Rack-Scale Memory Pooling for Datacenters

THÈSE N^{O} 7612 (2017)

PRÉSENTÉE LE 1ER JUIN 2017

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DES SYSTÈMES DE CENTRES DE CALCUL
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Stanko NOVAKOVIC

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. E. Bugnion, Prof. B. Falsafi, directeurs de thèse
Dr D. Narayanan, rapporteur
Dr T. Harris, rapporteur
Prof. J. Larus, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

Acknowledgements

First and foremost, I would like to thank my advisor, Ed Bugnion and my co-advisor, Babak Falsafi. I feel extremely lucky to have worked with them over the past five years. Ed has a knack for identifying important research challenges and deep, insightful intuition on how to approach them. On top of that, his technical expertise is beyond compare. This unique combination of research and technical skills had a profound impact on me growing up as a researcher. Ed's fast-paced, results-driven approach to problem solving is particularly interesting and has left a huge mark on me. Ed is also a great friend, his optimism gave me the courage to focus all these years and it shaped up my future career directions. I would also like to thank Babak for his support, guidance, and commitment to my success as a Ph.D. student. Babak, along with Ed, has thought me how to identify meaningful research problems, apply the scientific method to solve them, and communicate my research results to the wider community through writing and presentation. I also thank Boris Grot and Alexandros Daglis, with whom I collaborated on the Scale-Out NUMA project. It was a true priviledge to work with them and I thank them for the numerous discussions we have had over the past few years. In addition, I would like to thank the members of my thesis committee, Willy Zwaenepoel, James Larus, Dushyanth Narayanan, and Tim Harris for their valuable feedback on my thesis draft. Dushyanth was my informal advisor and I thank him for his willingness to discuss research on many occasions. I would also like to thank the people I worked with during my internships at Microsoft Research and HP Labs, Aleksandar Dragojevic, Miguel Castro, Kim Keeton, and Paolo Faraboschi.

I thank my colleagues and fellow graduate students from DCSL and PARSA laboratories for all the moments, either fun or busy, that we shared together: Jonas Fietz, Marios Kogias, George Prekas, Mia Primorac, Dmitrii Ustiugov, Sam Whitlock, Adrien Ghosn, Lisa Zhou, Margaret Es-

candari, Djordje Jevdjic, Onur Kocberber, Stavros Volos, Cansu Kaynak, Pejman Lotfi-Kamran, Evangelos Vlachos, Sotiria Fytraki, Javier Picorel, Hussein Kassir, Mario Drumond, Nooshin Mirzadeh, Arash Pourhabibi, Mark Sutherland, and Stephanie Baillargues. I would also like to thank the folks from Jim's group, Bogdan Stoica, Stuart Byma, and David Aksun with whom I shared offices for about a year and had lunch meetings every Wednesday.

Finally, I would like to thank my parents Ika and Branislav, my sister Jelena, and my wife Milica for their immense support and encouragement.

Abstract

The rise of web-scale services has led to a staggering growth in user data on the Internet [85]. To transform such a vast raw data into valuable information for the user and provide quality assurances, it is important to minimize access latency and enable in-memory processing. For more than a decade, the only practical way to accommodate for ever-growing data in memory has been to scale out server resources, which has led to the emergence of large-scale datacenters [19] and distributed non-relational databases (NoSQL) [41, 56]. Such horizontal scaling of resources translates to an increasing number of servers that participate in processing individual user requests. Typically, each user request results in hundreds of independent queries targeting different NoSQL nodes - servers, and the larger the number of servers involved, the higher the *fan-out*. To complete a single user request, all of the queries associated with that request have to complete first, and thus, the slowest query determines the completion time. Because of skewed popularity distributions and resource contention, the more servers we have, the harder it is to achieve high throughput and facilitate server utilization, without violating service level objectives [54, 127].

This thesis proposes *rack-scale memory pooling* (RSMP), a new scaling technique for future datacenters that reduces networking overheads and improves the performance of core datacenter software. RSMP is an approach to building larger, rack-scale capacity units for datacenters through specialized fabric interconnects with support for one-sided operations, and using them, in lieu of conventional servers (e.g. 1U), to scale out. We define an RSMP unit to be a server rack connecting 10s to 100s of servers to a secondary network enabling direct, low-latency access to the global memory of the rack. We, then, propose a new RSMP design - Scale-Out NUMA that leverages integration and a NUMA fabric to bridge the gap between local and remote memory to only 5× difference in access latency. Finally, we show how RSMP impacts NoSQL data serving, a key datacenter service used by most web-scale applications

today. We show that using fewer larger data shards leads to less load imbalance and higher effective throughput, without violating applications' service level objectives. For example, by using Scale-Out NUMA [126], RSMP improves the throughput of a key-value store up to 8.2× over a traditional scale-out deployment [128].

Key words: datacenters, rack-scale systems, RDMA, distributed non-relational databases (NoSQL)

Résumé

L'avènement de l'ère internet a été marqué par la prolifération de services web, qui à leur tour ont entrainé une véritable explosion, en termes de volumes, des données utilisateurs circulant sur le net [85]. L'exploitation de ces données brutes ainsi que leur transformation en informations utiles doit se faire tout en garantissant une certaine qualité de services. Pour cela, il est important de minimiser les temps d'accès aux données et de permettre leur traitement en mémoire. Pendant plus d'une décennie, la solution adoptée par défaut pour répondre à un volume toujours croissant de données à traiter en mémoire a été d'accroître les ressources et la puissance des serveurs, conduisant de fait à la création de centres de données colossaux [19] ainsi qu'à l'apparition de bases de données distribuées non-relationnelles [41, 56]. Une telle augmentation des ressources côté serveur passe inévitablement par l'ajout de machines. Le traitement d'une requête utilisateur requière ainsi la participation de plusieurs serveurs. Chaque requête utilisateur génère une multitude de sous-requêtes indépendantes visant des bases de données réparties sur différents serveurs ou nœuds NoSQL. Plus le nombre de serveurs impliqués est grand, plus le *fan-out* est important. Le résultat d'une requête utilisateur n'est obtenu que lorsque toutes les sous-requêtes ont été traitées. Par conséquent, le temps nécessaire à l'exécution d'une requête utilisateur est déterminé par celui de la sous-requête la plus lente. Le grand nombre de serveurs impliqués dans une requête empêche de garantir l'utilisation uniforme des ressources et le maintien du débit de traitement à un niveau respectant les *service level objectives* (SLO) [54, 127]. Cela s'explique d'une part par l'inégale popularité des différents nœuds, mais également par les conflits d'utilisation des ressources nécessaires à chacune des requêtes.

Cette thèse propose *rack-scale memory pooling* (RSMP), une nouvelle technique de *scaling* pour les futurs centres de données permettant de réduire les délais réseaux et d'améliorer les performances des logiciels au cœur des centres de données. RSMP présente une approche

permettant de construire de larges unités, à l'échelle des *racks*, pour les centres de données, via l'utilisation de *fabric interconnects* spécialisés supportant les opérations *one-sided*. Ces unités sont destinées à remplacer les serveurs conventionnels (e.g., 1U) lors d'un *scale out*. Nous définissons une unité RSMP comme un *rack* connectant jusqu'à plusieurs centaines de serveurs à un sous-réseau. Ce sous-réseau permet un accès direct et avec peu de latence à la mémoire globale du rack. Nous proposons ensuite un nouveau design RSMP, Scale-Out Numa. Ce nouveau design tire profit de l'intégration des interfaces réseaux ainsi que des *NUMA fabrics* afin de réduire les délais d'accès à la mémoire globale, comparés aux coûts d'accès de la mémoire locale (facteur de 5). Enfin, nous montrons comment RSMP impacte un service NoSQL, l'une des bases de données les plus populaires parmi les applications web devant traiter de grands volumes de données. Nous montrons qu'il est possible, en utilisant seulement quelques larges *data shards*, d'obtenir une meilleure répartition des requêtes et un débit de sortie plus élevé, tout en respectant le SLO de l'application. Par exemple, en utilisant Scale-Out NUMA [126], RSMP améliore le débit d'un *key-value store* par un facteur supérieur à 8.2, comparé à la méthode usuelle consistant à augmenter les ressources serveurs (*scale-out*) [128].

Mots clés : centre de données, système en rack, RDMA, bases de données distribuées non-relationnelles

Contents

List of Figures

List of Tables

1 Introduction

The surge in popularity of online services and new technologies is transforming the Internet into a true worldwide network, connecting almost 50% of the world's population at the time of writing [2]. Thanks to the Internet, online services are available everywhere and to everyone, allowing people around the planet to communicate and share, search, learn, shop, organize themselves, just by using a computing device, be it a personal computer or a mobile device. The biggest challenge associated with this trend is to enable fast access to vast user-generated data, allowing applications to perform computation at high speed. A shared memory machine is the ideal candidate, but it cannot physically accommodate exceedingly large datasets, whereas scaling up shared memory is expensive and introduces verification and fault-containment challenges [42, 126].

The in-memory scale-out model has emerged as a solution enabling low-latency local access to data [12]. A scale-out deployment usually assumes a datacenter – a large warehouse full of servers, running a diverse set of in-memory services and applications that retrieve and process data in a distributed manner. In datacenters, data is partitioned and distributed among many servers via a *distributed NoSQL database* [4], where each server hosts a chunk of that data in unstructured or semi-structured format [31, 41, 56, 62, 69, 146]. This thesis focuses only on in-memory workloads where the data is served from DRAM. Unfortunately, even though the actively used portion of data typically resides in memory, many datacenter applications and services require frequent data transfers from remote to local memory and vice verse, primarily because of core data structures and algorithms used or for load balancing, fault tolerance, or

coordination purposes. As compared to local memory, the network is much slower and poor remote access latency is often a major performance obstacle [126].

This thesis proposes *rack-scale memory pooling* (RSMP), an approach to aggregating memory at rack-scale granularity enabling low–latency access to nearby data. We argue that aggregating racks' servers into rack-scale logical entities naturally overcomes a range of issues in the datacenter that arise due to the specifics of the adopted scale-out model, such as load imbalance [127] and resource contention [54]. §1.1 focuses on in-memory data serving using distributed NoSQL databases, a critical datacenter service that is the backbone of most Web applications. §1.2 provides an overview of rack-scale memory pooling (RSMP), a technique whose primary goal is to improve the performance of such systems. §1.3 briefly describes a high–performance rack–scale memory pooling architecture that leverages integration and NUMA. Finally, §1.4 summarizes how NoSQL databases can take advantage of the RSMP datacenter architecture.

1.1 Data Serving in Datacenters

To be able to store and operate on vast amounts of data, large-scale datacenters, also known as warehouse-scale computers [19], have emerged as the core component of any popular service on the Internet today. Modern datacenters consist of thousands of independent servers communicating over a TCP/IP network. Each server of the datacenter's back-end performs computation on a chunk of data to extract meaningful information, or just serves that data in its raw format to the applications the end users interact with.

To facilitate fast access to vast data, most datacenter services tend to keep the data, or at least the popular subset that is actively used, in DRAM [69, 169]. Because of its size, data is typically distributed across many servers, allowing for parallel processing and serving using the concept of distributed NoSQL databases [31, 41, 56, 62, 69, 146]. Even though the scale-out NoSQL model is advantageous from the cost–performance perspective and has become a key concept in the cloud computing industry, it introduces a range of challenges, from the low-level datacenter organization to data distribution, running computation in a distributed setting, dealing with load imbalance, interference, just to name a few [16, 32, 54].

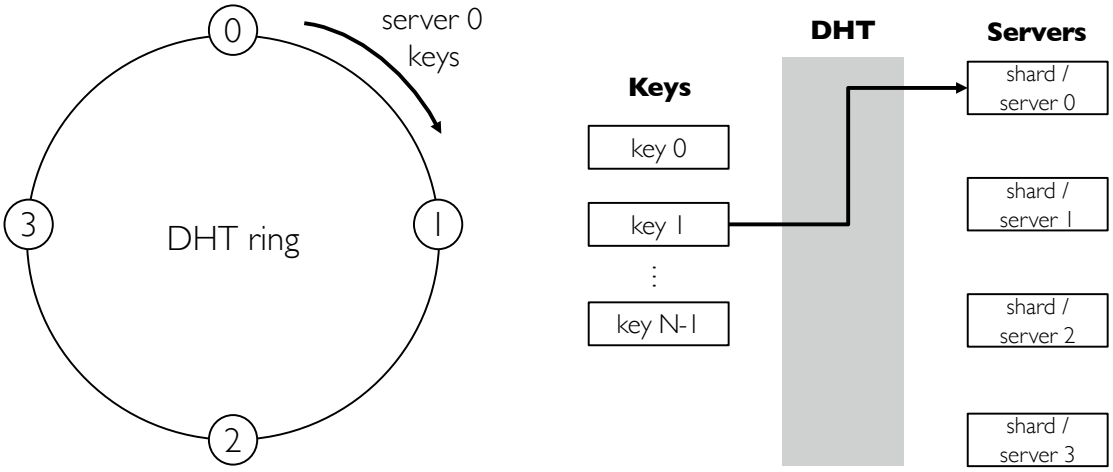


Figure 1.1 – Scale-out key-value store. Distributed hash table (DHT) ring contains key-to-server mappings.

Data serving - NoSQL services are characterized by fine grain, random accesses to large amounts of data [14, 16, 104]. In such services, unlike in analytics, aggregating network transfers is a non-trivial task. Also, unlike relational databases, most NoSQL implementations do not support more complex operations through ACID transactions, such as *join*, for scalability reasons. The classic examples of distributed NoSQL databases include key-value stores (KVS) [56, 62, 69] and their variants, such as graph stores [31] and wide column stores - multi-dimensional maps [13, 41], typically using distributed hash table (DHT) rings or B+ tree-like (index) data structures for client request routing. NoSQL databases store data in trivial unstructured (schema-free) or semi-structured formats, such as key-value pairs (tuples), wide column, or graph, which are different from relational databases that store data using tabular relations. Unlike relational databases, NoSQL databases scale horizontally to thousands of machines and provide the clients with simple *put/get* API for access to individual data items. They sacrifice consistency in favor of availability, partition tolerance, and speed (CAP theorem [29]). NoSQL data serving is at the core of every web-scale application today and is one of the most important services running in today's datacenters [41, 56, 62, 69]. This thesis focuses on key-value stores (KVS), but the conclusions we make are equally applicable to the other types of distributed NoSQL databases running in-memory data serving workloads [31, 41].

Fig. 1.1 illustrates a classic key-value store (KVS) design, in which one or more front-end servers use keys to retrieve data from the back-end servers storing the data in the key-value

pair format [31, 58, 62, 69, 135, 146]. In a typical KVS, a DHT ring maps keys to servers and is used by the clients to look up data in a scalable manner. One simple way a KVS client can locate individual data items using a DHT ring is as follows:

$$serverID = \frac{hash(key)\,\%\,total\_key\_count}{shard\_key\_count}$$

The DHT ring on Fig. 1.1 maps contiguous ranges of keys to four different servers and their respective hash tables. Hash values that exceed the total number of keys are mapped to the same four DHT ranges - servers. The server processing a key-value request uses the hash value to locate the bucket in its local memory. Collisions are typically handled through the concept of *chaining* using linked lists.

The high *fan-out* nature of data serving workloads requires guarantees on the maximum service time that should be within the agreed-upon service level objective (SLO) boundary [54]. Such highly parallel processing of requests can cause load imbalance when the popularity distribution of data items is skewed [81, 127]. The servers hosting highly popular data will run at higher utilization and saturate faster, potentially violating the application's SLO. In such a setting, it is necessary to think about the latency requirements. This thesis studies dynamic replication and proposes memory pooling as a complementary technique for mitigating skew-induced imbalance. In addition, the thesis provides key insights into how memory pooling can be used to accommodate for other sources of rack-level imbalance, such as resource contention [54].

1.2 Rack-Scale Memory Pooling (RSMP)

We argue that bringing the servers of a datacenter closer together within each rack, through a specialized, low–latency network, improves the performance of core datacenter workloads, such as in-memory data serving. We propose *rack-scale memory pooling* (RSMP), which is an approach to aggregating the main memories of a rack's servers and treating them as a shared pool for application data. For RSMP to make sense, the internal rack-scale network connecting the servers must support low-latency *one-sided* memory access primitives. RSMP leverages

one-sided primitives to build coarser grain server nodes – RSMP units, where each server explicitly accesses memory elsewhere in the rack through one-sided read operations.

RSMP can be achieved using RDMA [116], which is a state-of-the-art implementation of one-sided primitives. Today, RDMA exists in the form of specialized PCI cards, called host channel adapters (HCA), that implement the entire RDMA stack, including link, network, and transport layers, in hardware [115]. At the link level, RDMA requires either Converged Ethernet (RoCE) [22] or Infiniband [138], since one-sided operations work best over *lossless* connections. Assuming lossless links, the Infiniband transport implements the *go-back-0* retransmission algorithm [74]. However, it has been shown that large, datacenter-scale RDMA deployments based on Converged Ethernet (RoCE) do, in fact, lose packets, often leading to livelocks because of a naive transport-level algorithm for reliable delivery [74].

The support for one-sided access assumes: (i) the ability to initiate remote memory operations directly from user-space and (ii) the ability to access non-local memory without involving the owner of that memory – its CPU. RDMA HCA offers both by exposing a queue-pair-based (QP-based) interface, where the application and the controller interact directly via memory-mapped pairs of queues, and by supporting address translation through IOMMU.

Fig. 1.2 illustrates a datacenter where the building block is a rack. Unlike standard RDMA deployments that sometimes span multiple racks, RSMP confines the fabric to a rack and assumes TCP for inter-rack communication. By limiting the size of an RSMP unit to a rack, we avoid the emergent safety, performance and monitoring challenges of large-scale RDMA 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 [74, 170]. Using Ethernet for RDMA in datacenters is a common design decision since most datacenter software today is still IP-based. Even though the flow-control issues of commodity Ethernet may go away [74, 170], complementing datacenter-scale RDMA (or equivalent) fabrics with specialized ultra-low-latency and high-bandwidth rack-scale fabrics [15, 126, 130] will be of great importance to balance the load across small groups of servers, without the cost of data replication (see §5). For example, similar to the design from Fig. 1.2, it is easy to imagine specialized, high-performance rack-

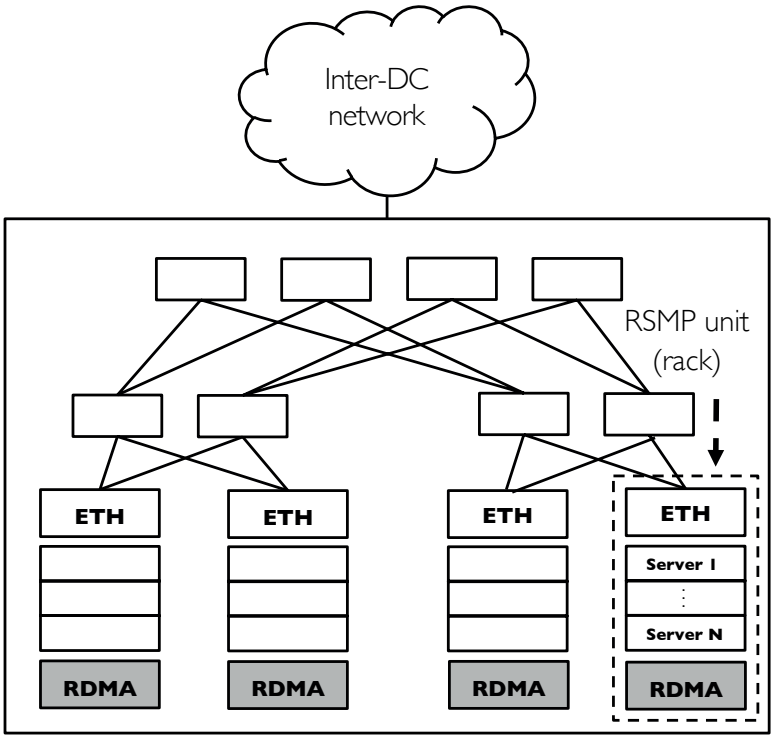


Figure 1.2 – RSMP datacenter architecture using RDMA. Each rack is equipped with an internal RDMA network to form an RSMP unit.

scale systems (e.g., Scale-Out NUMA [126]) in lieu of RDMA racks, and an RDMA network connecting the servers.

Unlike cache-coherent NUMA (ccNUMA), RSMP assumes virtual global address spaces spanning multiple servers that are not cache coherent with respect to each other. The software framework running on top of RSMP and exposing the abstraction of global virtual address spaces uses explicit one-sided primitives to access remote portions of an address space.

The software framework enables concurrent access and abstracts away the distributed nature of an RSMP unit. For the sake of correctness, all the servers belonging to an RSMP unit must follow a specific concurrency model. The RSMP software framework's responsibility is to implement the agreed-upon concurrency scheme and allow application code running on individual servers to access global data safely. The software framework is in charge of resolving memory references and allows the programmer to write shared–memory–like server code. Unlike existing distributed frameworks, such as FARM [58], the RSMP software framework has been designed to support the new datacenter architecture from Fig. 1.2 and includes a

client API based on RPC for server/rack resolution and scheduling. Clients use this API to understand the back-end organization and initiate operations. That is, which individual nodes have low–latency shared access to each other's main memory and constitute an RSMP node. Thus, the software framework allows application clients to easily take advantage of the RSMP architecture. In addition, the API enables layering of a distributed NoSQL database on top of the framework, which we demonstrate in §5.

1.3 Improving the Effectiveness of RSMP Through Integration

How effective RSMP is depends on the underlying fabric's latency. That is, the lower the latency, the higher the impact on application throughput. The key goal of any RSMP implementation should be to bring down the remote access latency as close to local access latency as possible. The main factors for this include: the underlying fabric's link performance, the switch latency, the network architecture (e.g., switch–based or *glueless*, CLOS or 3D torus) [98] and the end-to-end network performance as a function of the cluster size (i.e., average hop count), the low–level communication protocol, and, most importantly, the interface between an end–point's CPU and the fabric. As we pointed out earlier, the minimum requirement for RSMP to scale is to support one-sided primitives. Without such support the latency is guaranteed not to be sufficiently low [75].

The upper bound on the maximum sustainable remote access latency depends on the use case and the nature of the application itself. Some RSMP applications can tolerate higher access latency to remote memory [112], whereas some other applications require a minimal remote to local access ratio [58]. Nevertheless, any application that can take advantage of RSMP will also benefit from faster remote access and result in better performance [128].

This thesis introduces an RSMP design leveraging integrated controllers, a low-latency fabric, and small form factor to bridge the gap between the remote and local access latency. A key requirement for efficient RSMP is to support a low-overhead interface between the CPU and the controller. To achieve that, we propose folding the controller into the cache hierarchy of the CPU. By making the controller cache-coherent with the CPU cores, it is possible to keep the queue pairs in shared memory and allow applications to interact with the controller

through the standard load/store interface, thus minimizing the overhead of scheduling remote memory operations and receiving completion notification. The second requirement is to use a low-latency fabric interconnect. This thesis introduces a solution based on NUMA, but we also argue that a similar solution is feasible with other types of high-performance fabrics, such as InfiniBand [138]. Nevertheless, NUMA significantly reduces complexity, in terms of state, the messaging protocol, and the hardware-software interface.

§4 describes Scale-Out NUMA (soNUMA), an integrated RSMP design based on NUMA. soNUMA uses a NUMA memory subsystem, such as Intel QuickPath Interconnect (QPI) [120], to layer a simple cache-block request-reply one-sided access protocol. The basic one-sided primitives, such as remote read and write are implemented as part of a specialized hardware block, called *remote memory controller* (RMC). The RMC is integrated into the CPU's cache hierarchy, enabling user programs to interact with it directly via cache-resident queues. Applications schedule remote transfers via *work queues* (WQ), and asynchronously get notified by the RMC upon completion via *completion queues* (CQ). This programming model is similar to RDMA verbs [116], in which a pair of queues is used for sending/receiving messages and scheduling remote transfers, plus a completion queue for notifications. In RDMA, the controller must use DMA to perform transfers of queue-pair entries in and out of the main memory. Modern processor technologies, such as *direct cache access* (DCA) [82], reduce the overhead by allowing the controller to directly access the CPU cache, but still require DMA. Unlike RDMA, soNUMA implements a lean transport protocol with a narrow and cache-coherent hardware-software interface that improves the latency of remote operations and reduces hardware complexity. Our RSMP software framework described in §3.5 is compatible with both the state-of-the-art RDMA and our NUMA–based solution.

The high-performance characteristics of NUMA, which includes small *maximum transmission unit* (MTU) with low header overhead, and the ability to interact with the network interface directly from user-space through shared memory, enable applications running on soNUMA to access remote data at a small factor of the local access latency and stream at DDR rates. This thesis studies the advantages of integrated RSMP solutions, such as Scale-Out NUMA, over discrete solutions. We show that low remote access latency enables efficient RSMP and improves the performance of core datacenter workloads, such as NoSQL data serving.

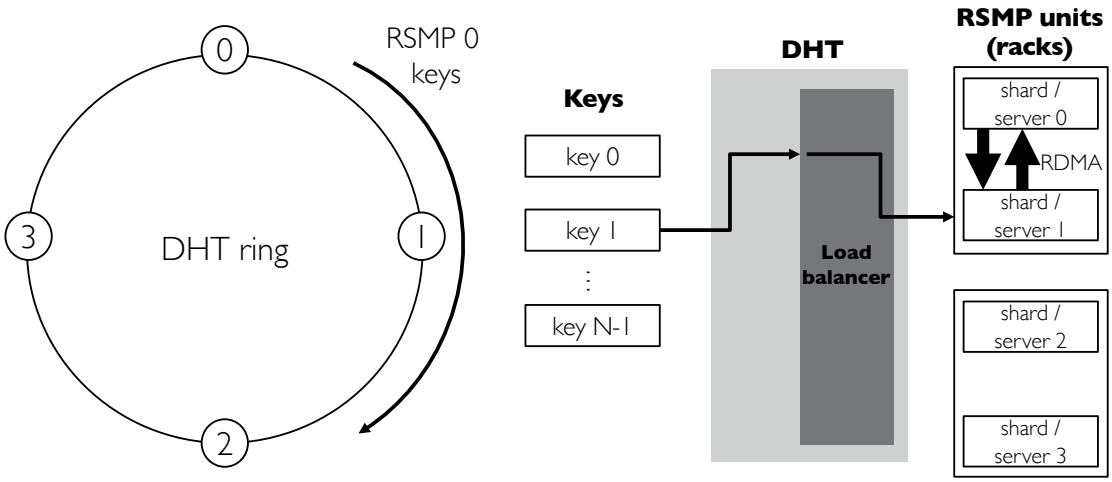


Figure 1.3 – A Rack-Out KVS architecture.

1.4 Using RSMP in Data Serving

To understand the benefits of RSMP, we study NoSQL databases and identify the inefficiencies and bottlenecks of in-memory data serving workloads. Data serving is an important class of datacenter services that most web applications today rely on, such as social networks or e-commerce [31, 56]. In particular, our focus is on hash–partitioned data serving systems, widely referred to as key-value stores (KVS) [45, 56, 62, 69, 106, 146]. Our conclusions also apply to the other types of distributed NoSQL databases storing data in simple, unstructured or semi-structured formats in memory, with weak consistency semantics [38].

We observe that the highly parallel nature of KVS, and distributed NoSQL databases in general, results in load imbalance that often leads to SLO violations. In particular, key popularity distributions are typically skewed, which directly translates to non-uniform utilization of the servers [127]. We define such non-uniform load distribution as *shard skew*. The larger the shard skew, the faster the system saturates, at which point the 99th percentile latency violates the SLO. Such sensitivity to skew prevents proper utilization of datacenter resources. Moreover, as the number of participating servers increases, the shard skew deteriorates and further lowers the utilization of resources. In addition to shard skew, other sources of imbalance, such as resource contention, present a threat to stable performance with the increase in server count [54].

To reduce load imbalance, we apply the concept of RSMP and propose to make the unit of sharding a rack [128]. Fig. 1.3 illustrates Rack-Out KVS, a key-value store design using RSMP. The DHT ring in Rack-Out KVS maps contiguous key ranges to RSMP units and features a load balancer that determines which server within the target RSMP unit should process an incoming key-value request.

In §5, we describe a simple load balancing approach for Rack-Out KVS that uses a random function, and, in §6, we present an adaptive load balancing approach leveraging workload characteristics and server occupancy to take scheduling decisions. Our RSMP design in Rack-Out KVS is based on the *concurrent-read/exclusive-write* (CREW) access model [102]. The CREW access model enables load balancing of read operations within each rack. We show that the larger the RSMP grouping (i.e., the number of servers within a rack), the more throughput we get out of the system, without violating the tail latency SLO.

We study the impact of Rack-Out KVS on in-memory data serving workloads using the state-of-the-art RoCE hardware [115] and an integrated solution based on Scale-Out NUMA [126]. We first use RDMA to show the benefit of RSMP for skewed workloads. RDMA provides sufficiently low remote access latency for RSMP to make an impact. We, then, evaluate Rack-Out KVS using Scale-Out NUMA, which provides lower remote access latency as compared to RDMA and further boosts the performance.

1.5 Thesis Goals and Primary Contributions

This thesis proposes *rack-scale memory pooling* (RSMP), an approach to aggregating the main memories of a datacenter rack using a low-latency fabric interconnect, with the goal to create a scalable building block for future datacenters. The concept of RSMP assumes hardware support for one–sided operations and adequate software support implementing the abstraction of a global address space. The software support for RSMP exposes a server API for allocation and transparent access to data via one-sided primitives, and a client API for request routing and access to data via RPC. To demonstrate the impact of RSMP, this thesis focuses on distributed non-relational (NoSQL) databases and in-memory data serving workloads.

The statement of this thesis is as follows:

> *Pooling the main memory within datacenter racks using a low-latency fabric improves the performance of in-memory data serving workloads.*

We first identify the performance bottlenecks in popular datacenter services, focusing on in-memory data serving using distributed non-relational (NoSQL) databases. We show that data popularity skew that exists in most data serving workload translates to load imbalance and requires the servers of a datacenter's back-end to communicate, mainly for load balancing reasons.

Second, we introduce a set of design principles for making communication within a rack more efficient. We propose *rack-scale memory pooling* (RSMP), a technique that relies on fast one-sided operations to enable shared access to a rack's data. RSMP exposes a different kind of programming abstraction that follows the shared-memory model, rather than messaging. In RSMP, improving the underlying rack-scale fabric is advantageous and directly impacts applications' performance. To build such a fabric, we identify key bottlenecks in existing implementations, such as InfiniBand, and argue for a lightweight solution that leverages integration.

Third, we design and implement a high-performance rack-scale system based on NUMA and a custom one-sided access protocol. The low-overhead, RDMA-like protocol is layered on top of a NUMA cache-block request/reply fabric, and is implemented in hardware as part of a specialized on-chip block. The software interacts directly with the on-chip controller through cache coherent memory accesses. Using a set of benchmarks we evaluate our solution and show the benefits of using a NUMA-based fabric and making the controller a part of a node's coherence domain.

Fourth, we propose Rack-Out KVS, a distributed NoSQL database design where the unit of sharding is one RSMP unit. By using fewer larger units, and hence fewer data shards, Rack-Out KVS reduces the load imbalance that exists in a majority of data serving workloads, leading to higher throughput, while respecting tight service-level objectives (SLO). We show that Rack-Out KVS is synergistic with dynamic replication and together lead to higher throughput than

RSMP or dynamic replication alone. However, RSMP significantly reduces the need for dynamic replication and, thus, reduces the overhead that comes with frequent load monitoring, creating replicas, maintaining consistency, etc.

Fifth, we propose a new scheduling mechanism for the Rack-Out KVS client that adapts the workload to the skewed write distribution and the variable service times within an RSMP unit. The new mechanism comprises two independent algorithms that create and adjust per-rack read distributions to reflect the imbalance within each rack. The read distributions are computed periodically and sampled by the clients. The goal of adaptive load balancing is to keep the servers of each RSMP unit, in the ideal case, equally occupied.

To summarize, this thesis claims the following contributions:

- A detailed analysis of the impact of data popularity skew on load imbalance in datacenter environments. We use in-memory data serving as an example to motivate for memory aggregation through low-latency one-sided primitives.
- *Rack-scale memory pooling* (RSMP), an approach to aggregating the main memory of a rack using one-sided operations. We introduce a set of design principles and technical requirements for RSMP. We, then, identify bottlenecks in existing fabrics and describe how tight integration can drive the remote latency down to a small factor of the local access latency.
- Scale-Out NUMA, an integrated and scalable RSMP design based on NUMA. We explain the design and implementation of Scale-Out NUMA in detail, and evaluate it using a set of micro-benchmarks.
- A key-value store (KVS) design leveraging RSMP. We explain how distributed NoSQL databases can take advantage of RSMP to improve data serving throughput, without violating applications' SLO. Most data serving workloads are skewed, which translates to load imbalance across the servers of a scale-out deployment. By grouping the servers within each rack using RSMP and, thus, reducing the number of data shards, applications suffer less from load imbalance and achieve higher throughput.

- An adaptive load balancing extension for RSMP-based data serving. We observe that the static RSMP approach has limited impact on the performance of read-write workloads and systems under contention because of the random scheduling of read requests. We show how adaptive scheduling of read requests that levarages the information about the workload and server occupancy leads to higher throughput for read-write workloads.

§2 focuses on skew analysis in classic data serving and studies datacenter technology trends. In §3, we introduce the concept of *rack-scale memory pooling* (RSMP) and state the main prerequisites for RSMP that are necessary to provide performance benefits over scale-out. In §4, we propose a rack-scale system design based on NUMA enabling highly effective RSMP. §5 describes how RSMP impacts the performance of distributed NoSQL databases and in-memory data serving. Finally, §6 proposes an adaptive load balancing mechanism that pushes the limits of CREW, the concurrency model enabling RSMP.

This thesis does not cover the security and fault tolerance aspects of RSMP. The thesis also does not study higher level programming models built over the classic global address space abstraction that is used in RSMP. Next, we focus on distributed in-memory NoSQL databases exclusively, even though we see other use cases for RSMP, such as rack-scale analytics (check §4.7.5). In the Scale-Out NUMA work, we do not focus on the network but, instead, study the end-point design and its impact on the remote access performance.

1.5.1 Bibliographic Notes

Portions of this thesis are based on the work I have previously published with my advisors Edouard Bugnion and Babak Falsafi, and two of my colleagues, Alexandros Daglis and Boris Grot. Chapter 4 is based on a conference paper published in the *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* in 2014 [126], and a patent granted by the *US Patent & Trademark Office* in 2017 [129]. Chapter 5 is based on an extended abstract published in the *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)* in 2016 [127], and a conference paper published in the *Proceedings of the ACM Symposium on Cloud Computing (SOCC)* in 2016 [128]. Chapter 6 is based on a pending journal submission.

2 Data Serving in Datacenters

The entire datacenter stack, from the low–level server architecture to application code, needs to be carefully crafted in order to maximize user experience. The number of users of any popular Web application is growing and requires constant effort of fine tuning datacenter software, including the optimization of specific application features and system software [23, 110, 169], rearchitecting datacenter networks to support higher bisection bandwidth and lower the oversubscription rate [61], adding accelerators to servers to speed up critical application features [93, 139], horizontally scaling server resources to accommodate for increasingly larger datasets [12], etc.

The classic scale-out design has a number of advantages, such as the ability to arbitrarily scale the system, but it is often a major performance-limiting factor. First, the distributed nature of scale-out requires using the network that is much slower as compared to CPU and local memory. Some applications exhibit more locality than the other, but, in general, most applications rely on the network for coordination, data transfers, and replication [55, 81, 83]. Second, to increase DRAM capacity and accommodate for increasingly larger datasets in memory, the straightforward solution is to just throw in more servers. However, horizontal scaling increases a provider's costs without a proportional increase in performance. For example, partitioning a graph results in more edges spanning two servers, which directly translates to more network traffic and slower computation; adding more servers to a NoSQL database results in more imbalance and worse resource utilization [127]; distributing a web index across a larger number of servers makes the search engine more vulnerable to tail latency

violations [54].

This chapter describes what today's datacenter infrastructure looks like and discusses the implications of the scale-out model on the performance of in-memory data serving workloads and distributed NoSQL databases. §2.1 focuses on data serving and key-value stores (KVS) in particular, since they are widely used in popular web applications [31, 56, 62, 69, 146, 169]. We identify major bottlenecks in KVS and show why the standard scale-out model in combination with dynamic replication alone is not a great fit for heavely skewed data serving workloads.

Designing and implementing efficient datacenter services is a first-order concern in deploying large-scale applications on the Internet. In a nutshell, a typical application is organized in two or more tiers, where the front-end tier is stateless and runs Web server applications that the users directly interact with, and the back-end runs a wide variety of datacenter services that the Web server applications use to produce meaningful content for the user. We first list and briefly discuss a few important aspects of datacenter infrastructure. In §2.1, we focus on data serving and provide an extensive analysis of how the scale-out nature of datacenters impacts the performance of this core datacenter service [41, 56, 62, 123].

Resource provisioning. A datacenter service should be provisioned with enough physical resources, such that it (almost) never lacks CPU cycles, memory, or network bandwidth. This requires planning and cleverly mapping different services onto datacenter resources to maximize utilization, but at the same time allow each service to take full advantage of the available resources. This is usually achieved through monitoring and analyzing each individual service in isolation or in production, and achieving balance by moving/replicating data to less utilized servers [81] or entirely migrating/replicating computation in form of virtual machines (VM) [125, 166]. Resource provisioning requires complex system support that is typically implemented as part of the datacenter scheduler. The scheduler is the only entity that has a global view of the system and, thus, is in charge of approximating the optimal placement of data/VMs and allocation of resources [32, 149, 161].

Networking. The characteristics of different datacenter services vary in terms of algorithms and data structures used, communication patterns, user activity (input load), popularity

distribution, etc [68]. Pure scale-out services are rare, and, in the common case, they require significant network support to accomodate for: intensive communication patters induced by specific data structures or algorithms used [55, 112], load spikes necessitating proper resource provisioning through data or VM migration and replication, etc. Thus, some of the key questions in datacenter design include: what should the network topology look like, how much bandwidth should we provision, what should be the network oversubscription ratio, how can we improve the network latency to facilitate access to non-local data, etc. Most datacenter operators have adopted CLOS topologies, which can scale horizontally to arbitrary sizes with high link redundancy using low-radix switches [8, 72, 153]. Migrating a data shard or a VM may occupy the network for too long and violate application SLO. Also, some services may require fine-grain low-latency access to remote data, which requires specialized hardware [115, 126]. Thus, with the rise of cloud computing, the network becomes a key resource that should enable core datacenter software to bridge the gap between local and remote memory.

Performance isolation. Datacenters run a diverse set of applications and services concurrently. Web search, for instance, depends on its indexing and crawling services running on thousands of servers in the datacenter [30, 53]. At the same time, services like analytics (e.g. MapReduce [55]) or data serving (e.g. Bigtable [41]), are collocated on the same servers, competing for resources such as CPU, memory, and network. Consolidation is necessary in order to adequately utilize available resources and reduce costs. Deploying a large datacenter for just one service would not pay off and most definitely result in waste of resources. Consolidation is often the main contributor to performance interference between collocated services. Therefore, technologies like virtual machines and containers have emerged [17]. The main responsibility of a virtual machine monitor (hypervisor) is to isolate independent services and reduce performance interference among them. When using such technologies, each service or application runs inside its virtual envirnoment and is supplied with a prespecified amount of resources. Nevertheless, performance interference does occur even in today's virtualized or containerized envirnoments, and it is often the case that one service degrades the performance of another collocated service due to resource contention [54]. Performance interference also occurs within individual virtual machines / processes due to *daemons* (i.e. background processes), garbage collection, maintenance activities, energy management.

In §2.1, we focus on the concept of distributed NoSQL databases, with emphasis on key-value stores (KVS), and identify key inefficiencies. §2.2 describes how popularity skew leads to load imbalance and underutilized resources. In §2.3, we argue that dynamic replication alone introduces significant overhead in heavily skewed data serving workloads. Finally, §2.4 introduces the problem of variable service times that is due to performance interference, background activities, and non-uniform memory accesses.

2.1 Distributed NoSQL Databases

Data serving is a critical datacenter service that is central to the big data revolution. Applications like social networks, search engines, and e-commerce extensively use data serving to retrieve data quickly and return it to the user in some readable form. The data is usually distributed among many servers and is kept in memory. Upon a user request, the web server running the application concurrently issues a number of data lookups or updates to a collection of back-end servers. In a nutshell, after the application had retrieved the requested data items, it assembles a page (or a subset of it [1]) and sends it to the user's web browser. Such high *fan–out* nature of data serving workloads makes the tail latency the most relevant metric; the slowest lookup determines an application's performance. The larger the number of partipating servers per request, the higher the likelihood of violating the tail latency objective [54]. Such latency considerations, for example, force Facebook to restrict the number of sequential data accesses to fewer than 150 per rendered web page [56].

Data serving services are commonly implemented as distributed NoSQL databases hosting data in simple unstructured or semi-structured formats, such as tuples, graphs, sparse multi-dimensional maps, etc [31, 41, 56, 62, 69, 146]. Unlike relational databases, NoSQL databases scale horizontaly to thousands of machines [91]. They sacrifice consistency in favor of availability, partition tolerance, and speed (CAP theorem [29]). Fig. 2.1 illustrates a datacenter-scale NoSQL deployment, where each server hosts a group of small data chunks, often referred to as *micro-shards* [81] or *tablets* [41]. A *micro-shard* is the unit of migration and replication in distributed NoSQL databases.

A significant fraction of distributed NoSQL databases leverage consistent hashing to partition

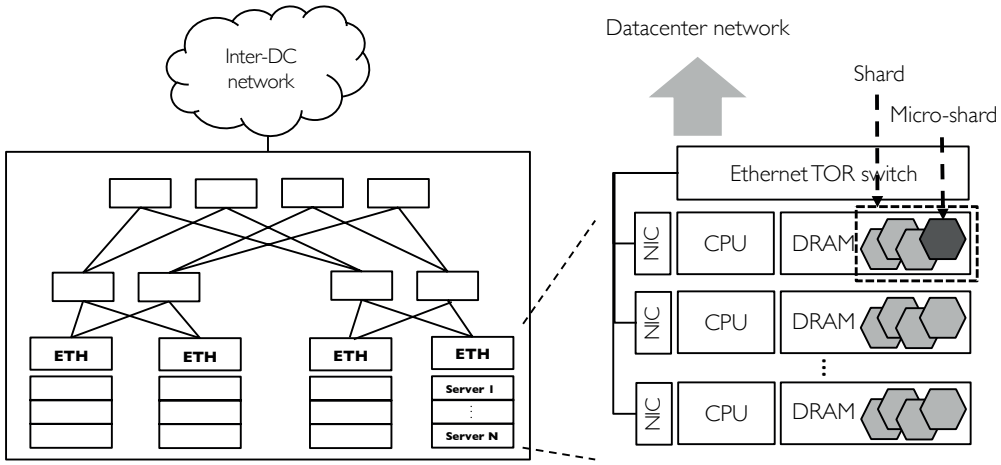


Figure 2.1 – A typical datacenter/rack architecture. Each server hosts a group of micro-shards of a distributed NoSQL database.

the data among the participating servers. We refer to hash-partitioned NoSQL databases that store data in a trivial key-value format as key-value stores (KVS). In KVS, all the keys and the associated values that map to one contiguous range reside in the memory of the server responsible for that range. This enables client-side hash partitioning, in which the client computes the hash of the key and contacts the owner directly [69, 146]. Alternativly, other designs allow the servers to coordinate among themselves and route the key to the owner [62]. When replication is enabled, mainly for fault tolerance reasons, each server is responsible for multiple subsequent key ranges, typically three.

Often times, distributed NoSQL databases replicate data purely for load balancing reasons. In this work, we refer to such replication as dynamic replication, where the data is replicated at *micro-shard* granularity, and to an arbitrary number of servers, depending on the load. This thesis focuses on distributed data serving with support for dynamic replication. We assume a key-value store with support for dynamic replication as the baseline for the design described in the next section.

In the rest of this section, we cover the basics of data serving using key-value stores (KVS) (§2.1.1). Then, in the rest of the chapter, we study the load imbalance problems in data serving workloads. The conclusions we make hold for the general class of distributed NoSQL databases, because of the similar characteristics of the various implementations; for example, a NoSQL database could use a lexicographically ordered index instead of a hash, and store data

in giant tables (wide column) partitioned across the servers, rather than key-value tuples [41]. Nevertheless, only the simple put/get operations on individual data items (rows) would apply, while the index is, like consistent hashing, distribution-agnostic, as data popularity distributions cannot be predicted. Thus, both of these NoSQL designs are prone to the same load imbalance problems, such as skewed data serving workloads.

In the following section, we use a skew analysis to motivate for a different kind of datacenter architecture, one in which the building block is a rack, rather than a server. While most concepts we introduce also apply to the data processing category of services, such as graph processing, throughout this thesis we focus primarily on in-memory data serving. We evaluate the impact of our techniques described in the following sections on this particular class of datacenter services.

2.1.1 Case Study: Key-Value Stores

An in-memory key-value store (KVS) is the most critical component of many modern cloud applications. Several such large–scale applications 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 [16, 31, 56, 123]. KVS such as Memcached [69], Redis [146], Dynamo [56], TAO [31], and Voldemort [106] are used in production environments of large service providers such as Facebook, Amazon, Twitter, Zynga, and LinkedIn [9, 105, 123, 159].

The popularity of key-value stores has led to considerable research and development efforts, including open-source implementations [62], research efforts [11, 132] and a wide range of sophisticated, highly tuned frameworks that aspire to become the state-of-the-art of KVS [58, 101, 102]. In principle, all these different KVS systems are distributed hash tables that organize data in a simple key–value pair format, where the clients use a hash function to look-up individual pairs.

Each server hosts multiple *micro–shards*, where a single micro–shard represents the unit of migration and replication, as illustrated on Fig. 2.1. The metadata, which essentially consists of a hash space ring, needs to be kept up–to–date constantly and it should be available to both clients and servers. Upon a server failure, for instance, the associated hash range needs to

be reassigned to another server, and the clients need to learn about this change in a timely manner, so that they can route their requests correctly.

KVS commonly deploy weaker consistency models, since under strong consistency, a KVS is not guaranteed to be available in the presence of network failures (partitions). As previously said, KVS usually maintain multiple copies of the data on multiple servers, for both load balancing and fault tolerance reasons. To accelerate write operations, upon a data update request, the receiver performs a local update, notifies the client, and then in the background propagates the update to the other replicas. Strong consistency requires the KVS to guarantee to always return the most recent version of any data, which requires extra checks. In particular, each read needs to trigger a retrieval of all the copies of the requested data item and return the most recent version to the application. Luckily, most applications can tolerate weaker consistency models, where the receiving server is not obliged to return the most recent copy. The number of replicas involved in each read operation defines the consistency level [62].

Service providers set strict SLO to ensure high quality, designing the service to respond to user requests within a short and bounded delay. For high fan–out applications, such as a social networking website, designing for the average latency is not enough; a good service guarantees that the vast majority of requests will meet the SLO, and thus targets a 99th or even 99.9th percentile latency of just a few milliseconds [54, 56]. At such tight latency constraints, keeping all or the majority of data in memory is common practice. However, given the sheer amount of data modern services handle, adequate memory capacity can only be supplied by deploying a vast number of servers, which directly contradicts with the tail latency requirement [54, 127].

2.2 Scale-Out-Induced Imbalance

Distributed NoSQL databases 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 the popularity of the data items varies greatly. Previous work has shown that popularity distributions in real-world data serving workloads follow a power-law distribution [14, 16, 67, 81], resulting in an access frequency imbalance, commonly referred to as *skew*. A skewed distribution is accurately represented by the power-law Zipfian distribution [16, 40, 49, 67, 151].

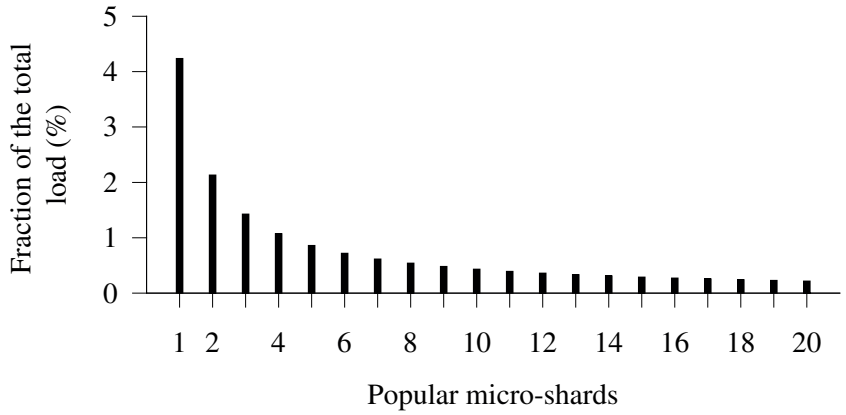


Figure 2.2 – The 20 most popular micro-shards out of 512000 micro-shards (sorted), Zipfian input key distribution.

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 data serving research [46, 58, 79, 101, 102]. Some studies show that the popularity distribution skew in real-world datasets can be lower than that (e.g., $\alpha = 0.6$ [151], $\alpha = 0.7 - 0.9$ [141, 151]), but also even higher (e.g., up to $\alpha = 1.01$ [40, 67]).

Sharding the data across the deployment's collection of servers is done by grouping the data items into *micro-shards*, each server being responsible for hosting and serving hundreds or thousands of them from its local memory [54, 68]. This data distribution is typically done by applying a hash function to the key, which maps each data item to a micro-shard (e.g., [62, 113, 156]) 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 static, stateless, and distribution-agnostic, as data item popularities cannot be predicted in advance or controlled, and may change over time. The same holds for lexicographically ordered index, which is another popular data placement method in distributed NoSQL databases [41].

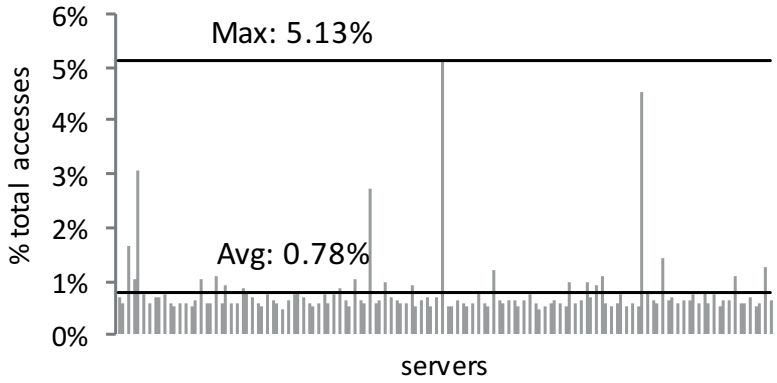


Figure 2.3 – Grouping of 250M data items with a power-law popularity distribution in 128 servers.

Fig. 2.2 shows the 20 most popular micro-shards in a one billion key dataset. The hottest micro-shard alone accounts for 4.2% of the total traffic, which is 20× the average load of an entire server in a 512-node cluster, and 85× in a 2048-node cluster of the same aggregate capacity. 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 of data items is still observable as popularity skew across shards [127].

Fig. 2.3 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 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 number of servers.

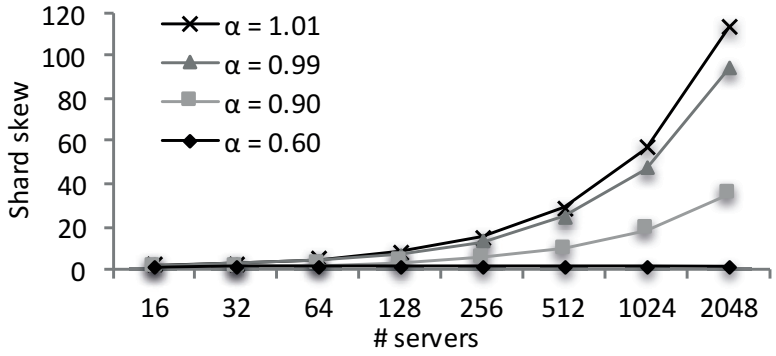


Figure 2.4 – Impact of the Zipfian exponent (250M keys).

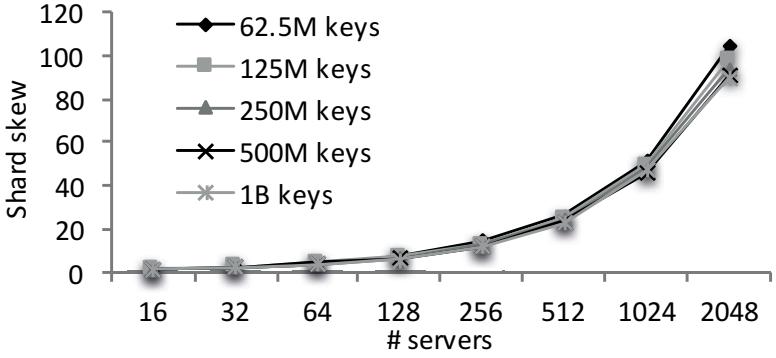


Figure 2.5 – Impact of the dataset size (Zipfian $\alpha = 0.99$).

Fig. 2.4 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×.

Fig. 2.5 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. Also, given a set of keys, the choice of the hash function used to distribute the data items to micro-shards has negligible impact on the resulting shard skew. Our experiments with different hash functions corroborate this observation.

2.3 Replication: a Thorny Solution

Service providers are well aware of the problems that arise from skew-induced load imbalance [53, 54, 81]: 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 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. 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. Finally, replication requires a mechanism to keep all the copies consistent. Consistency is a hard problem to deal with, as it introduces both correctness and performance concerns. The variety and complexity of the implementation frameworks dealing with various forms of consistency is an indication of this problem's challenging nature [28, 33, 56, 83, 96, 131, 158].

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* [123]). 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 [79, 81]. 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, etc. [31, 79, 81, 84].

Dynamic replication is never free: (i) load monitoring, 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. Even in a weakly consistent KVS [56],

each replica must eventually be updated, which reduces the system's effective throughput.

2.4 Service Time Variability

In addition to popularity skew, a major source of imbalance in data serving systems, performance interference with either background activities or other applications, maintenance activities, energy management, remote memory accesses or accesses to disk, etc., all lead to variable service times and are, often, the main cause of tail latency violations [54]. Any service that is triggered periodically, such as garbage collection, may contribute to load imbalance. Also, the request processing times may vary depending on the amount of work that needs to be done. Longer processing times require from a server to stall the other incoming requests, which reduces the server's processing rate.

Because of the high fan-out nature of data serving, the larger the number of servers involved in servicing a single user request, the higher the likelihood that the tail latency SLO will be violated [54]. The trend towards extreme scale-out will further exacerbate this problem [63].

Dean et al. [54] proposed *hedged requests* where a client sends a secondary request after a short delay (1ms). First, such an approach works for systems with end-to-end latencies in excess of a few milliseconds. At microsecond-scale, where the propagation delay and the service times are much smaller, because of a low-latency datacenter fabric and in-memory workloads, it is not clear whether issuing concurrent requests is the optimal solution.

The dynamic replication approach would require creating replicas of small data chunks (micro-shards) upon detecting delays in processing incoming requests. The cost of load monitoring, moving the data across the network, and maintaining consistency of the replicas make dynamic replication impractical for solving the service time variability problem.

2.5 Datacenter Technology Trends

The rise of new technologies and architectures for datacenters, such as hardware accelerators, specialized servers, and high–performance interconnects, will change the way we think about designing and building datacenter services. New server technologies, such as micro–

servers [27, 57, 78, 107], improve energy efficiency and are likely to replace conventional blade servers. A micro–server features a low–power System–on–chip and plugs into a small chassis or rack of 10s to 100s of micro–servers. Such a small cluster typically features an internal network based on Ethernet. The servers within the same chassis are treated as independent servers, like in conventional scale–out deployments.

Other than small form factor and integration, researchers and industry leaders have proposed various hardware accelerators to speed up application processing in the datacenter. Catapult [139], for instance, features an FPGA connected to the NIC of each server, allowing developers to accelerate network operations or simply off–load part of the computation to the FPGA. With the rise of "dark silicon" [76], the trend towards specialized ASICs and FPGAs will become more popular in the coming years.

Finally, most datacenter services rely on the network to produce meaningful output [55, 112, 121]. Because the network is multiple orders of magnitude slower than local memory, such transfers are usually exceedingly long and dominate the execution time. How long a transfer will take depends on how big the transfer is, the latency and bandwidth of the network, and the computational paradigm used. For small transfers, it is important that the network provides low propagation delay - latency, whereas for big transfers it is more important that the network provides high bandwidth. How big a transfer is depends on the use case; applications that support batch computation or applications that require replication (VM or data) are likely to initiate larger network transactions. Applications performing fine-grain accesses to data, such as key-value stores, require a low-latency network. In many cases, the programming paradigm allows for the overlap of large network transactions and computation [112]. Applications using replication to scale in throughput are directly dependent on network bandwidth. A slow network will take more time to replicate data and, thus, the clients to take advantage of the extra replicas for load balancing.

To complement novel server designs and accelerators, high–performance networks, such as InfiniBand (in further text IB), are making inroads into the datacenter and will alter the trade-offs for building datacenter services [3, 5]. IB is a complete network stack implemented in hardware, allowing user-level programs to send and receive messages with minimal CPU overhead.

Programs using IB as the means for communication interact directly with the network interface using memory mapped queues, avoiding any processing in the kernel and minimal processing in the user-space. The transport layer of IB, which implements the queue-pair (QP) interface, is implemented in the network interface, allowing for direct interaction between user-space software and the interface. Besides classic send and receive operations, the hardware-software interface of IB includes *remote direct memory access* (RDMA) operations [116], such as remote read and write.

Using remote reads and writes, an application can access the data in the memory of another server without CPU interrupts. In many cases, RDMA provides performance boost and we will demonstrate that on the data serving example. We will also identify the major bottlenecks in IB implementations and propose a more efficient design that leverages integration and delivers higher performance boost than the state-of-the-art RDMA. We will also discuss a hypothetical design combining IB with the concepts introduced in this thesis.

2.6 Summary

Designing datacenter services is a challenging task. Most such services require intensive communication across the datacenter network because the data is distributed. We focused on the specifics of distributed NoSQL databases and key-value stores in particular, which require communication among the servers for load balancing reasons. A typical distributed key-value store shards the data using a hash function and, thus, is 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 [31, 54, 79, 81]. However, dynamic replication comes with considerable overheads: the consistency semantics expected by the application, the dataset's change rate [56], the precision of the monitoring algorithm, and the micro-shard size, all critically impact the system's behavior. Fortunately, RDMA technologies are making inroads into datacenters and will significantly reduce the cost of data transfers, changing the trade-offs in designing and implementing future datacenter services.

3 Rack-Scale Memory Pooling (RSMP)

3.1 Introduction

Rack-scale memory pooling (RSMP) is an approach to aggregating the memory of a datacenter rack for the purpose of mitigating load imbalance in in-memory data serving workloads. RSMP leverages a low-latency fabric with support for *one-sided* primitives [116] to provide shared data access to a group of servers. In this section, we focus on the fundamentals of RSMP, the RSMP node architecture, and the underlying fabric technologies. We also identify major performance obstacles in existing fabrics and propose integration as an approach to lowering the remote access latency.

RSMP enables any application process to transparently and securely access the memory within the rack. Under RSMP, the memory of a rack is treated as a single pool through which multiple application processes, running on distinct servers, can share data. From the application perspective, RSMP exposes the abstraction of a partitioned global address space (PGAS [44]), providing access to both local and remote data through a well-defined interface. The RSMP software framework resolves remote references and performs transfers from remote to local memory and vice verse.

In one-sided, remote reads, the transfer initiator directly accesses the memory of another server, without interrupting its CPU, and copies the data into its local memory. One-sided operations assume a hardware transport, meaning applications can initiate remote transfers

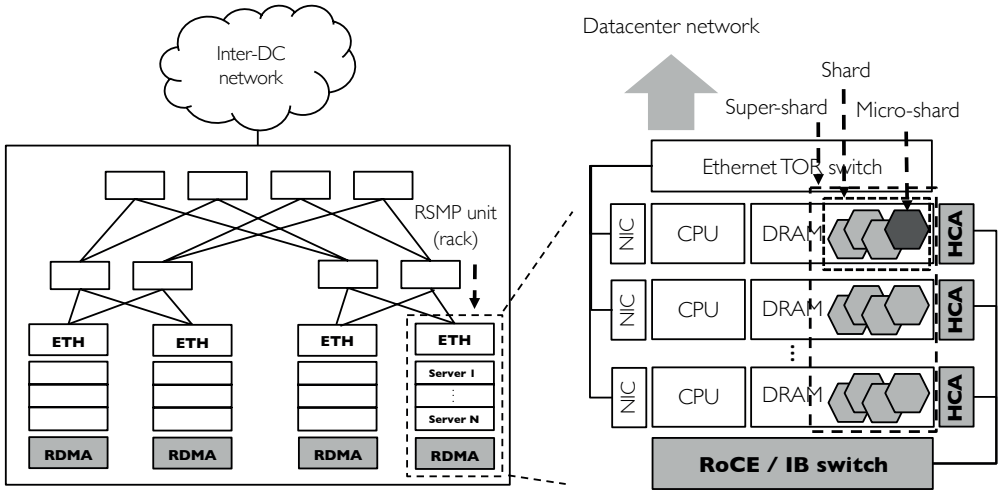


Figure 3.1 – RSMP datacenter architecture using RDMA. The aggregate memory of each rack is perceived as one shared pool of memory for data. Remote data is accessed through the internal RDMA network.

directly from user space and at low CPU overhead. *Remote direct memory access* (RDMA) is an existing technology providing one-sided access. RDMA is often referred to as Infiniband, a network stack providing support for RDMA [5]. Infiniband is implemented in a specialized network controller - *host channel adapter* (HCA) that applications interact with to initiate remote transfers using pairs of memory-mapped queues (i.e., bounded buffers). The Infiniband transport layer, which is also implemented in the network controller, exposes a queue-pair-based interface and performs remote transfers using the underlying fabric. Besides actual Infiniband fabrics [138], the Infiniband transport is compatible with classic IP networks layered on top of lossless Ethernet (RoCE [74, 115]). RDMA works best over lossless fabrics, because losing a packet at the link layer triggers the *back-to-0* algorithm at the Infiniband transport layer that restarts the whole transfer, often leading to livelocks [74]. RoCE is typically used in datacenters, simply because IP traffic is still predominantly used by datacenter software.

Fig. 3.1 illustrates how the servers of a datacenter rack form one giant node - RSMP unit using a secondary rack-scale RDMA network. Such an architecture allows any server within the rack to read any data using the underlying RDMA capability. Because of RSMP, the aggregate data within a rack in distributed NoSQL databases can be perceived as one *super-shard*, as opposed to numerous *micro-shards* or a group of *shards*. In §5, we demonstrate the impact of the RSMP datacenter organization on distributed key-value stores, an important class of

distributed NoSQL databases.

Unlike cache-coherent NUMA (ccNUMA), RSMP assumes virtual global address spaces spanning multiple servers that are not cache coherent with respect to each other. The software framework running on top of RSMP and exposing the abstraction of a global virtual address space uses explicit one-sided primitives to access remote portions of the address space.

Integrating the fabric controller in the CPU's cache hierarchy enables a low-overhead interface between software and the controller. We show how integration can drive the latency down to a small factor of the local access latency. We first introduce integration as a general principle applicable to any implementation of one-sided operations. Then, in the following chapter, we demonstrate this principle through Scale-Out NUMA [126]. Another factor playing key role in minimizing the remote access latency and complexity is the choice of the fabric. Scale-Out NUMA leverages a memory interconnect, where the nodes connect to each other using point-to-point low-latency links.

Finally, RSMP requires a concurrency model to guarantee correctness. As we previously pointed out, one-sided operations execute without the intervention of the server hosting the target data. If a concurrent write access occurs on the host server, the reader could retrieve partially written data. Thus, a concurrency control mechanism is necessary to synchronize local accesses executed on the CPU with remote accesses executed by the fabric controller concurrently. In RSMP, the software framework enables concurrent access and abstract away the distributed nature of an RSMP unit. The RSMP software framework's responsibility is to implement the agreed-upon concurrency scheme and allow application code running on different servers to access global data safely.

The software framework is in charge of resolving memory references and allows the programmer to write shared–memory–like server code. Unlike existing distributed frameworks, such as FARM [58], the RSMP software framework has been designed to support the new datacenter architecture from Fig. 3.1 and includes client API for server/rack resolution and scheduling. By using the RSMP API, the client code can understand the back-end organization. That is, which individual nodes have low–latency shared access to each other's main memory and constitute an RSMP node. Thus, the software framework allows application clients to easily

take advantage of the RSMP architecture.

We first provide some evidence to support our two-fabric design decision (§3.2). Then, we focus on key technical requirements for RSMP and discuss existing technologies (§3.3). In §3.4, we explore additional techniques that make RSMP much more efficient. We, then, describe the programming interface and the software support for resolving memory references in RSMP (§3.5). Finally, in §3.6, we provide a short summary of the RSMP proposal.

3.2 When Does RSMP Make Sense?

RSMP assumes a two-fabric datacenter architecture, where the servers use a primary, datacenter-wide fabric for cross-rack communication, and a secondary, rack-scale fabric for rack-local communication, as illustrated on Fig. 3.1. A rack-scale RDMA fabric consists of per-server fabric controllers (HCA) connected to either an RDMA-enabled *top-of-rack* Ethernet or Infiniband switch (labeled as RDMA). Unlike standard RDMA deployments that sometimes span multiple racks, the RSMP design from Fig. 3.1 confines the fabric to a rack and assumes the classical Ethernet (ETH) at datacenter scale. By limiting the size of an RSMP unit to a rack, such a design avoids the emergent safety, performance and monitoring challenges of large-scale RDMA fabrics. For example, recent work has shown that scaling RDMA over lossless Ethernet introduces issues of congestion control, dealing with deadlocks and livelocks, and other subtleties of priority-based flow control [74, 170]. Using lossless Ethernet for RDMA (i.e., RoCE) in datacenters is common, since most datacenter software today is still IP-based.

Even though the current scalability problems of RoCE are likely to be eliminated in the near future [74, 170], complementing datacenter-scale RDMA (or equivalent) fabrics with specialized ultra-low latency rack-scale fabrics [15, 126, 130] will be of great importance to balance the load across groups of servers, without the cost of replication (see §5). Such a design would require an adequate network oversubscription ratio, because the bottleneck could shift from the CPU to the datacenter-scale fabric, sufficient rack-scale fabric bandwidth to allow for high-throughput rack-local access, and enough per-server memory bandwidth. Similar to the design from Fig. 3.1, it is easy to imagine specialized rack-scale systems with ultra-low memory access latency and high bandwidth in lieu of RDMA racks, and an RDMA network connecting

the servers. To put things into perspective, Scale-Out NUMA [126], a specialized rack-scale system, provides a significant improvement in access latency over the state-of-the-art RDMA, and two to three orders of magnitude as compared to standard Ethernet.

Assuming a scalable RDMA or equivalent design, we now define the two key requirements for a two-fabric solution - RSMP to provide a performance improvement over the single-fabric solution for in-memory data serving workloads. We define *Remote_ST* and *Local_ST* as the average times to process an incoming client request remotely, within the rack, and locally, within the server (respectively). Assuming the *run-to-completion* model, *Remote_ST* includes the primary fabric processing overhead (e.g., TCP/IP in Fig. 3.1), application processing, and the latency of one or multiple remote memory accesses via the secondary fabric. Thus, the ratio between *Remote_ST* and *Local_ST* depends not just on the latency of the secondary fabric, but also the software overheads. We define the *Grouping Factor* (*GF*) as the size of an RSMP unit - how many servers have direct access to each other's memory via one-sided access primitives. Assuming a skewed data serving workload, the following must hold for a two-fabric solution (i.e., RSMP) to provide a performance impact, independent of the fabric technologies:

$$\frac{Remote\_ST}{Local\_ST} = k : k \geq 1, \exists GF : speedup(GF) > 1$$

For a given service time ratio *k*, there should exist some *GF*, such that the maximum speedup, defined as the throughput improvement of the corresponding RSMP deployment over scale-out (i.e., GF=1), is greater than one. For such a *GF* to exist, the following must also hold:

$$\forall\ i \in \{1..\frac{N}{GF_{racks}}\} : primary\_fab\_BW_i \leq secondary\_fab\_BW_i \leq node\_mem\_BW$$

For each RSMP unit - rack in our deployment, there should be enough bisection bandwidth (*BW*) within the rack (secondary fabric) and enough memory bandwidth within each server, such that none of the two ever becomes a throughput bottleneck. Hence, the bottleneck should be either the bandwidth of the primary - datacenter-scale fabric connecting the servers of a rack with the servers in the other racks, or the maximum CPU processing rate. Which of the two saturates first depends on the request size, the per-request software overhead, the number of CPU cores, the primary fabric bandwidth, and the latency of the secondary fabric (assuming synchronous processing). In the following text, we look at the key techniques that are necessary to satisfy the two requirements from the above.

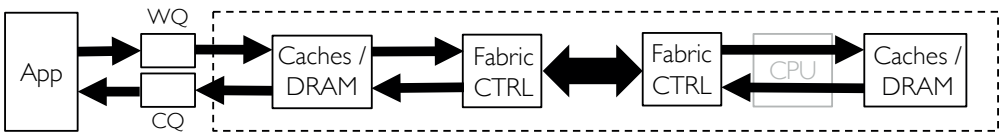


Figure 3.2 – One-sided access illustration. Application uses memory-mapped queues to request remote transfers. CPUs not participating on the data path.

3.3 Design Principles and Techniques Enabling RSMP

Rack-scale memory pooling (RSMP) is a set of general design principles and techniques for aggregating memory within a rack that is applicable to any fabric implementation or technology. This section will introduce those principles as the main prerequisites for effective memory pooling. In §3.3.1, we focus on one-sided primitives, which allow applications to access the memory within the local rack directly and with minimal overhead. §3.3.2 studies fabric interconnects or simply fabrics, which provide the foundation for one-sided communication. Finally, §3.3.3 focuses on concurrency issues in RSMP and explains why a concurrency control mechanism is necessary for correct execution.

3.3.1 One-Sided Memory Access Primitives

One-sided memory access primitives require: (i) the involvement of only one endpoint (usually the source) in a network transfer, and (ii) the ability to initiate and complete such transfers without OS involvement. The typical one-sided primitives include: remote read, remote write, and remote atomic operations, such as *compare-and-swap* or *fetch-and-add*. As Fig. 3.2 shows, one-sided access primitives do not engage the CPU of the server hosting the target data, and, thus, avoid the overhead associated with interrupts. Also, one-sided primitives can be executed from user space using memory-mapped queue pairs, without trapping into the kernel, which is illustrated on the lefthand side of the figure. To support such direct user-space access to remote memory, one-sided primitives require a fabric controller implementing a complete network stack with support for one-sided primitives.

For clarity, we distinguish between the support for one-sided access and the underlying fabric interconnect. The support for one-sided access includes a low-overhead hardware/software interface and a state machine for direct memory access. Such support is typically part of the

upper layers of the network stack (i.e., transport). Fabrics consist of the layers below, such as network/routing, link and physical layers. The fabric layers define the link-level messaging protocol between endpoints, congestion control, routing, link-level flow control, etc. We discuss fabric interconnects in the following subsection.

Fabric controllers enabling one-sided access expose a programming interface based on pairs of queues which are directly accessible to user-space software. A queue pair (QP) consists of two to three circular buffers or queues responsible for initiating one-sided or classic messaging operations (i.e., send/recv) and notifying applications upon their completion. The queues are typically memory mapped into the application's address space, allowing application code to schedule one-sided accesses directly from user space. In its simplest form, a QP-based interface consists of a work queue (WQ) or send queue (SQ) and a completion queue (CQ). When an application wants to access remote memory, it initiates a transfer by encoding all the necessary information into a WQ entry. These parameters typically include remote node ID, offset, key identifying the region exposed for remote access, size of the transfer, etc. The fabric controller polls on WQs and once it detects a new WQ entry, it decodes the passed information and initiates a data transfer. Upon completion, the controller notifies the application via a CQ.

The remote access protocol interfaces with applications through the QP-based interface and transforms application requests into fabric transactions (e.g., IP packets). Such protocol is implemented as a state machine and consists of one or more hardware pipelines that layer remote data transfers on top of the fabric's transport. The protocol also defines a set of message headers carrying essential information between two endpoints. Besides local requests for remote access, the remote access protocol is responsible for executing incoming requests issued by other nodes in the system. For one-sided operations, the protocol must not involve the CPU and it needs to load and subsequently inject the retrieved data into the fabric.

In this thesis, we propose and discuss the details of one such one-sided protocol – Scale-Out NUMA (soNUMA) [50, 126]. The soNUMA transport is layered on top of a NUMA fabric exposing a simple cache block request-reply interface. We also discuss RDMA as the state-of-the-art in one-sided communication. We use RDMA based on lossless Ethernet to evaluate the benefits of RSMP and, also, compare RDMA to soNUMA.

3.3.2 Fabric Interconnects

We define a fabric interconnect or fabric as a medium that carries network traffic in one-sided communication. One-sided protocols run on top of fabrics; they translate application requests for remote transfers into low-level request-reply fabric transactions. For example, a remote read request requires sending a packet to the server hosting the target data. Upon reception, the target's controller reads the requested data from the local memory, breaks it up into multiple packets and sends them to the requestor. The number of such packets depends on the *maximum transmission unit* (MTU) that the underlying fabric has been configured with.

Fabric interconnects define packet formats, headers and specific networking features that the transport protocol above uses to perform data transfers. At each layer of the stack, packets are encapsulated or decapsulated, depending on the direction. The header overhead can be significant depending on how complex the protocol is (e.g., how many features are supported) and the MTU size. For example, supporting routing across different subnets requires carrying network IDs, such as IP addresses, in every packet. Such feature is supported in Infiniband and lossless Ethernet fabrics. Finally, the smaller the MTU, the larger the header overhead, independent of the header size.

The MTU size is one of the determining factors for a fabric's performance. Large MTUs can amortize the significant header overhead of complex protocols, while smaller MTUs provide better propagation delay, which improves the latency of small transfers. We provide a comparison between the Infiniband and NUMA MTUs in §4.8. The MTU can be configured and varies from cache-block size (i.e., 64B) in NUMA to 8KB in Infiniband. Note that the physical interconnect can further break packets into smaller chunks of data (e.g., *flits* and *phits* in memory subsystems) before sending them across the wire.

A major requirement for one-sided communication is that the underlying fabric does not drop packets (i.e., lossless). In Infiniband, if a packet gets lost, the whole transfer is reinitiated at the transport layer and this can lead to livelocks [74]. To prevent such events, link-level flow control is a common solution that prevents buffer overflow in the switches and/or endpoints. Infiniband, for instance, implements credit-based flow control where each node keeps track

of empty space in downstream entities. One alternative to credit-based flow control is priority-based flow control (PFC), where endpoints and switches pause upstream entities using specific types of frames when an overflow is about to occur, and resume them when there is enough space to safely proceed with transfers. Such flow control mechanism is used in lossless Ethernet to ensure efficient RDMA. It has been shown that even flow-controlled fabrics can lose packets due to various hardware bugs and errors, which is why an efficient retransmission mechanism is required at the transport level [74].

3.3.3 Concurrency Models

RSMP requires a concurrency mechanism for correctness reasons. Accessing remote data using one-sided operations implies that conflicts with local CPU accesses at the target node can occur. For instance, if the target node is updating an object, a remote reader could read a partially updated version of that object. To prevent such inconsistent accesses, either a hardware or software synchronization mechanism is required.

Optimistic concurrency control (OCC) is one common way to minimize the number of round trips required to perform a remote transfer [58]. Under software OCC, the reader copies the requested data to its local memory and performs atomicity checks to verify that the object has not been modified during the transfer. In this thesis, we use a modified version of *fast and available remote memory* (FARM) [58] framework to provide the RSMP abstraction to applications. In FARM, OCC is implemented in software and assumes a data layout in which all cache blocks contain a version number. When an object is copied from remote to local memory, all the per-cache-block versions are compared against each other to detect atomicity violations that may have occured at the target node.

With the emergence of low-latency rack-scale fabrics, a hardware OCC mechanism will be of great importance [126]. Using a software mechanism in combination with today's Infiniband fabrics is a reasonable choice, given the relatively high latencies of such fabrics (i.e., >1us). In §4, we study an integrated fabric providing ramote access latency that is only a small factor higher than the local access latency. Due to such ulta-low access latency to remote memory, a software mechanism is not sufficient as the atomicity checks and extracting useful data takes

up a significant fraction of the end-to-end latency [50].

Finally, ACID transactions are necessary in RSMP to ensure consistent operations on multiple objects residing on different servers of the rack. An OCC mechanism guarantees linearizability, meaning accesses to a single object are serialized, whereas transactions guarantee serializability - interleaved accesses to a set of objects appear as if they were executed one at a time. FARM supports distributed transactions using a variant of the two-phase commit protocol. In FARM, the number of necessary remote operations is minimized through a combination of RDMA and standard RPC. This thesis, however, focuses on distributed NoSQL databases with weak consistency semantics that do not require support for ACID transactions. NoSQL databases typically assume the scale-out model, which is the model RSMP follows, where distributed ACID transactions are not practical [28].

3.4 Integrated RSMP

The impact of RSMP on datacenter's performance depends on the underlying fabric's performance characteristics. RDMA provides low-latency access to remote memory, and, thus, enables effective RSMP. For most applications where RSMP is applicable, RDMA is sufficiently fast to provide performance benefits as compared to standard scale-out deployments. To provide a more effective RSMP, one could think of a different kind of one-sided architecure that delivers lower latency and/or more bandwidth than existing RDMA implementations. The goal of such a new design should be to achieve the latency that is as close to the latency of a cache-coherent NUMA machine as possible. In this section, we will first identify the main performance bottlenecks in Infiniband/RDMA. Then, we will outline a set of design principles for building integrated one-sided fabrics that outperform the existing Infiniband implementations.

3.4.1 Obstacles to Low-Latency RSMP Using RDMA

RDMA is implemented through Infiniband, which assumes specialized discrete network cards, also known as *host channel adapters* (HCA), Converged Ethernet or Infiniband switches and physical links connecting the hosts. The Infiniband transport is implemented in the HCA

and is layered either over an existing IP/Ethernet network, for compatibility reasons, or an actual Infiniband network. For RDMA to work, the underlying fabric must support *lossless* communication. In IP/Ethernet networks, this is achieved through priority-based link-level flow control [6], whereas Infiniband fabrics rely on credit-based flow control [5] to provide the lossless property.

Because the entire network stack is implemented in hardware and the fact that only one endpoint is involved in one-sided communication, Infiniband delivers latencies on the order of only a few microseconds [58]. A hardware implementation of the transport allows applications to directly and at low overhead request remote data transfers from the locally-attached HCA via a set of registered memory-mapped queue pairs. The Infiniband transport implements one-sided operations and, thus, data can be retrieved from remote memory without involving the CPU of the owner node. HCAs access requested data from the local memory via DMA completions over the PCI bus and return it to the requestor using the fabric.

Even though Infiniband provides low-latency access to remote memory, its implementations are not well-aligned with the concept of RDMA. In particular, RDMA assumes one-sided communication that brings significant latency improvements over messaging, but initiating one-sided accesses dominates the end-to-end latency, primarily because of the PCI/DMA interface between the CPU and the HCA [18]. The PCI/DMA interface has been considered reasonably fast for decades in conventional TCP/IP networks, as the on-loaded network packet processing time dominates the end-to-end latency.

The second source of overhead is the switch that connects the servers within the rack. A single Ethernet switch adds up to 200ns of latency [114]. Also, the links may not provide sufficiently high signaling rate and/or width for applications to communicate at low latency and high bandwidth. More recent Infiniband signaling technologies, such as EDR, provide signaling rates up to 25Gbp/s per link.

Finally, protocol complexity affects the complexity of the HCA, increases the header overhead, and the amount of state kept in the network and its endpoints. In §4.8, we provide an analysis of different MTU sizes and header overheads to conclude that low complexity results in smaller header sizes and enables the use of smaller MTU, which reduces the end-to-end latency.

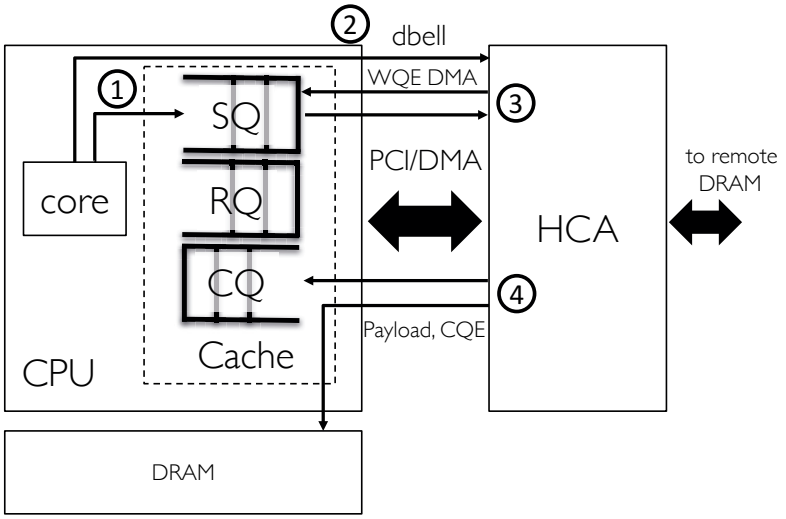


Figure 3.3 – PCIe/DMA interaction between a core and a host channel adapter (HCA).

PCIe/DMA. PCIe/DMA is a standard technique for connecting IO devices, such as graphic processing units, solid state disks, network interfaces, etc. The PCI bus connects the device to the CPU and the memory, allowing data transfers from the CPU caches or DRAM to the device and vice verse. The queue-pair-based interface (QP-based) in Infiniband leverages PCIe/DMA for transferring queue-pair entries between the application and the HCA. The HCA can be configured to accept QP entries via PCI writes or to pull QP entries using DMA. Which of the two mechanisms is more efficient depends on the number of round trips between the CPU and the device required for each application request. The size of the WQ entry often makes it more efficient to use DMA completions as opposed to direct PCI writes [89]. Nevertheless, the QP interaction over PCI/DMA can add up to 1us of delay in transferring data across the network [18].

When using PCIe writes, the software uses memory-mapped IO to write directly into the address space of the device. Another approach is to configure the controller to pull fresh WQ entries using PCIe completions upon doorbell notifications. The latter approach is illustrated on Fig. 3.3, where (i) the core creates one or multiple SQ entries, (ii) notifies the HCA via a doorbell, (iii) the HCA pulls fresh SQ entries via PCIe completions, and, finally, (iv) executes the requests, writes the requested data to memory (for reads) and notifies the application via a new CQ entry. This combination of PCI reads and completions are considered more bandwidth-efficient, as the packets in aggregate carry less header metadata, whereas direct

PCI writes often provide better latency.

Switches and Links. The second source of inefficiency is the switch and the links connecting the servers. A single Ethernet switch can add up to 200ns of latency [114], and, thus, the more servers we connect, the higher the latency (e.g., two racks require three switches in total). Link latency depends on the signaling rate, the width, and the MTU size. Infiniband has significantly improved since its first version based on Single Data Rate (SDR) with the latency of 5us. Currently, Infiniband *enhanced data rate* (EDR) can provide the latency of 0.61us and bandwidth of 100Gbps.

The signaling rate depends on how many bits can be transferred per clock cycle and the clock frequency; Quad Data Rate, for example, transfers 4 bits per cycle by using both rising and falling edges of the signal on two distinct wires (differential signaling). The available widths in Infiniband include 1, 4, and 12×.

Protocol complexity. Finally, HCAs are highly complex devices implementing an entire network stack. Such a network stack consists of transport, network, link, and physical layers. The transport layer is always Infiniband, while the rest of the stack is either an actual Infiniband network, consisting of Infiniband L3, L2, and PHY, or an IP/Ethernet network with support for link-level flow control (i.e., lossless Ethernet). IP over lossless Ethernet is commonly used in datacenters, because most of the traffic is still based on IP. HCAs are complex devices because Infiniband supports many features that add state and network packet header overhead. For instance, the requirement to establish connections between potentially each pair of QPs prior to accessing remote memory can lead to exceedingly large amount of state and thrashing from the HCA to DRAM [58].

Infiniband requires routing across subnets to support larger server deployments, which translates to bigger packet headers. Each packet also carries the information associated with the requested operation, such as the opcode, virtual address, etc. Thus, Infiniband headers can be as large as 98 bytes for RDMA operations [5], which is why it is important to support large MTUs in Infiniband and amortize the header overhead. The downside of large MTU headers is longer propagation delay - higher latency for small data transfers (see §4.8).

3.4.2 Integrated Fabric Interconnects

Tight integration can reduce the latency, improve bandwidth, and reduce complexity of one-sided fabrics. We first propose a set of design principles that lead to faster one-sided communication and reduce protocol complexity. In the next chapter, we describe Scale-Out NUMA (Fig. 3.4) an actual implementation of integrated one-sided fabric. Our focus is on rack-scale one-sided fabrics and we assume integration only within a single rack and not across multiple racks.

The main goal of integration is to: (i) improve the interaction between the software and the fabric controller, and (ii) improve transfer efficiency using one-sided primitives. We show that integration leads to a design that is much better aligned with the idea of one-sided communication, where servers can access each other's memory at a small factor of the local access latency. The reduced remote access latency leads to significant performance benefits through a better use of the fabric.

The first step towards a more efficient one-sided fabric is to design a lightweight hardware-software interface between the software and the controller. Such an interface should allow software to initiate remote transfers at much lower cost, such that a larger fraction of the end-to-end latency is attributed to the propagation delay of the interconnect, rather than the interface itself. A low-overhead interface is feasible through integration, where the fabric controller is a specialized and hardwired part of the CPU, allowing cores to interact with the controller faster than using PCI/DMA. Such an approach assumes removing the PCI bus, a mechanism that falls behind the concept of RDMA due to its overheads. By integrating the controller in the CPU, the cores can interact with the controller through cache-coherent shared memory and leverage the cache coherence protocol for core-to-controller transfers and vice verse.

Second, a faster one-sided fabric also necessitates the use of specialized network elements. This includes removing the switches and using integrated low-latency routers instead, as well as using multiple low-latency and high-bandwidth links to connect the nodes. A low-latency router in *glueless*/supercomputer-like fabrics can have a pin-to-pin delay of only a few tens of nanoseconds [118]. Then, low-latency and wide links can further improve transfer times.

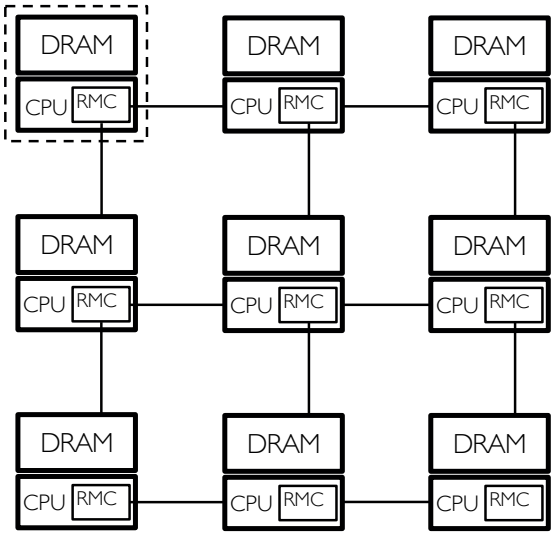


Figure 3.4 – A 2D mesh Scale-Out NUMA architecure.

In addition, connecting the nodes using high-degree network topologies, such as 3D torus, improves the bisection bandwidth. Integrated fabrics assume the use of a printed circuit board, which makes all the elements of the rack integrated.

Finally, a low-complexity transport protocol for one-sided access reduces the state and header overhead. To integrate a fabric controller into the CPU cache hierarchy, it is necessary to support a minimal number of features, which requires minimal state to be kept in the network endpoints and less metadata to be carried, along with the data, across the network (header). By reducing the header overhead, it is possible to reduce the MTU size and minimize the propagation delay.

Fig. 3.4 illustrates a 2D mesh Scale-Out NUMA (soNUMA) architecture that we cover in the next chapter. soNUMA is an integrated one-sided fabric design layering a low-latency and low-complexity one-sided protocol over a rack-scale NUMA fabric. Each node in a soNUMA fabric consists of a System-on-chip (SoC) with CPU cores, a *remote memory controller* (RMC), and local DRAM. RMC is the soNUMA's fabric controller that layers one-sided remote transfers over NUMA. The unique combination of a lightweight one-sided protocol and NUMA improves the remote access latency by an order of magnitude as compared to the state-of-the-art RDMA designs. In the next chapter, we discuss Scale-Out NUMA in detail and describe what it takes to build a faster alternative to RDMA.

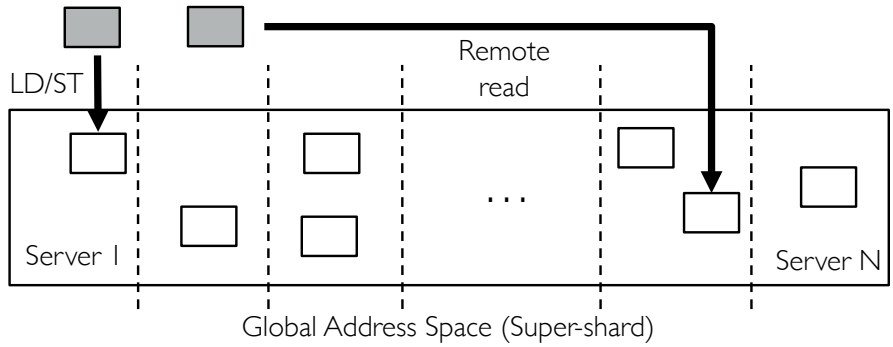


Figure 3.5 – A virtual global address space with objects spread across the N servers of an RSMP unit. Memcopy semantics for accessing both local and remote objects.

3.5 RSMP Software Framework

A key prerequisite for RSMP is a software framework that provides the abstraction of virtual global address spaces. Such a software framework is necessary to implement applications in a transparent way, while taking advantage of the RSMP architecture. By using the software framework, programmers do not have to spend too much time thinking about the intricacies of the underlying fabric and the architecture of the back-end. That is, the RSMP software framework provides: (i) an API for managing virtual global address spaces and (ii) an RPC mechanism with associated API for the client-server interaction. The server-side abstraction providing virtual global address spaces allows applications to transparently keep their data in the memory of multiple nodes connected using a one-sided fabric (i.e., RSMP unit). The client-side API allows applications to locate data and assign tasks (e.g., key lookup) to individual servers assuming shared access to data at the level of a rack.

Data layout. The default data layout in RSMP is *objects*, where each object is associated with a global virtual address and is accessed through the framework's API, independent of the object's location. Fig. 3.5 illustrates a global address space of stand-alone objects spread across an RSMP unit. Upon a request for access to a specific object, the RPC handler resolves the reference and issues a remote read if the object is not stored in the local memory. The fabric transfers the object to an application buffer and the framework returns a reference to the payload. It is up to the application programmer to decide whether the same object should be transferred repeatedly from remote memory or cached for some period of time.

An object in RSMP could be a key-value pair, a table row, graph vertex, etc. Depending on the kind of distributed NoSQL database layered on top of the framework, the RPC handler uses application-specific metadata, such as DHT ring, index, or adjacency matrix, to locate data in memory.

One of the downsides of the global address space abstraction is spatial locality; frequent access to remote objects might hurt performance. However, with the advance of high-performance one-sided fabrics, locality will become less of an issue. In the next chapter, we introduce a state-of-the-art one-sided fabric, delivering remote latency that is only a small factor higher than the local DRAM latency [126].

Programming interface and implementation. The RSMP software framework provides an RPC mechanism with the client-side API for requesting data structure-specific operations from the servers via TCP/IP, such as key-value *lookup* and *insert* in distributed hash tables. The RPC API assumes read-shared data access within each rack, allowing clients to take advantage of the RSMP architecture. The API helps clients understand the architecture of the back-end - which nodes are connected to each other and form RSMP units. The clients use the API to make smarter scheduling decisions leading to better performance as compared to the straightforward design in which work is distributed uniformly across an RSMP unit. In §6, we describe one such design for data serving applications. The current version of the RSMP software framework does not assume a networking abstraction where an RSMP unit is associated with a single IP address. A networking abstraction would further simplify application development and allow applications to think of an RSMP unit as one giant server.

RSMP software framework is based on FaRM [58], a distributed object store providing basic mechanisms that enable the *concurrent-read/exclusive-write* (CREW) concurrency model [102]. In particular, FaRM implements atomic remote object reads via RDMA using optimistic concurrency control that encodes 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 Scale-Out NUMA, propose to add hardware support for atomic, multi-cache line remote reads that eliminate the software overheads associated with atomicity checks and data layout [50].

	Client read API
RSMP	int RpcRead(server_endp_t server_endp, int rpc_type, buf_t params, rpc_callback_t AppCb)
FaRM	int AtomicRead(addr_t addr, size_t block_size, read_callback_t SearchDataCb)

Table 3.1 – Client read function comparison between RSMP and FARM. FARM assumes direct access to globally distributed data.

We perform three major modifications of FaRM: (i) FaRM has been 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 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 Scale-Out NUMA operations.

Table 3.1 shows the read function declarations on RSMP and FaRM. Because of the RSMP datacenter organization assumption, the RSMP software framework uses RPC to retrieve data from the servers via the TCP/IP network. Unlike RSMP, FaRM assumes a globally available RDMA fabric where any client can access any data [58]. FaRM clients typically use `AtomicRead` to retrieve a bigger chunk of data (e.g., table row) and then search for specific - requested item (e.g., column) locally, once the data has been transferred via RDMA. In RSMP, servers use `AtomicRead` from inside the RPC handler to retrieve data from the global address space of the local RSMP unit and return it to the client.

Using the framework API, it is possible to layer a distributed NoSQL database on top of the RSMP abstraction, such as key-value stores, sparse multi-dimensional maps, graph stores [31, 41, 58, 128]. In a nutshell, what distinguishes one NoSQL database from another is: (i) how memory gets allocated (e.g., tables, graphs, key-value pairs) and (ii) the type of metadata that the clients use to determine the location of data (i.e., which server / RSMP unit) - that the servers use to find the global virtual address associated with the requested data from inside the RPC handler.

Independent of the NoSQL database, the clients can use the knowledge of the back-end RSMP configuration (i.e., server groupings) to perform load balancing. In §5 and §6, we describe two different approaches to load balancing in a distributed key-value store, which we had previously implemented on top of the RSMP framework.

Run-to-completion model. The RSMP framework is optimized for latency and processes client requests on the server side in a *run-to-completion* manner [23], even though the QP-based interfaces of RDMA and Scale-Out NUMA support asynchronous remote operations. First, given the high overhead of processing inbound client requests (kernel and user space overheads), there is very little opportunity for pipelining; for example, Table 5.1 shows that the service time for local KVS lookups is 5us, which is well above the Scale-Out NUMA latency of only 240ns [50]. In the case of RDMA, using the asynchronous API may lead to a slight improvement in throughput, given the higher remote access latency of RDMA, but would make programming more complex [20]. Second, besides the performance aspect and more importantly, the run-to-completion model allows us to reason and carefully analyze application behavior. In §5, we present a queuing model for a KVS based on the RSMP framework. Assuming the run-to-completion model, we use the request processing times (i.e., service times) to instrument the model and accurately match the KVS's performance with the model results.

Consistency semantics. RSMP assumes the scale-out architecture, which means ACID transactions across different racks are not practical. Fortunately, NoSQL databases, unlike relational databases, often times imply only *put* and *get* operations on individual data items and, thus, do not require ACID transactions, but only need to guarantee linearizability [97]. Also, because of the scale-out assumption, NoSQL databases by definition do not provide strong consistency guarantees for replication (CAP theorem [29]), but rather implement eventual consistency models for availability reasons.

3.6 Summary

RSMP is a technique for aggregating memory at rack-scale using a fabric interconnect with support for one-sided operations. One can also think of RSMP as building fewer larger servers in a scale-out - datacenter envirnoment. A larger server in this context is a tightly interconnected cluster of nodes that can be perceived as a single capacity unit (i.e., server) in a datacenter. To build such clusters, we introduced the concept of one-sided operations and discussed fabric interconnects enabling low-latency one-sided transfers. One-sided operations bypass the OS

kernel and do not involve the owner of the target data - destination, thus providing a significant performance improvement over TCP/IP. We studied Infiniband as the state-of-the-art implementation of one-sided operations. Infiniband is a full network stack providing reliable delivery using link-level credit-based flow control.

We identified the major bottlenecks in existing RDMA implementations based on Infiniband. We identified PCI/DMA as the major source of overhead. Other sources include the switches, link signaling rate, network architecture, and complexity of the one-sided protocol. We, then, introduced integration as an effective approach to reducing the remote access latency, where the fabric controller resides on the chip and is cache coherent with the CPU cores, which minimizes the hardware-software interface overhead.

Finally, we discussed software support for RSMP. We proposed the RSMP software framework exposing the abstraction of a global address space to applications, with the object data layout. We described the API and how applications - NoSQL databases should be implemented on top of this API. We, also, described the run-to-completion model and the consistency semantics for distributed NoSQL databases running on top of RSMP.

4 Scale-Out NUMA: An Integrated RSMP Design

We introduce Scale-Out NUMA (soNUMA) - an architecture, programming model, and communication protocol for integrated, low-latency *rack-scale memory pooling* (RSMP). soNUMA layers an RDMA-inspired programming model directly on top of a NUMA memory fabric via a *stateless* messaging protocol. To facilitate interactions between the application, OS, and the fabric, soNUMA relies on the *remote memory controller* (RMC) - a new architecturally-exposed hardware block integrated into the node's local coherence hierarchy. Our results based on cycle-accurate full-system simulation show that soNUMA performs remote reads at latencies that are within 4x of local DRAM, can fully utilize the available DDR3 memory bandwidth using a single CPU core, and can issue up to 10M remote memory operations per second per core.

Because of the ability to access remote memory at a small factor of the local access latency, soNUMA enables highly effective RSMP. In this chapter, we focus on the architecture of soNUMA, and in the next chapter we show how RSMP using soNUMA can impact distributed key-value stores (KVS). §4.1 introduces the problem of low-latency RSMP. In §4.2 we discuss datacenter trends and obstacles to low-latency RSMP. We then describe the essential elements of the soNUMA architecture (§4.3), followed by a description of the design and implementation of the RMC (§4.4), the software support (§4.5), and the proposed communication protocol (§4.6). We evaluate our design (§4.7) and discuss additional aspects of the work (§4.8). Finally, we place soNUMA in the context of prior work (§4.9) and conclude (§4.10).

4.1 Introduction

Datacenter applications are rapidly evolving from simple data-serving tasks to sophisticated analytics operating over enormous datasets in response to real-time queries. To minimize the response latency, datacenter operators keep the data in memory. As dataset sizes push into the petabyte range, the number of servers required to house them in memory can easily reach into hundreds or even thousands.

Because of the distributed memory, applications that traverse large data structures (e.g., graph algorithms) or frequently access disparate pieces of data (e.g., key-value stores) must do so over the datacenter network. As today's datacenters are built with commodity networking technology running on top of commodity servers and operating systems, node-to-node communication delays can exceed 100μs [145]. In contrast, accesses to local memory incur delays of around 60ns – a factor of 1000 less. The irony is rich: moving the data from disk to main memory yields a 100,000x reduction in latency (10ms vs. 100ns), but distributing the memory eliminates 1000x of the benefit.

The reasons for the high communication latency are well known and include deep network stacks, complex network interface cards (NIC), and slow chip-to-NIC interfaces [70, 145]. RDMA reduces end-to-end latency by enabling memory-to-memory data transfers over InfiniBand [109] and Converged Ethernet [6] fabrics. By exposing remote memory at user-level and offloading network processing to the adapter, RDMA enables remote memory read latencies as low as 1.19μs [47]. Even though RDMA enables RSMP, its latency is still >10x higher than the local DRAM latency.

We introduce Scale-Out NUMA (soNUMA), an architecture, programming model, and communication protocol for distributed, in-memory applications that reduces the access latency to nearby remote memory (RSMP unit) to within a small factor (~4x) of local memory. soNUMA leverages two simple ideas to minimize latency. The first is to use a stateless request/reply protocol running over a NUMA memory fabric to drastically reduce or eliminate the network stack, complex NIC, and switch gear delays. The second is to integrate the protocol controller into the node's local coherence hierarchy, thus avoiding state replication and data movement across the slow PCI Express (PCIe) interface.

soNUMA exposes the abstraction of a partitioned global virtual address space, which is useful for big-data applications with irregular data structures such as graphs. The programming model is inspired by RDMA [116], with application threads making explicit remote memory read and write requests with copy semantics. The model is supported by an architecturally-exposed hardware block, called the *remote memory controller* (RMC), that safely exposes the global address space to applications. The RMC is integrated into each node's coherence hierarchy, providing for a frictionless, low-latency interface between the processor, memory, and the interconnect fabric.

Our primary contributions are:

- the RMC – a simple, hardwired, on-chip architectural block that services remote memory requests through locally cache-coherent interactions and interfaces directly with an on-die network interface. Each operation handled by the RMC is converted into a set of stateless request/reply exchanges between two nodes;
- a minimal programming model with architectural support, provided by the RMC, for one-sided memory operations that access a partitioned global address space. The model is exposed through lightweight libraries, which also implement communication and synchronization primitives in software;
- a preliminary evaluation of soNUMA using cycle-accurate full-system simulation demonstrating that the approach can achieve latencies within a small factor of local DRAM and saturate the available bandwidth using a single core;
- an soNUMA emulation platform built using a hypervisor that runs applications at normal wall-clock speeds and features remote latencies within 5x of what a hardware-assisted RMC should provide.

4.2 Why Scale-Out NUMA?

In this section, we discuss key trends in datacenter applications and servers, and identify specific pain points that affect the latency of such deployments.

4.2.1 Datacenter Trends

Applications. Today's massive web-scale applications, such as search or analytics, require thousands of computers and petabytes of storage [165]. Increasingly, the trend has been toward deeper analysis and understanding of data in response to real-time queries. To minimize the latency, datacenter operators have shifted hot datasets from disk to DRAM, necessitating terabytes, if not petabytes, of DRAM distributed across a large number of servers.

Recent work examining sources of network latency overhead in datacenters found that a typical deployment based on commodity technologies may incur over 100μs in round-trip latency between a pair of servers [145]. According to the study, principal sources of latency overhead include the operating system stack, NIC, and intermediate network switches. While 100μs may seem insignificant, we observe that many applications, including graph-based applications and those that rely on key-value stores, perform minimal computation per data item loaded. For example, read operations dominate key-value store traffic, and simply return the object in memory. With 1000x difference in data access latency between local DRAM (100ns) and remote memory (100μs), distributing the dataset, although necessary, incurs a dramatic performance overhead.

In addition to costly remote accesses, excessive horizontal scaling leads to significant load imbalance and higher probability of violating *service-level objectives* (SLO) [54, 127, 128]. In data serving workloads, for example, skewed key popularities emerge regularly and directly translate to load imbalance across the servers. Because of this, achieving high throughput without violating applications' tail latency objectives is a difficult problem. In addition, resource contention, garbage collection, energy management, maintenance activities, and other sources of performance interference, all increase the likelihood of tail latency violations. Amazon, for instance, reported that the rendering of a single page typically requires access to over 150 services, which is why achieving low tail latency is important [56]. These interactions introduce significant overheads that constrain the practical extent of sharding and the complexity of deployed algorithms. Also, tail latency considerations force Facebook to restrict the number of sequential data accesses to fewer than 150 per rendered web page [145]. Therefore, to provide the required quality of service and, at the same time, avoid wasting server resources, it

is necessary to reduce the number of participating nodes through RSMP or similar techniques.

Server architectures. Today's datacenters employ commodity technologies due to their favorable cost-performance characteristics. The end result is a *scale-out* architecture characterized by a large number of commodity servers connected via commodity networking equipment. Two architectural trends are emerging in scale-out designs.

First, System-on-Chips (SoC) provide high chip-level integration and are a major trend in servers. Current server SoCs combine many processing cores, memory interfaces, and I/O to reduce cost and improve overall efficiency by eliminating extra system components. For example, Calxeda's ECX-1000 SoC [36] combines four ARM Cortex-A9 cores, memory controller, SATA interface, and a fabric switch [35] into a compact die with a 5W typical power draw.

Second, system integrators are starting to offer *glueless fabrics* that can seamlessly interconnect hundreds of server nodes into fat-tree or torus topologies [57]. For instance, Calxeda's on-chip fabric router encapsulates Ethernet frames while energy-efficient processors run the standard TCP/IP and UDP/IP protocols as if they had a standard Ethernet NIC [52]. The tight integration of NIC, routers and fabric leads to a reduction in the number of components in the system (thus lowering cost) and improves energy efficiency by minimizing the number of chip crossings. However, such glueless fabrics typically do not reduce the end-to-end latency, because of the high cost of protocol processing at the end points.

Remote DMA. RDMA enables memory-to-memory data transfers across the network without processor involvement on the destination side. By exposing remote memory and reliable connections directly to user-level applications, RDMA eliminates all kernel overheads. Furthermore, one-sided remote memory operations are handled entirely by the adapter without interrupting the destination core. RDMA is supported on lossless fabrics such as InfiniBand [138] and Converged Ethernet [86] that scale to hundreds of nodes and can offer remote memory read latency as low as $1.19\mu s$ [47]. Although historically associated with the high-performance computing market, RDMA is now making inroads into web-scale datacenters, such as Microsoft Bing [152]. Latency-sensitive key-value stores such as RAMCloud [132] and Pilaf [117] are using RDMA fabrics to achieve object access latencies of as low as $5\mu s$.

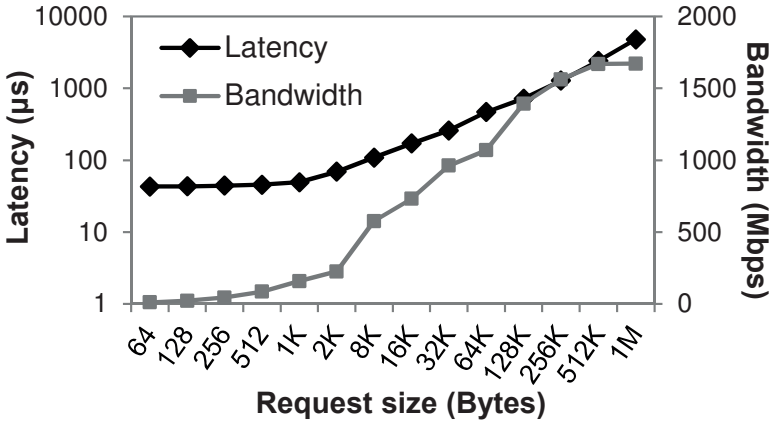


Figure 4.1 – Netpipe benchmark on a Calxeda microserver.

4.2.2 Obstacles to Low-Latency Distributed Memory

As datasets grow, the trend is toward more sophisticated algorithms at ever-tightening latency bounds. While SoCs, glueless fabrics, and RDMA technologies help lower network latencies within the rack, the network delay per byte loaded remains high. Here, we discuss principal reasons behind the difficulty of further reducing the latency for in-memory applications.

Node scalability is power-limited. As voltage scaling grinds to a halt, future improvements in compute density at the chip level will be limited. Power limitations will extend beyond the processor and impact the amount of DRAM that can be integrated in a given unit of volume (which governs the limits of power delivery and heat dissipation). Together, power constraints at the processor and DRAM levels will limit the server industry's ability to improve the performance and memory capacity of scale-up configurations, thus accelerating the trend toward distributed memory systems.

Deep network stacks are costly. Distributed systems rely on networks to communicate. Unfortunately, today's deep network stacks require a significant amount of processing per network packet which factors considerably into end-to-end latency. Figure 4.1 shows the network performance between two Calxeda EnergyCore ECX-1000 SoCs, measured using the `netpipe` benchmark [154]. The fabric and the NICs provide 10Gbps worth of bandwidth.

Despite the immediate proximity of the nodes and the lack of intermediate switches, we observe high latency (in excess of 40μs) for small packet sizes and poor bandwidth scalability (under 2 Gbps) with large packets. These bottlenecks exist due to the high processing requirements of TCP/IP and are aggravated by the limited performance offered by ARM cores.

Large-scale shared memory is prohibitive. One way to bypass complex network stacks is through direct access to shared physical memory. Unfortunately, large-scale sharing of physical memory is challenging for two reasons. First is the sheer cost and complexity of scaling up hardware coherence protocols. Chief bottlenecks here include state overhead, high bandwidth requirements, and verification complexity. The second is the fault-containment challenge of a single operating system instance managing a massive physical address space, whereby the failure of any one node can take down the entire system by corrupting shared state [42]. Sharing caches even within the same socket can be expensive. Indeed, recent work shows that partitioning a single many-core socket into multiple coherence domains improves the execution efficiency of scale-out workloads that do not have shared datasets [111].

PCIe/DMA latencies limit performance. I/O bypass architectures have successfully removed most sources of latency except the PCIe bus. Studies have shown that it takes 400-500ns to communicate short bursts over the PCIe bus [70], making such transfers 7-8x more expensive, in terms of latency, than local DRAM accesses. Furthermore, PCIe does not allow for the cache-coherent sharing of control structures between the system and the I/O device, leading to the need of replicating system state such as page tables into the device and system memory. In the latter case, the device memory serves as a cache, resulting in additional DMA transactions to access the state. SoC integration alone does not eliminate these overheads, since IP blocks often use DMA internally to communicate with the main processor [24].

Switch latency and link signaling rate. A single Ethernet switch adds up to 200ns of latency [114], and, thus, the more servers we connect, the higher the latency (e.g., two racks require three switches in total). The link latency depends on the signaling rate, the width, and the MTU size.

Infiniband, for instance, has improved substantially since its first version based on *single data rate* (SDR) with the latency of 5us. Currently, Infiniband *enhanced data rate* (EDR) can provide the latency of 0.6us and bandwidth of 100Gbps.

State and header overhead. RDMA network interfaces (HCA) are complex devices because Infiniband supports many features that add state and network packet header overhead. For instance, the requirement to establish connections between potentially each pair of QPs prior to accessing remote memory can lead to exceedingly large amount of state and thrashing from the HCA to DRAM [58]. Also, in Infiniband, each packet carries the information associated with the requested operation, such as the opcode, virtual address, network IDs (if routing across L2 domains), etc. Thus, Infiniband headers can be as large as 98 bytes for RDMA operations [5], which is why it is important to support large MTUs in Infiniband and amortize the header overhead. The downside is that large MTUs increase the propagation delay for small transfers.

4.3 Scale-Out NUMA

This work introduces soNUMA, an architecture and programming model for low-latency RSMP. soNUMA addresses each of the obstacles to low-latency described in §4.2.2. soNUMA is designed for a scale-out model with physically distributed processing and memory: (i) it replaces deep network stacks with a lean memory fabric supporting simple headers and small MTU (64B), low-latency low-radix switches; (ii) eschews system-wide coherence in favor of a global partitioned virtual address space accessible via RMDA-like remote memory operations with copy semantics; (iii) replaces transfers over the slow PCIe bus with cheap cache-to-cache transfers; and (iv) is optimized for rack-scale deployments (RSMP), where distance is minuscule. In effect, our design goal is to borrow the desirable qualities of ccNUMA and RDMA without their respective drawbacks.

Fig. 4.2 identifies the essential components of soNUMA. At a high level, soNUMA combines a lean memory fabric with an RDMA-like programming model in a rack-scale system. The figure shows a 2D mesh soNUMA, but any other topology, such as 2D or 3D torus, can be used.

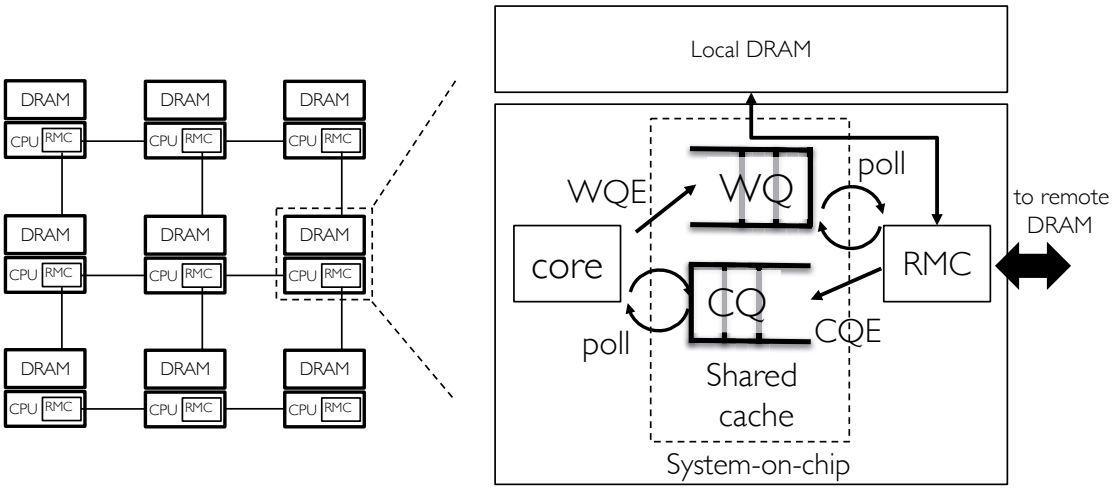


Figure 4.2 – Scale-Out NUMA fabric and node architecture. Software communicates with the local RMC through cache-to-cache transactions.

Applications access remote portions of the global virtual address space through remote memory operations. A new architecturally-exposed block, the *remote memory controller* (RMC), converts these operations into network transactions and directly performs the memory accesses. Applications directly communicate with the RMC using a memory-mapped queue pair, bypassing the operating system, which gets involved only in setting up the necessary in-memory control data structures.

Unlike traditional implementations of RDMA, which operate over the PCI bus, the RMC benefits from a tight integration into the processor's cache coherence hierarchy. In particular, the processor and the RMC share the control data structures, such as work queues (WQ), completion queues (CQ), page tables, and application data, via the cache hierarchy. The implementation of the RMC is further simplified by limiting the architectural support to one-sided remote memory read, write, and atomic operations, and by unrolling multi-line requests at the source RMC, thus, minimizing the state and header overheads. As a result, the protocol can be implemented in a stateless manner by the destination node.

The RMC converts application commands into remote requests that are sent to the *network interface* (NI). The NI is connected to an on-chip low-radix router with reliable, point-to-point links to other soNUMA nodes. The notion of fast low-radix routers borrows from supercomputer interconnects; for instance, the mesh fabric of the Alpha 21364 connected 128

nodes in a 2D torus using an on-chip router with a pin-to-pin delay of just 11ns [118].

soNUMA's memory fabric bears semblance (at the link and network layer, but not at the transport layer) to the QPI and HTX solutions that interconnect sockets together into multiple NUMA domains. In such fabrics, parallel transfers over traces minimize pin-to-pin delays, short messages (header + a payload of a single cache line) minimize buffering requirements and the propagation delay, topology-based routing eliminates costly CAM or TCAM lookups, and virtual lanes ensure deadlock freedom. Although Fig. 4.2 illustrates a 2D-torus, the design is not restricted to any particular topology.

4.4 Remote Memory Controller

The foundational component of soNUMA is the RMC, an architectural block that services remote memory accesses originating at the local node, as well as incoming requests from remote nodes. The RMC integrates into the processor's coherence hierarchy via a private L1 cache and communicates with the application threads via memory-mapped queues. We first describe the software interface (§4.4.1), provide a functional overview of the RMC (§4.4.2), and describe its microarchitecture (§4.4.3).

4.4.1 Hardware/Software Interface

soNUMA provides application nodes with the abstraction of globally addressable, virtual address spaces that can be accessed via explicit memory operations. The RMC exposes this abstraction to applications, allowing them to safely and directly copy data to/from global memory into a local buffer using remote write, read, and atomic operations, without kernel intervention. The interface offers atomicity guarantees at the cache-line granularity, and no ordering guarantees within or across requests.

soNUMA's hardware/software interface is centered around four main abstractions directly exposed by the RMC: (i) the context identifier (`ctx_id`), which is used by all nodes participating in the same application to create a global address space; (ii) the context segment, a range of the node's address space which is globally accessible by others; (iii) the queue pair (QP), used

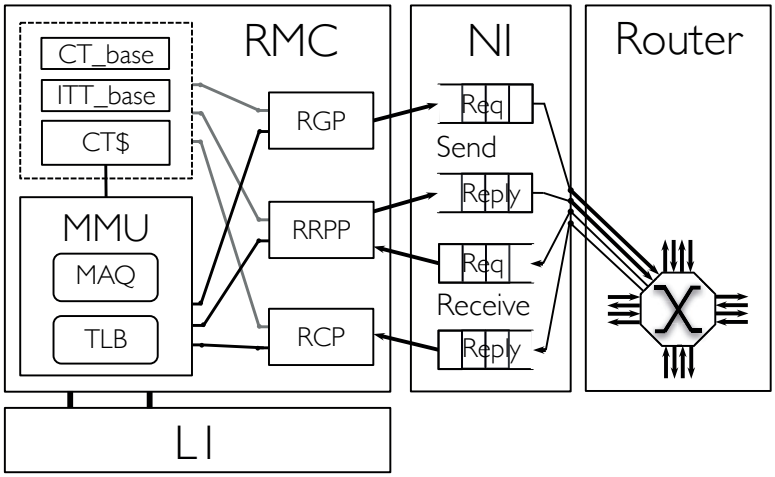


Figure 4.3 – RMC's internals: all memory requests of the three pipelines access the cache via the MMU. The CT_base register, the ITT_base register, and the CT$ offer fast access to the basic control structures.

by applications to schedule remote memory operations and get notified of their completion; and (iv) local buffers, which can be used as the source or destination of remote operations.

The QP model consists of a work queue (WQ), a bounded buffer written exclusively by the application, and a completion queue (CQ), a bounded buffer of the same size written exclusively by the RMC. The CQ entry contains the index of the completed WQ request. Both are stored in main memory and coherently cached by the cores and the RMC alike. In each operation, the remote address is specified by the combination of `<node_id, ctx_id, offset>`. Other parameters include the length and the local buffer address.

4.4.2 RMC Overview

The RMC consists of three hardwired pipelines that interact with the queues exposed by the hardware/software interface and with the NI. These pipelines are responsible for request generation, remote request processing, and request completion, respectively. They are controlled by a configuration data structure, the *Context Table* (CT), and leverage an internal structure, the *Inflight Transaction Table (ITT)*.

The CT is maintained in memory as a table and is initialized by the RMC device driver (see §4.5.1). The CT keeps track of all registered context segments, queue pairs, base addresses,

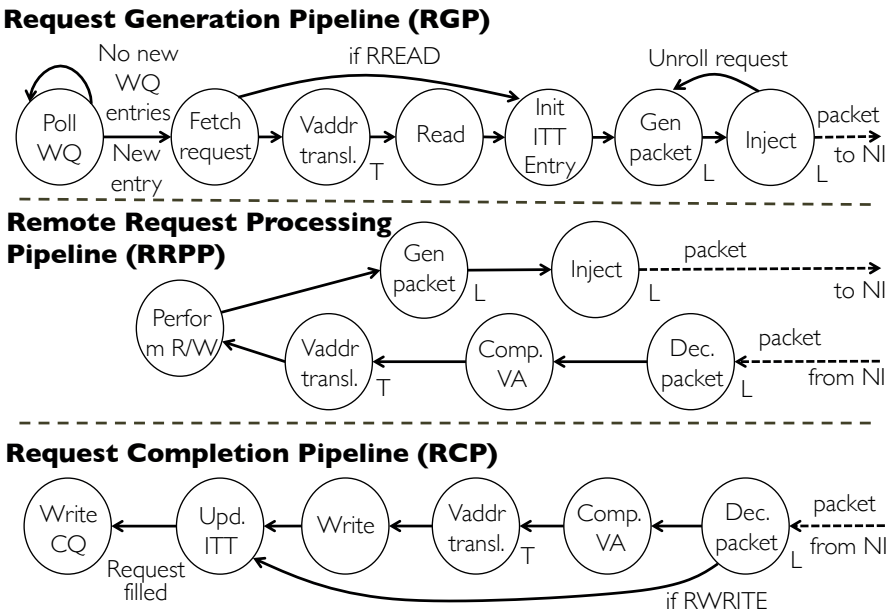


Figure 4.4 – RMC internal architecture and functional overview of the three pipelines.

and page table root addresses. Each CT entry – table row, indexed by its `ctx_id`, specifies the address space and a list of registered QPs (WQ, CQ) for that context. Multi-threaded processes can register multiple QPs for the same address space and `ctx_id`. The RMC driver preallocates as many QP entries in each row as there are physical CPU cores, assuming one QP per core per context in the extreme case. The QP entries in CT store the physical addresses of the queues and the corresponding head/tail indices. Storing physical addresses instead of virtual and bypassing the TLB is necessary to enable low-overhead polling on the queues. The RMC driver also supports sharing of QPs across different contexts within the same address space by reusing the physical addresses of previously allocated QPs when registering new contexts.

Multiple processes are also allowed to register contexts and access remote memory concurrently, but there should be no overlap among registered `ctx_id`s, within or across processes. If the driver tries to register a new context with a `ctx_id` that has been already taken, the RMC will return an error. The reason why there should be no overlap among `ctx_id`s is because the RMC uses the `ctx_id` as the index for the CT.

The RMC TLB supports *address space identifiers* (ASID), which is similar to process identifiers (PID), but usually they have fewer bits (e.g., 8 bits for the ASID versus 32 bits for a PID). The RMC uses the received `ctx_id` (either from the application or another node in the system)

to look up the page table root address in CT, and uses the 8 least significant bits as the ASID for the address space that the context is associated with. With ASID, the system avoids the overhead of TLB shootdowns for contexts residing within the same address space. Meanwhile, the ITT is used exclusively by the RMC and keeps track of the progress of each WQ request.

Fig. 4.3 shows the high-level internal organization of the RMC and its NI. The three pipelines are connected to distinct queues of the NI block, which is itself connected to a low-radix router block with support for two virtual lanes. While each of the three pipelines implements its own datapath and control logic, all three share some common data structures and hardware components. For example, they arbitrate for access to the common L1 cache via the MMU.

Fig. 4.4 highlights the main states and transitions for the three independent pipelines. Each pipeline can have multiple transactions in flight. Most transitions require an MMU access, which may be retired in any order. Therefore, transactions will be reordered as they flow through a pipeline.

Request Generation Pipeline (RGP). The RMC initiates remote memory access transactions in response to an application's remote memory requests (reads, writes, atomics). To detect such requests, the RMC polls on each registered WQ. Upon a new WQ request, the RMC generates one or more network packets using the information in the WQ entry. For remote writes and atomic operations, the RMC accesses the local node's memory to read the required data, which it then encapsulates into the generated packet(s). For each request, the RMC generates a transfer identifier (`tid`) that allows the source RMC to associate replies with requests.

Remote transactions in soNUMA operate at cache line granularity. Coarser granularities, in cache-line-sized multiples, can be specified by the application via the `length` field in the WQ request. The RMC unrolls multi-line requests in hardware, generating a sequence of line-sized read or write transactions. To perform unrolling, the RMC uses the ITT, which tracks the number of completed cache-line transactions for each WQ request and is indexed by the request's `tid`.

Remote Request Processing Pipeline (RRPP). This pipeline handles incoming requests originating from remote RMCs. The soNUMA protocol is stateless, which means that the RRPP can process remote requests using only the values in the header and the local configuration state. Specifically, the RRPP uses the `ctx_id` to access the CT, computes the ASID and the virtual address by adding the offset to the base virtual address, translates it to the corresponding physical address, and then performs a read, write, or atomic operation as specified in the request. The RRPP always completes by generating a reply message, which is sent to the source. Virtual addresses that fall outside of the range of the specified security context are signaled through an error message, which is propagated to the offending thread in a special reply packet and delivered to the application via the CQ.

Request Completion Pipeline (RCP). This pipeline handles incoming message replies. The RMC extracts the `tid` and uses it to identify the originating WQ entry. For reads and atomics, the RMC then stores the payload into the application's memory at the virtual address specified in the request's WQ entry. For multi-line requests, the RMC computes the target virtual address based on the buffer base address specified in the WQ entry and the offset specified in the reply message.

The ITT keeps track of the number of completed cache-line requests. Once the last reply is processed, the RMC signals the request's completion by writing the index of the completed WQ entry into the corresponding CQ and moving the CQ head pointer. Requests can therefore complete out of order and, when they do, are processed out of order by the application. Remote write acknowledgments are processed similarly to read completions, although remote writes naturally do not require an update of the application's memory at the source node.

The RCP pipeline is responsible for detecting failures. If a node does not respond or if a single packet does not arrive in a preconfigured amount of time, the RCP notifies the application by passing the corresponding error code and node ID (in case of node failures) via the CQ. This is implemented through an equivalent of the x86's time stamp counter (TSC) and the ITT storing transactions' start times.

4.4.3 Microarchitectural Support

The RMC implements the logic described above using a set of completely decoupled pipelines, affording concurrency in the handling of different functions at low area and design cost. The RMC features two separate interfaces: a coherent memory interface to a private L1 cache and a network interface to the on-die router providing system-level connectivity. The memory interface block (MMU) contains a TLB for fast access to recent address translations, required for all accesses to application data. TLB entries are tagged with address space identifiers corresponding to the application context. TLB misses are serviced by a hardware page walker. The RMC provides two interfaces to the L1 cache – a conventional word-wide interface as well as a cache-line-wide interface. The former is used to interact with the application and to perform atomic memory operations. The latter enables efficient atomic reads and writes of entire cache lines, which is the granularity of remote memory accesses in soNUMA.

The RMC's integration into the node's coherence hierarchy is a critical feature of soNUMA that eliminates wasteful data copying of control structures, and of page tables in particular. It also reduces the latency of the application/RMC interface by eliminating the need to set up DMA transfers of ring buffer fragments. To further ensure high throughput and low latency at high load, the RMC allows multiple concurrent memory accesses in flight via a *memory access queue* (MAQ). The MAQ handles all memory read and write operations, including accesses to application data, WQ and CQ interactions, page table walks, as well as ITT and CT accesses. The number of outstanding operations is limited by the number of miss status handling registers at the RMC's L1 cache. The MAQ supports out-of-order completion of memory accesses and provides store-to-load forwarding.

Each pipeline has its own arbiter that serializes the memory access requests from the pipeline's several stages and forwards the requests to the MAQ. The latter keeps track of each request's originating arbiter, and responds to that once the memory access is completed. Upon such a response, the arbiter feeds the data to the corresponding pipeline stage.

Finally, the RMC dedicates two registers for the CT and ITT base addresses, as well as a small lookaside structure, the *CT cache (CT$)* that caches recently accessed CT entries to reduce pressure on the MAQ. The CT$ includes the context segment base addresses and bounds, PT

roots, and the queue addresses, including the queues' head and tail indices. The base address registers and the CT$ are read-only-shared by the various RMC pipeline stages.

4.5 Software Support

We now describe the system and application software support required to expose the RMC to applications and enable the soNUMA programming model. §4.5.1 describes the operating system device driver. §4.5.2 describes the lower-level wrappers that efficiently expose the hardware/software interface to applications. Finally, §4.5.3 describes higher-level routines that implement unsolicited communication and synchronization without additional architectural support.

4.5.1 Device Driver

The role of the operating system on an soNUMA node is to establish the global virtual address spaces. This includes the management of the context namespace, virtual memory, QP registration, etc. The RMC device driver manages the RMC itself, responds to application requests, and interacts with the virtual memory subsystem to allocate and pin pages in physical memory. The RMC device driver is also responsible for allocating the CT and ITT on behalf of the RMC.

Unlike a traditional RDMA NIC, the RMC has direct access to the page tables managed by the operating system, leveraging the ability to share cache-coherent data structures. As a result, the RMC and the application both operate using virtual addresses of the application's process once the data structures have been initialized.

The RMC device driver implements a simple security model in which access control is granted on a per `ctx_id` basis. To join a global address space `<ctx_id>`, a process first opens the device `/dev/rmc_contexts/<ctx_id>`, which requires the user to have appropriate permissions. All subsequent interactions with the operating system are done by issuing `ioctl` calls via the previously-opened file descriptor. In effect, soNUMA relies on the built-in operating system mechanism for access control when opening the context, and further assumes that all operating system instances of an soNUMA fabric are under a single administrative domain.

4.5.2 Access Library

The QPs are accessed via a lightweight API, a set of C/C++ inline functions that issue remote memory commands and synchronize by polling the completion queue. We expose a synchronous (blocking) and an asynchronous (non-blocking) set of functions for both reads and writes. The asynchronous API is comparable in terms of functionality to the Split-C programming model [48].

Fig. 4.5 illustrates the use of the asynchronous API for the implementation of the classic PageRank graph algorithm [136]. `rmc_wait_for_slot` processes CQ events (calls `pagerank_async` for all completed slots) until the head of the WQ is free. It then returns the freed slot where the next entry will be scheduled. `rmc_read_async` (similar to Split-C's `get`) requests a copy of a remote vertex into a local buffer. Finally, `rmc_drain_cq` waits until all outstanding remote operations have completed while performing the remaining callbacks.

This programming model is efficient as: (i) the callback (`pagerank_async`) does not require a dedicated execution context, but instead is called directly within the main thread; (ii) when the callback is an inline function, it is passed as an argument to another inline function (`rmc_wait_for_slot`), thereby enabling compilers to generate optimized code without any function calls in the inner loop; (iii) when the algorithm has no read dependencies (as is the case here), asynchronous remote memory accesses can be fully pipelined to hide their latency.

To summarize, soNUMA's programming model combines true shared memory (by the threads running within a cache-coherent node) with explicit remote memory operations (when accessing data across nodes). In the PageRank example, the `is_local` flag determines the appropriate course of action to separate intra-node accesses (where the memory hierarchy ensures cache coherence) from inter-node accesses (which are explicit).

Finally, the RMC access library exposes atomic operations such as compare-and-swap and fetch-and-add as inline functions. These operations are executed atomically within the local cache coherence hierarchy of the destination node.

```
float *async_dest_addr[MAX_WQ_SIZE];
Vertex lbuf[MAX_WQ_SIZE];

inline void pagerank_async(int slot, void *arg) {
  *async_dest_addr[slot] += 0.85 *
    lbuf[slot].rank[superstep%2] / lbuf[slot].out_degree;
}

void pagerank_superstep(QP *qp) {
  int evenodd = (superstep+1) % 2;
  for(int v=first_vertex; v<=last_vertex; v++) {
    vertices[v].rank[evenodd] = 0.15 / total_num_vertices;
    for(int e=vertices[v].start; e<vertices[v].end; e++) {
      if(edges[e].is_local) {
      // shared memory model
      Vertex *v2 = (Vertex *)edges[e].v;
      vertices[v].rank[evenodd] += 0.85 *
        v2->rank[superstep%2] / v2->out_degree;
      } else {
        // flow control
        int slot = rmc_wait_for_slot(qp, pagerank_async);
        // setup callback arguments
        async_dest_addr[slot] = &vertices[v].rank[evenodd];
        // issue split operation
        rmc_read_async(qp, slot,
                edges[e].nid,     //remote node ID
                edges[e].offset,  //offset
                &lbuf[slot],      //local buffer
                sizeof(Vertex));  //len
      }
    }
  }
  rmc_drain_cq(qp, pagerank_async);
  superstep++;
}
```

Figure 4.5 – Computing a PageRank superstep in soNUMA through a combination of remote memory accesses (via the asynchronous API) and local shared memory.

4.5.3 Messaging and Synchronization Library

By providing architectural support for only read, write and atomic operations, soNUMA reduces hardware cost and complexity. The minimal set of architecturally-supported operations is not a limitation, however, as many standard communication and synchronization primitives can be built in software on top of these three basic primitives. In contrast, RDMA provides hardware support (in adapters) for unsolicited send/receive messages and remote read/write/atomic on top of reliable connections, thus introducing significant complexity (e.g., per-connection state) into the design [142].

Unsolicited communication. To communicate using `send` and `receive` operations, two application instances must first each allocate a bounded buffer from their own portion of the global virtual address space. The sender always writes to the peer's buffer using `rmc_write` operations, and the content is read locally from cached memory by the receiver. Each buffer is an array of cache-line sized structures that contain header information (such as the length, memory location, and flow-control acknowledgements), as well as an optional payload. Flow-control is implemented via a credit scheme that piggybacks existing communication.

For small messages, the sender creates packets of predefined size, each carrying a portion of the message content as part of the payload. It then *pushes* the packets into the peer's buffer. To receive a message, the receiver polls on the local buffer. In the common case, the send operation requires a single `rmc_write`, and it returns without requiring any implicit synchronization between the peers. A similar messaging approach based on remote writes outperforms the default send/receive primitives of InfiniBand [108].

For large messages stored within a registered global address space, the sender only provides the base address and size to the receiver's bounded buffer. The receiver then *pulls* the content using a single `rmc_read` and acknowledges the completion by writing a zero-length message into the sender's bounded buffer. This approach delivers a direct memory-to-memory communication solution, but requires synchronization between the peers.

At compile time, the user can define the boundary between the two mechanisms by setting a minimal message-size threshold: push has lower latency since small messages complete through a single `rmc_write` operation and also allows for decoupled operations. The pull mechanism leads to higher bandwidth since it eliminates the intermediate packetization and copy step.

Barrier synchronization. We have also implemented a simple barrier primitive such that nodes sharing a `ctx_id` can synchronize. Each participating node broadcasts the arrival at a barrier by issuing a `write` to an agreed upon offset on each of its peers. The nodes then poll locally until all of them reach the barrier.

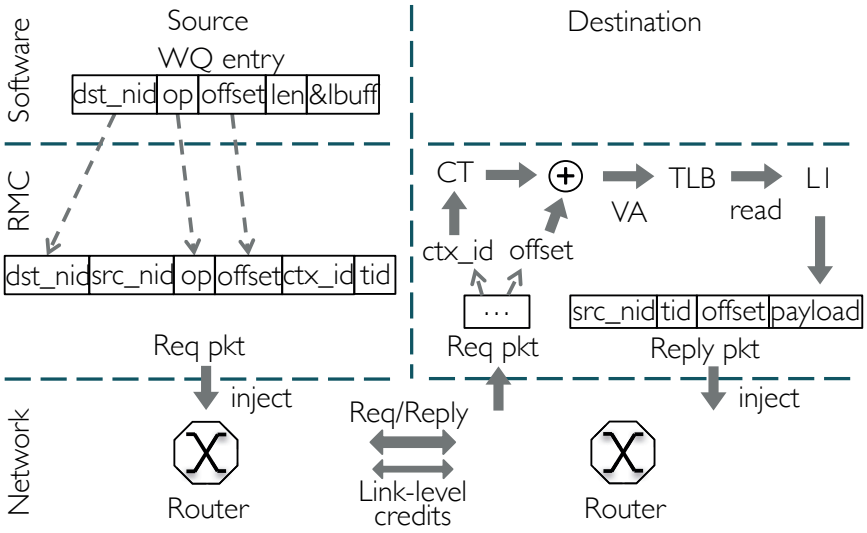


Figure 4.6 – Communication protocol for a remote read.

4.6 Communication Protocol

soNUMA's communication protocol naturally follows the design choices of the three RMC pipelines at the protocol layer. At the link and routing layers, our design borrows from existing memory fabric architectures (e.g., QPI or HTX) to minimize pin-to-pin delays.

Link layer. The memory fabric delivers messages reliably over high-speed point-to-point links with credit-based flow control. The message MTU is large enough to support a fixed-size header and an optional payload. Each point-to-point physical link has two virtual lanes to support deadlock-free request/reply protocols.

Routing (Network) layer. The routing-layer header contains the destination and source address of the nodes in the fabric (`<dst_nid, src_nid>`). `dst_nid` is used for routing, and `src_nid` to generate the reply packet.

The router's forwarding logic directly maps destination addresses to outgoing router ports, eliminating expensive CAM or TCAM lookups found in networking fabrics. While the actual choice of topology depends on system specifics, low-dimensional k-ary n-cubes (e.g., 3D torii) seem well-matched to rack-scale deployments [57].

Transport layer. The RMC transport protocol is a simple request-reply protocol, with exactly one reply message generated for each request. The WQ entry specifies the `dst_nid`, the command (e.g., `read`, `write`, or `atomic`), the `offset`, the length and the local buffer address. The RMC copies the `dst_nid` into the routing header, determines the `ctx_id` associated with the WQ, and generates the `tid`. If the `ctx_id` is not specified by the library, the RMC will access the context that the WQ is associated with. If the WQ is associated with more than one context, the application needs to specify which context it would like to access.

The `tid` serves as an index into the ITT and allows the source RMC to map each reply message to a WQ and the corresponding WQ entry. The `tid` is opaque to the destination node, but is transferred from the request to the associated reply packet. The `tid`, `ctx_id`, offset, and opcode are a part of the transport header, whereas `src_id` and `dst_id` belong to the routing header. The routing layer, along with the layers underneath belong to the NUMA fabric (e.g., QPI). The transport layer on soNUMA is custom and replaces the cache coherence protocol of ccNUMA.

Fig. 4.6 illustrates the actions taken by the RMCs for a remote read of a single cache line. The RGP in the requesting side's RMC first assigns a `tid` for the WQ entry and the `ctx_id` corresponding to that WQ. The RMC specifies the destination node via a `dst_nid` field of the routing header. The request packet is then injected into the fabric and the packet is delivered to the target node's RMC. The receiving RMC's RRPP decodes the packet, computes the local virtual address using the `ctx_id` and the `offset` found in it and translates that virtual address to a physical address. For correctness reasons, the RMC uses the concept of *address space identifiers* (ASID) to pick the right virtual-to-physical translation.

This stateless handling does not require any software interaction on the destination node. As soon as the request is completed in the remote node's memory hierarchy, its RMC creates a reply packet and sends it back to the requesting node. Once the reply arrives to the original requester, the RMC's RCP completes the transaction by writing the payload into the corresponding local buffer and by notifying the application via a CQ entry (not shown in Fig. 4.6).

4.6.1 Dealing with packet loss and node failures

Reliable delivery of packets is commonly enforced at the link and transport layers. In TCP/IP networks, the TCP layer is responsible for flow control, congestion management, retransmissions in case of packet loss, connection management, etc. Most of TCP processing is onloaded to the CPU, which makes commodity TCP networks a bad candidate for RSMP because of high remote access latency.

In RDMA networks, the Infiniband transport layer is implemented inside the NIC (i.e., HCA), thus, significantly reducing the CPU load and improving the latency. The Infiniband transport layer assumes lossless links and, thus, implements a trivial *back-to-0* algorithm for retransmissions [74]. Therefore, IB requires either a Converged Ethernet network with priority-based flow control (PFC) or an actual Infiniband network with support for credit-based flow control at the link layer to support lossless delivery.

A recent study on full-scale datacenter RDMA networks has shown that lossless links, in fact, do lose packets [74, 170], because of frame check sequence erros and bugs in switch hardware and software, concluding that a smarter retransmission mechanism is necessary for large - datacenter scale deployments.

NUMA fabrics implement credit-based flow control at the link layer, guaranteeing lossless delivery under congestion. One of the goals of soNUMA is to scale beyond what is possible (or affordable) with ccNUMA, which inspired us to think about failures. A failure on soNUMA can lead to SoC socket loss or, less dramatic, packet loss. In both cases, again, to simplify the design of the RMC, we expose failures to applications directly, letting them decide what action to take when nodes do not respond or there are unexpectedly long running remote transfers. The RMC notifies the application framework via the CQ and passes an error code signaling node or packet loss. The default routing algorithm on soNUMA is destination based and individual node failures may appear as if multiple nodes have failed. To deal with this problem, a more sophisticated adaptive algorithm for routing is required.

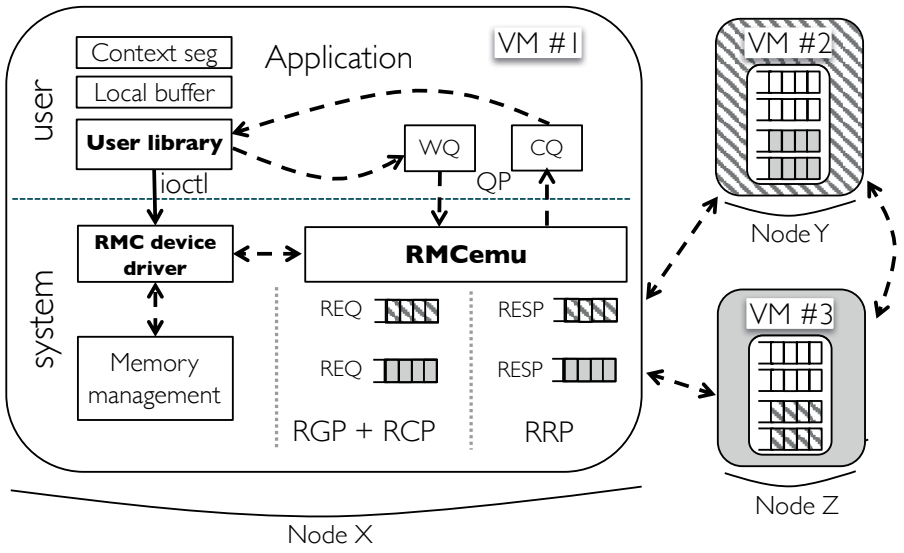


Figure 4.7 – soNUMA development platform. Each node is implemented by a different VM. `RMCemu` runs on dedicated virtual CPUs and communicates with peers via shared memory.

4.7 Evaluation

4.7.1 Methodology

To evaluate soNUMA, we designed and implemented two platforms: (i) development platform – a software prototype of soNUMA based on virtual machines used to debug the protocol stack, formalize the API, and develop large-scale applications; and (ii) cycle-accurate model – a full-system simulation platform modeling the proposed RMC.

Development platform. Our software soNUMA prototype is based on the Xen hypervisor [17] and a conventional ccNUMA server, on top of which we map (pin) multiple virtual machines to distinct NUMA domains. This includes both virtual CPUs and memory page frames. The server we use for the prototype is a modern AMD Opteron server with 4 CPU sockets (12 cores each, three-level cache hierarchy, 16MB LLC) and 256GB of RAM. The memory subsystem provides us with 8 NUMA domains (2 per socket).

Fig. 4.7 shows our setup. Each individual VM represents an independent soNUMA node, running an instance of the full software stack. The stack includes all user-space libraries, applications, the OS kernel, as well as the complete RMC device driver inside it.

Core	ARM Cortex-A15-like; 64-bit, 2GHz, OoO, 3-wide dispatch/retirement, 60-entry ROB
L1 Caches	split I/D, 32KB 2-way, 64-byte blocks, 2 ports, 32 MSHRs, 3-cycle latency (tag+data)
L2 Cache	4MB, 2 banks, 16-way, 6-cycle latency
Memory	cycle-accurate model using DRAMSim2 [144]. 4GB, 8KB pages, single DDR3-1600 channel. DRAM latency: 60ns; bandwidth: 12GBps
RMC	3 independent pipelines (RGP, RCP, RRPP). 32-entry MAQ, 32-entry TLB
Fabric	Inter-node delay: 50ns

Table 4.1 – System parameters for simulation on Flexus.

The driver is a Linux kernel module that responds to user library commands through `ioctl`, enabling WQ/CQ registration, buffer management, and security context registration.

In this platform, we have implemented an RMC emulation module (RMCemu), which runs in kernel space. RMCemu implements the RMC logic and the soNUMA wire protocol (for a total of 3100LOC). The module exposes the hardware/software interface described in §4.4.1 to the RMC device driver and applications. RMCemu runs as a pair of kernel threads pinned to dedicated virtual CPUs, one running RGP and RCP, the other RRPP of Fig. 4.4. All of the user-level data structures and buffers get memory-mapped by the device driver into the kernel virtual address space at registration time, and thus become visible to the RMCemu threads.

We emulate a full crossbar and run the protocol described in §4.6. Each pair of nodes exchanges protocol request/reply messages via a set of queues, mapped via the hypervisor into the guest physical address spaces of the VMs (there are two queue pairs per VM pair, emulating virtual lanes). To model the distributed nature of an soNUMA system, we pin each emulated node to a distinct NUMA domain such that every message traverses one of the server's chip-to-chip links. However, for the 16-node configuration, we collocate two VMs per NUMA domain.

Cycle-accurate model. To assess the performance implications of the RMC, we use the Flexus full-system simulator [164]. Flexus extends the Virtutech Simics functional simulator

with timing models of cores, caches, on-chip protocol controllers, and interconnect. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications. In its detailed OoO timing mode with the RMCs implemented, Flexus simulates "only" 5000 instructions per second, a slowdown of about six orders of magnitude compared to real hardware.

We model simple nodes, each featuring a 64-bit ARM Cortex-A15-like core and an RMC. The system parameters are summarized in Table 4.1. We extend Flexus by adding a detailed timing model of the RMC based on the microarchitectural description in §4.4. The RMC and its private L1 cache are fully integrated into the node's coherence domain. Like the cores, the RMC supports 32 memory accesses in flight. Fig. 4.4 illustrates how the logic is modeled as a set of finite state machines that operate as pipelines and eliminate the need for any software processing within the RMC. We model a full crossbar with reliable links between RMCs and a flat latency of 50ns, which is conservative when compared to modern NUMA interconnects, such as QPI and HTX.

4.7.2 Microbenchmark: Remote Reads

We first measure the performance of remote read operations between two nodes for both the development platform and the Flexus-based simulated hardware platform. The microbenchmark issues a sequence of read requests of varying size to a preallocated buffer in remote memory. The buffer size exceeds the LLC capacity in both setups. We measure (i) remote read latency with synchronous operations, whereby the issuing core spins after each read request until the reply is received, and (ii) throughput using asynchronous reads, where the issuing core generates a number of non-blocking read requests before processing the replies (similar to Fig. 4.5). We run the micro-benchmark in both single-sided (only one node reads) and double-sided (both nodes read from each other) mode.

Fig. 4.8 shows the remote read latency on the simulated hardware as a function of the request size. For small request sizes, the latency is around 300ns, of which 80ns are attributed to accessing the memory (cache hierarchy and DRAM combined) at the remote node and 100ns to round-trip socket-to-socket link latency.

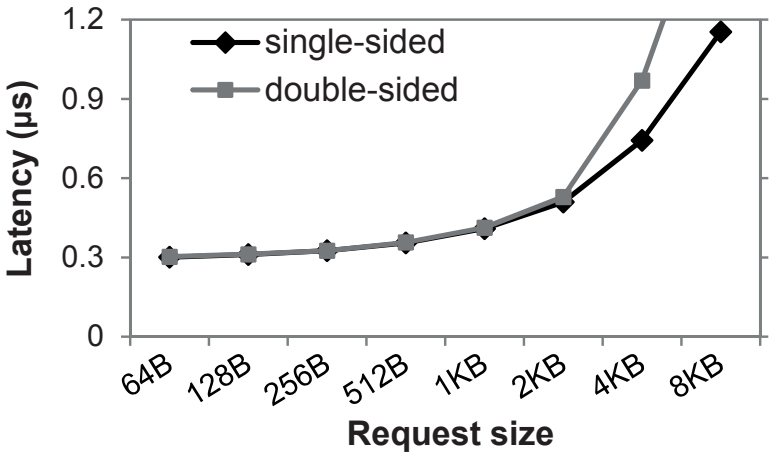


Figure 4.8 – Remote read latency (sim'd HW).

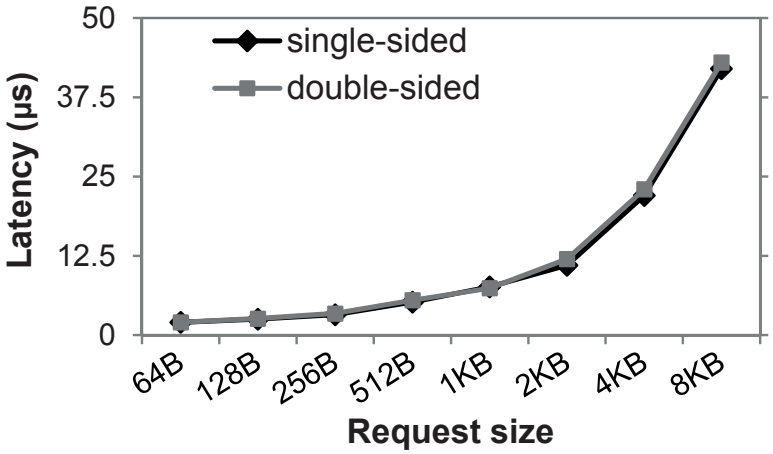


Figure 4.9 – Remote read latency (dev. platform).

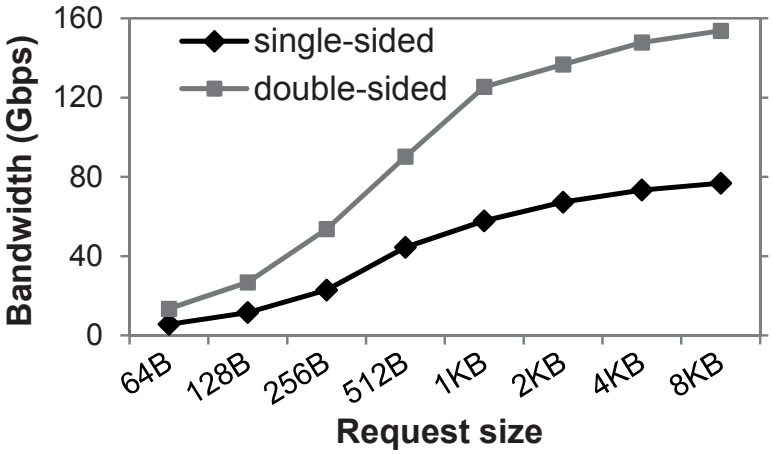


Figure 4.10 – Remote read throughput (sim'd HW).

The end-to-end latency is within a factor of 4 of the local DRAM access latency. In the double-sided mode, we find that the average latency increases for larger message sizes as compared to the single-sided case. The reason for the drop is cache contention, as each node now has to both service remote read requests and write back the reply data.

Fig. 4.9 shows the latency results on the development platform. The baseline latency is 1.5μs, which is 5x the latency on the simulated hardware. However, we notice that the latency increases substantially with larger request sizes. On the development platform, the RMC emulation module becomes the performance bottleneck as it unrolls large WQ requests into cache-line-sized requests.

Fig. 4.10 plots the bandwidth between two simulated soNUMA nodes using asynchronous remote reads. For 64B requests, we can issue 10M operations per second. For page-sized requests (8KB), we manage to reach 9.6GBps, which is the practical maximum for a DDR3-1600 memory channel. Thanks to the decoupled pipelines of the RMC, the double-sided test delivers twice the single-sided bandwidth.

4.7.3 Microbenchmark: Send/Receive

We build a Netpipe [154] microbenchmark to evaluate the performance of the soNUMA unsolicited communication primitives, implemented entirely in software (§4.5.3). The microbenchmark consists of the following two components: (i) a ping-pong loop that uses the smallest message size to determine the end-to-end one-way latency and (ii) a streaming experiment where one node is sending and the other receiving data to determine bandwidth.

We study the half-duplex latency (Fig. 4.11) and bandwidth (Fig. 4.13) on our simulation platform. The two methods (`pull`, `push`) expose a performance tradeoff: `push` is optimized for small messages, but has significant processor and packetization overheads. `pull` is optimized for large transfers, but requires additional control traffic at the beginning of each transfer. We experimentally determine the optimal boundary between the two mechanisms by setting the threshold to 0 and ∞ in two separate runs.

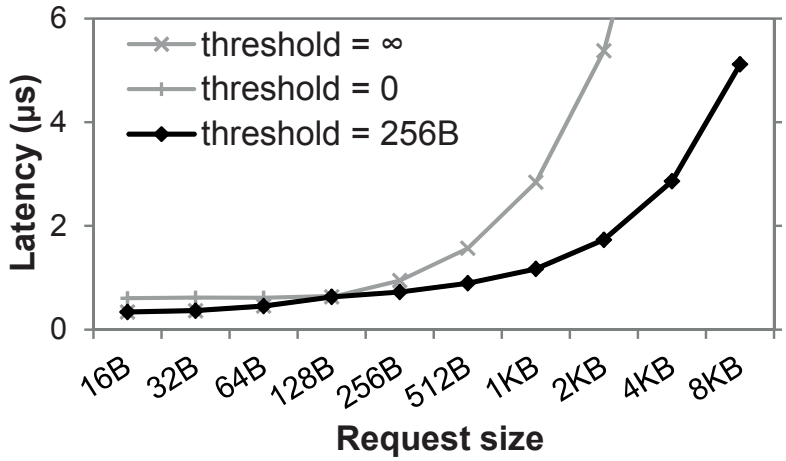


Figure 4.11 – Send/receive read latency (sim'd HW).

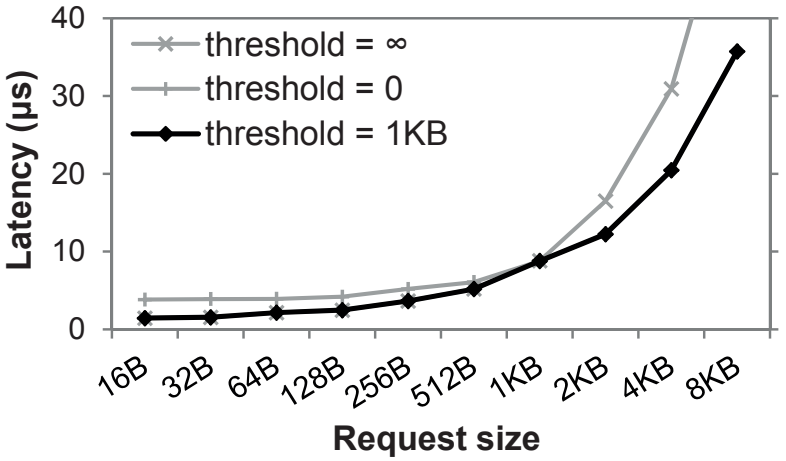


Figure 4.12 – Send/receive read latency (dev. platform).

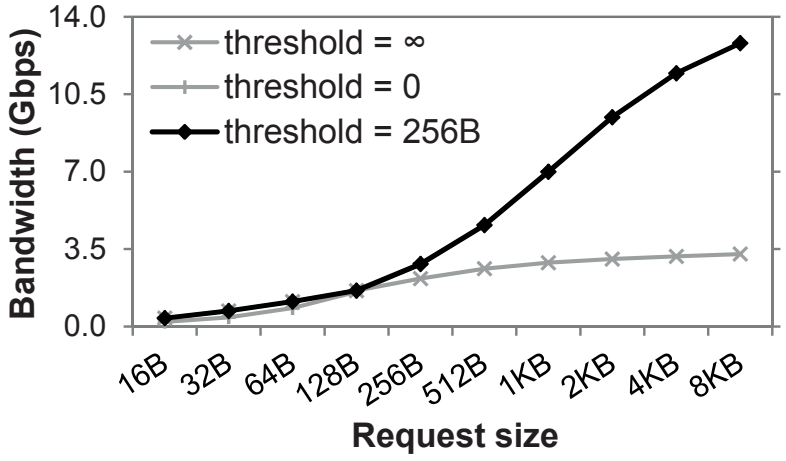


Figure 4.13 – Send/receive read throughput (sim'd HW).

Transport	soNUMA		RDMA/IB
	Dev. Plat.	Sim'd HW	[47]
Max BW (Gbps)	1.8	77	50
Read RTT (μs)	1.5	0.3	1.19
Fetch-and-add (μs)	1.5	0.3	1.15
IOPS (Mops/s)	1.97	10.9	35 @ 4 cores

Table 4.2 – A comparison of soNUMA and InfiniBand.

The black curve shows the performance of our unsolicited primitives with the threshold set to the appropriate value and both mechanisms enabled at the same time. The minimal half-duplex latency is 340ns and the bandwidth exceeds 10Gbps with messages as small as 4KB. For the largest request size evaluated (8KB), the bandwidth achieved is 12.8 Gbps, a 1.6x increase over Quad Data Rate InfiniBand for the same request size [80]. To illustrate the importance of having a combination of `push` and `pull` mechanisms in the user library, we additionally plot in grey their individual performance.

We apply the same methodology on the development platform. The minimal half-duplex latency (see Fig. 4.12) is 1.4μs, which is only 4x worse than the simulated hardware. However, the threshold is set to a larger value of 1KB for optimal performance, and the bandwidth is 1/10th of the simulated hardware. The relatively low bandwidth and a different threshold are due to the overheads of running the fine-grain communication protocol entirely in software (§4.7.2).

4.7.4 Comparison with InfiniBand/RDMA

To put our key results in perspective, Table 4.2 compares the performance of our simulated soNUMA system with an industry-leading commercial solution that combines the Mellanox ConnectX-3 [115] RDMA host channel adapter connected to host Xeon E5-2670 2.60Ghz via a PCIe-Gen3 bus. In the Mellanox system [47], the servers are connected back-to-back via a 56Gbps InfiniBand link. We consider four metrics – read bandwidth, read latency, atomic operation latency, and IOPS.

As Table 4.2 shows, compared to the state-of-the-art RDMA solution, soNUMA reduces the latency to remote memory by a factor of four, in large part by eliminating the PCIe bus overheads. soNUMA is also able to operate at peak memory bandwidth. In contrast, the

PCIe-Gen3 bus limits RDMA bandwidth to 50 Gbps, even with 56Gbps InfiniBand. In terms of IOPS, the comparison is complicated by the difference in configuration parameters: the RDMA solution uses four QPs and four cores, whereas the soNUMA configuration uses one of each. Per core, both solutions support approximately 10M remote memory operations.

We also evaluate the performance of atomic operations using fetch-and-add, as measured by the application. For each of the three platforms, the latency of fetch-and-add is approximately the same as that of the remote read operations on that platform. Also, soNUMA provides more desirable semantics than RDMA. In the case of RDMA, fetch-and-add is implemented by the host channel adapter, which requires the adapter to handle all accesses, even from the local node. In contrast, soNUMA's implementation within the node's local cache coherence provides global atomicity guarantees for any combination of local and remote accesses.

4.7.5 Rack-Scale Graph Analytics

Rack-scale analytics and distributed NoSQL databases are the obvious candidate applications for soNUMA. All of them perform very little work per data item (if any) and operate on large datasets, and hence typically require a scale-out configurations to keep all the data memory resident. Most importantly, they exhibit poor locality as they frequently access non-local data. We first study the impact of soNUMA on rack-scale graph analytics, and, in the next chapter, we show how soNUMA can be used as a building block (i.e. RSMP unit) in large-scale key-value stores.

We picked the canonical PageRank algorithm [136], and compared three parallel implementations of it. All three are based on the widely used Bulk Synchronous Processing model [160], in which every node computes its own portion of the dataset (range of vertices) and then synchronizes with other participants, before proceeding with the next iteration (so-called superstep). Our three implementations are:

(i) `SHM(pthreads)`: The baseline is a standard pthreads implementation that assumes cache-coherent memory rather than soNUMA. For the simulated hardware, we model an eight-core multiprocessor with 4MB of LLC per core. We provision the LLC so that the aggregate cache size equals that of the eight machines in the soNUMA setting. Thus, no benefits can be attributed to

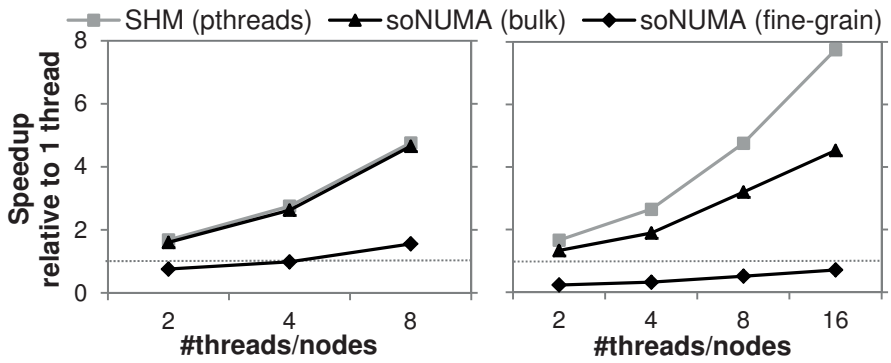


Figure 4.14 – PageRank speedup on simulated HW (left) and on the development platform (right).

larger cache capacity in the soNUMA comparison. For the development platform, we simply run the application on our ccNUMA server described in §4.7.1, but without a hypervisor running underneath the host OS. In this implementation, the application stores two rank values for each vertex: the one computed in the previous superstep and the one currently being computed. Barrier synchronization marks the end of each superstep.

(ii) `soNUMA(bulk)`: This scale-out implementation leverages aggregation mechanisms and exchanges ranks between nodes at the end of each superstep (after the barrier) using soNUMA's one-sided reads. A similar approach is supported by Pregel [112] as it amortizes high inter-node latencies and makes use of high-capacity Ethernet links. In this implementation we exploit spatial locality within the global address space by using multi-line requests at the RMC level. At the end of each superstep, every node uses multiple `rmc_read_async` operations (one per peer) to pull the remote vertex information from each of its peers into the local memory. This allows a concurrent shuffle phase limited only by the bisection bandwidth of the system.

(iii) `soNUMA(fine-grain)`: This variant leverages the fine-grain memory sharing capabilities of soNUMA, as shown in Fig. 4.5. Each node issues one `rmc_read_async` operation for each non-local vertex. This implementation resembles the shared-memory programming model of `SHM(pthreads)`, but has consequences: the number of remote memory operations scales with the number of edges that span two partitions rather than with the number of vertices per partition.

We evaluate the three implementations of PageRank on a subset of the Twitter graph [95] (30GB) using a naive algorithm that randomly partitions the vertices into sets of equal cardinality. We run 30 supersteps for up to 16 soNUMA nodes on the development platform. On the simulator, we run a single superstep on up to eight nodes because of the high execution time of the cycle-accurate model. We do not compare soNUMA against a scale-out Ethernet configuration because of disproportionate latency and bandwidth.

Fig. 4.14 (left) shows the speedup over the single-threaded baseline of the three implementations on the simulated hardware. Both `SHM(pthreads)` and `soNUMA(bulk)` have near identical speedup. In both cases, the speedup trend is determined primarily by the imbalance resulting from the graph partitioning scheme, and not the hardware. However, `soNUMA(fine-grain)` has noticeably greater overheads, primarily because of the limited per-core remote read rate (due to the software API's overhead on each request) and the fact that each reference to a non-local vertex results in a remote read operation. Indeed, each core can only issue up to 10 million remote operations per second. As shown in Fig. 4.10, the bandwidth corresponding to 64B requests is a small fraction of the maximum bandwidth of the system.

Fig. 4.14 (right) shows the corresponding speedup on the software development platform. We identify the same general trends as on the simulated hardware, with the caveat that the higher latency and lower bandwidth of the development platform limit performance.

4.8 Discussion

We now discuss the lessons learned in developing our soNUMA prototype, known open issues, killer applications, and deployment and packaging options.

Lessons learned. During our work, we appreciated the value of having a software development platform capable of running at native speeds. By leveraging hardware virtualization and dedicating processing cores to emulate RMCs, we were able to run an (emulated) soNUMA fabric at wall-clock execution time, and use that platform to develop and validate the protocol, the kernel driver, all user-space libraries, and applications.

Open issues. Our design provides the ability to support many variants of remote memory operations that can be handled in a stateless manner by the peer RMC. This includes read, write, and atomic operations. A complete architecture will probably require extensions such as the ability to issue remote interrupts as part of an RMC command, so that nodes can communicate without polling. This will have a number of implications for system software, e.g., to efficiently convert interrupts into application messages, or to use the mechanisms to build system-level abstractions such as global buffer caches.

Killer applications. Our primary motivation behind soNUMA is the conflicting trend of (i) large dataset applications that require tight and unpredictable sharing of resources; and (ii) manycore designs, which are optimized for throughput rather than resource aggregation. Our soNUMA proposal aims to reconcile these two trends. In our evaluation, we chose a simple graph application because it allows for multiple implementation variants that expose different aspects of the programming model, even though the regular, batch-oriented nature of that application is also a good fit for coarse-grain, scale-out models. Implementing these variants using the RMC API turned out to be almost as easy as using the conventional shared-memory programming abstractions.

Many applications such as on-line graph processing algorithms, in-memory transaction processing systems, and key-value stores demand low latency [145] and can take advantage of one-sided read operations [117]. These applications are designed to assume that both client and server have access to a low-latency fabric [117, 132], making them killer applications for soNUMA. In §5, we show how a scale-out key-value store can take advantage of soNUMA. We design and implement a KVS that leverages the soNUMA fabric for load balancing purposes, delivering better application throughput. Beyond these, we also see deployment opportunities in upper application tiers of the datacenter. For example, Oracle Exalogic today provides a flexible, low-latency platform delivered on a rack-scale InfiniBand cluster [134]. Such deployments are natural candidates for tighter integration using SoC and soNUMA.

System packaging options. The evaluation of the impact of system scale and fabric topologies is outside of the scope of this thesis since we focused on the end-behavior of the RMC. To be successful, soNUMA systems will have to be sufficiently large to capture very large

datasets within a single cluster, and yet sufficiently small to avoid introducing new classes of problems, such as the need for fault containment [42]. Industry solutions today provide rack-insert solutions that are ideally suited for soNUMA, e.g., HP's Moonshot with 45 cartridges or Calxeda-based Viridis systems with 48 nodes in a 2U chassis. Beyond that, rack-level solutions seem like viable near-term options; a 44U rack of Viridis chassis can thus provide over 1000 nodes within a two-meter diameter, affording both low wire delay and massive memory capacity.

NUMA vs. IP over lossless Ethernet/Infiniband. The Scale-Out NUMA (soNUMA) transport assumes a NUMA memory subsystem as the underlying mechanism for transferring data from remote to local memory and vice verse. Unlike soNUMA, the Infiniband transport relies either on lossless Ethernet or an actual Infiniband network. With Infiniband reaching the bandwidth of local DRAM [109], it is not hard to imagine a rack-scale system based on Infiniband that is similar to soNUMA, even though folding a full-blown RDMA HCA into the cache hierarchy of the CPU is not a trivial task.

An integrated Infiniband design would feature a lightweight hardware-software interface, just like soNUMA, but it would still incur overhead associated with other components on the data path. First, to initiate and complete operations it would require the same amount of work done by the CPU (i.e., instruction footprint). Second, assuming the same feature set supported by the original Infiniband, the amount of state kept in the network end-points would still be the same (e.g. connections). Finally, the typical size of the MTU in Infiniband is 4KB, and the smallest supported MTU is 256B, whereas the header overhead is 98 bytes for RDMA operations [5].

The header overhead of 98 bytes in size assumes the original feature set, such as routing across L2 domains, which requires network headers - global IDs in Infiniband (40B). The 4KB MTU increases the propagation delay, which may impact the latency for small messages, whereas for 256B MTU the headers take up much of the packet space (almost 40%) and, thus, requires more packets in aggregate size to be transferred for the same amount of data. On top of that, a single Ethernet switch introduces an additional delay of 0.2us [114].

On soNUMA, we strip much of the RDMA functionality and simplify the transport protocol.

The choice of NUMA encouraged us to implement fewer features to reduce the header overhead to 16 bytes due to the fixed MTU supported by NUMA (64 bytes). The MTU of 64 bytes, in combination with low-latency switches, significantly reduces the propagation delay and improves the latency for small messages. Because of the simple NUMA request/reply network protocol, the RMC maintains much less state, which further simplifies the design and enables integration. Finally, the simplified API leads to smaller intruction footprint and, hence, lower CPU overhead as compared to Infiniband.

We now provide a quick comparison between Infiniband and soNUMA at the wire level. That is, we ignore the hardware-software interface overheads, and focus only on actual data transfers crossing the network links and switches. We assume the same link bandwidth (100Gbps), but different MTU, different header sizes, and different physical interconnect implementations (i.e., switch-based vs. "glueless"). The MTU sizes supported by Infiniband range from 256B to 4KB, whereas the header size is 98B, independent of the MTU [138]. On soNUMA, the MTU is 64B and the header size is 16B. We leverage the following formula to compute the transfer time T:

$$T = propagation\_delay + \frac{transfer\_size - MTU}{bandwidth}$$

The *propagation_delay* is the time that it takes for the first packet, which is maximum MTU in size, to reach the destination, and includes the time to acquire all the resources for the subsequent packets that are part of the flow. The second half of the equation represents the serialization delay for the rest of the transfer. The equation is trivial in sense that it assumes an uncongested network - no queuing delay in network elements.

The *propagation_delay* is computed as follows:

$$propagation\_delay = link\_delay * hop\_cnt + switch\_cnt * switch\_latency$$

$$link\_delay = \frac{MTU}{bandwidth}$$

MTU size	4KB	50MB
4KB (RDMA/IB)	0.8us	4.8ms
256B (RDMA/IB)	0.7us	6.5ms
64B (soNUMA)	0.5us	5.1ms

Table 4.3 – Remote write performance comparison of Infiniband and soNUMA for different MTU sizes and header overhead.

To compute the propagation delay, we assume one switch (i.e., two hops) for Infiniband and the average of eight hops (seven switches) for a 2D torus soNUMA. We focus on two transfer sizes, 4KB and 50MB, and three different MTU sizes; 4KB Infiniband, 256 bytes Infiniband, and 64 bytes soNUMA. The header overhead on Infiniband is 98B, whereas on soNUMA it is 16B. We assume low-radix routers in soNUMA with 10ns pin-to-pin latency [118]. The comparison focuses on one-way transfers, such as remote writes.

Table 4.3 shows the transfer times computed using the model. For 4KB transfer size and 4KB IB MTU, the serialization delay is small and the propagation time dominates the total transfer time. For 256B MTU, the propagation time is smaller, but the serialization delay is much higher due to a high per-byte header overhead; 38% versus 2% for 4KB MTU. soNUMA's per-byte header overhead is 24%, but the propagation delay is much smaller than on Infiniband, resulting in 200-300ns improvement in latency.

For 50MB transfer size, the MTU of 4KB on Infiniband is superior to smaller MTU sizes. This is because the serialization delay dominates the transfer time for all three MTU sizes and 4KB MTU provides the lowest per-byte header overhead. However, soNUMA could provide multi-path routing to utilize the multiple links coming in and out of each soNUMA server node, which may require higher core count, a high-throughput RMC implementation [50], and more memory controllers. In such a scenario, soNUMA would provide much higher bisection bandwidth, which would further lead to better application performance.

Follow-up research and future directions. This work has demonstrated the benefits on simple microbenchmarks and one application. We have also evaluated the impact of soNUMA on latency-sensitive applications such as in-memory distributed databases (NoSQL). We also see research questions around system-level resource sharing: for example, to create a single-system image or a global filesystem buffer cache, or to rethink resource management

in hypervisor clusters. Multikernels, such as Barrelfish [21], could be an ideal candidate for soNUMA.

We have investigated the micro-architectural aspects of the RMC. This includes further optimizations in the RMC and the processor that reduce the remote latency, a reassessment of the RMC's design for SoC's with high core counts [50], and new one-sided primitives such as atomic bulk reads [51].

Our study focused on data sharing within a single soNUMA fabric. For very large datasets, datacenter deployments could connect multiple soNUMA systems using conventional networking technologies. This opens up new system-level research questions in the areas of resource management and networking (e.g., how to use the soNUMA fabric to run standard network protocols).

4.9 Related Work

Many of the concepts found in remote memory architectures today and our soNUMA proposal originate from research done in the '80s and '90s. In this section we look at the relationship between soNUMA and several related concepts. We group prior work into six broad categories.

Partitioned global address space. PGAS relies on compiler and language support to provide the abstraction of a shared address space on top of non-coherent, distributed memory [44]. Languages such as Unified Parallel C [168, 44] and Titanium [168] require the programmer to reason about data partitioning and be aware of data structure non-uniformity. However, the compiler frees the programmer from the burden of ensuring the coherence of the global address space by automatically converting accesses to remote portions into one-sided remote memory operations that correspond to soNUMA's own primitives. PGAS also provides explicit asynchronous remote data operations [26], which also easily map onto soNUMA's asynchronous library primitives. The efficiency of soNUMA remote primitives would allow PGAS implementations to operate faster.

Software distributed shared memory. Software distributed shared memory (DSM) provides global coherence not present in the memory hierarchies of PGAS and soNUMA. Pure software DSM systems such as IVY [100], Munin [37] and Threadmarks [10] expose a global coherent virtual address space and rely on OS mechanisms to "fault in" pages from remote memory on access and propagate changes back, typically using relaxed memory models. Similarly, software DSM can be implemented within a hypervisor to create a cache-coherent global guest-physical address space [43], or entirely in user-space via binary translation [147]. Like software DSM, soNUMA operates at the virtual memory level. Unlike software DSM, soNUMA and PGAS target fine-grained accesses whereas software DSM typically operates at the page level. Shasta [147] and Blizzard [148] offer fine-grain DSM through code instrumentation and hardware assistance, respectively, but in both cases with non-negligible software overheads.

Cache-coherent memory. ccNUMA designs such as Alewife [7], Dash [99], FLASH [94], Typhoon [143], Sun Wildfire [124], and today's Intel QPI and AMD HTX architectures create a compute fabric of processing elements, each with its own local memory, and provide cache-coherent physical memory sharing across the nodes. FLASH and Sun's Wildfire also provide advanced migration and replication techniques to reduce the synchronization overheads [66].

soNUMA shares the non-uniform aspect of memory with these designs and leverages the lower levels of the ccNUMA protocols, but does not attempt to ensure cache coherence. As a result, soNUMA uses a stateless protocol, whereas ccNUMA requires some global state such as directories to ensure coherence, which limits its scalability. The ccNUMA designs provide a global physical address space, allowing conventional single-image operating systems to run on top. The single-image view, however, makes the system less resilient to faults [42]. In contrast, soNUMA exposes the abstraction of global virtual address spaces on top of multiple OS instances, one per coherence domain.

User-level messaging. User-level messaging eliminates the overheads of kernel transitions by exposing communication directly to applications. Hybrid ccNUMA designs such as FLASH [77], Alewife [7], and Typhoon [65] provide architectural support for user-level messaging in conjunction with cache-coherent memory. In contrast, soNUMA's minimal design

allows for an efficient implementation of message passing entirely in software using one-sided remote memory operations.

SHRIMP [25] uses a specialized NIC to provide user-level messaging by allowing processes to directly write the memory of other processes through hardware support. Cashmere [155] leverages DEC's Memory Channel [71], a remote-write network, to implement a software DSM. Unlike SHRIMP and Cashmere, soNUMA also allows for direct reads from remote memory.

Fast Messages [137] target low latency and high bandwidth for short user-level messages. U-Net [162] removes the entire OS/Kernel off the critical path of messaging. These systems all focus on the efficient implementation of a message send. In soNUMA, the RMC provides architectural support for both one-sided read and write operations; messaging is implemented on top of these basic abstractions.

Remote memory access. Unlike remote write networks, such as SHRIMP and DEC Memory Channel, Remote Memory Access provides for both remote reads and remote writes. Such hardware support for one-sided operations, similar to soNUMA's, was commercialized in supercomputers such as Cray T3D [92] and T3E [150]. Remote memory access can also be implemented efficiently for graph processing on commodity hardware by leveraging aggressive multithreading to compensate for the high access latency [122]. soNUMA also hides latency but uses asynchronous read operations instead of multithreading.

Today, user-level messaging and RDMA is available in commodity clusters with RDMA host channel adapters such as Mellanox ConnectX-3 [115] that connect into InfiniBand or Converged Ethernet switched fabrics [86]. To reduce complexity and enable SoC integration, soNUMA only provides a minimal subset of RDMA operations; in particular, it does not support reliable connections, as they require keeping per-connection state in the adapter.

NI integration. One advantage of soNUMA over prior proposals on fast messaging and remote one-sided primitives is the tight integration of the NI into the coherence domain. The advantage of such an approach was previously demonstrated in Coherent Network Interfaces (CNI) [119], which leverage the coherence mechanism to achieve low-latency communication

of the NI with the processors, using cacheable work queues. More recent work showcases the advantage of integration, but in the context of kernel-level TCP/IP optimizations, such as a zero-copy receive [24]. Our RMC is fully integrated into the local cache coherence hierarchy and does not depend on local DMA operations. The simple design of the RMC suggests that integration into the local cache-coherence domain is practical. Our evaluation shows that such integration can lead to substantial benefits by keeping the control data structures, such as the QPs and page tables, in caches. soNUMA also provides global atomicity by implementing atomic operations within a node's cache hierarchy.

4.10 Summary

Scale-Out NUMA (soNUMA) is an architecture, programming model, and communication protocol for low-latency RSMP. soNUMA eliminates kernel, network stack, and I/O bus overheads by exposing a new hardware block –the remote memory controller– within the cache coherent hierarchy of the processor. The remote memory controller is directly accessible by applications and connects directly into a NUMA fabric. Our results based on cycle-accurate full-system simulation show that soNUMA can achieve remote read latencies that are within 4x of local DRAM access, stream at full memory bandwidth, and issue up to 10M remote memory operations per second per core.

The purpose of soNUMA is primarily to empower modern datacenters to execute software faster. However, how to take advantage of RSMP solutions, such as soNUMA, in various software systems running in datacenters is not a given. We previously showed how graph computation can use the soNUMA fabric to perform the shuffle phase faster. Batching of vertex updates during the communication phase alone over soNUMA improves the end-to-end performance. Unlike graphs, other software systems require some more thought to figure out how to take advantage of RSMP. The purpose of the next chapter is exactly that: to study one such scale-out application and identify the pain points that RSMP can help neutralize.

5 Leveraging RSMP in Data Serving

To provide low latency and high throughput guarantees, distributed NoSQL databases 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 chapter, we explore how data in distributed NoSQL databases, such as key-value stores, can be aggregated at rack-scale granularity using *rack-scale memory pooling* (RSMP) in order to reduce load imbalance. We will refer to this technique as Rack-Out KVS throughout this chapter. In Rack-Out KVS, all of the participating servers in each rack are jointly servicing all of the rack's micro-shards using the underlying fabric's one-sided primitives.

We develop a queuing model to evaluate the impact of Rack-Out KVS at datacenter scale. In addition, we implement a proof-of-concept Rack-Out KVS (RO-KVS), evaluate it on two experimental platforms based on RDMA and soNUMA, and use these results to validate the model. Our results show that Rack-Out KVS increases 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.

5.1 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 [16, 54, 123]. To meet these objectives, most such applications keep the data (e.g., a social graph) in memory-resident distributed NoSQL databases, such as distributed key-value stores (KVS) [56, 62, 69, 146]. 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 chapter, we introduce an application of RSMP to large distributed key-value stores that we refer to as Rack-Out KVS. The conclusions we make about using RSMP in KVS also apply to other instances of distributed NoSQL databases, because of similar data layout and workload properties [31, 41]; for example, a NoSQL database could use a lexicographically ordered index instead of a hash, and store data in tables rather than key-value tuples [41]. Nevertheless, only the simple put/get operations on individual data items (rows) would apply, while the index is distribution-agnostic. A Rack-Out KVS unit is equivalent to an RSMP unit, a group of servers (i.e., a rack) augmented with an internal secondary fabric allowing any node within the rack to read from the memory of other nodes through one-sided operations (i.e., without involving the remote CPU). Consequently, the data in Rack-Out KVS is partitioned at rack-scale granularity, with the entire rack responsible for a collection of data items mapped to the rack's servers. Rack-Out KVS leverages the *concurrent-read/exclusive-write* (CREW) data

access model within individual RSMP units, which has previously been shown to dramatically improve the scalability of single-server performance on multicore servers [58, 102, 117]. By decoupling data access from data storage, Rack-Out KVS substantially reduces the negative impact of skew on the entire KVS' performance.

To support the CREW model in Rack-Out KVS, the client identifies the server holding the target micro-shard and therefore the RSMP unit it belongs to. Read requests are load-balanced among all the servers within each RSMP unit, while write requests are always directed to the server holding the micro-shard. Even though data is never replicated within an RSMP unit, Rack-Out KVS is compatible and synergistic with dynamic replication [31, 54, 79, 81]. Since the Rack-Out KVS technique already balances requests within an RSMP unit (using one-sided operations rather than replication), replication is only required across RSMP units, which reduces the overheads associated with load monitoring, creating replicas, and ongoing consistency maintenance.

This chapter describes the following contributions:

• We provide a detailed analysis of the impact of rack-level aggregation on mitigating the effect of skew. We develop a queuing model for Rack-Out KVS that assumes globally accessible memory within the rack. We evaluate the benefits of Rack-Out KVS as a function of server count, RSMP unit size, 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 Rack-Out KVS deployment of 512 servers grouped into 16-server RSMP units increases throughput by 6× with RDMA and 8.6× with Scale-Out NUMA (soNUMA) [50, 51, 126] without violating SLO.

• We evaluate the combination of Rack-Out KVS with an idealized dynamic replication scheme. Our results show that the Rack-Out KVS model is synergistic with dynamic replication and dramatically reduces the number of replicas required for load balancing.

• We implement Rack-Out 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 [58] and is ported to both Mellanox RDMA [116] 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 RSMP 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.

The rest of the paper is organized as follows: §5.2 discusses the key architectural trends providing cost-effective RSMP. §5.3 provides a detailed analysis of the Rack-Out KVS queuing model, which we validate in §5.4 using an implementation of our proof-of-concept RO-KVS. We discuss related work in §5.6 and conclude in §5.7.

5.2 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 in order to have fewer larger shards. As we showed in §2, 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 using RSMP. The design space for such solutions is broad, but can be broken down into architectural considerations (§5.2.1), concurrency issues (§5.2.2) and fault tolerance (§5.2.3).

5.2.1 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, which is the main topic of this thesis.

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, that we refer to as *rack-scale memory pooling* (RSMP) leverages conventional datacenter-grade servers or individually less capable server building blocks [11, 39, 60, 63, 111, 157] augmented with a rack-level RDMA fabric. A similar approach is used in commercial products providing analytical (e.g., Oracle ExaData / Exalogic [133]) or storage (e.g. EMC/Isilon [59]) solutions to clients connected via a conventional network. AppliedMicro's X-Gene2 server SoC [107] and Oracle's Sonoma [73] integrate the RDMA controller directly on chip, HP Moonshot [78] combines low-power processors with RDMA NICs, and research proposals further argue for on-chip support for one-sided remote access primitives [50, 126]. Building larger logical entities using the RSMP approach instead of the cache-coherent NUMA approach comes at lower cost and complexity.

The fundamental premise for RSMP 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 [116].

5.2.2 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* (CREW) data access model 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 [101, 102]. As most workloads are read-dominated [16, 31, 46, 140], 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 [58] and Pilaf [117] 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.

5.2.3 Availability and Durability

RSMP 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 [56, 135, 156]. 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 [34]) 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.

5.3 Rack-Out Data Serving

The basic building block of Rack-Out KVS is an RSMP unit, 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.

RSMP 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. In Rack-Out KVS, we refer to the number of intra-rack servers pooling memory as the *Grouping*

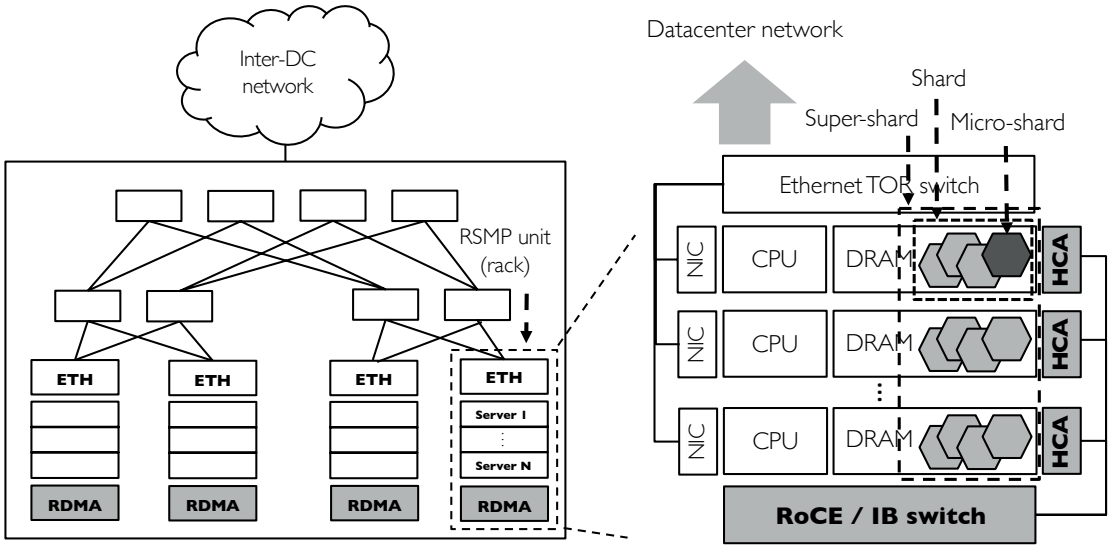


Figure 5.1 – Rack-Out KVS architecture using RDMA hardware. N x 4 micro-shards within each rack perceived as one super-shard. Non-local micro-shards accessed through RDMA reads.

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 of the hot server is fully utilized.

By aggregating the memory of a rack through RSMP, the per-node load increases, but only marginally, while the compute power 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. In §5.3.1, we provide a detailed analysis of this phenomenon.

Fig. 5.1 illustrates the RSMP architecture of a datacenter and how Rack-Out KVS takes advantage of it. Each unit connects the servers via an internal RDMA fabric (either RoCE or Infiniband), which supports one-sided read and write operations. Each server in the unit stores a group of *micro-shards* forming a *shard*, as in conventional KVS, while RSMP unit exposes the abstraction of a *super-shard* comprising all the micro-shards within a unit.

Fig. 5.1 also clearly illustrates the key assumption behind the model: by confining the fabric to the unit, the RSMP model sits between the traditional scale-out model and the full-scale

RDMA fabric deployment. From an application perspective, the proposed RSMP and scale-out models are similar: clients connect to servers via the network (assuming clients and servers run in different racks [64]) and applications rely on replication to scale beyond the unit of capacity (i.e., rack or server) and ensure data availability.

Recent work has shown that scaling RDMA over commodity Ethernet introduces issues of congestion, dealing with deadlocks and livelocks, and other subtleties of priority-based flow control [74, 170]. Even though the flow-control issues of commodity Ethernet may go away [74, 170], complementing datacenter-scale RDMA (or equivalent) fabrics with specialized ultra-low-latency rack-scale fabrics [15, 126, 130] will be of great importance to balance the load across small groups of servers using the Rack-Out KVS approach, without the cost of data replication. In such designs, the bottleneck would possibly shift from the CPU to the network interface, still requiring RSMP or a similar mechanism to mitigate the imbalance (see §5.5). To make RSMP compatible with future networking technologies, it will be key to provision enough network bandwidth among the servers of different racks and also enable high-throughput access to rack-local data for load balancing purposes. In §3.2, we state two key requirements for two-fabric solutions to provide a performance impact over single-fabric solutions.

Rack-Out KVS 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 RSMP unit.

In the rest of this section, we present a first-order analysis showing the maximal speedup attainable by Rack-Out KVS as a function of the read/write mix and the GF (§5.3.1), and subsequently build a queuing model for Rack-Out KVS to study service time implications at the tail latency (§5.3.2). We analyze the sensitivity of the skew to the dataset distribution (§5.3.3), the synergy of Rack-Out KVS with dynamic replication and migration (§5.3.4), and the impact of remote read penalties on performance (§5.3.5). Throughout the analysis, we assume identical server building blocks in terms of processing and memory capacity for both

scale-out and Rack-Out KVS configurations.

5.3.1 Load Balancing in Rack-Out Data Serving

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 Rack-Out KVS 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 Rack-Out KVS 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 Rack-Out KVS is $compute_{GF} = GF \times compute_1$. Overall, for a read-only workload, the ideal speedup derived from the Rack-Out KVS model over the scale-out model is:

$$Ideal\ speedup = \frac{\frac{shard\_skew_1 \times L_{avg}}{compute_1}}{\frac{rack\_skew_{GF} \times GF \times L_{avg}}{compute_{GF}}} = \frac{shard\_skew_1}{rack\_skew_{GF}}$$

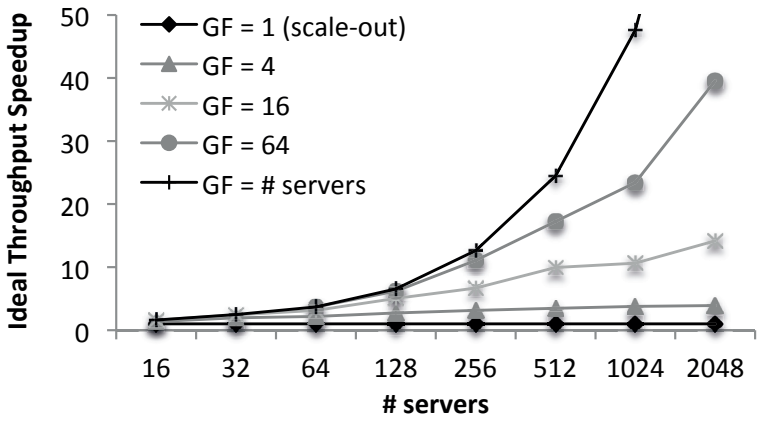


Figure 5.2 – Read-only, perfect intra-rack balancing.

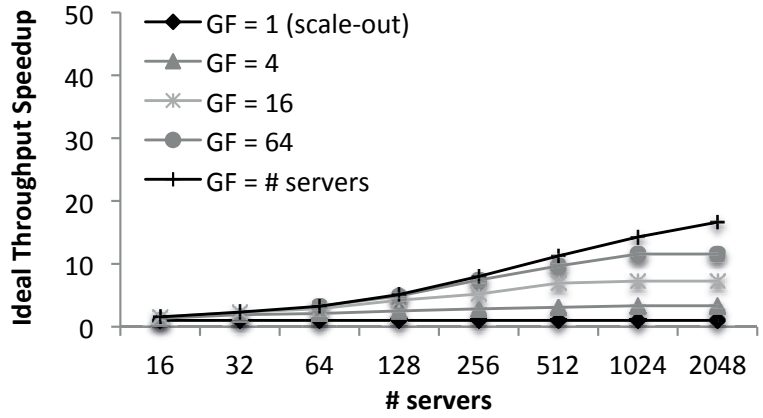


Figure 5.3 – 5% writes, CREW access.

Given the CREW model, *Ideal speedup* 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 Rack-Out KVS, a GF-sized rack's popularity is the sum of the popularity of the GF servers in that rack.

Fig. 5.2 shows the benefit of perfect load balancing within a rack of a given GF according to the *Ideal speedup* equation; for a given 512-server configuration, grouping the servers into RSMP units of 64 servers (i.e., GF=64) provides an ideal speedup opportunity of 16×. Fig. 5.3

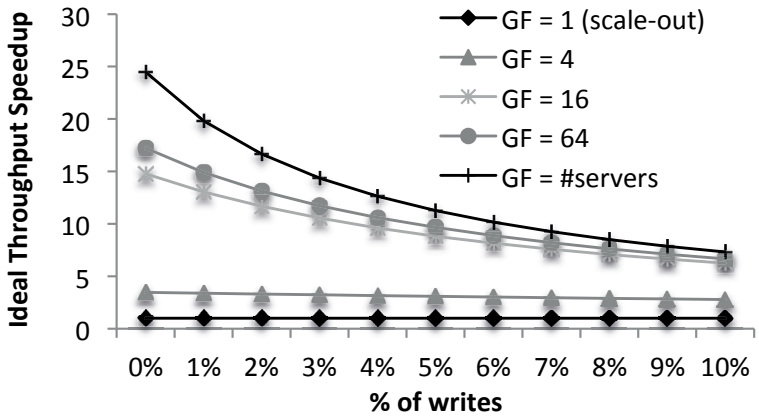


Figure 5.4 – Sensitivity to write %, 512 servers.

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×. Fig. 5.4 illustrates the sensitivity to the read/write mix for various GF configurations.

5.3.2 A Queuing Model for Rack-Out Data Serving

Ideal speedup suggests that the expected benefit of a Rack-Out KVS 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. 5.5 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

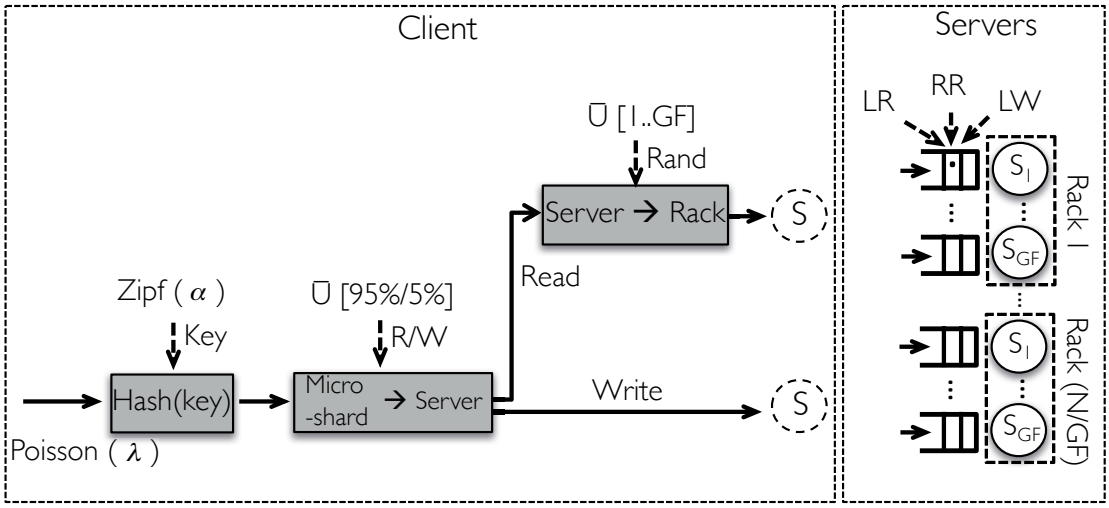


Figure 5.5 – CREW client and server queuing model.

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 RSMP. 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 over the 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 GF and the read/write mix together determine the per-server arrival rate for each request type t, λ_{it}:

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

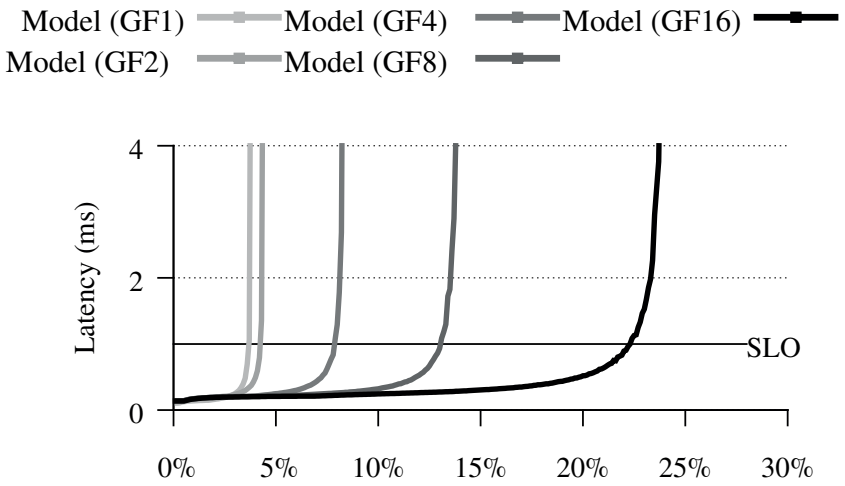
The system is stable if and only if:

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

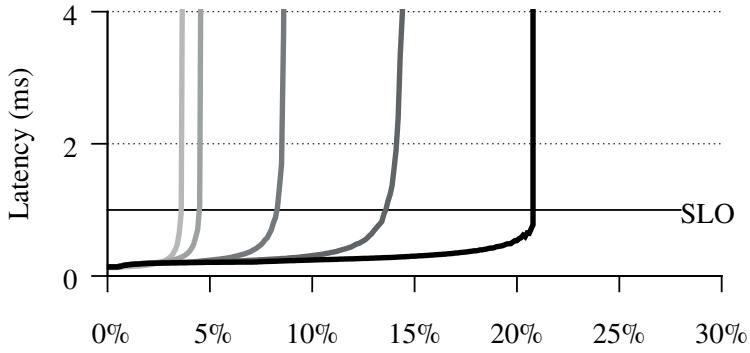
The resulting queuing model depends on the service time distributions S_t. To extract performance results from our queuing model, we instrument it with realistic service times, which we derive from a Rack-Out KVS implementation (RO-KVS) described in §5.4. Using RO-KVS, we measure the maximum node throughput $1/\bar{S_t}$ 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 5.1). The service times include all the CPU processing, such as network packet processing, to service a single key-value lookup or update.

Fig. 5.6a and Fig. 5.6b study 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 5.1). 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. 5.6a shows the impact of Rack-Out KVS organization on read-only workloads. For smaller GFs (including scale-out, GF=1), the datacenter-wide tail latency spikes rapidly as the hottest RSMP 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, Rack-Out KVS 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



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



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

Figure 5.6 – Datacenter-wide 99th percentile latency vs. utilization, determined using the queuing model and RDMA parameters (512 servers, Zipfian $\alpha = 0.99$).

associated with accessing remote memory over RDMA. In comparison, Fig. 5.2'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.

Fig. 5.6b 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 RSMP unit, are still far from saturated. This is particularly obvious with $GF \geq 16$, where the tail latency hardly increases before saturation.

5.3.3 Sensitivity to Skew

Fig. 5.6a and Fig. 5.6b show 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. 5.7 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%.

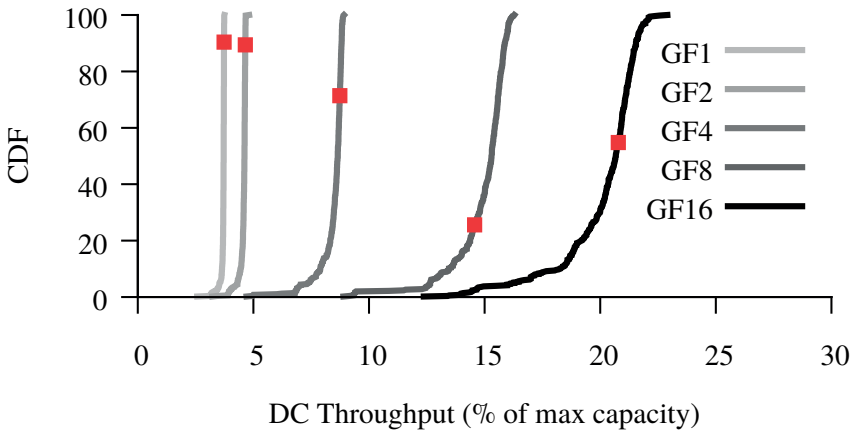


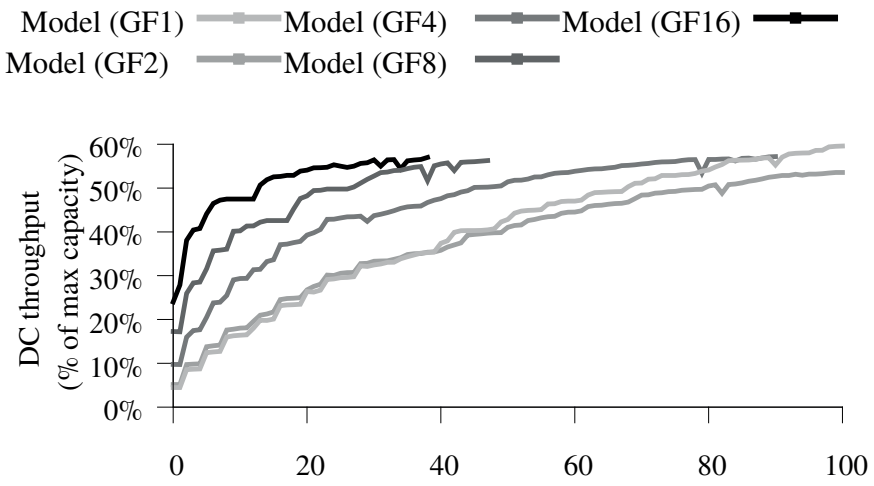
Figure 5.7 – Datacenter utilization for 500 different datasets of 50 million objects with Zipfian $\alpha = 0.99$ data popularity distribution (5% writes). The red dots represent the key distribution used throughout this paper.

5.3.4 Synergy with Dynamic Replication and Migration

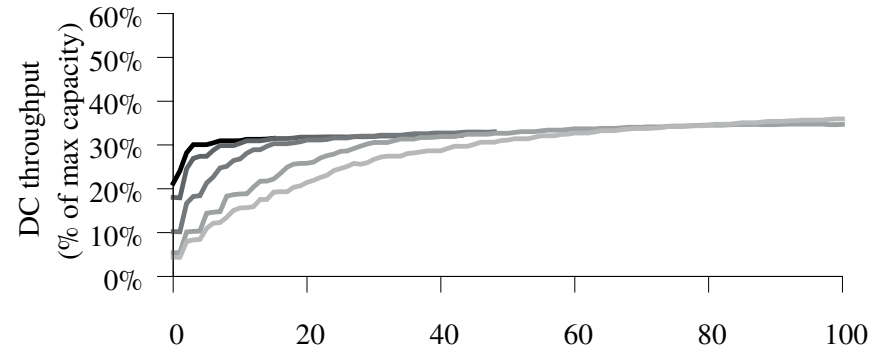
The results of the queuing model in §5.3.2, including Fig. 5.6a and Fig. 5.6b, assume that the key-value store has no provision for dynamic replication or migration. We extend the queuing model to include the dynamic migration and replication of micro-shards. The model extension runs a greedy algorithm operating as follows: (i) it identifies the hottest server node in the cluster and its corresponding RSMP unit; (ii) for that RSMP 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 RSMP 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 overhead (e.g., load monitoring) and consistency maintenance costs, and would have to rely on partial and potentially outdated metrics to make policy decisions [56, 67, 79, 81].

Fig. 5.8a and Fig. 5.8b evaluate the improvement in datacenter utilization after each step in the greedy algorithm for the scale-out (GF=1) and Rack-Out (GF>1) configurations. As in previous experiments, the model is sized with RDMA service times from Table 5.1. The model is configured with 1 billion keys, with Zipfian α = 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 [58] defaults to ~100 micro-shards on a server with only 16 GB of RAM). Fig. 5.8a shows the trend for a read-



(a) Number of replications, read-only



(b) Number of replications, 5% writes

Figure 5.8 – Dynamic replication applied to Rack-Out KVS.

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 Rack-Out KVS when given enough replicas, simply because all of its accesses are local.

Fig. 5.8b 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, Rack-Out KVS 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 [81]. With GF16, only 3 replicas achieve the same result as GF1 with 45 replicas. Therefore, Rack-Out KVS is superior to scale-out KVS as it requires fewer replicas to absorb a given load skew, and consequently reduces the overheads associated with dynamic replication.

5.3.5 The Impact of Faster Remote Reads

The impact of RSMP in Rack-Out KVS 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 Rack-Out KVS on RDMA to estimate its impact through simulation. Modern RDMA technology already provides remote memory access latency that is low enough for effective RSMP [145]. 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 Rack-Out KVS approach.

A representative of such emerging tightly integrated solutions is Scale-Out NUMA (soNUMA) [50, 51, 126], which we discussed in detail in §4. As a reminder, 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. 5.9 shows the speedups at saturation of Rack-Out KVS over a traditional scale-out deployment for different grouping factors as a function of the RR/LR ratio, derived by our Rack-Out KVS queuing model, using the same server configuration and dataset as in §5.3.2 (Fig. 5.6a, Fig. 5.6a). 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

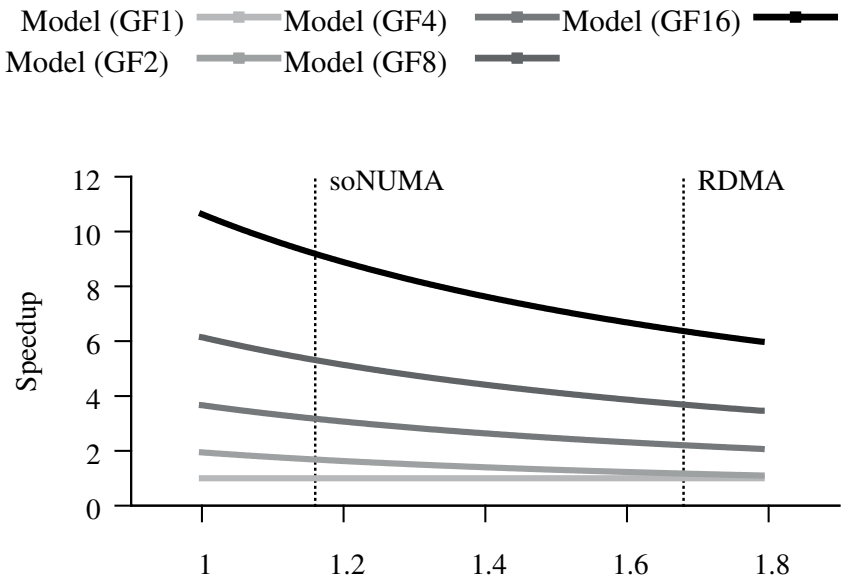


Figure 5.9 – Speedup for different RR/LR ratio (read-only).

on a commercially available solution based on Mellanox ConnectX-3 Pro adapters and (ii) the soNUMA ratio extracted from a soNUMA cycle-accurate simulation model [50] (Table 5.1). While Rack-Out KVS using RDMA already delivers substantial performance improvements over the traditional scale-out approach, soNUMA's faster remote memory access further highlights the potential of Rack-Out KVS. For instance, for GF16, soNUMA delivers 1.45× higher performance improvement than RDMA.

5.4 A Rack-Out Key-Value Store (RO-KVS)

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

5.4.1 RO-KVS Architecture

RO-KVS is built on top of the RSMP software framework described in §3.5. RO-KVS consists of (i) a coordinator server managing the key hash space; (iii) a set of Rack-Out KVS nodes - RSMP units holding the data in a key-value format; (ii) a large number of clients running a key-value workload; The coordinator maintains a distributed hash table (DHT) ring mapping key hash ranges to servers and a map associating each server with its respective RSMP unit.

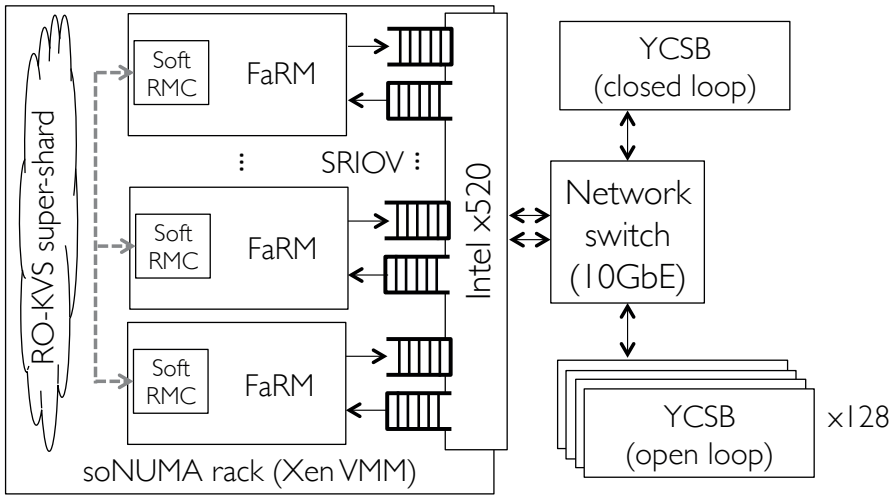


Figure 5.10 – Experimental setup for Rack-Out KVS evaluation.

This information is available to both the clients and the servers. The clients use the DHT ring to route requests directly to the appropriate server for writes or RSMP unit for reads.

To initiate a key-value operation on a server, the clients use `RpcRead` from Table 3.1. RO-KVS servers rely on the DHT ring to ensure the integrity of the KVS. They leverage the server-side API of the RSMP software framework to allocate space for the data in a simple key-value format. That is, each server allocates a contiguous array of objects, where each object represents a hash table bucket. The servers use `AtomicRead` from Table 3.1 to access hash table buckets from the global address space. After they read the target bucket, they search for the requested key-value pair, which sometimes requires additional remote accesses in case there is an overflow chain.

By default, RO-KVS clients rely on random selection of nodes within RSMP units for read requests for load balancing purposes, whereas for writes, RO-KVS always selects the server owning the key-value pair, which is in accordance with the CREW concurrency model. §6 describes a modified version of the RO-KVS client that accounts for the skew in write distribution and average service times to improve the throughput over the original static model.

We decided to study a key-value store as a representative of the NoSQL family of in-memory distributed databases [4]. What distinguishes one NoSQL database from another on RSMP is: (i) the data layout within the global virtual address space (e.g. arrays of objects representing rows - tables or stand-alone tuples), and (ii) the metadata used for routing (DHT or index).

Thus, it is easy to imagine any other type of NoSQL database built on top of the RSMP abstraction using the same framework API [31, 41]. Because of the scale-out property, most such systems suffer from skew and the conclusions we make in this thesis using RO-KVS are equally applicable to the general class of NoSQL databases.

5.4.2 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 §5.3.2. We use the latter to measure the throughput of RO-KVS under SLO tail latency constraints for different RSMP configurations, and compare the results to the 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 5.1 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 figures 5.6a, 5.6b, 5.7, 5.8a, 5.8b and 5.9. Table 5.1 also shows projected service times for soNUMA, 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 2.8μsin our RDMA setup for a 512-byte hashtable bucket) and the soNUMA remote read latency (~240ns [50]). For that projected latency, the resulting RR/LR ratio for soNUMA is 1.16.

Table 5.1 – Average service time of basic operations used to size the queuing model.

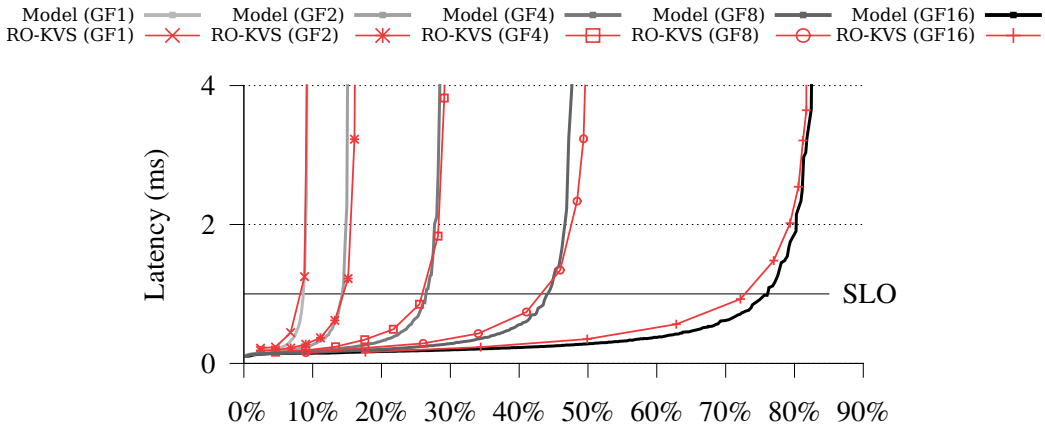
Operation	LR	RR	LW	RR/LR	Propagation
RDMA	5µ*s*	8.4µ*s*	6.9µ*s*	1.68	34.9µ*s*
soNUMA (projected)		5.8µ*s*		1.16	

Fig. 5.10 describes the emulation platform for the soNUMA architecture. This platform [126] 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 RSMP 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 5.1).

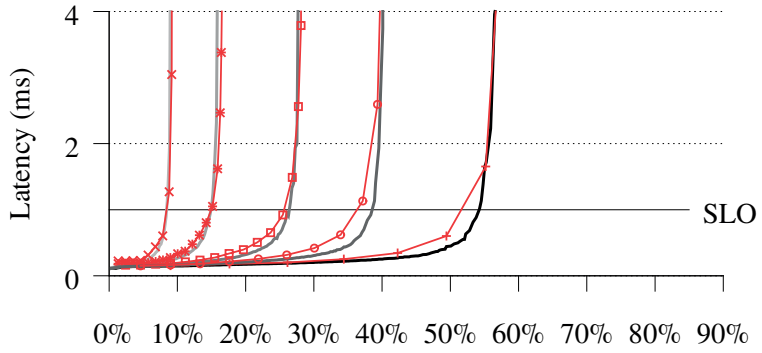
The load generators use the RO-KVS client library to run a YCSB workload [46]. The client library uses SpookyHash [87], a public domain hash function that produces well-balanced hash values, to map keys to servers. Nine external servers generate load, and a tenth runs a closed-loop YCSB process issuing 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.

5.4.3 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. 5.7. Even though 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., hottest rack). These 16 servers are organized in $16/GF$ RSMP units for the various experiments. Focusing only on a subset of servers is sufficient as long as that subset processes a significant fraction of the incoming traffic (hottest). This is because in a full-scale RSMP environment, typically, the hottest RSMP unit alone determines the tail latency. We report the



(b) Rack throughput (% of max capacity), read-only



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

Figure 5.11 – 99th percentile latency vs. throughput for the hottest rack measured on the experimental platform.

tail latency of requests issued to the hottest 16-server group in our testbed. For comparison purposes, we extract the tail latency for the same group of servers from the queuing model.

Fig. 5.11a and Fig. 5.11b show 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 Rack-Out KVS queuing model configured with the soNUMA parameters (Table 5.1). 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. 5.11a), 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. 5.11b), the inability to load-balance writes limits the maximum rack throughput with GF=16 to 57% vs. 84% in the read-only workload. 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 Rack-Out KVS 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. The key to the model's precision was to: i) implement a simple, *run-to-completion* request processing in the RSMP software framework (no pipelining), which is how requests get processed in our discrete event-based simulation environment, and ii) instrument the model with the service times measured on a real platform.

A Rack-Out KVS 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 [16, 31, 46, 140]. In the next chapter, we extend the original RO-KVS design with support for adaptive load balancing and demonstrate how RSMP and CREW can help improve throughput for workloads with a larger fraction of writes.

Overall, the Rack-Out KVS 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 Rack-Out KVS organizations with larger GFs, which exceed our platform's scale limitations.

5.5 Discussion

User-level network stacks and domain-specific operating systems. A key assumption of RSMP is that servers have low-latency access to each other's memory within their respective racks, enabling efficient load balancing. The major source of overhead in processing client requests on Linux-based systems comes from TCP/IP. Yasukata et al. [167] showed that processing a single packet on a Linux server takes 3.75us. The local read service time (LR) on Rack-Out KVS is 5us (Table 5.1), which means a significant fraction of time is spent on packet

processing. To balance such a high network overhead accross an RSMP unit, it is sufficient to use a microsecond-scale fabric (even in the synchronous mode), such as RDMA, to enable shared access to data.

The emergence of lightweight, user-level network stacks [88] and specialized, domain-specific operating systems [23] will change the trade-offs in RSMP. In particular, like it does not make much sense to use TCP/IP to enable shared access under Linux-based RSMP, in the future, it will not make much sense to combine synchronous RDMA and a specialized OS enabling a high packet processing rate at low latency. Our expectation is that a more advanced, nanosecond-scale fabric, such as Scale-Out NUMA, will be necessary for RSMP to make an impact and maintain good productivity (i.e., ease of programming). An extreme case of this would be to use RDMA user-level messaging for the RPC traffic (client-server), which has become popular recently [90], and soNUMA for remote access. Also, in such a design, it would make sense to enable the RDMA HCAs to convert remote accesses into soNUMA rack-local remote accesses, because an HCA could saturate due to skew, while the rest of the HCAs are underutilized. So, the same load balancing approach as in Rack-Out KVS would be applicable to distribute the load across a rack, assuming sufficient fabric bisection bandwidth.

Network oversubscription. The Rack-Out KVS model assumes that the maximum server processing rate is limited by the CPU and the latency of the internal fabric. The high packet processing overhead puts significant pressure on the CPU and, thus, the rate at which a server processes client key-value requests is quite low. As a consequence of the low processing rate, the network carrying the client-server traffic is underutilized. It takes three hyper-threaded cores and a specialized, domain-specific OS that has been optimized for high-throughput, low-latency key-value processing to saturate a 10Gbps NIC, assuming only local accesses to memory (i.e., no RSMP) [23]. The Rack-Out KVS assumes Linux for packet processing and RDMA or soNUMA for memory accesses, allowing for high network oversubscription at the rack level. Our soNUMA configuration allocates 1.6Gbps per server node; one 10Gbps Intel NIC connected via SRIOV to 16 servers (GF=16). Assuming only local reads (LR), a single server node processes 200.000 key-value lookups per second, which is 0.76Gbps, assuming 512 byte messages. Thus, a 10 to 1 oversubscription ratio seems more than sufficient for our system.

ACID transactions and consistency. A key advantage of distributed NoSQL databases over traditional relational databases is the ability to horizontaly scale the server resources at the expense of weaker consistency guarantees. Unlike relational databases, NoSQL systems store data in an unstructured or semi-structured format rather than tabular relations. The combination of scale-out architecture and unstructured data formats limits the programming semantics of NoSQL databases. They typically expose a narrow API, consisting of trivial *get* and *put* operations, without support for ACID transactions. FaRM does not assume scale-out architecture and, thus, provides serializable transactions over RDMA. Unlike FaRM, RSMP is a scale-out system where transaction may span multiple racks, making them costly for the user. Even though transactions are not efficient under RSMP, a consistency model is necessary for replication. In RSMP, replication happens only across different racks and never within the same rack. For performance reasons, we assume an eventual consistency model, which is what most distributed NoSQL databases assume.

5.6 Related Work

Resource pooling. RSMP 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 [103] 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 [59, 133].

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 [102] 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 [135] is based on EREW, while FaRM [58] and Pilaf [117] implement CREW, where reads are direct one-sided accesses to remote memory, while writes are transformed into RPCs. To our knowledge, DrTM [163] 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 [81, 141]. Rack-Out KVS is synergistic with dynamic replication and substantially reduces the need to dynamically replicate content. Fan et al. [67] 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.

Front-end caching. Besides dynamic replication and RSMP, an alternative way to deal with popularity skew is caching. Fan et al. [67] proposed leveraging skew to achieve better locality through caching the most popular items in front-end servers (i.e. web servers). This approach assumes distributions in which very few items contribute to the imbalance. Less skewed distributions in which a larger number of somewhat popular items contribute to the imbalance may be harder to deal with due limited cache capacity. Most importantly, front-end caching introduces consistency issues among front-end servers, as well as between front-end and back-end servers. Depending on scale (GF), an RSMP datacenter configuration is highly likely to be sufficient for an online service provider to absorb the skew, thus, steering clear of consistency overheads.

5.7 Summary

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 RSMP, 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 Rack-Out KVS 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 Rack-Out KVS 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.

6 Adaptive Load Balancing Using RSMP for Data Serving

Rack-scale memory pooling (RSMP) reduces the scale-out-induced load imbalance that exists in data serving workloads. In the previous chapter, we showed how using RSMP in distributed key-value stores (KVS) reduces skew and, thus, facilitates resource utilization, while respecting tight tail latency SLO. However, the impact of RSMP gradually diminishes with the increase in write load, which is a direct consequence of using the *concurrent-read/exclusive write* (CREW) model in combination with random scheduling of reads. This chapter proposes a new scheduling mechanism for Rack-Out KVS that provides greater performance benefits of RSMP for read-write data serving workloads (≥ 5% writes). To distinguish between different scheduling approaches in Rack-Out KVS, we refer to the original *stateless* (random) scheduling that we described in the previous chapter as `Static-Rack-Out`, and we refer to the new adaptive scheduling as `Adaptive-Rack-Out`.

6.1 Skew and Service Time Variability

High-performance fabrics with support for one-sided access primitives, such as Infiniband or Scale-Out NUMA (soNUMA), are the key enablers of RSMP. RSMP leverages such fabrics to expose a global address space to software and ease application development, while providing significant performance benefits to the user. Using RSMP, a group of servers can be perceived as one giant server, where the RSMP software framework allocates resources on behalf of applications and exposes an API for transparent access to the global address space (Table 3.1).

However, building scalable capacity units using RSMP can lead to load imbalance within a single unit. In particular, scheduling read requests randomly across an RSMP unit may not be the optimal choice; the larger the RSMP unit and the lower the read/write ratio, the higher the impact of writes on imbalance due to CREW [128]. On top of that, performance interference with either background services or other applications leads to high variance in service times and, hence, tail latency violations [54].

The skew that exists within an RSMP unit is the work that is not distributed evenly across the rack and leads to non-uniform utilization of resources. In the case of Rack-Out KVS, that work represents the write portion of the input load. Even for what most people consider to be a read-dominated workload - 5% writes, rack-level load imbalance is visible and negatively impacts the application throughput. Under CREW, writes are hashed to individual servers and, thus, an RSMP unit is typically not uniformly utilized because the time spent on processing writes varies across the servers. For workloads with a larger fraction of writes in particular, the `Static-Rack-Out` scheduling may not be sufficient, as writes represent a major source of imbalance.

Besides writes, any service that is triggered periodically, such as garbage collection, may contribute to load imbalance. Next, processing times of different requests may vary depending on the amount of work that needs to be done. Longer processing times require from a server to stall other incoming requests, which reduces the processing rate. In addition to that, because of the high fan-out nature of data serving, the larger the number of servers involved in servicing a single user request, the higher the likelihood that the tail latency SLO will be violated [54].

To deal with load imbalance within RSMP units more effectively, we propose adaptive load balancing - `Adaptive-Rack-Out`. Our approach consists of a monitoring system that keeps track of the servers' load, and a client-side scheduler that leverages the metrics obtained from the monitor to take smarter scheduling decisions. In the following text, we describe the design and implementation of the monitor and the `Adaptive-Rack-Out` algorithms, the goal of which is to improve upon the original static design (`Static-Rack-Out`) of Rack-Out KVS from the previous chapter.

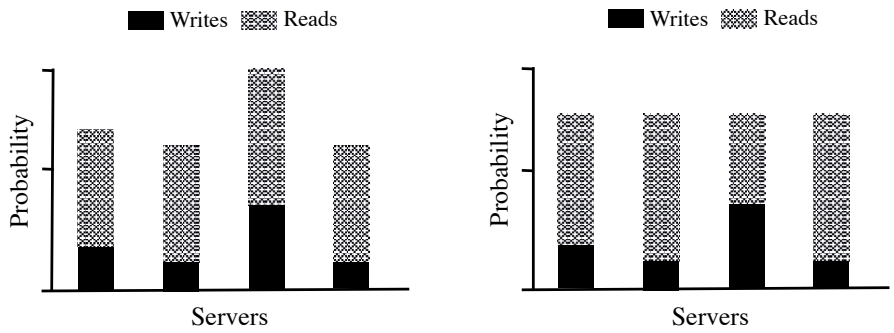


Figure 6.1 – Illustration of combined read/write distributions for `Static-Rack-Out` (left) and `Adaptive-Rack-Out` (right).

6.2 Adaptive Load Balancing for Read-Write Workloads

`Static-Rack-Out` is an efficient, stateless approach to load balancing for skewed workloads dominated by reads. The key limitation of `Static-Rack-Out` is that it does not account for the imbalance within individual RSMP units. Such imbalance exists due to performance interference, periodic activity within applications (e.g. garbage collection), skewed write distribution, remote accesses or accesses to disk, etc. For instance, `Static-Rack-Out` is oblivious to the CREW-induced skew and it randomly schedules read requests across an RSMP unit, without taking into account the *exclusive-writes* property of CREW. Thus, `Static-Rack-Out` often leads to skewed aggregate distribution of load across the servers, which is illustrated on the left hand side of Fig. 6.1. `Static-Rack-Out` also does not account for variable service times, but assumes that each server of an RSMP unit processes incoming requsts at similar pace. Uniform service times is, at least, not trivial to achieve in shared envirnoments where slowdowns happen regularly.

Adaptive load balancing for Rack-Out KVS - `Adaptive-Rack-Out` is a scheduling mechanism that adapts the read workload to the load imbalance within individual RSMP units. For read-write workloads ($\geq$ 5% writes), `Adaptive-Rack-Out` generates a read distribution for each RSMP unit that is inversely proportional to its write distribution. The outcome of `Adaptive-Rack-Out` is illustrated on the right hand side of Fig. 6.1, where the read and the write distributions form a uniform aggregate distribution. By sampling the computed read distribution when scheduling read requests, the clients adapt to the write load to achieve, in the ideal case, uniform utilization of the RSMP unit. To cope with non-uniform service times,

`Adaptive-Rack-Out` further adjusts the read distribution to reflect the processing rates of different servers.

The `Adaptive-Rack-Out` 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 (Fig. 6.2). A coordinator client uses the information collected by the monitors to generate the read distribution of its associated RSMP unit that it subsequently broadcast to the other clients. All the clients then regularly sample the read distributions when scheduling read requests. We first look at the monitor design and explain how the clients take advantage of the collected information.

6.2.1 Load Monitoring

The monitor in `Adaptive-Rack-Out` 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 6.2 illustrates the process of periodically obtaining the number of processed writes (`wr_cnt`) and the average service time (`avg_st`) from each server of a single rack. The coordinator clients compute the read distributions for their respective RSMP units based on the information obtained from the servers.

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 concept of Rack-Out KVS 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 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 transfering 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

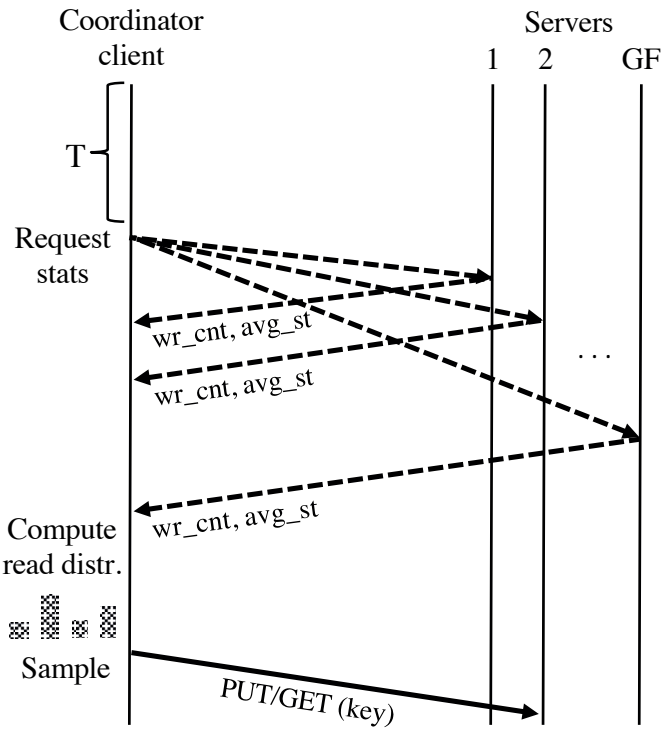


Figure 6.2 – Message exchange in `Adaptive-Rack-Out`.

from remote reads alone is negligible as compared to disk writebacks, garbage collection, or interference with any batch application performing heavy computation.

6.2.2 Scheduling Algorithms

In `Adaptive-Rack-Out`, clients take 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 write requests, the client schedules read requests such that each server's CPU within an RSMP unit is, in the ideal case, equally occupied. This is achieved by periodically computing a read probability distribution that in combination with the current write probability distribution should, in the ideal case, lead to uniform utilization of the associated RSMP unit. For workloads with a large fraction of writes (i.e. above 10%), it is not possible to distribute the load evenly across the servers, which is a limitation of CREW.

Coordinator clients retrieve the information collected by the monitors periodically either

through an *out-of-band* mechanism or using regular data traffic (piggybacking) (Fig. 6.2). 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 set based on how dynamic the system is; that is, how often the popularity data distribution and the service times change. In reality, we do not expect the key distribution to change frequently [81].

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). The `Adaptive-Rack-Out` mechanism comprises two algorithms: (i) Algorithm 1, which generates the read probability distribution based on the current write distribution, and (ii) Algorithm 2, which adjusts the read distribution generated by Algorithm 1 to reflect the variability in average service times.

Figure 6.1 illustrates the effect of Alg. 1; the figure on the left illustrates the output of the original, `Static-Rack-Out` scheduling, and the figure on the right shows how `Adaptive-Rack-Out` adjusts the read distribution such that each server receives an equal amount of read and write requests. The coordinator clients compute the read probability for each server i of their associated RSMP units 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 an RSMP 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 adjusts the read distributions using the mean service times obtained from the monitors. The scheduler first normalizes the service times to 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 $\frac{1}{GF}$.

Instead of computing the variance and then standard deviation, the algorithm subtracts the difference from the read probability:

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

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.

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 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
```

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 with more than 10% of writes, the aggregate probability of selecting the hottest server is 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 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 ← 0, i < GF, i+ = 1 do
4:     avg_st_total ← avg_st_total + avg_st[i]

5: for i ← 0, i < GF, i+ = 1 do
6:     avg_st_nr[i] ← avg_st[i] / avg_st_total

7: for i ← 0, i < GF, i+ = 1 do
8:     diff ← rd_pr[i] − (avg_st_nr[i] − 1 / GF)
9:     if diff < 0 then
10:        to_sub ← abs(diff) / (GF − 1)
11:        rd_pr[i] ← 0
12:    else
13:        rd_pr[i] ← diff

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

Algorithm 2 first normalizes the service times to 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 [54].

By using the periodically adjusted read distribution, each client decides which server within the target RSMP unit should process the next request. Each request results in: (i) determining

the RSMP unit ID based on key, and (ii) sampling the read distribution of the associated rack to pick a server.

6.3 Implementation in RO-KVS

We implement the `Adaptive-Rack-Out` 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 10 seconds, which we find sufficient given that key distributions in general do not change frequently [81]. When a coordinator client receives the updated variables, it adjusts the read distribution of its associated RSMP unit 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 envirnoment, 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 a replicated in-memory key-value store that does not use disks. Nevertheless, our `Adaptive-Rack-Out` mechanism (Alg. 2, in particular) accounts for even the slightest variations in service times, such as those caused by remote accesses.

For the purpose of evaluating the impact of `Adaptive-Rack-Out`, we use the same setup described in the previous chapter and focus only on platform evaluation. We focus on the most popular group of 16 servers in a 512-server configuration. We dedicate another 5 servers to run a group of load generator clients, plus one more server for the coordinator client. We evaluate `Adaptive-Rack-Out` using our RSMP unit configuration based on soNUMA comprising 16 nodes.

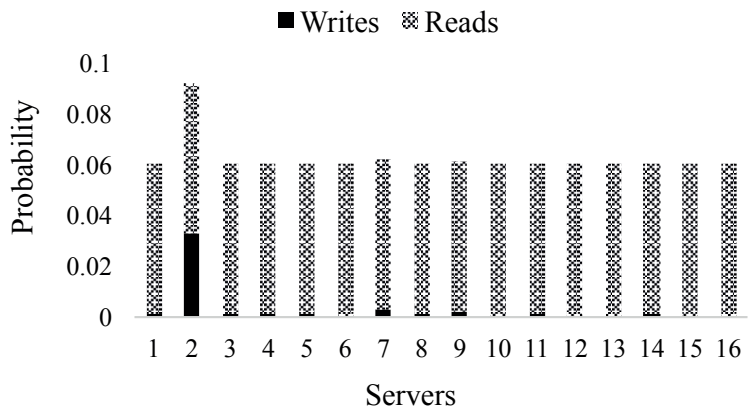


Figure 6.3 – Combined read/write probability distribution for the hottest RSMP unit in `Static-Rack-Out`

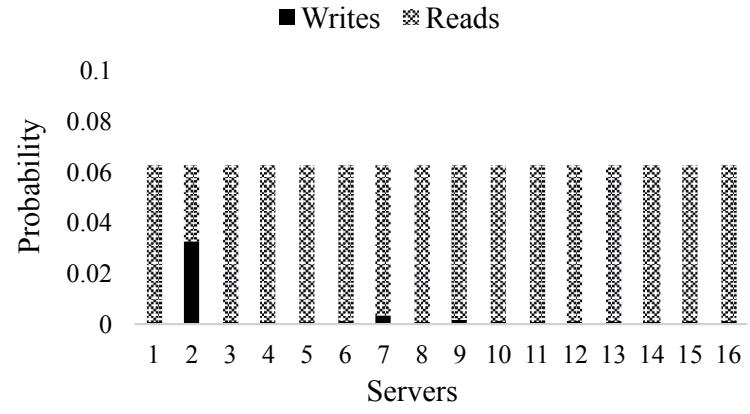


Figure 6.4 – Combined read/write probability distribution for the hottest RSMP unit in `Adaptive-Rack-Out` (Alg. 1)

6.4 Impact of Adaptive Load Balancing on Throughput

To emphasize the benefits of `Adaptive-Rack-Out` (ARO), we focus on read-write workloads and the GF=16 RSMP configuration. We assume the same workload with 5% of writes as in the previous experiments. We, first, study how Algorithm 1 (`RO-KVS (ARO1)`) adjusts the read probability distribution 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%.

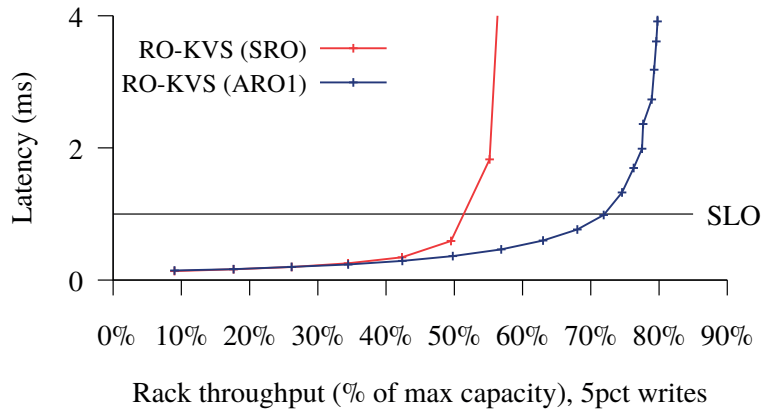


Figure 6.5 – `Static-Rack-Out` vs. `Adaptive-Rack-Out` (Alg. 1), 99th-pct latency for 5% of writes

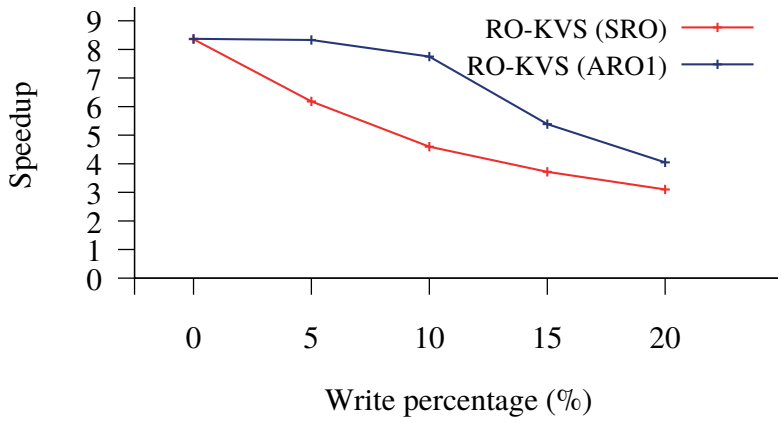


Figure 6.6 – Speedup at saturation for different write percentages.

Fig. 6.3 shows the combined read/write probability distribution for `Static-Rack-Out` (SRO) for the hottest 16-server RSMP 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 creates a read distribution such that each server within the hottest RSMP unit processes a similar aggregate amount of read and write requests (Fig. 6.4). The impact of `Adaptive-Rack-Out` is illustrated on Fig. 6.5, where Algorithm 1 (`RO-KVS (ARO1)`) outperforms `Static-Rack-Out` and reaches almost 80% of rack utilization, similar to `Static-Rack-Out` scheduling and a read-only workload (Fig 5.11a), and a speedup of 1.36× over `Static-Rack-Out` for the same 95/5 workload. The improvement comes from making the read distribution inversely proportional to the write distribution, as illustrated on Fig. 6.4. Unlike `Static-Rack-Out` scheduling, where a single server saturates and drives the

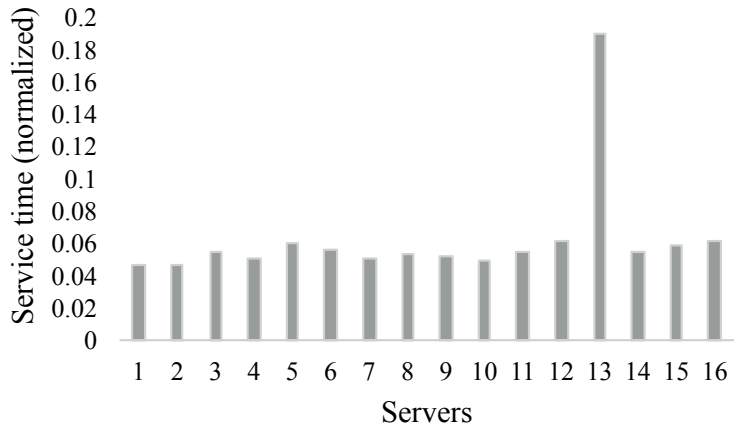


Figure 6.7 – Average service times. Interference injected on Server 13.

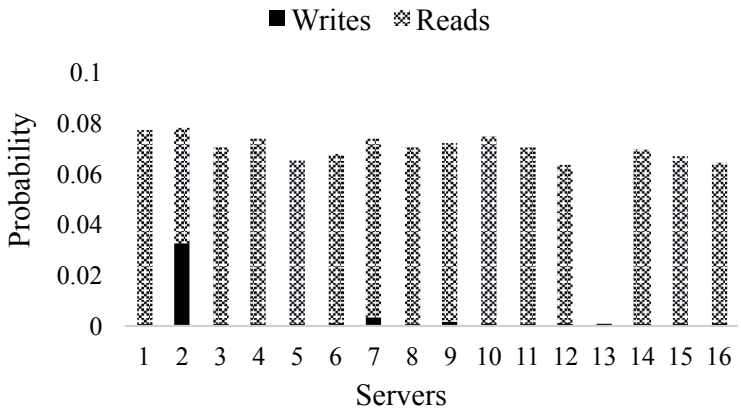


Figure 6.8 – Combined read/write probability distribution in `Adaptive-Rack-Out` (Alg. 1 and Alg. 2). Server 13. is slowed down due to interference and is assigned with 0 read probability.

99th-percentile latency above the SLO, with `Adaptive-Rack-Out`, all the servers of the hottest RSMP unit saturate concurrently, utilizing the servers uniformly and pushing the aggregate throughput (Fig. 6.5). The behavior of RO-KVS under `Adaptive-Rack-Out` 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, 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 $\frac{1}{GF}$. The last code snippet of Algorithm 1. handles this special case.

Fig. 6.6 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,

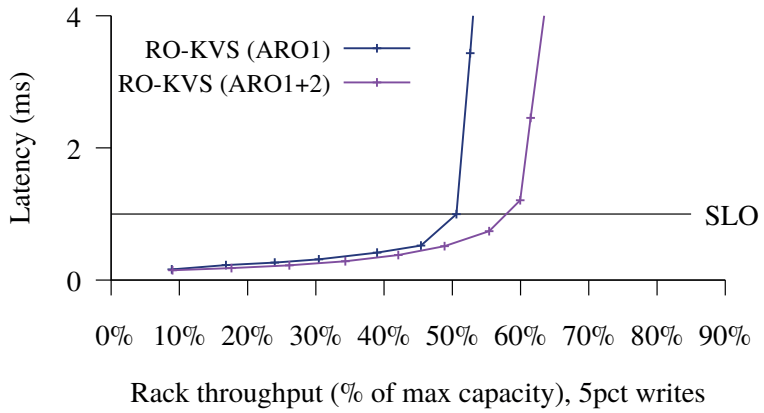


Figure 6.9 – `Adaptive-Rack-Out` (Alg. 1) vs `Adaptive-Rack-Out` (Alg. 1 and Alg. 2), 99th-pct latency for 5% of writes, with interference on one node injected.

the speedup decreases. We observe that the standard `Static-Rack-Out` scheduling with uniform read distribution performs worse than `Adaptive-Rack-Out` because the hottest server of the rack saturates faster; it handles both read and write load, whereas with adaptive load balancing the hottest server is unballasted from the read load at the expense of handling a higher read load on the other servers. In `Static-Rack-Out`, the larger the fraction of write requests, the faster the system saturates and the smaller the speedup, due to CREW. Similarly, with `Adaptive-Rack-Out`, the larger the fraction of writes, the harder it is to utilize the servers of an RSMP 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 (`RO-KVS (ARO1+2)`), we inject interference into one of the servers and study how the scheduler adjusts the read distribution accordingly. We, first, look at the service times that the clients periodically obtain from the monitors. Figure 6.7 shows the service times normalized to 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 on Figure 6.8. Server 13 is assigned with 0 read probability because of the high average service time.

Figure 6.9 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 Fig. 6.8, boosting the rack's utilization to 65% at saturation. We measure the overhead of code profiling to be 9% and the throughput loss for one node is 6.2%. Reducing the sampling frequency leads to smaller profiling overhead, but it degrades the accuracy and, consequently, results in untimely SLO violation. One approach to reducing the cost and improving the effectiveness of Algorithm 2 would be to measure the service time on the network interface for each inbound packet carrying a key-value request, and tag the reply packet with the service time.

6.5 Summary

We further explored the role of RSMP in scale-out data serving using CREW. The static scheduling approach - Static-Rack-Out described in the previous chapter assumes read-dominated workloads where writes have negligible impact on the overall throughput. For lower read to write ratios, scheduling reads uniformly across racks is not optimal as some servers will be under higher load than the other because of exclusive writes. Therefore, we proposed a monitoring system and a new scheduling mechanism - Adaptive-Rack-Out that accounts for skewed server utilization to make smarter scheduling decisions. We observe an improvement of 1.36× as compared to Static-Rack-Out for workloads with 5% of writes using Adaptive-Rack-Out.

7 Conclusion

Rack-scale memory pooling (RSMP) is a research proposal to aggregate the resources of a rack using a one-sided fabric, and use such a rack as the building block for large-scale datacenters. This thesis studies the necessary requirements to efficiently connect the servers within a rack and provide the abstraction of a virtual global address space. On top of that, the thesis describes how to take advantage of a coarser-grain scale-out architecture comprising a smaller number of larger capacity units - RSMP units or racks in core datacenter software.

We first looked at an RSMP design leveraging RDMA technology. To build an RSMP unit using RDMA hardware, it is necessary to equip each server with an RDMA controller - host channel adapter (HCA) and connect them to a secondary *top-of-rack* switch supporting RDMA. We concluded that the existing RDMA technology is sufficient for RSMP, providing an improvement over the standard scale-out deployment. This thesis also showed that existing RDMA designs have scalability limitations, such as the PCIe/DMA mechanism that is commonly used to connect network adapters.

To improve the effectiveness of RSMP, we proposed a set of design principles that significantly improve upon the RDMA technology. In particular, we proposed Scale-Out NUMA (soNUMA), which is a rack-scale system design leveraging a memory subsystem (NUMA) to connect a group of servers and provide low-latency, high-bandwidth access to remote memory. soNUMA does not provide global cache coherence within the rack for scalability reasons, but exposes a hardware-software interface for explicit remote memory access. The key concept in soNUMA is

to fold the fabric controller into the cache hierarchy of the CPU, reducing the cost of interaction between applications and the controller. The use of a NUMA interconnect simplifies the protocol design and, thus, reduces the amount of state and header size. This simplification of the underlying interconnect allowed us to reduce the complexity of the transport protocol that is in charge of transferring data from and to remote memory, and make it easier to intergrate the protocol controller. We showed that soNUMA reduces the remote access latency by 5x as compared to the best RDMA hardware available at the time of writing.

We described how future datacenter software could take advantage of RSMP. A scale-out architecture comprising a set of RSMP units can help improve the throughput of in-memory data serving workloads, without violating application's SLO. Our distributed NoSQL design, dubbed Rack-Out KVS, treats RSMP units as large KVS nodes hosting *super-shards*, where each server can look up any key within its local rack using one-sided reads. By effectively reducing the number of participating units, the load imbalance induced by heavily-skewed popularity distribution becomes less significant, leading to better application throughput, while maintaining low latency. To demonstrate the impact of this KVS design, we use both RDMA and soNUMA hardware, with up to 16 servers per RSMP unit. Our results show that Rack-Out KVS achieves an increase of 6x in throughput with RDMA over the standard scale-out alternative, and 8.6x using soNUMA. In addition to a working prototype, we introduced a queueing model that one can use to evaluate Rack-Out KVS under an arbitrary RSMP configuration and predict the performance impact. We validated the model using the results obtained on our RSMP platforms.

Finally, we proposed an adaptive load balancing algorithm that makes the most out of CREW. In particular, our algorithm uses the information about server load and average service time to schedule read requests such that each server of an RSMP unit is, in the ideal case, equally occupied. We showed how this new approach to scheduling leads to additional 1.36× improvement for workloads with 5% of writes over the original RSMP design.

Bibliography

[1] BigPipe: Pipelining web pages for high performance. https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919.

[2] Internet Growth Statistics. http://www.internetworldstats.com/emarketing.htm.

[3] RAMCloud Infiniband Network. https://ramcloud.atlassian.net/wiki/display/RAM/New+Infiniband+Fabric+Notes.

[4] Your Ultimate Guide to the Non-Relational Universe. http://nosql-database.org.

[5] Infiniband architecture specification volume 1. 2002.

[6] Ieee. 802.11qbb. priority based flow control. 2011.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. A. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pages 2–13, 1995.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, pages 63–74, 2008.

[9] Amazon. Amazon Elasticache. http://aws.amazon.com/elasticache/, 2011.

[10] C. Amza, A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. ThreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[11] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, 2009.

[12] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks Of Workstations). *IEEE Micro*, 15(1):54–64, 1995.

[13] Apache Software Foundation. Apache HBase. http://hbase.apache.org., 2016.

[14] 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.

[15] K. Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. USENIX FAST keynote. 2014.

[16] 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.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[18] L. Barroso. Programming a warehouse-scale computer. PLDI keynote. 2016.

[19] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[20] L. A. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[21] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[22] M. Beck and M. Kagan. Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching*, pages 9–15, 2011.

[23] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[24] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, pages 315–324, 2006.

[25] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 142–153, 1994.

[26] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, Version 2.0. 2007.

[27] Boston Limited. Boston Limited Unveil Their Revolutionary Boston Viridis, 2011.

[28] 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)*, page 335, 2010.

[29] E. A. Brewer. Pushing the CAP: Strategies for Consistency and Availability. *IEEE Computer*, 45(2):23–29, 2012.

[30] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.

[31] 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.

[32] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[33] 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.

[34] 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.

[35] Calxeda Inc. Calxeda Energy Core ECX-1000 Fabric Switch, 2012.

[36] Calxeda Inc. ECX-1000 Technical Specifications, 2012.

[37] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 152–164, 1991.

[38] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[39] Cavium Networks. Cavium Announces Availability of ThunderX^{TM} 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.

[40] 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.

[41] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 205–218, 2006.

[42] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–25, 1995.

[43] M. Chapman and G. Heiser. vNUMA: A Virtual Shared-Memory Multiprocessor. In *Proceedings of the 2009 conference on USENIX Annual Technical Conference*, 2009.

[44] C. Coarfa, Y. Dotsenko, J. M. Mellor-Crummey, F. Cantonnet, T. A. El-Ghazawi, A. Mohanti, Y. Yao, and D. G. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 36–47, 2005.

[45] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[46] 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.

[47] D. Crupnicoff. Personal communication (Mellanox Corp.), 2013.

[48] D. E. Culler, A. C. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (SC)*, pages 262–273, 1993.

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

[50] 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.

[51] 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)*, pages 1–13, 2016.

[52] M. Davis and D. Borland. System and Method for High-Performance, Low-Power Data Center Interconnect Fabric. WO Patent 2,011,053,488, 2011.

[53] 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.

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

[55] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[56] 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.

[57] A. Dhodapkar, G. Lauterbach, S. Li, D. Mallick, J. Bauman, S. Kanthadai, T. Kuzuhara, G. S. M. Xu, and C. Zhang. SeaMicro SM10000-64 Server: Building Datacenter Servers Using Cell Phone Chips. In *Proceedings of the 23rd IEEE HotChips Symposium*, 2011.

[58] 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.

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

[60] 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.

[61] Facebook. Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network.

[62] Facebook. Apache Cassandra. http://cassandra.apache.org/, 2008.

[63] 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-/., 2015.

[64] Facebook. Data Sharing on traffic pattern inside Facebook's datacenter network. https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/., 2017.

[65] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC)*, pages 380–389, 1994.

[66] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 229–240, 1997.

[67] 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.

[68] 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.

[69] B. Fitzpatrick. Memcached. http://memcached.org/, 2003.

[70] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 333–346, 2013.

[71] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2):12–18, 1996.

[72] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 51–62, 2009.

[73] L. Group. Oracle Shrink Sparc M7. *Microprocessor Report*, 2015.

[74] 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*, pages 202–215, 2016.

[75] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 139–152, 2015.

[76] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, 2011.

[77] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 38–50, 1994.

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

[79] 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.

[80] HPC Advisory Council. Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing. http://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf, 2009.

[81] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceed-*

ings of The 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII), pages 8:1–8:7, 2014.

[82] R. Huggahalli, R. R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 50–59, 2005.

[83] 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.

[84] 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.

[85] IDC. THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. https://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf, 2012.

[86] IEEE. *IEEE 802.1Qbb: Priority-Based Flow Control*, 2011.

[87] B. Jenkins. SpookyHash: a 128-bit noncryptographic hash. http://burtleburtle.net/bob/hash/spooky.html., 2011.

[88] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[89] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 437–450, 2016.

[90] A. Kalia, M. Kaminsky, and D. G. Andersen. 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)*, pages 185–201, 2016.

[91] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on the Theory of Computing (STOC)*, pages 654–663, 1997.

[92] R. Kessler and J. Schwarzmeier. Cray T3D: A New Dimension for Cray Research. In *Compcon Spring '93, Digest of Papers*, 1993.

[93] Y. O. Koçberber, B. Grot, J. Picorel, B. Falsafi, K. T. Lim, and P. Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, 2013.

[94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 302–313, 1994.

[95] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 17th International Conference on World Wide Web (WWW)*, pages 591–600, 2010.

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

[97] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. K. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, 2015.

[98] S. Legtchenko, N. Chen, D. Cletheroe, A. Rowstron, H. Williams, and X. Zhao. Xfabric: a reconfigurable in-rack network for rack-scale computers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 15–29, 2016.

[99] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[100] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[101] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey. Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform. *ACM Trans. Comput. Syst.*, 34(2):5:1–5:30, 2016.

[102] 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.

[103] 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.

[104] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 36–47, 2013.

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

[106] LinkedIn. Project Voldemort. http://www.project-voldemort.com/, 2009.

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

[108] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[109] Q. Liu and R. D. Russell. A performance study of InfiniBand fourteen data rate (FDR). page 16, 2014.

[110] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Improving Resource Efficiency at Scale with Heracles. *ACM Trans. Comput. Syst.*, 34(2):6:1–6:33, 2016.

[111] 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.

[112] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[113] 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.

[114] Mellanox Corp. Mellanox SX6025 Switch Product Brief. http://www.mellanox.com/related-docs/prod_ib_switch_systems/PB_SX6025.pdf.

[115] Mellanox Corp. Mellanox ConnectX-3 Pro, Single/Dual-Port Adapters. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_EN.pdf., 2013.

[116] 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.

[117] 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.

[118] S. S. Mukherjee, P. J. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. *IEEE Micro*, 22(1):26–35, 2002.

[119] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, pages 247–258, 1996.

[120] B. Mutnury, F. Paglia, J. Mobley, G. K. Singh, and R. Bellomio. Quickpath interconnect (qpi) design and analysis in high speed servers. In *Electrical Performance of Electronic*

Packaging and Systems (EPEPS), 2010 IEEE 19th Conference on, pages 265–268. IEEE, 2010.

[121] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 291–305, 2015.

[122] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching Large Graphs with Commodity Processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism (HotPar)*, 2011.

[123] 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.

[124] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's WildFire Prototype. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC)*, page 38, 1999.

[125] D. M. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 219–230, 2013.

[126] 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.

[127] 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.

[128] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, pages 182–195, 2016.

[129] S. Novakovic, A. Daglis, B. Grot, E. Bugnion, and B. Falsafi. Scale-out non-uniform memory access, Aug. 27 2015. US Patent App. 14/634,391.

[130] S. Novakovic, K. Keeton, P. Faraboschi, R. Schreiber, E. Bugnion, and B. Falsafi. NeVer Mind Networking: Using Shared Non-Volatile Memory in Scale-Out Software. In *Intl. Workshop on Rack-Scale Computing (WRSC)*, 2015.

[131] 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.

[132] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, 2011.

[133] 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.

[134] Oracle Corp. Oracle Exalogic Elastic Cloud X3-2 (Datasheet). http://www.oracle.com/us/products/middleware/exalogic/overview/index.html, 2013.

[135] 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):7:1–7:55, 2015.

[136] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[137] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. ACM, 1995.

[138] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[139] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[140] 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.

[141] 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.

[142] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. Technical report, 2007.

[143] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 325–336, 1994.

[144] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[145] 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.

[146] S. Sanfilippo. Redis. http://redis.io/, 2009.

[147] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), pages 174–185, 1996.

[148] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 297–306, 1994.

[149] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 2013 EuroSys Conference*, pages 351–364, 2013.

[150] S. L. Scott and G. M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Hot Interconnects*, 1996.

[151] 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.

[152] S. Shelach. Mellanox wins $200m Google, Microsoft deals. http://www.globes.co.il/serveen/globes/docview.asp?did=1000857043&fid=1725, 2013.

[153] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 183–197, 2015.

[154] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6, 1996.

[155] R. Stets, S. Dwarkadas, N. Hardavellas, G. C. Hunt, L. I. Kontothanassis, S. Parthasarathy, and M. L. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered

Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–183, 1997.

[156] 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.

[157] 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.

[158] 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.

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

[160] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[161] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 2015 EuroSys Conference*, pages 18:1–18:17, 2015.

[162] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 40–53, 1995.

[163] 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.

[164] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26:18 –31, 2006.

[165] WinterCorp. Big Data and Data Warehousing. http://www.wintercorp.com/.

[166] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[167] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 43–56, 2016.

[168] K. A. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. N. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. L. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. pages 24–32, 2007.

[169] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2012.

[170] 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.

Stanko Novakovic

Data Center Systems & Parallel Systems Architecture Laboratories
INN 233, Station 14, CH-1015 Lausanne
stanko.novakovic@epfl.ch | http://parsa.epfl.ch/~snovakov

RESEARCH INTERESTS

Broad: Distributed systems, operating systems, computer architecture

Current focus: Datacenter systems, network protocols and hardware-software interfaces for one-sided memory access, distributed NoSQL databases, performance analysis

EDUCATION

Swiss Federal Institute of Technology in Lausanne (EPFL) Lausanne, CH
Ph.D., Computer Science 2011-2017
Thesis: "Rack-Scale Memory Pooling for Datacenters"
Advisors: Prof. Edouard Bugnion and Prof. Babak Falsafi

University of Novi Sad Novi Sad, RS
M.S., Electrical and Computer Engineering 2010-2011

University of Novi Sad Novi Sad, RS
B.S., Electrical and Computer Engineering 2006-2010

AWARDS

EPFL Award for achievements above expectations, EPFL, 2016

EPFL I&C departmental fellowship, EPFL, 2011/2012

Dositeja award from the Serbian Ministry of Youth and Sports (awarded to the best students in the Republic of Serbia) for academic years 2011/2012, 2012/2013

Special award for the accomplished results in 2009/2010 from the National R&D Institute RT-RK for Computer Based Systems

EXPERIENCE

Swiss Federal Institute of Technology in Lausanne (EPFL) Lausanne, CH

Doctoral Assistant Sep, 2011 – Jun, 2017

Systems and architectures for future datacenters, queuing models and analysis, hardware-software system co-design, prototyping and evaluation, teaching assignments/exams, talks and presentations.

Hewlett-Packard Laboratories Palo Alto, CA

Research Associate June, 2014 – Sept, 2014

Rack-scale systems with shared non-volatile memories (NVM) based on memristor technology, software design principles for such systems, design and implementation of a graph processing engine and a distributed NoSQL database, two patent applications.

Microsoft Research Cambridge, UK

Research Associate Aug, 2013 – Oct, 2013

Datacenter architectures with support for remote direct memory access (RDMA). Design and implementation of a distributed transaction engine using multi-version concurrency control and RDMA.

RT-RK, National R&D Institute for Computer Based Systems Novi Sad, RS

Software Engineer Mar, 2011 – Aug, 2011

Android OS for an ARM-based set-top box.

RT-RK, National R&D Institute for Computer Based Systems Novi Sad, RS

Research Associate (part-time) June, 2008 – June, 2011

Android OS for MIPS processor architecture. Initial version of MIPSAndroid for MIPS big-endian processors and a TV based on it.

Micronas/Novi Sad Institute of Information Technologies Novi Sad, RS

Research Associate (part-time) June, 2007 - June, 2008

Parallel programming on IBM's Cell Broadband Engine and a motion detection algorithm in FPGA.

PATENTS

S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, B. Grot.

Scale-Out Non-Uniform Memory Access. US Patent Application 20,150,242,324, 2015

- licensed by a telecommunications equipment vendor, granted in 2017.

S. Novakovic, P. Faraboschi, K. Keeton, J. Zhao, R. Schreiber**.**

Responsive Server Identification Among Multiple Data Servers Linked to a Shared Memory WO Patent Application 2016137496, 2016

S. Novakovic, P. Faraboschi, K. Keeton, J. Zhao, R. Schreiber.

Graph Update Flush to a Shared Memory WO Patent Application 2016144299, 2016

PUBLICATIONS

S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, B. Grot.

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

2016 ACM Symposium on Cloud Computing (**SOCC**), 2016

A. Daglis, D. Ustiugov, S. Novakovic, E. Bugnion, B. Falsafi, B. Grot.

SABRes: Atomic Object Reads for In-Memory Rack-Scale Computing.

49th Annual IEEE/ACM International Symposium on Microarchitecture (**MICRO**), 2016

S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, B. Grot.

An Analysis of Load Imbalance in Scale-out Data Serving.

ACM International Conference on Measurement and Modeling of Computer Science (**SIGMETRICS**) – extended abstract, 2016.

A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, B. Grot.

Manycore Network Interfaces for InMemory Rack-Scale Computing.

International Symposium on Computer Architecture (**ISCA**), 2015.

S. Novakovic, K. Keeton, P. Faraboschi, R. Schreiber, E. Bugnion, B. Falsafi.

NeVer Mind Networking: Using Shared Non-Volatile Memory in Scale-Out Software.

International Workshop on Rack-Scale Computing (**WRSC**), 2015

S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, B. Grot.
Scale-Out NUMA.
Architectural Support for Programming Languages and Operating Systems (**ASPLOS**), 2014.

D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, R. Bianchini.
DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments.
USENIX Annual Technical Conference (**USENIX**), 2013

TALKS

Rack-Scale Memory Pooling for Datacenters.
VMware Research, Palo Alto, 2016
Oracle Labs, Redwood City, 2016
Microsoft Research, Cambridge (UK), 2016
Microsoft Research-EPFL-ETHZ JRC Workshop, Zurich, 2016

The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems.
ACM SOCC, Santa Clara, 2016

NeVer Mind Networking: Using Shared Non-Volatile Memory in Scale-Out Software.
ACM EuroSys (WRSC), Bordeaux, 2015

Scale-Out NUMA.
Nano-Tera Annual Event, Bern, 2015
EcoCloud Annual Event, Lausanne, 2014
ACM ASPLOS, Salt Lake City, 2014
ACM EuroSys, Prague, 2013

A Transactional System for Modern Networks.
Microsoft Research, Cambridge (UK), 2013

TEACHING (TA)

Advanced Multiprocessor Architecture, EPFL, Fall 2015

Computer Architecture, EPFL, Fall 2014

Introduction to Database Systems, EPFL, Spring 2014

Operating Systems, EPFL, Spring 2012

Advanced Computer Networks and Distributed Systems, EPFL, Fall 2012