Mitigating Load Imbalance in Distributed Data Serving with Rack-Scale Memory Pooling

STANKO NOVAKOVIC, VMware
ALEXANDROS DAGLIS, Georgia Institute of Technology
DMITRII USTIUGOV, EDOUARD BUGNION, and BABAK FALSAFI, EPFL
BORIS GROT, University of Edinburgh

6

To provide low-latency and high-throughput guarantees, most large key-value stores keep the data in the memory of many servers. Despite the natural parallelism across lookups, the load imbalance, introduced by heavy skew in the popularity distribution of keys, limits performance. To avoid violating tail latency service-level objectives, systems tend to keep server utilization low and organize the data in micro-shards, which provides units of migration and replication for the purpose of load balancing. These techniques reduce the skew but incur additional monitoring, data replication, and consistency maintenance overheads.

In this work, we introduce RackOut, a memory pooling technique that leverages the one-sided remote read primitive of emerging rack-scale systems to mitigate load imbalance while respecting service-level objectives. In RackOut, the data are aggregated at rack-scale granularity, with all of the participating servers in the rack jointly servicing all of the rack's micro-shards. We develop a queuing model to evaluate the impact of RackOut at the datacenter scale. In addition, we implement a RackOut proof-of-concept key-value store, evaluate it on two experimental platforms based on RDMA and Scale-Out NUMA, and use these results to validate the model. We devise two distinct approaches to load balancing within a RackOut unit, one based on random selection of nodes—`RackOut_static`—and another one based on an adaptive load balancing mechanism—`RackOut_adaptive`. Our results show that `RackOut_static` increases throughput by up to 6× for RDMA and 8.6× for Scale-Out NUMA compared to a scale-out deployment, while respecting tight tail latency service-level objectives. `RackOut_adaptive` improves the throughput by 30% for workloads with 20% of writes over `RackOut_static`.

CCS Concepts: • **Computer systems organization** → *Distributed architectures*;

Additional Key Words and Phrases: Rack-scale systems, key-value stores, RDMA, one-sided operations, load imbalance, latency-critical applications, CREW, SLO, tail latency

This work has been partially funded by the Nano-Tera *YINS* project, the CHIST-ERA *DIVIDEND* project, and the *Scale-Out NUMA* project of the Microsoft-EPFL Joint Research Center.

Authors' addresses: S. Novakovic, Microsoft Research, One Microsoft Way 99/2713, Redmond, WA, 98052, USA; email: stnovako@microsoft.com; A. Daglis, Georgia Institute of Technology, 266 Ferst Dr. NW, Atlanta, GA, 30332, USA; email: alexandros.daglis@cc.gatech.edu; D. Ustiugov, EPFL IC, INJ 236 (Bâtiment INJ), Station 14, CH-1015 Lausanne, Switzerland; email: dmitrii.ustiugov@epfl.ch; E. Bugnion, EPFL IC, INN 237 (Bâtiment INN), Station 14, CH-1015 Lausanne, Switzerland; email: edouard.bugnion@epfl.ch; B. Falsafi, EPFL IC, INJ 233 (Bâtiment INJ), Station 14, CH-1015 Lausanne, Switzerland; email: babak.falsafi@epfl.ch; B. Grot, University of Edinburgh, 10 Crichton Street, EH8 9YL, Edinburgh, United Kingdom; email: boris.grot@ed.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
0734-2071/2019/04-ART6 $15.00
https://doi.org/10.1145/3309986

ACM Reference format:
Stanko Novakovic, Alexandros Daglis, Dmitrii Ustiugov, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2019. Mitigating Load Imbalance in Distributed Data Serving with Rack-Scale Memory Pooling. *ACM Trans. Comput. Syst.* 36, 2, Article 6 (April 2019), 37 pages.
https://doi.org/10.1145/3309986

1 INTRODUCTION

Datacenter services and cloud applications such as search, social networking, and e-commerce are redefining the requirements for system software. A single application can consist of hundreds of software components, deployed on thousands of servers organized in multiple tiers. It must support high connection counts and operate with tight user-facing service-level objectives (SLO), often defined in terms of tail latency [7, 20, 55]. To meet these objectives, most such applications keep the data (e.g., a social graph) in memory-resident distributed non-relational databases, such as Key-Value Stores (KVS). Distributed KVS use consistent hashing to shard and distribute the data among the servers of the leaf tier.

The use of sharding has inherent scalability benefits: Applications perform lookups in parallel by leveraging a hash function to locate the requested data in the leaf tier quickly. In its basic form, however, sharding data limit the maximum throughput respecting the SLO whenever the popularity distribution of data items is skewed and unknown. A high-popularity skew among data items can result in severe load imbalance, as a small subset of the leaf servers will saturate—either the CPU or the network interface controller (NIC)—thereby violating the tail latency SLO, while the majority of the other leaf servers remains mostly idle.

In this work, we introduce RackOut, a memory pooling technique that leverages the one-sided remote read primitive of emerging rack-scale systems to mitigate the skew-induced load imbalance and achieve high throughput at low latency. A RackOut unit is a group of servers (i.e., a rack) augmented with an internal secondary fabric that allows any node within the rack to access the memory of other nodes through one-sided operations (i.e., without involving the remote CPU). The load associated with *network protocol processing* is imbalanced across a rack's servers, because of a small set of highly popular data items (Section 4.4). RackOut leverages one-sided reads to evenly redistribute that load across all the CPUs of the rack (Section 5). Consequently, the data in RackOut are partitioned at rack-scale granularity, with the entire rack responsible for a collection of data items mapped to the rack's servers (*super-shard*). RackOut leverages the concurrent-read/exclusive-write (CREW) data access model, which has previously been shown to dramatically improve the scalability of single-server performance on multicore servers [22, 45, 53]. By decoupling data access from data storage, RackOut substantially reduces the negative impact of skew on the entire KVS' performance.

To implement the CREW model in RackOut, the client identifies the server holding the target micro-shard and therefore the RackOut unit it belongs to. Read requests are load balanced among all the servers within the target RackOut unit, while write requests are always directed to the server holding the master copy of the micro-shard. A few replicas of each micro-shard are required for availability reasons, and they must reside on different RackOut units. We distinguish between `RackOut_static`, where the clients schedule read request uniformly across the servers of a RackOut unit, and `RackOut_adaptive`, where the clients adaptively schedule read requests based on the write distribution and average service times. Even though data are never replicated within a RackOut unit, RackOut is compatible and synergistic with dynamic replication [10, 20, 37, 38]. Since the RackOut technique already balances requests within a RackOut unit (using one-sided operations rather than replication), replication is only required across RackOut units, which

reduces the overheads associated with load monitoring, replication itself, and ongoing consistency maintenance. This article makes the following contributions:

- We provide a detailed analysis of the impact of data popularity skew on the load imbalance across the servers of a datacenter deployment. Technology trends toward extreme scale-out server deployments [26] will further exacerbate this problem, increasing the pressure on existing skew mitigation techniques, such as dynamic replication.
- We provide a detailed analysis of the benefits of rack-level aggregation in mitigating the effect of skew. We develop a queuing model for `RackOut_static` that assumes global access to memory within the rack and uniform scheduling of read requests within individual RackOut units. We evaluate the benefits of `RackOut_static` as a function of server count, size of the RackOut unit, and read/write ratio for datasets that follow a power-law distribution. For a Zipfian, read-only, distribution with $\alpha = 0.99$, the model predicts that a deployment of 512 servers grouped into 16-server RackOut units with `RackOut_static` scheduling increases throughput by 6× for RDMA and 8.6× for soNUMA without violating SLO.
- We evaluate the combination of `RackOut_static` with an idealized dynamic replication scheme. Our results show that RackOut is synergistic with dynamic replication and dramatically reduces the number of replicas required for load balancing.
- We propose `RackOut_adaptive`, an adaptive scheduling mechanism for RackOut that improves the throughput of read-write workloads (≥5% writes). In `RackOut_adaptive`, the clients take scheduling decisions for reads based on the current write distribution and average service times within the target RackOut unit. A coordinator client periodically retrieves the number of processed writes and the average service time from each server of its associated RackOut unit and uses that information to compute the read distribution, with the goal of achieving uniform utilization across the RackOut unit's servers. The coordinator clients propagate the read distributions to all the other clients.
- We implement RackOut KVS (RO-KVS), a proof-of-concept KVS using a conventional network for client access and an RDMA fabric for memory access. RO-KVS is based on FaRM [22] and is ported to both Mellanox RDMA [52] and Scale-Out NUMA [17, 56]. We evaluate RO-KVS using `RackOut_static` scheduling in terms of its 99th percentile tail latency for the hottest rack of a 512-server deployment. We show that organizing this rack as a 16-server RackOut unit using soNUMA can deliver 6× more requests than 16 independent servers for a workload with 5% writes and 8.2× more requests for a read-only workload. Discrepancies between the model and the measured system remain below 6%, which validates the model.
- Finally, we evaluate `RackOut_adaptive` using the implementation of the proposed adaptive load balancing mechanism. `RackOut_adaptive` improves the throughput of RO-KVS for read-write workloads; for example, for workloads with 20% of writes, RO-KVS achieves 30% higher throughput than `RackOut_static`.

The rest of the article is organized as follows: Section 2 motivates the problem in terms of application trends and dataset access patterns. Section 3 discusses the key architectural trends that provide cost-effective, rack-scale memory pooling. Section 4 provides a detailed analysis of the RackOut queuing model with `RackOut_static` scheduling. Section 5 introduces an adaptive load balancing extension - `RackOut_adaptive`, which boost performance for write-intensive workloads. In Section 6, we use an implementation of our proof-of-concept RO-KVS to validate the queuing model for `RackOut_static` and illustrate the benefits of `RackOut_adaptive`. We discuss related work in Section 8 and conclude in Section 9.

2 BACKGROUND

A significant portion of important web-scale applications is latency sensitive [7, 20, 55]. Designing a datacenter-scale system that delivers tight latency guarantees for the majority of user requests is notoriously challenging [20, 21]. The flexibility and scalability of using KVS implemented as an in-memory, distributed hash table has led to its broad use as a state-of-the-art approach for low-latency data serving applications. In this section, we explain the sensitivity of the traditional scale-out approach to the skewed popularity distribution that naturally exists in large data collections and argue that existing approaches to alleviate load imbalance through data replication are not adequate.

2.1 In-memory Key-Value Stores

An in-memory KVS is a critical component of many modern cloud systems. Several large-scale services are powered by well-engineered KVS, which are designed to scale to thousands of servers and petabytes of data and serve billions of requests per second [7, 10, 21, 55]. KVS such as Memcached [31], Redis [65], Dynamo [21], TAO [10], and Voldemort [48] are used in production environments of large service providers such as Facebook, Amazon, Twitter, Zynga, and LinkedIn [3, 47, 55, 71]. The popularity of these systems has resulted in considerable efforts, including open source implementations [25], research efforts [4, 60], and a wide range of sophisticated, highly tuned frameworks that aspire to become the state-of-the-art of KVS [22, 44, 45].

Service architects set strict service-level objectives (SLO) to ensure high quality, designing the service deployment to respond to user requests within a short and bounded delay. For high fan-out applications, designing for the average latency is not enough; a good service guarantees that the vast majority of the requests will meet the SLO and thus targets a 99th or even 99.9th percentile latency of just a few milliseconds [20, 21].

2.2 Skew in Scale-out Architectures

KVS typically handle very large collections of data items and millions or billions of requests per second. The scale of a KVS deployment is driven by memory capacity (i.e., all servers have their memory fully populated). Despite the uniform distribution of keys to servers, skewed access distributions emerge naturally, as the popularity of the data items varies greatly. Previous work has shown that popularity distributions in real-world KVS workloads follow a power-law distribution [7, 10, 28, 38], resulting in an access frequency imbalance, commonly referred to as *skew*. This skewed distribution is accurately represented by the power-law Zipfian distribution [7, 14, 16, 28, 45, 66]. Based on Zipf's law, and given a collection of popularity-ranked data items, the popularity y of a data item is inversely proportional to its rank r: $y \sim r^{-\alpha}$, with α close to unity. A classic example of such skewed popularity distribution is a social network, where a very small subset of users is extremely popular as compared to the average user. These two distinct user categories (popular versus the rest) result in a popularity distribution with a skyrocketing peak and a long tail.

The exponent α is determined by the dataset that it models; $\alpha = 0.99$ is the typical data popularity distribution used in KVS research [15, 22, 37, 44, 45]. Some studies show that the popularity distribution skew in real-world datasets can be lower than that (e.g., $\alpha = 0.6$ [66], $\alpha = 0.7$–0.9 [5, 63]) but also even higher (e.g., up to $\alpha = 1.01$ [14, 28]).

Sharding the data across the deployment's collection of servers is typically done by grouping data items into so-called *micro-shards*, each server being responsible for hosting and serving hundreds or thousands of them from its local memory [20, 30]. This data distribution is done by applying a hash function on the key of the data items, which maps each of them to a micro-shard (e.g., References [25, 51, 67]) and each micro-shard to a server. Hash functions are aimed at probabilistically reducing load imbalance by evenly distributing data items to servers but are distribution

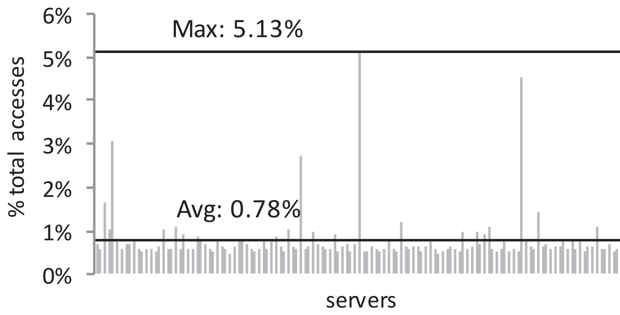


Fig. 1. Grouping of 250M data items with a power-law popularity distribution in 128 servers.

agnostic, as data item popularities cannot be predicted in advance or controlled and may change over time. In practice, a collection of micro-shards that maps to a single server represents a single data *shard* that is served by its corresponding server. Thus, even after grouping data items together into shards, the presence of the inherent popularity skew is still observable as popularity skew across shards [57].

Figure 1 illustrates the access distribution of a dataset of 250 million items of randomly generated keys, distributed across 128 servers, and accessed with a power-law (Zipfian) popularity distribution of exponent $\alpha = 0.99$. In such a distribution, the most popular item is accessed 11 million times more than the average item. After sharding the dataset across 128 servers through a hash function, the hottest server holds a shard with a set of keys that is 6.5× more popular than the average shard. This has significant implications as this hottest server will receive a 6.5× higher load than the average and may become overloaded while the majority of the servers are largely idle. In the absence of any dynamic replication or migration scheme, the number of micro-shards per server does not impact the skew.

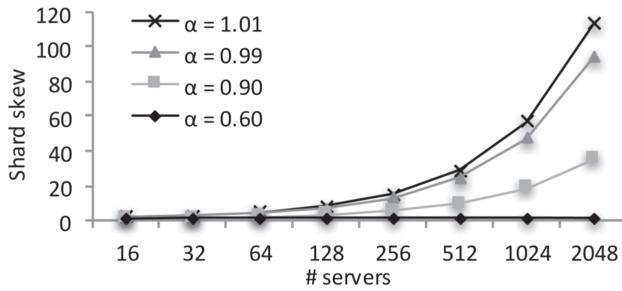
We define the *shard skew* as the access ratio between the hottest and the average server. Shard skew arises in a datacenter as a function of three parameters: (i) the exponent α of the dataset's power-law distribution, (ii) the number of items comprising the dataset, and (iii) the function used to distribute data items to micro-shards. We discuss the impact of each parameter below.

Figure 2(a) shows the shard skew as the number of servers scales for different popularity distribution exponents α. While the shard skew is insensitive to scaling out for low α values (e.g., $\alpha = 0.6$), large exponents dramatically increase shard skew, which in turn becomes a performance limiter. For example, doubling the number of servers from 1024 to 2048 with $\alpha = 0.99$ leads to shard skew increasing near-linearly by 1.97×, meaning that the resulting improvement in expected performance would be only 1.01×.

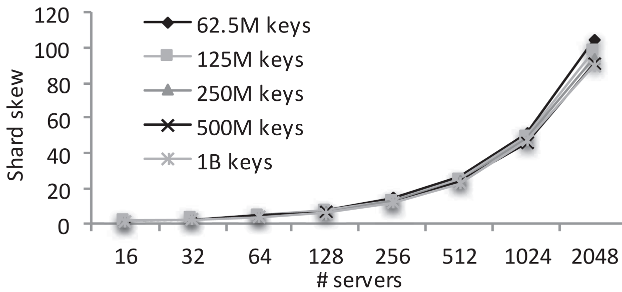
Figure 2(b) shows the impact of the size of the data item population on the shard skew. While larger datasets result in better load distribution to shards and hence lower shard skew, the variation of shard skew with the dataset size is minimal. Finally, given a set of keys, the hash function used to distribute the data items to shards affects the absolute value of the resulting shard skew. However, since hash functions are static, stateless, and distribution agnostic, they cannot predict or dynamically handle the shard skew that is inevitably derived from a heavily skewed data item popularity distribution.

2.3 Replication: A Thorny Solution

Service providers are well aware of the problems that arise from skew-induced load imbalance [19, 20, 38]: Servers that hold the most popular micro-shards can quickly become overwhelmed with user requests, degrading the whole system's performance and service quality. Data replication is



(a) Impact of the Zipfian exponent (250M keys).



(b) Impact of the dataset size (Zipfian $\alpha = 0.99$).

Fig. 2. Shard-skew sensitivity for a Zipfian distribution as a function of its exponent, server count, and dataset size.

a technique that is widely used to better deal with such load imbalance in the datacenter. It is based on the simple concept that by replicating each data N times, N servers rather than one can operate on the same piece of data, thus providing higher flexibility and robustness against skew. In its simplest form, replication is static; the whole dataset is replicated a fixed number of times. However, this approach comes at a great cost, as it multiplies the memory capacity requirements, which is the most critical resource of modern datacenters [46, 60]. Furthermore, static replication only provisions for a predefined amount of skew; any skew higher than the replication factor was provisioned for results in the service's quality degradation.

Recent research has focused on optimizations beyond static replication. In particular, dynamic replication can flexibly calibrate memory overprovisioning and sudden changes in skew (i.e., *thundering herds* [55]). The main principle is that the system dynamically monitors the load on an ensemble of micro-shards and takes replication decisions on the fly to mitigate the load imbalance. Dynamic replication could operate on individual keys or on entire micro-shards [37, 38]. The latter is often preferred as it simplifies the updates of the KVS metadata and minimizes routing overheads. The effectiveness of such methods is highly dependent on the fine-tuning of several parameters, such as the timeliness, cost, and accuracy of load imbalance detection, the speed of replication, tracking the changing number of replicas and timely deallocation when the replicas are no longer useful, and so on [10, 37, 38, 40].

Dynamic replication is never free: (i) Load monitoring to detect bursts in micro-shard popularity, the actual replication of micro-shards, and the updates to the KVS metadata all add CPU, memory, and network overheads; (ii) a replicated micro-shard is more expensive to maintain as consistency requirements introduce both correctness and performance concerns [9, 11, 21, 39, 43, 58, 70]. Even in a weakly consistent KVS [21], each replica must eventually be updated, which reduces the system's effective throughput.

2.4 Summary

Distributed KVS shard the data using a hash function and hence are subject to skewed access patterns. The popularity skew that appears in most real-world cloud applications directly translates to load imbalance that manifests itself in poor datacenter utilization. To make better use of resources and achieve higher throughput without violating the agreed-upon SLO, dynamic replication techniques have emerged [10, 20, 37, 38]. However, dynamic replication comes with considerable overheads: The consistency semantics expected by the application, the dataset's change rate [21], the precision of the monitoring algorithm, and the micro-shard size all impact the system's behavior [28, 37, 38].

3 RACK-SCALE MEMORY POOLING

An intuitive approach to mitigating the effects of shard skew while avoiding the challenges and overheads of dynamic replication is to reduce the number of nodes involved by increasing each node's size to have fewer larger shards. As Figure 2 illustrates, a reduction in the node count can dramatically reduce the shard skew. Even though the overall architecture remains that of a KVS consisting of multiple independent building blocks, each building block is designed to scale in terms of throughput and memory capacity. The design space for such solutions is broad but can be broken down into architectural considerations (Section 3.1), concurrency (Section 3.2), and fault tolerance (Section 3.3).

3.1 Architectural Building Blocks

Since the CPU or the NIC is the performance bottleneck at high load, growing the node size mandates increasing the per-node processing and networking capacity. Addressing this challenge involves either building bigger, more capable server nodes or aggregating multiple existing server nodes into larger logical entities.

The first approach simply leverages the technologies enabled in large-scale cache-coherent NUMA servers (e.g., based on Intel's QuickPath Interconnect or AMD's HyperTransport technology). Such machines provide the convenient shared memory abstraction and a low-latency high-bandwidth inter-node network. The downside of such large-scale machines is that their cost grows exponentially with the number of CPUs due to the complexity of scaling up the coherence protocol, increased system design and manufacturing cost, as well as a focus on low-volume, high-value applications such as online transaction processing for a market that is less price-sensitive.

The second approach leverages conventional datacenter-grade servers or individually less capable server building blocks [4, 13, 24, 26, 50, 68] augmented with a rack-level RDMA fabric. This approach is used in commercial solutions providing analytical (e.g., Oracle ExaData / Exalogic [59]) or storage (e.g. EMC/Isilon [23]) solutions to clients connected via a conventional network. AppliedMicro's X-Gene2 server SoC [49] and Oracle's Sonoma [34] integrate the RDMA controller directly on chip, HP Moonshot [36] combines low-power processors with RDMA NICs, and research proposals further argue for on-chip support for one-sided remote access primitives [17, 56]. The benefit of such rack-scale memory pooling approaches is that building larger logical entities comes at a lower cost and complexity as compared to the cache-coherent NUMA (ccNUMA) approach.

The fundamental premise for rack-scale memory pooling is that all servers within a rack can access all memory in the rack within a small premium over local memory, and thus the rack's aggregate memory can be perceived as a single, partitioned global address space. We further assume that remote memory can be accessed through *one-sided operations* and that the fabric efficiently supports the access of data items residing in remote memory. Such remote access capability is readily available in commercial fabrics such as InfiniBand or RoCE [52].

3.2 Concurrency Model

In a traditional scale-out deployment, each server manages a collection of micro-shards, stored in its own memory. Despite the simplicity of the design, such deployments offer a wide range of concurrency models that can independently provide concurrent or exclusive access to either read or write objects. The concurrent-read/exclusive-write data access model (CREW) has proven to provide solid scalability at low complexity. In CREW, the memory is managed as a single read-only pool, with changes being handled by a specific thread based on the location of the object in memory. Recent work has demonstrated the scalability benefits of the CREW model on Xeon-class servers [44, 45]. As most workloads are read dominated [7, 10, 15, 62], CREW offers a sweet spot in terms of scalable performance by keeping synchronization requirements to a minimum.

The suitability of the CREW model has also been demonstrated on rack-scale systems using RDMA. FaRM [22] and Pilaf [53] follow a CREW model where each server is responsible for the modification of objects stored in its memory, but other servers can directly read them using one-sided RDMA read operations.

3.3 Availability and Durability

Memory pooling is not a substitute for replication when it comes to availability and durability of the data. Indeed, scale-out applications are fundamentally designed to handle node failures [21, 60, 67]. In such cases, the central system relies on replication across nodes to ensure the availability of the data and removes the faulty node from the KVS. While this may seem problematic when an individual node consists of an entire rack, we note that datacenter deployments already assume that physical racks have single points of failure in the infrastructure (e.g., in terms of power distribution and top-of-rack networking [12]). Future rack-scale systems will likely consist of more tightly coupled nodes, introducing additional points of failure. Thus, fault tolerance must be handled across rack-scale building blocks.

Failures must also be handled within the rack-scale building block to detect and report partial failures of the system (e.g., the failure of individual nodes that would make portions of the dataset inaccessible). If a node fails, then its replica from another rack must take over as a new master. The clients need to be aware of this change to be able to direct requests to the right rack. Depending on the application's consistency and durability requirements, writes may be propagated asynchronously.

4 RACKOUT DATA SERVING

The basic building block of RackOut is a group of servers, typically within the same datacenter rack, that are tightly interconnected with an internal high-bandwidth, low-latency fabric and can directly access each other's memory by leveraging one-sided operations. Thanks to this internal fabric, the aggregate memory of the group can be perceived as unified.

RackOut leverages the capability for fast memory access within the boundaries of a rack to achieve better load balancing across the set of servers participating in memory pooling. We refer to the number of intra-rack servers pooling memory as the *Grouping Factor (GF)*. Popular data residing in the local memory of heavily loaded servers can be directly accessed by less loaded servers in the same rack, thus alleviating the queuing effects on the busy server owning that memory. This optimization is enabled by the fact that the internal fabric of the rack and memory bandwidth of each individual server are under-subscribed while the CPU or the external-facing NIC of the hot server is fully utilized. Also, by aggregating the memory of a rack, the per-node load increases marginally, while the maximum node throughput grows almost linearly with GF. For example, for a workload with one billion keys, going from 512 servers to 32 racks increases the load on the hottest rack only by 5%, as compared to the hottest server of the equivalent scale-out configuration.

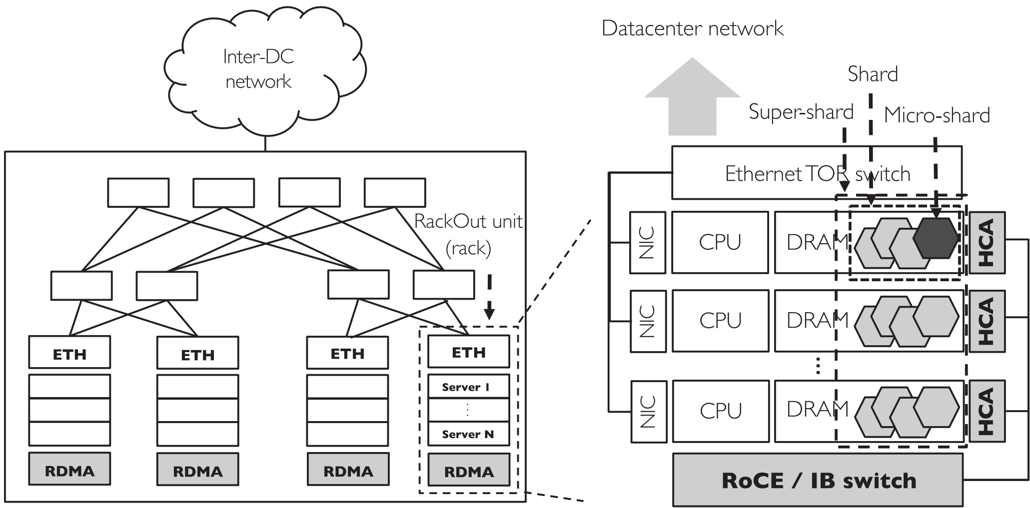


Fig. 3. RackOut datacenter architecture based on RDMA. A super-shard comprises all the micro-shards within a rack. HCA stands for Host Channel Adapter, which is a NIC with support for RDMA.

Figure 3 illustrates the RackOut datacenter architecture. Each RackOut unit connects a group of servers via an internal rack-scale fabric that supports one-sided RDMA operations (e.g., Infiniband, IB). Each server in the unit stores hundreds or thousands of micro-shards as in conventional KVS. A RackOut unit exposes the abstraction of a single *super-shard* composed of all the micro-shards within the unit.

Figure 3 also clearly illustrates the key assumption beyind the model: By confining the fabric to the unit, the RackOut model sits between the traditional scale-out model and the full-scale RDMA fabric deployment. While such large-scale RDMA fabrics do exist, they are still largely confined to the supercomputing space rather than cloud computing infrastructure. Recent work has even shown that scaling RDMA over commodity Ethernet introduces emergent safety, performance, and monitoring challenges, such as issues of congestion, dealing with deadlocks and livelocks, and other subtleties of priority-based flow control [35, 74]. Using Ethernet for RDMA in today's datacenters is common, since most software systems still use TCP/IP for communication. In addition to the flow-control issues of commodity Ethernet, specialized rack-scale systems [6, 29, 56] are likely to become the building blocks of future datacenters, where intra-rack and inter-rack access latencies are expected to remain non-uniform; similarly to the design from Figure 3, it is easy to imagine specialized rack-scale systems in lieu of RDMA racks, and an RDMA network connecting the rack-scale systems. In such designs, the bottleneck would possibly shift from the CPU to the network interface, requiring RackOut or a similar mechanism to mitigate the imbalance. From an application perspective, the proposed RackOut and scale-out models are similar: Clients connect to servers via the network (assuming clients and servers run in different racks [27]) and applications rely on replication to scale beyond the unit of capacity (i.e., rack or server) and ensure data availability.

RackOut leverages the CREW model. This means that write requests must be directed to the specific server within the rack that owns the data, but that read requests can be load-balanced to any server of the rack. Therefore, from the perspective of a KVS, the hash function determines the micro-shard and its associated server, which is used directly to select a specific server on write requests. For reads, the client schedules the request to a random server within the target server's RackOut unit.

In the rest of this section, we present a first-order analysis showing the maximal speedup attainable by RackOut as a function of the read/write mix and the GF (Section 4.1) and subsequently build a queuing model for RackOut to study service time implications at the tail latency (Section 4.2). We analyze the sensitivity of the skew to the dataset distribution (Section 4.3), the synergy of RackOut with dynamic replication and migration (Section 4.7), and the impact of remote read penalties on performance (Section 4.5). Throughout the analysis, we assume identical server building blocks in terms of processing and memory capacity for both scale-out and RackOut configurations.

4.1 Load Balancing with RackOut

We define the *rack skew* as the rack-scale analog to the shard skew, specifically as the ratio of the traffic on the most popular rack over the average traffic per rack. In the following analysis, we use L_{max} to refer to the load (expressed as a fraction of requests) of the server/rack with the hottest shard/super-shard and L_{avg} for the average server load. The straggler (i.e., the system's node with the highest load) determines the highest aggregate stable throughput. The straggler in a traditional scale-out environment is the server with the highest load:

$$L_{max} = shard_skew_1 \times L_{avg},$$

where $shard_skew_1$ is the shard skew in the scale-out deployment. In a RackOut organization, the building blocks are racks rather than servers. In such a deployment with a rack size of GF servers, the straggler rack's load is

$$L_{max} = rack_skew_{GF} \times GF \times L_{avg},$$

where $rack_skew_{GF}$ is the rack skew in a RackOut environment with a Grouping Factor of GF.

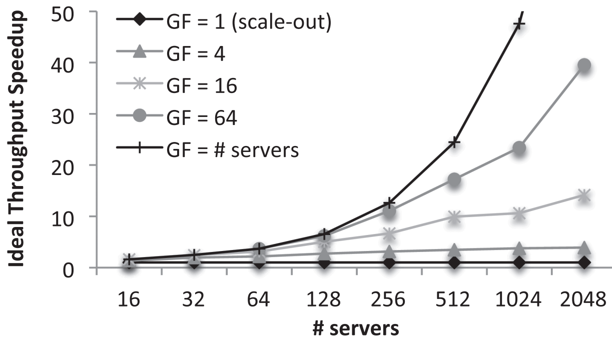
The time a straggler needs to crunch through its load is inversely proportional to the available resources that can be utilized. We assume that the single-server compute power that can be used to serve the hottest shard in the scale-out model is $compute_1$. The compute power that can be used on the hottest super-shard in the case of RackOut is $compute_{GF} = GF \times compute_1$. Overall, for a read-only workload, the ideal speedup derived from the RackOut model over the scale-out model is

$$Ideal\ speedup = \frac{\frac{shard_skew_1 \times L_{avg}}{compute_1}}{\frac{rack_skew_{GF} \times GF \times L_{avg}}{compute_{GF}}} = \frac{shard_skew_1}{rack_skew_{GF}}. \tag{1}$$

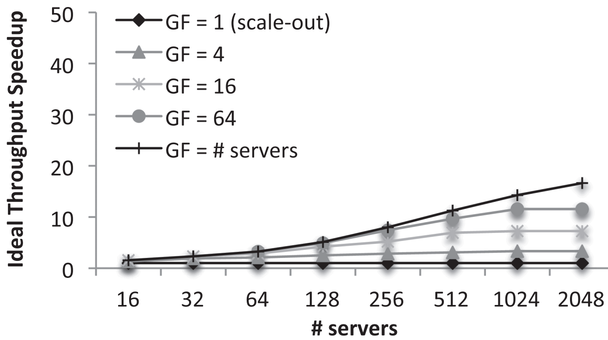
Given a CREW model, Equation (1) provides only an upper bound on the speedup for read-write workloads, as writes cannot be balanced within the rack.

We are not aware of a closed-form formula that determines the per-server load, L_{server}. Instead, we perform the following experiment: We generate a 250M-key dataset built out of a randomly generated sparse key space, then allocate keys to micro-shards according to a hash function, and, finally, compute each key's popularity according to the power-law distribution. A server's popularity is the sum of its keys' popularities; for RackOut, a GF-sized rack's popularity is the sum of the popularity of the GF servers in that rack.

Figure 4 shows the impact of GF on the rack skew. Figure 4(a) shows the benefit of perfect load balancing within a rack of a given GF according to Equation (1); for a given 512-server configuration, grouping the servers into RackOut units of 64 servers (i.e., GF=64) provides an ideal speedup opportunity of 16×. Figure 4(b) quantifies the impact of having 5% of writes on the ideal throughput speedup. Even though such a workload is clearly read-dominated, the CREW model bounds the speedup, similarly to Amdahl's law; the maximum performance improvement with GF=64 drops from 16× to 9×. Figure 5 illustrates the sensitivity to the read/write mix for various GF configurations.



(a) Read-only, perfect intra-rack balancing.



(b) 5% writes, CREW access.

Fig. 4. Ideal throughput speedup for different GFs (250M keys, Zipfian $\alpha = 0.99$).

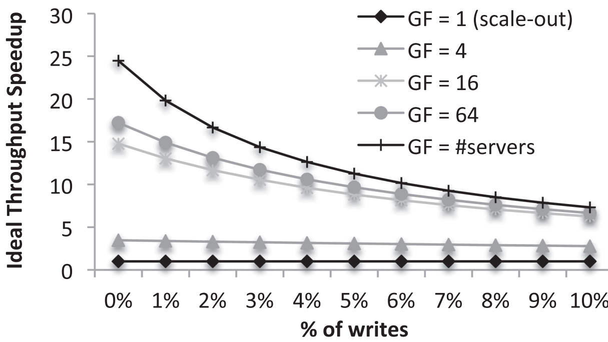


Fig. 5. Sensitivity to write %, 512 servers.

4.2 A Queuing Model for RackOut_static

Equation (1) suggests that the expected benefit of a RackOut organization is commensurate to the reduction in shard skew. We now reinforce the claim that this is a good approximation metric for the expected improvement in performance under a given SLO by leveraging basic queuing theory principles. Under a simple open-queuing system approach for real-world systems that service millions of client requests and serve huge datasets, we assume that the requests follow a Poisson distribution, data popularity follows a power-law (Zipfian) distribution, and that each key has the

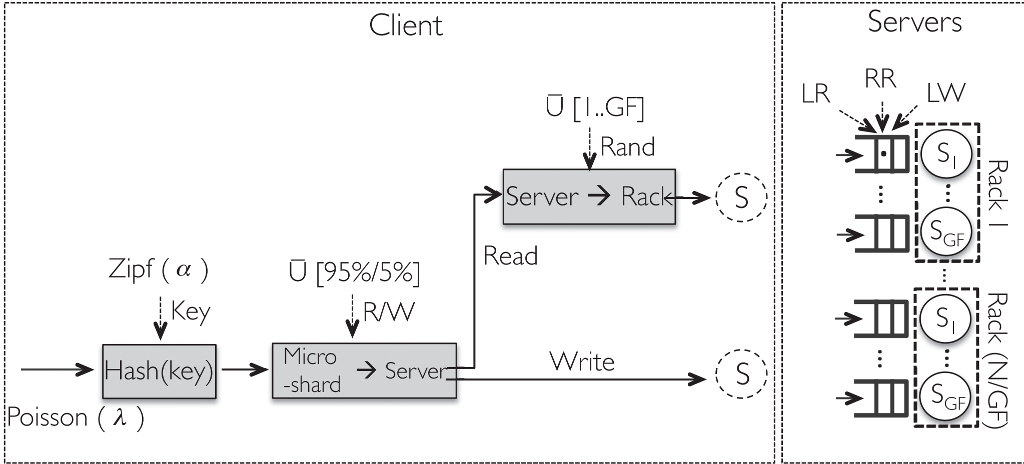


Fig. 6. CREW client and server queuing model for `RackOut_static`.

same average read/write ratio. The three distributions are orthogonal but equally critical to the queuing effects that arise in the system. Given Poisson request arrivals, queuing theory provides the tools to determine stability conditions (arrival rate < service rate, or $\lambda < \mu$), as well as each server's performance under a given SLO.

Figure 6 describes the key elements of the queuing model for `RackOut_static`. Requests follow an open-loop arrival process with a given rate λ with Poisson inter-arrivals. Each request therefore consists of a timestamp, a key selected randomly according to popularity, and a read/write tag selected uniformly according to the set probability. The client-side process maps the key to a micro-shard using a perfect hash function. Requests for each micro-shard P follow a Poisson arrival process with a rate:

$$\lambda_P = \lambda \times \sum_{k \in K_P} zipf(k); K_P = \{k | hash(k) = P\}.$$

According to CREW, the micro-shard directly determines the server node for write requests, but read requests are load balanced among the nodes of the selected RackOut unit. In our model, queuing happens on the server side, with one queue per server, and requests are served in FIFO order. The model distinguishes between three types of requests T: (i) read requests that can be served from the local memory of the server (*LR*)), (ii) read requests that require one-sided operations on the RackOut fabric interconnect (*RR*), and (iii) local write requests (*LW*). On any given server, the power-law distribution of the keys, the hash function, the RackOut GF, and the read/write mix together determine the arrival rate for each request type,

$$\lambda = \sum_{i=1}^{N_{nodes}} \sum_{\{t \in T\}} \lambda_{it}; T = \{LR, RR, LW\}.$$

The system is stable if and only if:

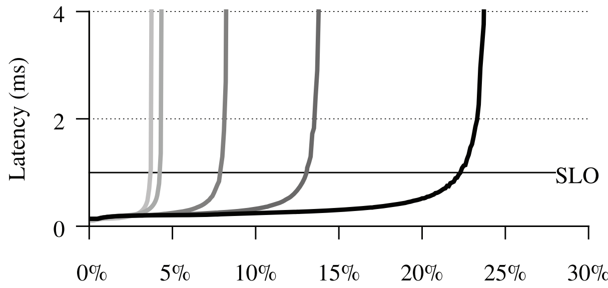
$$\forall i \in \{1..N_{nodes}\} : \sum_{\{t \in T\}} \lambda_{it} \times \bar{S_t} < 1.$$

The resulting queuing model depends on the service time distributions S_t. To extract performance results from our queuing model, we instrument it with realistic service times, which we

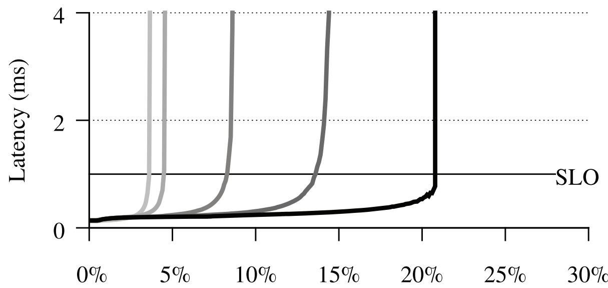
Table 1. Average Service Time of Basic Operations Used to Size the Queuing Model

Operation	LR	RR	LW	RR/LR	Propagation
RDMA	$5\mu s$	$8.4\mu s$	$6.9\mu s$	1.68	$34.9\mu s$
soNUMA (projected)		$5.8\mu s$		1.16	

Model (GF1) Model (GF4) Model (GF16)
Model (GF2) Model (GF8)



(a) DC throughput (% of max capacity), read-only



(b) DC throughput (% of max capacity), 5% writes

Fig. 7. Datacenter-wide 99th percentile latency vs. utilization, determined using the queuing model and RDMA parameters (512 servers, Zipfian $\alpha = 0.99$).

derive from the RackOut KVS system (RO-KVS) described in Section 6. Using RO-KVS, we measure the maximum node throughput $1/\bar{S_t}$ for each of the three request types. To simplify the model, we assume that each of these has a deterministic distribution of service times equal to its average. Local reads are the most lightweight operations and as such are associated with the lowest service time. In RO-KVS running on top of our Mellanox RDMA cluster, local writes and remote reads are 1.38× and 1.68× slower than local reads, respectively (see Table 1). The service times include all the CPU processing, such as network packet processing, to service a single key-value lookup or update.

Figure 7 studies the 99th percentile behavior of this queuing model for a deployment of 512 nodes and a Zipfian key popularity distribution with $\alpha = 0.99$. Given the lack of a closed form, we rely on discrete event simulation with 10 million arrival events for each measurement. We configure the model with RO-KVS service times and the propagation delay that we obtain on RDMA (Table 1). We define the *datacenter throughput* (on the x-axis) as the fraction of the throughput achieved as compared to a uniform workload with 100% read requests on a scale-out deployment (GF=1). We

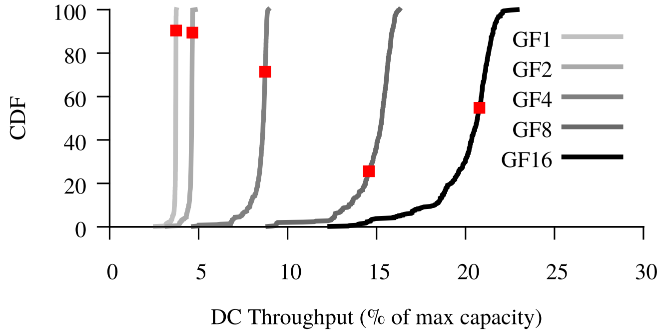


Fig. 8. Datacenter utilization for 500 different datasets of 50 million objects with Zipfian $\alpha = 0.99$ data popularity distribution (5% writes). The red dots represent the key distribution used throughout this article.

use the model to determine the maximum utilization at an SLO defined as handling 99% of requests in less than 1 millisecond, provided that none of the nodes are saturated.

Figure 7(a) shows the impact of `RackOut_static` on read-only workloads. For smaller GFs (including scale-out, GF=1), the datacenter wide tail latency spikes rapidly as the hottest RackOut unit (or scale-out node) reaches saturation. With larger groupings, the intra-rack load balancing reduces the skew in arrival rates and the tail latency rises with load according to the familiar pattern of open-loop models. When considering the SLO, RackOut with GF=16 achieves a speedup of 6× over scale-out. This is a substantial increase in performance due to better load balancing, even though the majority of requests will suffer the 1.68× penalty associated with accessing remote memory over RDMA. In comparison, Figure 4(a)'s idealized model predicts the maximum speedup for the same power-law distribution of keys to be of 9.9×. However, the idealized model does not account for the remote memory access penalty or the requirement to meet any particular SLO.

Figure 7(b) shows the same results for a workload with 5% of writes. While the key distribution remains identical, the inability to balance the write requests limits the performance improvements. The imbalance in writes is also the reason for the latency spike at the saturation point: While the average datacenter utilization is still low, resulting in low 99th percentile latency, the server with the hottest shard saturates while all the other servers, including the ones in the same RackOut unit, are still far from saturated. This is particularly obvious with $GF \geq 16$, where the tail latency hardly increases before saturation.

4.3 Sensitivity to Skew

Figure 7 shows the result of experiments conducted on a single, randomly generated dataset of sparse keys, using a balanced hash function. We performed these experiments multiple times, with different random seeds, and noticed some non-trivial variability in the results.

Figure 8 shows the CDF of the saturation points for 500 different datasets of 50 million objects for the configuration of 512 servers with 5% writes. Each point in this figure corresponds to the datacenter saturation point of one randomly generated dataset. The distribution of these saturation points shows that (i) traditional scale-out achieves consistently very low utilization, as it always suffers from high shard skew; (ii) the distributions do not overlap, meaning that the datacenter's utilization grows monotonically with an increase in GF; (iii) the relative standard deviation for the five shown GFs ranges from 2.7% to 8.6%.

4.4 Hottest Key Analysis

This work builds on popularity distributions following Zipf's law, which accurately represent real-world KVS workloads and hence are commonly used in KVS research [15, 22, 37, 44, 45]. An

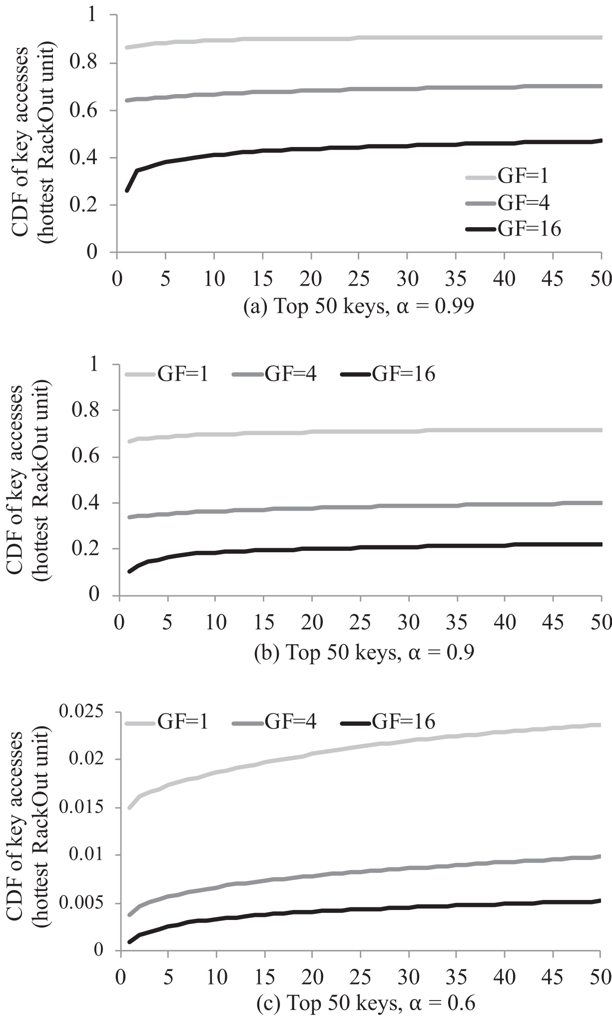


Fig. 9. CDF of key accesses to the hottest RackOut unit for Zipfian $\alpha = 0.99$ and $\alpha = 0.9$ data popularity distributions. Top 50 keys of the hottest RackOut unit sorted by popularity in decreasing order.

inherent property of Zipf is that most of the load on the hottest node is associated with a tiny fraction of keys. Zipf aims to reflect the behavior of a large majority of data serving applications where the popularity of different keys is heavily skewed and/or load spikes occur regularly.

Figure 9 shows the CDFs of key accesses on the hottest RackOut unit for different Zipfian distributions and different grouping factors. The figures focus on the top 50 keys (i.e., with the highest popularity) of the hottest RackOut unit. For GF = 1 and $\alpha = 0.99$, the first 50 keys contribute up to 90% of the load, for GF = 4 up to 70%, and for GF = 16 up to 46%. For smaller skew factors, key accesses are more spread out and the hottest keys are less critical. For higher grouping factors, the top 50 keys contribute much less to the overall load of the hottest RackOut unit.

To avoid early saturation in scale-out (i.e., GF=1), it is possible to replicate hot keys or microshards across different nodes. However, a multitude of replicas will be required to absorb the skew, and this will further lead to excessive update and storage overhead. Replicating individual keys reduces storage overhead but increases routing complexity and the per-key metadata overhead [63].

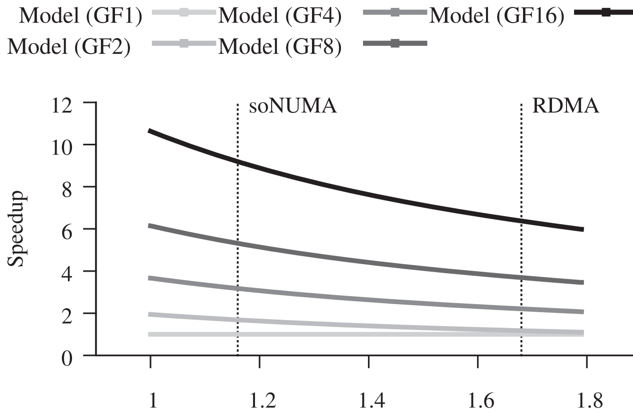


Fig. 10. Speedup for different RR/LR ratio (read-only).

In addition, unexpected temporal shifts in popularity, which are unfortunately not captured by Zipf, would make replication all that much harder to use as a skew mitigation mechanism. In the next section, we analyze a dynamic migration and replication algorithm and show how RackOut can reduce the number of replicas necessary to achieve maximum utilization. The larger the grouping factor, the less overhead there is from updating replicated keys.

4.5 The Impact of Faster Remote Reads

The impact of grouping the nodes together in RackOut depends on the ratio between RR and LR service times. The higher the ratio, the smaller the impact. So far, we relied on the performance of RO-KVS on RDMA to estimate the impact of RackOut through simulation. Modern RDMA technology already provides remote memory access latency that is low enough for effective rack-scale resource aggregation [64]. In addition, the increasing trend toward higher integration and low-latency fabrics will further lower the RR/LR ratio and improve the effectiveness of RackOut.

A representative of such emerging tightly integrated solutions is Scale-Out NUMA (soNUMA) [17, 56], which delivers remote memory access within a small factor over local memory access. soNUMA is an architecture and protocol that supports one-sided remote read and write operations, i.e., a strict subset of RDMA operations.

soNUMA relies on a remote memory controller (RMC), which is integrated within the cache-coherence hierarchy of the CPU, and layers a lean remote memory access protocol on top of the standard NUMA transport, thereby removing all major sources of latency and throughput overheads found in today's commodity networks: the PCIe bus, DMA, and deep network stack. Prior work showed that soNUMA delivers memory access latency that is $3-4\times$ of local DRAM access, with low CPU overheads.

Figure 10 shows the speedups at saturation of RackOut over a traditional scale-out deployment for different grouping factors as a function of the RR/LR ratio, derived by our RackOut queuing model, using the same server configuration and dataset as in Section 4.2 (Figure 7). We present only the read-only data as they are the most sensitive to the RR/LR ratio and highlight two points on the RR/LR curve, which correspond to (i) the actual ratio measured on a commercially available solution based on Mellanox ConnectX-3 Pro adapters and (ii) the extrapolated ratio expected from an soNUMA cycle-accurate simulation model [17] (Table 1). While RackOut over RDMA already delivers substantial performance improvements over the traditional scale-out approach, soNUMA's faster remote memory access further highlights the potential of RackOut. For instance, for GF16, soNUMA delivers 1.45× higher performance improvement than RDMA.

4.6 Determining the Maximum RackOut Unit Size (GF)

A key design goal in RackOut is to ensure that the internal (secondary) fabric does not become a bottleneck for the maximum GF we would like to support. The key metric that drives sizing decisions is throughput (IOPS), because data serving workloads are inherently throughput-bound; the NIC processing overheads dominate because of the workload's fine-grain access nature. The NIC's maximum IOPS and the maximum per-core remote read rate determine the maximum number of nodes that can be grouped together. Assuming the worst-case workload, where only a single item residing on a single server is being looked up, the sustainable IOPS of that server's NIC should be higher than the sum of the remote read rates of all the other nodes in the RackOut unit. The maximum per-core remote read rate depends on the remote service time (RR), which in turn depends on the processing overheads of the primary (datacenter-scale) network and the application processing overheads.

In Section 6.2, we report the service times for our RackOut implementation for both soNUMA and RDMA. Assuming an RDMA NIC supporting up to 40 million requests per second, the maximum RackOut unit size supported is 16 nodes, leaving about 50% of NIC throughput unused to avoid detrimental queuing effects. This provisioning is based on the aforementioned worst-case workload. In reality, other items on other nodes are being accessed, too, and there is a small fraction of writes, both of which reduce the remote read rate to any particular node. Such load distribution and additional local processing puts less pressure on individual NICs.

In data serving, bandwidth is typically not a concern, unless the items are multiple KBs in size. For sufficiently large data item sizes (2–4KB in RDMA), the following condition must hold:

$$\forall i \in \left\{1..\frac{N}{GF_{racks}}\right\} : primary_fabric_BW_i < secondary_fabric_BW_i < node_mem_BW.$$

For each RackOut unit, there should be enough bisection bandwidth (BW) within the unit and enough memory bandwidth within each server, such that none of the two becomes a bottleneck. Per-server memory bandwidth is unlikely to introduce a bottleneck, especially with emerging high-bandwidth memory technologies such as HBM [1]. The bottleneck in most deployments is expected to be either the bandwidth of the primary datacenter-scale fabric or the maximum CPU processing rate. Similarly to our throughput analysis, in reality the CPU load is somewhat distributed across the servers of the hot rack, plus there are writes, which, under CREW, are always locally processed.

4.7 Combining Dynamic Replication and Migration with RackOut

The results of the queuing model in Section 4.2, including Figure 7, assume that the key-value store has no provision for dynamic replication or migration. We extend the queuing model to include the dynamic migration and replication of micro-shards on a fixed cluster size of 512 servers. The model runs a greedy algorithm operating as follows: (i) it identifies the hottest server in the cluster and its corresponding RackOut unit; (ii) for that RackOut unit, it identifies the micro-shard contributing most to the load; (iii) if this micro-shard has never been migrated or replicated, it first migrates it to the least loaded node in the cluster; (iv) else it replicates the micro-shard to the least loaded node in the cluster; (v) in both cases, the hash table metadata is updated and the system load-balances the read traffic equally across replicas. Replicas are only made across different RackOut units. The model assumes that writes eventually propagate to all of the replicas. Using this model, we determine analytically the datacenter-wide utilization after each migration/replication step.

The model is optimistic in a number of ways: First, it assumes that migration and replication decisions and events do not impact CPU utilization but instead happen instantly. The model does account for the full cost of maintaining consistency but only to the extent of updating each replica;

any additional overheads associated with replication (e.g., quorum-based decisions) are not taken into consideration. Furthermore, the model assumes perfect monitoring of the load on each server and that the popularity distribution does not change over time. Any realistic implementation of micro-shard dynamic replication and migration would incur higher CPU overhead (e.g., load monitoring) and higher consistency maintenance costs and would have to rely on partial, and potentially outdated metrics to make policy decisions [21, 28, 37, 38]. Finally, in real deployments, the number of steps required depends on the system's load and decisions are neither instantaneous nor free in terms of resource utilization. In a practical implementation, the system will gradually keep migrating/replicating until the load on the hottest server or RackOut unit is reduced.

Figure 11 evaluates the improvement in datacenter utilization after each step in the greedy algorithm for the scale-out (GF=1) and RackOut (GF>1) configurations. As in previous experiments, the model is sized with RDMA service times from Table 1. The model is configured with 1 billion keys, with Zipfian $\alpha = 0.99$ key popularity, hashed into 512×1000 micro-shards on the cluster (i.e., with 1000 micro-shards per server). The key-space is representative of key-value stores used in large social networks, and the granularity of micro-sharding is aggressive for modern key-value stores (e.g., FaRM [22] defaults to ~100 micro-shards on a server with only 16GB of RAM). Figure 11(a) shows the trend for a read-only workload (read $\alpha = 0.99$). This is a degenerate case where consistency maintenance is a non-issue. All configurations converge to a point where skew is eliminated (i.e., load is distributed uniformly across servers) and utilization is primarily determined by the local:remote service times; hence scale-out can outperform RackOut when given enough replicas, simply because all of its accesses are local. With an unbounded number of replicas, datacenter utilization eventually reaches 100% (not shown on Figure 11).

Figure 11(b) shows the trend assuming a workload with 5% of writes following a Zipfian distribution that is significantly less skewed (i.e., write $\alpha = 0.9$). The read distribution is computed based on the write α and the read-write ratio to match the aggregate distribution of $\alpha = 0.99$. Using different distributions for reads and writes aims to reflect the behavior of the applications where writes are not necessarily strongly correlated with the access popularity distribution. Also, with less skewed writes, the aim is to introduce at least one write to low-popularity keys. Based on our simulations, we conclude that the overhead of updating a multitude of replicas is still high; the maximum DC throughput for GF=1 is 63%. Since our model is highly optimistic, maintaining hundreds of replicas would in reality introduce much more overhead and lead to faster saturation. For 1% of writes and less, replication introduces smaller performance penalty [32]. As Figure 11 shows, a combination of moderate replication with RackOut represents an effective design point; for example, GF=16 with only a few replicas matches the utilization achieved by scale-out with tens of replicas.

Figure 11(c) shows the trend assuming a workload with 5% writes, where both reads and writes follow the same Zipfian distribution of $\alpha = 0.99$. We observe that (i) the recurring cost of maintaining replica consistency with each write inherently limits the performance in all configurations; (ii) for small replica counts, RackOut configurations with larger GFs outperform smaller GFs; (iii) given enough replicas, all configurations tend to converge to roughly the same overall datacenter utilization; (iv) the number of replication/migration steps required increases substantially for smaller GFs and scale-out (GF=1). All curves plateau at a datacenter utilization of ~30%. For GF1, 45 replicas are required to reach the plateau, including 20 for the hottest micro-shard alone. In practice, scale-out KVS systems tend to cap the number of replicas to a much smaller number, e.g., according to Huang et al., Facebook allows for up to 10 replicas of each micro-shard [38]. With GF16, only 3 replicas achieve the same result as GF1 with 45 replicas. Therefore, RackOut is superior to scale-out as it requires fewer replicas to absorb a given load skew and, consequently, reduces the overheads associated with dynamic replication.

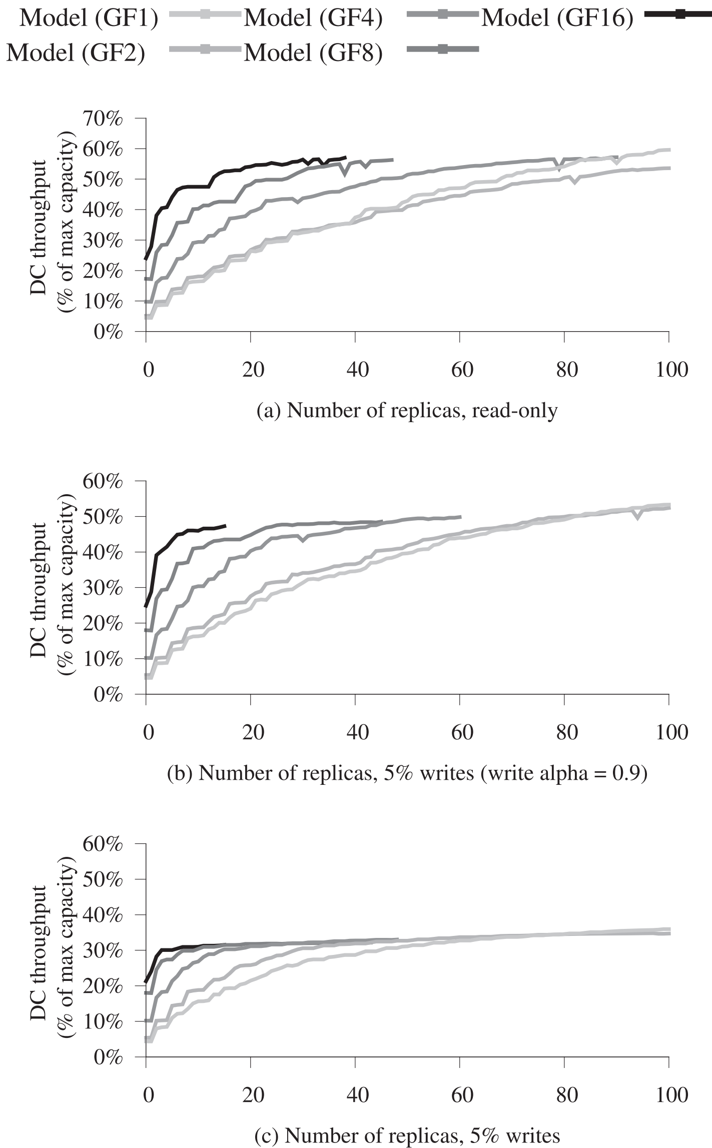


Fig. 11. Dynamic replication in RackOut. $\alpha = 0.99$ in all the cases. On figure (b), writes follow alpha=0.9.

5 ADAPTIVE LOAD BALANCING FOR READ-WRITE WORKLOADS

`RackOut_static` is an efficient, stateless approach to load balancing for skewed workloads dominated by reads. The key limitation of `RackOut_static` is that it does not account for the imbalance within individual RackOut units. Such imbalance exists due to performance interference, periodic activities within applications (e.g., garbage collection), skewed write distribution, remote accesses or accesses to disk, and so on. For instance, `RackOut_static` is oblivious to the CREW-induced skew and it randomly schedules read requests across a RackOut unit, without taking into account the *exclusive-writes* property of CREW. Thus, `RackOut_static` often leads to skewed aggregate distribution of load across the servers for read-write workloads (≥5% writes), as illustrated on the

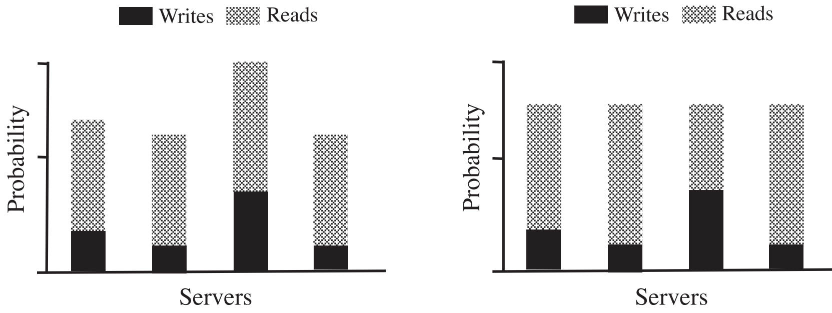


Fig. 12. Illustration of combined read/write distributions for `RackOut_static` (left) and `RackOut_adaptive` (right). The figures illustrate trends rather than actual numbers.

left-hand side of Figure 12. `RackOut_static` also does not account for variable service times but assumes that each server of a RackOut unit processes incoming requests at a similar pace. Uniform service times is, at least, not trivial to achieve in shared environments where slowdowns happen regularly.

Adaptive load balancing for RackOut (`RackOut_adaptive`) is a scheduling mechanism that adapts the read workload to the load imbalance within individual RackOut units. For read-write workloads, `RackOut_adaptive` generates a read distribution for each RackOut unit that is inversely proportional to its write distribution. The outcome of `RackOut_adaptive` is illustrated on the right-hand side of Figure 12, where the read and write distributions form a uniform aggregate distribution. By sampling the computed read distributions when scheduling read requests, the clients adapt to the write load to achieve, in the ideal case, uniform utilization of the RackOut unit. To cope with non-uniform service times, `RackOut_adaptive` further adjusts the read distribution to reflect the processing rates of different servers.

The `RackOut_adaptive` mechanism consists of a server-side monitoring system and a client-side scheduler. To tackle the load imbalance at the level of a rack, the monitors maintain statistics about each individual server's progress and transfer that data to the coordinator clients periodically and upon request (Figure 13). A coordinator client uses the information collected by the monitors to generate the read distribution of its associated RackOut unit that it subsequently broadcasts to the other clients. The clients then regularly sample the per-rack read distributions when scheduling read requests. We first look at the monitor design and then explain how the clients take advantage of the collected information.

5.1 Load Monitoring

The monitor in `RackOut_adaptive` is an application component that keeps track of the number of processed requests and the average service time on the local server. The coordinator clients periodically obtain this information and use it to determine how to schedule read requests in the future to achieve fair resource utilization and better throughput. Figure 13 illustrates the process of periodically obtaining the number of processed writes (`wr_cnt`) since the last polling and the average service time (`avg_st`) from each server of a single rack. The coordinators compute the per-rack read distributions based on the information obtained from the servers of their respective RackOut units.

Collecting one of the two metrics alone is not sufficient. The average service time indicates how long it takes to complete one request, but it does not capture the skew in the input key distribution. In particular, the RackOut concept assumes single writer (i.e., CREW) and thus writes cannot be balanced. The servers receiving larger fractions of writes due to skew will be under higher load

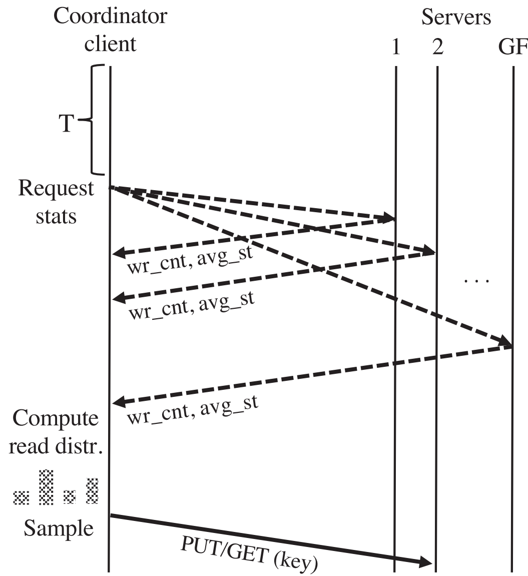


Fig. 13. Message exchange in `RackOut_adaptive`. wr_cnt - processed writes; avg_st - average service time.

than those processing fewer writes. Thus, to capture the skew, we propose keeping track of the number of processed write requests on each server of the rack and transferring these counters along with the average service times to the coordinator clients.

The service time varies across the servers due to interference with background activities, such as garbage collection or interference with collocated applications. The overhead coming from remote reads alone is negligible as compared to disk writebacks, garbage collection, or interference with any batch application performing heavy computation.

5.2 Scheduling Algorithms

In `RackOut_adaptive`, clients make scheduling decisions for reads based on the current write distribution and average service times. By knowing which servers are slower (i.e., have longer processing times) and/or execute a larger fraction of writes, clients schedule read requests such that each server's CPU within a RackOut unit is, in the ideal case, equally occupied. Coordinator clients retrieve the information collected by the monitor periodically either through an *out-of-band* mechanism or using regular data traffic (piggybacking) (Figure 13). The polling interval is configurable and for short intervals the overhead may be high, but the scheduler may be taking more accurate decisions. With longer intervals, the polling overhead reduces, but the scheduler's accuracy may degrade. The interval should be configured based on how frequently the popularity distribution and the service times change. Our adaptive approach works best when shifts in popularity are relatively infrequent and/or slow [38]. We set the polling interval to 10s, which accommodates for moderately frequent changes in popularity.

Using the information collected by the monitor, the scheduler computes the following: (i) fraction of reads (F_r), (ii) fraction of writes (F_w), (iii) write distribution (P_w), and (iv) read distribution (P_r). F_r and F_w are global variables that are computed based on the workload's read-write ratio and are maintained by the coordinator. The `RackOut_adaptive` mechanism comprises two algorithms: (i) Algorithm 1, which generates the read probability distribution based on the write distribution, and (ii) Algorithm 2, which adjusts the read distribution of Algorithm 1 to reflect the variability in average service times.

Algorithm 1: Figure 12 illustrates the effect of Algorithm 1; the figure on the left illustrates the output of the original, `RackOut_static` scheduling, and the figure on the right shows how `RackOut_adaptive` adjusts the read distribution such that each server receives an equal amount of read and write requests. In `RackOut_adaptive`, a coordinator client computes the read probability for each server i of a RackOut unit as follows:

$$P_r[i] = \frac{\frac{1}{GF} - P_w[i] \times F_w}{F_r}.$$

$P_r[i]$ is computed such that the total fraction of the input load per server equals $\frac{1}{GF}$, where GF is the grouping factor (i.e., the size of a RackOut unit). Algorithm 1 does not account for the variability in mean service times but only considers the skew imposed by the workload and the CREW access model.

Algorithm 2: Algorithm 2 adjusts the read distribution using the mean service times obtained from the monitor. The scheduler first normalizes the service times to the 0–1 range and then adjusts the read probabilities using the standard deviation of service times—the difference between the mean service time of each server $ST_{norm}[i]$ and $\frac{1}{GF}$. Rather than computing the variance and then standard deviation, the algorithm subtracts the difference from the read probability:

$$P_r[i] = P_r[i] - \left(ST_{norm}[i] - \frac{1}{GF}\right).$$

For negative differences, for example, the read probability will increase, because a negative difference means the mean service time on that server is lower than the global mean and that server should be able to process more requests.

ALGORITHM 1: Computing read distribution based on skewed write distribution

```
1:  Input: Write count from each server, total read and write counts
2:  Output: Read probability distribution

3:  wr_fr ← wr_total / (wr_total + rd_total)
4:  rd_fr ← 1 − wr_fr

5:  for i ← 0, i < GF, i+ = 1 do
6:      wr_pr[i] ← wr_cnt[i] / wr_total

7:  if wr_total > 0 and rd_total > 0 then
8:      for i ← 0, i < GF, i+ = 1 do
9:          diff ← (1 / GF − wr_pr[i] ∗ wr_fr) / rd_fr
10:         if diff < 0 then
11:             to_sub ← abs(diff) / (GF − 1)
12:             rd_pr[i] ← 0
13:         else
14:             rd_pr[i] ← diff

15:     if to_sub > 0 then
16:         for i ← 0, i < GF, i+ = 1 do
17:             if rd_pr[i] > 0 then
18:                 rd_pr[i] = rd_pr[i] − to_sub
```

In the following text, we disclose the algorithms for computing the read distribution based on the write distribution and the mean service times. Algorithm 1 shows the pseudo code for creating and Algorithm 2 for adjusting the read probability distributions.

Algorithm 1 first computes the read to write ratio of the workload using the counters obtained from the monitor. Then, using the previously computed write distribution, the algorithm computes the read distribution, such that the aggregate probability of selecting any server equals $\frac{1}{GF}$ in the ideal case. For low read-to-write ratios, the algorithm will assign 0 probability for reads to the hottest server and adjust other read probabilities accordingly. For highly skewed workloads, the aggregate probability of selecting the hottest server may be above the ideal target of $\frac{1}{GF}$, which imposes balancing only across GF-1 servers. Algorithm 1 assumes at most one overloaded server, but it can be extended to support less skewed Zipfian distributions where we may have more than one server with probability above $\frac{1}{GF}$.

Algorithm 2 first normalizes the service times to the 0–1 range and computes the mean service time across all the servers. The algorithm then adjusts the read distribution computed using Algorithm 1 based on the standard deviation of service times. Algorithm 2 is necessary if the service time varies across the servers due to, for example, resource contention among multiple processes running on the same server or periodic background activity such as garbage collection [20].

By using the read distributions, periodically adjusted and propagated by the coordinator clients, each client decides which server within the target RackOut unit should process the next request. Each request results in (i) determining the RackOut unit ID based on key and (ii) sampling the read distribution associated with the target RackOut unit to pick a server.

6 A RACKOUT KEY-VALUE STORE

This section describes RackOut KVS (RO-KVS), a proof-of-concept KVS tailored to the RackOut model that we use to validate the RackOut queuing model.

ALGORITHM 2: Adjusting read distribution based on average service times

1: **Input:** Average service time for each server, read probability distribution
2: **Output:** Read probability distribution

3: **for** $i \leftarrow 0, i < GF, i+ = 1$ **do**
4: $\quad avg_st_total \leftarrow avg_st_total + avg_st[i]$

5: **for** $i \leftarrow 0, i < GF, i+ = 1$ **do**
6: $\quad avg_st_nr[i] \leftarrow avg_st[i] \; / \; avg_st_total$

7: **for** $i \leftarrow 0, i < GF, i+ = 1$ **do**
8: $\quad diff \leftarrow rd_pr[i] - (avg_st_nr[i] - 1 \; / \; GF)$
9: $\quad$ **if** $diff < 0$ **then**
10: $\quad\quad to_sub \leftarrow abs(diff) \; / \; (GF - 1)$
11: $\quad\quad rd_pr[i] \leftarrow 0$
12: $\quad$ **else**
13: $\quad\quad rd_pr[i] \leftarrow diff$

14: **if** $to_sub > 0$ **then**
15: $\quad$ **for** $i \leftarrow 0, i < GF, i+ = 1$ **do**
16: $\quad\quad$ **if** $rd_pr[i] > 0$ **then**
17: $\quad\quad\quad rd_pr[i] = rd_pr[i] - to_sub$

6.1 RO-KVS Architecture

RO-KVS consists of (i) a coordinator node managing the key hash space, (ii) a client library and an access protocol, and (iii) RackOut nodes that hold the data. RO-KVS is built on top of FaRM [22], a framework for distributed data serving applications that provides the basic mechanism enabling a CREW key-value store. In particular, FaRM offers atomic one-sided remote access to objects over RDMA. FaRM implements atomic remote object reads via optimistic concurrency control by encoding versions in objects. Should an object write overlap with a remote read request, the framework detects the inconsistency and retries the operation. Next-generation rack-scale fabrics such as soNUMA propose to add hardware support for atomic, multi-cache line remote reads [18] that eliminate the software overheads associated with atomicity checks and data layout.

At the lowest level, FaRM organizes data into objects, each associated with a global address. A global address consists of a 32-bit region identifier and a 32-bit offset within the region. FaRM relies on consistent hashing to locate the owner region using the region identifier. RO-KVS is layered on top of FaRM's object abstraction. Both RO-KVS and the original FaRM hashtable assume that the required metadata is known by each node. The collection and broadcasting of the metadata is performed by a coordinator node. Besides hashtables, one can build other types of data structures, such as B+ trees, on top of the object abstraction.

We perform three major modifications of a version of FaRM, which we obtained from the authors: (i) FaRM was designed with the core assumption that the clients would have direct access to the RDMA fabric; instead we augment FaRM with a TCP/IP network front-end that receives client requests, processes them in a run-to-completion manner using the FaRM framework, and sends back the replies; (ii) out of necessity we ported FaRM to Linux and its RDMA stack—OFED; (iii) we ported FaRM from the standard RDMA interface to use the low-overhead soNUMA operations.

The coordinator maintains a distributed hash table ring mapping key hash ranges to servers and a map associating each server with its own RackOut unit. This information is available to both clients and servers. Client nodes use the ring to route requests directly to the appropriate server node (for writes) or RackOut unit (for reads). Server nodes rely on the hash table ring to ensure the integrity of the KVS.

We implement the `RackOut_adaptive` scheduling mechanism in RO-KVS. The implementation is contained within the hash table module of RO-KVS and includes modifications to both the client and server portions of the code. We extend the server code with extra variables to keep track of the metrics that are necessary to create and adjust the per-rack read probability distributions. The client code uses the RO-KVS's RPC mechanism to retrieve these variables from the servers. We set the polling interval to 10s, which we find sufficient given that key distributions in general do not change frequently [38]. When a coordinator client receives the updated variables, it adjusts its per-rack read distribution accordingly using one or both of the algorithms described in the previous section. The read distributions are regularly sampled upon new key-value read requests.

The monitor computes the average service time for a server as a simple moving average of the N most recent service times. In our RO-KVS environment, we identify remote accesses and background processes as the main sources of service time variability. RO-KVS itself does not perform periodic activities (e.g., no garbage collection), and it is an in-memory key-value store that does not use disks. Nevertheless, our `RackOut_adaptive` mechanism (Algorithm 2, in particular) accounts for even the slightest variations in service times, such as those caused by remote accesses.

6.2 Experimental Methodology

We evaluate RO-KVS on two platforms: RDMA and soNUMA. We use the former to measure the latencies of basic RO-KVS operations on existing hardware, used to instrument the model, as

described in Section 4.2. We use the latter to measure the throughput of RO-KVS under SLO tail latency constraints for different RackOut configurations and compare the results to the RackOut queuing model.

Our RDMA setup comprises six Intel Xeon E5-based servers running at 2.4GHz, each featuring 128GB of DRAM and a Mellanox ConnectX-3 Pro adapter. The adapters connect the servers via Converged Ethernet (RoCE), allowing them to access each other's memory using standard RDMA verbs. We use this setup to measure the throughput of RO-KVS performing GET and PUT operations that drive the elementary CREW operations: local read (LR), remote read (RR), and local write (LW).

Table 1 shows the average service times measured on our RDMA platform, as well as the client propagation delay, which is a constant offset used to compare latency measurements done from a client machine. These obtained service times are used to size the queuing model for Figures 7, 8, 11, and 10. Table 1 also shows projected service times for soNUMA, which does not currently exist in hardware. We derive the remote read service time (RR) for RO-KVS on a soNUMA platform to be 5.8μs by subtracting the bare latency of a remote read operation on RDMA (measured to be 2.8μs in our RDMA setup for a 512-byte hashtable bucket) from the measured RDMA RR latency (8.4μs), plus the soNUMA remote read latency (~240ns) [17]. For that projected RR latency, the resulting RR/LR ratio for soNUMA is 1.16.

With a maximum per-core throughput of 200,000 local lookups per second (5μs service time), RO-KVS performs worse than memcached (service time of about 2.5μs). The overhead in RO-KVS is inherent to the event-based polling model, where each KVS thread manages distributed state, polls on timers, completion queues, and network events, and so on, making processing more complex than in other, non-RDMA KVS, such as memcached. Lower service times could potentially be achieved through a user-level network stack, which would reduce the network processing overhead that is due to client–server communication [8]. Nevertheless, we project the RR/LR service time ratios for a hypothetical RackOut KVS with an LR of 2.5μs to be 1.2 for soNUMA and 2.5 for RDMA. With an RR/LR of 1.2, RackOut provides a significant performance boost, as is shown in Figure 10. The ratio of 2.5 for RDMA would make less impact, but the performance of RDMA networks constantly improves (e.g., through better signaling), and faster RDMA alternatives are likely to emerge in the near future [33].

Figure 14 describes the emulation platform for the soNUMA architecture. This platform [56] is designed to (i) run server nodes at regular wall-clock speed and (ii) approximate the latency and bandwidth of the fabric. The emulation platform relies on hardware virtualization to create a RackOut unit of up to 16 nodes. Each node comprises a dedicated CPU, dedicated NIC for client-facing traffic (exposed via PCI SR-IOV), and an RMC, implemented on a dedicated CPU. We fine-tune the remote access latency exposed by the RMC to match the RR/LR ratio of 1.16 (Table 1).

The load generators use the RO-KVS client library to run a YCSB workload [15]. The client library uses SpookyHash [41], a public domain hash function that produces well-balanced hash values, to map keys to servers. We run the load generators on up to nine distinct external servers. One server runs a closed-loop YCSB process issuing synchronous read/insert/update requests to the RO-KVS nodes for the purpose of measuring the latency. Another eight servers run 128 throughput YCSB generators in total, whose purpose is to put a specific load on the system under evaluation. Each generator issues up to 8 concurrent operations, each on a separate TCP/IP connection.

6.3 Validation of the Queuing Model

We use our soNUMA platform to evaluate RO-KVS for a workload that follows the key distribution highlighted in Figure 8. Although the system manages a distributed hash table ring for 512 servers, the clients only issue requests to the same group of 16 servers with the most traffic (i.e., the "hottest

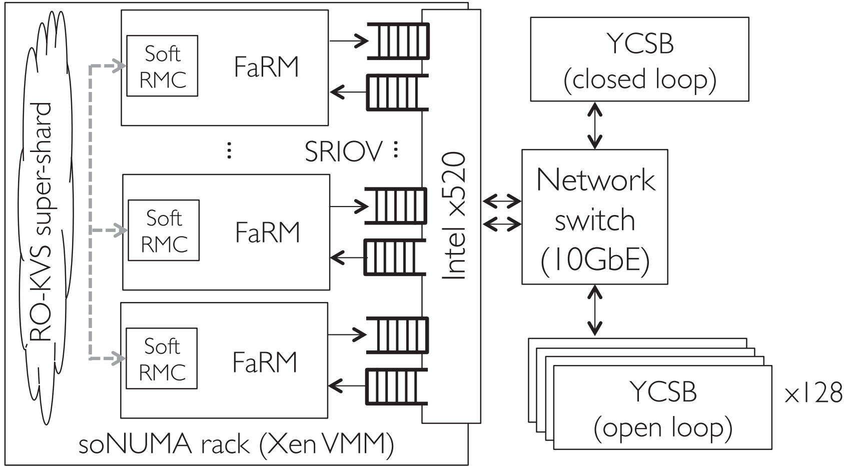


Fig. 14. Experimental setup for RackOut evaluation.

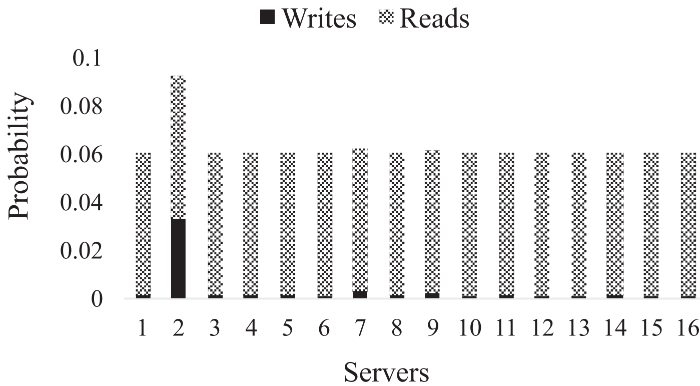


Fig. 15. Combined read/write probability distribution for the hottest RackOut unit in `RackOut_static`.

rack"). These 16 servers are organized in 16/*GF* RackOut units for the various experiments. We report the tail latency of requests issued to that 16-server group. For comparison purposes, we extract from the queuing model the tail latency for the same group of servers. Because the queuing model assumes `RackOut_static`, we configure the RO-KVS clients to perform random scheduling of requests. Figure 15 shows the combined read/write distribution for the hottest GF=16 RackOut unit in our configuration and a workload with 5% of writes. Server 2 saturates first because it hosts highly popular keys, and `RackOut_static` assumes read-dominated workloads where writes have little impact on imbalance.

Figure 16 shows the 99th percentile latency for the hottest rack in the datacenter, with throughput expressed as a fraction of the maximum processing capacity of 16 nodes. We compare the experimental results with the behavior predicted by the RackOut queuing model configured with the parameters for soNUMA from Table 1. We do not deploy our dynamic replication algorithm as the experimental setup is limited to a single rack.

The platform's behavior is closely predicted by the model, with the tail latency spiking as the server saturates. For the YCSB-C read-only workload (Figure 16(a)), each node observes the same arrival rate, but the node with the least popular keys issues mostly remote read requests, which are

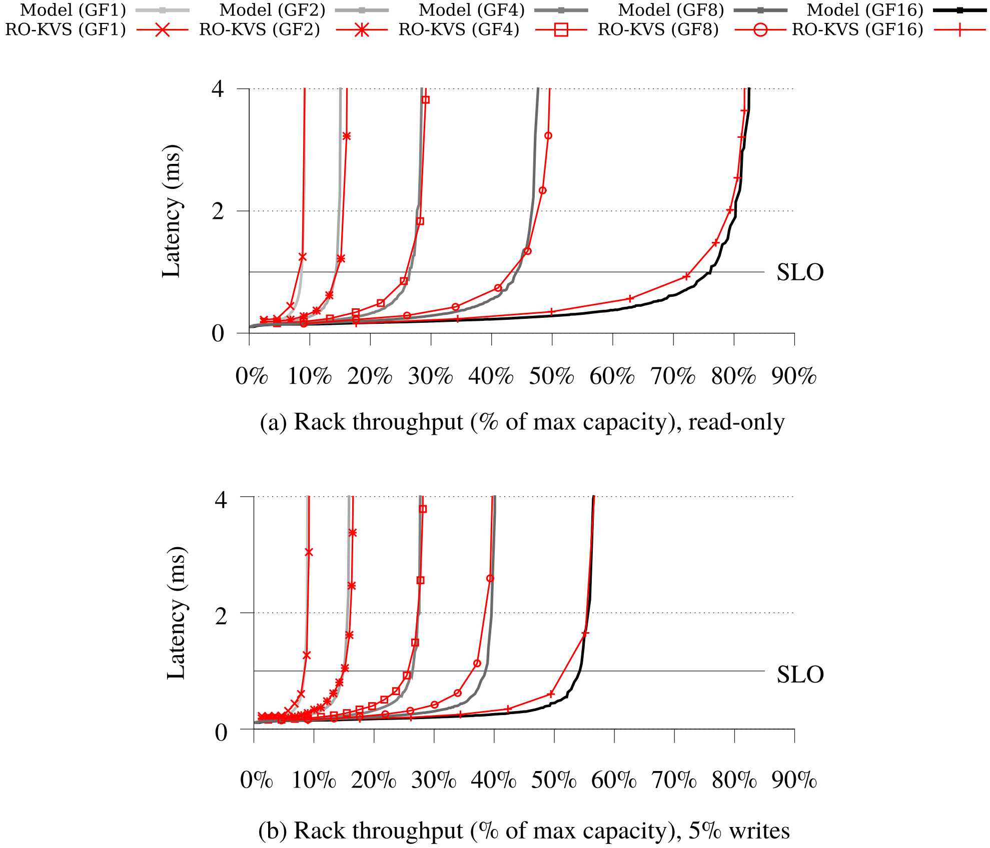


Fig. 16. Ninety-ninth percentile latency vs. throughput for the hottest rack measured on the experimental platform.

more expensive than local memory accesses. For YCSB-B workload with 5% writes (Figure 16(b)), the inability to load-balance writes limits the maximum rack throughput with GF=16 to 57% vs. 84% in the read-only workload. A workload with 20% of writes reduces the speedup from 6× for YCSB-B to 3.2×, and YCSB-A (50% writes) reduces it to 1.7×. However, workloads with atypically high fraction of writes are rare [7, 10, 15, 62]. We observe a difference below 6% between the model and the platform at 1ms SLO. The difference is likely due to contention within the emulation platform that is not captured by the model. Despite such system complexity of the RackOut platform—in terms of the application itself, the robust FaRM framework, the soNUMA fabric, and the underlying emulation platform—the queuing model provides a solid approximation of the behavior of the actual system, including the tail latency behavior.

Overall, the RackOut queuing model serves as a useful tool to analyze the impact of skew and skew mitigation techniques on KVS. Furthermore, it enables us to predict the performance improvement with high accuracy for arbitrarily large datacenter configurations and RackOut organizations with larger GFs, which exceed our platform's scale limitations.

6.4 Impact of Adaptive Load Balancing on Throughput

To emphasize the benefits of `RackOut_adaptive` (ARO), we focus on read-write workloads (≥5% writes) and the GF=16 RackOut configuration. We start with the same workload as in the

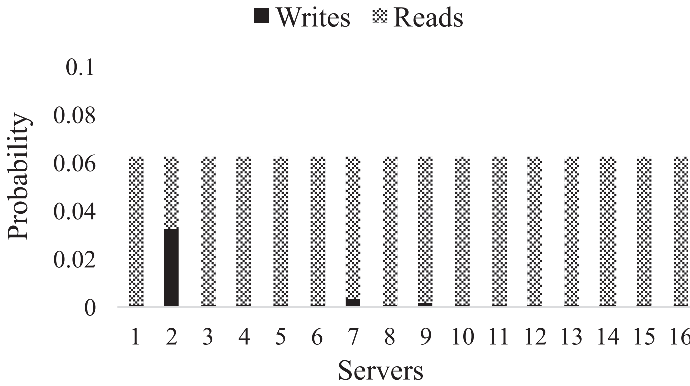


Fig. 17. Combined read/write probability distribution for the hottest RackOut unit in RackOut_adaptive (Algorithm 1).

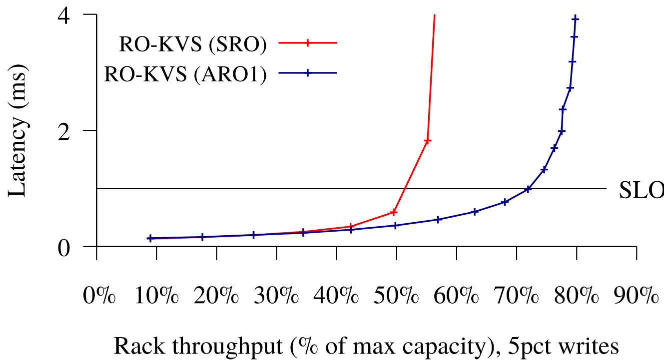


Fig. 18. RackOut_static vs. RackOut_adaptive (Algorithm 1), 99th-pct latency for 5% of writes, GF=16.

previous experiments (5% of writes). We first study how Algorithm 1 (RO-KVS (ARO1)) creates the read probability distribution for the hottest RackOut unit based on writes. We then look at how Algorithm 1 (RO-KVS (ARO1)) impacts the 99th-percentile latency and improves the speedup as we vary the fraction of writes from 0% to 20%.

Figure 15 shows the combined read/write probability distribution for RackOut_static (SRO) for the hottest 16-server RackOut unit. Server 2 is the hottest server in the rack and, thus, is the first to saturate as we drive the input load. To improve the throughput without premature SLO violation, Algorithm 1 generates a read distribution such that each server within the hottest RackOut unit processes a similar aggregate amount of read and write requests (Figure 17). Algorithm 1 assumes similar processing rates across the servers and across different operations (LR, RR, LW).

The impact of RackOut_adaptive is illustrated on Figure 18, where Algorithm 1 (RO-KVS (ARO1)) outperforms RackOut_static and reaches almost 80% of rack utilization, similarly to RackOut_static scheduling and a read-only workload (Figure 16(a)); this is expected, since in both cases the load is distributed evenly across the rack. RackOut_adaptive achieves a speedup of 1.36× over RackOut_static for the same 95/5 workload. The improvement comes from making the read distribution inversely proportional to the write distribution, as illustrated on Figure 17. Unlike RackOut_static scheduling, where a single server saturates and drives the 99th-percentile latency above the SLO, with RackOut_adaptive, all the servers of the hottest RackOut unit saturate concurrently, utilizing the servers uniformly and pushing the aggregate

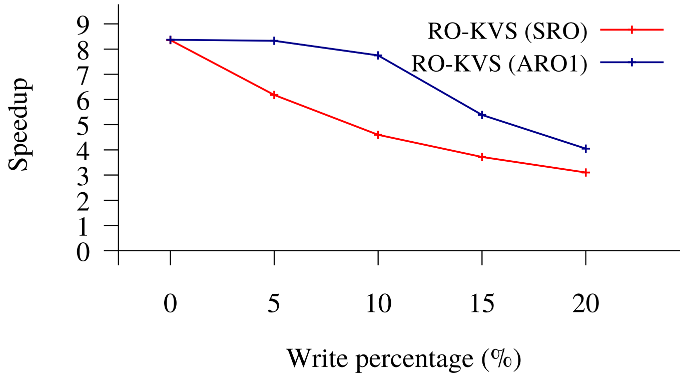


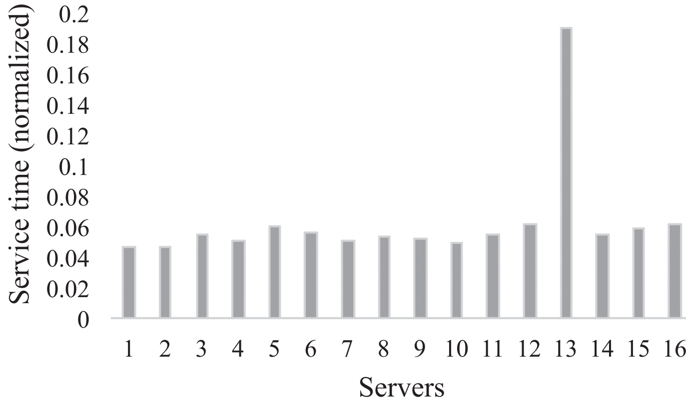
Fig. 19. Speedup at saturation for different write percentages, GF=16.

throughput (Figure 18). The behavior of RO-KVS under `RackOut_adaptive` scheduling is as expected; that is, the latency increases as the load reaches the maximum aggregate processing rate. We observe that for workloads with more than 10% of writes and GF=16, it is not possible to achieve uniform utilization of the rack's servers, simply because the probability of selecting the hottest server for writes is above $\frac{1}{GF}$ (Figure 19). The last code segment of Algorithm 1 handles this special case. For such workloads, `RackOut_adaptive` behaves similarly to a design in which reads are always routed to any node in the rack other than the owner.

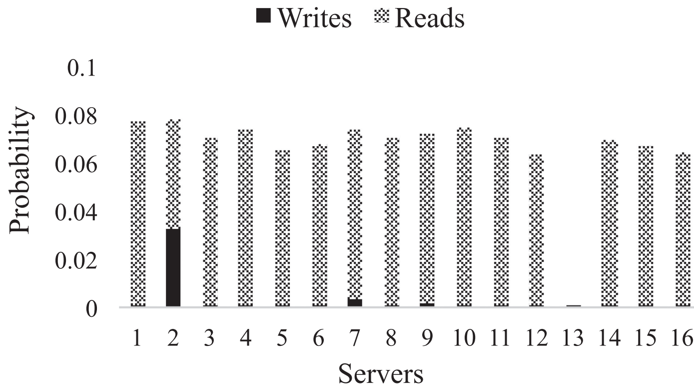
Figure 19 shows the speedups for different write percentages. 0% is a read-only workload where both scheduling techniques achieve similar performance. As we increase the write percentage, the speedup decreases. We observe that the standard `RackOut_static` scheduling with uniform read distribution performs worse than `RackOut_adaptive`, because the hottest server of the rack saturates faster; it handles both read and write load, whereas with `RackOut_adaptive` the hottest server is unballasted from the read load at the expense of handling a higher read load on the other servers. In `RackOut_static`, the larger the fraction of write requests, the faster the system saturates and the smaller the speedup due to CREW. Similarly, with `RackOut_adaptive`, the larger the fraction of writes, the harder it is to utilize the servers of a RackOut unit uniformly. In particular, we observe that for read-write workloads with more than 10% of writes, the speedup deteriorates even with adaptive load balancing, because write operations start dominating and CREW becomes the main performance obstacle.

Finally, to evaluate Algorithm 2, we inject interference into one server and study how the scheduler adjusts the read distribution accordingly. We look at the service times that the clients periodically obtain from the monitors. Figure 20(a) shows the service times normalized to the 0–1 range, where Server 13 appears much slower as compared to the rest of the servers. Thus, Algorithm 2 further adjusts the read distribution to reflect the skewed service times, as illustrated in Figure 20(b). Server 13 is assigned with zero read probability because of the high average service time.

Figure 21 illustrates the combined impact of Algorithm 1 and Algorithm 2 (`RO-KVS (ARO1+2)`) on the 99th-percentile latency and throughput. First, by injecting interference on one server, with `RO-KVS (ARO1)` the throughput decreases by 30% and is only 1.03× higher than the throughput achieved using `RO-KVS (SRO)` without injected interference. `RO-KVS (ARO1+2)` combines Algorithm 1 and Algorithm 2 to deal with both write distribution skew and variable service times. In our experiment, `RO-KVS (ARO1+2)` adjusts the read distribution as illustrated on Figure 20(b), boosting the rack's utilization to 65% at saturation.



(a) Average service times. Interference injected on server 13.



(b) Combined read/write probability distribution in `RackOut_adaptive`.

Fig. 20. Service times of a GF=16 RackOut unit with injected interference on one server, and the corresponding access distribution computed using Algorithm 1 and Algorithm 2 of `RackOut_adaptive`.

7 DISCUSSION

Using RPC for load balancing: We rely on the assumption that two different protocols with different strengths and weaknesses can co-exist. Conventional TCP/IP requires costly CPU processing, which, as we find in this work, can introduce hotspots. However, the protocol is inherently scalable and offers global connectivity. The second protocol is a lightweight user-level alternative, such as RDMA or an RDMA-like variant (e.g., soNUMA), which introduces significantly lower processing overhead but may impose stricter scalability boundaries. We combine the strengths and weaknesses of the two different networks to decouple protocol processing from data location while maintaining scalability. In particular, the key benefit of RackOut is the ability to uniformly spread network processing across a rack and mitigate the load imbalance that is inherent to the key popularity distribution. In this article, we rely on the one-sided operations of RDMA and soNUMA to showcase the strengths of such an architecture, but we could have used a lightweight RPC mechanism, which is an alternative way to retrieve and service remote data for the purpose

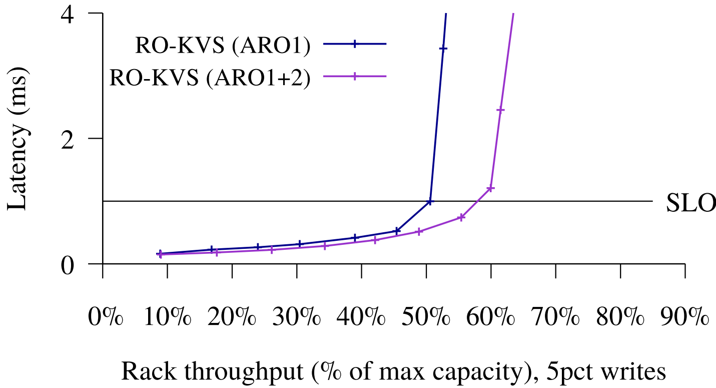


Fig. 21. `RackOut_adaptive` (Algorithm 1) vs `RackOut_adaptive` (Algorithm 1 and Algorithm 2), 99th-pct latency for 5% of writes, injected interference on one node.

of load balancing. Such a design would be less performant, as the scalability of RackOut is highly dependent on the latency and maximum IOPS of the secondary network. To show that, we adapt FaSST [42] on our RDMA testbed and measure the maximum RPC rate and the maximum remote read rate. FaSST is an RDMA-based KVS that uses RPC rather than one-sided operations to perform KV lookups, as well as other KVS operations. We measure the maximum RPC rate of 23.7 millions of requests per second (Mrps), whereas with remote reads we achieve 38.3Mrps. That said, when multiple nodes within the rack start reading a hot item, the node that hosts the hot item will sustain a higher request rate if the requesters use one-sided reads rather than RPCs. This further implies that using one-sided reads in RackOut supports larger units (i.e., GF). In reality, the RPC rate may be even lower if there is other code running on the CPU. For example, in RackOut all the nodes process incoming network packets associated with client requests. Thus, the hot node would have to process client requests, in addition to serving requests for the hot item from the other nodes within the rack. Because RackOut is CPU-bound, our measured RPC rate of 23.7Mrps would be even lower if any other form of processing on the receiver node was taken into consideration.

Network-level interference in adaptive load balancing: In our adaptive load balancing study, we focused on CPU contention and how RackOut can help avoid the contended node through remote reads. Besides the CPU, the secondary RDMA/soNUMA network itself could be contended because of a long running remote transfer streaming large amounts of data out of the server. We designed an experiment in which a node issues a long-running remote transfer to another node. At the same time, the destination node serves regular, key-value remote reads coming from the nodes in the same rack. We measure the latency of such requests on RDMA and conclude that the loaded latency (i.e., when the long-running transfer is active) is 4.4μs, as compared to the unloaded latency of 2.2μs. Such modest degradation in latency is because the latest generations of Mellanox HCAs have significantly better support for QoS.

Multi-get queries: The RackOut approach focuses on load balancing in classical key-value stores. For other applications with leaf-aggregator access patterns, RackOut can help reduce the network round-trip overhead; the receiver node pulls the requested set of objects concurrently using the secondary fabric and replies to the client. In applications exhibiting the leaf-aggregator pattern, skew is less of a problem, as each request requires accessing potentially multiple servers. Nevertheless, the additional network communication overhead can lead to more delay, and RackOut can help mitigate that problem.

8 RELATED WORK

Resource pooling: RackOut leverages fast remote access within a rack to tackle shard-skew through memory pooling. Multi-socket shared memory machines (ccNUMA) also offer this feature but have known scalability limitations. Disaggregated memory [46] introduces dedicated memory blades to increase the memory-to-compute capacity ratio and enable sharing between servers but does not consider fast remote memory access to combine memory chunks into a larger memory pool. Remote regions introduced memory pooling through the mmap interface, which is convenient but provides low performance [2]. Finally, the benefits of resource pooling powered by rack-level RDMA solutions are also leveraged in commercial database and storage solutions [23, 59].

Transactional RDMA-based data stores: One-sided reads have been first championed in FaRM [22], which we used as a base for RO-KVS. An alternative approach is the use of two-sided (RPC) RDMA communication for the purpose of performing remote data lookups [42]. Unlike RackOut, which focuses on key-value stores, both FaRM and RPC-based data stores provide serializable transactions. FaRM introduces the hopscotch data structure to enforce a single roundtrip in the common case, while RPC-based systems support regular hash tables. More recently, DrTM+H [72] proposed using both one-sided reads and RPCs for different phases of the two-phase commit protocol. As we pointed out earlier, RackOut could use RPCs instead of one-sided reads. While RPCs would enable more general computation on the receiver, they would limit the impact of RackOut.

Concurrency models in KVS: The concurrency model critically impacts performance. State-of-the-art KVS implementations represent a compromise between maximum theoretical performance and implementation complexity. Specifically, (i) concurrent-read/concurrent-write (CRCW): the memory is managed as a single pool, which can be concurrently accessed for reading and writing by any thread running on the server. This is the case of a single-instance deployment of memcached; (ii) exclusive-read/exclusive-write (EREW): The memory is managed as N distinct pools. This is typical for multi-instance deployments of memcached; (iii) CREW specifically has been proven to provide solid scalability at low complexity. MICA [45] shows that CREW delivers scalable performance for read-dominated workloads, circumventing the complexity and synchronization overhead of CRCW.

Most RDMA-based distributed KVS systems also avoid the complexity of CRCW. RamCloud [60] is based on EREW, while FaRM [22] and Pilaf [53] implement CREW, where reads are direct one-sided accesses to remote memory, while writes are transformed into RPCs. To our knowledge, DrTM [73] is the only RDMA-based distributed KVS system that implements CRCW by building a sophisticated concurrency mechanism that relies on HTM. Our RO-KVS is based on a modified version of FaRM for soNUMA that uses TCP/IP to receive and reply to client requests.

Replication: Replication is the common remedy for load imbalance in scale-out environments. Static replication [69] is a simple technique providing robustness to skew but incurs fixed increased memory requirements and is not flexible to skew changes. Dynamic replication effectively addresses these limitations but has intrinsic CPU, memory and network overheads [38, 63]. RackOut is synergistic with dynamic replication and substantially reduces the need to dynamically replicate content. Fan et al. [28] observe that the load imbalance introduced by highly popular data items can be turned into an opportunity by exploiting temporal locality using software caching techniques at the front-end (client). Our work focuses on the back-end without assuming front-end caching steering clear of the consistency issues.

Adaptive load balancing: Adaptive load balancing techniques have been proposed in prior work. The Cell work [54], for example, proposes using RDMA to process requests on the client side,

without involving the contented CPU of the server. When, instead, the NIC becomes a bottleneck, Cell relies on the CPU to process requests on the server side. Unlike Cell, the RackOut KVS does not assume a globally accessible RDMA fabric (for scalability reasons [35, 74]) but instead relies on regular Ethernet for client-server communication. In `RackOut_adaptive`, the global knowledge about load imbalance is readily available to each client through the coordinators. Based on that knowledge, a client chooses a server from the target RackOut unit to process the request. LARD [61] distributes incoming requests across the back-end nodes to achieve high cache locality. In skewed data serving workloads, good locality on the backend servers is a given. In fact, in all of our GF=1 (scale-out) experiments, because of the skewed access distributions, there is more reuse on the hot nodes compared to the other, GF>1 deployments. RackOut's greatest strength lies in relieving the hottest backend nodes when they become CPU bottlenecked.

9 CONCLUSION

The recent evolution of datacenters points to a steady increase in the overall size of the deployment, and to a standardization of the compute infrastructure at the rack level, with each rack a unit of purchase, operation, IP routing, and possibly failure domain. With RackOut, we advocate for augmenting this building block with a NUMA-style internal fabric and a fast one-sided read primitive to enable memory pooling at the rack level. The RackOut model quantifies the scalability benefits of the internal fabric as a function of key popularity distribution, the number of nodes within each rack, the number of racks, and input load. We study the benefits of RackOut when serving datasets that follow a power-law distribution and show, both through simulation and with a proof-of-concept prototype, that the approach can provide substantial benefits over the scale-out baseline and that it is synergistic with dynamic replication of micro-shards. We propose two scheduling mechanisms for RackOut: `RackOut_static`, an efficient, stateless mechanism for workloads dominated by read requests, and `RackOut_adaptive`, an adaptive load balancing mechanism tackling the rack-level imbalance in read-write workloads (≥5%) at the expense of additional messaging and computational overheads.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their helpful comments and feedback.

REFERENCES

[1] 2015. AMD High-Bandwidth Memory (HBM). Retrieved from https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: A simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 775–787.

[3] Amazon. 2011. Amazon Elasticache. Retrieved from http://aws.amazon.com/elasticache/.

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2011. FAWN: A fast array of wimpy nodes.*Commun. ACM* 54, 7 (2011), 101–109. DOI:https://doi.org/10.1145/1965724.1965747

[5] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A database benchmark based on the Facebook social graph. In *Proceedings of the SIGMOD Conference*. 1185–1196. DOI:https://doi.org/10.1145/2463676.2465296

[6] Krste Asanovic. 2014. Firebox: A hardware building block for 2020 warehouse-scale computers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'14)*.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 53–64. DOI:https://doi.org/10.1145/2254756.2254766

[8] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2017. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.* 34, 4 (2017), 11:1–11:39. DOI: https://doi.org/10.1145/2997641

[9] Eric A. Brewer. 2010. A certain freedom: Thoughts on the CAP theorem. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC'10)*. 335. DOI: https://doi.org/10.1145/1835698.1835701

[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)*. 49–60.

[11] Michael Burrows. 2006. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI'06)*. 335–350.

[12] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. 143–157. DOI: https://doi.org/10.1145/2043556.2043571

[13] Cavium Networks. 2014. Cavium Announces Availability of ThunderX™ Industry's First 48 Core Family of ARMv8 Workload Optimized Processors for Next Generation Data Center & Cloud Infrastructure. Retrieved from http://www.cavium.com/newsevents-Cavium-Announces-Availability-of-ThunderX.html.

[14] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue B. Moon. 2007. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Workshop on Internet Measurement (IMC'07)*. 1–14. DOI: https://doi.org/10.1145/1298306.1298309

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SOCC'10)*. 143–154. DOI: https://doi.org/10.1145/1807128.1807152

[16] Carlos Cunha, Azer Bestavros, and Mark Crovella. 1995. *Characteristics of WWW Client-based Traces*. Technical Report. Boston, MA.

[17] Alexandros Daglis, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2015. Manycore network interfaces for in-memory rack-scale computing. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15)*. 567–579. DOI: https://doi.org/10.1145/2749469.2750415

[18] Alexandros Daglis, Dmitrii Ustiugov, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. SABRes: Atomic object reads for in-memory rack-scale computing. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 6:1–6:13.

[19] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: Invited talk. In *Proceedings of the 2nd International Conference on Web Search and Web Data Mining (WSDM'09)*. 1. DOI: https://doi.org/10.1145/1498759.1498761

[20] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. DOI: https://doi.org/10.1145/2408776.2408794

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*. 205–220. DOI: https://doi.org/10.1145/1294261.1294281

[22] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.

[23] EMC Isilon. 2015. Isilon Scale-Out Storage Data Sheet. Retrieved from www.emc.com/collateral/software/data-sheet/h10541-ds-isilon-platform.pdf.

[24] EZchip Semiconductor Ltd. 2015. EZchip Introduces TILE-Mx100 World's Highest Core-Count ARM Processor Optimized for High-Performance Networking Applications. *Press release*. Retrieved from http://www.tilera.com/News/PressRelease/?ezchip=97.

[25] Facebook. 2008. Apache Cassandra. Retrieved from http://cassandra.apache.org/.

[26] Facebook. 2015. Introducing "Yosemite": The First Open Source Modular Chassis for High-powered Microservers. Retrieved from https://code.facebook.com/posts/1616052405274961/introducing-yosemite-the-first-open-source-modular-chassis-for-high-powered-microservers-/.

[27] Facebook. 2017. Data Sharing on Traffic Pattern Inside Facebook's Datacenter Network. Retrieved from https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/.

[28] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2011 ACM Symposium on Cloud Computing (SOCC'11)*. 23. DOI: https://doi.org/10.1145/2038916.2038939

[29] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan S. Milojicic. 2015. Beyond processor-centric operating systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS-XV)*.

[30] Michael Ferdman, Almutaz Adileh, Yusuf Onur Koçberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*. 37–48. DOI: https://doi.org/10.1145/2150976.2150982

[31] Brad Fitzpatrick. 2003. Memcached. Retrieved from http://memcached.org/.

[32] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-Out ccNUMA: Exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*. DOI: https://doi.org/10.1145/3190508.3190550

[33] GigaIO. 2018. GigaIO Link Express. Retrieved from http://gigaio.com.

[34] Linley Group. 2015. Oracle shrink sparc M7. *Microprocess. Rep.* 29, 9 (2015), 28–31.

[35] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference*. 202–215. DOI: https://doi.org/10.1145/2934872.2934908

[36] Hewlett-Packard Development Company. 2014. HP Moonshot System Family Guide. Retrieved from http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=4AA4-6076ENW.

[37] Yu-Ju Hong and Mithuna Thottethodi. 2013. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 2013 ACM Symposium on Cloud Computing (SOCC'13)*. 13:1–13:17. DOI: https://doi.org/10.1145/2523616.2525970

[38] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*. 8:1–8:7. DOI: https://doi.org/10.1145/2670518.2673882

[39] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC'10)*.

[40] Jinho Hwang and Timothy Wood. 2013. Adaptive performance-aware distributed memory caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*. 33–43.

[41] Bob Jenkins. 2011. SpookyHash: A 128-bit Noncryptographic Hash. Retrieved from http://burtleburtle.net/bob/hash/spooky.html.

[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI'16)*. 185–201.

[43] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. DOI: https://doi.org/10.1145/279227.279229

[44] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15)*. 476–488. DOI: https://doi.org/10.1145/2749469.2750416

[45] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI'14)*. 429–444.

[46] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. 267–278. DOI: https://doi.org/10.1145/1555754.1555789

[47] LinkedIn. 2009. How Linkedin Uses Memcached. Retrieved from http://www.oracle.com/technetwork/server-storage/ts-4696-159286.pdf.

[48] LinkedIn. 2009. Project Voldemort. Retrieved from http://www.project-voldemort.com/.

[49] Linley Group. 2014. X-Gene 2 aims above microservers. *Microprocess. Rep.* 28, 9 (Sep. 2014), 20–24.

[50] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Yusuf Onur Koçberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Özer, and Babak Falsafi. 2012. Scale-out processors. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12)*. 500–511.

[51] Petar Maymounkov and David Mazières. 2002. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Conference on Peer-to-peer Systems (IPTPS'02)*. 53–65.

[52] Mellanox Corp. 2015. RDMA Aware Networks Programming User Manual, Rev 1.7. Retrieved from www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[53] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)*. 103–114.

[54] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and network in the cell distributed B-tree store. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC'16)*. 451–464.

[55] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI'13)*. 385–398.

[56] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*. 3–18. DOI: https://doi.org/10.1145/2541940.2541965

[57] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. An analysis of load imbalance in scale-out data serving. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 367–368. DOI: https://doi.org/10.1145/2896377.2901501

[58] Diego Ongaro and John K. Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 305–319.

[59] Oracle. 2012. Exalogic & Exadata: The Optimal Platform for Oracle Knowledge. Retrieved from http://www.oracle.com/us/products/applications/knowledge-management/exalogic-exadata-optl-knolg-1509222.pdf.

[60] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud storage system.*ACM Trans. Comput. Syst.* 33, 3 (2015), 7:1–7:55. DOI: https://doi.org/10.1145/2806887

[61] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. 1998. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. 205–216. DOI: https://doi.org/10.1145/291069.291048

[62] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradkar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jagadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. 2013. On brewing fresh espresso: LinkedIn's distributed data serving platform. In *Proceedings of the SIGMOD Conference*. 1135–1146. DOI: https://doi.org/10.1145/2463676.2465298

[63] Venugopalan Ramasubramanian and Emin Gün Sirer. 2004. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. 99–112.

[64] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. 2011. It's time for low latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*.

[65] Salvatore Sanfilippo. 2009. Redis. Retrieved from http://redis.io/.

[66] Navin Sharma, Sean Kenneth Barker, David E. Irwin, and Prashant J. Shenoy. 2011. Blink: Managing server clusters on intermittent power. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)*. 185–198. DOI: https://doi.org/10.1145/1950365.1950389

[67] Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: A scalable peer-to-peer lookup protocol for internet applications.*IEEE/ACM Trans. Netw.* 11, 1 (2003), 17–32.

[68] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy nodes with 10GbE: Leveraging one-sided operations in soft-RDMA to boost memcached. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*. 347–353.

[69] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. 2018. Enabling efficient RDMA-based synchronous mirroring of persistent memory transactions. arxiv:1810.09360 http://arxiv.org/abs/1810.09360.

[70] Birjodh Tiwana, Mahesh Balakrishnan, Marcos K. Aguilera, Hitesh Ballani, and Zhuoqing Morley Mao. 2010. Location, location, location!: Modeling data proximity in the cloud. In *Proceedings of the 9th ACM Workshop on Hot Topics in Networks (HotNets-IX)*. 15. DOI: https://doi.org/10.1145/1868447.1868462

[71] Twitter. 2010. Memcached SPOF Mystery. Retrieved from https://blog.twitter.com/2010/memcached-spof-mystery.

[72] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled distributed transactions: Hybrid is better!. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18).* 233–251. https://www.usenix.org/conference/osdi18/presentation/wei.

[73] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15).* 87–104. DOI: https://doi.org/10.1145/2815400.2815419

[74] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference.* 523–536. DOI: https://doi.org/10.1145/2785956.2787484

Received March 2017; revised April 2018; accepted January 2019