Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier

Yu-Ju Hong and Mithuna Thottethodi
Purdue University
{yujuhong, mithuna}@purdue.edu

Abstract

Distributed memory caching systems (e.g., memcached) offer tremendous performance improvements for multi-tiered applications compared to architectures that directly access the storage layer. Unfortunately, the performance improvements are artificially limited by load imbalance in the memcached server pool. Specifically, we show that skewed key popularity induces significant load imbalance, which in turn can cause significant degradation in the tail (i.e., $90+^{th}$ %ile) latency. Based on this understanding, we design and implement `SPORE` – an augmented memcached variant which uses self-adapting, popularity-based replication to mitigate the effects of such load imbalance. `SPORE` uses *reactive internal key renaming* as a basic mechanism to efficiently achieve replication without excessive communication and/or coordination among servers and clients. Further, our `SPORE` design offers the same consistency model (with added time-bounds on write propagation) as a system with memcached. Based on evaluations on a "wimpy-node" testbed and on Amazon EC2, we show that `SPORE` achieves significantly higher performance than the baseline memcached.

1 Introduction

The use of a memory-caching tier between the web/business-logic tiers and the storage tier is increasingly common because of the significant performance advantages it offers. Memcached [13], in particular, is a widely-used distributed memory cache design. It is a key component of the infrastructure at Facebook, Youtube, and Zynga, among others. Facebook attributes a 10X performance improvement to the use of a memcached based caching tier [21]. Cloud platforms such as Amazon Web Services and Google App Engine also offer APIs for memcached services.

At a high level, memcached offers a basic key-value caching architecture. Within each server, memcached implements a simple in-memory hash-table. Pools of memcached servers serve large key-spaces by sharding the key space. As a consequence of this architecture, any access proceeds as a two stage lookup. The first stage uses hashing to identify the server, and the second stage uses traditional hash-table lookup in memory. Ideally, we want the load across the memcached servers to be balanced. However, we show later in Section 2 that, primarily because of non-uniformities in key popularity, there is significant load imbalance across the servers of a memcached pool. The load imbalance causes significant degradation in tail latency (e.g., latency of the 90^{th}%ile lookup). Because a single page-rendering request at the application level leads to fetching of hundreds of objects (as reported by Facebook [31]), increasing tail latency directly impacts application performance. Finally, we note that addressing the tail latency is not amenable to easy solutions. For example, increasing the number of servers linearly improves average load per server. However, the impact on tail latency is not as significant especially if there is load imbalance because of differences in key popularity. (Moreover, expanding the size of a memcached pool is not always an option as the increased "fanout" worsens the problem of incast congestion [31].)

This paper addresses the challenge of improving load balance in the memory caching tier. To that end, we design and develop an augmented variant of memcached called `SPORE` to proactively balance load among memcached servers via **S**elf-adapting **PO**pularity-based **RE**plication (`SPORE`). `SPORE` achieves either better performance with the same number of servers or the

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1.
http://dx.doi.org/10.1145/2523616.2525970

same performance with fewer servers. Prior work has examined techniques to reduce the load on the memcached layer by caching the most popular keys [11]. Such load reduction does not obviate the need for balancing the remaining load which will still have significant popularity disparity across keys.

Our design goals for SPORE are threefold: First, preserving the simple architecture of memcached, wherein each server is a standalone server is an important goal. It is not our goal to develop a complicated distributed system that achieves global load balancing via global coordination as that would destroy the ease-of-use, maintainability, and scalability of memcached. Second, we wish to achieve better load balance while incurring minimal space and performance overheads in order to preserve performance benefits of load balancing. Finally, we wish to offer the same consistency as memcached.

We satisfy the first design goal by allowing each server to identify and replicate popular key-value tuples individually and using a technique called *reactive internal key renaming (*RIKR*)* for key-to-server mapping. This enables the server and the clients to locate replicas by convention, rather than by explicit communication. A popular key-value tuple $<k,v>$ is said to be renamed if a client/server can map another key k' to a server s' and access the same tuple $<k,v>$ at server s'. Because the key-server mapping is achieved by hashing, the renamed key k' will likely be mapped to a server s' that is different from the server that key k was originally mapped to, provided the cluster of servers is reasonably large. As long as the server and the clients are pre-configured to rename a key using the same method, no additional communication and/or a central proxy is required to find the location of a replica. Note that this renamed key k' is only used to find server s'; it is not used in the client-server communication, nor is it exposed externally to the backend store and to the external clients.

We maximize load balance with minimal overheads by identifying what, when, and where to replicate through sampling and selective hotspot triggering mechanisms. Each SPORE server uses sampling to selectively and adaptively identify popular tuples to replicate. Sampling not only filters out noise (accesses to unpopular keys) but also avoids extra bookkeeping for every request. Our technique achieves the same performance as an impractical variant of SPORE with perfect knowledge of key popularity and read/write ratios. Furthermore, we use a hotspot-triggering mechanism to activate sampling and replication only on heavily loaded servers. This allows SPORE to achieve better load balance, without global coordination, by relieving the load at the hotspots through replication and preventing unnecessary replication (and bookkeeping overhead) at lightly-loaded servers. Therefore, SPORE does not need to rely on keeping a high number of replicas (i.e., high overhead) to achieve nearly the same level of load balance.

Finally, the consistency offered by a memcached pool depends on the context. Seen as a standalone system (i.e., with no underlying persistent store), memcached offers linearizability when there are no server failures. Such strong consistency is possible because, in the absence of any memcached replication, writes to a single copy in memcached are inherently atomic. However, this view of memcached as a standalone system is not meaningful because server failures are not uncommon and such failures can cause data loss (only writes to persistent stores are durable). Further, with a realistic system view wherein a memcached pool is part of a larger system and caches tuples from an underlying persistent datastore, implementing stronger consistency can be challenging because of the difficulty of writing atomically to the memcached tier and the underlying datastore (while allowing for arbitrary machine failures).

We demonstrate two variants of SPORE that vary in write-atomicity within the memory caching tier. The primary variant, SPORE, is designed to match the consistency of widely-deployed memcached systems (similar to the Facebook architecture [31]) wherein writes are not atomic and the system as a whole offers weaker, eventual consistency. We also demonstrate a second variant – SE−SPORE – which trades off a fraction of SPORE's performance benefits to enforce write-atomicity across the replicas that we create in the memory caching tier. SE−SPORE is thus equivalent to the standalone memcached in consistency.

We evaluate the performance of our implementation of SPORE on a *wimpy node* testbed as well as on a larger-scale Amazon EC2 cluster. Results show that the SPORE, with 12 nodes, achieves the same performance (throughput improvement at the same 90^{th} %ile latency) as a 16-node baseline memcached. Further, SPORE shows performance benefits for read-only, read-mostly, read/write, and mixed workloads.

In summary, the major contributions of this paper are:

- We identify the key sources of load imbalance and quantify the impact of load imbalance on the tail of the latency distribution.
- We design and implement SPORE which uses replication to achieve better load balance while preserving the key advantages of memcached. A 12-node SPORE pool achieves the same performance as that of a 16-node memcached pool.

The rest of this paper is organized as follows. Section 2 provides a brief background on the operation of memcached, the sources of load imbalance, the opportunity for performance improvement, and the consistency models. Section 3 describes the design of SPORE. Section 4 describes our experimental methodology and Section 5 presents our results. Section 6 briefly discusses possible limitations of our design. Section 7 discusses related work on load balancing and consistency models. Finally, Section 8 concludes the paper.

2 Background

We provide a brief background on the basic operation of memcached and its typical access characteristics. We then quantify the performance degradation due to load imbalance; which can be interpreted as the opportunity for our dynamic replication techniques to reclaim. Finally, we provide a brief background on consistency models.

2.1 Basic Operation of Memcached

Memcached supports the basic dynamic set operations of storage, (set(k,v), add(k,v), replace(k,v), increment(k), etc), retrieval (get(k)), and deletion (delete(k)). In the rest of the paper, we use the term *read (requests)* for retrieval operations and *write (requests)* for storage operations. The deletion operation simply deletes all copies of the tuple in the system, thus we do not discuss it in particular in the following sections. The memory caching tier is a pool of such memcached servers, each of which is a standalone server that need not be aware of the other servers in the pool. Each server is responsible for a shard of the key space. Such sharding is realized by using a hash function that maps keys to individual servers. Table-lookup-based techniques are unsuitable because of large, sparse key spaces (the typical motivation for the use of hashing). To ensure minimal remapping of keys to servers when the number of servers change (e.g., when servers are added/removed to/from the pool), memcached uses consistent hashing [18]. Because clients are responsible for directing accesses to the appropriate shard, they are aware of the number of servers in the pool. Note, the memcached clients are entities that are entirely within the organization behind the web-tier (and not at the browser/end-user end). However, requests from a single end-user are directed through a given client which implies that ensuring ordering at a client also ensures ordering is seen by the end-users. Within an individual memcached server, the storage is effectively a hash-table with chaining to handle collisions.

Access Characteristics. The typical workload for memcached is read-mostly as memcached can only speed-up reads. Memcached cannot filter writes from the underlying (non-volatile) datastore because memory-

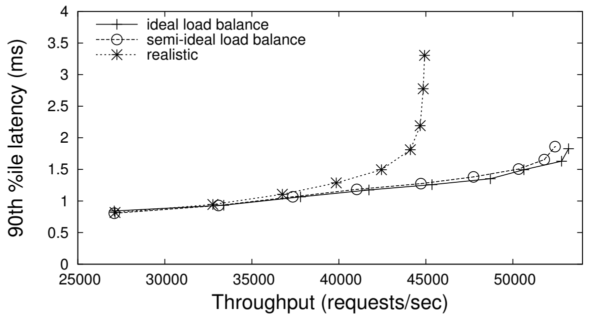


Figure 1: The 90th %ile latency and achieved memcahced pool throughput. Multiple points for each curve acquired by varying the number of clients.

writes are not durable. Because it is important to ensure durability of writes, a write may be deemed to be complete only if the write to the underlying datastore is also complete. Previous studies have confirmed the read-mostly nature of memcached workloads. For example, Facebook reports reads are two order of magnitudes higher than writes [12]. Others report that writes constitute from 3 –12% of all accesses [2, 10]. Previous research has characterized the popularity distribution of keys as Zipfian with an α value between 0.7 and 1.01 [11, 4, 34].

2.2 Understanding load imbalance and quantifying the opportunity

The load on each server is driven by two factors; the number of keys mapped to that server, and the popularity of those keys. Ideally, we want the load to be balanced across all servers. Unfortunately, there are imbalances in each of the two factors. First, there is some non-uniformity in key distribution even when using good hash functions for the key-to-server mapping. Second, the popularity of keys is quite skewed; possibly Zipfian. Load imbalance among servers can be a critical concern in such memcached pools because overloaded servers may see latency degradation. While the increased latency on a small subset of servers is unlikely to significantly affect average or median latency, the tail latency (e.g., 90th %ile latency) can degrade significantly.

Fig. 1 plots the 90th percentile latency (Y-axis) achieved by a pool of 12 memcached servers exposed to various load-levels. Note, the X-axis plots achieved system throughput (rather than applied load) which is a dependent variable. We use three configurations (corresponding to the three curves) to isolate the impact of each source of imbalance. The first curve (labeled *ideal load balance*) uses perfect (and impractical) key distribution in which every server has the same number of keys. Further, the popularity of various keys follows a uniform random distribution. The curve effectively represents the ideal load-balanced case as both keys per server and key popularity distribution are uniform. As load increases, the memcached layer eventually saturates and latency begins to degrade (i.e., increase) because of queuing delays. The second curve (labeled *semi-ideal load balance*) retains uniform popularity distribution, but uses a realistic hash function to map keys to servers. This configuration incurs some load imbalance because of non-uniform key-to-server distribution. We observe that for any given latency, the configuration achieves lower throughput. Finally, the third curve (labeled *realistic*) includes the effects of both popularity skew (with Zipfian $\alpha = 0.99$ key popularity) and imperfect key-to-server mappings (because of hashing). We observe that the performance is much worse (i.e., sustaining lower throughput while maintaining acceptable latency).

We make two observations from Fig. 1. First, we see that the degradation due to non-uniform (Zipfian) key popularity is the major contributor to performance degradation. Second, the gap between the ideal curve and the realistic curve represents an upper-bound on opportunity for our techniques, which we describe next.

2.3 Consistency Models

Informally, consistency refers to the ordering of reads and writes emanating from multiple threads/processes as seen by one another. If a consistency model is intuitive and well-defined, it is easier for programmers/users to reason about system behavior. In general, stronger flavors of consistency require memory operations to be globally ordered such that two key properties are true: (1) writes appear atomic, and (2) the perceived global order respects the local program order. For example, sequential consistency [22] requires that all writes are atomic (or appear atomic) and all reads/writes occur (or appear to occur) in program order. Linearizability, a flavor stronger than sequential consistency, requires atomic writes to be further constrained between physical time markers [17].

Our goal is to ensure that `SPORE` achieves the same consistency as baseline memcached. To that end, we provide a brief description of write atomicity in memcached and in `SPORE`, respectively.

A *standalone* pool of memcached servers (ignoring the persistent store and ignoring server failures) offers write-atomicity because clients perform operations one-at-a-time and because every write is atomic since there is a single copy of any tuple in the memcached pool. However, when considering real-world applications wherein memcached pools operate in conjunction with the backend storage, such write-atomicity across both tiers (i.e., atomic execution of both the write to the backend store *and* the update/invalidate of the copy in memcached) is more challenging especially when machine failures are also considered. Consequently, many deployed systems that include memcached do not offer linearizability [31]. Rather, such systems offer a weaker *eventual consistency*. Our design of `SPORE` leverages this fact to further relax write-atomicity even within the memory-tier while retaining the same consistency model at the system level (i.e., system-equivalence). Specifically, `SPORE`'s design uses multiple replicas of the same tuple in the memory caching tier and does not enforce write-atomicity when updating the replicas in the memory caching tier.

`SPORE` relaxes write-atomicity, but it matches baseline memcached in the other consistency properties. Specifically, `SPORE`, like memcached, offers read-monotonicity, which guarantees that a write once seen, may not be unseen. Further, a writer always sees its own writes immediately after the operation is complete. While the above guarantees are trivial in memcached in the absence of replication, ensuring the same guarantees in `SPORE` with replication and with non-atomic writes requires careful design, as we explain later in Section 3.

`SPORE`'s benefits do not fundamentally depend on relaxing write-atomicity. We demonstrate a variant of `SPORE` called `SE-SPORE` which enforces write-atomicity and is thus equivalent to a standalone version of traditional memcached. `SE-SPORE` still achieves performance improvements over traditional memcached because of improved load balancing.

3 SPORE Design

When there are no popular tuples, our `SPORE` behaves very similar to the baseline memcached system, where each key of a tuple is associated with one memcached server by using consistent hashing. This server is called the *home* server of the tuple.

Functionally, we want the following behavior when mostly-read tuples become popular. We would like those tuples to be replicated on additional servers. We refer to the servers with a replica as shadow servers. Read requests must be distributed to all replicas while ensuring that requests from a single client are directed to a single replica (requests from a single client are not load-balanced across multiple replicas to ensure read-monotonicity; as explained in Section 2). Write requests, on the other hand, must always go to the home server to preserve a sequential write order as the home server acts as an ordering point.

We describe the design of `SPORE` as answers to the following questions.

1. What is the mechanism that enables replication of tuples on shadow servers such that all clients can

discover the location of the replica(s)?

2. How does SPORE identify tuples to replicate?
3. How are clients notified of the creation and destruction of replicas? How do clients distribute load across replicas?
4. What are the policies that determine when replicas are created/retired?
5. How are writes handled? What is the consistency model?
6. How does SPORE handle server failures?

3.1 Reactive Internal Key Renaming (RIKR)

All clients and servers need to agree on where a key-value tuple should be replicated to. We define the number of replicas (not including the original tuple at the home server) as the *replica count*, γ.

Replication requires two key steps. First, the tuple must be replicated from the home server to one or more shadow servers. This is achieved by server-to-server communication. We choose the shadow servers using our RIKR technique. RIKR appends a suffix to the key for the limited purpose of determining the shadow servers. This suffix ranges from $1, 2, \ldots, \gamma$. Fig. 2 shows an example of key `foo` with $\gamma = 2$. The key `foo` can be renamed to be `foo1` and `foo2` resulting in a total of three copies. Each server is augmented with a "stub" that uses the same consistent hashing function as the clients (see Section 4 for details). This approach retains the design wherein each server is a standalone server. However, it does result in memcached servers being aware that there are other memcached servers in the pool. The home server identifies shadow servers by using the key-to-server mapping functionality (i.e., consistent hashing) in the stub in conjunction with the renamed key values. The home server then issues a `set(k,v)` to the shadow servers. Note that the renamed keys are used only for identifying the shadow server. Subsequent operations use the original (non-renamed) key at the shadow server instead of the renamed key to rule out the possibility of any aliasing. For example, clients `c2`, `c3` and `c4` (see Fig. 2) determine the server by using renamed keys `foo1` or `foo2`; but the actual operation uses the original key `foo`. Consider the case where a real tuple with key `foo1` exists. For correctness, that tuple should not be aliased with the renamed version of key `foo`). SPORE's design ensures that even though operations on a true key `foo1` and a renamed key `foo1` (original key `foo`) will be placed on the same server, they will not be aliased because the operations use the original keys.

The second step requires clients to discover that replicas exist. One advantage of RIKR is that there is minimal

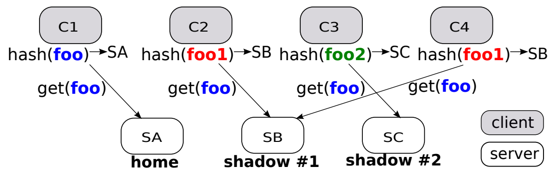


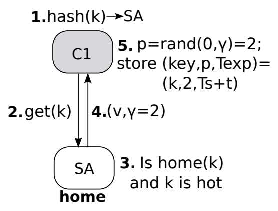
Figure 2: An example of how clients access replicas using our reactive internal key renaming (RIKR) scheme for a hot key `foo` with $\gamma = 2$.

overhead in conveying the location of replicas to clients. The server needs to communicate only γ to the clients (which is piggy-backed on existing communication, as we explain below). The client can then independently generate the names of all replicated tuples and locate them using the underlying key-to-server mapping.

3.2 Replica Discovery and Retirement

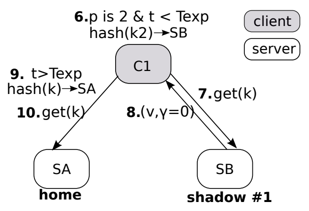
Clients discover the presence of replicas when they see non-zero γ in piggy-backed communication from the server during routine operation. (Steps 1–4 in Fig. 3(a).) The client, after receiving this information, chooses a replica for further accesses to this key by randomly choosing a suffix from 0 (the original copy) to γ, if $\gamma > 0$. If the suffix is non-zero (recall, $\gamma = 0$ corresponds to baseline memcached), the key and the suffix are stored in a client-side metadata (CSM) cache to ensure that future accesses to the key go to the same replica. (Step 5 in Fig. 3(a).) Note, to facilitate read-monotonicity, the suffix is randomly chosen exactly once for the client; not for every access at the client. As such, our goal is to balance the loads of different clients on multiple replicas while still sending all requests from a single client to a single replica. Before sending out a read request, the client looks up the CSM cache to check if the tuple has been replicated. If a tuple has been replicated, it appends the chosen suffix to the key and identifies the server to send the requests to. Evicting any entry (without deleting the replica) in the CSM cache will not cause any correctness problems because the client will simply revert to sending requests to the home server.

Finally, there are several ways in which the clients discover the retirement of replicas. First, we assign a predefined, short expiration time T_s for entries in the CSM cache. This short expiration time can be thought of as a short-term lease granting the client access to the replicas. As mentioned earlier, when receiving a new lease, the client will randomly choose one replica to send requests to. This decision is honored for the duration of current lease. Fig. 3(b) shows that a client accesses the replica, discovers the lease is expired before the next operation and reverts to accessing the home server. As long as there is not a valid lease, a client can accept a new lease when receiving the piggy-backed $\gamma > 0$ from the



1.hash(k)→SA
C1
5. p=rand(0,γ)=2; store (key,p,Texp)= (k,2,Ts+t)
2.get(k)
4.(v,γ=2)
SA
3. Is home(k) and k is hot
home

(a) Discovery



(b) Access and retirement.

Figure 3: An example of replica discovery and retirement.

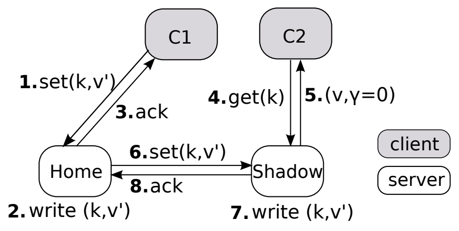


Figure 4: Other clients may see stale data when writes are not atomic for a hot key.

home server, and randomly choose one of the replicas for the new lease. Second, even with a valid lease, clients may find that a renamed key is missing in the shadow server (in case it has been deleted). Clients interpret that to mean the γ value is stale and revert to the home server.

3.3 Identifying popular tuples

We define popular key-value tuples as tuples that are accessed frequently within the recent time period. To capture the popular tuples, each server maintains a small *server-side metadata (SSM) cache*, which tracks popular tuples. Each entry has a key, a set of four counters, the state of the key, and a timestamp. The state of each entry indicates whether that particular tuple is replicated or not. The timestamp is used for termination of the replication, as we explain later. Finally, the counters are used to track popularity by maintaining an exponentially-weighted moving average (EWMA) of accesses to the tuple. Each tuple enters the cache in the `COLD` state and transits to the `HOT` state (via the transient `HOT PENDING` state till write-propagation is complete) when its EWMA access count exceeds a threshold, as shown in Fig. 5. Only a home server of a tuple maintains the metadata for it, (i.e., a key can reside in at most one SSM cache). This prevents a key from being replicated again at the shadow server.

Please note that although the metadata can also be stored in the main memcached cache by expanding each entry, it unnecessarily increases the required space in RAM by allocating metadata space for *all* tuples. Our small SSM cache is sufficient for capturing the metadata *only* for popular key-value tuples.

The access patterns to the key-value tuples are dynamic and change over time. When a tuple becomes less popular, the home server may decide to retire replicas. To terminate the replication of a tuple, the home server needs to be sure that no client will ever send a request to a shadow server for the replica. One naive way to achieve this is to explicitly send notifications to all the clients that have accessed this tuple before. This is impractical because the servers would need to record all clients for every tuple and the notifications to all clients would incur significant communication overhead.

At the home server, recall that there is a timestamp in the SSM cache entry. This timestamp records the last time a tuple is read while in the replicated state. Instead of directly retiring the tuple's replicas, the server waits for an additional lease time in the `COLD PENDING` state until the replicas can no longer be accessed by any client. While in the `COLD PENDING` state, to prevent new clients from getting pointers to the replicas, the server then responds to all incoming read requests to the tuple with $\gamma = 0$.

3.4 Handling Writes

This section elaborates on how `SPORE` handles writes. Recall, the high-level goal is to offer (a) read monotonicity and (b) eventual consistency with time-bounds on write propagation delay.

In `SPORE`, after receiving a write request from a client, the home server updates its own copy of the tuple and replies back to the client. The home server then propagates the update down to the shadow servers. During the propagation, other clients may still read the stale copy from the shadow servers (see Fig. 4). More importantly, other clients which have picked the home server as the replica, can see the updated copy before all replicas are updated; this is the key source of non-atomicity of writes. However, because every client chooses one replica to access at the beginning of their lease, they will either see the stale copy or the new copy during the propagation, but never see the stale copy after seeing the new copy (i.e., read monotonicity). `SPORE` uses another transient state (`HOT PENDING` in Fig. 5) while propagating writes to shadow servers. In the `HOT PENDING` state, the home server waits for the completion of write propagation. To ensure that a client which sees a new value at the home server does not revert to a shadow server that may potentially have stale values, we revoke the leases of any clients that request the tuple (by piggy-backing $\gamma = 0$ in the response) till write-propagation completes. Upon completion, the leases are renewed.

With the above operational method, we guarantee that a client that performs a write sees updates immediately. Other clients are guaranteed not to see stale data for

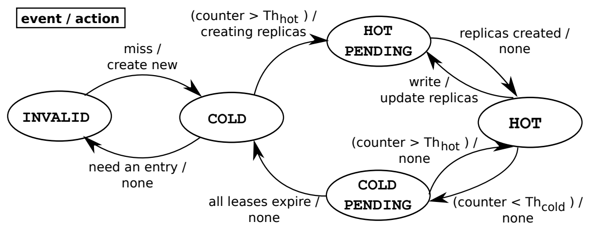


Figure 5: State transition diagram for an entry of the metadata cache at the server side.

more than T_s seconds. Finally, because all writes to any key are always sent to the home server, all replicas see the same order of writes as seen by the home server. As such, SPORE guarantees eventual consistency with (tunable) time-bound consistency.

SE-SPORE. To demonstrate that SPORE's basic architecture of replicating popular tuples is independent of write-atomicity in the memory caching tier, we develop an alternate variant of SPORE – SE-SPORE– by implementing atomic write operations. (Note atomicity is restricted to the memory caching tier.) As such, SE-SPORE is equivalent to a standalone pool of memcached servers from a consistency point-of-view.

We implement atomic writes in SE-SPORE by waiting for the replica update at *all* nodes before *any* node can supply the value to a reader using the well-understood two-phase atomic-update protocol [1]. Our measurements reveal that the atomic update can be up to two times slower than the non-atomic writes (for one to two replicas), and the slowdown increases with the replica count. Later, in Section 5.1.2, we show the cost of using only atomic writes in SE-SPORE.

One may implement atomic updates in other ways. For example, one may use CRAQ [37], which offers an update-based atomic write protocol for static, non-selective replication. Alternatively, one can adopt invalidation-based techniques from the cache coherence domain wherein the write is complete only after all replicas are invalidated (which can be confirmed by waiting for all invalidation acknowledgments) [1]. However, this design may require re-replication of hot tuples after each write; which is time-consuming.

Summary of Operation. Fig. 5 shows the precise state transition diagram for an entry in the metadata cache at the *server* side. There are five states in total: INVALID, COLD, HOT PENDING, COLD PENDING and HOT; the figure shows the events and actions that are relevant to state the transitions. The HOT PENDING state is used as a transient state until the replicas are created and ready for accesses; while the COLD PENDING state awaits until all leases at the clients expire. The home server continues propagating the writes to all replicas during the COLD PENDING state to ensure all clients see up-to-date replicas. Among all the states, a server replies with γ greater than zero only when in the HOT state, which grants leases to clients to access the replicas. In both the COLD PENDING and HOT PENDING states, although old clients may still access the replicas with their active lease, no new leases are issued. The SSM cache adopts a hybrid replacement policy based on access recency and frequency. Only the less frequently accessed tuples, the cold tuples, can be evicted (as indicated in Fig. 5). Among the cold tuples, the Least Recently Used (LRU) replacement policy is adopted to select the victim. The replacement policy was designed to be simple; more sophisticated replacement policies (e.g., [25]) may further improve the performance of SPORE. Finally, evicting only cold tuples may seem like a strict restriction, however, the tuples in transient states will transit to a stable state soon. If the cache is full with hot tuples already, there is no need to add more entries to the cache.

3.5 Other Optimizations

Sampling. Memcached is a simple and effective piece of software where short response latency is critical. To reduce the critical path, we only sample a small percentage (e.g., 3%) of requests for our SSM cache. The reason why we can have such a low sampling rate is that the popular tuples have much higher access rates. Even using a small sampling frequency, the cache is still able to distinguish the popular tuples. In fact, the sampling method acts as a filter to reduce the cache pollution. Therefore, although every write request has to be looked up in the SSM cache (to ensure all writes are propagated to replicas for hot tuples), only the sampled requests need to be looked up to update the EWMA counters and states of the tuples for the read requests.

Hotspot triggering. Our internal key renaming technique allows key-to-server remapping without global knowledge and coordination. One downside of preserving the distributed nature of memcached is that SPORE cannot explicitly shift the extra load to lightly-loaded servers for better load balance. One approach is to increase the replication count, γ, so that there is a higher probability to achieve more evenly-distributed load. However, higher γ implies higher overhead of updating the replicas which offsets the performance benefits of load balancing.

We observe that hotspots are usually formed by one or two servers in a cluster. Replicating hot tuples in the hotspot greatly improves the balance of the load. However, for servers with less load, replication has little benefit at best, and adverse effects at worst. Based on this observation, we use hotspot-triggering mechanism which activates SPORE on a server only when the server is heavily loaded. Hotspot-triggering mechanism is able to relieve the load at the hotspots while preventing unnecessary replication at the lightly-loaded servers to achieve better load balance. We use EWMA count of accesses for each server to track the activity at each server. Only the servers beyond a certain threshold of load will further replicate their hot keys. Note a fixed threshold set at the server's saturation capacity at which it can still meet the desired latency requirements is adequate. This is because no replication is needed at low loads when even the most popular servers are serving under their serving capacity. At high loads, when nearly all servers are nearing saturation, one may think that it is possible that a domino effect could occur triggering replication at all servers. However, if such a case does occur, it implies that the existing cluster simply cannot handle the load. Instead of balancing the load, one should expand the size of the cluster.

Workloads of mixed read/write ratio key-value tuples Load balancing with replication inherently relies on the assumption that replicated tuples are read-mostly. Without this assumption, the overhead of propagating the writes to the replicas could outweigh the benefits of distributing the reads to multiple servers. In Section 3.3, we describe how we select the hot tuples based on the access frequency in the recent time period with our EWMA popularity counter. For workloads with a mix of read-only, read-mostly, to write-heavy tuples, this popularity-based approach may capture the undesirable write-heavy tuples. Hence it is crucial for `SPORE` to distinguish read-mostly tuples from the rest.

To that end, `SPORE`'s SSM cache EWMA counters use weighted increments on reads and weighted decrements on writes. With the appropriate weights, we can target any desired read-to-write ratio. For example, with unit increments on reads and decrement-by-9 on writes, the long term moving average will be near zero for a 9:1 read/write ratio. Tuples with higher (lower) read/write ratios will have positive trending (negative-trending) counter values. With this addition, tuples have to exceed both thresholds (popularity and read/write ratio) to be eligible for replication.

3.6 Server Failures

In the baseline memcached system, server failure usually causes a temporary window of data staleness. For example, when a client discovers that a server is inaccessible (due to failed server or network), it may rehash and map the key among the remaining memcached pool. To ensure consistent rehashing among all clients, the system may need to issue a global configuration change to update the list of available servers on all clients. Nonetheless, if such a global configuration is not executed atomically (because atomicity in this case can significantly degrade system performance), a client may access a stale copy before the change completes. Alternatively, Facebook handles unreachable servers by resending the request to a backup memcached pool (a gutter pool) to avoid rehashing[31]. Because tuples in the gutter pool are not invalidated/maintained, they are set with a short expiration time. Users may still see a stale copy in the gutter pool.

When there is no replica, `SPORE` behaves similar to the baseline memcached. When there are replicas and a failed server is the home of some hot tuples, the replicas of these hot tuples become orphans as they will no longer receive any update. `SPORE` reduces the staleness of the data using our short-term lease mechanism. Clients are guaranteed to stop accessing such potentially stale replicas after T_s seconds.

In summary, with or without replicas, `SPORE` is susceptible to the same transient data staleness as the baseline memcached upon server failures. Therefore, atomic writes (even when limited to the memory caching tier) cannot be guaranteed in this situation. Finally, `SPORE` focuses on improving load balance for a single pool of memcached servers. It does not intend to handle network partitions in a pool any differently than what a conventional memcached system does.

3.7 Overhead and Limitations of SPORE

Space overhead. Our approach adds a small metadata cache on both the server (SSM cache) and the client (CSM cache). For the servers, it is a relatively small price to pay because the hot tuples are less than 0.01% of the total tuples. In addition, any entry in the small CSM cache may be evicted without causing correctness problems. Hence the cache size can be shrunk or tailored individually on each client without impacting the correctness of the system. Moreover, these are *metadata* caches, so the values of the tuples are not stored.

Latency overhead. For a server, the metadata cache lookup can be done in parallel with a normal memcached lookup on a multi-core server. The impact of maintaining the cache (e.g. state transitions, updating the counter) is minimized because we impose a 3% sampling frequency. Write propagation does not directly affect the write latency of a hot tuple, but it takes up some system resources and may increase the queuing latency of other requests. However, the hot tuples are rarely written to. With hotspot triggering, this overhead is even lower because only the hotspot servers replicate hot tuples. On the other hand, the CSM cache is even smaller and simpler, hence not contributing much to the overall latency.

Communication overhead. The piggy-backed γ takes up no more than one byte for a maximum of 256 replicas. The exact number of replicas will depend on the workload. However, we do not believe that practical implementations will even come close to needing 256 replicas.

4 Experimental Methodology

Memcached Configurations. We implemented `SPORE` as a modification of memcached [28]. Recall that `SPORE` needs a stub to push renamed tuples out to the server pool. We integrate the built-in Ketama consistent hashing for key-to-server mapping [20] from a C/C++ memcached library [27] for this purpose.

Workloads and Client Configurations We use Yahoo! Cloud Serving Benchmark (YCSB) [6] to generate workloads for our experiments. YCSB is designed to evaluate key-value and cloud serving stores. YCSB supports typical database operations such as read, insert, update, etc. To support key-value store type of requests (e.g. set, get, add), We ported the changes in an memcached-compatible YCSB version [26] to a newer YCSB version 0.14. We use a Java-based memcached client, Spymemcached 2.8 [35] with an added layer of a metadata cache for our replication mechanism. Spymemcached implements a series of performance optimizations, including utilizing a single connection and batching multiple requests in a single packet. Note that YCSB is very CPU-intensive compared with memcached. To ensure that YCSB does not become the performance bottleneck, YCSB and Spymemcached are run on three quad-core Linux machines, which are much more powerful than the memcached servers to be described later. We verify in our experiments that adding more memcached servers increases the system throughput, indicating that the client machines are well-provisioned and are not the limiting factor.

Our workload consists of 1 million key-value tuples. The length of the keys ranges from 4 to 10 bytes (each key has a prefix and an ID number), and the length of the values is 200 Bytes. The popularity distribution of the tuples is Zipfian with an α parameter of 0.99. We discuss the impact of lowering α in Section 6. Because the keys are hashed to determine the home server, the naming of the keys affects the key distribution among the servers. We use 6 key prefixes (6 runs) to include the effects of variation in key-distribution and compute the mean for each set of results. (Note, the key-prefixes are independent of the suffixes used for key renaming.) In each run, each client sends 100,000 requests. A total of 6 to 66 client threads are used in our wimpy node testbed, and up to 832 client threads are used on Amazon EC2. We specify the exact number of servers in the experiment descriptions in Section 5. Our experimental results show evaluations using get(k) and set(k,v) operations, but as mentioned in Section 2, any write operation can be used to replace set(k,v).

Table 1: Amazon EC2 Configurations

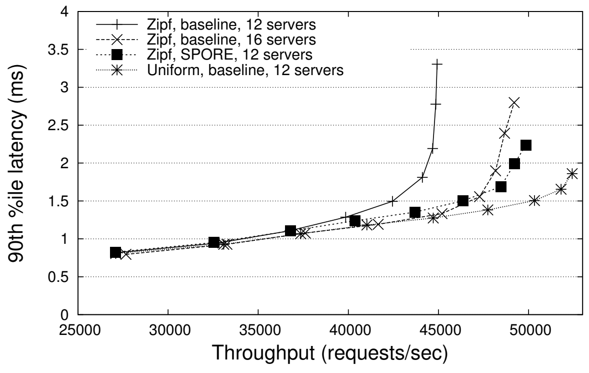
Type	Num	Arch	Instance
Server	64	32-bit	standard small
Client	64	64-bit	high-CPU extra large

Evaluation Platform. The bulk of our experimental measurements are on our "wimpy node" testbed based on Gumstix Overo Earth Computer-On-Modules(COMs) [15]. Note, our wimpy node testbed is merely a cluster that serves as our evaluation platform. We do not claim any novelty for the testbed. Nor do we implement any customized memcached implementation to exploit the characteristics of our testbed.

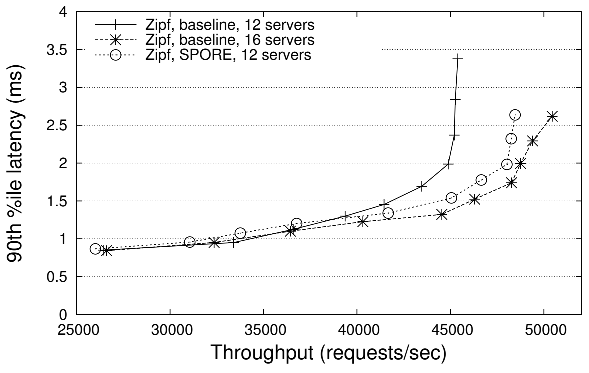
Each Gumstix Overo COM in our testbed has an ARM Cortex-A8 CPU at 600MHz, 512MB of RAM, and a 32GB SDHC MicroSD card as the main storage. The Overo COMs run Linux systems with a 3.0.0. kernel. The COMs are interconnected in a two level network. At the first level, up to seven COMs may be interconnected by a Stagecoach expansion board [16], where the Overo COMs are connected by an on-board 10/100Mbps Ethernet switch. The second level, inter-board communication is handled by an external 1Gbps Ethernet swtich.

Amazon Elastic Compute Cloud (EC2). To confirm that our results from our wimpy-node testbed hold for larger pools of real cloud instances, we validate a subset of results on 64-server memcached pool on Amazon EC2. Specifically, we use Amazon Virtual Private Cloud (VPC) service. The VPC function creates a virtual network and allows us to assign private IP addresses for each server. Other configurations are listed in Table 1. As mentioned above, the clients are run on more powerful instances to ensure that they are not the performance bottleneck. Note that we did not opt in for *dedicated hardware* nor did we use *cluster compute instances* for resource isolation. Consequently, our experiments include variations caused by resource sharing that naturally exist in a public cloud. Finally, we also scale up our synthetic workload to use 10 million tuples to suitably exercise our scaled server pool.

SPORE configuration. For `SPORE`, we use 3% of sampling frequency. Only the servers with the highest load triggers the replication function. The replica count, γ, is set to be one, unless otherwise specified. The lease time T_s for accessing the replicas is 10 seconds, but the performance is relatively insensitive to this parameter as we show later.



(a) Static key popularity



(b) Time-varying key popularity.

Figure 6: 90^{th} %ile Latency and throughput using read-mostly (99% reads) workload.

5 Results

The primary results of our evaluation are as follows.

1. **Overall cost savings and performance improvement:** SPORE achieves significantly higher throughput than an unmodified memcached at comparable latencies. For similar performance (throughput, latency) SPORE performs comparably to a configuration with 33% more machines. Further, SPORE maintains its performance advantage when the set of popular keys change.

2. **Evaluation with real cloud instances:** While the absolute throughput is higher on Amazon EC2 compared to our wimpy-node platform, the key qualitative result remains unchanged – SPORE achieves significantly better performance than the baseline.

3. **Sensitivity** The key trends are relatively insensitive to parameter changes (e.g., thresholds, 95^{th} %ile latency vs. 90^{th} %ile latency, number of replicas, cache lease periods).

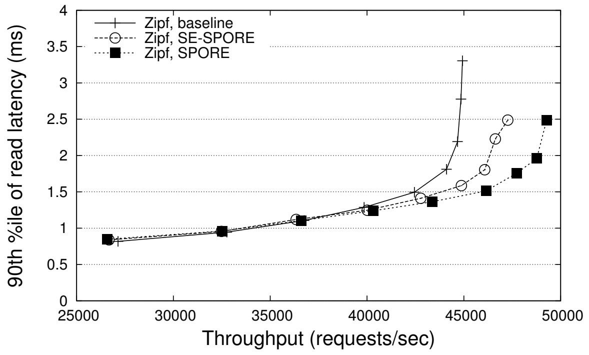
5.1 Overall performance and cost improvements

5.1.1 Read-mostly workloads

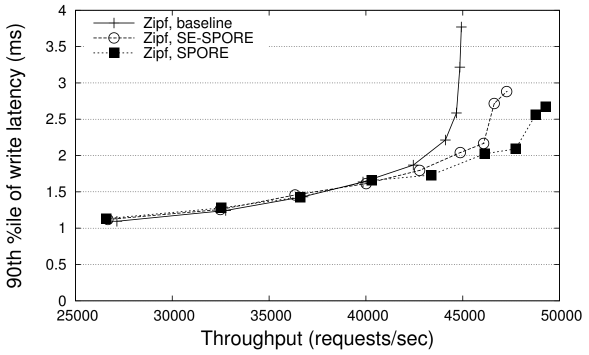
In this subsection, we discuss the performance of SPORE under read-mostly workloads. Fig. 6(a) plots the latency- throughput for four different configurations using a read-mostly (99% read) workload, which is consistent with Facebook's report that reads are two orders of magnitude higher than writes. Each point of the figure represents the average 90^{th} %ile latency and average throughput for 6 key prefixes, and the multiple points in one curve is obtained by varying the number of YCSB client threads. The first two configurations use the workloads with Zipfian distribution of key popularity on baseline memcached with pools of 12 and 16 servers, respectively. The third configuration uses SPORE with 12 servers. The last configuration is baseline memcached with 12 servers and a workload with uniform popularity distribution.

Recall that we show the performance difference of uniform and Zipfian distribution for the baseline memcached in Fig. 1. The uniform distribution is the upperbound for SPORE. We make two important observations from Fig. 6(a). First, simply adding more servers does not improve the performance as much as one would expect because of the way tail latency behaves. Although the 16-server configuration achieves higher throughput, it is not nearly as good as the uniform distribution with only 12 servers. The reason behind this trend is that increasing the number of servers does not necessarily eliminate a hotspot, as the new key-to-server mapping may not be better than the original mapping with 12 servers. Even though the average load per server reduces, the existence of a hotspot limits the overall throughput achieved at a given tail latency. Second, SPORE has higher throughput than the 16-server (33% more servers) configuration with Zipfian distribution.

To demonstrate that SPORE is indeed adaptive, we ran similar experiments but with popularity changing over time. While we maintain the Zipfian popularity distribution, we change the popularity rank of the keys every minute. Fig. 6(b) shows the performance comparison of the baseline and SPORE with 12 servers. The results are qualitatively similar in that SPORE outperforms the baseline; thus proving the adaptivity of SPORE. Though SPORE is better than the baseline, it does incur a slight degradation relative to the 16-server combination when comparing static and dynamic popularity. While in the static-popularity case, SPORE is marginally better than the 16-server baseline configuration, SPORE (with 12

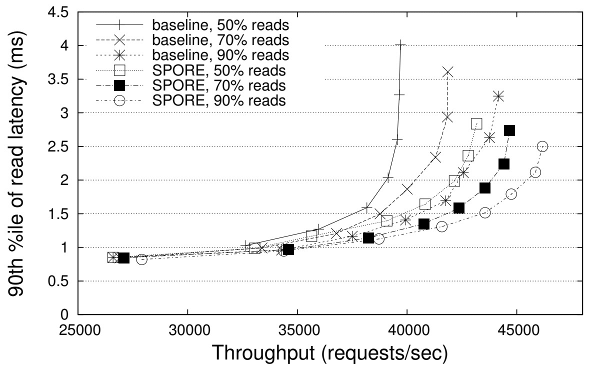


(a) Throughput vs. 90^{th} %ile of read latency.

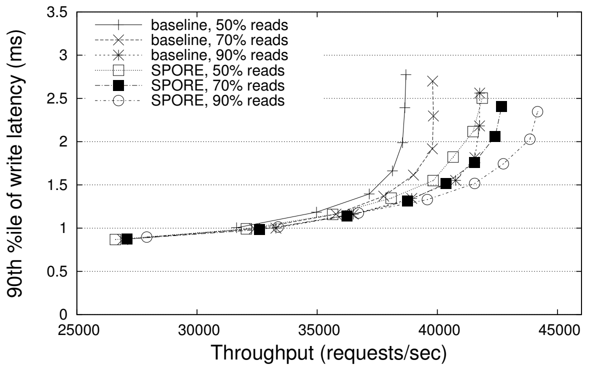


(b) Throughput vs. 90^{th} %ile of write latency.

Figure 7: Throughput and latency for workload of 99% reads.



(a) Throughput vs. 90^{th} %ile of read latency.



(b) Throughput vs. 90^{th} %ile of write latency.

Figure 8: Throughput and latency for a mixed workload where there are 20% read-only tuples in the most popular 500 tuples, and various read-write ratios for the rest of the read-write tuples.

servers) is marginally worse than the baseline 16-server configuration in the dynamic popularity case.

5.1.2 SE-SPORE

We compare system-equivalent SPORE (SPORE), standalone-equivalent SPORE (SE−SPORE), and the baseline memcached using read-mostly workloads. Fig. 7 plots the latency-throughput graph using a 99% read workload, with 90^{th} %ile read latency shown in Fig. 7(a) and the corresponding 90^{th} %ile write latency in Fig. 7(b). First, recall that on unloaded memcached servers, atomic writes across replicas in SE−SPORE are up to two times slower than non-atomic writes in SPORE. This difference in latency is not clearly visible in Fig. 7(b) for the following reasons. At low loads, replication is turned off by hotspot-triggering mechanism, hence no write propagation is needed. On the other hand, queuing latency dominates at high loads. Moreover, writes to most tuples in SE−SPORE do not incur longer latency. Second, as expected, SE−SPORE achieves better performance than the baseline, but is not as good as SPORE. This performance gap is mainly because that the two-phase commit for atomic writes blocks any access to the hot tuple until every shadow server has received the update. Since there are many requests for the hot tuples, this short period of blocking leads to many retries and thus lower throughput.

5.1.3 Mixed workloads

This subsection examines mixed workloads, where there are read-only tuples and read-write tuples. We modified YCSB benchmark to generate one read-only tuple for every five tuples in the most popular 500 tuples; the non-read-only tuples are configured to have mixed read/write ratios. Fig. 8 plots the latency-throughput for three such mixed workloads - 50%, 70%, and 90% reads for non-read-only tuples. Compared with the baseline, SPORE achieves higher throughput and lower latency for all three mixed workloads. This result demonstrates that as long as there are popular, read-only (or read-mostly) tuples in the workload, SPORE can distinguish and replicate them to alleviate the hotspot.

5.1.4 Amazon EC2 results

Because most of our results are from our limited-sized, wimpy-node platform, we validated a subset of our results on a larger pool of Amazon EC2 machine instances. Recall that the client and server configurations for the EC2 experiments are summarized in Table 1. Fig. 9 plots the throughput-latency curves for the Amazon EC2 experiment. Observe that the absolute aggregated saturation throughput for the 64 EC2 instances (approx. 350,000 requests/second) is higher than the corresponding throughput achieved by the 12 wimpy nodes (45,000 requests/second). The per-server throughput of the small instance is 46% higher than that of our wimpy node, although the latency is significantly better (63% lower). Nevertheless, the comparison of the baseline with SPORE reveals that the key trends are the same as in the wimpy-node case; SPORE achieves significantly better performance than the baseline with the same number of nodes.

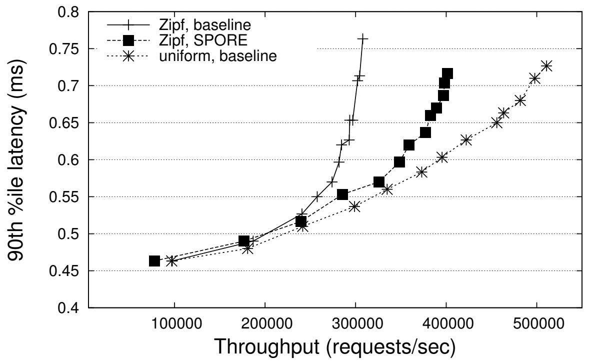


Figure 9: Scaled results: Amazon EC2, 64-nodes, read-only workloads

5.2 Overhead Minimization

We have already shown that SPORE is able to sustain higher throughput (or lower latency) at the same load when comparing with the baseline memcached. This subsection discusses the results that show how effective SPORE is in minimizing the overheads.

Performance overhead. Both our sampling technique and hotspot-triggering mechanism are crucial in minimizing the overhead. We summarize the related experimental results below.

• To assess the impact of sampling frequency, we measured the latency of requests on a lightly loaded server (where queuing delay is minimized). At 100% of sampling frequency, the average (90^{th} %ile) latency is 7% (10%) higher than the corresponding latency in the baseline memcached. However, at the default 3% sampling frequency, the average and the 90^{th} %ile latencies of SPORE are both within 3% difference of those in the baseline memcached. We can therefore draw the conclusion that when there are no popular read-mostly tuples in the workload (hence no replication), SPORE performs comparably to the baseline memcached because the latency overhead is low.

• When the load is universally low, SPORE turns off all replication and perform comparably to the baseline memcached.

• In the read-mostly workload, SPORE performs much better than the baseline memcached. However, when hotspot-triggering is disabled (i.e. every SPORE server always samples and replicates), the overall throughput/latency is only marginally better than the baseline memcached. This is due to redundant replication at the lightly loaded servers, which not only adds overhead of replica updates but also achieves worse load balance.

Space overhead. We observe that the space overhead for storing the replicas in SPORE is extremely small (less than 0.01% of all tuples), and the size of metadata cache is approximately 60KB per SPORE server. Hence, the effects on the cache miss rate of the memcached pool is negligible.

5.3 Sensitivity analysis

We conducted sensitivity tests by varying key parameters in our experiments. In this section, we present a subset of the sensitivity studies in detail and summarize the others for brevity.

• Fig. 10(a): For our workload, SPORE captures a large fraction of the benefits with one replica (excluding the original copy). Varying the number of replicas in the 2 to 3 range showed modest performance improvements for read-only workloads. Although having higher replica counts may help load balance in some workloads, the associated overhead could degrade the overall performance for read/write workloads.

• Fig. 10(b): SPORE is also insensitive to variations in lease times. Varying the lease time from 1 to 10 seconds does not change the results in any significant way. This is not surprising because renewing the lease every 1 second is relatively infrequent as a typical server serves thousands of requests per second. Thus, the overhead of renewing the lease each second is dwarfed by the performance improvement from replication.

• The bulk of our paper used 90^{th} %ile latency comparisons. For completeness, we examined median (i.e, 50^{th} %ile) latency and 95^{th} %ile latencies. The results indicate that the key observations still hold qualitatively – SPORE achieves higher throughput at comparable latencies and thus mimics the effect of additional machines. Quantitatively, we observe that the benefit of SPORE is magnified when considering higher percentile latencies.

• We compared a practical SPORE with an impractical SPORE that uses oracular knowledge of read popularity and server popularity. We found that our practical heuristics were nearly indistinguishable from oracular hot-block and hot-server identification. This implies that there is little motivation to develop more sophisticated hot block predictors.

• SPORE is relatively insensitive to changes in the EWMA counter hot/cold detection thresholds. This is not surprising given the Zipf popularity distribution in which the hot tuples are significantly more frequently accessed than relatively cold tuples. Thus, a broad range of thresholds can correctly identify the hot tuples for replication.

• Direct measurements of load served by each server reveal both the imbalance in the baseline memcached as well as the improved load balance in SPORE.

6 Discussion

In this section, we briefly discuss some possible issues, implications, limitations, and concerns that arise from some of our design choices.

Probabilistic load balancing. Ideally, a load balancing strategy would offload requests from heavily-loaded servers to lightly-loaded servers. However, SPORE's replication strategy does not seek/track lightly-loaded servers to place replicas. Instead, SPORE places replicas of popular tuples on arbitrarily chosen (by renaming and hashing) servers. While it is possible that the shadow server chosen in this manner may be heavily loaded, the probability of mapping to another heavily loaded server that affects the tail latency is low because the number of such heavily loaded servers is also low. As such, SPORE uses a probabilistic approach to load balancing. Exploring the performance/complexity tradeoffs of an alternative, load-aware approach may be an interesting study.

While the focus of SPORE was on the class of applications where eventual consistency and the possibility of (time-bound) data staleness was acceptable, we are also able to handle more demanding applications via SE-SPORE, which is still better than the baseline memcached (Section 5.1.2).

Impact of fixed SSM/CSM cache size. Some of the key structures used by SPORE are size-limited (e.g., the SSM cache and the CSM cache). One may think that, depending on workload, such size limits may cause "performance cliffs" – sudden points beyond which our design degrades. However, that is not likely because of two reasons. First, our caches hold a small number of popular tuples which can cause load imbalance; these popular tuples fundamentally have to be few in number. A large number of popular tuples will be load balanced because they will be evenly distributed over all available servers in the pool. Second, even if SPORE runs into capacity constraints in the caches, the design will be no worse than the baseline because the keys that cannot be tracked in the caches will simply not be replicated.

Workload sensitivity. Workloads with less popularity skew will likely lower the opportunity for SPORE because reduced popularity skew reduces load imbalance which SPORE targets. Social-media and web access popularity are widely reported to have highly skewed popularity distributions [4, 38, 29, 2].

7 Related Work

We provide an overview of previous work on load balancing in distributed key-value storage and in distributed hash tables. We then briefly discuss the consistency models used in distributed storage systems.

Caching. Load imbalance is often caused by skewed popularity distribution of keys. Prior work [11] proposed to place a small, popularity-based cache at a frontend load balancer to serve the requests to popular keys. This cache is small enough to fit in an L3 cache and significantly reduces the total access to the memcached layer. Our design differs from [11] in that SPORE preserves the distributed feature of memcached by not using a centralized point (a load dispatcher). Memcached relies heavily on the distributed communication pattern between clients and servers for its low latency and scalability. Any centralized point can be a performance bottleneck. A possible variation of [11] is to place a popularity-based cache in each memcached client. However, maintaining memory consistency for a large number of caches severely degrades system performance.

Migration. Replication is useful when there are popular read-mostly tuples, which is a reasonable assumption for most of the workloads. Nonetheless, in the case where such tuples are rare, replication is ineffective. One may think that instead of replicating a tuple, a key can be simply migrated (remapped) to another server to reduce the load on the hotspot. In fact, a similar key remapping technique is adopted in [34] through a centralized proxy server to operate clusters on intermittent power. However, the centralized point limits the scalability of memcached. Another remapping technique uses virtual buckets that have been adopted in Couchbase [8], and is also partially supported in the latest version of memcached [28]. Virtual buckets decouple key hashing and server mapping by adding another layer of indirection – keys are hashed to virtual buckets, and buckets are explicitly assigned to servers through configuration. Virtual buckets can be used to gradually migrate keys to a newly started server. Nonetheless, in order to balance the

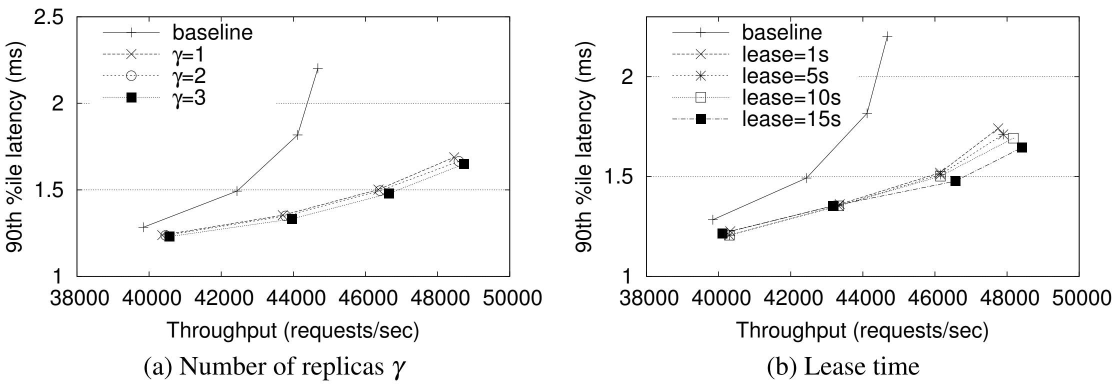


Figure 10: Sensitivity study.

load, it needs a global coordinator to monitor traffic and to remap the buckets, while SPORE's server can each act individually. SPORE's fine replication granularity (a key) also incurs lower overhead than remapping a bucket (migrating tens or even hundreds of thousands of keys). Finally, unlike replication, migration cannot deal with extremely popular keys which may contribute a significant share of a single server's load. For example, there are known cases where a single key accounts for approximately 20% of a server's workload [31].

Replication in key-value stores. Facebook handles scaling of memcached at different granularities from cluster, region, to across regions [31]. Across regions, keeping local replica is necessary for reasonable performance. In a region, Facebook decides whether to keep a single replica in a large memcached pool or multiple replicas for a set of objects based on manual heuristics (e.g. access rates, data set size, etc) derived from their experiences to avoid memory inefficiency from over-replicating data. Depending on the application characteristics, Facebook may also replicate within a pool for small data sets (those that fit in one or two memcached servers) under limited circumstances. In contrast, SPORE achieves in-pool selective replication independent of data-set size. SPORE self-adapts and reacts quickly to replicate only the popular objects without needing intervention from engineers.

Replication in Web caching. Replication of popular objects has also been extensively studied in Web caching [3, 30, 39]. Popular objects are often propagated to many caches so that they can be located near the source of demand, while less popular objects may incur a longer search path with series of cache queries. In memcached, all servers within a pool are considered to have equal distance to the clients, hence cache proximity is not a concern for replica placements.

Distributed hash tables. Distributed hash tables (DHTs) have been widely used in many distributed lookup protocols and applications ([33, 36, 19, 9]). Most of the DHTs use a variant of consistent hashing to map a key to a node. Nevertheless, load imbalance remains in the systems using DHTs, even when the servers are homogeneous [36]. Virtual servers [19] are proposed to resolve the problem by allowing each physical server to act as several virtual servers. This enables (1) more uniform load distribution in consistent hashing, and (2) the ability to potentially increase/decrease number of virtual servers on each physical server based on the computing/storage capacity of the server and the current load level. In SPORE the Ketama consistent hashing library [20] already handles the first case by generating multiple IDs (virtual servers) for each server to achieve better load balancing. We have shown that the skewed popularity has significantly more impact on the load balance than the imperfect hashing in our experiments. However, SPORE does not manually change individual number of virtual servers, which is similar to the virtual buckets mentioned above.

Consistency models in wide-area distributed systems. In general, strong flavors of consistency are known to be incompatible with the needs of wide-area, replicated distributed systems [14]. Google's Spanner is a rare geographically distributed/replicated database designed to achieve linearizability using global timestamps [7]. However, Spanner is still subject to the CAP theorem [14] which implies that strong consistency is achieved at the cost of either partition tolerance or availability. As such, weaker consistency models[1] have been explored by many distributed systems. For example, there are systems that achieve causal consistency [32, 24], per-key sequential consistency [5], and mixed models where a subset of operations have strong consistency [23]. All the above systems effectively correspond to the storage tier whereas SPORE is in the memory caching tier. While it is possible that applications with strong consistency requirements may

[1]Weaker relative to linearizability, but stronger than eventual consistency.

choose to avoid the use of memcached, SPORE focuses on the scalable distributed systems with more relaxed consistency requirements where the 10X performance improvement offered by memory caching is extremely attractive. In this context, SPORE's techniques remain relevant independent of the underlying storage system.

8 Conclusion

Memcached offers significant performance benefits by obviating the need for expensive storage tier accesses. For example, the memcached-based memory caching tier is credited with offering a 10X performance improvement at Facebook. Unfortunately, the performance of the memory caching tier is artificially limited by load imbalance. Compared to an ideally load-balanced memcached pool, a realistic memcached pool suffers from poor tail latency because of load imbalances induced primarily by popularity skew of tuples. To address this problem, we design and implement an improved version of memcached with support for self-adapting, popularity-based replication of mostly-read "hot" tuples on heavily loaded servers– SPORE. SPORE identifies popular mostly-read tuples and replicates them efficiently without excessive coordination with other clients/servers. SPORE uses time-bound *leases* – permission to access replicas – which helps guarantee (tunable) time-bounds on write-propagation. Further, we show that SPORE is fairly insensitive to parameter variations. Finally, a system with SPORE offers the same consistency as a system with baseline memcached. Our experimental results show that a 12-node pool of SPORE achieves tail latency comparable to that of baseline memcached with 16 servers.

Acknowledgment

We thank the anonymous reviewers and the shepherd Mahesh Balakrishnan for their feedback. This work is supported in part by the National Science Foundation (Grant no. CCF-0644183).

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[3] G. Barish and K. Obraczka. World wide web caching: Trends and techniques. *IEEE Communications Magazine*, 38:178–184, 2000.

[4] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 1–14, New York, NY, USA, 2007. ACM.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB*, 1(2):1277–1288, Aug. 2008.

[6] 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, 2010.

[7] J. C. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264. USENIX Association, 2012.

[8] Couchbase: a NoSQL database. http://www.couchbase.com.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 202–215. ACM, 2001.

[10] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36, 2011.

[11] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 23:1–23:12, 2011.

[12] Facebook's nsdi'13 presentation. https://www.usenix.org/conference/nsdi13/scaling-memcache-facebook.

[13] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, (124), Aug. 2004.

[14] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[15] Gumstix Overo Earth COM. `http://www.gumstix.com/store/product_info.php?products_id=211`.

[16] Gumstix Stagecoach expansion board. `https://www.gumstix.com/store/product_info.php?products_id=247`.

[17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[19] D. R. Karger and M. Ruhl. Diminished chord: a protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proceedings of the Third international conference on Peer-to-Peer Systems*, IPTPS'04, pages 288–297. Springer-Verlag, 2004.

[20] libketama: a consistent hashing algorithm for Memcached clients. `http://github.com/RJ/ketama`.

[21] S. Kumar. HPCA/PPoPP 2012 Keynote Talk. `http://www.ece.lsu.edu/hpca-18/files/HPCA2012_Facebook_Keynote.pdf`.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[23] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278. USENIX Association, 2012.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, 2011.

[25] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130. USENIX Association, 2003.

[26] A version of YCSB which supports Memcached operations. `https://github.com/mikewied/Memcached-Java-Load-Client`.

[27] libemcached: A C and C++ client library for Memcached. `https://code.launchpad.net/libmemcached`.

[28] Memcached: distributed memory object caching system. `http://memcached.org`.

[29] J. V. D. Merwe, S. Sen, and C. Kalmanek. Streaming video traffic: Characterization and network impact. In *In Proc. of International Web Content Caching and Distribution Workshop*, 2002.

[30] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching: towards a new global caching architecture. *Comput. Netw. ISDN Syst.*, 30(22-23):2169–2177, Nov. 1998.

[31] R. Nishtala, H. Fugal, et al. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398. USENIX Association, 2013.

[32] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, 1997.

[33] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 188–201. ACM, 2001.

[34] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: managing server clusters on intermittent

power. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 185–198, 2011.

[35] Spymemcached: A Java client for Memcached. `http://code.google.com/p/spymemcached`.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160. ACM, 2001.

[37] J. Terrace and M. J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 11–11, 2009.

[38] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Comput. Netw.*, 53(11):1830–1845, July 2009.

[39] J. Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, Oct. 1999.