The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems

Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, Boris Grot*
EcoCloud, EPFL *University of Edinburgh

Abstract

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 is 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. Our results show that RackOut can increase throughput 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.

Categories and Subject Descriptors C.4 [*PERFORMANCE OF SYSTEMS*]: Modeling techniques

Keywords Rack-scale systems, RDMA, data skew

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.
SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10...$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987577

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 comprise hundreds of software components, deployed on thousands of servers organized in multiple tiers. Such applications must support high connection counts and operate with tight user-facing service-level objectives (SLO), often defined in terms of tail latency [9, 21, 46]. To meet these objectives, most such applications keep the data (e.g., a social graph) in memory-resident distributed 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. In its basic form, however, sharding data limits the maximum throughput respecting the SLO whenever the popularity distribution of data items is skewed and unknown. A large popularity skew among data items can result in severe load imbalance, as a small subset of the leaf servers will saturate either their CPUs or their NICs, 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 allowing any node within the rack to access the memory of other nodes through one-sided operations (i.e., without involving the remote CPU). Consequently, the data in RackOut is partitioned at rack-scale granularity, with the entire rack responsible for a collection of data items mapped to the rack's servers. RackOut leverages the concurrent-read/exclusive-write (CREW) data access model at rack scale, which has previously been shown to dramatically improve the scalability of single-server performance on multicore servers [23, 37, 45]. 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 each RackOut unit, while write requests are always directed to the server holding the micro-shard. Even though data is never replicated within a RackOut unit, RackOut is compatible and synergistic with dynamic replication [11, 21, 31, 32]. 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, creating replicas, and ongoing consistency maintenance.

This paper 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 towards 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 impact of rack-level aggregation on mitigating the effect of skew. We develop a queuing model for RackOut that assumes globally accessible memory within the rack. We evaluate the benefits of RackOut as a function of server count, size of the RackOut unit, and read/write ratio for datasets following a power-law popularity distribution. For a Zipfian read-only distribution with $\alpha = 0.99$, the model predicts that a RackOut deployment of 512 servers grouped into 16-server RackOut units increases throughput by $6\times$ with RDMA and $8.6\times$ with Scale-Out NUMA (soNUMA) [18, 19, 47] without violating SLO.

• We evaluate the combination of RackOut 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 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 [23] and is ported to both Mellanox RDMA [44] and soNUMA. We evaluate RO-KVS 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 $6\times$ more requests than 16 independent servers for a workload with 5% writes and $8.2\times$ more requests for a read-only workload. Discrepancies between the model and the measured system remain below 6%, which validates the model.

The rest of the paper is organized as follows: §2 motivates the problem in terms of application trends and dataset access patterns. §3 discusses the key architectural trends providing cost-effective, rack-scale memory pooling. §4 provides a detailed analysis of the RackOut queuing model, which we validate in §5 using an implementation of our proof-of-concept RO-KVS. We discuss related work in §6 and conclude in §7.

2. Background

A significant portion of important contemporary web-scale applications is latency sensitive [9, 21, 46]. Designing a datacenter-scale system that delivers tight latency guarantees for the majority of user requests is notoriously challenging [21, 22]. 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 vulnerability 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 solutions.

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 [9, 11, 22, 46]. KVS such as Memcached [2], Redis [3], Dynamo [22], TAO [11], and Voldemort [5] are used in production environments of large service providers such as Facebook, Amazon, Twitter, Zynga, and LinkedIn [6, 39, 46, 59]. The popularity of these systems has resulted in considerable research and development efforts, including open-source implementations [1], research prototypes [7, 51] and a wide range of sophisticated, highly tuned frameworks that aspire to become the state-of-the-art of KVS [23, 36, 37].

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 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 [21, 22].

2.2 Skew in scale-out architectures

KVS typically handle very large collections of data items and millions or billions of requests per second. In such a setting, skewed distributions emerge naturally, as data popularity varies greatly. Previous work has shown that data popularity distributions in real-world KVS workloads follow a power-law distribution [9, 11, 27, 32], resulting in an access frequency imbalance, commonly referred to as *skew*. This skewed distribution is accurately represented by the power-law Zipfian distribution [9, 15, 17, 27, 37, 55]. 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

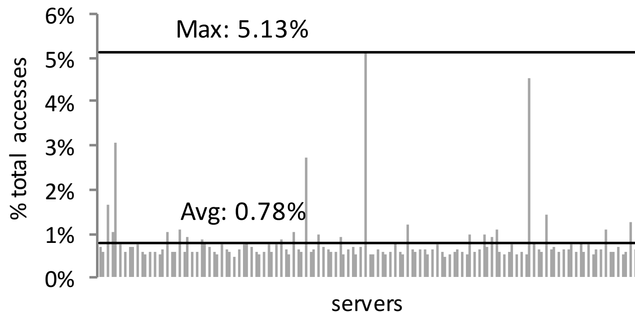


Figure 1: Grouping of 250M data items with a power-law popularity distribution in 128 servers.

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 modeled dataset. $\alpha = 0.99$ is the typical data popularity distribution used in KVS research [16, 23, 31, 36, 37]. Some studies show that the popularity distribution skew in real-world datasets can be lower than that (e.g., $\alpha = 0.6$ [55], $\alpha = 0.7-0.9$ [8, 53]), but also even higher (e.g., up to $\alpha = 1.01$ [15, 27]).

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 [21, 28]. 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., [1, 43, 56]) and each micro-shard to a server. Hash functions aim at probabilistically reducing load imbalance by evenly distributing data items to servers, but are distribution-agnostic, as data item popularities cannot be predicted in advance or controlled, and may change over time. In practice, a collection of micro-shards mapping 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 of data items is still observable as a popularity skew across shards [48].

Fig. 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. Popularity skew has significant implications as the hottest server will receive a 6.5× higher load than the average, and may saturate 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 deployment as a function of three parameters: (i) the exponent α of the dataset's power-law distribution, (ii) the number of data items comprising the dataset, and (iii) the function used to distribute data items to micro-shards. We discuss the impact of each parameter below.

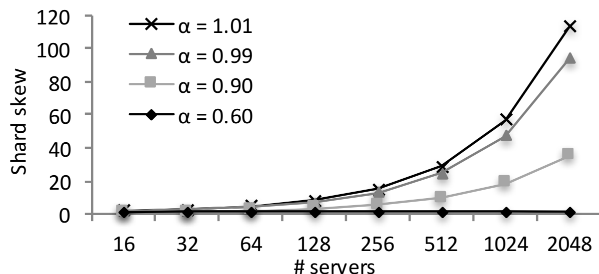
Fig. 2a shows the shard skew as the number of servers scales for different popularity distribution exponents α. While the shard skew is insensitive to scaling out for low α values (e.g., $\alpha = 0.6$), large exponents dramatically increase shard skew, which in turn becomes a performance limiter. For 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×.

Fig. 2b shows the impact of the data item population size 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.

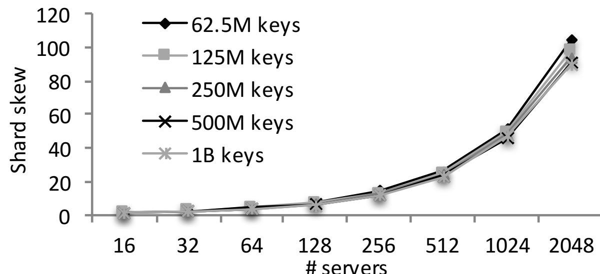
2.3 Replication: a thorny solution

Service providers are well aware of the problems that arise from skew-induced load imbalance [20, 21, 32]: servers holding 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 widely used technique that deals with such load imbalance in the datacenter. It is based on a simple concept: by replicating data N times, N servers, rather than a single server, can access the same data items, 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 [38, 51]. 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* [46]). The main principle is that the system dynamically monitors the load on a set of micro-shards, and takes replication decisions on the fly to mitigate load imbalance. Dynamic replication could oper-



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



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

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

ate on individual keys or on entire micro-shards [31, 32]. 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, timely deallocation of replicas that are no longer useful, etc. [11, 31, 32, 34].

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 [10, 12, 22, 33, 35, 49, 58]. Even in a weakly consistent KVS [22], 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 [11, 21, 31, 32]. However, dynamic replication comes with considerable overheads: the consistency semantics expected by the application, the dataset's change rate [22], the precision of the monitoring algorithm, and the micro-shard size, all critically impact the system's behavior [27, 31, 32].

3. Memory pooling at rack scale

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 in order to have fewer larger shards. As Fig. 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 (§3.1), concurrency issues (§3.2) and fault tolerance (§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 [7, 14, 25, 26, 42, 57] augmented with a rack-level RDMA fabric. This approach is used in commercial products providing analytical (e.g., Oracle ExaData / Exalogic [50]) or storage (e.g. EMC/Isilon [24]) solutions to clients connected via a conventional network. Applied-Micro's X-Gene2 server SoC [40] and Oracle's Sonoma [41] integrate the RDMA controller directly on chip, HP Moonshot [30] combines low-power processors with RDMA NICs, and research proposals further argue for on-chip support for one-sided remote access primitives [18, 47]. Building larger logical entities using such rack-scale memory

pooling approaches instead of the cache-coherent NUMA approach comes at a lower cost and complexity.

The fundamental premise for rack-scale memory pooling is that all servers within a rack can access the whole memory of the rack within a small premium over local memory, 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 [44].

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 been shown 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 [36, 37]. As most workloads are read-dominated [9, 11, 16, 52], 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 [23] and Pilaf [45] 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 data availability and durability. Indeed, scale-out applications are fundamentally designed to handle node failures [22, 51, 56]. 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, 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 [13]) and thus fault tolerance must be handled across rack-scale building blocks. Failures must also be handled within the rack to detect and report partial system failures, such as individual node failures, which would make portions of the dataset inaccessible.

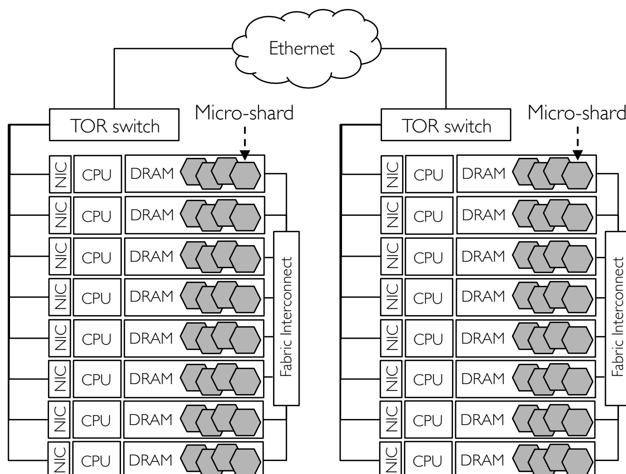


Figure 3: Two RackOut units, each with eight servers (GF=8) connected with an internal RDMA fabric. Each server hosts multiple micro-shards.

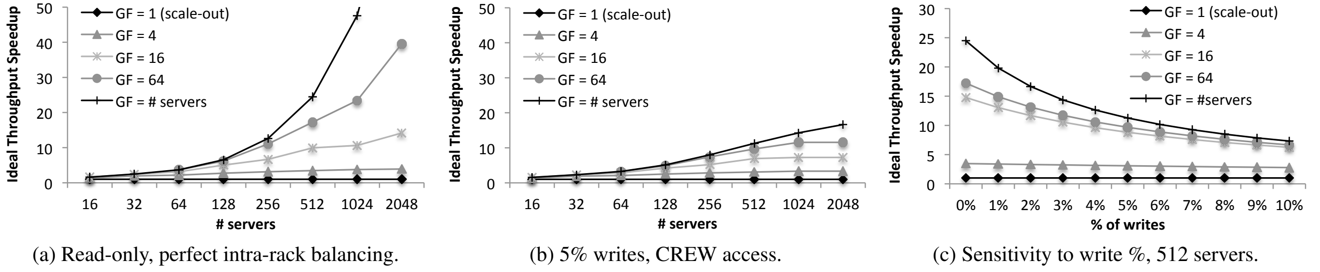
4. RackOut data serving

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 using one-sided operations. Thanks to this internal fabric, the aggregate memory of the server group is 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 rack's internal fabric and each individual server's memory bandwidth are under-subscribed while the CPU or the external-facing NIC is fully utilized.

Fig. 3 illustrates two RackOut units of eight servers each. Each RackOut unit connects the eight servers via an internal fabric which supports one-sided RDMA operations. 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* comprised of all the micro-shards within the unit.

Fig. 3 also clearly illustrates the key assumption behind 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. From an application perspective, the proposed RackOut and scale-out models are similar: clients connect to servers via the network and applications rely on replication to scale beyond the unit of capacity (i.e., rack or server) and ensure data availability.



(a) Read-only, perfect intra-rack balancing. (b) 5% writes, CREW access. (c) Sensitivity to write %, 512 servers.

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

Confining the fabric to a bounded-size unit avoids the emergent safety, performance and monitoring challenges of large-scale fabrics. For example, recent work has shown that scaling RDMA over commodity Ethernet introduces issues of congestion control, dealing with deadlocks and livelocks, and other subtleties of priority-based flow control [29, 61].

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 (§4.1), and subsequently build a queuing model for RackOut to study service time implications at the tail latency (§4.2). We analyze the sensitivity of the skew to the dataset distribution (§4.3), the synergy of RackOut with dynamic replication and migration (§4.4), and the impact of remote read penalties on performance (§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:

$$Ideal\ speedup = \frac{\frac{shard_skew_1 \times L_{avg}}{compute_1}}{\frac{rack_skew_{GF} \times GF \times L_{avg}}{compute_{GF}}} = \frac{shard_skew_1}{rack_skew_{GF}} \quad (1)$$

Given a CREW model, Eq. 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.

Fig. 4 shows the impact of GF on the rack skew. Fig. 4a shows the benefit of perfect load balancing within a rack of a given GF according to Eq. 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×. Fig. 4b 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, similar to Amdahl's law; the maximum performance improvement with GF=64 drops from 16× to 9×.

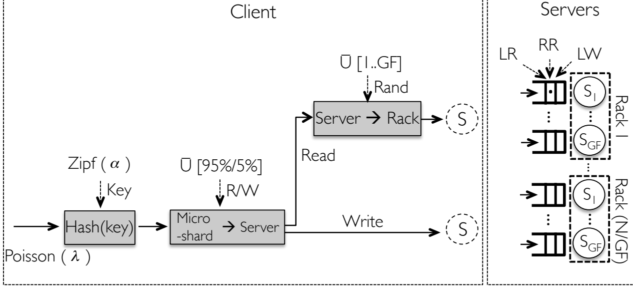


Figure 5: CREW client and server queuing model.

Fig. 4c illustrates the sensitivity to the read/write mix for various GF configurations.

4.2 A queuing model for RackOut

Eq. 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 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 ($\lambda < \mu$), as well as each server's performance under a given SLO.

Fig. 5 describes the queuing model's key elements. Requests follow an open-loop arrival process with a given rate λ and Poisson inter-arrivals. Each request carries a timestamp, a key selected randomly according to popularity, and a read/write tag selected uniformly according to the modeled 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 i, the power-law distribution of the keys, the hash function, the RackOut GF and the read/write mix together determine the per-server arrival rate for each request type t, λ_{it}:

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

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 derive from the RackOut KVS system (RO-KVS) described in §5. 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 $1.38\times$ and $1.68\times$ 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.

Fig. 6 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 compared to a uniform workload with 100% read requests on a scale-out deployment (GF=1). We 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.

Fig. 6a shows the impact of RackOut 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 $6\times$ over scale-out. This is a substantial increase in performance due to better load-balancing, even though the majority of requests will suffer the $1.68\times$ penalty associated with accessing remote memory over RDMA. In comparison, Fig. 4a's idealized model predicts the maximum speedup for the same power-law distribution of keys to be of $9.9\times$. However, the idealized model does not account for the remote memory access penalty or the requirement to meet any particular SLO.

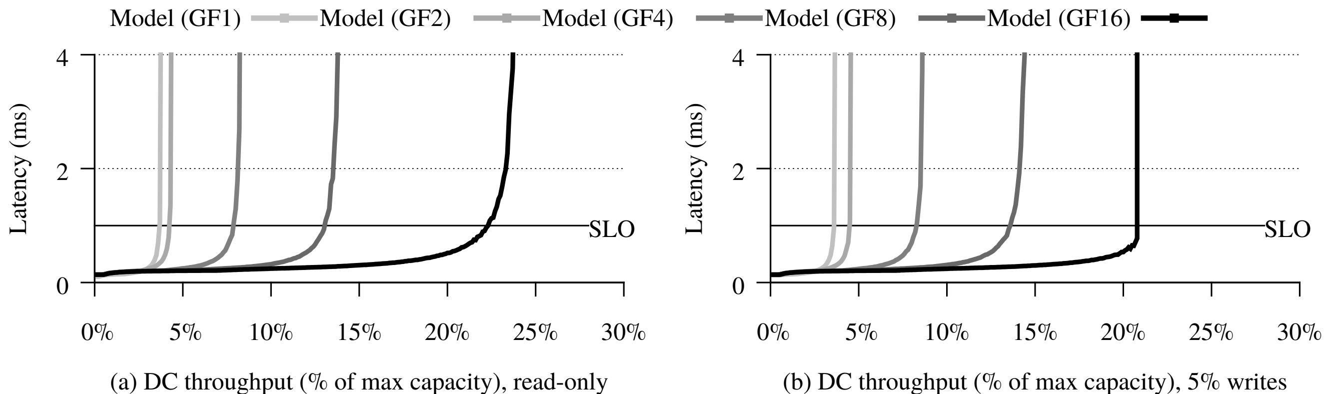


Figure 6: Datacenter-wide 99th percentile latency vs. utilization, determined using the queuing model and RDMA parameters (512 servers, Zipfian $\alpha = 0.99$).

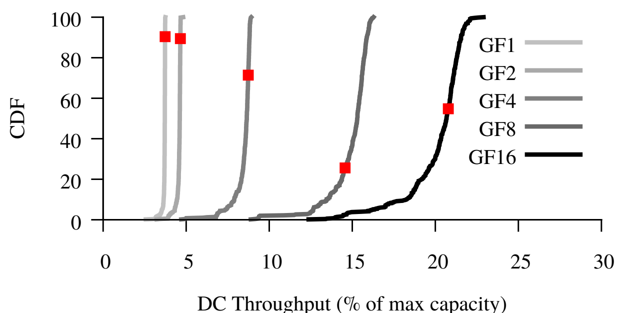


Figure 7: Datacenter utilization for 500 different datasets of 50 million objects with Zipfian $\alpha = 0.99$ data popularity distribution (5% writes). The red dots represent the key distribution used throughout this paper.

Fig. 6b 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

Fig. 6 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.

Fig. 7 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 Synergy with dynamic replication and migration

The results of the queuing model in §4.2, including Fig. 6, 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. The model extension runs a greedy algorithm operating as follows: (i) it identifies the hottest server node 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 all writes to a micro-shard eventually propagate to all of its 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 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. 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

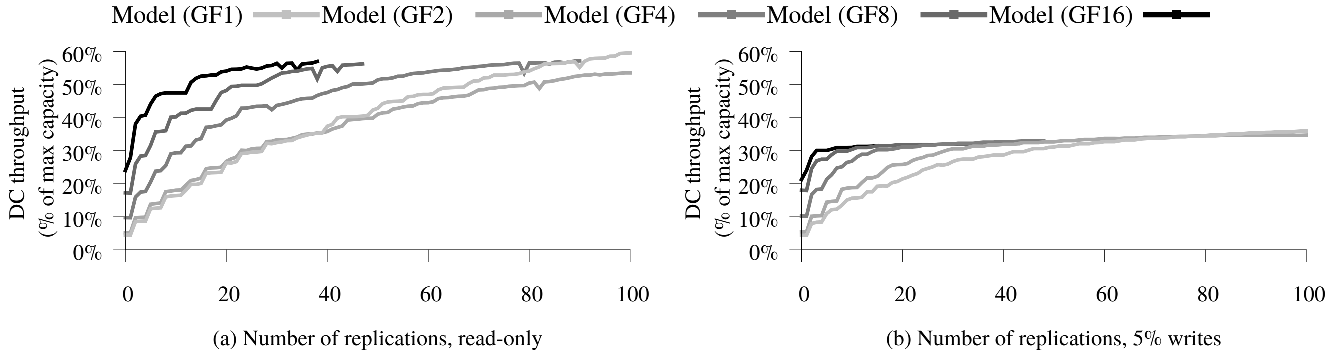


Figure 8: Dynamic replication applied to RackOut.

overhead (e.g., load monitoring) and consistency maintenance costs, and would have to rely on partial and potentially outdated metrics to make policy decisions [22, 27, 31, 32].

Fig. 8 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 [23] defaults to $\sim$100 micro-shards on a server with only 16 GB of RAM). Fig. 8a shows the trend for a read-only workload. This is a degenerate case where consistency maintenance is a non-issue. All configurations converge to a point where skew is eliminated 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.

Fig. 8b shows the trend assuming a workload with 5% writes. 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 $\sim$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 [32]. 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.

4.5 The impact of faster remote reads

The impact of node grouping 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 [54]. In addition, the increasing trend toward higher integration and low-latency fabrics will further lower the RR/LR ratio and improve the effectiveness of the RackOut approach.

A representative of such emerging tightly integrated solutions is Scale-Out NUMA (soNUMA) [18, 47], which delivers remote memory access latency 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.

Fig. 9 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 §4.2 (Fig. 6). We present only the read-only data as it is 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 soNUMA ratio extracted from a soNUMA cycle-accurate sim-

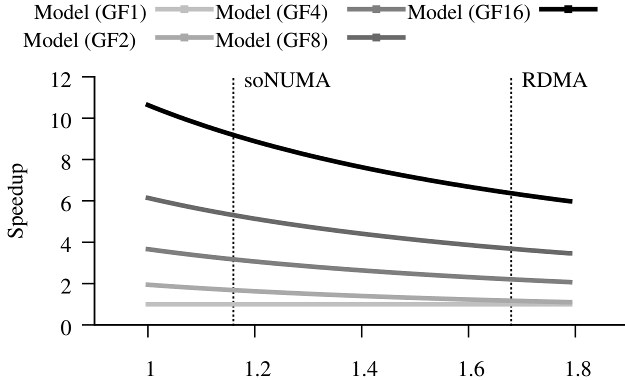


Figure 9: Speedup for different RR/LR ratio (read-only).

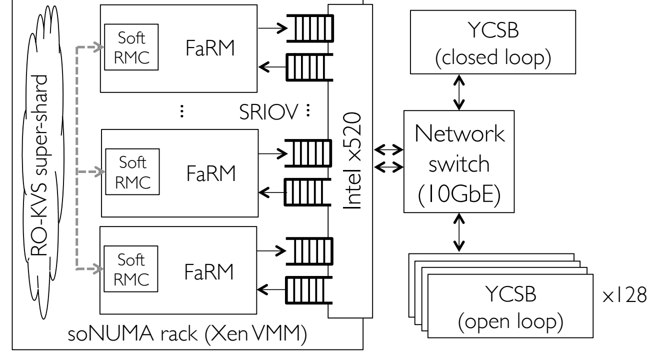


Figure 10: Experimental setup for RackOut evaluation.

ulation model [18] (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.

5. A RackOut key-value store

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

5.1 RO-KVS architecture

RO-KVS consists of (i) a coordinator node managing the key hash space; (ii) a client library and an access protocol; (iii) RackOut nodes that hold the data. RO-KVS is built on top of FaRM [23], 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 [19] that eliminate the software overheads associated with atomicity checks and data layout.

We perform three major modifications of FaRM: (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 the FaRM core 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.

5.2 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 §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 service times are used to size the queuing model for Fig. 6, 7, 8 and 9. Table 1 also shows projected service times for soNUMA,

Table 1: Average service time of basic operations used to size the queuing model.

Operation	LR	RR	LW	RR/LR	Propagation
RDMA	5μs	8.4μs	6.9μs	1.68	34.9μs
soNUMA (projected)		5.8μs		1.16	

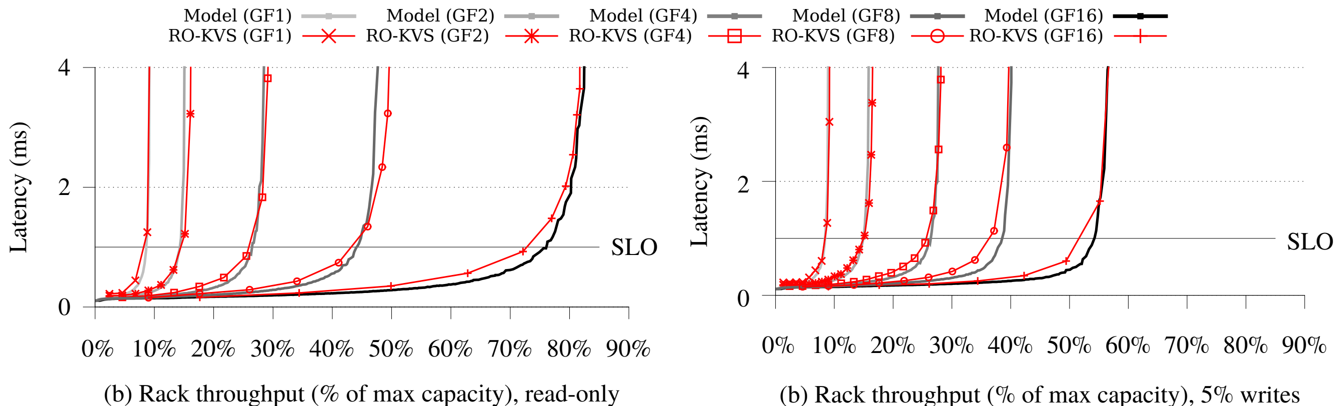


Figure 11: 99th percentile latency vs. throughput for the hottest rack measured on the experimental platform.

which does not currently exist in hardware. We derive the remote read service time for RO-KVS for soNUMA to be 5.8µs by adjusting the measured RDMA RR latency (8.4µs) by the difference between 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) and the soNUMA remote read latency (∼240ns [18]). For that projected latency, the resulting RR/LR ratio for soNUMA is 1.16.

Fig. 10 describes the emulation platform for the soNUMA architecture. This platform [47] 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 [16]. The client library uses SpookyHash [4], a public domain hash function that produces well-balanced hash values, to map keys to servers. Nine external servers generate load, and a tenth 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.

5.3 Validation of the queuing model

We use our soNUMA platform to evaluate RO-KVS for a workload that follows the key distribution highlighted in Fig. 7. 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 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 the tail latency for the same group of servers from the queuing model.

Fig. 11 shows the 99th percentile latency for the hottest rack, 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 soNUMA parameters (Table 1). We do not deploy a 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 (Fig. 11a), each node observes the same arrival rate, but the node with the least popular keys issues mostly remote read requests, which are more expensive than local memory accesses. For YCSB-B workload with 5% writes (Fig. 11b), 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 [9, 11, 16, 52]. 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 accurately predict the performance improvement for arbitrarily large datacenter configurations and RackOut organizations with larger GFs, which exceed our platform's scale limitations.

6. Related work

Resource pooling. RackOut leverages fast remote access within a rack to tackle shard-skew through memory pooling. Multi-socket shared memory machines also offer this feature, but have known scalability limitations. Disaggregated memory [38] 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. Finally, the benefits of resource pooling powered by rack-level RDMA solutions are also leveraged in commercial database and storage solutions [24, 50].

Concurrency models in KVS. 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) concurrent-read/exclusive-write (CREW) specifically has been proven to provide solid scalability at low complexity. MICA [37] 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 [51] is based on EREW, while FaRM [23] and Pilaf [45] implement CREW, where reads are direct one-sided accesses to remote memory, while writes are transformed into RPCs. To our knowledge, DrTM [60] 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 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 [32, 53]. RackOut is synergistic with dynamic replication and substantially reduces the need to dynamically replicate content. Fan et al. [27] 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.

7. 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 analytically 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.

Acknowledgements

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

References

[1] Apache Cassandra. `http://cassandra.apache.org/`.

[2] Memcached. `http://memcached.org/`.

[3] Redis. `http://redis.io/`.

[4] SpookyHash: a 128-bit noncryptographic hash. `http://burtleburtle.net/bob/hash/spooky.html`.

[5] Project Voldemort. `http://www.project-voldemort.com/`.

[6] Amazon. Amazon Elasticache. `http://aws.amazon.com/elasticache/`.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. *Commun. ACM*, 54(7):101–109, 2011.

[8] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a database benchmark based on the Facebook social graph. In *SIGMOD Conference*, pages 1185–1196, 2013.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. 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*, pages 53–64, 2012.

[10] E. A. Brewer. A certain freedom: thoughts on the CAP theorem. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2010.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[12] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–350, 2006.

[13] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

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

[15] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. B. Moon. 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)*, pages 1–14, 2007.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[17] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of www client-based traces. Technical report, Boston, MA, USA, 1995.

[18] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot. Manycore network interfaces for in-memory rack-scale computing. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 567–579, 2015.

[19] A. Daglis, D. Ustiugov, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot. SABRes: Atomic Object Reads for In-Memory Rack-Scale Computing. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[20] J. Dean. 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)*, page 1, 2009.

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

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[23] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

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

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

[26] Facebook. Introducing "Yosemite": the first open source modular chassis for high-powered microservers. `https://code.facebook.com/posts/1616052405274961/introducing-yosemite-the-first-open-source-modular-chassis-for-high-powered-microservers-/`., May 2015.

[27] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2011 ACM Symposium on Cloud Computing (SOCC)*, page 23, 2011.

[28] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. 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)*, pages 37–48, 2012.

[29] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016.

[30] Hewlett-Packard Enterprise. HP Moonshot System Family Guide. `http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=4AA4-6076ENW`., 2014.

[31] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 2013 ACM Symposium on Cloud Computing (SOCC)*, pages 13:1–13:17, 2013.

[32] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of The 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*, pages 8:1–8:7, 2014.

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

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

[35] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[36] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. 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), pages 476–488, 2015.

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

[38] K. T. Lim, J. Chang, T. N. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2009.

[39] LinkedIn. How Linkedin Uses Memcached. `http://www.oracle.com/technetwork/server-storage/ts-4696-159286.pdf`.

[40] Linley Group. X-Gene 2 Aims Above Microservers. *Microprocessor Report*, September 2014.

[41] Linley Group. Oracle Shrink Sparc M7. *Microprocessor Report*, September 2015.

[42] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, Y. O. Koçberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Özer, and B. Falsafi. Scale-out processors. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, pages 500–511, 2012.

[43] P. Maymounkov and D. Mazières. 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)*, pages 53–65, 2002.

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

[45] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 103–114, 2013.

[46] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[47] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 3–18, 2014.

[48] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. 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*, pages 367–368, 2016.

[49] D. Ongaro and J. K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

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

[51] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7, 2015.

[52] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradkar, C. Beaver, G. Brandt, M. Gandhi, K. Gopalakrishna, W. Ip, S. Jagadish, S. Lu, A. Pachev, A. Ramesh, A. Sebastian, R. Shanbhag, S. Subramaniam, Y. Sun, S. Topiwala, C. Tran, J. Westerman, and D. Zhang. On brewing fresh espresso: LinkedIn's distributed data serving platform. In *SIGMOD Conference*, pages 1135–1146, 2013.

[53] V. Ramasubramanian and E. G. Sirer. 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)*, pages 99–112, 2004.

[54] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[55] N. Sharma, S. K. Barker, D. E. Irwin, and P. J. Shenoy. 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)*, pages 185–198, 2011.

[56] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[57] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 347–353, 2012.

[58] B. Tiwana, M. Balakrishnan, M. K. Aguilera, H. Ballani, and Z. M. Mao. Location, location, location!: modeling data proximity in the cloud. In *Proceedings of The 9th ACM Workshop on Hot Topics in Networks (HotNets-IX)*, page 15, 2010.

[59] Twitter. Memcached SPOF Mystery. `https://blog.twitter.com/2010/memcached-spof-mystery`., 2010.

[60] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[61] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 523–536, 2015.