FPM (FastCGI Process Manager) [12] to handle PHP content. When the Nginx web server gets a PHP request, it forwards it to FPM through a socket, and receives the web page in html format after PHP is executed. PHP-FPM's key parameter is the *pm.max_children*, which controls how many FPM child processes are running. Thus, we now have to optimize Nginx and FPM parameters.

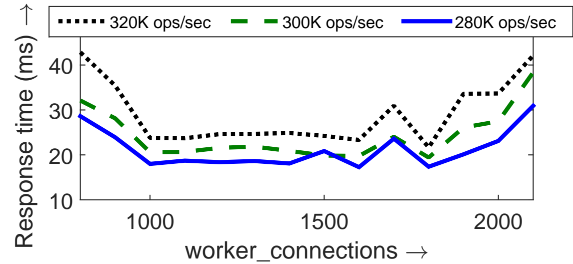


Figure 4. Tuning the *worker_connections* for Nginx LB.

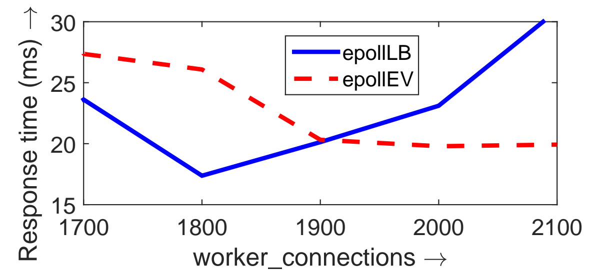


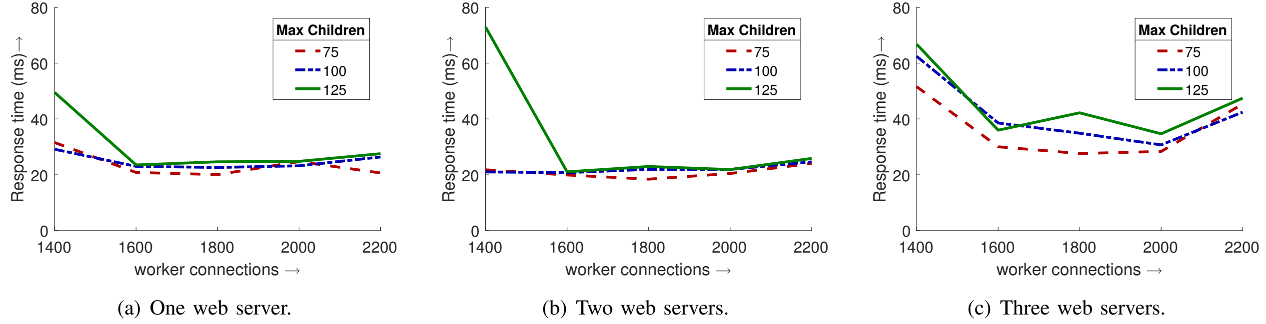
Figure 5. Tuning the *worker_connections* for Nginx LB when epoll is selected for LB and web servers.

For the Nginx web server, we again set the number of Nginx processes to be equal to the number of vCPUs, which is 4 for the web server VMs. We then vary the *pm.max_children* and *worker_connections* parameters at the Nginx web server. Figure 6 shows our results for different number of web servers and 1 Memcached server under high load. Under this load, the throughput is 320K ops/sec, 420K ops/sec, and 500K ops/sec, respectively, for 1 web, 2 web, and 3 web servers. For all plots, we see that *pm.max_children* of 75 provides low response times, likely due to low overhead. Likewise, *worker_connections* of 1800 provides a good balance between overhead and parallelization, resulting in low response times. These are the settings we use for Nginx web servers. We run the same experiment under moderate load as well (throughput of 280K ops/sec, 360K ops/sec, and 380K ops/sec, respectively, for 1 web, 2 web, and 3 web servers); results are qualitatively similar and are thus omitted.

### 4.3. Main Result: Sync v/s Async connection model

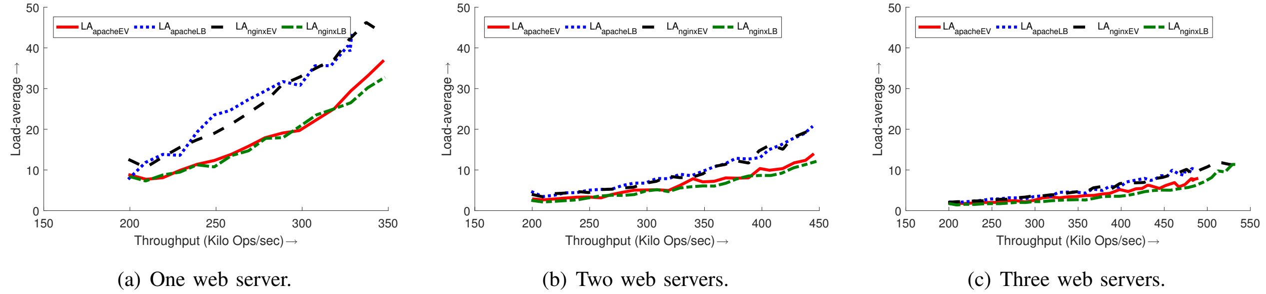
We now discuss the key results of this paper. Given the different communication models of Apache (sync) and Nginx (async), an important question is to evaluate their performance. Given the multi-tier nature of our application, we consider four deployment options: Apache LB + Apache web server (apacheEV), Apache LB + Nginx web server (apacheLB), Nginx LB + Nginx web server (nginxEV), and Nginx LB + Apache web server (nginxLB). Intuitively, we expect the async Nginx to perform better than Apache given that our goal is to maximize throughput, and thus the application will experience high request rates; this hypothesis is supported by prior work [16].

Figures 7 and 8 show our experimental results for load average and response time, respectively, under four different LB + web server configurations. Here, we vary the number of web servers and use only 1 Memcached server. From left to right, we see that the peak throughput (x-axis) increases



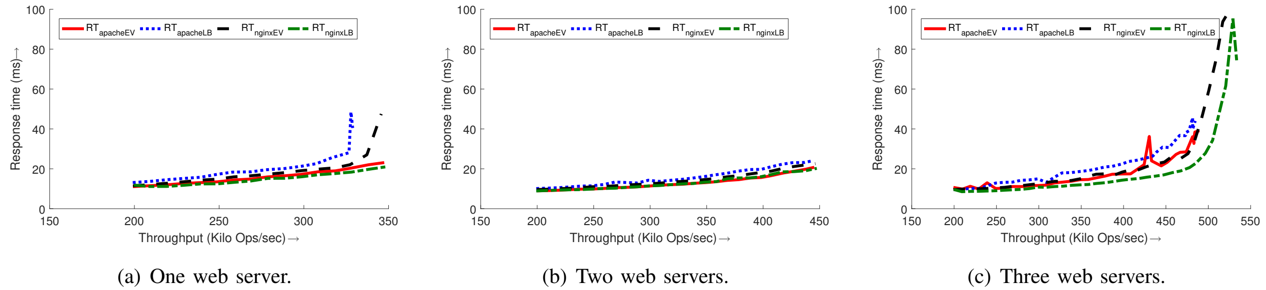

(a) One web server. (b) Two web servers. (c) Three web servers.

Figure 6. Response time as a function of *worker_connections* of Nginx web server for different FPM *pm.max_children* values under high request rate.




(a) One web server. (b) Two web servers. (c) Three web servers.

Figure 7. Load average versus throughput comparison for different tier configurations, for 1, 2 and 3 web servers and 1 Memcached server.




(a) One web server. (b) Two web servers. (c) Three web servers.

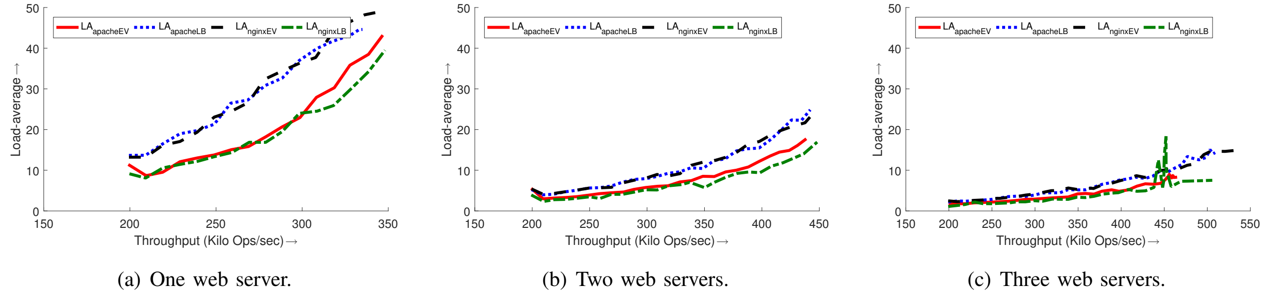
Figure 8. Response time versus throughput comparison for different tier configurations, for 1, 2 and 3 web servers and 1 Memcached server.

from 350K ops/sec to 450K ops/sec to almost 550K ops/sec, while the load average and response time largely decrease (except for the highest throughput values under three web servers). This suggests that the bottleneck is the number of web servers. Our application achieves a peak throughput of around 0.55 million ops/sec after OS- and process-level tuning; the configuration that achieves this throughput is nginxLB (Nginx LB + Apache web servers). By comparison, we easily surpassed 1 million ops/sec when using only 1 Memcached server (no LB or web servers) driven by the Mutilate memcached load generator [17]. This shows that it is not enough to simply optimize the Memcached tier since the end-to-end application throughput can be much lower.

It is interesting to note that the configurations that achieve low load averages and response times, apacheEV and nginxLB, both have Apache as the web server. This is non-trivial since we expect the async Nginx to outperform Apache. We believe that the reason for the worse-than-expected performance of Nginx web server is the communication overhead between Nginx and FPM via unix sockets. Apache, on the other hand, has a `mod_php` interpreter as a module that reduces this overhead. Note that Nginx, when using FPM, is still asynchronous; it forwards the PHP request to the FPM and continues handling other connections without blocking. Thus, despite its async nature, Nginx may not be the best choice as a web server; Apache (the newer versions) continues to perform well as LB and web server.

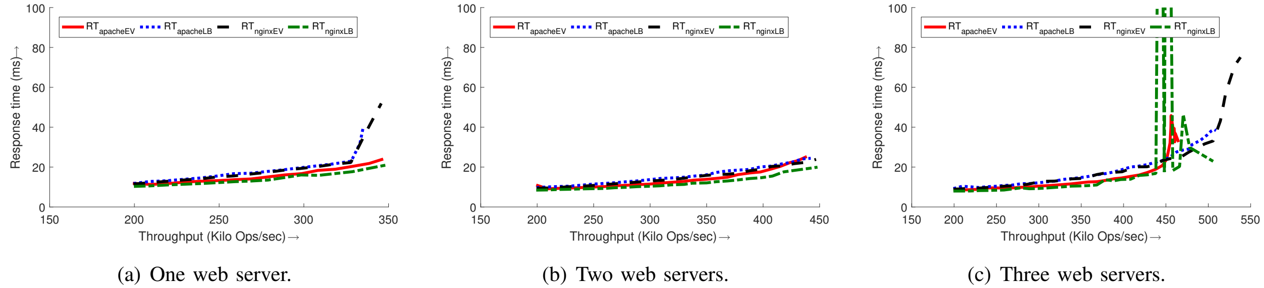
Figures 9 and 10 show our experimental results for load average and response time, respectively, when using 3 Memcached servers. Generally speaking, these results follow the same trends as for the 1 Memcached server configuration. Note that the peak throughput does not change much, validating our hypothesis that Memcached is not the bottleneck. Upon close observation, we find that the performance is slightly worse (about 10%) under 3 Memcached servers than under 1 Memcached server. This is because under 3 Memcached servers, each web server now has to maintain a lot more open connections with the Memcached tier, resulting in higher overhead. Our results for 2 Memcached servers (not shown due to lack of space) confirm this trend.

An interesting observation for the 3 Memcached servers setup is the case of 3 web servers in Figures 9(c) and 10(c). Here, the usually superior apacheEV and nginxLB both perform poorly and do not achieve the high throughput that nginxEV does. Further, as they approach the high throughput




(a) One web server. (b) Two web servers. (c) Three web servers.

Figure 9. Load average versus throughput comparison for different tier configurations, for 1, 2 and 3 web servers and 3 Memcached servers.




(a) One web server. (b) Two web servers. (c) Three web servers.

Figure 10. Response time versus throughput comparison for different tier configurations, for 1, 2 and 3 web servers and 3 Memcached servers.

values, the load average and response times increase sharply. We believe that the reason for this discrepancy is the high CPU overhead of `mpm_preform` that uses 500 processes for Apache web. Combined with the high number of connections maintained by the web servers to the 3 Memcached servers, likely results in lower-than-expected throughput. Note that this is not the case for 1 and 2 web servers.

In summary, while we are able to obtain an end-to-end throughput of about 0.5 million ops/sec, there is no single optimal configuration as the optimal depends on the number of web servers and the number of Memcached servers.

## 5. Conclusion

This paper evaluates the software tuning of multi-tier Memcached-backed applications, such as modern web applications, deployed on the cloud. Without tuning, the end-to-end throughput of such applications is much lower than that achieved by the Memcached tier, suggesting bottlenecks in the remaining service chain. Our experiments reveal that tuning the number of worker threads at the load balancer and web server tiers can significantly improve throughput. Importantly, for our specific setup with Apache and Nginx, we show that an asynchronous communication model at the web server does not always improve throughput.

## Acknowledgments

This work was supported by the U.S. National Science Foundation under grants CNS-1622832 and CNS-1617046.

## References


[1] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 2004, no. 124, pp. 5–5, Aug. 2004.

[2] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *NSDI 2013*, Lombard, IL, USA, pp. 385–398.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *SIGMETRICS 2012*, London, England, UK, pp. 53–64.

[4] mediawiki.org, "memcached," http://www.mediawiki.org/wiki/Memcached, 2014.

[5] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy Nodes with 10GbE: Leveraging One-sided Operations in soft-RDMA to Boost Memcached," in *USENIX ATC 2012*, Boston, MA, USA.

[6] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems," in *ISPASS 2012*, New Brunswick, NJ, USA, pp. 88–98.

[7] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek, "CPHASH: A Cache-partitioned Hash Table," in *PPoPP 2012*, New Orleans, LA, USA, pp. 319–320.

[8] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *NSDI 2013*, Lombard, IL, USA, pp. 371–384.

[9] S. Hart, E. Frachtenberg, and M. Berezecki, "Predicting Memcached Throughput Using Simulation and Modeling," in *TMS/DEVS 2012*, Orlando, FL, USA, pp. 40:1–40:8.

[10] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," in *SOCC 2014*, Seattle, WA, USA, pp. 9:1–9:14.

[11] D. Mosberger and T. Jin, "httperf—A Tool for Measuring Web Server Performance," *ACM Sigmetrics: Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[12] "PHP-FPM - A simple and robust FastCGI Process Manager for PHP," https://php-fpm.org/.

[13] "ardb," https://github.com/yinqiwen/ardb.

[14] "RocksDB — A persistent key-value store," http://rocksdb.org/.

[15] N. Sharma, S. Barker, D. Irwin, and P. Shenoy, "Blink: Managing server clusters on intermittent power," in *ASPLOS 2011*, Newport Beach, CA, USA, pp. 185–198.

[16] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations," in *ICDCS 2017*, Atlanta, GA, USA.

[17] "Mutilate: high-performance memcached load generator," https://github.com/leverich/mutilate.