Scaling Data Stores with Skewed Data Access: Solutions and Opportunities*

Shahram Ghandeharizadeh, Haoyu Huang

Database Laboratory Technical Report 2019-03

Computer Science Department, USC

Los Angeles, California 90089-0781

{shahram,haoyuhua}@usc.edu

Abstract

With a multi-node data store, a skewed pattern of access to data results in formation of hot spots and bottleneck servers. In its most extreme, one server out of many may dictate the overall processing capability of the system, reducing the throughput of a thousand node system to that of one node. This paper presents alternative bottlenecks due to a skewed pattern of data access and a taxonomy of solutions to resolve them. We identify use of the emerging RDMA as an opportunity to design and implement novel load balancing algorithms.

1 Introduction

Real-world workloads exhibit a highly skewed access pattern to data [3, 11]. For example, Facebook [3, 19] reports their access pattern consists of 90% of requests referencing 10% of keys with a long tail where less than 1% of requests reference 30% of keys.

With a multi-server data store or cache manager, a skewed pattern of access to data may result in different forms of bottleneck that limit performance, scalability, and elasticity during peak system load. We name this the Skewed Data Access (SkeDA) challenge. SkeDA may be attributed to the following forms of bottleneck:

1. NIC-bottleneck: The network interface card (NIC) bandwidth of a server becomes fully utilized. This happens when many concurrent requests read (write) large entries from (to) the same server.

2. CPU-bottleneck: All CPU cores become fully utilized. This is typically due to the processing of data prior to its transmission across the network, e.g., decompressing values or decrypting them for use by a client.

*Appeared in the *8th Workshop on Scalable Cloud Data Management*, co-located with IEEE BigData, Los Angeles, CA, December 9-12, 2019.

3. Secondary storage bottleneck: The bandwidth of the secondary storage device (disk, solid state disk, NVDIMM-N) is exhausted due to many concurrent transactions being directed to a few nodes. Once these transactions commit, they flush their log records to secondary storage devices to implement atomicity and durability properties of transactions. This may exhaust bandwidth of the secondary storage of a few (potentially one) nodes.

4. Core-bottleneck: A few (potentially one) cores out of many become fully utilized. This typically happens when threads are pinned to cores with one thread assigned a portion of memory [30, 8]. Keys assigned to this portion of memory are frequently accessed, causing this thread to utilize its assigned core fully.

5. Memory-bottleneck: The memory of one server out of many exhibits a high cache miss rate. In its extreme, the average cache hit rate across all servers is high while one server exhibits a thrashing behavior. This is due to a skewed pattern of data access that causes the fraction of working set assigned to the bottleneck server to exceed its memory size. This phenomena may occur even though the administrators provisioned total memory size of the servers to exceed the working set size.

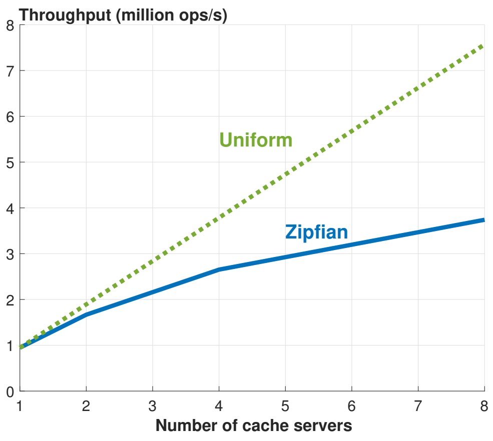
6. Process-bottleneck: A few (potentially one) processes out of many dictate the overall performance while system resources such as CPU and NIC are underutilized. Accesses to key data structures protected by synchronization primitives cause this bottleneck [43, 42]. Example data structures include a hash table to facilitate entry lookup or an LRU queue that identifies entries to evict. Concurrent threads in a process block on one synchronization primitive (semaphore/mutex) and wait instead of using system resources to perform useful work. This limits vertical scalability of the system.

The first five focus on the impact of SkeDA on a physical system resource and its utilization. The sixth focuses on synchronization primitives as programming constructs and how SkeDA causes concurrent requests to wait on one another using one or more of these primitives.

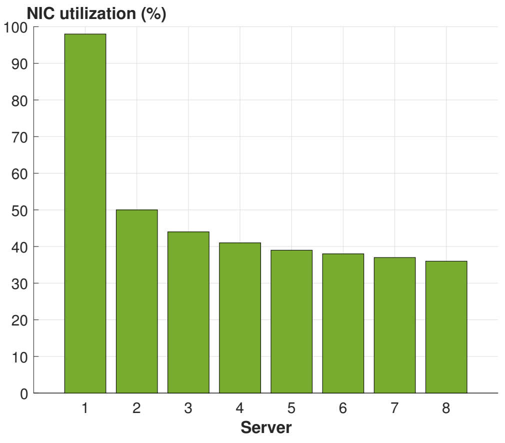
To illustrate the impact of a resource bottleneck due to SkeDA, consider the NIC-bottleneck. Fig. 1(a) shows the horizontal scalability of a multi-node memcached server with the YCSB workload C using both a uniform and a Zipfian (skewed) pattern of data access. We report the observed increase in throughput on the y-axis as we increase the number of cache servers from 1 to 8 on the x-axis, see Fig. 1(a). With the uniform access pattern, the configuration scales almost[1] linearly. With the Zipfian access pattern, scalability is sub-linear. Relative to 1 server, it approximates a 4 fold increase in throughput with 8 servers. In essence, the processing capability of the system with 8 servers is halved with the Zipfian access pattern. This is due a NIC-bottleneck. Fig. 1(b) shows NIC of one server is almost fully utilized while the remaining 7 are idle more than 50% of the time.

The contributions of this paper are twofold. First, we provide a taxonomy of existing solutions to mitigate the above bottlenecks. Second, we introduce use of the new *Remote Direct Memory Access* (RDMA) [24] as an opportunity to develop novel load balancing algorithms that have not been fully explored to date.

[1]It suffers from temporary bottlenecks attributed to randomly generated requests that collide on one server.



(a) Performance as a function of the number of cache servers.



(b) Network interface card (NIC) utilization of each individual cache servers with a Zipfian access pattern.

Figure 1: Scalability and NIC utilization with a uniform and a Zipfian access pattern.

2 Solutions

2.1 Mitigating NIC, CPU, and Secondary Storage Bottleneck

We classify studies that mitigate NIC, CPU, and secondary storage bottlenecks into three categories: 1) front-end caching [15], 2) erasure coding [35], 3) data migration and replication techniques to balance load across servers [8, 37, 1, 26]. These studies treat a server as overloaded once its NIC, CPU, or secondary storage exhibits a high utilization for a sustained period of time.

Front-end caching is based on a fast cache (named front-end cache) with the processing capability equal to or exceeding the total processing capability of all N cache servers [15]. It assigns the most popular data items to the front-end cache. It is based on the theoretical result that proves a server receives the same amount of load T with a high probability when the total system load is N*T. The number of data items assigned to the front-end cache is O(N log N). Hence, the memory size of the front-end server is a function of the number of servers (and not the number of data items). This theoretical result is implemented in different ways. Facebook [25] embeds a small cache in its front-end web servers that absorbs traffic to the most popular items. NetCache [27] stores the most popular items in the top-of-rack switch. The switch serves a read request when its referenced data item observes a NetCache hit. A single switch is able to process 2 billion requests per second.

Erasure coding distributes the load of a popular data item across multiple servers by encoding it into k data strips and r parity strips. A read request fetches any k strips out of k+r strips to reconstruct the data item. A larger k distributes the load more evenly. Erasure coding has two major overheads. First, it incurs a higher network usage since it must store and transmit the additional metadata with a replica to reconstruct a data item. For example, EC-Cache [35] reports that erasure coding incurs 10% network bandwidth overhead. Second, a put becomes more expensive as the data item must be encoded and written to multiple servers. With a write-heavy workload, a caching layer must adjust this technique to ensure its overhead do not outweigh its benefits.

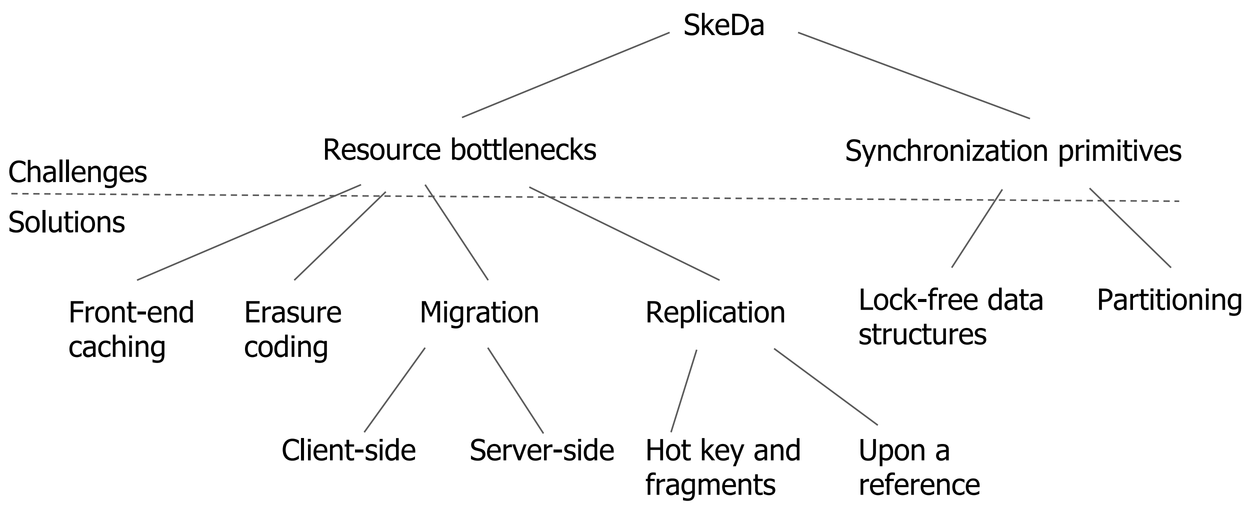


Figure 2: Taxonomy of challenges and solutions.

Data re-organization techniques may migrate or replicate data to balance load. A typical cluster has a coordinator that partitions key space into *fragments* and assigns one or more fragments to a server. A monitoring infrastructure determines the maximum load a server may handle, its current load, and load imposed by requests referencing each fragment. It identifies servers with high CPU or NIC utilization and reports them to a coordinator. The coordinator computes a new assignment that assigns fragments from an overloaded servers to underutilized servers. Next, the coordinator notifies servers of the new assignment. Finally, clients and servers either migrate or replicate an impacted fragment from its source server to its destination server.

A data re-organization algorithm has two objectives: 1) distribute the load evenly across servers, 2) minimize the number of impacted fragments. A formal specification of the first objective is as follows:

$$\text{minimize} \sum_{i=1}^{S} |L_i - L/S| \tag{1}$$

where S is the total number of servers, L is the total load, and L_i is the load of server i. The second objective may be formalized as:

$$\text{maximize} \sum_{i=1}^{S} F_{i,j} \cap F_{i,j+1} \tag{2}$$

where $F_{i,j}$ is the assigned fragments to server i with the current placement j. $F_{i,j+1}$ is a candidate placement computed by a migration algorithm.

Several studies propose a class of heuristics to compute a new assignment in polynomial time since computing its optimal solution is NP-hard [26, 1, 37]. E-Store [37] manages load balancing at the granularity of a data item. It monitors utilization of each server and starts monitoring statistics on individual data items only when it detects an overloaded server. It employs a greedy algorithm that assigns the hottest data item to the least loaded server.

Slicer [1] defines the weight of a move $F_{i,j+1} - F_{i,j}$ as its reduction in load imbalance divided by its key churn. The algorithm executes moves with the highest weight and terminates when it exhausts the key churn budget, e.g., 5% of the key space.

An objective of a data migration and replication technique is to be performed as a part of the system processing a request. Several studies use either a client-side or a server-side solution to migrate or replicate an impacted data item from its source server to its destination server. Migration maintains one replica of a data item [28, 13, 18, 17, 32, 26, 45]. Replication increases the number of replicas of a data item and is most effective for read-heavy workloads [22, 23, 16, 7, 10, 25, 36, 2].

A client-side solution populates the destination server using the data items fetched from source server [18, 17, 32, 26, 45]. Specifically, a client looks up a data item in the destination server first and consumes the value immediately upon a cache hit. Otherwise, the client looks up the source server. If found, it inserts the data item into the destination server. With migration, the client also deletes the data item in the source server. Clients must use leases to ensure consistency of data with concurrent readers and writers during either migration or replication [18, 23].

A server-side solution migrates identified fragments to the destination server asynchronously. Rocksteady [28] and Squall [13] are two migration techniques for a data store. A client directs requests to the destination server immediately after the assignment change. The destination server pulls the requested data from the source if it is not available. The destination-driven data migration is motivated by the following. First, it sheds load away from the overloaded source server immediately. Second, the migration completes faster with the destination server pulling data items since it has idle resources. Third, the destination server synchronizes client requests to ensure consistency.

Replication is predominantly used to enhance the availability of data in the presence of failures, e.g., quorum based replication [29]. We focus on replication techniques used to resolve load imbalance. These can be classified into techniques that replicate either hot data items [16, 23] and fragments [25] or a data item upon reference [7, 10]. ccNUMA [16] identifies the hot data items and replicates them across all servers. A client issues requests to any server. A server provides a response if the referenced data item is identified as hot and it has a copy of it. Otherwise, it fetches the data item from the primary server via RDMA. If this data item is identified as hot then the server makes a replica of it. SPORE [23] replicates a key when its access frequency exceeds a threshold. Similarly, Facebook's memcached cluster [25] replicates a fragment when its hosting server has less than 20% CPU and network capacity and the fragment contributes to at least 25% of its load. GAM [7] and PNUTS [10] are examples of the second category. They allow clients to issue requests to any server. The server replicates the referenced data item and serve future reads locally.

A related topic is replica selection. The objective is to issue requests to a replica that minimizes the response time. C3 [36] uses a combination of replica ranking and rate control. C3 clients gather real-time queue length and response time from each request to a replica. Its client favors issuing requests to the replica that has the lowest product of queue length and response time. Its clients also control the send rate to a replica using a cubic function. Google's search backend uses a voting scheme to select replicas [2].

2.2 Mitigating Process and Core Bottlenecks

The inter-thread synchronization may limit vertical scalability as a function of the number of cores [43, 42]. This is particularly true for in-memory data stores and cache managers. For example, a cache server's hash table, memory allocator, and data structures that implement its eviction policy require synchronization. Twemcache [40], Twitter's clone of memcached, uses a global lock to synchronize threads accessing these data structures. Redis [9] eliminates locking by using a single thread to process requests. Memcached [31] uses three types of locks: 1) an array of locks to look up the hash table, 2) a lock for each LRU queue, 3) a lock for each memory allocator of a specific data item size. These locks may result in covoys [6] where many threads wait on a lock instead of performing useful work. In general, there are three approaches to mitigate the process bottleneck [8, 14, 30].

The first approach is to design concurrent data structures that allow multiple readers and writers to access entries. MemC3 [14] uses optimistic cuckoo hashing that allows multiple readers and a single writer to access the hash table. MemC3 also uses a compact LRU-approximating eviction algorithm based on CLOCK. This approximation eliminates the lock on LRU queue. However, a more recent study [30] shows that the carefully designed data structures still struggle to scale to a large number of cores.

The second approach is to deploy multiple cache processes in a server. For example, one may deploy at least 16 Redis instances to fully utilize a 16-core server. Twitter employs this approach but also reports that it significantly increases the management cost [44]. A cluster of 100 servers and 64 cores per server requires monitoring 6400 cache processes instead of 100 cache processes. Also, when a core becomes the bottleneck, the expensive inter-process communication impedes reorganizing data between cache processes [8].

The third approach is to partition the cache space across threads. Each thread processes requests referencing its assigned data items [8, 30]. A thread has its own hash table, memory allocator, and evicts entries independently of other threads. This approach scales linearly with the number of cores when the workload has no requests referencing entries across different partitions, e.g., multi-get. Yu et al. [43] show that all concurrency control algorithms exhibit bottlenecks that hamper the scalability. The authors also suggest that it requires software and hardware co-design to scale to a thousand cores.

Deciding the number of partitions is another challenge of the second and third approaches. A recent study [38] shows that many critical factors impact the miss ratio of a partitioned cache, e.g., data item size, request rate, and data item's popularity. The study also demonstrates that partitioning the cache space may be preferable than sharing when data item sizes vary significantly.

2.2.1 Mitigating Core Bottleneck

The second and third approach may pin a thread or a process to a core to maximize performance. A core owns one or more threads or processes (tasks). We may apply techniques of Sect. 2.1 to mitigate the core bottleneck.

A reciprocal swap of two tasks may be sufficient when a core owns more than one task. To elaborate, a swap between an overloaded task in a busy core and a lightly loaded task in an underutilized core balances the load across these two cores. MBal [8] reports a maximum of 8% increase in throughput and 14% reduction in tail latency with this technique.

2.3 Mitigating Memory Bottleneck

With this bottleneck, a server exhibits a high miss rate and thrashing behavior. A server becomes memory bottlenecked for several reasons. First, the working set that maps to its assigned key space is too large relative to its memory size. Second, its eviction policy performs poorly for the workload. For example, an LRU cache is used for a workload dominated by large sequential scans. An evolving access pattern exacerbates the second challenge as one eviction policy may not be suitable for different workloads.

While memory management and automatic database tuning have been studied extensively [39], there remain many open research questions. An an example of the first challenge, consider a deployment with 100,000 servers. Is it best to have a deployment where (a) one server exhibit a low (say 30%) hit rate due to thrashing while the remaining 99,999 servers exhibit a high (say 99%) hit rate or (b) all 100,000 servers exhibit a more uniform and a high (say 90%) hit rate? Assuming Scenario b is preferred, there are several challenges that must be addressed. First, what is the proper cache size for each individual server? Second, what is the observed response time due to adjustments that increase the cache hit rate of the bottleneck server? Adaptive hashing [26] uses a load-aware consistent hashing scheme that adjusts the assignment based on cache hit rate and load. It assigns the same cache size to each server S_i and adjusts the assignment towards scenario b. It assumes a centralized load balancer that intercepts all client requests and directs them to cache servers. For each server i, the load balancer monitors its miss ratio M_i and usage ratio U_i defined as $T_i/max(T_i)$ where T_i is the throughput of a server S_i. The cost of a server S_i is $C_i = \alpha * (M_i) + (1-\alpha) * (U_i)$ where α is a configurable parameter. The objective of the load balancer is to minimize $\sum_{i=1}^{n} C_i$ with n servers. The load balancer moves a portion of keyspace from the server with the highest cost to the server with the lowest cost. In the above example of scenario a, when $\alpha = 1$, the objective of the load balancer is to minimize the overall miss ratio. The load balancer moves a portion of keyspace from the server with 30% hit ratio to a server with 99% hit ratio. Adaptive hashing is sub-optimal for several reasons. First, the centralized load balancer is the bottleneck that prevents the system to scale up to provide a higher throughput. Second, migrating cold entries may pollute the cache of other servers, resulting in a higher overall miss ratio.

There are many newly introduced cache management techniques with different complexity and claimed hit rates for different workloads, e.g., LHD [4] and CAMP [21]. An open research question is whether a technique with high complexity (CPU processing time) is worth the higher hit rate for an application? A system may evaluate an answer in an online manner as follows. For example, one may adapt principles of set dueling [34] technique that assigns a small portion of the cache space to each of two competing eviction policies to use the one that produces the lowest miss ratio. While set dueling is used in CPU caches, miniature simulation [41] models miss ratio curves and is general purpose. A miss ratio curve reports the miss ratio as a function of the cache size. A miniature simulator reports the miss ratio for one eviction policy of a particular cache size. A server downsamples data item references and feeds them into the simulator. It may launch M simulators to compute miss ratios of M cache sizes. A server that supports N eviction policies may launch $N * M$ miniature simulators. It may use the eviction policy with the lowest miss ratio at its current cache size. Miniature simulation assumes that cache entries have the same size. The modeled miss ratio curve may not be accurate for workloads with varied cache entry sizes.

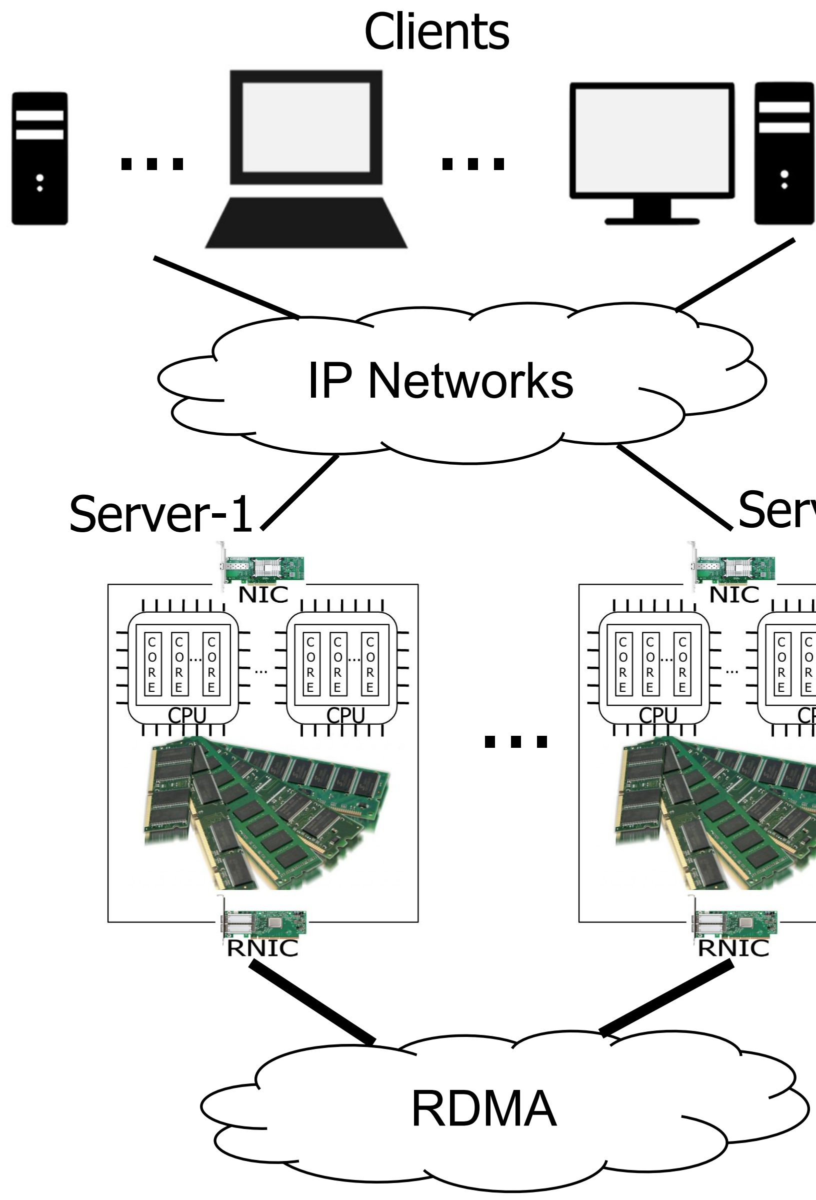


Figure 3: An architecture of a distributed key-value store.

3 Opportunities

The emerging RDMA networks motivates novel architectures to address the SkeDA challenge. A key property of these networks is their high bandwidth and low latencies [5, 24, 20]. We benchmarked an off-the-shelf Mellanox FDR CX3 single port mezz card [12] and observed it to provide a 50 Gbps bandwidth with single-digit microsecond latency [24].

Fig. 3 shows one possible architecture with the RDMA network complementing the IP network. The software system may separate client transmissions from data processing transmissions. More specifically, it may use the RDMA network to implement techniques that address the SkeDA challenge and use the IP network to deliver (a) the client requests to the servers and (b) the server produced results to clients. Its migration or replication technique may transmit terabytes of data across nodes using the RDMA network without interfering with communications to and from the clients. Such a technique may use the low RDMA latency to access important statistics published by each node, e.g., its current CPU and network utilization. This facilitates an implementation of efficient decentralized migration and replication algorithms.

Below, we focus on the architecture of Fig. 3 and a key-value store to present a simple technique named server-side redirection. This technique is designed to address the NIC bottleneck. We extend this simple technique to propose the proxy technique. Finally, we discuss extensions of these two techniques with data migration and replication as interesting short-term research directions.

With *server-side redirection*, once a server S_i's NIC utilization exceeds a certain threshold (say 80%), this technique employs NICs of other servers to transmit S_i's key-value pairs referenced by clients. It redirects client requests referencing keys owned by S_i to another peer server S_j. The directed request to S_j piggybacks the following information: the size of the cache entry, and S_i's memory address that stores the referenced cache entry. S_j uses RDMA READ to fetch the entry from S_i's memory address and uses its NIC to transmit the value to the client. Note that this simple technique does not change the placement of a key-value pair or construct replicas of it. Hence, it preserves the simple architecture of a cache manager such as memcached [31] or an in-memory key-value store such as RAMCloud [33]. It is simple to implement because it does not require a server to be aware of how data is placed across the different servers.

Fig. 4 shows an example of server-side redirection where a client issues a get request referencing Key k_i. The client first locates the primary fragment of k_i which maps the request to server S_i ①. It then issues the get request to S_i ②. Assuming NIC utilization of S_i is high (exceeds 80%), S_i performs a lookup ③ and redirects the client to issue its request to S_j ④. It provides the client with sufficient information to enable S_j to fetch k_i from S_i using the RDMA network. The client provides this information to S_j as a part of issuing its request to S_j ⑤. S_j uses RDMA READ verb to fetch the value given the memory address from S_i ⑥ and provides the value to the client ⑦.

The main advantage of the server-side redirection technique is its simplicity. However, it suffers from several limitations. First, a redirected client request performs two round-trips from the client to a server: one to S_i and a second to S_j. Second, this technique will not resolve either NIC or CPU bottlenecks with small value sizes because re-directing requests does not reduce either the network or processing load of S_i.

We extend server-side redirection to address its limitations, using the term *proxy* to refer to the new technique. As suggested by its name, this technique identifies a list of peer servers as proxies. Next, it directs clients referencing entries of S_i to the proxy servers S_j. S_j issues two RDMA READs to S_i to fetch the value. The first RDMA READ fetches the memory address that stores the referenced cache entry and its size. The second RDMA READ fetches the value from S_i's memory address and uses its NIC to transmit the value to the client.

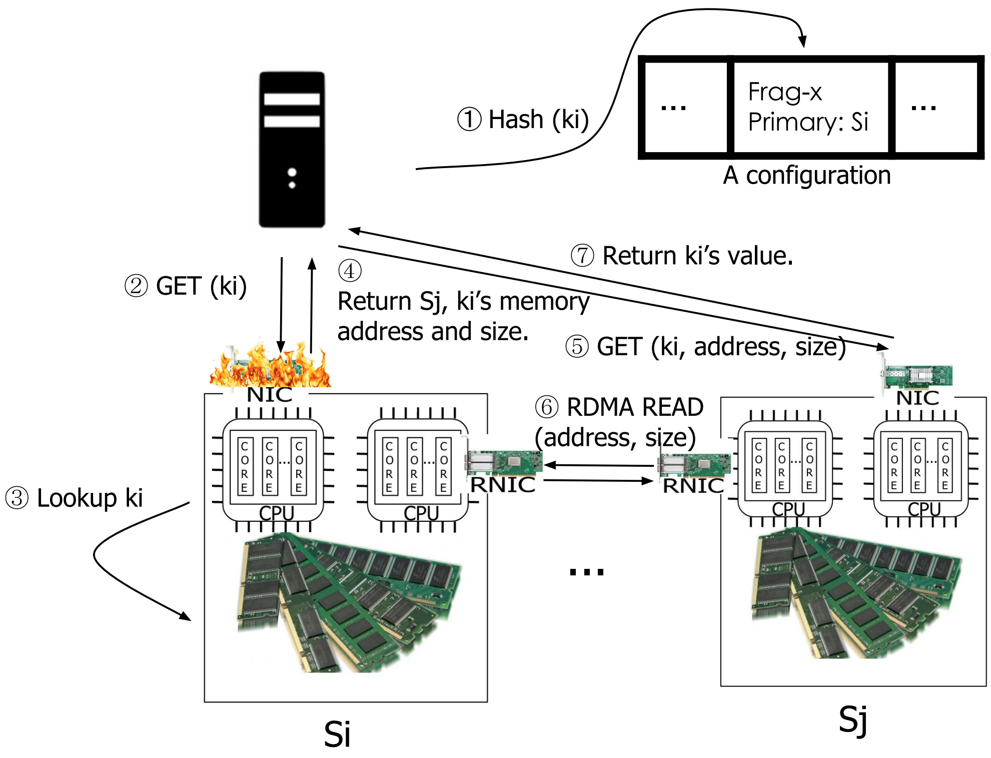


Figure 4: An example of server-side redirection.

A proxy server S_j may cache the memory address and size of the referenced cache entry so that future requests may fetch the value by issuing one RDMA READ.

Fig. 5 shows an example of the proxy technique with a client get request referencing key k_i. The client first locates the primary server of k_i in the configuration, S_i ①. The configuration identifies proxy servers that process requests for the fragment containing k_i, S_j. In this example, the client issues the get request to one of the proxy servers S_j ②. S_j has cached the memory address of k_i in S_i along with its value size. Hence, it issues an RDMA READ to fetch the value from S_i ③. Finally, S_j transmits the value to the client ④.

Both server-side redirection and proxy techniques do not change the placement of data. This may cause the RDMA network interface card (RNIC) of a bottleneck server to become fully utilized, dictating the overall processing capabilityof the system. However, Proxy eliminates an extra round-trip from client to server. On the other hand, it may potentially require additional RNIC round-trips between servers and propagation of proxy configuration to clients.

One may extend the above two technique with data migration and replication [28, 13, 18, 17, 32, 26, 45] to further enhance their performance. With migration, a bottleneck server S_i uses RDMA to migrate its popular entries to other servers before it performs server-side redirection or the proxy technique. This extension must address the following challenges. First, what is the granularity of data migration? It is a single key-value pair or a fragment consisting of many key-value pairs (that are potentially referenced together)? Second, how to efficiently support migration at the granularity of a single key-value pair? Studies such as [37] highlight the overhead of monitoring and performing such migrations may potentially

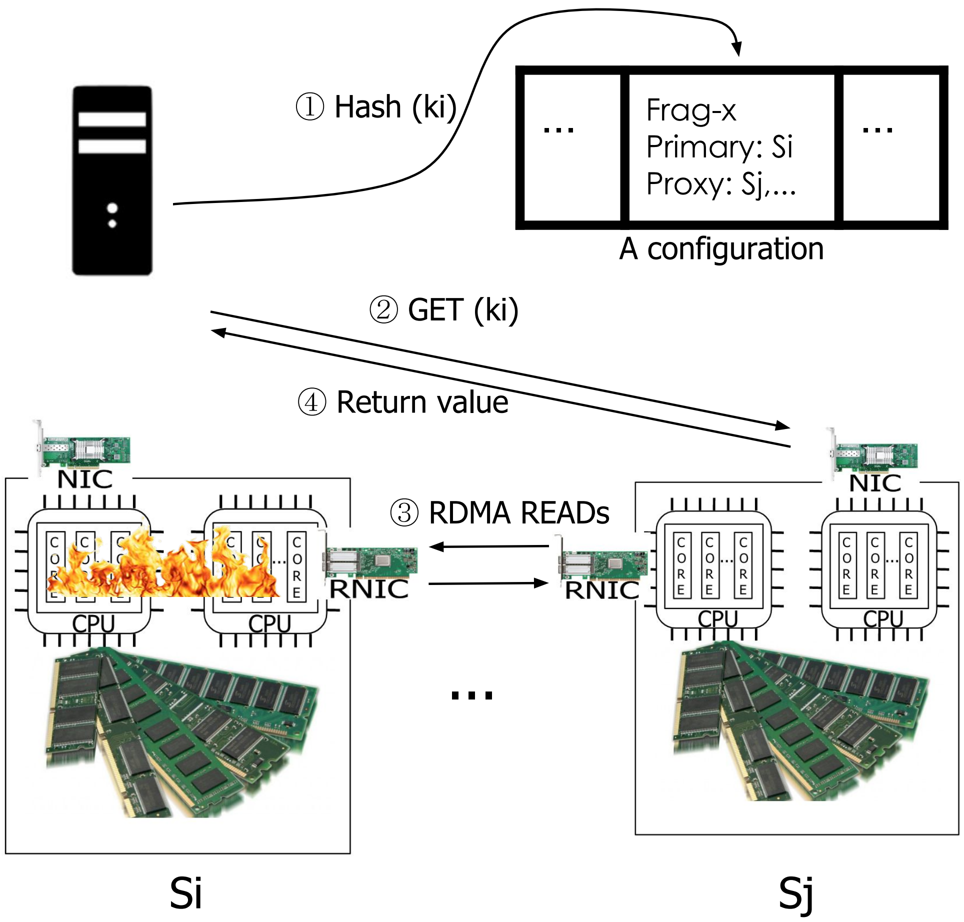


Figure 5: An example of the proxy technique.

outweighing its benefits. Third, how to ensure consistency during the migration? This is challenging since the bottleneck server must be aware of remote RDMA READs issued from the idle servers. The low latency of the RDMA may be particularly beneficial in this context. Fourth, what happens when data migration causes an idle server S_j to become a bottleneck? Should S_j reject this migration or should it continue to apply server-side redirection or proxy for the migrated entry? Finally, once a migrated entry becomes cold, should it be returned to its original owner?

With replication, a bottleneck server S_i uses RDMA to replicate its popular data items to other servers, e.g., S_j. In addition to S_i, other servers with a replica of the popular data items may serve requests referencing these data items. Replication raises its own challenges. First, how to ensure consistency across replicas? If S_i uses the one-sided RDMA WRITE verb to update the replicas, how does S_j handle race conditions between a local read with a remote RDMA WRITE when S_j is not aware of this write? Second, what is the best approach to add and remove replicas?

With both migration and replication, a key question is how do clients discover the new owners of a data item so that they can issue requests to them directly? As the bottleneck server S_i may change the ownership of a data item, how do clients discover the latest owners?

From a theoretical perspective, our use of RDMA raises the following obvious question.

How would the throughput of the new techniques with RDMA compare with a system configuration that simply increases the traditional network bandwidth with the bandwidth provided by the RDMA? For example, if the IP bandwidth is 40 Gbps and the RDMA bandwidth is 50 Gbps then how does the configuration of Fig. 3 compare with one that consists of only the IP network with a bandwidth of 90 Gbps? Obviously, the latency of the traditional network is higher than the RDMA network. However, a higher latency may not result in a lower throughput as throughput is dictated by the available bandwidth. An evaluation of the alternative networks and their comparison with use of the RDMA is a low hanging fruit.

4 Evaluation

We consider two bottlenecks: CPU and NIC. With value size of 329 bytes (Facebook's average value size [3]), the CPU of one server becomes the bottleneck. With larger value sizes, the outbound NIC of one server becomes the bottleneck. The database contains 10 million records range partitioned across N servers, $N = 8$. Similar to the discussions of Fig. 1(a), a client references a data item based on the Zipfian distribution. This causes the first server S_1 to become the bottleneck. The workload is read only. We study the different techniques with a heavy system load. If M is the number of concurrent client requests that saturates a single server, the number of concurrent requests with N servers is $M \times N$. In our experiments, the value of M is smaller for larger value sizes. With value size of 329 and 1024 bytes, M is 32 and 18, respectively.

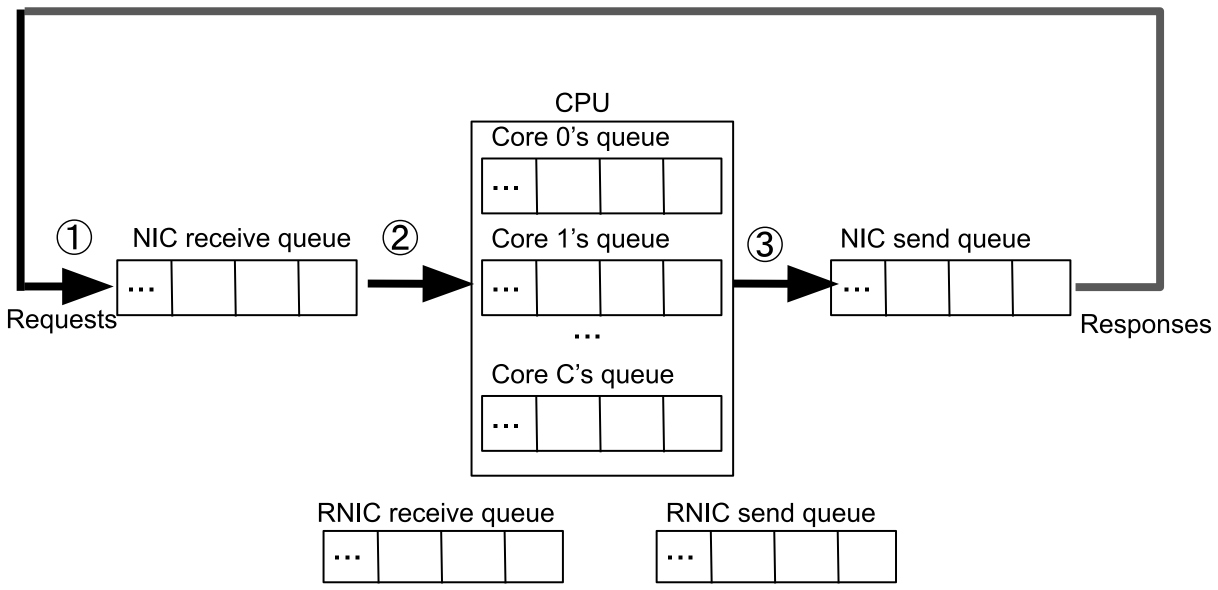


Figure 6: Modeled queues of a server.

We compare the response time and throughput of the baseline system by itself, and extended with each of the server-side re-direction and proxy techniques. The main *lessons* of this evaluation are:

1. Server-side redirection does not resolve CPU bottleneck.

2. Server-side redirection provides a higher throughput than the baseline when the NIC is the bottleneck.

3. Proxy provides a higher throughput and faster response times than both the baseline and server-side redirection.

4. The benefits of both server-side redirection and proxy level off when the RNIC of the bottleneck server becomes fully utilized.

Details of our experimental setup are as follows. We use a simulator that models each server resource as a queue, see Fig. 6. A server has one queue for each of its 32 cores and four queues for its NIC and RNIC to send and receive messages. The NIC bandwidth β_{NIC} is 10 Gbps. The RNIC bandwidth β_{RNIC} is 56 Gbps. A server processes a local read request as follows. It first adds the request to its NIC receive queue ①. Once it completes, it adds the request to one of the CPU queues ②. Lastly, the request is added to the NIC send queue ③.

The service time of a request in a queue depends on its size. We use Facebook [3] reported average key size of 36 bytes and vary the value size from 329 bytes to 16 KB. A request takes $\frac{key\ size}{\beta_{NIC}}$ and $\frac{value\ size}{\beta_{NIC}}$ to complete in a NIC receive queue and send queue. A core takes 12 μs, 14 μs, 48 μs, 108 μs, and 200 μs to process a request fetching a value of 329 bytes, 1 KB, 4 KB, 8 KB, and 16 KB, respectively.

A server S_i issues a remote RDMA READ request to S_1 by first adding the request to its RNIC's send queue with service time $\frac{8}{\beta_{RNIC}}$. Then, the request is added to S_j's RNIC receive queue with the same service time. S_j adds the request to its RNIC's send queue with service time $\frac{value\ size}{\beta_{RNIC}}$. Lastly, the request is added to the RNIC receive queue of S_i.

Our simulator implements the server-side and proxy as follows. With server-side, when the NIC is the bottleneck, the bottleneck server S_1 redirects a fraction of requests equal to the percentage of its idle CPU to the other servers. Each server has the same probability of being referenced. For example, with value size of 1 KB, S_1 provides a response directly for 53% of its requests and redirects the remaining 47% of its requests to the other 7 servers. It redirects requests to each server with the same probability, 6.7%. The redirect response size is 16 bytes. If the CPU is the bottleneck, this technique does not perform redirection because it is not beneficial. In this case, redirecting requests increases response time without resolving the bottleneck.

With proxy, when S_1's NIC and/or CPU is the bottleneck, each client redirects $\frac{N-1}{N}$ of its requests destined for S_1 to other servers where N is the number of servers. In these experiments, each client redirects 12.5% of its S_1 requests to each of the other servers. A proxy server uses one RDMA READ to fetch an index entry of 8 bytes followed by a second READ to fetch the value.

Table 1 shows the relative throughput of the baseline, server-side, and proxy relative to the maximum theoretical throughput supported by the eight node system for different value sizes. The theoretical maximum is defined as the maximum throughput of one node multiplied by eight. The throughput of the baseline is 15% of the maximum since S_1 limits the overall throughput. Server-side redirection enhances throughput with larger value sizes

Table 1: Throughput relative to the theoretical maximum.

Value size (bytes)	Baseline	Server-side redirection	Proxy
329	15.56	15.56	89.39
1024	15.29	28.14	85.59
4096	15.21	32.82	85.65
8192	15.18	29.26	86.71
16384	15.44	32.17	85.55

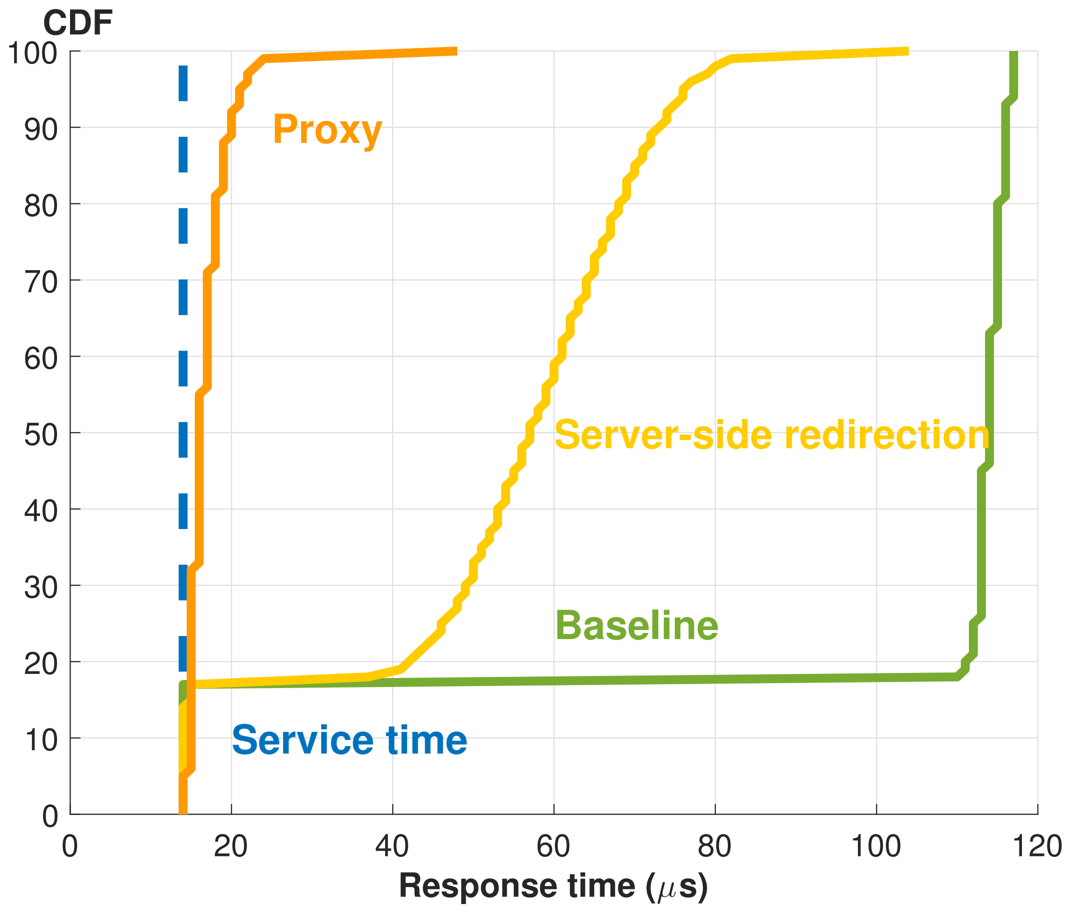


Figure 7: Response time of alternative techniques with 1 KB value size.

(starting with 1 KB) because the NIC is the bottleneck[2]. The proxy provides the most benefit, harnessing more than 80% of the theoretical maximum.

Fig. 7 shows the cumulative percentage of requests that observe a response time with 1 KB value size. The baseline has the slowest response time due to queuing delays. To elaborate, with the 1 KB value size, the 18th percentile of the response time is 110 μs. This is 8 times the service time (14 μs). Proxy is the best technique to approximate the

[2]It would provide no enhancement if the CPU was the bottleneck.

service time. Server-side redirection provides a benefit as long as the NIC is the bottleneck. Its response time includes two roundtrips from a client to a server to process a re-directed request. In our experiments with 329 bytes value sizes, the CPU is the bottleneck and server-side redirection has response time comparable to the baseline.

The improvements with the proxy technique are similar for CPU and NIC bottleneck. Proxy's throughput is 15% lower than the maximum because of random collision of requests on the same server which results in temporary queueing delays.

With both server-side and proxy, the performance benefits level off when the RNIC of the bottleneck server is saturated. One may extend both techniques with migration and replication described in Section 2.1. The low latency of RDMA and its ability to bypass the bottleneck server's CPU fastens migration and replication.

5 Conclusion

The paper identifies six forms of bottlenecks due to a skewed pattern of access to data. We surveyed existing solutions that mitigate each of these bottlenecks. A low hanging fruit is use of modern hardware such as RDMA to efficiently implement novel load balancing techniques that address different forms of bottlenecks.

Acknowledgment

We gratefully acknowledge use of CloudLab network testbed [12] for all experimental results presented in this paper. We also thank the anonymous referees of the 8th Workshop on Scalable Cloud Data Management co-colocated with IEEE BigData for their valuable comments.

References

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, 2016. USENIX Association.

[2] A. Archer, K. Aydin, M. H. Bateni, V. Mirrokni, A. Schild, R. Yang, and R. Zhuang. Cache-aware Load Balancing of Data Center Applications. *Proc. VLDB Endow.*, 12(6):709–723, Feb. 2019.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[4] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, 2018. USENIX Association.

[5] 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.

[6] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, Apr. 1979.

[7] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[8] Y. Cheng, A. Gupta, and A. R. Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 4:1–4:16, New York, NY, USA, 2015. ACM.

[9] R. contributors. Redis, 2018.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[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] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[13] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, New York, NY, USA, 2015. ACM.

[14] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, 2013. USENIX.

[15] 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.

[16] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[17] S. Ghandeharizadeh, M. Almaymoni, and H. Huang. Rejig: A Scalable Online Algorithm for Cache Server Configuration Changes. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, TLDKS '19, 2019. To Appear.

[18] S. Ghandeharizadeh and H. Huang. Gemini: A Distributed Crash Recovery Protocol for Persistent Caches. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, pages 134–145, New York, NY, USA, 2018. ACM.

[19] S. Ghandeharizadeh and H. Huang. Hoagie: A Database and Workload Generator using Published Specifications. In *2nd IEEE International Workshop on Benchmarking, Performance Tuning and Optimization for Big Data Applications, co-located with IEEE BigData*, pages 3847–3852, Dec 2018.

[20] S. Ghandeharizadeh, H. Huang, and H. Nguyen. Nova: Diffused Database Processing using Clouds of Components [Vision Paper]. In *15th IEEE International Conference on Beyond Database Architectures and Structures (BDAS)*, Ustron, Poland, 2019.

[21] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: A Cost Adaptive Multi-queue Eviction Policy for Key-value Stores. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 289–300, New York, NY, USA, 2014. ACM.

[22] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. *SIGMOD Rec.*, 25(2):173–182, June 1996.

[23] 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.

[24] H. Huang and S. Ghandeharizadeh. An Evaluation of RDMA-based Message Passing Protocols. In *6th Workshop on Performance Engineering with Advances in Software and Hardware for Big Data Science, co-located with IEEE BigData*, Dec 2019. To Appear.

[25] 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.

[26] J. Hwang and T. Wood. Adaptive Performance-Aware Distributed Memory Caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, 2013. USENIX.

[27] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[28] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 390–405, New York, NY, USA, 2017. ACM.

[29] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[30] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[31] memcached contributors. memcached, 2018.

[32] 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 *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[33] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[34] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[35] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, 2016. USENIX Association.

[36] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association.

[37] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[38] J. Tan, G. Quan, K. Ji, and N. Shroff. On Resource Pooling and Separation for LRU Caching. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(1):5:1–5:31, Apr. 2018.

[39] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proc. VLDB Endow.*, 12(10):1221–1234, June 2019.

[40] Twitter. Twemcache: Twitter memcached, 2018.

[41] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[42] Y. Wu and K.-L. Tan. Scalable In-memory Transaction Processing with HTM. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 365–377, Berkeley, CA, USA, 2016. USENIX Association.

[43] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[44] Y. Yue. How Twitter Uses Redis To Scale - 105TB RAM, 39MM QPS, 10,000+ Instances, 2014.

[45] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving Cash by Using Less Cache. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, 2012. USENIX.