Symmetric Caching: Exploiting Skew for High Performance Data Serving

Antonios Katsarakis

Master of Science by Research

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2017

Abstract

Data-serving systems constitute the backbone of today's large-scale applications, which are characterized by read-intensive and heavily skewed data accesses. As a consequence, the performance and scalability of the most prominent applications, which are growing both in terms of volume and popularity, heavily rely on such systems. However, existing data-serving solutions are offering sub-optimal performance with limited scalability, due to the load imbalances caused by the skewed accesses. Existing replication schemes increase the concurrency of the requests to alleviate the problem, but are not transparent to the clients, require to track sharers and their benefits are limited to eventual consistency. On the other hand, traditional caching techniques can be used to filter the skew, even though they are prone to bottlenecks and require additional resources.

In this work, we introduce Symmetric Caching, a novel skew mitigation technique that combines the best of caching and replication, to diminish load imbalance and increase performance in data-serving systems. This method is flexible, and it can be adopted transparently from the clients, by various data-serving architectures. Moreover, it avoids the cost of tracking replica sharers, and it achieves consistency over the replicas exploiting efficient protocols, that accomplish distribute write-serialization and reduce the costly network round-trips. We developed a discrete-event simulator to provide insights and evaluate the impact of Symmetric Caching in different datacenter deployments. Our results, using a representative configuration, indicate that Symmetric Caching can increase the throughput of state-of-the-art system by 7.8x in eventual consistent setting and more than 5.4x while offering strong consistency.

Acknowledgements

I would first like to thank my advisor Dr. Boris Grot, for his constructive comments, patience, motivation and for the continuous support of my thesis and related research. Besides my advisor, I would like to thank my collaborators, since this work consists a part of a bigger project, Vasilis Gavrielatos, Arpit Joshi and Dr. Vijay Nagarajan, for directing me into making better decisions during the shaping and the application of the idea.

Finally, this work was also supported in part by Microsoft, the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Antonios Katsarakis*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Data Serving

Data serving is considered as one of the core utilities in a datacenter, as it constitutes the fundamental building block for many datacenter applications. These applications, which span multiple fields and offer a variety of services, such as e-commerce, social networking, and electronic marketplaces, rely on data serving systems to store, modify and access their datasets.

The majority of requests in data serving systems refer to relatively small-sized objects[5, 14]. This is a repercussion of the primary purpose of those services, which is to offer a personalized user experience with dynamic content. As a consequence, each user request is translated into hundreds of smaller internal requests[14, 43], which are then sent to the data serving system in order to create an up to date on-demand response for the user.

Moreover, in data-serving workloads reads dominate writes and the access traffic is characterized as read-mostly[9, 14]. For instance, Facebook reports that just 0.2% of their total traffic consists of writes[9]. This is a consequence of the fact that in those services a piece of content is written by a single or small number of users (i.e. a post in a social network), while it can be read by a broader group of users (i.e. friends, followers).

Furthermore, data serving systems are required to offer both high performance and massive storage capacity, since the above applications, which are usually identified as large scale, are constantly increasing both in terms of popularity and volume[2]. More precisely, two aspects of performance are critical for data serving systems. Firstly, the user-facing nature of requests implies that latency should respect tight bounds

defined by service level objectives (SLO), because response time is crucial to user satisfaction[13] and even small delays can have immense financial impacts. Secondly, there is also a need for high throughput, due to the extensive amount of concurrent user requests, which produce a massive load of internal traffic[39].

To cope with those demands, data serving systems partition and distribute the datasets across multiple machines, using a mechanism called sharding. This approach leverages hashing techniques, such as consistent hashing[31], to evenly distribute the dataset across various nodes. Thus, it can offer an excessive amount of storage. Furthermore, since data are distributed among many nodes, ideally the hardware resources of every single machine could be exploited by issuing requests in parallel to increase the performance of the system.

1.2 Performance Bottlenecks in Data Serving

1.2.1 Skewed Data Access

Although the dataset is partitioned almost evenly across the machines, the data accesses are skewed. More precisely, several studies[5, 41, 26, 47] has shown that real world applications of this scale demonstrate data access patterns that follow a power-law distribution as illustrated in the Figure1.1. In other words, just a few objects are responsible for most of the accesses.

This phenomenon leads to high load imbalance in the system since nodes that are responsible for the most popular objects receive much more requests than the rest of the machines. Consequently, the majority of the requests are serialized in a small number of nodes, which limits the parallelism of the system and impacts its performance.

1.2.2 Inefficiencies of Traditional TCP/IP

TCP/IP is an inter-temporal protocol designed at the 70s and is still used today for communication over the network. This protocol is a *de facto*, and it is not only applied in wide area networks, such as the internet, but it is also used for the intra-datacenter communication. Although the protocol seems to satisfy the requirements of individuals to communicate and access content over the globe, several studies[18, 33] have revealed that it can be a huge burden for performance-oriented datacenter systems, such as data serving, where the network is a dominant factor.

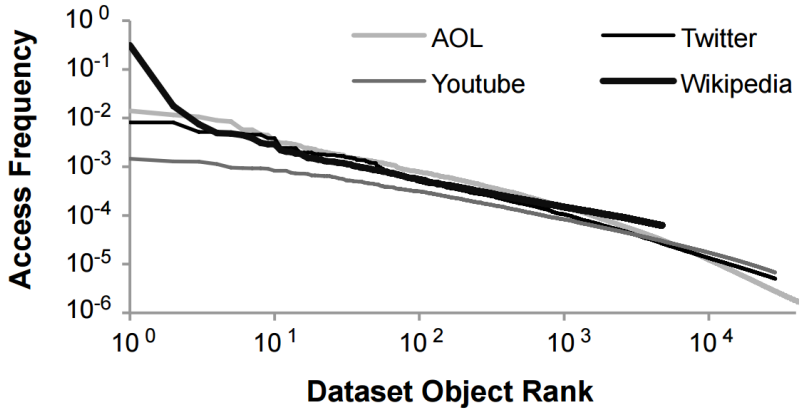

Figure 1.1: Large scale data accesses following power-law (linear in log-log scale) trends. (Figure 1 from [47])

TCP/IP can be characterized as a computational-hungry network protocol, which affects systems' performance, for numerous distinct factors. First of all, as Figure1.2 depicts TCP/IP network stack is composed of multiple layers. Although the layered model makes the design and implementation easier, the hefty headers contained in each packet consume computational resources at the expense of the application logic, both at the transmission and reception phase. Furthermore, the processing of a TCP/IP packet involves interrupts and happens in kernel mode, thus in addition to preventing application threads to run, it also introduces expensive context switches between user and kernel space. Last but not least, during header processing and as the packet moves from one stage to another, data is copied multiple times until it reaches the application layer, which contributes to the overall cost.

Apparently, the heavy processing requirements of the TCP/IP stack further adds to the load imbalance problem caused by the skewed data accesses among the servers. As a result, the overall performance of those systems is bottlenecked by the serialization of the requests in the few nodes storing the hottest items, while the majority of the machines remain under-utilized.

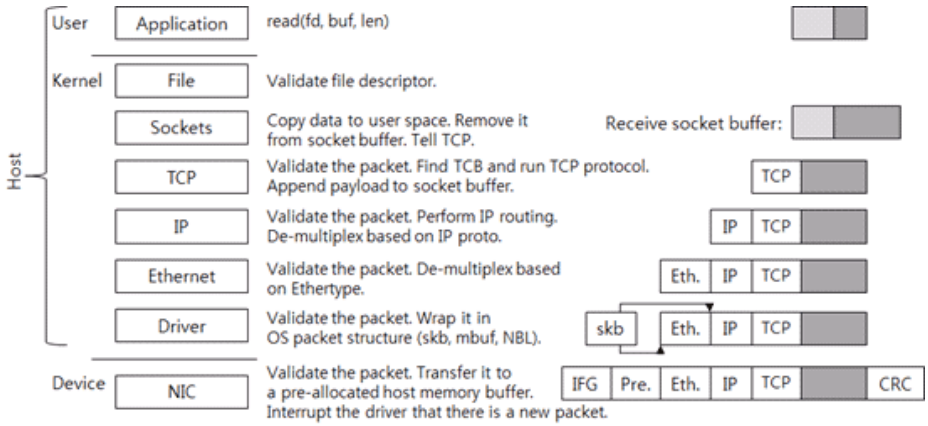

Figure 1.2: Procedure of receiving a single message over TCP/IP.

1.3 Contributions of the Thesis

The aim of this thesis is to reveal useful insights and provide an efficient solution to the load imbalance problem caused by skewed data accesses, which affects the performance and scalability of existing data-serving systems. More precisely, this work makes the following contributions:

- We propose and evaluate ***Symmetric Caching*, a novel skew mitigation technique that offers load balancing and increases the performance of data-serving systems**, by combining the best attributes of replication and caching, while being transparent to the clients and avoiding the costly requirement of tracking shares. (Section **3.1**)

- We developed and verified for safety, **efficient protocols that evenly spread the cost of required actions for consistency** and significantly increase performance, by enabling writes to be executed in any replica and achieving distributed write-serialization without incurring any network hops. (Section **3.3**)

- We introduce ***Incremental Concurrency Control* a new dynamic synchronization scheme**, where an object according to its popularity can be benefited either by the lack of synchronization overheads or by the concurrency offered by optimistic concurrency control methods. (Section **3.4**)

- We implement **"*Eureka*": A configurable discrete event simulator for data serving systems**, that enhances the design exploration and enables the compar-

ison between distinct systems with different peculiarities on the complex distributed environments. (Section **4.1**)

The rest of this thesis is organized as follows: Chapter 2 provides background knowledge and discusses the related work. Chapter 3 presents and explains the benefits of the Symmetric Caching technique. Chapter 4 gives more details about the simulator, the experiments conducted and the evaluation results, while Chapter 5 provides conclusion and future work.

Chapter 2

Background and Related Work

2.1 Key-Value Stores

For more than a decade, key-value stores(KVS) consist the most common paradigm for high-performance data serving systems. In detail, a KVS is a convenient and simple abstraction of storing a dataset at object granularity, where each object is stored as a value, and it is referenced using a unique key identifier. Their simplicity is reflected to their API, which is usually limited to primitive functions, such as GET(key) and PUT(key, value) that correspond to single object read and write operations.

In-memory Key Value Stores

Numerous research studies have been focused on KVS designs by prioritizing differently the characteristics of availability, performance, consistency, and fault-tolerance. A key-value store can be classified and used either as a cache[21, 35] or as a storage solution[24, 14]. The primary difference between those designs is that a storage KVS is a standalone solution that requires persistence, while a cache is used on top of disk-oriented storage systems, to accelerate slow disk accesses and to provide a performance boost by temporarily keeping the objects in memory.

Although the conventional implementations of those two KVS classes were different since one was optimized for memory and the other for disk, recently both of the approaches have converged to in-memory designs. The drop in the price of DRAM[43] and the need for high-performance, lead state-of-the-art storage systems also to move into in-memory schemes[18, 43], by leveraging replication and check-pointing to disk, in order to tolerate failures of the system.

2.2 Low Latency Network in Datacenters

The last few years, research has been focusing on low-latency networking solutions[23, 40, 45, 16] for intra-datacenter communication. The main goal is to moderate the cost of the required network processing. This is usually achieved by re-designing the network stack and by providing specialized hardware, to avoid occupying critical for performance CPU resources.

Innovations such as RoCE[27] enabled the popular, in the high-performance community, RDMA protocol to run on top of Ethernet, which helped in the adoption of low-latency networks by datacenter providers. In more details, RDMA is an efficient networking alternative that tries to overcome most of the TCP/IP expenses. To achieve that, it offers features such as *kernel bypass* and *zero-copy*, as shown in Figure 2.1. Additionally, it pushes most of the complexity to the network interface card(NIC), where specialized hardware performs the required actions efficiently. As a result, it occupies as fewer CPU resources as possible and avoids packet processing to interfere with the application execution.

Enhancing Data Serving with RDMA

As large-scale applications, become more and more popular, their performance requirements keep increasing. To cope with those demands data serving systems, such as key-value stores, try to squeeze every drop of performance. This has motivated multiple researchers both from academia[46, 7, 28, 37, 29] and industry[18, 48], to enhance these systems with cost-effective RDMA networking, that overcomes traditional overheads. Most works choose to completely re-design and implement systems from scratch, to find the most efficient way to endorse this low-latency communication.

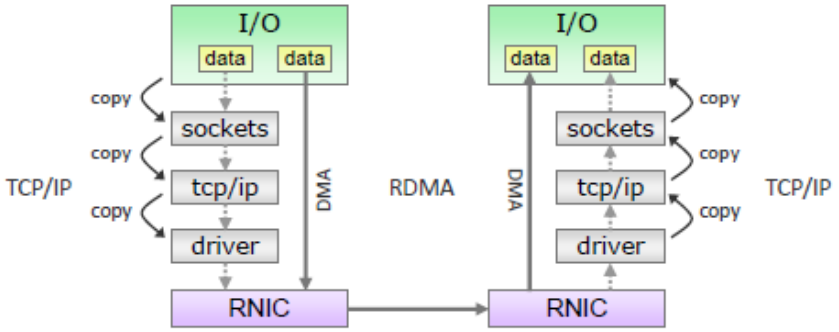

Figure 2.1: Data communication over TCP/IP vs. RDMA (Figure 1. from [6])

However, the community seems to be divided in terms of how to leverage this new technology as efficiently as possible. More precisely, RDMA provides two different methods of communication, using either two-sided or one-sided operations. The former is the traditional way of communication, similar to the conventional protocols, where both parties involved in the communication, consume CPU resources. On the other hand, the latter consists a new communication primitive where a node can remotely read or write directly to or from another machine's memory, using specialized hardware on the NIC of the targeting node, but without occupying any of its CPU resources. One-sided operations seem more attractive to the research community. Nevertheless, they introduce several challenges[49, 50, 17], and recent works[29, 30] have shown that long-established two-sided operations can be equally efficient for KVS designs.

2.3 Sharding and Skewed Data Accesses

Due to the amount of the storage capacity and performance demanded by large-scale applications, the dataset is sharded across several machines. In such scenarios, using the unique key identifier and a hash function is enough to locate the single machine, also known as the home of the key, which is responsible for storing and serving requests for the particular object.

Although designs that exploit in-memory and low latency networks boost performance, the parallelism offered by sharding is the primary factor that determines it. In large-scale data serving systems that can span hundreds or thousands of machines, the concurrency of the system can provide tremendous benefits regarding performance. In the utopian scenario of uniform access traffic, the performance always increases as more servers are added to the system, following a linear trend. Moreover, a state-of-the-art study for in-memory key-value store[35] has provided evidence that in the absence of skew in the workload, it is even more beneficial to partition the dataset in core granularity instead of the server. This approach can further boost performance by avoiding intra-node synchronization overheads, such as locks.

Limits of Sharding in the presents of Skew

The phenomenon that real-world data serving workloads follow skewed data access patterns is also known as popularity skew. Prior studies, has shown that these accesses

can be represented accurately by a Zipfian distribution[5, 8, 49, 20, 44]. More precisely, Zipf's law can be applied by sorting the keys in descending order based on their accesses. Then the popularity p of a key, given by (2.1), is inversely proportional to its rank r. The exponent α consists the main determining factor of this equation and according to the literature representative values for realistic accesses are usually close to unity. Although the precise value is workload-specific, typically the research community uses $\alpha = 0.99$[18, 11, 25, 34], with some exceptions that have used either lower (i.e. $\alpha = 0.9$)[4] or even higher (i.e. $\alpha = 1.01$)[20] values.

$$p = 1/r^{-\alpha} \tag{2.1}$$

Techniques that simply partition the dataset, such as consistent hashing, are not designed to deal with highly skewed access traffic. Sharding methods can just evenly spread the dataset among the nodes, since by their nature they are not sufficient to handle skew, because of two factors. Firstly, due to the fact that the access patterns can not be predicted a priori and thus are unknown at the partitioning phase, and secondly because the key popularity can dynamically shift during the execution.

Therefore, despite the fact that partitioning can provide high concurrency, in the ideal case of uniform workloads, the opposite holds when it is applied to realistic systems that exhibit skewed data accesses. This is because of the incident that the majority of access traffic which is generated, is targeting only a few nodes that store the most popular objects. As illustrated in Figure 2.2, machines that store the hottest objects receive more than $7x$ requests compared to the average node.

Consequently, there is a high load imbalance in the system, creating hot-spots, which serializes most of the requests, towards the path where the popular keys are stored in the set-up. More precisely, either just a few network links in this direction, or one of the performance critical edge-component, such as the memory, NIC(s), or cores of the most popular machines, limit the performance of those systems. Thus, large-scale systems are constrained by the minority of the overloaded resources while the rest are under-utilised.

2.4 Skew Alleviation Techniques

There have been several works which attempt to solve the load imbalance problem, caused by skewed data accesses, in data serving systems. Some of those take actions pro-actively, while others are reactive. Although existing methods can noticeably im-

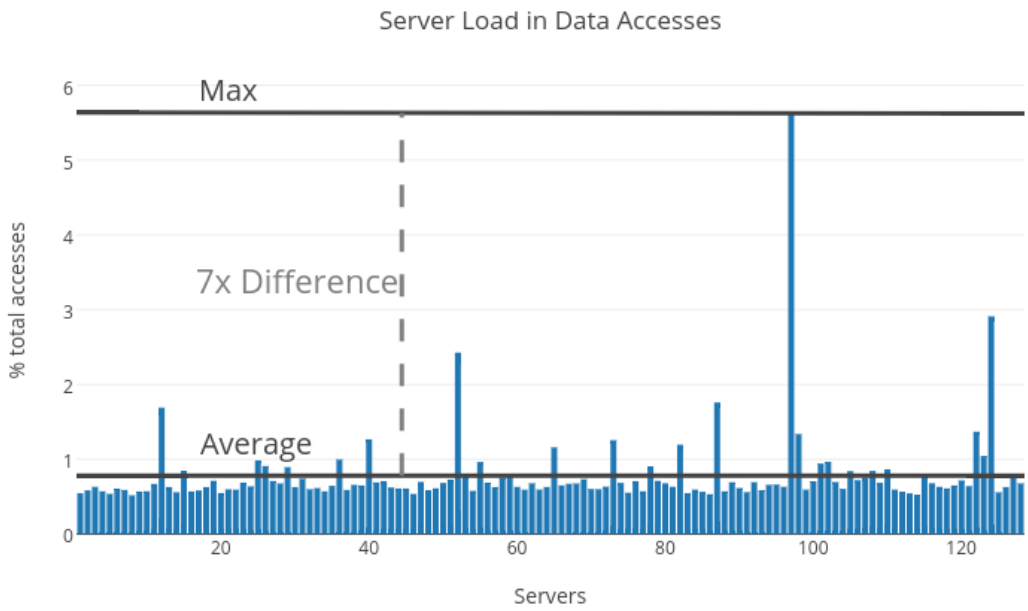

Figure 2.2: Per machine load of a partitioned dataset in 128 nodes that follows Zipfian distribution with exponent factor $\alpha = 0.99$

prove the performance of current systems, they either offer sub-optimal solutions, or they introduce consistency maintenance burdens. Additionally, it worth mentioning that some of the approaches either are unable to support performance-critical features, that further boost the efficiency of such systems, or they assume scalable RDMA networks.

2.4.1 Dynamic Replication

Dynamic replication is the oldest and the most commonly used reactive approach, that targets the problem of skewed data accesses. In this method, each key is initially serviced only by its home node. When load spikes, a load-balancer triggers a replication of hot objects to another node. Therefore, by definition, dynamic replication does not equally spread the load, but instead, once the resources of a single or a group of nodes are insufficient then it takes action.

In detail, this approach usually can not provide a deterministic way to predict the location of the keys if replication actions have occurred; thus a dynamic global map that locates keys in the system has to be known. This can be either achieved by redirecting all the client traffic through load balancers, that increase the number of total hops, or by managing and communicating this global state to each client. Additionally, a lot of metadata is required to track the accesses and the placements of keys,

thus usually the keys are grouped to reduce this overhead, in shards[9]. However, this increases the cost of replication, since a single popular key leads into a replication of its whole group.

A recent dynamic replication scheme, proposed by Hong and Thottethodi, SPORE [25], seem to overcome most of the previous difficulties. It leverages the existing KVS API and consistent hashing, to replicate and locate a single key deterministically. A new replica is created just by appending the replica number at the end of the original key as a suffix, and then this key is inserted into the system leveraging the consistent hashing mechanism. Although this technique minimizes the cost of both tracking accesses and locating keys, still the clients have to be aware of the replication factor of each key. Furthermore, since this approach deterministically chooses the key for the replica, which is cloned to a node ”blindly”, without considering any important factors, such as the location, the load or even if the targeted node already contains a replica for the same key. Thus, it is not unusual to lead into a domino effect.

Finally, either by the conventional method or the SPORE's approach, offering strong consistency is a burden. In both of the cases, due to the replicas placement, where they might end up being far away, achieve that cost-effectively is not feasible, especially when the TCP/IP protocol is used. In addition to that, in the conventional approach, there is a further cost in terms of consistency, since even non-popular keys that were replicated because they belonged to a heavy hitter group have to be kept consistent. These two reasons are contributing to the case, that most, if not all, of the dynamic replication approaches, are beneficial only over weaker consistency guarantees.

2.4.2 Caching Techniques

Caching is one of the most pervasively used techniques, that is applied in almost every system either in software or hardware. This method exploits the locality of data, and it is used to reduce the access latency and save network bandwidth.

Techniques that leverage caching are also extensively used in the context of the data serving systems. Existing approaches are applying caching to filter the skew of the workload, using diverse architectures, as shown in Figure 2.3, in different locations in-between the clients and the back-end nodes, and exploiting various hardware features. Nevertheless, in most cases caching in large-scale systems is either vulnerable to hot-spots or it introduces consistency issues, similar to the dynamic replication techniques.

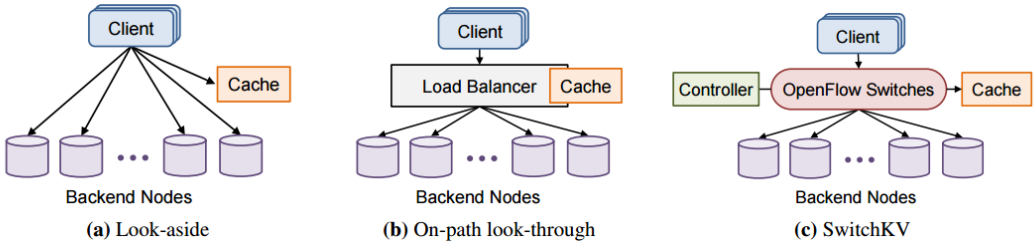

Figure 2.3: Existing caching architectures. (Figure 1 from [34])

The work by Fan et al.[20], proves that a small popularity-based cache that can fit on the last level hardware cache(LLC) of a scale-up server, can be surprisingly effective, for large-scale systems. More precisely, this design argues about a scale-up server that resides in-between clients and servers and acts as a load balancer implemented as an on-path look-through cache. Although the cache itself can be effective, this design increases the number of hops, and the load balancer consists a single point of failure for the system. Additionally, in the typical case where the request traffic is huge, a single load balancer is a potential bottleneck degrading the system's performance, since it has to receive and process the requests of each individual client.

Furthermore, Bronson et al.[9] discuss a design where a similar small cache resides in every client node, even though all cache hits have to be sent to the next layer to validate that the value is not stale. As a result, just a small fraction of network bandwidth is saved, in the case of the home node confirming that the cache value is consistent, where the value does not need to be sent back to the client. However, the response time is not reduced, and the skewed accesses are not filtered by the caches.

On the other hand, approaches based on the look-aside cache architecture such as the one proposed by Nishtala et al.[39], present alternative obstacles. Their difficulties include significant latencies, due to the increased number of hops, in the case of cache misses while the complexity of handling those is imperfectly pushed to the client side.

Moreover, Li et al.[34] discuss a technique that leverages specialized hardware on the SDN switches to efficiently route a key either to the cache or the back-end nodes. However, this approach is limited by the processing and storage capability of the switches, and more importantly, this method restricts each packet to contain only a single key request. Thus, batching techniques that are beneficial for amortizing the costly TCP/IP processing can not be used.

Finally, an interesting approach presented by Liu et. al[36], argues for distributed in-network caching spread across the datacenter network. Although this approach can reduce the network traffic and potentially save energy on the back-end nodes, it re-

quires specialized hardware accelerators and more importantly it is highly complex to keep those caches coherent.

2.4.3 TCP/IP Offloading & Alternatives

Another way to tolerate skew in such workloads is by reducing the cost of their operations. Multiple studies[33, 42], have identified that the cost of such systems is dominated by the TCP/IP protocol. Therefore, this section discusses some methods for TCP/IP offloading and alternative networking approaches.

High expenses of TCP/IP are originated from the costly context switches and the transitions between kernel and user-space, caused by traditional kernel drivers. A recently open-sourced library DPDK[1], utilized in the work of Lim et al.[35], tries to overcome such overheads, by efficiently implementing network drivers in user-space. This approach, reduce the context switches and focuses on offering a cost-effective communication between the driver and the NIC(s) by efficiently utilizing the PCIe and the memory.

Furthermore, modern NICs include specialized hardware that offloads TCP/IP processing. More accurately, they offer what is known as TCP offload engine (TOE)[38], an integrated circuit included on the card which processes the TCP headers. This avoids the slow header processing on software, which also occupies CPU resources.

An alternative way to cope with the cost of TCP/IP processing is by batching requests [4] from clients to servers, which means that a single packet can contain multiple key-value operations. Although batching can alleviate the problem, it is not easily applicable for two reasons. Firstly, it requires finding several requests, originated from a single machine with the same destination node, in systems where the dataset is randomly sharded across multiple nodes. Additionally, this has to be done in a tiny fraction of time since all the user requests have to respect tight latencies.

However, none of the above offloading methods can make TCP/IP as cost-effective as RDMA, thus others [37, 29, 18] choose to replace the TCP/IP protocol, to get maximum performance. Nevertheless, since RDMA is prone to scalability issues, a more realistic approach for datacenter-scale systems, is RackOut[42] proposed by Novakovic et al. In more details, this work assumes a hybrid network, where the clients are communicating with the back-end nodes through TCP/IP, and the back-end servers are grouped and interconnected with an RDMA network. Leveraging this design, the cost of TCP/IP processing is amortized by evenly spreading the client requests across

the servers of a group. Consequently, once the packet processing is done, a back-end node need to check if the key resides locally. In the exceptional case of a local request, the execution proceeds normally, otherwise the request is served by directly accessing the memory of the key's home node, leveraging one-sided RDMA primitives. As a result, this method requires an excessive amount of bandwidth and further increases the response time of the request, since most requests incur an additional RDMA operation. Also, although this method evenly spreads the load among the CPU resources of a group, other performance-critical components such as the NICs or the memory of a popular node can still become a hot-spot in the system.

2.5 Summary

To summarize, none of the above techniques are able to completely mitigate the problem of skew efficiently. Existing replication techniques are reactive, can lead into consecutive replication actions, present difficulties to track replicas and do not offer strong consistency guarantees. On the other hand, caching techniques usually filter the skew, even though they are prone to hot-spot issues. Finally, RackOut evenly spreads the computation among nodes of the same group, but still, NICs and memory components can be affected by skewed data accesses. Furthermore, most of the requests require an additional hop through the network, which increases latency and bandwidth requirements.

Chapter 3

Symmetric Caching

3.1 Core Idea

As it is mentioned already, data-serving systems are characterized by skewed data accesses, which lead to serialization and hot-spots that limit their performance. The primary purpose of this thesis is to reverse this fact by concentrating on the cause of the problem. To achieve that, we propose *Symmetric Caching* a novel technique, which exploits skew to offer load balance and increase performance, by combining the best of caching and replication.

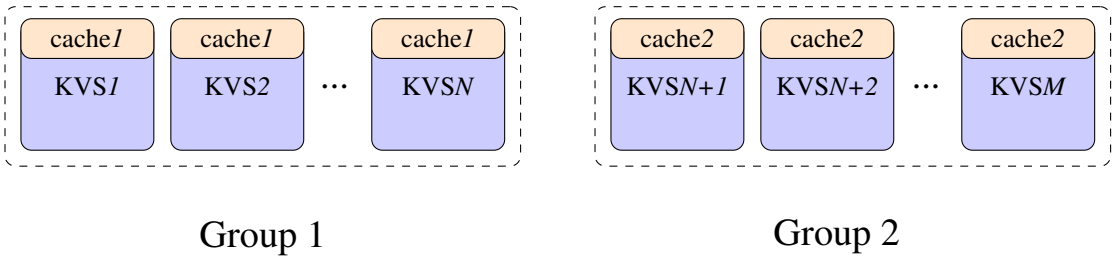

Figure 3.1: *M* node partitioned key-value store, clustered into two symmetric caching groups

In *Symmetric Caching*, data-serving nodes are grouped, and each one is equipped with an in-memory software cache, as shown in Figure 3.1. Machines that belong in the same group have replicated or identical caches that contain the most popular items stored in the group. In this case, skew works in favour of this method, since a small cache can be effective[20]. As the Figure 3.2 illustrates, caching a really small portion of the dataset can ensure more than half of the workload accesses to hit the cache.

Consequently, requests for the most frequently accessed keys, which are responsible for the load imbalances, can be served independently by any node within the group. As a result, there are no hot-spots in the system, and high concurrency can be achieved.

Although the solution may seem obvious at first glance, such a design may arise several questions. Thus, in the next sections, we provide answers to some of the most relevant and interesting questions.

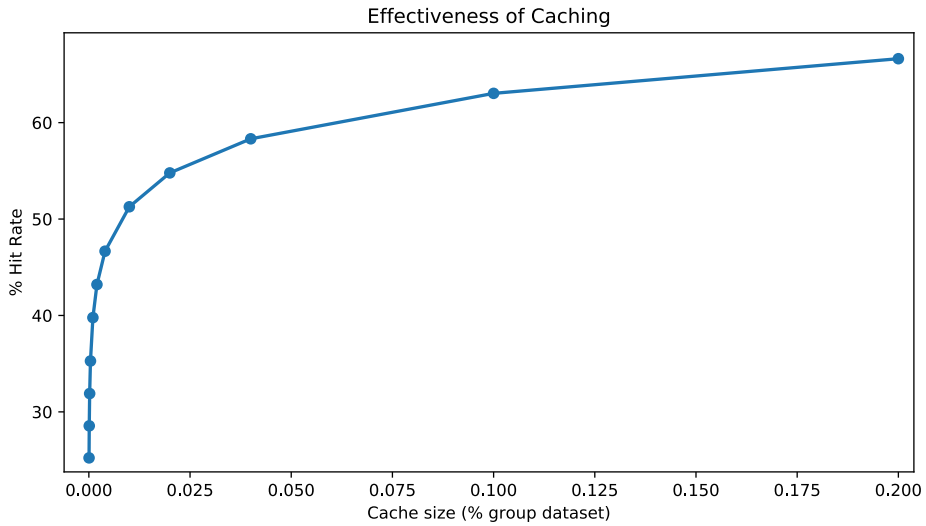

Figure 3.2: Cache effectiveness according to a dataset of 250 M keys following Zipfian accesses with $\alpha = 0.99$.

3.2 Scratching the Surface

Is the approach flexible and adaptable?

To begin with, data-serving systems consist the back-end of online services, where the data accesses and trends constantly change over time. The caches of the approach always contain the most popular objects, similarly to the work of Fan et. al[20]. Thus it differs from the conventional designs that require bringing every object in the cache before it gets accessed. This method avoids cache pollution with cold objects which are not accessed frequently. However, it requires a way to identify the hottest keys and update the contents of the caches on demand. This can be achieved in epochs, leveraging streaming algorithms[15, 12], such as probabilistic lossy counting, that approximate the hottest keys in a low-cost manner by sampling the accesses. Consequently, in our

design, the size of the cache can be dynamically changed from zero to an upper-bound, according to the on-demand requirements.

Furthermore, *symmetric caching* is designed to support different consistency models, that are further discussed in 3.3, which makes it adaptable to the system that is applied. As a result, this technique can enhance either systems that sacrifice consistency over performance, such as in-memory key-value caches, or others that require stronger consistency guarantees, like key-value storage or transactional-based systems.

How does it achieve load balance while being transparent?

In contrast with most related work on replication and caching that demand the intervention of client, Symmetric Caching has the advantage that systems can adopt it transparently. Existing solutions push complexity to the client-side, by either relying on the clients to handle cache misses or by requiring them to communicate with the back-end service and maintain an up-to-date location of the replicas. However, in our approach, the client just identifies the home server of the key, similarly to a system in the absence of caching and replication, and therefore the symmetric group it belongs to.

Once a group is determined by a key, the request is sent to any server in the group chosen either randomly or in a round-robin manner. Then due to the popularity cache the majority of the requests are served within a single hop, while the ones that miss both the cache and the local storage are propagated to the home node of the key in order to be served. Consequently, symmetric caching could be used to construct scale-out architectures very similar to ccNUMA[32], especially if RDMA communication is supported within those groups, where all the accesses go through a cache and in the case of a miss they are served through the fast internal network.

Our approach is able to offer load balance while being transparent, leveraging the replicated caches. More precisely, the requests are evenly spread across the machines, where the identical caches can serve requests for the hottest keys, which are dominating the traffic and are also responsible for causing load imbalances in the system. Finally, the cold requests that also miss the local storage require one more hop, even though they are served fast since the skew is filtered by the caches.

3.3 Replication Challenges

Replication, which is a known way to increase the concurrency beyond the scope of a single node, is the enabling mechanism for our load balancing technique. However, replication usually comes with a lot of burdens, which include tracking sharers, maintaining consistency and serialization of writes over replicas. In this section, we discuss how such problems are handled in the case of symmetric caching.

Does the approach require tracking replicas?

One major limitation of existing replication techniques in data-serving systems is the problem of tracking replicas either for dispatching the requests or for providing consistency among the replicas. In large-scale data serving environments, which are constituted by a large number of nodes and an enormous amount of distinct objects, the complexity and cost of tracking sharers in key granularity are very challenging. In contrast to related work, symmetric caching overcomes this difficulty since it does not require any meta-data, for tracking sharers. This is because once a key has been identified as hot, then a machine is a sharer if and only if it belongs in the same group with the home node of the key.

How does it ensure consistency?

Although data-serving accesses are characterized as read-mostly, when replication is applied handling writes is a major factor that determines system's performance. In a replicated environment reads can be served efficiently, in contrast to writes that require special actions to ensure consistency among the replicas. According to multiple studies, consistency maintenance is challenging[3] or almost impossible to be achieved[20] in non-trivial scenarios. However, we show that is not true when using *Symmetric Caching* and considering the low write-ratio of large-scale workloads.

In details, consistency maintenance of the replicated keys in Symmetric Caching can be reduced to the problem of keeping the caches within a group coherent, similar to traditional hardware caching. Thus, protocols for coherency actions are likewise required. However, data-serving systems have to apply such actions through software, over slower interconnects and in large-scale. As a result, there is a need to reduce the costly hops and make the protocols as distributed as possible, to avoid potential

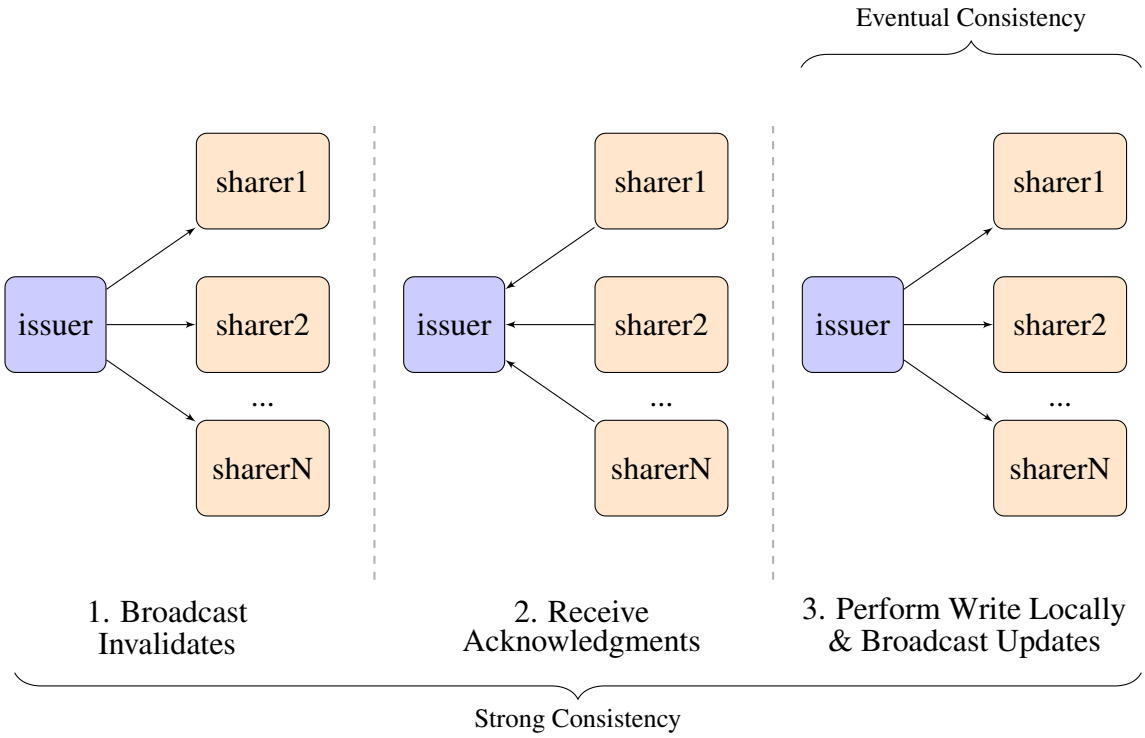

Figure 3.3: Required write actions for each consistency model.

hot-spots.

Implementing protocols based on a traditional directory would violate both of the above considerations. Instead, our protocols, as we already mentioned, leverage the symmetric feature of our technique. Thus, once a write hit the cache, it deterministically knows that every other node in the group is a sharer and needs to receive the new value.

The protocols that are used to handle writes in symmetric caching propagate the new values supporting two consistency models, as shown in Figure 3.3. The first one that offers *Eventual Consistency* (EC), saves performance by directly broadcasting the written value to the group. On the other hand, the *Strong Consistency* (SC) version trades some of its performance, to provide stronger guarantees by further ensuring write atomicity, through a classic two-phase commit protocol. More precisely, the SC model first sends invalidates and wait for acknowledgements. Once all of the acknowledgements have been received, proceeds to the final step which is the same as EC, where the write is performed locally, and updates to the sharers are sent.

Table 3.1: Protocols for Write Serialization

	Write Serialization	latency for serialization	Cost of Broadcast Actions
Primary Broadcast	propagate the write to the home of the key	1 hop	Home
Primary Timestamp	1.ask the home of the key for a timestamp 2. execute the write & broadcast actions using the timestamp	2 hops	Distributed
All Peers	distributed serialization using per key global timestamps (versions & machine id)	0 hops	Distributed

How does it accomplish write-serialization?

In both for both of the above consistency models, there is still the need for a consensus between the nodes for a single global ordering of writes that are proceeding concurrently on different nodes, also known as write serialization. In this work, we study three different protocols, which are summarized in Table 3.1, that accomplish write serialization.

More precisely, existing replication techniques usually serialize the writes by sending and executing all of them in just a single replica. In our case, this would mean that once a write hit a cache will have to be propagated to the primary(home) node that is responsible for executing all of the writes for the specific key, we refer to such protocols as *Primary Broadcast*. However, in the presence of skew, which is when replication is reasonable, serializing the writes on a single node, is even worse than serializing the reads. This is a consequence of the required costly write actions that have to be performed to ensure consistency, especially in the case of SC.

Nevertheless, protocols that achieve write serialization through the primary can be optimized in terms of load balance. Thus, another protocol that we study is the *Primary Timestamp*, which accomplishes serialization by requesting a timestamp from the primary node, and once the timestamp is received from the primary the execution of write proceeds locally. Such a protocol can evenly spread the costly broadcast actions of writes across all the replicas.

All Peers Protocol for Distributed Write-Serialization

However, both Primary protocols require extra network hops, which degrade performance, especially the *Primary Timestamp* that balances the load. To avoid those additional costs, we developed the *All Peers* protocol that accomplishes distributed write serialization, which is inspired by [10, 22] and is implemented via global virtual timestamps. In details, every replica maintains a virtual timestamp, which is implemented

with a version and the node id of the machine that is performing the write to the object. Furthermore, the timestamp is formed by the concatenation of these two variables. As a result, two concurrent writes on different machines always have different timestamps, and thus all the nodes can agree on a single global sequence of writes.

In the case of eventual consistency, this serialization method requires only small modifications in the execution of writes and updates, as listed in pseudocode 3.1. More precisely, once a write is performed, the local version is incremented, and the last writer variable is updated to the local machine id, afterwards the key, the value and the virtual timestamp (version, machine id) are broadcasted. When an update is received, is applied only if the received timestamp is higher than the local.

Finally, in contrast with eventual, the strong consistency version of this protocol is more complicated. Therefore, we provide a more detailed implementation of this protocol through the state transition matrix 3.4, which is formally verified for correctness. This implementation consists of 5 states and requires only 8 bytes of meta-data per cached object, which includes the virtual timestamp(5 Bytes), a lock(1 Byte), the current state(1 Byte) and a field for remaining acknowledgements(1 Byte).

	Read	Write	Receive Inv	Receive Ack	Receive Update
V	V -> V	Update_Timestamp() V -> W^A Broadcast Invs	if (Higher_Timestamp()) Store_Timestamp() V -> I^U Send Ack	X	X
I^U	Buffer the Read I^U -> I^{UR}	Update_Timestamp() I^U -> W^A Broadcast Invs	if (Higher_Timestamp()) Store_Timestamp() I^U -> I^U Send Ack	X	if (Same_Timestamp()) I^U -> V
I^{UR}	Buffer the Read	Buffer the write	if (Higher_Timestamp()) Store_Timestamp() I^{UR} -> I^{UR} Send Ack	X	if (Same_Timestamp()) I^{UR} -> V Replay Buffer
W^A	Buffer the Read W^A -> W^{AR}	Update_Timestamp() Broadcast Invs	If (Higher_Timestamp()) Store_Timestamp() W^A -> I^U Send Ack	if (Same_Timestamp() && Is_Last_Ack()) Broadcast Updates W^A -> V	X
W^{AR}	Buffer the Read	Buffer the write	If (Higher_Timestamp()) Store_Timestamp() W^{AR} -> I^{UR} Send Ack	if (Same_Timestamp() && Is_Last_Ack()) Broadcast Updates W^{AR} -> V Replay Buffer	X

Figure 3.4: State transitions and corresponding executing conditions of *All Peers-SC* protocol.

Listing 3.1: Pseudocode for eventual consistency with distributed write serialization.

```
perform_write(key, value){
  lock_object(key);
  write(key, value);
  key.timestamp.version++;
  key.timestamp.last_writer = local_node_id;
  timestamp = key.timestamp;
  unlock_object(key);
  broadcast_update(key, value, timestamp);
}

perform_update(key, timestamp){
  lock_object(key);
  if(key.timestamp < timestamp){
      write(key, value);
      key.timestamp = timestamp;
  }
  unlock_object(key);
}
```

3.4 One Step Further

What about concurrency over synchronization?

Traditional implementations of data-serving applications apply fine-grain locks to handle synchronization. Although this enables any core to process a request for a local object, it does not allow concurrent requests for the same key to proceed; instead it serializes them. Consequently, in the presence of skewed accesses, such systems consume multiple cores in order to serve requests for hot keys serially.

However, inside a single node concurrency for an object can be offered through efficient synchronization methods. Modern data-serving systems [19, 18] utilize optimistic concurrency control, allowing concurrent reads and exclusive writes(CREW) which increase the performance in the presence of skew. Nevertheless, a recent study[35] argues about the cost of synchronization and proposes to partition the address space in core granularity instead of a node. Despite the fact that this technique leads to exclu-

sive reads and exclusive writes(EREW) model, which performs poorly with a skewed workload, it eliminates the requirement of synchronization. Thus it can achieve better performance in the ideal case of uniform workloads.

Incremental Concurrency Control

One more benefit of symmetric caching is that it enables the architecture of systems, with the synchronization scheme, that we call *Incremental Concurrency Control*. This attribute achieves the best of both worlds, by exploiting the fact that heavy hitters are decoupled from the rest of the dataset. Therefore, caches, which are benefited by concurrent accesses since they serve the hottest keys, utilize optimistic concurrency control to implement the CREW architecture. On the contrary, the back-end system, which receives requests only for cold objects, can leverage the core-granularity dataset partitioning or EREW architecture, similarly to [35], to avoid synchronization costs. As a result, an item that is initially cold is served fast by a single core without any synchronization costs, and once its popularity reaches a threshold, it is replicated and moved on the symmetric caches, that leverage optimistic concurrency control to offer high concurrency and increase performance.

Is this solution beneficial anywhere else?

The grouping of back-end nodes, enforced with a cache, can provide further performance benefits. One existing problem of todays large-scale data serving systems is the increased number of required connections[17, 18, 39] between the number of clients and the back-end side since each client needs a connection with every single servicing node. In symmetric caching, the required connections of a client are one per group of servers. In other words, a client could be pinned to a single server in the group, and reduce the number of required connections dramatically. Although from clients perspective this is similar to the existing solution of Fan et al.[20], it avoids the overloading of the machine that holds the cache since different clients can be pinned to distinct machines that contain identical caches. However, this approach may affect the parallelism of the requests produced by a single client. Thus, there is a sweet-spot for choosing the number of connections between a client and a group that provides good performance and a small number of connections.

Symmetric caching provides additional advantages if the communication between the clients and the group happens via the costly TCP/IP protocol. This is due to the

ability of further enforcing the idea of batching multiple key requests into a single packet. In more details, in a system that supports symmetric caching, requests for a whole group could be batched in a single packet, in contrast to the common case of a single server. Finally, this can be even more beneficial in a set-up, where the intra-group network has higher bandwidth, or it supports RDMA capabilities similar to the one that RackOut[42] assumes.

Can you optimize Symmetric Caching further?

Although the requests are load balanced across all the nodes of the group, requests for cold keys that miss in the cache may require being served remotely. However, this replication approach can become more efficient by a more sophisticated request dispatching on the client side. Providing that the client can distinguish if a key is hot or not, which enables better decisions to be made about the destination of a request. More precisely, hot keys could be equally distributed among the nodes of the group and served in parallel from caches avoiding hot spots, while cold items could be directly sent to the home node. This optimization enables, all the request to require at most one hop, and avoids the probe of cache for cold requests. As a result, this set-up provides the illusion of a scale-up load balanced component, constituted by a group of collaborative nodes that serve requests for the most popular keys, while each server is still responsible for their own cold accesses. Finally, a way to distinguish popular from cold keys on the client side could be achieved by communicating a bloom filter per group, at the beginning of each epoch, initialized with the contents of the caches. Such a bloom filter for 100k keys and with false positive less than 0.1% would cost around 230KBytes.

3.5 Summary and Comparison with Related Work

In summary, symmetric caching is a skew mitigation technique that load balances and increases the parallelism of a system, by enforcing every node in the group with an identical popularity cache. Its symmetry, prevents the formation of hot-spots in the system, while it resolves the problem of tracking sharers. Furthermore, it is flexible due to its ability to adapt, both the cache size and the consistency guarantees, to the occasion. Additionally, this technique can be applied transparently to the clients and boost performance, exploiting novel replication protocols and synchronization schemes. Fi-

nally, symmetric caching can mitigate the problem of required connections and it can further enhance batching techniques, which offload the costly TCP/IP.

Symmetric caching performs better or equal in all critical aspects, compared to related work. More accurately, as Table 3.2 summarizes, any deployment of our technique requires equal or fewer hops than other caching methods, while it avoids serialization-points, by spreading the load evenly across the nodes of a group. Finally, Figure 3.5 depicts several benefits that our technique can offer related to skew, which most of the state-of-the-art solutions, are either missing or performing worse.

Table 3.2: Traditional vs Symmetric Caching

	Traditional Caching: Look-aside	**Traditional Caching: Look -through**	**Symmetric Caching: Transparent**	**Symmetric Caching: Optimal**
Clients responsibilities	handle cache misses	None	None	identify if key is hot
Cache load	100% queries	100% queries	(100% queries) / servers	(Hot queries) / servers
Hops with cache miss	3 machine transits	2 machine transits	2 machine transits*	No cache miss

* Latency when both cache and local storage miss.

Caching / Replication Technique	Determi-nistic Hashing (Transpa-rency)	TCP/IP amortization		Replication			Concurrency	
		Batching	Required Conne-ctions	Granu-larity	Tracking sharers (Meta-data)	Consistency	For Hot Items	Synch. Overhead & Contention
Symmetric Caching (Hybrid)	Yes	Per Group	1 per group	Per Key	Zero	Strong & Eventual	# total cores in the group (CREW)	Zero for cold keys (EREW)
SPORE (Replication)	No	Per server	All-2-All	Per Key	Per key	Eventual & Strong (mediocre performance)	Sharers	Every access
Facebook TAO(Hybrid)	No	Per server	All-2-All	Per shard	Per Shard	Eventual	Sharers	Every access
SwitchKV (Caching)	Yes	No batching	All-2-All	N/A	N/A	N/A	Single Scale-up Node	Every access

Figure 3.5: Overview comparison between skew alleviation techniques.

Chapter 4

Evaluation

4.1 Metodology

In this chapter, we evaluate the performance of the three different write serialization protocols and the impact of transparent Symmetric Caching when it is applied on existing data-serving systems, without any optimizations such as batching, using discreet-event simulations. We examine the influence of our technique over different consistency models, workload characteristics, infrastructure, and system settings.

Baselines & Consistency models

We choose to compare both strong and eventual modes of our technique over two state-of-the-art data-serving architectures, that reside in-memory and leverage low-latency RDMA communications that avoid the TCP/IP overheads. More precisely, we simulate an EREW based model using two-sided RDMA operations, that corresponds to FaSST [30] or Herd[29] design, which achieves the highest performance on uniform workloads. On the other hand, we emulate RackOut [40] a state-of-the-art skew mitigation method where reads are concurrently served by a group of nodes similarly to our technique, but instead of replication this technique leverages one-sided RDMA operations to implement a cross-machine CREW model.

Workloads

We created a trace generator that produces representative workloads, which match the characteristics of studied large-scale applications. More precisely, the trace generator provides an accurate trace according to a dataset size, a distribution of data accesses,

request arrival rate, and a particular write ratio. For this evaluation, we created datasets of 250 million keys, which follow Zipfian data access patterns with exponent $\alpha = 0.99$ while we vary the write ratio from read-only to 15%, since as we have already mentioned these workloads are read-intensive.

4.1.1 EUREKA Simulator

To provide insights and evaluate our technique about symmetric caching, we build *EUREKA* a discreet-even simulator for distributed data-serving systems. In more details, this simulator mimics the operation of a particular system, on a configurable infrastructure, consisting of distinct nodes and the network that it connects them. Furthermore, each simulated machine consists of latency and bandwidth operating components, such as cores (latency), NICs and memory (bandwidth). The internal implementation of such components is based on queues, and a simulated infrastructure looks similar to the Figure 4.1.

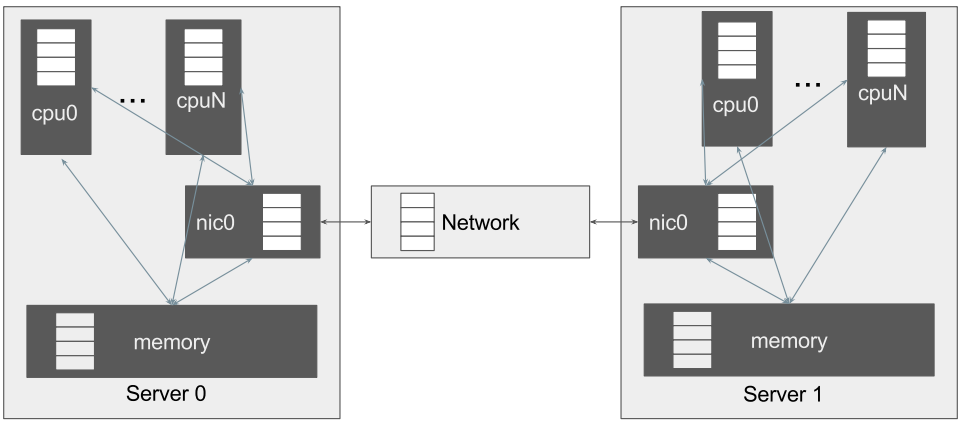

Figure 4.1: Simulator overview of two nodes, each with N+1 cores, a NIC and memory.

To perform precise simulations, *EUREKA* gets a trace of requests or operations, such as GETs and PUTs, a configuration of the infrastructure and a description of a data-serving system. More precisely, this description maps the real system's implementation of such operations, to a series of latency and bandwidth actions, which are executed by the corresponding components of the simulator. Consequently, the smaller granularity of actions results into more representative simulations. Therefore, in our

simulations, each operation usually consists of more than ten primitive actions, where each one cost was extracted from the literature. For example, serving an operation for a key that resides on a remote server, is broken into the actions similar to the Figure 4.2.

```
[ Local_CPU_processing_for_A, Latency_CPU_2_MEM,
  Local_Memory_BW_req_for_B, Latency_MEM_2_CPU,
  Local_CPU_processing_for_C, Latency_CPU_2_NIC,
  Local_NIC_BW_req_for_D, Network_latency,
  Remote_NIC_BW_req_for_E, ..., Local_CPU_processing_for_H ]
```

Figure 4.2: Simulator actions for a remote operation.

Additionally, EUREKA supports the simulation of the configured systems with or without the technique of symmetric caching. Where each node is enhanced with a popularity cache, with configurable size, protocol and consistency actions. Consequently, we can extract information and argue about the costs of different protocols, and also compare state-of-the-art systems with our solution.

The simulator is able to provide over-time and average results, for the configured set-up, in a component, node, and system granularity. Furthermore these results, for memory, network and cores include metrics like idleness, latency, throughput, utilization and the cost of coherence actions in each component, when symmetric caching is enabled. Finally, the simulator also provides metrics for the requests, such as latency and read staleness statistics.

4.2 Results

In this section, we first analyse the performance of the three different coherence protocols, when our technique is applied on both substrates. We then compare the performance between pure baseline systems and a system enhanced with Symmetric Caching. Afterwards, we present three case studies that demonstrate the sensitivity of our technique based on workload and infrastructure parameters. The evaluation parameters used for these experiments are summarized in Table 4.1, where in the case of multiple values the bold value is the default one, which is used unless stated otherwise.

Table 4.1: Evaluation Parameters

Configuration	Parameter	Values
Infrastructure	servers	**20**, 40, 60, 100
	cores per server	32
	network B/W	10, **40**, 56, 100, 200, 400, 560 Gbits/s
	memory B/W	∼35 GBytes/s (effective B/W of 4 channels)
	cache size	250k keys (0.1% of the dataset)
System	baseline / substrate	CREW, EREW
	symmetric caching	on / off
	protocol	Primary Broadcast, Primary Timestamp, **All Peers**
	consistency	Eventual(EC) , Strong(SC)
Workload	write rate	0, 1, 2, **5**, 10, 15 %
	zipfian exponent	0.99
	size	250 Milion keys

4.2.1 Protocols & Performance of Symmetric Caching

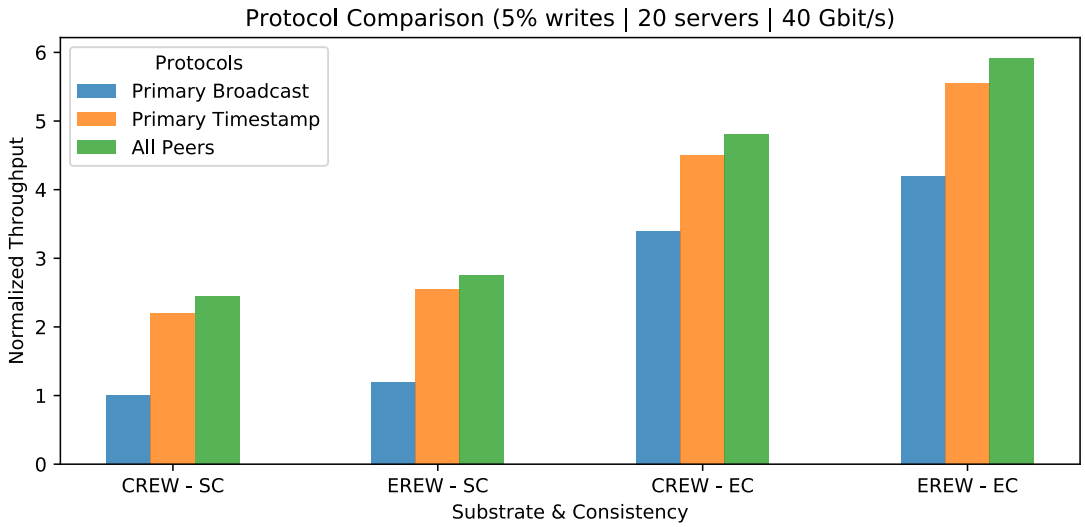

Figure 4.3: Symmetric Caching applied to EREW/CREW substrates, using different protocols and consistency models (Normalized to CREW - SC: Primary Broadcast).

To quantitatively determine the best coherence protocol for Symmetric Caching, when it is applied to different substrates, we ran simulations varying the protocols and the consistency models. Comparing the protocols in Figure 4.3, the Primary Broadcast performs worse than the other two, especially on the strong consistency setting. This is because the home node of each key is responsible for performing the computational heavy broadcast actions, which consist of multiple phases in strong consistency. As expected, the primary node of a hot key, which is receiving a lot of writes, becomes overloaded and limits the performance of the system. On the contrary, Primary Timestamp and All Peers are performing the broadcast coherence actions from any node, avoiding hot-spots. However, Primary Timestamp requires two additional network hops, which degrade its performance, compared to the All Peers protocol, which achieves write serialization in a distributed manner. Finally, it worth mentioning that the performance gap between the All Peers and the rest of the protocols would be enlarged if the evaluation was over TCP/IP protocol where network hops are more costly.

Furthermore, the Figure 4.3 depicts that Symmetric Caching is even more beneficial when it is applied on an EREW substrate system. As it is described earlier, this is because skew is filtered by the caches then the rest of the accesses for cold items are sufficiently served in core granularity, where the absence of synchronization overheads boosts performance.

In summary, the best configuration according to our results is when Symmetric Caching is combined with an EREW substrate model and utilizes the All Peers protocol for the coherency actions. Consequently, in the rest of the results, this setting is used unless it is stated otherwise, to evaluate and compare against the baselines.

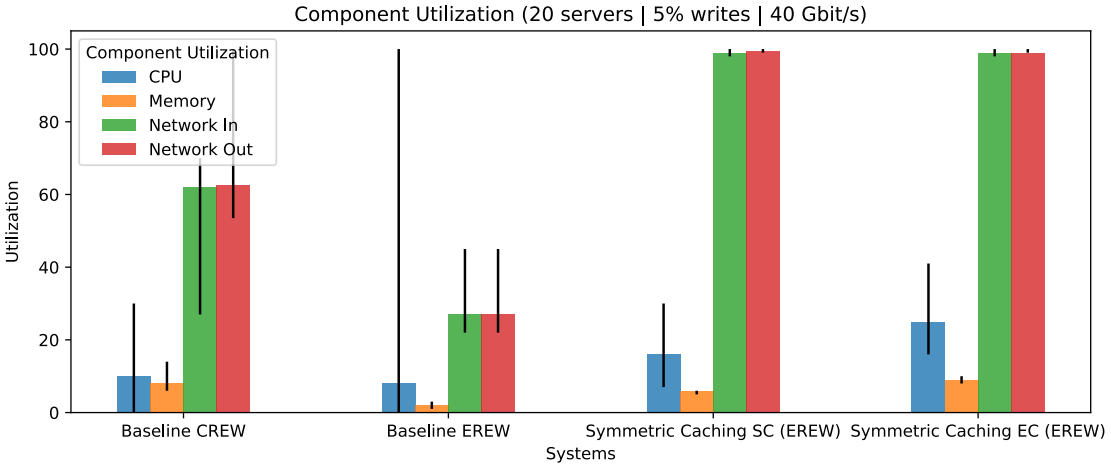

Figure 4.4: Load (im)balance and bottlenecks of systems with(out) symmetric caching.

To provide more insights about the bottlenecks of each system in the presence of skew, Figure 4.4 summarizes the utilization of the systems' performance-critical components. In this graph, the coloured and the thin black bars represent the utilization of the average, and the range between the least and the most utilized, component in the system respectively. We can deduce that there is high load imbalance in both baselines since the black lines span a broad range of values. More precisely, the core utilization of the EREW baseline stretches over the whole scale, while the average utilization is below 10%. In other words, the entire system is bottlenecked by a single or small amount of cores, which are responsible for the hottest keys, while the majority of cores is operating below 10%.

On the other hand, the CREW baseline has a more balanced core utilization, since the computational part of the requests is spread among all the servers. This is due to the use of one-sided RDMA to access the remote data, which does not occupy any CPU resources on the home node. However, the network traffic is still imbalanced, since most of the requests are targeting the nodes that are storing the most popular objects. More accurately, in this scenario the outgoing network traffic from the hottest nodes is the main bottleneck, since this system's design fetches multiple remote objects, via one-sided RDMA accesses, to avoid multiple round trips[18].

On the contrary to the baselines, the system with Symmetric Caching, either on

eventual or strong consistency setting balances almost perfectly the network traffic, while keeping the variation of utilization across the cores smaller than the CREW baseline, even though it is applied on an EREW substrate. This result validates that symmetric caching achieve load balance and filters the skew by serving hot requests concurrently from all servers in a group, and that is enabling the saturation of the NICs in all of the nodes.

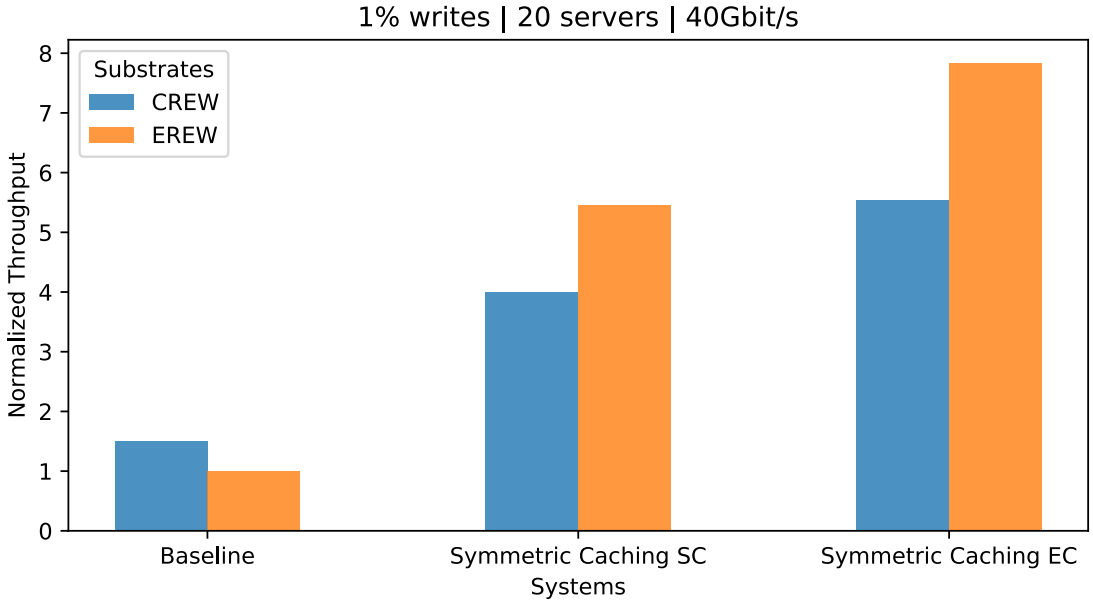

Figure 4.5: Performance comparison between native and baselines enhanced with Symmetric Caching (Normalized to EREW - Baseline).

Consequently, our technique can multiply the performance of existing state-of-the-art systems. The Figure 4.5, shows that in a read-mostly workload(5x more writes than what Facebook reports), our technique can empower systems to accomplish exceptional performance in both eventual and strong consistency. More precisely, the EREW system enhanced with Symmetric Caching achieves 5.2x and 3.5x over the CREW baseline, that already attempts to mitigate the skew, and 7.8x and 5.4x over the EREW baseline, for eventual and strong consistency respectively.

Finally, keeping the consistency for the hottest items, when each one is replicated 20 times, is not a considered a trivial scenario, even with our efficient consistency maintenance. Thus, Figure 4.1 quantifies the cost of the coherency actions for 5% writes, both for strong and eventual consistency, where faded and coloured bars correspond to the total and coherence utilization respectively. It is evident from the graph that the cost of coherence actions is affecting mainly the network bandwidth. More

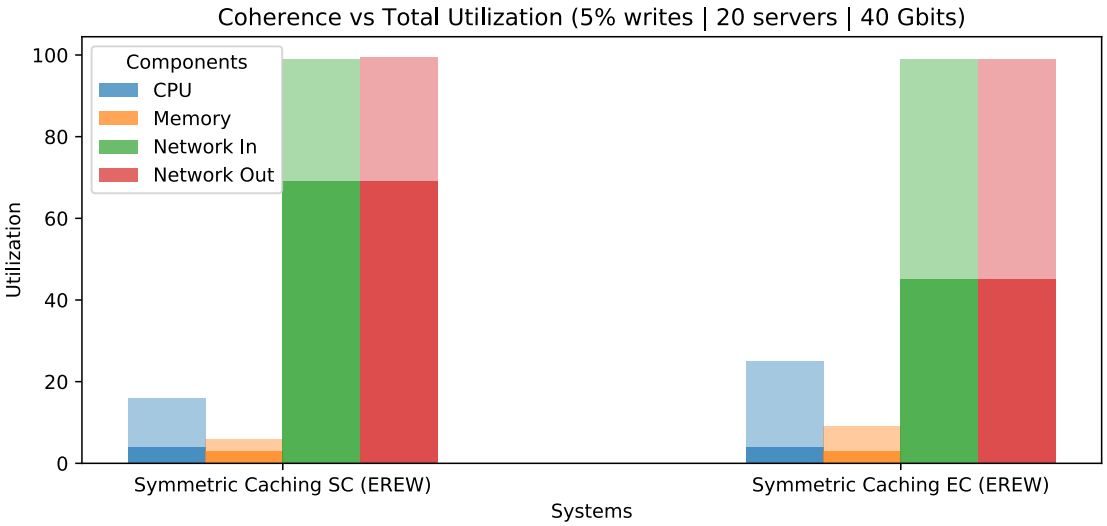

Figure 4.6: Average utilization for coherence actions (coloured bars) over the average total utilization (faded bars).

precisely, the cost in the eventual setting is 45% of the available bandwidth, while it rises to almost 70% for the strong consistency. Consequently, there is an opportunity to further increase the performance of the approach by reducing the cost of such actions.

4.2.2 Sensitivity of Symmetric Caching

The performance of our approach is mainly influenced by the write ratio, the available network bandwidth and the number of replicas (servers in the group). Thus, in order to examine the behaviour and limits of symmetric caching compared to the native baselines, we conducted three studies where we vary each of the critical factors.

The first study, depicted in Figure 4.7, is based on available network bandwidth, which ranges from 10 to 560 Gbit/s. The graph confirms that increasing the network bandwidth is not affecting the performance of the EREW baseline, which is bottlenecked by the CPU. Furthermore, the CREW baseline performs steadily worse for both the EC and SC set-ups of Symmetric Caching, even on the excess bandwidth configurations, where it achieves 1.75x and 1.5x less throughput respectively. Another interesting point is that Symmetric Caching-EC achieves saturation on 200 Gbit/s, in contrast to SC and CREW baseline, which requires double the bandwidth, due to costly actions on writes and load imbalances respectively. Finally, it worth noticing that the performance difference between the EC and SC is quite small when they have both reached their peak, which implies that their performance difference is heavily influ-

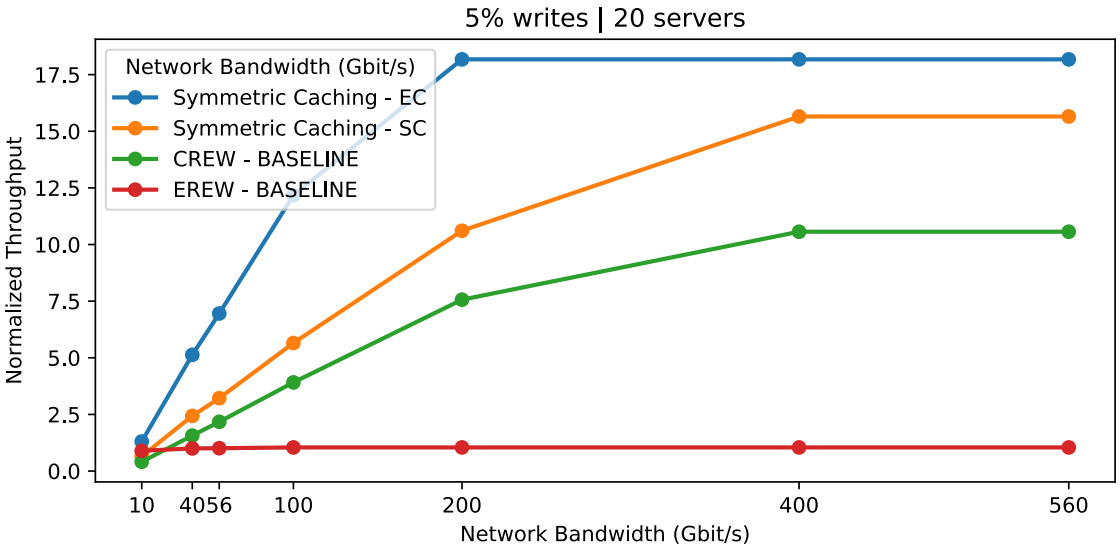

Figure 4.7: Study of baselines and symmetric caching to network bandwidth (Normalized to EREW - Baseline: 10 Gbit/s).

enced by the lack of network bandwidth.

The Figure 4.8 evaluates the sensitivity of the systems in the presence of different write ratios, which we varied from 0 to 15%. This graph illustrates that the CREW baseline slightly increases performance, in contrast to the EREW that is unaffected, as the write ratio rises. This is due to the lower cost of writes regarding network bandwidth in the case of the CREW model. Moreover, we see that the strong consistency setting of symmetric caching for this configuration is beneficial with write ratios below 10%. On the other hand, the eventual set-up, even on 15% writes, is still achieving almost 1.5x compared to the performance of the best baseline.

Finally, the scalability of the baselines and our approach in the presence of skew is illustrated in Figure 4.9, where we vary the number of servers, which also correspond to the quantity of replicas, from 20 to 100. Where the strong consistency variant of Symmetric Caching is not scaling beyond the 40 servers similarly to the EREW baseline, however, it still achieves better performance than the leading baseline. On the contrary, Symmetric Caching with eventual consistency is scaling quite well achieving 9x and almost 3.5x from EREW and CREW baseline respectively.

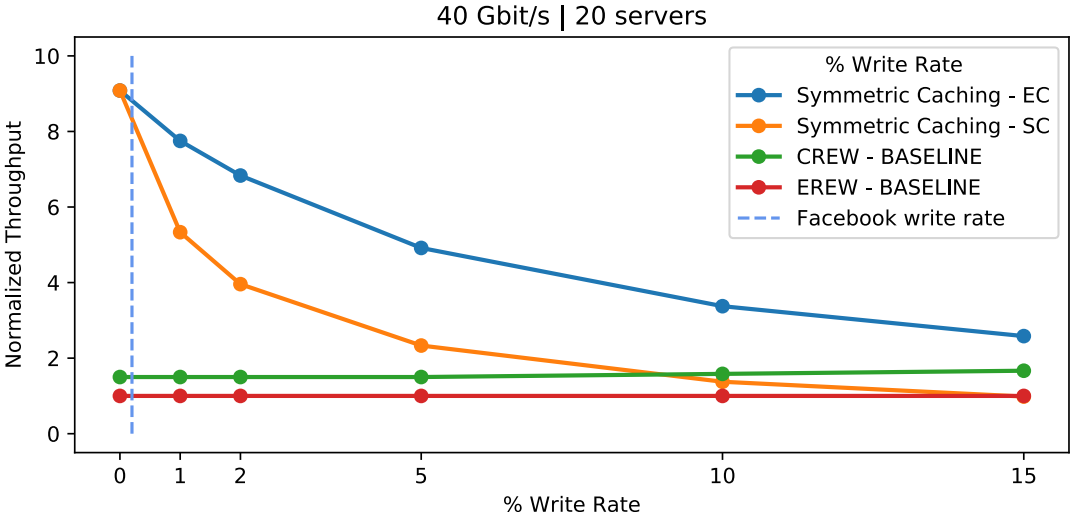

Figure 4.8: Sensitivity of baselines and symmetric caching to write ratio (Normalized to EREW - Baseline: 0% writes).

4.3 Summary

In summary, deploying an efficient protocol such as All Peers can significantly increase the effectiveness of our technique, even on low write rates. Additionally, we provided evidence that EREW can not tolerate skew, and that CREW mitigates the load imbalance from the CPU, but it can not accomplish the same for the network. However, when our technique is applied, even on an EREW substrate, load balance in all three major components, is ensured. Consequently, when the baselines are enforced with Symmetric Caching, they can achieve up to 7.8x and 5.4x increase in their initial performance, with EC and SC respectively. Based on the sensitivity studies, our technique can tolerate write ratios almost up to 10% for SC and more than 15% for EC, and still be advantageous. Finally, the studies revealed that Symmetric Caching remains beneficial as the network bandwidth or the number of servers increases, while the EC further enforces the scalability of the system.

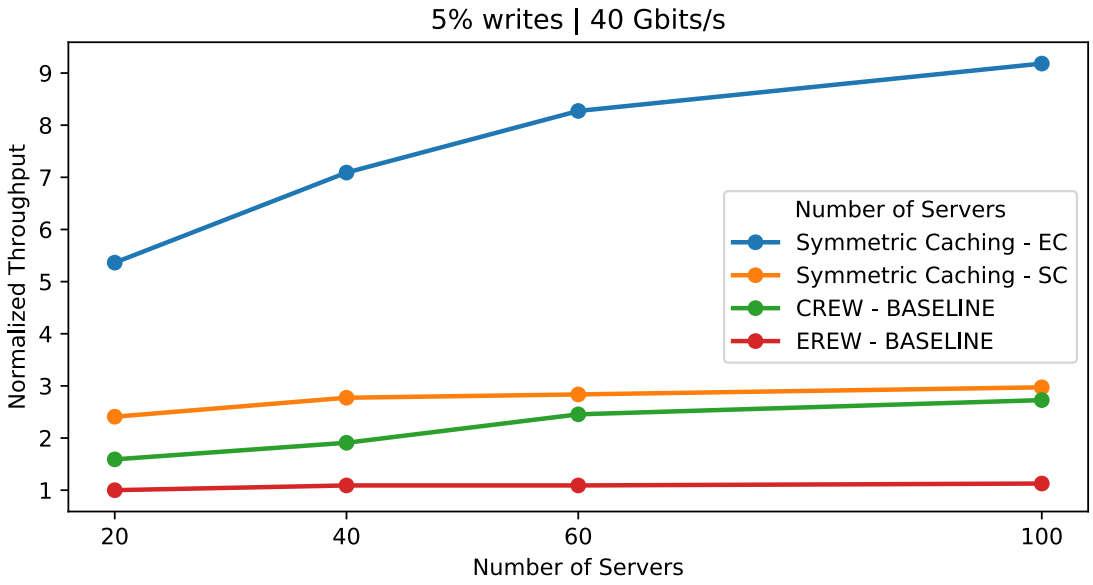

Figure 4.9: Scalability comparison between symmetric caching and baselines in terms of servers (Normalized to EREW - Baseline: 20 servers).

Chapter 5

Conclusions and Future Work

This thesis proposed and evaluated *Symmetric Caching*, a novel technique that combines the best of caching and replication, to mitigate skew of large-scale applications and increase the performance of data-serving systems. To achieve that, network inefficiencies, workload characteristics, and system considerations were taken into account, while the main focus was laid on replication and synchronization.

To provide consistency over non-trivial scenarios of replication, we designed and implemented protocols that load balance the consistency actions over all the replicas of an object, by offering distributed write-serialization that avoid costly network round trips. We provided evidence that such protocols can significantly increase the performance over the traditional protocols, especially on strong consistency setting where they can achieve more than 2x increase in a scenario of just 5% writes.

We show that although synchronization across machines is able to increase the concurrency, is not as effective as replication, even when they are applied over low-latency networks. However, the intra-node synchronization matters and thus we proposed *Incremental Concurrency Control*. A new synchronization scheme, that according to the online demand, an object can be either accessed by avoiding any synchronization overheads or through the optimal synchronization method of optimistic concurrency control.

Through the discreet-event simulator that we build, we proved that symmetric caching can almost perfectly balance the load among the nodes and achieve numerous times the performance of state-of-the-art systems that take skew into consideration, both in eventual and strong consistency setting. Additionally, we provided evidence that our approach remains beneficial in the excess of network bandwidth and increases the scalability of existing systems.

Finally, despite the fact that Symmetric Caching is able to balance the load and offer the performance of existing systems multiple times, there is plenty of space for improvements. Motivated by the results of the simulator, reducing the cost of consistency actions seems promising to boost the performance of the approach further and close the gap between strong and eventual consistency. A direction to achieve this is to leverage the hardware, either by exploiting existing multicast capabilities of switches to propagate the broadcast actions or by totally implementing the consistency protocols into FPGAs. Lastly, another interesting path for future research is to study the implication of system failures into the approach and increase its fault tolerance, possibly through the use of non-volatile memory.

Bibliography

[1] Dpdk boosts packet processing, performance, and throughput. https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html. (Accessed on 08/13/2017).

[2] Facebook now has 2 billion monthly users and responsibility. https://techcrunch.com/2017/06/27/facebook-2-billion-users/. Accessed: 2017-07-20.

[3] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 13–13, Berkeley, CA, USA, 2015. USENIX Association.

[4] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. 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, pages 1185–1196, New York, NY, USA, 2013. ACM.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[6] M. Beck. Maximize your software-defined data center infrastructure efficiency with rdma-enabled interconnects - the data center journal. https://tinyurl.com/y9r5w85p, 10 2015. (Accessed on 08/13/2017).

[7] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, Mar. 2016.

[8] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 241–252, New York, NY, USA, 2010. ACM.

[9] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[10] S. Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, Oct. 2014.

[11] 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*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[12] G. Cormode and M. Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19(1):3–20, Feb. 2010.

[13] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[15] X. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic lossy counting: An efficient algorithm for finding heavy hitters. *SIGCOMM Comput. Commun. Rev.*, 38(1):5–5, Jan. 2008.

[16] L. Ding, P. Kang, W. Yin, and L. Wang. Hardware tcp offload engine based on 10-gbps ethernet for low-latency network communication. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 269–272, Dec 2016.

[17] A. Dragojevic, D. Narayanan, and M. Castro. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 40(1):3–14, 2017.

[18] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[19] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[20] 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*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[21] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[22] R. Guerraoui, D. Kostic, R. R. Levy, and V. Quema. A high throughput atomic storage algorithm. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, pages 19–, Washington, DC, USA, 2007. IEEE Computer Society.

[23] T. Hamada and N. Nakasato. Infiniband trade association, infiniband architecture specification, volume 1, release 1.0, http://www.infinibandta.com. In *in International Conference on Field Programmable Logic and Applications, 2005*, pages 366–373.

[24] O. Hermes. *Leveldb*. Bellum Publishing, 2011.

[25] Y.-J. Hong and M. Thottethodi. 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, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[26] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches.

In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[27] InfiniBand Trade Association. *RDMA over Converged Ethernet (RoCE)*, 4 2010.

[28] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Washington, DC, USA, 2011. IEEE Computer Society.

[29] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, Aug. 2014.

[30] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016. USENIX Association.

[31] 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*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[32] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. *SIGARCH Comput. Archit. News*, 25(2):241–251, May 1997.

[33] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM.

[34] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 31–44, Berkeley, CA, USA, 2016. USENIX Association.

[35] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. 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, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.

[36] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGOPS Oper. Syst. Rev.*, 51(2):795–809, Apr. 2017.

[37] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[38] J. C. Mogul. Tcp offload is a dumb idea whose time has come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[40] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out numa. *SIGARCH Comput. Archit. News*, 42(1):3–18, Feb. 2014.

[41] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. An analysis of load imbalance in scale-out data serving. *SIGMETRICS Perform. Eval. Rev.*, 44(1):367–368, June 2016.

[42] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The case for rackout: Scalable data serving using rack-scale systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 182–195, New York, NY, USA, 2016. ACM.

[43] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.

[44] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: Managing server clusters on intermittent power. *SIGARCH Comput. Archit. News*, 39(1):185–198, Mar. 2011.

[45] D. Sidler, Z. Istvn, and G. Alonso. Low-latency tcp/ip stack for data center applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.

[46] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, New York, NY, USA, 2017. ACM.

[47] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot. Fat caches for scale-out servers. *IEEE Micro*, 37(2):90–103, Mar. 2017.

[48] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. Hydradb: A resilient rdma-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 22:1–22:11, New York, NY, USA, 2015. ACM.

[49] Y. Yang and J. Zhu. Write skew and zipf distribution: Evidence and implications. *Trans. Storage*, 12(4):21:1–21:19, June 2016.

[50] X. Zhao, P. Balaji, and W. Gropp. Scalability challenges in current mpi one-sided implementations. In *2016 15th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 38–47, July 2016.