Scale-Out ccNUMA: Exploiting Skew with Strongly Consistent Caching

Citation for published version:
Gavrielatos, V, Katsarakis, A, Joshi, A, Oswald, N, Grot, B & Nagarajan, V 2018, Scale-Out ccNUMA: Exploiting Skew with Strongly Consistent Caching. in *Proceedings of the Thirteenth EuroSys Conference.*, 21, ACM, European Conference on Computer Systems (EuroSys), Porto, Portugal, 23/04/18. https://doi.org/10.1145/3190508.3190550

Digital Object Identifier (DOI):
10.1145/3190508.3190550

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the Thirteenth EuroSys Conference

Scale-Out ccNUMA: Exploiting Skew with Strongly Consistent Caching

Vasilis Gavrielatos*
The University of Edinburgh
Vasilis.Gavrielatos@ed.ac.uk

Antonios Katsarakis*
The University of Edinburgh
Antonios.Katsarakis@ed.ac.uk

Arpit Joshi
The University of Edinburgh
Arpit.Joshi@ed.ac.uk

Nicolai Oswald
The University of Edinburgh
Nicolai.Oswald@ed.ac.uk

Boris Grot
The University of Edinburgh
Boris.Grot@ed.ac.uk

Vijay Nagarajan
The University of Edinburgh
Vijay.Nagarajan@ed.ac.uk

ABSTRACT

Today's cloud based online services are underpinned by distributed key-value stores (KVS). Such KVS typically use a scale-out architecture, whereby the dataset is partitioned across a pool of servers, each holding a chunk of the dataset in memory and being responsible for serving queries against the chunk. One important performance bottleneck that a KVS design must address is the load imbalance caused by skewed popularity distributions. Despite recent work on skew mitigation, existing approaches offer only limited benefit for high-throughput in-memory KVS deployments.

In this paper, we embrace popularity skew as a performance opportunity. Our insight is that aggressively caching popular items at all nodes of the KVS enables both load balance and high throughput – a combination that has eluded previous approaches. We introduce *symmetric caching*, wherein every server node is provisioned with a small cache that maintains the most popular objects in the dataset. To ensure consistency across the caches, we use high-throughput fully-distributed consistency protocols. A key result of this work is that strong consistency guarantees (per-key linearizability) need not compromise on performance. In a 9-node RDMA-based rack and with modest write ratios, our prototype design, dubbed *ccKVS*, achieves 2.2× the throughput of the state-of-the-art KVS while guaranteeing strong consistency.

KEYWORDS

RDMA, Key-Value Stores, Replication, Consistency

ACM Reference Format:
Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-Out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190550

1 INTRODUCTION

Today's online services like search, e-commerce and social networking are underpinned by cavernous Key-value stores (KVS). Such KVS' must provide high throughput in order to serve millions of user requests simultaneously, while meeting online response time requirements. In order to sustain these performance objectives, the dataset is typically maintained *in memory* and is sharded across multiple servers using techniques such as consistent hashing [25].

Although sharding data among individual servers enables massive parallelism, such a design can suffer from hot spots. This is because the popularity distribution of objects is highly skewed, typically following power law distributions [5, 21, 37, 43]. In other words, in the presence of skew, the server(s) serving the most popular objects will become saturated, thus becoming a bottleneck and limiting the throughput of the entire KVS.

The skew problem is well-established, and a number of techniques have been proposed to mitigate it. The techniques can be classified into two categories. The first class of techniques [16, 22, 32] uses a dedicated cache for storing popular keys to filter the skew. The second class of techniques (FaRM [14] and RackOut [37]) mitigate skew by evenly distributing read requests across all servers of the KVS regardless of the object's location. For low latency, the servers use an RDMA-enabled interconnect to access objects that reside at other servers. In essence, this class of techniques exposes a non-uniform memory access (NUMA) shared memory abstraction across the servers of the KVS.

The first approach is not scalable because the limited computational resources of a single cache node may not be able to keep up with the load. In contrast, the second approach is scalable in its processing capability, but is network bound because the vast majority of accesses is serviced by remote nodes.

In this paper, we view skew as an opportunity and leverage it to improve KVS performance. Taking inspiration from the effectiveness of caches in shared memory multiprocessors, we propose a *Scale-Out ccNUMA* architecture which augments *each* server node in a distributed KVS deployment with a small cache of hot items. Because item popularity is a function of the entire dataset and not of individual shards, all cache instances maintain an identical set of objects, which are the most popular objects in the dataset. Such a *symmetric cache* not only ensures a high hit rate, but also relieves the clients from knowing which caches maintain what objects, and avoids the need for costly metadata to track sharers on the KVS side.

Replicating data in multiple caches raises the problem of ensuring consistency in the presence of writes. While *eventually consistent* [8] systems are thought to be beneficial performance-wise, they are difficult to reason about and can cause unexpected behavior for both developers and users. Meanwhile, stronger consistency models require all sharers to agree on the order of writes; some models even require writes to be performed synchronously. We ask

[*] The first two authors contributed equally to this work.

EuroSys '18, April 23–26, 2018, Porto, Portugal
2018. ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190550

the question of whether aggressive replication through caching of popular objects can be achieved with strong consistency guarantees *and* high throughput.

To answer this question, we study non-blocking, fully distributed consistency protocols that leverage logical timestamps to achieve two strong consistency guarantees: *per-key sequential consistency* (SC) and *per-key linearizability* (Lin). We then develop *ccKVS* – a distributed RDMA-based KVS that employs a Scale-Out ccNUMA architecture, featuring symmetric caching with consistency guarantees enforced via the two protocols.

Our evaluation on a 9-node rack-based cluster shows that in comparison with a state-of-the-art KVS (Section 7), *ccKVS* achieves 2.5× (2.2×) improvement in throughput for a workload with 1% writes while satisfying SC (Lin).

Summarizing, our contributions are as follows:

- We introduce ***Symmetric Caching***: a novel and transparent caching strategy that replicates the most popular objects in all caches, thus enabling high throughput and load balance, while eliminating the costly requirement of tracking sharers. (Section **4**)
- In order to keep the caches consistent, we employ two ***fully distributed protocols*** that equally spread the cost of consistency actions by enabling writes to be performed directly in any replica. The first of the protocols guarantees per-key Sequential Consistency; the second guarantees per-key Linearizability, which we verify for safety and deadlock freedom in a model checker. (Section **5**)
- We build ***ccKVS***: an RDMA-based KVS that implements symmetric caching with our fully distributed protocols and achieves a throughput improvement of 2.5× (2.2×) over a state-of-the-art RDMA KVS for a workload with modest write ratios while satisfying SC (Lin). (Section **8**)

2 MOTIVATION

2.1 Skew and Load Imbalance

Prior research characterizing data access patterns in real-world settings has shown that popularity of individual items in a dataset often follows a power-law distribution [5, 6, 21, 37, 39, 44]. In such a distribution, a small number of hot items receives a disproportionately high share of accesses, while the majority of the dataset observes relatively low access frequency. The resulting *skew* can be accurately represented using a Zipfian distribution, in which an item's popularity, y, is inversely proportional to its rank r:y $r^{-\alpha}$. The exponent α is a function of the dataset and access pattern, and has been shown to lie close to unity. The most common value for α in recent literature is 0.99 [14, 20, 22, 32, 38], with 0.90 and 1.01 also frequently used and cited in KVS research [4, 16].

An important implication of popularity skew is the resulting load imbalance across the set of servers maintaining the dataset. As shown in Figure 2a, the server(s) responsible for the hottest keys may experience several times more load than an average server storing a slice of the dataset [37]. For instance, Figure 1 shows an example deployment of 128 servers and a data-serving workload with an access skew of $\alpha = 0.99$. As seen in the figure, the server storing the hottest key receives over 7x the average load in the system.

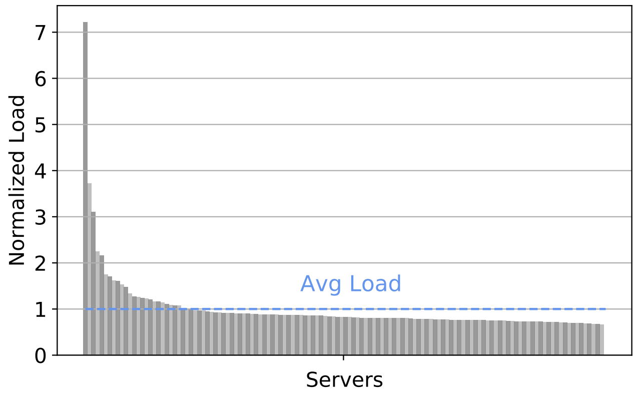
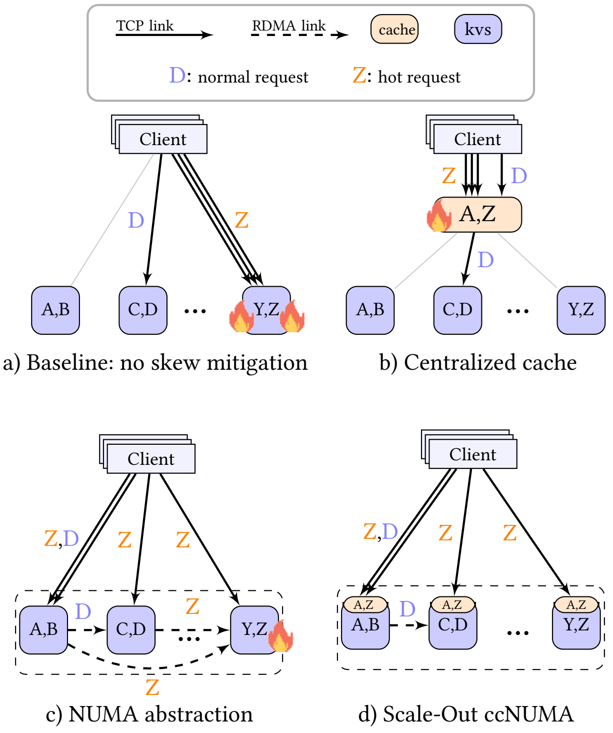


Figure 1: Load imbalance in a cluster of 128 servers caused by skewed workload with $\alpha = 0.99$.



a) Baseline: no skew mitigation

b) Centralized cache

c) NUMA abstraction

d) Scale-Out ccNUMA

Figure 2: Design space for skew mitigation techniques.

2.2 Existing Solutions for Skew Mitigation

Figures 2b and 2c depict two approaches for skew mitigation that have emerged in recent literature: caching and the NUMA abstraction.

Caching: Noting that a small fraction of the keys are responsible for the load imbalance, recent work has suggested using a dedicated cache to filter the skew from the access stream before it hits the data serving nodes (Figure 2b). Various flavours of the idea have been proposed, which are: (i) placing a cache at the front-end load balancer [16]; (ii) using a programmable switch to steer requests for hot objects to the cache node [32]; and (iii) using a programmable switch as a cache node [22].

These caching approaches suffer from two important limitations. First, they usually target storage clusters where the back-end nodes are limited by performance of the storage I/O [32]. Thus, a powerful server with an in-memory object cache is sufficient to keep up with the load. The same is not true if the data store is in-memory, in which case the high request rate it can sustain would overwhelm a single cache node. Secondly, these approaches do not offer a viable strategy to scale the cache beyond a single node to accommodate larger deployments. True, simple partitioning of hot keys across servers is one way to scale to multiple cache nodes; in the limit, however, this strategy is fundamentally limited by the ability of the cache node with the hottest key to keep up with the load.

NUMA abstraction: Pioneered in FaRM [14] and leveraged in RackOut [38], this approach offers a NUMA-like shared memory abstraction across the servers storing the dataset via remote access primitives over a low-latency RDMA-enabled network, as shown in Figure 2c. More specifically, the one-sided RDMA reads allow any server to directly access the memory of any other server in the deployment. The design exploits this remote access capability to offer a *black-box abstraction* to the outside world – a client can send a request to any node in the deployment without regard to the data's location. By allowing requests for any object to be evenly distributed across the entire deployment, load imbalance is mitigated in the face of a skewed access distribution.

The key limitation of this approach is that the vast majority of requests require remote access. Indeed, the fraction of requests satisfied locally is inversely proportional to the number of servers in a deployment. Subsequent work (FaSST [24]) improved on the network performance by replacing the one-sided primitives with two-sided RDMA communication, reducing the overall network overhead of the approach. Novakovic et al. [36] demonstrated that integrated on-chip NICs can further enhance performance by lowering the remote access latency. Nevertheless, network bandwidth has persisted as the main performance limiter of the NUMA shared memory abstraction [24].

To summarize, existing skew mitigation techniques either (1) use a powerful cache node to filter the skew from the access stream before it hits the storage nodes, or (2) exploit a NUMA-like shared memory abstraction that relies on remote access primitives to distribute the load across all servers. The first approach is processing bound because a single cache node may not be able to keep with the load, which makes it applicable mainly in a disk-based cluster environment. Meanwhile, the latter approach is scalable in its processing capability, but is network bound because the vast majority of requests requires a remote access.

3 Scale-Out ccNUMA

The central thesis of this work is that *a small cache of hot items at each data serving node can effectively shave the skew while scaling cache throughput with the number of servers.* Figure 2d demonstrates the proposed approach, which combines the best features of caching and the NUMA abstraction in an architecture we call *Scale-Out ccNUMA*. As shown in the figure, Scale-Out ccNUMA augments each node in a pure NUMA deployment with a cache of hot objects. Whenever a client request hits in a server's cache, that node can immediately return the data, thus avoiding a remote access to the node containing the corresponding shard.

Intuitively, the proposed approach has the following benefits:

- Compared to existing cache proposals that have a centralized cache at a load balancer or a network switch [16, 22, 32] and are thus limited by the throughput of that cache, the per-node cache naturally scales its throughput with the size of the deployment. Moreover, the per-node cache avoids the need for heterogeneous or exotic hardware required by prior work, such as more powerful server in the cache node [16, 32] and/or programmable network switches [22, 32]. Avoiding hardware heterogeneity in a datacenter setting is beneficial from a cost, maintenance and engineering (programmability) perspective.
- Compared to a pure NUMA abstraction (e.g., FaRM [14], RackOut [37], FaSST [24]), adding a cache to each node can significantly lower the incidence of remote accesses. As Figure 3 shows, for a zipfian skew with an exponent $a = 0.99$ and a cache storing as little as 0.1% of the hottest data, 65% of requests will hit in the cache. Thus, only the remaining 35% of the accesses (i.e., cache misses) may require remote access.
  Critically, the use of caching does not compromise the black-box abstraction presented by the NUMA shared memory architecture. Thus, any client can send a request to any server in the deployment without the knowledge of the data's location. By load balancing the requests across the nodes and avoiding the majority of remote accesses, co-locating a cache with each node naturally improves the scalability of the shared memory architecture.

Despite the benefits, the proposed approach introduces a significant challenge in requiring the caches to be consistent with respect to each other whenever a write occurs. The consistency challenge can further be broken down into two components.

The first is how to determine which caches store what items. This is necessary to find the set of replicas, which is needed for consistency-preserving actions (e.g., an invalidation or an update). Consistency protocols used in scalable multi-processors use a directory to track replicas; however, the node holding the directory can potentially become a performance bottleneck. While a directory can be distributed, a skewed access distribution would naturally make certain directory nodes more loaded than others, potentially negating the benefits of caching.

The second challenge relates to write-serialization, which is an important consistency requirement: all sharers must agree on the order of writes. Scalable multi-processors accomplish this by physically serializing at the directory, which again can cause a bottleneck in our setting.

Finally we note that in addition to the consistency challenges, Scale-Out ccNUMA also introduces the need for *update* based protocols. Protocols used in scalable multi-processors tend to employ *invalidating* protocols, meaning that a writer will invalidate all sharers, which then must re-read the item to bring it back into the cache. This strategy is aimed for parallel workloads where, for example, a variable can be updated multiple times before being read by another thread. In contrast, with read-intensive workloads that are the target of this work, an item that was updated will very likely be read in the nearest future at other nodes. This motivates

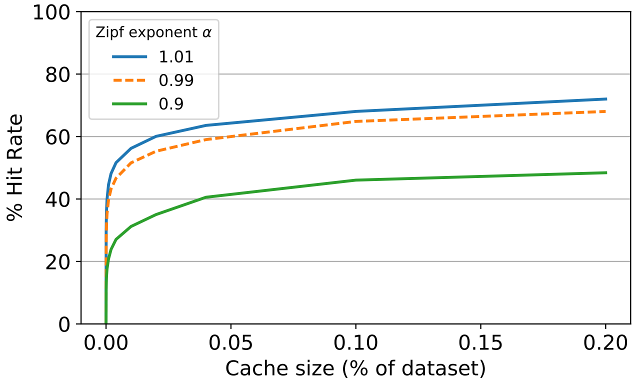


Figure 3: Effectiveness of caching under popularity skew.

the need for an updating protocol, since it proactively pushes the updated object to all of the caches.

In the next two sections, we describe a cache organization, followed by two consistency protocols that address the challenges outlined above.

4 SYMMETRIC CACHE

We exploit a simple insight in designing a scalable cache architecture that helps address the concerns outlined in the previous section. Specifically, we observe that the most popular items are, by their very nature, the most likely ones to be accessed; hence, despite the fact that there are multiple cache nodes, they should all cache the same set of items – the most popular ones. This idea, which we call *symmetric caching*, is illustrated in Figure 2d.

Despite its apparent simplicity, the symmetric cache architecture turns out to be extremely powerful, as it naturally resolves a number of challenges. For one, because all caches keep the same set of items, there is no need to inform clients of which node caches what items. Thus, clients can leverage the black-box abstraction and send requests to any node in the data serving deployment, with probability of a cache hit being dependent solely on the requested key and not the choice of the node. This ensures both a load balanced request distribution and a high cache hit rate.

Another advantage of the symmetric cache is that a node can find out which, if any, nodes cache an item just by querying its local cache; if an item is found there, then *all* nodes have it; otherwise, none do. Such ability to query a local cache to learn the status of an item naturally avoids the need for a directory, whose role in cache-coherent multiprocessors is to track the set of caches that have a copy of a cache block. By not having a directory through which consistency actions would need to serialize, the symmetric cache eliminates a potential serialization bottleneck and enables fully distributed consistency protocols, described in the next section.

An important feature of symmetric caching is that the caches are write-back. This means that writes to an item residing in the symmetric cache do not update the underlying KVS until the item is evicted from the cache. This feature is critical in avoiding a throughput degradation at the home node of a popular item, whenever writes follow a skewed distribution. Because all caches maintain the same set of items in the cache, on eviction, only the node containing the shard with the evicted key needs to check if the item has been modified and, if so, update the underlying KVS.

In order for the symmetric cache to be effective, it is essential to be able to identify the most popular items with minimal overhead. This problem has been well-researched with highly-efficient solutions proposed in recent work. A particularly attractive approach for symmetric caching is one proposed by Li et al. [32], which relies on memory-efficient top-k algorithms [11, 35] to dynamically learn the popularity distribution. In the algorithm proposed by Li et al. [32], each server maintains a key-popularity list with k entries, approximating the popularity of the k hottest keys, and a frequency counter that keeps track of recently visited keys, such that newly popular keys can be detected. The scheme uses an epoch-based approach, whereby the key-popularity list gets updated and propagated to the cache at the end of each epoch. Finally, request sampling is used to alleviate the performance impact of updating the frequency counter upon each request.

Conveniently, because symmetric caching exposes a NUMA abstraction, whereby clients spread their requests across all servers, each server sees the same access distribution as do the other servers in the deployment. Therefore, in our setting (and in contrast to [32]), it is sufficient for just a single server to act as the cache coordinator, responsible for identifying the most popular items and informing the other nodes. Centralizing the process of classifying an item as popular not only reduces the overhead of tracking hot items, but also naturally alleviates the burden of reaching a consensus on which items are popular, thus simplifying the entire process. While our evaluation does not consider shifts in popularity skew, we expect the set of most popular keys to evolve slowly, with only a handful of keys removed/added to the cache every few seconds [32].

5 ENFORCING CACHE CONSISTENCY

With symmetric caching, whilst we are able to serve a significant chunk of read requests (the cache hits) locally, ensuring consistency in the presence of writes is challenging. To ensure this, a consistency protocol must propagate writes that hit in one cache to all of the caches. What determines *when* and *how* writes are propagated is the consistency model.

A plethora of weaker consistency models abound under the umbrella of *eventual consistency*. The only requirement is that all replicas must eventually converge on a value in the absence of new updates, allowing for writes to be propagated asynchronously in any order. Performance-wise, this is beneficial, but eventual consistency can be be hard to reason about and can cause nasty surprises for both developers and clients [42].

More intuitive models mandate *write serialization*: all sharers must agree on the order of writes to a single key. Stronger yet models mandate that writes must propagate synchronously, i.e. in a blocking fashion. Enforcing such strong guarantees in a high-throughput fashion is challenging. One natural way to enforce write serialization is to employ passive replication (also known as primary-backup replication), where writes of a specific key serialize at a designated primary, as shown in Figure 4a. Such passive protocols are commonly used in shared memory multiprocessors [40] and other distributed systems [15] that demand strong consistency.

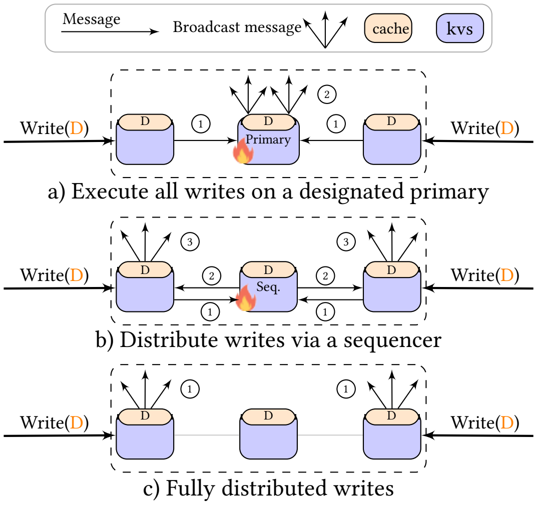

Message
Broadcast message
cache
kvs
Write(D)
Write(D)
D
Primary
1
2
Seq.
3
a) Execute all writes on a designated primary
b) Distribute writes via a sequencer
c) Fully distributed writes

Figure 4: Design space for guaranteeing a single global order for writes on a per-key basis.

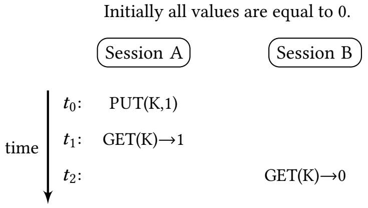


Figure 5: Session B seeing the old value is a violation of Lin, but not SC.

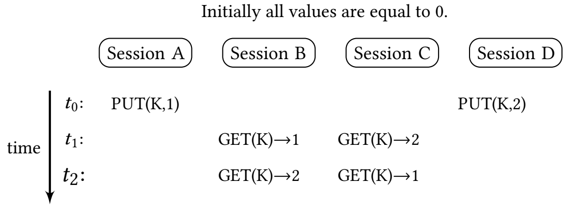


Figure 6: Sessions B and C do not agree on the order of writes and hence this is a SC violation (also a Lin violation).

In the same spirit, distributed systems can opt to achieve write serialization through a sequencer, as shown in Figure 4b, which assigns monotonically increasing timestamps to writes and their consistency actions (i.e. invalidations, updates).

However, in the presence of skew, the primary (or sequencer) in the two approaches could easily become a hotspot on writes to a hot item, as consistency actions related to that item must serialize through it. We address this issue by employing fully distributed protocols that achieve write serialization in a fully distributed manner, shown in Figure 4c, while still guaranteeing strong consistency. We elaborate on the protocols specifics below.

In this work we consider two strong models: *per-key sequential consistency* (SC) and *per-key linearizability* (Lin). We employ fully distributed protocols that enforce the models efficiently. In the rest of the section, we first introduce the two consistency models, then briefly overview existing protocols and their limitations, and finally describe the protocols in depth.

5.1 Consistency Models

For this discussion, let us assume that a number of clients (modelled as *sessions*) are interacting with a replicated data store by issuing get and put requests [8]. The *session order* is a per-session total order which represents the order in which gets and puts appear in each session. A put writes the provided object (abstracted as a value) of the corresponding key, whereas a get reads and returns the value.

A consistency model must formally specify what value a get must return. *Since we are considering data serving applications, our focus is on consistency models that provide guarantees on a per-key basis [12]; there are no guarantees between gets and puts of different keys.* We focus on SC and Lin, the latter strictly stronger than the former. It is worth noting that in both of the above models, updates to any object must happen atomically; a get must return a value written in its entirety by exactly one put – it cannot return a mishmash of the values written by two different puts.

Per-key Sequential Consistency. Informally, SC allows for writes to propagate asynchronously, as long as all sessions can *agree on the order of writes to the same key* . The following sequence of operations to key K, depicted in Figure 5, does not violate SC. Session A writes a value of 1 at real time t_0, then reads its own write at time t_1. Later, at time t_2, it is permissible for session B to read the old value of 0, since SC does not mandate that write must happen synchronously. However, the following sequence illustrated in Figure 6 is not allowed. There are two concurrent puts to same key (from sessions A and D). From the point of view of session B, the put from session A appears to perform before that of session D. Session C, however, observes a different order. Since the two sessions don't agree on the order of puts, there is a SC violation.

Formally, SC mandates that: (i) every put must eventually propagate (i.e. it must be made visible) to all sessions; (ii) all sessions must agree on the order of puts to the same key (writes must serialize) and (iii) gets and puts from the same session to the same key must appear to take effect in session order. More formally, gets and puts of the same key from all sessions must appear to perform in some total order that is consistent with the session order. It is worth noting that a similar guarantee has been formalized as *coherence* in the world of shared memory multiprocessors [3].

Per-key Linearizability. Informally, the data store must behave as if there exists only a single copy of every object in the data store, and at any time-step at most one session may issue an operation (get or a put) in session order. Thus, Lin preserves real-time behaviour, with each call to get and put appearing to take place some time between their invocation and completion. Therefore, the behavior shown in Figure 5 is impossible in a system that guarantees Lin. Lin must satisfy all the constraints of SC along with two additional conditions: (i) a put must return only after the value written has become visible to all sessions (writes are synchronous); (ii) a get may return a value only after the put (from which it reads its value) has returned.

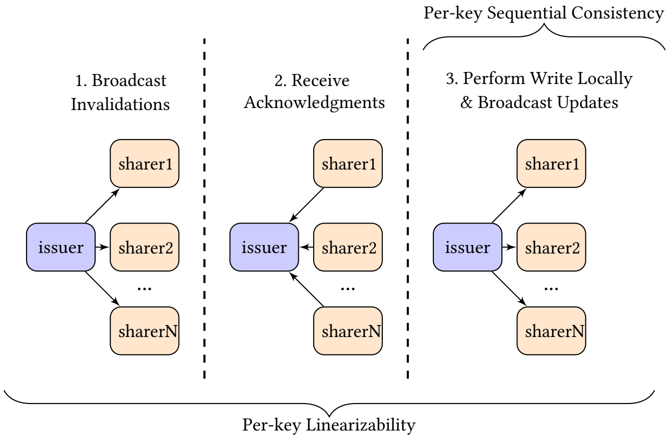

Per-key Sequential Consistency
1. Broadcast Invalidations
2. Receive Acknowledgments
3. Perform Write Locally & Broadcast Updates
issuer
sharer1
sharer2
...
sharerN
Per-key Linearizability

Figure 7: Required write actions for each consistency model.

5.2 Protocols

SC Protocol. We adapt the update-based protocol developed by Burckhardt [8] that uses Lamport timestamps for achieving write serialization. Each object in the symmetric cache is tagged with a Lamport logical clock [29], along with the session id of the last writer. (Together, the clock and session id are referred as Lamport timestamp.) On a put that hits in the cache, the writer: (1) increments the Lamport clock, (2) writes the new value, and (3) broadcasts an update that contains the new value and the timestamp.

On receiving an update, a node will apply the change only if the received value has a bigger Lamport clock than the stored Lamport clock. If the two Lamport clocks are equal, the session ids are used as tie breakers.

Figure 4c shows the fully distributed nature of the protocol, where write serialization is achieved via the use of Lamport timestamps, thus avoiding the use of serialization points (Figures 4a and 4b). The protocol is non-blocking: writes are asynchronous and can be locally applied immediately (on any replica), allowing for reads following the write to return the new value without waiting for the broadcast.

Invariant. Each write is tagged with a Lamport clock and the session id of the writer, which gives a unique timestamp for every write. This is the invariant that ensures write serialization. The reader is referred to Burckhardt [8] for a formal proof.

Lin Protocol. Recall that with Lin, a put can return only after it has propagated to all sessions. This calls for a two-phase protocol: the writer needs to broadcast invalidations and get an acknowledgement from every other sharer; only then can the value be broadcast. Of course, there can be multiple puts happening concurrently and all sessions need to agree on the same order. Therefore, in the first phase invalidations are tagged with Lamport timestamps to achieve write serialization. Figure 7 illustrates the phases involved.

The protocol we use is an adaptation of the one proposed by Guerraoui et al. [17]. Because writes are synchronous (blocking), there are situations when the protocol is waiting for an event and must deal with intervening requests correctly, without violating safety or causing deadlocks. Thus, we must account for and handle such *transient* states.

Protocol actions. A session that performs a put increments its Lamport clock and transitions the state of the cached object to a transient *Write* state (indicating that it is waiting for acknowledgements). It then broadcasts invalidations that include the key, and its Lamport timestamp. When a node receives an invalidation, it compares the incoming timestamp with the stored timestamp. If the received timestamp is greater, the machine transitions the state of the cached object to *Invalid* and responds to the writer with an acknowledgment. When a node receives an acknowledgement, it increments a counter that holds the received acknowledgements for the relevant key. When the counter indicates that all nodes have acknowledged the invalidation, the cached object is transitioned to *Valid* state and the new value and its timestamp are broadcast. Upon receiving an update, a node will check whether the relevant key is currently in *Invalid* state, waiting for the particular update (checked via comparing the timestamps); if the timestamps match, the update is applied and the object transitions to *Valid* state, otherwise the update is discarded.

Verification. The Lin protocol has one stable state and two transient states, making it more complex than the SC protocol, which has only one stable state and no transient states and has been formally shown to be correct by Burckhardt [8]. Therefore, we wanted to formally verify the Lin protocol. To this end, we expressed the Lin protocol in the model checker Murφ [13] and verified the protocol for safety and the absence of deadlocks. For safety, we verified two invariants. First, the *single-writer-multiple-reader* invariant (SWMR) [40]: at any given time there can be exactly one writer that can update an object (multiple readers can safely read an object). Second, the *data value invariant*: if an object is in a valid state (that can be read), then it must hold the most recent value that was written to that object. Our Murφ model allows for the number of processors, addresses, and timestamp size to be configured. We have verified with three processors, two addresses and timestamp of size two bits. A detailed state transition table as well as the Murφ model is available online.[1]

6 ccKVS

In order to understand the benefits and limitations of the proposed Scale-Out ccNUMA architecture, we build ccKVS, an in-memory RDMA-based distributed key-value store which combines a NUMA abstraction [14] with the consistent symmetric cache. The code of ccKVS is available online.[2]

Each node in ccKVS is composed of two entities: a shard of the KVS and an instance of the cache; each of the entities has an object store and a dedicated pool of threads for request processing. As described in Section 4, the content of all caches is identical, composed of the most popular items in the dataset. The caches are kept consistent using the fully-distributed protocols described in Section 5. The nodes of a ccKVS deployment are connected via RDMA with two-sided primitives used for communication. Clients load balance their requests (both reads and writes) across all nodes in a ccKVS deployment, e.g., by picking a server at random or in a round-robin fashion.

[1] https://github.com/icsa-caps/Linearization-Protocol
[2] https://github.com/icsa-caps/ccKVS

6.1 Functional overview

Reads: When a client request arrives at a ccKVS server, the server probes its instance of the symmetric cache. If the requested key is found, the associated object is retrieved from the cache and the server directly responds to the client. In case of a miss, the server determines whether the key belongs to a local or remote KVS partition. If remote, the server issues a remote access to the server containing the requested key using a two-sided RDMA primitive. On the destination side, the server picks up the remote access and responds with the data to the requesting server. Once the object is available, either by virtue of being in the local partition or through a remote access, the server handling the request responds to the client.

Writes: Similar to reads, write requests are also load balanced across all nodes in a ccKVS deployment, thus avoiding write-induced load imbalance. If the write request hits in the cache, the server handling the request executes the steps necessary to maintain consistency across all symmetric caches in accordance with the chosen consistency protocol. In brief, for the SC protocol, this means immediately propagating the new data to all of the caches, whereas for the Lin protocol, it requires first invalidating the caches and then propagating the new data. Regardless of the protocol, the communication required for maintaining consistency relies on two-sided RDMA primitives. If the write request misses in the cache, the server forwards it to the home node (if remote), which performs the write.

6.2 Cache and KVS Implementation Details

Thread Partitioning. The threads inside a machine in ccKVS are partitioned into two pools: cache threads and KVS threads. The cache threads receive the requests from the outside clients and are responsible for the cache accesses. The back-end KVS is handled by the KVS threads; thus, in case of a cache miss the request must be propagated from a cache thread to a KVS thread (local or remote). Finally, the cache threads also communicate with each other to exchange consistency messages: updates in SC; invalidations, acknowledgements and updates in Lin. Notably, the KVS threads do not communicate with each other.

Concurrency Control. Among the cache threads (threads responsible for servicing requests to the most popular items in the request stream) ccKVS leverages the Concurrent Read Concurrent Write model (CRCW): any cache thread can read or write any item in the cache. Despite the mandatory synchronization overheads, we find that this design maximizes throughput given the demand for the most popular keys in the dataset.

The KVS design is more involved. Conventional wisdom [33] is that when the requests are load balanced across all machines, it is beneficial to partition the KVS at a core granularity (i.e. Exclusive Reads Exclusive Writes - EREW) to avoid inter-thread synchronization on data accesses. Our design, however, employs the CRCW model for the KVS, despite the fact that, with the skew shaved by the caches, KVS accesses enjoy an access distribution that closely approaches uniform.

The reason we choose CRCW is that it allows us to minimize the connections among the cache threads and the KVS threads in the deployment. Our experiments show that this is a favourable design choice, as the benefits of limiting the connectivity, among threads on different machines, trump the overhead of the concurrency control in CRCW. We elaborate on these benefits in section 6.4. Finally, the CRCW concurrency model in KVS increases the ability of cache threads to batch multiple requests in a single packet, alleviating network-related bottlenecks. We explore the benefits of this optimization in section 8.5.

To ensure high read/write performance under the CRCW model, ccKVS synchronizes accesses using *sequential locks* (seqlocks) [19, 28], which allow lock-free reads without starving the writes. The seqlock is composed of a spinlock and a version. The writer acquires the spinlock and increments the version, goes through its critical section, increments the version again and releases the lock. Meanwhile, the reader never needs to acquire the spinlock; the reader simply checks the version right before entering the critical section and right after exiting. If in either case the version is an odd number, or if the version has changed, then a write has happened concurrently with the read and thus the reader retries.

The seqlocks are implemented in the header of each object. The header contains a version number that has a dual role: it is used both to implement seqlocks and as the Lamport clock for the consistency protocols. Therefore, we only need to add one Byte to the header to implement the spinlock. Our seqlock implementation is inspired by the OPTIK design pattern [18].

All consistency messages are treated as writes, as they need to modify metadata in the header of the key-value pair. Meanwhile, reads to the cache do not modify state and thus they happen "lock-free" and in parallel.

KVS. We use MICA [33] as a state-of-the-art KVS and leverage the source code for EREW found in [23] to build our KVS. Since ccKVS adopts the CRCW model, the KVS is concurrently accessed by all KVS threads; therefore, we implement seqlocks over MICA. Our evaluation considers both EREW and CRCW design choices. Finally, we note that Scale-Out ccNUMA and symmetric caching are not tied to any particular KVS.

Symmetric Cache. The symmetric cache is a data structure that is concurrently accessed by all the cache threads within a node; it inherits its structure from our KVS (and thus by extension from MICA [33]), and also implements appropriate support for SC and Lin. We extend the KVS's API and functionality to provide support for the consistency protocols (i.e. stable and transient states) and the consistency related operations (i.e. updates, invalidations and acknowledgments). For example, a read request under Lin may hit in the cache but it may not succeed, if the key-value pair is in Invalid state.

Each key-value pair stored in the cache has an 8B header, where the necessary metadata for synchronization and consistency are efficiently maintained. The metadata include: the consistency state (1B, only used in Lin), the version (i.e. Lamport clock, 4B), the id of the last writer (1B), a counter for the received acknowledgements (1B, only used in Lin) and the spinlock required to support the seqlock mechanism (1B).

6.3 Communication Layer

RDMA. There are two prevalent techniques to build an RDMA-based KVS: (i) using RDMA Reads via one-sided primitives, similar to FaRM [14] or (ii) with Remote Procedure Calls (RPCs) over Unreliable Datagram Sends (UD Sends) similarly to FaSST [24]. We choose the more general RPCs over UD Sends approach, but note that the Scale-Out ccNUMA paradigm is not constrained by the choice of the communication primitive and could equally work with one-sided accesses. For convenience, we utilize the elegant RDMA wrappers built by Kalia et al. in [23], modifying them wherever more fine-grained control is required.

Flow Control. The communication between cache threads and KVS threads is facilitated by a credit-based flow control mechanism [27]. The cache threads have a number of credits for each remote KVS thread, and the KVS threads have a matching amount of buffer space for each remote cache thread. Each time a cache thread sends a request, the credits for the receiving KVS thread are decremented. Similarly, the credits are incremented whenever the KVS responds. Because a request always receives a response, the flow control does not require additional credit update messages; the responses to the requests are implicitly used as credit update messages.

In contrast, the communication between cache threads on consistency actions requires explicit credit updates because not all messages receive a response. For example, a cache thread that broadcasts updates to all other machines does not receive acknowledgements for those updates. Thus, ccKVS uses explicit credit update messages to inform cache threads of buffer availability across the symmetric cache nodes. Section 6.4 describes optimizations to alleviate the network bandwidth overhead of credit updates.

Broadcast Primitive. To facilitate both protocols we implement a software broadcast, where the sender prepares and sends a separate message to each receiver. The application sends a linked list of work requests to the NIC as a batch; all work requests point to the same payload but each work request points to a different destination (i.e. Address Handle). When a cache thread intends to send more than one broadcast, we batch these broadcasts together to the NIC to amortize the PCIe overheads.

We also implement the broadcast primitive for SC using the RDMA Multicast, but do not observe any benefit. The semantics of the RDMA Multicast are that the sender node transmits a single message to the switch and the switch propagates it to all registered recipients. Therefore, RDMA Multicast optimizes the send side, but the bottleneck persists in the receive side. Thus, although machines have available send-bandwidth to send additional messages, they are still bandwidth-limited on the receive side. In practice, using RDMA Multicast slightly decreases ccKVS performance; we attribute this decrease to the switch's multicast implementation overheads.

6.4 Performance Optimizations

Reducing Connections. One of our goals in implementing ccKVS is to maintain RDMA scalability by limiting the number of threads that communicate with each other. Despite using the more scalable UD transport, all-to-all communication at the thread level can still prove challenging to scale because of the required buffer space and the posted RDMA Receives that scale linearly with connection count [24].

Partitioning the threads (Section 6.2) works toward limiting the extent of all-to-all communication, as KVS threads of different nodes do not need to communicate with each other. Additionally, we bind each cache thread to exchange messages with just two threads in each remote machine: one cache thread and one KVS thread. This optimization is enabled by the use of the CRCW model in both the symmetric cache and the KVS, since each thread has full access to the dataset (cache or KVS, respectively).

Reducing the connections minimizes the required posted RDMA Receives and the required buffer space that is registered with the NIC. As discussed in section 6.2, transitioning the KVS from the EREW to the CRCW model incurs a concurrency control overhead. However in our experiments we measure a performance increase of up to 10% when employing CRCW instead of EREW, which we attribute to the reduction of the connections between cache and KVS threads.

RDMA optimizations. Using the UD transports allows us to perform opportunistic batching in all communication with the NIC to amortize the PCIe overheads. We post Receive and Send work requests as linked lists and notify the NIC about their existence through an MMIO write. The NIC is then able to read these requests in bulk, amortizing the PCIe overheads. In order to additionally alleviate PCIe overheads, we inline payloads inside their respective work requests, whenever the payloads are small enough (less than 189 Bytes), such that the NIC does not need a second round of DMA reads to fetch the payloads after reading the work requests.

We follow the guideline to use multiple Queue Pairs (QPs) per thread [23]; for example, a cache thread uses different QPs for the remote requests, the consistency messages and the credit updates. Furthermore, we utilize the capability to use Selective Signaling when sending messages: the sender needs to poll for only one completion every time it sends a fixed-size batch of messages. We get additional performance by fine-tuning the Selective Signaling batches to match the size of the queues, such that one completion is created every time the Send Queue is filled.

Flow Control optimizations. To prevent flow control from becoming an important factor in network bandwidth consumption, we apply a batching optimization on the credit updates: we do not send a credit update for each received message; rather, we send a credit update after receiving a number of consistency messages to amortize the network cost of the credits. Additionally, the credit update messages have no payload (i.e. they are header-only messages), reducing the required PCIe transactions and network traffic for sending and receiving them. In Section 8 we show that through these optimizations the overhead of the credit update message becomes trivial.

7 METHODOLOGY

In this section, we first present the designs that we evaluate and then describe our evaluation infrastructure.

7.1 Evaluated Systems

We evaluate *Scale-out ccNUMA* by comparing it with a state-of-the-art skew mitigation approach based on FaSST [23]. Although FaSST is designed for transaction processing, it has two key attributes that make it a good baseline for a system to tackle skew: it offers a

NUMA abstraction like FaRM [14] and RackOut [38], and it leverages several design techniques to achieve high performance using RDMA [24]. We modify FaSST to implement efficient single-key Get and Put operations by stripping off all of the transaction processing overheads. We apply all of our optimizations (discussed in Section 6.4) to maximize the performance of this baseline and negate any implementation-specific advantages of ccKVS. The performance of our baseline system is on par with the reported FaSST results (subject to different evaluation setups).
We evaluate three flavours of the FaSST-based baseline design:

- **Base-EREW**: has its KVS partitioned at a core granularity similarly to MICA [33]. We expect this system to suffer under a skewed distribution as the performance will be limited by the core responsible for the hottest shard.
- **Base**: has its KVS partitioned at a server-granularity (CRCW). Compared to Base-EREW, we expect this system to perform better under skew, while still being bottlenecked by the server with the hottest shard.
- **Uniform**: represents the performance of Base under a uniform distribution. This establishes an upper bound on the performance of baseline designs.

We build ccKVS by adding symmetric caches on top of Base. More specifically, we add a cache to each node and implement a system as described in Section 6 that supports the SC and Lin consistency protocols specified in Section 5.2. We refer to these variants of ccKVS as **ccKVS-SC** and **ccKVS-Lin**, respectively. We configure the symmetric cache size to 0.1% of the total dataset (250K objects of up to 1KB each, with overall memory footprint of up to 1GB). In accordance with Figure 3, the expected cache hit ratio is 46%, 65% and 69% for skew exponents of α equal to 0.9, 0.99 and 1.01, respectively.

7.2 Evaluation Setup

Infrastructure: We conduct our experiments on an isolated cluster of 9 servers interconnected via a 12-port Infiniband switch (Mellanox MSX6012F-BS). Each machine runs Ubuntu server 14.04 and is equipped with two 10-core CPUs (Intel Xeon E5-2630) with 64 GB of system memory and a single-port 56Gb Infiniband NIC (Mellanox MCX455A-FCAT PCIe-gen3 x16) connected on socket 0. Each CPU has 25 MB of L3 cache and two hyper-threads per core. We disable turbo-boost, pin threads to cores and use huge pages (2MB) for both the KVS and the cache.
Workloads: Our evaluation is performed on workloads that follow a Zipfian access distribution. We use the skew exponent $\alpha = 0.99$ as the default value (as used in YCSB [10]) and also study $\alpha = \{0.90, 1.01\}$. For comparison, we also assess a uniform access distribution. We evaluate both a read-only workload and workloads with modest write ratios, which are representative of large-scale data serving deployments (e.g. Facebook reports a write ratio of 0.2% [7]). The KVS consists of 250 million distinct key-value pairs, thus making each node responsible for nearly 28 million keys. Unless stated otherwise, we use keys and values of 8 and 40 bytes, respectively, thus allowing a direct comparison with FaSST [24]. Finally, we apply a request coalescing optimization only in Sections 8.4, 8.5 and 8.6.

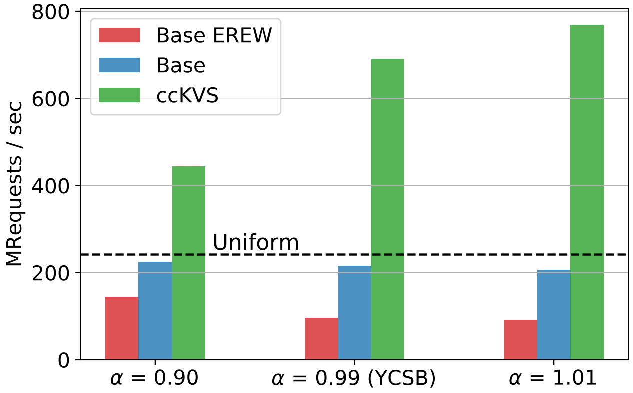


Figure 8: Throughput comparison for a read-only workload with varying skew. [9 nodes]

8 EVALUATION

8.1 Read-Only Performance

We first evaluate the performance of all the designs for a read-only workload. Figure 8 shows the performance of Base-EREW, Base and ccKVS under three different skewed distributions (α = 0.9, 0.99, 1.01). The results are similar for all three distributions and thus we focus our discussion on $\alpha = 0.99$.

As expected, Base-EREW has poor performance and achieves only 95 million requests per second (MRPS), as the whole system is bottlenecked by the throughput of the core responsible for the hottest shard. On the other hand, Base achieves 215 MRPS, significantly mitigating the skew, as the bottleneck shifts from the hottest core to the hottest server. In fact, the performance of Base is within 10% of Uniform, which achieves 240 MRPS. It is worth noting that this performance gap is strongly correlated with the skew exponent (α) and the number of servers in the deployment.

ccKVS achieves 690 MRPS, which is 3.2× higher than the throughput of Base and 2.85× higher than Uniform. The significantly higher throughput of ccKVS compared to Uniform highlights the fact that the baseline systems are network limited. ccKVS is able to achieve considerably higher throughput by avoiding the need to access remote nodes for cached objects, thus reducing network bandwidth pressure. ccKVS also benefits from the fact that symmetric caches allow all the nodes in the KVS to serve requests for hot objects, thus distributing the load evenly among them.

To better understand the reasons behind the significant performance improvement provided by ccKVS, we analyze its throughput. Figure 9 shows the breakdown of ccKVS throughput in terms of the number of cache hits and misses for a read-only workload with varying skew. In general, as the skew increases, the cache hit rate will also increase. Cache hits require compute resources, whereas cache misses mostly require network resources due to remote KVS access. We observe that the cache-miss throughput of ccKVS is equal to the entire throughput of Uniform and that the cache-miss throughput stays constant even though the cache miss rate is higher with lower skew exponents. This leads to the conclusion that both ccKVS and Uniform are network bound. Meanwhile, the cache hit throughput increases as the cache hit rate increases, which hints

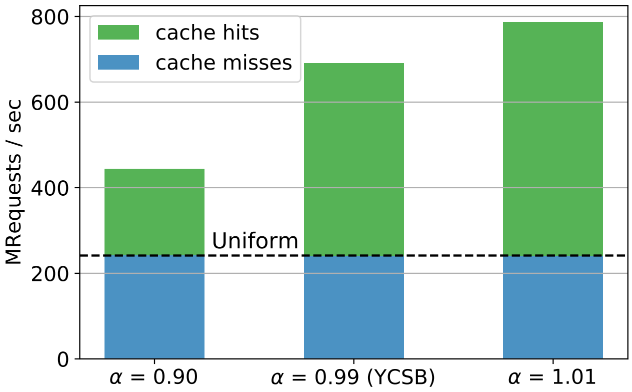


Figure 9: Break-down of completed requests in ccKVS for a read-only workload with varying skew. [9 nodes]

that CPU is not the bottleneck. We confirm these hypotheses in section 8.4.

8.2 Performance under writes

We now analyze the performance of ccKVS in the presence of writes. Figure 10 shows the throughput of the evaluated systems for varying write ratios with $\alpha = 0.99$. None of the baselines are sensitive to the write ratio, as they are all bottlenecked by the network. Note that in the baseline design, the network traffic does not change with varying write ratio as remote read and remote write requests both consume the same amount of network bandwidth. In contrast, the throughput for ccKVS decreases with increasing write ratio. This decrease is caused by the additional consistency actions for every cache write, such as broadcasting updates over the network. These actions consume network resources and thus diminish the throughput of the system, which is network-bound even in the read-only case (Section 8.1).

However, for realistic write ratios, such as 0.2% for Facebook workload [7], both ccKVS-SC and ccKVS-Lin provide throughput within 3% of a read-only workload. In fact, ccKVS outperforms Base even for write ratios as high as 5%, while providing the strongest consistency guarantee (per-key linearizability). This is a particularly important result which shows that *contrary to conventional wisdom, it is possible to achieve high throughput in the presence of aggressive replication under strong consistency guarantees.*

To further analyze the throughput of ccKVS with increasing write ratios, we show the breakdown of the network traffic for ccKVS-SC and ccKVS-Lin for 1% and 5% write ratios in Figure 11. As the write ratio increases, consistency actions (i.e. updates, invalidates and acks) claim an increasingly larger percentage of the available network bandwidth. As a result, less bandwidth is available for remote KVS accesses triggered by cache misses. Since the system is network-bound, a reduction in available bandwidth for remote KVS accesses proportionately lowers total system throughput. Finally, we note that, thanks to batching of credits (Section 6.4), flow control consumes a negligible amount of bandwidth.

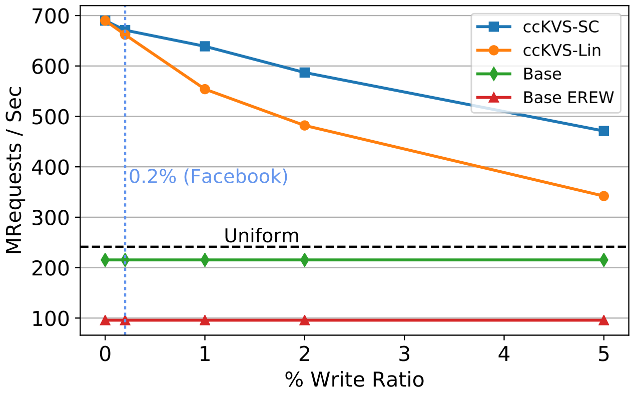


Figure 10: Sensitivity to write ratio. [9 nodes, $\alpha = 0.99$]

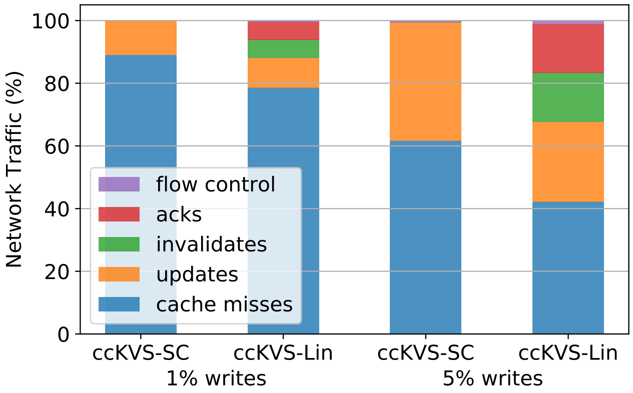


Figure 11: Network traffic breakdown. [9 nodes, $\alpha = 0.99$]

8.3 Sensitivity to Object Size

We next study the performance of ccKVS with varying object sizes. Figure 12 shows the throughput of ccKVS in comparison to Base for various object sizes with 0% and 1% writes and $\alpha = 0.99$.

In the read-only scenario (0% writes), the relative performance of Base and ccKVS follows the same trend irrespective of the object size: ccKVS still outperforms Base by over 3× for bigger objects. With writes, however, increasing the object size lowers the performance difference between ccKVS-Lin and ccKVS-SC. This occurs because with large object sizes, the bulk of network bandwidth is consumed by the data payloads and only a small fraction of the bandwidth is spent on consistency messages (i.e. invalidations and acknowledgements) inherent in ccKVS-Lin. This result demonstrates that *for workloads with large objects, it is possible to provide stronger consistency guarantees at very low performance overhead.*

8.4 System Bottlenecks

In order to identify the system bottlenecks, we analyze the hardware counters for the NIC, PCIe and memory[3]. We also profile ccKVS (using the Zoom profiler [1]) and use busy-wait counters within the ccKVS. After inspecting all measurements, we observe that

[3] We used the Mellanox's NEO-Host suite [34] for NIC profiling and Intel's pcm [2] for the PCIe and memory measurements.

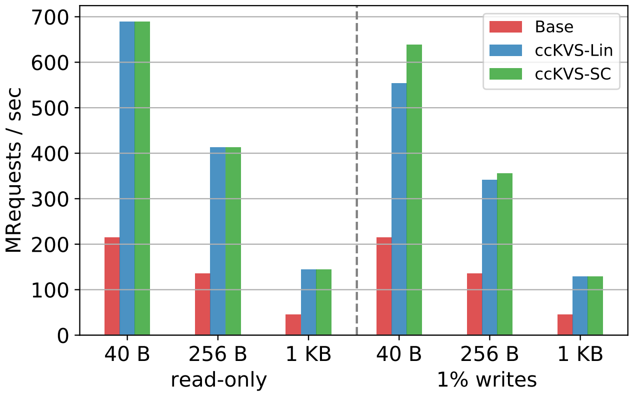


Figure 12: Read-only and 1% writes while varying object size. [9 nodes, α = 0.99]

bottleneck shifts depending on the network packet size. We identify two distinct cases: big objects that result in big packet sizes, and small objects that result in small packet sizes.

For big objects, network utilization in ccKVS closely approaches the available network bandwidth, while the rest of the resources remain underutilized; thus, we can safely infer that the bottleneck in this case is the available network bandwidth. On the other hand, with small objects, CPU, PCIe, memory bandwidth and network bandwidth are all underutilized. To our surprise, the bottleneck in the case of small object sizes appears to be the packet processing rate of the switch.

To validate our claim, we conduct the following experiment: we measure the maximum packet rate using Mellanox's microbenchmark (ib_send_bw), when connecting two machines directly (i.e. without the switch) and when connecting them through the switch. We observe that the maximum rate of sent/received packets per second is significantly higher (by up to 25%), when the servers are connected directly.[4] The results hold in ccKVS as well.

For simplicity, throughout the paper we assume that the bottleneck is in the network in both cases, as the limited switch processing rate for small packets can be viewed as an artificial network bandwidth limitation. We measure the maximum achievable bandwidth to be around 21.5 Gbps for small packets, while the NIC nominally supports 54 Gbps.

8.5 Request Coalescing

In order to demonstrate and alleviate the bottlenecks imposed by transmitting small packets, we enable *request coalescing*, whereby multiple requests destined to the same node are opportunistically coalesced into a single network packet. We only apply request coalescing to cache misses (requests and the associated responses), since these dominate the network traffic in ccKVS at modest write ratios.

Figure 13a shows the network utilization of ccKVS with and without request coalescing. In the figure, the network utilization is broken down into packet header and payload (i.e. data traffic), illustrated with striped and solid bars respectively. Coalescing multiple requests results in larger network packets, shifting the bottleneck from the switch's packet processing rate to network bandwidth. As a result, the optimized ccKVS that supports coalescing increases the throughput by almost 3× for the 40B values.

For a fair comparison, we also add support for coalescing to Base and we present the optimized performance of ccKVS and Base in Figure 13b, for read-only and 1% writes, while varying the object size. As expected, comparing the results presented in Figures 12 and 13b, both ccKVS and Base enjoy increased throughput for small object sizes when coalescing is applied. However, for larger objects coalescing is less beneficial as the system is already bottlenecked by the network bandwidth.

In detail, examining the effect of coalescing for small (40B) objects, we observe that the performance of Base is almost 950 MRPS for both read-only and 1% writes workloads, which yields an improvement of over 4× relative to the no-coalescing Base. In turn, ccKVS achieves a 3× improvement in performance with coalescing enabled, delivering over 2 billion requests per second, which is more than twice the performance of Base with coalescing.

The benefits of coalescing diminish in ccKVS on the 1% writes workload, because a fraction of network traffic carries consistency messages, which we do not coalesce. Nonetheless, even with writes, request coalescing improves the performance of the SC (Lin) variant of ccKVS by 2.6× (2×) over no-coalescing.

8.6 Latency

Figure 13c illustrates the average and the 95-percentile latency of a read-only workload (ccKVS) and a workload with 1% writes (ccKVS-SC and ccKVS-Lin) with varying load and with request coalescing enabled. We observe that, even at high loads, tail latency is about an order of magnitude lower than the target of 1ms for a typical KVS service [30]. In fact, at maximum load, the 95th percentile of both read-only and ccKVS-SC (1% writes) is quite close to the average latency. However, when ccKVS-Lin is at high load, its 95-percentile is noticeably higher than its average latency, which is expected since writes in ccKVS-Lin are blocking (i.e. sending invalidations and waiting for acknowledgments is in the critical path).

8.7 Analytical model

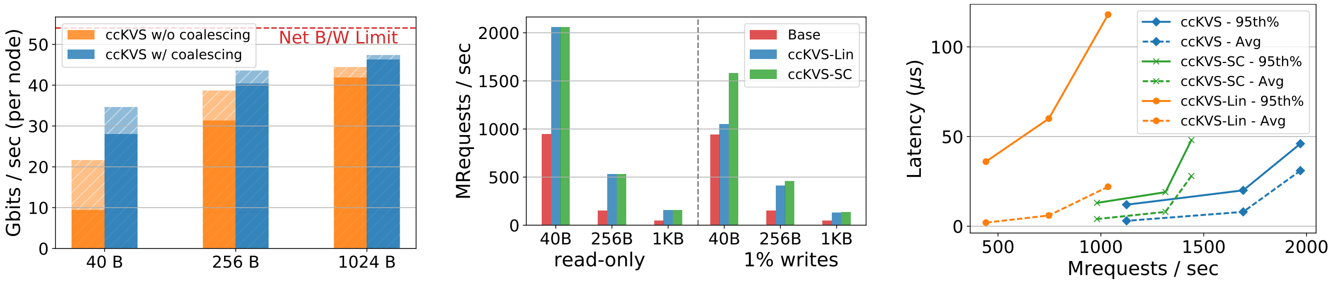
Since our 9-machine deployment prevents us from directly evaluating the scalability of ccKVS, we build an analytical model that models the throughput of ccKVS. The model leverages the fact that ccKVS is bottlenecked by the network bandwidth (Section 8.4); therefore, the throughput of ccKVS is inversely proportional to the overall network traffic.

There are two sources of network traffic. The first source is due to requests that miss in the cache, whose keys are mapped to a remote node. A request is a cache miss with probability $(1 - h)$, where h denotes the hit ratio; the cache miss is mapped to a remote node with probability $1 - \frac{1}{N}$, where N denotes the number of servers. A remote request generates two messages: one request and one reply; the total size of these two messages, in bytes, is denoted as B_{RR}. On average, the cache miss-related traffic (TR_{CM}) generated per request is given by:

$$TR_{CM} = (1 - h) * (1 - \frac{1}{N}) * B_{RR} \quad (1)$$

The second source of traffic are the messages for consistency actions, generated by hot writes (i.e., those writes that hit in the

[4] Our findings were confirmed by the manufacturer of the switch.



(a) Network utilization of a read-only workload with different object size. Solid bars: data payload, stripes: packet headers.

(b) Performance impact of coalescing for read-only and 1% writes while varying object size.

(c) 95th percentile and average latency for 40B objects at various load levels for read-only (ccKVS) and 1% writes (ccKVS-SC/Lin) with coalescing.

Figure 13: Analysis of coalescing and latency. [9 nodes, $\alpha = 0.99$]

cache). Consistency actions vary depending on the protocol. Consistency actions in Lin include invalidations, acknowledgements and updates; these three messages amount to B_{Lin} bytes, with each hot writing generating $(N-1)$ of each of these. Additionally, the probability of a hot write is given by $h * w$, where w denotes the write ratio. Therefore the overall consistency-related traffic (TR_{Lin}) generated per request in ccKVS-Lin, is given by:

$$TR_{Lin} = h * w * (N-1) * B_{Lin} \tag{2}$$

From 1 and 2, each request in ccKVS-Lin generates $TR_{CM} + TR_{Lin}$ bytes worth of traffic. Because ccKVS is network-limited, the throughput (i.e. the number of requests per second) of a ccKVS-Lin server can be computed as the available network bandwidth (BW) divided by the bytes required per request. Naturally, to compute the total throughput (T_{Lin}) of ccKVS-Lin, we need to multiply by the number of servers as shown in equation 3.

$$T_{Lin} = N * \frac{BW}{TR_{CM} + TR_{Lin}} \tag{3}$$

In contrast to Lin, a hot write in ccKVS-SC generates only $(N-1)$ updates, where each update requires B_{SC} bytes to be transferred over the network. In total, the traffic of a hot write amounts to $(N-1) * B_{SC}$ bytes in ccKVS-SC. Therefore, the overall consistency related traffic (TR_{SC}) generated by a request in ccKVS-SC, is given by:

$$TR_{SC} = h * w * (N-1) * B_{SC} \tag{4}$$

From 1 and 4, we can compute the total throughput (T_{SC}) of ccKVS-SC as shown in 5.

$$T_{SC} = N * \frac{BW}{TR_{CM} + TR_{SC}} \tag{5}$$

In the Uniform design, network traffic is generated for requests that map to a remote node. Requests map to a remote node with the probability $1 - \frac{1}{N}$ and such requests generate a request and a reply message, similar to cache misses in ccKVS, that amount to B_{RR} bytes transferred over the network. Therefore, the total traffic (TR_U) generated by a request in Uniform is given by:

$$TR_U = (1 - \frac{1}{N}) * B_{RR} \tag{6}$$

And the total throughput (T_U) of Uniform is as shown in Equation 7.

$$T_U = N * \frac{BW}{TR_U} \tag{7}$$

8.7.1 *Scalability study.*

In the presence of writes, we anticipate the per-server throughput of ccKVS to degrade as the number of servers increase, due to the proportional increase in consistency traffic. We employ the proposed analytical model to conduct a scalability study and understand the extent of this degradation.

To validate the model with our existing setup, we feed the model with the same parameters as in our implementation with request coalescing disabled. We set the cache hit ratio (h) to 65% and we set the message sizes with the exact numbers used in our evaluation for small objects (Sections 8.1, 8.2): $B_{RR} = 113$ bytes, $B_{SC} = 83$ bytes and $B_{Lin} = 183$ bytes (including network headers). Finally, we set the available network bandwidth (BW) at 21.5 Gbps, which is the network bandwidth observed for the configuration with small objects (Figure 13a).

Figure 14 shows the estimated throughput of Uniform, ccKVS-SC and ccKVS-Lin, when scaling the number of servers of the deployment from 5 to 40, while fixing the write ratio at 1%. As expected, the scaling of Uniform is almost perfectly linear. However, ccKVS-SC and ccKVS-Lin scale sublinearly with the number of servers; this is because, as the number of servers increases, the consistency traffic increases too. ccKVS-Lin scales more poorly than ccKVS-SC, because ccKVS-Lin requires more consistency messages due to the two-phase nature of its protocol.

We also plot the measured throughput of our system for up to 9 machines (i.e. the size of our deployment). As we can see, the analytically computed throughput is similar to the measured throughput for both ccKVS and Uniform. With 9 servers, ccKVS-SC and ccKVS-Lin are estimated to achieve 628 MRPS and 554 MRPS, respectively, which is within 2% of the measured throughput in our implementation (639 MRPS for ccKVS-SC and 554 MRPS for ccKVS-Lin).

In general, we find that the analytical model predicts the performance of ccKVS designs with sufficient accuracy. Using the validated model, we find that the performance of both ccKVS-SC and ccKVS-Lin is significantly better than the upper bound for the baseline (i.e. Uniform) for moderately sized deployments with a 1% write ratio.

8.7.2 *When does Symmetric Caching break even?*

Next, we use our analytical model to answer the following question: for a deployment of X servers, what is the write ratio at which ccKVS yields the same throughput as Uniform. We call this write ratio

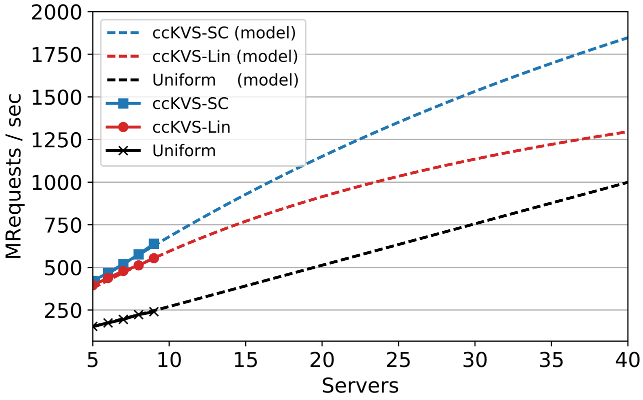


Figure 14: ccKVS scalability study using the model (dashed) and real-system validation (solid). [1% writes, α = 0.99]

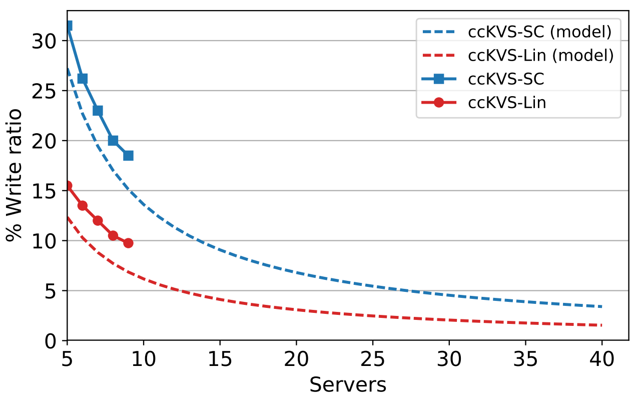


Figure 15: Break-even write ratio model (dashed) and real-system validation (solid) for up to 9 nodes. [α = 0.99]

the *break-even write ratio*. In order to calculate the break-even write ratio for ccKVS-SC, we equate the throughput of Uniform, T_U (Equation 5), with the throughput of ccKVS-SC, T_{SC} (Equation 7), and solve for the write ratio. We follow the same procedure to calculate the break-even write ratio for ccKVS-Lin.

Figure 15 illustrates the break-even write ratio for ccKVS-SC and ccKVS-Lin deployments of up to 40 servers. For example, a ccKVS-SC deployment with 20 servers will yield the same performance as Uniform at a write ratio of 8%. Therefore, a 20-server deployment with write ratio below 8% can benefit from employing ccKVS-SC.

To validate the model, Figure 15 also depicts the measured break-even write ratios for actual deployments of up to 9 machines. We observe that the trend is similar for both the model and actual measurements; however, the real system can sustain slightly higher break-even write ratios than what the model predicts. The reason behind this slight discrepancy is that, as noted in Section 8.4, the actual bottleneck for small packets is in the switch packet processing and, because the update messages in Lin and SC are big (contain both key and value) the system achieves higher network bandwidth than predicted for high write ratios.

As expected, the break-even write ratios for ccKVS-Lin are consistently lower than those for ccKVS-SC. Furthermore, as the number of servers increases, the break-even write ratio decreases in both consistency models. The decrease happens because the consistency traffic increases linearly with the number of servers, since a write to a hot object must be propagated to all servers. With 40 servers, the break-even write ratio is almost 4% for ccKVS-SC and 1.7% for ccKVS-Lin. This indicates that in a moderately sized deployment with low write ratios, ccKVS should outperform the baseline while maintaining strong consistency guarantees; however, at higher write ratios or in larger deployments, the performance benefit of ccKVS may vanish.

9 DISCUSSION

Scalability. We have established that the benefits of symmetric caching decrease with increasing size of the deployment. However, this constraint does not strictly prohibit the application of symmetric caching in large deployments. To scale beyond a rack-scale or small cluster sized deployment, we believe our ideas can be applied by simply partitioning bigger deployments into smaller Scale-Out ccNUMA clusters, each of which can independently apply symmetric caching. For example, a KVS that spans 100 nodes can be split into five 20-machine groups (similar to [38]), where each group employs symmetric caching for its portion of the KVS.

Resilience. We do not handle failures in the ccKVS design. However, existing resilience mechanisms, such as the one proposed in FaRM [15], can be readily applied to ccKVS, and, more generally, to Scale-Out ccNUMA. Most generally, we note that the principle objective of this work is to demonstrate that caching with aggressive replication presents a performance opportunity even with strong consistency models. Nonetheless, there is interesting interplay between replication, resilience and the strength of the consistency model, which we leave for future work to explore.

10 RELATED WORK

Data replication is often used by service providers to improve system performance, particularly due to load imbalance. While conceptually straight forward – replicate hot data across some number of servers [20, 21] – it comes with a number of practical shortcomings as detailed in [37]. These include determining the appropriate level of replication granularity (object, partial shard or entire shard), tracking replicas, maintaining replicas consistent, and informing clients of the replica’s locations. The latter can be particularly onerous if the number of clients is much greater than that of servers, which is often the case. In practice, these problems tend to have ad-hoc solutions requiring complex engineering and with significant system-level overheads, hence spurring the recent work on alternative approach using fast remote access and caching, as discussed in Section 2.2.

Our work takes the best features of replication, caching and fast remote access. Compared to traditional replication, our solution allows fine-granularity replication of individual keys and does not require client-side knowledge of replicas while affording strongly consistency across all replicas.

Distributed Shared Memory (DSM). In principle, a distributed KVS is not all that different from a DSM [9, 26, 31, 41]. The underlying problem boils down to enforcing a consistency model in the presence of replication. However, there is one important difference:

the workloads. Whereas the goal of DSM is to support scalable parallel programs, the goal of KVS is to support data-serving workloads. Whereas the former is characterized by CPU-intensive programs that ideally do not spend all their time waiting for memory, the latter does little more than data accesses to main memory. Whereas locality in DSM is because of program working sets, locality in a KVS can be explained by a skewed access distribution.

These workload differences and differences in sharing patterns translate into significant differences in the design of caches and the consistency protocols. Particularly, the popularity skew naturally dictates that only the popular items should be cached. Similarly, it dictates that there is no need to have different items in different caches, thus avoiding the need to track sharers (e.g., through a directory) or migrate pages.

Cache Coherence. In shared memory multiprocessors, the local caches of each processor are typically kept coherent using hardware based coherence protocols [40]. Our approach is inspired by the effectiveness of coherent caches in such architectures. Furthermore, the per-key consistency guarantees we offer are similar to the per-memory location guarantees offered by coherence protocols.[5] On the other hand, as discussed in Section 3, the protocols we employ (update based, full distributed) are very different from the ones typically employed in shared memory multiprocessors (invalidation based, serializing).

11 CONCLUSION

Popularity skew is a well-known bottleneck in existing KVS deployments. Existing skew-mitigation techniques are limited in their efficacy when applied to a distributed in-memory KVS. This work embraces skew as an opportunity through aggressive caching of popular items across all nodes of the KVS. While aggressive replication is generally thought to be a challenge in scale-out settings due to the perceived cost of keeping replicas consistent, we show otherwise. Using a low-overhead *symmetric cache* architecture combined with two fully-distributed variants of strongly consistent protocols, we show that our prototype ccKVS outperforms a state-of-the-art KVS on workloads with a moderate write ratio. This strong result paves the way for future work on enforcing strong consistency without sacrificing performance in scale-out settings.

ACKNOWLEDGMENTS

We would like to express our deepest appreciation to Haggai Eran for the technical discussions and valuable comments on RDMA; Anuj Kalia for his help with the MICA infrastructure; Vasileios Trigonakis for his support on the concurrency control and Stephan Diestelhorst, Aleksandar Dragojevic, our shepherd, Kimberly Keeton, and the anonymous reviewers for their valuable comments. This work was supported in part by EPSRC (grants EP/M027317/1 and EP/L01503X/1 to The University of Edinburgh), ARM and Microsoft Research through their PhD Scholarship Programmes.

REFERENCES

[1] 2015. Zoom profiler. http://www.rotateright.com/. (2015). Accessed: 2018-02-07.

[2] 2017. Intel® Performance Counter Monitor. https://www.intel.com/software/pcm. (2017). (Accessed on 03/05/2018).

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats. *ACM TOPLAS* 36, 2 (jul 2014), 1–74. https://doi.org/10.1145/2627752

[4] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1185–1196. https://doi.org/10.1145/2463676.2465296

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 53–64. https://doi.org/10.1145/2318857.2254766

[6] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2010. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 241–252. https://doi.org/10.1145/1807128.1807166

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 49–60. http://dl.acm.org/citation.cfm?id=2535461.2535468

[8] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150. https://doi.org/10.1561/2500000011

[9] John B. Carter. 1995. Design of the Munin Distributed Shared Memory System. *J. Parallel Distrib. Comput.* 29, 2 (Sept. 1995), 219–227. https://doi.org/10.1006/jpdc.1995.1119

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[11] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1530–1541. https://doi.org/10.14778/1454159.1454225

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[13] David L Dill. 1996. The Murphi Verification System. In *CAV 96: Computer-Aided Verification*, Vol. 1102. 390–393.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[16] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/2038916.2038939

[17] Rachid Guerraoui, Dejan Kostic, Ron R. Levy, and Vivien Quema. 2007. A High Throughput Atomic Storage Algorithm. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*. IEEE Computer Society, Washington, DC, USA, 19–. https://doi.org/10.1109/ICDCS.2007.80

[18] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic concurrency with OPTIK. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. 18:1–18:12. https://doi.org/10.1145/2851141.2851146

[19] S Hemminger. 2002. Fast reader/writer lock for gettimeofday 2.5. 30. Linux kernel mailing list August 12, 2002. (2002).

[20] Yu-Ju Hong and Mithuna Thottethodi. 2013. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 13, 17 pages. https://doi.org/10.1145/2523616.2525970

[21] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*. ACM, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2670518.2673882

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores

[5] The per-key linearizability guarantee is analogous to physical time SWMR and per-key sequential consistency is analogous to logical time SWMR [40].

with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 437–450. http://dl.acm.org/citation.cfm?id=3026959.3027000

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 185–201. http://dl.acm.org/citation.cfm?id=3026877.3026892

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. 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 (STOC '97)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[26] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1267074.1267084

[27] H. T. Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM '94)*. ACM, New York, NY, USA, 101–114. https://doi.org/10.1145/190314.190324

[28] Christoph Lameter. 2005. Effective synchronization on Linux/NUMA systems.

[29] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[30] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 4.

[31] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.* 7, 4 (Nov. 1989), 321–359. https://doi.org/10.1145/75104.75105

[32] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2016. Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform. *ACM Trans. Comput. Syst.* 34, 2, Article 5 (April 2016), 30 pages. https://doi.org/10.1145/2897393

[33] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 429–444. http://dl.acm.org/citation.cfm?id=2616448.2616488

[34] Mellanox. 2018. NEO™ Host. http://www.mellanox.com/page/products_dyn?product_family=278&mtag=mellanox_neo_host. (2018). (Accessed on 03/05/2018).

[35] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT'05)*. Springer-Verlag, Berlin, Heidelberg, 398–412. https://doi.org/10.1007/978-3-540-30570-5_27

[36] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 3–18. https://doi.org/10.1145/2654822.2541965

[37] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. An Analysis of Load Imbalance in Scale-out Data Serving. *SIGMETRICS Perform. Eval. Rev.* 44, 1 (June 2016), 367–368. https://doi.org/10.1145/2964791.2901501

[38] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 182–195. https://doi.org/10.1145/2987550.2987577

[39] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. 2011. Blink: Managing Server Clusters on Intermittent Power. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 185–198. https://doi.org/10.1145/1961295.1950389

[40] Daniel J Sorin, Mark D Hill, and David A Wood. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture* 6, 3 (2011), 1–212.

[41] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. 1997. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 170–183. https://doi.org/10.1145/268998.266675

[42] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[43] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. 2017. Fat Caches for Scale-Out Servers. *IEEE Micro* 37, 2 (March 2017), 90–103. https://doi.org/10.1109/MM.2017.32

[44] Yue Yang and Jianwen Zhu. 2016. Write Skew and Zipf Distribution: Evidence and Implications. *Trans. Storage* 12, 4, Article 21 (June 2016), 19 pages. https://doi.org/10.1145/2908557